1.- ¿POR QUÉ SPRING?

CASO DE USO

En el transcurso del desarrollo de un proyecto es muy común encontrarnos con las siguientes situaciones:

- Los requerimientos cambian constantemente; el cliente, al cual hemos presentado el trabajo actual quiere cambios que debemos aplicar.
- Los componentes deben ser testeables; siguiendo la metodología TDD queremos probar cada componente en el momento de su desarrollo y no esperar a la prueba de la aplicación final.
- Queremos una arquitectura flexible; hace unos años no pensábamos en tener clientes móviles y la pregunta que nos hacemos todos es que nos deparará el futuro. Nuestra aplicación debe estar preparada para adoptar esos cambios.

Nuestro reto es encontrar un framework que nos facilite el desarrollo y sobre todo evitar un problema: el **acoplamiento**.

Vamos a poner un ejemplo para explicar el concepto de acoplamiento.

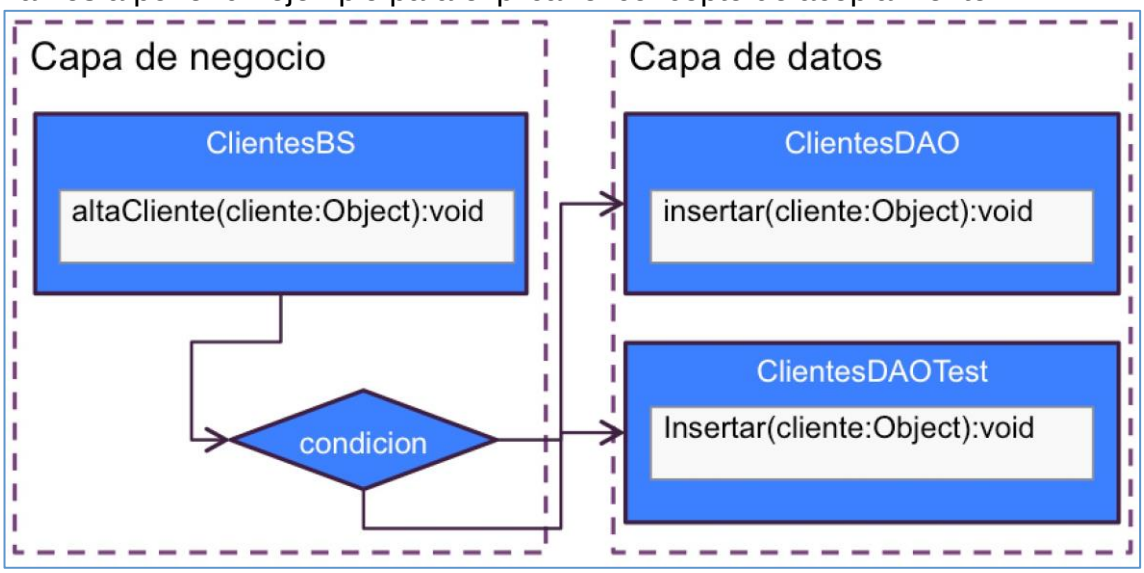


Gráfico 1. Acoplamiento en la capa de negocio.

Como vemos en la imagen anterior, si tenemos dos clases DAO en la capa de persistencia y debemos instanciar una de ellas desde nuestra clase de negocio, delegamos en una condición.

La pregunta que se puede hacer es: Que ocurre si en el futuro tenemos una tercera clase DAO? En ese caso debemos modificar la clase de negocio para que la tenga en cuenta en la condición.

En este caso podemos decir que existe acoplamiento entre las clases ya que se genera una fuerte dependencia entre ellas.

Como evitarlo? Se nos pueden ocurrir varias ideas: uso de interfaces, patrón Factory.

