Configuración y despliegue de aplicaciones

Índice

Ahora que ya tenemos los conocimientos necesarios para el desarrollo de aplicaciones con Grails y su completo entorno de desarrollo, es hora de pensar en lo que viene después de la fase de desarrollo y testeo, la fase de producción.

En esta sesión veremos como configurar nuestra aplicación en función del entorno en el que estemos trabajando. Posteriormente veremos como realizar el empaquetamiento de aplicaciones, para terminar hablando de la actualización de aplicaciones ante nuevas versiones de Grails.

1. Configuración de aplicaciones

1.1. 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 `com.biblioteca.miParametro = "dato"`, ésta va a ser accesible desde cualquier controlador, vista, servicio, etc. mediante la expresión `grailsApplication.config.com.biblioteca.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.location`: ubicaciones donde encontrar otros archivos de configuración que se fundirán con el principal *Config.groovy*
- `grails.enable.native2ascii`: en caso de que el valor sea true, Grails utilizará native2ascii para convertir los archivos properties al formato unicode
- `grails.views.default.codec`: especifica el formato por defecto de nuestras páginas GSPs. Puede tomar el valor 'none', que es el valor por defecto, 'html', o 'base64'
- `grails.views.gsp.encoding`: codificación de las páginas GSP
- `grails.converters.encoding`: codificación de los convertidores
- `grails.mime.file.extensions`: habilita el uso de la extensión en la url para fijar el *content-type* de la respuesta. Por ejemplo, si se añade la extensión *.xml* al final de cualquier url, el content-type se fijará automáticamente a 'text/xml', ignorando la cabecera Accept del navegador
- `grails.mime.types`: indica un mapa con los posibles tipos mime soportados en nuestra aplicación
- `grails.serverURL`: la parte "fija" de nuestros enlaces cuando queremos generar rutas absolutas

1.2. 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 diferente para cada uno, puesto que los requerimientos serán distintos. Por ejemplo, en un entorno de producción posiblemente nos sea suficiente utilizar una base de datos en memoria como HSQLDB, pero probablemente para los entornos de test y producción es más que probable que este tipo de bases de datos no nos sirvan y tengamos que 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
        driverClassName = "org.hsqldb.jdbcDriver"
        username = "sa"
        password = ""
}
hibernate {
    cache.use_second_level_cache=true
    cache.use_query_cache=true
cache.provider_class='com.opensymphony.oscache.hibernate.OSCacheProvider'
}
// environment specific settings
environments {
        development {
                dataSource {
                        dbCreate = "create-drop" // one of
'create', 'create-drop','update'
                        url = "jdbc:hsqldb:mem:devDB"
                }
        }
        test {
                dataSource {
                        dbCreate = "update"
                        url = "jdbc:hsqldb:mem:testDb"
                }
        }
        production {
                dataSource {
                    pooled = true
                    dbCreate = "update"
                    url = "jdbc:mysql://localhost/biblioteca"
                    driverClassName = "com.mysql.jdbc.Driver"
                    username = "user_biblioteca"
```

```
                    password = "pwdbiblio"
                }
        }
}
```

En el primer bloque *dataSource* se definen una serie de valores genéricos, que posteriormente podremos sobreescribir 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	`grails dev run-app` o `grails run-app`
Test	`grails test run-app`
Producción	`grails prod run-app`

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 HSQLDB, eso si, diferente para cada uno de ellos (*devDB* y *testDb*).

Para el entorno de producción vamos a 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.

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.

1.3. El archivo BootStrap.groovy

A lo largo de las sesiones relativas a Grails, hemos utilizado un archivo de configuración llamado *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.

Teniendo en cuesta esto, necesitaremos diferenciar en cada momento el entorno de ejecución de nuestra aplicación para insertar datos o no. Para ello, Grails dispone del paquete `grails.util.GrailsUtil` y la variable `GrailsUtil.environment` 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.GrailsUtil

class BootStrap {
        def init = { servletContext ->
                switch (GrailsUtil.environment){
                        case "development":
configuracionDesarrollo()
                                break;
                        case "test":configuracionTest()
                                break;
                        case "production":configuracionProduccion()
                                break;
                }
        }
        def destroy = {
                switch (GrailsUtil.environment){
                        case "development": salirDesarrollo()
                                break;
                        case "test":salirTest()
                                break;
                        case "production":salirProduccion()
                                break;
                }
        }
}
```

1.4. 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.

```
class UrlMappings {
    static mappings = {
      "/$controller/$action?/$id?"{
              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, si en nuestra aplicación escribiéramos algo como http://localhost:8080/biblioteca/libro/3, desearíamos ver lo mismo que si escribiésemos http://localhost:8080/biblioteca/libro/show/3, es decir, querríamos ver los datos del libro que tenga el identificador 3.

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.

