

---

# Grails

Fran García <<fgarciarico@gmail.com>>

## Table of Contents

1. Introducción a Groovy .....	7
1.1. ¿Qué es Groovy? .....	7
Integración con Java .....	7
¿A quién va dirigido Groovy? .....	8
A los programadores Java .....	8
A los programadores de scripts .....	8
A los programadores ágiles y extremos .....	8
1.2. Instalación .....	9
1.3. Hola Mundo .....	9
groovysh .....	9
groovyConsole .....	10
groovy .....	10
1.4. Características .....	11
Comentarios .....	11
Comparando la sintaxis de Java y Groovy .....	11
Brevedad del lenguaje .....	12
Aserciones .....	13
Un primer vistazo al código en Groovy .....	14
Declaración de clases .....	14
Scripts en Groovy .....	14
GroovyBeans .....	15
Todo son objetos .....	15
Operador == .....	16
La verdad en Groovy .....	16
Estructuras de control en Groovy .....	17
1.5. Tipos de datos en Groovy .....	19
Tipos primitivos y referencias .....	19
Boxing, unboxing y autoboxing .....	20
Tipado dinámico .....	21
Trabajo con cadenas .....	22
La librería GString .....	23
Expresiones regulares .....	23
Números .....	26
1.6. Colecciones .....	27
Rangos .....	27
Listas .....	29
Mapas .....	30
1.7. Ejercicios .....	33
De Java a Groovy (0.25 puntos) .....	33
GroovyBeans (0.50 puntos) .....	34
Expresiones regulares (0.5 puntos) .....	34
2. Aspectos avanzados del Lenguaje Groovy. Metaprogramación. ....	36
2.1. Closures .....	36
Definición de closure .....	36
Declarando closures .....	36

---

Los closures como objetos .....	38
Usos de los closures .....	39
Valores por defecto .....	40
Más métodos de los closures .....	40
Valores devueltos en los closures .....	40
2.2. Groovy como lenguaje orientado a objetos .....	41
Clases y scripts .....	41
Organizando nuestras clases y scripts .....	45
Características avanzadas del modelo orientado a objetos .....	47
GroovyBeans .....	47
Operador * .....	49
2.3. Metaprogramación .....	49
Definición de metaprogramación .....	49
Metaprogramación en Groovy .....	50
Clase Expando .....	50
Objeto delegate .....	51
Metaclass .....	52
2.4. Groovy Builders .....	54
MarkupBuilder .....	54
JsonBuilder .....	56
2.5. Ejercicios .....	58
Closures sobre colecciones (0.25 puntos) .....	58
Clase completa en Groovy (0.5 puntos) .....	58
Metaprogramación (0.5 puntos) .....	58
3. Introducción a Grails. Scaffolding. ....	60
3.1. ¿Qué es? .....	60
¿Qué empresas utilizan Grails? .....	60
3.2. Características de Grails .....	61
Convención sobre configuración .....	61
Tests .....	61
Scaffolding .....	61
Mapeo objeto-relacional .....	61
Plugins .....	62
Internacionalización (i18n) .....	62
Software de código abierto .....	62
Groovy .....	62
Framework Spring .....	62
Hibernate .....	62
SiteMesh .....	63
Tomcat y Jetty .....	63
H2 .....	63
Spock .....	63
Gant .....	63
3.3. Arquitectura .....	63
3.4. Instalación de Grails .....	64
Instalación de Grails en línea de comandos .....	65
Instalación en IntelliJ IDEA .....	65
3.5. Scaffolding .....	66
Descripción de la aplicación ejemplo .....	67
Creación del proyecto Grails .....	67
Creación de clases de dominio .....	69
Creación de controladores .....	71
Scaffolding estático .....	76

---

---

3.6. Ejercicios .....	78
Marcar como realizada una tarea (0.25 puntos) .....	78
Modificación pantalla inicial (0.25 puntos) .....	78
Scaffolding estático sobre la clase Categoría (0.75 puntos) .....	78
4. Patrón MVC: Vistas y controladores. ....	80
4.1. Estructura de las vistas en Grails .....	80
4.2. Plantillas .....	82
Plantilla de pie de página .....	82
Plantilla de encabezado .....	83
4.3. Páginas .....	84
Variables y alcance de las mismas en páginas .....	85
Directivas de páginas .....	86
Expresiones .....	86
4.4. Etiquetas .....	86
Etiquetas lógicas .....	86
Etiquetas de iteración .....	87
Etiquetas de asignación .....	88
Etiquetas de enlaces .....	88
Etiquetas de formularios .....	88
Etiquetas de renderizado .....	90
Etiquetas de validación .....	91
4.5. Librería de etiquetas .....	91
Etiquetas simples .....	92
Etiquetas lógicas .....	93
Generador de código HTML .....	93
4.6. Controladores .....	94
Acciones en los controladores .....	94
Ámbitos de los controladores .....	96
Relación entre vistas y controladores .....	96
Renderizando vistas .....	99
Redirecciones y encadenamientos .....	99
4.7. Filtros .....	100
4.8. Ejercicios .....	103
Modificando las vistas (0.25 puntos) .....	103
Etiqueta para valores booleanos (0.50 puntos) .....	103
Página de confirmación de eliminación de etiquetas (0.50 puntos) .....	103
5. Patrón MVC: Dominios y servicios. ....	105
5.1. Dominios .....	105
Creación de dominios .....	105
Relaciones entre clases de dominio .....	107
Uno a uno .....	107
Uno a muchos .....	108
Muchos a muchos .....	109
5.2. Restricciones .....	110
Restricciones predefinidas en GORM .....	110
Construir tus propias restricciones .....	112
Mensajes de error de las restricciones .....	114
5.3. Aspectos avanzados de GORM .....	114
Ajustes de mapeado .....	114
Nombres de las tablas y las columnas .....	115
Deshabilitar el campo version .....	115
Carga perezosa de los datos .....	115
Sistema caché .....	116

---

---

Modificar la clave primaria .....	117
Herencia de clases .....	117
Propiedades transitorias .....	118
Eventos GORM .....	119
Fechas automáticas .....	119
5.4. Interactuar con la base de datos .....	120
Consultas dinámicas de GORM .....	120
Consultas HQL de Hibernate .....	122
Consultas Criteria de Hibernate .....	123
5.5. Servicios .....	126
Ámbito de los servicios .....	128
Servicios web .....	129
5.6. Ejercicios .....	131
Construye tus propias restricciones (0.25 puntos) .....	131
Asegurar los borrados de las categorías y las tareas (0.25 puntos) .....	131
Consultas a base de datos (0.25 puntos) .....	131
¿Cuándo han sido realizadas las tareas? (0.50 puntos) .....	131
6. Framework de test Spock. ....	133
6.1. Introducción a los tests .....	133
¿Qué son los tests? .....	133
Tipos de tests de software .....	133
Otros frameworks de tests .....	133
6.2. Framework de tests Spock .....	134
Instalación en Grails .....	134
Tests en Spock .....	134
6.3. Spock en Grails .....	137
Tests con Spock sobre clases de dominio en Grails .....	137
Tests con Spock sobre controladores en Grails .....	140
Tests con Spock sobre librerías de etiquetas, vistas y plantillas en Grails .....	143
Tests con Spock sobre servicios en Grails .....	145
Tests con Spock mockeando objetos .....	146
Algunas anotaciones interesantes .....	148
Ignore .....	148
IgnoreRest .....	148
Ignorelf .....	149
Requires .....	149
Stepwise .....	149
Timeout .....	150
ConfineMetaClassChanges .....	150
Title y Narrative .....	151
6.4. Ejercicios .....	152
Testeando nuestra propia restricción (0.25 puntos) .....	152
Testeando la clase TodoController (0.50 puntos) .....	152
Testeando la clase TodoService (0.50 puntos) .....	152
7. Seguridad con Spring Security plugin. ....	153
7.1. Instalación del plugin .....	153
7.2. Configuración del plugin .....	154
7.3. Clases de dominio .....	155
Clase de dominio Person .....	155
Clase de dominio Role .....	157
Clase de dominio PersonRole .....	157
Clase de dominio RequestMap .....	160
7.4. Asegurar peticiones .....	160

---

---

Anotaciones .....	161
Archivo de configuración Config.groovy .....	162
Instancias de la clase RequestMap .....	163
Expresiones específicas .....	164
Aspectos importantes .....	166
Mediante anotaciones .....	166
Mediante instancias de la clase de dominio RequestMap .....	166
Mediante reglas en el archivo Config.groovy con InterceptUrlMap .....	166
7.5. Controladores .....	167
7.6. Librerías adicionales .....	170
Librería de etiquetas: SecurityTagLib .....	170
ifLoggedIn .....	170
ifNotLoggedIn .....	170
ifAllGranted .....	171
ifAnyGranted .....	171
ifNotGranted .....	171
loggedInUserInfo .....	171
username .....	171
access .....	171
noAccess .....	172
Servicio: SpringSecurityService .....	172
getCurrentUser() .....	172
isLoggedIn() .....	172
getAuthentication() .....	173
getPrincipal() .....	173
encodePassword() .....	173
updateRole() .....	174
deleteRole() .....	174
clearCachedRequestmaps() .....	175
reauthenticate() .....	175
7.7. Otros aspectos interesantes de Spring Security .....	176
Restricciones por IP .....	176
Internacionalización .....	176
7.8. Ejercicios .....	178
Roles y usuarios (0.50 puntos) .....	178
Modificar plantillas y vistas (0.25 puntos) .....	178
Diferente rol, diferentes reglas (0.25 punto) .....	178
Modificar los tests unitarios (0.25 puntos) .....	178
8. Configuración de aplicaciones. Plugins interesantes .....	180
8.1. Configuración de aplicaciones .....	180
El archivo Config.groovy .....	180
Sistema de logs .....	180
El archivo BuildConfig.groovy .....	182
El archivo DataSource.groovy .....	183
El archivo BootStrap.groovy .....	186
El archivo UrlMappings.groovy .....	187
8.2. Empaquetamiento de aplicaciones .....	189
8.3. Otros comandos interesantes de Grails .....	190
8.4. Plugins .....	191
Plugin console .....	192
Plugin para la búsqueda de contenido: Searchable .....	193
Plugin para exportar a otros formatos: Export .....	197
8.5. Ejercicios .....	201

---

Log de acceso a controladores (0.25 puntos) .....	201
URL limpia y pública para las tareas de los usuarios (0.50 puntos) .....	201
Buscador de tareas integrado (0.50 puntos) .....	201

## 1. Introducción a Groovy

En esta primera sesión del módulo, empezaremos explicando qué es el lenguaje de programación Groovy y, cómo, dónde y porqué se creo para ponernos un poco en situación. Posteriormente, pasaremos a implementar nuestro primer ejemplo en Groovy, el típico Hola Mundo y lo compararemos con lo que sería el mismo ejemplo en Java.

Seguidamente describiremos detalladamente algunos aspectos básicos de Groovy, tales como la sintaxis, los tipos de datos y el trabajo con colecciones.

### 1.1. ¿Qué es Groovy?

Groovy nació con la intención de ser un lenguaje rico en características típicas de lenguajes dinámicos, así como para interactuar sin problemas en el entorno Java, para de esta forma, utilizar los beneficios de estos lenguajes dinámicos en una plataforma tan robusta como es Java y todo su entorno. Posiblemente, habrá lenguajes que tengan más características típicas de los lenguajes dinámicos que Groovy o que se integren mejor en la Plataforma Java, pero ninguno de estos, presenta una combinación de estos detalles (lenguaje dinámico y entorno Java) tan eficiente como Groovy. Pero, *¿cómo podríamos definir exactamente Groovy?*

Una buena definición de Groovy podría ser la que se nos proporciona desde la propia página web de Groovy (<http://groovy.codehaus.org>):



#### Definición

Groovy es un lenguaje de programación ágil y dinámico diseñado para la Plataforma Java con determinadas características inspiradas en lenguajes como Python, Ruby o Smalltalk, poniéndolas a disposición de los programadores Java mediante una sintaxis típica de Java. Groovy se integra sin ningún problema con cualquier librería Java y se compila directamente a código byte, con lo que Java y Groovy pueden convivir sin ningún tipo de problemas.

En ocasiones se tacha a Groovy de ser un lenguaje meramente de script, pero esto es un error, a pesar de que Groovy funciona realmente bien como lenguaje script. Groovy puede ser precompilado a Java bytecode, estar integrado en aplicaciones Java, construir robustas aplicaciones web e incluso ser la base de una aplicación completa por si solo.

Se puede afirmar rotundamente que Groovy está destinado a ser utilizado en la Plataforma Java, no en vano muchas partes de Groovy están desarrolladas en Java, así que cuando decimos que *al programar en Groovy, estamos realmente programando en Java*, no estamos diciendo ninguna barbaridad. Todas las características de Java están esperando a ser utilizadas en Groovy, incluido todo el conjunto de librerías de Java.

### Integración con Java

Cuando hablamos de *integración con Java*, nos referimos a dos aspectos. Por un lado, de integración imperceptible, puesto que Groovy es simplemente una nueva forma de crear nuestras habituales clases Java. Al fin y al cabo, Groovy es Java con un archivo jar como dependencia. Por otro lado, la sintaxis de Groovy es más precisa y compacta que en Java y a pesar de eso, cualquier programador Java puede entender el funcionamiento de un fragmento de código escrito en Groovy.

## ¿A quién va dirigido Groovy?

### A los programadores Java

Un programador Java con muchos años de experiencia conocerá todas las partes importantes del entorno Java y su API, así como otros paquetes adicionales. Sin embargo, algo aparentemente tan sencillo como listar todos los archivos recursivamente que cuelgan de un directorio se hace demasiado pesado en Java y suponiendo que existiera la función `eachFileRecurse()` tendríamos algo similar a:

```
public class ListFiles {
    public static void main(String[] args){
        new java.io.File(".").eachFileRecurse(
            new FileListener() {
                public void onFile (File file) {
                    System.out.println(file.toString());
                }
            }
        );
    }
}
```

Mientras que con Groovy podría quedar en una sola línea que podríamos ejecutar incluso en línea de comandos de la siguiente forma:

```
groovy -e "new File('.').eachFileRecurse { println it}"
```

### A los programadores de scripts

Groovy también va dirigido a aquellos programadores que han pasado muchos años trabajando con lenguajes de programación como Perl, Ruby, Python o PHP. En ocasiones, sobre todo en las grandes empresas, los clientes requieren obligatoriamente el uso de una Plataforma robusta como puede ser Java. Estos programadores deben reciclarse, ya que no es factible cualquier solución que pase por utilizar un lenguaje script como los mencionados anteriormente.

Continuamente este tipo de programadores afirman que se sienten frustrados por toda la verborrea que conlleva un lenguaje como Java y la frase *"con un lenguaje como Ruby, podría escribir todo este método en una sola línea"* es su pan de cada día. Groovy puede darles lo que ellos piden y supone una vía de escape hacia adelante a este tipo de programadores.

### A los programadores ágiles y extremos

Este tipo de programadores están acostumbrados a trabajar con la próxima revisión tan cercana, que pararse a pensar en utilizar un nuevo lenguaje de programación queda fuera de sus principales objetivos. Sin embargo, una de los aspectos más importantes para los programadores ágiles, consiste en tener siempre a su disposición los mejores recursos y metodologías para llevar a cabo su trabajo.

Y ahí es donde Groovy les puede aportar mucho, ya que Groovy es un lenguaje perfecto para realizar esas tareas de automatización tan habituales en sus proyectos. Estas tareas



van desde la integración continua o la puesta en práctica de teorías como TDD (Test Driven Development).

## 1.2. Instalación

Antes de avanzar en la sesión, vamos a ver como se realizaría la instalación de Groovy en nuestro ordenador. Aunque podríamos descargarnos la última versión de Groovy desde la web oficial <http://groovy.codehaus.org/Download>, nosotros vamos a utilizar un gestor de paquetes conocido como *gvmtool* (<http://gvmtool.net/>) que nos permitirá gestionar varias versiones de Groovy en una misma máquina de forma muy sencilla. Además, esta misma herramienta nos servirá para instalar y gestionar Grails, entre otras interesantes librerías como Gradle o Vert.x.

Para instalar *gvmtool* y *groovy* simplemente debemos en línea de comandos:

```
.....  
curl -s get.gvmtool.net | bash  
  
gvm list groovy  
  
gvm install groovy 2.4.0  
.....
```

Si necesitáramos alternar entre varias versiones de Groovy, simplemente deberías ejecutar:

```
.....  
gvm use groovy 1.8.0  
.....
```

## 1.3. Hola Mundo

Ahora que ya tenemos una primera idea de lo que es Groovy, vamos a pasar a desarrollar nuestro primer ejemplo y a realizar nuestras primeras pruebas de ejecución y compilación de archivos Groovy. Pero antes de eso, vamos a introducir las diferentes formas de ejecutar nuestros programas en Groovy.

Tras la instalación de Groovy con la herramienta *gvm*, vamos a disponer de tres comandos que son *groovysh*, *groovyConsole* y *groovy*, entre otros. Podemos utilizar cualquiera de estos tres archivos para empezar a realizar nuestras pruebas con código Groovy. Vamos a verlos uno a uno:

### groovysh

Permite ejecutar en línea de comandos código Groovy de forma interactiva. Más información en <http://groovy.codehaus.org/Groovy+Shell>.

```
.....  
groovy:000> "Hola Mundo!"  
  
==== > Hola Mundo!  
.....
```

Otro ejemplo podría ser:

```
.....  
groovy:000> saludo = "Hola"  
====> Hola  
groovy:000> "${saludo} Mundo!"  
.....
```

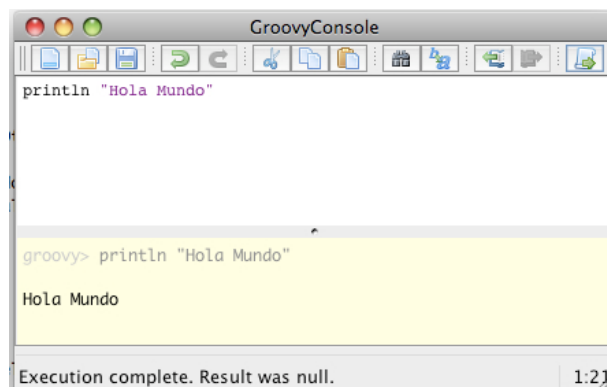
```
====> Hola Mundo!
```

E incluso podemos definir funciones con parámetros:

```
groovy:000> def hola(nombre){
groovy:001> println "Hola $nombre"
groovy:002> }
====> true
groovy:000> hola("Fran")
Hola Fran
===== > null
```

## groovyConsole

Mediante una interfaz Swing, groovyConsole actua como intérprete de comandos Groovy. Tiene la posibilidad de cargar y salvar ficheros desde su menú File. Más información en <http://groovy.codehaus.org/Groovy+Console>

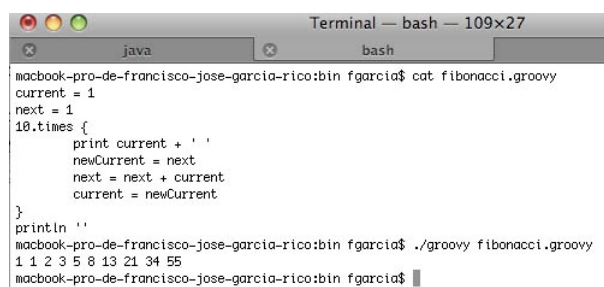


Como podemos observar, en la parte superior podemos escribir nuestro código Groovy, mientras que la parte inferior nos muestra la salida del programa ejecutado.

Además la groovyConsole nos permite también indicarle archivos jar externos que serán importados automáticamente en el código de nuestros scripts.

## groovy

El último método sirve para ejecutar archivos groovy y simplemente ejecutando en la línea de comandos `groovy holamundo.groovy`, podemos ver la salida producida por el programa. En la siguiente imagen, puedes comprobar el funcionamiento de este método con un fichero de ejemplo que realiza el cálculo de la sucesión de Fibonacci.



Quizás ya te hayas dado cuenta, pero a diferencia de en Java, Groovy puede ejecutar sus archivos sin necesidad de compilarlos previamente. Esta forma de utilizar Groovy se llama **Modo directo** y nuestro código es ejecutado sin necesidad de generar archivos ejecutables. Se puede decir que nuestros archivos son interpretados. Sin embargo, existe otra forma de ejecutar nuestros programas Groovy y se llama **Modo precompilado**. En el modo precompilado, en primer lugar se compila el programa groovy y posteriormente se ejecuta. El programa compilado es compatible con los archivos bytecode de Java.

Para compilar nuestros programas Groovy, utilizamos una sentencia en línea de comandos llamada **groovyc** (también disponible tras la instalación de Groovy). Si quisieramos compilar el anterior ejemplo (*fibonacci.groovy*), ejecutaríamos `groovyc -d classes fibonacci.groovy`. Con el parámetro `-d` le hemos indicado que nos genere los archivos compilados en el directorio *classes*, y en el caso de que no existiera, lo crearía. Si observamos el contenido de este directorio, veremos que hay un par de ficheros `.class`. El número de archivos dependerá del propio script, de la misma forma que sucede cuando compilamos archivo en Java.

Una vez ya tenemos nuestro programa compilado, vamos a ver como podríamos ejecutarlo con Java. ¡Sí, con Java!, ya que para ejecutar un programa en Groovy debemos hacer exactamente lo mismo que hacemos en Java. Simplemente, debemos añadir al classpath el archivo jar de Groovy. Debemos hacer lo mismo con el directorio donde residen nuestros archivos compilados. Quedaría algo así:

---

```
java -cp $GROOVY_HOME/embeddable/groovy-all-2.4.0.jar:classes fibonacci
```

---

## 1.4. Características

En la siguiente sección, vamos a introducir diferentes aspectos básicos del lenguaje Groovy, tales como la introducción de comentarios, diferencias y similitudes con Java, la brevedad del código en Groovy y otros pequeños detalles del lenguaje.

### Comentarios

Los comentarios en Groovy siguen el mismo formato que en Java. Esto es:

- `//`, comentarios de una línea
- `/* .... */`, comentarios multilínea
- `/** .... */`, comentarios del estilo Javadoc, que nos permitirá documentar nuestros programas con *Groovydoc*

Sin embargo, también podemos introducir en la primera línea comentarios del estilo shebang con la combinación de caracteres `#!`

### Comparando la sintaxis de Java y Groovy

Es habitual leer que la sintaxis de Groovy es una superclase de Java y esto es debido a las grandes similitudes que hay entre ambos lenguajes. Sin embargo, no es correcto afirmar que un lenguaje es una superclase del otro. Existen pequeños detalles de programación en Java que no existen en Groovy, como puede ser que el operador `==` que en Groovy tiene unas connotaciones diferentes.

Pero aparte de algunas pequeñas diferencias, podemos decir que gran parte de la sintaxis de Java es válida en Groovy. Estas similitudes son:

- Mecanismo de paquetes
- Sentencias
- Definición de clases y métodos
- Estructuras de control
- Operadores, asignaciones y expresiones
- Manejo de excepciones
- Declaración de literales
- Instanciación de objetos y llamadas a métodos

No obstante, Groovy posee un valor añadido gracias a:

- Fácil acceso a los objetos Java a través de nuevas expresiones y operadores
- Nuevas formas de declarar objetos
- Nuevas estructuras de control
- Nuevos tipos de datos con sus correspondientes operadores y expresiones
- Todo se gestiona como un objeto

Estas nuevas características hacen que Groovy sea un lenguaje fácil de leer y de entender.

### Brevidad del lenguaje

Groovy pretende evitar en todo momento toda la ceremonia que acompaña a Java en sus programas y nos permite obviar determinados aspectos obligatorios y centrarnos en nuestro objetivo. Para empezar, se elimina la obligatoriedad de utilizar los puntos y comas (;) al finalizar cada línea. Si lo pensamos un poco es lógico, porque ¿cuántos de nosotros escribimos más de una sentencia en la misma línea del código fuente?. Esto no quiere decir que en Groovy no podamos escribir puntos y coma al final de cada línea, simplemente no es obligatorio.

Además, también se elimina la obligatoriedad de utilizar paréntesis al pasar parámetros en la llamada a funciones.

Otro ejemplo de la brevedad del lenguaje que aporta Groovy es el siguiente ejemplo. Mientras que en Java tendríamos lo siguiente:

```
.....  
java.net.URLEncoder.encode ("a b");  
.....
```

en Groovy tendríamos

```
.....  
URLEncoder.encode 'a b'  
.....
```

Esto no sólo significa que el código en Groovy va a ser más corto, sino también más expresivo evitando toda la parafernalia que acompaña a Java en determinadas ocasiones.

Groovy además importa automáticamente los paquetes *groovy.lang.\**, *groovy.util.\**, *java.lang.\**, *java.util.\**, *java.net.\** y *java.io.\**, así como las clases *java.Math.BigInteger* y

*java.Math.BigDecimal*, así que siempre vas a poder utilizar todas sus clases sin necesidad de que sean importados sus paquetes al inicio del programa. Esto también es diferente a Java, donde sólo se importa automáticamente el paquete *java.lang.\**.

Resumiendo. Groovy nos permite olvidarnos de los paréntesis, la declaración de paquetes, puntos y comas y centrarnos en la funcionalidad de nuestros programas y métodos.

## Aserciones

Las aserciones permiten desde Java 1.4 comprobar si nuestro programa funciona como debería funcionar y Groovy no iba a ser menos, así que también las incluye. Las aserciones se insertan en medio de nuestro código y se utilizan, por ejemplo, para comprobar posibles incongruencias en las variables pasadas a un método. Fundamentalmente las aserciones se utilizarán en los tests que desarrollemos para nuestra aplicación y que asegurarán que las cosas funcionan correctamente. Algunos ejemplos de aserciones serían:

---

```
assert(true)
assert 1 == 1
def x = 1
assert x == 1
def y = 1; assert y == 1
```

---

Como se puede comprobar en los ejemplos las aserciones pueden tomar expresiones completas y no únicamente variables, como en el ejemplo `assert 1 == 1`. Además las aserciones no sólo nos servirán para comprobar el estado de nuestro programa, sino que también podremos reemplazar determinados comentarios por aserciones, que además de clarificar el código fuente de nuestro programa, nos sirve también para pasar pequeñas pruebas a nuestro código.

Cuando usamos aserciones, es una buena práctica incluir un mensaje que nos clarifique el error que se ha producido. Esto lo podemos hacer añadiendo después de la aserción el mensaje de texto que queremos precedido de `:`, como en el siguiente ejemplo:

---

```
assert 1==2 : "Desde cuando 1 es igual a 2"
//Obteniendo el siguiente mensaje
//Exception thrown: Desde cuando 1 es igual a 2. Expression: (1 == 2)
```

---

Además, en las últimas versiones de Groovy se ha mejorado el sistema de mensajes mostrados cuando fallan las aserciones fallan. Si por ejemplo, ejecutamos el siguiente fragmento de código:

---

```
assert new File('foo.bar') == new File('example.txt')
```

---

Groovy nos mostraría el siguiente mensaje de error

---

```
Exception thrown

Assertion failed:

assert new File('foo.bar') == new File('example.txt')
```

---

```
|
foo.bar

| |
| example.txt
false
```

---

## Un primer vistazo al código en Groovy

Antes de comenzar a ver las principales características de Groovy, podéis encontrar toda la documentación de sus clases y librerías en la dirección <http://groovy.codehaus.org/gapi>.

### Declaración de clases

Las clases son la clave de cualquier lenguaje orientado a objetos y Groovy no iba a ser menos. El siguiente fragmento consiste en la declaración de la clase *Libro*, con una sola propiedad *Título*, el constructor de la clase y un método *getter* para obtener el título del libro.

---

```
class Libro {
    private String titulo

    Libro (String elTitulo){
        titulo = elTitulo
    }
    String getTitulo(){
        return titulo
    }
}
```

---

Se puede comprobar como el código es bastante similar a Java. Sin embargo, no hay la necesidad de emplear modificadores de acceso a los métodos, ya que por defecto siempre serán públicos.

### Scripts en Groovy

Los scripts en Groovy son archivos de texto plano con extensión *.groovy* y que pueden ser ejecutados, tal y como comentábamos en una sección anterior mediante el comando *groovy miarchivo.groovy*. A diferencia de en Java, en Groovy no necesitamos compilar nuestros scripts, ya que son precompilados por nosotros al mismo tiempo que lo ejecutamos, así que a simple vista, es como si pudiéramos ejecutar un archivo groovy.

En Groovy podemos tener sentencias fuera de la definición de las clases e incluso podemos añadir métodos a nuestros scripts que también queden fuera de las definiciones de nuestras clases, si con ello conseguimos clarificar nuestro código, con lo que el método *main()* típico de Java, no es necesario en Groovy y sólo lo necesitaremos en el caso de que nuestro script necesite parámetros pasados en línea de comandos. Veamos un ejemplo:

---

```
Libro cgg = new Libro('Curso GroovyGrails')

assert cgg.getTitulo() == 'Curso GroovyGrails'
assert getTituloAlReves(cgg) == 'sliarGyvoorG osruC'

String getTituloAlReves(libro) {
    titulo = libro.getTitulo()
    return titulo.reverse()
}
```

```
}
```

Podemos comprobar que llamamos al método `getTituloAlReves()` incluso antes de que éste sea declarado. Otro aspecto importante es que no necesitamos compilar nuestra clase `Libro`, ya que Groovy lo hace por nosotros. El único requisito es que la clase `Libro` se encuentre en el `CLASSPATH`.

## GroovyBeans

Fundamentalmente los JavaBeans son clases que definen propiedades. Por ejemplo, en nuestro ejemplo de la clase `Libro`, sus propiedades serían el título, el autor, la editorial, etc. Estos JavaBeans tienen métodos para obtener los valores de estas propiedades (getters) y modificarlas (setters). Estos métodos deben ser definidos en cada JavaBean, con lo que esta tarea se vuelve algo repetitiva.

```
class Libro{
    String titulo;

    String getTitulo(){
        return this.titulo
    }

    void setTitulo(String str){
        this.titulo = new String(str)
    }
}
```

Sin embargo, Groovy gestiona esta definición de los métodos `get()` y `set()` por nosotros, con lo que ya no debemos preocuparnos. Con esto, el código anterior, en Groovy quedaría así:

```
class Libro{
    String titulo
}
```

Además, tenemos la gran ventaja de que podemos sobrescribir estos métodos en el caso de que queramos modificar su comportamiento básico.

En realidad, lo que hace Groovy es tratar de entender el comportamiento de las propiedades de un bean a partir de como están definidas. Con esto tenemos los siguientes casos:

- Si una propiedad de una clase es declarada sin ningún modificador, se genera un campo privado con sus correspondientes métodos públicos `getter()` y `setter()`
- Si una propiedad de una clase es declarada como `final`, dicha propiedad será privada y se definirá también su `getter()`, pero no su `setter()`
- Si necesitamos una propiedad de tipo privado o protegida debemos declarar sus métodos `getter()` y `setter()` de forma privada o protegida

## Todo son objetos

En Groovy, a diferencia que en Java, los números son tratados como objetos y no como tipos primitivos. En Java no podemos utilizar métodos de la clase `Integer` si el entero es de tipo

primitivo y tampoco podemos hacer  $y*2$  si la variable  $y$  es un objeto. Esto también es diferente en Groovy, ya que podemos utilizar operadores y métodos indiferentemente.

```
def x = 1
def y = 2
assert x + y == 3
assert x.plus(y) == 3
assert x instanceof Integer
```

En el ejemplo anterior se puede observar que hemos podido utilizar el operador  $+$  de los tipos primitivos y el método `plus()` de la clase `Integer`.

En realidad, en Groovy, todo es un objeto, con lo que aunque declaremos variables como tipos de datos primitivos en realidad Groovy los convierte a tipo de dato referencia, tal y como puedes ver en los siguientes ejemplos:

```
double d = 10.23
assert d instanceof Double
char a = 'a'
assert a instanceof Character
```

## Operador `==`

Es importante comentar también que en Groovy el operador `==` tiene una connotación diferente a la que tiene en Java. En Java, este operador significa igualdad en variables primitivas mientras que en objetos significa objetos idénticos. En Groovy, el significado de este operador es el mismo que la función `equals()`. Si necesitamos comprobar la identidad de dos variables siempre podremos utilizar el método `is()` tal y como vemos en el siguiente ejemplo:

```
if (foo.is(bar))
    println "foo y bar son el mismo objeto"
```

## La verdad en Groovy

Cabe aquí comentar lo que se conoce como *la verdad en Groovy* que es un término que se refiere a que se considera como cierto en Groovy y que no. Veamos algunos ejemplos:

```
def a = true
def b = true
def c = false
assert a
assert a && b
assert a || c
assert !c
```

Hasta aquí ninguna novedad, pero ¿qué pasa si no estamos evaluando directamente un *booleano*? Pues que Groovy añade una serie de reglas para indicar si esa expresión es cierta o no. Por ejemplo, en las colecciones (listas y mapas) se entenderá que una variable de estos tipos será cierta cuando contenga al menos un elemento.



```

def numeros = [1,2,3]
assert numeros //cierto, ya que numeros no está vacía
numeros = [] //vacíamos la lista
assert !numeros //cierto, ya que ahora la lista está vacía

assert ['one':1] //este mapa contiene un elemento con lo que se evalúa a
cierto
assert ![:] //este mapa está vacío

```

En lo que a las cadenas de texto se refiere, siempre que éstas no sean la cadena vacía, se evaluarán a cierto.

```

// Strings
assert 'Esto es cierto'
assert !''
//GStrings
def s = ''
assert !("$s")
s = 'x'
assert ("$s")

```

Si evaluamos un número, siempre que éste no sea cero se evaluará a cierto.

```

assert !0
assert 1

```

Por último, siempre que un objeto no sea *null* éste se evaluará a cierto.

```

assert new Object()
assert !null

```

## Estructuras de control en Groovy

Algunas estructuras de control en Groovy son tratadas de la misma forma que en Java. Estos bloques son *if-else*, *while*, *switch* y *try-catch-finally*, aunque es conveniente hacer algunas anotaciones. En los condicionales, *null* es tratado como *false*, mientras que *!null* será *true*.

Con respecto a la sentencia *if*, Groovy soporta la sentencia de forma anidada.

```

if ( ... ) {
  ...
} else if ( ... ) {
  ...
} else {
  ...
}

```

De igual forma, se introduce también el operador ternario.

```
def y = 5
def x = (y > 1) ? "funciona" : "falla"
assert x == "funciona"
```

Así como el operador Elvis

```
def usuario = [nombre:"Fran"]
def nombre = usuario.nombre ?: "Anónimo"
assert nombre == "Fran"
usuario = [:]
nombre = usuario.nombre ?: "Anónimo"
assert nombre == "Anónimo"
```

Con respecto a la estructura *switch*, la única diferencia entre Groovy y Java es que en la condición se puede utilizar cualquier tipo de comprobación, tal y como vemos en el siguiente ejemplo:

```
def x = 1.23
def result = ""

switch ( x ) {
  case "foo":
    result = "found foo"
    // lets fall through

  case "bar":
    result += "bar"

  case [4, 5, 6, 'inList']:
    result = "list"
    break

  case 12..30:
    result = "range"
    break

  case Integer:
    result = "integer"
    break

  case Number:
    result = "number"
    break

  default:
    result = "default"
}

assert result == "number"
```

Groovy permite realizar las siguientes comprobaciones en una estructura *switch*:

- Comparar la clase de una variable

- Expresiones regulares
- Comprobar si la variable está contenida en una colección
- Y en último lugar, se comprueba la igualdad entre la variable evaluada y el caso en particular

Los bloques de código del tipo *for* utilizan la notación *for(i in x){cuerpo}* donde *x* puede ser cualquier cosa que Groovy sepa como iterar (iteradores, enumeraciones, colecciones, mapas, rangos, etc.)

```
for(i in 1..10)
    println i

for(i in [1,2,3,4,5,6,7,8,9,10])
    println i
```

En Groovy es una práctica habitual utilizar un closure para simular un bucle *for* como por ejemplo en el siguiente código

```
def alumnos = ['Pedro', 'Miguel', 'Alejandro', 'Elena']
alumnos.each{nombre -> println nombre}
```

## 1.5. Tipos de datos en Groovy

### Tipos primitivos y referencias

En Java, existen los tipos de datos primitivos (*int*, *double*, *float*, *char*, etc) y las referencias (*Object*, *String*, etc), Los tipos de datos primitivos son aquellos que tienen valor por si mismos, bien sea un entero, un carácter o un número en coma flotante y es imposible crear nuevos tipos de datos primitivos.

Mientras que las referencias son identificadores de instancias de clases Java y, como su nombre indica, simplemente es una referencia a un objeto. En los tipos de datos primitivos es imposible realizar llamadas a métodos y éstos no pueden ser utilizados en aquellos lugares donde se espera la presencia de un tipo *java.lang.Object*. Esto hace que determinados fragmentos de código de nuestros programas, se compliquen demasiado, como puede ser el siguiente ejemplo que realiza la suma posición por posición de un par de vectores de enteros.

```
ArrayList resultados = new ArrayList();
for (int i=0; i < listaUno.size(); i++){
    Integer primero = (Integer)listaUno.get(i);
    Integer segundo = (Integer)listaDos.get(i);

    int suma = primero.intValue() + segundo.intValue();
    resultados.add(new Integer(suma));
}
```

En Groovy, todo es un objeto y una solución al ejemplo anterior podría ser utilizando el método *plus()* que Groovy añade al tipo *Integer*: *resultados.add(primero.plus(segundo))*, con lo que nos podríamos ahorrar el paso de la conversión de tipo de dato referencia a tipo de dato primitivo (*primero.intValue()*).

Sin embargo, esta solución también se podría conseguir en Java si se añadiera el método `plus` a la clase `Integer`. Así que Groovy decide ir un poco más lejos y permite la utilización de operadores entre objetos, con lo que la solución en Groovy sería `resultados.add (primero + segundo)`.

De esta forma, lo que en Groovy puede parecer una variable de tipo de dato primitivo, en realidad es una referencia a una clase Java, tal y como se muestra en la siguiente tabla.

Tipo primitivo	Tipo referencia
byte	java.lang.Byte
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double
char	java.lang.Character
boolean	java.lang.Boolean

Así que cada vez que utilices un tipo de datos primitivo en tus programas en Groovy, en realidad estás utilizando la correspondiente clase indicada en la tabla anterior, con lo que, puedes ahorrarte el uso de tipos primitivos en Groovy, porque por detrás se está haciendo una conversión a un tipo de dato referencia.

## Boxing, unboxing y autoboxing

La conversión de un tipo de dato primitivo a un tipo de dato referencia se conoce en Java como *boxing*, mientras que la conversión de un tipo de dato referencia a un tipo de dato primitivo se conoce *unboxing*. Groovy automatiza estas operaciones en lo que se conoce como *autoboxing*.

Pero, si Groovy convierte todo a un tipo de dato referencia, ¿qué pasa con aquellos métodos de Java que esperan un parámetro de tipo de dato primitivo? No hay de que preocuparse, el autoboxing de Groovy se encarga de eso. Por ejemplo, en el método `indexOf` de la clase `java.lang.String` se espera como parámetro un entero (`int`) que indica el carácter buscado en la cadena, devolviendo también un entero indicando la posición en la que se ha encontrado el carácter. Si probamos el siguiente ejemplo, veremos como todo funciona correctamente, ya que Groovy se encarga de realizar el autoboxing allí donde considere oportuno, en este caso, en el paso del parámetro a la función `indexOf`. El siguiente código trata de obtener la posición de la primera 'o' de la cadena.

```
assert 'Ho1a Mundo'.indexOf(111) == 1
```

En un principio Groovy, debería convertir el tipo de datos `int` del valor 111 a `Integer`, sin embargo, la función `indexOf()` requiere un parámetro de tipo `int`, con lo que el autoboxing de Groovy funciona de tal forma para convertir el parámetro al tipo de dato requerido, en este caso, `int`.

Otro aspecto interesante del autoboxing de Groovy es que no siempre se ejecuta el autoboxing para la realización de determinadas operaciones. Por ejemplo, en la operación `1 + 2`, podemos pensar que los valores 1 y 2 son del tipo referencia `Integer`, lo cual es cierto y que para poder realizarse la operación, éstos deben ser convertidos a tipo `int`, lo cual no es cierto.

De Groovy se dice que es incluso más orientado a objetos que Java y se dice por cuestiones como esta. En la operación  $1 + 2$ , lo que Groovy está realmente ejecutando es *1.plus(2)*, con lo que no es necesaria ninguna conversión para realizar esta operación.

## Tipado dinámico

Hasta el momento, en prácticamente todos los ejemplos que hemos visto en Groovy, hemos obviado especificar los tipos de datos utilizados, dejando que Groovy lo haga por nosotros. Esto es lo que se conoce como tipado dinámico y en este punto, vamos a ver los pros y los contras de su uso. La siguiente tabla, muestra un ejemplo con definiciones de variables y como actúa Groovy en cada caso.

Sentencia	Tipo de variable
<code>def a = 2</code>	<code>java.lang.Integer</code>
<code>def b = 0.4f</code>	<code>java.lang.Float</code>
<code>def c = 'a'</code>	<code>java.lang.String</code>
<code>int d = 3</code>	<code>java.lang.Integer</code>
<code>float e = 4</code>	<code>java.lang.Float</code>
<code>char f = '1'</code>	<code>java.lang.Character</code>
<code>Integer g = 6</code>	<code>java.lang.Integer</code>
<code>String h = '1'</code>	<code>java.lang.String</code>
<code>Character i = 'a'</code>	<code>java.lang.Character</code>

```
def a = 2
def b = 0.4f
def c = 'a'
int d = 3
float e = 4
char f = '1'
Integer g = 6
String h = '1'
Character i = 'a'
```

```
assert a instanceof java.lang.Integer
assert b instanceof java.lang.Float
assert c instanceof java.lang.String
assert d instanceof java.lang.Integer
assert e instanceof java.lang.Float
assert f instanceof java.lang.Character
assert g instanceof java.lang.Integer
assert h instanceof java.lang.String
assert i instanceof java.lang.Character
```

La palabra reservada *def* se utiliza cuando no queremos especificar ningún tipo de dato en especial y dejamos que Groovy decida por nosotros, tal y como aparece en los tres primeros ejemplos. En los tres ejemplos siguientes, podemos ver como independientemente de declarar una variable como tipo de dato primitivo, ésta acabará siendo un tipo de dato referencia. Los tres últimos ejemplos, servirán a la persona que esté leyendo el código para entender que esa variable es un objeto.

Aquí es importante resaltar que Groovy es un lenguaje de tipado dinámico de datos seguro, lo que quiere decir, que Groovy no nos va a permitir realizar operaciones de una determinada clase a un objeto definido de forma diferente. Por ejemplo, en el trozo de código *String f = '1'*, la variable *f* nunca va a poder ser utilizada para realizar operaciones matemáticas como si fuera de la clase *java.lang.Number* salvo que hagamos la correspondiente conversión.

Poder elegir si utilizamos tipado dinámico o estático, es una de las mejores cosas que tiene Groovy. En Internet existen muchos foros de discusión creados a partir de este debate donde se exponen los pros y los contras de cada método. El *tipado estático* nos proporciona más información para la optimización de nuestros programas y revelan información adicional sobre el significado de la variable o del parámetro utilizado en un método determinado.

Por otro lado, el *tipado dinámico* no sólo es el método utilizado por los programadores vagos que no quieren estar definiendo los tipos de las variables, sino que también se utiliza cuando la salida de un método es utilizado como entrada de otro sin tener que hacer ningún trabajo extra por nuestra parte. De esta forma, el programador deja a Groovy que se encargue de la conversión de los datos en caso de que sea necesario y factible.

Otro aspecto interesante del tipado dinámico, es lo que se conoce como el *duck typing* (tipado de patos) y es que, si hay algo que camina como un pato y habla como un pato, lo más probable es que sea un pato. El tipado dinámico es interesante utilizarlo cuando se desconoce a ciencia cierta el tipo de datos de una determinada variable o parámetro. Esto nos proporciona un gran nivel de reutilización de nuestro código, así como la posibilidad de implementar funciones genéricas.

## Trabajo con cadenas

Groovy nos facilita el trabajo con las cadenas de texto en mayor medida que lo hace Java, añadiendo su propia librería *groovy.lang.GString*, con lo que además de los métodos ofrecidos por la clase *java.lang.String*, Groovy cuenta con más métodos ofrecidos por esta librería. Una característica de como trabaja Groovy con las cadenas de texto es que nos permite introducir variables en las cadenas sin tener que utilizar caracteres de escape como por ejemplo *"hola \$minombre"*, donde en la misma cadena se introduce el valor de una variable. Esto es típico de algunos lenguajes de programación como PHP y facilita la lectura de nuestro código ya que no tenemos que estar escapando constantemente.

La siguiente tabla muestra las diferentes formas que hay en Groovy para crear una cadena de texto:

Caracteres utilizados	Ejemplo	Soporte GString
Comillas simples	'hola Juan'	No
Comillas dobles	"hola \$nombre"	Sí
3 comillas simples	"""----- Total:0.02 -----"""	No
3 comillas dobles	"""----- Total:\$total -----"""	Sí
Símbolo /	/x(\d*)y/	Sí

La diferencia entre las comillas simples y las dobles es la posibilidad de incluir variables precedidas del símbolo \$ para mostrar su valor. Las cadenas introducidas con el símbolo / son utilizadas con expresiones regulares, como veremos más adelante.

## La librería GString

La librería GString (*groovy.lang.GString*) añade determinados métodos para facilitarnos el trabajo con cadenas de texto, las cuales normalmente se crean utilizando comillas dobles. Básicamente, una cadena de tipo GString nos va a permitir introducir variables precedidas del símbolo \$. También es posible introducir expresiones entre llaves (*\$(expression)*), tal y como si estuviéramos escribiendo un *closure*. Veamos algunos ejemplos:

```
nombre = 'Fran'
apellidos = 'García'
salida = "Apellidos, nombre: $apellidos, $nombre"

fecha = new Date(0)
salida = "Año $fecha.year, Mes $fecha.month, Día $fecha.date"

salida = "La fecha es ${fecha.toGMTString()}"

sentenciasql = """
SELECT nombre, apellidos
FROM usuarios
WHERE anyo_nacimiento=$fecha.year
"""
```

Ahora que ya podemos declarar variables de texto, vamos a ver algunos métodos que podemos utilizar en Groovy:

```
saludo = 'Hola Juan'

assert saludo.startsWith('Hola')

assert saludo.getAt(3) == 'a'
assert saludo[3] == 'a'

assert saludo.indexOf('Juan') == 5
assert saludo.contains('Juan')

assert saludo[5..8] == 'Juan'

assert 'Buenos días' + saludo - 'Hola' == 'Buenos días Juan'

assert saludo.count('a') == 2

assert 'b'.padLeft(3) == '  b'
assert 'b'.padRight(3, '_') == 'b__'
assert 'b'.center(3) == ' b '
assert 'b' * 3 == 'bbb'
```

## Expresiones regulares

Las expresiones regulares nos permiten especificar un patrón y buscar si éste aparece en un fragmento de texto determinado. Groovy deje que sea Java la encargada del tratamiento de las expresiones regulares, pero además, añade tres métodos para facilitarnos este trabajo:

- El operador  `=~` : *find*

- El operador `==~`: *match*
- El operador `~String`: *pattern*

Con los patrones de las expresiones regulares, realmente estamos indicando que estamos buscando exactamente. Veamos algunos ejemplos:

Patrón	Significado
algo de texto	simplemente encontrará la frase "algo de texto"
algo de\s+texto	encontrará frases que empiecen con "algo de", vayan seguidos por uno o más caracteres y terminen con la palabra texto
\d\d/\d\d/\d\d\d\d	detectará fechas como por ejemplo 28/06/2008

El punto clave de los patrones de las expresiones regulares, son los símbolos, que los podemos sustituir por determinados fragmentos de texto. La siguiente tabla presenta estos símbolos:

Símbolo	Significado
.	Cualquier carácter
^	El inicio de una línea
\$	El final de una línea
\d	Un dígito
\D	Cualquier cosa excepto un dígito
\s	Un espacio en blanco
\S	Cualquier cosa excepto un espacio en blanco
\w	Un carácter de texto
\W	Cualquier carácter excepto los de texto
\b	Límite de palabras
()	Agrupación
(x y)	O x o y
x*	Cero o más ocurrencias de x
x+	Una o más ocurrencias de x
x?	Cero o una ocurrencia de x
x{m,n}	Entre m y n ocurrencias de x
x{m}	Exactamente m ocurrencias de x
[a-d]	Incluye los caracteres a, b, c y d
[^a]	Cualquier carácter excepto la letra a

Las expresiones regulares nos ayudarán en Groovy a:

- Indicarnos si un determinado patrón encaja completamente con un texto
- Si existe alguna ocurrencia de un patrón en una cadena



- Contar el número de ocurrencias
- Hacer algo con una determinada ocurrencia
- Reemplazar todas las ocurrencias con un determinado texto
- Separar una cadena en múltiples cadenas a partir de las ocurrencias que aparezcan en la misma

El siguiente fragmento de código muestra el funcionamiento básico de las expresiones regulares.

```
refran = "tres tristes tigres tigraban en un tigral"

//Compruebo que hay al menos un fragmento de código que empieza por t,
//le siga cualquier caracter y posteriormente haya una g
assert refran =~ /t.g/

//Compruebo que el refrán esté compuesto sólo
//por palabras seguidas de un espacio
assert refran ==~ /(\w+ \w+)* /

//Compruebo que el valor de una operación de tipo match es un booleano
assert (refran ==~ /(\w+ \w+)* /) instanceof java.lang.Boolean

//A diferencia que una operación de tipo find,
//las operaciones match se evalúan por completo contra una cadena
assert (refran ==~ /t.g/) == false

//Sustituyo las palabras por el caracter x
assert (refran.replaceAll(/\w+/, 'x')) == 'x x x x x x x'

//Devuelve un array con todas las palabras del refrán
palabras = refran.split(/ /)
assert palabras.size() == 7
assert palabras[2] == 'tigres'
assert palabras.getAt(3) == 'tigraban'
```

Es importante resaltar la diferencia entre el operador *find* y el operador *match*. El operador *match* es más restrictivo puesto que intenta hacer coincidir un patrón con la cadena entera, mientras que el operador *find*, sólo pretende encontrar una ocurrencia del patrón en la cadena.