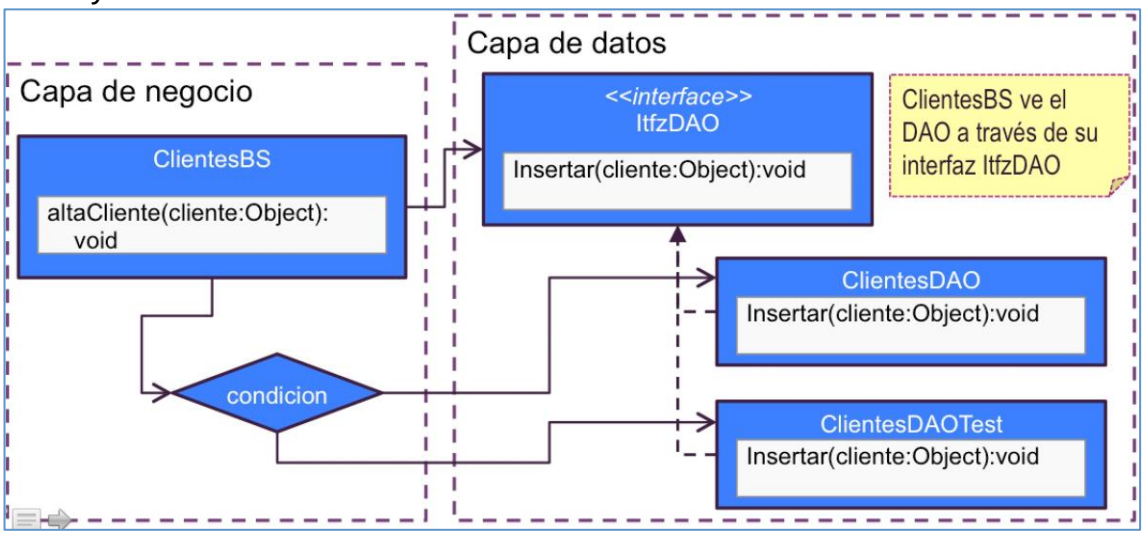


Gráfico 2. Uso de interfaces para evitar acoplamiento

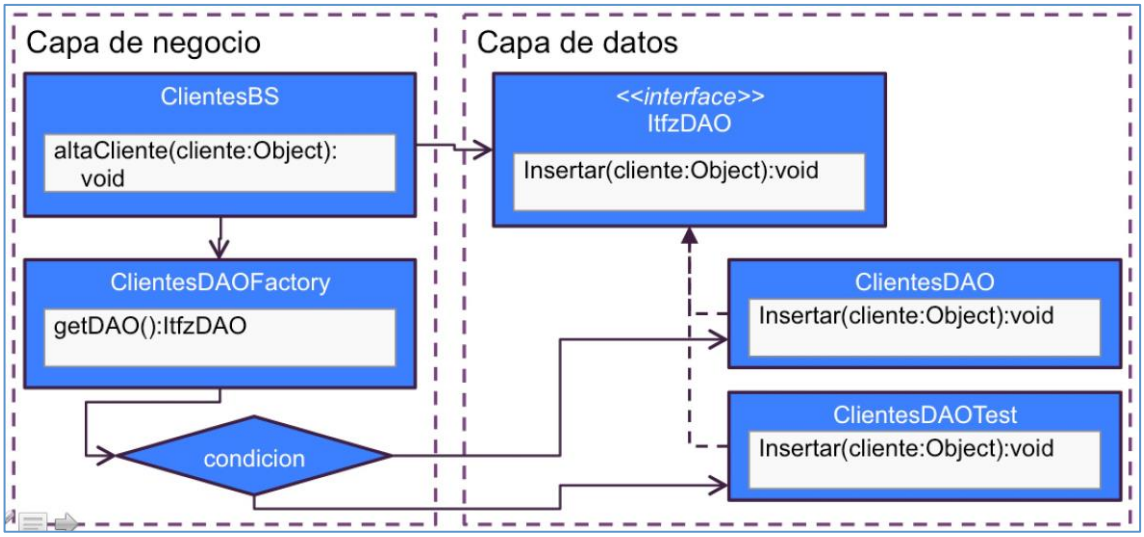


Gráfico 3. Patrón Factory

Finalmente nos damos cuenta que hemos ocultado el problema pero no lo hemos resuelto. Lo único que nos funcionaría sería aislar la creación del objeto y luego

inyectarlo en la clase de negocio. Casualidad? Eso es justo lo que nos proporciona Spring.

BENEFICIOS DE SPRING

¿Porque se utiliza tanto Spring? ¿Es una moda?. Desarrollar aplicaciones utilizando Spring presenta los siguientes beneficios:

- Simplificar el desarrollo en Java. Spring es un framework no intrusivo, esto quiere decir que nuestras clases serán simples clases Java (POJO´s) no teniendo que heredar de otras clases, ni implementar otras interfaces propias de Spring.
- Evitar el acoplamiento. Spring utiliza la inyección de dependencias. No crearemos el objeto sino que esperaremos a que nos lo faciliten.
- Reutilización de objetos. Mas adelante veremos como los beans creados por Spring se mantienen en un contenedor y de esta forma podremos reutilizarlos.
- Reducir el código reutilizable mediante aspectos y plantillas. Tomando como ejemplo JDBC vemos la cantidad de código que se repite para lanzar dos queries muy parecidas. Spring soluciona esto utilizando plantillas.

ARQUITECTURA DEL FRAMEWORK

A diferencia de otros frameworks como Hibernate, JSF, Struts2, …etc. Spring es un framework que no actúa en una sola capa sino que podemos utilizarlo en toda la aplicación. Esto hace que se distribuya en módulos siendo el imprescindible Spring Framework.

La siguiente figura nos muestra como está estructurado el framework.

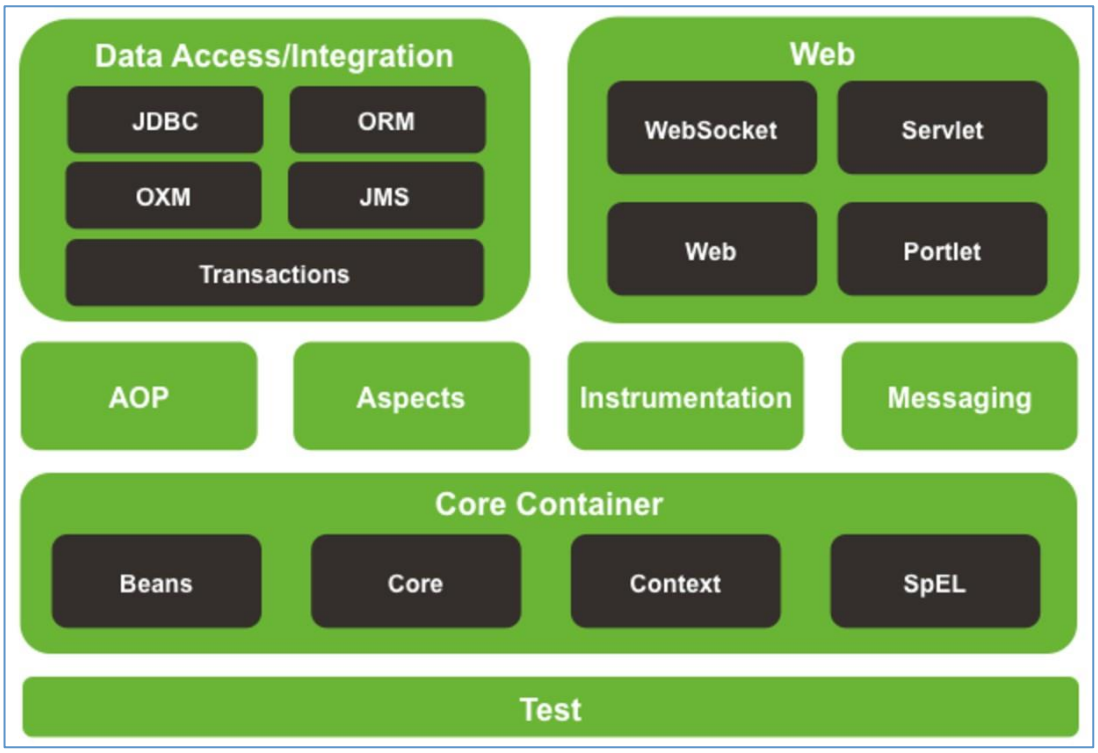


Gráfico 4. Estructura del Framework

Spring está organizado en aproximadamente 20 módulos agrupados en:

- Core Container
- Data Access/Integration
- Web
- AOP and Instrumentation
- Test

CORE CONTAINER

- Modulo Core; es el núcleo de Spring
- Modulo Beans; junto al modulo Core recoge todos los fundamentos de Spring, incluyendo IoC (Inversión de control) y DI (Inyección de dependencias)
- Modulo Context; añade soporte para la internacionalización, propagación de eventos, carga de recursos
- Modulo Expression Language; lenguaje de expresiones de Spring.

DATA ACCESS/INTEGRATION

- Modulo JDBC; Acceso a base de datos mediante JDBC
- Modulo ORM; Acceso a base de datos a través JPA, Hibernate, Ibatis,..etc.
- Modulo OXM; soporta el mapeo Objeto/XML como por ejemplo JAXB.
- Modulo JMS; Producir y consumir mensajes asíncronos.
- Modulo Transaction; Para manejar transacciones programáticas o declarativas.

WEB

- Modulo Web; Funcionalidades básicas de aplicaciones web.
- Modulo Web-Servlet; más conocido como Spring MVC • Modulo Web-Struts; permite integrar Struts 2.0 con Spring 3.0.
- Modulo Web-Portlets; integración de Portlets.

AOP AND INSTRUMENTATION

- Modulo AOP; Programación Orientada a Aspectos.
- Modulo Instrumentation; implementación del cargador de clases para ser utilizado en determinados servidores de aplicaciones.

TEST

- Modulo Test; permite probar componentes Spring con JUnit.

Todos los paquetes en desuso, y muchas clases y métodos en desuso se han eliminado con la versión 4.0.

