La capa de negocio en Spring: el contenedor de beans

Índice

1. El framework Spring

Spring es un framework que se propone como alternativa al uso de EJBs y servidores de aplicaciones para el desarrollo de aplicaciones J2EE. Intenta basarse en una serie de "buenos principios" de desarrollo, como la inversión de control (Inversion of Control), programación orientada a aspectos (AOP) y otros, que permitan superar la excesiva complejidad a la que se han enfrentado tradicionalmente los desarrolladores J2EE.

El framework está organizado en 7 módulos diferentes, y diseñado de forma que se pueden usar todos los módulos o solo los que se necesiten:

- **El núcleo (core):** contiene las clases básicas, y el contenedor de beans, que veremos en este tema
- **Los módulos ORM y DAO** facilitan la implementación de DAOs, permitiendo integrar implementaciones alternativas de acceso a datos (por ejemplo JPA o JDBC) dentro del mismo framework. Los veremos en el tema 2.
- **El módulo AOP** proporciona soporte a la mayor parte de servicios que ofrece Spring a los objetos de negocio: transaccionalidad, seguridad,... Será el objeto del tema 3.
- **El módulo Web** representa el interfaz entre la capa de negocio y la de presentación en aplicaciones web. Puede usarse solo, con un framework MVC de terceros (p.ej. Struts) o bien con el MVC propio de Spring. Veremos una breve introducción en el tema 4.
- **El módulo Context** proporciona una forma de acceder a los beans de modo similar a como lo hace JNDI, además de dar soporte a tareas de comunicación diversas como mail, acceso a EJBs, etc.
- **El módulo JEE** permite la integración con diversos APIs JavaEE, como JMS, EJB, y el acceso remoto a beans gestionados por Sprin.

La versión actual de Spring en el momento de escribir estas páginas es la 2.5, que es bastante reciente (noviembre 2007), por lo que la bibliografía impresa sobre ella es prácticamente inexistente. La novedad básica de esta versión es la mejora de la configuración a través de anotaciones, y la adaptación a las nuevas versiones de diversos APIs, aunque es bastante similar a la anterior, 2.0, para la que sí hay bastante bibliografía disponible. La "generación" 2.x incorporó bastantes mejoras en cuanto a potencia y mecanismos de configuración más sencillos que las versiones 1.X. El framework dispone de una documentación excelente, accesible a través de su web, que hemos tomado como referencia básica para estos apuntes.

2. El papel del contenedor de beans

El módulo **Core** de Spring desempeña un papel equivalente al contenedor de EJBs de un servidor de aplicaciones. Instancia los objetos de negocio y permite controlar su ciclo de vida, proporcionando también servicios declarativos como por ejemplo la

transaccionalidad. Una diferencia básica es que el contenedor de beans de spring es un contenedor **ligero**, independiente del resto de servicios del *framework*. Es decir, no es más que un .jar que se puede desplegar en cualquier servidor web java, como por ejemplo Tomcat. Esto permite ahorrar costes y aligerar la ejecución del servidor, ya que habitualmente los servidores de aplicaciones son una cuestión "todo o nada" (ofrecen multitud de servicios útiles, pero tenemos que soportar la carga de todos aunque solo necesitemos unos pocos).

Otros puntos importantes en la filosofía del contenedor son su énfasis en la **inyección de dependencias** y en los beans al estilo **POJO**. En la "era" post EJB 3.0, esto no parece nada revolucionario, pero precisamente fue el éxito del propio Spring el que, en la época de EJB 2.x, demostró que estas ideas permitían hacer aplicaciones más ligeras, portables y *testables*.

3. Cómo definir e instanciar beans

Hasta Spring 2.5, lo habitual era almacenar la información sobre los beans en un fichero de configuración XML independiente del código fuente. En la versión 2.5 se introduce la posibilidad de usar anotaciones en el propio fuente. El uso de anotaciones presenta algunas ventajas frente al uso de XML:

- Los ficheros fuente se convierten en autocontenidos
- En general, es mucho menos farragoso escribir la especificación con anotaciones que el equivalente en XML.

no obstante el XML también tiene puntos fuertes:

- Al ser independiente del fichero fuente se pueden hacer cambios en los objetos de negocio sin necesidad de recompilación, ya que el XML se interpreta cuando arranca el contenedor. Esto incluye la posibilidad de especificar el valor inicial para las propiedades de los beans.
- Permite crear varios beans de la misma clase Java pero con características distintas (por ejemplo distinto ámbito o ciclo de vida).