Ya sabemos como localizar fragmentos de texto en cadenas, pero ¿y si queremos hacer algo con estas cadenas encontradas? Groovy nos vuelve a facilitar esta tarea y pone a nuestra disposición un par de formas para recorrer las ocurrencias encontradas: *each* y *eachMatch*. Por un lado, al método *eachMatch* se le pasa una cadena con un patrón de expresión regular como parámetro: *String.eachMatch(patron)*, mientras que al método *each* se le pasa directamente el resultado de una operación de tipo *find()* o *match()*: *Matcher.each()*. Veámos ambos métodos en funcionamiento.

```
refran = "tres tristes tigres tigraban en un tigral"

//Busco todas las palabras que acaben en 'es'
rima = ~/\b\w*es\b/
resultado = ''
```

```

refran.eachMatch(rima) { match ->
    resultado += match + ' '
}

assert resultado == 'tres tristes tigres '

//Hago lo mismo con el método each
resultado = ''
(refran =~ rima).each { match ->
    resultado += match + ' '
}

assert resultado == 'tres tristes tigres '

//Sustituyo todas las rimas por guiones bajos
assert (refran.replaceAll(rima){ it-'es'+'__'} == 'tr__ trist__ tigr__
    tigraban en un tigral')

```

## Números

El GDK de Groovy introduce algunos métodos interesantes en cuanto al tratamiento de números. Estos métodos funcionan como closures y nos servirán como otras formas de realizar bucles. Estos métodos son:

- *times()*, se utiliza para realizar repeticiones
- *upto()*, utilizado para realizar una secuencia de acciones de forma creciente
- *downto()*, igual que el anterior pero de forma decreciente
- *step()*, es el método general para realizar una secuencia paso a paso

Pero como siempre, veamos varios ejemplos.

```

def cadena = ''
10.times {
    cadena += 'g'
}
assert cadena == 'gggggggggg'

cadena = ''
1.upto(5) { numero ->
    cadena += numero
}

assert cadena == '12345'

cadena = ''
2.downto(-2) { numero ->
    cadena += numero + ' '
}

assert cadena == '2 1 0 -1 -2 '

cadena = ''
0.step(0.5, 0.1) { numero ->
    cadena += numero + ' '
}

```

```
}  
  
assert cadena == '0 0.1 0.2 0.3 0.4 '
```

---

## 1.6. Colecciones

Ahora que ya hemos introducido los tipos de datos simples, llega el turno de hablar de las colecciones presentes en Groovy. En este apartado vamos a ver tres tipos de datos. Por un lado, las *listas* y los *mapas*, que tienen prácticamente las mismas connotaciones que en Java, con alguna nueva característica que añade Groovy, y por otro, los *rangos*, un concepto que no existe en Java.

### Rangos

Empecemos por lo novedoso. Cuantas veces no nos habremos encontrado con un bloque de código similar al siguiente

---

```
for (int i=0;i<10;i++){  
    //hacer algo con la variable i  
}
```

---

El anterior fragmento de código se ejecutará empezando en un límite inferior (0) y terminará de ejecutarse cuando la variable *i* llegue al valor 10. Uno de los objetivos de Groovy consiste en facilitar al programador la lectura y la comprensión del código, así que los creadores de Groovy pensaron que sería útil introducir el concepto de rango, el cual tendría por definición un límite inferior y uno superior.

Para especificar un rango, simplemente se escribe el límite inferior seguido de dos puntos y el límite superior, *limiteInferior..limiteSuperior*. Este rango indicaría que ambos valores establecidos están dentro del rango, así que si queremos indicarle que el límite superior no está dentro del rango, debemos utilizar el operador *..<*, *limiteInferior.<limiteSuperior*. También existen los *rangos inversos*, en los que el límite inferior es mayor que el límite superior. Veamos algunos ejemplos:

---

```
//Rangos inclusivos  
assert (0..10).contains(5)  
assert (0..10).contains(10)  
  
//Rangos medio-exclusivos  
assert (0.<10).contains(9)  
assert (0.<10).contains(10) == false  
  
//Comprobación de tipos  
def a = 0..10  
assert a instanceof Range  
  
//Definición explícita  
a = new IntRange(0,10)  
assert a.contains(4)  
  
//Rangos para fechas  
def hoy = new Date()
```

```
def ayer = hoy - 1
assert (ayer..hoy).size() == 2

//Rangos para caracteres
assert ('a'..'f').contains('e')

//El bucle for con rangos
def salida = ''
for (elemento in 1..5){
    salida += elemento
}
assert salida == '12345'

//El bucle for con rangos inversos
salida = ''
for (elemento in 5..1){
    salida += elemento
}
assert salida == '54321'

//Simulación del bucle for con rangos inversos
//y el método each con un closure
salida = ''
(5..<1).each { elemento ->
    salida += elemento
}
assert salida == '5432'
```

---

Los rangos son objetos y como tales, pueden ser pasados como parámetros a funciones o bien ejecutar sus propios métodos. Un uso interesante de los rangos es el filtrado de datos. También es interesante verlos como clasificador de grupos y su utilidad se puede comprobar en los bloques de código *switch*.

---

```
//Rangos como clasificador de grupos
edad = 31
switch (edad){
    case 16..20: interesAplicado = 0.25; break
    case 21..50: interesAplicado = 0.30; break
    case 51..65: interesAplicado = 0.35; break
}
assert interesAplicado == 0.30

//Rangos para el filtrado de datos
edades = [16,29,34,42,55]
joven = 16..30
assert edades.grep(joven) == [16,29]
```

---

Como se ha podido comprobar, podemos especificar rangos para fechas e incluso para cadenas. En realidad, cualquier tipo de dato puede ser utilizado en un rango, siempre que se cumplan una serie de condiciones:

- El tipo implemente los métodos *next()* y *previous()*, que sobrecargan los operadores ++ y --
- El tipo implemente *java.lang.Comparable*, implementando el método *compareTo()* que sobrecarga el operador <#

## Listas

En Java, agregar un nuevo elemento a un array no es algo trivial. Una solución es convertir el array a una lista del tipo *java.util.List*, añadir el nuevo elemento y volver a convertir la lista en un array. Otra solución pasa por construir un nuevo array del tamaño del array original más uno, copiar los viejos valores y el nuevo elemento. Eso es la parte negativa de los arrays en Java. La parte positiva es que nos permite trabajar con índices en los arrays para recuperar su información, así como modificar su valor, como por ejemplo, *miarray[indice] = nuevoelemento*. Groovy se aprovecha de la parte positiva de Java en este sentido, y añade nuevas características para mejorar su parte negativa.

La definición de una lista en Groovy se consigue utilizando los corchetes [] y especificando los valores de la lista. Si no especificamos ningún valor entre los corchetes, declararemos una lista vacía. Por defecto, las listas en Groovy son del tipo *java.util.ArrayList*. Podemos rellenar fácilmente las listas a partir de otras con el método *addAll()*. También se pueden definir listas a partir de otras con el constructor de la clase *LinkedList*.

```
miLista = [1,2,3]

assert miLista.size() == 3
assert miLista[2] == 3
assert miLista instanceof ArrayList

listaVacia = []
assert listaVacia.size() == 0

listaLarga = (0..1000).toList()
assert listaLarga[324] == 324

listaExplicita = new ArrayList()
listaExplicita.addAll(miLista)
assert listaExplicita.size == 3
listaExplicita[2] = 4
assert listaExplicita[2] == 4

listaExplicita = new LinkedList(miLista)
assert listaExplicita.size == 3
listaExplicita[2] = 4
assert listaExplicita[2] == 4
```

En el fragmento de código anterior, hemos visto como se puede especificar un valor a un elemento de la lista. Pero, ¿qué pasa si queremos especificar un mismo valor a toda la lista o un trozo de la misma? Los creadores de Groovy ya han pensado en ese problema y podemos utilizar rangos y colecciones en las listas.

```
miLista = ['a','b','c','d','e','f']

assert miLista[0..2] == ['a','b','c']//Acceso con Rangos
assert miLista[0,2,4] == ['a','c','e']//Acceso con colección de índices

//Modificar elementos
miLista[0..2] = ['x','y','z']
assert miLista == ['x','y','z','d','e','f']
```

```
//Eliminar elementos de la lista
miLista[3..5] = []
assert miLista == ['x', 'y', 'z']

//Añadir elementos a la lista
miLista[1..1] = ['y', '1', '2']
assert miLista == ['x', 'y', '1', '2', 'z']

miLista = []

//Añado objetos a la lista con el operador +
miLista += 'a'
assert miLista == ['a']

//Añado colecciones a la lista con el operador +
miLista += ['b', 'c']
assert miLista == ['a', 'b', 'c']

miLista = []
miLista << 'a' << 'b'
assert miLista == ['a', 'b']

assert miLista - ['b'] == ['a']

assert miLista * 2 == ['a', 'b', 'a', 'b']
```

En ocasiones las listas son utilizadas junto a estructuras de control para controlar el flujo de nuestro programa.

```
miLista = ['a', 'b', 'c']

//Listas como clasificador de grupos
letra = 'a'
switch (letra){
    case miLista: assert true; break;
    default: assert false
}

//Listas como filtrado de datos
assert ['x', 'y', 'a'].grep(miLista) == ['a']

//Bucle for con lista
salida = ''
for (i in miLista){
    salida += i
}
assert salida == 'abc'
```

Las listas tienen una larga lista de métodos disponibles en el API de Java en la interfaz *java.util.List* para por ejemplo ordenar, unir e intersectar listas.

## Mapas

Un mapa es prácticamente igual que una lista, con la salvedad de que los elementos están referenciados a partir de una clave única (sin caracteres extraños ni palabras reservadas por

Groovy), `miMapa[clave] = valor`. Podemos especificar un mapa al igual que lo hacíamos con las listas utilizando los corchetes, pero ahora debemos añadir la clave a cada valor introducido, como por ejemplo `miMapa = [a:1, b:2, c:3]`. Los mapas creados implícitamente son del tipo `java.util.HashMap`. Veámoslo con ejemplos:

```
def miMapa = [a:1, b:2, c:3]

assert miMapa instanceof HashMap
assert miMapa.size() == 3
assert miMapa['a'] == 1

//Definimos un mapa vacío
def mapaVacio = [:]
assert mapaVacio.size() == 0

//Definimos un mapa de la clase TreeMap
def mapaExplicito = new TreeMap()
mapaExplicito.putAll(miMapa)
assert mapaExplicito['c'] == 3
```

Las operaciones más comunes con los mapas se refieren a la recuperación y almacenamiento de datos a partir de la clave. Veamos algunos métodos de acceso y modificación de los elementos de un mapa:

```
def miMapa = [a:1, b:2, c:3]

//Varias formas de obtener los valores de un mapa
assert miMapa['a'] == 1
assert miMapa.a == 1
assert miMapa.get('a') == 1
//Si no existe la clave, devuelve un valor por defecto, en este caso 0
assert miMapa.get('a',0) == 1

//Asignación de valores
miMapa['d'] = 4
assert miMapa.d == 4
miMapa.e = 5
assert miMapa.e == 5
```

Los mapas en Groovy utilizan los mismos métodos que los mapas en Java y éstos están en el API de Java referente a `java.util.Map`, pero además, Groovy añade un par de métodos llamados `any()` y `every()`, los cuales, utilizados como closures, permite evaluar si todos (`every`) o al menos uno (`any`) de los elementos del mapa cumplen una determinada condición. Además, en el siguiente fragmento de código, vamos a ver como iterar sobre los mapas.

```
def miMapa = [a:1, b:2, c:3]

def resultado = ''
miMapa.each { item ->
    resultado += item.key + ':'
    resultado += item.value + ', '
}
assert resultado == 'a:1, b:2, c:3, '
```

```

resultado = ''
miMapa.each { key, value ->
    resultado += key + ':'
    resultado += value + ', '
}
assert resultado == 'a:1, b:2, c:3, '

```

```

resultado = ''
for (key in miMapa.keySet()){
    resultado += key + ':'
    resultado += miMapa[key] + ', '
}
assert resultado == 'a:1, b:2, c:3, '

```

```

resultado = ''
for (value in miMapa.values()){
    resultado += value + ' '
}
assert resultado == '1 2 3 '

```

```

def valor1 = [1, 2, 3].every { it < 5 }
assert valor1

```

```

def valor2 = [1, 2, 3].any { it > 2 }
assert valor2

```

---

Y para terminar con los mapas, vamos a ver otros métodos añadidos por Groovy para el manejo de los mapas, que nos permitirán:

- Crear un submapa de un mapa dado a partir de algunas claves: *subMap()*
- Encontrar todos los elementos de un mapa que cumplen una determinada condición: *findAll()*
- Encontrar un elemento de un mapa que cumpla una determinada condición: *find()*
- Realizar operaciones sobre los elementos de un mapa: *collect()*

---

```

def miMapa = [a:1, b:2, c:3]
def miSubmapa = miMapa.subMap(['a', 'b'])
assert miSubmapa.size() == 2

```

```

def miOtroMapa = miMapa.findAll { entry -> entry.value > 1 }
assert miOtroMapa.size() == 2
assert miOtroMapa.c == 3

```

```

def encontrado = miMapa.find { entry -> entry.value < 3 }
assert encontrado.key == 'a'
assert encontrado.value == 1

```

```

def miMapaDoble = miMapa.collect { entry -> entry.value *= 2 }
//Todos los elementos son pares
assert miMapaDoble.every { item -> item % 2 == 0 }

```

---



## 1.7. Ejercicios

### De Java a Groovy (0.25 puntos)

Modificar la siguiente clase en Java para convertirla en una clase en Groovy y que quede lo más simplificada posible con la misma funcionalidad.

```
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;

public class Todo {
    private String titulo;
    private String descripcion;

    public Todo() {}

    public Todo(String tit, String des) {
        this.titulo = tit;
        this.descripcion = des;
    }

    public String getTitulo(){
        return titulo;
    }

    public void setTitulo(String tit){
        this.titulo = tit;
    }

    public String getDescripcion(){
        return descripcion;
    }

    public void setDescripcion(String des){
        this.descripcion = des;
    }

    public static void main (String[] args){
        List todos = new ArrayList();
        todos.add(new Todo("Lavadora", "Poner lavadora"));
        todos.add(new Todo("Impresora", "Comprar cartuchos impresora"));
        todos.add(new Todo("Películas", "Devolver películas videoclub"));

        for (Iterator iter = todos.iterator();iter.hasNext();) {
            Todo todo = (Todo)iter.next();
            System.out.println(todo.getTitulo()+
"+todo.getDescripcion());
        }
    }
}
```

## GroovyBeans (0.50 puntos)

Crear la clase *Libro* con las propiedades mínimas necesarias que lo describan. La información a almacenar de estos libros será el *nombre* del libro, el *año de edición* y el *autor* del mismo.

Simplemente crearemos un constructor de la clase *Libro* que permita especificar las tres propiedades de la clase (nombre, año de edición y autor). Para comprobar que nuestra clase funciona correctamente, insertaremos los siguientes ejemplos de libros:

- Nombre: La colmena, año de edición: 1951, autor: Cela Trulock, Camilo José
- Nombre: La galatea, año de edición: 1585, autor: de Cervantes Saavedra, Miguel
- Nombre: La dorotea, año de edición: 1632, autor: Lope de Vega y Carpio, Félix Arturo

Posteriormente comprueba que se han insertado correctamente cada uno de los libros indicados anteriormente mediante el uso de sentencias *asserts* y las funciones tipo *getter()* generadas automáticamente en Groovy

A continuación añade una nueva propiedad a la clase *Libro* para almacenar la editorial de los libros. Sin modificar la inserción de libros realizada anteriormente, inserta la editorial de cada uno de los libros utilizando los *setters()* definidos automáticamente por Groovy. Comprueba que los cambios se han efectuado correctamente nuevamente mediante la utilización de *asserts*.

Por último, modifica el comportamiento del `getter()` correspondiente a la propiedad *autor* para que modifique su comportamiento por defecto y devuelva en su lugar el autor del libro con el formato nombre y apellidos.

Como siempre, comprueba que el nuevo método funciona correctamente mediante la utilización de `assert`.

## Expresiones regulares (0.5 puntos)

Crea un script en Groovy que permite agrupar las palabras que aparecen en un párrafo en función del número de letras que contengan. Almacenaremos este catálogo de palabras en un mapa cuya clave será el tamaño y el valor será una lista con todas las palabras de ese mismo tamaño.

El párrafo a analizar será el inicio de *El Ingenioso Hidalgo Don Quijote de la Mancha*:



En un lugar de la Mancha, de cuyo nombre no quiero acordarme, no ha mucho tiempo que vivía un hidalgo de los de lanza en astillero, adarga antigua, rocín flaco y galgo corredor. Una olla de algo más vaca que carnero, salpicón las más noches, duelos y quebrantos los sábados, lentejas los viernes, algún palomino de añadidura los domingos, consumían las tres partes de su hacienda. El resto della concluían sayo de velarte, calzas de velludo para las fiestas con sus pantuflos de lo mismo, los días de entre semana se honraba con su vellori de lo más fino. Tenía en su casa una ama que pasaba de los cuarenta, y una sobrina que no llegaba a los veinte, y un mozo de campo y plaza, que así ensillaba el rocín como tomaba la podadera. Frisaba la edad de nuestro hidalgo con los cincuenta años, era de complexión recia, seco de carnes, enjuto de rostro; gran madrugador y amigo de la caza. Quieren decir que tenía el sobrenombre de Quijada o Quesada (que en esto hay alguna diferencia en

los autores que deste caso escriben), aunque por conjeturas verosímiles se deja entender que se llama Quijana; pero esto importa poco a nuestro cuento; basta que en la narración dél no se salga un punto de la verdad.

Para comprobar si tu script funciona correctamente, puedes añadir estas líneas:

```
.....  
assert palabras[1] == ['y', 'a', 'o']  
assert palabras[2] == ['en', 'un', 'de', 'la', 'no', 'ha', 'su', 'el', 'lo', 'se']  
assert palabras[3] ==  
  ['que', 'los', 'una', 'más', 'las', 'con', 'sus', 'ama', 'así', 'era', 'hay', 'por', 'dél']  
assert palabras[4] ==  
  ['cuyo', 'olla', 'algo', 'vaca', 'tres', 'sayo', 'para', 'días', 'fino', 'casa', 'mozo', 'como', '  
assert palabras[5] ==  
  ['lugar', 'mucho', 'vivía', 'lanza', 'rocín', 'flaco', 'galgo', 'algún', 'resto', 'della', 'mism  
assert palabras[6] ==  
  ['mancha', 'nombre', 'quiero', 'tiempo', 'adarga', 'noches', 'duelos', 'partes', 'calzas', 'sem  
assert palabras[7] ==  
  ['hidalgo', 'antigua', 'carnero', 'sábados', 'viernes', 'velarte', 'velludo', 'fiestas', 'honr  
assert palabras[8] ==  
  ['corredor', 'salpicón', 'lentejas', 'palomino', 'domingos', 'hacienda', 'cuarenta', 'podader  
assert palabras[9] ==  
  ['acordarme', 'astillero', 'añadidura', 'consumían', 'concluían', 'pantuflos', 'ensillaba', '  
assert palabras[10] ==  
  ['quebrantos', 'complexión', 'madrugador', 'diferencia', 'conjeturas']  
assert palabras[11] == ['sobrenombre', 'verosímiles']  
.....
```

## 2. Aspectos avanzados del Lenguaje Groovy. Metaprogramación.

En la sesión anterior hacíamos un repaso completo sobre los tipos de tipos de datos simples y algunas características. Posteriormente hablábamos sobre colecciones y comentábamos las excelencias de los rangos, una tipo de datos propio de Groovy, además de los anteriormente conocidos listas y mapas.

Por último, echábamos un vistazo a las estructuras de control que podíamos utilizar en Groovy y a la combinación de éstas con los nuevos tipos de datos.

En esta sesión, veremos a fondo un nuevo concepto llamado *closure*. Ya hemos comentado algo sobre ellos en la sesión anterior, pero en esta veremos todas sus características.

Una vez vistos los closures, pasaremos a hablar sobre las características de Groovy como lenguaje orientado a objetos y terminaremos viendo un concepto conocido como *metaprogramación* que te permitirá extender la funcionalidad del lenguaje Groovy de forma muy sencilla.

### 2.1. Closures

Aunque en apartados anteriores ya hayamos visto algunos ejemplos de closures, es conveniente dedicarle más tiempo a su explicación, ya que son una de las partes más importantes del lenguaje Groovy y más utilizados en Grails, y al mismo tiempo, puede ser un concepto difícil de entender, ya que no aparece en otros lenguajes de programación. Así que volveremos a ver lo que son los closures, como se declaran y como pueden ser posteriormente referenciados.

Más adelante, pasaremos a ver otros métodos disponibles en los closures y el ámbito de aplicación de los mismos, es decir, quien puede acceder a los mismos. Finalmente, definiremos varias tareas comunes que pueden ser realizadas con los closures, que hasta ahora se hacen de otras formas.

#### Definición de closure



#### Definición

Un *closure* es un trozo de código empaquetado como un objeto y definido entre llaves. Actúa como un método, al cual se le pueden pasar parámetros y pueden devolver valores. Es un objeto normal y corriente al cual se pasa una referencia de la misma forma que se le pasa a cualquier otro objeto.

Posiblemente estés pensando, que de momento los closures no te aportan nada que no puedas hacer nada con cualquier otro lenguaje de programación y posiblemente sea cierto. Sin embargo, los closures nos aportan agilidad a la hora de programar, que es lo que en principio buscamos utilizando un lenguaje como Groovy.

#### Declarando closures

Como comentábamos anteriormente, los closures son bloques de código encerrado entre llaves `{}`. Y para entrar en calor, vamos a definir un closure para imprimir nuestro nombre.

```
def nombre = 'Juan'  
def imprimeNombre = { println "Mi nombre es $nombre" }
```

```
imprimeNombre()

nombre = "Yolanda"
imprimeNombre()
```

En primer lugar, cabe destacar en el ejemplo el uso de una variable dentro del closure definido fuera del ámbito del mismo. Es lo que se conoce *free variable*.

Además, te habrás dado cuenta de que el closure que acabamos de crear no está parametrizado, con lo que si se cambiara el nombre de nuestra variable, el closure no se ejecutaría correctamente. Para definir parámetros en nuestros closures, podemos hacerlo al inicio del mismo introduciendo el nombre de nuestros parámetros (separados por comas si hay más de uno) seguido de los caracteres `→`. El ejemplo anterior parametrizado quedaría así:

```
def imprimeNombre = { nombre -> println "Mi nombre es ${nombre}"}

imprimeNombre("Juan")
imprimeNombre "Yolanda" //Los paréntesis son opcionales

//Con múltiples parámetros
def quintetoInicial = { base, escolta, alero, alapivot, pivot ->
  println "Quinteto inicial compuesto por: $base, $escolta, $alero,
  $alapivot y $pivot"}

quintetoInicial "Calderón", "Navarro", "Jiménez", "Garbajosa", "Pau Gasol"
```

En aquellos closures que sólo tienen un parámetro, es posible obviar su declaración al inicio del closure, puesto que Groovy pone a nuestra disposición la variable *it*. El siguiente ejemplo es idéntico al closure *imprimeNombre* anterior, pero sin declarar sus parámetros.

```
def imprimeNombre = { println "Mi nombre es $it" }

imprimeNombre("Juan")
imprimeNombre "Yolanda"
```

Por último, existe otra forma de declarar un closure y es aprovechando un método ya existente. Con el operador *referencia* `&` podemos declarar un closure a partir de un método de una clase ya creada. El siguiente ejemplo, tenemos la clase *MetodoClosureEjemplo*, en la que existe un método para comprobar si la longitud de la cadena pasada por parámetro es superior a un límite. A partir de este método, crearemos un closure sobre dos instancias de esta clase creadas con límites diferentes. Se puede comprobar como ejecutando el mismo método, obtenemos resultado diferentes.

```
class MetodoClosureEjemplo {
  int limite

  MetodoClosureEjemplo (int limite){
    this.limite = limite
  }

  boolean validar (String valor){
```

```

        return valor.length() <= limite
    }
}

MetodoClosureEjemplo primero = new MetodoClosureEjemplo(8)
MetodoClosureEjemplo segundo = new MetodoClosureEjemplo(5)

Closure primerClosure = primero.&validar

def palabras = ["cadena larga", "mediana", "corta"]

assert "mediana" == palabras.find(primerClosure)
assert "corta" == palabras.find(segundo.&validar)

```

Con la variable *primero* estamos creando una instancia de la clase *MetodoClosureEjemplo* que validará aquellas palabras que tengan como mucho 8 caracteres, mientras que la variable *segundo* validará aquellas palabras con 5 caracteres o menos. Posteriormente, el closure *primerClosure* devolverá la primera palabra encontrada con 8 o menos caracteres y de la lista de palabras coincidiría con la palabra "mediana". En el segundo closure, el que valida las palabras de 5 o menos caracteres, la palabra devuelta sería "corta".

Otra característica interesante de los closures se refiere a la posibilidad de ejecutar diferentes métodos en función de los parámetros pasados y se conoce como *multimétodo*. La idea es crear una clase que sobrecarga un determinado método y posteriormente crear un closure a partir de ese método sobrecargado.

```

class MultimetodoClosureEjemplo{

    int metodoSobrecargado(String cadena){
        return cadena.length()
    }

    int metodoSobrecargado(List lista){
        return lista.size()
    }

    int metodoSobrecargado(int x, int y){
        return x * y
    }
}

MultimetodoClosureEjemplo instancia = new MultimetodoClosureEjemplo()
Closure multiclosure = instancia.&metodoSobrecargado

assert 21 == multiclosure("una cadena cualquiera")
assert 4 == multiclosure(['una', 'lista', 'de', 'valores'])
assert 21 == multiclosure(7, 3)

```

## Los closures como objetos

Anteriormente comentábamos que los closures son objetos y que como tales, pueden ser pasados como parámetros a funciones. Un ejemplo de este caso que ya hemos visto con anterioridad es el método *each()* de las listas, al cual se le puede pasar un closure para realizar una determinada operación sobre cada elemento de la lista.

Si echamos un vistazo al API de Groovy, veremos que el método `each` recibe como parámetro un objeto de tipo *Closure* <http://groovy.codehaus.org/api/org/codehaus/groovy/runtime/DefaultGroovyMethods.html>

```
def quintetoInicial = ["Calderón", "Navarro", "Jiménez", "Garbajosa", "Pau Gasol"]

salida = ''
quintetoInicial.each {
    salida += it + ', '
}
assert salida.take(salida.size()-2) == 'Calderón, Navarro, Jiménez, Garbajosa, Pau Gasol'
```

## Usos de los closures

Ahora que ya sabemos como declarar los closures, vamos a ver como utilizarlos y como podemos invocarlos. Si tenemos definido un *Closure* `x` y queremos llamarlo podemos hacerlo de dos formas:

- `x.call()`
- `x()`

```
def suma = { x, y ->
x + y
}
assert 10 == suma(7,3)
assert 13 == suma.call(7,6)
```

A continuación, veremos un ejemplo sobre como pasar un closure como parámetro a un método. El ejemplo nos permitirá tener un campo de pruebas para comprobar que código es más rápido.

```
def campodepruebas(repeticiones, Closure proceso){
    inicio = System.currentTimeMillis()
    repeticiones.times{proceso(it)}
    fin = System.currentTimeMillis()
    return fin - inicio
}

lento = campodepruebas(999999) { (int) it / 2 }
rapido = campodepruebas(999999) { it.intdiv(2) }

//El método lento es al menos 3 más lento que el rápido
assert rapido * 3 < lento
```

Cuando ejecutamos `campodepruebas(999999)` le estamos pasando el primer parámetro, el que indica el número de repeticiones del código a ejecutar, mientras que el código encerrado entre llaves se corresponde con el *Closure* pasado como segundo parámetro. Cuando definimos un método que recibe como parámetro un closure, es obligatorio que éste sea definido el último parámetro del método.

## Valores por defecto

Hasta el momento, siempre que hemos creado un closure con parámetros, le hemos pasado tantas variables como parámetros tenía el closure. Sin embargo, al igual que en los métodos, es posible establecer un valor por defecto para los parámetros de un closure de la siguiente forma:

```
def suma = { x, y=3 ->
  suma = x + y
}
assert 7 == suma(4,3)
assert 7 == suma(4)
```

## Más métodos de los closures

La clase `groovy.lang.Closure` (<http://groovy.codehaus.org/api/groovy/lang/Closure.html>) es una clase como cualquier otra, aunque es cierto que con una potencia increíble. Hasta ahora, sólo hemos visto la existencia del método `call()`, pero existen muchos más, de los que vamos a ver los más importantes.

Es probable que en alguna ocasión necesites conocer el número de parámetros pasados a un closure para saber como actuar y para ello, los Closures disponen del método `getParameterTypes()`.

```
def llamador (Closure closure){
  closure.getParameterTypes().size()
}

assert llamador { uno -> } == 1
assert llamador { uno, dos -> } == 2
```

Existe una técnica en programación llamada *currying* en honor a su creador *Haskell Brooks Curry*, que consiste en transformar una función con múltiples parámetros en otra con menos parámetros. Un ejemplo puede ser la función que suma dos valores. Si tenemos esta función con dos parámetros, y queremos crear otra que acepte un sólo parámetro, está claro que debe ser perdiendo el segundo parámetro, lo que conlleva a sumar siempre el mismo valor en esta nueva función. El método `curry()` devuelve un clon de la función principal, eliminando uno o más parámetros de la misma.

```
def suma = { x, y -> x + y }
def sumaUno = suma.curry(1)

assert suma(4,3) == 7
assert sumaUno(5) == 6
```

El nuevo closure `sumaUno` siempre toma como segundo parámetro el valor 1.

## Valores devueltos en los closures

Los *closures* tienen dos formas de devolver valores:



- *De forma implícita.* El resultado de la última expresión evaluada por el closure, es lo que éste devuelve. Esto lo que hemos hecho hasta ahora.
- *De forma explícita.* La palabra reservada `return` también nos servirá en los closures para devolver valores

En el siguiente código de ejemplo, ambos closures tienen el mismo efecto, que es la duplicación de los valores de la lista.

```
assert [2,4,6] == [1,2,3].collect { it * 2 }
assert [2,4,6] == [1,2,3].collect { return it * 2 }
```

Si queremos salir de un closure de forma prematura, también podemos hacer uso del `return`. Por ejemplo, si en el ejemplo anterior sólo queremos duplicar aquellos valores impares, deberíamos tener algo así.

```
assert [2,2,6] == [1,2,3].collect {
    if (it%2==1)
        return it * 2
    return it
}
```

## 2.2. Groovy como lenguaje orientado a objetos

Un concepto erróneo que se suele decir de los lenguajes scripts, debido a su dejadez con el tipado de datos y determinadas estructuras de control, es que son lenguajes destinados más a los hackers que a los programadores serios. Esta reputación viene de las primeras versiones del lenguaje Perl, donde la falta de encapsulación y la falta de otras características típicas del modelo orientado a objetos, provocaba un mala gestión del código, con frecuentes trozos de código duplicados e indescifrables fallos de programación.

Sin embargo, este panorama ha cambiado drásticamente en los últimos años, ya que lenguajes como el mismo Perl, Python y más recientemente, Ruby, han añadido características del modelo orientado a objetos, que los hacen incluso más productivos que lenguajes como Java o C++. Groovy también se ha subido al carro de estos lenguajes ofreciendo características similares, con lo que ha pasado de ser un lenguaje de script basado en Java, a ser un lenguaje que nos ofrece nuevas características del modelo orientado a objetos.

Hasta ahora hemos visto que Groovy nos ofrece tipos de datos referencia donde Java simplemente nos ofrecía tipos de datos simples, tenemos rangos y closures, y muchas estructuras de control para trabajar de forma muy ágil y sencilla con colecciones de objetos. Pero esto es sólo la punta del iceberg, y a partir de ahora veremos otras características, que hacen de Groovy un lenguaje con mucho futuro. Empezaremos repasando las clases y los scripts en Groovy, seguiremos por la organización de nuestras clases y terminaremos viendo características avanzadas del modelo orientado a objetos en Groovy.

### Clases y scripts

La definición de clases en Groovy es prácticamente idéntica a como se hace en Java. Las clases se declaran utilizando la palabra reservada `class` y una clase puede tener *campos*, *constructores* y *métodos*. Los métodos y los constructores pueden utilizar *variables locales*.

Por otro lado tenemos los *scripts* que puede contener la definición de variables y métodos, así como la declaración de clases.

La **declaración de variables** debe realizarse antes de que se utilicen. Declarar una variable supone indicar un nombre a la misma y un tipo, aunque en Groovy esto es opcional.

Groovy utiliza los mismos modificadores que Java, que son: *private*, *protected* y *public* para modificar la visibilidad de las variables; *final* para evitar la modificación de variables; y *static* para la declaración de las variables de la clase.

La definición del tipo de la variable es opcional en Groovy y cuando no se especifica, debemos introducir previamente al nombre de la variable, la palabra reserva *def*. Por último y aunque pueda resultar obvio, en Groovy es imposible asignar valores a variables que no coincidan en el tipo. Por ejemplo, un valor numérico no puede ser asignado a una variable definida de tipo *String*. Como vimos anteriormente, Groovy se encarga de hacer el llamado *autoboxing* siempre y cuando sea posible.

Otro aspecto interesante de Groovy es la asignación de valores a las propiedades de las clases. Si hemos definido un campo en una clase en Groovy, podemos acceder al valor del mismo de la forma habitual *objeto.campo* o bien *objeto['campo']*. Esto nos permite una mayor facilidad para acceder a los campos de las clases de forma dinámica, tal y como se hace en el siguiente fragmento de código.

```
class miClase {
    public campo1, campo2, campo3, campo4 = 0
}

def miobjeto = new miClase()

miobjeto.campo1 = 2
assert miobjeto.campo1 == 2

miobjeto['campo2'] = 3
assert miobjeto.campo2 == 3

for(i=1;i<=4;i++)
    miobjeto['campo'+i] = i - 1

assert miobjeto.campo1 == 0
assert miobjeto['campo2'] == 1
assert miobjeto.campo3 == 2
assert miobjeto['campo4'] == 3
```

La **declaración de los métodos** sigue los mismos criterios que acabamos de ver con las variables. Se pueden utilizar los modificadores Java, es opcional devolver algo con la sentencia *return* y si no se utilizan modificadores ni queremos especificar el tipo de dato a devolver, debemos utilizar la palabra reservada *def* para declarar nuestros métodos. Por defecto, la visibilidad de los métodos en Groovy es *public*.

Veamos una clase ejemplo y con ella, algunas diferencias con la misma clase en Java.

```
class MiClase{
```

```
static main(args){
    def algo = new MiClase()
    algo.metodoPublicoVacio()
    assert "hola" == algo.metodoNoTipado()
    assert 'adios' == algo.metodoTipado()
    metodoCombinado()
}

void metodoPublicoVacio(){
    ;
}

def metodoNoTipado(){
    return 'hola'
}

String metodoTipado(){
    return 'adios'
}

protected static final void metodoCombinado(){
}
}
```

---

En la primera sesión comentábamos que el método *main* típico de Java y C, ya no era necesario en Groovy, puesto que podíamos ejecutar nuestro código sin necesidad de incluirlo en dicho método. Sin embargo, si queremos introducir parámetros a nuestro programa, tendremos que utilizarlo, tal y como aparece en el ejemplo. No obstante, este método *main* es algo diferente al de Java, puesto que no es necesario indicarle que el método es público, ya que, por defecto y salvo que se diga que lo contrario, todo método en Groovy es *public*. La segunda diferencia con Java es que los argumentos debían ser del tipo *String[]* mientras que en Groovy simplemente es un objeto y no es necesario especificarle el tipo. Puesto que en la función *main* no se va a devolver nada, es posible obviar la etiqueta *void* en la declaración del método. Resumiendo, mientras que en Java tendríamos esto *public static void main (String[] args)*, en Groovy quedaría algo así *static main (args)*.

Groovy nos ahorra también bastante trabajo en cuanto a la comprobación de errores. En este sentido, cuando intentamos llamar a un método o un objeto cuya referencia es *null*, obtendremos una excepción del tipo *NullPointerException*, lo cual es muy útil para comprobar que nuestro código funciona tal y como debe funcionar. Veamos un ejemplo:

---

```
def mapa = [a:[b:[c:1]]]

assert mapa.a.b.c == 1

//Protección con cortocircuito
if (mapa && mapa.a && mapa.a.x){
    assert mapa.a.x.c == null
}

//Protección con un bloque try/catch
try{
    assert mapa.a.x.c == null
} catch (NullPointerException npe){}
```

```
//Protección con el operador ?.  
assert mapa?.a?.x?.c == null
```

---

En el ejemplo, estamos intentando acceder a una propiedad que no existe como es *mapa.a.x*. Antes de acceder a dicha propiedad, protegemos el acceso para comprobar que no sea *null* y en caso de que no lo sea, acceder a su valor. Aparecen tres tipos de protección de este tipo de errores. La comprobación en cortocircuito con un bloque *if*, es decir, que cuando se detecte una condición de la expresión que sea falsa, nos salimos del *if*. En segundo lugar, intentamos proteger el acceso erróneo con un bloque *try/catch*. Y por último, con el operador *?.*, el cual no es en sí una comprobación y es la opción que menos código emplea.

Por último, es necesario mencionar algo sobre los constructores en Groovy. Los constructores tienen la función de inicializar los objetos de una determinada clase y en caso de que no se especifique ningún constructor para la clase, éstos son creados directamente por el compilador. Nada que no se haga ya en Java. Sin embargo, era extraño que la gente de Groovy no hiciera algo más y así es, hay más.

Existen dos formas de llamar a los constructores de las clases creadas en Groovy. Por un lado, el método tradicional pasando parámetros de forma posicional, donde el primer parámetro significa una cosa, el segundo otra y así sucesivamente. Y por otro lado, tenemos también la posibilidad de pasar los parámetros a los constructores utilizando directamente los nombres de los campos. Esta característica surge debido a que pasar parámetros según su posición tiene el inconveniente de aquellos constructores que tienen demasiados campos y no es sencillo recordar el orden de los mismos.

Cuando creamos un objeto de una clase mediante su constructor, podemos pasarle los parámetros de tres formas diferentes en Groovy:

- Mediante la forma tradicional, pasando parámetros en orden
- Mediante la palabra reservada *as* y una lista de parámetros
- Mediante una lista de parámetros

Veamos un ejemplo:

---

```
class Libro{  
    String titulo, autor  
  
    Libro(titulo, autor){  
        this.titulo = titulo  
        this.autor = autor  
    }  
}  
  
//Forma tradicional  
def primero = new Libro('Groovy in action', 'Dierk König')  
  
//Mediante la palabra reservada as y una lista de parámetros  
def segundo = ['Groovy in action', 'Dierk König'] as Libro  
  
//Mediante una lista de parámetros  
Libro tercero = ['Groovy in action', 'Dierk König']
```

```
assert primero.getTitulo() == 'Groovy in action'  
assert segundo.getAutor() == 'Dierk König'  
assert tercero.titulo == 'Groovy in action'
```

Además del motivo de tener que recordar el orden de los parámetros pasados al constructor de la clase, otra razón para utilizar este tipo de llamadas a los constructores es que podemos ahorrarnos la creación de varios constructores. Imagina el caso en que tengamos dos campos de la clase, y ambos sean opcionales. Esto supone crear cuatro constructores: sin parámetros, con uno de los campos, con el otro campo y con los dos campos. Si utilizamos este tipo de llamadas a los constructores nos ahorraremos la creación de estos cuatro constructores. Veamos el mismo caso que antes con la clase *Libro*. En el ejemplo, dejamos que Groovy cree por nosotros los constructores.

```
class Libro {  
    String titulo, autor  
}  
  
def primero = new Libro()  
def segundo = new Libro(titulo: 'Groovy in action')  
def tercero = new Libro(autor: 'Dierk König')  
def cuarto = new Libro(titulo: 'Groovy in action', autor: 'Dierk König')  
  
assert primero.getTitulo() == null  
assert segundo.titulo == 'Groovy in action'  
assert tercero.getAutor() == 'Dierk König'  
assert cuarto.autor == 'Dierk König'
```

## Organizando nuestras clases y scripts

En este apartado, vamos a ver como podemos organizar nuestras clases y ficheros de la aplicación, y la relación entre ellos. También veremos la utilización de paquetes (*packages*) y el alias de tipos. Comencemos por la relación entre las clases y los ficheros fuentes.

La relación entre los ficheros fuente y las clases en Groovy no es tan estricta como en Java y los ficheros fuente en Groovy pueden contener tantas definiciones de clases como queramos, siguiente una serie de reglas:

- Si el fichero *.groovy* no tiene la declaración de ninguna clase, éste se trata como si fuera un *script* y automáticamente se genera una clase de tipo *Script* con el mismo nombre que el fichero *.groovy*.
- Si el fichero *.groovy* contiene una sola clase definida con el mismo nombre que el fichero, la relación es la misma que en Java, es decir, un fichero *.class* por cada fichero *.groovy*
- Si el fichero *.groovy* contiene más de una clase definida, el compilador de groovy, *groovyc*, creará tantos ficheros *.class* como sean necesarios para cada una de las clases definidas en el fichero *.groovy*. Si quisiéramos llamar a nuestros scripts directamente a través de la línea de comandos, deberíamos añadir el método *main()* a la primera de las clases definidas en el fichero *.groovy*.
- Un fichero Groovy puede mezclar la definición de clases con el código script. En este caso, el código script se convierte en la clase principal a ejecutar, con lo que no se puede declarar una clase con el mismo nombre que el fichero fuente.

Otro aspecto importante para la organización de los ficheros de nuestros proyectos en Groovy es la **organización en paquetes**. En este sentido, Groovy sigue el mismo convenio que Java y su organización jerárquica. De esta forma, la estructura de paquetes se corresponde con los ficheros `.class` de la estructura de directorios.

Sin embargo, como ya se ha comentado en la sesión anterior, no es necesario compilar nuestros archivos `.groovy`, con lo que se añade un problema a la hora de ejecutar dichos archivos. Si no existen los archivos compilados `.class`, ¿dónde los va a buscar? Groovy soluciona este *problema*, buscando también en los archivos `.groovy` aquellas clases necesarias. Así, que el `classpath` no sólo nos va a servir para indicar los directorios donde debe buscar los archivos `.class`, sino también para decirle donde pueden estar los archivos `.groovy`, en caso de que los archivos `.class` no estén. En el caso de que Groovy encuentre en el `classpath`, tanto ambos archivos, `.groovy` y `.class`, se quedará con el más reciente, y así se evitarán los problemas producidos porque se nos haya olvidado compilar nuestro archivo `.groovy`.

Al igual que en Java, las clases Groovy deben especificar su pertenencia a un paquete antes de su declaración. El siguiente fragmento de código muestra un ejemplo de un archivo con dos clases definidas que forman parte de un mismo paquete.

---

```
package negocio

class Cliente {
    String nombre, producto
    Direccion direccion = new Direccion()
}

class Direccion {
    String calle, ciudad, provincia, pais, codigopostal
}
```

---

Para poder utilizar las clases declaradas *Cliente* y *Direccion*, debemos *importar* el paquete correspondiente *negocio*, tal y como aparece en el siguiente ejemplo.

---

```
import negocio.*

def clienteua = new Cliente()
clienteua.nombre = 'Universidad de Alicante'
clienteua.producto = 'Pizarras digitales'

assert clienteua.getNombre() == 'Universidad de Alicante'
```

---

El último aspecto importante a comentar en cuanto a la organización de las clases y los archivos de nuestras aplicaciones, se refiere a la posibilidad de establecer alias a los paquetes importados. Imaginad el caso de tener dos paquetes procedentes de terceras partes que contengan una clase con el mismo nombre. En Groovy podemos solucionar este conflicto de nomenclatura utilizando los alias. Veamos como:

---

```
import agenteexterno1.OtraClase as OtraClase1
import agenteexterno2.OtraClase as OtraClase2

def otraClase1 = new OtraClase1()
```

---

```
def otraClase2 = new OtraClase2()
```

---

## Características avanzadas del modelo orientado a objetos

Ahora que ya tenemos unos conocimientos básicos de lo que Groovy nos permite en cuanto al modelo orientado a objetos, pasemos a ver conceptos algo más avanzados de los vistos hasta ahora, como son la *herencia*, los *interfaces* y los *multimétodos*.

Cuando hablamos de **herencia** en el modelo orientado a objetos, nos referimos a la posibilidad de añadir campos y métodos a una clase a partir de una clase base. Groovy permite la herencia en los mismos términos que lo hace Java. Es más, una clase Groovy puede extender una clase Java y viceversa.

Groovy también soporta el modelo de *interfaces* de Java. En Java, un interface es una clase especial en la que todos sus métodos son abstractos y públicos. Sin embargo, estos métodos no están implementados en la clase interface, sino que esa labor se deja para la clase que implemente esa interface. Los interfaces son lo más parecido a la herencia múltiple, ya que una clase puede implementar más de un interface pero sólo puede extender una clase.

Por último, comentar que en Groovy el tipo de los datos se elige de manera dinámica cuando éstos son pasados como parámetros a los métodos de nuestras clases. Esta característica de Groovy se le conoce como *multimétodo* y lo mejor es que veamos un ejemplo.

---

```
def multimetodo(Object o) { return 'objeto' }
def multimetodo(String o) { return 'string' }

Object x = 1
Object y = 'foo'

assert 'objeto' == multimetodo(x)
assert 'string' == multimetodo(y)//En Java, esta llamada hubiera devuelto
la palabra 'objeto'
```

---

En el ejemplo anterior, el tipo de datos estático de la variable *x* es *Object* mientras que su tipo dinámico es *Integer*, mientras que el tipo estático de *y* es *Object* mientras que su tipo dinámico es *String*. Debido a que Groovy intenta utilizar el tipo dinámico de las variables, en el segundo caso se ejecuta el método que tiene como parámetro una variable de tipo *String*.

## GroovyBeans

En Java, los JavaBeans se introdujeron en su momento para definir un modelo de componentes para la construcción de aplicaciones Java. Básicamente el modelo consiste en una serie de convenciones en cuanto a los nombres que permiten a las clases Java comunicarse unas con otras. Groovy utiliza también el concepto inherente a los JavaBeans, pero lo transforma para dar paso a los GroovyBeans, con unas mejoras particulares, como facilitar el acceso a los métodos. Vayamos paso a paso.

Empecemos por la declaración de los GroovyBeans. Imaginemos que tenemos de nuevo la clase *Libro* con tan solo la propiedad *título*. En Java tendríamos el siguiente *JavaBean*:

---

```
public class Libro implements java.io.Serializable {
    private String titulo;
```



```
public String getTitulo(){
    return titulo;
}

public void setTitulo(String valor){
    titulo = valor;
}
}
```

---

Mientras que en Groovy, simplemente necesitaríamos tener esto otro:

---

```
class Libro implements java.io.Serializable {
    String titulo
}
```

---

Las diferencias son evidentes, ¿no crees? Pero no sólo es una cuestión de ahorro en código, sino también de eficiencia. Imagina simplemente que por cualquier motivo, tuvieras que cambiar el nombre al campo *titulo*, ese simple cambio en Java supondría cambiar el código en tres lugares diferentes, sin embargo, en Groovy, simplemente cambiando el nombre de la propiedad lo tendríamos todo. Además, Groovy también nos ahorra tener que escribir los métodos *setTitulo()* y *getTitulo()*. Esto lo hace únicamente cuando no se han especificado dichos métodos en el GroovyBean. Además, Groovy sabe perfectamente cuando debe especificar un método *set* para acceder a una propiedad. Por ejemplo, si establecemos que una propiedad de nuestro GroovyBean es *final*, esta propiedad solamente será de lectura, así que Groovy no implementa su correspondiente método *set*.

