



GRAILS

Groovy & Grails: Desarrollo rápido de aplicaciones

Sesión 9: Dominios y servicios (I)



Dominios y servicios (I)

- GORM (Grails Object Relational Mapping)
- Validación



GORM

- Creación de dominios
- Relaciones entre clases de dominio
- Aspectos avanzados de GORM



GORM

- En algunos frameworks el término *dominio* se sustituye por *modelo*
- Ambos se refieren a que los objetos de la aplicación sean persistidos contra una base de datos
- GORM tiene por debajo Hibernate 3
- GORM facilita el trabajo en términos de usabilidad



Creación de dominios

```
grails create-domain-class Usuario
```

- ¿Qué pasa cuando creamos una *clase de dominio*?



Creación de dominios

- Con la siguiente clase de dominio

```
class Operacion {  
    String tipo  
    Boolean estado  
    Date fechaInicio  
    Date fechaFin  
    Usuario usuario  
    Libro libro  
  
    static belongsTo = [Usuario, Libro]  
}
```



Creación de dominios

- Se generará la siguiente tabla en la base de datos

Campo	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Extra
<u>id</u>	bigint(20)			No		auto_increment
version	bigint(20)			No		
estado	bit(1)			No		
fecha_fin	datetime			No		
fecha_inicio	datetime			No		
libro_id	bigint(20)			No		
tipo	varchar(8)	latin1_swedish_ci		No		
usuario_id	bigint(20)			No		



Creación de dominios

- **Nuevas columnas**
 - *ID* (clave primaria)
 - *version* (integridad transaccional)



Creación de dominios

- **Nombres de las columnas**
 - Clases de dominio con convenio *Camel/Case*
 - Tablas con convenio *snake_case*



Creación de dominios

- **Claves ajenas**
 - Nombre de la clase/tabla referencia seguido de *_id*



Creación de dominios

- **Tipos de datos**
 - Depende de la base de datos utilizada
 - *String* ↔ *varchar()*
 - *Date* ↔ *Datetime*



Creación de dominios

- **Valores por defecto**
 - Se pueden definir en la propia clase de dominio

String tipo = “prestamo”



Creación de dominios

- **Ficheros**

- Para almacenar ficheros en la base de datos la propiedad de la clase de dominio se define como *byte[]*
- En la tabla se crea una columna de tipo *tinyblob*



Relaciones entre clases de dominio

- Uno a uno
- Uno a muchos
- Muchos a uno
- Muchos a muchos



Relaciones entre clases de dominio

- Las relaciones se definen con las palabras reservadas
 - *hasMany*
 - *belongsTo*
 - *hasOne*



Relaciones entre clases de dominio

- **Uno a uno**
 - Un objeto de la clase A está únicamente relacionado con un objeto de la clase B

```
class Libro{
    ....
    OperacionActiva operact
    ....
}
class OperacionActiva{
    ....
    Libro libro
    ....
}
```




Relaciones entre clases de dominio

- **Uno a uno**
 - Inserciones y borrados en cascada.

```
class Libro{
    ...
    OperacionActiva operact
    ...
}
class OperacionActiva{
    ...
    static belongsTo = [operact:OperacionActiva]
    ...
}
```



Relaciones entre clases de dominio

- **Uno a uno**

```
class Libro{
    ...
    statichasOne = [operact:OperacionActiva]
    ...
}
class OperacionActiva{
    ...
    Libro libro
    ...
}
```



Relaciones entre clases de dominio

- **Uno a muchos / Muchos a uno**
 - 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 A

```
class Usuario{
    ...
    static hasMany = [operaciones:Operacion]
}
class Operacion {
    ...
    Usuario usuario
}
```



Relaciones entre clases de dominio

- **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

```
class Alumno{
    static hasMany = [asignaturas:Asignatura]
}
class Asignatura{
    static belongsTo = Alumno
    static hasMany = [alumnos:Alumno]
}
```



Aspectos avanzados en GORM

- Ajustes de mapeado
- Herencia de clases
- Habilitando la caché
- Propiedades transitorias
- Eventos GORM



Ajustes de mapeado

- En ocasiones es necesario seguir una serie de criterios a la hora de crear la base de datos y debemos modificar el comportamiento de Grails
- Grails utiliza un *closure* llamado *mapping* donde se define dicho comportamiento

```
static mapping = {  
    //Todo el mapeado de la tabla aquí  
}
```