```
"/libro/$id"{
      controller = "libro"
      action = "show"
}
```

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.

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 castellano y por lo tanto las URLs se muestran también en castellano. Si deseamos que estas URLs se muestren también en inglés, podemos crear una serie de reglas en el archivo *UrlMappings.groovy*.

```
"/book/$action/$id"{
      controller = "libro"
}

"/user/$action/$id"{
      controller = "usuario"
}

"/operation/$action/$id"{
      controller = "operacion"
}
```

Las reglas de mapeo también permiten la introducción de restricciones que deben cumplir

las partes de la URL. Por ejemplo, algo típico de los blogs es mostrar la dirección de un artículo con la fecha (año y mes) en la que fue publicado seguido del identificador del artículo, como por ejemplo, http://localhost/blog/2009/06/2. 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. Para que la aplicación supiera que hacer con este tipo de direcciones debemos introducir la siguiente regla de mapeo.

```
"/blog/$anyo/$mes/$id" {
        controller = "blog"
        action = "show"
        constraints {
                anyo(matches:/d{4}/)
                mes(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:'errores', action:'notFound')`, para que sea un controlador quien haga este trabajo.

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 *biblioteca*, seguido de la versión de la aplicación. En nuestro caso, la primera vez que ejecutemos el comando `grails war` el fichero generado se llamará *biblioteca-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.

1. Actualizar el código fuente del repositorio de control de versiones para asegurarse de que todas las partes del proyecto están actualizadas
2. Ejecutar los tests de integración, unitarios y funcionales que hayamos implementado para comprobar que todo funciona tal y como esperamos
3. Incrementar la variable *app.version* del archivo *application.properties* manualmente o bien mediante el comando `grails set-version 0.2`

4. Limpiar el proyecto de archivos temporales mediante el comando `grails clean`
5. 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, GlassFish, Apache Geronimo, BEA WebLogic o IBM WebSphere o incluso en un contenedor web como Apache Tomcat o Jetty. 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 de la web oficial de Grails puedes encontrar información sobre como desplegar los archivos *WAR* en varios servidores.

Como puedes observar, la variedad de servidores donde se pueden desplegar los archivos *WAR* generados es muy elevada. Quizás sea ese el motivo por el que no existen ningún comando `grails deploy`, pero nosotros vamos a crear nuestro propio script para realizar esta tarea.

Si nos fijamos cuando ejecutamos cualquiera de los comandos vistos hasta ahora, por ejemplo el último del que hemos hablado `grails war`, en la línea de comandos se nos muestra una serie de datos, y prácticamente la primera de ellas nos indica que script se está ejecutando mediante el comando en cuestión. En el caso de `grails war`, podemos ver como se ejecuta el comando *War.groovy* que está ubicado en el directorio *scripts* del directorio donde está instalado Grails.

Si abrimos este directorio, podemos comprobar la existencia de una serie de archivos con extensión *.groovy* que coinciden con las opciones que tenemos con el comando `grails`. Con esto tenemos, que si creamos cualquier script dentro de este directorio, vamos a poder ejecutarlo en línea de comandos. Pero no sólo tenemos este directorio para copiar nuestros scripts sino que también vamos a poder ubicarlos en *USER_HOME/.grails/scripts*, *PROJECT_HOME/scripts*, *PROJECT_HOME/plugins/*/scripts/* o en *GRAILS_HOME/scripts*.

Lo que vamos a hacer ahora, es crear un nuevo script en *GRAILS_HOME/scripts* para que el proceso de despliegue de una aplicación en un servidor no sea tan traumática como en ocasiones puede llegar a ser. En este script vamos a suponer que la aplicación se va a desplegar en un servidor JBoss, el cual permite el despliegue automático de aplicaciones simplemente copiando el archivo *WAR* en un directorio especial de despliegue.

```
/**
* Script Gant que copia un archivo WAR
* en un directorio de despliegue de un servidor
*/

Ant.property(environment:"env")
grailsHome = Ant.antProject.properties."env.GRAILS_HOME"
```

```
includeTargets << new File ( "${grailsHome}/scripts/War.groovy" )

target ('default':'''Copia un archivo WAR al directorio de
despliegue de un servidor.

 Ejemplo:
 grails deploy
 grails prod deploy
 ''') {
   deploy()
}

target (deploy: "Despliegue en el servidor") {
   depends( war )

   def deployDir = "jbossserver"

   Ant.copy(todir:"${deployDir}", overwrite:true) {
     fileset(dir:"${basedir}", includes:"*.war")
  }

  event("StatusFinal", ["Archivo WAR copiado en ${deployDir}"])
}
```

En primer lugar se ejecuta el comando para generar un archivo *WAR* de la aplicación y posteriormente se copia este archivo al directorio indicado en la variable *deployDir*. Nombramos este archivo como *Deploy.groovy*, lo guardamos en el directorio *scripts* de la instalación de Grails y posteriormente ejecutamos el comando `grails deploy`.