De igual forma, Groovy también permite el acceso a las propiedades utilizando el operador *.*, como por ejemplo *libro.titulo = 'Groovy in action'*. Pero, como siempre, hay algo más. Observemos detenidamente el siguiente ejemplo.

---

```
class Persona {
    String nombre, apellidos

    String getNombreCompleto(){
        return "$nombre $apellidos"
    }
}

def juan = new Persona(nombre:"Juan")
juan.apellidos = "Martínez"

assert juan.nombreCompleto == "Juan Martínez"
```

---

Tenemos el método *getNombreCompleto()* para obtener el nombre completo de la persona en cuestión. Pues Groovy además, crea al vuelo una propiedad equivalente al método *get()*, en nuestro caso, *nombreCompleto*.

En ocasiones, los métodos *get()* no tienen porque devolver el valor concreto del campo en cuestión, sino que es posible que hayan hecho algún tratamiento sobre dicho campo para modificar el valor devuelto. Imaginemos una clase que duplique el valor de su única propiedad cuando se invoca por los métodos tradicionales (bien mediante el método *get* o mediante el operador *.*).



Además, Groovy también nos ofrece la posibilidad de recuperar el valor original del campo sin pasar por su correspondiente getter con el operador `@`, tal y como aparece en la última línea de este ejemplo.

```

class DobleValor {
    def valor

    void setValor(valor){
        this.valor = valor
    }

    def getValor(){
        valor * 2
    }
}

def doble = new DobleValor(valor: 300)

assert 600 == doble.getValor()
assert 600 == doble.valor
assert 300 == doble.@valor

```

## Operador \*

Groovy presenta además, otra característica conocida como el operador *spread* `*`. Este operador permite pasar una lista a un método que contiene una serie de parámetros. De esta forma, se consigue en cierta forma sobrecargar el método en cuestión como si permitiese también la introducción de sus parámetros en forma de lista. Veamos un ejemplo. Imagina que tienes un método que devuelve una lista de resultados, los cuales deben pasados uno por uno a otro método.

```

def getLista(){
    return [1,2,3,4,5]
}

def suma(a, b, c, d, e){
    return a + b + c + d + e
}

assert 15 == suma(*lista)

```

Sencillo, y sin tener que destripar la lista en varios parámetros.

## 2.3. Metaprogramación

### Definición de metaprogramación

Según Wikipedia:



#### Definición de Metaprogramación

La metaprogramación consiste en escribir programas que escriben o manipulen otros programas (o a sí mismos) como datos, o que hacen en

tiempo de compilación parte del trabajo que, de otra forma, se haría en tiempo de ejecución. Esto permite al programador ahorrar tiempo en la producción de código.

Básicamente esto significa que en tiempo de ejecución nuestro programa podrá cambiar su funcionamiento sin necesidad de ser compilado nuevamente.

## Metaprogramación en Groovy

Entre las mejores cualidades de Groovy como lenguaje de programación destaca las facilidades que presenta para realizar dicha metaprogramación. ¿Y cuáles son estas características?

- Todo método de cualquier objeto invocado desde Groovy pasa por un intermediario (MOP - MetaObjectProtocol)
- Posibilidad de interceptar llamadas a métodos o acceso a propiedades
- Crear y modificar métodos en tiempo de ejecución para por ejemplo extender directamente la clase *java.lang.String*

Pero veamos algún ejemplo de lo que podemos hacer con metaprogramación en Groovy.

### Clase Expando

Groovy dispone de una clase llamada *Expando* (<http://groovy.codehaus.org/api/groovy/util/Expando.html>). Esta es una clase especial de Groovy que nos permitirá añadir métodos, constructores, propiedades y métodos estáticos utilizando una sintaxis basada en closures, tal y como demuestra el siguiente ejemplo.

```
def miExpando = new Expando()

miExpando.factor = 5

miExpando.multiplica = { a -> factor * a }

assert miExpando.multiplica(4) == 20

assert miExpando.resto == null
```

¿Mágia? Por supuesto que no. Esto es lo que realmente está pasando.

```
miExpando.factor = 5
//Realmente...
miExpando.setProperty('factor',5)

miExpando.factor
//Realmente...
miExpando.getProperty('factor')

miExpando.multiplica(4)
//Realmente...
miExpando.invokeMethod('multiplica', [4] as Object[])
```

Veamos otro ejemplo de lo que supone la metaprogramación en Groovy:

```
class MiExpando {
    private dynamicProperties = [:]

    void setProperty(String propName, val){
        dynamicProperties[propName] = val
    }

    def getProperty(String propName) {
        dynamicProperties[propName]
    }

    def methodMissing(String methodName, args){
        def prop = dynamicProperties[methodName]
        if (prop instanceof Closure) {
            return prop(*args)
        }
    }
}

def miExpando = new MiExpando()
miExpando.a = 4
miExpando.b = 5
miExpando.suma = { x, y -> x + y }

assert miExpando.a == 4
assert miExpando.b == 5
assert miExpando.suma(4,5)
```

Como podrás imaginar, la nueva clase *MiExpando* que acabamos de crear tiene la misma funcionalidad que la clase de Groovy *Expando*. Por supuesto, la magia reside en los métodos *setProperty(String, Object)*, *getProperty(String)* y *methodMissing(String, Object[])*. Cuando en Groovy se realiza una llamada a un método que no existe, se intenta llamar al método *methodMissing(String, Object[])* con lo que si sobrecargamos ese método vamos a poder hacer cualquier cosa, como por ejemplo invocar un closure o bien escribir un log indicando que alguien ha llamado a un método que no existe.

## Objeto delegate

Todo closure tiene asociado un objeto conocido como *delegate* que puede ser cualquier tipo de objeto y es la forma de tener un objeto que responda a determinadas llamadas a métodos y propiedades.

```
def miclosure = {
    concat " Mundo!"
}

def s = "Hola"
miclosure.delegate = s
assert miclosure.call() == "Hola Mundo!"
```

Pero, ¿qué pasaría si tuviéramos un método *concat(String)*? ¿Cuál de los dos métodos se ejecutaría al realizar la llamada desde el closure?

```
def concat(String arg) {
    return "Concat llamado con arg = $arg"
}

def miclosure = {
    concat " Mundo!"
}

def s = "Hola"
miclosure.delegate = s
assert miclosure.call() == "Concat llamado con arg = Mundo!"
```

Como véis, el método invocado ha sido el que hemos creado con lo que el objeto *delegate* no ha sido contemplado dentro del closure a la hora de invocar el método *concat(String)*. Aquí es donde surge el concepto de la estrategia que tienen los closures para resolver las llamadas a métodos y tenemos cuatro estrategias diferentes: *OWNER\_FIRST*, *DELEGATE\_FIRST*, *OWNER\_ONLY* y *DELEGATE\_ONLY* y en función de que estrategia se utilice, la llamada al método *concat* se resolverá de una forma u otra. Por defecto, la estrategia seguida si no se indica otra es *OWNER\_FIRST*. Comentar también que cada closure tiene un *owner* (*propietario*) y que siempre es aquel objeto o clase que cree el closure.

```
def concat(String arg) {
    return "Concat llamado con arg = $arg"
}

def miclosure = {
    concat " Mundo!"
}

def s = "Hola"
miclosure.delegate = s

assert miclosure.resolveStrategy == Closure.OWNER_FIRST
assert miclosure.call() == "Concat llamado con arg = Mundo!"

miclosure.resolveStrategy = Closure.OWNER_FIRST
assert miclosure.call() == "Concat llamado con arg = Mundo!"

miclosure.resolveStrategy = Closure.DELEGATE_FIRST
assert miclosure.call() == "Hola Mundo!"

miclosure.resolveStrategy = Closure.OWNER_ONLY
assert miclosure.call() == "Concat llamado con arg = Mundo!"

miclosure.resolveStrategy = Closure.DELEGATE_ONLY
assert miclosure.call() == "Hola Mundo!"
```

Comentar por último que, el object *delegate* por defecto de cualquier closure siempre será el propietario del closure.

## Metaclass

Toda clase creada en Groovy tiene un objeto asociado conocido como *metaClass*. Este objeto es del tipo *MetaClassImpl* (<http://groovy.codehaus.org/gapi/groovy/lang/MetaClassImpl.html>)

y gracias a él vamos a poder por ejemplo extender en tiempo de ejecución cualquier clase, como por ejemplo *java.lang.String*.

Habitualmente, en cualquier proyecto con Groovy tendremos una clase llamada *StringUtils* donde solemos meter todo tipo de métodos estáticos para realizar operaciones sobre las cadenas de caracteres. Sin embargo, con Groovy vamos a poder extender la clase *java.lang.String* sin necesidad de crear una nueva clase extendida. Para ello utilizaremos el objeto *metaClass* que, recordad, tienen todas las clases creadas en Groovy.

En el siguiente ejemplo vamos a modificar en tiempo de ejecución la clase *java.lang.String* para añadir un nuevo método que corte aquellas cadenas de caracteres a los primeros 140 caracteres.

```
def textoLargo = """La metaprogramación consiste en escribir programas que
    escriben o manipulan otros programas
(o a sí mismos) como datos, o que hacen en tiempo de compilación parte del
trabajo que, de
otra forma, se haría en tiempo de ejecución. Esto permite al programador
ahorrar tiempo en
la producción de código."""

def textoCorto = "La metaprogramación consiste en escribir programas que
    escriben o manipulan otros programas"

assert textoLargo instanceof java.lang.String
assert textoCorto instanceof java.lang.String

String.metaClass.cortaLosPrimeros140Caracteres = {
    delegate.size() >= 140 ? "${delegate.take(137)}..." : delegate
}

println textoLargo.cortaLosPrimeros140Caracteres()
println textoCorto.cortaLosPrimeros140Caracteres()
```

Además de las clases, los objetos creados en Groovy también tienen el objeto *metaClass* con lo que en ocasiones es posible que nos interese simplemente modificar el comportamiento de un sólo objeto y no de todos los objetos de una determinada clase. Esto nos será de gran utilidad cuando veamos la parte de los tests unitarios en Grails.

```
def texto1 = "texto 1"
def texto2 = "texto 2"

texto1.metaClass.foo = {
    "${delegate}foo"
}

assert texto1.foo() == "texto 1foo"

try {
    assert texto2.foo() == "texto 2foo"
} catch (MissingMethodException mme) {
    println "El método foo no existe"
}
```

Es importante saber, que cuando no necesitemos los métodos especificados en el objeto *metaClass* debemos referenciar dicho objeto a *null* para evitar problemas.

Además de poder crear nuevo métodos, también vamos a poder sobrescribir aquellos ya existentes en la propia clase por nuestra propia implementación. En el siguiente ejemplo vamos a modificar el comportamiento del método *substring(int beginIndex)* de la clase *java.lang.String* para que el índice pasado por parámetro empiece por el número 1.

```
def texto = "En Groovy podemos sobrescribir métodos ya existentes"

def textoZeroBased = texto.substring(0)

String.metaClass.substring = { int beginIndex ->
    delegate[beginIndex-1..delegate.size()-1]
}

def textoOneBased = texto.substring(1)

assert textoOneBased == textoZeroBased

String.metaClass = null
```

Si lo que estamos sobrecargando es un método estático, debemos hacer referencia a la metaclasses de la siguiente forma:

```
String.metaClass.'static'.valueOf = { Boolean b ->
    b ? "false" : "true"
}

assert "false" == String.valueOf(true)
assert "true" == String.valueOf(false)
```

## 2.4. Groovy Builders

Los *Builders* en Groovy se utilizan para hacer que determinadas tareas complejas se conviertan en un juego de niños. Con ellos veremos que la construcción de archivos XML, la automatización de tareas con Ant o el diseño de interfaces gráficas se facilita muchísimo. Veremos como generar archivo XML y JSON gracias a estos builders.

### MarkupBuilder

Los archivos XML son un tipo de archivo ampliamente extendido para el intercambio de información entre aplicaciones, así que Groovy quiere ayudarnos en esa labor, tratando que escribamos el código para construir esos archivos XML de la forma más sencilla y clara posible y el *builder* encargado de esa labor es *MarkupBuilder*, el cual nos ayudará a escribir tanto archivos XML como HTML.

El ejemplo que vamos a ver consiste en crear un archivo XML referido a facturas de una empresa. Cada factura contendrá una serie de ítems, cada uno con un producto.

```
def builder = new groovy.xml.MarkupBuilder()
def facturas = builder.facturas {
```

```
for (dia in 1..3) {
    factura(fecha: new Date() - dia) {
        item(id:dia){
            producto(nombre: 'Teclado', euros:876)
        }
    }
}

facturas
```

Lo que produciría el siguiente archivo XML.

```
<facturas>
  <factura fecha='Sun Jan 03 00:00:00 CET 2014'>
    <item id='1'>
      <producto nombre='Teclado' euros='876' />
    </item>
  </factura>
  <factura fecha='Mon Jan 02 00:00:00 CET 2014'>
    <item id='2'>
      <producto nombre='Teclado' euros='876' />
    </item>
  </factura>
  <factura fecha='Tue Jan 01 00:00:00 CET 2014'>
    <item id='3'>
      <producto nombre='Teclado' euros='876' />
    </item>
  </factura>
</facturas>
```

Se puede comprobar la limpieza del código utilizado para generar el archivo XML y prácticamente, el código sigue el mismo tabulado que el posterior archivo XML, con lo que la lectura del código por terceras personas se facilita muchísimo.

Pero *MarkupBuilder* no sólo nos va a servir para generar archivos XML, sino también archivos HTML. No es raro, puesto que HTML es en el fondo XML. Veamos como construir el mismo ejemplo anterior para que se vea como una página web. Un ejemplo de página web podría ser el siguiente:

```
<html>
  <head>
    <title>Facturas</title>
  </head>
  <body>
    <h1>Facturas</h1>
    <ul>
      <li>Sun Jan 03 00:00:00 CET 2015</li>
      <ul>
        <li>1.- Teclado => 876euros</li>
      </ul>
    </ul>
    <ul>
      <li>Sun Jan 02 00:00:00 CET 2015</li>
```

```
<ul>
  <li>2.- Teclado => 876euros</li>
</ul>
</ul>
<ul>
  <li>Sun Jan 01 00:00:00 CET 2015</li>
  <ul>
    <li>3.- Teclado => 876euros</li>
  </ul>
</ul>
</body>
</html>
```

Este ejemplo de HTML se podría construir con el siguiente código, haciendo uso de MarkupBuilder.

```
def builder = new groovy.xml.MarkupBuilder()

builder.html {
  head {
    title 'Facturas'
  }

  body {
    h1 'Facturas'

    for (dia in 1..3){
      ul{
        li (new Date() - dia).toString()
        ul {
          li "$dia.- Teclado => 876euros"
        }
      }
    }
  }
}
```

El código HTML generado por MarkupBuilder será compatible con los estándares y no tendremos incluso que preocuparnos por la conversión de determinados caracteres como el símbolo < (&lt;), ya que él mismo será el encargado de realizar dicha conversión.

## JsonBuilder

Si en lugar de generar XML queremos generar JSON, Groovy también dispone de otro builder que nos permite generar estos JSON de forma muy sencilla y de forma muy similar a como veíamos con el generador de XML. Veamos un ejemplo:

```
def builder = new groovy.json.JsonBuilder()
def root = builder.teachers {
  professor {
    firstName 'Fran'
    lastName 'Garcia'
    address(
      city: 'Oxford',
```



```
        country: 'UK',
        zip: 12345,
    )
    married true
    modules 'Groovy', 'Grails'
}
}
```

```
assert root instanceof Map
```

```
assert builder.toString() == '{"teachers":{"professor":
{"firstName":"Fran","lastName":"Garcia","address":
{"city":"Oxford","country":"UK","zip":12345},"married":true,"modules":
["Groovy","Grails"]}}}'
```

---

## 2.5. Ejercicios

### Closures sobre colecciones (0.25 puntos)

Implementar un closure que realice la operación matemática factorial del número pasado como parámetro.

A continuación crear una lista o un mapa de números enteros y utilizar algunos de los métodos vistos para recorrer la lista o el mapa, generar el número factorial de cada uno de sus elementos con el closure que hemos creado acabamos de crear y mostrarlo por pantalla.

Por último, crear dos closures llamados *ayer* y *mañana* que devuelvan las fechas correspondientes a los días anterior y posterior a la fecha pasada. Podemos crear fechas mediante la sentencia `new Date().parse("d/M/yyyy H:m:s", "28/6/2008 00:30:20")`.

Posteriormente, crear una lista de fechas y utilizar los closures recién creados para obtener las fechas del día anterior y posterior a todos los elementos de la lista.

### Clase completa en Groovy (0.5 puntos)

Crear una clase llamada *Calculadora* que permita la realización de cálculos matemáticos con dos número enteros. Estos cálculos serán la suma, resta, multiplicación y división.

Los parámetros serán solicitados como entrada a la aplicación. Para leer parámetros por línea de comandos se pueda utilizar el siguiente closure:

```
System.in.withReader {
    print 'Introduzca operador: '
    op1 = it.readLine()
    println(op1)
}
```

El sistema nos debe pedir que introduzcamos los operandos así como el tipo de operación que queremos realizar y mostrará por pantalla el resultado de la operación.

### Metaprogramación (0.5 puntos)

Utilizando los conceptos sobre metaprogramación vistos en la parte teórica, añade un método llamado *moneda(String)* que recibirá por parámetro una cadena que representa un determinado *locale* y en función de este *locale* imprimirá un símbolo u otro. Para simplificar, vamos a limitar el número de posibles *locales* a tres: *es\_ES*, *en\_EN* y *en\_US*. Para comprobar si tu método funciona correctamente, puedes pasarle las siguientes aserciones tras modificar adecuadamente la clase correspondiente:

```
assert 10.2.moneda("en_EN") == "£10.2"
assert 10.2.moneda("en_US") == "\$10.2"
assert 10.2.moneda("es_ES") == "10.2€"

assert 10.moneda("en_EN") == "£10"
assert 10.moneda("en_US") == "\$10"
assert 10.moneda("es_ES") == "10€"

assert new Float(10.2).moneda("en_EN") == "£10.2"
```

```
assert new Float(10.2).moneda("en_US") == "\\$10.2"  
assert new Float(10.2).moneda("es_ES") == "10.2€"  
  
assert new Double(10.2).moneda("en_EN") == "£10.2"  
assert new Double(10.2).moneda("en_US") == "\\$10.2"  
assert new Double(10.2).moneda("es_ES") == "10.2€"
```

---

## 3. Introducción a Grails. Scaffolding.

En esta primera sesión de Grails, comenzaremos, como no puede ser de otra forma, viendo una descripción de lo que es Grails, sus características y las ventajas que conlleva su uso como entorno para el desarrollo rápido de aplicaciones. Seguiremos analizando su arquitectura y las herramientas de código abierto que aúna Grails y terminaremos desarrollando nuestra primera aplicación en Grails aprovechándonos del *scaffolding*.

### 3.1. ¿Qué es?

Grails es un framework para el desarrollo de aplicaciones web basado en el lenguaje de programación Groovy, que a su vez se basa en la Plataforma Java. Grails está basado en los paradigmas *convención sobre configuración* y *DRY (don't repite yourself)* o no te repitas, los cuales permiten al programador olvidarse en gran parte de los detalles de configuración de más bajo nivel.

Como la mayoría de los framework de desarrollo web, Grails está basado en el patrón *Modelo Vista Controlador (MVC)*. En Grails los *modelos* se conocen como *clases de dominio* que permiten a la aplicación mostrar los datos utilizando la *vista*. A diferencia de otros frameworks, en Grails las clases de dominio de Grails son automáticamente persistidas y es incluso posible generar el esquema de la base de datos. Los *controladores* por su parte, permiten gestionar las peticiones a la aplicación y organizar los servicios proporcionados. Por último, la *vista* en Grails son las conocidas como *Groovy Server Pages (GSP)* (análogamente a las Java Server Pages -JSP-) y habitualmente se encargan de generar el contenido de nuestra aplicación en formato HTML.

Como comentábamos anteriormente, Grails permite al programador olvidarse de gran parte de la configuración típica que incluyen los frameworks MVC. Además Grails se aprovecha de un lenguaje dinámico como Groovy para acortar los tiempos de desarrollo y que el equipo de desarrolladores puedan centrarse simplemente en escribir código, actualizar, testear y depurar fallos. Esto hace que el desarrollo de la aplicación sea mucho más ágil que con otros frameworks MVC.

Habitualmente cuando hablamos de framework, se entiende como un marco de los programadores pueden utilizar las técnicas del Modelo Vista Controlador para el desarrollo rápido de sus aplicaciones. Pero, ¿qué pasa con el resto de elementos necesarios para el desarrollo de una aplicación como pueden ser los servidores web o los gestores de bases de datos?. En este sentido, Grails no es simplemente un framework de desarrollo de aplicaciones sino que es más una plataforma completa, puesto que incluye también un contenedor web, bases de datos, sistemas de empaquetado de la aplicación y un completo sistema para la realización de tests. De esta forma, no debemos perder el tiempo buscando y descargando un servidor web para nuestra futura aplicación o un gestor de base de datos. Ni tan siquiera será necesario escribir complicados scripts de configuración para el empaquetado de la aplicación. Todo esto se convierte en una tarea tan sencilla como instalar Grails.

### ¿Qué empresas utilizan Grails?

Desde los inicios de Grails, muchas han sido las empresas que apostaron por este framework y que a día de hoy siguen utilizándolo para desarrollar sus aplicaciones.

- Netflix, <http://www.netflix.com>
- Sky, <http://www.sky.com>
- Secret Escapes, <http://www.secretescapes.com>

- Engage Sciences, <http://www.engagesciences.com>
- Odoobo, <http://www.odobo.com>

## 3.2. Características de Grails

Algunas de las características más importantes que presenta Grails son las siguientes:

### Convención sobre configuración

En lugar de tener que escribir interminables archivos de configuración en formato XML, Grails se basa en una serie de convenciones para que el desarrollo de la aplicación sea mucho más rápido y productivo. Además, gracias al uso de convenciones, se refuerza el otro principio del que hablábamos anteriormente, *DRY (don't repite yourself)* o no te repitas.

### Tests

Una de las partes más importantes en el desarrollo de software se refiere a los tests implementados que garantizan un software de calidad y el fácil mantenimiento de una aplicación. Gracias a estos tests, es muy sencillo detectar y solucionar fallos provocados por cambios en el código. Cada vez que se genera en Grails una clase de dominio o un controlador, paralelamente es generado también un test para comprobar la nueva clase o controlador, que por supuesto nosotros deberemos completar (Grails es muy bueno, pero no lo hace todo).

Grails distingue entre tests unitarios y tests de integración. Los tests unitarios son tests sin dependencias de ningún tipo, salvo algún que otro objeto *mock*. Por otro lado, los tests de integración tienen acceso completo al entorno de Grails, incluyendo la base de datos. Además, Grails permite también la creación de tests funcionales para comprobar la funcionalidad de nuestra aplicación web a nivel de interacción con el usuario.

### Scaffolding

El scaffolding es una característica de determinados frameworks que permite la generación automática de código para las cuatro operaciones básicas de cualquier aplicación, que son la *creación, lectura, edición y borrado*, lo que en inglés se conoce como *CRUD (create, read, update and delete)*.

Grails permite también utilizar scaffolding en nuestras aplicaciones a través de un plugin que viene instalado por defecto. En versiones anteriores, el scaffolding venía como parte del *core* de Grails, sin embargo en las últimas versiones esta característica ha sido movida a un plugin. El scaffolding en Grails se consigue escribiendo muy pocas líneas de código, con lo que podemos centrarnos en especificar las propiedades, comportamientos y restricciones de nuestras clases de dominio.

Una gran cantidad de aplicaciones consisten en simplemente ofrecer la posibilidad a los usuarios de realizar esas cuatro operaciones básicas que comentábamos anteriormente sobre algunas entidades o clases de dominio con lo que en ocasiones este scaffolding nos ahorrará mucho tiempo.

### Mapeo objeto-relacional

Grails incluye un potente framework para el mapeo objeto-relacional conocido como *GORM (Grails Object Relational Mapping)*. Como cualquier framework de persistencia, GORM permite mapear objetos contra bases de datos relacionales y representar relaciones entre dichos objetos del tipo *uno-a-uno, uno-a-muchos y muchos-a-muchos*.

Por defecto, Grails utiliza Hibernate por debajo para realizar este mapeo, con lo que no debemos preocuparnos de que tipo de base de datos vamos a utilizar en nuestra aplicación (MySQL, Oracle, SQL Server, etc) porque Grails se encargará por nosotros de esta parte. Además, en los últimos tiempo están proliferando lo que se conoce como las bases de datos *noSQL* como MongoDB, Redis, Neo4j, etc. Aunque éstos tipos de bases de datos todavía no están soportadas en el núcleo de Grails, si están apareciendo al mismo tiempo diversos plugins para la utilización de éstas sin apenas modificar nada en nuestra aplicación.

## Plugins

Sin embargo, Grails no siempre es la solución a cualquier problema que se nos pueda plantear en el desarrollo de aplicaciones web. Para ayudarnos, Grails dispone de una arquitectura de plugins con una comunidad de usuarios detrás (cada vez más grande) que ofrecen plugins para seguridad, AJAX, testeo, búsqueda, informes, servicios web, etc. Este sistema de plugins hace que añadir complejas funcionalidades a nuestra aplicación se convierte en algo muy sencillo. En la actualidad hay alrededor de 1200 plugins desarrollados para Grails.

## Internacionalización (i18n)

Cuando creamos un proyecto en Grails, automáticamente se crea toda la estructura de datos necesaria para la internacionalización de la aplicación sin ningún problema.

## Software de código abierto

Por suerte, Grails no sufre del síndrome *Not Invented Here (NIH)*<sup>1</sup> y lo hace integrando en su arquitectura las mejores soluciones de software libre del mercado para crear un framework robusto. Veamos cuales son estas soluciones.

## Groovy

**Groovy**<sup>2</sup> es la parte fundamental en la que se basa Grails. Como vimos en las dos primeras sesiones, Groovy es un potente y flexible lenguaje de programación. Su integración con Java, las características como lenguaje dinámico y su sintaxis sencilla, hacen de este lenguaje de programación el compañero perfecto para Grails.

## Framework Spring

El framework **Spring**<sup>3</sup> ofrece un alto nivel de abstracción al programador que en lugar de tratar directamente con las transacciones, proporciona una forma para declarar dichas transacciones utilizando los *POJOs* (*Plain Old Java Objects*), con lo que el programador se puede centrar en la implementación de la lógica de negocio.

## Hibernate

**Hibernate**<sup>4</sup> es un framework de persistencia objeto-relacional y constituye la base de GORM. Es capaz de mapear complejas clases de dominio contra las tablas de una base de datos, así como establecer las relaciones entre las distintas tablas.

---

<sup>1</sup> [http://en.wikipedia.org/wiki/Not\\_Invented\\_Here](http://en.wikipedia.org/wiki/Not_Invented_Here)

<sup>2</sup> <http://groovy.codehaus.org/>

<sup>3</sup> <http://www.springsource.org>

<sup>4</sup> <https://www.hibernate.org>

## SiteMesh

[SiteMesh](#)<sup>5</sup> es un framework web para el renderizado de documentos HTML que implementa el patrón de diseño [Decorator](#)<sup>6</sup> con componentes como cabeceras, pies de páginas y sistemas de navegación.

## Tomcat y Jetty

Como comentábamos anteriormente, Grails es una plataforma completa, y esto es en parte gracias a la inclusión de un servidor web como [Jetty](#)<sup>7</sup> o [Tomcat](#)<sup>8</sup>. Esto no significa que Grails solo funcione con estos dos servidores de aplicaciones sino que en nuestra instalación local de Grails podremos utilizar cualquiera de ellos.

## H2

Grails también incluye un gestor de bases de datos relacionales como [H2](#)<sup>9</sup>. H2 es un gestor de base de datos en memoria de tamaño muy reducido y que en la mayoría de los casos nos servirá para nuestra fase de desarrollo (nunca para producción). Esto tampoco quiere decir que nuestras aplicaciones desarrolladas en Grails sólo funcionen con H2, sino que para desarrollar nuestra aplicación no vamos a necesitar ningún gestor de base de datos externo.

## Spock

En las primeras versiones de Grails, se utilizaba como framework para el desarrollo de tests el ampliamente aceptado JUnit, pero en las últimas versiones se ha decidido utilizar otro framework que ha tenido gran aceptación por parte de la comunidad llamado [Spock](#)<sup>10</sup>. Spock nos permitirá realizar tests unitarios, de integración y funcionales, tal y como veremos en la sesión 6.

## Gant

[Gant](#)<sup>11</sup> es un entorno en línea de comandos que envuelve el archiconocido sistema de automatización de tareas [Apache Ant](#)<sup>12</sup>. Con Gant vamos a poder realizar todo tipo de tareas en un proyecto Grails, como la creación de todo tipo de artefactos en la aplicación, la generación del código de scaffolding necesario, la realización de las pruebas unitarias y de integración y el empaqueta de la aplicación, entre otras cosas.

## 3.3. Arquitectura

Ahora que ya tenemos una idea de lo que es Grails y las ventajas que nos puede aportar a la hora de desarrollar nuestros proyectos web, veamos gráficamente su arquitectura.

---

<sup>5</sup> <http://www.opensymphony.com/sitemesh>

<sup>6</sup> [http://es.wikipedia.org/wiki/Decorator\\_\(patr%C3%B3n\\_de\\_dise%C3%B1o\)](http://es.wikipedia.org/wiki/Decorator_(patr%C3%B3n_de_dise%C3%B1o))

<sup>7</sup> <http://www.mortbay.org/jetty>

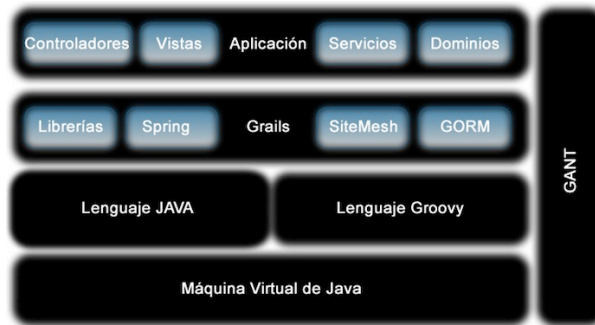
<sup>8</sup> <http://tomcat.apache.org>

<sup>9</sup> <http://www.h2database.com>

<sup>10</sup> <http://spock-framework.readthedocs.org>

<sup>11</sup> <http://gant.codehaus.org>

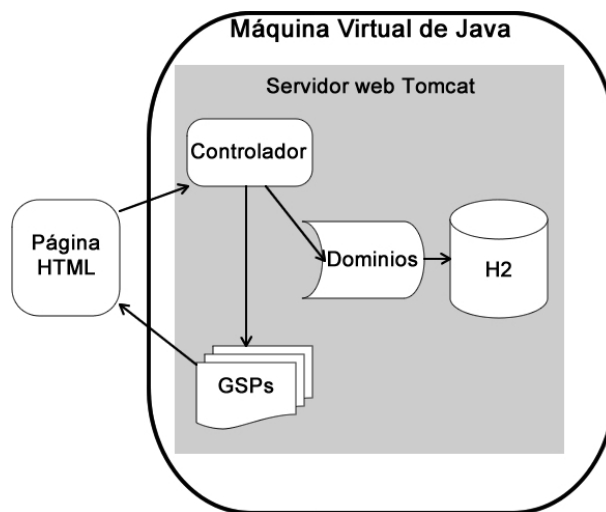
<sup>12</sup> <http://ant.apache.org>



Podemos decir que la arquitectura de Grails está formada por cuatro capas claramente diferenciadas. La base de la arquitectura de Grails es la máquina virtual de Java. Por encima de ésta se encuentran los lenguajes de programación en lo que está basado Grails, Java y Groovy. En la tercera capa tenemos el propio framework Grails, al cual le acompañan todos los frameworks de los que hablábamos en la sección anterior, SiteMesh, Spring, GORM, etc. En esta capa también se ha añadido una opción abierta como es la posibilidad de incluir otras librerías externas que no están incluidas en Grails. La última capa está formada por la aplicación en sí, siguiendo el patrón modelo vista controlador.

Además, envolviendo a todo el proyecto Grails aparece Gant, la herramienta en línea de comandos que nos permitirá una gran variedad de acciones sobre el mismo.

Desde el punto de vista de la ejecución de un proyecto Grails, se podría esquematizar de la siguiente forma:



De la imagen podemos concluir que una página web realiza una petición al servidor web. Esta petición se pasa a un controlador, el cual podrá utilizar o no una clase de dominio (modelo). Esta clase de dominio puede ser a su vez estar persistida en una base de datos gracias a GORM. Una vez el controlador termina, pasa la petición al correspondiente GSP para que renderice la vista y sea enviada de nuevo al navegador en forma de página HTML.

### 3.4. Instalación de Grails

Si tenemos en cuenta lo que decíamos anteriormente de que Grails es un completo framework con un servidor web, un gestor de base de datos y el resto de características ya comentadas,



podríamos pensar que su instalación puede ser muy complicada. Sin embargo, la instalación de Grails se convierte en un juego de niños, ya que nos olvidamos de tener que buscar soluciones externas para cada uno de los aspectos relacionados con el desarrollo de una aplicación web.

Aquí vamos a configurar Grails para poder utilizarlo tanto desde línea de comandos como desde el entorno de desarrollo IntelliJ IDEA.

## Instalación de Grails en línea de comandos

Tal y como hacíamos en la instalación de Groovy, vamos a utilizar el gestor de paquetes *gvmtool* (<http://gvmtool.net/>). Estos son los pasos para instalar Grails en nuestro ordenador:

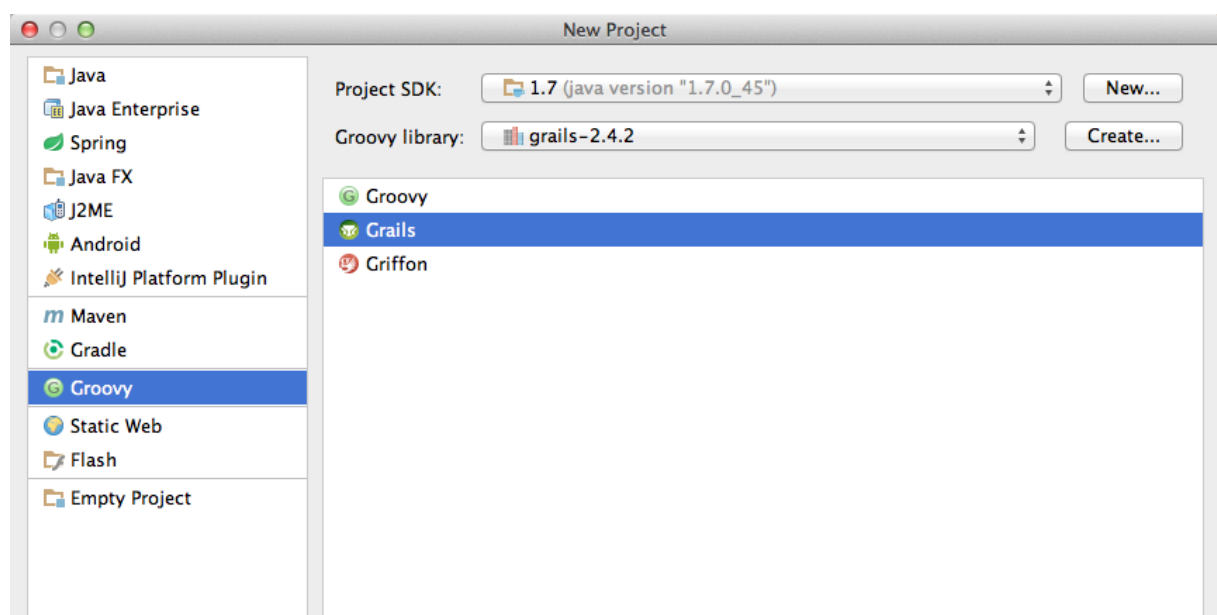
```
curl -s get.gvmtool.net | bash  
  
gvm list grails  
  
gvm install grails 2.4.4
```

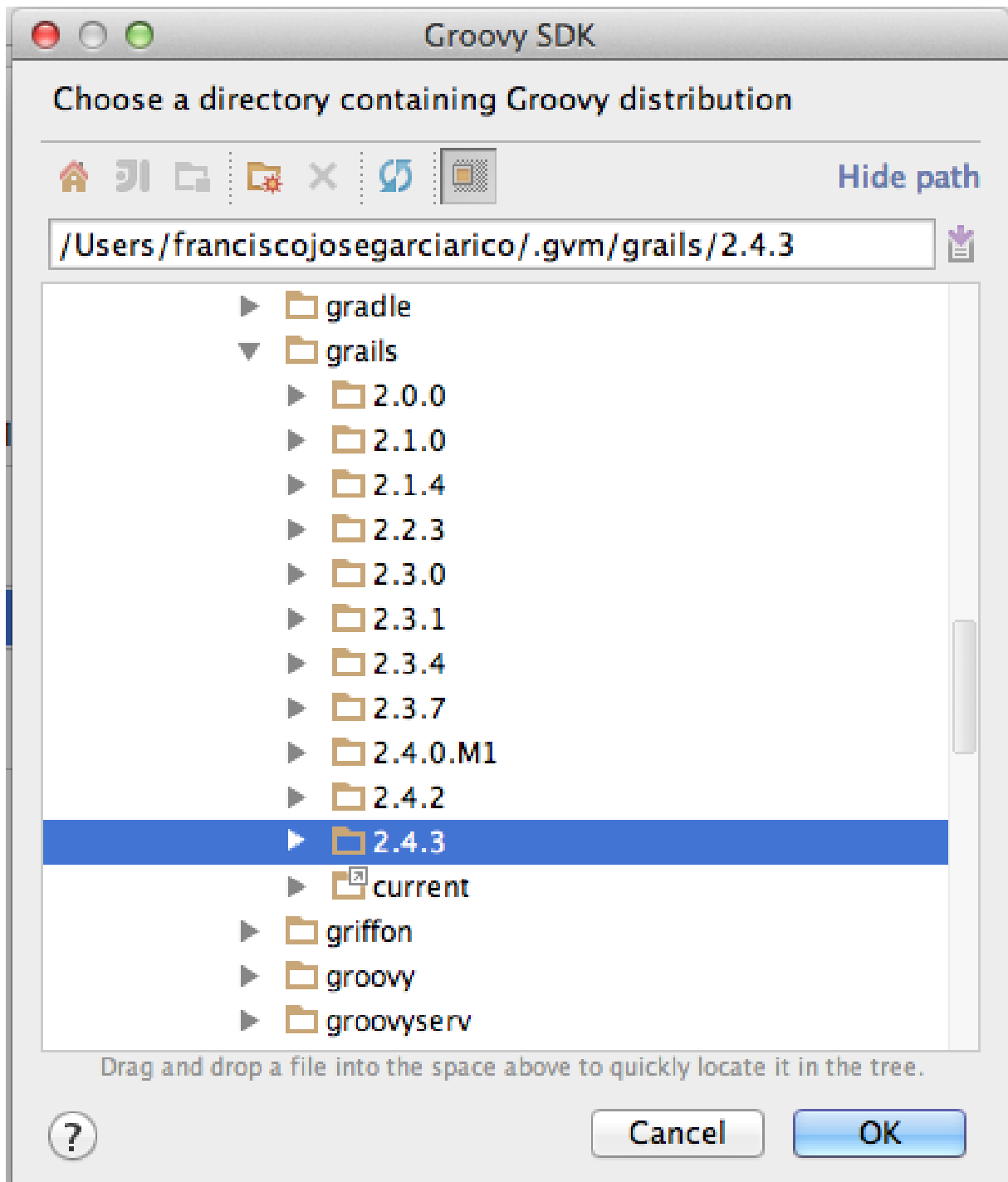
Si necesitáramos alternar entre varias versiones de Grails, simplemente deberías ejecutar:

```
gvm use grails 2.1.0
```

## Instalación en IntelliJ IDEA

En principio IntelliJ IDEA ofrece automáticamente la posibilidad de crear proyectos Grails gracias al plugin instalado por defecto. Puedes comprobarlo en las preferencias de *IntelliJ IDEA* > *IDE Settings* > *Plugins* y ahí buscar el plugin de Grails. Cuando intentemos crear un nuevo proyecto en Grails, IntelliJ nos pedirá que le indiquemos donde se encuentra el framework instalado.





### 3.5. Scaffolding

Una vez ya tenemos configurado nuestro entorno de desarrollo, pasemos a la acción desarrollando con Grails una sencilla aplicación. En este módulo, vamos a desarrollar un ejemplo de lo que sería el completo desarrollo de una aplicación web que nos permita manejar una lista de tareas. Empezaremos desarrollando este ejemplo de aplicación para explicar como Grails implementa el scaffolding. Pero, ¿qué es el scaffolding?

El scaffolding es un término utilizado en programación para designar la construcción automática de aplicaciones a partir del esquema de la base de datos. Está soportado por varios frameworks MVC y Grails no podía ser menos y el equipo de desarrollo decidió incluirlo

entre sus características más importantes. La idea del scaffolding es, partiendo del esquema de la base de datos, generar el código necesario para implementar las cuatro operaciones básicas en cualquier aplicación, que son: creación, lectura, actualización y borrado. Este tipo de aplicaciones se las conoce como *CRUD* (*create, read, update y delete*).

Grails además nos ofrece dos tipos de scaffolding, el *dinámico*, en el cual Grails genera por nosotros al vuelo el código referente a las vistas y a los controladores y por otro lado el *scaffolding estático* en el que Grails generará de forma estática el código de controladores y vistas.

Como comentábamos anteriormente, vamos a ver un ejemplo de lo que es el scaffolding, desarrollando la típica aplicación *todo*, aplicación que iremos mejorando durante todo este módulo.

## Descripción de la aplicación ejemplo

Básicamente el funcionamiento de la aplicación debe permitir la creación y mantenimiento de tareas así como la posibilidad de categorizarlas. Para simplificar la aplicación, una tarea sólo podrá pertenecer como mucho a una categoría mientras que una categoría podrá tener por supuesto muchas tareas.

Además, también vamos a permitir la posibilidad etiquetar dichas tareas, con lo que una tarea puede tener varias etiquetas y una etiqueta estar asignada a varias tareas.

Por el momento, no nos vamos a preocupar de la gestión de usuarios, ya que lo haremos en la sesión 7 utilizando un plugin que se encargará por nosotros de todo el tema de seguridad de nuestra aplicación.

## Creación del proyecto Grails

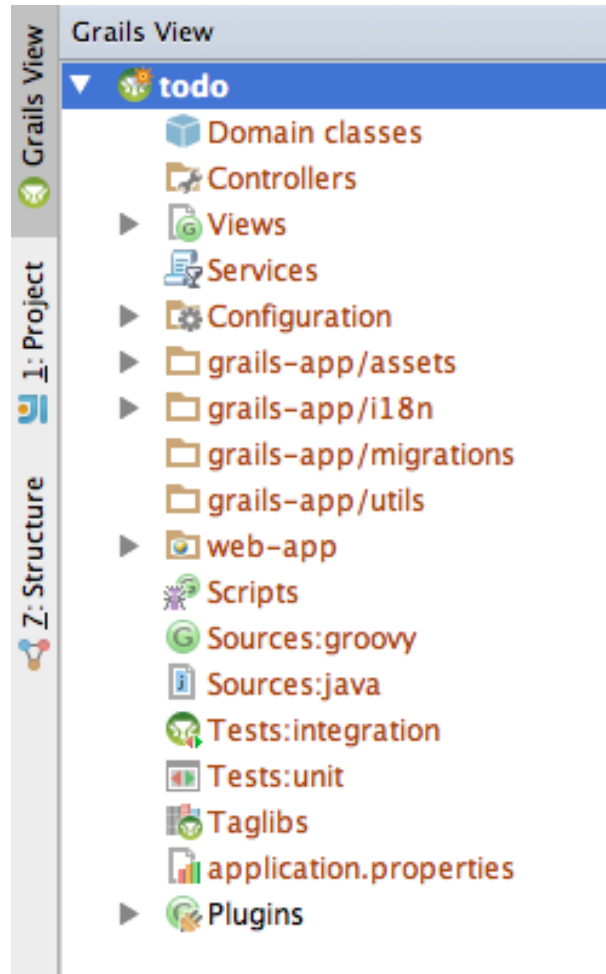
Si todo ha ido bien en la instalación de Grails, podremos crear nuestra primera aplicación en Grails gracias al comando *grails*. Este será el comando que utilizaremos a lo largo de todo el curso para crear todas las partes de nuestra aplicación. Para ver un listado completo de las opciones del comando *grails*, podemos ejecutar *grails help*. Si echamos un primer vistazo a esta listado, descubriremos la opción *create-app*, la cual nos servirá para crear la estructura de directorios de la aplicación. Ejecutamos *grails create-app todo* y se nos generará el directorio *todo* con una serie de subdirectorios que comentaremos en breve.

Esta estructura de directorios generada automáticamente por Grails viene como consecuencia de lo que comentábamos anteriormente como uno de los paradigmas en que se basa Grails, *convención sobre configuración*. De esta forma, Grails nos genera la estructura de directorios que albergará todo nuestro proyecto para que nosotros lo vayamos *completando*. Veamos para que sirven los directorios más importantes generados dentro de nuestro proyecto *todo*.

Directorio	Descripción
grails-app/conf	Ficheros de configuración de la aplicación
grails-app/conf/hibernate	Archivos de mapeado de Hibernate
grails-app/conf/spring	Archivos de mapeado de Spring
grails-app/controllers	Controladores de la aplicación que gestionan las peticiones
grails-app/domain	Clases de dominio del modelo

Directorio	Descripción
grails-app/i18n	Mensajes para la internacionalización de la aplicación
grails-app/services	Servicios
grails-app/taglib	Librerías de etiquetas dinámicas
grails-app/utils	Utilidades específicas de Grails
grails-app/views	Archivos GSP
grails-app/views/layout	Archivos de diseño de las páginas web
grails-app/assets/images	Imágenes utilizadas por la aplicación en el entorno del plugin pipeline
grails-app/assets/javascripts	Archivos javascript utilizados por la aplicación en el entorno del plugin pipeline
grails-app/assets/stylesheets	Hojas de estilo utilizadas por la aplicación en el entorno del plugin pipeline
lib	Archivos JAR de terceras partes, tales como controladores de bases de datos
scripts	Scripts <i>GANT</i> para el automatizado de tareas
src/java	Archivos fuente adicionales en Java
src/groovy	Archivos fuente adicionales en Groovy
test/integration	Tests de integración
test/unit	Tests unitarios
web-app	Artefactos web que finalmente serán comprimidos a un <i>WAR (Web Application Archive)</i>
web-app/css	Hojas de estilo
web-app/images	Imágenes de la aplicación
web-app/js	Javascript
web-app/WEB-INF	Archivos de configuración para Spring o SiteMesh

Ahora que ya tenemos una primera idea de lo que significa cada uno de los directorios generados por Grails al crear un proyecto, vamos a abrirlo con el editor IntelliJ IDEA. Grails es capaz de crear también los archivos de configuración necesarios para que podamos editar el proyecto con diferentes editores como Eclipse, Textmate y NetBeans, con lo que simplemente abrimos el editor y localizamos el proyecto *todo* y lo abrimos como un nuevo proyecto. Como vemos en la imagen, IDEA ha cambiado la estructura de directorios por una más clara para el desarrollador, aunque siempre podemos ver la estructura real de directorios si accedemos a la vista *Project*.



Nuestro proyecto ya está listo para ser ejecutado, aunque imaginarás que por el momento no hará nada. Para ver la primera versión del proyecto *todo* podemos ejecutar el comando *grails run-app*, que nos generará una aplicación en la dirección <http://localhost:8080/todo>. El comando *grails run-app* lo que ha hecho es crear una instancia del servidor web Tomcat en el puerto 8080 y cargar en él la aplicación *todo*. Por ahora esta aplicación hace más bien poco y simplemente nos muestra un mensaje de bienvenida, pero vamos a ver lo sencillo que es generar su contenido.

## Creación de clases de dominio

Al desarrollar una aplicación de este tipo, lo habitual es crear en primer lugar las clases de dominio necesarias para después pasar a generar los controladores, así que vamos a empezar creando la clase de dominio referente a los *todos*. En Grails tenemos el comando *grails create-domain-class* para generar una determinada clase de dominio, con lo que si ejecutamos *grails create-domain-class es.ua.expertojava.todo.todo*, Grails nos creará la estructura necesaria para las tareas de nuestra aplicación. En caso de no indicar el nombre de la clase, el sistema nos preguntará por él.

