# Spring

## Índice


1 Introducción a Spring.....4
1.1 ¿Qué es Spring?.....4
1.2 Estereotipos configurables.....5
1.3 Inyección de dependencias.....11
1.4 Alternativas a las anotaciones para la configuración.....16
2 Ejercicios del contenedor de beans de Spring.....21
2.1 Configuración del proyecto.....21
2.2 Estructura de la aplicación.....24
2.3 Crear la capa de negocio (1 punto).....24
2.4 Crear la capa de acceso a datos y enlazarla con la de negocio (1.5 puntos).....25
2.5 Configurar beans en el XML (0.5 puntos).....26
3 Acceso a datos.....27
3.1 La filosofía del acceso a datos en Spring.....27
3.2 Uso de JDBC.....28
3.3 Uso de JPA.....32
3.4 Transaccionalidad declarativa.....36
4 Ejercicios de Acceso a datos en Spring.....41
4.1 Uso de JDBC en Spring (1 punto).....41
4.2 Transaccionalidad declarativa (1 punto).....42
4.3 Uso de JPA en Spring (1 punto).....43
5 Introducción a MVC en Spring.....45
5.1 Procesamiento de una petición en Spring MVC.....45
5.2 Configuración básica.....46
5.3 Caso 1: petición sin procesamiento de datos de entrada.....47
5.4 Caso 2: procesamiento de un formulario.....52
6 Ejercicios de MVC en Spring.....59


Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

6.1 Configurar el proyecto para Spring MVC (0.5 puntos)..... 59
6.2 MVC sin procesamiento de datos de entrada (1 punto)..... 60
6.3 MVC con procesamiento de datos de entrada (1 punto)..... 60
6.4 Taglibs de Spring y validación básica de datos (1 punto)..... 61
7 Aplicaciones AJAX y REST con Spring MVC..... 63
7.1 AJAX con Spring..... 63
7.2 Servicios web REST..... 67
7.3 Tratamiento de errores en aplicaciones AJAX y REST..... 72
8 Ejercicios de AJAX y REST..... 75
8.1 AJAX (1 punto)..... 75
8.2 Servicios REST (1.5 puntos)..... 76
8.3 Gestión de errores en servicios REST (0.5 puntos)..... 78
9 Validación e internacionalización con Spring MVC..... 79
9.1 Validación en Spring..... 79
9.2 Internacionalización y formateo de datos..... 82
10 Ejercicios de validación e internacionalización..... 87
10.1 Conversión y formateo de datos (0.5 puntos)..... 87
10.2 Validación (1.5 puntos)..... 87
10.3 Internacionalización (1 punto)..... 88
11 Acceso remoto. Pruebas..... 90
11.1 Acceso remoto..... 90
11.2 Pruebas..... 95
12 Ejercicios de acceso remoto y pruebas..... 104
12.1 Acceso remoto con HttpInvoker (1 punto)..... 104
12.2 Pruebas de la capa DAO (0.5 puntos)..... 105
12.3 Pruebas de la capa BO con y sin objetos mock (1 punto)..... 106
12.4 Pruebas de la capa web (0.5 puntos)..... 107
13 Seguridad..... 108
13.1 Conceptos básicos de seguridad..... 108
13.2 Una configuración mínima para una aplicación web..... 109
13.3 Autentificación contra una base de datos..... 112
13.4 Seguridad de la capa web..... 115



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

13.5 Seguridad de la capa de negocio..... 120
14 Ejercicios de Spring Security.....124
14.1 Seguridad de la capa web (1 punto)..... 124
14.2 Personalización de la seguridad web (1 punto) .....124
14.3 Seguridad en los JSP (0.5 puntos).....125
14.4 Seguridad de la capa de negocio (0.5 puntos).....125
15 Desarrollo rápido de aplicaciones con Spring Roo.....126
15.1 Introducción rápida a Spring Roo..... 126
15.2 La capa de acceso a datos.....134
15.3 La capa web.....138
15.4 Refactorización del código Roo..... 140
16 Programación orientada a aspectos (AOP) en Spring.....143
16.1 Introducción a la AOP.....143
16.2 AOP en Spring..... 145
16.3 Puntos de corte (pointcuts).....146
16.4 Advices.....148
16.5 AOP "implícita" en Spring 3.....152



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

# 1. Introducción a Spring

## 1.1. ¿Qué es Spring?

Spring es un *framework* alternativo al *stack* de tecnologías estándar en aplicaciones JavaEE. Nació en una época en la que las tecnologías estándar JavaEE y la visión "oficial" de lo que debía ser una aplicación Java Enterprise tenían todavía muchas aristas por pulir. Los servidores de aplicaciones eran monstruosos devoradores de recursos y los EJB eran pesados, inflexibles y era demasiado complejo trabajar con ellos. En ese contexto, Spring popularizó ideas como la inyección de dependencias o el uso de objetos convencionales (POJOs) como objetos de negocio, que suponían un soplo de aire fresco. Estas ideas permitían un desarrollo más sencillo y rápido y unas aplicaciones más ligeras. Eso posibilitó que de ser un *framework* inicialmente diseñado para la capa de negocio pasara a ser un completo *stack* de tecnologías para todas las capas de la aplicación.

Las ideas "innovadoras" que en su día popularizó Spring se han incorporado en la actualidad a las tecnologías y herramientas estándar. Así, ahora mismo no hay una gran diferencia entre el desarrollo con Spring y el desarrollo JavaEE "estándar", o al menos no tanta como hubo en su día. No obstante, Spring ha logrado aglutinar una importante comunidad de desarrolladores en torno a sus tecnologías y hoy por hoy sigue constituyendo una importante alternativa al estándar que merece la pena conocer. En la actualidad, las aportaciones más novedosas de Spring se centran en los campos de Big Data/NoSQL, HTML5/móviles y aplicaciones sociales.

Básicamente, la mayor diferencia práctica que podemos encontrar hoy en día entre desarrollar con Spring y con JavaEE estándar es la posibilidad de usar un servidor web convencional al estilo Tomcat para desplegar la aplicación. Las tecnologías JavaEE más sofisticadas requieren del uso de un servidor de aplicaciones, ya que los APIs los implementa el propio servidor, mientras que Spring no es más que un conjunto de librerías portables entre servidores. En otras palabras, usando JavaEE estándar, nos atamos al servidor de aplicaciones y usando Spring nos atamos a sus APIs. Eso sí, los desarrolladores de Spring se han preocupado bastante de armonizar con el estándar en la medida de lo posible, por ejemplo dando la posibilidad de usar anotaciones estándar aun con implementaciones propias por debajo. La idea es obstaculizar lo menos posible una posible portabilidad a JavaEE, idea que es de agradecer en un mundo en que todos los fabricantes intentan de una forma u otra mantener un público cautivo.

Hay una abundante bibliografía sobre Spring, aunque la documentación del propio proyecto es excelente y bastante exhaustiva, pudiéndose utilizar perfectamente no solo como manual de referencia sino como tutorial detallado. La hemos tomado como referencia básica para la elaboración de estos apuntes.

Desde un punto de vista genérico, Spring se puede ver como un soporte que nos



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

proporciona tres elementos básicos:

- **Servicios *enterprise*:** podemos hacer de manera sencilla que un objeto sea transaccional, o que su acceso esté restringido a ciertos roles, o que sea accesible de manera remota y transparente para el desarrollador, o acceder a otros muchos servicios más, sin tener que escribir el código de manera manual. En la mayoría de los casos solo es necesario anotar el objeto.
- **Estereotipos configurables** para los objetos de nuestra aplicación: podemos anotar nuestras clases indicando por ejemplo que pertenecen a la capa de negocio o de acceso a datos. Se dice que son configurables porque podemos definir nuestros propios estereotipos "a medida": por ejemplo podríamos definir un nuevo estereotipo que indicara un objeto de negocio que además sería cacheable automáticamente y con acceso restringido a usuarios con determinado rol.
- **Inyeccion de dependencias:**: ya hemos visto este concepto cuando se hablaba de CDI de JavaEE. La inyección de dependencias nos permite solucionar de forma sencilla y elegante cómo proporcionar a un objeto cliente acceso a un objeto que da un servicio que este necesita. Por ejemplo, que un objeto de la capa de presentación se pueda comunicar con uno de negocio. En Spring las dependencias se pueden definir con anotaciones o con XML.

## 1.2. Estereotipos configurables

Spring puede gestionar el ciclo de vida de los objetos que queramos. Los objetos gestionados por el framework se denominan genéricamente *beans* de Spring (no confundir con el concepto estándar de bean en Java). Esto no es nada extraño, es lo que sucede por ejemplo con los servlets, normalmente no los instancia el desarrollador sino que lo hace el contenedor web cuando es necesario. Spring extiende esta idea permitiéndonos gestionar el ciclo de vida de cualquier objeto. Para ello tendremos que anotarlo (o crear un fichero de configuración XML, aunque esta opción tiende a estar en desuso).

Spring ofrece una serie de anotaciones estándar para los objetos de nuestra aplicación: por ejemplo, **@Service** indica que la clase es un bean de la capa de negocio, mientras que **@Repository** indica que es un DAO. Si simplemente queremos especificar que algo es un bean sin decir de qué tipo es podemos usar la anotación **@Component**. Por ejemplo:

```
package es.ua.jtech.spring.negocio;

import org.springframework.stereotype.Service;

@Service("usuariosBO")
public class UsuariosBOSimple implements IUsuariosBO {
   public UsuarioTO login(String login, String password) {
     ...
   }
   public bool logout() {
     ...
   }
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
    ...
}
```

El parámetro "usuariosBO" de la anotación sirve para darle un nombre o identificador al bean, que deberá ser único. Veremos ejemplos de su uso en el apartado siguiente. Si no le diéramos uno, el identificador por defecto sería el nombre de la clase pero con la inicial en minúscula.

**@Service vs. @Repository vs. @Component**

Nótese que en la versión actual de Spring la anotación @Service no tiene una semántica definida distinta a la de @Component. Es decir, simplemente le ayuda al que lee el código a saber que el bean pertenece a la capa de negocio y por lo demás es indiferente usar una u otra. La anotación @Repository sí tiene efecto sobre la transaccionalidad automática, como veremos en la siguiente sesión. No obstante, el equipo de desarrollo de Spring se reserva la posibilidad de añadir semántica a estas anotaciones en futuras versiones del *framework*.

Para que nuestro bean funcione, falta un pequeño detalle. Spring necesita de un fichero XML de configuración mínimo. Cuando los beans no se definen con anotaciones sino con XML, aquí es donde se configuran, en lugar de en el fuente Java. Pero aunque usemos anotaciones en el fuente, como en nuestro caso, el XML de configuración mínimo sigue siendo necesario. En él debemos decirle a Spring que vamos a usar anotaciones para definir los beans.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd
         http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

     <context:component-scan base-package="es.ua.jtech.spring"/>
</beans>
```

La etiqueta **<context:component-scan>** es la que especifica que usaremos anotaciones para la definición de los beans, y que las clases que los definen van a estar en una serie de paquetes (todos van a ser subpaquetes de **base-package**). Por ejemplo, el propio `es.ua.jtech.spring` o paquetes como `es.ua.jtech.spring.negocio` o `es.ua.jtech.spring.base.to`

**¡Cuidado!**

Las anotaciones puestas en clases que no estén definidas en el paquete "base-package" o en alguno de sus subpaquetes no tendrán ningún efecto. Es un error muy típico olvidarse de esto y desesperarse ante el hecho de que Spring "ignora" las anotaciones.

### 1.2.1. Solicitarle beans al contenedor



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

Evidentemente, para que la gestión del ciclo de vida funcione, tiene que haber "alguien" que se encargue de realizarla. En el caso de los servlets el encargado es el contenedor web (por ejemplo Tomcat). En Spring existe un concepto equivalente: el contenedor de beans, que hay que configurar e instanciar en nuestra aplicación (el contenedor web es algo estándar de JavaEE y por tanto su arranque es "automático", pero Spring no es más que una librería Java y por tanto no goza de ese privilegio). El contenedor implementa la interfaz `ApplicationContext`. Spring ofrece diferentes implementaciones de este interfaz, algunas apropiadas para aplicaciones de escritorio y otras para aplicaciones web.

En **aplicaciones de escritorio** es común usar la clase `ClassPathXmlApplicationContext`, a la que hay que pasarle el nombre del fichero de configuración que vimos en el apartado anterior. Como su propio nombre indica, esta clase buscará el archivo en cualquier directorio del *classpath*.

```
ApplicationContext contenedor =
    new ClassPathXmlApplicationContext("configuracion.xml");
```

El application context actúa como si fuera una factoría de beans, de modo que podemos obtener de él el objeto deseado:

```
ApplicationContext contenedor =
    new ClassPathXmlApplicationContext("configuracion.xml");
IUsuariosBO iub = contenedor.getBean(IUsuariosBO.class);
UsuarioTO uto = igu.login("javaee", "javaee");
```

El context nos permite recuperar los beans por clase (o interfaz, como en el ejemplo), pero también podríamos recuperarlos por su identificador (recordar que habíamos asignado un identificador en la anotación `@Service`).

```
...
IUsuariosBO iub = contenedor.getBean("usuariosBO", IUsuariosBO.class);
...
```

El acceso por identificador nos será útil si hemos definido distintos beans de la misma clase, con distintas propiedades cada uno (aunque esto no se puede hacer únicamente con anotaciones, necesitaríamos usar la configuración XML o Java).

A primera vista parece que no hemos obtenido nada con Spring que no hubiéramos podido conseguir de manera mucho más sencilla con un modesto `new UsuariosBOSimple()`, pero esta forma de hacer las cosas presenta ciertas ventajas:

- El *application context* se puede ver como una implementación del patrón *factory*. Este patrón nos independiza de la clase concreta que use nuestra implementación. El código que hace uso del application context para obtener el gestor de usuarios no necesita saber qué clase concreta se está usando ni debe cambiar si cambia ésta. Todo esto, como ya se ha visto, a costa de introducir complejidad adicional en el proyecto (¡nada es gratis!).
- Podemos cambiar ciertos aspectos del ciclo de vida del bean de manera declarativa.



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

Por ejemplo, el ámbito: es muy posible que un solo objeto pueda hacer el trabajo de todos los clientes que necesiten un `IGestorUSuarios` (un *singleton*, en argot de patrones). O al contrario, podría ser que cada vez hiciera falta un objeto nuevo. O también podría ser que si estuviéramos en una aplicación web, cada usuario que va a hacer login necesitara su propia instancia de `IGestorUSuarios`. Todo esto lo podemos configurar en Spring de manera declarativa, sin necesidad de programarlo, y que sea el contenedor el que se ocupe de estas cuestiones del ciclo de vida.

En **aplicaciones web** la configuración del contenedor se hace con la clase `WebApplicationContext`, definiéndola como un *listener* en el fichero descriptor de despliegue, web.xml.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app ...>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/misBeans.xml</param-value>
  </context-param>

  <listener>
     <listener-class>
        org.springframework.web.context.ContextLoaderListener
     </listener-class>
  </listener>

  <!--  resto de etiquetas del web.xml -->
  ...
</web-app>
```

La clase `ContextLoaderListener` carga el fichero o ficheros XML especificados en el `<context-param>` llamado `contextConfigLocation` (suponemos que el fichero `misBeans.xml` está en el directorio `WEB-INF`). Como `<param-value>` se puede poner el nombre de varios ficheros XML, separados por espacios o comas.

Una vez arrancado el contenedor, podemos acceder a un bean a través de la clase `WebApplicationContext`, que funciona de manera prácticamente igual que la ya vista `ClassPathXmlApplicationContext`. El `WebApplicationContext` es accesible a su vez a través del contexto del servlet, por lo que en un JSP podríamos hacer algo como:

```
<%@ page import ="org.springframework.web.context.*" %>
<%@ page import ="org.springframework.web.context.support.*" %>
<%@ page import ="es.ua.jtech.spring.negocio.*" %>
<html>
<head>
   <title>Acceso a beans de spring desde un JSP</title>
</head>
<body>
<%
  ServletContext sc = getServletContext();
  WebApplicationContext wac =
          WebApplicationContextUtils.getWebApplicationContext(sc);
  IUsuariosBO iub = wac.getBean(IUsuariosBO.class);
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
  UsuarioTO uto = iub.login("javaee", "javaee");
%>
</body>
</html>
```

**Pero esto es demasiado complicado, ¿no?...**

Toda esta parafernalia para obtener objetos parece excesiva. ¿No sería mejor que los objetos se obtuvieran automáticamente cuando los necesitáramos?. De ese modo, en una aplicación web, cuando se recibiera una determinada petición HTTP se obtendría automáticamente un bean de la capa de presentación, que a su vez podría acceder automáticamente al/los que necesitara de negocio, y éstos a su vez a otros de acceso a datos, y... Efectivamente, esa es la forma habitual de trabajar en aplicaciones web con Spring: el web application context está presente pero no es necesario acudir a él de manera explícita. Lo que ocurre es que para poder hacer esto necesitamos dos elementos que todavía no hemos visto: la inyección de dependencias y el framework para la capa web, Spring MVC. El primero de ellos lo abordaremos en breve, y el segundo en la sesión 3.

### 1.2.2. Ámbito de los beans

Por defecto, los beans en Spring son *singletons*. Esto significa que el contenedor solo instancia un objeto de la clase, y cada vez que se pide una instancia del bean en realidad se obtiene una referencia al mismo objeto. Recordemos que se solicita una instancia de un bean cuando se llama a `getBean()` o bien cuando se "inyecta" una dependencia del bean en otro.

El ámbito *singleton* es el indicado en muchos casos. Probablemente una única instancia de la clase `GestorPedidos` pueda encargarse de todas las tareas de negocio relacionadas con pedidos, si la clase no tiene "estado" (entendiendo por esto que no tiene variables miembro y que por tanto varios hilos independientes que accedan concurrentemente al mismo objeto no van a causar problemas). Los DAO son otro ejemplo típico de objetos apropiados que se suelen definir en Spring como singletons, ya que no suelen guardar estado.

Podemos asignar otros ámbitos para el bean usando la anotación `@Scope`. Por ejemplo, para especificar que queremos una nueva instancia cada vez que se solicite el bean, se usa el valor `prototype`

```
package es.ua.jtech.spring.negocio;

import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Service;
...

@Scope("prototype")
@Service
public class UsuariosBOSimple implements IUsuariosBO {
   ...
}
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

En **aplicaciones web**, se pueden usar además los ámbitos de `request` y `session` (hay un tercer ámbito llamado `globalSession` para uso exclusivo en portlets). Para que el contenedor pueda gestionar estos ámbitos, es necesario usar un `listener` especial cuya implementación proporciona Spring. Habrá que definirlo por tanto en el `web.xml`

```
...
<web-app>
  ...
  <listener>
    <listener-class>
       org.springframework.web.context.request.RequestContextListener
     </listener-class>
  </listener>
  ...
</web-app>
```

Ahora ya podemos usar los ámbitos especiales para aplicaciones web. Por ejemplo para definir un bean que tenga como ámbito la sesión HTTP simplemente usaremos "session" como valor de la anotación `@Scope`.

### 1.2.3. Configurar el estereotipo

Podemos aprovechar la posibilidad de definir anotaciones Java a medida para definir nuestros propios estereotipos, combinando anotaciones de Spring. Por ejemplo para un objeto de negocio con ámbito de sesión y con transaccionalidad automática podríamos definir la siguiente anotación:

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Service;

@Service
@Scope("session")
@Transactional
@Retention(RetentionPolicy.RUNTIME)
public @interface ServicioTransaccional {
}
```

Una vez hecho esto, el compilador reconocerá la anotación `@ServicioTransaccional` como propia, por ejemplo:

```
@ServicioTransaccional
public class UsuariosDAO implements IUsuariosDAO {
   public void UsuarioTO leer(String login) {
      ...
   }
   ...
}
```

### 1.2.4. Control del ciclo de vida



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

En algunos casos puede ser interesante llamar a un método del bean cuando éste se inicializa o destruye. Para ello se usan respectivamente las anotaciones `@PostConstruct` y `@PreDestroy`. Por ejemplo:

```
@Component
public class MiBean {
   @PostConstruct
   public void inicializa() {

   }

   @PreDestroy
   public void libera() {
   }

}
```

Ambos deben ser métodos sin parámetros. El método de inicialización se llama justo después de que Spring resuelva las dependencias e inicialice las propiedades del bean.

## 1.3. Inyección de dependencias

La inyección de dependencias es uno de los pilares fundamentales en que se basa Spring. Aunque no fue el primer *framework* que usaba este concepto, sí fue el que lo popularizó en el mundo Java enterprise. Durante mucho tiempo era una de las diferencias más destacadas entre el "enfoque Spring" y el JavaEE estándar. En este último, cuando un objeto necesitaba de otro o de un recurso debía localizarlo él mismo mediante el API JNDI. No obstante, en la actualidad, como ya se ha visto en otros módulos, el estándar ha ido incorporando la inyección de dependencias en múltiples tecnologías (JPA, JSF, web,...).

### 1.3.1. Uso de anotaciones estándar

Spring permite el uso de anotaciones de JSR330, para reducir el acoplamiento con los APIs de Spring y mejorar la portabilidad. Veremos en primer lugar el uso de estas anotaciones con Spring y luego las propias del *framework*.

Para anotar componentes, en el estándar se usa `@Named`, que sería la equivalente a la `@Component` de Spring (un bean, sin especificar si es de presentación, negocio o acceso a datos).

Para las dependencias ya se vio en otros módulos, por ejemplo en el de componentes web, el uso de `@Inject`. En Spring es idéntico. Continuando con los ejemplos ya vistos, supongamos que nuestro `IUsuariosBO` necesita de la colaboración de un bean auxiliar `IUsuariosDAO` para hacer su trabajo:

```
@Named("usuariosBO")
public class UsuariosBOSimple implements IUsuariosBO {
  @Inject
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
  IUsuariosDAO udao;

  public UsuarioTO login(String login, String password) {
    UsuarioTO uto = udao.recuperar(login);
    if (uto.getPassword().equals(password)) {
       ...
    }
  }

  ...
}
```

Evidentemente, para que esto funcione tiene que existir alguna clase que implemente el interfaz `IUsuariosDAO` y que esté anotada como un bean de Spring

Hay que destacar que la inyección se puede hacer en las variables miembro pero también en métodos o constructores. Esto posibilita resolver las dependencias "manualmente" sin el concurso de Spring. Por ejemplo:

```
@Named("usuariosBO")
public class UsuariosBOSimple implements IUsuariosBO {
  IUsuariosDAO udao;

  @Inject
  public setUsuariosDAO(UsuariosDAO udao) {
    this.udao = udao;
  }

  public UsuarioTO login(String login, String password) {
    UsuarioTO uto = udao.recuperar(login);
    if (uto.getPassword().equals(password)) {
       ...
    }
  }

  ...
}
```

Así, si no pudiéramos o quisiéramos usar Spring (por ejemplo, para pruebas), podríamos hacer:

```
IUsuariosBO ubo = new UsuariosBOSimple();
ubo.setUsuariosDAO(new UsuariosDAOJPA());
```

Mientras que "dentro" de Spring la dependencia se seguiría resolviendo e instanciando automáticamente.

Por defecto, la creación de todos los beans y la inyección de dependencias se efectúa cuando se arranca el contenedor (el Application Context). Se crea un bean de cada clase (ya que el ámbito por defecto es `singleton`) y se "enlazan" de modo apropiado. Si no fuera posible resolver alguna dependencia el arranque de la aplicación fallaría. Si queremos que algún bean no se inicialice cuando arranca el contenedor, sino en el momento que se solicite con `getBean`, podemos anotarlo con `@Lazy`.

No es necesario añadir Weld al proyecto ni ninguna otra implementación externa de



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

JSR330 porque en realidad se está usando la implementación propia de Spring, por lo que basta con añadir la dependencia del artefacto que define las anotaciones en sí:

```
<dependency>
    <groupId>javax.inject</groupId>
    <artifactId>javax.inject</artifactId>
    <version>1</version>
</dependency>
```

**JSR330 vs. JSR299**

Spring no permite el uso de anotaciones JSR299, *Contexts and Dependency Injection*. Estas últimas son, de hecho, más sofisticadas en ciertos aspectos que las que ofrece actualmente Spring de manera nativa.

### 1.3.2. Uso de anotaciones Spring

Usando las anotaciones propias de Spring para la inyección de dependencias perdemos portabilidad pero podemos aprovechar todas las posibilidades del framework.

El equivalente Spring a `@Inject` sería `@Autowired`. La anotación de Spring es algo más flexible, ya que nos permite especificar dependencias opcionales. En ese caso la aplicación no fallará si no es posible resolver la dependencia.

```
@Service("usuariosBO")
public class UsuariosBOSimple implements IUsuariosBO {
  @Autowired(required=false)
  IUsuariosDAO udao;

  public UsuarioTO login(String login, String password) {
    if (udao==null) {
      throw new Exception("No es posible realizar la operación");
    }
    else {
      ...
    }
  }
  ...
}
```

En el ejemplo anterior la aplicación arrancaría correctamente, aunque no fuera posible resolver la dependencia. Lo que ocurriría es que el valor miembro `udao` quedaría a `null`.

El problema contrario a no ser capaz de resolver una dependencia es **encontrar varios beans candidatos a resolverla**, ya que hay que elegir uno de ellos. Por ejemplo, supongamos que tuviéramos dos implementaciones distintas de `IUsuariosDAO`: `UsuariosDAOJPA` y `UsuariosDAOJDBC`. ¿Cuál deberíamos seleccionar para inyectarle al `UsuariosBOSimple`?. La opción es la misma que se usa en el estándar: la anotación `@Qualifier`. Con ella marcaremos el bean al definirlo y al inyectarlo, por ejemplo:

```
@Repository
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
@Qualifier("JDBC")
public class UsuariosDAOJDBC implements IUsuariosDAO {
    ...
}
```

```
@Service
public class UsuariosBOSimple implements IUsuariosBO {
  @Autowired
  @Qualifier("JDBC")
  IUsuariosDAO udao;
  ...
}
```

Al igual que vimos en el módulo de componentes web, podríamos crearnos nuestras propias anotaciones `@Qualifier` personalizadas, por ejemplo una `@JDBCDAO`.

### 1.3.3. Inyectando expresiones

Con la anotación `@Value`, podemos inyectar valores en variables miembro o parámetros de métodos. En su variante más simple este sería un valor constante, por ejemplo:

```
@Component MiBean {
  @Value(100)
  private creditoInicial;

}
```

Evidentemente inyectar un valor constante no es muy útil, es más sencillo hacer la inicialización directamente con código Java. Lo interesante de `@Value` es que podemos usar expresiones en **SpEL** (*Spring Expression Language*), un lenguaje de expresiones similar en concepto al JSP o JSF EL pero propio de Spring, y que nos permite, entre otras cosas:

- Realizar operaciones aritméticas y lógicas
- Llamar a métodos y acceder a propiedades de objetos. Podemos acceder a beans referenciándolos por su id.
- Hacer *matching* con expresiones regulares
- Acceso a las propiedades del sistema y a variables de entorno

A continuación se muestran algunos ejemplos de uso.

```
//acceso al bean con id "miConfig" y llamada al método "getLocale"
//(supuestamente este método devuelve el objeto Locale que aquí
inyectamos)
@Value("#{miConfig.defaultLocale}")
private Locale locale;

//otro acceso a un getter de un bean, ahora usando también operadores
lógicos
@Value("#{almacen.stock>0")
private boolean hayStock

//acceso a las propiedades del sistema.
//Las variables de entorno son accesibles con "systemEnvironment"
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
@Value("#{systemProperties['user.name']}")
private String userName;
```

El uso detallado de SpEL queda fuera del ámbito de estos apuntes introductorios. Se recomienda consultar la documentación de Spring para obtener más información sobre el lenguaje.

### 1.3.4. Dependencias de recursos externos

El API estándar de JavaEE para el acceso a recursos externos (conexiones con bases de datos, colas de mensajes, etc) es JNDI. Este API no se basa en inyección de dependencias, todo lo contrario: el objeto cliente del servicio es el que debe "hacer el esfuerzo" para localizar por él mismo los objetos de los que depende. La "solicitud" o búsqueda de dichos objetos se le pide a un servicio de directorio que tiene un registro de servicios por nombre. Spring hace lo posible por integrar su mecanismo estándar de dependencias con este modo de funcionamiento y nos permite un acceso relativamente sencillo a recursos JNDI

Primero hay que asociar el recurso con su nombre JNDI en el .xml de configuración. Lo más sencillo es usar el espacio de nombres `jee`. Para usar este espacio de nombres hay que definir el siguiente preámbulo en el XML de configuración (en negrita aparece la definición del espacio de nombres propiamente dicho)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns:jee="http://www.springframework.org/schema/jee"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans.xsd
            http://www.springframework.org/schema/jee
     http://www.springframework.org/schema/jee/spring-jee.xsd">
   ...
</beans>
```

Hacer que un DataSource cuyo nombre JNDI es `jdbc/MiDataSource` se "convierta" en un bean de Spring (y por tanto sea inyectable en otros) es muy sencillo con la etiqueta `jee:jndi-lookup`

```
<jee:jndi-lookup id="miBean" jndi-name="jdbc/MiDataSource"
       resource-ref="true"/>
```

Nótese que el atributo identificador (id) del nuevo bean es un valor requerido en la etiqueta XML, aunque es arbitrario. Simplemente debemos asegurarnos de que sea único.

Donde el atributo `resource-ref="true"` indica que el DataSource lo gestiona un servidor de aplicaciones y que por tanto al nombre JNDI del objeto hace falta precederlo de `java:comp/env/`

Podemos mejorar un poco el ejemplo: cuando se introducen valores en el XML de configuración susceptibles de cambios, como los nombres JNDI, es mejor externalizarlos,



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

ya que el XML es bastante crítico y no conviene andar editándolo sin necesidad. Por ello, Spring prevé la posibilidad de usar constantes, con la sintaxis `${nombre.constante}`, y tomar sus valores de ficheros `.properties` estándar. Por ejemplo:

```
<jee:jndi-lookup id="miBean" jndi-name="${datasource}"
        resource-ref="true"/>
<context:property-placeholder
        location="classpath:es/ua/jtech/ds.properties"/>
```

donde la etiqueta `property-placeholder` especifica la localización física del fichero `.properties`, en este caso, en el classpath en un fichero llamado `ds.properties` dentro de las carpetas `es/ua/jtech`.

Finalmente, podemos inyectar nuestro DataSource, ahora convertido en un bean de Spring y por tanto inyectable, usando `@Autowired` como de costumbre:

```
@Repository
public class UsuariosDAOJDBC implements IUsuariosDAO {
  @Autowired
  DataSource ds;
  ...
}
```

Spring es lo suficientemente "hábil" para deducir el tipo de un objeto a partir de su nombre JNDI, y por tanto sabrá que el recurso JNDI llamado jdbc/MiDataSource es un candidato válido para ser inyectado en esta variable.

## 1.4. Alternativas a las anotaciones para la configuración

Aunque es el tipo de configuración más usado actualmente, las anotaciones no son el único método de configuración en Spring. Se pueden definir beans con XML y también con código Java. Vamos a ver muy brevemente las ventajas de estas alternativas y su forma de uso.

### 1.4.1. Configuración en XML

En lugar de anotaciones, se puede hacer uso del XML de configuración para definir los beans. Esto tiene dos ventajas básicas: eliminar toda referencia a Spring de nuestro código, lo que mejora la portabilidad, y permitir la definición de más de un bean de la misma clase con distintas propiedades. No obstante, el XML resultante es bastante farragoso, por ejemplo, el caso del GestorUsuarios, un BO que depende del DAO UsuariosDAO se haría en notación XML:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- definimos un bean de la clase UsuariosDAO
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
         y le damos nombre -->
    <bean id="miUsuariosDAO"
        class="es.ua.jtech.spring.datos.UsuariosDAO">
    </bean>

    <!-- definimos otro bean de la clase UsuariosDAO
         pero con ámbito "prototype",
         solo para que veas que se pueden definir varios beans
         de la misma clase con propiedades distintas -->
    <bean id="otroUsuariosDAO"
          class="es.ua.jtech.spring.datos.UsuariosDAO"
                scope="prototype">
    </bean>

    <bean id ="miGestorUsuarios"
      class="es.ua.jtech.spring.negocio.GestorUsuarios">
        <!-- la propiedad "udao" referencia al bean antes definido -->
        <!-- Cuidado, la propiedad debe llamarse igual que en el fuente
Java -->
        <property name="udao" ref="miUsuariosDAO"/>
    </bean>
</beans>
```

Hay varias cosas a notar de este código: la más evidente es que la etiqueta `bean` es la que se usa para definir cada uno de los beans gestionados por el contenedor. Además, para especificar que una variable de un bean es otro bean que Spring debe instanciar se usa la etiqueta `property` con el atributo `ref` referenciando al bean.

Aunque por supuesto todo lo que hemos hecho con anotaciones se puede hacer con XML (y más cosas, además) no vamos a ver la sintaxis, salvo para ver cómo definir propiedades de los beans. Se recomienda consultar la documentación de Spring, en concreto el capítulo 3, *The IoC Container*.

**Podemos especificar valores iniciales para las propiedades de un bean**. Así podremos cambiarlos sin necesidad de recompilar el código. Lógicamente esto no se puede hacer con anotaciones sino que se hace en el XML. Las propiedades del bean se definen con la etiqueta `<property>`. Pueden ser Strings, valores booleanos o numéricos y Spring los convertirá al tipo adecuado, siempre que la clase tenga un método `setXXX` para la propiedad. Podemos convertir otros tipos de datos (fechas, expresiones regulares, URLs, ...) usando lo que en Spring se denomina un `PropertyEditor`. Spring incorpora varios predefinidos y también podemos definir los nuestros.

Por ejemplo, supongamos que tenemos un buscador de documentos `DocsDAO` y queremos almacenar en algún sitio las preferencias para mostrar los resultados. La clase Java para almacenar las preferencias sería un *JavaBean* común:

```
package es.ua.jtech.spring.datos;


public class PrefsBusqueda {
    private int maxResults;
    private boolean ascendente;
    private String idioma;

    //Aquí faltarían los getters y setters
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
    ...
}
```

No se muestra cómo se define la relación entre `DocsDAO` y `PrefsBusqueda`. Ya conocemos cómo hacerlo a partir de los apartados anteriores.

Los valores iniciales para las propiedades pueden configurarse en el XML dentro de la etiqueta `bean`

```
...
   <bean id="prefsBusqueda"
class="es.ua.jtech.spring.datos.PrefsBusqueda">
      <property name="maxResults" value="100"/>
      <property name="ascendente" value="true"/>
      <property name="idioma" value="es"/>
   </bean>
...
```

A partir de la versión 2 de Spring se añadió una forma alternativa de especificar propiedades que usa una sintaxis mucho más corta. Se emplea el espacio de nombres `http://www.springframework.org/schema/p`, que permite especificar las propiedades del bean como atributos de la etiqueta `bean`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
   <bean id="misPrefs" class="es.ua.jtech.spring.datos.PrefsBusqueda"
       p:maxResults="100" p:ascendente="true">
   </bean>
</beans>
```

Las propiedades también pueden ser colecciones: `Lists`, `Maps`, `Sets` o `Properties`. Supongamos que en el ejemplo anterior queremos una lista de idiomas preferidos:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
   <bean id="prefsBusqueda"
class="es.ua.jtech.spring.datos.PrefsBusqueda">
    <property name="listaIdiomas">
      <list>
       <value>es</value>
       <value>en</value>
      </list>
    </property>
    <!-- resto de propiedades -->
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
    ...
   </bean>
</beans>
```

Para ver cómo se especifican los otros tipos de colecciones, acudir a la documentación de referencia de Spring.

### 1.4.2. Configuración Java

El último "formato" que podemos usar para configurar los beans es el propio lenguaje Java. La idea básica es definir un método por cada bean y que éste devuelva el objeto instanciado con las propiedades y dependencias que deseemos. Aquí tenemos el ejemplo del `IUsuariosBO` y el `IUsuariosDAO` con configuración Java:

```
@Configuration
public class SampleConfig {
  @Bean
  public IUsuariosDAO udao() {
    return new UsuariosDAOJPA();
  }

  @Bean
  public IUsuariosBO ubo() {
    IUsuariosBO ubo = new UsuariosBOSimple();
    ubo.setCredito(100);
    ubo.setIUsuariosDAO(udao());
    return ubo;
  }
}
```

Como se ve, cada bean se define mediante un método anotado con `@Bean` y que devuelve una instancia del objeto deseado (simplemente usando `new()`). La clase que agrupa estos métodos se anota con `@Configuration`. Nótese además que:

- Spring toma como identificador del bean el nombre del método que lo genera. Por ejemplo, el identificador del bean de tipo `IUsuariosDAO` será "udao".
- Si queremos asignarle una propiedad al bean simplemente lo hacemos con código Java. En el ejemplo anterior, hemos supuesto que el `IUsuariosBO` tiene un *setter* para una propiedad llamada "credito".
- Igualmente, para establecer una dependencia entre beans lo hacemos con código Java convencional. Esto nos obliga a pasar el bean "colaborador" como un parámetro en el constructor o en un setter del bean "dependiente". Esto es lo que hemos hecho al construir el `IUsuariosBO`.

En el código de la configuración Java hay más cosas de las que se aprecian a simple vista. La más destacable es que Spring no ejecuta "textualmente" el código cada vez que se solicita un bean. Si lo hiciera, cada vez que le pidiéramos al contenedor un bean `IUsuariosDAO`, nos estaría dando una nueva instancia, mientras que ya hemos visto que en Spring por defecto los beans son *singletons*. ¿Qué sucede entonces?: lo que ocurre es que Spring intercepta la llamada al método `udao` y si ya se ha creado una instancia del bean devuelve la misma. Esto lo hace usando una técnica denominada programación



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

orientada a aspectos o AOP.

**La AOP y Spring**

La AOP o Programación Orientada a Aspectos es una tecnología básica en Spring y que explica cómo funcionan internamente casi todas las capacidades que ofrece el *framework*. La AOP permite interceptar la ejecución en ciertos puntos del código para ejecutar en ellos nuestras propias tareas antes o después del código original. En el caso de la configuración Java, antes de crear el bean se comprueba si existe ya otra instancia creada. Como se ha dicho, la AOP se usa de manera extensiva en Spring. Es la que permite que se gestionen de manera automática las transacciones, la seguridad, la cache, y otros muchos aspectos. Para saber algo más de qué es la AOP y cómo funciona en Spring puedes consultar el apéndice de estos apuntes.

¿Por qué puede interesarnos usar la configuración Java en nuestros proyectos?: frente a la basada en anotaciones tiene la ventaja de que la configuración está centralizada y por tanto es más fácil de entender y modificar. Frente al XML ofrece la ventaja de que al ser código Java podemos usar las capacidades de chequeo de código de nuestro IDE y del compilador para verificar que todo es correcto. Por ejemplo, en el XML no podemos saber si nos hemos equivocado en el identificador de un bean "colaborador" o en el nombre de la clase del bean. Si esto pasa en la configuración Java nos daremos cuenta porque el código no compilará.

Por supuesto esta es una introducción muy básica a la configuración Java, podemos hacer muchas más cosas como configurar distintos grupos de beans en distintas clases `@Configuration` e importar en cada `@Configuration` las configuraciones que necesitemos que sean visibles. Para más información,como siempre, consultar la documentación del framework.