De esta breve discusión queda claro que no hay una "forma mejor" de hacerlo, sino que dependerá de los requerimientos y las circunstancias particulares de nuestra aplicación. Aquí veremos los dos estilos de manera indistinta, aunque la gran mayoría de ejemplos existentes en la bibliografía impresa y en la web usan el XML. Por ello en cada caso primero explicaremos el XML y luego las anotaciones.

3.1. Definir beans

Supongamos que tenemos el siguiente POJO, que queremos convertir en un bean de Spring

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

public class GestorUsuarios {
   public UsuarioTO login(String login, String password) {
     ...
   }
}
```

Deberíamos definir un fichero XML como el siguiente:

```
<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-2.5.xsd">

  <bean id="gu" class="es.ua.jtech.spring.negocio.GestorUsuarios">
  </bean>

  <!-- si hubieran más beans, podríamos ponerlos aquí -->
  <bean id="miOtroBean" class="mipaquete.MiOtraClase">
  </bean>

  ...
</beans>
```

El atributo **id** especifica el nombre que vamos a darle al bean y con el que podremos acceder a él a través del contenedor.

Nota:

En Spring 2.X se usan *schemas* para la definir la gramática del XML, en lugar de los DTDs que se usaban en la 1.2 y anteriores. Aunque el DTD antiguo sigue siendo válido, se anima a los desarrolladores a usar el schema, ya que es mucho mejor desde el punto de vista de la capacidad de validación y además incorpora las mejoras de sintaxis de la versión 2.

3.2. Definir beans con anotaciones

El "estilo" XML es el que se usa por defecto en Spring. Si vamos a usar anotaciones debemos especificarlo en el fichero de configuración XML, lo que nos obliga a crearlo, como en el caso anterior, aunque no necesitaremos etiquetas **bean** y en su lugar usaremos anotaciones en el fuente.

```
<?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-2.5.xsd
```

```
        http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">

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

La etiqueta **<context:component-scan/>** 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**). En negrita en el XML se muestran además las líneas necesarias para que se reconozca el espacio de nombres "context", usado por dicha etiqueta.

Para definir un bean de Spring podemos usar varias anotaciones en el código fuente: 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
public class GestorUsuarios {
   public UsuarioTO login(String login, String password) {
     ...
   }
}
```

Nota:

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. La anotación @Repository sí tiene efecto sobre la transaccionalidad, aunque no en más cuestiones. No obstante, el equipo de desarrollo de Spring se reserva la posibilidad de añadir semántica a estas anotaciones en futuras versiones del *framework*.

Por defecto, el nombre del bean gestionado por Spring será el mismo que el de la clase pero sin "cualificar" y sin la inicial en mayúscula, en nuestro caso "gestorUsuarios". Podemos especificar el nombre indicándolo en la anotación

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

import org.springframework.stereotype.Service;

@Service("miGestor")
public class GestorUsuarios {
   public UsuarioTO login(String login, String password) {
     ...
   }
}
```

En el apartado siguiente veremos cómo acceder a un bean conociendo su nombre.

3.3. Instanciar beans

El que nos da acceso a un bean es el contenedor, por lo que nuestro primer problema es cómo arrancarlo. Esto es bastante sencillo: si estamos en una **aplicación web** (el caso que aquí nos ocupa), en el `web.xml` especificaremos la lista de ficheros XML con definiciones de beans y el contenedor web será el encargado de poner en marcha el contenedor de beans de Spring al arrancar la aplicación. Para ello se usa el mecanismo estándar de ejecutar código al arrancar una aplicación web: los *listener*. El `web.xml` quedaría así:

```
<?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 desempeña un papel "similar" al de JNDI en JavaEE. Dicha clase 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.*,
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);
  GestorUsuarios gu = (GestorUsuarios)
wac.getBean("gestorUsuarios");
%>
</body>
</html>
```

Donde suponemos que, o bien en el XML de definición de beans hemos definido uno con `id="gestorUsuarios"` de la clase `GestorUsuarios`, o bien que hemos usado anotaciones y hemos puesto la anotación `@Service` en dicha clase.