2.- CONCEPTOS FUNDAMENTALES DE SPRING

INYECCION DE DEPENDENCIAS

Un bean es una instancia creada desde una clase, almacenada en el contenedor, que puede ser inyectado en otro bean.

La inyección de dependencias evita crear el objeto y asignarlo a una propiedad. Esto generaría una fuerte dependencia y el problema de acoplamiento que ya hemos visto.

Sin embargo Spring permite inyectar un bean residente en el contenedor en la propiedad de otro bean.

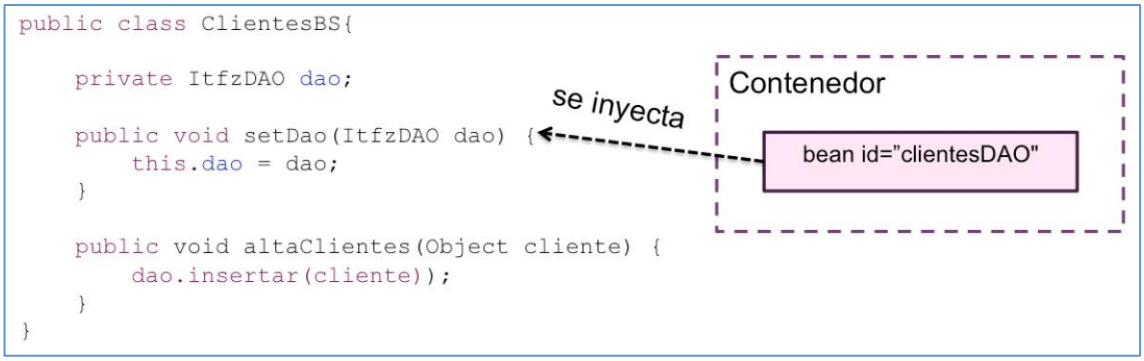


Gráfico 5. Inyección de dependencias

Actualmente existen 3 formas de trabajar con Spring:

- A través de XML
- Uso de Anotaciones
- Configuración con clases Java

CONTENEDOR DE BEANS

La filosofía de Spring es la reutilización de objetos (en adelante beans). Se trata de crear un objeto y almacenarlo en un contenedor para su posterior uso.

En una aplicación desarrollada con Spring, los beans (objetos de aplicación) van a residir dentro del contenedor de beans.

Dicho contenedor se encuentra en el núcleo (Core).

El contenedor utiliza DI (Inyección de Dependencias) para crear los beans, conectarlos con otros, configurarlos y administrar su ciclo de vida.

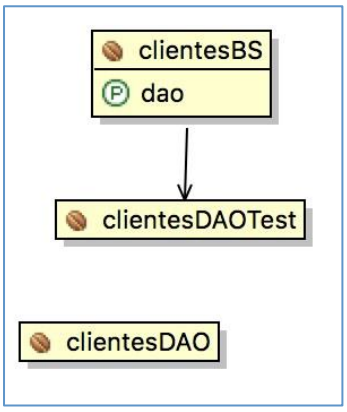


Gráfico 6. Beans en el contenedor

En adelante, se utiliza indistintamente el nombre contenedor, core-container, Spring IoC container o contexto de Spring para referirse a éste

APPLICATIONCONTEXT Y FACTORYBEAN

Spring construye el core-container en un objeto del tipo BeanFactory, a partir de un archivo de configuración típicamente llamado applicationContext.xml, del cual se puede extraer los objetos llamados "bean" ya configurados para utilizarlos.

Spring provee una interfaz genérica extendida de BeanFactory, llamada ApplicationContext, que es la que se utiliza.

ClassPathXmlApplicationContext es una implementación que permite obtener el archivo de configuración desde el classpath de la aplicación.

La interfaz BeanFactory provee todo el mecanismo de configuración para manejar los objetos del contenedor.

La interfaz ApplicationContext agrega integración con otras funcionalidades, como el manejo de aspectos, manejo de recursos, eventos, y características especiales, por lo que es la que se utiliza en la práctica.

CONTEXTOS DE SPRING

Para crear el contexto de Spring, lo podemos hacer por medio de 5 clases:

- ClassPathXmlApplicationContext; carga la definición de contexto a partir de un archivo XML situado en el classpath.

- FileSystemXmlApplicationContext; carga la definición de contexto a partir de un archivo XML situado en el sistema de archivos.
- XmlWebApplicationContext; carga la definición de contexto a partir de un archivo XML situado en una aplicación web.
- AnnotationConfigApplicationContext; carga la definición de contexto a partir de una clase de configuración basada en Java.
- AnnotationConfigWebApplicationContext; carga la definición de contexto web a partir de una clase de configuración basada en Java.

```
ApplicationContext context = new
        ClassPathXmlApplicationContext("archivo.xml");
```

```
ApplicationContext context = new
        FileSystemApplicationContext("C:/archivo.xml");
```

```
ApplicationContext context = new
        AnnotationConfigApplicationContext("paquete.ClaseConfig.class");
```

3.- CONFIGURACION CON XML

La configuración por XML es la más antigua de las tres. Hasta la versión 2.5 no se pudieron utilizar anotaciones por lo cual es bastante común que nos encontremos con un montón de proyectos donde se sigue utilizando.

Si optamos por esta forma de trabajar necesitamos incorporar a nuestro proyecto el archivo applicationContext.xml. No es obligatorio que se llame así, también lo reconoceréis como config.xml, context.xml, springContext.xml.

applicationContext.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://www.springframework.org/schema/beans
5      http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">
```

Gráfico 7. applicationContext.xml

DECLARAR UN BEAN

Ahora ya estamos listos para declarar nuestros beans. El espacio de nombres beans nos permite incorporar la etiqueta bean con los atributos id (identificador) y class (clase a instanciar).

```
<!-- ClientesDAO clientesDAO = new ClientesDAO(); -->
<bean id="clientesDAO" class="com.grupoatrium.persistencia.ClientesDAO" />
```

Gráfico 8. Declaración de un bean

Tengamos en cuenta lo siguiente:

- El identificador es único.
- Se utiliza para referenciar un bean desde otro.
- También se utiliza para recuperar un bean desde el contenedor.
- Convención de nombres para el identificador: aunque no hay una convención fija, suele utilizarse la interfaz o clase, cambiando la primera letra a minúscula.
- El nombre de la clase que se indica en el atributo class debe ser un qualified name (paquete.clase).

NAME Y ALIAS

En el caso que se quiera utilizar otros nombres, por ejemplo para que sean utilizados externamente, se pueden definir a través del "name":

```
<!-- El name es un nombre alternativo al id del bean -->
<bean id="clientesDAO" class="com.grupoatrium.persistencia.ClientesDAO"
    name="real"/>
```

Gráfico 9. Uso de name

En ese caso, el bean se puede referenciar de una de las dos formas: por su id o por su name.

También se puede definir uno o más alias asociados al identificador:

```
<bean id="clientesDAOTest" class="com.grupoatrium.persistencia.ClientesDAOTest" />

<!-- El alias es un nombre alternativo al id -->
<alias name="clientesDAOTest" alias="DAOprueba"/>
```

Gráfico 10. Uso de alias

En ese caso, el bean se puede referenciar de la forma:

```
<bean id="clientesBS" class="com.grupoatrium.negocio.ClientesBS">
    <property name="dao" ref="DAOprueba" />
</bean>
```

Gráfico 11. Inyectar el bean por su alias

INYECCION DE PROPIEDADES

Una vez creado un bean, podemos asignar sus propiedades. Para ello se utilizarán los métodos setxxx().

Un bean de Spring puede tener atributos de tipo básico, los cuales se pueden definir desde Spring. En este caso, se utiliza el "value" del tag <property>:

Utilizamos el atributo ref para asignar una referencia de otro bean a la propiedad.