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

# 2. Ejercicios del contenedor de beans de Spring

## 2.1. Configuración del proyecto

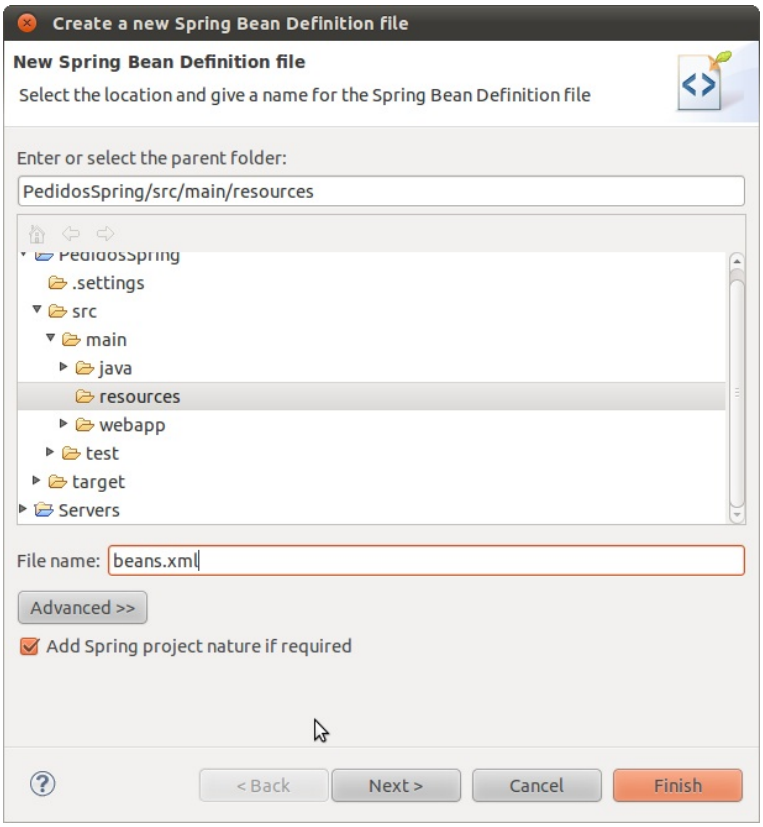
**SpringSource Tool Suite**

En el módulo de Spring usaremos como IDE SpringSource Tool Suite, que es una versión de Eclipse con un conjunto de plugins instalados que facilitan bastante el trabajo con Spring. Os podéis bajar la versión actual de STS de la web de SpringSource (en la máquina virtual no es necesario, ya la tenéis preinstalada). Como en el módulo vamos a desarrollar básicamente aplicaciones web necesitamos un servidor donde desplegarlas. STS viene ya con una instancia preconfigurada del "VMWare vFabric tc server" que es una versión de Tomcat modificada por SpringSource. No obstante, vamos a usar Tomcat tal cual ya que es algo más ligero. Por tanto tendréis que crea una nueva instancia del servidor `File > New > Other... > Server`. Recordad que Tomcat está instalado en `/opt/apache-tomcat-7.0.29`

1. En las plantillas de la sesión hay un proyecto Spring ya creado que usa Maven. Para importarlo en STS usar (como siempre) `File > Import... > Existing Maven Projects`. En el "pom.xml" se incluye una dependencia del módulo "spring-context" porque éste, a su vez, depende del resto de módulos básicos de Spring, que así Maven incluirá automáticamente. Además, para poder instanciar beans desde la capa web necesitamos el módulo "spring-web".
2. Lo primero que vamos a hacer es **Crear el fichero XML de configuración de Spring**. La localización del fichero es libre, pero nosotros vamos a colocarlo en la carpeta "src/main/resources". Hacer clic con botón derecho sobre esta carpeta, y en el menú contextual, seleccionar "New > Spring Bean Configuration File". Darle como nombre "beans.xml". Asegurarse de que está marcada la casilla de "Add Spring Project Nature if required", que activará el soporte de Spring para el proyecto. **No pulsar sobre Finish, sino sobre Next**, para poder ir al siguiente paso del asistente.



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

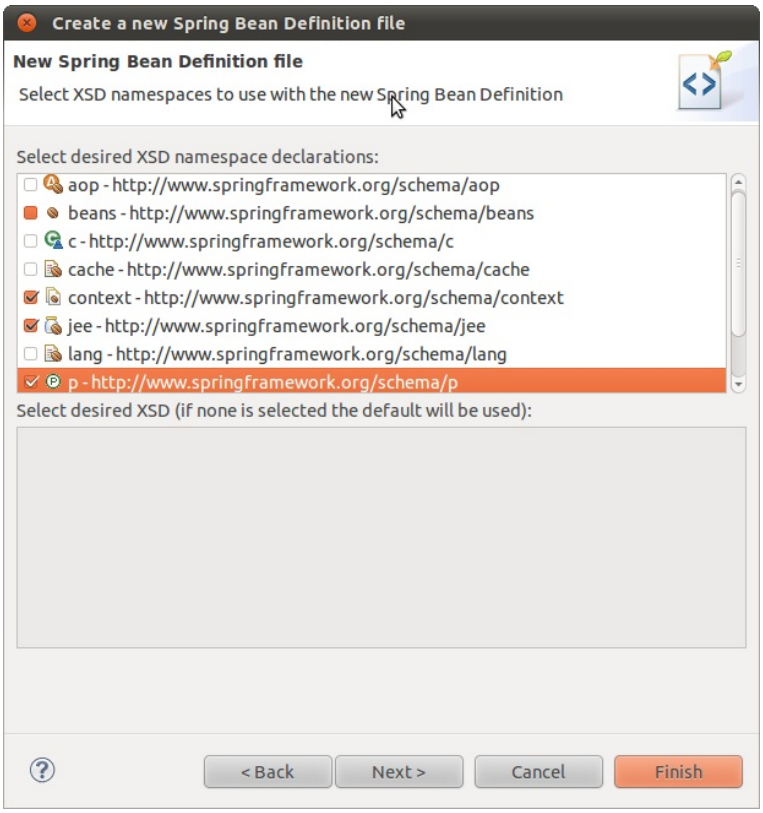



Crear beans.xml. Paso 1

En el segundo paso marcamos los espacios de nombres que necesitamos: "context", "jee" y "p".



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.




Crear beans.xml. Paso 2

3. Ahora tenemos que **referenciar el fichero de configuración de Spring en el web.xml** (recordad que está en src/main/webapp/WEB-INF), para que Spring arranque cuando arranque la aplicación web. Introduciremos en el web.xml el siguiente código:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:beans.xml</param-value>
</context-param>

<listener>
    <listener-class>
    org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

Vamos a hacer una pequeña aplicación con acceso a base de datos, por lo que tenemos que preparar la conexión y la propia base de datos:

1. Asegurarse de que el .jar con el driver de mysql está en el directorio "lib" de Tomcat (/opt/apache-tomcat-7.0.29/lib). Si no lo está, copiarlo allí (se incluye en las plantillas de la sesión).
2. Recuerda que en Tomcat la configuración del DataSource se hace en el context.xml



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

(en `src/main/webapp/META-INF`). Este archivo ya está creado, no tienes que modificar nada.

3. Crear la propia base de datos: en las plantillas de la sesión se incluye un script SQL con la base de datos. Se puede ejecutar con el programa "MySQL Query Browser" o bien desde un terminal con la orden

```
(nos pedirá el password de root de mysql)
mysql -u root -p < pedidos.sql
```

## 2.2. Estructura de la aplicación

La aplicación servirá para hacer pedidos de productos, funcionando según las siguientes reglas:

- La capa de negocio debe chequear que el número de unidades pedidas no supere un cierto límite. Si lo hace, el pedido no se puede procesar automáticamente, por lo que se genera una excepción.
- En la capa de acceso a datos cuando se haga un pedido correctamente se insertará la información en la tabla "pedidos".

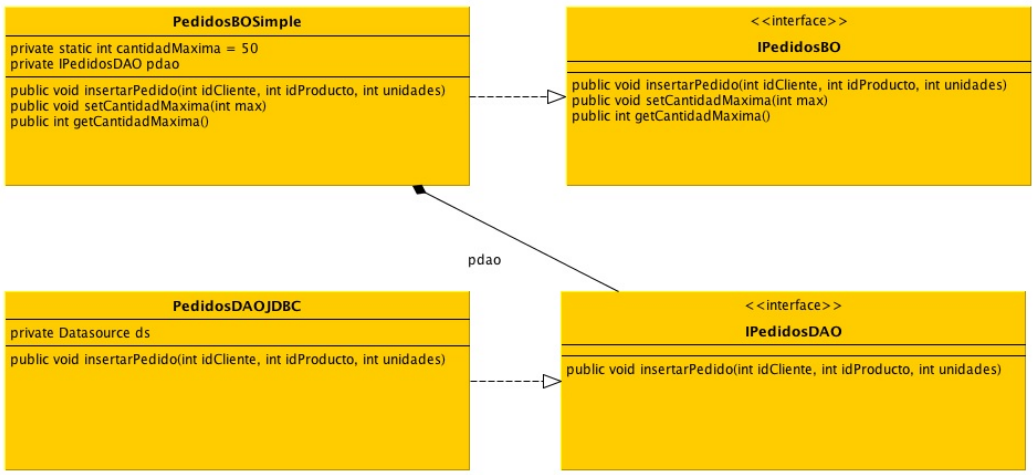



diagrama UML de la aplicación de pedidos

En la figura anterior se muestra el diagrama UML de la aplicación

## 2.3. Crear la capa de negocio (1 punto)

Vamos a empezar por crear la capa de negocio, aunque de momento no funcionará del todo al faltar el DAO.

1. Crear el interfaz `IPedidosBO` y la clase que lo implementa `PedidosBOSimple` en el paquete `es.ua.jtech.spring.negocio`. Usad como guía el diagrama UML para ver la signatura de los métodos.



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

- Por el momento no hay DAO, así que no es necesario que definas el campo "pdao" del objeto.
- Define el campo "cantidadMaxima" como static con un valor de 50.
- Define un constructor y en él simplemente imprime un mensaje en la consola. Así podrás ver el momento en que Spring crea el bean y si lo crea más de una vez.
- Implementa el método "insertarPedido": debe comprobar que la cantidad pedida no supera "cantidadMaxima". Si se supera, se lanzará una `PedidosException` (ya incluida en la plantilla). En caso contrario se llamaría al DAO, pero esta parte todavía no la vamos a hacer, simplemente si no se supera la cantidad máxima imprime un mensaje en la consola con "Pedido realizado".

2. Convertir la clase `PedidosBOSimple` en un bean de Spring añadiéndole la anotación `@Service`.
3. Modificar el fichero de configuración XML de Spring (beans.xml) para que busque las anotaciones Spring en el paquete que queremos, y en sus subpaquetes (es decir, en `es.ua.jtech.spring`). Recuerda que se usa la etiqueta <context-component-scan/>.

**Ayudas del STS**

Usa el autocompletar de STS siempre que puedas, por ejemplo en el valor del atributo "base-package", para no tener que teclear manualmente el nombre del paquete y no cometer errores. Es posible que tengas que introducir la primera letra del nombre del paquete para que el autocompletar funcione.

4. En el servlet `HacerPedido` del paquete `es.ua.jtech.spring.web` hay que obtener un bean de tipo `IPedidosBO` y llamar al método insertarPedido con los valores de idCliente, idProducto y unidades.
5. Comprueba que cada vez que insertas un pedido de menos de 50 unidades el servlet muestra el mensaje, y que cuando se piden más salta la excepción. Finalmente comprueba que Spring solo crea una instancia del bean (solo se llama una vez al constructor), aunque obtengas varias veces el bean haciendo varios pedidos. Esto sucede porque usamos el ámbito por defecto, o sea "singleton". Además la creación se hace por defecto cuando se arranca el contenedor de Spring, no cuando se hace el primer pedido.

## 2.4. Crear la capa de acceso a datos y enlazarla con la de negocio (1.5 puntos)

1. en el "beans.xml", configura el Datasource para acceder a la BD con Spring. Consulta los apuntes o transparencias para ver el uso de la etiqueta "jee:jndi-lookup". El id que le des al bean es indiferente si luego usas @Autowired para acceder a él.
2. Crea el interface `IPedidosDAO` y la clase `PedidosDAOJDBC` en el paquete `es.ua.jtech.spring.datos`.
   - Debes convertir la clase `PedidosDAOJDBC` en un bean de Spring anotándola con `@Repository`
   - Anota la variable miembro `ds` con `@Autowired` para que Spring busque el DataSource en el "beans.xml".
   - El codigo del método `insertarPedido` de `PedidosDAOJDBC` lo podéis tomar de



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

las plantillas de la sesión para ahorrar tiempo

3. Modifica el `PedidoBOSimple` para que haga uso del DAO
   - Debes definir en `PedidoBOSimple` un campo de tipo IPedidosDAO y anotarlo con @Autowired, para que Spring resuelva e instancie automáticamente la dependencia

```
@Autowired
IPedidosDAO pdao;
```

   - Haz uso de este DAO en el insertarPedido. Es decir, el gestor de pedidos debe hacer uso del objeto "pdao" para insertar el pedido en la BD. Aunque el DAO devuelve el id del pedido, por el momento no es necesario que hagas nada con él. **No hagas un new() para inicializar "pdao"**. Si todo está correctamente configurado Spring inicializará esta variable, ya que la has marcado con @Autowired.
4. Finalmente, comprueba que ahora cuando haces un pedido a través del servlet `HacerPedido` éste se inserta en la base de datos.

## 2.5. Configurar beans en el XML (0.5 puntos)

Tal y como está el código de `PedidoBOSimple`, hay que cambiar el código fuente para cambiar la cantidad máxima de unidades por pedido (50). Es mucho mejor usar Spring para externalizar este valor a través del "beans.xml" y poder cambiar la cantidad sin recompilar el código. El problema es que entonces la anotación @Service de `PedidoBOSimple` debe ser sustituida por una configuración completa del bean en el XML.

1. Comenta la anotación @Service de `PedidoBOSimple`.
2. Usando la etiqueta "bean", configura en el "beans.xml" un bean de la clase `PedidoBOSimple`. Cuidado: debes poner el nombre completo de la clase, incluyendo "packages".
3. Consulta los apuntes o transparencias para ver cómo fijar en el XML el valor de la propiedad "cantidadMaxima".
4. Comprueba que se inicializa la propiedad correctamente imprimiendo un mensaje desde el método "setCantidadMaxima", al que Spring llamará automáticamente para darle valor a la propiedad.



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

# 3. Acceso a datos

En este tema veremos las facilidades que proporciona Spring para implementar nuestros objetos de la capa de acceso a datos. Veremos que nos permite simplificar el código, reduciendo el código repetitivo, y uniformizar el tratamiento independientemente de la implementación subyacente (JPA, JDBC, ...). Finalmente abordaremos la transaccionalidad, que es un aspecto íntimamente ligado con el acceso a datos, aunque su gestión no suele estar localizada en el código de los DAOs sino en la capa de negocio.

Debido al auge en los últimos años del *big data* en la web, la "punta de lanza" de Spring en la capa de acceso a datos se ha movido más hacia la integración en Spring de bases de datos no relacionales (NoSQL). El proyecto Spring que se ocupa de estos aspectos es Spring Data, que, no obstante, queda fuera del ámbito de estos apuntes.

## 3.1. La filosofía del acceso a datos en Spring

Spring proporciona básicamente dos ventajas a la hora de dar soporte a nuestros DAOs:

- Simplifica las operaciones de acceso a datos en APIs tediosos de utilizar como JDBC, proporcionando una capa de abstracción que reduce la necesidad de código repetitivo. Para ello se usan los denominados *templates*, que son clases que implementan este código, permitiendo que nos concentremos en la parte "interesante".
- Define una rica jerarquía de excepciones que modelan todos los problemas que nos podemos encontrar al operar con la base de datos, y que son independientes del API empleado.

Un **template** de acceso a datos en Spring es una clase que encapsula los detalles más tediosos (como por ejemplo la necesidad de abrir y cerrar la conexión con la base de datos en JDBC), permitiendo que nos ocupemos únicamente de la parte de código que hace realmente la tarea (inserción, consulta, ...)

Spring ofrece diversas *templates* entre las que elegir, dependiendo del API de persistencia a emplear. Dada la heterogeneidad de los distintos APIs La implementación del DAO variará según usemos un API u otro , aunque en todos ellos Spring reduce enormemente la cantidad de código que debemos escribir, haciéndolo más mantenible.

Por otro lado, en APIs de acceso a datos como JDBC hay dos problemas básicos con respecto a la **gestión de excepciones**:

- Hay muy pocas excepciones distintas definidas para acceso a datos. Como consecuencia, la más importante, `SQLException`, es una especie de "chica para todo". La misma excepción se usa para propósitos tan distintos como: "no hay conexión con la base de datos", "el SQL de la consulta está mal formado" o "se ha producido una violación de la integridad de los datos". Esto hace que para el desarrollador sea tedioso escribir código que detecte adecuadamente el problema. Herramientas como



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

Hibernate tienen una jerarquía de excepciones mucho más completa, pero son excepciones propias del API, y referenciarlas directamente va a introducir dependencias no deseadas en nuestro código.
- Las excepciones definidas en Java para acceso a datos son *comprobadas*. Esto implica que debemos poner `try/catch` o `throws` para gestionarlas, lo que inevitablemente llena *todos* los métodos de acceso a datos de bloques de gestión de excepciones. Está bien obligar al desarrollador a responsabilizarse de los errores, pero en acceso a datos esta gestión se vuelve repetitiva y propensa a fallos, descuidos o a caer en "malas tentaciones" (¿quién no ha escrito *nunca* un bloque `catch` vacío?). Además, muchos métodos de los DAO generalmente poco pueden hacer para recuperarse de la mayoría de excepciones (por ejemplo, "violación de la integridad"), lo que lleva al desarrollador a poner también `throws` de manera repetitiva y tediosa.

La solución de Spring al primer problema es la definición de una completa jerarquía de excepciones de acceso a datos. Cada problema tiene su excepción correspondiente, por ejemplo `DataAccessResourceFailureException` cuando no podemos conectar con la BD, `DataIntegrityViolationException` cuando se produce una violación de integridad en los datos, y así con otras muchas. Un aspecto fundamental es que estas excepciones *son independientes del API usado para acceder a los datos*, es decir, se generará el mismo `DataIntegrityViolationException` cuando queramos insertar un registro con clave primaria duplicada en JDBC que cuando queramos persistir un objeto con clave duplicada en JPA. La raíz de esta jerarquía de excepciones es `DataAccessException`.

En cuanto a la necesidad de gestionar las excepciones, Spring opta por eliminarla haciendo que todas las excepciones de acceso a datos sean *no comprobadas*. Esto libera al desarrollador de la carga de los `try-catch/throws` repetitivos, aunque evidentemente no lo libera de su responsabilidad, ya que las excepciones tendrán que gestionarse en algún nivel superior.

## 3.2. Uso de JDBC

JDBC sigue siendo un API muy usado para el acceso a datos, aunque es tedioso y repetitivo. Vamos a ver cómo soluciona Spring algunos problemas de JDBC, manteniendo las ventajas de poder trabajar "a bajo nivel" si así lo deseamos. Probablemente las ventajas quedarán más claras si primero vemos un ejemplo con JDBC "a secas" y luego vemos el mismo código usando las facilidades que nos da Spring. Por ejemplo, supongamos un método que comprueba que el login y el password de un usuario son correctos, buscándolo en la base de datos con JDBC:

```
private String SQL ="select * from usuarios where login=? and password=?";

public UsuarioTO login(String login, String password) throws DAOException
{
    Connection con=null;
    try {
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
        con = ds.getConnection();
           PreparedStatement ps = con.prepareStatement(SQL);
        ps.setString(1, login);
        ps.setString(2, password);
        ResultSet rs = ps.executeQuery();
        if (rs.next()) {
            UsuarioTO uto = new UsuarioTO();
            uto.setLogin(rs.getString("login"));
            uto.setPassword(rs.getString("password"));
            uto.setFechaNac(rs.getDate("fechaNac"));
            return uto;
        }
        else
            return null;
    } catch(SQLException sqle) {
        throw new DAOException(sqle);
    }
    finally {
        if (con!=null) {
            try {
                con.close();
            }
            catch(SQLException sqle2) {
                throw new DAOException(sqle2);
            }
        }
    }
}
```

Se destacan las líneas de código que hacen realmente el trabajo de buscar el registro y devolver la información. El resto es simplemente la infraestructura necesaria para poder hacer el trabajo y gestionar los errores, y que, curiosamente *ocupa más líneas que el código "importante"*. Evidentemente la gestión de errores se habría podido "simplificar" poniendo en la cabecera del método un `throws SQLException`, pero entonces ya estaríamos introduciendo dependencias del API JDBC en la capa de negocio.

Veamos cómo nos puede ayudar Spring a simplificar nuestro código, manteniendo la flexibilidad que nos da SQL. El primer paso será elegir el *template* apropiado.

### 3.2.1. JDBCTemplate

Como ya hemos dicho, los *templates* son clases que encapsulan el código de gestión de los detalles "tediosos" del API de acceso a datos. En Spring, para JDBC tenemos varios templates disponibles, según queramos hacer consultas simples, con parámetros con nombre,... Vamos a usar aquí `JdbcTemplate`, que aprovecha las ventajas de Java 5 (autoboxing, genéricos, ...) para simplificar las operaciones. El equivalente si no tenemos Java 5 sería `JdbcTemplate`, que tiene una sintaxis mucho más complicada.

Lo primero que necesitamos es instanciar el *template*. El constructor de `JdbcTemplate` necesita un `DataSource` como parámetro. Como se vio en el tema anterior, los `DataSource` se pueden definir en el fichero XML de los beans, gracias al espacio de nombres `jee`:

```
<jee:jndi-lookup id="ds" jndi-name="jdbc/MiDataSource"
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
resource-ref="true"/>
```

Con lo que el `DataSource` se convierte en un bean de Spring llamado `ds`. La práctica habitual es inyectarlo en nuestro DAO y con él inicializar el *template*, que guardaremos en el DAO:

```
import org.springframework.jdbc.core.simple.JbcTemplate;
import org.springframework.stereotype.Repository;
//Resto de imports...
...

@Repository("JDBC")
public class UsuariosDAOJDBC implements IUsuariosDAO {
    private JdbcTemplate jdbcTemplate;

    @Autowired
    public void setDataSource(DataSource ds) {
        this.jdbcTemplate = new JdbcTemplate(ds);
    }

    ...
}
```

Recordemos que la anotación `@Repository` se usa para definir un DAO. Recordemos también que `@Autowired` inyecta la dependencia buscándola por tipo. En este caso no hay ambigüedad, ya que solo hemos definido un `DataSource`.

**SimpleJdbcTemplate**

Antes de Spring 3, la funcionalidad más "simple" de JDBC en Spring la implementaba `SimpleJdbcTemplate`, mientras que el API de `JdbcTemplate` era más complicado y por tanto solo recomendable para operaciones más complejas que las que vamos a ver aquí. A partir de la versión 3, `SimpleJdbcTemplate` está *deprecated*.

### 3.2.2. Consultas de selección

Normalmente en un SELECT se van recorriendo registros y nuestro DAO los va transformando en objetos Java que devolverá a la capa de negocio. En Spring, el trabajo de tomar los datos de un registro y empaquetarlos en un objeto lo hace `RowMapper`. Este es un interface, por lo que nuestro trabajo consistirá en escribir una clase que lo implemente. Realmente el único método estrictamente necesario es `mapRow`, que a partir de un registro debe devolver un objeto. En nuestro caso podría ser algo como:

```
//esto podría también ser private y estar dentro del DAO
//ya que solo lo necesitaremos dentro de él
public class UsuarioTOMapper implements RowMapper<UsuarioTO> {

    public UsuarioTO mapRow(ResultSet rs, int numRow) throws SQLException
{
        UsuarioTO uto = new UsuarioTO();
        uto.setLogin(rs.getString("login"));
        uto.setPassword(rs.getString("password"));
        uto.setFechaNac(rs.getDate("fechaNac"));
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
            return uto;
        }
}
```

Ahora solo nos queda escribir en el DAO el código que hace el SELECT:

```
private static final String LOGIN_SQL = "select * " +
 "from usuarios where login=? and password=?";

public UsuarioTO login(String login, String password) {
    UsuarioTOMapper miMapper = new UsuarioTOMapper();

    return this.jdbcTemplate.queryForObject(LOGIN_SQL, miMapper,
                                            login, password);
}
```

Como se ve, no hay que gestionar la conexión con la base de datos, preocuparse del `Statement` ni nada parecido. El *template* se ocupa de estos detalles. El método `queryForObject` hace el SELECT y devuelve un `UsuarioTO` ayudado del *mapper* que hemos definido antes. Simplemente hay que pasarle el SQL a ejecutar y los valores de los parámetros.

Tampoco hay gestión de excepciones, porque Spring captura todas las `SQLException` de JDBC y las transforma en excepciones no comprobadas. Por supuesto, eso no quiere decir que no podamos capturarlas en el DAO si así lo deseamos. De hecho, en el código anterior hemos cometido en realidad un "descuido", ya que podría no haber ningún registro como resultado del SELECT. Para Spring esto es una excepción del tipo `EmptyResultDataAccessException`. Si queremos seguir la misma lógica que en el ejemplo con JDBC, deberíamos devolver `null` en este caso.

```
private static final String LOGIN_SQL = "select * " +
        "from usuarios where login=? and password=?";

public UsuarioTO login(String login, String password) {
   UsuarioTOMapper miMapper = new UsuarioTOMapper();

   try {
       return this.jdbcTemplate.queryForObject(LOGIN_SQL, miMapper,
                                               login, password);
   }
   catch(EmptyResultDataAccessException erdae) {
       return null;
   }
}
```

La amplia variedad de excepciones de acceso a datos convierte a Spring en un *framework* un poco "quisquilloso" en ciertos aspectos. En un `queryForObject` Spring espera obtener *un registro y sólo un registro*, de modo que se lanza una excepción si no hay resultados, como hemos visto, pero también si hay más de uno: `IncorrectResultSizeDataAccessException`. Esto tiene su lógica, ya que `queryForObject` solo se debe usar cuando esperamos como máximo un registro. Si el SELECT pudiera devolver más de un resultado, en lugar de llamar a `queryForObject`, emplearíamos **`query`**, que usa los mismos parámetros, pero devuelve una lista de objetos.



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

### 3.2.3. Consultas de actualización

Las actualizaciones se hacen con el método `update` del *template*. Por ejemplo, aquí tenemos el código que da de alta a un nuevo usuario:

```
private static final String REGISTRAR_SQL =
        "insert into usuarios(login, password, fechaNac) values (?,?,?)";

public void registrar(UsuarioTO uto) {
   this.jdbcTemplate.update(REGISTRAR_SQL, uto.getLogin(),
             uto.getPassword(), uto.getFechaNac());
}
```

## 3.3. Uso de JPA

Veamos qué soporte ofrece Spring al otro API de persistencia que hemos visto durante el curso: JPA. El caso de JPA es muy distinto al de JDBC, ya que es de mucho más alto nivel y más conciso que este último. Por eso, aunque Spring implementa un *template* para JPA, también se puede usar directamente el API sin escribir mucho más código. Además otra decisión que debemos tomar es quién gestionará los Entity Managers, si lo haremos "manualmente", al estilo Java SE, o bien a través del servidor de aplicaciones.

### 3.3.1. Formas de acceder a la "EntityManagerFactory"

Para poder trabajar con JPA lo primero que necesitamos es una `EntityManagerFactory` a través de la que crear los *Entity Managers*. Spring nos da tres posibilidades para acceder a ella:

- Como se hace en Java SE. Es decir, Entity Managers gestionados por la aplicación. En este caso, la mayor parte de la configuración se hace en el persistence.xml. Esta forma no se recomienda en general, solo para pruebas.
- Usando el soporte del servidor de aplicaciones. Evidentemente esto no funcionará en servidores que no tengan soporte JPA, como Tomcat, pero es la forma recomendada de trabajo en los que sí lo tienen. El acceso a la factoría se hace con JNDI.
- Usando un soporte propio de Spring. Esto funciona incluso en servidores sin soporte "nativo", como Tomcat. La configuración se hace sobre todo en el fichero de configuración de beans de Spring. El problema es que depende del servidor y también de la implementación JPA que estemos usando, pero está explicada con detalle en la documentación de Spring para varios casos posibles (sección 7.8.4.5).

Estas dos últimas posibilidades serían lo que en JPA se conoce como Entity Managers gestionados por el contenedor, con la diferencia de que una de ellas es una implementación nativa y la otra proporcionada por Spring. En cualquier caso, en realidad como desarrolladores todo esto nos da casi igual: elijamos la opción que elijamos Spring va a gestionar la `EntityManagerFactory` por nosotros. Esta se implementa como un bean de Spring, que podemos inyectar en nuestro código usando la anotación estándar de



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

`@PersistenceUnit`. También podemos inyectarlo con `@Autowired` o `@Resource`, como vimos en el tema anterior. Dicho bean debemos definirlo en el fichero XML de configuración. Aquí tenemos un ejemplo usando JPA gestionado por el contenedor pero en Tomcat (es decir, gestionado en realidad por Spring)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd
         http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="es.ua.jtech"/>

    <jee:jndi-lookup id="miDS" jndi-name="jdbc/MiDataSource"
       resource-ref="true"/>

    <bean id="miEMF" class="org.springframework.orm.jpa.
LocalContainerEntityManagerFactoryBean">
      <property name="dataSource" ref="miDS"/>
      <property name="jpaVendorAdapter">
         <bean class="org.springframework.orm.jpa.vendor.
                        HibernateJpaVendorAdapter">
               <property name="showSql" value="true"/>
               <property name="generateDdl" value="true"/>
               <property name="databasePlatform"
                          value="org.hibernate.dialect.HSQLDialect"/>
        </bean>
      </property>
    </bean>
</beans>
```

Como vemos el bean es una clase propia de Spring, de ahí que no podamos definirlo mediante anotaciones.El identificador del bean (miEMF) es arbitrario e irrelevante para nuestro ejemplo. Especificamos dos propiedades del bean: el datasource que se usará para conectar con la BD (aquí definido con JNDI) y la implementación JPA que estamos usando, en nuestro caso Hibernate. Finalmente, la etiqueta "component-scan" es necesaria para que Spring reconozca y procese las anotaciones `@PersistenceUnit` y `@PersistenceContext`.

**Cuidado con la implementación JPA**

En nuestro caso la configuración se simplifica porque usamos Hibernate, pero otras implementaciones JPA requieren *load time weaving*, una técnica que manipula el *bytecode* de las clases JPA en tiempo de ejecución y que es necesaria para que funcione la gestión por el contenedor. Si usáramos otra implementación JPA para la que fuera necesario el *load time weaving* habría que configurarlo, configuración que es además dependiente del servidor web en que hagamos el despliegue. Consultad la documentación de Spring para más información.

Con esto ya podemos inyectar el EntityManager o bien la factoría de EntityManagers en nuestro código:



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
package es.ua.jtech.datos;

//faltan los import...
@Repository("JPA")
public class UsuariosDAOJPA implements IUsuariosDAO {

    @PersistenceContext
    EntityManager em;

    public UsuarioTO login(String login, String password) {
        UsuarioTO uto = em.find(UsuarioTO.class, login);
        if (uto!=null && password.equals(uto.getPassword()))
            return uto;
        else
            return null;
    }
}
```

### 3.3.2. JpaTemplate

Esta clase facilita el trabajo con JPA, haciéndolo más sencillo. No obstante, al ser JPA un API relativamente conciso, no es de esperar que ahorremos mucho código. De hecho, los propios diseñadores de Spring recomiendan usar directamente el API JPA para aumentar la portabilidad del código. No obstante, vamos a verlo brevemente.

Aquí tenemos el esqueleto de una nueva implementación de `IUsuariosDAO` usando ahora `JpaTemplate` en lugar de `JdbcTemplate`:

```
package es.ua.jtech.spring.datos;

import es.ua.jtech.spring.dominio.UsuarioTO;
import javax.persistence.EntityManagerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.orm.jpa.JpaTemplate;
import org.springframework.stereotype.Repository;

@Repository("JPATemplate")
public class UsuariosDAOJPATemplate implements IUsuariosDAO {
    private JpaTemplate template;

    @Autowired
    public void setEntityManagerFactory(EntityManagerFactory emf) {
        this.template = new JpaTemplate(emf);
    }

    //Falta la implementación de este método
    public UsuarioTO login(String login, String password) {
        ...
    }

    //Falta la implementación de este método
    public void registrar(UsuarioTO uto) {
        ...
    }
```

Como se ve, para instanciar un `JpaTemplate` necesitamos un `EntityManagerFactory` (de ahí el trabajo que nos habíamos tomado en la sección anterior). En este caso usamos



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

la anotación `@Autowired` para que Spring resuelva automáticamente la dependencia y la inyecte a través del *setter*. También podríamos haber usado `@PersistenceUnit`, que al ser estándar hace nuestro código más portable. Una vez creado el Template, se puede reutilizar en todos los métodos ya que es *thread-safe*, al igual que en JDBC.

Ahora veamos cómo implementar los métodos del DAO. En lugar de ver sistemáticamente el API de `JpaTemplate`, nos limitaremos a mostrar un par de ejemplos. Primero, una consulta de selección:

```
private static String LOGIN_JPAQL = "SELECT u FROM UsuarioTO u
                                     WHERE u.login=?1 AND u.password=?2";
public UsuarioTO login(String login, String password) {
   List<UsuarioTO> lista;
   lista = this.template.find(LOGIN_JPAQL, login, password);
   return (UsuarioTO) lista.get(0);
}
```

Obsérvese que en el ejemplo anterior **no es necesario instanciar ni cerrar ningún Entity Manager**, ya que la gestión la lleva a cabo el *template*. En cuanto al API de acceso a datos, como se ve, el método `find` nos devuelve una lista de resultados y nos permite pasar parámetros a JPAQL gracias a los *varargs* de Java 5. En el ejemplo anterior hemos "forzado un poco" el API de `JpaTemplate` ya que `find` devuelve siempre una lista y en nuestro caso está claro que no va a haber más de un objeto como resultado. Se ha hecho así para mantener el paralelismo con el ejemplo JDBC, aunque aquí quizá lo más natural sería buscar por clave primaria. Tampoco se han tratado adecuadamente los errores, como que no haya ningún resultado en la lista.

Otros métodos de `JpaTemplate` nos permiten trabajar con parámetros con nombre y con *named queries*.

Las actualizaciones de datos no ofrecen gran ventaja con respecto al API directo de JPA, salvo la gestión automática del Entity Manager, por ejemplo:

```
public void registrar(UsuarioTO uto) {
   this.template.persist(uto);
}
```

**¡Cuidado con la transaccionalidad!**

Spring gestiona automáticamente los Entity Manager en función de la transaccionalidad de los métodos. Si no declaramos ningún tipo de transaccionalidad, como en el ejemplo anterior, podemos encontrarnos con que al hacer un `persist` no se hace el `commit` y el cambio no tiene efecto. En el último apartado del tema veremos cómo especificar la transaccionalidad, pero repetimos que es importante darse cuenta de que *si no especificamos transaccionalidad, la gestión automática de los entity manager no funcionará adecuadamente*.

Una ventaja del uso de *templates* es la jerarquía de excepciones de Spring. En caso de usar directamente el API JPA nos llegarán las propias de la implementación JPA que estemos usando. Si queremos que Spring capture dichas excepciones y las transforme en



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

excepciones de Spring, debemos definir un bean de la clase `PersistenceExceptionTranslationPostProcessor` en el XML de definición de beans:

```
<bean class="org.springframework.dao. <!--Esto debería estar en 1 sola
línea-->
annotation.PersistenceExceptionTranslationPostProcessor"/>
```

Como se ve, lo más complicado de la definición anterior ¡es el nombre de la clase!

## 3.4. Transaccionalidad declarativa

Abordamos aquí la transaccionalidad porque es un aspecto íntimamente ligado al acceso a datos, aunque se suele gestionar desde la capa de negocio en lugar de directamente en los DAOs. Vamos a ver qué facilidades nos da Spring para controlar la transaccionalidad de forma declarativa.

### 3.4.1. El "Transaction Manager"

Lo primero es escoger la estrategia de gestión de transacciones: si estamos en un servidor de aplicaciones podemos usar JTA, pero si no, tendremos que recurrir a la implementación nativa del API de acceso a datos que estemos usando. Spring implementa clases propias para trabajar con JTA o bien con transacciones JDBC o JPA. Estas clases gestionan las transacciones y se denominan "Transaction Managers". Necesitaremos definir uno de ellos en el fichero XML de definición de beans.

