

### Tema 23. Groovy y Grails

#### Contenidos

<b>1. Introducción</b>	1
<b>2. Groovy</b>	2
Elementos específicos de Groovy	2
Operadores	3
Listas y Aplicaciones (List y Map)	5
Cadenas (String)	6
Expresiones Regulares	7
Sentencias de Control	8
Definición de métodos y paso de parámetros	10
Clases y propiedades	12
Metaprogramación	13
<b>3. Grails</b>	13
Clases de Dominio	14
Servicios	20
Controles	21
Vistas	29
Configuración	40

#### 1. Introducción

En capítulos anteriores hemos visto los pasos necesarios para modelar un dominio, realizar casos de uso y diseñar los interfaces usuario. Todo ello con el objetivo de implementar una aplicación web que esté de acuerdo con los requisitos expresados por los clientes y capturados en el modelo de dominio, los casos de uso y los esquemas de interfaces.

Ahora tenemos que escoger un lenguaje de programación y una tecnología que nos permita concretar los detalles y por lo tanto hacer posible la construcción de la aplicación web buscada.

Como lenguaje de programación vamos a escoger *Groovy* y como tecnología *Grails*. Es una tecnología suficientemente madura que permite construir rápidamente prototipos razonablemente buenos de aplicaciones web.

Veamos primero *Groovy* y posteriormente los detalles adicionales de *Grails* para implementar el modelo de dominio.

## 2. Groovy

*Groovy* es una extensión de Java completamente compatible con él. Vamos a ver aquí los aspectos que los diferencian del mismo considerando que el lector conoce los elementos básicos de Java en su versión 8. Solo veremos los principales elementos de *Groovy*. Una documentación más completa puede encontrarse [aquí](#).

### Elementos específicos de Groovy

- Los puntos y comas son opcionales.
- Usar *return* es opcional. Un método devuelve el valor de la última expresión.
- Se puede usar *this* dentro de un método estático refiriéndose a la clase correspondiente.
- Se pueden omitir muchos paréntesis en *Groovy* en expresiones de alto nivel. Por ejemplo en las llamadas a métodos se puede escribir:

```
println "Hola"  
método a, b
```

En vez de

```
println ("Hola")  
método (a, b)
```

- En *Groovy* *protected* tiene el mismo significado que en Java.
- La cláusula *throws* en la cabecera de un método no comprobada por el compilador de *Groovy*, porque no hay diferencia entre las excepciones comprobadas y no comprobadas.
- El acceso a un atributo de un objeto se puede hacer con el operador *.*, *objeto.atributo* como en Java, o como si fuera un índice *objeto[atributo]*.
- El compilador de *Groovy* no dará errores de compilación, como lo haría el de Java, por el uso de los miembros no definidos o el paso de parámetros de tipo incorrecto.
- El operador *==* representa la igualdad para todos los tipos. Si se necesita la identidad en *Groovy* se puede utilizar el método *e1.is(e2)*.
- Todos los objetos pueden ser obligados a tomar valor booleano. Todo lo que se evalúa a *null*, *void* o *empty* se convierte en *false*, y en otro caso *true*.
- Los números reales por defecto son *BigDecimal*s. Así que cuando se escribe 3.14, *Groovy* no creará un *double* o un *float*. Creará un *BigDecimal*.

- *Groovy* tiene clausuras, o expresiones lambda, con la misma sintaxis que Java 8.
- Algunas palabras reservadas específicas de *Groovy*
  - *def* es un sustituto de un nombre de tipo. Se utiliza para indicar que no estamos interesados por el tipo de la variable y dejamos que el compilador los deduzca. En las declaraciones de variables es obligatorio, ya sea proporcionar un nombre de tipo explícita o utilizar *def*.
  - *it* es un nombre de variable especial, que se define automáticamente dentro de una clausura. Se refiere siempre al primer parámetro de la clausura, o null, si no tiene ningún parámetro.
  - *in* es una palabra clave que representa un operador contiene. La expresión *e in a* es equivalente a la expresión *a.contains(e)*.
  - *as* es una palabra clave que representa un operador para el cambio forzado de tipo (casting).
  - *this* tiene el mismo significado que en Java. Es decir la instancia de la clase en la que nos encontramos.
  - *owner: this* o la clausura dentro de la que nos encontramos)
  - *delegate*: por defecto el mismo que *owner*, pero modificable.
- Una variable puede representar valores de diferentes tipos a lo largo de un programa. Por ejemplo:

```
def a
assert a == null
def b = 1
assert b == 1
b = 2
assert b == 2
b = 'cat'
assert b == 'cat'
b = null
assert b == null
```

## Operadores

*Groovy* soporta la sobrecarga de operadores. Esto hace que trabajar con números, colecciones, aplicaciones y otros tipos de datos sea más fácil. La idea general es que cada operador se asocia a un método con un nombre predefinido según la siguiente tabla. Por lo tanto cualquier tipo que tenga métodos como los de la tabla tiene predefinidos los operadores asociados.

Operador	Método
$a + b$	<i>a.plus(b)</i>
$a - b$	<i>a.minus(b)</i>
$a * b$	<i>a.multiply(b)</i>
$a ** b$	<i>a.power(b)</i>

$a / b$	<i>a.div(b)</i>
$a \% b$	<i>a.mod(b)</i>
$a   b$	<i>a.or(b)</i>
$a \& b$	<i>a.and(b)</i>
$a \wedge b$	<i>a.xor(b)</i>
$a++$ or $++a$	<i>a.next()</i>
$a--$ or $--a$	<i>a.previous()</i>
$a[b]$	<i>a.getAt(b)</i>
$a[b] = c$	<i>a.putAt(b, c)</i>
$a << b$	<i>a.leftShift(b)</i>
$a >> b$	<i>a.rightShift(b)</i>
$\text{switch}(a) \{ \text{case}(b) : \}$	<i>b.isCase(a)</i>
$\sim a$	<i>a.bitwiseNegate()</i>
$-a$	<i>a.negative()</i>
$+a$	<i>a.positive()</i>

Los siguientes operadores soportan *null* impidiendo el disparo de la excepción [java.lang.NullPointerException](#).

*Operator    Method*

$a == b$	<i>a.equals(b)</i> or <i>a.compareTo(b) == 0</i> **
$a != b$	<i>! a.equals(b)</i>
$a <=> b$	<i>a.compareTo(b)</i>
$a > b$	<i>a.compareTo(b) &gt; 0</i>
$a >= b$	<i>a.compareTo(b) &gt;= 0</i>
$a < b$	<i>a.compareTo(b) &lt; 0</i>
$a <= b$	<i>a.compareTo(b) &lt;= 0</i>

Groovy tiene otros operadores útiles:

- *Operador de extensión (\*.)*: Invoca una clausura sobre todos los elementos de un agregado

```
assert ['cat', 'elephant']*.size() == [3, 8]
```

- *Operador Elvis (?):* Devuelve un valor por defecto si el operando es false.

```
def displayName = user.name ?: "Anonymous"
```

- *Operador de navegación segura (?.):* Se utiliza para evitar una *NullPointerException*. Normalmente cuando se tiene una referencia a un objeto que pueda necesitar comprobar que no es nulo antes de acceder a sus métodos o propiedades. Para evitar esto, el operador de una navegación segura simplemente devolver *null* en lugar de lanzar una excepción.

```
def streetName = user?.address?.street
```

## Listas y Aplicaciones (List y Map)

En *Groovy* se usa una sintaxis especial para representar listas y aplicaciones. En ambos casos se usa el símbolo `[]` para contener los elementos de la lista o los pares clave-valor de la aplicación. Los pares clave valor se separan por el símbolo `:`.

```
def list= [1, 2, 3]
list= [] //lista vacía
list = [1, 'b', false, 4.5 ] // una lista puede tener elementos
                        //de distintos tipos
assert list[0]==1 && list[1]=='b' && ! list[2] && list[3] == 4.5
//podemos referirnos a un elemento por su índice
def map= [1:'a', 2:'b', 3:'c']
map= [:] //map vacío
map= ['a': 1, 'b': 'c', 'groovy': 78.9, 12: true]
                        //valores de distintos tipos
assert map['a']==1
assert map['b']=='c'
assert map['groovy']==78.9
assert map[12] //podemos referirnos a valores por una clave
```

Las listas y las aplicaciones tienen unos métodos y operadores específicos:

- *each* clausura: Ejecuta la clausura sobre cada elemento de las lista o de la aplicación

```
[2, -17, +987, 0 ].each { n -> println n}
[1: 3, 2: 6, 3: 9, 4: 12 ].each{k, v-> assert k * 3 ==v }
```

Podemos definir listas como rangos de valores:

```
(3..7).each{ println it } //3, 4, 5, 6, y 7
```

```
(3..<7).each{ println it } //3, 4, 5, and 6 y excluido 7
```

Los *conjuntos* son colecciones no ordenadas con objetos sin duplicados. Se obtienen a partir de una lista aplicando el operador correspondiente:

```
def s1= [1,2,3,3,3,4] as Set,  
s2= [4,3,2,1] as Set,  
s3= new HashSet( [1,4,2,4,3,4] )
```

Más métodos para listas, conjuntos y aplicaciones pueden encontrarse en los tipos [Collection](#), [Object](#), [List](#) y otros tipos del API de Groovy.

Para acceder al valor asociado a una clave de un Map hay varias posibilidades.

```
def myMap = [a:1, b:2, c:3]  
assert myMap['a'] == 1  
assert myMap.a == 1  
assert myMap.get('a') == 1
```

## Cadenas (String)

Hay varios tipos de cadenas de caracteres en Groovy: simples comillas, dobles comillas, triples comillas y triples comillas dobles.