```
<!-- ClientesBS clientesBS = new ClientesBS();
     clientesBS.setDAO(clientesDAOTest) -->
<bean id="clientesBS" class="com.grupoatrium.negocio.ClientesBS">
    <property name="dao" ref="clientesDAOTest" />
</bean>
```

Gráfico 12. Atributo ref

A la hora de inyectar el valor a la propiedad podemos proceder de dos formas:

- Delegar en spring la detección de tipo: en el xml, el valor de la propiedad está en formato texto. Si la propiedad es de tipo simple (string, número, boolean), y también de algunos otros tipos que pueden interpretar valores (class, resource), entonces el valor es interpretado (parsed) e inyectado al bean.

```
<bean id="empleado1" class="com.grupoatrium.modelo.Empleado">
    <property name="numeroEmpleado" value="127" />
    <property name="nombre" value="Juan" />
    <property name="apellido" value="Lopez" />
    <property name="jefe" value="false" />
    <property name="sueldo" value="27000" />
</bean>
```

Gráfico 13. Spring detecta el tipo de la propiedad

- Especificar el tipo del atributo.

```
<bean id="empleado2" class="com.grupoatrium.modelo.Empleado">
    <property name="numeroEmpleado">
        <value type="int">128</value>
    </property>
    <property name="nombre">
        <value type="java.lang.String">Maria</value>
    </property>
    <property name="apellido">
        <value type="java.lang.String">Sanchez</value>
    </property>
    <property name="jefe">
        <value type="boolean">true</value>
    </property>
    <property name="sueldo">
        <value type="double">56525.45</value>
    </property>
</bean>
```

Gráfico 14. Especificamos el tipo de la propiedad

En cuanto a la inyección de dependencias:

- Se realiza a través del setter del atributo definido en el "name" del tag <property>. La propiedad podría incluso llamarse de otra forma.
- Inyecta el bean referenciado por su id.

REF LOCAL Y REF BEAN

Mas adelante veremos que lo habitual es trabajar con más de un archivo xml en la aplicación. En este caso podemos utilizar lo siguiente: ref local:

El propio XML valida que el nombre de la referencia exista dentro del archivo. Esta opción está solo disponible hasta versiones Spring 3.

```
<!-- Buscara el bean empleado1 en este mismo archivo -->
<ref local="empleado1" />
```

Gráfico 15. Ref local

ref bean:

Equivalente a utilizar ref dentro del tag property.

```
<!-- El bean empleado2 puede estar en otro archivo -->
<ref bean="empleado2" />
```

Gráfico 16. Ref bean

PROPIEDADES DE TIPO COLECCIÓN

No siempre vamos a trabajar con datos simples como propiedades, también podemos asignar propiedades de tipo Colección como List, Array, Set, Map o Properties.

```
public class Empresa {

    private List<Empleado> empleados;
    private Set<String> proyectos;
    private Empleado[] equipos;
    private Map<String, Empleado> jefesProyecto;
    private Properties emails;
```

Gráfico 17. Propiedades de tipo collection

LIST Y ARRAYS

Al igual que utilizamos la etiqueta <ref> también podemos utilizar <value>, <bean> y <null>

```
<bean id="empresa" class="com.grupoatrium.modelo.Empresa">
    <property name="empleados">
        <list value-type="com.grupoatrium.modelo.Empleado">
            <ref local="empleado1" />
            <ref bean="empleado2" />
            <!-- Bean interno -->
            <bean id="empleado3" class="com.grupoatrium.modelo.Empleado">
                <property name="numeroEmpleado" value="129" />
                <property name="nombre" value="Pedro" />
                <property name="apellido" value="Arias" />
                <property name="jefe" value="true" />
                <property name="sueldo" value="67000" />
            </bean>
        </list>
    </property>
```

Gráfico 18. Inyección de List

```
<property name="equipos">
    <array>
        <ref bean="empleado1" />
        <ref bean="empleado2" />
    </array>
</property>
```

Gráfico 19. Inyección de Array

SET

Al igual que List lo podemos utilizar para definir un array. Podemos utilizar <value>, <bean> y <null>

```
<property name="proyectos">
    <set value-type="java.lang.String">
        <value>Mercadona</value>
        <value>BBVA</value>
        <value>Gas Natural</value>
    </set>
</property>
```

Gráfico 20. Inyección de Set

MAP

En lugar de <key> podemos utilizar <key-ref> y en lugar de <value> la alternativa es <value-ref>

```
<property name="jefesProyecto">
    <map>
        <entry key="Mercadona" value-ref="empleado1" />
        <entry key="BBVA" value-ref="empleado2" />
    </map>
</property>
```

Gráfico 21. Inyección de Map

PROPERTIES

La clase Properties es prácticamente lo mismo que un Map con la excepción que solo soporta tipos String para claves y valores.

```
<property name="emails">
    <props>
        <prop key="Juan">juan@gmail.com</prop>
        <prop key="Maria">maria@gmail.com</prop>
    </props>
</property>
```

Gráfico 22. Inyección de Properties

otra forma de utilizar propiedades

```
<property name="emails">
    <value>
        Juan=juan@gmail.com
        Maria=maria@gmail.com
    </value>
</property>
```

Gráfico 23. Otra forma de inyectar Properties

INYECCION DE DEPENDENCIAS A TRAVES DEL CONSTRUCTOR

CONSTRUCTOR SIN ARGUMENTOS

Cuando declaramos un bean en el archivo applicationContext.xml Spring lo creará utilizando el constructor sin argumentos. De ahí la necesidad de que este exista en nuestra clase.

```
<!-- ClientesDAO clientesDAO = new ClientesDAO(); -->
<bean id="clientesDAO" class="com.grupoatrium.persistencia.ClientesDAO" />
```

Gráfico 24. Uso de constructor sin argumentos

CONSTRUCTOR CON ARGUMENTOS

También podemos utilizar constructores con argumentos para dar valores iniciales a nuestros beans o incluso realizar inyección de dependencias.

También podemos utilizar argumentos de constructor de tipo colección como ya vimos anteriormente en la inyección por propiedades.