Por ejemplo, en el caso de estar usando JDBC sin soporte JTA, como en Tomcat, necesitaríamos una implementación de Transaction Manager que referencia un DataSource.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jee="http://www.springframework.org/schema/jee"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/jee
        http://www.springframework.org/schema/jee/spring-jee.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd">


    <!-- el DataSource que usará el Transaction Manager -->
    <jee:jndi-lookup id="miDataSource" jndi-name="DStestSpring"
resource-ref="true" />

    <!-- Elegimos el tipo apropiado de "Transaction Manager" (JDBC) -->
    <bean id="miTxManager"
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="miDataSource"/>
    </bean>

    <!-- Decimos que para este Transaction Manager vamos a usar
anotaciones -->
    <tx:annotation-driven transaction-manager="miTxManager"/>
 </beans>
```

### 3.4.2. La anotación @Transactional

Para entender mejor el uso de esta anotación, vamos a plantear un ejemplo. Supongamos que al registrar un nuevo usuario, además de introducir sus datos en la tabla de usuarios, también lo debemos dar de alta en otra tabla para enviarle publicidad, pero si no es posible este alta de publicidad por lo que sea (dirección de email no válida, digamos) hay que anular toda la operación. Supongamos que al detectar el error nuestro DAO lanzaría un `DataAccessException` que, recordemos, es la raíz de la jerarquía de excepciones de acceso a datos en Spring.

Colocaremos `@Transactional` delante de los métodos que queramos hacer transaccionales. Si la colocamos delante de una clase, estamos diciendo que **todos** sus métodos deben ser transaccionales. En nuestro caso:

```
//Faltan los imports, etc.
...
@Service
public class GestorUsuarios {
  @Autowired
  @Qualifier("JPA")
  private IUsuariosDAO udao;

  public void setUdao(IUsuariosDAO udao) {
      this.udao = udao;
  }

  @Transactional
  public void registrar(UsuarioTO uto) {
      udao.registrar(uto);
      udao.altaPublicidad(uto);
  }
}
```

El comportamiento por defecto de `@Transactional` es **realizar un *rollback* si se ha lanzado alguna excepción no comprobada**. Recordemos que, precisamente, `DataAccessException` era de ese tipo. Por tanto, se hará automáticamente un *rollback* en caso de error.

**¡Cuidado con los métodos no públicos!**

Todos los métodos que deseamos hacer transaccionales deben ser públicos, no es posible usar `@Transactional` en métodos `protected` o `private`. La razón es que cuando hacemos un método transaccional y lo llamamos desde cualquier otra clase quien está recibiendo la llamada en realidad es el gestor de transacciones. El gestor comienza y acaba las transacciones y "entre medias" llama a nuestro método de acceso a datos, pero eso no lo podrá hacer si este no es



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

`public`. Por la misma razón, la anotación no funcionará si el método transaccional es llamado desde la misma clase que lo define, aunque esto último se puede solucionar haciendo la configuración adecuada.

`@Transactional` tiene varias implicaciones por defecto (aunque son totalmente configurables, como ahora veremos):

- La propagación de la transacción es REQUIRED. Esto significa que se requiere una transacción abierta para que el método se ejecute en ella. Si no hubiera ninguna, Spring automáticamente la crearía. Esto funciona igual que los EJBs del JavaEE estándar.

**Transaccionalidad declarativa en JavaEE**

En el bloque de Aplicaciones Enterprise se verá con mucho más detalle todo el tema de la transaccionalidad declarativa, y los diferentes tipos de gestión de la transacción típicos en JavaEE. Por el momento nos limitaremos a casos relativamente simples.

- Cualquier excepción no comprobada dispara el *rollback* y cualquiera comprobada, no.
- La transacción es de lectura/escritura (en algunos APIs de acceso a datos, por ejemplo, Hibernate, las transacciones de "solo lectura" son mucho más eficientes).
- El *timeout* para efectuar la operación antes de que se haga *rollback* es el que tenga por defecto el API usado (no todos soportan *timeout*).

Todo este comportamiento se puede configurar a través de los atributos de la anotación, como se muestra en la siguiente tabla:

| Propiedad | Tipo | Significado |
|---|---|---|
| propagation | enum: Propagation | nivel de propagación (opcional) |
| isolation | enum: Isolation | nivel de aislamiento (opcional) |
| readOnly | boolean | solo de lectura vs. de lectura/escritura |
| timeOut | int (segundos) | |
| rollbackFor | array de objetos Throwable | clases de excepción que deben causar rollback |
| rollbackForClassName | array con nombres de objetos Throwable | nombres de clases de excepción que deben causar rollback |
| noRollbackFor | array de objetos Throwable | clases de excepción que no deben causar rollback |
| noRollbackForClassName | array con nombres de objetos Throwable | nombres de clases de excepción que no deben causar rollback |



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

Por ejemplo supongamos que al producirse un error en la llamada a `altaPublicidad()` lo que se genera es una excepción propia de tipo `AltaPublicidadException`, que es comprobada pero queremos que cause un *rollback*:

```
@Transactional(rollbackFor=AltaPublicidadException.class)
public void registrar(UsuarioTO uto) {
      udao.registrar(uto);
      udao.altaPublicidad(uto);
}
```

Finalmente, destacar que podemos poner transaccionalidad "global" a una clase y en cada uno de los métodos especificar atributos distintos:

```
//Faltan los imports, etc.
...
@Service
//Vale, el REQUIRED no haría falta ponerlo porque es la opción
//por defecto, pero esto es solo un humilde ejemplo!
@Transactional(propagation=Propagation.REQUIRED)
public class GestorUsuarios {
  @Autowired
  @Qualifier("JPA")
  private IUsuariosDAO udao;

  public void setUdao(IUsuariosDAO udao) {
      this.udao = udao;
  }

  @Transactional(readOnly=true)
  public UsuarioTO login(String login, String password) {
      return udao.login(login, password);
  }

  @Transactional(rollbackFor=AltaPublicidadException.class)
  public void registrar(UsuarioTO uto) {
      udao.registrar(uto);
      udao.altaPublicidad(uto);
  }

  public void eliminarUsuario(UsuarioTO uto) {
       udao.eliminar(uto);
  }
}
```

Nótese que `eliminarUsuario` es también transaccional, heredando las propiedades transaccionales de la clase.

### 3.4.3. Transaccionalidad y uso directo de JDBC

Si usamos el API JDBC directamente, sin ninguna de las facilidades que nos da Spring, se nos va a plantear un problema con la transaccionalidad declarativa: si cada método del DAO abre y cierra una conexión con la BD (lo más habitual), va a ser imposible hacer un rollback de las operaciones que hagan distintos métodos, ya que la conexión ya se habrá cerrado. Para evitar este problema, Spring nos proporciona la clase `DataSourceUtils`,



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

que nos permite "liberar" la conexión desde nuestro punto de vista, pero mantenerla abierta automáticamente gracias a Spring, hasta que se cierre la transacción y no sea necesaria más. Su uso es muy sencillo. Cada vez que queremos obtener una conexión hacemos:

```
//El DataSource se habría resuelto por inyección de dependencias
@Autowired
private DataSource ds;

...
Connection con = DataSourceUtils.getConnection(ds);
```

Y cada vez que queremos liberarla:

```
DataSourceUtils.releaseConnection(con, ds);
```

Nótese que al liberar la conexión no se puede generar una SQLException, al contrario de lo que pasa cuando cerramos con el `close` de JDBC, lo que al menos nos ahorra un `catch`.



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

# 4. Ejercicios de Acceso a datos en Spring

Continuaremos en esta sesión con la aplicación de pedidos, simplificando el código del DAO gracias a Spring y añadiendo transaccionalidad declarativa.

## 4.1. Uso de JDBC en Spring (1 punto)

Tendrás que añadirle al pom.xml una nueva dependencia con las librerías de Spring para JDBC:

```
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-jdbc</artifactId>
   <version>3.2.0.RELEASE</version>
</dependency>
```

Vamos a añadirle al DAO un método `List<Pedido> listarPedidos()` que devuelva los pedidos hechos hasta el momento. Usaremos el API de Spring en el código, en concreto `SimpleJdbcTemplate`. Para ello sigue estos pasos:

1. Modifica el interfaz `IPedidosDAO` para incluir la nueva operación.
2. Tienes que definir un nuevo campo en `PedidosDAOJDBC` con el template:

   ```
   private .JdbcTemplate template;
   ```

   Puedes inicializar este campo del mismo modo que en los apuntes, usando el "truco" del método `setDataSource`. Por tanto, debes crear el propio método setDataSource y quitar la anotación @Autowired del campo "ds" y ponerla en el método. Este método debe inicializar la variable ds y el template. Necesitas las dos porque aunque el método que vas a introducir usa el template, el código de insertarPedido usa directamente el "ds".
3. Tendrás que crear la clase Pedido en el paquete `es.ua.jtech.spring`. Será simplemente un javabean con *getters* y *setters*. Ponle los campos correspondientes a las columnas de la BD (id, idProducto, idCliente, unidades).
4. Necesitarás una clase que implemente el interfaz `RowMapper<Pedido>`, que a partir de un registro de la BD devuelva un objeto Pedido. Puede ser la propia `PedidosJDBCDAO` o una clase adicional.
5. Por último tendrás que escribir el código de "listarPedidos". Observa que es mucho más simple de lo que sería si tuvieras que usar JDBC "puro".

Puedes comprobar el funcionamiento con el servlet "ListarPedidos" , disponible en las plantillas de la sesión. Cópialo en el paquete `es.ua.jtech.spring.web` y fíjate que está mapeado con la URL "listarPedidos". El código no es muy "correcto", ya que accede directamente al DAO sin pasar por la capa de negocio, pero así no tendrás que modificar `IPedidosBO` y `PedidosBOSimple`.



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

## 4.2. Transaccionalidad declarativa (1 punto)

Supongamos que cada vez que se hace un pedido se tiene que avisar al proveedor, para que nos sirva los productos. Si el aviso no puede enviarse, el pedido debería anularse y por tanto se debería hacer un *rollback* de las operaciones realizadas en la base de datos.

En las plantillas de la sesión hay un interfaz `IMensajeria` y una clase `MensajeriaDummy`, que lo implementa. `MensajeriaDummy` supuestamente se encarga de enviar el aviso (y decimos "supuestamente" porque en realidad lo único que hace es imprimir un mensaje en la salida estándar). La clase permite simular el mal funcionamiento del aviso a través de una variable static. Sigue estos pasos:

1. Copia el interface `IMensajeria` y la clase `MensajeriaDummy` en el *package* `es.ua.jtech.spring.negocio`
2. Copia el JSP "mensajeria.jsp" en "src/main/webapp". Este JSP te permite ver el "estado del servicio de avisos" y simular el fallo en el aviso poniendo el estado en "off"
3. `PedidosBOSimple` necesitará de un bean de tipo `IMensajeria` para funcionar. Define un campo de ese tipo igual que hiciste con el DAO y anótalo para que Spring gestione el objeto:

```
    @Autowired
    private IMensajeria m;
```

4. Modifica el código de "insertarPedido" de `GestorPedidosSimple` para que
   - Después de llamar al DAO llame al método "enviarAvisoPedido" del objeto de tipo `IMensajeria`
   - Sea transaccional, de modo que si se produce un `PedidosException` se haga un *rollback*.
   - Para que todo funcione, tendrás además que configurarlo en el fichero `src/main/webapp/WEB-INF/beans.xml`. Consulta apuntes y/o transparencias para ver un ejemplo:
     1. Definir un "transaction manager" para JDBC. Cuidado, su propiedad "dataSource" debe referenciar ("ref") al DataSource que definiste en la sesión anterior.
     2. Usar la etiqueta `tx:annotation-driven` y referenciar el "transaction-manager" que acabas de definir. Para esta etiqueta necesitas el espacio de nombres "tx" en el XML, así que puedes cambiar la cabecera del beans.xml por esta, que ya lo tiene incluido:

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:jee="http://www.springframework.org/schema/jee"
  xmlns:p="http://www.springframework.org/schema/p"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
   http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.2.xsd
    http://www.springframework.org/schema/jee
     http://www.springframework.org/schema/jee/spring-jee-3.2.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-3.2.xsd">
```

5. Cambia el código de "insertarPedido" en el DAO para que la conexión con la BD (variable "con") se obtenga y se libere a través de la clase de Spring `DataSourceUtils`, ya que si Spring no controla la conexión no será posible hacer el *rollback*.
6. Comprueba que todo funciona, accediendo a "mensajeria.jsp" para poner el estado del "servidor de avisos" a OFF e insertando un pedido como hacías hasta ahora. Si todo es correcto, se generará un PedidosException y el pedido no debería aparecer en la BD al no hacerse un "commit" de la operación.

## 4.3. Uso de JPA en Spring (1 punto)

Vamos a crear una clase de negocio que se ocupe de los productos usando el API JPA. Para simplificar, nos permitirá únicamente mostrar productos sabiendo su código. Sigue estos pasos:

1. Introduce en la sección de dependencias del pom.xml las dependencias de Spring ORM, Hibernate JPA y slf4j:

```
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-orm</artifactId>
   <version>3.2.0.RELEASE</version>
</dependency>
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-entitymanager</artifactId>
   <version>3.5.6-Final</version>
</dependency>
<dependency>
   <groupId>org.slf4j</groupId>
   <artifactId>slf4j-log4j12</artifactId>
   <version>1.5.8</version>
   <scope>runtime</scope>
</dependency>
```

2. Crea la clase `ProductoEntity` en el package `es.ua.jtech.spring` con el siguiente código (cuidado, faltan getters y setters)

```
package es.ua.jtech.spring;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
@Table(name="productos")
public class ProductoEntity {
    @Id
    private int id;
    private String nombre;

    //¡¡Faltan getters y setters, genéralos!!
}
```

3. Como fichero `persistence.xml` puedes usar el siguiente. Créalo simplemente como un XML, dentro de "src/main/resources". Crea allí una carpeta "META-INF" y pon el "persistence.xml" ahí dentro. ¡No lo confundas con el "META-INF" de "webapp"!.

```
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">

    <persistence-unit name="pedidosSpringPU"
transaction-type="RESOURCE_LOCAL">
        <class>es.ua.jtech.spring.ProductoEntity</class>
    </persistence-unit>

</persistence>
```

4. Crea el interface `IProductosDAO` en el package `es.ua.jtech.spring.datos`

```
package es.ua.jtech.spring.datos;

import es.ua.jtech.spring.ProductoEntity;

public interface IProductosDAO {
    public ProductoEntity buscarProducto(int id);
}
```

5. Crea una clase `ProductosDAOJPA` que implemente el interfaz anterior, en el mismo package. Inyecta en ella el EntityManager con la anotación @PersistenceContext, e implementa el método "buscarProducto" usando el API JPA. Finalmente, anota la clase con `@Repository`, para que sea un bean de Spring.
6. Introduce la configuración necesaria en el fichero "beans.xml" como se muestra en los apuntes. Básicamente tienes que definir un bean de la clase `org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean`. Consulta apuntes o transparencias para ver las propiedades que debe tener. Te valdrán las que se muestran allí, salvo por la propiedad "dataSource", cuyo "ref" debe referenciar el identificador del DataSource que definiste en la sesión 1 del módulo.
7. Para probar la implementación usa el servlet `VerProducto`, disponible en las plantillas de la sesión. Tendrás que llamarlo pasándole el `id` del producto que quieres ver como parámetro. Por ejemplo, verProducto?id=1 para ver los datos del producto 1.



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

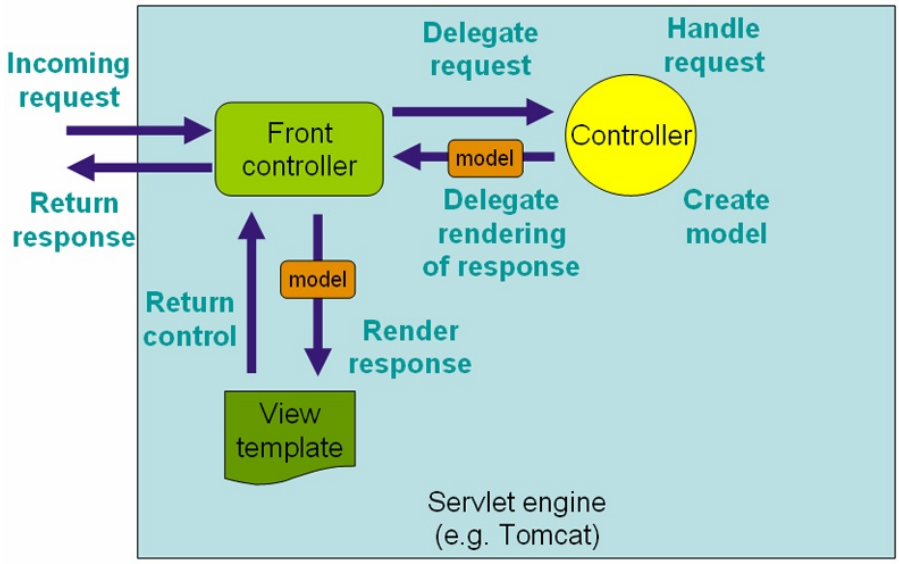
# 5. Introducción a MVC en Spring

En este tema se hará una introducción a las características del *framework* modelo-vista-controlador que incorpora Spring. Veremos que tiene una completa y bien pensada arquitectura, altamente configurable, que a primera vista lo hace parecer bastante complejo, siendo aún así fácil de usar en los casos más frecuentes.

En esta primera sesión dedicada a Spring MVC veremos aplicaciones web del tipo que podríamos llamar "clásico" o "pre-web 2.0". Es decir, aplicaciones en las que cuando el usuario rellena un formulario y se envían los datos al servidor se produce un salto a otra página o una recarga de la misma. Actualmente es mucho más común el uso de aplicaciones AJAX en las que la comunicación con el servidor es más transparente para el usuario y no implica normalmente cambios de página. Veremos cómo trabajar con AJAX en Spring en sesiones posteriores.

## 5.1. Procesamiento de una petición en Spring MVC

A continuación se describe el flujo de procesamiento típico para una petición HTTP en Spring MVC. Esta explicación está simplificada y no tiene en cuenta ciertos elementos que comentaremos posteriormente. Spring es una implementación del patrón de diseño "front controller", que también implementan otros frameworks MVC, como por ejemplo, el clásico Struts.




Procesamiento de una petición HTTP en Spring (tomado de la documentación del framework)

- Todas las peticiones HTTP se canalizan a través del *front controller*. En casi todos los



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

frameworks MVC que siguen este patrón, el *front controller* no es más que un servlet cuya implementación es propia del framework. En el caso de Spring, la clase `DispatcherServlet`.

- El *front controller* averigua, normalmente a partir de la URL, a qué `Controller` hay que llamar para servir la petición. Para esto se usa un `HandlerMapping`.
- Se llama al `Controller`, que ejecuta la lógica de negocio, obtiene los resultados y los devuelve al servlet, encapsulados en un objeto del tipo `Model`. Además se devolverá el nombre lógico de la vista a mostrar (normalmente devolviendo un String, como en JSF).
- Un `ViewResolver` se encarga de averiguar el nombre físico de la vista que se corresponde con el nombre lógico del paso anterior.
- Finalmente, el *front controller* (el `DispatcherServlet`) redirige la petición hacia la vista, que muestra los resultados de la operación realizada.

En realidad, el procesamiento es más complejo. Nos hemos saltado algunos pasos en aras de una mayor claridad. Por ejemplo, en Spring se pueden usar interceptores, que son como los filtros del API de servlets, pero adaptados a Spring MVC. Estos interceptores pueden pre y postprocesar la petición alrededor de la ejecución del `Controller`. No obstante, todas estas cuestiones deben quedar por fuerza fuera de una breve introduccion a Spring MVC como la de estas páginas.

## 5.2. Configuración básica

Lo habitual es que se configure la aplicación web de manera que todas las peticiones cuya URL sigue un determinado patrón pasen a través de Spring MVC. Así, por ejemplo, podemos "redirigir" a través de Spring MVC todas las peticiones que acaben en ".do". Esto nos permite por ejemplo servir los recursos estáticos aparte, ya que no es necesario que estos pasen por el flujo de ejecución MVC.

Necesitaremos configurar el `web.xml` para que todas las peticiones HTTP con un determinado patrón se canalicen a través del mismo servlet, en este caso de la clase `DispatcherServlet` de Spring. Como mínimo necesitaremos incluir algo como esto:

```
<servlet>
   <servlet-name>dispatcher</servlet-name>
   <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
   </servlet-class>
   <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
   <servlet-name>dispatcher</servlet-name>
   <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

Con esta configuración, todas las peticiones acabadas en `.do`, como `getPedido.do` o `verClientes.do`, se redirigirían al servlet de Spring.



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

Para aumentar la modularidad de la aplicación, se pueden configurar la capa web en un XML distinto al resto de la aplicación. Al arrancar, Spring buscará automáticamente un fichero con el mismo nombre del servlet que hace de dispatcher, seguido de la terminación `-servlet.xml`. La búsqueda se hace en el directorio `WEB-INF`. En nuestro caso, el fichero buscado automáticamente sería `dispatcher-servlet.xml`.

Por tanto, la forma habitual de trabajar es usar un XML para los beans de la capa web y otro (u otros) distinto para los de la capa de negocio y DAOs. Spring establece una jerarquía de contextos de modo que en el XML de la capa web se heredan automáticamente los otros beans, lo que nos permite referenciar los objetos de negocio en nuestro código MVC.

**¡Cuidado con las anotaciones!**

Recordad que hay que configurar la etiqueta `component-scan` para que Spring examine ciertos packages en busca de anotaciones. Si usamos dos XML, uno para la capa web y otro para las demás capas, cada XML **debe** tener su propio `component-scan`. Aunque por nuestra organización de código el `component-scan` de las capas de negocio y datos "cubra" también a las clases de web, necesitamos ponerlo explícitamente en el `dispatcher-servlet.xml`

Suponiendo que nuestros componentes web están implementados en el paquete `es.ua.jtech.spring.mvc`, en el `dispatcher-servlet.xml` aparecería:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context.xsd
       http://www.springframework.org/schema/mvc
       http://www.springframework.org/schema/mvc/spring-mvc.xsd">

  ...
       <context:component-scan base-package="es.ua.jtech.spring.mvc"/>
       <mvc:annotation-driven/>
  ...
</beans>
```

La etiqueta `<mvc:annotation-driven/>` permite usar anotaciones y hace una configuración por defecto de distintos elementos que iremos viendo, como la validación y conversión de datos. Está disponible a partir de la versión 3.0 de Spring

## 5.3. Caso 1: petición sin procesamiento de datos de entrada

La elaborada arquitectura de Spring MVC, y las muchas posibilidades que tiene el usuario de configurar a su medida el procesamiento que hace el *framework* hacen que sea poco intuitivo hacer una descripción general de Spring MVC, al menos si no se dispone del



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

suficiente tiempo para hacerlo de manera pausada, lo que no es el caso. En su lugar, hemos preferido aquí describir cómo se implementarían un par de casos típicos en una aplicación web, indicando cómo implementar cada caso y las posibilidades adicionales que ofrece Spring MVC. El lector tendrá que consultar fuentes adicionales para ver con detalle el resto de opciones.

El primer caso sería el de una petición que no necesita interacción por parte del usuario en el sentido de proceso de datos de entrada: por ejemplo sacar un listado de clientes, mostrar los datos de un pedido, etc. La "no interacción" aquí se entiende como que no hay que procesar y validar datos de entrada. Es decir, que no hay un formulario HTML. Esto no quiere decir que no haya parámetros HTTP, pero entonces suelen estar fijos en la URL de un enlace o de modo similar, no introducidos directamente por el usuario. Estas peticiones suelen ser simplemente listados de información de "solo lectura".

Vamos a poner estos ejemplos en el contexto de una hipotética aplicación web para un hotel, en la cual se pueden ver y buscar ofertas de habitaciones, disponibles con un determinado precio hasta una fecha límite. Aquí tendríamos lo que define a una oferta:

```
 package es.ua.jtech.spring.dominio;

import java.math.BigDecimal;
import java.util.Date;

public class Oferta {
        private BigDecimal precio;
        private Date fechaLimite;
        private TipoHabitacion tipoHab;
        private int minNoches;

        //..aquí vendrían los getters y setters
}
```

TipoHabitación es un tipo enumerado que puede ser `individual` o `doble`.

Lo primero es definir el controlador, que será una clase java convencional. Con anotaciones le indicamos a Spring qué métodos procesarán las peticiones y cómo enlazar sus parámetros con los parámetros HTTP.

### 5.3.1. Definir el controller y asociarlo con una URL

Supongamos que queremos sacar un listado de ofertas del mes. Así, el esqueleto básico de nuestro `Controller` sería:

```
import es.ua.jtech.spring.negocio.IGestorOfertas;

@Controller
@RequestMapping("/listaOfertas.do")
public class ListaOfertasController  {
   @Autowired
   private IGestorOfertas miGestor;
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
   ...
}
```

La anotación `@Controller` indica que la clase es un bean controlador y nos ahorra el trabajo de definir en XML el bean correspondiente. Con `@RequestMapping` asociamos una URL a este controlador.

Por supuesto, cualquier `Controller` necesitará para hacer su trabajo de la colaboración de uno o más objetos de negocio. Lo lógico es que estos objetos sean beans de Spring y que instanciemos las dependencias haciendo uso del contenedor. En nuestro caso dicho objeto es "miGestor". Supondremos que él es el que "sabe" sacar las ofertas del mes con el método `public List<Oferta> getOfertasDelMes(int mes)`. Probablemente a su vez este gestor deba ayudarse de un DAO para hacer su trabajo, pero esto no nos interesa aquí.

Por tanto, hemos usado `@Autowired` para resolver dependencias por tipo, suponiendo para simplificar que no hay ambigüedad (solo existe una clase que implemente `IGestorOfertas`)

Para que Spring sepa qué método del controlador debe procesar la petición HTTP se puede usar de nuevo la anotación `@RequestMapping`. Podemos especificar qué método HTTP (GET, POST,...) asociaremos al método java.

```
@Controller
@RequestMapping("/listaOfertas.do")
public class ListaOfertasController {
   @Autowired
   private GestorOfertas miGestor;

   @RequestMapping(method=RequestMethod.GET)
   public String procesar(HttpServletRequest req) {
      int mes = Integer.parseInt(req.getParameter("mes"));
      ...
   }
}
```

Aquí estamos asociando una llamada a "listaOfertas.do" de tipo GET con el método java `procesar`. Una llamada a la misma URL con POST produciría un error HTTP de tipo 404 ya que no habría nada asociado a dicha petición. Si no necesitamos tanto control sobre el método HTTP asociado podemos poner la anotación `@RequestMapping("/listaOfertas.do")` directamente en el método `procesar` en lugar de en la clase.

### 5.3.2. Leer parámetros HTTP

La primera tarea que debe hacer un método de procesamiento de una petición es **leer el valor de los parámetros HTTP**. Spring automáticamente detectará que `procesar` tiene un argumento de tipo `HttpServletRequest` y lo asociará a la petición actual. Esta "magia" funciona también con otros tipos de datos: por ejemplo, un parámetro de tipo `Writer` se asociará automáticamente a la respuesta HTTP y uno de tipo `Locale` al *locale*



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

de la petición. Consultar la documentación de Spring para ver más detalladamente las asociaciones que se realizan automáticamente.

Otra posibilidad es asociar explícitamente parámetros HTTP con parámetros java mediante la anotación `@RequestParam`. Usando esta anotación, el ejemplo quedaría:

```
@Controller
@RequestMapping("/listaOfertas.do")
public class ListaOfertasController {
   @Autowired
   private IGestorOfertas miGestor;

   @RequestMapping(method=RequestMethod.GET)
   public String procesar(@RequestParam("mes") int mes) {
      ...
   }
}
```

De este modo Spring hace por nosotros el trabajo de extraer el parámetro y convertir el tipo de String a int. Hay que tener en cuenta que esta forma de trabajar puede generar múltiples excepciones. Si el parámetro no está presente en la petición HTTP se producirá una excepción de tipo `MissingServletRequestParameterException`. Si está presente y no se puede convertir se generará una `TypeMismatchException`. Para especificar que el parámetro no es obligatorio, podemos hacer:

```
public String procesar(@RequestParam(value="mes",required=false) int mes)
{
    ...
}
```

En este caso, se recomienda usar un Integer en lugar de un int para que si no existe el parámetro Spring nos pueda pasar el valor null (si no existe Spring no lo interpreta como 0).

Aunque en el ejemplo anterior hemos tenido que vincular el parámetro HTTP con el del método del controller de manera explícita, en Spring esta vinculación se hace muchas veces de modo automático. Por ejemplo si un método del controller tuviera un parámetro de tipo HttpRequest Spring lo asociaría automáticamente con el HttpRequest actual. Así, podríamos hacer algo como:

```
public String procesar(HttpRequest peticion) {
    int mes = Integer.parseInt(peticion.getParameter("mes");
    ...
}
```

Esta vinculación automática funciona con objetos de varios tipos: HttpRequest, HttpResponse, HttpSession,... Se recomienda consultar la documentación de Spring para más detalle (sección 15.3.2.3)

Una vez leídos los parámetros, lo siguiente suele ser **disparar la lógica de negocio y colocar los resultados en algún ámbito al que pueda acceder la vista**. Spring ofrece



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

algunas clases auxiliares para almacenar de manera conveniente los datos obtenidos. En realidad son simplemente `Map` gracias a los que se puede almacenar un conjunto de objetos dispares, cada uno con su nombre. Spring hace accesible el modelo a la vista como un atributo de la petición y al controlador si le pasamos un parámetro de tipo `ModelMap`. Aquí tenemos la lógica de `procesar`:

```
@RequestMapping(method=RequestMethod.GET)
public String procesar(@RequestParam("mes") int mes,
                       ModelMap modelo) {
   modelo.addAttribute("ofertas", miGestor.getOfertasDelMes(mes));
   return "listaOfertas";
}
```

### 5.3.3. Mostrar la vista

Finalmente, nos queda **especificar qué vista hemos de mostrar**. Hemos estado viendo en todos los ejemplos que el método `procesar` devuelve un String. Dicho String se interpreta como el nombre "lógico" de la vista a mostrar tras ejecutar el controlador. En el JSP será sencillo mostrar los datos ya que, como hemos dicho, se guardan en un atributo de la petición:

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core"prefix="c"%>
...
<c:forEach items="${ofertas}" var="o">
        Habitación ${o.tipoHab} un mínimo de
        ${o.minNoches} noches por solo ${o.precio}eur./noche
</c:forEach>
...
```

Queda pendiente cómo se resuelve el nombre lógico de la vista, es decir, cómo sabe Spring a qué archivo corresponde en realidad el nombre lógico. Para ello necesitamos un `ViewResolver`. Debemos definir un bean en el XML con el `id=viewResolver` y la implementación de ViewResolver que nos interese. De las que proporciona Spring una de las más sencillas de usar es `InternalResourceViewResolver`. Esta clase usa dos parámetros básicos: `prefix` y `suffix`, que puestos respectivamente delante y detrás del nombre lógico de la vista nos dan el nombre físico. Ambos son opcionales. Por ejemplo:

```
<bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
         <property name="prefix" value="/jsp/"/>
         <property name="suffix" value=".jsp"/>
</bean>
```

Así, si el nombre lógico de la vista de nuestro ejemplo era `ofertas`, se acabaría buscando el recurso físico `/jsp/ofertas.jsp`.

En algunos casos, en lugar de mostrar una vista, lo adecuado es ejecutar una operación



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

MVC. Por ejemplo, tras dar de alta una oferta podría ser adecuado mostrar el listado de ofertas, y probablemente ya tengamos un controller que se ocupe de esto. Para ello podemos usar el prefijo "redirect:". Así, tras insertar una nueva oferta en el controller podríamos hacer algo como

```
return "redirect:/listarOfertas.do"
```

Para saltar al listado de ofertas, suponiendo que esa URL esté asociada con la operación correspondiente.

## 5.4. Caso 2: procesamiento de un formulario

Este caso es más complejo ya que implica varios pasos:

- El usuario introduce los datos, normalmente a través de un formulario HTML
- Los datos se validan, y en caso de no ser correctos se vuelve a mostrar el formulario para que el usuario pueda corregirlos.
- En caso de pasar la validación, los datos se "empaquetan" en un objeto Java para que el `controller` pueda acceder a ellos de modo más sencillo que a través de la petición HTTP.
- El `controller` se ejecuta, toma los datos, realiza la tarea y cede el control para que se muestre la vista.

En Spring la práctica habitual es mostrar el formulario y procesar los datos dentro del mismo controlador, aunque podríamos hacerlo en controladores distintos.

### 5.4.1. Javabean para almacenar los datos de entrada

En Spring se suele usar un objeto para almacenar los datos que el usuario teclea en el formulario. Habitualmente habrá una correspondencia uno a uno entre las propiedades del objeto (accesibles con getters/setters) y los campos del formulario. Por ejemplo, este podría ser un objeto apropiado para buscar ofertas. Solo contiene los campos estrictamente necesarios para la búsqueda, no todos los datos que puede contener una oferta:

```
package es.ua.jtech.spring.mvc;
import es.ua.jtech.spring.dominio.TipoHabitacion;

public class BusquedaOfertas {
        private int precioMax;
        private TipoHabitacion tipoHab;

        //..ahora vendrían los getters y setters
}
```

### 5.4.2. El controller



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

Desde el punto de vista de lo que tenemos que implementar, este caso solo se diferenciará del caso 1 (sin procesamiento de datos de entrada) en el `controller` y en que para la vista podemos usar *tags* de Spring, para que se conserve el valor de los campos y el usuario no tenga que volver a escribirlo todo si hay un error de validación. La asociación entre la URL y el controlador y entre la vista lógica y el recurso físico serán igual que antes. Además, por supuesto, tendremos que implementar la validación de datos. Por el momento vamos a hacer la validación de manera manual y dejaremos para la sesión siguiente las facilidades de validación que nos ofrece Spring, para no distraernos.

Mediante la anotación `@RequestMapping` mapearemos la petición que muestra el formulario a un método java (será con GET) y la que lo procesa a otro distinto (POST):

```
...
@Controller
@RequestMapping("/busquedaOfertas.do")
public class BusquedaOfertasController {
    @Autowired
    private GestorOfertas miGestor;

    @RequestMapping(method=RequestMethod.GET)
    public String preparaForm(Model modelo) {
       ...
    }

    @RequestMapping(method=RequestMethod.POST)
    public String procesaForm(BusquedaOfertas bo) {
       ...
    }
}
```

Como ya hemos dicho, los datos del formulario se van a almacenar en un objeto de tipo `BusquedaOferta`. El método `preparaForm`, que se ejecutará *antes* de mostrar el formulario, debe crear un `BusquedaOferta` con los valores por defecto y colocarlo en el modelo para que puedan salir en el formulario. El modelo no es más que un "almacén" de objetos al que tiene acceso tanto la vista como los controllers. Está representado en Spring por el interfaz `Model`, y se le añaden objetos con el método `addAttribute` (más o menos como el `request` estándar del API de servlets). Finalmente, se devolverá el nombre lógico de la vista que contiene el formulario, que en nuestro caso representa a la página "busquedaOfertas.jsp". Aquí tenemos el código:

```
@RequestMapping(method=RequestMethod.GET)
public String preparaForm(Model modelo) {
    modelo.addAttribute("bo", new BusquedaOfertas());
    return "busquedaOfertas";
}
```

### 5.4.3. La vista con las taglibs de Spring

En la página `busquedaOfertas.jsp` colocaremos un formulario usando las *taglibs* de Spring. Estas etiquetas nos permiten vincular el modelo al formulario de forma sencilla y además mostrar los errores de validación como veremos posteriormente.



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<html>
    <head>
        <title>Esto es busquedaOfertas.jsp</title>
    </head>
    <body>
        <form:form modelAttribute="bo">
            <form:input path="precioMax"/> <br/>
            <form:select path="tipoHab">
                <form:option value="individual"/>
                <form:option value="doble"/>
            </form:select>
            <input type="submit" value="buscar"/>
        </form:form>
    </body>
</html>
```

Obsérvense varios puntos interesantes:

- La vinculación entre el objeto `BusquedaOfertas` del modelo y los datos del formulario se hace dándole un nombre al objeto al añadirlo al modelo y usando este mismo nombre en el atributo `modelAttribute` de la etiqueta "form".
- Las etiquetas de Spring para formularios son muy similares a su contrapartida HTML. El atributo `path` de cada una indica la propiedad de `BusquedaOfertas` a la que están asociadas.
- El botón de enviar del formulario no necesita usar ninguna etiqueta propia de Spring, es HTML convencional
- El formulario no tiene "action" ya que llamará a la misma página. Implícitamente Spring lo convierte en un formulario con action vacío y método POST.

Llegados a este punto, el usuario rellena el formulario, lo envía y nosotros debemos procesarlo. Esto lo hace el controlador en el método `procesaForm`. Veamos su código, por el momento sin validación:

```
@RequestMapping(method=RequestMethod.POST)
public String procesaForm(BusquedaOfertas bo, Model modelo) {
   //buscamos las ofertas deseadas
   modelo.addAttribute("ofertas", miGestor.BuscaOfertas(bo));
   //y saltamos a la vista que muestra los resultados
   return "listaOfertas";
 }
```

*Antes* de ejecutar este método Spring habrá tomado los datos del formulario y creado un `BusquedaOferta` con ellos. Spring vincula automáticamente este bean con el parámetro del mismo tipo del método `procesaForm`. Podemos hacer explícita esta vinculación con la anotación `@ModelAttribute` en el parámetro. Esto podría ser necesario si tenemos más de un parámetro del mismo tipo

### 5.4.4. Validación de los datos

Por supuesto, el ejemplo anterior no es demasiado correcto, ya que antes de disparar la lógica de negocio tendríamos que **validar los datos**. El código con validación sería algo



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

como:

```
@RequestMapping(method=RequestMethod.POST)
public String procesaForm(@ModelAttribute("bo") BusquedaOfertas bo,
                          BindingResult result,
                          Model modelo) {

   //El precio no puede ser negativo
   if (bo.getPrecioMax()<0)
       result.rejectValue("precioMax", "precNoVal");
   //si Spring o nosotros hemos detectado error, volvemos al formulario
   if (result.hasErrors()) {
       return "busquedaOfertas";
   }
   //si no, realizamos la operación
   modelo.addAttribute("ofertas", miGestor.BuscaOfertas(bo));
   //y saltamos a la vista que muestra los resultados
   return "listaOfertas";
 }
```

Cuando se llama al `procesaForm` Spring ya habrá hecho una pre-validación comprobando que los datos son del tipo adecuado para encajar en un `BusquedaOfertas`: por ejemplo, que el precio es convertible a `int`. El resultado de esta "pre-validación" es accesible a través del segundo parámetro. De nuevo, tenemos una vinculación automática entre un parámetro e información del contexto: el parámetro de tipo `BindingResult` y el resultado de la validación. Condición indispensable es que este parámetro venga justo después del bean que estamos validando. La clase `BindingResult` tiene métodos para averiguar qué errores se han producido, y como ahora veremos, añadir nuevos.

**La anotación @ModelAttribute**

Si usamos BindingResult tendremos que usar obligatoriamente la anotación @ModelAttribute anotando el objeto que se está "validando".

Nuestra lógica de negocio puede tener requerimientos adicionales a la mera conversión de tipos. Por ejemplo, en nuestro caso está claro que un precio no puede ser un valor negativo. Por tanto, lo comprobamos y si es negativo usamos el método `rejectValue` para informar de que hay un nuevo error. Este método tiene dos parámetros: el nombre de la propiedad asociada al error y la clave del mensaje de error. El mensaje estará guardado en un fichero properties bajo esa clave. Si hay errores retornamos a la vista del formulario. Si no, disparamos la lógica de negocio, obtenemos los resultados, los añadimos al modelo y saltamos a la vista que los muestra.

### 5.4.5. Los mensajes de error

Para que en el JSP aparezcan los mensajes, usaremos la etiqueta `<form:errors/>` de Spring. Por ejemplo, supongamos que al lado del precio queremos mostrar sus posibles errores

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
<html>
    <head>
        <title>Esto es busquedaOfertas.jsp</title>
    </head>
    <body>
       <form:form modelAttribute="bo">
            <form:input path="precioMax"/>
            <form:errors path="precioMax" cssClass="rojo"/>
            ...
```

Con el atributo `path` decimos el campo para el que queremos mostrar los errores. El atributo `cssClass` hace que el texto del error venga "envuelto" en un `<span class="rojo">`, útil para hacer que el error aparezca con un estilo especial.

El texto de los mensajes de error no se define en código java, para que sea más sencillo modificarlos y no sea necesario recompilar el código. Se definen en ficheros `.properties`. En el `dispatcher-servlet.xml` debemos definir un bean de Spring que representa al archivo de mensajes. Se suele usar la clase `ResourceBundleMessageSource`. El `id` del bean debe ser `messageSource`

```
<bean id="messageSource"
class="org.springframework.context.support.ResourceBundleMessageSource">
     <property name="basename"
value="es/ua/jtech/spring/mvc/mensajesWeb"/>
</bean>
```

Y aquí tenemos el archivo `mensajesWeb.properties` (que, como indicaba la propiedad "basename", está colocado dentro del paquete `es.ua.jtech.spring.mvc`):

```
precNoVal = precio no válido
typeMismatch.precioMax = el precio no es un número
```

Las claves a las que se asocian los mensajes en este fichero pueden ser propias (como el caso de `precNoVal`, que hemos definido en la llamada a `rejectValue` del apartado anterior) o bien seguir una convención propia de Spring. Por ejemplo, `typeMismatch` se asocia por defecto a los errores de conversión de tipos, de modo que si introdujéramos un valor no convertible a entero, se mostraría este error. Iniicialmente Spring probaría a encontrar una clave de typeMismatch específica para el campo precioMax, que es la que tenemos. Si no se encontrara esta, se buscaría solamente typeMismatch y si ni siquiera estuviera esta se mostraría un mensaje de error por defecto (por supuesto en inglés). Se recomienda consultar la documentación de la clase DefaultMessageCodesResolver , que es la encargada de buscar las claves apropiadas por orden.

### 5.4.6. Validación fuera del controller

**¡No hagáis esto en casa!**

A partir de Spring 3, la validación se puede hacer de manera mucho más sencilla con



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

anotaciones, como veremos en la última sesión dedicada a MVC. No obstante vamos a explicar aquí la validación manual con el objetivo de que el tema sea autocontenido. Pero no es recomendable usarla salvo para aplicaciones *legacy*.

Evidentemente es mucho mejor separar el código de validación del bean del código del controller. Para ello debemos proporcionar una clase propia que implemente el interfaz `Validator`. En cierto método de dicha clase meteremos el código de validación, con lo que el controller quedará mucho más limpio.

Para validar los datos necesitamos una clase que implemente el interfaz `org.springframework.validation.Validator` . Supongamos que queremos rechazar la oferta buscada si el precio está vacío o bien no es un número positivo (para simplificar vamos a obviar la validación del tipo de habitación). El código sería:

```
package es.ua.jtech.spring.mvc;

import org.springframework.validation.Errors;
import org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;


public class OfertaValidator implements Validator {

	public boolean supports(Class arg0) {
		return arg0.isAssignableFrom(BusquedaOfertas.class);
	}

	public void validate(Object obj, Errors errors) {
		ValidationUtils.rejectIfEmpty(errors, "precioMax",
"precioVacio");
		BusquedaOfertas bo = (BusquedaOfertas) obj;
		//comprobar que el precio no esté vacío
		// (para que no haya null pointer más abajo)
		if (bo.getPrecioMax()==null)
			return;
		//comprobar que el número sea positivo
		if (bo.getPrecioMax().floatValue()<0)
			errors.rejectValue("precioMax", "precNoVal");
	}