Como podemos comprobar en nuestro proyecto con IDEA, se nos ha creado una clase de dominio llamada *Todo*, que como ves empieza por mayúscula a pesar de que nosotros introdujimos el nombre de la clase de dominio en minúsculas. Esto es debido a que Grails sigue una serie de convenios para el nombre de las clases. El contenido de *Todo.groovy* es el siguiente:

---

```
package es.ua.expertojava.todo

class Todo {

    static constraints = {
    }
}
```

---

Además, también se crea un test unitario llamado *es.ua.expertojava.todo.TODOSpec*, que de momento dejaremos tal cual está pero que volveremos a él en la sesión 6 en la que hablaremos de los tests unitarios.

---

```
package es.ua.expertojava.todo

import grails.test.mixin.TestFor
import spock.lang.Specification

/**
 * See the API for {@link
 * grails.test.mixin.domain.DomainClassUnitTestMixin} for usage instructions
 */
@TestFor(Todo)
class TODOSpec extends Specification {

    def setup() {
    }

    def cleanup() {
    }

    void "test something"() {
    }
}
```

---

La clase de dominio *Todo* está preparada para que le añadamos los campos necesarios, tales como título, descripción, fecha, etc. La información referente a las tareas serán *título*, *descripción*, *fecha*, *fechaRecordatorio* y *url*. Con estos datos, la clase de dominio *Todo* quedaría así:

---

```
package es.ua.expertojava.todo

class Todo {
    String title
    String description
    Date date
    Date reminderDate
    String url

    static constraints = {
        title(blank:false)
        description(blank:true, nullable:true, maxSize:1000)
        date(nullable:false)
        reminderDate(nullable:true)
    }
}
```

---

```

        url(nullable:true, url:true)
    }

    String toString(){
        title
    }
}

```

Como puedes comprobar, en ningún momento se ha indicado ninguna propiedad como clave primaria y esto es debido a que Grails añade siempre las propiedades *id* y *version*, los cuales sirven respectivamente como clave primaria de la tabla en cuestión y para garantizar la integridad de los datos. La propiedad *version* es un mecanismo utilizado en *Hibernate* para el bloqueo de las tablas y que evita que se produzcan inconsistencias en los datos.

Por otro lado, la clase *Todo* no sólo contiene propiedades sino que también se han añadido una serie de restricciones (*constraints*) que deben cumplir dichas propiedades para que una tarea se pueda insertar en la base de datos. Por ejemplo, la propiedad *title* no puede dejarse en blanco, la propiedad *date* no puede ser *null*, mientras que las propiedades *description*, *reminderDate* y *url* pueden tener un valor *null* y además ésta última, en caso de contener valor, debe ser una dirección url bien formada. Además, observa también como la descripción no puede exceder de 1000 caracteres.

## Creación de controladores

Ahora que tenemos nuestra primera clase de dominio creada, necesitamos un controlador que gestione todas las peticiones que le lleguen a esta clase de dominio. El controlador es el encargado de gestionar la interacción entre la vista y las clases de dominio, con lo que podemos decir sin género de dudas que es la parte más ardua del sistema. Sin embargo, gracias a que Grails nos permite utilizar *scaffolding* esto se convierte de nuevo en un juego para niños.

Para crear un controlador en Grails, debemos ejecutar el comando *grails create-controller* y se generará un nuevo controlador en el directorio *grails-app/controllers*, así como un test unitario en el directorio *test/unit/<paquete>*. Antes de seguir vamos a crear el controlador de la clase *Todo* ejecutando *grails create-controller es.ua.expertojava.todo.todo*, el cual creará el archivo *grails-app/controllers/es/ua/expertojava/todo/ToDoController.groovy* con el siguiente contenido

```

package es.ua.expertojava.todo

class ToDoController {

    def index() { }
}

```

Además, tal y como pasaba cuando creábamos la clase de dominio, Grails crea por nosotros un test unitario llamado *es.ua.expertojava.todo.ToDoControllerSpec* que nos permitirá más adelante testear el controlador.

Para poder utilizar *scaffolding* de la clase *Todo* simplemente debemos cambiar la línea

```

def index = {}

```

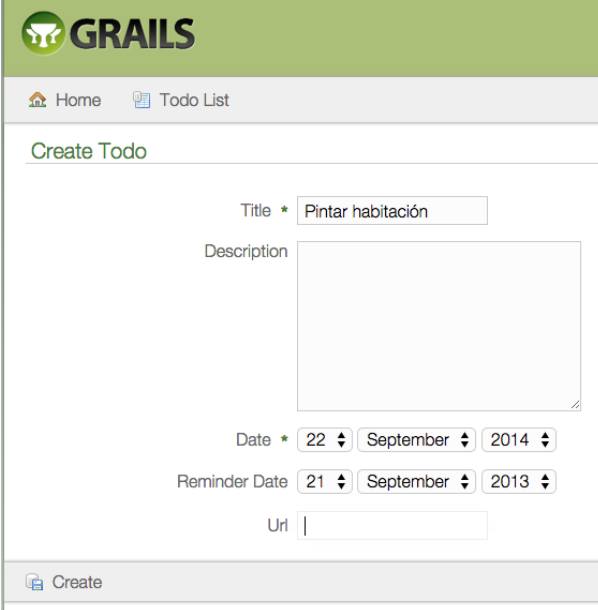
por el siguiente código

```
def scaffold = Todo
```

o bien por

```
def scaffold = true
```

Si actualizamos la web de la aplicación veremos como nos aparece un nuevo enlace en el que podremos controlar las tareas de nuestra aplicación. El scaffolding de Grails ha creado por nosotros los cuatro métodos necesarios para la gestión de las tareas *creación*, *lectura*, *edición* y *borrado*. En la siguiente imagen podemos ver el formulario para añadir una nueva tarea.



The screenshot shows a web browser window with the Grails logo and navigation links for 'Home' and 'Todo List'. The main content area is titled 'Create Todo'. It contains a form with the following fields:

- Title \***: A text input field containing 'Pintar habitación'.
- Description**: A large text area for entering details.
- Date \***: A date picker showing '22', 'September', and '2014'.
- Reminder Date**: A date picker showing '21', 'September', and '2013'.
- Url**: An empty text input field.

At the bottom of the form is a 'Create' button.

Si nos fijamos bien, observaremos que el orden para el formulario de creación de una nueva tarea sigue el mismo orden en el que aparecen definidas las restricciones para la validación de los datos. También es interesante ver como para la propiedad *description* ha creado un *textarea* en lugar de un campo *text* como ha hecho con el resto de propiedades de tipo *String*. Esto es porque para el campo *description* hemos establecido la restricción del tamaño máximo.

Probemos ahora a insertar tareas que no cumplan alguna de las restricciones que le impusimos en la definición de la clase de dominio. Si por ejemplo intentamos insertar una nueva tarea sin especificar su *título* el sistema nos proporcionará un error indicando que el campo en cuestión no puede estar vacío. Además, el campo de texto aparece remarcado en rojo para que el usuario sepa que ahí está pasando algo no permitido.

Si solucionamos este problema con el *título*, se insertará una nueva tarea en la base de datos de nuestra aplicación, que posteriormente podremos visualizar, editar y eliminar.

Continúemos con nuestro pequeño ejemplo de scaffolding, y para ello vamos a definir también la clase de dominio y el controlador para las *categorías* de las *tareas*. Empecemos creando las clases de dominio necesarias y empecemos por las *categorías* de las *tareas*. Para ello ejecutamos el comando `grails create-domain-class es.ua.expertojava.todo.category`. De estas categorías necesitamos conocer su *nombre* y una *descripción*.



---

```
package es.ua.expertojava.todo

class Category {

    String name
    String description

    static hasMany = [todos:Todo]

    static constraints = {
        name(blank:false)
        description(blank:true, nullable:true, maxSize:1000)
    }

    String toString(){
        name
    }
}
```

---

Lo novedoso en la declaración de la clase *Category* es la aparición de la especificación de la relación entre ésta y la clase *Todo*. Si pensamos en la relación existente entre las tareas y las categorías, está claro que esta relación es del tipo uno a muchos, es decir, que una categoría puede tener muchas tareas y que una tarea sólo pertenecerá por simplicidad a una categoría. Teniendo esto en cuenta, debemos especificar en la clase de dominio *Todo* la otra parte de la relación. Esto lo vamos a hacer añadiendo una nueva propiedad estática en la clase de dominio del siguiente modo:

---

```
package es.ua.expertojava.todo

class Todo {
    String title
    String description
    Date date
    Date reminderDate
    String url
    Category category

    static constraints = {
        title(blank:false)
        description(blank:true, nullable:true, maxSize:1000)
        date(nullable:false)
        reminderDate(nullable:true)
        url(nullable:true, url:true)
        category(nullable:true)
    }

    String toString(){
        "title"
    }
}
```

---

Una vez ya tenemos definidas la clase de dominio *Category*, podemos crear el controlador e indicarle que queremos utilizar el scaffolding para gestionar su información. Para ello ejecutamos `grails create-controller es.ua.expertojava.todo.category`.

---

```
package es.ua.expertojava.todo

class CategoryController {

    def scaffold = Category
}

```

---

Por último, vamos a añadir la clase de dominio referente a las etiquetas que vamos a poder asignar a las tareas. Ejecutamos *grails create-domain-class es.ua.expertojava.todo.tag*:

---

```
package es.ua.expertojava.todo

class Tag {

    String name

    static hasMany = [todos:Todo]

    static constraints = {
        name(blank:false, nullable:true, unique:true)
    }

    String toString(){
        name
    }
}

```

---

Como puedes observar, una etiqueta puede estar asignada a muchas tareas. De igual forma, una tarea podrá tener muchas etiquetas. Más adelante veremos a fondo como se deben establecer este tipo de relaciones.

---

```
package es.ua.expertojava.todo

class Todo {
    String title
    String description
    Date date
    Date reminderDate
    String url
    Category category

    static hasMany = [tags:Tag]
    static belongsTo = [Tag]

    static constraints = {
        title(blank:false)
        description(blank:true, nullable:true, maxSize:1000)
        date(nullable:false)
        reminderDate(nullable:true)
        url(nullable:true, url:true)
        category(nullable:true)
    }
}

```

---

```
String toString(){
    "title"
}
}
```

Como siempre, no olvidemos crear el controlador que se encargará de gestionar las etiquetas:

```
package es.ua.expertojava.todo

class TagController {

    def scaffold = Tag
}
```

Ahora sí, nuestra aplicación empieza a tomar forma de como debe ser la aplicación final, aunque por supuesto todavía quedan muchas cosas que iremos viendo a lo largo del módulo. Ahora mismo, podemos empezar a probar la aplicación insertando datos en cada una de las clases y comprobar que todo funciona correctamente. Sin embargo, la labor de introducción de los datos a mano en la aplicación es algo muy repetitivo y aburrido, así que vamos a ver como podemos convertir esta tarea en algo más sencillo y no tener que repetirla cada vez que probamos la aplicación (recuerda que ahora mismo la base de datos está en memoria con lo que se destruye cada vez que reiniciamos la aplicación).

En el directorio de configuración de nuestra aplicación (*grails-app/conf*) tenemos un archivo llamado *BootStrap.groovy* cuya funcionalidad es posibilitar la realización de acciones al arrancar y al finalizar nuestra aplicación. Como ya estaréis imaginando, vamos a aprovechar este fichero para introducir algunos datos en nuestra aplicación para tener algo de información ya introducida en nuestra aplicación. El siguiente ejemplo, inserta 4 usuarios diferentes y 4 mensajes enviados por estos usuarios.

```
import es.ua.expertojava.todo.*

class BootStrap {

    def init = { servletContext ->
        def categoryHome = new Category(name:"Hogar").save()
        def categoryJob = new Category(name:"Trabajo").save()

        def tagEasy = new Tag(name:"Fácil").save()
        def tagDifficult = new Tag(name:"Difícil").save()
        def tagArt = new Tag(name:"Arte").save()
        def tagRoutine = new Tag(name:"Routine").save()
        def tagKitchen = new Tag(name:"Cocina").save()

        def todoPaintKitchen = new Todo(title:"Pintar cocina", date:new
Date()+1)
        def todoCollectPost = new Todo(title:"Recoger correo postal",
date:new Date()+2)
        def todoBakeCake = new Todo(title:"Cocinar pastel", date:new
Date()+4)
        def todoWriteUnitTests = new Todo(title:"Escribir tests
unitarios", date:new Date())
    }
```

```
        todoPaintKitchen.addToTags(tagDifficult)
        todoPaintKitchen.addToTags(tagArt)
        todoPaintKitchen.addToTags(tagKitchen)
        todoPaintKitchen.category = categoryHome
        todoPaintKitchen.save()

        todoCollectPost.addToTags(tagRoutine)
        todoCollectPost.category = categoryJob
        todoCollectPost.save()

        todoBakeCake.addToTags(tagEasy)
        todoBakeCake.addToTags(tagKitchen)
        todoBakeCake.category = categoryHome
        todoBakeCake.save()

        todoWriteUnitTests.addToTags(tagEasy)
        todoWriteUnitTests.category = categoryJob
        todoWriteUnitTests.save()
    }
    def destroy = { }
}
```

---

## Scaffolding estático

Una vez hemos visto lo rápido que hemos desarrollado la base de nuestra aplicación, vamos a ver como podemos llegar algo más lejos con nuestras aplicaciones gracias al scaffolding estático. Hasta el momento, el código de nuestros controladores y vistas se genera al vuelo y no podemos realizar ni siquiera una pequeña modificación a estos artefactos.

Para pasar de scaffolding dinámico a estático, Grails dispone de tres comandos para realizar esta tarea:

- generate-controller
- generate-views
- generate-all

Los tres comandos esperan como parámetro una clase de dominio para la cual generar el código necesario. Veamos un ejemplo sobre las clases de dominio de la aplicación.

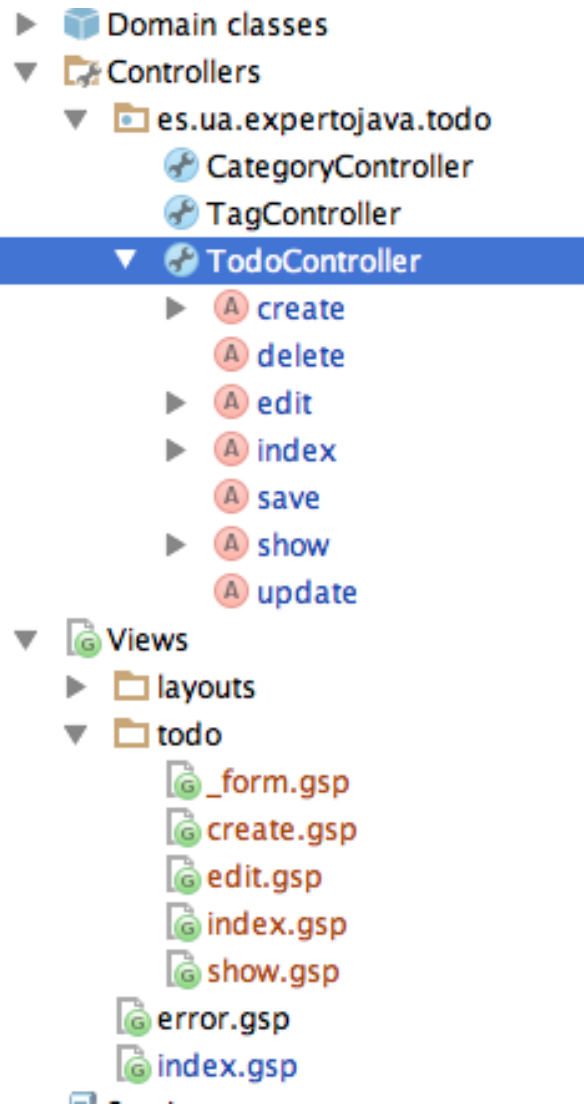
---

```
grails generate-all es.ua.expertojava.todo.TODO
```

---

Tras ejecutar este comando, si echamos un vistazo a nuestro proyecto en IntelliJ IDEA, veremos como el controlador *es.ua.expertojava.todo.TODOController* y el directorio de las vistas *views/todo* han cambiado considerablemente.

Por un lado, el controlador ahora tiene todo el contenido necesario de forma estática de forma que vamos a poder modificarlo según nuestras necesidades. Además, en el directorio de las vistas correspondientes a la clase de dominio *todo* podemos ver los archivos necesarios para poder completar el CRUD de la clase *Todo*.



Los otros dos comandos comentados anteriormente van a realizar en conjunto las mismas acciones que el comando *generate-all*. Veamos un ejemplo con la clase de dominio *Tag*.

```
grails generate-controller es.ua.expertojava.todo.Tag
grails generate-views es.ua.expertojava.todo.Tag
```

Como podrás comprobar en el proyecto, el controlador *es.ua.expertojava.todo.TagController* se ha definido de forma estática y las vistas han sido generadas para poder ser modificadas más fácilmente.

## 3.6. Ejercicios

### Marcar como realizada una tarea (0.25 puntos)

En la definición de la aplicación *todo* hemos pasado por alto algo fundamental y es que una tarea puede haber sido realizada o no. Añade la información necesaria a nuestras clases de dominio para saber si una tarea ha sido ya completada.

No olvides añadir el valor deseado en el archivo *Bootstrap.groovy* para la nueva propiedad añadida.

### Modificación pantalla inicial (0.25 puntos)

Vamos a modificar la pantalla de bienvenida a la aplicación con las siguientes indicaciones:

- Eliminar el logo de Grails de la parte superior para indicar el nombre la aplicación
- Indicar nuestro nombre, apellidos y DNI
- Añadir una imagen de vuestro perfil (puede valer la misma que tenéis en el campus virtual)
- Eliminar el bloque de la izquierda donde aparece toda la información relativa a nuestra aplicación
- Añadir una descripción apropiada de la aplicación
- Añadir los enlaces correspondientes a las acciones que podemos realizar por el momento en nuestra aplicación

Podría quedar algo así:



### Scaffolding estático sobre la clase Categoría (0.75 puntos)

La única clase de dominio que nos queda por definir de forma estática es la referente a las categorías. Utilicemos también scaffolding estático para esta clase. Además, vamos a realizar las siguientes modificaciones sobre las vistas de las categorías:

- Cuando creamos una categoría, Grails nos ofrece un enlace para crear *tareas* que por el momento vamos a eliminar.

## Create Category

Name \*

Description

Todos [Add Todo](#)

- En el listado de categorías, vamos a eliminar el campo descripción porque en ocasiones esa descripción puede ser demasiado extensa.
- Cuando creamos una categoría utilizando la aplicación, el mensaje que aparece indica que la categoría con identificado X ha sido creada correctamente. Realiza los cambios necesarios en el controlador para que en lugar de indicar el identificador, aparezca el nombre de la nueva categoría creada.

 Category 3 created

 Category A fourth category created

## 4. Patrón MVC: Vistas y controladores.

Tras una primera sesión introductoria a Grails en la que ya hemos visto como crear una sencilla aplicación utilizando el scaffolding que nos ofrece Grails. Aunque en esta primera sesión no hemos comentado nada sobre las vistas, ya podemos suponer como Grails las organiza siguiendo el principio de *convención sobre configuración*.

En esta sesión profundizaremos en más aspectos relativos a las vistas, tales como creación de plantillas o etiquetas. También aprovecharemos para hablar más detalladamente sobre los controladores en Grails.

### 4.1. Estructura de las vistas en Grails

Lo primero que hay que comentar es que en Grails los archivos de las vistas de nuestras aplicaciones tienen extensión *GSP*. Además, y siguiendo con el paradigma de *convención sobre configuración*, Grails aconseja que las vistas utilizadas en nuestra aplicación se alojen en el directorio *grails-app/views*. Dentro de este directorio, al crear nuestro proyecto se ha creado también el subdirectorio *layouts*. Este directorio viene por defecto con un archivo *main.gsp*, que podríamos decir que es el encargado de pintar la apariencia de toda la aplicación en Grails.

Como comentábamos en la sesión de introducción a Grails, éste utiliza el framework Sitemesh para renderizar una aplicación en Grails y una de las características de Sitemesh es que utiliza lo que podríamos llamar un archivo padre (layout) y dentro de este archivo padre se pintarán todas las vistas de la aplicación. Veamos ahora el contenido del único archivo (*/grails-app/views/layouts/main.gsp*) que aparece por el momento en el directorio *layouts*.

```

<!DOCTYPE html>
<!--[if lt IE 7 ]> <html lang="en" class="no-js ie6"> <![endif]-->
<!--[if IE 7 ]> <html lang="en" class="no-js ie7"> <![endif]-->
<!--[if IE 8 ]> <html lang="en" class="no-js ie8"> <![endif]-->
<!--[if IE 9 ]> <html lang="en" class="no-js ie9"> <![endif]-->
<!--[if (gt IE 9)|!(IE)]><!-->
<html lang="en" class="no-js"><!--<![endif]-->
  <head>
    <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
    <title><g:layoutTitle default="Grails"/></title>
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <link rel="shortcut icon" href="${assetPath(src:
'favicon.ico')}" type="image/x-icon">
    <link rel="apple-touch-icon" href="${assetPath(src: 'apple-touch-
icon.png')}">
    <link rel="apple-touch-
icon" sizes="114x114" href="${assetPath(src: 'apple-touch-icon-
retina.png')}">
    <asset:stylesheet src="application.css"/>
    <asset:javascript src="application.js"/>
    <g:layoutHead/>
  </head>
  <body>
    <div id="grailsLogo" role="banner"><a href="http://
grails.org"><asset:image src="grails_logo.png" alt="Grails"/></a></div>

```



```

    <g:layoutBody/>
    <div class="footer" role="contentinfo"></div>

    <div id="spinner" class="spinner" style="display:none;"><g:message code="spinner.alt"
  ></div>
    </body>
  </html>

```

Si analizamos un poco el código fuente de este archivo *main.gsp*, comprobaremos la existencia de una serie de etiqueta que veremos posteriormente cuyo espacio de nombres es *g*, como por ejemplo `<g:layoutTitle default="Grails"/>`, `<g:layoutHead/>` o `<g:layoutBody/>`. Estas tres etiquetas están indicando a Grails que deben ser sustituidas por el título, la cabecera y el contenido de las páginas que estemos renderizando.

Por ejemplo, otro archivo que ha creado Grails automáticamente es el archivo que se encuentra en la raíz del directorio *views* llamado *error.gsp*. El contenido de este archivo es el siguiente

```

<!DOCTYPE html>
<html>
  <head>
    <title>
      <g:if env="development">Grails Runtime Exception</
g:if><g:else>Error</g:else>
    </title>
    <meta name="layout" content="main">
    <g:if env="development"><asset:stylesheet src="errors.css"/></
g:if>
  </head>
  <body>
    <g:if env="development">
      <g:renderException exception="{exception}" />
    </g:if>
    <g:else>
      <ul class="errors">
        <li>An error has occurred</li>
      </ul>
    </g:else>
  </body>
</html>

```

¿Cómo actuaría Grails cuando quiera pintar este archivo *error.gsp*? En primer lugar, detecta que este archivo se tiene que pintar envuelto en el layout *main* puesto que tenemos la etiqueta `<meta name="layout" content="main">`. Una vez Grails ya sabe que esa página debe ser pintada utilizando un envoltorio (layout), realizará las sustituciones correspondientes en las etiquetas de dicho envoltorio `<g:layoutTitle default="Grails"/>` (sustituida por el texto Grails Runtime Exception), `<g:layoutHead/>` (sustituido por todo lo que esté dentro de la cabecera de *error.gsp*, que en este caso es únicamente la referencia a los estilos `<g:layoutHead/>`) y `<g:layoutBody/>`, que será sustituido por la salida que produzca la etiqueta `<g:renderException exception="{exception}" />` en caso de que estemos en el entorno de desarrollo o bien por un texto indicando "An error has occurred".

Una vez tenemos claro como se organizan las vistas en un proyecto desarrollado en Grails, ¿cómo podemos organizar estas vistas de forma lógica? En Grails vamos a distinguir dos tipos de vistas: las plantillas y las vistas. Básicamente las primeras serán fragmentos de

código que podremos utilizar en varias partes de nuestra aplicación mientras que las vistas se corresponderán con páginas renderizadas gracias a una llamada de un controlador.

## 4.2. Plantillas

En Grails las plantillas se diferencian de las vistas porque en las primeras su nombre siempre empieza por un subrayado bajo. Otra diferencia notable entre las vistas y las plantillas, es su utilidad y es que las plantillas normalmente las utilizaremos insertadas en vistas y nos facilitará la labor en aquellas partes de la interfaz de usuario que aparezca en varias páginas. Un buen ejemplo de una plantilla podría ser el menú con todas las opciones de nuestra aplicación.

Las buenas prácticas de Grails indican que cualquier plantilla relacionada con una clase de dominio deben estar alojadas todas en un directorio, como por ejemplo *grails-app/views/todo/\_mytemplate.gsp*. De igual forma, aquellas plantillas que vayan a ser utilizadas de una forma más genérica en varias clases de dominio, deben alojarse en un lugar común, por ejemplo en *grails-app/views/common*. Las plantillas tendrán extensión GSP que son las siglas de *Groovy Server Pages*.

### Plantilla de pie de página

Para comprobar el funcionamiento de las plantillas en Grails, vamos a crear una que nos permita mostrar un pie de página en toda nuestra aplicación que sustituirá al que Grails ha creado por nosotros. Como es una plantilla común a toda la aplicación, la almacenaremos en el directorio *grails-app/views/common* y la vamos a llamar *\_footer.gsp* y su contenido será algo parecido a esto:

```
<div class="footer" role="contentinfo">
  &copy; 2015 Experto en Desarrollo de Aplicaciones Web con JavaEE y
  Javascript<br/>
  Aplicación Todo creada por Francisco José García Rico (21.542.334F)
</div>
```

Una vez creada la plantilla, necesitamos poder incrustarla en nuestro proyecto. Para ello, vamos a modificar el layout *main.gsp* para incluir la llamada correspondiente a esta plantilla. Nosotros vamos a añadir esta plantilla justo después de la etiqueta *<g:layoutBody/>*, quedando de esta forma:

```
...
<g:layoutBody/>
<g:render template="/common/footer"/>
<div id="spinner"...
```

Para pintar la plantilla correspondiente hemos utilizado la etiqueta de Grails *g:render* y como habrás podido comprobar, no es necesario poner ni el subrayado bajo ni la extensión *gsp* en el parámetro *template*, puesto que Grails ya lo hace por nosotros.

Nuestra aplicación ya dispone de un pie de página con la información sobre los creadores de la aplicación. Pero, ¿qué pasa si queremos modificar la apariencia de como se muestra este pie de página para por ejemplo centrar el texto? Esto lo conseguiremos editando la información relativa a los estilos de la página (CSS). Las hojas de estilos de las aplicaciones en Grails se encuentran en el directorio *grails-app/assets/stylesheets/*, y concretamente tenemos un archivo llamado *main.css*. Lo que vamos a hacer a continuación es modificar la clase

correspondiente al pie de página (*.footer*). Abrimos la hoja de estilos *main.css* y localizamos la clase *.footer* para dejarla de la siguiente forma.

```
.footer {  
  background: #abbf78;  
  color: #000;  
  clear: both;  
  font-size: 0.8em;  
  margin-top: 1.5em;  
  padding: 1em;  
  min-height: 1em;  
  text-align:center;  
}
```

## Plantilla de encabezado

La primera plantilla que hemos creado no nos ha supuesto mucha complicación y era muy sencilla de implementar, pero al menos nos ha servido como introducción al sistema de plantillas de Grails y a comprender como se renderizan estas plantillas en el sistema. La segunda plantilla que vamos a implementar nos va a complicar algo más la vida.

Aunque nuestra aplicación todavía no tiene el concepto de usuario (es algo que añadiremos en la sesión 7), esta plantilla será la encargada de mostrar un enlace indicando la palabra *login* para que cuando los futuros usuarios hagan clic sobre ella, aparezca un formulario donde el usuario podrá identificarse en el sistema. Ahora bien, ¿qué pasa si el usuario ya se ha identificado? Pues lo que se hace habitualmente es modificar esa primera línea para que en lugar de mostrar un enlace con la palabra *login*, se muestre una línea con el nombre completo del usuario en cuestión y un enlace para que abandone el sistema de forma segura. Eso es lo que vamos a hacer con esta segunda plantilla.

En primer lugar, vamos a modificar el archivo *grails-app/views/layout/main.gsp* para que en la parte superior de la aplicación, aparezca un encabezado que definiremos posteriormente. El archivo *main.gsp* quedaría así:

```
...  
<body>  
<g:render template="/common/header"/>  
<div id="grailsLogo" role="banner">  
  <a href="http://grails.org">  
      
  </a>  
</div>  
<g:layoutBody/>  
<g:render template="/common/footer"/>  
...
```

Si intentas actualizar ahora la aplicación, verás como Grails nos devuelve un error y esto es debido a que todavía no hemos creado la correspondiente plantilla *\_header.gsp*, así que antes de continuar vamos a crearla en el directorio *grails-app/views/common/*, puesto que es una plantilla que se utilizará a lo largo de toda la aplicación independientemente de la clase que estemos utilizando en cada momento. Vamos a ver el contenido de esta nueva plantilla y luego lo comentamos más ampliamente.

```

<div id="menu">
  <nobr>
    <g:if test="{isUserLoggedIn}">
      <b>${userInstance?.name} ${userInstance?.surnames}</b> |
      <g:link controller="user" action="logout">Logout</g:link>
    </g:if>
    <g:else>
      <g:link controller="user" action="login">Login</g:link>
    </g:else>
  </nobr>
</div>

```

Si actualizamos ahora la aplicación en el navegador, veremos como el enlace con la palabra *Login* queda a la izquierda, lo que no es lo habitual en las aplicaciones web, donde suele aparecer casi siempre en la parte superior derecha de la pantalla. Para solucionar esto, volvemos a editar los estilos de nuestra aplicación y le añadimos las siguientes entradas:

```

#header {
  text-align:left;
  margin: 0px;
  padding: 0;
}

#header #menu{
  float: right;
  width: 240px;
  text-align:right;
  font-size:12px;
  padding:4px;
}

```

Ya tenemos alineado a la derecha el texto con la palabra *Login*, que además, aparecerá en cualquier página de la aplicación. Antes de continuar, vamos a explicar el contenido del archivo *grails-app/views/common/\_header.gsp*. Podemos comprobar como se hace una comprobación para saber si el usuario actual se ha identificado o no en nuestro sistema. Para realizar esta comprobación utilizamos la etiqueta *<g:if>*, a la que se le pasa el parámetro *test* con la variable *{isUserLoggedIn}* (variable que todavía no existe). En caso de que esta variable esté ya definida (el usuario ya se ha identificado en el sistema), se mostrará su nombre y apellidos seguido de un enlace para que abandone la aplicación de forma segura. En caso contrario, se le mostrará un enlace con la palabra *Login* para que se identifique. En ambos casos, se utiliza la etiqueta *<g:link>* que permite crear enlaces que apunten a una determinada acción de un controlador, tal y como veremos más adelante.

Si ahora hacemos clic sobre *Login*, veremos como Grails nos devuelve un error del tipo 404, puesto que no hemos definido nada para que la aplicación actúe en consecuencia. Lo dejaremos así hasta que nuestra aplicación soporte usuarios que veremos en la sesión de Spring Security.

### 4.3. Páginas

Las páginas en Grails podemos definirlas como aquellos archivos de la interfaz de usuario que se pintarán a partir de una llamada realizada desde un controlador. Por ejemplo, si

nuestra aplicación necesita una sección con las preguntas más frecuentes de la aplicación, podríamos crear una vista llamada *faqs.gsp* que se encargue de esto. Además, como esta vista no va a pertenecer a ninguna clase de dominio en cuestión sino que será común a toda la aplicación, la ubicaremos en el directorio */grails-app/views/common*. Más adelante crearemos un controlador que se encargará de pintar las preguntas más frecuentes de nuestra aplicación.

Aunque ya te habrás dado cuenta, a diferencia de las plantillas, las vistas no deben utilizar un subrayado bajo al inicio del nombre pero también deben tener extensión GSP.

## Variables y alcance de las mismas en páginas

En GSP podemos utilizar podemos utilizar los símbolos `<% %>` de la siguiente forma

```
<% now = new Date() %>
```

Y de forma similar podemos acceder a esa variables con

```
<%= now %>
```

Dentro del alcance de una página GSP, tendremos a nuestra disposición una serie de variables predefinidas, que son:

- application - Instancia de [javax.servlet.ServletContext](#)<sup>13</sup>
- applicationContext - Instancia de Spring ApplicationContext
- flash - El objeto flash que veremos más adelante en la sección de los controladores
- grailsApplication - Instancia de GrailsApplication
- out - Un response writer para escribir la salida
- params - Los parámetros pasados en la petición
- request - Instancia de [HttpServletRequest](#)<sup>14</sup>
- response - Instancia de [HttpServletResponse](#)<sup>15</sup>
- session - Instancia de [HttpSession](#)<sup>16</sup>
- webRequest - Instancia de [GrailsWebRequest](#)<sup>17</sup>

Utilizando los símbolos `<% y %>` vamos a poder realizar bucles:

```
<html>
  <body>
    <% [1,2,3,4].each { num -> %>
      <p><%= "Hello ${num}!" %></p>
    <%}%>
  </body>
</html>
```

<sup>13</sup> <http://docs.oracle.com/javaee/7/api/javax/servlet/ServletContext.html>

<sup>14</sup> <http://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpServletRequest.html>

<sup>15</sup> <http://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpServletResponse.html>

<sup>16</sup> <http://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpSession.html>

<sup>17</sup> <http://grails.org/doc/latest/api/org/codehaus/groovy/grails/web/servlet/mvc/GrailsWebRequest.html>

así como comprobaciones lógicas:

```
<html>
  <body>
    <% if (params.hello == 'true')%>
      <%= "Hello!" %>
    <% else %>
      <%= "Goodbye!" %>
    </body>
  </html>
```

## Directivas de páginas

Las páginas GSP soportan además algunas directivas típicas de las JSP tales como *importar clases*:

```
<%@ page import="java.awt.*" %>
```

o incluso las directivas para especificar el tipo de contenido de la página visualizada:

```
<%@ page contentType="text/json" %>
```

## Expresiones

La sintaxis vista en el punto anterior que utiliza los símbolos `<%` y `%>` sin embargo no es tan utilizada debido a la existencia de las expresiones GSP, que son similares a las expresiones JSP EL y tienen la forma `${expresion}`:

```
<html>
  <body>
    Hello ${params.name}
  </body>
</html>
```

## 4.4. Etiquetas

En los ejemplos que hemos visto hasta el momento, han aparecido algunas de las etiquetas más utilizadas en Grails. Sin embargo, son muchas más las disponibles y a continuación vamos a ver algunas de ellas. El número de etiquetas de Grails crece continuamente gracias al aporte de la cada vez mayor comunidad de usuarios con sus plugins. A continuación vamos a ver aquellas etiquetas que vienen con el *core* de Grails.

### Etiquetas lógicas

Estas etiquetas nos permitirán realizar comprobaciones del tipo *si-entonces-sino*.

Etiqueta	Descripción	API
<code>&lt;g:if&gt;</code>	Evalúa una expresión y actúa en consecuencia	<a href="http://www.grails.org/doc/latest/ref/Tags/if.html">http://www.grails.org/doc/latest/ref/Tags/if.html</a>

Etiqueta	Descripción	API
<g:else>	La parte <i>else</i> del bloque <i>if</i>	<a href="http://www.grails.org/doc/latest/ref/Tags/else.html">http://www.grails.org/doc/latest/ref/Tags/else.html</a>
<g:elseif>	La parte <i>elseif</i> del bloque <i>if</i>	<a href="http://www.grails.org/doc/latest/ref/Tags/elseif.html">http://www.grails.org/doc/latest/ref/Tags/elseif.html</a>

```
<g:if test="\${userInstance?.type == 'admin'}">
  <!-- mostrar funciones de administrador -->
</g:if>
<g:else>
  <!-- mostrar funciones básicas -->
</g:else>
```

## Etiquetas de iteración

Son utilizadas para iterar a través de colecciones de datos o hasta que una determinada condición se evalúe a falso.

Etiqueta	Descripción	API
<g:while>	Ejecuta un bloque de código mientras una condición se evalúe a cierto	<a href="http://www.grails.org/doc/latest/ref/Tags/while.html">http://www.grails.org/doc/latest/ref/Tags/while.html</a>
<g:each>	Itera sobre una colección de objetos	<a href="http://www.grails.org/doc/latest/ref/Tags/each.html">http://www.grails.org/doc/latest/ref/Tags/each.html</a>
<g:collect>	Itera sobre una colección y transforma los resultados tal y como se defina en el parámetro <i>expr</i>	<a href="http://www.grails.org/doc/latest/ref/Tags/collect.html">http://www.grails.org/doc/latest/ref/Tags/collect.html</a>
<g:findAll>	Itera sobre una colección donde los elementos se corresponden con la expresión GPath definida en el parámetro <i>expr</i>	<a href="http://www.grails.org/doc/latest/ref/Tags/findAll.html">http://www.grails.org/doc/latest/ref/Tags/findAll.html</a>

```
<g:set var="num" value="\${1}" />
<g:while test="\${num < 5 }">
  <p>Number ${num++}
</g:while>
```

/\*  
\*/

```
<g:each in="\${[1,2,3]}" var="num">
  <p>Number ${num}</p>
</g:each>
```

/\*  
\*/

```
Stephen King's Books:
<g:findAll in="\${books}" expr="it.author == 'Stephen King'">
  <p>Title: ${it.title}
</g:findAll>
```

## Etiquetas de asignación

Nos servirán para asignar valores a variables.

Etiqueta	Descripción	API
<code>&lt;g:set&gt;</code>	Define y establece el valor de una variable utilizada en una página GSP	<a href="http://www.grails.org/doc/latest/ref/Tags/set.html">http://www.grails.org/doc/latest/ref/Tags/set.html</a>

```

<g:set var="now" value="${new Date()}" />

/*****/
<g:set var="myHTML">
  Some re-usable code on: ${new Date()}
</g:set>

/*****/
<g:set var="now" value="${new Date()}" scope="request" />

```

## Etiquetas de enlaces

Crean enlaces a partir de los parámetros pasados.

Etiqueta	Descripción	API
<code>&lt;g:link&gt;</code>	Crea un enlace HTML utilizando los parámetros pasados	<a href="http://www.grails.org/doc/latest/ref/Tags/link.html">http://www.grails.org/doc/latest/ref/Tags/link.html</a>
<code>&lt;g:createLink&gt;</code>	Crea un enlace HTML que puede ser utilizado dentro de otras etiquetas	<a href="http://www.grails.org/doc/latest/ref/Tags/createLink.html">http://www.grails.org/doc/latest/ref/Tags/createLink.html</a>

```

<g:link action="show" id="1">Book 1</g:link>
<g:link action="show" id="${currentBook.id}">${currentBook.name}</g:link>
<g:link controller="book">Book Home</g:link>
<g:link controller="book" action="list">Book List</g:link>
<g:link url="[action: 'list', controller: 'book']">Book List</g:link>
<g:link params="[sort: 'title', order: 'asc', author:
  currentBook.author]" action="list">
  Book List
</g:link>

```

## Etiquetas de formularios

Las utilizaremos para crear nuestros propios formularios HTML.

Etiqueta	Descripción	API
<code>&lt;g:actionSubmit&gt;</code>	Crea un botón de tipo <i>submit</i>	<a href="http://www.grails.org/doc/latest/ref/Tags/actionSubmit.html">http://www.grails.org/doc/latest/ref/Tags/actionSubmit.html</a>



Etiqueta	Descripción	API
<g:actionSubmitImage>	Crea un botón de tipo <i>submit</i> con una imagen	<a href="http://www.grails.org/doc/latest/ref/Tags/actionSubmitImage.html">http://www.grails.org/doc/latest/ref/Tags/actionSubmitImage.html</a>
<g:checkBox>	Crea un elemento de formulario de tipo checkbox	<a href="http://www.grails.org/doc/latest/ref/Tags/checkbox.html">http://www.grails.org/doc/latest/ref/Tags/checkbox.html</a>
<g:currencySelect>	Crea un campo de tipo <i>select</i> con un listado de monedas	<a href="http://www.grails.org/doc/latest/ref/Tags/currencySelect.html">http://www.grails.org/doc/latest/ref/Tags/currencySelect.html</a>
<g:datePicker>	Crea un elemento de formulario para seleccionar una fecha con día, mes, año, hora, minutos y segundos	<a href="http://www.grails.org/doc/latest/ref/Tags/datePicker.html">http://www.grails.org/doc/latest/ref/Tags/datePicker.html</a>
<g:form>	Crea un formulario	<a href="http://www.grails.org/doc/latest/ref/Tags/form.html">http://www.grails.org/doc/latest/ref/Tags/form.html</a>
<g:hiddenField>	Crea un campo oculto	<a href="http://www.grails.org/doc/latest/ref/Tags/hiddenField.html">http://www.grails.org/doc/latest/ref/Tags/hiddenField.html</a>
<g:localeSelect>	Crea un elemento de formulario de tipo <i>select</i> con un listado de posibles localizaciones	<a href="http://www.grails.org/doc/latest/ref/Tags/localeSelect.html">http://www.grails.org/doc/latest/ref/Tags/localeSelect.html</a>
<g:radio>	Crea un elemento de formulario de tipo radio	<a href="http://www.grails.org/doc/latest/ref/Tags/radio.html">http://www.grails.org/doc/latest/ref/Tags/radio.html</a>
<g:radioGroup>	Crea un grupo de elementos de formulario de tipo radio	<a href="http://www.grails.org/doc/latest/ref/Tags/radioGroup.html">http://www.grails.org/doc/latest/ref/Tags/radioGroup.html</a>
<g:select>	Crea un elemento de formulario de tipo <i>select combo box</i>	<a href="http://www.grails.org/doc/latest/ref/Tags/select.html">http://www.grails.org/doc/latest/ref/Tags/select.html</a>
<g:textField>	Crea un elemento de formulario de tipo <i>text</i>	<a href="http://www.grails.org/doc/latest/ref/Tags/textField.html">http://www.grails.org/doc/latest/ref/Tags/textField.html</a>
<g:passwordField>	Crea un elemento de formulario de tipo <i>password</i>	<a href="http://www.grails.org/doc/latest/ref/Tags/passwordField.html">http://www.grails.org/doc/latest/ref/Tags/passwordField.html</a>
<g:textArea>	Crea un elemento de formulario de tipo <i>textarea</i>	<a href="http://www.grails.org/doc/latest/ref/Tags/textArea.html">http://www.grails.org/doc/latest/ref/Tags/textArea.html</a>
<g:timeZoneSelect>	Crea un elemento de formulario de tipo <i>select</i> con un listado de zonas horarias	<a href="http://www.grails.org/doc/latest/ref/Tags/timeZoneSelect.html">http://www.grails.org/doc/latest/ref/Tags/timeZoneSelect.html</a>

```
<g:form name="myForm" url="[controller:'book',action:'list']">...</g:form>
```

```
/*  
*****  
*/
```

```
<g:textField name="myField" value="${myValue}" />
```

```
/*  
*****  
*/
```

```
<g:actionSubmit value="Some update label" action="update" />
```

## Etiquetas de renderizado

Permite renderizar las páginas web de nuestras aplicaciones utilizando las plantillas.

Etiqueta	Descripción	API
<g:applyLayout>	Aplica un determinado diseño a una página o una plantilla	<a href="http://www.grails.org/doc/latest/ref/Tags/applyLayout.html">http://www.grails.org/doc/latest/ref/Tags/applyLayout.html</a>
<g:formatDate>	Aplica el formato <i>SimpleDateFormat</i> a una fecha	<a href="http://www.grails.org/doc/latest/ref/Tags/formatDate.html">http://www.grails.org/doc/latest/ref/Tags/formatDate.html</a>
<g:formatNumber>	Aplica el formato <i>DecimalFormat</i> a un número	<a href="http://www.grails.org/doc/latest/ref/Tags/formatNumber.html">http://www.grails.org/doc/latest/ref/Tags/formatNumber.html</a>
<g:layoutHead>	Muestra una determinada cabecera para una página	<a href="http://www.grails.org/doc/latest/ref/Tags/layoutHead.html">http://www.grails.org/doc/latest/ref/Tags/layoutHead.html</a>
<g:layoutBody>	Muestra un determinado contenido para una página	<a href="http://www.grails.org/doc/latest/ref/Tags/layoutBody.html">http://www.grails.org/doc/latest/ref/Tags/layoutBody.html</a>
<g:layoutTitle>	Muestra un determinado título para una página	<a href="http://www.grails.org/doc/latest/ref/Tags/layoutTitle.html">http://www.grails.org/doc/latest/ref/Tags/layoutTitle.html</a>
<g:meta>	Muestra las propiedades del archivo <i>application.properties</i>	<a href="http://www.grails.org/doc/latest/ref/Tags/meta.html">http://www.grails.org/doc/latest/ref/Tags/meta.html</a>
<g:render>	Muestra un modelo utilizando una plantilla	<a href="http://www.grails.org/doc/latest/ref/Tags/render.html">http://www.grails.org/doc/latest/ref/Tags/render.html</a>
<g:renderErrors>	Muestra los errores producidos en una página	<a href="http://www.grails.org/doc/latest/ref/Tags/renderErrors.html">http://www.grails.org/doc/latest/ref/Tags/renderErrors.html</a>
<g:pageProperty>	Muestra una propiedad de una página	<a href="http://www.grails.org/doc/latest/ref/Tags/pageProperty.html">http://www.grails.org/doc/latest/ref/Tags/pageProperty.html</a>
<g:paginate>	Muestra los típicos botones <i>Anterior</i> y <i>Siguiente</i> y las <i>migas de pan</i> cuando se devuelven muchos resultados	<a href="http://www.grails.org/doc/latest/ref/Tags/paginate.html">http://www.grails.org/doc/latest/ref/Tags/paginate.html</a>
<g:sortableColumn>	Muestra una columna de una tabla con la posibilidad de ordenarla	<a href="http://www.grails.org/doc/latest/ref/Tags/sortableColumn.html">http://www.grails.org/doc/latest/ref/Tags/sortableColumn.html</a>

```
<g:render template="bookTemplate" model="[book: myBook]" />
```

```
/*  
*****  
*/
```

```
<g:render template="bookTemplate" var="book" collection="${bookList}" />

/*****/
" />
```

## Etiquetas de validación

Se utilizan para mostrar errores y mensajes de advertencia.

Etiqueta	Descripción	API
<g:eachError>	Itera a través de los errores producidos en un bean o un modelo	<a href="http://www.grails.org/doc/latest/ref/Tags/eachError.html">http://www.grails.org/doc/latest/ref/Tags/eachError.html</a>
<g:hasErrors>	Comprueba si se ha producido algún error en un bean o un modelo	<a href="http://www.grails.org/doc/latest/ref/Tags/hasErrors.html">http://www.grails.org/doc/latest/ref/Tags/hasErrors.html</a>
<g:message>	Muestra un mensaje a partir de la propiedad pasada por parámetro	<a href="http://www.grails.org/doc/latest/ref/Tags/message.html">http://www.grails.org/doc/latest/ref/Tags/message.html</a>

```
<g:eachError bean="${book}">
  <li>${it}</li>
</g:eachError>
```

```
/*****/
<g:message code="my.message.code" />
```

## 4.5. Librería de etiquetas

Como acabamos de ver, Grails nos ofrece la posibilidad de utilizar un amplio rango de etiquetas tanto JSP como GSP, pero en ocasiones, es probable que necesitemos crear nuestras propias etiquetas. Estas etiquetas nos van a permitir realizar tareas repetitivas de forma rápida y sencilla.

Las librerías de etiquetas no requieren ninguna tarea de configuración y, como casi siempre, se recargan automáticamente sin reiniciar el servidor.

En Grails tenemos dos métodos para crear etiquetas. Por un lado mediante el comando *grails create-tag-lib* y por otro creando una nueva clase en el directorio *grails-app/taglib* cuyo nombre termine en *TagLib*. Nosotros utilizaremos como hasta ahora el comando *grails create-tag-lib es.ua.expertojava.todo.TODO*.

Este será el resultado de la nueva librería de etiquetas creada:

```
package es.ua.expertojava.todo

class TodoTagLib {
    static defaultEncodeAs = [taglib:'html']
    //static encodeAsForTags = [tagName: [taglib:'html'], otherTagName:
    [taglib:'none']]
}
```

```
}

```

Las dos propiedades estáticas creadas nos servirán para indicar de forma global o por etiqueta como queremos renderizarlas. Por ejemplo, con `static defaultEncodeAs = [taglib:'html']` estamos indicando que cuando la etiqueta imprima el símbolo `<` o `>` lo hará por su correspondiente entidad, ésto es, `&lt;` o `&gt;`.

Además de definirlo de forma global, también vamos a poder especificarlo por etiqueta utilizando la variable `encodeAsForTags` y por supuesto modificando las claves utilizadas en el ejemplo por los nombres de nuestras etiquetas.

En nuestros ejemplos, vamos a tener que modificar la variable `defaultEncodeAs` por el siguiente valor: `[taglib:'html']`.