Las cadenas con triples comillas sirven para escribir texto multilínea.

```
assert '''hello,  
world''' == 'hello,\nworld'
```

Dentro de las cadenas con dobles comillas se puede insertar código en la forma *\${expresión}*. La expresión se evaluará con los valores de las variables implicadas y se sustituirá por el resultado.

```
def a = 'How are you?'  
assert "The phrase '$a' has length ${a.size()}" ==  
    "The phrase 'How are you?' has length 12"
```

Los métodos específicos para este tipo en Groovy se pueden encontrar en [String](#).

```
assert 'abcde'.find{ it > 'b' } == 'c'  
assert 'abcde'.findAll{ it > 'b' } == ['c', 'd', 'e']  
assert 'abcde'.indexOf{ it > 'c' } == 3  
assert 'abcde'.every{ it < 'g' } && ! 'abcde'.every{ it < 'c' }
```

```

assert 'abcde'.any{ it > 'c' } && ! 'abcde'.any{ it > 'g' }
assert 'abcdefg'[ 3 ] == 'd'
assert 'abcdefg'[ 3..5 ] == 'def'
assert 'hello, ' + 'balloon' - 'lo' == 'hel, balloon'
assert 'a' * 3 == 'aaa' && 'a'.multiply(3) == 'aaa'
assert 'he she\t it'.tokenize() == ['he', 'she', 'it']
    //los caracteres separadores son ' \t\n\r\f'
assert 'he,she;it,;they'.tokenize(',') == ['he', 'she', 'it', 'they']
    //se proporcionan los caracteres separadores

```

## Expresiones Regulares

Las expresiones regulares son mecanismos para describir conjuntos de cadenas de mediante un patrón. Un patrón se escribe de la forma */patrón/*. Dónde patrón es una expresión formada por caracteres y los siguientes operadores:

a?	0 o 1 ocurrencias de *a*	'a' o la cadena vacía
a*	0 o más ocurrencias de *a*	La cadena vacía o 'a', 'aa', 'aaa', etc
a+	1 más ocurrencias de *a*	'a', 'aa', 'aaa', etc
a b	Lo descrito por *a* o por *b*	'a' o 'b'
[woeirjsd]	Cualquiera de los caracteres incluidos	'w', 'o', 'e', 'i', 'r', 'j', 's', 'd'
[1-9]	Cualquiera de los caracteres en el rango	'1', '2', '3', '4', '5', '6', '7', '8', '9'
[^13579]	Cualquiera de los caracteres no incluidos	Dígitos pares, o cualquier otro carácter
(ie)	Operador de agrupamiento	'ie'
^a	*a* en el comienzo de una línea	'a'
a\$	*a* a final de una línea	'a'
\d	Cualquier dígito	
\w	Cualquier carácter alfabético, numérico o el subrayado.	
\s	Espacio, tabulador, salto de línea, salto de página, return	
{n}	La expresión a la izquierda repetida n veces	"a{5}" == "aaaaa"
{m,n}	Entre m y n repeticiones de la expresión izquierda	"ba{1,3}" ej. "ba", "baa", "baaa"
{m,}	Al menos m repeticiones de la expresión izquierda	"\w\d{2,}" ej. "a12", "_456"
.	Un simple carácter	'a', 'q', 'l', '_', '+', etc

Una vez conocida la forma de escribir patrones vemos los operadores que combinan cadenas y patrones. Son los siguientes:

- `~/patrón/`: Crea un objeto de tipo *Pattern* de Java cuyos métodos se pueden encontrar [aquí](#).
- `cadena=~/patrón/`: Devuelve true si alguna subcadena pertenece al conjunto descrito por el patrón. Pueden haber varias subcadenas que concuerden con el patrón. Además, a partir de la *cadena* y el *patrón*, crea un objeto de tipo *Matcher* de Java cuyos métodos se pueden encontrar [aquí](#). Un objeto de tipo *Matcher* es similar a una lista y puede ser tratado como tal con los operadores adecuados: puede ser indexado para encontrar las coincidencias e iterado. Si *m* es un *Matcher* entonces `m[0]` devuelve una lista con la primera concordancia. El primer elemento de esa lista es la subcadena que concuerda con la expresión regular completa, y el resto de elementos son las cadenas que coinciden con cada grupo del patrón.

```
matcher = 'Groovy is groovy' =~ /(G|g)roovy/  
print "Size is ${matcher.size()} "  
println "with elements ${matcher[0]} and ${matcher[1]}."
```

Cuyo resultado sería:

```
Size is 2 with elements ["Groovy", "G"] and ["groovy", "g"].
```

- `cadena ==~/patrón/`: Devuelve true si la cadena completa pertenece al conjunto descrito por el patrón.
- *Sustituciones*: Se puede reemplazar texto usando los métodos `replaceFirst(cadena)` (para reemplazar solo la primera concordancia por *cadena*) o `replaceAll(cadena)` (para reemplazar todas la concordancias por *cadena*).

```
str = 'Groovy is groovy, really groovy'  
println str  
result = (str =~ /groovy/).replaceAll('hip')  
println result
```

Cuyo resultado sería:

```
Groovy is groovy, really groovy  
Groovy is hip, really hip
```

## Sentencias de Control



Las sentencias de control son similares a Java con algunas diferencias. La sentencia *for* usa *in* en vez de “:”. La mayoría de las iteraciones sobre listas y aplicaciones es más conveniente hacerlo mediante métodos específicos y clausuras:

```
def list = [10,11,12,13,14,15,16,17,18,19,20]
//Encuentra el primer par
def even = list.find {item -> item % 2 == 0}
//Una lista con todos los pares
def llist = list.findAll{it % 2 == 0}
//Son todos menores que 30
assert list.every { item -> item < 30}
//Hay alguno menor que 15
assert list.any { item -> item < 15}
// Añade cada item a store
def store = ''
list.each { item -> store += item }
alist << 27 //añadir elementos a una lista
alist << 26
// Crear una nueva lista con los elementos calculados por una
// expresión a partir de los antiguos
println alist.collect{it * 10}
// Crear una nueva lista con los elementos calculados por una
// expresión a partir de los antiguos
println alist*.length()
```

Usando los métodos apropiados de tipos como *Number* o *Range* podemos llevar a cabo iteraciones

```
5.times {
    println "Hello"
}
(6..10).each { e ->
    println e
}
```

La declaración de excepciones en la signatura de un método es opcional.

La sentencia *switch* es más potente que en Java. Un objeto que tenga el método *isCase* es un candidato para ser usado en un cae de una sentencia switch.

Así podemos escribir:

```
switch (10) {
    case 0 : assert false ; break
    case 0..9 : assert false ; break
    case [8,9,11] : assert false ; break
```

```
case Float : assert false ; break
case {it%3 == 0}: assert false ; break
case ~/./ : assert true ; break
default : assert false ; break
}
```

## Definición de métodos y paso de parámetros

Los métodos se declaran de forma similar a Java pero teniendo en cuenta que se puede prescindir del *return*, el tipo devuelto y los tipos de los parámetros formales.

*Groovy* soporta el tipo paso de parámetros que en Java. Este tipo lo denominaremos paso de parámetros por posición. Al declarar el método se incluyen una serie de parámetros formales cada uno de ellos con un nombre y un tipo (que puede ser omitido en *Groovy*). En la llamada al método se incluyen tantas expresiones como parámetros formales. Los valores de las expresiones se pasan a los parámetros según la posición en la que estén.

*Groovy* también soporta un número variable de parámetros como en Java.

```
class Calculator {
    //La convención en Groovy es que si el argument es de tipo Object[]
    //entonces la llamada al método puede usar un número variable de
    //parámetros
    def addAllGroovy( Object[] args ){
        int total = 0
        args.each { total += it }
        total
    }

    // La notación Java5 está también disponible en Groovy
    def addAllJava( int... args ){
        int total = 0
        args.each { total += it }
        total
    }
}

Calculator c = new Calculator()
assert c.addAllGroovy(1,2,3,4,5) == 15
assert c.addAllJava(1,2,3,4,5) == 15
```

*Groovy* dispone de otros mecanismos de paso de parámetros: parámetros con nombre y parámetros con valor por defecto.

Las llamadas a métodos con *parámetros con nombre* son de la forma *foo(p1:e1, p2:e2, ..., pn:en)*. Donde *p1, p2, ...pn* son los nombre de los parámetros con nombre y *e1,e2,...en* sus

correspondientes valores. Las llamadas a métodos con parámetros con nombre se corresponden con declaraciones de métodos que tienen un *Map* como primer parámetro. Dentro del cuerpo del método se accede a los parámetros vía las claves del *Map*.

```
class Math {
    static getSlope( Map args ){
        def b = args.b ? args.b : 0
        (args.y - b) / args.x
    }
}

assert 1 == Math.getSlope( x: 2, y: 2 )
assert 0.5 == Math.getSlope( x: 2, y: 2, b: 1 )
```

Un método puede tener parámetros posicionales, con nombre y número variable de parámetros. El parámetro que declara número variable de los mismos (de tipo *Object[]*) irá el último como en Java pero los parámetros por nombre y por posición pueden combinarse, en la llamada, de cualquier manera. El compilador los reordena. Así la llamada *foo(p1:e1, q1, p2:e2, ..., pn:en, q2, ..., qm)* será transformada a *foo([p1:e1, p2:e2, ..., pn:en], q1, q2, ..., qm)*.

