Introducción a Spring

Índice

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 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.

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() {
     ...
   }
   ...
}
```

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.

2.1. Solicitarle beans al contenedor

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. 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);
  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.

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 {
   ...
}
```

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`.

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) {
      ...
   }
   ...
}
```

2.4. Control 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 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.

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,...).

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
  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 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.

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
@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`.

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"
@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.

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, 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.

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.

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
         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
    ...
}
```

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 -->
    ...
   </bean>
```

```
</beans>
```

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

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 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.