## Etiquetas simples

El primer ejemplo de etiqueta que vamos a crear será una etiqueta que permita la inclusión de archivos de funciones javascript en el código de nuestras páginas GSPs. Para ello definimos un método en la nueva clase `TodoTagLib` con el siguiente contenido:

```
def includeJs = {attrs ->
    out << "<script src='scripts/${attrs['script']}.js' ></script>"
}

```

La creación de la etiqueta necesita como parámetro los atributos de la misma. En este primer ejemplo, sólo vamos a utilizar un atributo que será el nombre del archivo javascript que queremos invocar en nuestra página GSP.

Una vez creada la etiqueta, para realizar la invocación de la misma en las páginas GSP utilizaremos el siguiente código.

```
<g:includeJs script="miscrypt"/>

```

Para invocar la nueva librería creada se utiliza el `namespace` genérico `<g>`. Sin embargo, Grails nos permite crear nuestro propio espacio de nombres para que el código generado sea sencillo de leer. El espacio de nombres que vamos a utilizar será `me`, acrónimo de `mis etiquetas`. Para ello debemos definir una variable estática al inicio de la clase `TodoTagLib` llamada `namespace` e indicándole el valor del nuevo espacio de nombres, tal y como aparece en el siguiente fragmento de código.

```
package es.ua.expertojava.todo

class TodoTagLib {

    static defaultEncodeAs = [taglib:'html']
    //static encodeAsForTags = [tagName: [taglib:'html'], otherTagName:
    [taglib:'none']]

    static namespace = 'me'
}

```

```
def includeJs = {attrs ->
  out << "<script src='scripts/${attrs['script']}.js'/>"
}
}
```

De esta forma, en nuestras páginas GSP ya no tendríamos que utilizar `<g:includeJs script="miscrypt"/>` sino que podríamos emplear `<me:includeJs script="miscrypt"/>`, con lo que la persona que lea el código podrá detectar que esa etiqueta es una etiqueta propia de la aplicación.

En ocasiones, es posible que sea necesario referenciar a nuestras etiquetas desde controladores o desde otras etiquetas y no nos va a ser posible referenciar a estas etiqueta de la forma habitual. En este caso deberíamos llamar a la etiqueta de la siguiente forma:

```
def renderImage = { attrs ->
  println "Rendering the image ${attrs.image}"
  asset.image(src:attrs.image)
}
```

En este ejemplo, en primer lugar imprimimos un log para saber que estamos renderizando una image y después utilizamos una librería de etiquetas disponible en el plugin *asset-pipeline* que nos permite renderizar imágenes de forma optimizada en todos los entornos.

## Etiquetas lógicas

Con Grails también es posible crear etiquetas lógicas que evalúen una cierta condición y actúen en consecuencia en función de dicha evaluación. El siguiente ejemplo es una etiqueta que comprueba si el usuario autenticado es un administrador. El siguiente método realiza esta comprobación y en caso de ser cierto, se mostraría el contenido encerrado entre la etiqueta.

```
def esAdmin = { attrs, body ->
  def usuario = attrs['usuario']
  if(usuario != null && usuario.tipo=="administrador") {
    out << body()
  }
}
```

En esta ocasión, la nueva etiqueta no sólo recibe el atributo *attrs* sino que también necesita del atributo *body*, que se refiere a aquello que esté encerrado entre la apertura y el cierre de la etiqueta. El método comprueba si el usuario pasado por parámetro es un administrador. En la página GSP correspondiente deberíamos indicar el siguiente código.

```
<me:esAdmin usuario="${session.usuario}">
  Dar de baja a usuario
</me:esAdmin>
```

## Generador de código HTML

Grails ofrece la posibilidad de generar código HTML de forma muy sencilla gracias a un Builder conocido como *MarkupBuilder*. Para comprobar el funcionamiento de este Builder, vamos a

crear una nueva etiqueta que imprima un enlace cuyo atributo *href* coincida con el título. Por ejemplo, para imprimir un enlace a la página de Google, normalmente debemos incluir el código `<a href="http://www.google.com">http://www.google.com</a>`. Con esta etiqueta que vamos a crear nos ahorraremos escribir la dirección url dos veces.

En primer lugar creamos un nuevo método en el archivo *TodoTagLib.groovy* y lo llamaremos *printLink()*. Este método utilizará el *MarkupBuilder* e imprimirá un enlace con el valor del atributo *href* igual al contenido.

```
def printLink = { attrs, body ->
    def mkp = new groovy.xml.MarkupBuilder(out)
    mkp.a(href:body(),body())
}
```

Ahora en la vista podemos añadir una etiqueta como `<me:printLink>http://www.google.com</me:printLink>`, la cual será traducida al código HTML `<a href="http://www.google.com">http://www.google.com</a>`.

## 4.6. Controladores

Una vez visto a fondo todo lo relativo a las vistas en las aplicaciones en Grails, vamos a analizar los controladores. En la sesión anterior generábamos el scaffolding de forma estática de tal forma que ahora vamos a analizar con más detalle todos los aspectos relativos a los controladores.

### Acciones en los controladores

Como veíamos anteriormente, los controladores en Grails se ubican en el directorio *grails-app/controllers*. Si recordamos la imagen donde se explicaba el proceso típico de una aplicación que utiliza el patrón Modelo Vista Controlador, los controladores se encargan básicamente de dirigir las llamadas a nuestra aplicación y de ponerse en contacto tanto con las clases de dominio, las vistas y los servicios para ponerlos de acuerdo.

Para analizar el formato de los controladores en Grails, veamos uno cualquiera de los generados en la sesión anterior, como por ejemplo *TodoController.groovy*.

```
package es.ua.expertojava.todo

import static org.springframework.http.HttpStatus.*
import grails.transaction.Transactional

@Transactional(readOnly = true)
class TodoController {

    static allowedMethods = [save: "POST", update: "PUT",
delete: "DELETE"]

    def index(Integer max) {
        params.max = Math.min(max ?: 10, 100)
        respond Todo.list(params), model:[todoInstanceCount: Todo.count()]
    }

    ....
}
```

```
}
```

En primer lugar vemos que se especifica mediante la anotación `@Transactional` la transaccionalidad de todos los métodos del controlador. En la siguiente sesión aprovecharemos para hablar sobre los diferentes niveles de transaccionalidad cuando veamos los servicios en Grails.

La variable estática `allowedMethods` nos permite especificar para determinados métodos de nuestros controladores, el tipo de petición aceptada. En el caso del controlador generado vemos como el método `save()` sólo puede ser procesado por una petición de tipo `POST`, las actualizaciones por peticiones de tipo `PUT` y por último los borrados sólo serán válidos cuando se hagan mediante una petición de tipo `DELETE`.

Los parámetros pasados a los métodos serán mapeados directamente por Grails a partir de los parámetros pasados en la petición. Por ejemplo, si efectuamos la petición <http://localhost:8080/todo/todo/index?max=20>, el parámetro `max` de la petición con valor 20 será pasado como parámetro al método `index`.

Como has podido comprobar en el ejemplo anterior, no es necesario especificar lo que van a devolver los métodos de los controladores con lo que podemos dejar que sea Grails quien se encargue de definirlo especificando la palabra reservada `def`.

Comentar también que en la últimas versiones de Grails se ha añadido el método `respond` que provocará una salida diferente en función de la petición HTTP efectuada. Por ejemplo, si abrimos un terminal y ejecutamos:

```
curl -v -H "Accept: application/json" -H "Content-type: application/json"
-X GET http://localhost:8080/todo/todo/index
```

veremos que la salida del método será en formato json, mientras que si ejecutamos

```
curl -v -H "Accept: text/xml" -H "Content-type: text/xml" -X GET http://
localhost:8080/todo/todo/index
```

veremos que la salida del método será en formato xml, mientras que si desde un navegador vamos a la dirección <http://localhost:8080/todo/todo/index>, Grails renderizará una página html siguiendo una serie de convenios que veremos a continuación.

Además, cada controlador en Grails tiene un método por defecto que sigue las siguientes reglas:

- Si sólo hay un método, por supuesto ese será el método por defecto
- Si hay un método con el nombre `index()`, ese será el método por defecto
- Si la clase define la propiedad `defaultAction`, el método especificado será el escogido por defecto

```
static defaultAction = "list"
```



## Ámbitos de los controladores

Los ámbitos (*scopes*) son una serie de objetos que nos permitirán almacenar variables en una aplicación en Grails. Tenemos los siguientes ámbitos:

- *servletContext*: también conocido con el ámbito *aplicación* y que nos servirá para compartir variables a través de cualquier artefacto de la aplicación. Hablaremos de él cuando veamos como configurar una aplicación en Grails.
- *session*: es utilizada para controlar el estado de un determinado usuario de nuestra aplicación. Habitualmente, se utilizan cookies para asociar la *session* con el usuario.
- *request*: contendrá toda la información de la petición.
- *params*: es un mapa con la información pasada a la petición en forma de parámetro *GET* o *POST*.
- *flash*: es una variable *efímera* de tal forma que sólo dura una petición y la siguiente, con lo que no es necesario destruirla a mano. Un ejemplo de su uso es el paso de mensajes de error a las vistas.

## Relación entre vistas y controladores

Las vistas serán las encargadas de mostrar al usuario aquello que el controlador quiera mostrar. Si echamos un vistazo a los métodos del controlador *TodoController* nos costará entender esa relación pues apenas se hace mención a ninguna vista en este controlador. Esto es debido a que cuando no se especifica de forma explícita que vista renderizar, Grails utiliza lo que comentábamos en la sesión anterior al respecto de "convención sobre configuración" y directamente trata de renderizar una vista que coincida con el patrón *nombre-del-controlador/nombre-del-método.gsp*, con lo que por ejemplo, en el método

```
def show(Todo todoInstance) {  
    respond todoInstance  
}
```

al no haber indicado ninguna vista en el método *respond()*, Grails renderizará la vista ubicada en el directorio *grails-app/todo/show.gsp*. Como comentábamos anteriormente, el método *respond()* se encarga de decidir como renderizar la salida pero si tiene que renderizar una vista en HTML, bien buscará una siguiendo el patrón mencionado o bien le podemos pasar una de forma de implícita, como por ejemplo vemos en el método *save()* del controlador *TodoController*

```
...  
if (todoInstance.hasErrors()) {  
    respond todoInstance.errors, view: 'create'  
    return  
}  
...
```

Sigamos con la relación entre vistas y controladores. Si vemos las vistas y los métodos del controlador *TodoController* generados automáticamente por el scaffolding estático nos daremos cuenta rápidamente que cuatro de esos métodos tienen vistas asociadas (*index*, *create*, *edit* y *show*) mientras que otros tres no tienen ninguna vista asociada (*save*, *update* y *delete*). Esto es debido a que estos tres últimos métodos serán los encargados únicamente de



crear, actualizar o eliminar una tarea y serán los otros métodos los encargados de renderizar las vistas correspondientes.

Si abrimos la vista *todo/create.gsp* veremos que el formulario se debe procesar con la acción *save()*

```
<g:form url="[resource:todoInstance, action:'save']" >
  <fieldset class="form">
    <g:render template="form"/>
  </fieldset>
  <fieldset class="buttons">
    <g:submitButton name="create" class="save" value="${message(code:
'default.button.create.label', default: 'Create')}}" />
  </fieldset>
</g:form>
```

con esto conseguimos que al enviar el formulario, el encargado de procesarlo sea el método *save()*.

Veamos ahora todo el proceso de creación de una tarea. Si arrancamos nuestra aplicación y accedemos en el navegador a la dirección <http://localhost:8080/todo/todo/create>, el método que sale al rescate es *TodoController.create()*. Este método con una sola línea de código es capaz de renderizar una vista con una tarea asociada, que en principio tendrá todos sus campos vacíos.

```
def create() {
    respond new Todo(params)
}
```

Este método renderizará la vista *todo/create.gsp* y le pasará una variable llamada *todoInstance* que contendrá una tarea con sus campos vacíos.

Si abrimos ahora la vista *todo/create.gsp* vemos como esa vista únicamente llama a la plantilla *todo/\_form.gsp* que es la que contiene los elementos de formulario necesarios para crear una tarea.

```
<g:form url="[resource:todoInstance, action:'save']" >
  <fieldset class="form">
    <g:render template="form"/>
  </fieldset>
  <fieldset class="buttons">
    <g:submitButton name="create" class="save" value="${message(code:
'default.button.create.label', default: 'Create')}}" />
  </fieldset>
</g:form>
```

Una vez el usuario envía la información, el método encargado de procesar esa petición será *TodoController.save()*

```
@Transactional
def save(Todo todoInstance) {
```

```

if (todoInstance == null) {
    notFound()
    return
}

if (todoInstance.hasErrors()) {
    respond todoInstance.errors, view: 'create'
    return
}

todoInstance.save flush:true

request.withFormat {
    form multipartForm {
        flash.message = message(code: 'default.created.message', args:
[message(code: 'todo.label', default: 'Todo'), todoInstance.id])
        redirect todoInstance
    }
    '*' { respond todoInstance, [status: CREATED] }
}
}

```

Pongámonos en el caso en que no hayamos completado correctamente todos los campos requeridos. Si ésto sucede, el método volverá a renderizar la vista *todo/create.gsp* con los errores que se hayan producido.

En caso de que todo vaya bien, se almacena la nueva tarea

```

todoInstance.save flush:true

```

y se trata de renderizar el resultado en función del tipo de petición realizada. En caso de que hayamos enviado un formulario se redirige la aplicación para mostrar esa tarea recién creada con una variable de tipo flash para mostrar un mensaje indicando que la tarea se ha creado correctamente.

```

flash.message = message(code: 'default.created.message', args:
[message(code: 'todo.label', default: 'Todo'), todoInstance.id])
redirect todoInstance

```

Cuando hacemos esta redirección Grails buscará el método *show()* y lo renderizará. Este método pintará en el navegador la vista *todo/show.gsp*. Esta vista, como prácticamente cualquier vista, tiene un fragmento de código que comprueba la presencia de una variable de tipo *flash* para actuar en consecuencia. En este caso simplemente mostrará el mensaje que hemos completado en el método *save()* del *TodoController*.

```

<g:if test="${flash.message}">
    <div class="message" role="status">${flash.message}</div>
</g:if>

```

Si en lugar de haber enviado un formulario a través de una página HTML, la petición es de cualquier otro tipo, se trata de renderizar una salida acorde al tipo de petición.

```
'*' { respond todoInstance, [status: CREATED] }
```

Si por ejemplo lanzamos desde la terminal el siguiente comando

```
curl -v -H "Accept: application/json" -H "Content-type: application/json" -X POST -d '{"title':'Hacer los ejercicios de Groovy', 'date_day':25, 'date_month':3, 'date_year':2015, 'date':'date.struct'}" http://localhost:8080/todo/todo/save
```

la aplicación nos creará una nueva tarea y nos devolverá el resultado en formato json. Si por el contrario ejecutamos

```
curl -v -H "Accept: text/xml" -H "Content-type: application/json" -X POST -d '{"title':'Hacer los ejercicios de Grails', 'date_day':26, 'date_month':3, 'date_year':2015, 'date':'date.struct'}" http://localhost:8080/todo/todo/save
```

Con esto hemos podido comprobar como Grails con muy pocas líneas de código es capaz de generar una aplicación que será capaz de responder al cliente de varias formas con lo que podemos tener una aplicación multiplataforma todo integrado en el mismo código fuente.

## Renderizando vistas

En ocasiones, sobre todo cuando realicemos aplicaciones web, tendremos que simplemente renderizar vistas o fragmentos de texto. Para ello Grails dispone del método *render()* que puede renderizar desde un trozo de texto hasta una plantilla.

```
render "Hello World!"

render {
  10.times {
    div(id: it, "Div ${it}")
  }
}

render(view: 'show')

render(template: 'book_template', collection: Book.list())

render(text: "<xml>some xml</xml>", contentType: "text/xml",
  encoding: "UTF-8")
```

## Redirecciones y encadenamientos

Tal y como veíamos anteriormente, en ocasiones necesitaremos redireccionar nuestra aplicación a otros métodos. Para ello, disponemos del método *redirect()* que acepta varios parámetros, como en los siguientes ejemplos:

```
//Redirección que se hace dentro del mismo controlador
```

```
redirect(action: login)

//Redirección que se hace a otro controlador
redirect(controller: 'home', action: 'index')

//Redirección a una uri explícitamente
redirect(uri: "/login.html")

//Redirección a una url absoluta
redirect(url: "http://grails.org")

//Redirección pasando parámetros
redirect(action: 'myaction', params: [myparam: "myvalue"])
```

Además, Grails también ofrece la posibilidad de encadenar acciones. El beneficio de encadenar acciones en lugar de redirigir es que el modelo se pasa de una acción a la siguiente:

```
class ExampleChainController {
    def first() {
        chain(action: second, model: [one: 1])
    }

    def second () {
        chain(action: third, model: [two: 2])
    }

    def third() {
        [three: 3]
    }
}
```

Esta cadena de acciones terminaría con el método *third()* recibiendo los modelos

```
[one: 1, two: 2, three: 3]
```

Si necesitamos acceder a los modelos dentro de una cadena, podemos utilizar la variable dinámica *chainModel* que es un mapa con todos los modelos generados en la cadena.

```
class ChainController {
    def nextInChain() {
        def model = chainModel.myModel
        ...
    }
}
```

## 4.7. Filtros

Los filtros en una aplicación Grails permiten ejecutar acciones antes y después de que los métodos de los controladores sean invocados. El típico ejemplo de uso de los filtros es el de asegurar las peticiones a nuestra aplicación para comprobar si el usuario tiene acceso a un determinado recurso.

Otro ejemplo útil de los filtros puede ser el de mantener un log cada vez que se ejecuta una petición en nuestra aplicación, es decir, que cada vez que llamemos a un método de un controlador, imprimimos algo por pantalla.

Por supuesto que podríamos hacerlo directamente en cada método de los controladores, pero sin duda es algo poco mantenible y no es una buena práctica, con lo que vamos a utilizar los filtros.

En Grails, los filtros se generan creando una nueva clase que termine con la palabra *Filters* o bien mediante el comando `grails create-filters` para que sea Grails quien lo haga por nosotros y nos cree además un esqueleto de la clase en cuestión. Estas nuevas clases se deben crear en el directorio `grails-app/conf`. Cada filtro implementa la lógica a ejecutar antes y después de las acciones así como sobre que acciones se debe hacer. En nuestro caso, debemos crear un filtro ejecutando el comando `grails create-filters log` que se ubicará en el directorio `grails-app/conf/todo` y se llamará `LogFilters.groovy`. El esqueleto que Grails ha creado para este filtro es el siguiente:

```
package todo

class LogFilters {

    def filters = {
        all(controller:'*', action:'*') {
            before = {

            }
            after = { Map model ->

            }
            afterView = { Exception e ->

            }
        }
    }
}
```

Tal y como vemos en el código del esqueleto generado por Grails, podemos ejecutar el filtro antes (*before*) o después (*after*) de que se ejecute el método correspondiente del controlador o bien hacerlo después de que se genere la vista. Además, también podemos especificar sobre que controladores queremos aplicar estos filtros.

Nuestro filtro se aplicará a todos los controladores de nuestra aplicación pero sólo a aquellas acciones que traten de renderizar una vista, que serán *index*, *create*, *edit* y *show* con lo que podríamos tener algo así:

```
package todo

class LogFilters {

    def filters = {
        all(controller:'todo|category|tag', action:'create|edit|index|
show') {
            before = {
```

```
    }
    after = { Map model ->
        println "Controlador ${controllerName} - Accion
${actionName} - Modelo ${model}"
    }
    afterView = { Exception e ->
    }
}
}
}
```

---

La clase LogFilters únicamente dispone de un filtro, el denominado *all()*, pero una misma clase puede tener tantos filtros como queramos.

## 4.8. Ejercicios

### Modificando las vistas (0.25 puntos)

Si hacemos un rápido test de usabilidad en nuestra aplicación veremos que cada vez que necesitamos editar o eliminar una tarea, tenemos que hacer dos clicks sobre la misma, uno para ver la tarea y otro para editarla. Vamos a solucionar este problema de usabilidad modificando el listado de las tareas para añadir una nueva columna a la izquierda de cada tarea para mostrar dos opciones, una para editar y otra para eliminar una tarea.

Aprovecharemos también para eliminar las columnas *Description* y *Reminder Date* del listado de las tareas.

### Etiqueta para valores booleanos (0.50 puntos)

Vamos a crear una etiqueta que nos permita, a partir de un valor booleano, imprimir un icono u otro. La etiqueta se llamara *printIconFromBoolean* y únicamente recibirá como parámetro el atributo *value*. Modifica también el namespace de la librería creada para que podamos a ella a través del namespace *todo* con lo que para llamar a nuestra etiqueta tendríamos que ejecutar algo como:

```
<todo:printIconFromBoolean value=${todoInstance.done}/>
```

Y deberíamos imprimir algo parecido a esto:

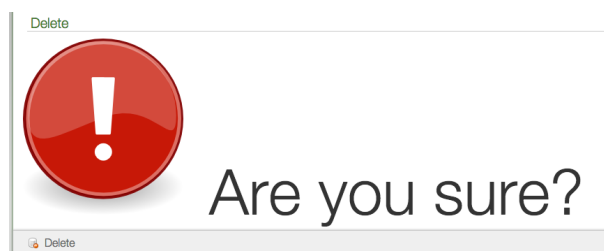
Title	Date	Url	Done
Pintar cocina	2014-10-21 00:00:00 BST		✓
Recoger correo postal	2014-10-22 19:37:47 BST		✗
Cocinar pastel	2014-10-24 19:37:47 BST		✗
Escribir tests unitarios	2014-10-20 19:37:47 BST		✗

Sería aconsejable que utilizaras el método *image()* de la librería de etiquetas del plugin *asset pipeline*. Aquí tienes un ejemplo:

```
asset.image(src:"icon.png")
```

### Página de confirmación de eliminación de etiquetas (0.50 puntos)

Cuando deseamos eliminar una etiqueta, al hacer click sobre el botón eliminar se nos muestra un mensaje en Javascript para confirmar si realmente queremos eliminar esta etiqueta. Vamos a modificar este sistema para que en lugar de mostrar un mensaje en Javascript, nos muestre una nueva página para pedir confirmación. Esto nos permitirá añadir algo más de seguridad en lo que el usuario está realizando ya que podemos mostrar un mensaje de alerta más evidente, como por ejemplo éste:







## 5. Patrón MVC: Dominios y servicios.

En esta tercera sesión sobre Grails veremos como Grails hace uso de GORM para mapear objetos contra la base de datos de la aplicación. Recordar que GORM es una capa superior a Hibernate con lo que gran parte de lo aprendido en módulos anteriores sobre Hibernate serán totalmente válidos para trabajar con Grails.

Por último, veremos lo sencillo que es en Grails implementar servicios que se encarguen de la lógica de negocio de nuestra aplicación creando un sencillo ejemplo.

### 5.1. Dominios

#### Creación de dominios

Como puedes imaginar, lo primero que se debe hacer con una clase de dominio es crearla. Como ya vimos en sesiones anteriores, el comando utilizado para generar una nueva clase de dominio es `grails create-domain-class` seguido del nombre de la nueva clase que queremos crear. Pero, ¿qué pasa en la base de datos cuando generamos nuestras clases de dominio? Para ver exactamente que es lo que pasa en la base de datos cuando creamos una nueva clase de dominio, vamos a tomar como ejemplo la clase *Todo*.

```
package es.ua.expertojava.todo

class Todo {
    String title
    String description
    Date date
    Date reminderDate
    String url
    Boolean done = false
    Category category

    static hasMany = [tags:Tag]
    static belongsTo = [Tag]

    static constraints = {
        title(blank:false)
        description(blank:true, nullable:true, maxSize:1000)
        date(nullable:false)
        reminderDate(nullable:true)
        url(nullable:true, url:true)
        done(nullable:false)
        category(nullable:true)
    }

    String toString(){
        title
    }
}
```

Pasemos a ver que tabla nos ha creado nuestra aplicación en la base de datos. Para esto, vamos a ver una de las características añadidas en las últimas versiones de Grails que es la posibilidad de utilizar una consola web para acceder a la estructura de la base de datos creada. Esta consola se puede acceder desde la dirección <http://localhost:8080/todo/>

*dbconsole*. Hay que tener en cuenta que como parámetro JDBC URL debemos introducir la misma url que tengamos configurada en el archivo *DataSource.groovy*, que por defecto será *jdbc:h2:mem:devDb*.

Una vez dentro de la consola de H2, podemos ejecutar el comando *SHOW COLUMNS FROM TODO* para ver el esquema de la tabla *TODO* y compararla de esta forma con la clase de dominio que nosotros hemos creado para encontrar las siguientes diferencias.

Run Run Selected Auto complete Clear SQL statement:

SHOW COLUMNS FROM TODO

SHOW COLUMNS FROM TODO;

FIELD	TYPE	NULL	KEY	DEFAULT
ID	BIGINT(19)	NO	PRI	(NEXT VALUE FOR PUBLIC.SYSTEM_SEQUENCE_C62D1D63_8703_41ED_8AC6_8DB3938A26B6)
VERSION	BIGINT(19)	NO		NULL
CATEGORY_ID	BIGINT(19)	YES		NULL
DATE	TIMESTAMP(23)	NO		NULL
DESCRIPTION	VARCHAR(1000)	YES		NULL
DONE	BOOLEAN(1)	NO		NULL
REMINDER_DATE	TIMESTAMP(23)	YES		NULL
TITLE	VARCHAR(255)	NO		NULL
URL	VARCHAR(255)	YES		NULL

(9 rows, 16 ms)

## Nuevas columnas

La primera diferencia que encontramos es la existencia de dos nuevas columnas en la tabla correspondiente de la base de datos. La primera de ellas es la columna *id* que además es la clave primaria de la tabla y autoincremental. Esto puede parecer que es algo en contra de Grails ya que no vamos a poder establecer la clave primaria que nosotros queremos, pero en la práctica se ha demostrado que es la mejor forma a la hora de interactuar con la base de datos. La otra nueva columna que se ha generado en la tabla es el campo *version* y servirá para garantizar la integridad transaccional y el bloqueo eficiente de las tablas cuando se realizan operaciones de entrada.

## Nombre de las columnas

El nombre de las propiedades que en la clase de dominio seguían el convenio *CamelCase* (palabras sin espacios en blanco y con la primera letra de cada una de ellas en mayúscula), en la tabla se convierten por defecto al formato *snake\_case* (todas las palabras en minúsculas y separadas por un subrayado bajo *\_*), a pesar de que en la consola de H2 nos aparezca todo

en mayúsculas. Si lo comprobáramos con una base de datos MySQL veríamos este cambio de nombre de las propiedades.

### Claves ajenas

GORM representa las claves ajenas en la tabla con el nombre de la clase/tabla referencia en minúsculas seguido de *\_id*. En nuestro caso tenemos la columna *category\_id* que identifica la relación entre las tareas y categorías, con lo que cada tarea sólo puede pertenecer a una categoría.

### Tipos de datos

Dependiendo de la base de datos utilizada (en nuestro caso H2), GORM transformará los tipos de datos de las propiedades de la clase de dominio en otros tipos de datos en la tabla. Por ejemplo, el tipo de dato *String* se sustituye en la base de datos por *varchar()*, mientras que el tipo de dato *Date* es reemplazado por *timestamp*. Todo esto dependerá de la base de datos utilizada y es posible que varíe entre ellas.

### Relaciones entre clases de dominio

Cualquier aplicación que vayamos a desarrollar, presentará relaciones entre sus clases de dominio. En nuestro ejemplo de scaffolding ya definimos algunas relaciones entre ellas y ahora vamos a ver más en detalle esas relaciones. Las posibles relaciones entre las clases de dominio de una aplicación son:

- Uno a uno
- Uno a muchos
- Muchos a muchos

Si hacemos algo de memoria de la definición de nuestras clases de dominio, recordaremos que utilizábamos la palabra reservada *hasMany* y *belongsTo* para establecer la relación entre las tareas y las etiquetas. También existen la palabra reservada *hasOne*. A continuación veremos en detalle cada una de las posibles relaciones y como especificarlas.

#### Uno a uno

Una relación uno-a-uno se da cuando un objeto de la clase A está únicamente relacionado con un objeto de la clase B y viceversa. Por ejemplo, imaginad la aplicación *Biblioteca* en donde un *libro* sólo puede tener una *operación activa*. Esto se representaría con una relación *uno-a-uno* y podríamos hacerlo de tres formas diferentes.

*Primera forma de establecer una relación uno a uno*

```
.....  
class Book{  
}  
  
class ActiveOperation{  
    ....  
    Book book  
    ....  
}
```

Aquí estamos representando una relación unidireccional desde las operaciones activas hasta los libros. Si necesitamos especificar una relación bidireccional podemos utilizar cualquiera de las siguientes formas.

*Segunda forma de establecer una relación uno a uno*

```

class Book{
    ....
    ActiveOperation actoper
    ....
}

class ActiveOperation{
    ....
    static belongsTo = [book:Book]
    ....
}

```

En este caso, las inserciones y los borrados se realizan en cascada. Si por ejemplo hacemos `new Book(actoper:new ActiveOperation()).save()` en primer lugar se creará una operación activa y posteriormente se creará el libro. De igual forma, si eliminamos el libro asociado a la operación activa, ésta se eliminará también de la base de datos.

*Tercera forma de establecer una relación uno a uno*

```

class Book{
    ....
    static hasOne = [actoper:ActiveOperation]

    static constraints = {
        actoper unique:true
    }
    ....
}

class ActiveOperation{
    ....
    Book book
    ....
}

```

En esta ocasión, se creará una relación bidireccional de uno a uno entre los libros y las operaciones activas y se creará una columna de clave ajena en la tabla `active_operation` llamada `book_id`.

En estos casos, desde la documentación de Grails se aconseja que especifiquemos también una restricción indicando que la propiedad será única.

**Uno a muchos**

Una relación uno-a-muchos se da cuando un registro de la tabla A puede referenciar muchos registros de la tabla B, pero todos los registros de la tabla B sólo pueden referenciar un registro de la tabla A. Por ejemplo, en nuestra aplicación de ejemplo, una tarea sólo puede estar relacionada con una categoría, pero una categoría puede tener muchas tareas. Esto se representa de la siguiente forma en la definición de las clases de dominio.

```

class Category {

```

```

    ....
    static hasMany = [todos:Todo]
    ....
}

class Todo {
    ....
    Category category
    ....
}

```

Con esta definición de las clases de dominio, las actualizaciones y los almacenamientos se realizan en cascada mientras que los borrados sólo se realizarán en cascada si especificamos la otra parte de la relación con un *belongsTo*.

```

class Category {
    ....
    static hasMany = [todos:Todo]
    ....
}

class Todo {
    ....
    static belongsTo = [category:Category]
    ....
}

```

Estas propiedades que representan la relación de uno a muchos son consideradas como colecciones de datos y para insertar o eliminar datos de las mismas únicamente debemos utilizar un par de métodos de GORM que son *addTo()* y *removeFrom()*.

```

//Ejemplo para insertar datos en relaciones
def todo = new Todo(title:"Limpiar cocina", description:"Limpiar la cocina
a fondo", ...)

def category = new Category(name:"Hogar")

category.addToTodos(todo)
category.save()

//Ejemplo para eliminar datos de relaciones
category.removeFromTodos(todo)
category.save()

//Ejemplo para buscar tareas dentro de una categoría
def todosfound = category.todos.find {it.title = "Limpiar cocina"}

```

## Muchos a muchos

En una relación muchos-a-muchos un registro de la tabla A puede referenciar muchos registros de la tabla B y un registro de la tabla B referenciar igualmente muchos registros de la tabla A. En nuestra aplicación ejemplo disponemos de la relación entre tareas y etiquetas en la que

una tarea puede estar relacionada con muchas etiquetas y viceversa. Esto se representa con una relación muchos-a-muchos de la siguiente forma.

```
.....  
class Tag{  
    ....  
    static hasMany = [todos:Todo]  
    ....  
}  
  
class Todo{  
    ....  
    static hasMany = [tags:Tag]  
    static belongsTo = Tag  
    ....  
}  
.....
```

En GORM, una relación muchos-a-muchos se representa indicando en ambas clases la propiedad *hasMany* y en al menos una de ellas la propiedad *belongsTo*.

## 5.2. Restricciones

El siguiente punto que vamos a ver trata a fondo las definiciones de restricciones en las clases de dominios.

### Restricciones predefinidas en GORM

Para controlar estas restricciones, GORM dispone de una serie de funciones comunes predefinidas para establecerlas. Además, también nos permitirá crear nuestras propias restricciones. Hasta ahora, ya hemos visto algunas de estas restricciones como pueden ser *size*, *unique* o *blank* entre otras. Recordamos que para establecer las restricciones de una clase de dominio debemos establecer la propiedad estática *constraints* como en el siguiente ejemplo.

```
.....  
package es.ua.expertojava.todo  
  
class Todo {  
    String title  
    String description  
    Date date  
    Date reminderDate  
    String url  
    Boolean done = false  
    Category category  
  
    static hasMany = [tags:Tag]  
    static belongsTo = [Tag]  
  
    static constraints = {  
        title(blank:false)  
        description(blank:true, nullable:true, maxSize:1000)  
        date(nullable:false)  
        reminderDate(nullable:true)  
        url(nullable:true, url:true)  
    }  
}  
.....
```

```

    done(nullable:false)
    category(nullable:true)
  }

  String toString(){
    title
  }
}

```

A continuación, vamos veremos las restricciones que podemos utilizar directamente con GORM.

Nombre	Descripción	Ejemplo
blank	Valida si la cadena puede quedar o no vacía	login(blank:false)
creditCard	Valida si la cadena introducida es un número de tarjeta correcto	tarjetaCredito(creditCard:true)
email	Valida si la cadena introducida es un correo electrónico correcto	correoElectronico(email:true)
password	Indica si la propiedad es una contraseña, con lo que al introducirla no se mostrará directamente	contrasenya(password:true)
inList	Valida que la propiedad contenga cualquiera de los valores pasados por parámetro	tipo(inList:["administrador", "bibliotecario", "profesor", "socio"])
matches	Valida la cadena introducida contra una expresión regular pasada por parámetro	login(matches:"[a-zA-Z]+")
max	Valida que el número introducido no sea mayor que el número pasado por parámetro	price(max:999F)
min	Valida que el número introducido no sea menor que el número pasado por parámetro	price(min:0F)
minSize	Valida que la longitud de la cadena introducida sea superior al número pasado por parámetro	hijos(minSize:5)
maxSize	Valida que la longitud de la cadena introducida sea inferior al número pasado por parámetro	hijos(maxSize:15)
notEqual	Valida que el objeto asignado a la propiedad	login(notEqual:"admin")

Nombre	Descripción	Ejemplo
	no sea igual que el objeto pasado por parámetro	
nullable	Valida si el objeto puede ser <i>null</i> o no	edad(nullable:true)
range	Valida que el objeto esté dentro del rango pasado por parámetro	edad(range:0..99)
scale	Indica el número de decimales de la propiedad	salario(scale:2)
size	Valida que la longitud de la cadena esté dentro del rango pasado por parámetro	login(size:5..15)
unique	Especifica que la propiedad debe ser única y no se puede repetir ningún objeto	login(unique:true)
url	Valida que la cadena introducida sea una dirección URL correcta	website(url:true)

Pero además de estas restricciones, Grails también tiene otras más que suponen una serie de cambios en la interfaz de usuario generado y aunque no es muy recomendable mezclar este tipo de información en las clases de dominio, cuando usamos el scaffolding en gran parte de nuestro proyecto, es muy cómodo indicarle algunas particularidades a nuestro proyecto para facilitarle su renderizado. Estas restricciones son las siguientes:

Nombre	Descripción	Ejemplo
display	Indica si la propiedad debe mostrarse o no en las vistas generadas por el scaffolding. El valor por defecto es <i>true</i>	password(display:false)
editable	Indica si la propiedad es editable o no. Se suele utilizar en aquellas propiedades de solo lectura	username(editable:false)
format	Especifica el formato aceptado en aquellos tipos que lo acepten, como pueden ser las fechas	dateOfBirth(format:'yyyy-MM-dd')
password	Indica si la propiedad es una contraseña, con lo que al introducirla no se mostrará directamente	password(password:true)

## Construir tus propias restricciones

Pero seríamos un tanto ilusos si pensáramos que el equipo de desarrollo de Grails ha sido capaz de crear todas las posibles restricciones existentes en cualquier aplicación. Ellos no lo han hecho, pero si han dejado la posibilidad de crear nuestras propias restricciones de una forma rápida y sencilla.



Si echamos un vistazo a la clase de dominio *Todo*, podemos detectar que una posible restricción que nos falta por añadir sería la comprobación de que la *fecha* sea posterior a la fecha actual. Para esta restricción, Grails se ha olvidado de nosotros ya que si repasamos la lista de restricciones predefinidas, no encontramos ninguna que satisfaga dicha restricción. Para crear nuevas restricciones, tenemos el closure *validator*. Veamos como quedaría esta restricción:

```
date(nullable:true,
  validator: {
    if (it) {
      return it?.after(new Date())
    }
    return true
  }
)
```

El closure *validator* debe devolver *true* en caso de que la validación se pase correctamente y *false* en caso contrario. Gracias a la variable *it* podemos acceder al valor introducido para analizarlo y comprobar que cumple los requisitos, en este caso, que la fecha no sea anterior a la actual.

Este closure *validator* también puede recibir un par de parámetros que nos permitirá evaluar el valor proporcionado a una propiedad con respecto a cualquier otra propiedad del objeto. El siguiente ejemplo de restricción nos permitirá comprobar que la fecha de recordatorio sea siempre anterior a la fecha de la propia tarea:

```
reminderDate(nullable:true,
  validator: { val, obj ->
    if (val && obj.date) {
      return val.before(obj?.date)
    }
    return true
  }
)
```

En ocasiones es conveniente reutilizar el código de una restricción en diferentes propiedades de diferentes clases de dominio. Para ello en lugar de definir la restricción en la clase de dominio la definiremos en el archivo de configuración *Config.groovy* de la siguiente forma:

```
grails.gorm.default.constraints = {
  max20chars(nullable: false, blank: false, maxSize:20)
}
```

Posteriormente en la propiedad que necesites utilizar esta restricción, debes referenciarla de esta forma:

```
class User {
  ...
  static constraints = {
    password shared: "max20chars"
  }
}
```

```
}  
}
```

---

## Mensajes de error de las restricciones

Sin lugar a dudas, los mensajes que se generan cuando se produce un error en las restricciones de una clase de dominio, no son lo más deseable para un usuario final, así que es necesario que entendamos el mecanismo que utiliza Grails para devolver estos mensajes cuando se producen estos errores.

Si recordamos la clase de dominio *Todo*, tenemos que el título no puede dejarse vacío. Esto lo hacíamos utilizando la restricción *blank:false*. Al intentar crear una tarea sin escribir su título, recibiremos un mensaje de error como el siguiente *La propiedad [title] de la clase [class Todo] no puede ser vacía*. Pero, ¿cómo se encarga Grails de mostrar estos mensajes de error?

Grails implementa un sistema jerárquico para los mensajes de error basado en diferentes aspectos como puede ser el nombre de la clase de dominio, la propiedad o el tipo de validación. Para el caso del error producido al saltarnos la restricción de no dejar en blanco el campo *title* de la clase de dominio *Todo*, dicha jerarquía sería la siguiente.

---

```
todo.title.blank.error.title  
todo.title.blank.error.java.lang.String  
todo.title.blank.error  
todo.title.blank.title  
todo.title.blank.java.lang.String  
todo.title.blank  
blank.title  
blank.java.lang.String  
blank
```

---

Este sería el orden en el que Grails buscaría en nuestro archivo *message.properties* hasta encontrar cualquier propiedad y mostrar su mensaje. En caso de no encontrar ninguna de estas propiedades, se mostraría la cadena asociado a la propiedad *default.blank.message*, que es la que encontramos por defecto en el archivo *message.properties*.

De esta forma, ya podemos personalizar los mensajes de error generados al incumplir las restricciones de nuestras clases de dominio. Simplemente comentar que para las restricciones creadas por nosotros mismos, la jerarquía de los mensajes en el validador introducido en la clase de dominio *Todo* empezaría por *todo.date.validator.error.date* y así sucesivamente tal y como hemos visto anteriormente.

## 5.3. Aspectos avanzados de GORM

Ahora que ya hemos visto la parte básica a la hora de crear una clase de dominio (establecer sus propiedades, las restricciones y las relaciones entre ellas), vayamos un paso más adelante y veamos aspectos algo más avanzados de GORM

### Ajustes de mapeado

Hay ocasiones en que los administradores de las bases de datos quieren, por cualquier motivo, que las columnas de las tablas de sus bases de datos sean nombradas siguiendo un

determinado patrón. Estos motivos pueden ir desde cuestiones de facilidad en la lectura de los campos hasta argumentos como *"así se ha hecho toda la vida y no se va a cambiar"*. El tema está en que hay que conformarse con sus ordenes y acatarlas y encontrar la mejor forma para adaptarse a ellas.

Para solucionar este posible problema que se nos puede plantear en cualquier organización, GORM dispone de una forma rápida y sencilla para ajustar estos nombres de las columnas de las tablas relacionadas. Para ello, GORM emplea una sintaxis de tipo DSL y exige crear un nuevo closure estático llamado *mapping* de la siguiente forma *static mapping = { //Todo el mapeado de la tabla aquí }*. Veamos los posibles cambios que podemos hacer con GORM.

### Nombres de las tablas y las columnas

Si necesitáramos cambiar por ejemplo el nombre de la tabla *todo* por *tbl\_todo*, deberíamos escribir dentro del closure *mapping* el siguiente código *table 'tbl\_todo'*. Si ahora quisiéramos cambiar el nombre de la columna *description* por *desc*, podríamos hacer lo siguiente:

```
class Todo {
    ....
    static mapping = {
        table 'tbl_todo'
        description column: 'desc'
    }
}
```

Con esto, ya tendríamos cambiados los nombres de las tablas y de las columnas para adaptarlos al convenio que nosotros queramos seguir.

### Deshabilitar el campo version

Por defecto, GORM utiliza un campo llamado *version* para garantizar la integridad de los datos y está demostrado que es un buen método. Sin embargo, por cualquier motivo es posible que no queramos tener este campo en nuestra tabla y para ello, simplemente debemos especificar en el mapeado de la clase lo siguiente: *version false*.

### Carga perezosa de los datos

Por defecto, Grails realiza una carga perezosa de los datos de las propiedades. Esto es que no se cargarán en memoria hasta que no sean solicitados por la operación, lo cual nos previene de posibles pérdidas de rendimiento al cargar demasiados datos innecesarios. Sin embargo, si queremos cambiar este comportamiento por defecto de Grails, podemos hacerlo de dos formas:

```
class Person {

    String firstName Pet pet

    static hasMany = [addresses: Address]

    static mapping = {
        addresses lazy: false
        pet fetch: 'join'
    }
}
```

```

    }
}

class Address {
    String street
    String postCode
}

class Pet {
    String name
}

```

Indicando *addresses lazy: false* nos aseguramos que cuando los datos de una persona se carguen en nuestra aplicación, sus direcciones serán cargadas al mismo tiempo. Por otro lado, con la opción *pet fetch:'join'* hacemos básicamente lo mismo salvo que en lugar de hacerlo con un *SELECT* como se estaría haciendo con la opción anterior lo estaríamos haciendo con una operación de tipo *JOIN*. Si queremos reducir el número de queries y mejorar el rendimiento de la operación, deberíamos utilizar la segunda opción. Sin embargo, hay que tener cuidado con estas opciones ya que podemos cargar demasiados datos innecesarios.

### Sistema caché

Uno de las grandes ventajas de Hibernate es la posibilidad de utilizar una caché de segundo nivel, que almacena los datos asociados a una clase de dominio. Para configurar esta caché de segundo nivel, en primer lugar debemos modificar el archivo de configuración *grails-app/conf/DataSource.groovy* para indicarle quien se encargará de gestionar esta caché.

```

hibernate {
    cache.use_second_level_cache = true
    cache.use_query_cache = true
    cache.region.factory_class
= 'net.sf.ehcache.hibernate.EhCacheRegionFactory'
    singleSession = true
    flush.mode = 'manual'
}

```

Posteriormente, para utilizar este sistema de caché, podemos especificarlo en la propia clase de dominio de la siguiente forma:

```

class User {
    ...
    static mapping = {
        table 'tbl_users'
        cache true
    }
}

```

Esta caché sería de tipo *lectura-escritura* e incluiría todas las propiedades, tanto las perezosas como las no perezosas. Pero también es posible ser un poco más específico e indicarle como queremos que se configure la caché de una determinada clase de dominio.

```
class User {
    ...
    static mapping = {
        table 'tbl_users'
        cache usage: 'read-only', include: 'non-lazy'
    }
}
```

Incluso también podemos especificar la caché para una asociación de la siguiente forma:

```
class Person {

    String firstName

    static hasMany = [addresses: Address]

    static mapping = {
        table 'people'
        version false
        addresses column: 'Address', cache: true
    }
}
```

De esta forma, sólo estaríamos cacheando la propiedad *addresses* de la clase *Person*.

### Modificar la clave primaria

En ocasiones podemos necesitar modificar el sistema de clave primaria que utiliza Grails por defecto en el que se añade una propiedad llamada *id* autonumérico. Podríamos por ejemplo establecer como clave primaria de la tabla de las tareas la composición del título y la categoría de la siguiente forma:

```
static mapping = {
    id composite: ['title', 'category']
}
```

### Herencia de clases

La herencia de clases es algo muy común y GORM lo implementa de forma tan sencilla como *extendiendo* la clase. Por ejemplo, imagina el caso de una clase de dominio Usuario que a su vez puede tener 3 tipos distintos. Podríamos tener algo así:

```
class User{
    ....
}

class Administrator extends User{
    ....
}

class Registered extends User{
```

```
    .....
```

```
}

class Guest extends User{
    .....
```

---

Pero, ¿cómo se transformaría esto en la base de datos? GORM habría almacenado todos los datos en una única tabla y además, hubiera añadido un campo llamado *class* que permitiría distinguir el tipo de instancia creada. No obstante, si optásemos por tener una tabla por cada tipo de usuario, podríamos tener lo siguiente:

---

```
class User{
    .....
```

```
}

class Administrator extends User{
    static mapping = {
        table = 'administrator'
    }
}

class Registered extends User{
    static mapping = {
        table = 'registered'
    }
}

class Guest extends User{
    static mapping = {
        table = 'guest'
    }
}
```

---

Otra opción sería especificar en la clase padre la propiedad *tablePerHierarchy* a falso de la siguiente forma:

---

```
class User{
    .....
```

```
    static mapping = {
        tablePerHierarchy false
    }
}
```

---

## Propiedades transitorias

Por defecto, con GORM todas las propiedades definidas en una clase de dominio son persistidas en la base de datos. Sin embargo, en ocasiones es posible que tengamos determinadas propiedades que no deseamos que sean almacenadas en la base de datos, tales como por ejemplo la confirmación de una contraseña por parte de un usuario. Para ello, podemos añadir una nueva propiedad llamada *transients* a la clase de dominio correspondiente con la propiedad que no deseamos persistir en la base de datos.

```
class User {
    static transients = ["confirmPassword"]

    String username
    String password
    String confirmPassword
    String name
    String surnames
}
```

## Eventos GORM

GORM dispone de dos métodos que se llaman automáticamente antes y después de insertar y actualizar las tablas de la base de datos. Estos métodos son *beforeInsert()* y *beforeUpdate()*. Gracias a estos métodos, vamos a poder realizar determinadas operaciones siempre antes de insertar o actualizar datos de nuestros registros. Por ejemplo, imagina el caso en el que tuviéramos que almacenar quien creó una tarea y quien fue la última persona en actualizarla. Necesitaríamos por ejemplo un par de propiedades llamadas *createdBy* y *lastModifiedBy*. Para automáticamente actualizar esta información en nuestra clase de dominio, podríamos utilizar los métodos *beforeInsert()* y *beforeUpdate()* de la siguiente forma:

```
class Todo {
    ....
    User createdBy
    User lastModifiedBy

    def beforeInsert = {
        createdBy = session?.user
        lastModifiedBy = session?.user
    }

    def beforeUpdate = {
        lastModifiedBy = session?.user
    }
}
```

Además de estos dos métodos comentados, GORM también dispone de los métodos *beforeDelete()*, *beforeValidate()*, *afterInsert()*, *afterUpdate()*, *afterDelete()* y *onLoad()*.

## Fechas automáticas

Grails además presenta una característica conocida como *automatic timestamping*. Con esta característica si queremos saber de una instancia de una determinada clase cuando se creó y cuando se modificó por última vez, simplemente debemos crear dos propiedades de tipo *Date* en la clase de dominio llamadas *dateCreated* y *lastUpdated* y Grails se encargará automáticamente de rellenar esta información por nosotros.

```
class Todo {
    ....
    Date dateCreated
    Date lastUpdated
}
```

---

```
}

```

---

Si por cualquier motivo, deseamos tener estas dos propiedades pero desactivar el *automatic timestamping*, podemos hacerlo de la siguiente forma:

---

```
static mapping = {
    autoTimestamp false
}
```

---

## 5.4. 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 *Todo*, detectaremos la presencia de los métodos *delete()* y *save()*. También podemos comprobar como no existe ninguno de estos dos métodos implementados en la clase de dominio *Todo*. Entonces, ¿cómo es capaz Grails de persistir la información en la base de datos si ninguno de estos dos métodos existen?