3. Actualización de aplicaciones

Desde los inicios de Grails, se vio claramente que, con las constantes actualizaciones del framework, se hacía necesario un método para actualizar la versión de Grails instalada sin que aquello provocara mayor problema. Como la mayoría de partes en Grails, esto se soluciona por medio de un comando, en este caso `grails upgrade`.

Cuando ejecutamos el comando `grails run-app` sobre una aplicación, Grails comprueba que la versión de Grails instalada coincida con la que se indica en el archivo ubicado en la raiz del proyecto *application.properties* y en caso de no ser así, se mostraría un mensaje de advertencia indicándonos que debemos actualizar la versión de Grails con el comando `grails upgrade`.

4. Tareas programadas con Quartz

Una de las partes habituales de cualquier aplicación web es la ejecución programada de determinadas tareas o scripts. Esta tareas deben ser lanzadas sistemáticamente a determinadas horas del día o cada cierto intervalo de tiempo. En la mayoría de ocasiones este tipo de sistemas suelen quedarse fuera de la propia aplicación, tal y como ocurre con Apache y el crontab de linux, lo que lo hace poco aconsejable cuando llega el momento

de exportar el proyecto a otros servidores.

Para solucionar esta falta de integración entre las tareas programadas y el proyecto de aplicación, Grails dispone del plugin Quartz, que permite la planificación de estas tareas insertándolas en una clase de nuestra aplicación.

Empecemos como siempre instalando este plugin. Para ello debemos ejecutar el comando `grails install-plugin quartz` y en caso de que este comando nos dé error, deberemos descargarlo en primer lugar desde la dirección http://svn.codehaus.org/grails-plugins/grails-quartz/tags/RELEASE_0_4_1/grails-quartz-0.4.1.zip y posteriormente instalarlo con el comando `grails install-plugin /path/donde/se/descargo/el/plugin`.

Una vez instalado el plugin, tendremos la posibilidad de utilizar dos nuevos comandos que se habrán agregado a la lista de posibles comandos de Grails, `grails create-job` o `grails install-quartz-config`.

Con el comando `grails install-quartz-config` vamos a poder configurar dos parámetros del plugin. Cuando ejecutemos este comando, se creará un archivo de configuración en el directorio de nuestra aplicación *grails-app/conf* llamado *QuartzConfig.groovy* y que nos permitirá controlar dos variables. La variable `autoStartup` indica si cuando arranque la aplicación arrancará también la ejecución de Quartz. Su valor por defecto es `true`. La otra variable se llama `jdbcStore` e indica si los trabajos programados deben ser persistidos en la base de datos. Su valor por defecto es `false`.

Mediante el comando `grails create-job`, el sistema nos pedirá el nombre del trabajo que queremos crear y se creará una nueva clase en el directorio *grails-app/jobs* con el nombre dado seguido de la palabra *Job*. Veamos un ejemplo.

```
class TareasJob {
    def startDelay = 30000
    def timeout = 1000

    def group = "GrupoDeTareas1"

    def execute(){
        print "Ejecuto la tarea programada!"
    }
}
```

Aquí de nuevo vuelve a aparecer la teoría de convenio sobre configuración de Grails. En esta ocasión, el convenio se refiere a la utilización de la variable `startDelay` para indicarle a la aplicación cuanto tiempo en milisegundos debe esperar para lanzar la tarea por primera vez (30 segundos en nuestro ejemplo).

Con la variable `timeout` le indicamos al sistema cuanto tiempo debe esperar en milisegundos para volver a ejecutar la tarea. El método `execute()` será el que se ejecute cada vez que lance la tarea. Además, podemos agrupar las tareas en grupos gracias a la

variable `group`. Si no especificamos ningún valor a las variables `startDelay` y `timeout`, el valor por defecto que tomarán serán 30 y 60 segundos respectivamente. No es conveniente especificar un valor a la variable `startDelay` por debajo de 30 segundos, ya que la aplicación debe estar completamente inicializada para empezar a ejecutar las tareas programadas.