Nota:

Nótese que en este ejemplo no hemos usado *dependency injection* para acceder al bean, sino lo que se denomina *dependency lookup*, ya que necesitamos pedirle explícitamente a Spring que nos lo localice. Si en la capa web usamos el módulo web de Spring conseguiremos inyección de dependencias, como veremos en sesiones posteriores del módulo.

En Spring los beans por defecto son *singletons*, lo cual quiere decir que *todas* las llamadas en el contenedor a `getBean("gestorUsuarios")` devolverán una referencia *al mismo objeto* (salvo que tengamos más de un contenedor arrancado, en cuyo caso hay *una instancia por contenedor*). El ámbito *singleton* es adecuado para objetos de negocio sin estado y para DAOs, que tampoco suelen guardar estado. No obstante este no es ni mucho menos el único ámbito disponible en Spring. Más adelante veremos cómo especificar el ámbito del bean.

4. Definir dependencias entre beans

Cuando un bean hace uso de otros, en Spring se dice que tiene "colaboradores" (*collaborators*). Por ejemplo, supongamos que nuestro `gestorUsuarios` del ejemplo anterior necesita de otro bean que haga de DAO:

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

import es.ua.jtech.spring.datos.UsuariosDAO;
import es.ua.jtech.spring.dominio.UsuarioTO;

public class GestorUsuarios {
  private UsuariosDAO udao;

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

  public UsuarioTO login(String login, String password) {
     return udao.login(login, password);
  }
}
```

En Spring tenemos dos formas de definir cuáles son exactamente los beans colaboradores de uno dado. Se puede hacer por nombre del bean y por tipo (lo que se denomina *autowiring*). Por supuesto, cada una de estas dos formas a su vez se puede especificar en XML o con anotaciones.

4.1. Dependencias por nombre

Nótese que en el "estilo XML" de definir dependencias no se introduce ninguna referencia al API de Spring en el fuente, lo que es bueno para la portabilidad de nuestro código. A cambio de esta "libertad", necesitamos definir el *setter* público para que Spring pueda inyectar la dependencia. Cuando se arranca el contenedor de beans, éste va a instanciar el DAO que necesita el objeto de negocio y a pasarle la referencia llamando al método `setUdao`. Esta forma de inyectar dependencias se denomina *setter injection*.

El código Java del DAO sería algo como:

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

import es.ua.jtech.spring.dominio.UsuarioTO;

public class UsuariosDAO {
    public UsuarioTO login(String login, String password) {
        //aquí vendría la implementación JDBC, JPA o lo que sea
    }

}
```

Finalmente, en el fichero XML que define los beans, tenemos que especificar que nuestro bean de la clase `GestorUsuarios` tiene una propiedad llamada `udao` que es un bean de la clase `UsuariosDAO`:

```


<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-2.5.xsd">

    <!-- definimos un bean de la clase UsuariosDAO y le damos
nombre -->
    <bean id="miUsuariosDAO"
class="es.ua.jtech.spring.datos.UsuariosDAO">
    </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>
```

Al llamarse la propiedad `udao` Spring "sabe" por convenio que debe inyectar la dependencia llamando a un *setter* denominado `setUdao`.