```
<!-- Empleado empleado3 = new Empleado(129, "Antonio",
    "Hernandez", false, 19000); -->
<bean id="empleado3" class="com.grupoatrium.modelo.Empleado">
    <constructor-arg value="129" />
    <constructor-arg value="Antonio" />
    <constructor-arg value="Hernandez" />
    <constructor-arg value="false" />
    <constructor-arg value="19000" />
</bean>
```

Gráfico 25. Uso de constructor con argumentos

RESOLVER AMBIGÜEDADES

Dado que en Java esta permitida la sobrecarga de constructores se nos puede presentar el caso que Spring no sabe exactamente a que constructor nos estamos refiriendo ya que por inferencia de tipos habría varias posibilidades.

Para solventar esta situación podremos ser mas específicos utilizando los atributos: type, index y name.

Atributo type

En este caso estamos indicando el tipo del argumento del constructor.

```
<bean id="empleado4" class="com.grupoatrium.modelo.Empleado">
    <constructor-arg type="int" value="129" />
    <constructor-arg type="java.lang.String" value="Antonio" />
    <constructor-arg type="java.lang.String" value="Hernandez" />
    <constructor-arg type="boolean" value="false" />
    <constructor-arg type="double" value="19000" />
</bean>
```

Gráfico 26. Resolver ambigüedades con type

Atributo index

Indicamos la posición del argumento en el constructor.

```
<bean id="empleado5" class="com.grupoatrium.modelo.Empleado">
    <constructor-arg index="0" value="129" />
    <constructor-arg index="1" value="Antonio" />
    <constructor-arg index="2" value="Hernandez" />
    <constructor-arg index="3" value="false" />
    <constructor-arg index="4" value="19000" />
</bean>
```

Gráfico 27. Resolver ambigüedades con index

Atributo name

Indicamos el nombre del argumento del constructor.

```
<bean id="empleado6" class="com.grupoatrium.modelo.Empleado">
    <constructor-arg name="numeroEmpleado" value="129" />
    <constructor-arg name="nombre" value="Antonio" />
    <constructor-arg name="apellido" value="Hernandez" />
    <constructor-arg name="jefe" value="false" />
    <constructor-arg name="sueldo" value="19000" />
</bean>
```

Gráfico 28. Resolver ambigüedades con name

BEANS ANONIMOS

Cuando declaramos un bean sin asignarle un identificador, este se conoce como un bean anónimo.

```
<!-- Bean anonimo -->
<bean class="com.grupoatrium.modelo.Empleado">
    <property name="numeroEmpleado" value="130" />
    <property name="nombre" value="Alfonso" />
    <property name="apellido" value="García" />
    <property name="jefe" value="false" />
    <property name="sueldo" value="25000" />
</bean>
```

Gráfico 29. Bean anónimo

El nombre del bean será el nombre cualificado de la clase seguido de #0 para la primera instancia y así los irá numerando Spring de forma incremental.

Paquete.clase#0

BEANS INTERNOS

Un bean puede referenciarse indicando su clase directamente en la referencia. Esto se conoce como un inner bean, por su similitud con una inner class en Java.

```
<property name="empleados">
    <list value-type="com.grupoatrium.modelo.Empleado">
        <ref local="empleado1" />
        <ref bean="empleado2" />

        <!-- Bean interno -->
        <bean id="empleado3" class="com.grupoatrium.modelo.Empleado">
            <property name="numeroEmpleado" value="129" />
            <property name="nombre" value="Pedro" />
            <property name="apellido" value="Arias" />
            <property name="jefe" value="true" />
            <property name="sueldo" value="67000" />
        </bean>
    </list>
</property>
```

Gráfico 30. Bean interno

En este caso, la clase es instanciada al momento de ser inyectada. Si la misma clase es inyectada en dos beans distintos como inner bean, se instancia dos veces

BEANS ABSTRACTOS

Si hacemos uso del atributo abstract=”true” significará que el bean es abstracto y que no será instanciado por Spring, luego no existirá en el contenedor.

```
<!-- Declarar un bean plantilla con las propiedades iniciales del empleado -->
<!-- Un bean abstracto no se puede recuperar del contenedor -->
<bean id="plantilla" class="com.grupoatrium.modelo.Empleado" abstract="true">
    <property name="jefe" value="false" />
    <property name="sueldo" value="22000" />
    <property name="skill" value="sin determinar" />
    <property name="proyecto" value="staff" />
</bean>
```

Gráfico 31. Bean abstracto

Si intentamos recuperar un bean abstracto del contenedor nos devolverá una excepción.

Este tipo de bean lo utilizamos como plantilla y así poder aplicar herencia.

HERENCIA

Si hay un concepto que conocemos muy bien en Java es la herencia. Nos han enseñado la necesidad de reutilizar código creando una clase que extiende otra.

Spring da un paso mas en cuanto a herencia y en este caso vamos a reutilizar los beans. Concretamente vamos a heredar las propiedades de otro bean.

El atributo parent en la declaración de un bean indica el id del bean a heredar.

```
<!-- Creamos el bean heredando de la plantilla y asi reutilizamos sus propiedades -->
<bean id="empleado1" class="com.grupoatrium.modelo.Empleado" parent="plantilla">
    <property name="numeroEmpleado" value="127" />
    <property name="nombre" value="Juan" />
    <property name="apellido" value="Lopez" />
</bean>
```

Gráfico 32. Herencia

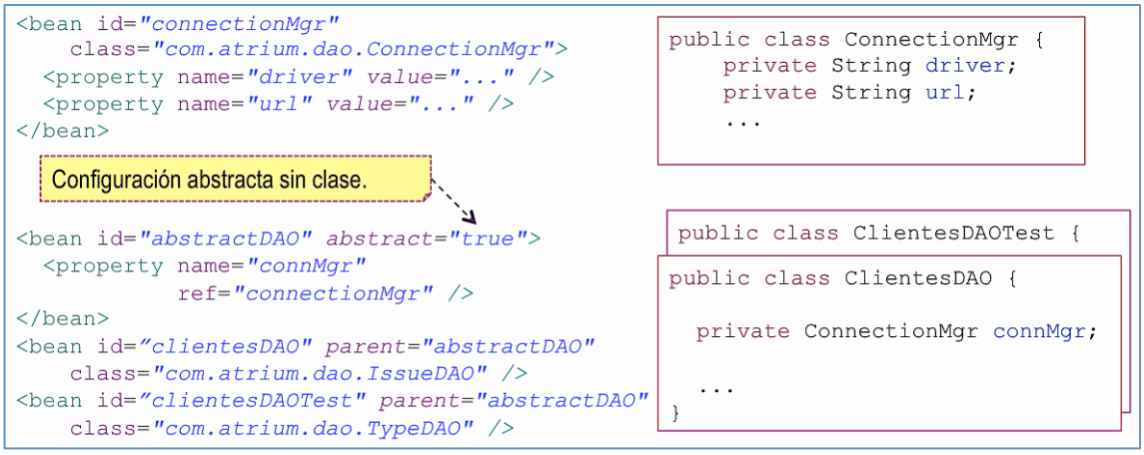
HERENCIA CON CLASES ABSTRACTAS

Cuando se hereda una configuración, debe existir coherencia con herencia de clases, es decir, la clase debe extender de la clase declarada en la configuración abstracta, si la hay.

Es recomendable intentar utilizar siempre una clase en la configuración abstracta. La clase no necesita ser abstracta, aunque es recomendable si no debe ser instanciada.

HERENCIA CON CLASES ABSTRACTAS SIN CLASE ASOCIADA

No necesariamente la configuración abstracta debe tener una clase asociada. Podría utilizarse sólo para centralizar configuración en Spring. En ese caso, los atributos que se heredan deben estar en las clases.