En el ejemplo anterior, veíamos como podíamos ejecutar una tarea programada indicándole al sistema cada cuanto tiempo debe ejecutarse mediante las variables `startDelay` y `timeout`. Sin embargo, existe otra forma más conocida sobre todo entre la gente que utiliza Linux como son las expresiones de tipo *cron*. Con una expresión de este tipo vamos a poder especificarle al sistema que los días 28 de cada mes se ejecute una determinada tarea o que todos los días a las 17:30 queremos hacer una determinada acción.

El siguiente ejemplo muestra el código de una nueva tarea programada que se ejecutaría todos los días a las 15h.

```
class TareasCronJob  {
 def cronExpression = "0 0 15 * * ?"

 def group = "GrupoDeTareas1"

 def execute(){
   print "Ha llegado la hora!"
 }
}
```

Las expresiones de tipo cron tienen 6 campos obligatorios más uno opcional. Por orden de aparición, los campos de una expresión tipo cron tienen el siguiente significado:

Campo	Valores permitidos	Valores especiales permitidos
Segundos	0-59	, - * /
Minutos	0-59	, - * /
Horas	0-23	, - * /
Día del mes	1-31	, - * ? / L W
Mes	1-12 o JAN-DEC	, - * /
Día de la semana	1-7 o SUN-SAT (la semana empieza los domingos)	, - * ? / L #
Año (opcional)	1970-2099	, - * /

- El carácter '*' se especifica para indicar todos los valores. Por ejemplo, si ponemos un '*' en el campo minutos, se indica que se debe ejecutar la tarea cada minuto
- El carácter '?' indica que no se especifica ningún valor determinado
- El carácter '-' se utiliza para especificar rangos. Por ejemplo, si indicamos en la hora el rango '10-12' le estamos indicando al sistema que la tarea se debe ejecutar a las 10, a

las 11 y a las 12

- El carácter ',' se utiliza para especificar valores adicionales. Por ejemplo, podemos indicar que los lunes, martes y miércoles se debe ejecutar determinada tarea especificando el valor '2,3,4' (recuerda que la semana empieza los domingos)
- El carácter '/' se utiliza para especificar incrementos. Por ejemplo, si especificamos en el campo minuto el valor '0/15', la tarea se ejecutará en los minutos 0, 15, 30 y 45 de cada hora. Utilizar el carácter '*' antes del carácter '/' es equivalente a especificar como valor inicial el 0.
- El carácter 'L' es el método abreviado de *last* (último), pero en los dos campos donde este carácter está permitido tiene un significado diferente en cada uno de ellos. Si especificamos el valor 'L' en el campo día del mes, estaremos indicándole el último día del més (28, 29, 30 o 31). Si indicamos 'L' en el campo día de la semana, significará el último día de la semana, o sea el sábado. Pero si utilizamos este carácter en el campo día de la semana seguido de otro valor, por ejemplo '6L', esto significará el último viernes de cada mes.
- El carácter 'W' solamente se permite en el campo día del mes y se utiliza para especificar el próximo día de entre semana más cercano a un valor especificado. Por ejemplo, si indicamos '15W' se indicará el día de lunes a viernes más cercano al día 15 de cada mes. Si el 15 de un mes fuera sábado, la tarea programada se ejecutaría el lunes 17 de ese mes.
- El carácter '#' sólo se permite para el campo día de la semana. Especificando el valor '6#3' estaríamos especificando que queremos ejecutar la tarea el tercer viernes de cada mes.

A continuación tenemos algunos ejemplos de expresiones de tipo cron

Expresión	Significado
0 0 12 * * ?	Todos los días a las 12h
0 15 10 * * ? *	Todos los días a las 10:15h
0 15 10 * * ? 2009	Todos los días a las 10:15h en el 2009
0 * 14 * * ?	Cada minuto desde las 14:00h hasta las 14:59h
0 0/5 14,18 * * ?	Cada 5 minutos desde las 14:00h hasta las 14:59h y cada 5 minutos desde las 18:00h hasta las 18:59h
0 15 10 ? * MON-FRI	Cada lunes, martes, miércoles, jueves y viernes a las 10:15h
0 15 10 L * ?	El última día del mes a las 10:15h
0 15 10 ? * 6L	El último viernes de cada mes a las 10:15h
0 15 10 ? * 6#3	El tercer viernes de cada mes a las 10:15h

Una vez visto las expresiones cron, veamos como ejecutar múltiples tareas en una sola

clase. Para ello, debemos configurar la variable `triggers` y podemos hacerlo de tres formas diferentes.