En lugar de hacer una referencia al "bean dependiente" con el atributo `ref` podemos usar un "bean interno" (un *inner bean*) poniendo directamente un bean dentro de otro. Obsérvese que el bean interno no necesita nombre si nunca lo vamos a instanciar directamente (si llamamos a `getBean()` para instanciar un bean necesitamos que tenga un nombre, para pasarlo como argumento).

```
<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-2.5.xsd">

    <bean id ="miGestorUsuarios"
class="es.ua.jtech.spring.negocio.GestorUsuarios">
        <property name="udao">
            <bean class="es.ua.jtech.spring.datos.UsuariosDAO">
            </bean>
        </property>
    </bean>

</beans>
```

4.2. Dependencias por tipo (autowiring)

Se pueden omitir las referencias a los colaboradores en el fichero de configuración, y decirle a Spring que mediante *reflection* determine cuáles son las referencias necesarias. Esto se denomina **autowiring**. La forma más común es `byType`, que consiste en que Spring asigna el bean del tipo adecuado de entre todos los definidos. Siguiendo con el ejemplo anterior:

```
<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-2.5.xsd
       http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">


    <bean id="miUsuariosDAO"
class="es.ua.jtech.spring.datos.UsuariosDAO">
    </bean>

    <bean id ="miGestorUsuarios"
class="es.ua.jtech.spring.negocio.GestorUsuarios"
```

```
autowire="byType">
    </bean>

</beans>
```

Como puede verse, no hace falta referenciar al bean `miUsuariosDAO` en las propiedades de `miGestorUsuarios`. Al usar *autowiring*, Spring detecta automáticamente la referencia (ya que en la clase java hay un método `setUdao`) y la resuelve por la clase del objeto (ya que el parámetro del setter es un `UsuariosDAO`, y en el fichero de definición de beans hay uno definido de esta clase). Otro tipo de *autowiring* es `byName`, en el que si hay una propiedad de un bean llamada XXX (o sea, hay un getXXX/setXXX) se asocia automáticamente con un bean de Spring que tenga el mismo nombre. En general, se recomienda usar el `byType` porque es menos propenso a errores.

Si se usa *autowiring* es recomendable decirle a Spring que chequee que todas las propiedades de los *beans* se han resuelto correctamente. Para ello se usa el atributo `dependency-check`.

```
<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-2.5.xsd
       http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">


    <bean id="miUsuariosDAO"
class="es.ua.jtech.spring.datos.UsuariosDAO">
    </bean>

    <bean id ="miGestorUsuarios"
class="es.ua.jtech.spring.negocio.GestorUsuarios" autowire="byType"
          dependency-check="objects">
    </bean>

</beans>
```

De este modo, si al arrancar el contenedor no se resuelve la dependencia Spring dará un mensaje de error indicando la dependencia sin resolver. Sin este chequeo nos daríamos cuenta del problema demasiado tarde y de manera indirecta: cuando el `gestorUsuarios` intente acceder al DAO, generando una `NullPointerException`.

Con `dependency-check="objects"` especificamos que se chequeen solo las referencias a objetos, pero no así las propiedades "simples" (tipos primitivos y cadenas). Esto último se consigue poniendo como valor del atributo `all` (o `simple` solo para los tipos "simples").

4.3. Definición de dependencias con anotaciones

Spring admite el uso de la anotación `@Resource` para definir dependencia por nombre. Nótese que esta anotación es parte del estándar JSR-250 y no es exclusiva de Spring. La anotación se puede colocar en el *setter* o bien en el campo, por ejemplo:

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

import es.ua.jtech.spring.datos.UsuariosDAO;
import es.ua.jtech.spring.dominio.UsuarioTO;
import javax.annotation.Resource;
import org.springframework.stereotype.Service;

@Service
public class GestorUsuarios {
  @Resource(name="usuariosDAO")
  private UsuariosDAO udao;

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

  public UsuarioTO login(String login, String password) {
     return udao.login(login, password);
  }
}
```

El atributo `name` se refiere al nombre del bean, que recordemos que por defecto es el nombre de la clase con la inicial en minúscula y *sin* el nombre del paquete.Si no ponemos atributo `name` en la anotación, se buscará un bean con nombre igual al del campo (en este caso, `udao`).

El código de `UsuariosDAO` sería:

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

import es.ua.jtech.spring.dominio.UsuarioTO;
import org.springframework.stereotype.Repository;

@Repository
public class UsuariosDAO {
    public UsuarioTO login(String login, String password) {
        //aquí vendría la implementación del login...
    }

}
```

Se puede especificar el *autowiring* usando la anotación `@Autowired` en el código fuente, en el campo o bien en el *setter*:

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

```
import es.ua.jtech.spring.datos.UsuariosDAO;
import es.ua.jtech.spring.dominio.UsuarioTO;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class GestorUsuarios {
  //También valdría poner la anotación en el setter
  @Autowired
  private UsuariosDAO udao;

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

  public UsuarioTO login(String login, String password) {
     return udao.login(login, password);
  }
}
```

4.4. Múltiples candidatos para autowiring

Los ejemplos anteriores de autowiring (dependencias por tipo) no planteaban problemas, ya que solo existía *un único candidato posible* para resolver la dependencia. Pero ¿qué ocurre si existe más de un bean del mismo tipo? supongamos que hemos decidido convertir UsuariosDAO en una interfaz, `IUsuariosDAO`, y vamos a crear implementaciones alternativas de la misma. Usaremos anotaciones a lo largo del ejemplo.

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

import es.ua.jtech.spring.dominio.UsuarioTO;

public interface IUsuariosDAO {
    public UsuarioTO login(String login, String password);
}
```

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

import es.ua.jtech.spring.dominio.UsuarioTO;
import org.springframework.stereotype.Repository;

@Repository
public class UsuariosDAOJDBC implements IUsuariosDAO {
    public UsuarioTO login(String login, String password) {
        //Aquí vendría la implementación JDBC...
    }

}
```

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

```
import es.ua.jtech.spring.dominio.UsuarioTO;
import org.springframework.stereotype.Repository;

@Repository
public class UsuariosDAOJPA implements IUsuariosDAO {
    public UsuarioTO login(String login, String password) {
        //Aquí vendría la implementación JPA...
    }

}
```

Por supuesto, el `GestorUsuarios` ahora dependerá del interfaz `IUsuariosDAO`:

```
package es.ua.jtech.spring.negocio;
import es.ua.jtech.spring.datos.IUsuariosDAO;
import es.ua.jtech.spring.dominio.UsuarioTO;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
@Service
public class GestorUsuarios {
  @Autowired
  private IUsuariosDAO udao;

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

  public UsuarioTO login(String login, String password) {
     return udao.login(login, password);
  }
}
```

Pero aquí aparece el problema: ¿cuál de las dos clases que podría satisfacer la dependencia hay que usar?. Si intentamos ejecutar el código anterior tal cual en Spring, el contenedor dará un mensaje de error indicando que hay una ambigüedad en las dependencias. Para solucionar estas ambigüedades, podemos darle un nombre a cada `@Repository` y luego usar la anotación `@Qualifier` en conjunción con `@Autowired` para resolver la ambigüedad. Por ejemplo:

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

import es.ua.jtech.spring.dominio.UsuarioTO;
import org.springframework.stereotype.Repository;

@Repository("JDBC")
public class UsuariosDAOJDBC implements IUsuariosDAO {
    public UsuarioTO login(String login, String password) {
        //Aquí vendría la implementación JDBC...
    }

}
```

```
package es.ua.jtech.spring.negocio;
import es.ua.jtech.spring.datos.IUsuariosDAO;
```

```
import es.ua.jtech.spring.dominio.UsuarioTO;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
@Service
public class GestorUsuarios {
  @Autowired
  @Qualifier("JDBC")
  private IUsuariosDAO udao;

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

  public UsuarioTO login(String login, String password) {
     return udao.login(login, password);
  }
}
```

4.5. Constructor injection

La forma de *dependency injection* que venimos usando se denomina técnicamente **setter injection** , ya que el contenedor le proporciona a los objetos las dependencias que necesitan a través de métodos `setXXX`. Otra forma de conseguir esto sería "inyectar" las dependencias en el propio constructor. Aunque lo más habitual en Spring es usar *setter injection*, el *framework* también implementa esta última posibilidad. Lo único que hay que hacer en el fichero de definición de beans es sustituir las etiquetas `property` por una serie de `constructor-arg`.