```
<bean id="connectionMgr"
    class="com.atrium.dao.ConnectionMgr">
  <property name="driver" value="..." />
  <property name="url" value="..." />
</bean>

<bean id="abstractDAO" abstract="true">
  <property name="connMgr"
            ref="connectionMgr" />
</bean>
<bean id="clientesDAO" parent="abstractDAO"
    class="com.atrium.dao.IssueDAO" />
<bean id="clientesDAOTest" parent="abstractDAO"
    class="com.atrium.dao.TypeDAO" />
```

```
public class ConnectionMgr {
    private String driver;
    private String url;
    ...
```

```
public class ClientesDAOTest {
public class ClientesDAO {

  private ConnectionMgr connMgr;

  ...
}
```

Gráfico 33. Herencia sin clase asociada

OTRAS FORMAS DE CREAR BEANS

Que ocurre cuando queremos utilizar una clase definida como un patrón Singletón? En este caso no podemos utilizar el constructor ya que este es privado.

En Java nos encontramos con este tipo de situaciones donde el constructor es privado pero nos proporcionan un método alternativo que es el que devuelve la instancia de la clase.

FACTORIA ESTATICA

Cuando el método alternativo al constructor es estático hablamos de una factoria estática.

```
public class Contabilidad {

    // Propiedad que almacena la unica instancia de la clase
    private static Contabilidad instance = new Contabilidad();

    // Constructor es privado para evitar la creacion de otras instancias
    private Contabilidad() {
        // TODO Auto-generated constructor stub
    }

    // Metodo que devuelve la unica instancia
    public static Contabilidad getInstance() {
        return instance;
    }

    // Metodos de negocio
    public void crearAsiento(String datos){
        System.out.println("Anotando en el libro de contabilidad " + datos);
    }

}
```

Gráfico 34. Clase con patrón Singleton

```
<!-- Factoria estatica -->
<!-- Contabilidad contabilidad = Contabilidad.getInstance(); -->
<bean id="contabilidad" class="com.grupoatrium.business.Contabilidad"
    factory-method="getInstance" />
```

Gráfico 35. Factoría estática

FACTORIA DE INSTANCIA

Cuando el método es dinámico y necesitamos una intancia de la clase entonces será una factoría de instancia.

```
public class ServicioContabilidad {

    public Contabilidad getLibroContable(){
        return Contabilidad.getInstance();
    }

}
```

Gráfico 36. Clase con método dinámico

```
<!-- Factoria dinamica -->
<!-- ServicioContabilidad servicio = new ServicioContabilidad();
     Contabilidad libro = servicio.getLibroContable(); -->
<bean id="servicio" class="com.grupoatrium.business.ServicioContabilidad" />
<bean id="libro" class="com.grupoatrium.business.Contabilidad"
    factory-bean="servicio" factory-method="getLibroContable"  />
```

Gráfico 37. Factoría dinámica

METODOS DE CICLO DE VIDA

Spring es el encargado de crear los beans por lo cual hablamos de componentes manejados.

Cuando se requiere ejecutar cierta lógica posterior a la construcción de un bean de Spring o antes de su eliminación, se proveen algunas formas de hacerlo.

INTERFACES INITIALIZINGBEAN Y DISPOSABLEBEAN

Una de las formas es que el bean implemente la interfaz InitializingBean, lo cual hace que se agregue el método afterPropertiesSet(), el cual se ejecuta después que se construye el bean.

La ventaja de esta funcionalidad es que permite ejecutar lógica con el bean ya inicializado, cosa que no se podría por ejemplo hacer en el constructor, ya que se invoca antes que se inyecten las dependencias con los setter. De hecho, el nombre "afterPropertiesSet" indica justamente eso.

Posibles aplicaciones:

- Opción para validar atributos inyectados.
- Inicializar otros objetos que utilizan los atributos inyectados.

```
public class Empleado implements InitializingBean, DisposableBean{
```

Gráfico 38. Clase que implementa interfaces de ciclo de vida

```
@Override
public void afterPropertiesSet() throws Exception {
    // TODO Auto-generated method stub
    System.out.println("Las propiedades ya han sido inyectadas");
}
```

Gráfico 39. Método afterPropertiesSet

Al igual como se requiere realizar inicialización, también se puede requerir controlar cuando se destruye el bean, como parte de la detención del contexto de Spring.

Esto puede ser útil cuando se requiere una parada "ordenada" de objetos, como por ejemplo:

- Liberar conexiones a destinos.
- Des-registrar drivers.
- Liberar caches

El equivalente, durante la destrucción del bean, a la interfaz InitializingBean es DisposableBean. El método a implementar es destroy().

```
@Override
public void destroy() throws Exception {
    // TODO Auto-generated method stub
    System.out.println("Metodo destroy de DisposableBean");
}
```

Gráfico 40. Método destroy

METODOS ANOTADOS @POSTCONTRUCT Y @PREDESTROY

Una segunda alternativa equivalente a la interfaz InitializingBean es agregar a la clase un método con la anotación @PostConstruct:

```java
@PostConstruct
public void postConstruct(){
    System.out.println("Metodo postConstruct");
}
```

Gráfico 41. Método anotado con @PostConstruct

Para que eso funcione, se debe habilitar el uso de anotaciones, agregando:

```xml
<!-- Para que Spring reconozca las anotaciones -->
<context:annotation-config />
```

Gráfico 42. Habilitar uso de anotaciones

Esto pertenece al namespace "context", que se agrega en el tag raíz del XML:

```xml
<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-3.1.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.1.xsd"
```

Gráfico 43. Espacio de nombres context

Otra forma alternativa de habilitar las anotaciones es agregar:

```xml
<bean class="org.springframework.context.annotation.
            CommonAnnotationBeanPostProcessor" />
```

Gráfico 44. Bean para la anotación @PostConstruct

@PreDestroy corresponde a las finalización de @PostConstruct.