}
```

Como vemos, un `Validator` debe implementar al menos dos métodos:

- supports: indica de qué clase debe ser el `Command` creado para que se considere aceptable. En nuestro caso debe ser de la clase `BusquedaOfertas`
- validate: es donde se efectúa la validación. El primer parámetro es el `Command`, que se pasa como un `Object` genérico (lógico, ya que Spring no nos obliga a implementar ningún interfaz ni heredar de ninguna clase determinada). El segundo es una especie de lista de errores. Como vemos, hay métodos para rechazar un campo si es vacío o bien por código podemos generar errores a medida (en este caso, si el precio es un número negativo).



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

# 6. Ejercicios de MVC en Spring

La aplicación "AmigosSpring" es una especie de red social muy simple en la que los usuarios pueden hacer login en el sistema y ver el perfil propio y el de otros. El núcleo de la aplicación ya está desarrollado, y el objetivo de la sesión es ponerle un interfaz web usando Spring MVC.

## 6.1. Configurar el proyecto para Spring MVC (0.5 puntos)

El proyecto está en las plantillas de la sesión. Es un proyecto Maven, así que lo puedes importar como "Existing Maven Projects".

Antes de probar la aplicación, **crea la base de datos del script "amigosSpring.sql"** incluido en la carpeta "database" del proyecto.

Tras desplegar el proyecto, comprueba que todo funciona con la página "testDAO.jsp". Tendrás que pasarle como parámetro "login" el login de un usuario de la base de datos. Por ejemplo, accede a `http://localhost:8080/AmigosSpring/testDAO.jsp?login=jsf` y comprueba que aparecen los datos de la usuaria "jsf".

Nótese que para simplificar, la aplicación no tiene capa de negocio. La capa de presentación va a acceder directamente a los DAOs, lo que no es en general una buena práctica.

Las dependencias de Spring 3 ya se han incluido en el pom.xml del proyecto, pero tendrás que hacer las siguientes tareas para configurarlo para Spring MVC:

- Échale un vistazo al XML de configuración de beans para la capa de datos, `src/main/webapp/WEB-INF/daos.xml`. Fijate en que está referenciado en el web.xml, como siempre hacemos para arrancar Spring. (Vale, en este punto no es que hayas tenido que hacer mucho, pero al menos recuerda que estos ficheros son necesarios, para cuando hagas tus propios proyectos).
- Modifica el web.xml para crear el servlet "dispatcher". Asócialo con las peticiones de tipo *.do, como hemos hecho en los apuntes.
- Crea el fichero XML de configuración de beans para la capa web, WEB-INF/dispatcher-servlet.xml (New > Spring Bean configuration file). En el segundo paso del asistente necesitamos el "context" y también el "mvc". Puedes tomar como modelo el "dispatcher-servlet.xml" de los apuntes. Pero cuidado, tendrás que cambiar el atributo "package" de la etiqueta component-scan por lo que consideres necesario (fijate en el package en que está el LoginController, que es donde vamos a colocar el resto de Controllers de Spring).
- Crea dentro del dispatcher-servlet.xml el bean "viewResolver", que asociará nombres lógicos de vistas con JSPs. Toma como modelo el de los apuntes, pero ten en cuenta que aquí los JSP no están dentro de ningún directorio especial (directamente están en



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

webapp) por lo que la propiedad "prefix" sobra.

Comprueba que la configuración es correcta entrando en la aplicación (por ejemplo como "jsf" con password "jsf"). . Una vez hecho login, debería aparecer la página del usuario actual, ya que el controlador que maneja el caso de uso "login" ya está implementado en la clase `es.ua.jtech.amigosspring.presentacion.LoginController`

## 6.2. MVC sin procesamiento de datos de entrada (1 punto)

Implementar la capa web para el caso de uso "ver amigo". Escribiendo un identificador de usuario y pulsando el botón "ver amigo" de la parte izquierda de la pantalla, veremos sus datos en un JSP ya incluido en la plantilla, "usuario.jsp".

Pasos a seguir para crear el controller:

1. Crear en el paquete `es.ua.jtech.amigosspring.presentacion` la clase `VerAmigoController`. Añádele la anotación `@Controller` para convertirlo en controlador de Spring.
2. La clase necesitará de un objeto de tipo `IUsuariosDAO` para hacer su trabajo, que debes inyectar con `@Autowired`
3. Crea un método java para procesar la petición:
   - Anota el método con `@RequestMapping` para asociarlo a la URL "verAmigo", que es a la que llama el formulario.
   - La petición disparada por el formulario tiene un único parámetro HTTP, de nombre "amigo", que es el del campo de texto donde se escribe el login del usuario a ver. Usa la anotación `@RequestParam` para asociar este parámetro HTTP con un parámetro del método java
   - El método tendrá un parámetro de tipo `Model` es para poder colocar el objeto Usuario. La vista, que es la página usuario.jsp, espera un Usuario bajo el nombre "amigo". (fíjate en que se usan expresiones del tipo ${amigo.edad})
   - Finalmente recuerda que el método java debe devolver un String que es el nombre lógico de la vista (en nuestro caso, el nombre real menos el .jsp)
   - Si el usuario no existe (si el DAO devuelve null), debes saltar a otra vista distinta. Crea una página noexiste.jsp que simplemente diga "el usuario no existe" y devuelve esta vista si el usuario buscado no existe.

Puedes probar a ver por ejemplo los usuarios `javaee` o `pepe`.

## 6.3. MVC con procesamiento de datos de entrada (1 punto)

Implementar la capa web para el caso de uso "buscar usuarios". Podemos buscar usuarios a partir de diversos criterios como edad, localidad, o sexo. El formulario de búsqueda está en la página "busqueda.jsp", ya incluída en las plantillas.

1. Definir una clase `es.ua.jtech.amigosspring.presentacion.CriteriosBusqueda` que pueda capturar los datos que se introducen en el formulario de búsqueda (con



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

propiedades `int edadMin, int edadMax, String localidad, String sexo`)

2. Crear el *controller* de Spring
   1. Crear la clase `es.ua.jtech.amigosspring.presentacion.BuscarAmigosController`. Usando anotaciones, convertirla en controlador y mapearla con la URL "/busqueda"
   2. La clase necesitará de un `IUsuariosDAO` para hacer la búsqueda, igual que los otros Controllers.
   3. El controlador tendrá dos métodos: uno de ellos mapeado con GET y otro con POST:
      - `public String preparaForm()`: por el momento simplemente devolverá el nombre lógico de la vista con el formulario ("busqueda").
      - `public String procesaForm(CriteriosBusqueda criterios, Model modelo)`:
        - Tomando los datos del objeto "criterios", llama al método "buscar" del DAO para hacer la búsqueda.
        - Coloca el resultado en el modelo bajo el nombre "encontrados", ya que la vista JSP espera una List<Usuario> bajo ese nombre
        - Finalmente devuelve el nombre lógico de la vista que mostrará los datos ("encontrados"). Se corresponde con la página "encontrados.jsp" que ya está implementada.

3. Prueba a hacer búsquedas. **No dejes en blanco los campos de edad mínima y máxima**, porque si no dará error. Lo arreglaremos en el siguiente ejercicio.

## 6.4. Taglibs de Spring y validación básica de datos (1 punto)

Un problema de las búsquedas es que hay que escribir manualmente los valores para edad mínima y máxima. Vamos a poner unos valores por defecto sin tocar el HTML. Para ello:

1. Modifica el `preparaForm` del controller añadiéndole un parámetro de tipo `Model`. En este Model debes añadir como atributo un objeto de la clase `CriteriosBusqueda` con una edad mínima de 18 y máxima de 99.
2. Para que estos valores se muestren en el formulario hay que usar las taglibs de Spring. Cambia las etiquetas del formulario por las de Spring. Recuerda que el "modelAttribute" de la etiqueta "form:form" es el que vincula el formulario con el `CriteriosBusqueda` que has añadido al `Model` en `preparaForm`.
3. Comprueba que al acceder a la página de búsqueda aparecen los valores por defecto en el formulario.

Todavía nos queda por resolver la validación de datos. Si pruebas a introducir una edad mínima o máxima no numérica verás que se genera un error, al no ser posible vincular el campo del formulario con el dato de tipo entero. Vamos a filtrar los errores y mostrar un mensaje apropiado:

1. En el método `procesaForm` del controller define un parámetro de tipo



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

`BindingResult` para "capturar" los errores. Tiene que estar **inmediatamente detrás** del parámetro de tipo `CriteriosBusqueda` ya que son los errores asociados a él. Recuerda también que al usar esto debes obligatoriamente anotar el parámetro CriteriosBusqueda con @ModelAttribute

2. En el código del método, antes de llamar al DAO comprueba si el `BindingResult` contiene errores (si "hasErrors()" devuelve *true*). En ese caso, hay que ir de nuevo a la página de búsqueda.
3. Comprueba que al meter un dato no numérico en la edad se vuelve a mostrar la misma página de búsqueda
4. Nos falta todavía mostrar un mensaje de error al usuario indicando qué ha pasado. Hay que:
   - Crear el fichero .properties con los mensajes. Créalo en "src/main/resources".
   - Crear el bean "messageSource" en el dispatcher-servlet.xml, referenciando el .properties creado. Si lo has hecho en "resources" no es necesario poner el package, solo el nombre del .properties, sin la extensión.
   - Usar etiquetas `<form:errors/>` en el formulario para mostrar los errores asociados a la edad mínima y máxima. Lleva cuidado de meter las etiquetas dentro del formulario, si las pones fuera no se encontrará la referencia a los campos "edadMin" y "edadMax".
5. Comprueba que al dejar vacía la edad o introducir un valor no numérico se muestra el error en el JSP. Por el momento no vamos a validar otros errores, como por ejemplo edades negativas, eso lo haremos en sesiones posteriores.



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

# 7. Aplicaciones AJAX y REST con Spring MVC

En la sesión anterior vimos cómo implementar aplicaciones web "clásicas", es decir, aquellas en las que cada "pantalla" de nuestra aplicación se implementa en un HTML distinto y cada comunicación con el servidor implica un cambio de página (y de "pantalla"). No es necesario decir que la gran mayoría de aplicaciones web actuales no son así. Casi todas usan AJAX y Javascript en el cliente de manera intensiva, lo que nos permite comunicarnos con el servidor sin cambiar de página y también cambiar dinámicamente la interfaz sin movernos a otro HTML.

Un paso más allá son las aplicaciones REST, en las que (entre otras cosas) el cliente no recibe el HTML sino directamente los datos en un formato estándar (JSON/XML) y los "pinta" él mismo (o los formatea en HTML para que los "pinte" el navegador) . Esta filosofía permite diseñar capas web multi-dispositivo (escritorio/web/móviles).

Veremos en esta sesión una introducción a las facilidades que nos da Spring para implementar estos dos tipos de aplicaciones.

## 7.1. AJAX con Spring

En una aplicación AJAX lo que necesitamos por parte del servidor es que nos permita intercambiar información hacia/desde el cliente fácilmente. En su variante más sencilla esa información sería simplemente texto o pequeños fragmentos de HTML. En casos más complejos serían objetos Java serializados a través de HTTP en formato JSON o XML. Vamos a ver qué funcionalidades nos ofrece Spring para implementar esto, planteando varios casos de uso típicos en AJAX.

### 7.1.1. Caso de uso 1: respuesta del servidor como texto/fragmento de HTML

En este caso, el cliente hace una petición AJAX y el servidor responde con un fragmento de texto plano o de HTML que el cliente mostrará en la posición adecuada de la página actual. Por ejemplo, el típico caso del formulario de registro en que cuando llenamos el campo de login queremos ver si está disponible, antes de rellenar los siguientes campos. Cuando el foco de teclado sale del campo de login, se hace una petición AJAX al servidor, que nos devolverá simplemente un mensaje indicando si está disponible o no.

La página HTML del cliente con el javascript que hace la petición AJAX y recibe la respuesta podría ser algo como:

**AJAX y Javascript**

Mostraremos aquí el código javascript del cliente para tener el ejemplo completo, aunque no podemos ver con detalle cómo funciona al no ser materia directa del curso. Usamos la librería jQuery en los ejemplos para simplificar al máximo el código.



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
<form id="registro" action="#">
	Login: <input type="text" name="login" id="campo_login">
		<span id="mensaje"></span><br>
	Password: <input type="password" name="password"> <br>
	Nombre y apellidos: <input type="text" name="nombre"> <br>
	<input type="submit" value="registrar">
</form>
<script type="text/javascript">
	$('#campo_login').blur(
		function() {
			$('#mensaje').load('loginDisponible.do',
"login="+$('#campo_login').val())
		}
	)
</script>
```

Del código jQuery anterior baste decir que cuando el foco de teclado se va (evento 'blur') del campo con id "campo_login" es cuando queremos disparar la petición AJAX. El método load() de jQuery lanza una petición AJAX a una determinada url con determinados parámetros y coloca la respuesta en la etiqueta HTML especificada (en este caso la de id "mensaje", un span que tenemos vacío y preparado para mostrar el mensaje).

El código Spring en el servidor, que respondería a la petición AJAX, sería el siguiente:

```
@Controller
public class UsuarioController {
	@Autowired
	private IUsuarioBO ubo;

	@RequestMapping("/loginDisponible.do")
	public @ResponseBody String loginDisponible(@RequestParam("login")
String login) {
		if (ubo.getUsuario(login)==null)
			return "login disponible";
		else
			return "login <strong>no</strong> disponible";
	}

}
```

La única diferencia con lo visto en la sesión anterior es que el valor de retorno del método no debe ser interpretado por Spring como el nombre lógico de una vista. Debe ser el contenido de la respuesta que se envía al cliente. Esto lo conseguimos anotando el valor de retorno del método con `@ResponseBody`. Cuando el valor de retorno es un String, como en este caso, simplemente se envía el texto correspondiente en la respuesta HTTP. Como veremos, si es un objeto Java cualquiera se serializará automáticamente.

Evidentemente no es una muy buena práctica tener "empotrados" en el código Java directamente los mensajes que queremos mostrar al usuario, pero este trata de ser un ejemplo sencillo. En un caso más realista usaríamos el soporte de internacionalización de Spring para externalizar e internacionalizar los mensajes. O quizá sería el propio javascript el que mostraría el mensaje adecuado.

### 7.1.2. Caso de uso 2: respuesta del servidor como objeto serializado



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

Continuando con el ejemplo anterior, supongamos que si el login no está disponible, queremos, además de saberlo, obtener como sugerencia algunos logins parecidos que sí estén disponibles, como se hace en muchos sitios web.

En lugar de enviarle al cliente simplemente un mensaje, le enviaremos un objeto con un campo booleano que indique si el login está disponible o no, y una lista de Strings con las sugerencias. En caso de que esté disponible, no habría sugerencias. La clase Java que encapsularía esta información desde el lado del servidor sería algo como lo siguiente (no se muestran constructores, getters o setters, solo las propiedades)

```
public class InfoLogin {
        private boolean disponible;
        private List<String> sugerencias;

        ...
}
```

Y el método de Spring que respondería a la petición HTTP ahora devolvería un objeto `InfoLogin`.

```
@Controller
public class UsuarioController {
        @Autowired
        private IUsuarioBO ubo;

        @RequestMapping("/loginDisponible.do")
        public @ResponseBody InfoLogin loginDisponible(
                                                @RequestParam("login") String
login) {
                if (ubo.getUsuario(login)==null)
                        //Si está disponible, no hacen falta sugerencias
                        return new InfoLogin(true, null);
                else
                        //si no lo está, generamos las sugerencias con la
ayuda del IUsuarioBO
                        return new InfoLogin(false,
ubo.generarSugerencias(login));
        }

}
```

Por lo demás, como se ve, a nivel de API de Spring no habría cambios. Automáticamente se serializará el objeto al formato adecuado. Por defecto, lo más sencillo en Spring es generar JSON. Si usamos Maven, basta con incluir en el proyecto la dependencia de la librería Jackson, una librería Java para convertir a/desde JSON que no es propia de Spring, pero con la que el framework está preparado para integrarse:

```
<dependency>
    <groupId>org.codehaus.jackson</groupId>
    <artifactId>jackson-mapper-asl</artifactId>
    <version>1.9.11</version>
</dependency>
```

En el lado del cliente, el Javascript debería obtener el objeto enviado desde Spring



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

(sencillo si lo que se envía es JSON) y mostrar las sugerencias en el HTML. Simplificando, podría ser algo como lo siguiente:

```
<form id="registro" action="#">
    Login: <input type="text" name="login" id="campo_login">
           <span id="mensaje"></span><br>
    Password: <input type="password" name="password"> <br>
    Nombre y apellidos: <input type="text" name="nombre"> <br>
    <input type="submit" value="registrar">
</form>
<script type="text/javascript">
$('#campo_login').blur(
        function() {
                $.getJSON('loginDisponible.do',
                                "login="+$('#campo_login').val(),
                                function(obj) {
                                        var mens;
                                        if (obj.disponible)
                                                mens = "El login está
disponible";
                                        else {
                                                mens = "El login no está
disponible. Sugerencias: ";
                                                for (var i=0;
i<obj.sugerencias.length; i++) {
                                                        mens +=
obj.sugerencias[i] + " ";
                                                }
                                        }
                                        $('#mensaje').html(mens);
                                }
                )
        }
)
</script>
```

Si quisiéramos serializar el objeto en formato XML, bastaría con anotarlo con anotaciones JAXB, como se vio en las sesiones de servicios REST del módulo de Componentes Web (solo se muestran los getters relevantes por estar anotados)

```
@XmlRootElement
public class InfoLogin {
        private boolean disponible;
        private List<String> sugerencias;

        @XmlElement
        public boolean isDisponible() {
                return disponible;
    }

        @XmlElementWrapper(name="sugerencias")
        @XmlElement(name="sugerencia")
        public List<String> getSugerencias() {
                return sugerencias;
        }

        ...
}
```

### 7.1.3. Caso de uso 3: enviar objetos desde el cliente



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

En AJAX lo más habitual es que el cliente envíe los datos a través de un formulario HTML. Ya vimos en la primera sesión de Spring MVC cómo tratar ese caso de uso, recordemos que los datos se podían "empaquetar" automáticamente en un objeto Java y, como ya veremos, validar declarativamente con JSR303. Pero también podríamos hacer que el cliente envíe al servidor un objeto serializado en JSON o XML. Este objeto se envía entonces en el cuerpo de la petición HTTP del cliente y el trabajo de Spring es deserializarlo y "transformarlo" a objeto Java. En el método del controller que responda a la petición simplemente anotamos el parámetro que queremos "vincular" al objeto con `@RequestBody`.

Continuando con el ejemplo del registro de usuarios, supongamos que queremos enviar desde el cliente el nuevo usuario en formato JSON (por ejemplo, porque usamos un cliente de escritorio). Desde el lado del servidor bastaría con usar `@RequestBody`:

```
@RequestMapping("/altaUsuario.do")
public void altaUsuario(@RequestBody Usuario usuario) {
	...
}
```

Para completar el ejemplo, mostraremos el código correspondiente del lado del cliente. Este código envía los datos del formulario pero en formato JSON. Como en los ejemplos anteriores, usamos el API de jQuery para la implementación, consultar su documentación para más información de cómo funciona el código.

```
	<form id="registro" action="#">
		Login: <input type="text" name="login" id="login"> <span
id="mensaje"></span><br>
		Password: <input type="password" name="password"
id="password"> <br>
		Nombre y apellidos: <input type="text" name="nombre"
id="nombre"> <br>
		<input type="submit" value="registrar">
	</form>
	<script type="text/javascript">
		$('#registro').submit(function(evento) {
			$.ajax({
				url: 'altaUsuario.do',
				type: 'POST',
				data: JSON.stringify({login:
$('#login').val(),
					password: $('#password').val(),
nombre: $('#nombre').val()}),
				processData: false,
				contentType: "application/json"
			})
			evento.preventDefault();
		});
	</script>
```

## 7.2. Servicios web REST



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

Desde Spring 3.0, el módulo MVC ofrece soporte para aplicaciones web RESTful, siendo precisamente esta una de las principales novedades de esta versión. Actualmente Spring ofrece funcionalidades muy similares a las del estándar JAX-RS, pero perfectamente integradas con el resto del *framework*. Nos limitaremos a explicar aquí el API de Spring para REST obviando los conceptos básicos de esta filosofía, que ya se vieron en el módulo de componentes web.

### 7.2.1. URIs

Como ya se vio en el módulo de servicios web, en aplicaciones REST cada recurso tiene una URI que lo identifica, organizada normalmente de modo jerárquico. En un sistema en el que tenemos ofertas de alojamiento de distintos hoteles, distintas ofertas podrían venir identificadas por URLs como:

```
/hoteles/excelsiorMad/ofertas/15
/hoteles/ambassador03/ofertas/15 (el id de la oferta es único solo dentro
del hotel)
/hoteles/ambassador03/ofertas/   (todas las del hotel)
```

Las ofertas aparecen determinadas primero por el nombre del hotel y luego por su identificador. Como puede verse, parte de la URL es la misma para cualquier oferta, mientras que la parte que identifica al hotel y a la propia oferta es variable. En Spring podemos expresar una URL de este tipo poniendo las partes variables entre llaves: `/hoteles/{idHotel}/ofertas/{idOferta}`. Podemos asociar automáticamente estas partes variables a parámetros del método java que procesará la petición HTTP. Por ejemplo:

```
@Controller
public class OfertaRESTController {
    @Autowired
    IOfertasBO obo;

    @RequestMapping(value="hoteles/{idHotel}/ofertas/{idOferta}",
                                method=RequestMethod.GET)
    @ResponseBody
    public Oferta mostrar(@PathVariable String idHotel, @PathVariable int
idOferta) {
        Oferta oferta =  obo.getOferta(idHotel, idOferta);
        return oferta;
    }
}
```

Las partes variables de la URL se asocian con los parámetros Java del mismo nombre. Para que esta asociación funcione automáticamente, hay que compilar el código con la información de debug habilitada. En caso contrario podemos asociarlo explícitamente: `@PathVariable("idHotel") String idHotel`. Spring puede convertir las `@PathVariable` a los tipos más típicos: String, numéricos o Date.

Nótese que Spring no asociará este *controller* a una URI como `hoteles/cr124/ofertas/`, ya que se espera una `PathVariable` para la oferta que aquí



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

no existe. Lo que tiene su lógica, ya que esta URL significa que queremos hacer alguna operación con *todas* las ofertas del hotel, y esto es mejor que sea tarea de otro método Java distinto a `mostrar`.

Nótese además que el método java `mostrar` viene asociado solo a las peticiones de tipo GET. En una aplicación REST, típicamente el método Java encargado de editar una oferta se asociaría a la misma URL pero con PUT, y lo mismo pasaría con insertar/POST, y borrar/DELETE

Por otro lado, como ya hemos visto en la sección de AJAX, la anotación `@ResponseBody` hace que lo que devuelve el método Java se serialice en el cuerpo de la respuesta HTTP que se envía al cliente. La serialización en JSON o XML se hace exactamente de la misma forma que vimos en AJAX.

Nótese que el ejemplo anterior se podría modificar de modo sencillo para una aplicación web convencional con JSP, como veníamos haciendo en las sesiones de MVC. Únicamente habría que poner un parámetro de tipo `Model`, añadirle el objeto `oferta` encontrado y devolver un `String` con el nombre lógico de la vista, donde se mostrarían los datos.

### 7.2.2. Obtener recursos (GET)

La implementación del apartado anterior era demasiado básica, ya que no estamos controlando explícitamente elementos importantes de la respuesta HTTP como el código de estado o las cabeceras. Para controlarlos, en Spring podemos hacer uso de la clase `ResponseEntity`, que modela la respuesta HTTP y con la que podemos devolver los objetos serializados, fijar el código de estado y añadir cabeceras. Vamos a ver cómo se implementaría una versión algo más sofisticada y con una sintaxis más "propia de REST"

```
@Controller
@RequestMapping("/hoteles/{idHotel}/ofertas")
public class OfertasController {
        @Autowired
        IOfertasBO obo;


        @RequestMapping(method=RequestMethod.GET,
                        value="{idOferta}",
                        produces=MediaType.APPLICATION_JSON_VALUE)
        public ResponseEntity<Oferta> obtenerOferta(@PathVariable String
idHotel,
                                                    @PathVariable int
idOferta)
                                                      throws
OfertaInexistenteException {
                Oferta oferta = obo.getOferta(idHotel, idOferta);
                return new ResponseEntity<Oferta>(oferta, HttpStatus.OK);
        }
}
```

Del código anterior destacar que:



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

- Para simplificar las URIs de cada método se hace que el controller en general responda a la parte "fija" y común a todas las URIs de las que se encargará, y en cada método se pone solo la parte que va a continuación. Nótese que los parámetros del método pueden referenciar cualquier `@PathVariable` de la URI, aunque aparezca en la anotación de la clase (como el id. del hotel).
- Usamos el atributo `produces` de `@RequestMapping` para fijar el valor de la cabecera HTTP "content-type", al igual que hacíamos en AJAX.
- Como ya hemos dicho, la clase `ResponseEntity` representa la respuesta HTTP. Cuando en el cuerpo de la respuesta queremos serializar un objeto usamos su clase para parametrizar `ResponseEntity`. Hay varios constructores de esta clase. El más simple admite únicamente un código de estado HTTP (aquí 200 OK). El que usamos en el ejemplo tiene además otro parámetro en el que pasamos el objeto a serializar.
- Nótese que en caso de solicitar una oferta inexistente se generaría una excepción, lo que en la mayoría de contenedores web nos lleva a una página HTML de error con la excepción, algo no muy apropiado para un cliente REST, normalmente no preparado para recibir HTML. Veremos luego cómo arreglar esto, convirtiendo automáticamente las excepciones en códigos de estado HTTP.
- Ya no hace falta la anotación `@ResponseBody` ya que al devolver un `ResponseEntity<Oferta>`, ya estamos indicando que queremos serializar un objeto en la respuesta HTTP.

### 7.2.3. Crear o modificar recursos (POST/PUT)

En este caso, el cliente envía los datos necesarios para crear el recurso y el servidor le debería responder, según la ortodoxia REST, con un código de estado 201 (Created) y en la cabecera "Location" la URI del recurso creado. Veamos cómo se implementaría esto en Spring:

```
@RequestMapping("/hoteles/{idHotel}/ofertas")
public class OfertasController {
        @Autowired
        IOfertasBO obo;

        //Ya no se muestra el código de obtenerOferta
        //...

        @RequestMapping(method=RequestMethod.POST, value="")
        public ResponseEntity<Void> insertarOferta(@PathVariable idHotel,
                                                   @RequestBody Oferta
oferta,
                                                   HttpServletRequest
peticion) {
                int idOFerta = obo.crearOferta(idHotel, oferta);
                HttpHeaders cabeceras = new HttpHeaders();
                try {
                        cabeceras.setLocation(new
URI(peticion.getRequestURL()+
Integer.toString(idOFerta)));
                } catch (URISyntaxException e) {
                        e.printStackTrace();
                }
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
                return new ResponseEntity<Void>(cabeceras,
HttpStatus.CREATED);
        }
}
```

A destacar del código anterior:

- Como no tenemos que serializar ningún objeto en el cuerpo de la respuesta, usamos un `ResponseEntity<Void>`. Le pasamos la cabecera "Location" generada y el código de estado 201, indicando que el recurso ha sido creado.
- Para generar la cabecera "Location" usamos el API de Spring (la clase `HttpHeaders`). La URI del nuevo recurso será la actual seguida del id de la nueva oferta.
- La anotación `@RequestBody` indica que en el cuerpo de la petición debemos enviar un objeto Oferta serializado. Esto guarda un paralelismo con el caso de uso de AJAX en que el cliente envía un objeto JSON al servidor. Nótese que el API es exactamente el mismo.

El caso de modificar recursos con PUT es similar en el sentido de que se envían datos serializados en la petición, pero es más sencillo en cuanto al valor de retorno, ya que nos podemos limitar a devolver un código de estado 200 OK si todo ha ido bien, sin necesidad de más cabeceras HTTP:

```
@RequestMapping(method=RequestMethod.PUT, value="{idOferta}")
public ResponseEntity<Void> insertarOferta(@PathVariable idHotel,
                                           @RequestBody Oferta oferta) {
        int idOFerta = obo.modificarOferta(idHotel, oferta);
        return new ResponseEntity<Void>(HttpStatus.CREATED);
}
```

### 7.2.4. Eliminar recursos (DELETE)

Este caso de uso suele ser sencillo, al menos en lo que respecta a la interfaz. Simplemente debemos llamar a la URI del recurso a eliminar, y devolver 200 OK si todo ha ido bien:

```
@RequestMapping(method=RequestMethod.DELETE, value="{idOferta}")
public ResponseEntity<Void> borrarOferta(@PathVariable
idHotel,@PathVariable idOferta) {
        obo.eliminarOferta(idHotel, idOferta);
        return new ResponseEntity<Void>(HttpStatus.OK);
}
```

### 7.2.5. Parte del cliente

El código del cliente REST podría escribirse usando directamente el API de JavaSE, o librerías auxiliares como Jakarta Commons HttpClient, que permite abrir conexiones HTTP de manera sencilla. No obstante, que sea sencillo no implica que no sea tedioso y necesite de bastantes líneas de código. Para facilitarnos la tarea, Spring 3 ofrece la clase `RestTemplate`, que permite realizar las peticiones REST en una sola línea de código Java.



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

| Método HTTP | Método de RestTemplate |
|---|---|
| DELETE | delete(String url, String? urlVariables) |
| GET | getForObject(String url, Class<T> responseType, String? urlVariables) |
| HEAD | headForHeaders(String url, String? urlVariables) |
| OPTIONS | optionsForAllow(String url, String? urlVariables) |
| POST | postForLocation(String url, Object request, String? urlVariables) |
| | postForObject(String url, Object request, Class<T> responseType, String? uriVariables) |
| PUT | put(String url, Object request, String?urlVariables) |

**Table 1: Métodos de RestTemplate y su correspondencia con los de HTTP**

Por ejemplo, una petición GET a la URL `hoteles/ambassador/ofertas/1` se haría:

```
RestTemplate template = new RestTemplate();
String uri =
"http://localhost:8080/ServidorREST/hoteles/{idHotel}/ofertas/{idOferta}";
Oferta oferta = template.getForObject(uri, Oferta.class, "ambassador", 1);
System.out.println(oferta.getPrecio() + "," + oferta.getFin());
```

Como se ve, en el cliente se usa la misma notación que en el servidor para las URLs con partes variables. En los parámetros del método Java `getForObject` se coloca el tipo esperado (la clase `Oferta`) y, por orden, las partes variables de la URL. Igual que en el servidor, si tenemos las librerías de Jackson en el *classpath* se procesará automáticamente el JSON. Así, Jackson se encarga de transformar el JSON de nuevo en un objeto Java que podemos manipular de modo convencional.

Si hubiéramos implementado en el servidor un método en el controller para dar de alta ofertas, podríamos llamarlo desde el cliente así:

```
RestTemplate template = new RestTemplate();
String uri =
"http://localhost:8080/TestRESTSpring3/hoteles/{idHotel}/ofertas";
//aquí le daríamos el valor deseado a los campos del objeto
Oferta oferta = new Oferta( ..., ..., ...);
URI result = template.postForLocation(uri, oferta, "ambassador");
```

El método de `RestTemplate` llamado `postForLocation` crea un nuevo recurso, y obtiene su URI, que necesitaremos para seguir interactuando con el objeto. La clase URI es del API estándar de JavaSE. En una aplicación REST el servidor debería devolver dicha URI como valor de la cabecera HTTP `Location`.

## 7.3. Tratamiento de errores en aplicaciones AJAX y REST



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

Como ya se ha visto en el módulo de componentes web, podemos especificar qué página HTML deseamos que se muestre cuando se genera una excepción en la capa web que acaba capturando el contenedor. Tanto la página de error que muestra el contenedor como la información que aparece en ella son totalmente configurables, dando una solución aceptable para aplicaciones web "clásicas". No obstante este mecanismo de gestión de errores no es apropiado para aplicaciones AJAX o REST, ya que el HTML con la información de error no es un formato apropiado ni para el javascript en el primer caso ni para el cliente REST en el segundo. Lo más adecuado en estos casos es devolver un código de estado HTTP que indique de la manera más precisa posible qué ha pasado y en el cuerpo de la respuesta un mensaje en texto plano con más datos sobre el error. La forma más sencilla de implementar esto en Spring es mediante el uso de *exception handlers*.

Un *exception handler* no es más que un método cualquiera del *controller* anotado con `@ExceptionHandler`. Este método capturará el o los tipos de excepciones que deseemos, y lo que haremos en el método será devolver la excepción en una forma amigable a los clientes AJAX/REST. Por ejemplo, para gestionar el caso de las ofertas inexistentes podríamos hacer algo como:

```
@ExceptionHandler(OfertaInexistenteException.class)
public ResponseEntity<String>
gestionarNoExistentes(OfertaInexistenteException oie) {
        return new ResponseEntity<String>(oie.getMessage(),
HttpStatus.NOT_FOUND);
}
```

En el ejemplo simplemente enviamos un código 404 y en el cuerpo de la respuesta colocamos el mensaje de la excepción, aunque podríamos hacer cualquier otra cosa que deseáramos (colocar cabeceras, serializar un objeto en la respuesta, o incluso devolver un String que se interpretaría, como es habitual, como el nombre de una vista que mostrar).

La anotación `@ExceptionHandler` admite varias excepciones, de modo que podemos usar un único método gestor para varias distintas, por ejemplo:

```
@ExceptionHandler({OfertaInexistenteException.class,
HotelInexistenteException.class})
public ResponseEntity<String> gestionarNoExistentes(Exception e) {
        return new ResponseEntity<String>(e.getMessage(),
HttpStatus.NOT_FOUND);
}
```

Spring MVC genera unas cuantas excepciones propias, caso por ejemplo de que se produzca un error de validación en un objeto que hemos querido chequear con JSR303, ya veremos cómo (`BindException`),o que el cliente no acepte ninguno de los formatos que podemos enviarle (`HttpMediaTypeNotAcceptableException`), o que intentemos llamar con POST a un método que solo acepta GET (`HttpRequestMethodNotSupportedException`), entre otros. En esos casos actúa un *exception handler* definido por defecto que lo único que hace es capturar las excepciones



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

y generar códigos de estado HTTP (400 en el primer caso, 406 en el segundo y 405 en el tercero). Si queremos que se haga algo más, como enviar un mensaje con más información en el cuerpo de la respuesta, tendremos que definir nuestros propios *handlers* para esas excepciones.



*Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.*

# 8. Ejercicios de AJAX y REST

## 8.1. AJAX (1 punto)

Vamos a cambiar la búsqueda de usuarios para que funcione mediante AJAX. Así los resultados se mostrarán directamente en la página de búsqueda sin necesidad de cambiar de página. Seguid estos pasos:

1. Incluir la librería Jackson en el pom.xml, puedes copiar la dependencia que aparece de los apuntes
2. **Parte del servidor**: añádele un método `buscarAJAX` al BuscarAmigosController, que se ocupe de esta funcionalidad.
   - El @RequestMapping puede ser a la misma URL que el buscarAmigos "convencional", con el mismo método POST, pero debes indicar que genera JSON usando el atributo `produces`
   - El método devolverá una List<Usuario>. Debes anotar el valor de retorno con @ResponseBody para que se devuelva en la respuesta HTTP.
3. **Parte del cliente**: copia el siguiente código dentro de la página busqueda.jsp, después del formulario

```
<div id="resultados"></div>
<script type="text/javascript"
src="http://code.jquery.com/jquery.min.js"></script>
<script>
function crearTabla(obj) {
        var tabla = "<table><tr> <th>Login</th> <th>Edad</th>
<th>Localidad</th> </tr>"
        for(i=0; i<obj.length; i++) {
          tabla += "<tr> <td>" + obj[i].login + "</td> <td>" + obj[i].edad
+
             "</td> <td>" + obj[i].localidad + "</td> </tr>"
        }
        tabla += "</table>"
        alert(JSON.stringify(obj, undefined, 2))
        //hide y show solo se usan aquí para molar
        $('#resultados').hide().html(tabla).show('slow')
}

function buscarAJAX() {
        $.ajax({
                type: 'POST',
                url: 'busqueda.do',
                dataType: 'json',
                data: $('#formulario').serialize(),
                success: crearTabla
        })
}
</script>
```

4. Para que el código anterior funcione, debes ponerle al formulario un atributo id="formulario". Además, debes añadir dentro del formulario un nuevo botón para la búsqueda con AJAX. No es necesario que borres el anterior, así puedes tener los dos



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

tipos de búsqueda:

```
<input type="button" value="buscar con AJAX" onclick="buscarAJAX()"/>
```

5. Cuando lo pruebes, primero verás que aparece en un cuadro de diálogo el JSON que nos envía el servidor (línea y luego se muestra la tabla formateada en HTML. Fijate en que aunque en el HTML no se ve el password del usuario, desde el servidor sí nos lo está enviando, lo que es un problema de seguridad. Puedes anotar la clase Usuario con la anotación Jackson `@JsonIgnoreProperties({"password", "credito"})`, para que la librería no serialice ninguno de los dos campos.
6. Una vez hecho lo anterior, puedes quitar la línea 11 (el 'alert'), solamente estaba puesta para que vieras "en crudo" los datos que envía el servidor

## 8.2. Servicios REST (1.5 puntos)

Queremos hacer una implementación REST de algunas funcionalidades de la aplicación. En concreto, queremos implementar:

- La búsqueda de usuarios
- Obtener los datos de un usuario a partir de su login ("ver amigo")
- Dar de alta un usuario

Debéis seguir estos pasos:

1. **Configuración**: usaremos otro dispatcher servlet adicional al que ya tenemos. Así podemos configurarlo por separado. En el web.xml introduce el siguiente código:

```
<servlet>
   <servlet-name>rest</servlet-name>
<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
   <load-on-startup>2</load-on-startup>
</servlet>
<servlet-mapping>
   <servlet-name>rest</servlet-name>
   <url-pattern>/REST/*</url-pattern>
</servlet-mapping>
```

como ves, le asociamos las URL que comienzan por /REST. Ahora crearemos el fichero de configuración .xml para dicho servlet, que se llamará rest-servlet.xml y estará en WEB-INF, siguiendo la convención por defecto de Spring MVC:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:context="http://www.springframework.org/schema/context"
        xmlns:mvc="http://www.springframework.org/schema/mvc"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd
                http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.2.xsd
                http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd">

        <context:component-scan
base-package="es.ua.jtech.amigosspring.rest" />
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
        <mvc:annotation-driven />
</beans>
```

2. **Crear el controller**: Crea el paquete `es.ua.jtech.amigosspring.rest`. Crea dentro de él la clase `AmigosRESTController`. Anótala con @Controller y mapéala con @RequestMapping a la URL "amigos"
3. Ahora "solo" te queda **implementar las tres funcionalidades**. Recuerda que:
   - Debes respetar la "filosofía" REST en cuanto al uso de los métodos HTTP. Buscar usuarios y ver un usuario usarán GET, mientras que crear usuario debería usar POST
   - Asímismo debes mapear adecuadamente las URLs. Buscar y crear usarán la misma URL que el controller "entero" (amigos), pero la URL de "ver amigo" debe acabar por el login del usuario que quieres ver (por ejemplo "amigos/jsf")
   - Buscar y ver usuario producen JSON, mientras que crear usuario consume JSON

Para probar los servicios REST puedes usar el cliente REST-shell, desarrollado por SpringSource. Si quieres más información puedes ver su repositorio en github. Ejecuta el programa "rest-shell" que está dentro del subdirectorio "bin" de "rest-shell-1.2.1.RELEASE". Es un prompt que acepta comandos que permiten hacer peticiones REST de modo relativamente sencillo. En todo momento podemos ver la ayuda general con 'help' o solo sobre un comando determinado, por ejemplo 'help get'.

Por ejemplo, para probar el "ver usuario" haríamos

```
#Establece la URL "base" por defecto para todas las peticiones a partir de
aquí
base http://localhost:8080/AmigosSpring/REST/amigos
#Para decirle al servidor que nos puede mandar JSON. Cabecera HTTP
"Accept:"
headers set --name Accept --value application/json
#hacer una petición GET a una URL
get jsf
```

Para buscar usuarios, suponiendo que ya hemos establecido la URL base y hemos fijado la cabecera Accept para aceptar JSON:

```
#podemos pasar parámetros HTTP escribiéndolos en formato JSON
#(pero no se envía JSON, se envían en la forma
param1=valor1&param2=valor&...)
get --params "{'edadMin':9, 'edadMax':90, 'sexo':'indiferente'}"
```

Para crear un nuevo usuario tendremos que hacer una petición POST y subir los datos JSON en el cuerpo de la petición.

```
#decimos que vamos a enviar JSON
headers set --name Content-Type --value application/json
base http://localhost:8080/AmigosSpring/REST/amigos
#enviamos los datos JSON (cuidado, meter todo lo siguiente en una sola
línea)
post --data "{'login':'test','password':'test','fechaNac':'1990-01-01',
    'localidad':'javaland', 'varon':'true', 'descripcion':'holaaaaaa'}"
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

**Every little thing gonna be alright...**

Por el momento vamos a suponer que no se produce ningún error al crear el nuevo usuario. En el último ejercicio gestionaremos alguna de las posibles excepciones y en la siguiente sesión veremos cómo validar los datos usando JSR303 (por ejemplo, que la fecha de nacimiento está en el pasado). Así que no probéis con datos erróneos ni intentéis crear un usuario con un login ya existente.

## 8.3. Gestión de errores en servicios REST (0.5 puntos)

Lo habitual en REST es devolver un código de estado HTTP si se ha producido un error y algún mensaje en el cuerpo de la respuesta con más información. Vamos a hacer esto en nuestro servicio:

- En primer lugar, cuando se quiere ver un usuario por login, si no existe se debe devolver un código de estado 404. Hazlo si no lo has hecho ya.
- Por otro lado, cuando se genera una excepción (por ejemplo al intentar crear dos usuarios con el mismo login) se ve en el cliente, lo que no es apropiado. Usa la gestión de excepciones que hemos visto en la sesión para transformar las posibles RuntimeException que devuelve el DAO en códigos de estado 400 (sí, le vamos a echar la culpa de todo al cliente, para simplificar el ejercicio). En el cuerpo de la respuesta envía el mensaje de la excepción, obtenido con getMessage().



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

# 9. Validación e internacionalización con Spring MVC

## 9.1. Validación en Spring

Desde la versión 3, Spring ha simplificado y potenciado en gran medida la validación de datos, gracias a la adopción de la especificación JSR 303. Este API permite validar los datos de manera declarativa, con el uso de anotaciones. Esto nos facilita la validación de los datos introducidos en formularios HMTL antes de llegar al controlador. Nosotros veremos aquí la validación en el módulo MVC, que es lo que nos interesa, aunque ésta se puede aplicar a cualquier capa de nuestra aplicación Spring.