Un método puede recibir una clausura como parámetro. Cuando es el último parámetro se puede poner fuera del paréntesis de cierre, e incluso omitir los paréntesis. Las siguientes expresiones son equivalentes:

```
list.each( { println it } )
list.each(){ println it }
list.each { println it }
```

*Groovy* también soporta parámetros con valores por defecto en la declaración. Un parámetro con un valor por defecto puede ser omitido en la llamada. En ese caso el parámetro toma el valor por defecto.

```
def foo(x=1,y=2) {x+y}
assert foo() == 3
assert foo(2) == 4
assert foo(5,10) == 15
```

Como ejemplo podemos considerar:

```
class Summer {
    def sumWithDefaults(a, b, c=0) { return a + b + c }

    def sumWithOptionals(a, b, Object[] optionals) {
        return a + b + optionals.inject(0) { sum, i -> sum += i }
    }

    def sumNamed(Map args) {
        ['a','b','c'].each{ args.get( it, 0 ) }
        //Da valores por defecto a los parámetros con nombre a, b,c
    }
}
```

```

        return args.a + args.b + args.c
    }
}
def summer = new Summer()
assert 2 == summer.sumWithDefaults(1,1)
assert 3 == summer.sumWithDefaults(1,1,1)
assert 2 == summer.sumWithOptionals(1,1)
assert 3 == summer.sumWithOptionals(1,1,1)
assert 2 == summer.sumNamed(a:1, b:1)
assert 1 == summer.sumNamed(c:1)

```

## Clases y propiedades

- Métodos, clases y atributos son públicos por defecto.
- Una propiedad pública modificable se declara en *Groovy* mediante un atributo con el mismo nombre que la propiedad con la primera letra minúscula. No hacen falta los correspondientes *getters* y *setters*. Implícitamente por cada propiedad el compilador crea un atributo y su correspondiente *getter* y *setter*.
- Un método de la forma *getNombre()* define una propiedad con identificador *nombre* que puede no tener asociado un atributo. Si también incluimos el método *setNombre(..)* la propiedad será modificable. En ese caso es preferible en general usar un atributo público.
- Una propiedad pública no modificable se declara en *Groovy* mediante un atributo con el cualificador *final*.
- Si desea una propiedad *private* o *protected* tiene que proporcionar su propio *get* y *set* que deben ser declarado *private* o *protected*.
- Para acceder a una propiedad (definida mediante un atributo público o mediante un método que comienza por *get*) podemos hacerlo mediante el operador *"."* o el operador *[]*.
- Para acceder a atributo podemos hacerlo mediante el operador *"@"*.
- Por cada clase se proporciona un constructor por defecto con parámetros con nombre

Veamos un ejemplo:

```

class Person {
    String name
    String lastname
    private int id

    String toString() { "${name} ${lastname}" }
    def getFullName() { "${name} ${lastname}" }
}

Person person = new Person( name: 'Andres', lastname: 'Almiray' )
assert "Andres Almiray" == person.toString()
person.lastname = "Jaramillo"
assert "Andres Jaramillo" == person.toString()

```

```
person['lastname'] = "Almiray"
assert "Andres Almiray" == person.toString()
assert person.name == "Andres" // acceso a la propiedad
assert person.@name == "Andres" // acceso al campo
assert person.fullName == "Andres Almiray"
try{ assert person.@fullName == "Andres Almiray" }
catch( MissingFieldException e ){
    println "ERROR, fullName no es un atributo de Person"
}

// imprime
// "ERROR, fullName no es un atributo de Person"
```

En el ejemplo anterior definimos un tipo llamado *Person*, que tiene 2 propiedades públicas [*nombre*, *apellido*] y atributo privado [*id*]. Luego creamos una instancia de esa clase mediante el constructor por defecto.

## Metaprogramación

Groovy tiene capacidades similares a la Programación Reflexiva en Java pero muchos más potentes. Dispone de varios mecanismos para conseguir este objetivo. Uno de ellos es la *ExpandoMetaClass* cuyos detalles pueden encontrarse [aquí](#).

Con ese mecanismo se pueden descubrir métodos y propiedades en tiempo de ejecución, añadir a una clase constructores, métodos, propiedades o tomar prestados métodos de otras clases.

### 3. Grails

*Grails* es un *framework* de desarrollo de aplicaciones web basado en *Groovy* y *Java* que se puede desplegar en servidores web existentes de *Java*.

*Grails* permite crear rápidamente aplicaciones web sin dedicar mucho tiempo a la configuración del mismo. *Grails* logra esto proporcionando automáticamente el contenedor web *Tomcat* y la base de datos *HSQLDB* durante el desarrollo. Aunque fácilmente se puede usar *MySQL* como base de datos durante el desarrollo. En el despliegue se puede sustituir el contenedor o la base de datos.

Veremos en un primer momento una visión general de *Grails* sin preocuparte por detalles de configuración que veremos más adelante.

*Grails* tiene disponibles los siguientes elementos: Clases de Dominio, Controladores, Vistas y Servicios. Veamos cada uno de ellos.

## Clases de Dominio

Son clases escritas en *Groovy* que sirven para implementar Entidades. Es decir tipos persistentes. *Grails* se encarga de gestionar la persistencia de instancias de una Clase de Dominio. Con estas clases podemos implementar todos los elementos de un modelo de dominio. Un ejemplo de Clase de Dominio es:

```
class Person {  
    String firstName  
    String lastName  
    int age  
}
```

Implicítamente *Grails* añade una propiedad más llamada *id* de tipo que es un identificador único para cada instancia del tipo y le da valores adecuados cuando se crea una instancia nueva.

Como veremos más adelante es posible escribir el invariante del tipo, las asociaciones del mismo con otros tipos y las relaciones de generalización en que pueda estar involucrado.

*Grails*, además, proporciona automáticamente métodos para consultar la población del tipo. Así para el tipo *Person* estarán disponibles automáticamente los métodos:

- *Person.list()*: Recupera toda la población
- *Person.get(id)*: Recupera la instancia con el identificador dado y *null* si este no existe.
- *Person.getAll(id1,id2,id3)*: Recupera una lista de instancias con los identificadores específicos. Si algunos de los identificadores proporcionados son nulos o no existen la lista resultante tendrá valores nulos en esas posiciones.
- *Person.load(id)*: Devuelve un proxy de la instancia con el identificador proporcionado. Si esa instancia ya está en memoria la devuelve. Si no está se crea un proxy que será inicializado cuando se llame a algún método del mismo.
- *Person.read(id)*: Devuelve una instancia no modificable con el identificador proporcionado.

A su vez si *p* es una instancia de cualquier tipo tiene disponibles los métodos:

- *p.save()*: Guarda en la base de datos, o actualiza, la instancia *p* y propaga las actualizaciones a las instancias con las que está asociada. Hay que tener en cuenta que *Grails valida* una instancia de dominio cada vez que la guarda. Si falla la validación no se guarda en la base de datos. Por defecto, *save()* simplemente devuelve *null* en este caso, pero si prefiere a que dispare una excepción se le proporciona el parámetro argumento *FailOnError: true*. Puesto que en *Groovy null* se convierte a *false* podemos escribir código el siguiente código:

```
if (!b.save()) {  
    b.errors.each {  
        println it  
    }  
}
```

```
}  
}
```

- *p.delete()*: Borra la instancia de la base de datos.
- *p.errors*: La propiedad *errors* es utilizada por *Grails* para guardar los errores que puedan aparecer cuando se llama al método *save()* o *validate()*. Es decir cuando se intenta comprobar el invariante.
- *p.validate()*: Comprueba si cumple el invariante
- *p.hasErrors()*: Indica si hay errores después de llamar a *save()* o *validate()*.

Y también están disponibles métodos, que denominaremos buscadores dinámicos (*Dynamic Finders*), cuyos nombres son una combinación de las palabras *findByPropiedadSufijo(param)* o *findAllByPropiedadSufijo(param)*. El primero devuelve una instancia y el segundo un conjunto de ellas. Algunos de los sufijos posibles junto con los parámetros permitidos son:

```
InList(p1,p2,...,pn): En la lista de valores  
LessThan(p): Menor que p  
LessThanEquals(p): Menor o igual que p  
GreaterThan(p): Mayor que p  
GreaterThanEquals(p): Mayor o igual que p  
NotEqual(p): No igual a p  
InRange(p1,p2): Entre p1 y p2  
Between(p1,p2): Entre p1 y p2  
NotNull(): No es null  
IsNull(): Es null
```

Algunos ejemplos son:

```
def results = Book.findAllByTitle("The Shining")  
results = Book.findAllByReleaseDateBetween(firstDate, new Date())  
results = Book.findAllByReleaseDateGreaterThanEquals(firstDate)  
results = Book.findAllByTitleNotEqual("Harry Potter")  
results = Book.findAllByAuthorInList(["Adams", "Thompson"])
```

Más variantes de estos métodos se pueden encontrar en la [documentación](#).

### *Invariante*

*Grails* permite establecer fácilmente el invariante de una entidad. Esto se hace con una sección etiquetada *constraints* dentro del código de una clase de dominio. En esta sección se establecen restricciones específicas para cada propiedad. Para ello cuenta con un conjunto de restricciones predefinidas que pueden aplicarse a las distintas propiedades según su tipo.

Nombre	Comentario	Ejemplo
<i>blank</i>	No es blank	<i>blank : false</i>
<i>creditCard</i>	Es un número válido de una tarjeta de crédito	<i>creditCard: true</i>
<i>email</i>	Es un email válido	<i>email: true</i>
<i>inList</i>	Está en la lista	<i>inList:["Joe","Fred","Bob"]</i>
<i>matches</i>	Concuerda con la expression regular	<i>matches: "[a-zA-Z]+"</i>
<i>max</i>	No excede el valor indicado	<i>max: new Date()</i>
<i>maxSize</i>	El tamaño no excede el valor indicado	<i>maxSize: 25</i>
<i>min</i>	El valor no es menor que el indicado	<i>min: 18</i>
<i>minSize</i>	El tamaño no es menor que el valor indicado	<i>minSize: 25</i>
<i>notEqual</i>	No es igual al valor indicado	<i>notEqual: "Bob"</i>
<i>nullable</i>	Permite que la propiedad tome el valor null	<i>nullable: true</i>
<i>range</i>	Está dentro del rango	<i>range: 18..65</i>
<i>scale</i>	Numero de dígitos de la parte decimal	<i>scale: 2</i>
<i>size</i>	El tamaño de una colección está en el rango	<i>size: 5..15</i>
<i>unique</i>	Tiene un valor único en la población	<i>unique: true</i>
<i>url</i>	Es un url válido	<i>url: true</i>
<i>validator</i>	Adds custom validation to a field.	<i>Una clausura</i>

Una restricción en general se puede implementarse con un *validator*. Este se implementa mediante una clausura que puede tener hasta tres parámetros. Si tiene un parámetro (o uno implícito en la variable *it*) el valor será el que se está validando. Si se acepta dos parámetros el primero es el valor de la propiedad que está siendo validada y el segundo la instancia de clase que se está validando. Esto es útil cuando la validación necesita tener acceso a las otras propiedades.