En Grails tenemos varias formas de realizar consultas a la base de datos que son:

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

### Consultas dinámicas de GORM

Básicamente, la consultas dinámicas de GORM son muy similares a los métodos que habéis visto en el módulo de JPA con lo que únicamente veremos un resumen de las mismas.

Método	Descripción	Ejemplo
<i>count()</i>	Devuelve el número total de registros de una determinada clase de dominio	<i>Todo.count()</i>
<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>Todo.countByTitle('Pintar cocina')</i>
<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>Todo.findByTitle('Pintar cocina')</i>
<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	<i>Todo.findWhere(["title":"Pintar cocina"])</i>
<i>findAllBy()</i>	Devuelve todos los registros encontrados de una determinada	<i>Todo.findAllByTitleIlike('%cocina%')</i>



Método	Descripción	Ejemplo
	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>Todo.findAllWhere(["title":"Pintar cocina"])</i>
<i>get()</i>	Devuelve el registro de una determinada clase de dominio cuyo identificador coincida con el pasado como parámetro	<i>Todo.get(1)</i>
<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	<i>Todo.getAll([1,3])</i>
<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>Todo.list(["max":10, "offset":0, "sort":"title", "order":"asc"])</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>	<i>Todo.listOrderBy(["max":10, "offset":0, "order":"asc"])</i>

Podemos además encadenar varias propiedades junto con el operador utilizado, como por ejemplo:

```
.....
Todo.findAllByTitleAndDescriptionAndDone('Pintar cocina', null, false)
.....
```

Seguido de la propiedad podemos indicar el método a seguir en la comparación de registros. En la siguiente tabla se muestra una tabla resumen de esos métodos que podemos utilizar en las comparaciones.

Método	Descripción	Ejemplo
<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>Category.findAllByNameInList(["Hogar", "Trabajo"])</i>
<i>LessThan</i>	Devuelve aquellos registros en los que el valor de la propiedad dada sea menor que el valor pasado por parámetro	<i>House.findAllByPriceLessThan(150000)</i>
<i>LessThanEquals</i>	Devuelve aquellos registros en los que el valor de la propiedad dada sea	<i>House.findAllByPriceLessThanEquals(150000)</i>

Método	Descripción	Ejemplo
	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	Car.findAllByPriceGreaterThan(25000)
GreaterThanEquals	Devuelve aquellos registros en los que el valor de la propiedad dada sea mayor o igual que el valor pasado por parámetro	Car.findAllByPriceGreaterThanEquals(25000)
Like	Es equivalente a una expresión <i>LIKE</i> en una sentencia SQL	Todo.findAllByTitleLike('%cocina%')
ILike	Similar al método <i>Like</i> salvo que en esta ocasión <i>case insensitive</i>	Todo.findAllByTitleIlike('%cocina%')
NotEqual	Devuelve aquellos registros en los que el valor de la propiedad dada no sea igual al valor pasado por parámetro	Todo.findAllByDoneNotEqual(true)
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.	Car.findAllByPriceBetween(10000,20000)
NotNull	Devuelve aquellos registros en los que el valor de la propiedad dada no sea nula. No se necesita ningún argumento.	User.findAllByDateOfBirthNotNull()
IsNull	Devuelve aquellos registros en los que el valor de la propiedad dada sea nula	User.findAllByDateOfBirthIsNull()

## 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 la primera tarea que encuentre en la clase de dominio *Todo*.

```
def sentenciaHQL1 = Todo.find("From Todo as t")
```

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 determinadas tareas de nuestra aplicación.

```
def hqlsentence2 = Todo.findAll("from Todo as t where t.title='Pintar
cocina'")

def hqlsentence3 = Todo.findAll("from Todo as t where t.title=?",
["Escribir tests unitarios"])

def hqlsentence4 = Todo.findAll("from Todo as t where t.title=:title",
[title:"Cocinar pastel"])
```

En la primera de ellas, la sentencia HQL se pasa como parámetro al método *findAll()* tal cual y sin utilizar ningún parámetro. También es posible pasar parámetros a la sentencia HQL tal y como se ha hecho en la segunda forma. Utilizando el carácter *?* le especificamos que es valor se pasará en forma de parámetro posicional, es decir, el primer carácter *?* 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 *title*, 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 la *fecha* de la tarea cuyo título sea "Pintar cocina".

```
Todo.executeQuery("select date from Todo t where t.title='Pintar cocina'")
```

## Consultas Criterias 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 las siguientes 15 tareas a realizar en los próximos 10 días ordenadas ascendentemente.

```
void nextTodos() {
    def c = Todo.createCriteria()
    def result = c{
        between("date", new Date(), new Date()+10)
        maxResults(15)
        order("date", "asc")
    }
}
```

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

<b>Criterio</b>	<b>Descripción</b>	<b>Ejemplo</b>
between	Cuando el valor de la propiedad se encuentra entre dos valores	<i>between("date",new Date()-10, new Date())</i>
eq	Cuando el valor de una propiedad es igual al valor pasado por parámetro	<i>eq("make","volkswagen")</i>
eqProperty	Cuando dos propiedades tienen el mismo valor	<i>eqProperty("startDate","endDate")</i>
gt	Cuando el valor de una propiedad es mayor que el valor pasado por parámetro	<i>gt("date",new Date()-5)</i>
gtProperty	Cuando el valor de una propiedad es mayor que el valor de la propiedad pasada por parámetro	<i>gtProperty("startDate","endDate")</i>
ge	Cuando el valor de una propiedad es mayor o igual que el valor pasado por parámetro	<i>ge("date", new Date()-5)</i>
geProperty	Cuando el valor de una propiedad es mayor o igual que el valor de la propiedad pasada por parámetro	<i>geProperty("startDate", "endDate")</i>
idEq	Cuando el identificador de un objeto es igual al valor pasado por parámetro	<i>idEq(1)</i>
ilike	La sentencia SQL like, pero en modo <i>case insensitive</i>	<i>ilike("name", "fran%")</i>
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	<i>'in'("age",[18..65])</i>
isEmpty	Cuando la propiedad se encuentra vacía	<i>isEmpty("description")</i>
isNotEmpty	Cuando la propiedad no se encuentra vacía	<i>isNotEmpty("description")</i>
isNull	Cuando el valor de la propiedad es <i>null</i>	<i>isNull("description")</i>
isNotNull	Cuando el valor de la propiedad no es <i>null</i>	<i>isNotNull("description")</i>
lt	Cuando el valor de una propiedad es menor que el valor pasado por parámetro	<i>lt("startDate",new Date()-5)</i>
ltProperty	Cuando el valor de una propiedad es menor que el valor de la propiedad pasada por parámetro	<i>ltProperty("startDate","endDate")</i>

Criterio	Descripción	Ejemplo
le	Cuando el valor de una propiedad es menor o igual que el valor pasado por parámetro	<i>le("startDate", new Date()-5)</i>
leProperty	Cuando el valor de una propiedad es menor o igual que el valor de la propiedad pasada por parámetro	<i>leProperty("startDate", "endDate")</i>
like	La sentencia SQL like	<i>like("name", "Fran%")</i>
ne	Cuando el valor de una propiedad no es igual al valor pasado por parámetro	<i>ne("model", "volkswagen")</i>
neProperty	Cuando el valor de una propiedad no es igual al valor de la propiedad pasada por parámetro	<i>neProperty("startDate", "endDate")</i>
order	Ordena los resultados por la propiedad pasada por parámetro y en el orden especificado (asc o desc)	<i>order("name", "asc")</i>
rlike	Similar a la sentencia SQL like, pero utilizando expresiones regulares. Sólo está disponible para Oracle y MySQL	<i>rlike("name", /Fran.+/)</i>
sizeEq	Cuando el tamaño del valor de una determinada propiedad es igual al pasado por parámetro	<i>sizeEq("name", 10)</i>
sizeGt	Cuando el tamaño del valor de una determinada propiedad es mayor al pasado por parámetro	<i>sizeGt("name", 10)</i>
sizeGe	Cuando el tamaño del valor de una determinada propiedad es mayor o igual al pasado por parámetro	<i>sizeGe("name", 10)</i>
sizeLt	Cuando el tamaño del valor de una determinada propiedad es menor al pasado por parámetro	<i>sizeLt("name", 10)</i>
sizeLe	Cuando el tamaño del valor de una determinada propiedad es menor o igual al pasado por parámetro	<i>sizeLe("name", 10)</i>
sizeNe	Cuando el tamaño del valor de una determinada propiedad no es igual al pasado por parámetro	<i>sizeNe("name", 10)</i>

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("date", new Date()-10, new Date())
    ilike("content", "%texto%")
}

or {
    between("date", new Date()-10, new Date())

```

```
    ilike("content", "%texto%")
}

not {
    between("date", new Date()-10, new Date())
    ilike("content", "%texto%")
}
```

---

## 5.5. 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 (<https://www.youtube.com/watch?v=91C7ax0UAAc>). 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 ubican en el directorio *grails-app/services*.

En la aplicación que estamos desarrollando como ejemplo, vamos a crear un método en un servicio que devuelva aquellas tareas que estén planeadas para los próximos X días. Para ello, en primer lugar debemos crear el esqueleto del servicio con el comando *grails create-service es.ua.expertojava.todo.todo*. Este comando, además del servicio, también creará un test unitario. La clase de servicio que hemos creado tiene la siguiente estructura:

---

```
package es.ua.expertojava.todo

import grails.transaction.Transactional

@Transactional
class TodoService {

    def serviceMethod() {

    }
}
```

---

Como vemos, la clase creada especifica la anotación *@Transactional* que indica que la clase completa con todos los métodos que añadamos serán transaccionales. Esto significa que si en algún método se produce algún error, las operaciones realizadas hasta el momento del error serán desechadas. Sin embargo, existe una forma de ser algo más selectivo e indicar que métodos serán transaccionales y cuales no. Esto se hace también con anotaciones y este es un ejemplo de lo que podríamos tener:

---

```
import grails.transaction.Transactional

class BookService {

    @Transactional(readonly = true)
```

```
def listBooks() {
    Book.list()
}

@Transactional
def updateBook() {
    // ...
}

def deleteBook() {
    // ...
}
}
```

En el método *listBooks()*, le estamos especificando que el método será transaccional de solo lectura. Por otro lado, el método *updateBook()* será transaccional de lectura-escritura y por último, y que sirva únicamente como ejemplo, tenemos el método *deleteBook()* que al no indicarle nada, le estamos diciendo a Grails que no es transaccional (por supuesto, esta operación debería ser transaccional. Tómallo únicamente como un ejemplo).

Cuando creábamos el servicio *TodoService*, Grails generaba automáticamente un método vacío llamado *serviceMethod()* que ahora vamos a sustituir por uno propio que devuelva una serie de tareas comprendidas entre un par de fechas. El método recibirá una fecha inicio y fin y además, un mapa con una serie de parámetros que utilizaremos para paginar en caso de que sea necesario. El método devolverá una colección de instancias de la clase *Todo*. También crearemos un método que devuelva un contador de cuantas tareas coinciden con ese criterio.

```
package es.ua.expertojava.todo

import grails.transaction.Transactional

class TodoService {

    def getNextTodos(Integer days, params) {
        Date now = new Date(System.currentTimeMillis())
        Date to = now + days
        Todo.findAllByDateBetween(now, to, params)
    }

    def countNextTodos(Integer days) {
        Date now = new Date(System.currentTimeMillis())
        Date to = now + days
        Todo.countByDateBetween(now, to)
    }
}
```

El siguiente paso, será modificar el controlador de la clase de dominio *Todo* para crear un nuevo método que utilice este servicio recién creado. El método lo vamos a llamar *listNextTodos()*.

```
class TodoController {
    def todoService

    ....
}
```



```

    def listNextTodos(Integer days) {
        respond todoService.getNextTodos(days, params),
            model:[todoInstanceCount:
todoService.countNextTodos(days)],
            view:"index"
    }
    ....
}

```

Al inicio del controlador debemos crear también la variable *def todoService* 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.

El método es muy similar al método *index()* con la excepción de que ahora le pasaremos el parámetro *view:"index"* para que cuando la petición se realice desde una aplicación web, esta sea renderizada en una página HTML.

Con esto tenemos que si accedemos a través del navegador a la url <http://localhost:8080/todo/todo/listNextTodos?days=2> la aplicación nos mostrará las tareas que tengamos que hacer en los próximos dos días. Si por otro lado ejecutamos desde la línea de comandos:

```

curl -v -H "Accept: text/xml" -H "Content-type: text/xml" -X GET http://
localhost:8080/todo/todo/listNextTodos?days=2

```

el resultado será un xml con el contenido de las próximas tareas. En el caso en que necesitemos el resultado en formato JSON:

```

curl -v -H "Accept: application/json" -H "Content-type: application/json"
-X GET http://localhost:8080/todo/todo/listNextTodos?days=2

```

## Ámbito de los servicios

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 TodoService{
    static scope = 'session'
    ...
}
```

## Servicios web

Ya hemos visto durante las sesiones anteriores que es muy sencillo generar en Grails una aplicación que sepa responder a las distintas peticiones de una aplicación web, bien sea a través de una página web o bien desde una aplicación nativa utilizando REST. Pero, ¿qué es exactamente REST?

Los servicios web REST son algo más que devolver datos en formato XML o JSON sino que consiste en una forma de trabajar que en función del método HTTP utilizado en la petición se debe procesar mediante una acción u otra, tal y como se representa en la siguiente tabla.

Método	Acción
GET	show()
PUT	update()
POST	save()
DELETE	delete()

A partir de esta tabla, sólo necesitaríamos una url para poder realizar todas las operaciones sobre nuestras clases de dominio. Imagina por ejemplo las operaciones de tipo CRUD sobre los usuarios. Con REST, sólo deberíamos tener una posible url que sería [http://localhost:8080/todo/todo/id\\_todo](http://localhost:8080/todo/todo/id_todo), donde el identificador de la tarea sería optativo. Veamos en otra tabla como quedarían todas las posibles peticiones para realizar las operaciones del usuario.

URL	Método HTTP	Acción
<a href="http://localhost:8080/todo/todo">http://localhost:8080/todo/todo</a>	GET	<i>show()</i> , debería mostrar un listado de las tareas
<a href="http://localhost:8080/todo/todo/4">http://localhost:8080/todo/todo/4</a>	GET	<i>show()</i> , debería mostrar los datos de la tarea con <i>id = 4</i>
<a href="http://localhost:8080/todo/todo/4">http://localhost:8080/todo/todo/4</a>	PUT	<i>update()</i> , debería actualizar los datos de la tarea con <i>id = 4</i>
<a href="http://localhost:8080/todo/todo">http://localhost:8080/todo/todo</a>	POST	<i>save()</i> , debería crear una nueva tarea con los datos pasados en el formulario
<a href="http://localhost:8080/todo/todo/4">http://localhost:8080/todo/todo/4</a>	DELETE	<i>delete()</i> , debería eliminar la tarea con <i>id = 4</i>



## 5.6. Ejercicios

### Construye tus propias restricciones (0.25 puntos)

En la clase de dominio de las tareas tenemos tanto una fecha para la tarea como una fecha de recuerdo. Vamos a imponer como restricción que la fecha para recordar la tarea deba ser siempre anterior a la fecha de realización de la misma. Crea para ello una restricción que nos permita conseguir esto.

Crea también una nueva propiedad en las etiquetas para especificar un color. Este color vendrá en formato RGB, que te recuerdo está especificado por el carácter # seguido de 6 caracteres hexadecimales, como por ejemplo #23FA8F. Define esta restricción globalmente con el nombre *rgbcolor* para que pueda ser reutilizada en cualquier otra propiedad de nuestra aplicación.

### Asegurar los borrados de las categorías y las tareas (0.25 puntos)

En la aplicación actualmente tenemos un problema que nos impide eliminar categorías por la relación con las tareas y la clave ajena que se genera a nivel de base de datos con lo que antes de proceder al borrado de la categoría, debemos asegurarnos que ésta no esté asignada a ninguna tarea.

Crea el servicio *CategoryService.groovy* y mueve el eliminado que ahora mismo se está realizando en el controlador *CategoryController.groovy* para que asigne a null todas las tareas de la categoría a eliminar.

Soluciona al mismo tiempo el problema existente a la hora de eliminar tareas por la relación existente entre ellas y las etiquetas. Mueve el borrado de las tareas a un método en el servicio adecuado.

### Consultas a base de datos (0.25 puntos)

Vamos a crear un nuevo informe en nuestra aplicación que nos permita listar todas las tareas ordenadas por fecha y cuya categoría coincida con las pasadas por parámetro. Necesitaremos crear una vista y su correspondiente método (*listByCategory()*) en el controlador *TodoController()* para poder seleccionar aquellas categorías de las que queramos listar sus tareas.

A continuación, deberás implementar un nuevo método en el controlador *TodoController()* que devuelva todas las tareas cuya categoría coincidan con aquellas que el usuario haya seleccionado en el paso anterior. En esta vista puedes mostrar todas las categorías junto a un checkbox para que el usuario pueda seleccionar varios al mismo tiempo.

Por último, edita los menús de navegación de la aplicación para añadir el nuevo informe que acabamos de generar en todas páginas relativas a las tareas.

### ¿Cuándo han sido realizadas las tareas? (0.50 puntos)

El autotimestamp es una característica de Grails que se suele utilizar para saber cuando se crean y actualizan las instancias de nuestras clases de dominio. En primer lugar vamos a añadir esta característica de Grails a las tareas.

A continuación, vamos a crear una nueva propiedad en las tareas que nos permite saber cuando se han realizado las tareas y llama a esta propiedad *dateDone*. Para rellenar esta

propiedad vamos a crear un servicio que se encargue de esta lógica y a su vez, mover el guardado que se está realizando actualmente en el controlador.

Para ello añadiremos en el servicio *TodoService* un nuevo método llamado *saveTodo(Todo todoInstance)* de tal forma que cuando la propiedad *done* sea cierta, completaremos con la fecha actual la nueva propiedad *dateDone*. Este método deberá ser utilizado tanto cuando se crea como cuando se actualiza una tarea.

A continuación, crearemos un nuevo método en el servicio *TodoService()* llamado *lastTodosDone(Integer hours)* que nos permitirá pasar como parámetro un número de horas y nos devuelva todas las últimas tareas realizadas en ese intervalo de tiempo, es decir, desde el momento actual hasta hace *hours* horas.

## 6. Framework de test Spock.

En esta sesión veremos en profundidad como comprobar que nuestra aplicación desarrollada en Grails funciona tal y como esperamos y para ello utilizaremos un framework para la realización de tests llamado Spock. Empezaremos viendo conceptos generales tanto de tests como de Spock y terminaremos viendo su aplicación en una aplicación en Grails.

### 6.1. Introducción a los tests

#### ¿Qué son los tests?



#### Definición

Los tests de software se pueden definir como el proceso empleado para comprobar la corrección de un programa informático y se puede considerar como una parte más en el proceso del control de calidad.

#### Tipos de tests de software

Básicamente existen tres tipos de tests de software:

- *Tests unitarios*, que consisten en comprobar el funcionamiento de un determinado módulo sin tener en cuenta a los demás. En este tipo de pruebas tampoco se tiene acceso a recursos como bases de datos, sistemas de ficheros o recursos de red.
- *Tests de integración*, suponen un nivel por encima a los tests unitarios y en ellos se añaden recursos como bases de datos, sistemas de ficheros o recursos de red.
- *Tests funcionales*, consiste en comprobar que la funcionalidad de nuestra aplicación es la esperada. Básicamente este tipo de pruebas se basan en automatizar las típicas pruebas de rellenar datos en formularios, hacer clicks en botones y analizar la respuesta obtenida.

#### Otros frameworks de tests

En Java disponemos de varias alternativas para el desarrollo de los tests, pero básicamente disponemos de dos que son las más conocidas y utilizadas:

- JUnit (<http://junit.org>), posiblemente sea el framework más extendido entre la comunidad Java.
- TestNG (<http://testng.org>), es un framework que permite testear nuestra aplicación con pruebas unitarias, de integración y funcionales.

Sin embargo, en la comunidad Groovy cada vez se está extendiendo más el uso de un framework conocido como Spock (<https://code.google.com/p/spock/>). Spock surgió en el año 2009 y su creador es Peter Niederwieser (@pniederw<sup>18</sup>) y sin duda alguna destaca entre el resto de frameworks por la expresividad de su lenguaje y su integración con Groovy y Java. Estas son algunas de las razones por las que vamos a utilizar Spock:

- Fácil de aprender
- Integración con Groovy

<sup>18</sup> <https://twitter.com/pniederw>

- Eliminación aserciones innecesarias
- Información detallada en las salidas de los tests
- Diseñado desde el punto de vista del usuario
- Independientemente de la teoría que sigas para desarrollar tus tests (TDD, BDD, etc), Spock se adaptará a tus necesidades
- Lenguaje altamente expresivo
- Compatible con JUnit
- Combina lo mejor de otros conocidos frameworks como JUnit o jMock

## 6.2. Framework de tests Spock

### Instalación en Grails

Spock en Grails viene como un plugin con lo que únicamente tendremos que añadir el plugin en el archivo de configuración *BuildConfig.groovy*, tal y como se explica en la página del plugin (<http://grails.org/plugin/spock>) para versiones superiores a Grails 2.2

```
grails.project.dependency.resolution = {
  repositories {
    grailsCentral()
    mavenCentral()
  }
  dependencies {
    test "org.spockframework:spock-grails-support:0.7-groovy-2.0"
  }
  plugins {
    test(":spock:0.7") {
      exclude "spock-grails-support"
    }
  }
}
```

Para que el plugin sea instalado y el IDE reconozca las nuevas clases, necesitaremos lanzar algún *target* de Grails, como por ejemplo *grails test-app*. Una vez instalado el plugin, ya podremos empezar a desarrollar nuestros primeros tests.

### Tests en Spock

Por defecto en Spock, todas las clases que creamos para nuestros tests deben extender la clase *spock.lang.Specification*, que inyectará una serie de métodos y utilidades en nuestra clase extendida.

A diferencia de otros frameworks de tests, Spock presenta una sintaxis muy sencilla de leer que se basa en la utilización de bloques de código. Este sería un ejemplo básico de test con Spock:

```
package es.ua.expertojava.todo

import spock.lang.Specification
```

```

class MyFirstTest extends Specification {

    def "El operador suma funciona correctamente"() {
        expect:
            10 == 2 + 8
        and:
            10 == 0 + 10
        and:
            10 == 10 + 0
    }
}

```

Este formato de test en Spock podríamos decir que es el más sencillo de los que podemos encontrar y como puedes observar la sintaxis es muy sencilla de entender. Lo primero que cabe destacar aunque probablemente ya te hayas dado cuenta es el nombre de los métodos que están entrecomillados y podemos llamarlo como queramos y utilizar espacios o caracteres especiales (no ingleses) como acentos.

Destacar también que no es necesario utilizar aserciones dentro del bloque *expect*. Pero veamos otro ejemplo un poco más elaborado y que nos introducirá un nuevo formato de test en Spock.

```

def "El método plus de los números funciona correctamente"() {
    when:
        def result = 10.plus(2)
    then:
        result == 12
}

```

Este formato de spock sea posiblemente el más común en Spock y en él vemos un bloque *when* y un bloque *then*. El bloque *when* se utiliza habitualmente para lanzar el método a probar y en este caso también para almacenar el resultado, mientras que el bloque *then* se utiliza para comprobar que la respuesta proporcionada por el método es la correcta. De nuevo, en el bloque *then* no es necesario utilizar aserciones.

Pero, ¿qué pasa si queremos probar varios casos en la misma sentencia? Para este caso, Spock introduce el bloque *where* en donde con una sintaxis muy sencilla de entender en forma de tabla, vamos a poder comprobar varios al mismo tiempo. Veamos el anterior ejemplo mejorado:

```

@Unroll
def "El método plus funciona con el sumador #sumador y el sumando #sumando"() {
    when:
        def result = sumador.plus(sumando)
    then:
        result == resultadoEsperado
    where:
        sumador | sumando | resultadoEsperado
        0        | 0       | 0
        0        | 1       | 1
        1        | 1       | 2
        -1       | 0       | -1
}

```

```

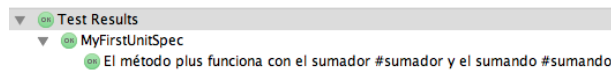
    0      |      -1      |      -1
    -1     |      -1      |      -2
    2      |      1        |      3
}

```

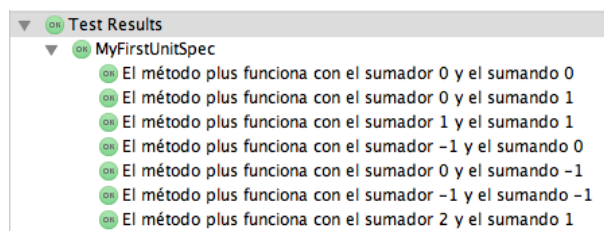
En este ejemplo vemos la forma en la que Spock crea variables dentro del bloque *where* sin definir su tipo o ni tan siquiera con la palabra reservada *def*. De esta forma, en un único test hemos podido comprobar el método *plus()* con varios valores en una única sentencia.

Por otro lado, vemos también como el nombre del test introduce los valores de las variables definidas dentro del bloque *where* de tal forma que cuando ejecutemos este test, si alguno falla veremos rápidamente que caso es el problemático.

Además, la anotación *@Unroll* nos permite que cada uno de los casos en el bloque *where* se despliegue como un nuevo test, lo cual de nuevo, facilita la detección rápida de los errores. Esta sería la salida producida en nuestro editor sin la anotación *@Unroll*.



y esta es la producida con la anotación *@Unroll*.



Como puedes observar, la anotación *@Unroll* nos proporciona más información del resultado de los tests.

El siguiente ejemplo es idéntico al anterior pero en lugar de utilizar una tabla utilizamos listas con valores:

```

@Unroll
def "El método plus funciona con el sumador #sumador y el sumando #sumando
    utilizando listas"() {
    when:
        def result = sumador.plus(sumando)
    then:
        result == resultadoEsperado
    where:
        sumador << [0, 0, 1, -1, 0, -1, 2]
        sumando << [0, 1, 1, 0, -1, -1, 1]
        resultadoEsperado << [0, 1, 2, -1, -1, -2, 3]
}

```

El último bloque de un test Spock que nos queda por ver es el primero de todos, el bloque *given*, que nos servirá para realizar todo lo necesario antes de ejecutar el método a testear. Este podría ser un ejemplo:



---

```

def "El método para concatenar cadenas añade un signo + entre las cadenas concatenadas"() {
    given:
        String.metaClass.concat = { String c ->
            "${delegate}+${c}"
        }
    expect:
        "cadena1".concat("cadena2") == "cadena1+cadena2"
    cleanup:
        String.metaClass = null
}

```

---

Y como siempre, no olvides devolver la clase metaprogramada con `String.metaClass = null`.

### 6.3. Spock en Grails

Una vez hemos visto los primeros ejemplos de tests con Spock en Groovy, vamos a ver ejemplos de como testear los diferentes artefactos en las aplicaciones desarrolladas con Grails. Pero antes de continuar, hay que comentar que en Spock disponemos de unos métodos que se ejecutan antes y después de cada tests o de cada clase Spock. Estos métodos son:

- `setupSpec()`: se ejecuta **antes del primer test** de una determinada clase de tests con Spock
- `cleanupSpec()`: se ejecuta **después del último test** de una determinada clase de tests con Spock
- `setup()`: se ejecuta **antes de cada test** de una determinada clase de tests con Spock
- `cleanup()`: se ejecuta **después de cada test** de una determinada clase de tests con Spock

### Tests con Spock sobre clases de dominio en Grails

Empecemos pues con los tests sobre los artefactos de una aplicación Grails y lo haremos sobre las clases de dominio y más concretamente sobre las restricciones.

Partiendo de nuestra clase de dominio `Category`,

---

```

package es.ua.expertojava.todo

class Category {

    String name
    String description

    statichasMany = [todos:Todo]

    static constraints = {
        name(blank:false)
        description(blank:true, nullable:true, maxSize:1000)
    }

    String toString(){
        name
    }
}

```

```
}
```

Vamos a comprobar que las restricciones impuestas sobre las propiedades *name* y *description* se cumplen. Para ello, vamos a reutilizar un test unitario creado cuando generábamos las clases de dominio llamado *CategorySpec.groovy* dentro del paquete *es.ua.expertojava.todo*.

Lo ideal es que cada test unitario compruebe únicamente un aspecto, así que vamos a seguir este consejo y crearemos los siguientes tests:

```
package es.ua.expertojava.todo

import grails.test.mixin.TestFor
import spock.lang.Specification
import spock.lang.Unroll

/**
 * See the API for {@link
 * grails.test.mixin.domain.DomainClassUnitTestMixin} for usage instructions
 */
@TestFor(Category)
class CategorySpec extends Specification {

    def setup() {
    }

    def cleanup() {
    }

    def "El nombre de la categoría no puede ser la cadena vacía"() {
        given:
            def c1 = new Category(name: "")
        when:
            c1.validate()
        then:
            c1?.errors['name']
    }

    def "Si el nombre no es la cadena vacía, este campo no dará problemas"() {
        given:
            def c1 = new Category(name: "algo")
        when:
            c1.validate()
        then:
            !c1?.errors['name']
    }

    def "La descripción de la categoría puede ser la cadena vacía"() {
        given:
            def c1 = new Category(description: "")
        when:
            c1.validate()
        then:
            !c1?.errors['description']
    }
}
```

```
def "Si la descripción es la cadena vacía, este campo no dará
problemas"() {
  given:
    def c1 = new Category(description: "")
  when:
    c1.validate()
  then:
    !c1?.errors['description']
}

def "La descripción de la categoría puede ser null"() {
  given:
    def c1 = new Category(description: null)
  when:
    c1.validate()
  then:
    !c1?.errors['description']
}

@Unroll
def "Si la descripción tiene menos de 1001 caracteres, no dará
problemas"() {
  given:
    def c1 = new Category(description: "a"*characters)
  when:
    c1.validate()
  then:
    !c1?.errors['description']
  where:
    characters << [0,1,999,1000]
}

@Unroll
def "Si la descripción tiene más 1000 caracteres, dará problemas"() {
  given:
    def c1 = new Category(description: "a"*characters)
  when:
    c1.validate()
  then:
    c1?.errors['description']
  where:
    characters << [1001,1002]
}
}
```

---

Con esta batería de tests nos aseguraremos de que las instancias creadas para la clase de dominio Categoría cumplen los requisitos definidos en las restricciones. De igual forma, vamos a realizar un último test para comprobar que cuando se crea una instancia de Categoría correctamente, el valor devuelto por ésta coincide con la sobrecarga del método *toString()* que hayamos hecho en la clase de dominio.

---

```
...
def "La instancia de Categoría devuelve su nombre por defecto"() {
  expect:
    new Category(name: "The category name").toString() == "The
category name"
```

```
    }
    ...

```

## Tests con Spock sobre controladores en Grails

Una vez hemos visto como podemos testear nuestras clases de dominio, vamos a ver como podemos hacer lo mismo con los controladores y para ello vamos a basarnos en el código generado para el controlador *CategoryController.groovy*. Cuando creábamos este controlador, al mismo tiempo se creaba un test *CategoryControllerSpec.groovy* con código también generado, pero que nosotros necesitaremos completar, en este caso, con la información necesaria procedente de la clase de dominio *Category*.

Si abrimos el archivo *CategoryControllerSpec.groovy* con nuestro editor veremos como en la parte superior hay definido un método con un comentario de tipo *TODO* para avisarnos de que hay algo que debemos completar, que básicamente será la información necesaria para crear una instancia de tipo *Tag* de forma correcta.

```
package es.ua.expertojava.todo

import grails.test.mixin.*
import spock.lang.*

@TestFor(CategoryController)
@Mock(Category)
class CategoryControllerSpec extends Specification {

    def populateValidParams(params) {
        assert params != null
        // TODO: Populate valid properties like...
        //params["name"] = 'someValidName'
    }
    ...

```

En nuestro caso, sólo será necesario añadir la información referente al nombre, para que los tests se ejecuten de forma satisfactoria, con lo que podríamos tener algo así:

```
def populateValidParams(params) {
    assert params != null
    params["name"] = 'Category name'
}

```

Si ahora ejecutamos en línea de comandos

```
grails test-app CategoryControllerSpec unit:

```

comprobaremos como todos los tests se han ejecutado correctamente. Veamos como quedaría el código del test:

```
package es.ua.expertojava.todo

import grails.test.mixin.*

```

```
import spock.lang.*

@TestFor(CategoryController)
@Mock(Category)
class CategoryControllerSpec extends Specification {

    def populateValidParams(params) {
        assert params != null
        params["name"] = 'Category name'
    }

    void "Test the index action returns the correct model"() {

        when:"The index action is executed"
            controller.index()

        then:"The model is correct"
            !model.categoryInstanceList
            model.categoryInstanceCount == 0
    }

    void "Test the create action returns the correct model"() {
        when:"The create action is executed"
            controller.create()

        then:"The model is correctly created"
            model.categoryInstance != null
    }

    void "Test the save action correctly persists an instance"() {

        when:"The save action is executed with an invalid instance"
            request.contentType = FORM_CONTENT_TYPE
            request.method = 'POST'
            def category = new Category()
            category.validate()
            controller.save(category)

        then:"The create view is rendered again with the correct model"
            model.categoryInstance != null
            view == 'create'

        when:"The save action is executed with a valid instance"
            response.reset()
            populateValidParams(params)
            category = new Category(params)

            controller.save(category)

        then:"A redirect is issued to the show action"
            response.redirectedUrl == '/category/show/1'
            controller.flash.message != null
            Category.count() == 1
    }

    void "Test that the show action returns the correct model"() {
        when:"The show action is executed with a null domain"
            controller.show(null)
    }
}
```

```
    then:"A 404 error is returned"
      response.status == 404

    when:"A domain instance is passed to the show action"
      populateValidParams(params)
      def category = new Category(params)
      controller.show(category)

    then:"A model is populated containing the domain instance"
      model.categoryInstance == category
  }

  void "Test that the edit action returns the correct model"() {
    when:"The edit action is executed with a null domain"
      controller.edit(null)

    then:"A 404 error is returned"
      response.status == 404

    when:"A domain instance is passed to the edit action"
      populateValidParams(params)
      def category = new Category(params)
      controller.edit(category)

    then:"A model is populated containing the domain instance"
      model.categoryInstance == category
  }

  void "Test the update action performs an update on a valid domain
instance"() {
    when:"Update is called for a domain instance that doesn't exist"
      request.contentType = FORM_CONTENT_TYPE
      request.method = 'PUT'
      controller.update(null)

    then:"A 404 error is returned"
      response.redirectedUrl == '/category/index'
      flash.message != null

    when:"An invalid domain instance is passed to the update action"
      response.reset()
      def category = new Category()
      category.validate()
      controller.update(category)

    then:"The edit view is rendered again with the invalid instance"
      view == 'edit'
      model.categoryInstance == category

    when:"A valid domain instance is passed to the update action"
      response.reset()
      populateValidParams(params)
      category = new Category(params).save(flush: true)
      controller.update(category)

    then:"A redirect is issues to the show action"
```

```

        response.redirectedUrl == "/category/show/$category.id"
        flash.message != null
    }

    void "Test that the delete action deletes an instance if it exists"()
    {
        when:"The delete action is called for a null instance"
            request.contentType = FORM_CONTENT_TYPE
            request.method = 'DELETE'
            controller.delete(null)

        then:"A 404 is returned"
            response.redirectedUrl == '/category/index'
            flash.message != null

        when:"A domain instance is created"
            response.reset()
            populateValidParams(params)
            def category = new Category(params).save(flush: true)

        then:"It exists"
            Category.count() == 1

        when:"The domain instance is passed to the delete action"
            controller.delete(category)

        then:"The instance is deleted"
            Category.count() == 0
            response.redirectedUrl == '/category/index'
            flash.message != null
    }
}

```

¿Qué podemos extraer de estos tests generados?

- Una serie de variables han sido automáticamente inyectadas en nuestro test como son *controller*, *model*, *view*, *response*, *request* y *flash*.
- Se utiliza la anotación `@Mock()` para mockear el comportamiento de la clase de dominio pasada por parámetro. De esta forma se simula su almacenamiento.
- Para invocar un método cualquiera del controlador simplemente debemos ejecutar *controller.method()*
- La variable *model* nos permitirá comprobar si el modelo que devolvemos es el esperado
- Las redirecciones se pueden comprobar con la variable *response.redirectedUrl*

## Tests con Spock sobre librerías de etiquetas, vistas y plantillas en Grails

Como siempre, cuando creábamos la librería de etiquetas *TodoTagLib.groovy*, se creaba automáticamente un esqueleto de test unitario:

```

package es.ua.expertojava.todo

import grails.test.mixin.TestFor
import spock.lang.Specification

/**

```

```

* See the API for {@link grails.test.mixin.web.GroovyPageUnitTestMixin}
for usage instructions
*/

```

```

@TestFor(ToDoTagLib)
class ToDoTagLibSpec extends Specification {

    def setup() {
    }

    def cleanup() {
    }

    void "test something"() {
    }
}

```

Mediante la anotación `@TestFor(ToDoTagLib)` la variable `tagLib` se inyecta automáticamente en nuestros tests. Si hacemos memoria, cuando hablábamos de las librerías de etiquetas poníamos como ejemplo

```

def includeJs = {attrs ->
    out << "<script src='scripts/${attrs['script']}.js'></script>"
}

```

Vamos a ver a continuación como podemos comprobar que esta etiqueta devuelve lo que pensamos.

```

void "La etiqueta includeJs devuelve una referencia a la librería
javascript pasada por parámetro"() {
    expect:
        applyTemplate('<todo:includeJs script="" />') == "<script
src='scripts/.js'></script>"
        applyTemplate('<todo:includeJs script="myfile" />') == "<script
src='scripts/myfile.js'></script>"
}

```

Además de testear las librerías de etiquetas, vamos a comprobar que las vistas y las plantillas devuelven lo que deben devolver. Si recordamos en la sesión de vistas y plantillas, creábamos una plantilla para pintar el pie de página. Vamos a poder comprobar que esta plantilla devuelve lo correcto mediante un test unitario:

```

void "El pie de página se renderiza correctamente"() {
    when:
        def result = render(template: '/common/footer')

    then:
        result == "<div class=\"footer\" role=\"contentinfo\">\n" +
            "    &copy; 2015 Experto en Desarrollo de Aplicaciones Web con
JavaEE y Javascript<br/>\n" +
            "    Aplicación Todo creada por Francisco José García Rico
(21.542.334F)\n" +
            "</div>"
}

```



## Tests con Spock sobre servicios en Grails

Tal y como sucede con el resto de artefactos de una aplicación en Grails, cuando creamos un servicio nuevo automáticamente se genera también un test unitario que nos permitirá testear los métodos del servicio.

```
package es.ua.expertojava.todo

import grails.test.mixin.TestFor
import spock.lang.Specification

/**
 * See the API for {@link grails.test.mixin.services.ServiceUnitTestMixin}
 * for usage instructions
 */
@TestFor(ToDoService)
class ToDoServiceSpec extends Specification {

    def setup() {
    }

    def cleanup() {
    }

    void "test something"() {
    }
}
```

En su momento creábamos un método llamado *getNextTodos()* que nos devolvía las tareas programadas para los próximos días. Para poder testear este método, vamos a necesitar crear una serie de instancias de la clase de dominio *ToDo*. Si recordamos el método en cuestión

```
def getNextTodos(Integer days, params) {
    Date now = new Date(System.currentTimeMillis())
    Date to = now + days
    ToDo.findAllByDateBetween(now, to, params)
}
```

vemos que se utiliza una llamada a un método dinámico de GORM, con lo que se se realizará una consulta a la base de datos. Pero, si recordamos la definición de tests unitarios que veíamos al inicio de esta sesión, se hacía mención a que los tests unitarios no tienen acceso a recursos externos tales como la base de datos, con lo que, ¿cómo vamos a poder solucionar este problema?

Podríamos optar por dos soluciones. Bien crear un test de integración para tener acceso a la base de datos o bien, *mockear* la creación de las clases de dominio necesarias. Nosotros optamos por la segunda solución, ya que en términos de velocidad de ejecución de los tests, los tests de integración son considerablemente más lentos que los tests unitarios.

Para *mockear* la creación de una serie de instancias de una determinada clase de dominio, vamos a utilizar el método llamado *mockDomain* al cual le podemos pasar que clase de dominio queremos mockear y cuales son esas instancias. Con esto podríamos tener un test como éste para comprobar el método *getNextTodos()*:

```

void "El método getNextTodos devuelve los siguientes todos de los días
pasado por parámetro"() {
    given:
        def todoDayBeforeYesterday = new Todo(title:"Todo day before
yesterday", date: new Date() - 2 )
        def todoYesterday = new Todo(title:"Todo yesterday", date: new
Date() - 1 )
        def todoToday = new Todo(title:"Todo today", date: new Date())
        def todoTomorrow = new Todo(title:"Todo tomorrow", date: new
Date() + 1 )
        def todoDayAfterTomorrow = new Todo(title:"Todo day after
tomorrow", date: new Date() + 2 )
        def todoDayAfterDayAfterTomorrow = new Todo(title:"Todo day after
tomorrow", date: new Date() + 3 )
    and:
        mockDomain(Todo, [todoDayBeforeYesterday, todoYesterday, todoToday,
todoTomorrow, todoDayAfterTomorrow, todoDayAfterDayAfterTomorrow])
    and:
        def nextTodos = service.getNextTodos(2, [:])
    expect:
        Todo.count() == 6
    and:
        nextTodos.containsAll([todoTomorrow, todoDayAfterTomorrow])
        nextTodos.size() == 2
    and:
        !nextTodos.contains(todoDayBeforeYesterday)
        !nextTodos.contains(todoToday)
        !nextTodos.contains(todoYesterday)
        !nextTodos.contains(todoDayAfterDayAfterTomorrow)
}

```

En el test, en primer lugar creamos todas las instancias de la clase de dominio *Todo* que vamos a utilizar para realizar las comprobaciones. La primera comprobación que realizamos es comprobar que el contador de instancias de la clase *Todo* es igual al número de elementos creados.

A continuación, ya comprobamos el método *getNextTodos()* para comprobar que sólo devuelve aquellas instancias creadas que coincidan con el criterio de que sean tareas a realizar en el futuro. Esto significa que sólo las instancias *todoTomorrow* y *todoDayAfterTomorrow* serán devueltas.

## Tests con Spock mockeando objetos

Los tests unitarios se pueden complicar mucho cuando entran en funcionamiento otros objetos en el método que queremos probar. Por ejemplo, en la sesión sobre vistas y etiquetas planteábamos un ejercicio para escribir una etiqueta que nos imprimiera por pantalla un icono diferente en función de si el valor pasado por parámetro era cierto o falso. Esa etiqueta se aconsejaba utilizar a su vez la etiqueta *asset.image()* procedente del plugin *asset pipeline*.

Si pensamos en que test podríamos utilizar para comprobar el funcionamiento de la etiqueta *printIconFromBoolean*, podríamos pensar algo así:

```

@Unroll
void "El método printIconFromBoolean devuelve una ruta a una imagen"() {

```

```

when:
  def output = applyTemplate('<todo:printIconFromBoolean
value="${value}" />', [value:value])
  then:
    output == expectedOutput
  where:
    value    |    expectedOutput
    true     |    "icontrue.png"
    false    |    "iconfalse.png"
}

```

Sin embargo, si ejecutamos este test, veremos como el test no se ejecuta correctamente debido a que no sabe como ejecutar un método llamado *assetPath*. ¿*assetPath*? ¿De dónde viene la llamada a ese método?

Si analizamos el código del método *image* de la librería de etiquetas *AssetsTagLib* veremos como se realiza una llamada al método *assetPath* que queda fuera del contexto de nuestro test unitario con lo que no podemos acceder directamente a él. Para solucionar este problema, debemos *mockear* la llamada que nuestra etiqueta *printIconFromBoolean* realiza al método *image()* de la librería de etiquetas del plugin *asset pipeline*. De esta forma, podemos aislar nuestra prueba unitaria de las llamadas realizadas por otros objetos colaboradores y nos centraremos en la lógica de nuestro método.

```

@Unroll
void "El método printIconFromBoolean devuelve una ruta a una imagen"() {
  given:
    def assetsTagLib = Mock(AssetsTagLib)
    tagLib.metaClass.asset = assetsTagLib
  when:
    def output = applyTemplate('<todo:printIconFromBoolean
value="${value}" />', [value:value])
  then:
    output == expectedOutput
  and:
    1 * assetsTagLib.image(_) >> { value ? "icontrue.png"
: "iconfalse.png" }
  where:
    value    |    expectedOutput
    true     |    "icontrue.png"
    false    |    "iconfalse.png"
}

```

En primer lugar, debemos mockear la librería *AssetsTagLib* para poder devolver lo que nosotros queramos y que no sea necesario invocar al método *assetPath*. Además, debemos indicar una referencia al *namespace* *asset* que nuestra librería *TodoTagLib* va a utilizar como colaborador. Esto lo hacemos con algo de metaprogramación

```
tagLib.metaClass.asset = assetTagLib
```

y básicamente estamos creando una referencia entre el objeto *asset* en el contexto de la librería de etiquetas *TodoTagLib* y nuestro objeto mockeado con la librería *AssetsTagLib*. De esta forma, lo único que nos queda por hacer es sobrecargar el método *image()*.

Para sobrecargar este método utilizamos una técnica que nos permite incluso saber cuantas veces un método ha sido invocado y con que parámetros. Esta técnica se conoce como *Tests basados en interacciones*.

Para ello lo primero que hemos hecho ha sido generar ese objeto mockeado con

```
def assetsTagLib = Mock(AssetsTagLib)
```

Por último, debemos indicar cuantas interacciones vamos a realizar con el método del objeto mockeado y cual va a ser su salida.

```
1 * assetsTagLib.image(_) >> { value ? "icontrue.png" : "iconfalse.png" }
```

Con esto estamos indicando que sólo se va a producir una llamada al método *image()* de la librería de etiquetas *AssetsTagLib* y que además, esta llamada se puede realizar con cualquier valor como primer (y único) parámetro (esto lo conseguimos utilizando el *placeholder* `_`). Si quisiéramos asegurarnos de que los parámetros pasados son los correctos podríamos tener algo así:

```
1 * assetsTagLib.image([src:expectedOutput]) >> { value ? "icontrue.png" : "iconfalse.png" }
```

## Algunas anotaciones interesantes

Spock presenta una serie de anotaciones que nos servirán de apoyo para realizar nuestros tests.

### Ignore

Esta anotación se puede utilizar a nivel de método o a nivel de clase y como su nombre indica sirve para indicar al proceso que ejecuta los tests que el método en cuestión o la clase no deben ser ejecutados.

```
@Ignore
def "my feature"() { ... }
```

```
@Ignore
class MySpec extends Specification { ... }
```

Incluso podemos pasar como parámetro a la anotación un motivo por el cual este test no va a ser ejecutado.

```
@Ignore(reason = "TODO")
def "my feature"() { ... }
```

### IgnoreRest

Esta anotación sólo se puede especificar a nivel de método y le indicará al proceso encargado de ejecutar los tests que sólo los métodos que proporcionen esta anotación deben ser ejecutados.

```
def "I'll be ignored"() { ... }

@IgnoreRest
def "I'll run"() { ... }

def "I'll also be ignored"() { ... }
```

## IgnoreIf

Esta anotación indica que el método anotado sólo se ejecutará si se cumplen una serie de condiciones.

```
@IgnoreIf({ System.getProperty("os.name").contains("windows") })
def "I'll run everywhere but on Windows"() { ... }
```

Para facilitar la comprensión de la condición existen una serie de propiedades disponibles dentro del closure:

- `sys` Un mapa con todas las propiedades del sistema
- `env` Un mapa con todas las variables de entorno
- `os` Información acerca del sistema operativo (ver `spock.util.environment.OperatingSystem`)
- `jvm` Información sobre la máquina virtual de Java (ver `spock.util.environment.Jvm`)

Con lo que el ejemplo anterior puede ser reescrito de la siguiente forma:

```
@IgnoreIf({ os.windows })
def "I'll run everywhere but on Windows"() { ... }
```

## Requires

Es la anotación contraria a *@IgnoreIf*. Los métodos anotados con esta anotación sólo se ejecutarán bajo ciertas condiciones.

```
@Requires({ os.windows })
def "I'll only run on Windows"() { ... }
```

## Stepwise

Es conveniente que los tests no dependan unos de otros para su correcta ejecución, pero en ocasiones por rapidez, necesitaremos ejecutar los tests en un determinado orden. Esto lo podemos conseguir con la anotación *@Stepwise* que se utiliza a nivel de clase.