Ajustes de mapeado

- **Nombres de las tablas y las columnas**

```
static mapping = {  
    table 'tbl_usuario'  
    columns {  
        login column:'username'  
        password column:'passwd'  
    }  
}
```



Ajustes de mapeado

- **Deshabilitar el campo *version***

```
static mapping = {  
    ....  
    version false  
    ....  
}
```




Ajustes de mapeado

- **Carga perezosa de los datos**
 - Los datos no se cargan en memoria hasta que no sean solicitados
 - Para cambiar este comportamiento y cargar los datos en memoria

```
static mapping = {  
    columns {  
        operaciones lazy:false  
    }  
}
```



Herencia de clases

- Simplemente hay que *extend* la clase

```
class Usuario{ .... }  
  
class Administrador extends Usuario{ .... }  
  
class Bibliotecario extends Usuario{ .... }  
  
class Profesor extends Usuario{ .... }  
  
class Socio extends Usuario{ .... }
```



Herencia de clases

- Grails almacena todos los datos en una única tabla
- Añadiría un campo *class* para distinguir el tipo de instancia creada
- Podemos optar por tener una tabla por cada tipo de usuario



Herencia de clases

```
class Usuario{ .... }
class Administrador extends Usuario{
    static mapping = { table 'administrador' }
}
class Bibliotecario extends Usuario{
    static mapping = { table 'bibliotecario' }
}
class Profesor extends Usuario{
    static mapping = { table 'profesor' }
}
class Socio extends Usuario{
    static mapping = { table 'socio' }
}
```



Herencia de clases

- También se puede especificar en la clase padre la propiedad *tablePerHierarchy*

```
class Usuario{  
    ....  
    static mapping = {  
        tablePerHierarchy false  
    }  
}
```



Habilitando la caché

- Hibernate cuenta con una caché de segundo nivel, que almacena los datos asociados a una clase de dominio
- Debemos editar el archivo *grails-app/conf/DataSource.groovy*

```
hibernate {  
    cache.use_second_level_cache=true  
    cache.use_query_cache=true  
    cache.provider_class =  
    'com.opensymphony.oscache.hibernate.OSCacheProvider'  
}
```



Habilitando la caché

- Y en cada clase que deseemos cachear

```
class Usuario{  
    ....  
    static mapping = {  
        nombre cache:true  
    }  
}
```



Propiedades transitorias

- Por defecto, todas las propiedades de una clase de dominio son persistidas en la base de datos
- Para no persistir determinadas propiedades, debemos utilizar la propiedad *transients*

```
class Usuario {  
    static transients = ["confirmarPassword"]  
    String login  
    String password  
    String confirmarPassword  
    String nombre  
    String apellidos  
}
```




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
- Los métodos son *beforeInsert()* y *beforeUpdate()*
- Un ejemplo podría ser indicar las fechas de creación y última modificación de un determinado registro



Eventos GORM

- Un ejemplo podría ser indicar las fechas de creación y última modificación de un determinado registro

```
def beforeInsert() = {  
    fechaCreacion = new Date()  
    fechaUltimaModificacion = new Date()  
}  
  
def beforeUpdate() = {  
    fechaUltimaModificacion = new Date()  
}
```



Eventos GORM

- Grails dispone también de los siguientes métodos:
 - *beforeDelete()*
 - *afterInsert()*
 - *afterUpdate()*
 - *afterDelete()*
 - *onLoad()*



Validación

- Restricciones predefinidas en GORM
- Construir tus propias restricciones
- Mensajes de errores de las restricciones



Validación

- Podemos imponer restricciones a las propiedades de las clases
- En caso de que no se cumplan, el objeto no será persistido en la base de datos y se mostrará el error causante del problema
- El método `save()` se encarga de realizar estas comprobaciones



Validación

- Las restricciones también nos servirán para imponer determinadas características a las tablas
- Podemos indicar que una propiedad de una clase de dominio puede contener el valor *null*



Restricciones predefinidas en GORM

- GORM dispone de una serie de funciones predefinidas
- Hemos visto hasta ahora unas cuantas como *size()*, *unique()*, *blank()* o *inList()*