### 9.1.1. JSR 303 - Bean Validation

Como ya hemos comentado, la especificación JSR 303 permite especificar la validación de datos de manera declarativa, usando anotaciones. La implementación de referencia de este JSR es Hibernate Validator, de la que recomendamos consultar su documentación . No obstante, en nuestra discusión vamos a restringirnos al estándar, así que todo lo que vamos a ver es válido para cualquier otra implementación.

Veamos un ejemplo que nos mostrará lo sencillo y potente que es este API. Siguiendo con las ofertas de hoteles del tema anterior, la siguiente clase representaría una reserva de habitación hecha por un cliente, expresando las restricciones que debe cumplir cada campo con JSR 303:

```
package es.ua.jtech.spring.modelo;
import java.util.Date;
import javax.validation.Valid;
import javax.validation.constraints.Future;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;

public class Reserva {
    @Future
    private Date entrada;
    @Range(min=1,max=15)
    private int noches;
    @Min(10)
    private BigDecimal pagoAnticipado;
    @NotNull
    private TipoHabitacion tipohabitacion;
    @NotNull
    private Cliente cliente;
    //ahora vendrían getters y setters
    ...
```

Aquí hemos especificado que la fecha de entrada en el hotel debe ser posterior a la fecha actual (la del sistema), se puede reservar entre 1 y 15 noches de estancia, por anticipado se debe pagar un mínimo de 10 euros y que el cliente y el tipo de habitación no pueden ser nulos. Nótese que especificar una restricción no implica que el objeto la deba cumplir



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

siempre (evidentemente en los históricos habrá reservas con fecha de entrada en el pasado). Simplemente, la especificación nos proporciona una manera sencilla de expresar las restricciones y de comprobar si se cumplen en el momento que nos interese, llamando a un método del API. Normalmente dispararemos la validación al dar de alta o editar el objeto. Aunque disparar la validación es muy sencillo usando directamente el API del JSR (solamente hay que llamar a `Validator.validate()` sobre el objeto a validar) veremos que en Spring es todavía más simple, gracias a su integración con dicho API.

Aunque como se ha visto se pueden anotar las propiedades, también se pueden anotar los getters para obtener el mismo efecto (pero NO se deben anotar los setters).

La especificación ofrece un amplio conjunto de restricciones predefinidas. Hibernate Validator proporciona algunas adicionales, y además el usuario puede definirse las suyas propias. Aunque no es excesivamente complicado esto queda fuera del alcance de este tema. Se recomienda consultar la documentación de Hibernate Validator para ver la lista de las restricciones predefinidas y si lo deseas, sobre la definición de restricciones propias del usuario.

Por defecto, si un objeto referencia a otros, al validarlo no se comprobarán las restricciones de los referenciados. Aunque en algunos casos nos puede interesar lo contrario, por ejemplo validar al `Cliente` que ha hecho la reserva. Para esto se usa la anotación `@Valid`:

```
public class Reserva {
    ...
    @NotNull
    @Valid
    private Cliente cliente;
    //ahora vendrían getters y setters
    ...
```

Esta "validación recursiva" se puede aplicar también a colecciones de objetos, de modo que se validarán todos los objetos de la colección. Por ejemplo, si en una misma reserva se pudieran reservar varias habitaciones a la vez, podríamos tener algo como:

```
public class Reserva {
    ...
    @NotNull
    @NotEmpty
    @Valid
    private List<TipoHabitacion> habitaciones;
    ...
```

Donde la anotación `@NotEmpty` significa que la colección no puede estar vacía.

### 9.1.2. Validación en Spring MVC

La activación de la validación JSR 303 en Spring MVC es muy sencilla. Simplemente hay que asegurarse de que tenemos en el classpath una implementación del JSR. (la más



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

típica es Hibernate Validator, como ya hemos comentado) y poner en el XML de definición de beans de la capa web la etiqueta `<mvc:annotation-driven/>`.

En la capa MVC el caso de uso más típico es validar el objeto cuando el usuario haya rellenado sus propiedades a través de un formulario. Normalmente en Spring MVC ese objeto nos llegará como parámetro del método del controller que procese los datos del formulario. Para validar el objeto antes de pasárselo al método lo único que hay que hacer es anotar el parámetro con `@Valid`. Por ejemplo:

```
package es.ua.jtech.spring.mvc;

import es.ua.jtech.spring.modelo.Reserva;
import javax.validation.Valid;
import org.springframework.stereotype.Controller;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping("/reserva")
public class ReservaController {

    @RequestMapping(method=RequestMethod.POST)
    public String efectuarReserva(@Valid Reserva reserva,
                                  BindingResult result) {
        if (result.hasErrors())
           return "rellenarReserva";
    }
}
```

Como vemos, la validación declarativa se integra con el API de Spring de manejo de errores. Los posibles errores de validación se almacenarán en el parámetro de tipo `BindingResult`, dándonos la posibilidad de examinarlos con el API de Spring, o saltar de nuevo al formulario y mostrarlos con la etiqueta `<form:error>`, como ya vimos en el tema anterior.

Los mensajes de error se buscarán en un fichero .properties, al igual que en el tema anterior. La clave bajo la que se buscará el mensaje es generalmente el nombre de la restricción. El primer parámetro que se suele pasar es este nombre, y a partir de aquí los parámetros de la anotación. Por ejemplo, en el caso de la restricción `@Min(1)` para las noches de estancia podríamos hacer en el .properties

```
Min.noches = hay un mínimo de {1} noche de estancia
```

Cuidado, el `{1}` no significa literalmente un 1, sino que hay que sustituir por el segundo argumento (recordemos que empiezan en 0). En este caso la sustitución sería por el valor 1, pero así podemos cambiar la restricción para forzar más de una noche de estancia sin cambiar el código del mensaje de error.

Aunque por defecto la clave bajo la que se busca el mensaje en el properties es el nombre de la restricción, podemos cambiarla en la propia anotación, por ejemplo:



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
public class Reserva {
   ...
   @Min(value=1,message="minimoNoches")
   private int noches;
   ...
}
```

Podemos efectuar la validación llamando directamente al API JSR303. Esto puede ser útil en los casos en que no podamos usar la anotación @Valid en nuestro código. Por ejemplo, @Valid no funciona en la versión 3.0 (aunque sí a partir de la 3.1 inclusive) cuando los datos de entrada vienen el cuerpo de la petición como JSON o XML en lugar de ser parámetros HTTP (típico de REST). Aun en ese caso no hay mucho problema, ya que el API JSR303 es bastante sencillo:

```
ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
Validator validator = factory.getValidator();
Set<ConstraintViolation<Oferta>> errores = validator.validate(oferta);
for (ConstraintViolation<Oferta> cv : errores) {
      System.out.println(cv.getMessage());
}
```

Para aplicar la validación solo hay que construir un `Validator` a través de una `ValidatorFactory` y llamar al método `validate` sobre el objeto a validar. La aplicación de la validación produce por cada error detectado un `ConstraintViolation`, parametrizado al tipo que se está validando. Este objeto contiene información sobre el error que se ha producido. Para más información se recomienda consultar la documentación del API.

## 9.2. Internacionalización y formateo de datos

Veremos en esta sección cómo preparar nuestra aplicación para que esté adaptada al idioma del usuario. A este proceso se le denomina **internacionalización** o *i18n* para abreviar (ya que en la palabra hay 18 letras entre la 'i' inicial y la 'n' final).

### 9.2.1. Internacionalización de los textos

La parte más tediosa de la i18n de una aplicación suele ser la traducción y adaptación de los mensajes del interfaz de usuario. En aplicaciones Java internacionalizadas casi siempre los mensajes se suelen almacenar en ficheros *properties* para no tener que modificar el código Java si hay que cambiar/añadir alguna traducción. La convención habitual es que los mensajes para cada idioma se almacenan en un fichero .properties separado, pero de modo que todos los ficheros comienzan por el mismo nombre aunque como sufijo del mismo se usa el *locale* del idioma en cuestión.

Un *locale* es una combinación de idioma y país (y opcionalmente, aunque la mayoría de veces no se usa, una variante o dialecto). Tanto el país como el idioma se especifican con códigos ISO (estándares ISO-3166 e ISO-639). Aunque el locale en Java exige



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

especificar tanto idioma como país, en Spring y en casi todos los frameworks podemos usar solamente el código de idioma si no deseamos especificar más. De este modo, los mensajes internacionalizados se podrían guardar en archivos como:

```
mensajes_es_ES.properties   //Español de España
mensajes_es_AR.properties   //Español de Argentina
mensajes_es.properties      //Español genérico a usar en otro caso
mensajes_en.properties      //Inglés
mensajes.properties         //fallback, o fichero a usar si no hay otro
apropiado
```

Así, el fichero mensajes_es_ES.properties, con los mensajes en idioma español (es) para España (ES), podría contener algo como lo siguiente:

```
saludo = Hola, bienvenido a la aplicación
error= lo sentimos, se ha producido un error
```

El *framework* usará automáticamente el fichero apropiado al locale actual. En caso de poder usar varios se elegirá el que mejor encaje. Así, si en el ejemplo anterior el locale actual fuera idioma español y país España se usaría el primer archivo en lugar del español genérico (que se seleccionaría por ejemplo si el locale actual fuera idioma español y país México). Si no hay ninguno que encaje se usará el que no tiene sufijo de locale (en nuestro ejemplo, el último de los archivos).

Como hemos visto en la sesión anterior, los mensajes de error en Spring se almacenan también en ficheros .properties y su localización se define habitualmente con un bean de tipo `ResourceBundleMessageSource`. Para los textos de la interfaz web se usa el mismo mecanismo. En realidad los mensajes de error también se pueden internacionalizar automáticamente de la misma forma que los demás mensajes, simplemente generando los .properties adecuados.

```
<bean id="messageSource"
class="org.springframework.context.support.ResourceBundleMessageSource">
     <property name="basename" value="es/ua/jtech/spring/mvc/mensajes"/>
</bean>
```

Solo nos falta colocar los mensajes internacionalizados en la interfaz web. Suponiendo, como hemos hecho hasta ahora, que usamos JSP para la interfaz, podemos emplear la etiqueta `<spring:message/>`. Esta etiqueta tiene como atributo principal `code`, que representa la clave del properties bajo la que está almacenado el mensaje. Por ejemplo:

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
...
<spring:message code="saludo"/>
```

de modo que en ese punto del JSP aparecería el mensaje que en el .properties del locale actual estuviera almacenado bajo la clave "saludo".



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

### 9.2.2. Cambio del locale

Para que una aplicación pueda estar internacionalizada se tiene que guardar de alguna manera cuál es el locale con el que desea verla el usuario. El sitio más habitual para guardar el locale es la sesión o una cookie en el navegador, consiguiendo así que cada usuario pueda mantener el suyo propio. Además tiene que haber algún mecanismo para consultar cuál es el locale actual y cambiarlo. Spring da soporte a todo esto: ofrece interceptores que nos permitirán cambiar el locale sin más que llamar a determinada URL y tiene clases propias que guardan el locale del usuario actual. Veamos cómo funciona todo esto.

Por defecto Spring toma el locale de las cabeceras HTTP que envía el navegador con cada petición. Por tanto, si el usuario tiene configurado el navegador en el idioma deseado, la aplicación aparecerá automáticamente en el mismo idioma. No obstante, es posible que el usuario tenga instalado el navegador en un idioma pero prefiera ver las páginas en uno distinto. En ese caso tenemos que cambiar el lugar de donde toma Spring el locale, pasando a guardarlo en una cookie o en la sesión, según la implementación elegida de la clase que guarda el locale. Además, podemos configurar un interceptor que permitirá cambiar el locale de manera sencilla. Esto lo configuraremos en el XML de la capa web (típicamente `dispatcher-servlet.xml`)

```
...
<!-- el id debe ser "localeResolver -->
<bean id="localeResolver"
class="org.springframework.web.servlet.i18n.CookieLocaleResolver"/>

<mvc:interceptors>
   <bean
class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor"/>
</mvc:interceptors>
...
```

En el fragmento de XML anterior, el `localeResolver` es el bean encargado de guardar el locale del usuario. La implementación elegida en este caso lo guarda en una cookie, como puede deducirse del nombre de la clase. Para guardarlo en la sesión usaríamos en su lugar la clase `SessionLocaleResolver`. Como decíamos antes, por defecto en Spring el `localeResolver` usa la cabecera `Accept-language` de las peticiones del navegador, pero este tiene el problema de que no podemos cambiar el locale (si lo intentamos se generará un error).

Por otro lado, la clase `LocaleChangeInterceptor` es un interceptor. Es decir, es una clase que intercepta las peticiones HTTP antes de que lleguen a los controllers. Esta clase cambia el locale si la petición HTTP actual tiene un parámetro `locale` igual al locale deseado. Por ejemplo, la petición `index.do?locale=en` cambiaría el locale actual a inglés, además de hacer lo que haga normalmente el controller asociado a `index.do`. Nótese que esto lo podríamos hacer "a mano", obteniendo una referencia al `localeResolver` actual y llamando a un método en él para cambiar el locale, pero es



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

mucho más cómodo hacerlo con el interceptor.

### 9.2.3. Formateo de datos

Según el locale, existen convenciones distintas en el formato para mostrar fechas y números. Sin Spring esto lo haríamos con etiquetas JSTL (por ejemplo . Sin embargo, para introducir datos en formularios hay que hacer la conversión inversa: de String a Date o a número, cosa que no podemos solucionar con JSTL. En Spring 3 se introducen varias anotaciones para expresar el formato de los campos, que internamente usan clases ("formatters") capaces de convertir a y desde String a otros tipos. Ya hay anotaciones definidas para formatear fechas y números, aunque podríamos hacer nuestros propios "formateadores".

Veamos cómo podríamos formatear los campos de la clase `Reserva`:

```
public class Reserva {
    @Future
    @DateTimeFormat(style="S-")
    private Date entrada;
    @Range(min=1,max=15)
    private int noches;
    @Min(10)
    @NumberFormat(style=NumberFormat.Style.CURRENCY)
    private BigDecimal pagoAnticipado;
    @NotNull
    private TipoHabitacion tipohabitacion;
    @NotNull
    private Cliente cliente;
    //ahora vendrían getters y setters
    ...
```

La anotación `@DateTimeFormat` formatea fechas. El atributo `style` indica el estilo de formateo con dos caracteres. El primer carácter es para el estilo de la fecha y el segundo para el de la hora. Se admiten los caracteres "S" (short), "M", (medium), "L" (long), "F" (full) y el guión que indica que no nos interesa mostrar esa parte. Así, en el ejemplo, `@DateTimeFormat(style="S-")` indica que queremos ver la fecha en formato corto, sin hora. En el locale español, por ejemplo el 28 de diciembre de 2009 a las 12:00 se mostraría como 28/12/09 (recordemos que no nos interesaba la hora), mientras que en el locale inglés sería 12/28/09. Se recomienda consultar el javadoc del API de Spring para más información sobre otros modos de formatear fechas.

**La librería JodaTime**

Hasta la versión 3.1 de Spring incluida, la anotación `@DateTimeFormat` requería que la librería JodaTime estuviera presente en el classpath. Esta librería *open source* no es parte de Spring, pero se puede obtener, junto con las demás dependencias de Spring, de la web de SpringSource, o bien de su propia web. JodaTime es una implementación alternativa al `Date` y `Time` del API de Java. Si has usado alguna vez fechas en Java habrás visto que su API es potente y flexible, pero también...digamos...algo retorcido. JodaTime simplifica considerablemente el manejo de fechas, manteniendo la flexibilidad si es necesaria. A partir de la versión 3.2, ya no es obligatorio el uso de JodaTime para esta anotación, pero si está disponible se usará en lugar de la librería estándar.



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

Como habrás "adivinado", la anotación `@NumberFormat` formatea números. El parámetro style puede tener valores distintos indicando si queremos mostrar un número (con coma decimal en el locale español, o punto decimal en el ingles, por ejemplo), una moneda (aparecerá el símbolo de la moneda del locale actual) o un porcentaje.

Con esto podemos leer y convertir datos de entrada en formularios. O sea, de String al tipo deseado. Para el caso contrario, en el que queremos mostrar un dato y que aparezca formateado (de objeto a String), podemos usar la etiqueta `<spring-eval/>`

```
Fecha de entrada: <spring:eval expression="reserva.entrada" />
```

Igual que se usan para convertir datos introducidos en formularios, estas anotaciones también se pueden usar para convertir parámetros HTTP al tipo deseado, por ejemplo:

```
public class TareasController {
  @RequestMapping("tareas/crear")
  public int nuevaTarea(@RequestParam
                        @DateTimeFormat(style="S-") Date fecha, ...) {
    ...
  }
}
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

# 10. Ejercicios de validación e internacionalización

En las plantillas de la sesión tienes un proyecto ya configurado con Spring. La aplicación es para la gestión de una agencia de alquiler de coches. Se pueden listar los coches en alquiler, insertar nuevos o editar los existentes. Para simplificar no hay autentificación de usuarios.

En la carpeta "database" del proyecto tienes el script SQL con la base de datos mySQL. Ejecútalo como lo hacemos habitualmente para crear la base de datos. Comprueba que el listado de coches aparece correctamente accediendo a "listar.do".

Si quieres probar a dar de alta un coche o modificar los datos de uno existente debes introducir la fecha en un formato como 01-ene-2010 o 05-oct-2011. En el segundo ejercicio veremos cómo cambiar el formato de las fechas.

## 10.1. Conversión y formateo de datos (0.5 puntos)

Vamos a añadir la funcionalidad de que se puedan listar solo los vehículos matriculados con posterioridad a una fecha dada. Modifica el método `listar` de la clase `es.ua.jtech.spring.mvc.CocheController` para que admita un nuevo parámetro HTTP "fecha", opcional. Si fecha es null, listará todos los coches. Si no lo es, listará solo los matriculados con posterioridad a esa fecha. Tendrás que:

1. Añadirle un parámetro al método `listar` y vincularlo al parámetro HTTP "fecha". Recuerda que esto se hacía con "@RequestParam". El parámetro HTTP debe ser opcional.
2. Anotar convenientemente el parámetro para que se haga la conversión de String a Date. El formato de entrada será de dos dígitos para día,mes y año separados por barras ("/"). Al ser un formato personalizado tendrás que usar un "pattern"
3. Modificar el código del método para que si la fecha es null use el "listar()" del DAO y en caso contrario llame al "listarPosterioresA(fecha)".
4. Conprueba que todo funciona pasando el parámetro "manualmente" en la URL: "http://localhost:8080/AlquilerCoches/listar.do?**fecha=01/01/09**"

## 10.2. Validación (1.5 puntos)

Modifica la aplicación para validar los datos del coche cuando se dé de alta o se edite. Tendrás que:

- Poner las anotaciones JSR303 pertinentes en los campos de la clase `org.especialistajee.spring.modelo.Coche`. Debes validar que la matrícula tiene 7 caracteres de largo, que ni el kilometraje del coche ni el precio pueden ser números negativos, y que la fecha de matriculación del coche debe ser ya pasada.
- Para que Spring convierta la fecha que escribas en el formulario, que será un String, a



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

un Date, debes anotar el campo "fechaMatriculacion" con `DateTimeFormat`. Usa el mismo formato con el que aparecen los coches en el listado (dos dígitos para día, mes y año y separados por barras).

- Modifica la cabecera de los métodos `procesarCrear` y `procesarEditar` de la clase `org.especialistajee.spring.mvc.CocheController` para asegurar que Spring valida los objetos Coche que se le pasan.
- Modifica el fichero `mensajes.properties` de "src/main/resources", añadiendo las claves necesarias para que se muestren los mensajes de error de validación. Puedes probar a dar de alta algún coche con datos no válidos para comprobar que se vuelve al mismo formulario. Como el log está configurado para ello, verás aparecer los errores de validación en la consola de Tomcat de Eclipse. Pero por supuesto no en el JSP, cosa que vamos a solucionar ahora. Cuando pruebes el funcionamiento, ten en cuenta que estás controlando los errores de validación, pero para los de conversión de tipos tendrías que crear los mensajes del tipo "typeMismatch" en el .properties, como hicimos en la sesión 3.
- Introduce en los JSP "crear.jsp" y "editar.jsp" (están en src/main/webapp/WEB-INF/views) *tags* de Spring para mostrar los errores al usuario. Al lado de cada campo debes meter una etiqueta form:errors, por ejemplo para la matrícula:

```
<form:errors path="matricula"/>
```

## 10.3. Internacionalización (1 punto)

Prepara la aplicación para que se vea en inglés y español. El localeChanger ya está configurado en el XML de Spring. Guardará el locale en una cookie del navegador. Tendrás que:

- Añadir un nuevo *locale* para inglés al `mensajes.properties`. Por tanto, tendrás que cambiar el nombre actual por `mensajes_es.properties` y crear una copia del archivo como `mensajes_en.properties`. Edita esta última y traduce los mensajes.
- Modificar los JSP para que muestren los textos con la etiqueta `message` de Spring en lugar de estar escritos directamente en el código. Traduce al menos la página del listado de coches al inglés y al español, y añade un enlace en dicha página para poder cambiar el idioma. El enlace debe apuntar a "listar.do?locale=en_us" para inglés de USA y lo mismo con "locale=es_es" para español. Si no pones el país en el locale, hay cosas que serán ambiguas, como la moneda.
- Formatea el kilometraje del coche, la fecha de matriculación y la moneda para que se adapte al idioma que se está usando. Para la fecha usa un formato corto, sin hora. Evidentemente el valor de la moneda no será correcto, ya que además de cambiar el símbolo habría que aplicar un tipo de cambio, cálculo que podríamos introducir en el "getter". Pero no te preocupes por eso, el ejercicio solo pretende probar el soporte de internacionalización de Spring. Recuerda que el formateo de Spring solo funciona con los campos de formulario, así que para probarlo tendrás que editar o dar de alta un



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

coche.

**Posibles problemas con la codificación de caracteres**

Cuidado con Google Chrome, algunas versiones pueden dar problemas con el símbolo de la moneda. Por otro lado fíjate en que la aplicación usa un filtro para procesar correctamente los caracteres UTF8. Si no lo usáramos se produciría un error al intentar leer caracteres como el del euro. Este filtro está ya implementado en Spring y se llama `characterEncodingFilter`, puedes ver su definición en el web.xml. No es necesario que modifiques nada.



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

# 11. Acceso remoto. Pruebas

## 11.1. Acceso remoto

Uno de los puntos que hacen atractivos a los EJB es que permiten simplificar la programación distribuida, proporcionando un acceso a objetos remotos "casi transparente" para el programador. Se puede localizar de manera sencilla un objeto remoto que reside en otra máquina cualquiera y llamar a sus métodos como si el objeto estuviera en la máquina local. En esta sección veremos alternativas para conseguir el mismo objetivo, que si bien no son tan sofisticadas, son más ligeras que la implementación de muchos contenedores de EJBs.

### 11.1.1. Evaluación de las alternativas

Spring no proporciona una única alternativa a EJBs para acceso remoto. Según los requerimientos de la aplicación y las características de la implementación será más apropiada una u otra alternativa. Veamos cuáles son:

- **RMI:** A pesar de que usar RMI directamente pueda parecer algo "primitivo", Spring implementa una serie de clases que proporcionan una capa de abstracción sobre el RMI "puro", de modo que por ejemplo no hay que gestionar directamente el servidor de nombres, ni ejecutar manualmente `rmic` y el cliente puede abstraerse totalmente de que el servicio es remoto. RMI será la alternativa adecuada cuando nos interese **buen rendimiento, clientes Java y sepamos que el servidor de nombres no es un problema (p.ej. con firewalls)**
- **HTTP invoker:** Es una opción muy similar a la de RMI, usando serialización de Java, pero a través del puerto HTTP estándar, con lo que eliminamos los posibles problemas de firewalls. Nótese que el cliente también debe ser Spring, ya que el protocolo está implementado en librerías propias del framework (un cliente RMI podría ser no-Spring). Será apropiado cuando nos interese **buen rendimiento, clientes Spring y tengamos posibles problemas con los puertos permitidos.** .
- **Protocolos Hessian y Burlap:** son protocolos que funcionan a través del puerto HTTP estándar. Hessian es binario y Burlap XML, por lo que el primero es más eficiente. Teóricamente pueden funcionar con clientes no-Java, aunque con ciertas limitaciones. No son protocolos originalmente diseñados en el seno de Spring, sino de una empresa llamada Caucho (aunque son también *open source*). Será interesante cuando queramos **buen rendimiento, clientes no-Java y tengamos posibles problemas con los puertos permitidos.**
- **Servicios web SOAP:** indicados para el acceso a componentes remotos en plataformas heterogéneas (con clientes escritos en casi cualquier lenguaje). Su punto débil es básicamente el rendimiento: la necesidad de transformar los mensajes que intercambian cliente servidor a formato neutro en XML hace que sean poco eficientes, pero es lo que al mismo tiempo los hace portables. Serán apropiados cuando **El**



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

**rendimiento no sea crítico, y queramos la máxima portabilidad en cuanto a clientes.**

- **Servicios web REST:** Superan la "pesadez" de sus hermanos SOAP gracias al uso de protocolos ligeros como HTTP. Como contrapartida, al carecer en general de una especificación formal tendremos que programar manualmente el servicio, tanto en la parte cliente como servidor. Serán los indicados si queremos **máxima portabilidad en cuanto a clientes y el API del servicio es simple y fácil de programar manualmente.**

**Limitaciones del acceso remoto en Spring**

Nótese que en el acceso remoto a componentes los EJBs siguen teniendo ciertas ventajas sobre Spring, en particular la propagación remota de transacciones, que no es posible en Spring. Es el precio a pagar por poder usar un contenedor web java convencional como Tomcat en lugar de un servidor de aplicaciones completo.

Discutiremos a continuación con más detalle las características de estas alternativas y cómo usarlas y configurarlas dentro de Spring. Obviaremos los servicios web SOAP, que por su complejidad quedan fuera del ámbito posible para esta sesión.

En todos los casos vamos a usar el siguiente ejemplo, muy sencillo, de componente al que deseamos acceder de forma remota, con su interfaz:

```
package servicios;

public interface ServicioSaludo {
   public String getSaludo();
}
```

```
package servicios;

@Service("saludador")
public class ServicioSaludoImpl implements ServicioSaludo {
   String[] saludos = {"hola, ¿qué tal?", "me alegra verte", "yeeeeeey"};

   public String getSaludo() {
         int pos = (int)(Math.random() * saludos.length);
         return saludos[pos];
   }
}
```

### 11.1.2. RMI en Spring

Aunque el uso directo de RMI puede resultar tedioso, Spring ofrece una capa de abstracción sobre el RMI "puro" que permite acceder de forma sencilla y casi transparente a objetos remotos.

Usando la clase `RmiServiceExporter` podemos exponer la interfaz de nuestro servicio como un objeto RMI. Se puede acceder desde el cliente usando RMI "puro" o bien, de modo más sencillo con un `RmiProxyFactoryBean`.



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

La configuración en el lado del servidor quedará como sigue:

```
<bean class="org.springframework.remoting.rmi.RmiServiceExporter">
    <!-- el nombre del servicio es arbitrario.
    Sirve para referenciarlo desde el cliente -->
    <property name="serviceName" value="miSaludador"/>
    <!-- el servicio es el bean que hacemos accesible -->
    <property name="service" ref="saludador"/>
    <!-- el bean debe implementar un interface -->
    <property name="serviceInterface" value="servicios.ServicioSaludo"/>
    <!-- Cómo cambiar el puerto para RMI (por defecto es el 1099) -->
    <property name="registryPort" value="1199"/>
</bean>
```

En la configuración anterior se ha cambiado el puerto del servidor de nombres RMI para evitar posibles conflictos con el del servidor de aplicaciones. A partir de este momento, el objeto remoto es accesible a través de la URL `rmi://localhost:1199/miSaludador`.

La configuración en el cliente quedaría como sigue:

```
<bean id="saludadorRMI"
        class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
    <property name="serviceUrl" value="rmi://localhost:1199/miSaludador"/>
    <property name="serviceInterface" value="servicios.ServicioSaludo"/>
</bean>
```

Nótese que el id del bean es arbitrario, aunque luego habrá que referenciarlo en el código del cliente. Los valores de las propiedades `serviceUrl` y `serviceInterface` evidentemente tienen que coincidir con los dados en la configuración en el servidor.

Para acceder al objeto de forma remota todo lo que necesitaríamos es tener en el cliente el código del interfaz `ServicioSaludo` y usar el siguiente código Java para conectar:

```
ClassPathXmlApplicationContext contexto =
         new ClassPathXmlApplicationContext("clienteRMI.xml");
ServicioSaludo ss =  contexto.getBean(ServicioSaludo.class);
System.out.println(ss.getSaludo());
```

Suponiendo que el fichero de configuración en el cliente lo hemos llamado `clienteRMI.xml`

### 11.1.3. Hessian y Burlap

Hessian y Burlap son dos protocolos diseñados originalmente por la empresa Caucho, desarrolladora de un servidor de aplicaciones J2EE de código abierto llamado `Resin`. Ambos son protocolos para acceso a servicios remotos usando conexiones HTTP estándar. La diferencia básica entre ambos es que Hessian es binario (y por tanto más eficiente que Burlap) y este es XML (y por tanto las comunicaciones son más sencillas de depurar). Para ambos también se han desarrollado implementaciones en distintos lenguajes de manera que el cliente de nuestra aplicación podría estar escrito en C++,



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

Python, C#, PHP u otros.

#### 11.1.3.1. Uso de los protocolos

Usaremos Hessian en el siguiente ejemplo, aunque la configuración de Burlap es prácticamente idéntica. Hessian se comunica mediante HTTP con un servlet . Por tanto el primer paso será crear dicho servlet en nuestro `web.xml`. Nos apoyaremos en la clase `DispatcherServlet` propia de Spring, ya que se integra de manera automática con el resto de elementos de nuestra configuración. A continuación se muestra el fragmento significativo del `web.xml`.

```
<servlet>
   <servlet-name>remoting</servlet-name>
   <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
   </servlet-class>
   <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
   <servlet-name>remoting</servlet-name>
   <url-pattern>/remoting/*</url-pattern>
</servlet-mapping>
```

Esto hace accesible el servlet a través de la URL `http://localhost:8080/remoting` (si por ejemplo usamos Tomcat cuyo puerto por defecto es el 8080). Recordemos del tema de MVC que en Spring, la configuración de un `DispatcherServlet` se debe guardar en un xml con nombre *nombreDelServlet*-servlet.xml (en nuestro caso `remoting-servlet.xml`). Aclarar que aunque podríamos hacer uso del mismo DispatcherServlet para gestionar tanto los controllers MVC como el acceso remoto, no es necesario, podemos usar una instancia distinta para cada cosa.

```
<bean name="/saludador"
class="org.springframework.remoting.caucho.HessianServiceExporter">
  <property name="service" ref="saludador"/>
  <property name="serviceInterface" value="servicios.ServicioSaludo"/>
</bean>
```

Aquí hacemos uso de la clase `HessianServiceExporter`, que nos permite exportar de forma sencilla un servicio Hessian. En nuestro caso estará accesible en la URL `http://localhost:8080/contexto-web/remoting/saludador`. Nos falta la configuración del cliente y el código que llama al servicio:

```
<bean id="miSaludador"
class="org.springframework.remoting.caucho.HessianProxyFactoryBean">
        <property name="serviceUrl"
                value="http://localhost:8080/contexto-web/remoting/saludador"/>
        <property name="serviceInterface"
value="servicios.ServicioSaludo"/>
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
</bean>
```

Suponiendo que el código XML anterior se guarda en un fichero llamado `clienteHessian.xml`

```
ClassPathXmlApplicationContext contexto =
        new ClassPathXmlApplicationContext("clienteHessian.xml");
ServicioSaludo ss =  (ServicioSaludo) contexto.getBean("miSaludador");
System.out.println(ss.getSaludo());
```

En el ejemplo anterior bastaría con escribir `Burlap` allí donde aparece `Hessian` y todo debería funcionar igual, pero ahora usando este procolo basado en mensajes XML.

#### 11.1.3.2. Autentificación HTTP con Hessian y Burlap

Al ser una conexión HTTP estándar podemos usar autentificación BASIC. De este modo podemos usar la seguridad declarativa del contenedor web también para controlar el acceso a componentes remotos. En la configuración del servidor podríamos añadir el siguiente código:

```
<!-- cuidado con el copiar/pegar,
el nombre de la clase está partido -->
<bean class="org.springframework.web.servlet.
        handler.BeanNameUrlHandlerMapping">
   <property name="interceptors">
      <list>
         <ref bean="authorizationInterceptor"/>
      </list>
   </property>
</bean>

<!-- cuidado con el copiar/pegar,
el nombre de la clase está partido -->
<bean id="authorizationInterceptor"
   class="org.springframework.web.servlet.
   handler.UserRoleAuthorizationInterceptor">
   <property name="authorizedRoles">
    <list>
      <value>admin</value>
      <value>subadmin</value>
    </list>
   </property>
</bean>
```

Usando AOP añadimos un interceptor que resulta ser de la clase `UserRoleAuthorizationInterceptor`. Dicho interceptor solo permite el acceso al bean si el usuario resulta estar en uno de los roles especificados en la propiedad `authorizedRoles`. El `BeanNameUrlHandlerMapping` es el objeto que "tras las bambalinas" se encarga de asociar los beans que comienzan con "/" con los servicios en la URL del mismo nombre (en nuestro caso el bean "/saludador").

### 11.1.4. HTTP invoker



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

Esta es una implementación propia de Spring, que utiliza la serialización estándar de Java para transmitir objetos a través de una conexión HTTP estándar. Será la opción a elegir cuando los objetos sean demasiado complejos para que funcionen los mecanismos de serialización de Hessian y Burlap.

La configuración es muy similar al apartado anterior, podemos usar el mismo `DispatcherServlet` pero ahora para el acceso al servicio se debe emplear la clase `HttpInvokerServiceExporter` en lugar de `HessianServiceExporter`

```
<bean name="/saludadorHTTP"
class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
  <property name="service" ref="saludador"/>
  <property name="serviceInterface" value="servicios.ServicioSaludo"/>
</bean>
```

En la parte del cliente la definición del bean sería:

```
<bean id="httpProxy"
class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
  <property name="serviceUrl"
value="http://localhost:8080/MiAplicacion/remoting/saludadorHTTP"/>
  <property name="serviceInterface" value="servicios.ServicioSaludo"/>
</bean>
```

Por defecto en el cliente se usan las clases estándar de J2SE para abrir la conexión HTTP. Además Spring proporciona soporte para el `HttpClient` de Jakarta Commons. Bastaría con poner una propiedad adicional en el `HttpInvokerProxyFactoryBean`:

```
<bean id="httpProxy"
class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
  ...
  <property name="httpInvokerRequestExecutor">
    <!-- cuidado con el copiar/pegar,
    el nombre de la clase está partido -->
    <bean class="org.springframework.remoting.httpinvoker.
             CommonsHttpInvokerRequestExecutor"/>
  </property>
  ...
</bean>
```

## 11.2. Pruebas

En cualquier metodología moderna de ingeniería de software las pruebas son un elemento fundamental. Reconociendo este hecho, Spring nos da soporte para poder implementar nuestras pruebas del modo más sencillo posible. Otro aspecto importante, sobre todo en prubeas de integración es trabajar en un entorno lo más parecido posible a cómo se ejecutará el código en producción. En Spring podemos hacer que las dependencias entre



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

objetos se satisfagan automáticamente en pruebas del mismo modo que en producción.

Vamos a usar aquí el soporte para JUnit 4 y superior, que es la recomendada en la versión actual de Spring. El soporte de Junit 3 está *deprecated* y se eliminará en futuras versiones.

### 11.2.1. Pruebas unitarias

Supongamos que en la capa de acceso a datos de nuestra aplicación tenemos un DAO que implementa este interfaz:

```
public interface IUsuariosDAO {
    public List<Usuario> listar();
    public Usuario getUsuario(String login);
    public void alta(Usuario u);
    ...
}
```

y que queremos escribir código de prueba para él. Lo primero que nos da Spring es la posibilidad de usar un fichero de configuración de beans para pruebas diferente al fichero de producción. El esqueleto de la clase que prueba el DAO podría ser algo como:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:config/daos-test.xml")
public class UsuariosDAOTest  {
  //Spring nos da la instancia del DAO a probar
  @Autowired
  IUsuariosDAO dao;

  //Esto ya no tiene nada de particular de Spring
  @Test
  public void testListar() {
    List<Usuario> lista = dao.listar();
    assertEquals(10, lista.size());
  }
}
```

La anotación `@RunWith` es la que nos "activa" el soporte de testing de Spring. Con `@ContextConfiguration` especificamos el fichero o ficheros de configuración de beans que vamos a usar para las pruebas. Aquí podemos configurar lo que queremos que sea distinto de producción, por ejemplo, que los dataSources conecten con bases de datos de prueba, que los objetos de los que dependemos sean *mocks*, etc.

**Profiles**

A partir de la versión 3.1 de Spring hay una forma alternativa de hacer esto usando lo que se denominan *profiles*. La idea es que podemos hacer que ciertos beans se definan dentro de un determinado perfil (p.ej. "test") y ciertos otros en uno distinto (p.ej. "produccion"). Usando diferentes mecanismos podemos especificar qué perfil queremos usar en un momento dado. Se recomienda consultar la documentación de Spring para más información.

Nótese que la referencia al DAO que queremos probar nos la da el propio Spring, lo que es una forma cómoda de instanciar el objeto, y además ofrece la ventaja adicional de que



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

si este dependiera de otros dichas dependencias se resolverían automáticamente, lo que nos facilita las pruebas de integración, como veremos después.

¿Qué tendría de especial nuestro fichero de configuración para pruebas?. Como ya hemos dicho, lo más habitual es definir la conexión con la base de datos de prueba en lugar de con la real. Yendo un paso más allá y para agilizar las pruebas, podríamos usar una base de datos embebida en lugar de usar simplemente otras tablas en el mismo servidor de bases de datos de producción:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:jee="http://www.springframework.org/schema/jee"
  xmlns:jdbc="http://www.springframework.org/schema/jdbc"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.2.xsd
    http://www.springframework.org/schema/jee
    http://www.springframework.org/schema/jee/spring-jee-3.2.xsd
    http://www.springframework.org/schema/jdbc
    http://www.springframework.org/schema/jdbc/spring-jdbc-3.2.xsd">

    <!-- conexión con la BD de pruebas -->
    <jdbc:embedded-database id="miDataSource">
        <jdbc:script location="classpath:db.sql"/>
        <jdbc:script location="classpath:testdata.sql"/>
    </jdbc:embedded-database>

    <!-- los DAOs están en este package -->
    <context:component-scan base-package="es.ua.jtech.spring.datos"/>

    <!-- soporte de transacciones en las pruebas -->
    <bean id="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
      <!-- importante: este "ref" coincide con el "id" del dataSource -->
      <property name="dataSource" ref="miDataSource"/>
    </bean>
</beans>
```

En el fichero anterior estaríamos definiendo un DataSource Spring que conectaría con una base de datos embebida, cuyos datos estarían en los dos scripts SQL a los que se hace referencia (supuestamente uno para crear las tablas y otro para insertar los datos). Spring usa HSQLDB como base de datos embebida por defecto, aunque podemos cambiar la configuración para usar otras. Por supuesto necesitaremos incluir la dependencia correspondiente en el pom.xml (suponiendo HSQLDB):

```
<dependency>
  <groupId>org.hsqldb</groupId>
  <artifactId>hsqldb</artifactId>
  <version>2.0.0</version>
  <scope>test</scope>
</dependency>
```

Por otro lado, vemos que en el fichero se define un "transaction manager". Este se usa



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

como soporte para gestionar la transaccionalidad de las pruebas. Un problema muy típico de las pruebas de acceso a datos es que la ejecución de una prueba altera el contenido de la base de datos y quizás interfiere con las pruebas siguientes. Esto nos obliga a restablecer los datos iniciales tras cada prueba, por ejemplo ejecutando el script SQL que inserta los datos. Una alternativa es el **soporte de transaccionalidad** que nos da Spring: podemos hacer un *rollback* automático de todos los cambios generados durante la prueba, lo que facilita mucho el trabajo.

Para que funcione esta transaccionalidad de las pruebas necesitamos un transaction manager en el fichero de configuración de pruebas al igual que es necesario para que funcione la transaccionalidad declarativa en producción. Una vez definido el gestor de transacciones, debemos anotar la clase de pruebas con `@TransactionConfiguration`. El valor del atributo "transactionManager" debe coincidir con el id del transaction manager definido en el fichero de configuración XML. Finalmente anotaremos con `@Transactional` los métodos de prueba para los que queramos hacer un *rollback* automático, o bien anotaremos la propia clase si queremos hacerlo en todos:

```
//modificación del test anterior
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:config/daos-test.xml")
@TransactionConfiguration(transactionManager = "txManager",
                          defaultRollback = true)
@Transactional
public class UsuariosDAOTest  {
...
```

### 11.2.2. Pruebas de integración. Uso de objetos mock

Supongamos que tenemos una capa de negocio en la que hay un "gestor de usuarios" con el siguiente interfaz:

```
public interface IUsuariosBO {
  //Este método debe comprobar que el password coincide con lo que
devuelve el DAO
  public Usuario login(String login, String password);
  //Estos métodos delegan el trabajo en el DAO
  public List<Usuario> listar();
  public void alta(Usuario u);
}
```

Las implementaciones de dicho interfaz hacen uso del DAO de usuarios:

```
@Service
public class UsuariosBOSimple implements IUsuariosBO {
  @Autowired
  IUsuariosDAO udao;

  @Override
  public Usuario login(String login, String password) {
    Usuario u = udao.getUsuario(login);
    if (u!=null  && u.getPassword().equals(password))
      return u;
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
    else
      return null;
  }
}
```

Además de probar por separado el BO y el DAO nos va a interesar hacer pruebas de integración de ambos. En Spring si hacemos test de la capa BO automáticamente estamos haciendo pruebas de integración ya que el framework resuelve e instancia la dependencia BO->DAO.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={"classpath:config/daos-test.xml",
   "classpath:config/bos-test.xml"})
public class UsuariosBOTest {
  @Autowired
  IUsuariosBO ubo;

  @Test
  public void testLogin() {
    //el usuario "experto" sí está en la BD
    assertNotNull(ubo.login("experto", "experto"));
    //pero no el usuario "dotnet" (¡jamás!)
    assertNull(ubo.login("dotnet", "dotnet"));
  }
}
```

Como antes, simplemente le pedimos a Spring con @Autowired que nos inyecte el objeto que deseamos probar. No necesitamos instanciar el DAO, lo hará Spring. La única diferencia con el ejemplo anterior es que usamos un fichero de configuración adicional para la capa BO (en un momento veremos por qué nos podría interesar tener la capa DAO y BO en ficheros de configuración distintos). Nótese que cuando se usan varios, hay que hacerlo con notación de inicialización de array (entre llaves), y explicitando el nombre del atributo: "location".