Imagina por ejemplo un test funcional con el que pretendemos comprobar que las 4 operaciones básicas sobre las tareas (creación, eliminación, actualización y borrado) se ejecutan correctamente. Lo ideal sería crear 4 tests distintos, uno para cada operación, pero, ¿qué pasaría si pretendemos eliminar una tarea que todavía ni siquiera ha sido creada? En este ejemplo, es necesario que todos los tests se ejecuten en un determinado orden.

## Timeout

La anotación `@Timeout` se puede aplicar a nivel de clase o a nivel de método y nos permitirá especificar un tiempo máximo de ejecución de un test o de una clase de tests. La unidad por defecto será en segundos, pero vamos a poder modificar esta unidad.

```
@Timeout(5)
def "I fail if I run for more than five seconds"() { ... }

@Timeout(value = 100, unit = TimeUnit.MILLISECONDS)
def "I better be quick" { ... }
```

Si utilizamos la anotación a nivel de clase y a nivel de método, el método siempre sobrescribirá a lo especificado a nivel de clase.

```
@Timeout(10)
class TimedSpec extends Specification {
    def "I fail after ten seconds"() { ... }
    def "Me too"() { ... }

    @Timeout(value = 250, unit = MILLISECONDS)
    def "I fail much faster"() { ... }
}
```

## ConfineMetaClassChanges

Utilizar metaprogramación puede ser muy beneficioso en nuestros tests pero al mismo tiempo, puede llegar a ser algo peligroso, ya que necesitamos asegurarnos que las clases que han sido metaprogramadas no conservan esa metaprogramación más allá de donde sea imprescindible.

Para ello, bien asegurar de dejar la metaclass sin ninguna modificación con

```
Class.metaClass = null
```

o bien, en los casos de los tests con Spock podemos utilizar la anotación `@ConfineMetaClassChanges`.

```
@Stepwise
class FooSpec extends Specification {
    @ConfineMetaClassChanges
    def "I run first"() {
        when:
            String.metaClass.someMethod = { delegate }

        then:
            String.metaClass.hasMetaMethod('someMethod')
    }

    def "I run second"() {
        when:
```

```
    "Foo".someMethod()

    then:
    thrown(MissingMethodException)
  }
}
```

---

## Title y Narrative

Podemos especificar lenguaje natural al nombre de nuestras clases con las anotaciones *@Title*

---

```
@Title("This is easy to read")
class ThisIsHarderToReadSpec extends Specification {
    ...
}
```

---

y *@Narrative*.

---

```
@Narrative("""
As a user
I want foo
So that bar
""")
class GiveTheUserFooSpec() { ... }
```

---

## 6.4. Ejercicios

### Testeando nuestra propia restricción (0.25 puntos)

Realiza los tests necesarios para comprobar el correcto funcionamiento de la restricción que creaste en sesiones anteriores para que la fecha de recordatorio nunca pueda ser posterior a la fecha de la propia tarea.

Este test o tests los vamos a escribir en la clase *TodoSpec.groovy*.

### Testeando la clase *TodoController* (0.50 puntos)

Escribe las modificaciones necesarias a la clase *TodoControllerSpec* para pasar los tests unitarios para la parte de scaffolding. Recuerda que estamos utilizando el servicio *todoService* en el controlador con lo que tendrás que inyectar el servicio en concreto en el controlador.

Además, realiza los tests unitarios necesarios para comprobar también el funcionamiento de los métodos *listNextTodos(Integer days)* y *showListByCategory()* (la vista que nos permite que categorías queremos filtrar en el listado de las tareas).

Estos tests los vamos a escribir en la clase *TodoControllerSpec.groovy*.

### Testeando la clase *TodoService* (0.50 puntos)

Para terminar con los tests, vamos a implementar un par de tests para testear el servicio *TodoService*. Estos dos métodos serán *countNextTodos()* y *saveTodo()* (que creamos en la sesión anterior).

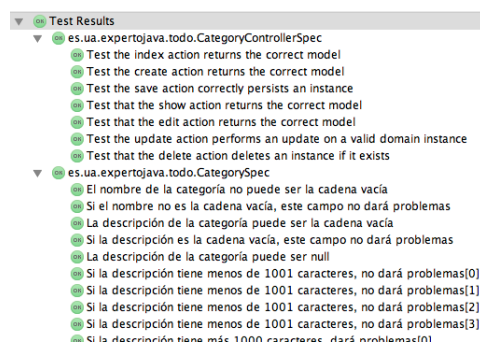
El método *saveTodo()* se encarga de almacenar la nueva instancia pero además os recuerdo que debía almacenar la fecha de realización de la tarea si esta había sido realizada.

Estos tests los vamos a escribir en la clase *TodoServiceSpec.groovy*.



### Importante

Es absolutamente necesario que cuando presentéis el proyecto todos los tests se pasen correctamente. Así que antes de entregar el proyecto final, recuerda ejecutar el comando *grails test-app unit:* para comprobar que tus tests se ejecutan correctamente. Es decir, así debería terminar vuestro proyecto antes de ser entregado.





## 7. Seguridad con Spring Security plugin.

En esta séptima sesión del módulo de Groovy&Grails veremos como podemos utilizar un plugin de Grails que nos permitirá descargar toda la implementación de la seguridad de nuestra aplicación en Spring Security.

Mediante este plugin podremos asignar determinados roles a usuarios y posteriormente crear algunos permisos que podrán ser asignados tanto a usuarios como a roles.

### 7.1. Instalación del plugin

En la introducción a Grails decíamos que, aquello que no está implementado en el núcleo de Grails, es muy probable que lo podamos encontrar entre los plugins, realizados tanto por la comunidad de usuarios como por la gente de Spring Source.

En este caso, el plugin es mantenido directamente por la gente de Spring Source, algo que nos garantiza su durabilidad y mantenimiento. Este plugin se llama *spring-security-core* y podemos encontrar toda la información relativa al mismo en la dirección <http://grails.org/plugin/spring-security-core>.

Tal y como se explica en la página del plugin, éste tiene una serie de plugins adicionales que permiten la autenticación mediante OpenID, LDAP o Kerberos (entre otras). Además, a esta lista se han añadido recientemente otros plugins que permiten la autenticación utilizando Facebook o Twitter.

El plugin de *spring-security-core* es uno de los mejor documentados de todos los desarrollados y podrás encontrar esta documentación en la dirección <http://grails-plugins.github.io/grails-spring-security-core/guide/index.html> y su autor, Burt Beckwith ([@burtbeckwith](https://twitter.com/burtbeckwith)<sup>19</sup>) uno de los *commiters* más activos de la comunidad.

En Grails, para instalar cualquier plugin tenemos el comando *grails install-plugin* con lo que para instalar el plugin de *spring-security-core* únicamente deberíamos ejecutar el comando *grails install-plugin spring-security-core*. Pero además, podemos editar el archivo *BuildConfig.groovy* y añadir una nueva línea en la parte dedicada a los plugins:

```
.....  
plugins {  
    ...  
    compile "spring-security-core:2.0-RC4"  
}
```

Nosotros vamos a optar por esta segunda opción puesto que quedarán contemplados todos los plugins que instalemos en nuestra aplicación en un único archivo que luego vamos a poder compartir con el resto de desarrolladores de la aplicación.

Durante la instalación del plugin se realizará automáticamente la descarga de todas las dependencias del plugin. En algunos casos, la instalación de plugins supone la creación de nuevos comandos en Grails y éste es el caso de *spring-security-core*, en el que, tal y como nos indican una vez finalizada la instalación nos indica que podemos ejecutar el comando *grails s2-quickstart* que nos servirá para inicializar el plugin y crear las clases de dominio necesarias.

El nuevo comando instalado *s2-quickstart* necesita de una serie de parámetros que serán los siguientes

<sup>19</sup> <https://www.twitter.com/burtbeckwith>

- *paquete* donde queremos crear las clases de dominio a generar
- *clase de dominio para los usuarios* que habitualmente se llamará User o Person
- *clase de dominio para los roles* que habitualmente se llamará Role o Authority
- *clase de dominio para el mapeo de peticiones* que habitualmente se llamará RequestMap. Este parámetro es opcional puesto que este mapeo podemos hacerlo directamente en el archivo de configuración *Config.groovy*

Nosotros ejecutaremos el comando `grails s2-quickstart es.ua.expertojava.todo Person Role RequestMap` que creará cuatro nuevas clases de dominio en nuestra aplicación en el paquete `es.ua.expertojava.todo` que son las siguientes:

- Person
- PersonRole
- Role
- RequestMap

## 7.2. Configuración del plugin

Toda la configuración del plugin se añade automáticamente en el archivo de configuración *Config.groovy* y básicamente lo que se indica en esta configuración es el nombre de cada una de las clases necesarias para el plugin así como el tipo de seguridad implementada.

```
// Added by the Spring Security Core plugin:
grails.plugin.springsecurity.userLookup.userDomainClassName
= 'es.ua.expertojava.todo.Person'
grails.plugin.springsecurity.userLookup.authorityJoinClassName
= 'es.ua.expertojava.todo.PersonRole'
grails.plugin.springsecurity.authority.className
= 'es.ua.expertojava.todo.Role'
grails.plugin.springsecurity.requestMap.className
= 'es.ua.expertojava.todo.RequestMap'
grails.plugin.springsecurity.securityConfigType = 'Requestmap'
grails.plugin.springsecurity.controllerAnnotations.staticRules = [
  '/': ['permitAll'],
  '/index': ['permitAll'],
  '/index.gsp': ['permitAll'],
  '/assets/**': ['permitAll'],
  '/**/js/**': ['permitAll'],
  '/**/css/**': ['permitAll'],
  '/**/images/**': ['permitAll'],
  '/**/favicon.ico': ['permitAll']
]
```

Como te habrás dado cuenta, todos los parámetros de configuración empiezan por `grails.plugin.springsecurity`. De esta forma podemos especificar por ejemplo que tipo de algoritmo queremos para encriptar la contraseña de los usuarios de la siguiente forma

```
grails.plugin.springsecurity.password.algorithm='SHA-512'
```

## 7.3. Clases de dominio

Como comentábamos anteriormente, cuatro son las clases que se han creado al ejecutar el comando `grails s2-quickstart`, tres obligatorias (Person, Role y PersonRole) y una opcional (RequestMap). La clase PersonRole es una clase que relaciona que implementa la relación muchos-a-muchos entre la clase Person y la clase Role. Veamos cada una de estas clases en detalle:

### Clase de dominio Person

La clase de dominio Person es la clase encargada de almacenar los datos de todos los usuarios de la aplicación. Estos datos son los típicos que desearíamos almacenar de un usuario como son el nombre de usuario y la contraseña. Además, también tiene una serie de propiedades que nos indicará si el usuario está activo o no, si la cuenta ha expirado, si la cuenta está bloqueada o si la contraseña ha expirado.

```
package es.ua.expertojava.todo

class Person {

    transient springSecurityService

    String username
    String password
    boolean enabled = true
    boolean accountExpired
    boolean accountLocked
    boolean passwordExpired

    static transients = ['springSecurityService']

    static constraints = {
        username blank: false, unique: true
        password blank: false
    }

    static mapping = {
        password column: '`password`'
    }

    Set<Role> getAuthorities() {
        PersonRole.findAllByPerson(this).collect { it.role }
    }

    def beforeInsert() {
        encodePassword()
    }

    def beforeUpdate() {
        if (isDirty('password')) {
            encodePassword()
        }
    }

    protected void encodePassword() {
```

```

    password = springSecurityService?.passwordEncoder ?
    springSecurityService.encodePassword(password) : password
  }
}

```

Además, también se implementa un método llamado *getAuthorities()* que nos devolverá todos los roles que el usuario tenga asignados. Este método podíamos decir que es análogo a la definición *static hasMany = [authorities: Authority]* de la forma tradicional.

Esta clase podrá ser modificada a nuestro gusto para añadirle tantas propiedades como necesitemos, como por ejemplo, nombre, apellidos o email. Sin embargo, es más conveniente dejar esta clase tal cual está y extenderla utilizando una clase llamada por ejemplo *User*.

```

package es.ua.expertojava.todo

class User extends Person{
  String name
  String surnames
  String confirmPassword
  String email
  Date dateOfBirth
  String description

  static hasMany = [todos:Todo]

  static constraints = {
    name(blank:false)
    surnames(blank:false)
    confirmPassword(blank:false, password:true)
    email(blank:false, email:true)
    dateOfBirth(nullable:true, validator: {
      if (it?.compareTo(new Date()) < 0)
        return true
      return false
    })
    description(maxSize:1000, nullable:true)
  }

  static transients = ["confirmPassword"]

  String toString(){
    "@${username}"
  }
}

```

Date cuenta de la relación que hemos establecido entre los usuarios y las tareas con *static hasMany = [todos:Todo]* con lo que habrá también que modificar la clase de dominio *Todo* para establecer la otra parte de la relación.

```

package es.ua.expertojava.todo

class Todo {
  ...
}

```

```
User user
```

```
    ...  
}
```

---

Para facilitar la generación de usuarios desde la propia aplicación, vamos a generar las vistas y los controladores referentes a los usuarios con

---

```
grails generate-all es.ua.expertojava.todo.User
```

---

## Clase de dominio Role

El plugin `spring-security-core` necesita de una clase que implemente los diferentes roles que pueden coexistir en nuestra aplicación. Habitualmente, esta clase se encarga de restringir el acceso a determinadas URLs a aquellos usuarios que tengan los permisos adecuados. Un usuario puede tener varios roles asignados que indiquen diferentes niveles de acceso a la aplicación y por lo normal, cada usuario debería tener al menos un rol asignado.

También es posible que un usuario no tenga ningún rol asignado (aunque no es muy habitual). En estos casos, cuando un usuario sin ningún rol asignado se identifica en la aplicación, automáticamente se le asigna un rol *virtual* llamado `ROLE_NO_ROLES`. Este usuario, sólo podrá acceder a aquellos recursos que no estén *asegurados*.

La clase `Role` tiene una única propiedad llamada *authority* de tipo `String` que será el nombre dado al rol. Esta propiedad debe ser única, tal y como vemos en las restricciones de la clase.

---

```
package es.ua.expertojava.todo  
  
class Role {  
  
    String authority  
  
    static mapping = {  
        cache true  
    }  
  
    static constraints = {  
        authority blank: false, unique: true  
    }  
}
```

---

## Clase de dominio PersonRole

Para especificar la relación de muchos-a-muchos entre las clases de dominio `Person` y `Role` utilizamos una nueva clase llamada `PersonRole`. Esta clase tiene únicamente dos propiedades que serán las referidas a la clase de dominio `Person` y a la clase de dominio `Role`. Además, le especificaremos que la clave primaria de la clase será la composición de ambas propiedades y que no necesitamos almacenar la versión actual del objeto.

---

```
package es.ua.expertojava.todo  
  
import org.apache.commons.lang.builder.HashCodeBuilder
```

---

```
class PersonRole implements Serializable {

    private static final long serialVersionUID = 1

    Person person
    Role role

    boolean equals(other) {
        if (!(other instanceof PersonRole)) {
            return false
        }

        other.person?.id == person?.id &&
        other.role?.id == role?.id
    }

    int hashCode() {
        def builder = new HashCodeBuilder()
        if (person) builder.append(person.id)
        if (role) builder.append(role.id)
        builder.hashCode()
    }

    static PersonRole get(long personId, long roleId) {
        PersonRole.where {
            person == Person.load(personId) &&
            role == Role.load(roleId)
        }.get()
    }

    static boolean exists(long personId, long roleId) {
        PersonRole.where {
            person == Person.load(personId) &&
            role == Role.load(roleId)
        }.count() > 0
    }

    static PersonRole create(Person person, Role role, boolean flush = false)
    {
        def instance = new PersonRole(person: person, role: role)
        instance.save(flush: flush, insert: true)
        instance
    }

    static boolean remove(Person u, Role r, boolean flush = false) {
        if (u == null || r == null) return false

        int rowCount = PersonRole.where {
            person == Person.load(u.id) &&
            role == Role.load(r.id)
        }.deleteAll()

        if (flush) { PersonRole.withSession { it.flush() } }

        rowCount > 0
    }
}
```

```
static void removeAll(Person u, boolean flush = false) {
    if (u == null) return

    PersonRole.where {
        person == Person.load(u.id)
    }.deleteAll()

    if (flush) { PersonRole.withSession { it.flush() } }
}

static void removeAll(Role r, boolean flush = false) {
    if (r == null) return

    PersonRole.where {
        role == Role.load(r.id)
    }.deleteAll()

    if (flush) { PersonRole.withSession { it.flush() } }
}

static constraints = {
    role validator: { Role r, PersonRole ur ->
        if (ur.person == null) return
        boolean existing = false
        PersonRole.withNewSession {
            existing = PersonRole.exists(ur.person.id, r.id)
        }
        if (existing) {
            return 'userRole.exists'
        }
    }
}

static mapping = {
    id composite: ['role', 'person']
    version false
}
}
```

---

También se han creado una serie de métodos que facilitan la asignación de roles a usuario así como la revocación. Con esto, suponiendo que ya tenemos creados un rol y un usuario, podemos asignar o revocar un rol a un usuario de la siguiente forma:

---

```
User user = ...
Role role = ...
//Asignación de rol a un usuario
UserRole.create user, role
//Asignación de rol a un usuario indicándole el atributo flush
UserRole.create user, role, true

//Revocación de un rol a un usuario
User user = ...
Role role = ...
UserRole.remove user, role
//Revocación de un rol a un usuario indicándole el atributo flush
```

```
UserRole.remove user, role, true
```

---

## Clase de dominio *RequestMap*

La clase de dominio *RequestMap* es una clase opcional que nos permite registrar que peticiones queremos asegurar introduciéndolas en la base de datos en lugar de tener que especificarlo en el archivo de configuración *Config.groovy* o mediante anotaciones (lo veremos a continuación).

Esta opción nos permite configurar estas entradas en tiempo de ejecución y podremos añadir, modificar o eliminar entradas sin tener que reiniciar la aplicación. Esta es la clase generada:

---

```
package es.ua.expertojava.todo

import org.springframework.http.HttpMethod

class RequestMap {

    String url
    String configAttribute
    HttpMethod httpMethod

    static mapping = {
        cache true
    }

    static constraints = {
        url blank: false, unique: 'httpMethod'
        configAttribute blank: false
        httpMethod nullable: true
    }
}
```

---

El campo *url* será la dirección accedida a asegurar mientras que el campo *configAttributeField* indicará los roles permitidos para acceder a esa dirección url. También podemos especificar que método HTTP vamos a permitir. Si no lo asignamos, significará que podemos utilizar cualquier método.

## 7.4. Asegurar peticiones

En esta sección vamos a ver las diversas formas que nos permite el plugin de Spring Security para configurar las diferentes peticiones que necesitemos asegurar en nuestra aplicación. Básicamente, tenemos cuatro formas de hacer esto, de las cuales debemos elegir sólo una para especificar que direcciones queremos asegurar en nuestra aplicación:

- Anotaciones
- Archivo de configuración *Config.groovy*
- Instancias de la clase *RequestMap*
- Expresiones específicas

Pero antes de ver cada una de estas formas, veamos unos cuantos aspectos interesantes. Habitualmente, las aplicaciones web son públicas con algunas páginas privadas que son las que aseguraremos. Sin embargo, si la mayor parte de nuestra aplicación es privada,



podemos usar la aproximación pesimista para denegar el acceso a todas las urls que no tengan una regla configurada. Esto lo podemos hacer especificando en el archivo de configuración *Config.groovy* el siguiente valor:

```
grails.plugin.springsecurity.rejectIfNoRule = true
```

De esta forma, cualquier url que no tenga una regla especificada, será denegada para cualquier usuario.

Cualquier dirección que necesitemos asegurar debe tener especificada una regla. Por ejemplo, podremos proteger la url */todo/tag/\*\** para que sólo los usuarios con el rol *ROLE\_ADMIN* puedan crear etiquetas. Pero además, estas entradas podemos combinarlas con una serie de tokens predefinidos:

- *IS\_AUTHENTICATED\_ANONYMOUSLY*, indica que cualquier usuario podría acceder a esa url.
- *IS\_AUTHENTICATED\_REMEMBERED*, indica que el usuario se ha autenticado en la aplicación con la opción de *remember me*.
- *IS\_AUTHENTICATED\_FULLY*, indica que el usuario se ha identificado completando el formulario de autenticación.

## Anotaciones

Si queremos utilizar este método en nuestra aplicación, debemos especificar el siguiente parámetro en el archivo de configuración *Config.groovy*

```
grails.plugin.springsecurity.securityConfigType = "Annotation"
```

Podemos especificar anotaciones tanto a nivel de clase como a nivel de método. En caso de que se especifiquen reglas tanto a nivel de clase como a nivel de método, serán estas últimas las que se apliquen.

En el siguiente ejemplo especificamos la seguridad a nivel de método:

```
package com.mycompany.myapp
import grails.plugin.springsecurity.annotation.Secured

class SecureAnnotatedController {

    @Secured(['ROLE_ADMIN'])
    def index() {
        render 'you have ROLE_ADMIN'
    }

    @Secured(['ROLE_ADMIN', 'ROLE_SUPERUSER'])
    def adminEither() {
        render 'you have ROLE_ADMIN or SUPERUSER'
    }

    def anybody() {
        render 'anyone can see this'
    }
}
```

```
}

```

Pero también podemos especificarlo a nivel de clase:

```
package com.mycompany.myapp
import grails.plugin.springsecurity.annotation.Secured

@Secured(['ROLE_ADMIN'])
class SecureClassAnnotatedController {

    def index() {
        render 'index: you have ROLE_ADMIN'
    }

    def otherAction() {
        render 'otherAction: you have ROLE_ADMIN'
    }

    @Secured(['ROLE_SUPERUSER'])
    def super() {
        render 'super: you have ROLE_SUPERUSER'
    }
}
```

En ocasiones, necesitaremos incluso definir reglas para proteger el acceso a determinados directorios, como por ejemplo, los directorios de imágenes o los archivos javascript. Para esto, debemos especificar reglas estáticas en el archivo de configuración *Config.groovy*:

```
grails.plugin.springsecurity.controllerAnnotations.staticRules = [
    '/js/admin/**': ['ROLE_ADMIN'],
    '/images/**': ['ROLE_ADMIN']
]
```

## Archivo de configuración *Config.groovy*

Para poder utilizar este método, debemos especificar en el archivo de configuración *Config.groovy* el siguiente parámetro:

```
grails.plugin.springsecurity.securityConfigType = "InterceptUrlMap"
```

Posteriormente, lo único que debemos hacer es especificar en el archivo *Config.groovy* todas las reglas que necesitemos de la siguiente forma:

```
grails.plugin.springsecurity.interceptUrlMap = [
    '/': ['permitAll'],
    '/index': ['permitAll'],
    '/index.gsp': ['permitAll'],
    '/assets/**': ['permitAll'],
    '/*/*js/**': ['permitAll'],
    '/*/*css/**': ['permitAll'],
    '/*/*images/**': ['permitAll'],

```

```

    '/**/favicon.ico':    ['permitAll'],
    '/login/**':         ['permitAll'],
    '/logout/**':        ['permitAll'],
    '/secure/**':        ['ROLE_ADMIN'],
    '/finance/**':       ['ROLE_FINANCE', 'isFullyAuthenticated()'],
  ]

```

En este listado, es importante que especifiquemos las reglas en el orden correcto y esto se consigue siendo más específico para pasar posteriormente a ser más general. El siguiente ejemplo sería incorrecto, puesto que estaríamos dando permisos a los usuarios de tipo `ROLE_ADMIN` a las direcciones `/secure/reallysecure/**`:

```

'/secure/**':           ['ROLE_ADMIN', 'ROLE_SUPERUSER'],
'/secure/reallysecure/**': ['ROLE_SUPERUSER']

```

Si un usuario de tipo `ROLE_ADMIN` intentase acceder a la dirección `/secure/reallysecure/list` se le daría acceso sin ningún problema puesto que casaría con la primera regla encontrada. La solución correcta sería la siguiente:

```

'/secure/reallysecure/**': ['ROLE_SUPERUSER']
'/secure/**':               ['ROLE_ADMIN', 'ROLE_SUPERUSER']

```

## Instancias de la clase RequestMap

Con esta forma, almacenaremos en la base de datos todas las urls que queremos proteger junto con la regla que debe cumplir, de forma muy similar a como hemos visto en el anterior apartado.

Lo primero que debemos hacer es especificar que utilizaremos este método en el archivo de configuración `Config.groovy`.

```

grails.plugin.springsecurity.securityConfigType = "Requestmap"

```

Posteriormente, podemos especificar estas urls junto con sus reglas en el archivo `BootStrap.groovy`.

```

for (String url in [
    '/', '/index', '/index.gsp', '/**/favicon.ico',
    '/assets/**', '/**/js/**', '/**/css/**', '/**/images/**',
    '/login', '/login.*', '/login/*',
    '/logout', '/logout.*', '/logout/*']) {
    new Requestmap(url: url, configAttribute: 'permitAll').save()
}
new Requestmap(url: '/profile/**',    configAttribute: 'ROLE_USER').save()
new Requestmap(url: '/admin/**',
    configAttribute: 'ROLE_ADMIN').save()
new Requestmap(url: '/admin/role/**',
    configAttribute: 'ROLE_SUPERVISOR').save()
new Requestmap(url: '/admin/user/**',
    configAttribute: 'ROLE_ADMIN,ROLE_SUPERVISOR').save()
new Requestmap(url: '/j_spring_security_switch_user',

```

```
configAttribute: 'ROLE_SWITCH_USER, isFullyAuthenticated()').save()
```

A diferencia del método anterior, en esta ocasión no es necesario que tengamos en cuenta el orden de las reglas, ya que será el propio plugin quien se encargue de calcular que regla es la más específica.

Por defecto, las reglas especificadas de esta forma son cacheadas en la aplicación, con lo que debemos tener en cuenta que cuando creemos, modifiquemos o eliminemos una regla, deberemos actualizar la caché para que se tengan en cuenta los cambios introducidos. Esto lo podemos hacer con el método `clearCachedRequestmaps()` del servicio `springSecurityService`, tal y como vemos en el siguiente ejemplo.

```
class RequestmapController {
    def springSecurityService

    ...

    def save() {
        def requestmapInstance = new Requestmap(params)
        if (!requestmapInstance.save(flush: true)) {
            render view: 'create', model: [requestmapInstance:
requestmapInstance]
            return
        }

        springSecurityService.clearCachedRequestmaps()

        flash.message = "${message(code: 'default.created.message',
args: [message(code: 'requestmap.label', default: 'Requestmap'),
requestmapInstance.id])}"
        redirect action: 'show', id: requestmapInstance.id
    }
}
```

## Expresiones específicas

En Spring Security podemos utilizar también *Spring Expression Language (SpEL)* para especificar las reglas de acceso a determinadas partes de nuestra aplicación. Mediante este lenguaje, vamos a poder ser algo más específicos. Veamos el mismo ejemplo utilizando cualquiera de los métodos que acabamos de ver. Empecemos por las anotaciones:

```
package com.yourcompany.yourapp
import grails.plugin.springsecurity.annotation.Secured

class SecureController {

    @Secured(["hasRole('ROLE_ADMIN')"])
    def someAction() {
        ...
    }

    @Secured(["authentication.name == 'ralph'"])
    def someOtherAction() {
```

```

    ...
  }
}

```

Sigamos con el método de almacenamiento de instancias de la clase `RequestMap`.

```

new Requestmap(url: "/secure/someAction",
               configAttribute: "hasRole('ROLE_ADMIN)').save()
new Requestmap(url: "/secure/someOtherAction",
               configAttribute: "authentication.name == 'ralph)').save()

```

Terminemos por la modificación del archivo de configuración `Config.groovy`:

```

grails.plugin.springsecurity.interceptUrlMap = [
  '/secure/someAction': ["hasRole('ROLE_ADMIN)"],
  '/secure/someOtherAction': ["authentication.name == 'ralph'"]
]

```

Aquí tienes una tabla con las expresiones que podemos utilizar:

Expresión	Descripción
<code>hasRole(role)</code>	Devuelve cierto si el usuario tiene asignado este rol
<code>hasAnyRole([role1,role2])</code>	Devuelve cierto si el usuario tiene asignado cualquiera de los roles pasados por parámetro
<code>principal</code>	Devuelve los datos del usuario registrado
<code>authentication</code>	Devuelve todos los datos relativos a la autenticación
<code>permitAll</code>	Siempre devuelve cierto
<code>denyAll</code>	Siempre devuelve falso
<code>isAnonymous()</code>	Devuelve cierto si el usuario actual es anónimo
<code>isRememberMe()</code>	Devuelve cierto si el usuario ha entrado con la opción <i>remember me</i>
<code>isAuthenticated()</code>	Devuelve cierto si el usuario no es anónimo
<code>isFullyAuthenticated()</code>	Devuelve cierto si el usuario no es anónimo o no ha entrado con la opción <i>remember me</i>
<code>request</code>	La petición HTTP, permitiendo expresiones como <code>isFullyAuthenticated() &amp;&amp; request.getMethod().equals('OPTIONS')</code>

La siguiente tabla muestra las equivalencias si lo hacemos de forma tradicional o mediante las expresiones *SpEL*.

Modo tradicional	Expresión
<code>ROLE_ADMIN</code>	<code>hasRole('ROLE_ADMIN')</code>
<code>ROLE_USER, ROLE_ADMIN</code>	<code>hasAnyRole('ROLE_USER','ROLE_ADMIN')</code>

Modo tradicional	Expresión
ROLE_ADMIN, IS_AUTHENTICATED_FULLY	hasRole('ROLE_ADMIN') and isFullyAuthenticated()
IS_AUTHENTICATED_ANONYMOUSLY	permitAll
IS_AUTHENTICATED_REMEMBERED	isAnonymous() or isRememberMe()
IS_AUTHENTICATED_FULLY	isFullyAuthenticated()

## Aspectos importantes

Como veremos más adelante, el plugin utiliza dos controladores para gestionar el proceso de *login* y *logout* del sistema. Estos controladores se llaman *LoginController* y *LogoutController*. Cualquier método de estos dos controladores debe ser accesible por cualquier usuario. Además, también debemos de dar acceso global a determinados recursos de nuestra aplicación.

Teniendo esto en cuenta, en función del método escogido para proteger nuestra aplicación, deberemos añadir una serie de reglas en nuestro archivo de configuración *Config.groovy*.

## Mediante anotaciones

```
grails.plugin.springsecurity.securityConfigType = 'Annotation'
grails.plugin.springsecurity.controllerAnnotations.staticRules = [
  '/': [ 'permitAll' ],
  '/index': [ 'permitAll' ],
  '/index.gsp': [ 'permitAll' ],
  '/assets/**': [ 'permitAll' ],
  '/**/js/**': [ 'permitAll' ],
  '/**/css/**': [ 'permitAll' ],
  '/**/images/**': [ 'permitAll' ],
  '/**/favicon.ico': [ 'permitAll' ],
  '/login/**': [ 'permitAll' ],
  '/logout/**': [ 'permitAll' ],
  '/dbconsole/**': [ 'permitAll' ]
]
```

## Mediante instancias de la clase de dominio RequestMap

```
grails.plugin.springsecurity.securityConfigType = 'Requestmap'

/* Esto debe ir en el archivo BootStrap.groovy */
for (String url in [
  '/', '/index', '/index.gsp', '/assets/**',
  '/**/js/**', '/**/css/**', '/**/images/**',
  '/**/favicon.ico', '/login/**', '/logout/**', '/dbconsole/**']) {
  new RequestMap(url: url, configAttribute: 'permitAll').save()
}
```

## Mediante reglas en el archivo Config.groovy con InterceptUrlMap

```
grails.plugin.springsecurity.securityConfigType = 'InterceptUrlMap'
grails.plugin.springsecurity.interceptUrlMap = [
```

```

    '/':                ['permitAll'],
    '/index':          ['permitAll'],
    '/index.gsp':      ['permitAll'],
    '/assets/**':      ['permitAll'],
    '/**/js/**':       ['permitAll'],
    '/**/css/**':      ['permitAll'],
    '/**/images/**':   ['permitAll'],
    '/**/favicon.ico': ['permitAll'],
    '/login/**':       ['permitAll'],
    '/logout/**':      ['permitAll'],
    '/dbconsole/**':   ['permitAll']
]

```

## 7.5. Controladores

Con el plugin de Grails tendremos acceso a un par de controladores, que aunque en un principio no vamos a tener que modificar, está bien que conozcamos de su existencia. Estos dos controladores son *LoginController* y *LogoutController* y este es su contenido:

```

package grails.plugin.springsecurity

import grails.converters.JSON

import javax.servlet.http.HttpServletResponse

import org.springframework.security.access.annotation.Secured
import org.springframework.security.authentication.AccountExpiredException
import org.springframework.security.authentication.CredentialsExpiredException
import org.springframework.security.authentication.DisabledException
import org.springframework.security.authentication.LockedException
import org.springframework.security.core.context.SecurityContextHolder as SCH
import org.springframework.security.web.WebAttributes

@Secured('permitAll')
class LoginController {

    /**
     * Dependency injection for the authenticationTrustResolver.
     */
    def authenticationTrustResolver

    /**
     * Dependency injection for the springSecurityService.
     */
    def springSecurityService

    /**
     * Default action; redirects to 'defaultTargetUrl' if logged in, /login/
     * auth otherwise.
     */
    def index() {
        if (springSecurityService.isLoggedIn()) {
            redirect uri:
                SpringSecurityUtils.securityConfig.successHandler.defaultTargetUrl

```

```
}
else {
  redirect action: 'auth', params: params
}
}

/**
 * Show the login page.
 */
def auth() {

  def config = SpringSecurityUtils.securityConfig

  if (springSecurityService.isLoggedIn()) {
    redirect uri: config.successHandler.defaultTargetUrl
    return
  }

  String view = 'auth'
  String postUrl
  = "${request.contextPath}${config.apf.filterProcessesUrl}"
  render view: view, model: [postUrl: postUrl,
                           rememberMeParameter:
config.rememberMe.parameter]
}

/**
 * The redirect action for Ajax requests.
 */
def authAjax() {
  response.setHeader 'Location',
SpringSecurityUtils.securityConfig.auth.ajaxLoginFormUrl
  response.sendError HttpServletResponse.SC_UNAUTHORIZED
}

/**
 * Show denied page.
 */
def denied() {
  if (springSecurityService.isLoggedIn() &&
      authenticationTrustResolver.isRememberMe(SCH.context?.authentication))
  {
    // have cookie but the page is guarded with IS_AUTHENTICATED_FULLY
    redirect action: 'full', params: params
  }
}

/**
 * Login page for users with a remember-me cookie but accessing a
IS_AUTHENTICATED_FULLY page.
 */
def full() {
  def config = SpringSecurityUtils.securityConfig
  render view: 'auth', params: params,
        model: [hasCookie:
authenticationTrustResolver.isRememberMe(SCH.context?.authentication),

postUrl: "${request.contextPath}${config.apf.filterProcessesUrl}"]
}
```



```
}

/**
 * Callback after a failed login. Redirects to the auth page with a
 warning message.
 */
def authfail() {

    String msg = ''
    def exception = session[WebAttributes.AUTHENTICATION_EXCEPTION]
    if (exception) {
        if (exception instanceof AccountExpiredException) {
            msg = g.message(code: "springSecurity.errors.login.expired")
        }
        else if (exception instanceof CredentialsExpiredException) {
            msg = g.message(code: "springSecurity.errors.login.passwordExpired")
        }
        else if (exception instanceof DisabledException) {
            msg = g.message(code: "springSecurity.errors.login.disabled")
        }
        else if (exception instanceof LockedException) {
            msg = g.message(code: "springSecurity.errors.login.locked")
        }
        else {
            msg = g.message(code: "springSecurity.errors.login.fail")
        }
    }

    if (springSecurityService.isAjax(request)) {
        render([error: msg] as JSON)
    }
    else {
        flash.message = msg
        redirect action: 'auth', params: params
    }
}

/**
 * The Ajax success redirect url.
 */
def ajaxSuccess() {
    render([success: true, username:
springSecurityService.authentication.name] as JSON)
}

/**
 * The Ajax denied redirect url.
 */
def ajaxDenied() {
    render([error: 'access denied'] as JSON)
}
}
```

---

```
package grails.plugin.springsecurity
```

```
import javax.servlet.http.HttpServletResponse
```

```

import org.springframework.security.access.annotation.Secured

@Secured('permitAll')
class LogoutController {

    /**
     * Index action. Redirects to the Spring security logout uri.
     */
    def index() {

        if (!request.post &&
            SpringSecurityUtils.getSecurityConfig().logout.postOnly) {
            response.sendError HttpServletResponse.SC_METHOD_NOT_ALLOWED // 405
            return
        }

        // TODO put any pre-logout code here
        redirect uri:
        SpringSecurityUtils.securityConfig.logout.filterProcessesUrl // '/'
        j_spring_security_logout'
    }
}

```

---

## 7.6. Librerías adicionales

El plugin de Spring Security viene además con una serie de librerías accesibles en forma de etiquetas o servicio. Veamos cada uno de ellos.

### Librería de etiquetas: SecurityTagLib

Con esta librería podremos insertar en nuestros archivos GSPs comprobaciones del tipo, ¿el usuario está identificado? o ¿tiene los permisos necesarios para ver esto?

#### ifLoggedIn

Muestra el contenido del cuerpo que pongamos dentro de esta etiqueta siempre que el usuario esté autenticado en el sistema.

---

```

<sec:ifLoggedIn>
  Welcome Back!
</sec:ifLoggedIn>

```

---

#### ifNotLoggedIn

Muestra el contenido del cuerpo que pongamos dentro de esta etiqueta siempre que el usuario no esté autenticado en el sistema.

---

```

<sec:ifNotLoggedIn>
  <g:link controller='login' action='auth'>
    Login
  </g:link>
</sec:ifNotLoggedIn>

```

---

### ifAllGranted

Muestra el contenido del cuerpo que pongamos dentro de esta etiqueta siempre que el usuario tenga todos los roles pasados por parámetro.

```
<sec:ifAllGranted roles="ROLE_ADMIN,ROLE_SUPERVISOR">secure stuff here</sec:ifAllGranted>
```

### ifAnyGranted

Muestra el contenido del cuerpo que pongamos dentro de esta etiqueta siempre que el usuario tenga al menos uno de los roles pasados por parámetro.

```
<sec:ifAnyGranted roles="ROLE_ADMIN,ROLE_SUPERVISOR">secure stuff here</sec:ifAnyGranted>
```

### ifNotGranted

Muestra el contenido del cuerpo que pongamos dentro de esta etiqueta siempre que el usuario no tenga asignado ninguno de los roles pasados por parámetro.

```
<sec:ifNotGranted roles="ROLE_USER">non-user stuff here</sec:ifNotGranted>
```

### loggedInUserInfo

Muestra el valor del campo especificado por parámetro del usuario identificado en el sistema.

```
<sec:loggedInUserInfo field="username"/>
```

### username

Muestra el nombre de usuario del usuario autenticado en el sistema

```
<sec:ifLoggedIn>
  Welcome Back <sec:username/>!
</sec:ifLoggedIn>
<sec:ifNotLoggedIn>
  <g:link controller='login' action='auth'>Login</g:link>
</sec:ifNotLoggedIn>
```

### access

Muestra el contenido del cuerpo que pongamos dentro de esta etiqueta siempre que la expresión se evalúe a cierto o el usuario tenga acceso a la url pasada por parámetro.

```
<sec:access expression="hasRole('ROLE_USER')">
  You're a user
</sec:access>
```

```
<sec:access url="/admin/user">
  <g:link controller='admin' action='user'>Manage Users</g:link>
</sec:access>
```

Incluso podemos especificar el controlador y la acción que queremos referenciar:

```
<sec:access controller='admin' action='user'>
  <g:link controller='admin' action='user'>
    Manage Users
  </g:link>
</sec:access>
```

### noAccess

Muestra el contenido del cuerpo que pongamos dentro de esta etiqueta siempre que la expresión se evalúe a falso o el usuario no tenga acceso a la url pasada por parámetro.

```
<sec:noAccess expression="hasRole('ROLE_USER')">
  You're not a user
</sec:noAccess>
```

### Servicio: SpringSecurityService

El servicio *grails.plugin.springsecurity.SpringSecurityService* ofrece algunos métodos que podremos utilizar en los controladores u otros servicios simplemente inyectándolos debidamente.

```
def springSecurityService
```

### getCurrentUser()

Obtiene una instancia de la clase de dominio correspondiente al usuario autenticado en el sistema.

```
class SomeController {
  def springSecurityService

  def someAction() {
    def user = springSecurityService.currentUser
    ...
  }
}
```

### isLoggedIn()

Comprueba si el usuario está autenticado en el sistema

```
class SomeController {
```

```

def springSecurityService

def someAction() {
    if (springSecurityService.isLoggedIn()) {
        ...
    }
    else {
        ...
    }
}
}

```

### getAuthentication()

Obtiene los datos del usuario autenticado en el sistema

```

class SomeController {
    def springSecurityService

    def someAction() {
        def auth = springSecurityService.authentication
        String username = auth.username
        def authorities = auth.authorities // a Collection of
        GrantedAuthority
        boolean authenticated = auth.authenticated
        ...
    }
}

```

### getPrincipal()

Obtiene los datos del usuario autenticado en el sistema

```

class SomeController {
    def springSecurityService

    def someAction() {
        def principal = springSecurityService.principal
        String username = principal.username
        def authorities = principal.authorities // a Collection of
        GrantedAuthority
        boolean enabled = principal.enabled
        ...
    }
}

```

### encodePassword()

Codifica la contraseña a partir del esquema de encriptación configurado. Por defecto, se utiliza el método SHA-256, pero éste se puede modificar en el archivo de configuración *Config.groovy* especificando el parámetro *grails.plugin.springsecurity.password.algorithm*, pudiendo utilizar cualquiera de los valores comentados en esta página de Java: <http://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#Architecture>.

---

```
class PersonController {
  def springSecurityService

  def updateAction() {
    def person = Person.get(params.id)

    params.salt = person.salt
    if (person.password != params.password) {
      params.password = springSecurityService.encodePassword(password,
salt)
      def salt = ... // e.g. randomly generated using some utility method
      params.salt = salt
    }
    person.properties = params
    if (!person.save(flush: true)) {
      render view: 'edit', model: [person: person]
      return
    }
    redirect action: 'show', id: person.id
  }
}
```

---

### updateRole()

Actualiza un rol, y en caso de que estemos utilizando el método *RequestMap*, también actualizará el nombre del rol en todas las instancias de la base de datos.

---

```
class RoleController {
  def springSecurityService

  def update() {
    def roleInstance = Role.get(params.id)
    if (!springSecurityService.updateRole(roleInstance, params)) {
      render view: 'edit', model: [roleInstance: roleInstance]
      return
    }

    flash.message = "The role was updated"
    redirect action: show, id: roleInstance.id
  }
}
```

---

### deleteRole()

Elimina un rol, y en caso de que estemos utilizando el método *Requestmap*, también lo eliminará de aquellas instancias donde estuviera especificado.

---

```
class RoleController {
  def springSecurityService

  def delete() {
    def roleInstance = Role.get(params.id)
    try {
```

```

        springSecurityService.deleteRole (roleInstance)
        flash.message = "The role was deleted"
        redirect action: list
    }
    catch (DataIntegrityViolationException e) {
        flash.message = "Unable to delete the role"
        redirect action: show, id: params.id
    }
}
}
}

```

### clearCachedRequestmaps()

Este método fuerza una actualización de toda la caché referente al sistema de autenticación de nuestra aplicación. Siempre que creamos, modifiquemos o eliminemos una instancia de la clase *RequestMap* debemos invocar este método. En los métodos *updateRole()* y *deleteRole()* ya se llama automáticamente al método *clearCachedRequestmaps()* para que nosotros no tengamos que preocuparnos.

```

class RequestmapController {
    def springSecurityService

    def save() {
        def requestmapInstance = new Requestmap(params)
        if (!requestmapInstance.save(flush: true)) {
            render view: 'create', model: [requestmapInstance:
requestmapInstance]
            return
        }

        springSecurityService.clearCachedRequestmaps()
        flash.message = "Requestmap created"
        redirect action: show, id: requestmapInstance.id
    }
}
}

```

### reauthenticate()

Este método renueva los datos de la autenticación del usuario. Esto se debe hacer después de que se actualicen los roles de un usuario.

```

class UserController {
    def springSecurityService

    def update() {
        def userInstance = User.get(params.id)

        params.salt = userInstance.salt
        if (params.password) {
            params.password =
springSecurityService.encodePassword(params.password, salt)
            def salt = ... // e.g. randomly generated using some utility method
            params.salt = salt
        }
    }
}

```

```

userInstance.properties = params
if (!userInstance.save(flush: true)) {
    render view: 'edit', model: [userInstance: userInstance]
    return
}

if (springSecurityService.loggedIn &&
    springSecurityService.principal.username ==
userInstance.username) {
    springSecurityService.reauthenticate userInstance.username
}

flash.message = "The user was updated"
redirect action: show, id: userInstance.id
}
}

```

## 7.7. Otros aspectos interesantes de Spring Security

El plugin de Spring Security es sin duda uno de los más completos de Grails y en la documentación oficial encontrarás otros detalles del mismo. Sin embargo, aquí vamos a resaltar un par de ellos como son las restricciones por IP o la internacionalización del plugin.

### Restricciones por IP

En ocasiones nos puede interesar proporcionar una serie de restricciones por IP de tal forma que únicamente las peticiones que vengan de esta IP podrán acceder al recurso. Esto lo debemos hacer en el archivo *Config.groovy* y este sería un ejemplo:

```

grails.plugin.springsecurity.ipRestrictions = [
    '/pattern1/**': '123.234.345.456',
    '/pattern2/**': '10.0.0.0/8',
    '/pattern3/**': ['10.10.200.42', '10.10.200.63']
]

```

### Internacionalización

El plugin de Spring Security viene traducido al inglés y al francés (entre otros idiomas), así que si necesitamos traducirlo al castellano por ejemplo, debemos crear las siguientes propiedades en el archivo de literales correspondiente:

Propiedad	Valor por defecto en inglés
springSecurity.errors.login.expired	Sorry, your account has expired
springSecurity.errors.login.passwordExpired	Sorry, your password has expired.
springSecurity.errors.login.disabled	Sorry, your password is disabled.
springSecurity.errors.login.locked	Sorry, your account is locked.
springSecurity.errors.login.fail	Sorry, we were not able to find a user with that username and password.
springSecurity.login.title	Login
springSecurity.login.header	Please Login..



<b>Propiedad</b>	<b>Valor por defecto en inglés</b>
springSecurity.login.button	Login
springSecurity.login.username.label	Username
springSecurity.login.password.label	Password
springSecurity.login.remember.me.label	Remember me
springSecurity.denied.title	Denied
springSecurity.denied.message	Sorry, you're not authorized to view this page.

## 7.8. Ejercicios

### Roles y usuarios (0.50 puntos)

En nuestra aplicación vamos a introducir dos tipos de usuarios: los administradores y los usuarios básicos. Los administradores serán capaces de crear nuevos usuarios, etiquetas y categorías mientras que los usuarios básicos simplemente podrán gestionar sus propias tareas.

Para ello introduce en el *Bootstrap.groovy* la creación de:

- Dos roles (ROLE\_ADMIN, ROLE\_BASIC)
- Un usuario administrador con nombre de usuario y contraseña "admin"
- Un par de usuarios básicos con nombres de usuario y contraseñas "usuario1" y "usuario2"
- Asigna varias tareas a ambos usuarios básicos

### Modificar plantillas y vistas (0.25 puntos)

En su momento creábamos una plantilla llamada *header.gsp* que se renderiza en la parte superior de la aplicación y que básicamente *pinta* un enlace para que el usuario se pueda identificar en el sistema.

En este ejercicio vamos a modificar esta plantilla para especificar los enlaces correctos */login/auth* para identificarse en el sistema y */logout/index* para abandonar el mismo. Una cosa que debes tener en cuenta es que para abandonar el sistema, el plugin Spring Security sólo acepta el método POST, con lo que tendrás que crear un botón dentro de un formulario para enviar esta petición utilizando el método adecuado. No olvides también imprimir en la cabecera el nombre de usuario de la persona identificada en el sistema.

Por otro lado, vamos a modificar la pantalla inicial de la aplicación para mostrar el contenido de la misma en función del tipo de usuario identificado, es decir, los administradores verán varios enlaces para mantener usuarios, etiquetas y categorías mientras que los usuarios básicos sólo podrán mantener sus tareas.

### Diferente rol, diferentes reglas (0.25 punto)

Por último, ahora que la aplicación soporta varios roles, deberemos distinguir que rol tiene acceso a que partes de la aplicación. Como comentábamos en el primer ejercicio, los administradores serán los encargados de gestionar usuarios, etiquetas y categorías, mientras que los usuarios básicos sólo podrán acceder a sus tareas.