- *simpleTrigger*, especificando el tiempo que tarda la tarea en empezar con `startDelay`, el `timeout` entre ejecuciones y el número de repeticiones en la variable `repeatCount`
- *cronTrigger*, especificando el tiempo que tarda la tarea en empezar con `startDelay` y una expresión cron
- *customTrigger*, una clase que implementa la interface Trigger y cualquier parámetro necesario por el trigger

```
class TareasTriggerJob {
    static triggers = {
        simpleTrigger startDelay:10000, timeout: 30000,
repeatCount: 10
        cronTrigger startDelay:10000, cronExpression: '0/6 * 15 * *
?'
        customTrigger claseTrigger:MiClaseTrigger,
miParametro:miValor, miOtroParametro:miOtroValor
    }

    def execute() {
        println "Ejecuto una tarea!"
    }
}
```

Con esta configuración, la tarea se ejecutará 11 veces cada 30 segundos, y además también se ejecutará cada 6 segundos desde las 15:00h a las 15:59h. Y por último, también se ejecutaría nuestra propia clase.

4.1. Envío de notificaciones automáticas

Si sois miembros de la Universidad de Alicante, supongo que en alguna ocasión habréis recibido algún correo electrónico procedente de la Biblioteca en el que os avisan de que un libro que tenéis reservado, ya está a vuestra disposición para que paséis a recogerlo. En otras ocasiones, ese correo electrónico se refiere a que ha caducado un préstamos de un libro y que debéis devolverlo cuanto antes.

Por supuesto, detrás de ese envío de correo electrónico no hay ninguna persona comprobando las operaciones caducadas o las reservas disponibles, sino que es un proceso automático y programado.

Aprovechando el servicio de mensajería desarrollado en la sesión 10 y el automatizado de tareas que acabamos de ver gracias al plugin Quartz, vamos a ver como quedaría un sistema similar al que hemos comentado.

En primer lugar, vamos a crear una nueva tarea con el comando `grails create-job` llamado *NotificacionOperacion*. En esta tarea necesitamos obtener aquellos préstamos caducados cuyo estado sea activo. Una vez obtenidas estas operaciones, enviaremos un

correo electrónico indicándole al usuario que su préstamo ha caducado y que debe devolver el libro. La tarea programada quedaría así:

```
class NotificacionOperacionJob  {
    def cronExpression = "0 0 0 * * ?"

    def notificadorService

    def execute(){
            def operacionesCaducadas = Operacion.findAll("from
Operacion as o where o.fechaFin < ? and o.tipo = ? and o.estado =
?", [new Date(), "prestamo", true])
        if (operacionesCaducadas.size()>0){
            operacionesCaducadas.each {
notificadorService.mandarMails(it.usuario.email,"Aviso de la
Biblioteca","El prestamo del libro ${it.libro.titulo} ha caducado")
            }
        }
    }
}
```

La tarea programada se ejecutará todos los días a las 00:00h y en primer lugar, se buscarán todas aquellas reservas cuya fecha fin sea anterior a la fecha actual y su estado sea activo. Podemos comprobar el funcionamiento de esta tarea modificando la fecha de lanzamiento de la tarea.

5. Otros comandos interesantes de Grails

A lo largo de todo el curso, 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
`grails bug-report`	Genera un archivo comprimido en ZIP con los archivos fuente de nuestro proyecto para el caso de que queramos informar de un bug
`grails clean`	Limpia el directorio *tmp* de nuestra aplicación. Este comando puede ser combinado con otros comandos como por ejemplo `grails clean run-app`
`grails console`	Nos muestra la consola de Groovy que veíamos en la primera sesión del curso para que podamos hacer nuestras pruebas.

`grails doc`	Genera la documentación completa de nuestro proyecto.
`grails help`	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 `grails help doc`
`grails list-plugins`	Muestra un listado completo tanto de los plugins disponibles como de los ya instalados en la aplicación.
`grails plugin-info`	Muestra la información completa del plugin pasado como parámetro al comando.
`grails run-app -https`	Ejecuta el comando `grails run-app` utilizando como servidor Jetty pero sobre un servidor seguro *https*. El puerto por defecto es 8443 y puede ser modificado añadiendo al comando `-Dserver.port.https=<numero_puerto>`
`grails schema-export`	Genera un fichero con las sentencias SQL necesarias para exportar la base de datos.
`grails set-version`	Establece la versión de la aplicación. Por ejemplo `grails set-version 1.0.4`
`grails stats`	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.
`grails uninstall-plugin`	Desinstala el plugin pasado como parámetro de la aplicación.

Podéis encontrar un listado completo de todos los comandos disponibles en Grails en la dirección http://www.grails.org/Command+Line.