El cierre puede devolver: *null* o *true* (o ningún valor de retorno) para indicar que el valor es válido, *false* para indicar que el valor es no válido y añadiendo un código de mensaje.

Como ejemplo podemos ver:

```
class User {
    String login
    String password
    String email
    Integer age

    static constraints = {
        login size: 5..15, blank: false, unique: true
        password size: 5..15, blank: false
    }
}
```



```
        email email: true, blank: false
        age min: 18
    }
}
```

Se establece, por ejemplo, que la edad mínima son 18, el *login* es único, debe tener un tamaño en caracteres entre 5 y 15 y no puede ser vacía (*blank*).

El orden en que se escriben las restricciones puede ser importante para el código que se puede generar automáticamente. Este orden es el orden en que se visualizarán las correspondientes propiedades en el interfaz web generado.

### *Relaciones y Asociaciones*

Como en Java las clases de dominio admiten herencia que se expresa de forma similar.

Como vimos en el capítulo anterior una asociación entre tipos puede tener unos roles en sus extremos y unos detalles como su navegabilidad, su cardinalidad y los mecanismos de propagación de las actualizaciones de las instancias. Las asociaciones se definen usando las palabras reservadas *hasOne*, *hasMany*, *belongsTo*, *embedded* y *mappedBy*. Las dos primeras definen la cardinalidad del rol correspondiente. La tercera establece el sentido del mecanismos de propagación de actualizaciones (o borrados). La cuarta si una instancia es parte de otra. La quinta define los roles de dos tipos que forman parte de una asociación.

Veamos una primera idea de cómo definir los detalles de una asociación. En la siguiente sección estudiaremos con más detalle estos elementos.

Sobre los nombre de los roles podemos establecer restricciones adicionales mediante la sección *constraints* vista arriba.

*Unidireccional muchos-a-uno de A hasta B:* Cada instancia de A está asociada a una de B. La instancia de B puede estar asociada a varias de A. Se indica con una propiedad de tipo B en la clase A.

```
class A {
    B b
}
class B {
}
```

*Bidireccional muchos-a-uno de A hasta B:* Se indica con una propiedad de tipo B en la clase A e indicando que la instancia B pertenece a A.

```
class A {
    B b
}
class B {
    static belongsTo = [a:A]
```

```
}
```

***Bidireccional uno-a-uno de A hasta B:***

```
class A {
    static hasOne = [b:B]

    static constraints = {
        b unique: true
    }
}
class B {
    A a
}
```

***Unidireccional uno-a-muchos:*** Una clase A tiene asociadas muchas instancias de otra clase B

```
class A {
    static hasMany = [bs: B]
    ...
}
class B {
    ...
}
```

La propiedad *bs* de la clase *A* es de tipo *Set* por defecto. Si queremos una lista habrá que definir la propiedad que define la asociación de tipo *List*. Podemos indicar, también, el mecanismo de propagación de actualizaciones. Por defecto, si no se indica con *belongsTo* se propagan las actualizaciones de *A* hasta *B* pero no el borrado de instancias. El borrado también se propaga si se incluye *belongsTo*.

```
class A {
    static hasMany = [bs: B]
    ...
}
class B {
    static belongsTo = [a:A]
    ...
}
```

***Bidireccional muchos-a-muchos:***

```
class A {
    static belongsTo = B
```

```

    static hasMany = [bs:B]
    ...
}
class A {
    static hasMany = [as:A]
    ...
}

```

*Composición:* Se soporta con la palabra *embedded*.

```

class Person {
    Address homeAddress
    Address workAddress
    static embedded = ['homeAddress', 'workAddress']
}
class Address {
    String number
    String code
}

```

### *Detalles de asociaciones*

Veamos con un algo más de detalles cada uno de los términos que usamos para definir las asociaciones: *hasOne*, *hasMany*, *belongsTo*, *embedded* y *mappedBy*.

- *belongsTo*: Define una asociación *pertenece a* entre un tipo *B* y otra *A*. El tipo especificado por *belongsTo* asume la propiedad de la asociación y lo denominaremos *propietario*. El otro *dependiente*. Ambos pueden ser del mismo tipo. Esto tiene el efecto de controlar la propagación en cascada, desde el propietario al dependiente, de la eliminación de instancias (*delete*) y la actualización (*save*). El comportamiento exacto depende del tipo de relación:
  - *Muchos-a-uno/uno-a-uno*: Guarda y borra en cascada desde propietario hacia el dependiente.
  - *Uno-a-muchos*: Por defecto propaga la actualización (*save*) desde el lado de uno hacia el lado de muchos. Si en el lado de muchos tiene *belongsTo* entonces también propaga la eliminación.
  - *Muchos-a-muchos*: propaga la actualización desde el tipo propietario al dependiente pero no la eliminación..
- *hasOne*: Define una asociación bidireccional uno-a-uno entre dos clases
- *hasMany*: Define una asociación uno-a-muchos entre dos clases. Por defecto se define una propiedad de tipo *Set*. Si se quiere de tipo *List*, *SortedSet*, *Collection* hay que definir explícitamente la propiedad del tipo correspondiente.
- *embedded*: Define una relación de composición entre tipos.

- *mappedBy*: Permite especificar si una asociación es unidireccional o bidireccional, y los roles de ambos extremos en el caso de las asociaciones bidireccionales. Con *mappedBy* se especifican pares clave-valor de propiedades (roles) en dos tipos relacionados por una asociación. La clave es una propiedad en la clase actual y el valor otra propiedad en la clase del otro extremo de la asociación (que puede ser en sí mismo). Si el valor asociado es *none* entonces la asociación es unidireccional. A partir de las propiedades definidas en dos tipos es posible deducir, en la mayoría de los casos, las características de la asociación entre ellos (*roles*, *cardinalidad*, *uni o bidireccionalidad*) pero para hacer éstas más explícitas o para cambiar estas propiedades podemos usar *mappedBy*. En el ejemplo siguiente se establece que propiedades *outgoingFlights*: *'departureAirport'* son dos extremos de una asociación.

```
class Airport {
    static mappedBy = [outgoingFlights: 'departureAirport',
                      incomingFlights: 'destinationAirport']
    static hasMany = [outgoingFlights: Route,
                     incomingFlights: Route]
}
class Route {
    Airport departureAirport
    Airport destinationAirport
}
```

En el ejemplo siguiente se establece que *parent* y *supervisor* son dos asociaciones unidireccionales en vez de los dos roles extremos de una asociación bidireccional.

```
class Person {
    String name
    Person parent
    static belongsTo = [ supervisor: Person ]
    static mappedBy = [ supervisor: "none", parent: "none" ]
    static constraints = { supervisor nullable: true }
}
```

## Servicios

Por cada tipo podemos definir un servicio con un conjunto de métodos necesarios asociados a los casos de uso básicos. Los casos de uso básico los denominados CRUD (*Create*, *Read*, *Update*, *Delete*) necesitan servicios asociados en el caso de que el tipo en cuestión tenga asociaciones. Los servicios asociados a una Clase de Dominio llamada *Nombre* les asignaremos el identificador *NombreService*.

- *Create*: Crea una entidad a partir de valor de sus propiedades. Este servicio es adecuado para crear una entidad a partir de sus propiedades e inicializar las instancias a las que está asociada. Por ejemplo:

```
Pet createPet(String name, Date birthDate, long petTypeId, long id) {
    def pet = new Pet(name: name, birthDate: birthDate,
        type: PetType.load(petTypeId), owner: Owner.load(ownerId))
    pet.save()
    pet
}
```

Podemos observar como para las entidades asociadas se crean proxis con el método `load` a partir de un parámetro que representa la identidad de la instancia correspondiente.

- *Update*: Actualiza una instancia a partir de unas nuevas propiedades y nuevas identidades si las propiedades son a su vez entidades.

```
void updatePet(Pet pet, String name, Date birthDate, long petTypeId,
    long ownerId) {
    pet.name = name
    pet.birthDate = birthDate
    pet.type = PetType.load(petTypeId)
    pet.owner = Owner.load(ownerId)
    pet.save()
}
```

- *Delete*: Elimina una instancia de la base de datos

```
void deletePet(long petId) {
    def pet = Pet.get(petId)
    pet.delete()
}
```

## Controles

Junto a las clases de dominio y los servicios los controles son el otro elemento básico de *Grails*. Un controlador (*Controller*) es una clase *Groovy* que sirve para

### *Direcciones en la web*

En la web hay muchos recursos. En informática, un identificador uniforme de recursos (*URI*, *uniform resource identifier*) es una cadena de caracteres que se utiliza para identificar el nombre de un recurso. Esta identificación permite interaccionar con un recurso a través de una red, aquí la *World Wide Web*, usando protocolos específicos. Las *URIs* vienen definidas mediante esquemas que especifican una sintaxis concreta y los protocolos asociados. Una forma muy común de *URI* es el localizador uniforme de recursos (*URL*, *uniform resource locator*), a la que frecuentemente se la denomina informalmente *dirección web*. Un *URL* es un *URI* que, además de la identificación de un recurso web, especifica los medios de actuar sobre

él o de obtener una representación del mismo, especificando un mecanismo de acceso y su ubicación de red. Por ejemplo, el URL *http://example.org/wiki/Main\_Page* se refiere a un recurso identificado como */wiki/Main\_Page* cuya representación, en forma de código *HTML*, se puede obtener a través del protocolo de transferencia de hipertexto (*http*) a partir de una autoridad cuyo nombre de dominio es *example.org*.