Pero ¿qué ocurre si queremos hacer pruebas unitarias de la capa BO, sin que "interfiera" el comportamiento de los DAO?. Necesitamos entonces sustituir estos últimos por **objetos *mock***. Podríamos escribirlos nosotros mismos, pero en Java hay varios *frameworks* que nos ayudarán a generar estos objetos. Vamos a usar aquí mockito. El objetivo no es ver aquí cómo se usa este *framework* en sí, sino ver una breve introducción a cómo usarlo dentro de Spring.

Supongamos que queremos construir un *mock* del interfaz `IUsuariosDAO` y hacer que tenga un determinado comportamiento. El API básico de mockito es bastante simple:

```
import static org.mockito.Mockito.*;
//creamos el mock
IUsuariosDAO udaoMock = mock(IUsuariosDAO.class);
//Creamos un usuario de prueba con login "hola" y password "mockito"
Usuario uTest = new Usuario("hola", "mockito");
//grabamos el comportamiento del mock
when(udaoMock.getUsuario("hola")).thenReturn(uTest);
//imprime el usuario de prueba
System.out.println(udaoMock.getUsuario("hola"));
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

Como se ve en el código anterior, primero creamos el *mock* con la llamada al método estático `mock` y luego especificamos para la entrada indicada (`when`) qué salida debe proporcionar (`thenReturn`). Nuestro *mock* será una especie de "tabla" con respuestas predefinidas ante determinadas entradas.

Ahora para integrar el mock dentro de Spring tenemos que resolver dos problemas: el primero es cómo sustituir nuestra implementación de IUsuariosDAO por el mock. Definiremos un bean de Spring que se construya llamando al método `mock` de mockito. Vamos a hacerlo en XML, aunque también lo podríamos hacer con configuración Java (pero no con la anotación @Repository, ya que la clase a anotar no es nuestra, sino generada por mockito):

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:jee="http://www.springframework.org/schema/jee"
  xmlns:jdbc="http://www.springframework.org/schema/jdbc"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.2.xsd
    http://www.springframework.org/schema/jee
    http://www.springframework.org/schema/jee/spring-jee-3.2.xsd
    http://www.springframework.org/schema/jdbc
    http://www.springframework.org/schema/jdbc/spring-jdbc-3.2.xsd">

  <bean id="usuarioDAO" class="org.mockito.Mockito" factory-method="mock">
    <constructor-arg value="es.ua.jtech.dao.IUsuariosDAO" />
  </bean>
</beans>
```

Ahora podemos ver por qué nos interesaba tener en ficheros de configuración distintos la capa DAO y BO. De este modo podemos tener un XML para la capa DAO de "producción" (que busque anotaciones @Repository digamos en es.ua.jtech.dao) y de pruebas de integración y este otro para pruebas con mocks.

El segundo problema a resolver es cómo especificar el comportamiento del *mock*. Aprovecharemos el `@Autowired` de Spring para acceder al objeto y el `@Before` para especificar su comportamiento. Si el bean es un *singleton*, el BO tendrá acceso a la misma instancia que hemos controlado nosotros y por tanto su comportamiento será el deseado.

```
@RunWith(SpringJUnit4ClassRunner.class)
//daos-mock-test.xml es el XML de configuración anterior,
//con la definición del mock de IUsuariosDAO
@ContextConfiguration(locations={"classpath:config/daos-mock-test.xml",
   "classpath:config/bos-test.xml"})
public class UsuariosBOMockTest {
  @Autowired
  IUsuariosBO ubo;

  //Esto nos dará acceso al mock
  @Autowired
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
  IUsuariosDAO udao;

  @Before
  public void setup() {
    when(udao.getUsuario("test")).thenReturn(new Usuario("test","test"));
    when(udao.getUsuario("test2")).thenReturn(new
Usuario("test2","test2"));
  }

  @Test
  public void testLogin() {
    //este usuario está "grabado" en el mock
    assertNotNull(ubo.login("test", "test"));
    //pero este no, por tanto el BO debería devolver null
    assertNull(ubo.login("experto", "experto"));
  }
}
```

### 11.2.3. Pruebas de la capa web

Una de las novedades de la versión 3.2 de Spring (la última en el momento de escribir estos apuntes) es la extensión del soporte de pruebas a la capa web. Sin necesidad de desplegar la aplicación en un contenedor web podemos simular peticiones HTTP y comprobar que lo que devuelve el controller, la vista a la que se salta o los objetos que se añaden al modelo son los esperados.

Hay dos posibilidades para ejecutar pruebas en la capa web, una de ellas es usar la misma configuración que se usa en producción y otra es establecer manualmente una configuración simplificada. Vamos a ver aquí la primera de ellas, ya que nos permite hacer pruebas más completas y más próximas a como se ejecuta el código dentro del contenedor web.

Veamos primero un ejemplo muy sencillo. Supongamos que tenemos el siguiente controller, que lo único que hace es devolver un mensaje de texto:

```
@Controller
public class HolaSpringController {
  @RequestMapping("/hola")
  public @ResponseBody String hola() {
    return "Hola Spring";
  }
}
```

Para las pruebas necesitamos una instancia de `MockMVC`, que se construye a partir del `WebApplicationContext` (el contexto de aplicación generado a partir del fichero de configuración de la capa web). Podemos pedirle a Spring una instancia de este último con `@Autowired`, así que no es excesivamente complicado construir el `MockMVC`:

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration(locations={"classpath:config/web-test.xml",
   "classpath:config/daos-test.xml", "classpath:config/bos-test.xml"})
public class HolaSpringControllerTest {
  @Autowired
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
  private WebApplicationContext wac;

  private MockMvc mockMvc;

  @Before
  public void setup() {
        this.mockMvc =
MockMvcBuilders.webAppContextSetup(this.wac).build();
  }

  //ahora vienen los test, un poco de paciencia...
  ...
}
```

Nótese que además necesitamos anotar la clase del test con `@WebAppConfiguration`, para indicarle a Spring que estamos usando un contexto de aplicación web.

El fichero de configuración de la capa web para testing (en el ejemplo, web-test.xml) puede ser el mismo de producción o uno simplificado. Como mínimo, si usamos anotaciones, deberá tener el context:component-scan. Además también debería tener un viewresolver, así podremos comprobar el nombre físico de la página a la que saltamos tras el controller.

El API para hacer las llamadas de prueba es bastante intuitivo. Por desgracia al ser una funcionalidad muy recientemente añadida a Spring (antes estaba en un proyecto aparte) la documentación de referencia todavía no es muy detallada y para algunas cosas hay que acudir al propio Javadoc del API. Vamos a ver aquí algunos ejemplos sencillos. Por ejemplo, para comprobar que la respuesta da un código de estado OK y que es la que deseamos, haríamos

```
import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;
...
...
@Test
public void testHola() throws Exception  {
  this.mockMvc.perform(get("/hola"))
    .andExpect(status().isOk())
    .andExpect(content().string("Hola Spring"));
}
```

como se ve, el esquema básico es que se hace una petición con `perform` y luego se espera una serie de resultados.

Vamos a ver varios ejemplos de cómo hacer peticiones y comprobar el resultado de la ejecución del controller. Para más información, se recomienda consultar la sección correspondiente de la documentación de Spring

#### 11.2.3.1. Simular la petición

Podemos especificar el método HTTP a usar y el tipo de datos que aceptamos (el valor de



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

la cabecera Accept):

```
mockMvc.perform(get("/usuarios/").accept(MediaType.APPLICATION_JSON));
```

Podemos usar URI templates como en REST

```
this.mockMvc.perform(put("/usuarios/{id}", 42)
    .content("{'login':'experto', 'password':'experto'}"));
```

O añadir parámetros HTTP a la petición

```
mockMvc.perform(get("/verUsuario").param("login", "experto"));
```

#### 11.2.3.2. Comprobar el resultado

Ya hemos visto cómo comprobar el código de estado y el contenido de la respuesta. Veamos otros ejemplos. Podemos comprobar la vista a la que se intenta saltar:

```
this.mockMvc.perform(post("/login").param("login","experto").param("password",
"123456"))
  .andExpect(view().name("home"));
```

También podemos verificar que el modelo generado por el controller es correcto. Por ejemplo, supongamos que el controller debería haber añadido al modelo un atributo "usuario" con el usuario a ver, cuyos datos se muestran a través de la vista "datos_usuario"

```
this.mockMvc.perform(get("/usuarios/experto"))
   .andExpect(model().size(1))
   .andExpect(model().attributeExists("usuario"))
   .andExpect(view().name("datos_usuario"));
```

Podemos verificar una parte del contenido de la respuesta. Si es JSON, podemos usar la sintaxis de JsonPath para especificar la parte que nos interesa. Este ejemplo buscaría una propiedad llamada "localidad" con valor "alicante".

```
this.mockMvc.perform(get("/usuarios/experto").accept("application/json;charset=UTF-8"))
  .andExpect(status().isOk())
  .andExpect(content().contentType("application/json"))
  .andExpect(jsonPath("$.localidad").value("Alicante");
```

En caso de que la respuesta sea XML podemos usar el estándar xpath.

Podemos comprobar otros aspectos del resultado. El framework nos ofrece distintos `ResultMatchers`, que nos sirven para esto. Se recomienda consultar directamente el JavaDoc del API de Spring, en concreto el package `org.springframework.test.web.servlet.result`



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

# 12. Ejercicios de acceso remoto y pruebas

## 12.1. Acceso remoto con HttpInvoker (1 punto)

Vamos a proporcionar acceso remoto a la aplicación de alquiler de coches de la sesión anterior. Usaremos el HttpInvoker, ya que es razonablemente eficiente y no presentará problemas con *firewalls*.

Configuración de la parte del servidor:

1. Fíjate que en el `web.xml` definimos un nuevo servlet de la clase `DispatcherServlet` al que llamamos `remoting` y lo mapeamos con las URL del tipo `/remoting/*` (no tienes que hacer nada, ya está definido). Aunque ya teníamos otro `DispatcherServlet` definido, se encargaba de la parte MVC y no es recomendable que un solo servlet se encargue de las dos cosas.
2. Modifica el `src/main/webapp/WEB-INF/config/remoting-servlet.xml` para añadir la configuración de la parte del servidor. Adapta la de los apuntes, para dar acceso remoto al interface `ICocheBO`. Cuidado con el atributo "ref": es el nombre del bean CocheBO. Como la anotación @Service en esta clase no lo especifica, el nombre por defecto será el mismo que el de la clase con la inicial en minúscula: cocheBO, como vimos en la primera sesión. Una vez creado, puedes comprobar que está inicializado intentando acceder a su URL con el navegador. Será "remoting/" seguido del "name" que le hayas puesto al bean. En realidad dará un error HTTP 500, ya que al HttpInvoker no se le puede llamar así, pero al menos debería mostrar una excepción de tipo EOFException, y sabremos "que está ahí". Si da otro tipo de excepción o un HTTP 404 es que hay algo mal configurado.

Parte del cliente:

1. Crea un proyecto de tipo "Maven project". Elige el arquetipo que te saldrá por defecto: "Maven-archetype-quickstart". Como groupId pon `es.ua.jtech` y como ArtifactId `ClienteRemotoCoches`.
2. Cambiar el pom.xml generado por Eclipse por el siguiente (tras hacer esto tendrás que ejecutar la opción Maven > Update project para que Eclipse tenga en cuenta los cambios)

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
   http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>es.ua.jtech</groupId>
    <artifactId>ClienteRemotoCoches</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>ClienteRemotoCoches</name>
    <url>http://maven.apache.org</url>
    <dependencies>
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.7</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>3.2.0.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-web</artifactId>
            <version>3.2.0.RELEASE</version>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.6</source>
                    <target>1.6</target>
                </configuration>
            </plugin>
        </plugins>
        <finalName>ClienteRemotoCoches</finalName>
    </build>
</project>
```

3. Vamos a acceder al BO remotamente, y este nos devolverá objetos `Coche`. Por lo tanto, en el nuevo proyecto necesitarás copiar `es.ua.jtech.spring.negocio.ICocheBO` y `es.ua.jtech.spring.modelo.Coche` del proyecto original. Tendrás que crear los packages correspondientes. Cuidado, en la clase Coche que copies en el nuevo proyecto debes borrar las anotaciones de validación para no introducir dependencias del API JSR303 en el cliente. Atento: no necesitas la clase `CocheBO`, solo el interface.
4. Crea una nueva "Source Folder" (File > New > source folder) dándole como nombre "src/main/resources".
5. Crea un fichero de configuración de beans XML de Spring llamado "cliente.xml" en la carpeta "resources" que acabas de crear (File > New > Spring bean configuration file). Solo necesitarás el espacio de nombres "beans". Pon en él la configuración de la parte del cliente, fijándote en la que aparece en los apuntes y adaptándola a lo que necesitas.
6. En el método `main` de la clase `App` del proyecto, escribe código que obtenga un ICocheBO (necesitarás un ClasspathApplicationContext), llame al método obtener(String matricula) y muestre los datos de uno de los coches por pantalla con `System.out.println`.

## 12.2. Pruebas de la capa DAO (0.5 puntos)



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

Vamos a implementar algunas pruebas para la capa DAO. Usaremos una base de datos embebida (HSQLDB) para acelerar las pruebas

1. Lo primero es **incluir las dependencias** necesarias en el pom.xml: spring-test y hsqldb (JUnit ya está incluida en la plantilla)

```
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-test</artifactId>
        <version>3.2.0.RELEASE</version>
        <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>org.hsqldb</groupId>
        <artifactId>hsqldb</artifactId>
        <version>2.0.0</version>
        <scope>test</scope>
    </dependency>
```

2. Ahora crearemos el **fichero de configuración de Spring para los test de la capa DAO** (créalo dentro de src/test/resources/config y llámalo daos-test.xml). Puedes tomar como modelo el de los apuntes y transparencias. En src/test/resources tienes un script para crear la base de datos e insertar los datos. Modifica el XML de los apuntes ya que éste espera que haya dos (uno para crear tablas y otro para insertar datos). *Disclaimer*: siento el trabajo rutinario, es simplemente para que no te limites a copiar y pegar literalmente el fichero entero en modo "piloto automático".
3. **Crea una clase de prueba** CocheDAOJDBCTest en el paquete es.ua.jtech.spring.datos de src/test/java (cuidado, no la metas en src/main/java). Implementa en ella una prueba del método listar del DAO. Por ejemplo puedes comprobar que hay dos coches en los datos de prueba y que la matrícula del primero de ellos es 1111JKG.

## 12.3. Pruebas de la capa BO con y sin objetos mock (1 punto)

1. **Crea una clase de prueba CocheBOTest** en el paquete es.ua.jtech.spring.bo de src/test/java, para probar la integración entre la capa de negocio y datos (es decir, sin usar mock). Implementa en ella alguna prueba que verifique que el listar del BO funciona correctamente.
2. Implementa **pruebas del BO con mocks** de la capa DAO. Tendrás que:
   - Introducir en el pom.xml la dependencia de Mockito:

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-all</artifactId>
    <scope>test</scope>
    <version>1.8.5</version>
</dependency>
```

   - Crear el fichero de configuración daos-mock-test.xml en src/test/resources/config. En este fichero se debe crear un mock de ICocheDAO. Puedes tomar como



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

modelo el que crea el IUsuarioDAO en los apuntes y transparencias.

- Crea una clase de prueba CocheBOMockTest en el paquete es.ua.jtech.spring.bo de src/test/java, para hacer pruebas unitarias del BO usando el mock. Prueba al menos el método listar(). Tendrás que preparar el mock en el @Before para que le devuelva al BO datos de prueba.

## 12.4. Pruebas de la capa web (0.5 puntos)

Como fichero de configuración de testing para la capa web, puedes usar directamente el mismo que se está usando "en producción", el dispatcher-servlet.xml que está en src/main/webapp/WEB-INF/config. Cópialo en src/test/resources.

Implementa pruebas de integración del controller CocheController. Hazla en la clase CocheControllerTest, en el paquete es.ua.jtech.spring.mvc de src/test/java. Prueba al menos el funcionamiento del método listar() del controller. Verifica que existe un atributo en el modelo llamado "listado" y que la vista a la que se salta se llama "listar".



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

# 13. Seguridad

En este tema vamos a introducir *Spring Security*, un proyecto "hijo" de Spring que permite controlar de forma declarativa y totalmente configurable la seguridad de nuestra aplicación. Además, nuestro proyecto será totalmente portable entre servidores, a diferencia de la seguridad declarativa estándar de JavaEE, que no lo es en varios aspectos, por ejemplo, la definición de usuarios y roles.

## 13.1. Conceptos básicos de seguridad

Lo primero que encuentra un usuario que intenta acceder a una aplicación segura es el mecanismo de **autentificación**. Para autentificarse, el usuario necesita un *principal*, que típicamente es un login y unas *credenciales*, normalmente un password. No siempre se usa login y password. El *principal* y las credenciales pueden proceder por ejemplo de un certificado digital o de otros mecanismos.

En Spring Security, el encargado de gestionar la autentificación es el *Authentication manager*. Este depende de uno o varios *authentication providers*, que son los que de manera efectiva obtienen el *principal* y credenciales del usuario. Spring security tiene implementados un gran número de proveedores de autentificación: login con formulario web, login con HTTP BASIC (el navegador muestra una ventana propia para introducir login y password), servidor LDAP, certificados digitales, etc.

La autentificación demuestra que el usuario es quien dice ser, pero queda por ver si tiene permiso de acceso al recurso que ha solicitado. Esto se denomina **control de acceso**. Aquí entra en juego el *Access manager*, que en función de las credenciales, toma la decisión de permitir o no el acceso. Normalmente cada usuario tiene asociado una serie de roles o, como se dice en Spring, de *authorities*, que se asocian a los recursos para permitir o no el acceso.

En una aplicación normalmente solo hay un *access manager*, aunque Spring permite el uso simultáneo de varios, que "por consenso" o "por votación" decidirán si conceden el acceso al recurso

En aplicaciones web sencillas el control de accesos declarativo suele ser una cuestión de "todo o nada" para un determinado rol. Una forma más avanzada de control de accesos es algo muy común en sistemas operativos: las *Access Control Lists* (ACL) que especifican qué operaciones (acceso/modificación/borrado) puede realizar cada usuario sobre cada recurso. Las aplicaciones con requerimientos de seguridad avanzados pueden asignar a cada recurso un ACL que controlará Spring Security, lo que proporciona una enorme flexibilidad de configuración.

Hay otros tipos de gestores de seguridad en Spring Security, como los *run-as managers*, que permiten ejecutar ciertas tareas cambiando temporalmente el rol del usuario (al estilo



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

de su en UNIX) o los *after invocation managers*, que controlan que todo es correcto *después* de acceder al recurso. No obstante, quedan fuera del ámbito de estos apuntes. Aunque la documentación de Spring Security no es tan exhaustiva como la del propio Spring, es bastante aceptable y pueden consultarse en ella todos estos conceptos.

## 13.2. Una configuración mínima para una aplicación web

Antes de realizar la configuración propiamente dicha debemos incluir en el proyecto las dependencias de los módulos de Spring Security que necesitemos. El framework está dividido en tres módulos: *core*, *configuration* y *web*. Los dos primeros son necesarios siempre. El tercero solo si vamos a asegurar una aplicación web. Con Maven basta con incluir los artefactos correspondientes en el pom.xml. Como se ve, en el momento de escribir estas líneas la versión actual es la 3.1.

```
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-core</artifactId>
  <version>3.1.3.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-web</artifactId>
  <version>3.1.3.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-config</artifactId>
  <version>3.1.3.RELEASE</version>
</dependency>
```

Ahora hay que configurar el framework. Vamos a ver aquí una configuración mínima por defecto para una aplicación web, y luego la iremos personalizando. **Lo primero es permitir que Spring Security intercepte las peticiones a nuesttra aplicación** para poder controlar la seguridad. El framework usa filtros de servlets para esto. Por ello lo primero es declararlos en el `web.xml`. Lo más sencillo es usar un filtro de la clase `DelegatingFilterProxy` ya implementada en Security. Esta clase hará de interfaz entre el mecanismo estándar de filtros y los beans que Spring security usa "por debajo" para implementar todas sus funcionalidades. A continuación se muestra el fragmento relevante del `web.xml`:

```
<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>
       org.springframework.web.filter.DelegatingFilterProxy
    </filter-class>
</filter>
<filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

**Cuidado con el nombre del filtro**

El nombre del filtro es significativo, y por defecto **debe** ser `springSecurityFilterChain`, ya que es lo que espera la configuración estándar. Por supuesto esto es modificable con algo de configuración adicional, que se puede consultar en la documentación de Spring security.

**Lo segundo es configurar las opciones del propio Spring Security**. Esta configuración se hace en un fichero de definición de beans de Spring. La configuración más sencilla que podemos hacer es la siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/security"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:beans="http://www.springframework.org/schema/beans"
  xsi:schemaLocation="
          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans.xsd
          http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security.xsd">
  <http auto-config="true">
    <intercept-url pattern="/**" access="ROLE_USER" />
  </http>
  <authentication-manager alias="authenticationManager">
    <authentication-provider>
      <user-service>
        <user authorities="ROLE_USER" name="guest" password="guest" />
      </user-service>
    </authentication-provider>
  </authentication-manager>
</beans:beans>
```

Aunque luego explicaremos el significado de las etiquetas con más detalle, puede intuirse echándole un breve vistazo al XML que estamos protegiendo todas las URL (/**, se utiliza la notación de Ant para los *paths*), permitiendo solo el acceso a los usuarios con rol `ROLE_USER`. A continuación definimos un usuario con este rol y con login y password "guest".

**Aviso:**

Por defecto, **los nombres de los roles deben comenzar por "`ROLE_`"**, otros nombres no serán considerados como válidos. Consultar la documentación para ver cómo cambiar dicho prefijo.

El atributo `auto-config="true"` activa por defecto los servicios de autentificación BASIC, autentificación a través de formulario autogenerado por Spring y gestión de logout. Iremos viendo dichos servicios con más detalle en los siguientes apartados, y cómo adaptarlos a nuestras necesidades.

Por último, para terminar con nuestra configuración mínima tenemos que **indicarle al contenedor de beans de Spring dónde está nuestro fichero de configuración de seguridad**. Como ya se vio en la primera sesión del módulo, en aplicaciones web la forma estándar de hacer esto es definiendo un *listener* de la clase



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

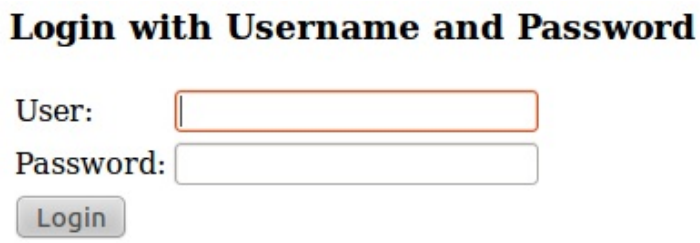
`ContextLoaderListener` en el web.xml y pasándole un parámetro `contextConfigLocation` con el/los nombre/s del/los fichero/s de configuración de beans:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
    /WEB-INF/spring/root-context.xml
    /WEB-INF/spring/security-context.xml
    </param-value>
</context-param>

<listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

En el ejemplo estamos suponiendo que el fichero con la configuración de seguridad que hemos mostrado antes se llama `security-context.xml` y que tenemos otro fichero de configuración de beans donde definimos el resto de elementos de nuestra aplicación no relacionados con la seguridad.

Si ejecutamos la aplicación e intentamos acceder a una URL cualquiera dentro de ella, veremos que nos aparece la siguiente página de login, generada automáticamente por Spring Security:




página de login generada automáticamente por Spring Security

Si introducimos "guest" como usuario y contraseña podremos acceder a la URL deseada. Si introducimos otras credenciales, volverá a mostrarse la página de login con un mensaje generado automáticamente por Spring ("Your login attempt was not successful, try again"). Evidentemente tal y como está la página de login y el mensaje de error no es usable en una aplicación real, porque no emplea ni el idioma ni el "look&feel" de nuestra aplicación, pero al menos así podemos comprobar de manera rápida que la seguridad funciona. Por supuesto todo esto es configurable y adaptable a nuestras necesidades de aspecto e idioma como veremos luego.

Llegados a este punto hay que hacer notar que hasta ahora, al menos a nivel funcional la seguridad que proporciona Spring Security y la estándar de JavaEE son muy similares: se protegen determinadas URL permitiendo solo el acceso a determinados roles y "alguien" (aquí Spring, en el estándar el contenedor web) intercepta las peticiones y comprueba que



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

se posean las credenciales adecuadas. No obstante y aun con este ejemplo tan sencillo Spring Security tiene dos ventajas sobre la seguridad estándar:

- La portabilidad: como se vio en el módulo de componentes web, aunque en JavaEE las etiquetas de configuración de seguridad del web.xml están estandarizadas no lo está la forma de definir los usuarios. En Tomcat por ejemplo se pueden definir en un archivo tomcat-users.xml, pero esto no es así en otros servidores. Curiosamente en este aspecto Spring resulta ser más portable que el estándar (!).
- Con la seguridad estándar no se puede saltar directamente a la página de login, siempre hay que usar el "truco" de intentar el salto a una URL protegida para que el contenedor nos redirija a la de login. Si se salta directamente a la de login y se introducen las credenciales se produce un error. Esto a veces hace un poco "retorcida" la configuración de la página inicial de la aplicación. Sin embargo en Spring Security no hay problema en saltar directamente a la página de login (spring_security_login por defecto, aunque por supuesto es configurable, como veremos) ya que se puede especificar a qué página saltar por defecto tras esta.

Una tercera ventaja que veremos conforme vayamos complicando los ejemplos es que Spring Security permite mucha más flexibilidad y es mucho más potente que la seguridad estándar, manteniendo la portabilidad: podemos por ejemplo autentificar contra un servidor LDAP, usar *single sign on* con varias aplicaciones, proteger los métodos de la capa de negocio y muchas otras cosas. Hay que destacar que muchas de estas características también están disponibles en prácticamente todos los servidores de aplicaciones, pero la configuración es propia de cada uno. Por ejemplo, en JBoss, Weblogic o Glassfish es posible autentificar con LDAP, pero en cada uno la configuración se hace de manera distinta.

## 13.3. Autentificación contra una base de datos

Hasta ahora hemos almacenado las credenciales de los usuarios en el propio fichero de configuración. Obviamente esto sirve para hacer pruebas sencillas pero no para una aplicación en producción, necesitamos un mecanismo más realista para obtener las credenciales. Como ya hemos dicho, en Spring Security esta tarea la realizan los proveedores de autentificación.

Spring Security incorpora diversos proveedores de autentificación "listos para usar", basados en tecnologías tan diversas como certificados digitales, LDAP, JAAS, sistemas *single sign-on* como CAS o OpenId,... no obstante, su uso y configuración quedan fuera del ámbito de estos apuntes. Vamos a ver lo más habitual en aplicaciones web, que es almacenar las credenciales en la base de datos, lo que permite modificarlas y gestionarlas de manera sencilla. En Spring se usa un *DAO authentication provider* para esta tarea. La configuración más sencilla que podemos usar es la que viene a continuación:

```
<authentication-manager alias="authenticationManager">
    <authentication-provider>
```

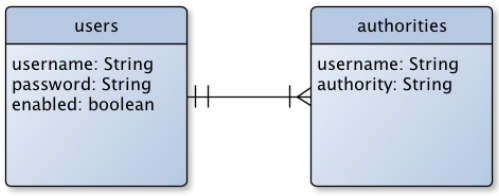


Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
      <jdbc-user-service data-source-ref="miDataSource"/>
    </authentication-provider>
</authentication-manager>

<jee:jndi-lookup id="miDataSource" jndi-name="jdbc/securityDS"
resource-ref="true"/>
```

Como las credenciales están en una base de datos debemos conectarnos con ella a través de un dataSource. Ya vimos en la primera sesión cómo acceder a dataSources JNDI con la etiqueta `<jndi-lookup>`. El *authentication provider* por defecto asumirá que la base de datos tiene una determinada estructura, que se muestra en la figura siguiente:




Esquema de BD por defecto para la autentificación JDBC

Como puede verse, se asume que tenemos dos tablas, una para guardar el login y password de cada usuario y otra para guardar los roles, que en Spring Security se denominan *authorities*. Entre ambas hay una relación uno a muchos ya que evidentemente un usuario puede tener varios roles. El campo enabled de la primera tabla indica si un usuario está habilitado o no (los usuarios deshabilitados no pueden acceder a los recursos).

Mucho más habitual que usar esta configuración por defecto será emplear el esquema de base de datos que tenga nuestra aplicación. Por ejemplo, vamos a suponer que nuestro esquema es el de la siguiente figura, que se muestra comparado con el esperado por Spring por defecto.


Nuestro esquema de BD comparado con el usado por defecto

Por desgracia, no podemos configurar esto directamente con la etiqueta jdbc-user-service. Lo más simple es definir un *bean* de la clase `JdbcDaoImpl`, que se encarga de pasarle los datos de los usuarios al *authentication provider* y es configurable en este y otros aspectos.

La adaptación a nuestro esquema de base de datos se basa en que el `JdbcDaoImpl` usa una consulta SQL predefinida para obtener login y password de un usuario y otra para obtener los roles asociados. Las dos consultas por supuesto presuponen el esquema anterior. Lo que tendremos que hacer es suministrar consultas propias que devuelvan los resultados con los mismos nombres. En primer lugar, para comprobar el password se hace:

```
SELECT username, password, enabled
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
  FROM users
  WHERE username = ?
```

Donde el campo `enabled`, del que carece nuestra base de datos, indica si el usuario está o no activado. Con nuestro esquema, para devolver los mismos resultados que la consulta anterior, haríamos:

```
SELECT login as username, password, true as enabled
  FROM usuarios
  WHERE login=?
```

Por otro lado, para obtener los roles (authorities) de un usuario, se hace:

```
SELECT username, authority
  FROM authorities
  WHERE username = ?
```

Con nuestro esquema de base de datos, haríamos:

```
SELECT login as username, rol as authority
   FROM roles
   WHERE login=?
```

Estas consultas se modifican a través de las propiedades `usersByUsernameQuery` y `authoritiesByUsernameQuery` de `JdbcDaoImpl`. Así, nuestro XML quedaría:

```
<authentication-manager alias="authenticationManager">
  <authentication-provider user-service-ref="miUserServiceJDBC" />
</authentication-manager>

<beans:bean id="miUserServiceJDBC"
  class="org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl">
  <beans:property name="dataSource" ref="miDataSource"/>
  <beans:property name="usersByUsernameQuery"
                value="SELECT login as username, password, true as enabled
                       FROM usuarios WHERE login=?"/>
  <beans:property name="authoritiesByUsernameQuery"
                value="SELECT login as username, rol as authority
                       FROM roles WHERE login=?"/>
</beans:bean>

<jee:jndi-lookup id="miDataSource" jndi-name="jdbc/securityDS"
resource-ref="true"/>
```

Donde definimos y configuramos un bean de la clase `JdbcDaoImpl`, le damos un nombre arbitrario y usamos ese nombre en el atributo `user-service-ref` de la etiqueta que define el proveedor de autentificación

**Passwords en claro**

Evidentemente, dejar los passwords en claro en la base de datos es una nefasta práctica de seguridad, muy peligrosa en caso de que alguien consiga la información de la tabla de usuarios. Por eso Spring Security nos permite usar hashes de los passwords originales de manera bastante sencilla y transparente al proceso de autentificación. El usuario escribirá el password en claro en el formulario de login, pero automáticamente se hará un hash de este y se comprobará contra el



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

hash almacenado en la base de datos. La configuración es muy sencilla pero el proceso se complica un poco (no demasiado) al tener que codificar el password para guardarlo en la base de datos cuando el usuario se da de alta. Se recomienda leer la documentación de Spring Security para ver cómo implementarlo.

## 13.4. Seguridad de la capa web

Vamos a ver aquí cómo configurar y controlar todo lo relacionado con la seguridad de la capa web: la visualización o no de ciertos fragmentos de HTML dependiendo del rol, el acceso a los métodos de los controladores, etc

### 13.4.1. Configuración del login basado en formularios

La mayoría de aplicaciones web usan un formulario HTML para que el usuario introduzca su login y password. Hemos visto que por defecto Spring crea automáticamente este formulario, pero lo habitual será que lo hagamos nosotros para poder darle el "look and feel" de la aplicación. Esto se consigue con la etiqueta `form-login`:

```
<http pattern="/login.html" security="none"/>

<http>
  <intercept-url pattern="/**" access="ROLE_REGISTRADO, ROLE_ADMIN" />
  <form-login login-page="/login.html" default-target-url="/main.html" />
</http>
```

Con el atributo `login-page` se especifica la página que contiene el formulario de login y con `default-target-url` la dirección a la que se saltará por defecto. Como ya hemos dicho, en este aspecto la autentificación con formulario de Spring se diferencia ligeramente de la seguridad declarativa estándar de JavaEE. En el estándar no se suele saltar directamente a la página de login, sino que esta se muestra automáticamente cuando el usuario intenta acceder a un recurso protegido. En Spring nada nos impide acceder directamente a la página de login, ya que se nos redirigirá una vez hecho login a la página indicada por `default-target-url`. Independientemente de ello, por supuesto, cuando en Spring se intenta acceder a un recurso protegido también "salta" la página de login.

Nótese que la existencia de la página de login nos obliga a desprotegerla para que los usuarios puedan acceder a ella. Esto se hace poniendo otro elemento `http` aparte que indique que esta página no debe estar protegida. Hay que llevar cuidado ya que Spring aplicará el primer patrón que encaje con la URL, de modo que si pusiéramos esta línea al final no funcionaría correctamente (ya que se aplicaría el patrón "/**").

**Bucles infinitos en la web**

Si se nos olvida desproteger la URL de la página de login, el navegador mostrará un mensaje del estilo "Esta página web tiene un bucle de redireccionamiento" o "demasiados redireccionamientos", al intentar el navegador múltiples redirecciones a la misma página que no



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

tienen éxito al ser un recurso protegido.

La página de login contendrá un formulario HTML cuyos campos deben tener un nombre estándar, al estilo de los que se usan en seguridad declarativa JavaEE:

```
<form action="j_spring_security_check" method="post">
    Usuario: <input type="text" name="j_username"/> <br/>
    Contraseña: <input type="password" name="j_password"/> <br/>
    <input type="submit" value="Entrar"/>
</form>
```

Spring nos ofrece un **servicio de logout** que se encarga de invalidar automáticamente la sesión HTTP y, si lo deseamos, redirigir al usuario a una página de "salida". Este servicio se configura con la etiqueta `logout`, que se debe colocar dentro de la de `http`:

```
<http>
   ...
   <logout logout-url="/logout" logout-success-url="/adios.jsp"/>
</http>
```

El atributo `logout-url` indica qué URL "disparará" el proceso. Por tanto, para que el usuario pueda hacer *logout* bastará con un enlace a esta URL en cualquier página. Por defecto esta URL es `/j_spring_security_logout`. Con `logout-success-url` indicamos a qué página se saltará tras invalidar la sesión. Por defecto es "/".

Por otro lado Spring nos da la posibilidad de **recordar que ya hemos hecho login** para ahorrarnos la operación en sucesivas visitas al sitio desde la misma máquina, aunque cerremos el navegador. Esta es una opción muy habitual en muchos sitios web y normalmente se implementa guardando en una *cookie* un *token* de autentificación, que asegura que en algún momento hemos hecho login correctamente. En Spring esta funcionalidad se llama "remember-me" y se implementa con una *cookie* que es por defecto un hash md5 del login y password del usuario, la fecha de expiración del *token* y una palabra clave propia de la aplicación. Para activar el "remember-me" hay que usar la etiqueta del mismo nombre dentro de la de `http`:

```
<http>
    ...
    <remember-me key="claveDeLaAplicacion"/>
</http>
```

`key` es simplemente un valor que solo nosotros deberíamos conocer y que se combina con el resto de campos para darle una mayor seguridad al *token* y que además este sea exclusivo de la aplicación web.