Tendremos que realizar una serie de cambios en nuestra aplicación para que al crear una tarea, ésta sea asignada automáticamente al usuario identificado en el sistema. Deberemos proceder de la misma forma con el listado de las tareas para que sólo se muestren al usuario sus propias tareas.

Además, deberemos eliminar cualquier enlace presente en la aplicación que enlace a alguna parte que el usuario no deba tener acceso, como por ejemplo, la posibilidad de crear etiquetas desde la página de creación de tareas.

### Modificar los tests unitarios (0.25 puntos)

Si volvemos a ejecutar los tests unitarios, nos daremos cuenta de que acabamos de romper aquellos que creaban tareas y ésto ha sido provocado por la introducción de la relación entre las tareas y los usuarios.

Soluciona estos problemas en los tests *TodoSpec.groovy*, *TodoServiceSpec.groovy* y *TodoControllerSpec.groovy*.

## 8. Configuración de aplicaciones. Plugins interesantes

En esta última sesión de Grails analizaremos los diferentes aspectos de configuración de nuestra aplicación y posteriormente instalaremos algunos plugins interesantes desarrollados por la comunidad de usuarios de Grails.

### 8.1. Configuración de aplicaciones

#### El archivo `Config.groovy`

El archivo `grails-app/conf/Config.groovy` contiene los parámetros de configuración general de nuestra aplicación. En este archivo se pueden declarar variables que estarán disponibles en cualquier artefacto de nuestra aplicación a través del objeto global `grailsApplication.config`. Por ejemplo si definimos la siguiente variable `es.ua.expertojava.todo.miParametro = "dato"`, ésta va a ser accesible desde cualquier controlador, vista, servicio, etc. mediante la expresión `grailsApplication.config.es.ua.expertojava.todo.miParametro`.

Además de poder declarar nuevas variables globales, el archivo `Config.groovy` utiliza una serie de variables definidas que tienen el siguiente significado:

- `grails.config.locations`: ubicaciones donde encontrar otros archivos de configuración que se fundirán con el principal `Config.groovy`. Esta variable está comentada por defecto con lo que sólo se tendrá en cuenta el archivo `Config.groovy`.
- `grails.project.groupId`: nombre por defecto del paquete en el que se crean todos los artefactos de la aplicación. Por defecto tenemos que este paquete coincidirá con el nombre de la aplicación. Lo podríamos modificar por ejemplo a "es.ua.expertojava.todo" para no ahorrarnos tener que especificar el paquete cada vez que creamos un nuevo artefacto.
- `grails.views.default.codec`: especifica el formato por defecto de nuestras páginas GSPs. Puede tomar el valor 'none', 'html', o 'base64'. Por defecto se utiliza el valor html.
- `grails.controllers.defaultScope`: el `scope` por defecto de los controladores, en principio `singleton`.
- `grails.views.gsp`: un mapa de configuración de como se van a renderizar nuestras páginas GSP.
- `grails.converters.encoding`: codificación de los convertidores
- `grails.scaffolding.templates.domainSuffix`: el sufijo añadido a las instancias de nuestras clases de dominio en el plugin `scaffolding`.
- `grails.mime.types`: indica un mapa con los posibles tipos mime soportados en nuestra aplicación

#### Sistema de logs

Grails utiliza la librería `log4j` para implementar todo el sistema de logs. Podemos configurar estos logs de nuestras aplicaciones en el propio archivo `Config.groovy`. Aquí encontraremos una variable llamada `log4j` que ya viene con algunas reglas.

```
log4j.main = {
    error 'org.codehaus.groovy.grails.web.servlet', // controllers
         'org.codehaus.groovy.grails.web.pages', // GSP
         'org.codehaus.groovy.grails.web.sitemesh', // layouts
         'org.codehaus.groovy.grails.web.mapping.filter', // URL mapping
         'org.codehaus.groovy.grails.web.mapping', // URL mapping
```

```
'org.codehaus.groovy.grails.commons', // core / classloading
'org.codehaus.groovy.grails.plugins', // plugins
'org.codehaus.groovy.grails.orm.hibernate', // hibernate
integration
    'org.springframework',
    'org.hibernate',
    'net.sf.ehcache.hibernate'
}
```

En esta configuración se especifica que aquellos problemas surgidos a nivel *error* serán añadidos al archivo de logs. Habitualmente tenemos ocho niveles estándar de logs.

- off
- fatal
- error
- warn
- info
- debug
- trace
- all

Esto significa que si por ejemplo especificamos *warn 'org.example.domain'*, todos los problemas surgidos de tipo *warn*, *error* o *fatal* serán impresos en el archivo. Los otros niveles serán ignorados.

Lo habitual es que necesitemos generar logs en controladores, servicios y otros artefactos. Estos artefactos se encuentran en el paquete *grails.app.<tipo\_de\_artefacto>.<nombre\_de\_clase>*.

```
log4j.main = {
    // Establecemos el nivel info para todos los artefactos de la aplicación
    info "grails.app"

    // Especificamos el nivel debug para el controlador de la clase Tag
    debug "grails.app.controllers.es.ua.expertojava.todo.TagController"

    // Especificamos el nivel error para el servicio de la clase Todo
    error "grails.app.services.es.ua.expertojava.todo.TODOService"

    // Especificamos el nivel para todas las librerías de etiquetas
    info "grails.app.taglib"
}
```

Los tipos de artefactos pueden ser los siguientes:

- *conf*: para todo lo que esté en el directorio de configuración *grails-app/conf*
- *filters*: para los filtros
- *taglib*: para las librerías de etiquetas
- *services*: para los servicios
- *controllers*: para los controladores
- *domain*: para las clases de dominio

Vamos a implementar un pequeño ejemplo en nuestra aplicación. En primer lugar, vamos a crear un `appender`, que no es más que un objeto en el cual especificamos como y donde queremos escribir los mensajes de log. En nuestro caso lo haremos escribiendo en un archivo de logs, aunque también podríamos haber optado por enviarlo a un correo electrónico o bien almacenarlo en una base de datos.

```
log4j.main = {
  appenders {
    file name: 'file', file: 'mylog.log'
  }
}
```

A continuación, vamos a indicarle a nuestra aplicación que queremos utilizar este `appender` en el controlador de la clase `Category`.

```
trace
file: "grails.app.controllers.es.ua.expertojava.todo.CategoryController"
```

Por último, añadiremos una línea en el método `index()` del controlador `CategoryController` para escribir un mensaje de prueba.

```
def index() {
  log.trace("Método index del controlador CategoryController")
  params.max = Math.min(max ?: 10, 100)
  respond Category.list(params), model:[categoryInstanceCount:
  Category.count()]
}
```

Si todo ha ido bien, tendremos un archivo en la raíz de nuestra aplicación llamado `mylog.log` y que añadirá el mensaje que hemos creado cada vez que se muestre el listado de usuarios.

```
2015-03-28 12:05:08,549 ["http-bio-8080"-exec-1] TRACE
es.ua.expertojava.todo.CategoryController -
Método index del controlador CategoryController
```

## El archivo `BuildConfig.groovy`

Además del archivo `Config.groovy`, todos los aspectos relativos a la configuración de la generación del proyecto se encuentran en el archivo `conf/BuildConfig.groovy`. En este archivo podemos encontrar las siguientes variables definidas:

- `grails.servlet.version`: indicamos la especificación de Servlets. Por defecto utilizaremos la versión 3.0.
- `grails.project.class.dir`: especificamos el directorio donde se alojarán todos los archivos compilados del proyecto. Por defecto se alojarán en el directorio `target/classes`
- `grails.project.test.class.dir`: especificamos el directorio donde se compilarán los tests de nuestros proyectos. Por defecto se alojarán en el directorio `target/test-classes`
- `grails.project.test.reports.dir`: especificamos el directorio donde se alojarán los informes de las ejecuciones de los tests. Por defecto los tendremos en `target/test-reports`

- *grails.project.target.level*: especificamos la versión de la máquina virtual de java. Por defecto será la 1.6
- *grails.project.war.file*: especificamos el nombre y la ubicación del archivo war que se generará. Por defecto, se alojarán en el directorio *target* y se llamará con el nombre de la aplicación seguido de la versión, empezando por la 0.1
- *grails.project.fork*: parámetros de configuración de la máquina virtual de Java utilizada en cada entorno de ejecución.
- *grails.project.dependency.resolution*: especificamos como se resuelven las dependencias de los archivos JAR de nuestros proyectos.
- *grails.project.dependency.resolution.plugins*: un listado de los plugins utilizados en nuestra aplicación.

## El archivo DataSource.groovy

Muchas empresas disponen de varios entornos que las aplicaciones deben superar para finalmente pasar a disposición de los usuarios finales. Los entornos más habituales son el *entorno de desarrollo* que se refiere al entorno propio del desarrollador, con su propio servidor local instalado en su ordenador, el *entorno de tests* en el que otras personas se encargan de comprobar que la aplicación funciona tal y como se espera de ella y por último, el *entorno de producción*, que es donde aparecen en escena los usuarios finales, que son quienes realmente "probarán" el buen funcionamiento de la aplicación.

En cada uno de estos entornos, lo habitual es tener una configuración de base de datos diferente para cada uno, puesto que los requerimientos serán distintos. Por ejemplo, en un entorno de desarrollo posiblemente nos sea suficiente utilizar una base de datos en memoria como H2, pero este tipo de base de datos será insuficiente para los entornos de test y producción con lo que deberemos utilizar un servidor de base de datos como por ejemplo MySQL.

Como no podía ser menos, Grails lo tiene todo preparado para realizar esta diferenciación sin problemas. El archivo *grails-app/conf/DataSource.groovy* se encarga de todo mediante la creación por defecto de tres entornos de desarrollo: *desarrollo*, *test* y *producción*, tal y como puedes comprobar en el siguiente ejemplo.

```
.....  
dataSource {  
    pooled = true  
    jmxExport = true  
    driverClassName = "org.h2.Driver"  
    username = "sa"  
    password = ""  
}  
hibernate {  
    cache.use_second_level_cache = true  
    cache.use_query_cache = false  
    //    cache.region.factory_class =  
    'net.sf.ehcache.hibernate.EhCacheRegionFactory' // Hibernate 3  
    cache.region.factory_class  
    = 'org.hibernate.cache.ehcache.EhCacheRegionFactory' // Hibernate 4  
    singleSession = true // configure OSIV singleSession mode  
    flush.mode = 'manual' // OSIV session flush mode outside of  
    transactional context  
}
```

```

// environment specific settings
environments {
    development {
        dataSource {
            dbCreate = "create-drop" // one of 'create', 'create-drop',
            'update', 'validate', ''
            url
            = "jdbc:h2:mem:devDb;MVCC=TRUE;LOCK_TIMEOUT=10000;DB_CLOSE_ON_EXIT=FALSE"
        }
    }
    test {
        dataSource {
            dbCreate = "update"
            url
            = "jdbc:h2:mem:testDb;MVCC=TRUE;LOCK_TIMEOUT=10000;DB_CLOSE_ON_EXIT=FALSE"
        }
    }
    production {
        dataSource {
            dbCreate = "update"
            url
            = "jdbc:h2:prodDb;MVCC=TRUE;LOCK_TIMEOUT=10000;DB_CLOSE_ON_EXIT=FALSE"
            properties {
                // See http://grails.org/doc/latest/guide/
                conf.html#dataSource for documentation
                jmxEnabled = true
                initialSize = 5
                maxActive = 50
                minIdle = 5
                maxIdle = 25
                maxWait = 10000
                maxAge = 10 * 60000
                timeBetweenEvictionRunsMillis = 5000
                minEvictableIdleTimeMillis = 60000
                validationQuery = "SELECT 1"
                validationQueryTimeout = 3
                validationInterval = 15000
                testOnBorrow = true
                testWhileIdle = true
                testOnReturn = false
                jdbcInterceptors = "ConnectionState"
                defaultTransactionIsolation =
                java.sql.Connection.TRANSACTION_READ_COMMITTED
            }
        }
    }
}

```

En el primer bloque *dataSource* se definen una serie de valores genéricos, que posteriormente podremos sobrescribir en cada uno de los entornos. El bloque *hibernate* se refiere a parámetros de configuración del propio hibernate, mientras que el último bloque de *environments* es el que nos va a permitir diferenciar cada uno de los entornos.

Por defecto Grails crea los tres entornos de los que hablábamos anteriormente. Con el comando *grails run-app* que hemos utilizado hasta ahora, la aplicación se ejecuta en el entorno de desarrollo, que es el utilizado por defecto, pero si quisiéramos utilizar otro entorno podríamos ejecutar los comandos de la siguiente tabla:



Entorno	Comando
Desarrollo	<code>grails dev run-app</code> o <code>grails run-app</code>
Test	<code>grails test run-app</code>
Producción	<code>grails prod run-app</code>

En el fichero `DataSource.groovy` se puede comprobar las tres configuraciones para cada uno de los entornos. Para los entornos de desarrollo y test se va a utilizar una base de datos en memoria como H2, eso si, diferente para cada uno de ellos (`devDB` y `testDb`).

Para el entorno de producción deberíamos utilizar una base de datos más robusta y propia de entornos de producción como es MySQL. Para ello le especificamos la URI de la base de datos, así como el controlador, el nombre de usuario y la contraseña de acceso a la misma.

```

production {
    dataSource {
        dbCreate = "update"
        url = "jdbc:mysql://localhost:3306/todo?
useUnicode=true&characterEncoding=UTF-8"
        username = "userprod"
        password = "pwdprod"
        properties {
            // See http://grails.org/doc/latest/guide/
            conf.html#dataSource for documentation
            jmxEnabled = true
            initialSize = 5
            maxActive = 50
            minIdle = 5
            maxIdle = 25
            maxWait = 10000
            maxAge = 10 * 60000
            timeBetweenEvictionRunsMillis = 5000
            minEvictableIdleTimeMillis = 60000
            validationQuery = "SELECT 1"
            validationQueryTimeout = 3
            validationInterval = 15000
            testOnBorrow = true
            testWhileIdle = true
            testOnReturn = false
            jdbcInterceptors = "ConnectionState"
            defaultTransactionIsolation =
            java.sql.Connection.TRANSACTION_READ_COMMITTED
        }
    }
}

```

Además, deberíamos disponer del driver correspondiente de MySQL (<http://www.mysql.com/downloads/connector/j/>) y copiarlo en el directorio `lib` de la aplicación o bien especificarlo en las dependencias en el archivo `BuildConfig.groovy`.

```

grails.project.dependency.resolution = {
    ...

    dependencies {
        runtime 'mysql:mysql-connector-java:5.1.34'
    }
}

```

```

    }
    ....
}

```

Además, mediante el parámetro *dbCreate* podemos especificar la forma en la que Grails debe generar el esquema de la base de datos a partir de la definición de las clases de dominio. Este parámetro puede tomar tres valores diferentes:

- *create-drop*. El esquema de la base de datos se creará cada vez que arranquemos la aplicación en el entorno dado, y será destruido al interrumpir su ejecución.
- *create*. Crea el esquema de la base de datos, en caso de que el éste no exista, pero no modifica el esquema existente, aunque si borrará todos los datos almacenados en la misma.
- *update*. Crea la base de datos si no existe, y actualiza las tablas si detecta que se han añadido entidades nuevas o campos a los ya existentes. Estas modificaciones no incluirán la eliminación o modificación de nada en la base de datos, con lo que hay que tener cuidado con este tipo de cambios en las clases de dominio y que luego no se ven reflejados automáticamente en la base de datos.
- *validate*. Simplemente compara nuestro esquema actual con la base de datos especificada para enviarnos algún que otro aviso de diferencia.
- *cualquier otro valor*. Si especificamos cualquier otro valor, Grails no hará nada por nosotros. No debemos especificar ningún valor a la variable *dbCreate* si nos encargaremos de la base de datos nosotros mismos mediante una herramienta externa.

## El archivo *BootStrap.groovy*

A lo largo de las sesiones relativas a Grails, hemos utilizado el archivo de configuración *BootStrap.groovy* para insertar ciertos datos de ejemplo en nuestra aplicación que nos han servido para comprobar su funcionamiento.

El archivo define dos closures, *init* y *destroy*. El primero de ellos se ejecutará cada vez que ejecutemos la aplicación mediante el comando *grails run-app* en cualquiera de los entornos de ejecución recientemente comentados. Mientras que el closure *destroy* se ejecutará cuando paremos la ejecución de la aplicación.

Hasta el momento, cuando hemos insertado los datos en la base de datos, lo hacíamos independientemente del entorno de ejecución en el que estuviéramos, algo que no será lo habitual, puesto que para los entornos de test y producción, es más que probable que los datos ya estén almacenados en la base de datos o bien se inserten mediante una batería de pruebas en el caso del entorno de tests.

Teniendo en cuenta esto, necesitaremos diferenciar en cada momento el entorno de ejecución de nuestra aplicación para insertar datos o no. Para ello, Grails dispone de la clase *grails.util.Environment* y la variable *current* que nos indica cual es el entorno de ejecución actual. El siguiente código de ejemplo, muestra como realizar esta distinción a la hora de insertar datos en diferentes entornos.

```

import grails.util.Environment

class BootStrap {
    def init = { servletContext ->
        switch (Environment.current) {
            case Environment.DEVELOPMENT:

```

```

        configuracionDesarrollo()
        break
    case Environment.PRODUCTION:
        configuracionProduccion()
        break
    case Environment.TEST:
        configuracionTest()
        break
    }
}
def destroy = {
    switch (Environment.current) {
        case Environment.DEVELOPMENT:
            salirDesarrollo()
            break
        case Environment.PRODUCTION:
            salirProduccion()
            break
        case Environment.TEST:
            salirTest()
            break
    }
}
}
}

```

---

## El archivo `UrlMappings.groovy`

Otro de los archivos interesantes en la configuración de una aplicación Grails es *UrlMappings.groovy*. Gracias a este archivo vamos a poder definir nuevas relaciones entre las URLs y los controladores.

Por defecto, este archivo indica que el primer parámetro que sigue al nombre de la aplicación se refiere al controlador, el segundo a la acción y el tercero al identificador de la instancia de la clase de dominio que estamos tratando. Además, en reciente versiones de Grails se ha añadido un nuevo parámetro que indica el formato de la petición, por ejemplo, json o xml.

---

```

class UrlMappings {

    static mappings = {
        "$controller/$action?/$id?(.$format)?"{
            constraints {
                // apply constraints here
            }
        }

        "/"(view: "/index")
        "500"(view: '/error')
    }
}

```

Un ejemplo típico del uso del mapeo de URLs es modificar este comportamiento para permitir otras URLs más limpias. Por ejemplo, como administradores de la aplicación estaría bien poder ver las tareas de los usuarios a partir de su nombre de usuario, algo como <http://localhost:8080/todo/usuario1>.

Para ello, podemos introducir una nueva regla en el archivo *UrlMappings.groovy* para que la aplicación sepa como actuar cuando le llegue una petición con una URL del estilo de la comentada. La siguiente regla se encargaría de este redireccionamiento encubierto.

```
"/todos/$username"(controller:"todo",action:"showTodosByUser")
```

O bien podemos optar por una sintaxis diferente.

```
"/$username"{
    controller = "todo"
    action = "showtodosbyuser"
}
```

En primer lugar le especificamos la parte de la URL a partir del nombre de la aplicación que necesitamos que concuerde y después definimos que controlador y que acción se deben encargar de procesar esta petición. En ambas opciones tendremos acceso al valor del nombre de usuario a través de la variable *params.username*.

Posteriormente, deberíamos implementar un método en el controlador *TodoController* llamado *showTodosByUser()* que nos devolviera aquellos tareas que pertenezcan al usuario cuyo nombre de usuario coincida con el pasado en la url.

Otra posible utilidad del mapeo de URLs es la internacionalización de las URLs. Por ejemplo, en nuestra aplicación hemos definido las clases de dominio en inglés y por lo tanto las URLs se muestran también en inglés. Si deseamos que estas URLs se muestren también en castellano, podemos crear una serie de reglas en el archivo *UrlMappings.groovy*.

```
"/tarea/$action?/$id?(.$format)?"{
    controller = "tarea"
}

"/usuario/$action?/$id?(.$format)?"{
    controller = "user"
}
```

Las reglas de mapeo también permiten la introducción de restricciones que deben cumplir las partes de la URL. Por ejemplo, si quisiéramos obtener un listado con todos las tareas creadas un determinado día podríamos tener la siguiente url <http://localhost:8080/todo/2015/03/28>. Las restricciones que debe cumplir la URL son que el año debe ser una cifra de cuatro dígitos mientras que el mes debe estar compuesta por dos números y el día por otras dos. Para que la aplicación supiera que hacer con este tipo de direcciones debemos introducir la siguiente regla de mapeo.

```
"/$year/$month/$day" {
    controller = "todo"
    action = "showtodosbyday"
    constraints {
        year(matches:/\d{4}/)
        month(matches:/\d{2}/)
        day(matches:/\d{2}/)
    }
}
```

```
}

```

Otro aspecto interesante del mapeo de URLs puede ser la captura de los códigos de error que se producen en el acceso a una aplicación web, como por ejemplo el típico error 404 cuando la página solicitada no existe. En este tipo de casos, estaría bien modificar la típica pantalla de este tipo de errores, por otra en que se mostrará información sobre nuestra aplicación, como por ejemplo un mapa de todas las opciones de la aplicación.

Para controlar la información mostrada al producirse estos errores, podemos añadir lo siguiente en el archivo *UrlMapping.groovy*

```
"404"(view: '/error')
```

para que sea una página GSP quien se encargue de esta gestión o bien

```
"404"(controller: 'error', action: 'notFound')
```

para que sea un controlador quien haga este trabajo.

Por último, también es muy interesante ver como Grails es capaz de reescribir los enlaces que generamos en nuestra aplicación. Imagina por ejemplo que tenemos la siguiente regla en nuestro archivo *UrlMappings.groovy*

```
static mappings = {
    "$blog/$year?/$month?/$day?/$id?"(controller:"blog", action:"show")
}
```

Y que en una página GSP tenemos el siguiente código

```
<g:link controller="blog" action="show" params="[blog:'fred', year:2014]">
  My Blog
</g:link>

<g:link controller="blog" action="show" params="[blog:'fred', year:2014,
  month:10]">
  My Blog - October 2014 Posts
</g:link>
```

Lo que Grails convertiría en lo siguiente

```
<a href="/fred/2014">My Blog</a>
<a href="/fred/2014/10">My Blog - October 2014 Posts</a>
```

## 8.2. Empaquetamiento de aplicaciones

Al terminar una nueva funcionalidad de una aplicación o una nueva versión de la misma, necesitamos generar el paquete *WAR* correspondiente a la nueva versión para desplegarla en el servidor de destino.

La generación del archivo *WAR* se realiza simplemente ejecutando el comando *grails war*, el cual nos generará un archivo con extensión *.war* con el nombre de la aplicación, en nuestro caso *todo*, seguido de la versión de la aplicación. En nuestro caso, la primera vez que ejecutemos el comando *grails war* el fichero generado se llamará *todo-0.1.war*.

Esto sería lo más básico para generar el archivo *WAR* de la aplicación, sin embargo, lo habitual es hacer alguna cosa más, tal y como se muestra en el siguiente listado.

- Actualizar el código fuente del repositorio de control de versiones para asegurarse de que todas las partes del proyecto están actualizadas.
- Ejecutar los tests de integración, unitarios y funcionales que hayamos implementado para comprobar que todo funciona tal y como esperamos.
- Incrementar la variable *app.version* del archivo *application.properties* manualmente o bien mediante el comando *grails set-version 0.2*.
- Limpiar el proyecto de archivos temporales mediante el comando *grails clean*.
- Generar el archivo *WAR* indicándole el entorno donde queremos desplegar este *WAR*. Por ejemplo, el comando *grails prod war* crearía un archivo *WAR* para ser desplegado en nuestro entorno de producción.

Una aplicación Grails empaquetada como un archivo *WAR* puede ser desplegada en servidores de aplicaciones JAVA EE tales como JBoss (<http://www.jboss.org/>), GlassFish (<https://glassfish.dev.java.net/>), Apache Geronimo (<http://geronimo.apache.org>), BEA WebLogic (<http://www.bea.com>) o IBM WebSphere (<http://www.ibm.com/software/websphere>) o incluso en un contenedor web como Apache Tomcat (<http://tomcat.apache.org>) o Jetty (<http://www.mortbay.com>).

Cada uno de estos servidores o contenedores tendrán su propia especificación y forma de desplegar los archivos *WAR* generados. Unos mediante unos directorios especiales donde copiar los *WAR*, otros mediante una consola basada en web, otros por línea de comandos e incluso mediante tareas de tipo Ant. En la sección Deployment (<http://www.grails.org/Deployment>) de la web oficial de Grails puedes encontrar información sobre como desplegar los archivos *WAR* en varios servidores.

### 8.3. Otros comandos interesantes de Grails

Durante este módulo dedicado a Grails, han ido apareciendo varios de los comandos más habituales que utilizaremos cuando creemos aplicaciones con Grails. Sin embargo, los comandos vistos hasta ahora no son los únicos y es ahora cuando veremos algunos de ellos. Ten en cuenta también que cuando instalamos plugins, es posible que también se añadan nuevos comandos.

Si ejecutamos el comando *grails help* veremos un listado con todos los posibles comandos que tenemos disponibles en nuestra instalación de Grails. La siguiente tabla muestra los comandos más interesantes que no hemos visto hasta ahora.

Comando	Descripción
<i>grails bug-report</i>	Genera un archivo comprimido en ZIP con los archivos fuente de nuestro proyecto para el caso de que queramos informar de un bug

Comando	Descripción
<i>grails clean</i>	Limpia el directorio <i>tmp</i> de nuestra aplicación. Este comando puede ser combinado con otros comandos como por ejemplo <i>grails clean run-app</i>
<i>grails console</i>	Nos muestra la consola de Groovy que veíamos en la parte del módulo dedicada a este lenguaje de programación.
<i>grails doc</i>	Genera la documentación completa de nuestro proyecto.
<i>grails help</i>	Muestra un listado de comandos disponibles en Grails. Si le pasamos como parámetro uno de esos posibles comandos, nos mostrará información adicional sobre el comando dado. Por ejemplo <i>grails help doc</i>
<i>grails list-plugins</i>	Muestra un listado completo tanto de los plugins disponibles como de los ya instalados en la aplicación.
<i>grails plugin-info</i>	Muestra la información completa del plugin pasado como parámetro al comando.
<i>grails run-app -https</i>	Ejecuta el comando <i>grails run-app</i> utilizando como servidor Tomcat pero sobre un servidor seguro <i>https</i> . El puerto por defecto es 8443 y puede ser modificado añadiendo al comando - <i>Dserver.port.https=&lt;numero_puerto&gt;</i>
<i>grails schema-export</i>	Genera un fichero con las sentencias SQL necesarias para exportar la base de datos.
<i>grails set-version</i>	Establece la versión de la aplicación. Por ejemplo <i>grails set-version 1.0.4</i>
<i>grails stats</i>	Nos muestra una serie de datos referentes a nuestro proyecto, con respecto al número de controladores, clases de dominio, servicios, librerías de etiquetas, tests de integración, etc. y al número de líneas totales en cada apartado.
<i>grails uninstall-plugin</i>	Desinstala el plugin pasado como parámetro de la aplicación.

## 8.4. Plugins

La comunidad de usuarios de Grails es cada vez mayor y eso hace que, cuando una característica no ha sido desarrollada en el núcleo de Grails, sean los propios usuarios quienes se encarguen de desarrollarla como un plugin externo. En el momento en que se escribieron estos apuntes eran casi 1200 los plugins desarrollados por la comunidad (<http://www.grails.org/plugins/>).

Estos plugins abarcan aspectos tan diversos como la seguridad como el plugin de Spring Security (<http://www.grails.org/plugin/spring-security-core>), el desarrollo de interfaces gráficas

ricas como Rich UI (<http://www.grails.org/plugin/richui>) o la exportación de datos a otros formatos con el plugin Export (<http://www.grails.org/plugin/export>).

Para ver el funcionamiento de los plugins en Grails, nosotros vamos a utilizar un plugin que nos permitirá buscar contenido en las clases de dominio de nuestra aplicación sin prácticamente esfuerzo alguno. También utilizaremos el plugin export para exportar la información de nuestras clases de dominio a otros formatos. Pero para empezar, veamos un plugin que nos permitirá utilizar una consola a través de nuestra aplicación para poder realizar todo tipo de operaciones.

## Plugin console

El plugin console no es más un plugin que permite la ejecución en línea de comandos. Desde esta consola tendremos acceso a determinados contextos como pueden ser la propia aplicación o las clases de dominio. Es muy útil cuando queremos realizar operaciones sobre las clases de dominio y no podemos o no queremos tocar la aplicación para esto.

La instalación del plugin se realiza con el comando *grails install-plugin console* o bien editando el archivo de configuración *BuildConfig.groovy* para añadir el plugin correspondiente:

```
.....  
grails.project.dependency.resolution {  
    ...  
    plugins {  
        ...  
        compile "console:1.5.4"  
    }  
}
```

.....

Una vez instalado el plugin y con la aplicación en funcionamiento, podremos acceder a esta consola desde la dirección <http://localhost:8080/todo/console>.

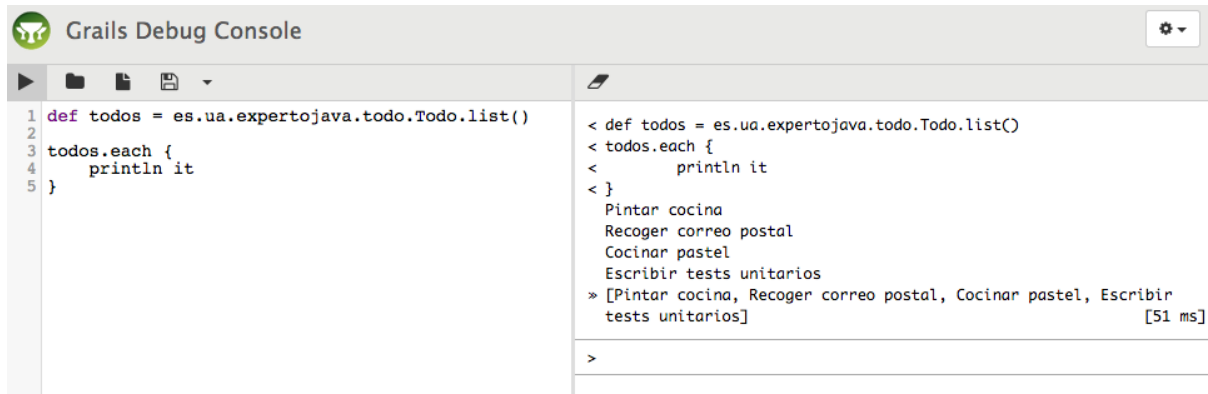
Sin embargo, es probable que, debido a la configuración segura de nuestra aplicación, necesites actualizar las reglas de seguridad para que por ejemplo permita el acceso a esta url a los administradores de la misma.

```
.....  
grails.plugin.springsecurity.controllerAnnotations.staticRules = [  
    "/console/**": ['ROLE_ADMIN'],  
    "/plugins/console*/**": ['ROLE_ADMIN']  
]
```

.....

Ten en cuenta que este ejemplo considera que estamos utilizando el método de Spring Security *Annotation*.





```

Grails Debug Console
1 def todos = es.ua.expertojava.todo.Todo.list()
2
3 todos.each {
4     println it
5 }
< def todos = es.ua.expertojava.todo.Todo.list()
< todos.each {
<     println it
< }
Pintar cocina
Recoger correo postal
Cocinar pastel
Escribir tests unitarios
» [Pintar cocina, Recoger correo postal, Cocinar pastel, Escribir
tests unitarios] [51 ms]
>

```

Con el ejemplo mostrado en la imagen podríamos obtener todos las tareas de nuestra aplicación e imprimirlos por pantalla. Pero además podemos acceder a cualquier artefacto de nuestra aplicación incluidos los servicios, tal y como si estuviéramos ejecutando este código dentro de nuestra aplicación.

Las siguientes variables son accesible desde esta consola:

- `ctx` - el contexto de la aplicación Spring
- `grailsApplication` - la instancia de la aplicación Grails
- `config` - la configuración Grails
- `request` - la petición HTTP actual
- `session` - la sesión HTTP actual

El objeto `ctx` sin duda es el más interesante puesto que nos va a permitir acceder al contexto de la aplicación y por consiguiente a varios artefactos como pueden ser los servicios de forma muy sencilla y por ejemplo eliminar tareas directamente.

---

```

Todo todo = es.ua.expertojava.todo.Todo.get(1)

```

```

ctx.todoService.deleteTodo(todo)

```

---

Este plugin es una forma rápida de probar determinadas funcionalidades de nuestra aplicación.

## Plugin para la búsqueda de contenido: *Searchable*

Para esta tarea de búsqueda, Grails cuenta con un plugin llamado *Searchable* que nos va a facilitar muchísimo esta labor. Este plugin además, está basado en el framework OpenSymphony Compass Search Engine (<http://www.opensymphony.com/compass/>) y que tiene por detrás a Apache Lucene, con lo que el plugin cuenta con mucho respaldo.

Para la instalación de plugin en Grails, debemos modificar el archivo de configuración de configuración *BuildConfig.groovy* y seguir las instrucciones que vienen en la página del plugin <http://grails.org/plugin/searchable>.

Estas instrucciones pasan por añadir un nuevo repositorio maven y añadir el plugin en cuestión.

---

```

grails.project.dependency.resolution = {

```

```
...
repositories {
    ...
    mavenRepo "http://repo.grails.org/grails/core"
}

plugins {
    ...
    compile "searchable:0.6.9"
}
}
```

---

Una vez instalado, podemos comprobar que en la dirección <http://localhost:8080/todo/searchable> (habrá que añadir también esta dirección en la configuración segura de nuestra aplicación), tenemos un buscador que se ha generado automáticamente. Sin embargo, si probamos a hacer alguna búsqueda, el buscador nos dirá que no ha encontrada nada. Esto es porque todavía no le hemos dicho al plugin donde debe buscar la información.

Si pensamos donde podemos añadir un buscador en nuestra aplicación encontraremos que los usuarios de nuestro aplicación querrán buscar tareas, con lo que debemos indicar en la clases de dominio *Todo* que necesitamos que nuestro plugin *searchable* sea capaz de buscar en ella.

Para que el plugin *searchable* sea capaz de encontrar tareas que contengan un determinado texto, necesitamos modificar la clase de dominio *Todo* y añadirle una nueva propiedad llamada *searchable*. Esta propiedad es *static* y debe contener un valor booleano indicando si permitimos la búsqueda o no, con lo que la clase de dominio *Todo* quedarían así:

---

```
package es.ua.expertojava.todo

class Todo {
    String title
    String description
    Date date
    Date reminderDate
    String url
    Boolean done = false
    Category category

    Date dateCreated
    Date lastUpdated

    Date dateDone

    User user

    static searchable = true

    ...
}
```

---

Si ahora realizamos alguna búsqueda de prueba, *Searchable* nos devolverá los resultados encontrados en todas las propiedades de las clases de dominio *Todo*. Ahora bien, estamos permitiendo que se busque el término de búsqueda en todas las propiedades de las clases

de dominio implicadas, algo que no siempre será lo deseado. Para ello, el plugin permite la utilización de dos variables con las que le indicaremos al sistema donde puede buscar o donde no.

Estas dos variables son *only* y *except* y podemos definir las de la siguiente forma:

```
static searchable = [only: ['title', 'description']]
```

para indicarle que debe buscar en las propiedades *title* y *description* o bien mediante:

```
static searchable = [except: 'date', 'reminderDate']
```

si lo que queremos es indicarle sólo aquellas propiedades en las que no debemos realizar la búsqueda. En este último caso, buscaremos en todas las propiedades de la clase de dominio *Todo* excepto en *date* y *reminderDate*.

En el momento de escribir estos apuntes el plugin tenía un problema para funcionar con Hibernate 4 con lo que vamos a tener que actualizar un par de ficheros para utilizar en su lugar Hibernate 3. Si abrimos el archivo de configuración *BuildConfig.groovy* veremos como hay una línea indicando

```
runtime "hibernate4:4.3.5.5" // or "hibernate:3.6.10.17"
```

que vamos a tener que modificar para utilizar Hibernate 3. Además, en el archivo de configuración *DataSource.groovy* encontraremos

```
//cache.region.factory_class =  
'net.sf.ehcache.hibernate.EhCacheRegionFactory' // Hibernate 3  
cache.region.factory_class  
= 'org.hibernate.cache.ehcache.EhCacheRegionFactory' // Hibernate 4
```

y de igual forma, vamos a tener que modificar para utilizar Hibernate 3 en lugar de Hibernate 4.

Con estas modificaciones, si ahora intentamos de nuevo realizar una búsqueda en <http://localhost:8080/todo/searchable> veremos como ya empieza a buscar en los campos *title* y *description* de la clase de dominio *Todo*.

Pero si esto se nos queda corto (y posiblemente sea así), podemos utilizar el servicio *SearchableService*. Además, los métodos de este servicio también están disponibles en las clases de dominio "anotadas" con la variable *searchable*. Estos son los métodos de los que disponemos:

- [search<sup>20</sup>](#): encuentra objetos que cumplen una determinada consulta.
- [countHits<sup>21</sup>](#): encuentra el número de objetos que cumplen una determinada consulta.
- [moreLikeThis<sup>22</sup>](#): encuentra objetos similares a la instancia pasada por parámetro.

<sup>20</sup> <https://grails.org/Searchable+Plugin++Methods++search>

<sup>21</sup> <https://grails.org/Searchable+Plugin++Methods++countHits>

<sup>22</sup> <https://grails.org/Searchable+Plugin++Methods++moreLikeThis>

- [suggestQuery](#)<sup>23</sup>: sugiere una nueva consulta a partir de la consultada pasada por parámetro.
- [termsFreqs](#)<sup>24</sup>: devuelve la frecuencia para los términos en el índice.

Un ejemplo para buscar en las tareas de nuestra aplicación podría ser el siguiente:

```
def searchResult = searchableService.search(
    "Tests",
    [offset: 0, max: 20]
)
println "${searchResult.total} hits:"
for (i in 0..<searchResult.results.size()) {
    println "${searchResult.offset + i + 1}: " +
        "${searchResult.results[i].toString()} " +
        "(score ${searchResult.scores[i]})"
}
```

o bien, también podemos buscar directamente en la clase de dominio en cuestión.

```
//Devuelve un único objeto de tipo Todo
//que coincidan con la búsqueda realizada
def todo = Todo.search(
    "Tests",
    [result: 'top']
)
assert todo instanceof Todo

//Devuelve todos los objetos de tipo Todo
//que coincidan con la búsqueda realizada
def todos = Todo.search(
    "Tests",
    [reload: true, result: 'every']
)
assert todos.each { it instanceof Todo }
```

Podemos comprobar estos ejemplos directamente en la consola vista en el plugin anterior.

En ocasiones es probable que necesitemos filtrar los resultados de una búsqueda por un determinado parámetro. Por ejemplo, en nuestra aplicación si quisiéramos buscar aquellas tareas que pertenezcan a la categoría "Hogar", deberíamos tener algo así:

```
def todos = Todo.search([result: 'every']) {
    must(queryString(params.q))
    must(term( '$/Todo/category/id', Category.findByName("Hogar")?.id))
}
```

Debemos también especificar que entre las propiedades a buscar en la clase de dominio `Todo` se encuentra la propiedad `category` y la clase `Category` también tendrá la propiedad estática `searchable` a cierto para que nos permitan buscar en las instancias de esta clase.

<sup>23</sup> <https://grails.org/Searchable+Plugin+-+Methods+-+suggestQuery>

<sup>24</sup> <https://grails.org/Searchable+Plugin+-+Methods+-+termFreqs>

## Plugin para exportar a otros formatos: Export

Hoy en día, cualquier aplicación que se precie debe ser capaz de exportar sus datos a otros formatos para que el usuario tenga la comodidad de elegir como quiere utilizar los datos. Los formatos más comunes a exportar la información de nuestra aplicación será *pdf*, *hojas de cálculo excel*, *csv (datos separados por comas)* u *ods (la hoja de cálculo de OpenOffice)*.

Esta tarea que en otros sistemas se vuelve complicada y pesada, en Grails podemos solucionarla utilizando un plugin disponible llamado *export*. Como siempre, lo primero que vamos a hacer es instalarlo añadiendo la siguiente línea a nuestro archivo *BuildConfig.groovy*.

```
grails.project.dependency.resolution = {
  ...

  repositories {
    ...
    mavenRepo "http://repo.grails.org/grails/core"
  }

  dependencies {
    ...
    compile 'commons-beanutils:commons-beanutils:1.8.3'
  }
  plugins {
    ...
    compile ":",export:1.7-SNAPSHOT"
  }
}
```

Una vez instalado el plugin *export* debemos añadir algunos *mime types* en la variable *grails.mime.types* del archivo *Config.groovy*. Los nuevos *mime types* serán los relativos a *csv*, *excel*, *pdf* y *ods*. La variable *grails.mime.types* quedaría así:

```
grails.mime.types = [
  all:          '*/*',
  atom:         'application/atom+xml',
  css:          'text/css',
  csv:          'text/csv',
  pdf:          'application/pdf',
  excel:        'application/vnd.ms-excel',
  ods:          'application/vnd.oasis.opendocument.spreadsheet',
  form:         'application/x-www-form-urlencoded',
  html:         ['text/html', 'application/xhtml+xml'],
  js:           'text/javascript',
  json:         ['application/json', 'text/json'],
  multipartForm: 'multipart/form-data',
  rss:          'application/rss+xml',
  text:         'text/plain',
  hal:          ['application/hal+json', 'application/hal+xml'],
  xml:          ['text/xml', 'application/xml']
]
```

Para empezar a utilizar el plugin, debemos incluir la etiqueta `<export:resource/>` en la cabecera del archivo GSP donde queramos incluir las opciones para exportar. Esto incluirá los archivos

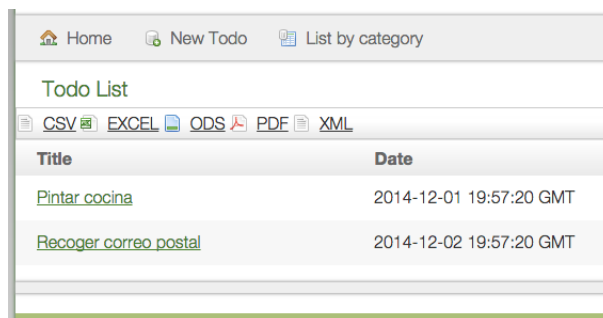
CSS necesarios para crear una barra de herramientas con las opciones para exportar la página actual.

El siguiente paso consistirá en incluir la barra de herramientas necesaria para exportar la página HTML generada a algún formato de los ya comentados. Para esto el plugin pone a nuestra disposición una nueva etiqueta `<export:formats />`, la cual acepta como parámetro un listado de los formatos a los que queremos exportar la página, como por ejemplo `<export:formats formats="['csv','excel','ods','pdf','xml']"/>`.

Si queremos añadir por ejemplo en la página que contiene el listado de las tareas una barra para exportar dicho listado, podríamos tener algo así en la página `grails-app/views/todo/index.gsp`.

```
<html>
  <head>
    ...
    <export:resource/>
  </head>
  <body>
    ...
    <h1><g:message code="default.list.label" args="[entityName]" /></h1>
    <g:if test="${flash.message}">
      <div class="message" role="status">${flash.message}</div>
    </g:if>
    <export:formats formats="['csv', 'excel', 'ods', 'pdf', 'xml']" />
    ...
  </body>
</html>
```

Si echamos un vistazo al listado de tareas comprobaremos como en la parte superior de los mismos aparecerá una barra con los formatos a los que podemos exportar dicho listado.



Sin embargo, si intentamos exportar el listado a cualquier de los formatos utilizados, veremos como no sucede nada nuevo y la aplicación vuelve a mostrarnos el listado tal y como ha aparecido siempre, es decir, en formato HTML. Para que la aplicación pueda exportar a los nuevos formatos, debemos modificar el controlador de la clase de dominio `Todo` y más en concreto el método `index()` para que acepte los nuevos formatos.

```
package es.ua.expertojava.todo
```

```
import static org.springframework.http.HttpStatus.*
```

```

import grails.transaction.Transactional

@Transactional(readOnly = true)
class TodoController {

    def todoService
    def springSecurityService
    def exportService

    static allowedMethods = [save: "POST", update: "PUT",
delete: "DELETE"]

    def index(Integer max) {
        params.max = Math.min(max ?: 10, 100)
        if(params?.f && params.f != "html"){
            response.contentType =
grailsApplication.config.grails.mime.types[params.f]
            response.setHeader("Content-disposition", "attachment;
filename=todos.${params.extension}")
            exportService.export(params.f, response.outputStream,
Todo.findAllByUser(springSecurityService.currentUser, params), [:], [:])
        }
        respond Todo.findAllByUser(springSecurityService.currentUser,
params), model:[todoInstanceCount:
Todo.countByUser(springSecurityService.currentUser)]
    }
    ...
}

```

Ahora nuestra aplicación sí va a ser capaz de exportar a otros formatos para que el usuario final elija cual de ellos utilizar en cada ocasión. Sin embargo, si echamos un vistazo por ejemplo al formato en PDF, comprobaremos como el listado que aparece muestra todas las propiedades de la clase *Todo*, a excepción de la propiedad *version*, lo cual no va a ser aconsejable.

Para mejorar esto, el plugin permite especificar que propiedades queremos mostrar e incluso la etiqueta que queremos que aparezca en la fila de encabezados de la tabla. Para conseguir esto, necesitamos completar los dos últimos parámetros del método *export()* que anteriormente dejábamos vacíos. Además, también necesitaremos indicarle el formato de como deseamos las propiedades de los usuarios y un mapa de parámetros para el fichero exportado. Así quedaría el método *index()*.

```

def index(Integer max) {
    params.max = Math.min(max ?: 10, 100)
    if(params?.f && params.f != "html"){
        response.contentType =
grailsApplication.config.grails.mime.types[params.f]
        response.setHeader("Content-disposition", "attachment;
filename=todos.${params.extension}")

        List props = ["title", "description", "date", "url", "done"]
        Map tags = ["title":"Título",
                    "description":"Descripción",
                    "date":"Fecha",
                    "url":"URL",
                    "done":"Hecho"]
    }
}

```

```
// Closure formateador
def uppercase = { domain, value -> return value.toUpperCase() }

Map formatters = [title: uppercase]
Map parameters = [title: "LISTADO DE USUARIOS"]

exportService.export(params.f, response.outputStream,
    Todo.findAllByUser(springSecurityService.currentUser, params),
    props, tags,
    formatters, parameters)
}
respond Todo.findAllByUser(springSecurityService.currentUser, params),
model:[todoInstanceCount:
    Todo.countByUser(springSecurityService.currentUser)]
}
```

Por último, comentar también que podemos cambiar los textos asociados a cada uno de los formatos que aparecen en la barra para exportar. Simplemente debemos crear nuevas entradas en el archivo *grails-app/i18n/messages.properties* correspondientes, tal y como aparece en el siguiente fragmento de código.

```
default.csv = CSV
default.excel = EXCEL
default.pdf = PDF
default.xml = XML
default.ods = ODS
```



## 8.5. Ejercicios

### Log de acceso a controladores (0.25 puntos)

Con lo visto en esta sesión sobre la configuración de los logs y lo visto en la sesión 4 de los filtros que se pueden crear en los controladores, genera logs de tipo *trace* en todos los métodos de los controladores de nuestra aplicación (sin incluir los controladores *login* y *logout* del plugin *spring security*) para saber el usuario, controlador, método y modelo utilizados en la petición.

```
2015-03-28 11:13:15,863 [http-bio-8080-exec-9] TRACE todo.LogFilters
- User admin - Controlador category - Accion index - Modelo
[categoryInstanceCount:2, categoryInstanceList:[Hogar, Trabajo]]
```

Estos logs deberán ser almacenados en un archivo llamados *logs/filters.log*.

### URL limpia y pública para las tareas de los usuarios (0.50 puntos)

Crea una url del tipo */todos/\$username* como la vista en la parte de teoría que muestre todas las tareas del usuario pasado por parámetro en la URL. Este listado de tareas sólo podrá ser accesible por los usuarios de tipo administrador.

Modifica también el listado de usuarios que pueden ver los administradores para mostrar un enlace en cada usuario para que les permita ver este listado de tareas. Además, este enlace sólo debe ser mostrado para los usuarios que puedan tener tareas, no para los administradores.

### Buscador de tareas integrado (0.50 puntos)

El listado de tareas pide a gritos un buscador integrado. Implementalo utilizando el plugin *searchable* y muéstralo en la parte superior del listado de las tareas. Date cuenta que las tareas están ahora asignadas a un usuario y que por lo tanto, el buscador sólo deberá buscar aquellas tareas que pertenezcan al usuario autenticado.



Estoy seguro que no hace falta que te lo recuerde, pero antes de entregar el proyecto ejecuta *grails test-app unit*: para comprobar que no hayas roto tus tests haciendo otras tareas.