Existe un *URL* único para cada página de cada uno de los documentos web. El *URL* de un recurso es su dirección en Internet, la cual permite que el navegador la encuentre y la muestre de forma adecuada. Por ello el *URL* combina el nombre del ordenador que proporciona la información, el directorio donde se encuentra, el nombre del archivo, y el protocolo a usar para recuperar los datos.

En general un *URI* consta de las siguientes partes:

- *Esquema*: nombre que se refiere a una especificación para asignar los identificadores, e.g. *urn*;, *tag*;, *cid*:. En algunos casos también identifica el protocolo de acceso al recurso, por ejemplo *http*;, *mailto*;, *ftp*;, etc.
- *Autoridad*: Es un identificador de la entidad a la que pertenece el recurso.
- *Ruta*: Información usualmente organizada en forma jerárquica, que identifica al recurso en el ámbito del esquema URI y la autoridad (*/domains/example*).
- *Consulta*: Información con estructura no jerárquica, usualmente en la forma de pares "clave=valor", que identifica al recurso en el ámbito del esquema URI y la autoridad especificada. El comienzo de este componente se indica mediante el carácter '?'.
- *Fragmento*: Permite identificar una parte del recurso principal, o vista de una representación del mismo. El comienzo de este componente se indica mediante el carácter '#'.

En definitiva la estructura sintáctica de un URI es:

<code>&lt;scheme name&gt; : //authority/path [ ? &lt;query&gt; ] [ # &lt;fragment&gt; ]</code>
--

La autoridad (*authority*) , contiene información opcional del usuario (en la forma *usuario: contraseña@*), un nombre de host (por ejemplo, el nombre de dominio o dirección IP), y un número de puerto opcional, precedido por dos puntos ":".

La ruta de acceso (*path*) define una ruta a un archivo físico o lógico.

Las vistas anteriormente se las denomina *URIs absolutas*. Junto a ellas podemos usar *URIs relativas*. Las *URIs relativas* prescinden de algunas partes de las URIs para hacerlas más breves. Como se trata de *URIs* incompletas, es necesario disponer de información adicional para obtener el recurso enlazado. En concreto, para que una *URI* relativa sea útil es imprescindible conocer la URI que establece el punto de referencia (posiblemente un directorio) que llamaremos *base*. El navegador calcula la URI absoluta a partir de la base y la URI relativa.

Las *URIs* relativas tienen las siguientes formas particulares (suponemos que el recurso se llama *fichero.ext*):

- El recurso está en el directorio base: *fichero.txt*
- El recurso se encuentra en el directorio padre de la base: *../fichero.ext*
- El recurso se encuentra en un subdirectorio de la base: *subdirectorio/fichero.ext*
- El recurso se encuentra en el directorio raíz: */fichero.txt*.

Dentro de los esquemas posibles aquí estamos interesados en el *HTTP*. El Protocolo de transferencia de hipertexto (*HTTP*) está diseñado para permitir la comunicación entre clientes y servidores. *HTTP* funciona como un protocolo de petición-respuesta entre un cliente y el servidor. Un navegador web puede ser el cliente y una aplicación en un equipo que aloja un sitio web puede ser el servidor.

Ejemplo: Un cliente (navegador) envía una solicitud *HTTP* al servidor; entonces el servidor devuelve una respuesta al cliente. La respuesta contiene información del estado de la solicitud y también puede contener el contenido solicitado.

Los dos métodos más comúnmente utilizados para una petición-respuesta entre un cliente y el servidor en el protocolo *HTTP* son: *GET* y *POST*.

- Una petición *GET* recupera datos de un servidor web mediante la especificación de parámetros en la parte correspondiente de la *URI* de la solicitud. Este es el principal método utilizado para la recuperación de documento estático.
- Una solicitud *POST* se utiliza para enviar datos al servidor, por ejemplo, información del cliente, carga de archivos, etc. usando formularios *HTML*.

Veamos algunas diferencias entre ellos.

- *GET* es idempotente. *POST* no lo es. Un método *HTTP* se dice que es idempotente si devuelve el mismo resultado cada vez. Debemos asegurarnos de que estos métodos devuelven siempre el mismo resultado. El método *HTTP POST* debemos utilizarlo para implementar algo que cambia con cada petición. Por ejemplo, para acceder a una página *HTML* o imagen, debemos utilizar *GET* porque siempre devolverá el mismo objeto, pero si tenemos que guardar la información del cliente en la base de datos, hay que utilizar el método *POST*.
- *GET* es el método *HTTP* predeterminado.
- Los hiperenlaces en una página utilizan el método *GET*.
- *GET* envía los datos como parte de la *URI* mientras que el método *POST* envía los datos como contenido *HTTP*. Los datos en peticiones *GET* se envían como la parte *query* en la *URI*.
- Los datos en una petición *GET* están limitados por la longitud máxima de la *URI* soportado por el navegador y el servidor web, mientras que la *POST* no tiene tales límites.
- El método *GET* no es seguro ya que los datos se exponen en la *URI*, podemos anotarlos y enviar la misma petición otra vez. *POST* es más seguro, ya que los datos se envían en el cuerpo de la solicitud aunque esto sin más no es suficientemente seguro.
- Una más diferencia más entre *GET* y *POST* es que la solicitud *GET* es marcable (*bookmarkable*), por ejemplo una búsqueda de Google, pero no se puede marcar una petición *POST*.

- Una solicitud GET también es almacenable (*cacheable*), mientras esto no es posible con peticiones POST.

### *URIs, controles y acciones*

Los controles son clases *Groovy* que gestionan las solicitudes que llegan a través de la web y crean o preparan la respuesta. Un controlador puede generar la respuesta directamente o delegar en una vista (que veremos más adelante). Para crear un controlador, basta con crear una clase cuyo *NombreController* en el directorio correspondiente. Es decir las clases que implementar controles tienen un identificador que acaba en *Controller*. Una acción es un método de un control y contiene el código necesario para crear o preparar la respuesta a partir de los datos aportados por la solicitud. La idea general es el una acción, es decir el método de un control, va a ser invocada cuando el navegador busque una determinada *URI*.

Las aplicaciones que desarrollemos en *Grails*, mientras las estamos desarrollando, tendrán *URIs* de la forma <http://localhost:8080/proyecto/control/accion/id?p1=v1&p2=v2>. Donde *proyecto* es el nombre del proyecto, *control* y *acción* son, respectivamente un control, cuyo identificador será *ControlController*, y una acción e *id*, que puede ser opcional, el identificador de un objeto. La parte tras el símbolo “?” es también opcional y está formada por un conjunto de pares clave valor que denominaremos parámetros de consulta (*query*).

Cada *URI* tiene asociada una acción dentro de un control. Esa acción (el método correspondiente) tiene disponibles como parámetros de entrada los siguientes variables:

- *id*: Especificado en la *URI*
- *request*: Es un instancia del tipo *HttpServletRequest* que permite almacenar información durante la solicitud actual
- *params*: Un *Map* que contiene los pares clave-valor incluidos en la *URI*
- *flash* : Un *Map* que sirve como almacén transitorio que está disponible entre una solicitud y la siguiente (veremos los detalles más adelante)
- Hay otras variables disponibles que veremos más adelante

Con la información de entrada el código es una acción tiene como objetivo:

- Construir un modelo (Un *Map* de pares clave-valor) para pasarlo a una vista que lo usará para construir un página
- Proporcionar una respuesta mediante el método *render*
- Redirigir a otra *URI* mediante el método *redirect* o *forward*.

La parte <http://localhost:8080/proyecto/> de la llamaremos *base* y define una referencia por defecto para construir una *URI* absoluta a partir de un control y una acción que pueden tomar valores por defecto. El valor por defecto de la acción es el método *index* o el nombre del método si existe sólo uno para un control. En muchos casos puede haber, también un valor por defecto del control que suele ser el control actual.

El método *redirect* tiene el formato y asumiendo que cada uno de los parámetros puede tener valores por defecto y por lo tanto no ser explícitos.



```
redirect(controller:"book",action:"list",id:4,params:[author:"King"])
```

El método *forward* tiene un comportamiento similar a *redirect* aunque existen diferencias.

```
forward controller:"book" action: "show", id: 4, params: [author:"King"]
```

El método *render* es de alguna de las formas siguientes:

```
render "texto"
render(view: "vista", model: [book: theShining])
render(template: "patron", bean: theShining)
```

Donde *view*, *template*, *bean*, *model* son elementos que veremos abajo.

En un control se puede limitar para cada acción el tipo de método de la solicitud HTTP mediante el método *allowedMethods* cuya sintaxis es:

```
allowedMethods = [action1:'POST',action3:['POST', 'DELETE']]
```

Si se intenta invocar una acción mediante un método HTTP no permitido se envía un error 405 (*Method Not Allowed*). Los detalles de su uso los veremos más adelante.

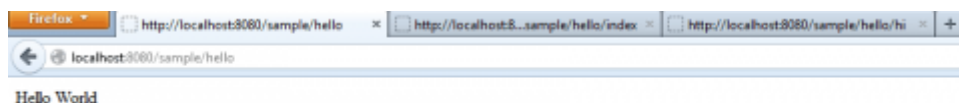
## Ejemplos

Los controles, como hemos dicho arriba, son clases Groovy que tomando los parámetros de entrada proporcionados hacen unas de las tareas que tienen como objetivo.