```java
@PreDestroy
public void preDestroy(){
    System.out.println("Metodo preDestroy");
}
```

Gráfico 45. Método anotado como @PreDestroy

METODOS INIT Y DESTROY

Una tercera alternativa equivalente es agregar a la declaración en el XML el atributo init-method:

```
<bean id="empleado1" class="com.grupoatrium.modelo.Empleado"
      init-method="initEmpleado" destroy-method="destroyMetodo">
    <property name="numeroEmpleado" value="127" />
    <property name="nombre" value="Juan" />
    <property name="apellido" value="Lopez" />
    <property name="jefe" value="false" />
    <property name="sueldo" value="27000" />
</bean>
```

Gráfico 46. Atributo init-method

En ambos casos el método debe ser público y sin parámetros. Si tiene retorno, se ignora, así que conviene que sea void.

```
public void initEmpleado(){
    System.out.println("Metodo initEmpleado");
}
```

Gráfico 47. Metodo init

Si en todas las clases creamos un método iniciar() y otro destruir() estos se pueden configurar de forma conjunta.

```
<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-3.1.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.1.xsd"
        default-init-method="init" default-destroy-method="destroyMetodo">
```

Gráfico 48. default-init-method

Es importante saber que en los beans declarados como prototype no se invoca al método destroy.

ORDEN DE EJECUCION

- @PostConstruct
- afterPropertiesSet
- init-method

En el caso que dos o más inicializaciones utilicen el mismo método, dicho método se ejecuta una sola vez.

Si creamos el ApplicationContext como AbstractApplicationContext podemos invocar al método registerShutdownHook para indicar que al finalizar la aplicación llame a todos los métodos destroy.

```
// Levantar el contexto de Spring
// Crear el contenedor de beans a partir del archivo applicationContext.xml
AbstractApplicationContext context =
        new ClassPathXmlApplicationContext("applicationContext.xml");
```

Gráfico 49. AbstractApplicationContext

```
// Eliminar todos los beans del contenedor
context.registerShutdownHook();
```

Gráfico 50. Eliminar beans del contenedor

USO DE ASSERT

A través del utilitario org.springframework.util.Assert se pueden realizar validaciones. Esto se puede aplicar, por ejemplo, en conjunto con una inicialización.

Cuando un assert no se cumple, se lanza internamente una excepción, que impide que se inicie el contexto de Spring.

Los assert permiten colocar opcionalmente mensajes.

```java
public void afterPropertiesSet()
        throws Exception {
    Assert.notNull(optionsDAO);
    List<String> options = optionsDAO.getOptions();
    Assert.notNull(options);
    Assert.notEmpty(options, "'options' es empty");
    int idx = 0;
    for (String option : options) {
        cache.put(idx++, option);
    }
}
```

Gráfico 51. Validaciones con Assert

CARGAR ARCHIVOS DE RECURSOS

Spring nos permite utilizar 3 clases para la carga de archivos de recursos:

- FileSystemResource: para un archivo en un sistema de archivos. También permite URL.
- ClassPathResource: para un recurso en el classpath.
- UrlResource: para un recurso que está en una URL. También se puede utilizar para archivos, con "file:".

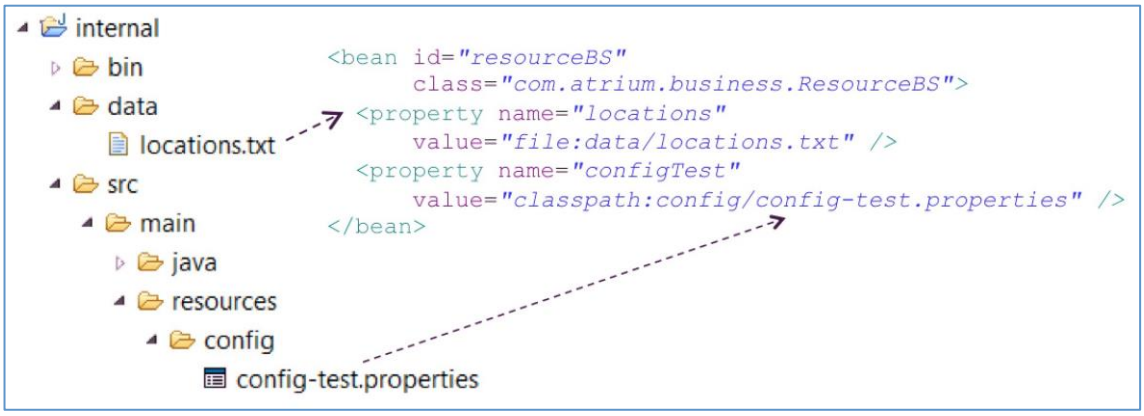


Gráfico 52. Cargar archivos de recursos

CARGAR ARCHIVOS DE PROPIEDADES

Spring provee la clase:

org.springframework.beans.factory.config.PropertyPlaceholderConfigurer

para la gestión de propiedades delegada a recursos, equivalentes a archivos properties. Por ejemplo, si el properties se llama config-test.properties, y está ubicado en el classpath:

```
<bean
 class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="locations">
    <list>
      <value>classpath:config/config-test.properties</value>
    </list>
  </property>

  <property name="ignoreUnresolvablePlaceholders" value="true" />
</bean>
```

Gráfico 53. Cargar archivo de propiedades

CARGAS EAGER Y LAZY

Si ponemos el atributo lazy-init a true, el bean es creado al invocarlo por primera vez. Su valor por defecto es false.

Es importante saber que esta propiedad no se hereda. Se debe definir específicamente para cada bean.

```
<bean id="empleado1" class="com.grupoatrium.modelo.Empleado" lazy-init="true">
    <property name="numeroEmpleado" value="127" />
    <property name="nombre" value="Juan" />
    <property name="apellido" value="Lopez" />
    <property name="jefe" value="false" />
    <property name="sueldo" value="27000" />
</bean>
```

Gráfico 54. Carga lazy

AMBITOS DE UN BEAN

A la hora de crear el bean se le puede definir un ámbito. Esto nos limitará el uso que le podemos dar. Los ámbitos definidos en Spring son:

- Singleton; Por defecto todos los beans son instancias únicas.
- Prototype; Permite que un bean se instancie tantas veces se quiera.
- Request; El bean pertenece al ámbito de petición. Solo para aplicaciones Web.
- Session; El bean pertenece al ámbito de sesión. Solo para aplicaciones Web.

- Global-session; El bean pertenece al ámbito de sesión global. Solo es válido para el contexto de un portlets.

El siguiente fragmento implica que cada vez que se utilice el método getBean() para recuperar el bean del contenedor, se crea una nueva instancia de este.

```
<bean id="empleado2" class="com.grupoatrium.modelo.Empleado" scope="prototype">
```

Gráfico 55. Bean con ámbito prototype

Los beans singleton se mantienen durante toda la aplicación. Los beans prototype, solo durante su utilización.

ESPACIO DE NOMBRES C

Podemos utilizar el espacio de nombres c para inyectar valores a través del constructor a partir de Spring 3.0.

Lo primero será agregar dicho espacio de nombres a nuestro applicationContext.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:c="http://www.springframework.org/schema/c"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">
```

Gráfico 56. Espacio de nombres c

Por cada parámetro del constructor Spring generara dos atributos que podremos utilizar en la declaración del bean. Uno de ellos será para inyectar un valor simple y el otro para inyectar la referencia de otro bean.