```
...
<bean id="usuariosDAO"
class="es.ua.jtech.spring.datos.UsuariosDAO">
   <!--propiedades de este bean -->
   ...
</bean>
<bean id="gestorUsuarios"
class="es.ua.jtech.spring.negocio.GestorUsuarios">
   <constructor-arg>
      <ref bean="usuariosDAO"/>
   </constructor-arg>
</bean>
```

Por supuesto, para que esto funcione, debe haber un constructor de `GestorUsuarios` que tenga como único argumento un objeto de la clase `UsuariosDAO`. Por limitaciones del API de *reflection*, si se usan varios argumentos del mismo tipo en un constructor, no es posible resolver la ambigüedad por nombre del parámetro, de modo que en la definición del bean habrá que especificar el orden. Esto se hace con el atributo `index`.

5. Especificar las propiedades de un bean

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
    ...
}
```

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-2.5.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-2.5.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 -->
  </bean>
</beans>
```

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

5.1. Conversión de tipos automática

Como ya se ha comentado, Spring convierte automáticamente los valores puestos en el XML a numéricos, Strings y booleanos, pero podemos convertir los datos a cualquier clase Java sin más que definir y/o usar un `PropertyEditor`, que no es más que una clase que se encarga de realizar esta conversión. Spring incluye bastantes de estos conversores predefinidos, y si no nos bastan podemos definir los nuestros propios. La definición de conversores propios queda fuera del ámbito de estos apuntes. No obstante, aunque algunos de ellos ya están definidos, como el conversor a `Date`, no están registrados por defecto en el contenedor y no pueden usarse directamente. Vamos a ver aquí un ejemplo de cómo registrar el `CustomDateEditor`, que nos permitirá convertir a `Date` a partir de patrones cadena del estilo de los que usa la clase `DateFormat`. El registro de un nuevo `PropertyEditor` se hace a través de la clase `CustomEditorConfigurer`

Por ejemplo, supongamos que al bean `PrefsBusqueda` se le añade una propiedad de tipo `java.util.Date` llamada `desde` para poder especificar la fecha más antigua que puede

tener un documento que devuelva el buscador.

Si miramos el API de Spring veremos que la clase `CustomEditorConfigurer` tiene una propiedad de tipo `Map` en la que hay que colocar la clase destino de la conversión (en nuestro caso `java.util.Date`) y el `PropertyEditor` registrado (en nuestro caso `CustomDateEditor`). El API de este último nos dice qué parámetros requiere (consultar la documentación de Spring al efecto). El XML cambiaría como sigue:

```
...
<bean id="prefsBusqueda"
class="es.ua.jtech.spring.datos.PrefsBusqueda"
    p:maxResults="100" p:idioma="es" p:ascendente="true"
p:desde="10/02/2007">
</bean>
<bean id="miConfig"
class="org.springframework.beans.factory.config.CustomEditorConfigurer">
    <property name="customEditors">
        <map>
            <entry key="java.util.Date">
                <bean
class="org.springframework.beans.propertyeditors.CustomDateEditor">
                    <constructor-arg index="0">
                        <bean class="java.text.SimpleDateFormat">
                            <constructor-arg value="dd/MM/yyyy"/>
                        </bean>
                    </constructor-arg>
                    <constructor-arg index="1" value="false"/>
                </bean>
            </entry>
        </map>
    </property>
</bean>
...
```

6. 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 un solo `GestorPedidos` comentado pueda encargarse de todas las tareas de negocio relacionadas con pedidos, y por dar otro ejemplo si el bean representa un `DataSource`, aunque lo referenciemos en varios sitios en realidad siempre queremos acceder al mismo objeto.

Podemos usar otros ámbitos para el bean, a través del atributo `scope` de la etiqueta `bean`. Por ejemplo, para especificar que queremos una nueva instancia cada vez que se solicite

el bean, se usa el valor `prototype`

```
...
  <bean id="miBean" class="mipaquete.MiClase" scope="prototype">
     <!-- propiedades del bean y referencias a otros beans -->
     ...
  </bean>
...
```

6.1. Ámbitos especiales para aplicaciones web

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</liste
  </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:

```
...
  <bean id="miBean" class="mipaquete.MiClase" scope="session">
     <!-- propiedades del bean y referencias a otros beans -->
     ...
  </bean>
...
```

6.2. Dependencias entre beans con distinto ámbito

Cuando un bean depende de otro que tiene un ciclo de vida más corto se pueden plantear problemas. Por ejemplo, supongamos que un bean de la clase `DocsDAO` con ámbito `singleton` depende de otro de la clase `PrefsBusqueda` con ámbito de sesión HTTP.