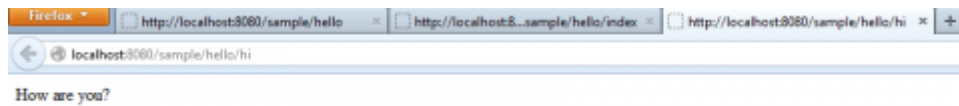
Asumiendo que nuestro proyecto se llama *sample* diseñamos el siguiente control:

```
class HelloController {
    def index() {
        render "Hello World"
    }
    def hi() {
        render "How are you?"
    }
}
```

Invocando la URL <http://localhost:8080/sample/hello> en un navegador aparece la pantalla:



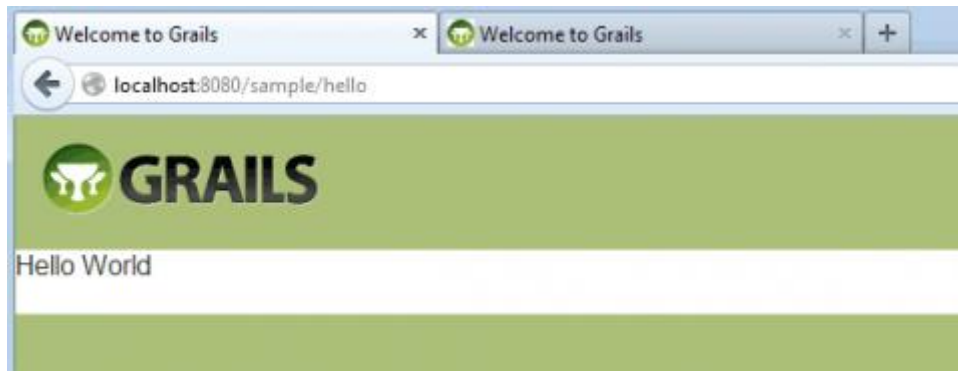
E invocando <http://localhost:8080/sample/hello/hi> obtendremos:



Si quitamos la llamada al método render invocamos implícitamente a una vista con el mismo nombre que la acción:

```
class HelloController {  
    def index() {  
    }  
    def hi() {  
    }  
}
```

Invocando la URL <http://localhost:8080/sample/hello> en un navegador aparece la pantalla:



E invocando <http://localhost:8080/sample/hello/hi> obtendremos:



Ahora es necesario crear las vistas *index.gsp* y *hi.gsp* que son dos ficheros que explicitan la forma e información que aparecerá en la página web.

Grails puede generar automáticamente un control por cada clase de dominio para hacer sobre ella las operaciones básicas (CRUD). Si asumimos la clase de dominio:

```
class Person {  
    String firstName
```

```

    String lastName
    Date dateOfBirth
    static constraints = {
        firstName (blank: false)
        lastName (blank: false)
        dateOfBirth (blank: false)
    }
}

```

El control generado es de la forma:

```

class PersonController {
    static allowedMethods=[save:"POST",update:"POST",delete:"POST"]
    def index() {
        redirect(action: "list", params: params)
    }
    def list(Integer max) {
        params.max = Math.min(max ?: 10, 100)
        [personInstanceList: Person.list(params),
         personInstanceTotal: Person.count()]
    }
    def create() {
        [personInstance: new Person(params)]
    }
    def save() {
        def personInstance = new Person(params)
        if (!personInstance.save(flush: true)) {
            render(view: "create",
                   model: [personInstance: personInstance]
                  )
            return
        }
        flash.message =
            message(code: 'default.created.message',
                   args: [message(code: 'person.label',
                                   default: 'Person'),
                        personInstance.id]
                  )
        redirect(action: "show", id: personInstance.id)
    }
    def show(Long id) {
        def personInstance = Person.get(id)
        if (!personInstance) {
            flash.message =
                message(code: 'default.not.found.message',
                       args: [message(code: 'person.label',
                                       default: 'Person'),
                            id]
                      )
            redirect(action: "list")
            return
        }
        [personInstance: personInstance]
    }
    def edit(Long id) {
        def personInstance = Person.get(id)
    }
}

```

```

        if (!personInstance) {
            flash.message =
                message(code: 'default.not.found.message',
                        args: [message(code: 'person.label',
                                      default: 'Person'),
                              id]
                        )
            redirect(action: "list")
            return
        }
        [personInstance: personInstance]
    }
    ...
}

```

El código tiene los siguientes comentarios:

- *Allowed Methods*: Restringe si GET o POST son permitidos para las acciones correspondientes. Así *static allowedMethods = [save: "POST", update: "POST", delete: "POST"]* significa que solamente HTTP POST se permite para acciones *save*, *update*, y *delete* (las últimas no explicitadas). GET se permite para estas acciones. Pero para el resto de las acciones, que no se indica nada, se permiten tanto POST como GET.
- *index*: Redirige a la acción *list*, pasándole los parámetros que había recibido.
- *list*: Recupera con el método adecuado la población del tipo (*Person.list(params)*) y el número de la mismas (*Person.count()*) y con ambas informaciones construye un modelo que pasa a la vista *list.gsp*. El método *list* acepta, entre otros los parámetros *max* y *offset*. Esto le permite recuperar un número *max* de elementos de la población a partir del *offset* especificado. Con este mecanismo es posible dividir una lista larga en trozos de tamaño *max* representables en una página. Con el código *Math.min(max?:10,100)* se establece un valor por defecto para *max* de 10 y un máximo de 100.
- *create*: Construye un modelo formado por una instancia del tipo *Person* y lo pasa a la vista correspondiente *create.gsp*.
- *save*: Con los parámetros recibidos en *params* construye una nueva instancia y la intenta guardar. Si se cumple el invariante, redirige a la acción *show* pasándole el *id* de la instancia. Si no se cumple el invariante muestra la vista *create.gsp* con los valores de la instancia.
- *show*: Recupera la instancia cuya identidad se pasa como parámetro. Si la instancia no se encuentra se redirige a la acción *list*.
- *edit*: Recupera la instancia cuya identidad se pasa como parámetro y se la pasa a la vista *edit.gsp*. Si la instancia no se encuentra se redirige a la acción *list*.

### Paso de parámetros a las acciones de un control

Para comprender el paso de parámetros entre las llamadas a acciones hay que tener en cuenta que éstas (métodos de los controles) pueden tener parámetros definidos por posición. Como sabemos, los parámetros a una acción se pasan en la *query* de la URI y se capturan en la variable *params*. Cuando redirigimos a otra acción, en el método *redirect*, indicamos los elementos del URI: control, acción y *params*. Si la acción tiene parámetros definidos por

posición entonces los valores, asociados a claves en *params* que coincidan con nombres de parámetros por posición, se convierte automáticamente al tipo especificado en la definición del método. Si la conversión del tipo de falla, entonces el parámetro será *null*. Así por ejemplo si teniendo el control:

```
class NuevoController {
    def sample(final String author, final Long id, final String book) {
        render "Params: author = $author, book= $book, id = $id"
    }
}
```

Si se invoca la URI siguiente se obtiene el resultado que se muestra.

<http://localhost:8080/proyecto/nuevo/sample?id=100&book=It&author=Stephen%20King>

```
Params: name= Stephen King, book = It, id = 100
```

Esto explica que si invocamos el control *person* con la acción por defecto (*index*) y con la *query* *?max=5* entonces este valor se captura en la variable *params* que se pasa a la acción *list* y ésta recibe el valor en el parámetro por posición *max*.

Junto con *params* tenemos también la posibilidad de usar la variable *flash* que también es un *Map* de pares clave-valor que pueden ser compartidos entre la petición actual y la siguiente. Después se borran. Es útil para configurar un mensaje directamente antes de redirigir a otra acción, por ejemplo. Para construir mensajes usamos el método *message*, que veremos con más detalle abajo, que tiene los siguientes parámetros:

- *code*: El código para construir el mensaje
- *default*: El valor por defecto si *code* no puede ser encontrado en *messages.properties*.
- *args*: Una lista de argumentos para construir el mensaje

Hay otras variables con un tiempo de vida más largo que veremos más adelante.

## Vistas

### Visión general

Las vistas son el mecanismo para generar las páginas web que vamos necesitando. Las vistas son ficheros GSP (*Groovy Servers Pages*).

Un fichero GSP usa un modelo, que es un *Map* de pares clave-valor que se pasa desde un control, y es típicamente un fichero *html* con etiquetas *GSP*. Veamos algunos ejemplos:

Partimos de la clase de dominio *Person*:

```
class Person {
    String firstName
    String lastName
    int age
}
```

Y el control *PersonController*:

```
class PersonController {
  def index() {
    redirect(action: "list", params:params)
  }
  def showList() {
    def list = []
    list << new Person(firstName: 'John', lastName:'Doe', age:50)
    list << new Person(firstName: 'Jane', lastName:'Smith', age:45)
    list << new Person(firstName: 'Sam', lastName:'Robinson', age:47)
    render(view:"list", model:[personInstanceList:list])
  }
  def list(Integer max){
    params.max = Math.min(max ?: 10, 100)
    [personInstanceList: Person.list(params),
     personInstanceTotal: Person.count()]
  }
  def create() {
    [personInstance: new Person(params)]
  }
  def show(long id) {
    def personInstance = Person.get(id)
    if (!personInstance) {
      redirect(action: "list")
      return
    }
    [personInstance: personInstance, ]
  }
  def save() {
    ...
  }
}
```

Y la vista *show.gsp*:

```
<!DOCTYPE html>
<html>
<head>
  <meta name="layout" content="main"/>
  <title>Render Domain</title>
</head>
<body>
Last Name: ${person.lastName} <br/>
First Name: ${person.firstName} <br/>
Age: ${person.age} <br/>
</body>
</html>
```

El resultado obtenido tiene un aspecto similar al que se ve a continuación.