Además para que se active esta funcionalidad debemos añadir un campo al formulario de login. Este campo le permitirá al usuario elegir si desea usar o no la característica, y por defecto debe llamarse `_spring_security_remember_me`. Nuestro formulario de login quedaría así:



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
<form action="j_spring_security_check" method="post">
    Usuario: <input type="text" name="j_username"/> <br/>
    Contraseña: <input type="password" name="j_password"/> <br/>
    <input type="checkbox" name="_spring_security_remember_me"/>
         Recordar mis credenciales <br>
    <input type="submit" value="Entrar"/>
</form>
```

### 13.4.2. Localización de los mensajes de error

Todas las excepciones que saltan cuando se produce algún problema de autorización tienen los mensajes internacionalizados y externalizados en ficheros `.properties`. Los mensajes en inglés están en `messages.properties` en el paquete `org.springframework.security` y también traducidos a más de una decena de idiomas siguiendo la convención habitual (incluyendo español, en `messages_es_ES.properties`). Por tanto, y siguiendo lo que vimos en sesiones anteriores basta con definir un messageSource en nuestro fichero de configuración de beans que referencie este archivo de mensajes (recordemos que en esta sesión hemos optado por tener un fichero únicamente con la configuración de seguridad y otro con la configuración del resto de beans. Esto se debería incluir en este último:

```
<bean id="messageSource"
class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
      <property name="basename"
value="classpath:org/springframework/security/messages"/>
</bean>
```

Podemos mostrar en cualquier JSP el último mensaje de error generado por Spring haciendo uso de SPRING_SECURITY_LAST_EXCEPTION.message, que Spring guarda en ámbito de sesión. Si hemos instanciado el messageSource como acabamos de explicar, el mensaje aparecerá automáticamente en el locale actual:

```
  ${sessionScope.SPRING_SECURITY_LAST_EXCEPTION.message}
```

### 13.4.3. Autentificación BASIC vs. basada en formulario

En la autentificación BASIC, el navegador muestra una ventana de tipo "popup" en la que introducir login y password. En realidad, la mayor utilidad de este mecanismo es para el acceso con un cliente de tipo REST o de escritorio, ya que la forma de envío de login y password al servidor es sencilla de implementar y no requiere el mantenimiento de sesiones, a diferencia del login con formulario.

Para usar autentificación BASIC, simplemente colocaríamos la etiqueta `http-basic` en el XML:

```
<http>
    <intercept-url pattern="/**" access="ROLE_USER" />
    <http-basic/>
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
</http>
```

La autentificación BASIC se puede tener funcionando simultáneamente con la de formulario. Así, un navegador que intente acceder a una URL protegida será redirigido al formulario, mientras que por ejemplo un cliente REST que envíe la cabecera "Authorization" con login y password (la usada por el estándar BASIC) tendrá el acceso permitido si las credenciales son correctas. De hecho, como ya hemos visto, la opción `auto-config=true` pone las dos en funcionamiento simultáneamente. También podemos configurar por separado la seguridad para clientes REST y web de modo que cada uno tenga un punto de entrada distinto en la aplicación:

```
<!-- servicios REST sin estado con autentificación Basic -->
<http pattern="/restful/**" create-session="stateless">
  <intercept-url pattern='/**' access='ROLE_REMOTE' />
  <http-basic />
</http>

<!-- Desproteger la página de login-->
<http pattern="/login.htm*" security="none"/>

<!-- Clientes web con autentificación basada en formulario -->
<http>
  <intercept-url pattern='/**' access='ROLE_USER' />
  <form-login login-page='/login.htm' default-target-url="/home.htm"/>
  <logout />
</http>
```

El `create-session="stateless"` del ejemplo anterior le indica a Spring que no es necesario mantener una HttpSession en el servidor para autentificar a los clientes REST, ya que estos enviarán en cada conexión las credenciales.

### 13.4.4. Seguridad de "grano fino" en los JSPs

Ya hemos visto cómo permitir o denegar el acceso a una página completa. Ahora vamos a ver cómo hacer que se muestre o no determinada sección de un JSP. Por ejemplo podríamos mostrar solo el menú de administrador a los usuarios con ROLE_ADMIN, o mostrar únicamente el logout a los usuarios que se hayan autentificado. Para ello lo más sencillo es usar la *taglib* de Spring Security. La etiqueta básica para implementar esta funcionalidad es `authorize`, que debe "envolver" la sección de JSP que deseamos mostrar (o no).

Primero hay que tener en cuenta que para usar esta *taglib* debemos introducir otra dependencia en el `pom.xml`

```
<dependency>
      <groupId>org.springframework.security</groupId>
      <artifactId>spring-security-taglibs</artifactId>
      <version>3.1.3.RELEASE</version>
</dependency>
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

Hay dos formas de controlar si se muestra o no la sección de código JSP. La primera es usar una URL como referencia, de modo que el código solo se mostrará a los usuarios que tengan permiso para acceder a ella. Para esto se usa el atributo llamado, precisamente, `url`

```
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags"
%>
...
<sec:authorize url="/admin/eliminar">
  <a href="/admin/eliminar">Eliminar</a>
</sec:authorize>
```

Como se ve, esta forma de uso es muy adecuada para los casos en los que solo queremos mostrar un enlace a los usuarios que tienen permiso para acceder a él. El resto no lo vería. Evidentemente aunque aquí hemos puesto un enlace, en general es un fragmento arbitrario de código JSP.

La otra forma de usar `authorize` es con una expresión en lenguaje SpEL (*Spring Expression Language*) que mostrará el código solo si se evalúa a `true`. Ya vimos en la primera sesión del módulo una muy breve introducción a este lenguaje, que nos permite entre otras cosas evaluar expresiones aritméticas y lógicas o llamar a métodos. Hay una serie de métodos de Spring Security específicamente diseñados para ser usados con SpEL, por ejemplo:

- `hasRole(rol)`: es cierta si el usuario actual que se ha autentificado tiene el rol especificado
- `hasAnyRole(rol1, rol2, ...)`: es cierta si el usuario actual que se ha autentificado tiene uno de los rol especificados (se ponen separados por comas)
- `permitAll()`: es cierta por defecto, indicando por tanto que queremos permitir el acceso a todos los usuarios. `denyAll()` sería lo contrario.
- `isFullyAuthenticathed()`: es cierta si el usuario se ha autentificado con login y password. Esto no sería así por ejemplo en el caso en que se haya usado el remember-me. En este caso, sería cierta la expresión `isRememberMe()`.
- `hasIpAddress(dir)`: cierta si el usuario se ha autentificado desde esta IP. Se pueden usar rangos como '192.168.1.1/20'

La expresión de SpEL hay que ponerla como valor del atributo `access` de la *tag* `authorize`. Por ejemplo, para permitir acceso solamente a un administrador que se conecte desde la máquina local:

```
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags"
%>
...
<sec:authorize access="hasRole('ROLE_ADMIN') and
hasIpAddress('127.0.0.1')">
  <p>Esto solo lo debería ver un admin conectado localmente</p>
</sec:authorize>
```

Cuidado: para poder usar expresiones de SpEL en chequeo de permisos es necesario



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

activar su soporte. La forma más sencilla (aunque engorrosa, y enseguida veremos por qué) es añadir el atributo `use-expressions="true"` en la etiqueta `http` que proteja la URL donde vamos a usar el *tag*. Y hemos dicho engorrosa, porque este atributo nos va a obligar a reescribir los valores de access de las etiquetas intercept-url que estén dentro, ya que ahora estos valores no se toman como nombres textuales de roles, sino como expresiones. Así, el equivalente a por ejemplo `ROLE_ADMIN` ahora sería `hasRole('ROLE_ADMIN')`. Otro ejemplo:

```
<http use-expressions="true">
    <intercept-url pattern="/**" access="hasAnyRole('ROLE_USER',
'ROLE_ADMIN')"/>
</http>
```

## 13.5. Seguridad de la capa de negocio

Para mayor seguridad podemos controlar los permisos al ejecutar cualquier método Java. Los métodos restringidos se pueden especificar de dos formas: con anotaciones en el código fuente o con etiquetas XML en el fichero de configuración. Como siempre, la ventaja de la anotación es que el código queda más claro y autocontenido. Por otro lado, si colocamos las restricciones de acceso en un fichero XML podemos hacer que estas afecten a múltiples métodos y no solo a uno, como luego veremos

Lo primero que necesitamos es la habilitar este tipo de seguridad. Esta configuració se hace con la etiqueta `global-method-security`

En una aplicación web, el intento de ejecutar código sin permiso acabará generando una respuesta HTTP con código 403 (acceso denegado), gracias a los filtros de Spring. Esto nos permite tratar de manera uniforme las denegaciones de acceso sean por URL o por código.

### 13.5.1. Seguridad con anotaciones estándar

Podemos anotar los métodos a los que queramos restringir el acceso. En el estándar JavaEE, las anotaciones del JSR-250 son las que se usan con esta finalidad. Spring ofrece soporte para este estándar, aunque también tiene una anotación propia más o menos equivalente llamada `@Secured`. Evidentemente, usar las anotaciones estándar aumentará la portabilidad de nuestro código, por lo que es lo más recomendable.

En este caso, en el XML de configuración de seguridad deberíamos incluir:

```
<global-method-security jsr250-annotations="enabled"/>
```

Si quisiéramos usar la anotación `@Secured` deberíamos incluir el atributo `secured-annotations="enabled"`. Ambos tipos de anotaciones pueden usarse simultáneamente.



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

En el código a proteger, escribiremos:

```
@RolesAllowed("ROLE_ADMIN")
public void eliminarUsuario(UsuarioTO uto) {
      ...
}
```

Al igual que en el estándar, si colocamos la anotación delante de la clase, estamos protegiendo todos sus métodos.

### 13.5.2. Seguridad con anotaciones de Spring Security 3

La versión 3 del framework añadió la posibilidad de usar expresiones SpEL en las anotaciones de seguridad del código. No obstante, las anotaciones que hemos visto (`@RolesAllowed` y `@Secure`) no soportan el uso de expresiones. Se introducen para ello dos anotaciones más: `@PreAuthorize` y `@PostAuthorize`. La primera de ellas, como su propio nombre indica, chequea que la expresión SpEL sea cierta antes de ejecutar el código. La segunda, tras ejecutarlo.

Para poder usar estas anotaciones lo primero es configurarlas en el XML de Spring Security:

```
<global-method-security pre-post-annotations="enabled"/>
```

Una vez hecho esto, podemos anotar cualquier método que queramos proteger y en que la condición de chequeo vaya más allá de tener un determinado rol. Además de usar las expresiones típicas de SpEL, podemos usar las funcionalidades propias de seguridad que ya vimos en el apartado de la capa web (hasIPAddress, isFullyAuthenticated,...). Además podemos referenciar los valores de los parámetros, precediéndolos del símbolo '#'. Por ejemplo supongamos una aplicación de mensajería en la que un usuario puede enviar un mensaje a otro a través de este interfaz:

```
public interface IMensajeriaBO {
   ...
   public void enviarMensaje(UsuarioTO uto, MensajeTO mto);
   ...
}
```

Y supongamos que queremos chequear no solo que el usuario actual se ha autentificado, sino que además tiene crédito para enviar el mensaje (que `uto.getCredito()` devuelve un valor mayor que cero.). Podríamos usar la siguiente anotación:

```
public interface IMensajeriaBO {
   ...
   @PreAuthorize("hasRole('ROLE_USER') and #u.credito>0")
   public void enviarMensaje(Usuario u, Mensaje m);
   ...
}
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

Nótese que en SpEL podemos llamar a los *getters* con la notación `objeto.propiedad`.

En ciertos casos puede ser interesante acceder a información sobre el usuario actual. Esta información la tenemos en el objeto `principal`. Por ejemplo, supongamos que queremos verificar que un usuario está intentando cambiar su propio password y no el de otro:

```
public interface IUsuarioBO {
   @PreAuthorize("#u.login == principal.username and
hasRole('ROLE_USER')")
   public void cambiarPassword(Usuario u, String nuevoPassword);
}
```

Por otro lado las anotaciones @PreFilter y @PostFilter sirven para filtrar colecciones. La más habitual es @PostFilter, que elimina los elementos que no cumplan la condición una vez se ha llamado al método. Por ejemplo, supongamos que los administradores pueden listar los datos de los demás usuarios, pero no los de los otros administradores, que solamente puede listar el usuario con rol 'ROLE_ROOT'. Vamos a suponer que la clase `Usuario` tiene un método `isAdmin`() que nos indica si un usuario es o no administrador. Podríamos hacer algo como:

```
public interface IUsuarioBO {
...
   @PostFilter("hasRole('ROLE_ROOT') or (hasRole('ROLE_ADMIN')
                and !filterObject.isAdmin())")
   public List<Usuario> getUsuarios();
...
}
```

Lo que va haciendo Spring es aplicar la expresión uno a uno a los objetos de la colección, eliminando aquellos que no la cumplan (visto de otro modo, que la expresión se cumpla indica los objetos que "dejamos pasar"). El `filterObject` es el objeto actual que estamos comprobando si filtrar o no.

### 13.5.3. Seguridad en el XML de configuración

La principal ventaja de esta forma de trabajar es que podemos cambiar la seguridad sin necesidad de tocar una sola línea de código. En el XML, con la etiqueta `protect-pointcut` podemos especificar una expresión AOP que indique qué puntos del código queremos proteger e indicar a qué roles les será permitido el acceso. Para hacernos una idea rápida (y un poco grosera), podríamos verlo como que estamos poniendo una expresión regular con la que deben encajar los métodos que queremos proteger. La sintaxis de esta pseudo-expresión regular es la de AspectJ, el framework AOP más difundido en el mundo Java y que es el que usa Spring.

```
<global-method-security>
    <protect-pointcut
                    expression="execution(* eliminarUsuario(..))"
                    access="ROLE_ADMIN"/>
</global-method-security>
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

Lo que viene a decir la expresión AspectJ anterior es que queremos interceptar la ejecución de cualquier método llamado `eliminarUsuario`, tenga el número y tipo de argumentos que tenga (el ".." es un comodín para uno o más *tokens*) y devuelva lo que devuelva el método (como solo habrá un valor de retorno ponemos el comodín para un solo *token*, "*"). Así, para ejecutar cualquier método que encaje con esta expresión habrá que tener rol `ROLE_ADMIN`. Se recomienda consultar el apéndice de AOP de los apuntes para más información sobre la sintaxis de las expresiones de AspectJ soportadas por Spring y sobre AOP en general.



*Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.*

# 14. Ejercicios de Spring Security

Vamos a añadirle Spring Security a la aplicación de alquiler de coches que llevamos usando desde la sesión 5. No importa si no has completado todavía los ejercicios de las sesiones 5 y 6, son independientes de estos.

## 14.1. Seguridad de la capa web (1 punto)

Lo primero que debes hacer es la **configuración básica de la seguridad en la capa web**. Las dependencias de Spring Security ya están incluidas en el pom.xml, por tanto lo que te queda por hacer, como se comenta en los apuntes, es:

1. Definir los filtros correspondientes en el web.xml para que Spring Security pueda interceptar las peticiones
2. También en el web.xml, especificar dónde va a estar y cómo se va a llamar nuestro fichero de configuración de seguridad. Debes llamarlo security.xml y colocarlo en src/main/webapp/config, con el resto de ficheros de configuración de Spring.
3. Crear el propio security.xml, puedes tomar como base el de los apuntes. Modifícalo para que haya dos roles, ROLE_USUARIO y ROLE_GESTOR. El primero será un usuario externo de la web, que únicamente podrá ver los coches. El segundo será un trabajador de la empresa, que además podrá hacer altas y modificaciones de coches. Ya implementaremos estos permisos, por el momento simplemente crea los dos roles. Crea además en el fichero algún usuario de prueba de cada tipo, poniéndole el login y password que desees.

Comprueba que la seguridad funciona. Al arrancar la aplicación, en lugar del listado de coches debería aparecer el formulario de login auto-generado por Spring Security. Comprueba que se puede entrar correctamente en la aplicación.

**Cómo hacer logout por las malas**

Para poder hacer logout puedes usar el "truco" de borrar en el navegador la cookie JSESSIONID. Si no, tendrás que cerrar el navegador.

**Seguridad y acceso remoto**

Si has hecho el ejercicio 1 de la sesión de acceso remoto, desprotege la URL /remoting/* si quieres que siga funcionando. Una solución mejor sería asignarle a esta URL autentificación BASIC y hacer que el cliente remoto la use para conectarse al servidor. Para ver cómo hacer esto, consultar por ejemplo este tutorial

## 14.2. Personalización de la seguridad web (1 punto)

Una vez hecho esto, **personaliza la seguridad web**:



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

1. Crea un formulario de login personalizado en la página login.jsp y configura Spring Security para que lo use. La default-target-url (a la que se saltará tras hacer login) debe ser "listar.do".
2. Configura el "servicio" de logout y añádele al menos a la página de listar coches (WEB-INF/views/listar.jsp) un enlace para hacer "logout"

Una vez tengas todo esto funcionando, cambia la configuración de la seguridad para **tomar los datos de los usuarios de la base de datos** en lugar de tenerlos definidos en el security.xml

## 14.3. Seguridad en los JSP (0.5 puntos)

Añade algo de **seguridad de "grano fino"** a listar.jsp para que el enlace "editar" al lado de cada coche y el botón de "crear" solo aparezcan si el usuario actual tiene rol ROLE_GESTOR.

## 14.4. Seguridad de la capa de negocio (0.5 puntos)

Queremos reservar los coches que llevan pocos kilómetros solo para ciertos clientes VIP. Añadir un filtro al método listar() de la capa de negocio de modo que solo los gestores puedan ver los coches con menos de 1000 km.



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

# 15. Desarrollo rápido de aplicaciones con Spring Roo

## 15.1. Introducción rápida a Spring Roo

### 15.1.1. ¿Qué es Roo?

Aunque casi nadie niega que JavaEE sea una plataforma robusta, escalable y que incorpora multitud de funcionalidades, no brilla precisamente por su productividad. Y eso que se ha simplificado mucho con los años. En gran medida el problema procede del propio lenguaje de desarrollo: Java es un lenguaje estáticamente tipado e induce a un estilo de programación demasiado verboso.

Como alternativa, las plataformas de desarrollo rápido (RAD) de aplicaciones web, como Ruby on Rails o Django, suelen usar lenguajes de tipado dinámico o filosofías como la "convención por encima de la configuración" (*convention over configuration*) con el objetivo de hacer el desarrollo más rápido y ágil y el producto resultante más ligero y simple. En el mundo Java tenemos un representante de estas ideas en **Groovy on Grails**, inspirado en Rails y usando el lenguaje dinámico Groovy pero con la plataforma JavaEE por debajo (Grails tiene su propio módulo en este curso).

Spring Roo toma prestadas algunas de las ideas típicas de las plataformas al estilo Rails (la convención por encima de la configuración, el *active record* para la persistencia,...), pero conceptualmente está más cerca de las herramientas MDA y de las de generación de código. Es decir, básicamente Roo nos va a generar el esqueleto de una aplicación Spring completa, ahorrándonos el tedioso trabajo de tener que configurarlo todo manualmente y tener que escribir el típico código repetitivo, como por ejemplo las partes de CRUD en JPA o en la web.

Roo respeta los siguientes principios:

- Uso de tecnologías "estándar" (en el sentido de ampliamente conocidas y probadas): genera un proyecto Maven con código Spring, JPA y Tiles entre otros.
- No incorpora componentes en tiempo de ejecución. No hay lenguajes interpretados ni librerías adicionales.
- Separa físicamente el código generado de los archivos editables por el desarrollador, de modo que al modificar estos últimos no "rompamos" o trastoquemos la aplicación, que es un problema típico de las herramientas de generación de código. Como veremos, esto se consigue mediante el uso de AOP. Esto no significa que no haya que tener cuidado al editar el código. Luego veremos algunas directrices.
- Intenta evitar convertirse en una dependencia imprescindible: de modo que si en algún momento deseamos dejar de usarlo podamos refactorizar el código de manera sencilla y pasar a tener un código fuente sin dependencias de él.

Evidentemente, en desarrollo software no hay fórmulas mágicas, así que Roo no puede



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

resolver todos nuestros problemas ni escribir automáticamente una aplicación no trivial (pero quizá pueda darnos un 80% del código hecho para concentrarnos en el 20% realmente interesante). Por otro lado, y como ya hemos dicho, al ser una herramienta que genera código hay que llevar cierto cuidado con los cambios que hacemos para evitar que la aplicación se "rompa".

### 15.1.2. Arquitectura de las aplicaciones Roo

Habitualmente las aplicaciones web JavaEE se estructuran en tres capas: la capa web, la de negocio y la de acceso a datos. En la de negocio suele haber una "sub-capa" de servicios (Business Objects) que ofrecen eso, servicios, a la capa web o a clientes remotos (REST y otros). Además aquí se suele considerar que está el modelo del dominio. Finalmente, en la capa de acceso a datos habitualmente tenemos los DAOs, que se encargan de la persistencia.

La arquitectura de las aplicaciones generadas por Roo por defecto es mucho más simple que la que acabamos de describir. Básicamente tiene dos capas: la web y la de entidades o modelo del dominio. Las entidades se convierten en una especie de "superentidades" que se encargan no solo de modelar el dominio sino también de tareas como su propia persistencia, la validación de sus datos e incluso su serialización a JSON. Esta idea de que los propios objetos del dominio se encarguen de su persistencia no es exclusiva de Roo, en realidad es muy típica de plataformas RAD como Rails, Django, Grails y otras, y se conoce habitualmente como *Active Record*.

En ciertos casos nos interesará tener en la aplicación servicios y/o DAOs. Aunque no los use por defecto, Roo también puede generarlos si se lo pedimos. No obstante, en estos apuntes nos concentraremos en el modelo más simple de dos capas.

### 15.1.3. Roo en diez minutos o menos

Desde el punto de vista de su uso, Roo está diseñado como un shell interactivo al estilo de Rails o de Grails. Para facilitar su uso tiene autocompletado de los comandos y ayuda contextual. Además en todo momento nos mostrará solo los comandos que sean válidos y nos dará pistas de cuál es la siguiente tarea a realizar si estamos un poco perdidos.

Vamos a ver aquí cómo se interactúa con Roo a nivel básico, teclearemos comandos sin explicar en mucho detalle qué hace cada uno, en eso ya entraremos en los siguientes apartados.

Crearemos la típica aplicación de gestión de tareas pendientes. Para comenzar, lo primero es crear un directorio para el proyecto:

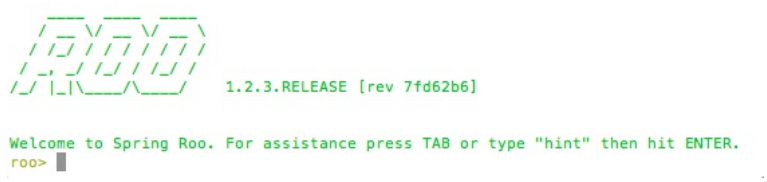
```
mkdir tareas
cd tareas
```

Y ahora ya podemos arrancar el shell de Roo. El script en sistemas Unix se llama `roo.sh`



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

(`roo.bat` en Windows). Aparecerá la siguiente pantalla:

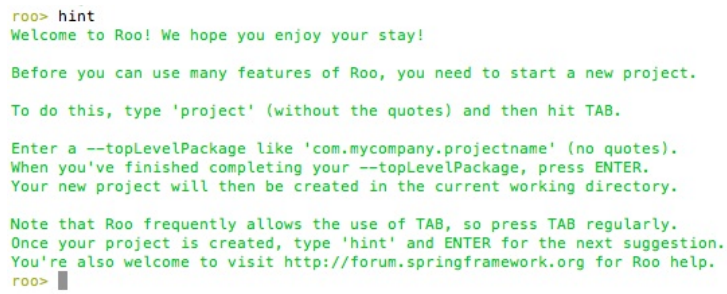



El shell de Spring Roo

**Cuidado con el directorio inicial**

Cuando arranca, el *shell* de Roo examina recursivamente los subdirectorios del directorio actual, de modo que si lo ejecutamos desde nuestro `$HOME` o desde la raíz del disco, puede pasarse mucho rato navegando por subdirectorios y examinando archivos y tener problemas de memoria. Por eso hay que ejecutarlo siempre desde el propio directorio del proyecto.

El propio *shell* nos sugiere que pulsemos TAB para autocompletar comandos. Si lo hacemos, aparecerá una lista con los comandos disponibles, que no son todos los de Roo, sino solo los que tiene sentido usar en este momento. Además, el *shell* nos indica que si tecleamos el comando hint, nos "dará una pista" de qué hacer a continuación:

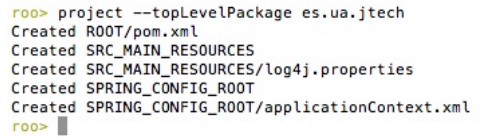



El comando 'hint' nos dice qué podemos hacer a continuación

Como vemos, lo primero que debemos hacer es crear la estructura del proyecto con el comando `project`. Para ahorrarnos tecleos innecesarios podemos emplear TAB para autocompletar, de modo que si tecleamos proj-TAB-TAB el resultado será:

```
project --topLevelPackage
```

Ahora tecleamos un nombre para el "dominio" base del proyecto, por ejemplo es.ua.jtech. Roo creará la estructura básica necesaria para comenzar a trabajar con nuestro proyecto:

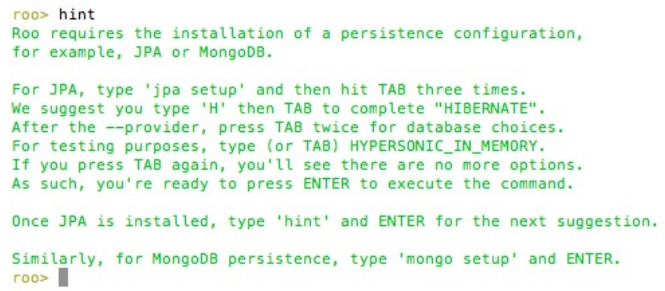



Creando la estructura del proyecto con el comando 'project'



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

Podemos ver que se ha creado el pom.xml de Maven y un fichero XML de configuración de beans de Spring. Ahora, si no sabemos qué hacer, podemos teclear de nuevo 'hint':




Empezando a trabajar con la capa de persistencia

Y vemos que Roo nos sugiere empezar a trabajar con la capa de persistencia de nuestra aplicación. Lo primero de todo es seleccionar lo que Roo llama un "proveedor de persistencia", es decir, una tecnología de persistencia de entre las que soporta Roo. Por defecto lo más habitual en Roo es usar JPA, aunque también viene preparado para bases de datos NoSQL como MongoDB.

Vamos entonces a configurar la capa de persistencia. Usaremos JPA con la implementación de Hibernate y como base de datos emplearemos hsqldb en memoria, para no depender de ningún servidor de base de datos externo.

```
roo> jpa setup --provider HIBERNATE --database HYPERSONIC_IN_MEMORY
```

El siguiente paso es crear las entidades del modelo del dominio. Crearemos una clase `Tarea` y luego le iremos añadiendo campos: titulo, fecha límite y prioridad. Ponemos todos los comandos seguidos, obviando la información que Roo nos da entre comando y comando de todos los archivos que va creando para dar soporte a la aplicación.

```
roo> entity jpa --class es.ua.jtech.domain.Tarea
~.domain.Tarea roo> field string --fieldName titulo
~.domain.Tarea roo> field date --fieldName limite --type java.util.Date
~.domain.Tarea roo> field number --type int --fieldName prioridad
```

**El archivo .log de Roo**

Cada uno de los comandos que tecleamos en el *shell* se guarda en un archivo llamado `log.roo` en el directorio del proyecto. Así, si queremos repetir los pasos para volver a crear de nuevo el mismo proyecto podemos ejecutar este fichero como un *script*, mediante el comando Roo del mismo nombre: `script --file log.roo`

Con todo esto hemos creado una entidad que automáticamente incorpora métodos de persitencia CRUD y algunos métodos de búsqueda básicos (buscar tareas por identificador - Roo crea una automáticamente, al no haberlo hecho nosotros -), Nótese que tras crear la entidad `Tarea`, el *prompt* de Roo cambia indicando que el foco de las siguientes operaciones está fijado en ella, es decir, que los campos se van a crear por



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

defecto en esa entidad. Roo abrevia el nombre del paquete de nivel superior de la aplicación (en nuestro ejemplo `es.ua.jtech`) con el símbolo de la tilde, ~. Podemos usar ese símbolo en nuestros comandos en lugar de teclear el nombre completo del paquete, aunque por desgracia no es una abreviatura muy práctica para el teclado en castellano. Para cambiar el foco a otra entidad podemos usar el comando `focus --class nombre_de_la_clase_cualificado`

Por otro lado, aunque aquí estamos creando los campos de la clase con comandos de Roo podríamos hacerlo también editando directamente el archivo `Tarea.java`, como haríamos si no estuviéramos usando Roo. Si abrimos este archivo, veremos que tiene un aspecto como el siguiente (obviando los imports):

```
@RooJavaBean
@RooToString
@RooJpaActiveRecord
public class Tarea {

    @Temporal(TemporalType.TIMESTAMP)
    @DateTimeFormat(style = "M-")
    private Date limite;

    private int prioridad;

    private String titulo;
}
```

Vemos que es código estándar con algunas anotaciones de Roo. Pero a poco que lo examinemos, es un poco sorprendente: ¿dónde están los *getters* y *setters*? ¿y el supuesto código CRUD que se había autogenerado? ¿no será un engaño todo esto de Roo ;)?. La respuesta es que para no interferir con los archivos que nosotros podemos modificar, Roo coloca todo este código generado en archivos aparte. El mecanismo que se usa es bastante ingenioso: estos archivos son lo que se denominan *Inter-Type Declarations* (ITD) de AspectJ, y son los ficheros con extensión `.aj` que Roo ha ido creando y actualizando tras cada uno de los comandos que hemos ejecutado sobre nuestra entidad. Por ejemplo el `Tarea_Roo_JavaBean.aj` contiene los *getters* y *setters*, el `Tarea_Roo_Jpa_ActiveRecord.aj` contiene el código de persistencia, ... En tiempo de compilación, el compilador de AspectJ "mezcla" (*weaves*, en argot AOP) este código generado con el código escrito por nosotros, produciendo ficheros .class convencionales con una combinación de nuestro código y el generado automáticamente.

Roo es lo suficientemente "inteligente" como para detectar en tiempo real los cambios que hacemos a nuestros .java y actuar en consecuencia. Por ejemplo si añadimos un campo para la descripción de la tarea

```
private String descripcion;
```

Al actualizar el archivo .java, y si seguimos teniendo abierto el shell de Roo, este detectará el cambio y actualizará los ITD pertinentes para incluir el nuevo campo.

Para terminar nuestra aplicación, vamos a añadirle una capa web y a generar



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

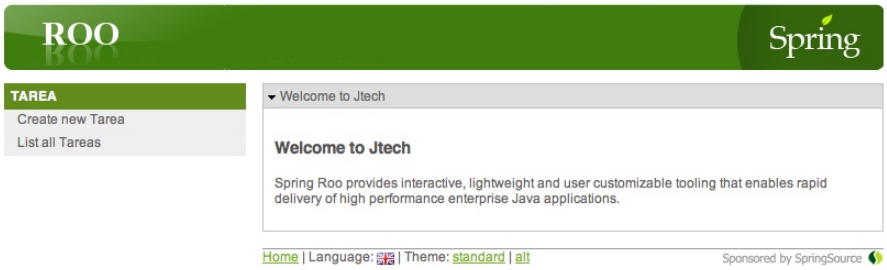
automáticamente la parte CRUD de la web. A esto último, en el ámbito de las herramientas web de desarrollo rápido, se le suele denominar *scaffolding*.

```
roo> web mvc setup
roo> web mvc all --package es.ua.jtech.web
```

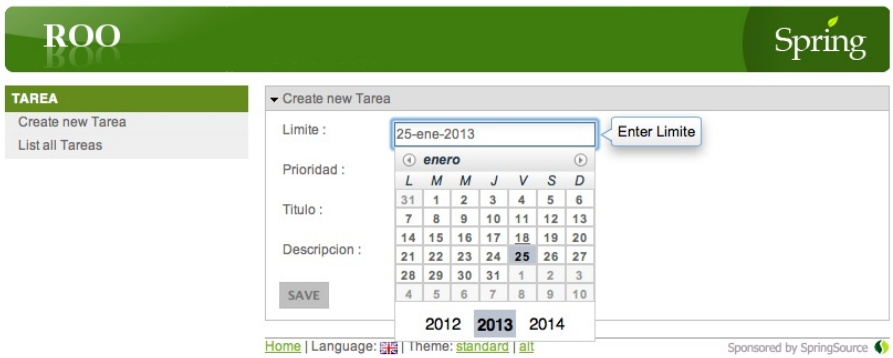
Finalmente, con el comando `quit` nos salimos del shell de roo y podemos proceder a desplegar nuestra aplicación. Al ser un proyecto web Maven, podemos desplegarla y ejecutarla automáticamente por ejemplo en el contenedor web Jetty con:

```
$ mvn jetty:run
```

Si es la primera vez que usamos Jetty desde Maven tardará un rato en bajarse las dependencias, pero el proceso es automático. Una vez aparezca en la consola el mensaje "Started Jetty Server", el servidor está listo, y podemos acceder a la aplicación abriendo la URL `http://localhost:8080/jtech` desde un navegador. Veremos el interfaz web generado por Roo, en el que podemos realizar las típicas operaciones CRUD.



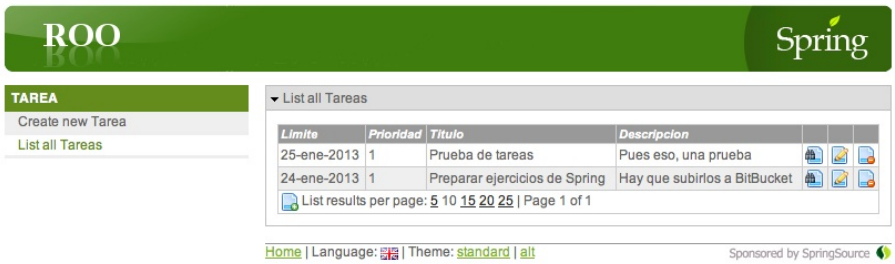

Pantalla inicial de la aplicación web




Creación de tarea



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.



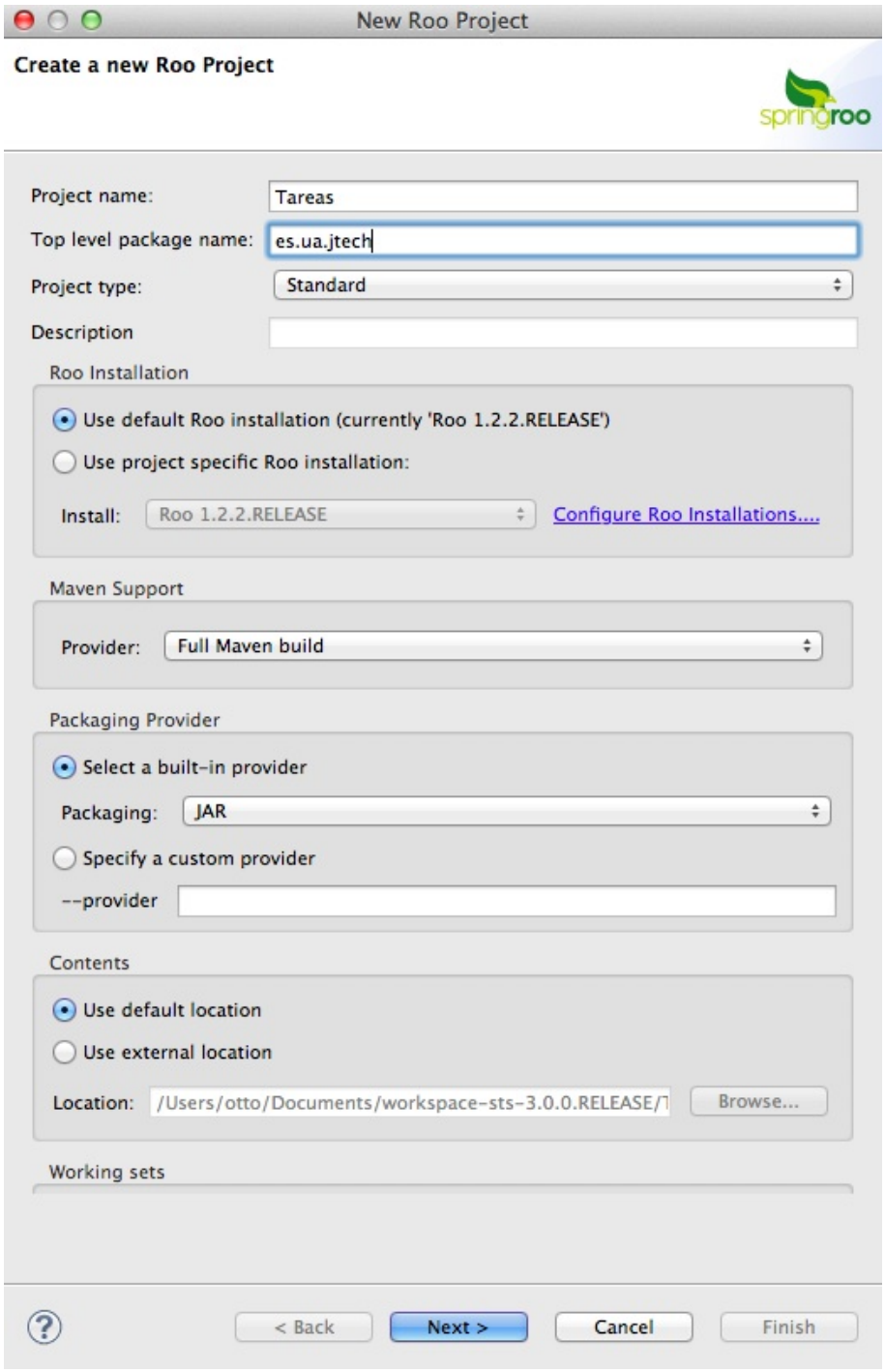

Lista de tareas

### 15.1.4. Trabajar con Roo desde SpringSource Tool Suite

Como no podía ser de otro modo, STS, el IDE de SpringSource, está preparado para trabajar con Roo. Entre los distintos tipos de proyectos que podemos crear está el "Spring Roo Project". El cuadro de diálogo para introducir los datos tiene muchas opciones pero podemos dejar la mayoría con los valores por defecto. Por ejemplo, para darle las mismas propiedades que el proyecto que hemos creado desde el *shell* de Roo, haríamos:



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.




Crear nuevo proyecto Roo en STS

Una vez creado el esqueleto del proyecto, aparecerá una ventana de Eclipse con un shell de Roo, que es el mismo que hemos usado antes desde fuera de STS. La diferencia más



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

aparente es que ya se está ejecutando por defecto en el directorio del proyecto y además que desde STS para autocompletar no se usa TAB sino Ctrl-Espacio, al estilo Eclipse. De hecho el propio shell de Roo lo detecta y cambia el mensaje en pantalla para indicarlo.

**Problema con el proyecto recién creado**

En la versión instalada en la máquina virtual hay un pequeño *bug*, ya que STS espera por defecto que haya una carpeta src/test/resources, mientras que Roo no la crea automáticamente. Para que desaparezca el error de "*Project 'XXXX' is missing required source folder: 'src/test/resources'*" tendremos que crear dichas carpetas manualmente.

Por defecto STS oculta los ITD con extensión .aj. Para verlos, en la ventana del `Package Explorer` hacemos clic sobre la pequeña flecha apuntando hacia abajo de la esquina superior derecha, seleccionamos `Filters...` y desmarcamos la opción de "Hide generated Spring ROO ITDs".

El desarrollo en Roo usando STS también se basa en comandos que hay que teclear en la terminal, con la comodidad adicional que nos ofrece el IDE al escribir código o ejecutar la aplicación (al crear Roo una aplicación web estándar, podemos ejecutarla desde STS con `Run As` tras añadirle la capa web). Además STS nos ofrece algunas funcionalidades adicionales, por ejemplo para refactorizar código, como veremos posteriormente.

### 15.1.5. Reglas generales sobre cómo editar código en una aplicación Roo

Al ser Roo una herramienta de generación de código, hay que llevar cuidado al modificar manualmente nuestro proyecto porque podemos eliminar o cambiar algo vital para Roo y "romper" el código. En la práctica, como todo el código generado por Roo está en los .aj esto no va a pasar con tanta facilidad como con otras herramientas de generación de código. Vamos a dar unas directrices generales de qué editar y cómo y qué no:

- *No se deben modificar/eliminar los archivos .aj* ya que es Roo el que los gestiona y modifica automáticamente.
- *Se puede modificar código de los .java generados por Roo*. Ya que Roo vigilará los cambios y modificará los .aj para que los reflejen. Como iremos viendo, algunos comandos de Roo lo único que hacen en realidad es introducir ciertas anotaciones propias del framework en el código Java. Estas anotaciones, de hecho, son las que hacen que Roo genere y actualice los .aj. De modo que también podemos introducirlas a mano en lugar de hacerlo mediante los comandos del shell de Roo. Evidentemente, esto quiere decir que si las eliminamos o modificamos inadvertidamente, la "magia" de Roo podría dejar de funcionar.
- En general *también se pueden editar los artefactos estándar*, por ejemplo el pom.xml para añadir dependencias o plugins, eso sí, llevando cuidado de no eliminar nada que haya introducido Roo.

## 15.2. La capa de acceso a datos



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

Vamos a ver en este apartado de manera un poco más detallada qué puede hacer Roo por nosotros al crear las entidades del dominio de nuestra aplicación. Como ya hemos visto en la introducción rápida este es el primer paso que se hace tras la creación de un proyecto.

Vimos que el primer paso era especificar qué proveedor de persistencia y qué base de datos íbamos a usar, con el comando `jpa setup`. En cualquier momento podemos cambiar la configuración de la base de datos y del proveedor de persistencia las veces que deseemos ejecutando de nuevo `jpa setup`. Así, en las primeras versiones de la aplicación podemos usar una base de datos `hsqldb` en memoria, como hicimos en el ejemplo de la aplicación de tareas, y luego cambiar a otra base de datos como MySQL u Oracle. En este último caso, lo normal será que tengamos que editar manualmente las propiedades de conexión con la base de datos, que Roo almacena en el archivo `src/main/resources/META-INF/spring/database.properties`. También se pueden mostrar/editar dichas propiedades desde el shell de Roo con el comando `properties`.

**Generar las entidades a partir de la BD**

Aquí vamos a crear y mantener las entidades manualmente, pero Roo también ofrece la posibilidad de generarlas automáticamente haciendo ingeniería inversa de la base de datos. No obstante esa forma de trabajar queda fuera del ámbito de estos apuntes. Se recomienda consultar la sección denominada DBRE (DataBase Reverse Engineering) de la documentación de Roo.