```
...
<bean id="misPrefs" class="es.ua.jtech.spring.datos.PrefsBusqueda"
scope="session"/>

<bean id="miBuscador" class="es.ua.jtech.spring.datos.DocsDAO">
    <property name="prefs" ref="misPrefs"/>
</bean>
...
```

La configuración anterior tiene un problema que quizá no sea evidente a primera vista: como el bean `miBuscador` es un *singleton*, el contenedor lo instancia una sola vez, inyectando la dependencia de `misPrefs` *también una sola vez*. Pero nosotros no queremos eso, queremos que `misPrefs` se conserve durante la duración de la sesión HTTP y que si ésta se invalida se instancie *un nuevo* `misPrefs`.

Spring soluciona este problema con el uso de lo que se denomina un *proxy AOP*, cuyo funcionamiento veremos en sesiones posteriores. Baste saber por ahora que es un objeto que se "interpone" entre los dos beans y que se encarga de manera "inteligente" de gestionar el ciclo de vida del bean con ámbito más corto, creando nuevas instancias de él cuando sea necesario. El bean `miBuscador` permanece ajeno al hecho de que trata no con el verdadero `misPrefs` sino con un proxy, que toma sus peticiones y las pasa al verdadero destinatario. En realidad, si nos ceñimos simplemente a la sintaxis de Spring, el uso de este proxy AOP no puede ser más sencillo:

```
...
<bean id="misPrefs" class="springbeans.PrefsBusqueda"
scope="session"/>
    <aop:scoped-proxy/>
</bean>
<bean id="miBuscador" class="springbeans.Buscador">
    <property name="prefs" ref="misPrefs"/>
</bean>
...
```

Por defecto, Spring usa una librería "open source" denominada CGLIB para generar los *proxies*, por lo que deberemos incluirla en el CLASSPATH. Las librerías estándar de Java permiten generar *proxies* pero solo funcionan con interfaces, no con clases directamente. Si nuestros beans son clases que implementan un determinado interfaz, no necesitamos CGLIB. En ese caso, en la etiqueta `aop:scoped-proxy` pondríamos el atributo `proxy-target-class="false"`.

6.3. Definición del ámbito con anotaciones

La anotación `@Scope` nos permite definir el ámbito de un bean. Por ejemplo:

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

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

@Scope("prototype")
@Service
public class GestorUsuarios {
    ...
```

Si tenemos dependencias entre beans de distinto ámbito, necesitaremos usar el "truco" del *proxy* (ver apartado anterior). En el XML tendremos que especificarlo:

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

Donde estamos diciendo que necesitamos generar proxies de clases (es decir, necesitamos la librería CGLIB) Si todos nuestros beans implementan alguna interfaz podemos "librarnos" de CGLIB poniendo el atributo `scoped-proxy="interfaces"`

6.4. Notificaciones del ciclo de vida

En algunos casos puede ser interesante llamar a un método del bean cuando éste se inicializa o destruye. Para ello se pueden usar dos atributos en la definición: `init-method` y `destroy-method`. Por ejemplo:

```
...
<bean id="miBuscador" class="springbeans.Buscador"
init-method="inicializa" destroy-method="destruye">
</bean>
...
```

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.

7. Acceso a recursos JNDI con beans de Spring

Un bean *singleton* es perfecto para representar una referencia a un recurso JNDI como un DataSource. Aunque en versiones anteriores la forma de asociar un bean a un recurso JNDI era más compleja, la sintaxis se ha simplificado mucho con la introducción del 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-2.5.xsd
            http://www.springframework.org/schema/jee
http://www.springframework.org/schema/jee/spring-jee-2.5.xsd">
...
</beans>
```

Hacer por ejemplo que un DataSource cuyo nombre JNDI es `jdbc/MiDataSource` sea un

bean de Spring es muy sencillo con la etiqueta `jee:jndi-lookup`

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

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/`