Last Name: "Perez"  
First Name: "Antonio"  
Age: "22"

Vista *list.gsp*:

```
<!DOCTYPE html>
<html>
<head>
  <meta name="layout" content="main"/>
  <title>Render Domain</title>
</head>
<body>
  <table>
    <thead>
      <tr>
        <th>Name</th>
        <th>Age</th>
      </tr>
    </thead>
    <g:each in="${list}" var="person">
      <tr>
        <td>${person.lastName}, ${person.firstName}</td>
        <td>${person.age}</td>
      </tr>
    </g:each>
  </table>
</body>
</html>
```

El resultado es:



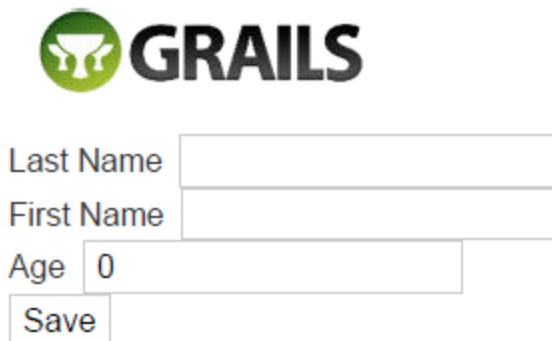
---

Name	Age
Doe, John	50
Smith, Jane	45
Robinson, Sam	47

Vista *create.gsp*:

```
<!DOCTYPE html>
<html>
<head>
  <meta name="layout" content="main"/>
  <title>Render Domain</title>
</head>
<body>
<g:form action="save" >
  <label for="lastName">Last Name</label>
  <g:textField name="lastName" value="${person.lastName}"/>
  <br/>
  <label for="firstName">First Name</label>
  <g:textField name="firstName" value="${person.firstName}"/>
  <br/>
  <label for="age">Age</label>
  <g:textField name="age" value="${person.age}"/>
  <br/>
  <g:submitButton name="create" value="Save" />
</g:form>
</body>
</html>
```

El resultado es:



The screenshot shows the Grails logo at the top. Below it is a form with the following elements:

- A label "Last Name" followed by a text input field.
- A label "First Name" followed by a text input field.
- A label "Age" followed by a text input field containing the value "0".
- A "Save" button.

Como podemos observar los resultados son pobres. No está incluida la navegación, en el formulario es posible presentar mejor los diferentes campos de entrada, no están tratados los posibles errores de validación, etc.

### *Estructura de un fichero gsp*

Si nos fijamos en los ficheros anteriores vemos que tienen su estructura tiene el formato de un documento *html* con un conjunto de etiquetas *gsp*. Si quitamos las etiquetas *gsp* en el fichero *list.gsp* obtenemos:

```
<!DOCTYPE html>
```



```

<html>
<head>
  <meta name="layout" content="main"/>
  <title>Render Domain</title>
</head>
<body>
  <table>
    <tr>
      <th>Name</th>
      <th>Age</th>
    </tr>
    <tr>
      <td>${person.lastName}, ${person.firstName}</td>
      <td>${person.age}</td>
    </tr>
  </table>
</body>
</html>

```

El documento tiene una cabecera y un cuerpo. En la cabecera se indica el título y se incluye un fichero con detalles importantes para la página. Veremos más adelante los detalles de es fichero que usualmente se denomina *main.gsp*.

El resto son etiquetas *html*. Un conjunto de etiquetas *html* son:

- `<table> ... </table>`: Presenta un tabla de datos
- `<tr> ... </tr>`: Presenta una fila de datos
- `<td> ... </td>`: Una celda de datos
- `<th> ... </th>`: Una celda de datos de la cabera de la tabla
- `<thead> ... </thead>`: Agrupa la cabecera en una tabla
- `<tbody> ... </tbody>`: Agrupa el cuerpo en una tabla
- `<tfoot> ... </tfoot>`: Agrupa el pie en una tabla
- `<br/>`: Salto de línea
- `<label for="name">Name</label>`: Un etiqueta de nombre *Name* para el elemento con identificador *name*.
- `<h1> ... </h1>`: Encabezado de nivel 1. Existen hasta nivel 6.
- `<div> ... </div>`: Una sección en un documento
- `<spam> ... </spam>`: Permite agrupar varios elementos en seguidos dentro de un mismo bloque
- `<p> ... </p>`: Un párrafo
- `<ul> ... </ul>`: Una lista
- `<li> ... </li>`: Elemento de una lista
- `<dl> ... </dl>`: Una lista de definiciones
- `<dt> ... </dt>`: Una término en una lista de definiciones
- `<dd> ... </dd>`: Una definición en una lista de definiciones
- `<a href="http://www.w3schools.com">Visit W3Schools.com!</a>`: Enlaces
- `<fieldset> ... </fieldset>`: Un grupo de elementos relacionados
- ``: Inserción de imágenes

- `<meta name="layout" content="main">`: Para incluir información en la página. Con los detalles que se muestran, información de *layout* contenida en el fichero *main.gsp*.

Con esas etiquetas se compone un documento *html* que, una vez procesado por el navegador, se convertirá en la página que vemos. No daremos más detalles sobre *html* que pueden ser encontrados en documentos más especializados.

Como se puede ver la estructura de un documento está compuesto de un conjunto de elementos que pueden estar anidados. Un elemento está delimitado por una etiqueta de inicio y otra de fin, por ejemplo `<div> ... </div>`, y puede tener atributos definidos como un conjunto de pares clave-valor. Entre los atributos más usuales tenemos:

- *class*: Especifica uno o más nombres de las clases para un elemento (Todos los elementos con la misma clase tendrán un mismo estilo que se tomará de una hoja de estilo)
- *id*: Especifica para un elemento un identificador único en toda la página
- *lang*: Especifica el lenguaje del contenido del elemento
- *role*: Especifica información estructural para el elemento. Algunos de sus valores son: *main* (contenido principal), *navigation* (barra de navegación), etc.

Junto a *html* necesitamos unos ficheros adicionales, los denominados hojas de estilo (*CSS*, acrónimo de *Cascading Style Sheets*) que añaden información de presentación a los sitios web. Estos especifican cuestiones como fuentes, colores, márgenes, líneas, altura, anchura, imágenes de fondo, posicionamiento avanzado de los elementos de *html* y muchos otros temas. En definitiva *html* se usa para estructurar el contenido, *css* para formatearlo.

Un documento **CSS** es un conjunto de declaraciones de la forma:

```
selector {
    property: value;
    ...
}
```

Donde selector es una etiqueta que indica a que elemento del documento *html* se aplica la definición. Una definición es un conjunto de pares clave-valor donde las claves son nombre de propiedades. Entre las propiedades posibles están: *color*, *background-color*, *font*, *text-indent*, *text-align*, etc.

En la mayoría de los casos queremos asignar una definición a un conjunto de elementos. Esto lo hacemos refiriéndonos a aquellos elementos que tengan asociado un valor al atributo *class*, a aquel elemento que tenga un *id* concreto, en la forma:

```
a {
    color: blue;
}
a.whitewine {
    color: #FFBB00;
}
a.redwine {
```

```
        color: #800000;
    }
    #c1-2 {
        color: red;
    }
```

En la primera declaración se asigna una definición a los elementos de tipo `a` (los enlaces), en la segunda a los enlaces con clase *whitewine*, en la tercera a los enlaces con clase *redwine* y en la cuarta al elemento con identificador *c1-2*.

Los elementos de *html* `<span>` y `<div>` se usan para agrupar y estructurar un documento, y junto con los atributos *class* e *id* permiten dar formato a un bloque.

Como en *html* los elementos pueden estar anidados con el modelo de caja *CSS* describe las cajas que se generan a partir de los mismos. El modelo de caja también contiene opciones detalladas en lo referente a la posición de cada elemento, al ajuste de márgenes, bordes, relleno (*padding*) y contenido de cada elemento.

Para posicionar un elemento de forma absoluta, la propiedad *position* se establece como *absolute*. Posteriormente puedes usar las propiedades *left*, *right*, *top*, y *bottom* para colocarlo dentro los vértices de la caja. Para posicionar un elemento de forma relativa, la propiedad *position* se establece como *relative*. Los elementos se pueden hacer flotar a la derecha o a la izquierda dando valores a la propiedad *float* (que puede tomar los valores *left*, *right*, o *none*).

En un proyecto *grails* los detalles de estilo vienen definidos en el fichero *main.gsp* por defecto. En él se definen no solamente los estilos (en un fichero incluido *main.css*), también la estructura fija de cada página y las imágenes que contenga. Cambiar este fichero supone cambiar el aspecto exterior de la aplicación web. O ajustarlo a distintos dispositivos como móviles, etc. Es un tema complejo en el que no entraremos en esta introducción.

## Etiquetas *gsp*

Hay un conjunto de etiquetas *gsp*, de las que solo presentaremos su sintaxis y cuya semántica puede consultarse en la documentación, entre las que tenemos:

- Declara la variable *now* con ámbito toda la página y le da valor

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

- Estructuras de control y paginación

```
<g:if test= "${session.role == 'admin'}">
    ...
</g:if>
<g:else>
    ...
</g:else>
```

```

<g:each in="\${[1,2,3]}" var="num">
  <p>Number \${num}</p>
</g:each>
<g:set var="num" value="\${1}" />
<g:while test="\${num < 5 }">
  <p>Number \${num++}</p>
</g:while>

<g:paginate controller="book" action="list" total="\${bookCount}" />

```

- Enlaces a acciones al pulsar un determinado texto y generadores de link

```

<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:createLink action="show" params="[foo: 'bar', boo: 'far']"/>

```

- Formularios: Presenta un formulario con un nombre dado que cuando está terminado se envía a la acción indicada o a una por defecto.

```

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

```

- Elementos de un formulario

```

<g:textField name="myField" value="\${myValue}" />
<g:field type="password" name="password" value="\${userInstance?.password}"/>
<g:datePicker name="myDate" value="\${myDomainClass?.myDateField}"
  default="\${new Date().plus(7)}"/>
<g:passwordField name="myPasswordField" value="\${myPassword}" />
<g:submitButton name="update" value="Update" />
<g:actionSubmit value="Delete" action="Delete"
  onclick="return confirm('Are you sure??')"/>

```

- Tablas

```

<g:sortableColumn property="title" title="Title" />

```

- Beans

```

<g:fieldValue bean="\${book}" field="title" />

```

- Formatos

```

<g:formatDate format="yyyy-MM-dd" date="\${date}"/>
<g:formatNumber number="\${myNumber}" format="\$###,##0" />
<g:formatBoolean boolean="\${myBoolean}" true="True!" false="False!" />