Restricciones predefinidas en GORM

```
class Usuario {
    String login
    String password
    String nombre
    String apellidos
    String tipo
    static constraints = {
        login(size:6..20, blank:false, unique:true)
        password(size:6..20, blank:false,password:true)
        nombre(blank:false)
        apellidos(blank:false)
        tipo(inList:["administrador", "bibliotecario", "profesor", "socio"])
    }
    ....
}
```




Restricciones predefinidas en GORM

Nombre	Ejemplo
<i>blank</i>	login(blank:false)
<i>creditCard</i>	tarjetaCredito(creditCard:true)
<i>email</i>	correoElectronico(email:true)
<i>password</i>	contrasenya(password:true)
<i>inList</i>	tipo(inList:["administrador","bibliotecario"])
<i>matches</i>	login(matches:"[a-zA-Z]+")
<i>max</i>	price(max:999F)
<i>min</i>	price(min:0F)
<i>minSize</i>	hijos(minSize:5)
<i>maxSize</i>	hijos(maxSize:15)



Restricciones predefinidas en GORM

Nombre	Ejemplo
<i>notEqual</i>	login(notEqual:"admin")
<i>nullable</i>	edad(nullable:true)
<i>range</i>	edad(range:0..99)
<i>scale</i>	salario(scale:2)
<i>size</i>	login(size:5..15)
<i>unique</i>	login(unique:true)
<i>url</i>	website(url:true)



Construir tus propias restricciones

- Grails no incluye todas las posibles restricciones que nos podemos encontrar en una aplicación
- Tenemos la posibilidad de crear nuestras propias restricciones
- Lo podemos hacer mediante un closure llamado *validator*
- Vamos a comprobar que las *fechas inicio y fin* no sean anteriores a la fecha actual



Construir tus propias restricciones

```
package biblioteca
class Operación {
    ....
    static constraints = {
        ....
        fechaInicio(nullable:false,
            validator: {
                if (it?.compareTo(new Date()) < 0)
                    return false
                return true
            }
        ) ....
    }
    ....
}
```



Construir tus propias restricciones

- El closure *validator* debe devolver *true* si la validación se efectúa correctamente y *false* en caso contrario
- Podemos utilizar la variable *it* para conocer el valor introducido y comprobar su validez
- Introduzcamos otra restricción que compruebe que la *fecha fin* no sea anterior a la *fecha inicio*



Construir tus propias restricciones

```
fechaFin(nullable:false,  
    validator: {  
        val, obj ->  
            if (val != null){  
                return val.after(obj.fechaInicio)  
            }  
        }  
    }  
)
```



Construir tus propias restricciones

- La variable *val* se refiere al valor de la variable (*fechaFin*)
- La variable *obj* se refiere al nuevo objeto que estamos creando (Operacion)
- Podemos incluso utilizar consultas a la base de datos



Construir tus propias restricciones

- La siguiente restricción evita que un usuario tenga más de un determinado número de préstamos

```
tipo(inList:["prestamo", "reserva"],
    validator: {
        val, obj ->
            if ((val=="prestamo") &&
                (Operacion.findAllByTipoAndUsuario(val,obj.usuario).size() > 5))
                return false
            return true
        }
    )
```




Mensajes de error de las restricciones

- Los mensajes de error producidos no son lo más deseable para un usuario final
- Para cambiar estos mensajes es necesario entender el sistema de mensajes que utiliza Grails
- Cuando no escribimos el nombre de un usuario, recibimos un mensaje de error del tipo *La propiedad [nombre] de la clase [class Usuario] no puede ser vacía*



Mensajes de error de las restricciones

- Grails implementa un sistema jerárquico para los mensajes de error basado en varios aspectos
 - Clase de dominio
 - Propiedad
 - Tipo de validación



Mensajes de error de las restricciones

- La jerarquía que Grails busca en este error será

```
usuario.nombre.blank.error.nombre  
usuario.nombre.blank.error.java.lang.String  
usuario.nombre.blank.error  
usuario.nombre.blank.nombre  
usuario.nombre.blank.java.lang.String  
usuario.nombre.blank  
blank.nombre  
blank.java.lang.String  
blank
```



Mensajes de error de las restricciones

- Este sería el orden en el que Grails buscaría en el archivo *message.properties*
- Para las restricciones creadas por nosotros mismos, la jerarquía de mensajes empezaría por *operacion.fechaInicio.validator.error.fechaInicio* y así sucesivamente