### 15.2.1. Active Record

Antes hemos comentado que Roo implementa el patrón de diseño *Active Record*, dando automáticamente a las entidades funcionalidad CRUD. Esta funcionalidad se implementa físicamente en el .aj cuyo nombre comienza con el de la entidad y acaba en `ActiveRecord`. El API es bastante sencillo, consistiendo en un conjunto de métodos muy similares a los de JPA. Por ejemplo:

```
//Create
Tarea t = new Tarea();
t.setLimite(new Date());
t.setPrioridad(1);
t.setTitulo("Probando");
t.persist();
//Read
for (Tarea t_i : Tarea.findAllTareas()) {
        System.out.println(t_i.getTitulo());
}
//Update
Tarea t2 = Tarea.findTarea(1L);
t2.setPrioridad(1);
t2.merge();
//Delete
Tarea t3 = Tarea.findTarea(2L);
t3.remove();
```

Como se ve, los métodos de este API son los mismos que los de JPA, pero los



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

implementa la propia clase de la entidad. El `find` de JPA se implementa como un método estático y se añade un `findAll`, un método de conveniencia para obtener todas las entidades sin necesidad de crear una query, también estático.

En *active record*, los métodos que sirven para buscar entidades especificando determinadas condiciones se denominan genéricamente *finders*. Además del `find` y del `findAll`, Roo puede generar por nosotros el código de muchos otros *finders* sin necesidad de escribir manualmente la *query* JPA-QL. Los *finders* codifican la *query* en el nombre del método según una serie de convenciones, que nos permiten buscar por un campo o por varios (findXXXByYYY, findXXXByYYYAndZZZ), o buscar aquellas entidades en las que un campo cumpla o no determinada condición (findXXXByYYYNotEquals,...). Algunos ejemplos concretos:

```
findTareasByTituloEquals(String titulo)
findTareasByTituloNotEquals(String titulo)
findTareasByTituloLike(String titulo)
findTareasByPrioridad(int prioridad)
findTareasByLimiteLessThan(Date limite)
findTareasByLimiteBetween(Date minLimite, Date maxLimite)
```

Para saber exactamente qué finders puede implementar Roo por nosotros, podemos usar el comando `finder list`, teniendo el foco en la entidad que nos interesa, lo que nos imprimirá en el shell una lista con todos los *finder* posibles para esta entidad. Ahora, para generar uno de ellos usamos el comando `finder add --finderName` seguido del nombre del *finder* a generar. Por ejemplo:

```
~.domain.Tarea roo> finder add --finderName findTareasByPrioridad
```

Como ya hemos dicho, podemos generar *finders* que combinen dos o más campos (en el nombre aparecerán unidos por `And` u `Or`,por ejemplo `findTareasByPrioridadAndLimiteLessThan`). Para listarlos, podemos añadir el parámetro `--depth` indicando cuántos campos queremos combinar. ¡Cuidado, fácilmente se producirá una explosión combinatoria en el número de *finders* posibles!.

### 15.2.2. Validación

Vamos a probar cosas un poco más sofisticadas que las que hicimos en la introducción "en diez minutos". Continuando con el ejemplo de las tareas, vamos a suponer ahora que tenemos proyectos y que cada proyecto está compuesto de una o varias tareas. Vamos a crear la entidad Proyecto con los campos título y prioridad. En el siguiente apartado nos ocuparemos de la relación uno a muchos entre proyecto y tareas. Por ahora vamos a decirle a Roo que queremos **validar** los campos título y prioridad: el título no puede estar vacío y la prioridad debe ser un entero de 1 a 3. La creación de la entidad la hacemos con los comandos:

```
roo> entity jpa --class es.ua.jtech.domain.Proyecto
~.domain.Proyecto roo> field string --fieldName nombre --notNull
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
~.domain.Proyecto roo> field number --fieldName prioridad --type int --min
1 --max 3
```

Lo que hace Roo es introducir anotaciones de JSR303 para validar los campos. También podríamos haber creado los campos e introducido las anotaciones manualmente en el código. Para más información sobre los distintos parámetros para la validación, consultar la referencia de Roo.

### 15.2.3. Pruebas

Para probar el código de persistencia y validación, podemos decirle a Roo que genere automáticamente un conjunto de tests. Dicho conjunto lo podemos generar con el comando `test integration`:

```
roo> test integration --entity es.ua.jtech.domain.Proyecto
```

También podíamos haber creado los test en el momento de crear la entidad con el comando `entity` añadiendo al final el *switch* `--testAutomatically`.

Una vez creados los test, veremos que se ha generado, entre otros, una clase de prueba `ProyectoIntegrationTest` en `src/test/java` que como no podía ser menos en Roo, no contiene en apariencia ningún test (solo uno que está vacío). En realidad los test de integración están en un .aj, y prueban todas las funcionalidades CRUD que Roo le proporciona a la entidad. Como el .aj se "mezcla" con el .java al compilarse, los test CRUD se ejecutarán automáticamente con Maven o bien podemos ejecutarlos manualmente con botón derecho y `Run As > JUnit Test`.

Por supuesto, podemos incluir nuestros propios test. Con el objeto de facilitarnos su implementación, Roo nos genera junto a los tests de CRUD una clase denominada `XXXDataOnDemand`, donde `XXX` es el nombre de la entidad. Esta clase nos proporciona bajo demanda nuevas instancias de la entidad con valores aleatorios, asegurándose de que cumplan las restricciones de validación.

```
@Test
public void testDeEjemplo() {
        ProyectoDataOnDemand pdod = new ProyectoDataOnDemand();
        Proyecto p = pdod.getNewTransientProyecto(1);
        p.persist();
        Proyecto p2 = Proyecto.findProyecto(p.getId());
        assertEquals(p.toString(),p2.toString());
}
```

### 15.2.4. Relaciones entre entidades

Por supuesto, tan importante como gestionar las entidades es gestionar las relaciones entre ellas. Al crear los campos, si especificamos que representan una relación con otra entidad introducirá las anotaciones JPA pertinentes. Como siempre, también podríamos escribir manualmente el código.



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

Vamos a modelar una relación uno a muchos entre proyecto y tareas. En el proyecto Añadiremos un campo "tareas" que referencie las tareas dentro de él. En Roo este tipo de campo se define como "set" (en Java será un Set<Tarea>). Por defecto se supone cardinalidad muchos a muchos, salvo que lo especifiquemos. Para el campo del lado "muchos a uno" se usa el tipo "reference".

```
roo> focus --class es.ua.jtech.domain.Proyecto
~.domain.Proyecto roo> field set --fieldName tareas --type Tarea
                                 --cardinality ONE_TO_MANY --mappedBy
proyecto
~.domain.Proyecto roo> focus --class es.ua.jtech.domain.Tarea
~.domain.Tarea roo> field reference --fieldName proyecto --type Proyecto
                                 --cardinality MANY_TO_ONE
```

Como es de suponer, el `--mappedBy` representa el atributo del mismo nombre de la anotación `@OneToMany`

## 15.3. La capa web

Roo puede generar el código de la capa web para diferentes *frameworks*. El usado por defecto es Spring MVC, que es el que vamos a ver aquí, pero también podemos emplear JSF, GWT u otros menos conocidos como Vaadin.

Lo primero es crear toda la configuración y los artefactos necesarios para la capa web. Entre otras cosas, si el proyecto era hasta ahora de tipo JAR, Roo cambiará la configuración a .WAR, creará el descriptor de despliegue web.xml, introducirá las nuevas dependencias en el pom.xml, creará el fichero de configuración de beans para la capa web... todo esto se consigue con el comando

```
roo> web mvc setup
```

### 15.3.1. Scaffolding

El *scaffolding* es una funcionalidad muy habitual en las herramientas de desarrollo rápido para web y consiste en que la herramienta nos va a generar todo lo necesario para un interfaz web CRUD de las entidades, desde las páginas JSP y el CSS hasta los *controllers* (en Spring MVC, en otros *frameworks* generará los artefactos equivalentes). Podemos decirle a Roo que queremos scaffolding para todas las entidades con

```
roo> web mvc all --package es.ua.jtech.web
```

Con `--package` estamos especificando el paquete donde queremos que Roo genere el código de los controllers. Por cada entidad se va a generar un controlador con métodos CRUD, accesibles mediante URLs al estilo REST. Así, por ejemplo, haciendo una petición GET a la URL `/tareas/` obtendríamos una lista de tareas, mientras que haciendo una petición POST a la misma URL crearíamos una tarea nueva, pasando los



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

datos como parámetros HTTP. Todos estos métodos se implementan en el archivo `XXXController_Roo_Controller.aj`, donde `XXX` es el nombre de la entidad. Se recomienda consultar la documentación de Spring Roo para más información sobre los métodos generados.

**Cuidado con los plurales**

Roo usa automáticamente el plural del nombre de la entidad para las URL asociadas. Por desgracia, usa las reglas del idioma inglés para generar los plurales. Eso ha funcionado bien con la entidad Tarea, pero Proyecto se asociará a "proyectoes" y no a "proyectos". Para solucionarlo, hay que editar la clase ProyectoController y cambiar la anotación `@RequestMapping` y el valor del atributo path de la anotación `@RooWebScaffold`.

Existe la posibilidad de hacer el *scaffolding* solo para una determinada entidad, con el comando `web mvc scaffold`. Primero tenemos que poner el foco en la entidad:

```
roo> focus es.ua.jtech.domain.Proyecto
~.domain.Proyecto roo> web mvc scaffold --class
es.ua.jtech.web.ProyectoController
```

Si no quisiéramos generar algún método CRUD podemos especificarlo con el parámetro `--disallowedOperations` y una lista de las operaciones separadas por comas (de entre 'create', 'update' o 'delete', por ejemplo `--disallowedOperations update,delete`. También podemos modificar la anotación `@RooWebScaffold` que Roo coloca en la clase controller, añadiendo como atributo nombre_operacion=false, por ejemplo:

```
@RooWebScaffold(path = "proyecto", formBackingObject = Proyecto.class,
                update = false, delete = false)
@RequestMapping("/proyectos")
@Controller
public class ProyectoController {}
```

El *scaffolding* es útil, pero hay muchos casos en los que queremos implementar operaciones distintas del simple CRUD. Para no tener que empezar a implementar el controlador desde cero, Roo nos ofrece la posibilidad de generar solo el esqueleto del controlador, con el comando "web mvc controller". Esto no solo genera el código (casi vacío) del controlador. Además, crea una página JSP asociada e incluye un enlace al controlador en el menú principal.

### 15.3.2. Clientes REST

Podemos serializar automáticamente las entidades en JSON, lo que nos será útil para los clientes REST. Para ello usaremos la secuencia de comandos:

```
roo> json all #da soporte JSON a todas las entidades
roo> web json setup #solo necesario si no hemos hecho ya web mvc setup
roo> web mvc json all #genera los controllers CRUD para json
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

El soporte para la serialización JSON lo ofrece la librería *open source* flexjson, por lo que para averiguar cómo se configura el JSON generado tendremos que acudir a su documentación.

Al igual que con otros comandos, lo que hacen los de json básicamente es incluir un par de anotaciones en las ciertas clases. Estas anotaciones le indican a Roo que debe crear o introducir ciertas funcionalidades en los ITD automáticamente. En concreto, "json all" lo que hace es anotar todas las entidades con `@RooJson`, y "web mvc json all" anota los controladores con `@RooWebJson`

### 15.3.3. Finders en la web

Podemos dar acceso en la capa web a los finders que hayamos creado en las entidades. Automáticamente Roo incluirá un método en el controlador para llamar al finder, lo mapeará con una URL determinada e incluso creará un formulario web para introducir los datos y un enlace en la página principal de la aplicación. El comando para dar acceso a todos los finders es "web mvc finder all", que introduce en los controladores la anotación `@RooWebFinder`.

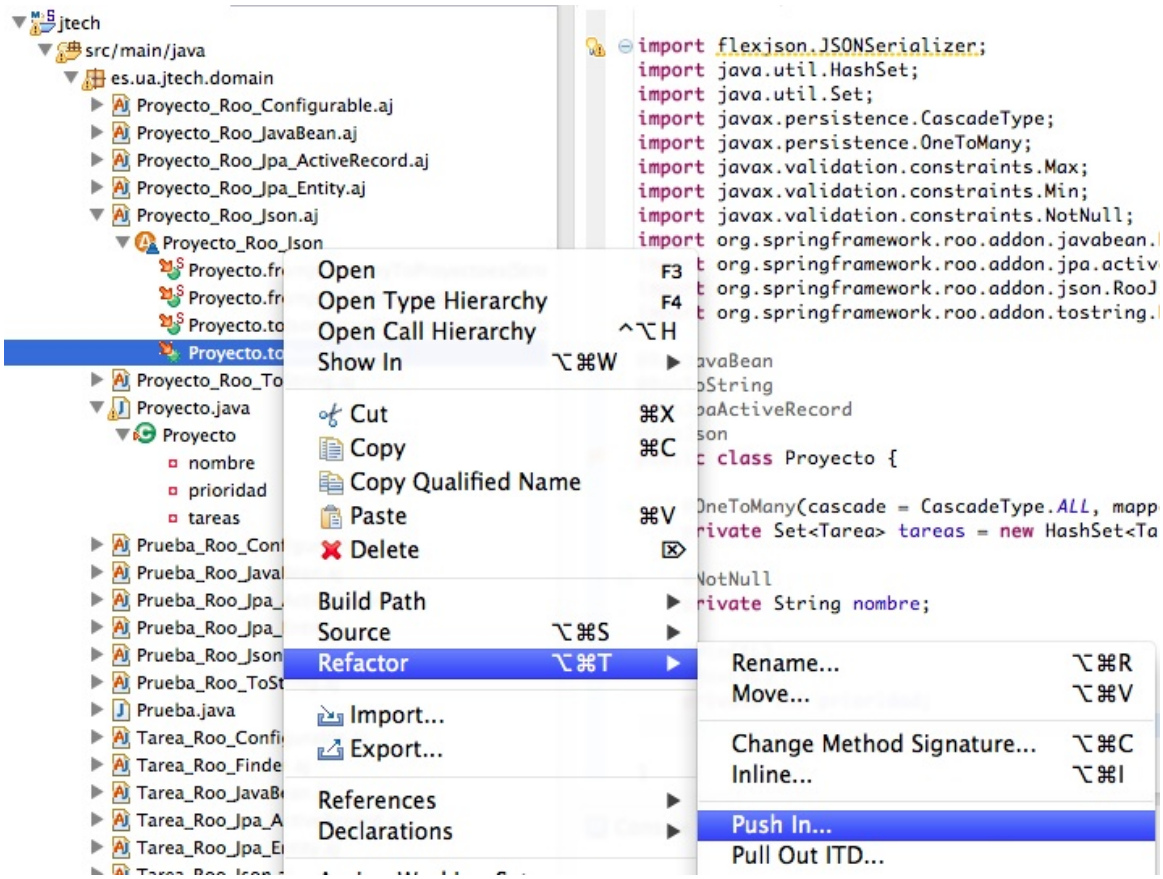
## 15.4. Refactorización del código Roo

En muchos casos, nos servirá aproximadamente el código generado por Roo pero querremos modificarlo para que se adapte a nuestras necesidades. Por ejemplo, al serializar los objetos en JSON, las relaciones uno a muchos no se serializan por defecto. Para cambiarlo tendríamos que cambiar código en un .aj. Pero ya hemos dicho que los .aj no se deberían editar, ya que Roo los gestiona y actualiza automáticamente y podría deshacer nuestros cambios de buenas a primeras. La solución es refactorizar el código para llevarnos el método que nos interese desde el .aj hasta el correspondiente .java. A esto se le denomina *push-in refactoring*.

El *push-in refactoring* está implementado en el IDE STS. Desde el package explorer seleccionamos el .aj del que nos interesa extraer el método, desplegamos sus métodos y pulsamos sobre el método objetivo con el botón derecho. Para refactorizar seleccionamos Refactor > Push In. Luego modificaríamos manualmente el código, ya en el .java, con la seguridad de que Roo no va a "meter las narices" en los cambios que hagamos.



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.



Push-in refactoring en STS

En el ejemplo que habíamos comentado, para serializar en JSON todas las tareas de un proyecto, haríamos un push-in del método toJson() del Proyecto_Roo_Json.aj. El código acabaría en Proyecto.java donde podríamos modificarlo, con la ayuda de la documentación de FlexJSON, para que se incluya el campo "tareas" en la serialización:

```
public String toJson() {
        return new JSONSerializer().include("tareas").serialize(this);
}
```

Aunque es de uso mucho menos habitual, la refactorización contraria al "push in" es "pull out" y consiste en que trasladamos código de una clase Java a un .aj. También está implementada en STS.

**Eliminar Roo de nuestro proyecto** se podría ver como "el push-in definitivo", o sea, trasladar el código de todos los .aj a las clases Java correspondientes. Además, eliminaríamos las anotaciones y algunas dependencias ya innecesarias del pom.xml. Seguiríamos estos pasos:

1. Hacer el push-in de todos los .aj: en STS seleccionaríamos el proyecto y con el botón



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

derecho, como antes, Refactor > Push In
2. Eliminar todas las anotaciones de Roo: es relativamente sencillo, ya que todas comienzan por `@Roo`.
3. Finalmente, eliminamos la dependencia de <artifactId>org.springframework.roo.annotations</artifactId> del pom.xml



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

# 16. Programación orientada a aspectos (AOP) en Spring

## 16.1. Introducción a la AOP

La programación orientada a aspectos (AOP - *Aspect Oriented Programming*) es un paradigma de programación que intenta formalizar y representar de forma concisa los elementos que son transversales a todo el sistema. En los lenguajes orientados a objetos, la estructura del sistema se basa en la idea de *clases* y *jerarquías de clases*. La herencia permite modularizar el sistema, eliminando la necesidad de duplicar código. No obstante, siempre hay aspectos que son transversales a esta estructura: el ejemplo más clásico es el de control de permisos de ejecución de ciertos métodos en una clase:

```
public class MiObjetoDeNegocio {
   public void metodoDeNegocio1() throws SinPermisoException {
        chequeaPermisos();
        //resto del código
        ...
   }

   public void metodoDeNegocio2() throws SinPermisoException {
        chequeaPermisos();
        //resto del código
        ...
   }

   protected void chequeaPermisos() throws SinPermisoException {
         //chequear permisos de ejecucion
      ...
   }
}
```

Como vemos, estructurando adecuadamente el programa se puede *minimizar* la repetición de código, pero es prácticamente imposible eliminarla. La situación se agravaría si además tuviéramos que controlar permisos en objetos de varias clases. El problema es que en un lenguaje orientado a objetos los aspectos transversales a la jerarquía de clases no son modularizables ni se pueden formular de manera concisa con las construcciones del lenguaje. La programación orientada a aspectos intenta formular conceptos y diseñar construcciones del lenguaje que permitan modelar estos aspectos transversales sin duplicación de código. En nuestro ejemplo, se necesitaría poder especificar de alguna manera concisa que *antes* de ejecutar *ciertos métodos* hay que llamar a *cierto código*.

En AOP, a los elementos que son transversales a la estructura del sistema y se pueden modularizar gracias a las construcciones que aporta el paradigma se les denomina **aspectos** (*aspects*). En el ejemplo anterior el control de permisos de ejecución, modularizado mediante AOP, sería un aspecto.

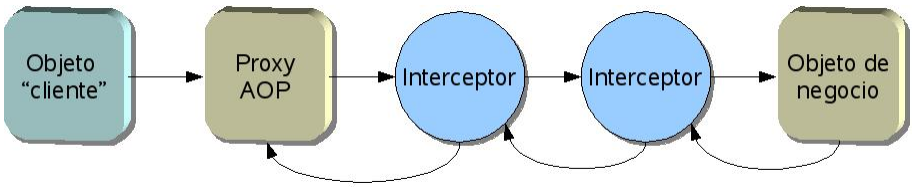
Un **consejo** (*advice*) es una acción que hay que ejecutar en determinado/s punto/s de un



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

código, para conseguir implementar un aspecto. En nuestro ejemplo, la acción a ejecutar sería la llamada a `chequeaPermisos()`. El conjunto de puntos del código donde se debe ejecutar un *advice* se conoce como **punto de corte** o *pointcut*. En nuestro caso serían los métodos `metodoDeNegocio1()` y `metodoDeNegocio2()`. Nótese que aunque se hable de "punto de corte" en singular, en general no es un único punto del código.

En muchos *frameworks* de AOP (Spring incluido), el objeto que debe ejecutar esta acción se modela en la mayoría de casos como un **interceptor**: un objeto que recibe una llamada a un método propio *antes* de que se ejecute ese punto del código. Los interceptores se pueden encadenar, si deseamos realizar varias acciones en el mismo punto, como puede observarse en la siguiente figura.




cadena de interceptores

Cuando algún objeto llama a un método que forma parte del pointcut, el *framework* de AOP se las "arregla" para que en realidad se llame a un objeto *proxy* o intermediario, que tiene un método con el mismo nombre y signatura pero cuya ejecución lo que hace en realidad es redirigir la llamada por una cadena de interceptores hasta el método que se quería ejecutar.

En algunas ocasiones nos interesará usar un interceptor para interceptar las llamadas a *todos* los métodos de una clase. En otras solo nos interesará interceptar *algunos* métodos. En Spring, cuando deseamos interceptar las llamadas solo a algunos métodos debemos definir un **advisor**, que será una combinación de *pointcut* (dónde hay que aplicar AOP) más *interceptor* (qué hay que ejecutar).

Aunque la terminología descrita en los párrafos anteriores será nueva para los no iniciados en AOP, las ideas resultarán bastante familiares a los que hayan trabajado con EJBs. En efecto, éstos funcionan de un modo muy parecido: el contenedor debe *interceptar* las llamadas a los métodos de negocio para poder ofrecer los servicios de seguridad, transaccionalidad y gestión de hilos. Los *puntos de corte* se especifican mediante los descriptores de despliegue y el `EJBObject` actúa de *proxy*. La diferencia básica es que AOP es más genérico que los EJBs, o dicho de otro modo, se podría considerar que *el mecanismo de provisión de servicios de los EJBs es un subconjunto de AOP*, ya que en EJBs:

- Los únicos objetos a los que se puede proporcionar servicios son por supuesto EJBs, pero no clases Java cualquiera (POJOs)
- Los puntos de corte que se puede especificar están limitados
- la implementación de los interceptores ya está hecha y no es posible cambiarla



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

## 16.2. AOP en Spring

Desde la versión 2, Spring ha mejorado mucho su módulo AOP de cara al usuario. La configuración necesaria se ha simplificado muy considerablemente. Además en lugar de usar una sintaxis propia de Spring se ha pasado a usar la de AspectJ. Siguiendo la filosofía de no reinventar la rueda, los desarrolladores de Spring han considerado que no era necesaria una sintaxis propia existiendo la de AspectJ, ampliamente probada en la práctica.

Hay que tener presente que *no es lo mismo usar la sintaxis de AspectJ que usar AspectJ* en sí. De hecho, lo recomendado en la documentación de Spring es usar la sintaxis de AspectJ pero con la implementación de AOP propia de Spring. Para casos en los que se necesite la potencia que puede dar AspectJ al completo, puede usarse su implementación sustituyendo a la de Spring, aunque cómo hacer esto último queda fuera del alcance de estas páginas.

Los apartados siguientes hacen un uso extensivo de la sintaxis de AspectJ, por lo que se recomienda al lector interesado que consulte su documentación, teniendo siempre en cuenta que Spring no soporta AspectJ al completo, sino solo un subconjunto.

### 16.2.1. Anotaciones vs. XML

Hay dos sintaxis alternativas para usar AOP en Spring 2.0. Una es mediante el uso de anotaciones en el propio código Java. La otra es con etiquetas en un fichero de configuración. El uso de XML es necesario cuando no podemos usar Java 5 (no tenemos anotaciones), no queremos tocar el código fuente o vamos a reutilizar la misma AOP en distintas aplicaciones y contextos. Sin embargo, usando anotaciones podemos encapsular el AOP junto con el código Java en un único lugar. En teoría este es el sitio en que debería estar si el AOP es un requisito de negocio que debe cumplir la clase.

En los ejemplos que siguen usaremos anotaciones. El lector puede consultar la excelente documentación de Spring para ver los equivalentes XML.

### 16.2.2. Añadir soporte para AOP a un proyecto

Para añadir soporte AOP a un proyecto Spring necesitaremos dos librerías: `aspectjweaver.jar`, y `aspectjrt.jar`, que podemos obtener de la distribución estándar de AspectJ o bien de la versión de Spring que viene con todas las dependencias incluidas. Además, si queremos usar AOP con clases que no implementen ningún `interface`, necesitaremos la librería CGLIB. En teoría es una buena práctica que todas nuestras clases de negocio implementen un interfaz, con lo que este último caso no debería darse demasiado.

**Proxies**



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

> Hay que recalcar de nuevo que la implementación AOP de Spring está basada en la idea de *proxy*. En Java existe un mecanismo estándar para generar automáticamente un *proxy* a partir de un interfaz, mecanismo que aprovecha Spring, pero no se puede generar a partir de una clase. Por ello en este caso se hacen necesarias herramientas de terceros como CGLIB.

Además debemos especificar que vamos a hacer uso de AOP en el XML de configuración de beans. La configuración se reduce a una sola etiqueta. En negrita se destaca la etiqueta y la definición del espacio de nombres necesaria para poder usarla.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
         xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

  <aop:aspectj-autoproxy/>
<beans>
```

## 16.3. Puntos de corte (pointcuts)

Como ya se ha dicho, un punto de corte o *pointcut* es un punto de interés en el código antes, después o "alrededor" del cual queremos ejecutar algo (un *advice*). Un *pointcut* no puede ser cualquier línea arbitraria de código. La versión actual de **Spring solo soporta puntos de corte en ejecuciones de métodos de beans**. La implementación completa de AspectJ permite usar también el acceso a campos, la llamada a un constructor, etc, aunque esto en AOP de Spring no es posible.

Es importante destacar que al definir un *pointcut* realmente no estamos todavía diciendo que vayamos a ejecutar nada, simplemente *marcamos un "punto de interés"*. La combinación de ***pointcut* + *advice*** es la que realmente hace algo útil. Por ello, los ejemplos dados en este apartado por sí solos no tienen demasiado sentido, no hay que intentar probarlos tal cual, aunque aquí los explicaremos aislados para poder describir con cierto detalle su sintaxis antes de pasar a la de los *advices*.

Lo que sigue es un conjunto de ejemplos que ilustran las opciones más comunes para *pointcuts*, no una referencia exhaustiva, que no tendría sentido estando ya la documentación de Spring y la de AspectJ para ello.

### 16.3.1. Expresiones más comunes

La expresión más usada en *pointcuts* de Spring es **`execution()`**, que representa la llamada a un método que encaje con una determinada signatura. Se puede especificar la signatura completa del método incluyendo tipo de acceso (public, protected,...), tipo de retorno, nombre de clase (incluyendo paquetes), nombre de método y argumentos.



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

Teniendo en cuenta:

- El tipo de acceso y el nombre de clase son opcionales, pero no así el resto de elementos
- Podemos usar el comodín * para sustituir a cualquiera de ellos, y también el comodín .., que sustituye a varios tokens, por ejemplo varios argumentos de un método, o varios subpaquetes con el mismo prefijo.
- En los parámetros, `()` indica un método sin parámetros, `(..)` indica cualquier número de parámetros de cualquier tipo, y podemos también especificar los tipos, por ejemplo `(String, *, int)` indicaría un método cuyo primer parámetro es `String`, el tercero `int` y el segundo puede ser cualquiera.

Por ejemplo, para especificar todos los métodos con acceso "public" de cualquier clase dentro del paquete `es.ua.jtech.aop` pondríamos:

```
execution( public * es.ua.jtech.aop.*.*(..))
```

Donde el primer * representa cualquier tipo de retorno, el segundo * cualquier clase y el tercer * cualquier método. Los .. representan cualquier conjunto de parámetros

Algunos ejemplos más de `execution()`

```
//Ejecución de cualquier getter (método público
//cuyo nombre comience por "get" y que no tenga parámetros).
execution(public * get*())
//Ejecución de cualquier método public de cualquier clase
//en el paquete es.ua.jtech o subpaquetes (fíjate en el "..")
execution(public * es.ua.jtech..*.*(..))
//Ejecución de cualquier método de cualquier clase
//en el paquete es.ua.jtech que devuelva void
//y cuyo primer parámetro sea de tipo String
//Se omite el modificador de acceso
execution (void es.ua.jtech.*.*(String,..))
```

**`within()`** permite especificar todos las llamadas a métodos dentro de un paquete o subpaquetes del mismo (usando el comodín .. al igual que en la sintaxis de `execution()`).

```
//Cualquier llamada a método dentro del paquete es.ua.jtech o subpaquetes
within(es.ua.jtech..*)
```

**`args()`** permite especificar el tipo deseado para los argumentos. No se suele usar tal cual, sino combinado con `execution` como un "truco" para darle nombre a los argumentos (ver el apartado siguiente).

```
//Cualquier método que tenga un solo parámetro y que implemente
Serializable
args(java.io.Serializable)
```

### 16.3.2. Combinar pointcuts



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

Se pueden combinar pointcuts usando los operadores lógicos &&, || y !, con el mismo significado que en el lenguaje C. Por ejemplo:

```
//todos los getters o setters de cualquier clase
execution (public * get*()) || execution (public void set*(*))
```

El operador && se suele usar en conjunción con `args` como una forma de "dar nombre" a los parámetros, por ejemplo:

```
execution (public void set*(*)) && args(nuevoValor)
```

Encajaría con un *setter* cualquiera, dándole el nombre *nuevoValor* al parámetro pasado al mismo. Veremos la utilidad de esto, cuando definamos `advices`, como método para acceder al valor del parámetro.

### 16.3.3. Pointcuts con nombre

Se le puede asignar un nombre arbitrario a un pointcut (lo que se denomina una *signatura*). Esto permite referenciarlo y reutilizarlo de manera más corta y sencilla que si tuviéramos que poner la expresión completa que lo define. La definición completa consta de la anotación `@Pointcut` seguida de la expresión que lo define y la signatura. Para definir la signatura se usa la misma sintaxis que para definir la de un método Java en un interfaz. Eso sí, el valor de retorno debe ser `void`. Por ejemplo:

```
@Pointcut("execution(public * get*())")
public void unGetterCualquiera() {}
```

Esta signatura se puede usar por ejemplo al combinar pointcuts:

```
@Pointcut("execution(public * get*())")
public void unGetterCualquiera() {}

@Pointcut("within(es.ua.jtech.ejemplo.negocio.*"))
public void enNegocio() {}

@Pointcut("unGetterCualquiera() && enNegocio()")
public void getterDeNegocio() {}
```

## 16.4. Advices

Con los *advices* ya tenemos la pieza del puzzle que nos faltaba para que todo cobre sentido. Un *advice* es algo que hay que hacer en un cierto punto de corte, ya sea antes, después, o "alrededor" (antes y después) del punto.

Los *advices* se especifican con una anotación con el pointcut y la definición del método Java a ejecutar (signatura y código del mismo). Como en Spring los puntos de corte deben ser ejecuciones de métodos los casos posibles son:



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

- Antes de la ejecución de un método (anotación `@Before`)
- Después de la ejecución normal, es decir, si no se genera una excepción (anotación `@AfterReturning`)
- Después de la ejecución con excepción/es (anotación `@AfterThrowing`)
- Después de la ejecución, se hayan producido o no excepciones (anotación `@After`)
- Antes y después de la ejecución (anotación `@Around`)

Un aspecto (*aspect*) es un conjunto de *advices*. Siguiendo la sintaxis de AspectJ, los aspectos se representan como clases Java, marcadas con la anotación `@Aspect`. En Spring, además, un aspecto debe ser un bean, por lo que tendremos que anotarlo como tal

```
import org.aspectj.lang.annotation.Aspect;
import org.springframework.stereotype.Component;

@Component
@Aspect
public class EjemploDeAspecto {

  //aquí vendrían los advices...
}
```

### 16.4.1. @Before

Esta anotación ejecuta un *advice* antes de la ejecución del punto de corte especificado. Por ejemplo:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class EjemploBefore {

  @Before("execution(public * get*())")
  public void controlaPermisos() {
    // ...
  }

}
```

Ejecutaría `controlaPermisos()` antes de llamar a cualquier *getter*.

### 16.4.2. @AfterReturning

Esta anotación ejecuta un *advice* después de la ejecución del punto de corte especificado, siempre que el método del punto de corte retorne de forma normal (sin generar excepciones). Por ejemplo:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterReturning;

@Aspect
public class EjemploAfterReturning {
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
  @AfterReturning("execution(public * get*())")
  public void log() {
    // ...
  }

}
```

Evidentemente para hacer *log* nos puede interesar saber el valor retornado por el método del punto de corte. Este valor es accesible con la sintaxis alternativa:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterReturning;

@Aspect
public class EjemploAfterReturning {

  @AfterReturning(
   pointcut="execution(public * get*())",
   returning="valor")
   public void log(Object valor) {
    // ...
  }
}
```

Al poner `Object` como tipo de la variable asociada al valor de retorno, estamos indicando que nos da igual el tipo que sea (incluso si es primitivo). Especificando un tipo distinto, podemos reducir el ámbito del *advice* para que solo se aplique a los puntos de corte que devuelvan un valor del tipo deseado.

### 16.4.3. @AfterThrowing

Esta anotación ejecuta un *advice* después de la ejecución del punto de corte especificado, si el método del punto de corte genera una excepción. Podemos tanto acceder a la excepción generada como restringir el tipo de las excepciones que nos interesan, usando una sintaxis como la siguiente:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterThrowing;

@Aspect
public class EjemploAfterThrowing {

  @AfterThrowing(
    pointcut="execution(public * get*())",
    throwing="daoe")
  public void logException(DAOException daoe) {
    // ...
  }
}
```

El ejemplo anterior indicaría que no hay que ejecutar el *advice* a menos que la excepción generada sea del tipo `DAOException`, y nos permite acceder a su valor a través del parámetro `daoe`.



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

### 16.4.4. @After

Esta anotación ejecuta un *advice* después de la ejecución del punto de corte especificado, genere o no una excepción, es decir, al estilo del `finally` de Java. Se usa típicamente para liberar recursos y otras tareas habituales para `finally`.

### 16.4.5. @Around

Esta anotación ejecuta parte del *advice* antes y parte después de la ejecución del punto de corte especificado. La filosofía consiste en que el usuario es el que debe especificar en el código del *advice* en qué momento se debe llamar al punto de corte. Por ello, el *advice* debe tener como mínimo un parámetro de la clase `ProceedingJoinPoint`, que representa el punto de corte. Llamando al método `proceed()` de esta clase, ejecutamos el punto de corte. Por ejemplo:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.ProceedingJoinPoint;

@Aspect
public class EjemploAround {

  @Around("execution(public * get*())")
  public Object ejemploAround(ProceedingJoinPoint pjp) throws Throwable {
        System.out.println("ANTES");
        Object valorRetorno = pjp.proceed();
        System.out.println("DESPUES");
        return valorRetorno;
  }
}
```

Hay que destacar varias cuestiones del código anterior. Como ya se ha dicho, cuando queremos llamar al punto de corte invocamos a `proceed()`. Además debemos devolver como valor de retorno del advice el devuelto por el punto de corte. Finalmente, si el método del punto de corte requiere parámetros, podemos pasarle un `Object[]`.

### 16.4.6. Acceder a los parámetros y otra información del punto de corte

Spring ofrece al *advice* acceso tanto a los parámetros del método del punto de corte como a información adicional sobre el mismo. Ya hemos visto cómo acceder al valor de retorno, en el ejemplo de `@AfterReturning`, y a la excepción lanzada en el caso del `@AfterThrowing`.

Para hacer accesibles al *advice* los argumentos del punto de corte se puede usar `args`. Por ejemplo:

```
//...
@AfterReturning("execution(public void set*(*)) && args(nuevoValor)")
public void log(int nuevoValor) {
```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

```
  // ...
}
//...
```

Con esto enlazamos el valor del argumento del punto de corte con la variable `nuevoValor`, y además al poner `int` como tipo de `nuevoValor` indicamos que solo queremos aplicar el *advice* si el argumento del punto de corte es `int`.

Otra forma de hacer lo anterior sería usar un punto de corte "con nombre":

```
//..
@Pointcut("execution(public void set*(*))&& args(nuevoValor)")
public void unSetter(int nuevoValor) {}

@AfterReturning("unSetter(nuevoValor)")
public void log(int nuevoValor) {
        //...
}
//..
```

Podemos acceder a información sobre el punto de corte en el primer parámetro del advice si es de tipo `JoinPoint`. Esta clase tiene métodos para obtener por ejemplo la signatura del método del punto de corte, o sus argumentos, etc.

```
@Before("execution(public void set*(*))")
public void beforeGetter(JoinPoint jp) {
        System.out.println("Antes de ejecutar " + jp.getSignature());
}
```

### 16.4.7. Más de un advice para el mismo punto de corte

Aunque no se ha dicho explícitamente hasta el momento, por supuesto se puede definir más de un *advice* que encaje con el mismo punto de corte. La pregunta surge entonces de forma natural: **¿cuál es el orden de aplicación de los *advices*?**.

**Para los *advices* especificados dentro del mismo aspecto**, se puede tomar como una regla básica que el orden de ejecución es el mismo que el de declaración. Es decir, que si declaramos varios *advices* de tipo *before*, se ejecutará primero el que primero aparezca declarado, y si declaramos varios de tipo *after* ocurrirá lo mismo (en realidad en el caso *after* si se piensa un poco se verá que el de mayor importancia es el que se ejecuta el último).

El caso en que tenemos *advices* definidos en aspectos distintos es más complejo, ya que en principio no está definida la precedencia por defecto y para especificarla hay que escribir algo de código Java. En concreto el aspecto debe implementar el interface `org.springframework.core.Ordered`. Remitimos al lector a la documentación de Spring para más información.

## 16.5. AOP "implícita" en Spring 3



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

La AOP constituye uno de los pilares en que se fundamenta Spring, incluso aunque nunca la usáramos en nuestros proyectos de modo explícito. A continuación resumimos alguno de los usos directos de AOP en el framework:

- **Transaccionalidad declarativa:** la gestión automática de las transacciones implica la intercepción AOP de los métodos marcados como transaccionales. El TransactionManager es el que intercepta los métodos y se encarga de crear nuevas transacciones, hacer rollback automáticos, etc. según sea necesario.
- **Seguridad:** Spring Security es un proyecto que intenta mejorar la seguridad declarativa estándar de JavaEE, haciéndola más potente sin aumentar la complejidad de uso. Mediante este proyecto se puede por ejemplo controlar de manera declarativa el acceso a los métodos de los beans, de manera similar a como se hace en Servlets 3.0:

  ```
  @RolesAllowed("ROLE_ADMIN")
  public void eliminarUsuario(Usuario usuario) {
        ...
  }
  ```

  Pero va mucho más allá, permitiendo el uso de expresiones en el lenguaje EL de Spring para evaluar los permisos de acceso. Por ejemplo, queremos que también el propio usuario pueda borrar su cuenta, además del administrador:

  ```
  @PreAuthorize("#usuario.login == authentication.name or
  hasRole('ROLE_ADMIN')")
  public void eliminarUsuario(Usuario usuario) {
        ...
  }
  ```

  `authentication.name` sería el login del usuario que se ha autentificado, y estamos intentando comprobar si corresponde con el del usuario que queremos borrar (el # nos da acceso al parámetro).
- **Spring Roo**: es un framework de desarrollo rápido de aplicaciones. Crea la estructura CRUD de la aplicación usando Spring en todas las capas (presentación, negocio y datos). Todo el código necesario se introduce de forma "no invasiva" usando AOP. Por ejemplo si decimos que queremos usar JPA en la capa de persistencia, Roo se encarga de convertir nuestra clase de dominio en una entidad JPA y de gestionar el Entity Manager y los métodos JPA básicos, pero todo este código no será visible directamente en nuestras clases y no nos "molestará". Todo esto se hace usando AspectJ, aunque es un uso más avanzado que el que nosotros hemos visto aquí.
- **Cache declarativa:** es una de las nuevas características incorporadas en la versión 3.1. Nos permite usar de manera transparente frameworks para gestión de cache ya conocidos y muy probados como Ehcache. Por ejemplo, podemos hacer que un método de un DAO cachee automáticamente los resultados sin más que usar la anotación correspondiente:

  ```
  @Cacheable("libros")
  public LibroTO getLibro(String isbn) {
     ...
  }
  ```



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

Si ya se ha llamado anteriormente al método con el mismo valor de "isbn", en lugar de ejecutar el método, se sacará el LibroTO de la cache. Lo que está pasando es que con AOP se está interceptando el método con @Before y comprobando si es necesario o no ejecutarlo.



Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.

Copyright © 2012-2013 Depto. Ciencia de la computación e IA All rights reserved.