```

## Mensajes y Gestión de Errores

En *Grails* se *no* se valida una restricción se busca un mensaje con un código de la forma:

```
[Class Name].[Property Name].[Constraint Code]
```

Si en la clase de dominio *User* la restricción *blank* no se valida el mensaje asociado tiene el código *user.login.blank*. En el fichero *grails-app/i18n/messages.properties* hay definidos un buen número de mensajes asociados a sus códigos. Este en particular es de la forma:

```
user.login.blank=Your login name must be specified!
```

El mensaje asociado a un código dado puede tener parámetros como por ejemplo:

```
book.delete.message="Book {0} deleted."
```

Para construir el mensaje concreto se deben aportar los parámetros necesarios. Normalmente la información necesaria para construir un mensaje se pasa a través de la variable *flash*. En esta variable se añaden pares clave-valor en alguna de las formas posibles por ejemplo:

```
flash.message = "book.delete.message"
flash.args = ["The Stand"]
flash.default = "book deleted"
```

Con esa información, y usando la etiqueta `<g:message .../>`, podemos construir el mensaje. Esa etiqueta tiene como parámetros:

- *code*: El código del mensaje
- *default*: El mensaje opcional si el código no se puede encontrar en el fichero *messages.properties*.
- *args*: Una lista de parámetros
- *locale*: Indica los detalles locales

El mensaje se construye entonces de la forma:

```
<g:message code="{flash.message}" args="{flash.args}"
           default="{flash.default}"/>
```

Con el resultado: *"Book The Stand deleted."*

Con *Grails* es posible personalizar el texto que aparece en una vista basándose en la configuración regional del usuario. Sigue el estándar (i18n) y usa objetos que representan una región geográfica, política o cultural específico. Estos objetos se designan con un código de

idioma y un código de país. Por ejemplo, "en\_US" es el código para inglés de EE.UU, mientras que "en\_ES" es el código de inglés británico.

Por cada configuración local se crea un archivo con nombre *message\_lo.properties*. Donde *lo* es el código de la configuración correspondiente. Por ejemplo *messages\_en\_GB.properties* de inglés británico. Ese archivo contiene un grupo de mensajes asociados a sus códigos y está en el directorio *grails-app / i18n*. Es un archivo de propiedades Java. Por defecto se busca en *messages.properties* a menos que el usuario haya especificado una configuración local.

La configuración local se puede cambiar como el parámetro *lang* de la *URI*

```
/book/list?lang=es
```

O con el parámetro *locale* de la etiqueta *<g:message .../>*.

Cuando validamos una instancia, posiblemente al almacenarla, pueden producirse errores. Una forma de construir mensajes explicativos de los mismos es construir mensajes con la etiqueta *<g:message .../>*, si la usamos en una vista, o el método equivalente *g.message(...)*, si lo usamos en un control. El texto producido puede guardarse en la variable flash en un control y recuperarlo en la vista. Por ejemplo:

```
flash.message = message(code: 'default.not.found.message', args:
                        [message(code: 'person.label', default: 'Person'),
                         id])
```

Los mensajes por defecto asociados a los códigos son:

```
default.not.found.message={0} not found with id {1}
default.show.label=Show {0}
```

El código *'person.label'* no tiene mensaje por defecto asociado. Si lo estimamos oportuno incluimos el mensaje en el fichero o ficheros que corresponda.

Y en la vista el mensaje se recupera de la forma:

```
h1><g:message code="default.show.label" args="[entityName]" /></h1>
<g:if test="${flash.message}">
<div class="message" role="status">${flash.message}</div>
</g:if>
```

### *Ejemplo de fichero gsp*

Como ejemplo veamos un fichero completo

```
<%@ page import="asia.grails.test.scaff.Person" %>
<!DOCTYPE html>
<html>
  <head>
```

```

<meta name="layout" content="main">
<g:set var="entityName" value="${message(code: 'person.label',
                                     default: 'Person'))}"
/>
<title><g:message code="default.list.label" args="[entityName]" />
</title>
</head>
<body>
<a href="#list-person" class="skip" tabindex="-1">
  <g:message code="default.link.skip.label"
    default="Skip to content..." />
</a>
<div class="nav" role="navigation">
  <ul>
    <li><a class="home" href="${createLink(uri: '/')}">
      <g:message code="default.home.label" />
    </a>
    </li>
    <li><g:link class="create" action="create">
      <g:message code="default.new.label"
        args="[entityName]" />
    </g:link>
    </li>
  </ul>
</div>
<div id="list-person" class="content scaffold-list" role="main">
  <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>
  <table>
    <thead>
      <tr>
        <g:sortableColumn property="firstName"
          title="${message(code:
                                'person.firstName.label',
                                default: 'First Name'))}"
        />
        <g:sortableColumn property="lastName"
          title="${message(code:
                                'person.lastName.label',
                                default: 'Last Name'))}"
        />
        <g:sortableColumn property="dateOfBirth"
          title="${message(code:
                                'person.dateOfBirth.label',
                                default: 'Date Of Birth'))}"
        />
      </tr>
    </thead>
    <tbody>
      <g:each in="${personInstanceList}" status="i"
        var="personInstance">
        <tr class="${(i % 2) == 0 ? 'even' : 'odd'}">
          <td><g:link action="show" id="${personInstance.id}">
            ${fieldValue(bean: personInstance,
                          field: "firstName")}
          </g:link>
          </td>
          <td>${fieldValue(bean: personInstance,
                          field: "lastName")}
          </td>
          <td><g:formatDate
            date="${personInstance.dateOfBirth}" />
          </td>
        </tr>
      </g:each>
    </tbody>
  </table>

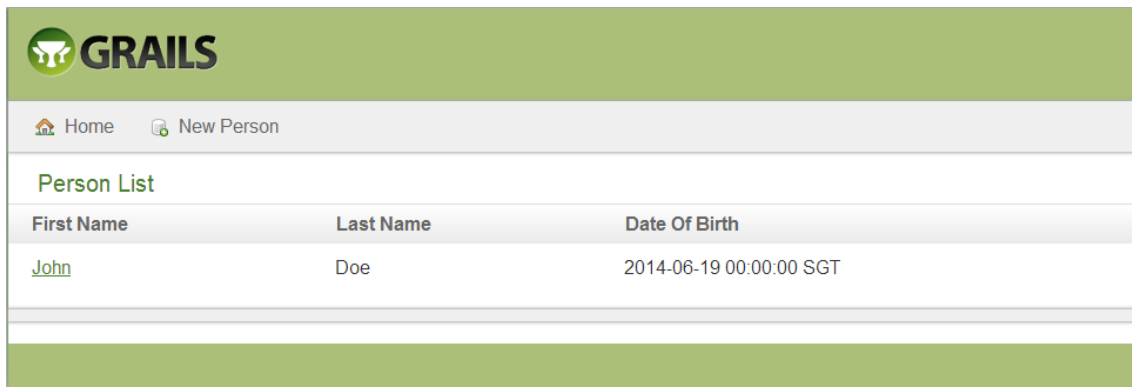
```

```

        </g:each>
    </tbody>
</table>
<div class="pagination">
    <g:paginate total="{personInstanceTotal}" />
</div>
</div>
</body>
</html>

```

Que, asumiendo la acción *list* en el control *PersonController* visto arriba, daría lugar a una página de este tipo:



First Name	Last Name	Date Of Birth
<a href="#">John</a>	Doe	2014-06-19 00:00:00 SGT

Veamos los detalles

## Configuración

*Grails* puede ser configurado para ajustarse a diversas necesidades. Sólo veremos algunas ideas aquí. *Grails* tiene el concepto de configuración definido por cada entorno. Los archivos *Config.groovy*, *DataSource.groovy*, *BootStrap.groovy* y *BuidConfig.groovy*, ubicados en el directorio *grails-app /conf* pueden ser editados para introducir detalles de configuración. La configuración depende del entorno escogido. Hay tres entornos posibles que corresponden con tres momentos del software: desarrollo (*development*), pruebas (*test*), producción (*production*). Es posible cambiar de un entorno a otro en los ficheros de configuración. Sólo daremos aquí algunos detalles para el primero.

Para cada uno de ellos se puede definir un almacén de datos y unos datos iniciales, además de otros detalles de configuración.

### *Datos iniciales*

En la clase *BootStrap.groovy* añadir líneas en el método *init* creando las instancias que consideremos y guardarlas.



```
def init = { servletContext ->
    def radiology = new Speciality(name: 'radiology').save(failOnError: true)
    def surgery = new Speciality(name: 'surgery').save(failOnError: true)
    ...
}
```

Esas instancias, si no tienen errores, se crearán y guardarán y formarán parte de la población inicial.

### *Conexión a bases de datos*

En el modo de desarrollo Grails usa una base de datos interna que viene proporcionada por la aplicación. Pero con la configuración por defecto no se mantienen los datos de una sesión a otra. Una solución sencilla es usar MySQL como base de datos. Para ello se requiere añadir a las librerías del proyecto un archivo *.jar* que contiene el driver JDBC para *MySQL*, que se puede obtener de:

<http://dev.mysql.com/downloads/connector/j/>

Incluir en la sección dependencias del fichero *BuidConfig.groovy* la línea:

```
dependencies {

    runtime 'mysql:mysql-connector-java:5.1.29'

}
```

Añadir la fuente de datos al fichero *DataSource.groovy* en sustitución de otra declaración del mismo tipo y en el entorno adecuado. Previamente habrá que crear la base de datos *nombre* en el entorno de MySQL y dar autorización al usuario adecuado con su *password* correspondiente.

```
dataSource{
    pooled = true
    dbCreate = "update"
    url = "jdbc:mysql://localhost/nombre"
    driverClassName = "com.mysql.jdbc.Driver"
    dialect = org.hibernate.dialect.MySQL5InnoDBDialect
    username = "----"
    password = "----"
}
```

De forma similar se podrían conectar otras bases de datos o, incluso, una distinta para cada entorno.