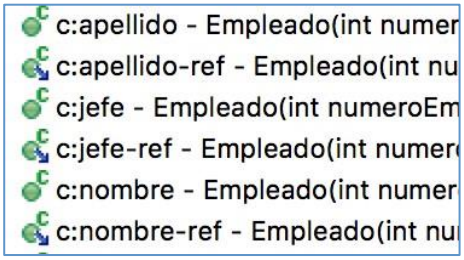


Gráfico 57. Atributos generados

```
<!-- Empleado empleado3 = new Empleado(129, "Antonio",
    "Hernandez", false, 19000); -->
<bean id="empleado" class="com.grupoatrium.modelo.Empleado"
    c:numeroEmpleado="129" c:nombre="Antonio" c:apellido="Hernandez"
    c:jefe="false" c:sueldo="19000" />
```

Gráfico 58. Inyección por nombre de constructor

También podemos hacer referencia a la posición del parámetro. Como XML no permite usar dígitos como primer carácter de un atributo, hemos tenido que añadir un guión bajo como prefijo.

```
<bean id="empleado2" class="com.grupoatrium.modelo.Empleado"
    c:_0="130" c:_1="Miguel" c:_2="Gonzalez" c:_3="true" c:_4="46000" />
```

Gráfico 59. Inyección por posición de argumentos en el constructor

Si solo hay un parámetro en el constructor podemos no identificarlo.

```
<bean id="empleado3" class="com.grupoatrium.modelo.Empleado"
    c:_="Maria" />
```

Gráfico 60. Inyección de un solo argumento en el constructor

ESPACIO DE NOMBRES P

De una forma similar a lo visto en el apartado anterior, tenemos otro espacio de nombres para hacer la inyección de propiedades.

El espacio de nombres p nos permite asignar propiedades al bean de otra forma más rápida. Es necesario agregarlo al fichero xml.

```xml
<?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-4.3.xsd">

    <bean id="empleado1" class="com.grupoatrium.modelo.Empleado"
        p:numeroEmpleado="127" p:nombre="Juan"
        p:apellido="Lopez" p:jefe="false" p:sueldo="27000" />

</beans>
```

Gráfico 61. Espacio de nombres p

ESPACIO DE NOMBRES UTIL

El espacio de nombres P no nos permite inyectar colecciones para eso necesitamos otro espacio de nombres: útil.

Los elementos que me aporta son:

- <util:constant> Lo utilizamos para crear un bean una propiedad estatica
- <util:list> Para crear una lista como bean
- <util:map> Para crear un mapa como bean
- <util:properties> Se usa para crear una colección de tipo Properties como bean
- <util:property-path>
- <util:set> Lo utilizamos para crear un bean de tipo Set

Como ya intuís lo primero será agregarlo a nuestro applicationContext.xml:

```xml
<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"
    xmlns:util="http://www.springframework.org/schema/util"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
        http://www.springframework.org/schema/util
        http://www.springframework.org/schema/util/spring-util-4.3.xsd">
```

Gráfico 62. Espacio de nombres util

Luego crearemos la colección:

```
<util:list id="empleados" value-type="com.grupoatrium.modelo.Empleado">
    <ref bean="empleado1" />
    <ref bean="empleado2" />
    <bean class="com.grupoatrium.modelo.Empleado">
        <property name="numeroEmpleado" value="140" />
        <property name="nombre" value="Antonio" />
        <property name="apellido" value="Sanz" />
        <property name="jefe" value="false" />
        <property name="sueldo" value="15000" />
    </bean>
</util:list>
```

Gráfico 63. Crear un bean de tipo List

```
<util:list id="proyectos" value-type="java.lang.String">
    <value>Gas Natural</value>
    <value>Sabadell</value>
    <value>Mercadona</value>
</util:list>
```

Gráfico 64. Un array se puede crear como un bean de tipo List

```
<util:set id="equipos" value-type="com.grupoatrium.modelo.Empleado">
    <ref bean="empleado1" />
    <ref bean="empleado2" />
</util:set>
```

Gráfico 65. Crear un bean de tipo Set

```
<util:map id="jefesProyecto" key-type="java.lang.String"
    value-type="com.grupoatrium.modelo.Empleado">
    <entry key="Sabadell" value-ref="empleado1" />
    <entry key="Mercadona" value-ref="empleado2" />
</util:map>
```

Gráfico 66. Crear un bean de tipo Map

```
<util:properties id="emails">
    <prop key="Juan">juan@everis.com</prop>
    <prop key="Maria">maria@everis.com</prop>
</util:properties>
```

Gráfico 67. Crear un bean de tipo Properties

Por ultimo la inyectamos en la propiedad del bean necesaria.

```
<bean id="empresa2" class="com.grupoatrium.modelo.Empresa"
    p:empleados-ref="empleados"
    p:proyectos-ref="proyectos"
    p:equipos-ref="equipos"
    p:jefesProyecto-ref="jefesProyecto"
    p:emails-ref="emails" />
```

Gráfico 68. Inyección de colecciones con espacio de nombres p

CONEXIONES AUTOMATICAS CON AUTOWIRE

Spring nos ofrece conectar los beans de forma automática es decir, ya no vamos a necesitar hacer la inyección de dependencias en la propiedad necesaria ya que Spring detectará que bean declarado puede ser inyectado en otro de forma automática.

Podemos efectuar conexiones automáticas de un bean con sus dependencias de 3 formas:

- default; El valor asignado por defecto es no.
- no; Significa que no habrá conexión automática y el bean se debe inyectar haciendo referencia a través del atributo ref.
- byName; una propiedad se conecta con un bean que tenga el mismo nombre de esta.
- byType; una propiedad se conecta con un bean que sea del mismo tipo que esta.
- constructor; busca los beans que coinciden con los argumentos del constructor. Es igual que byType pero con constructores en vez de propiedades.

```
<!-- ClientesBS clientesBS = new ClientesBS();
     clientesBS.setDAO(clientesDAOTest) -->
<bean id="clientesBS" class="com.grupoatrium.negocio.ClientesBS"
    autowire="byType" />
```

Gráfico 69. Autowire byType

Podemos establecer una conexión automática predeterminada para todos los beans.

```
<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-4.3.xsd"
    default-autowire="byType">
```

Gráfico 70. Autowire predeterminado

Si queremos excluir aquellos beans que no queremos que se tengan en cuenta para la conexión automática,

```
<!-- ClientesDAO clientesDAO = new ClientesDAO(); -->
<bean id="clientesDAO" class="com.grupoatrium.persistencia.ClientesDAO"
    autowire-candidate="false"/>
```

Gráfico 71. Excluir un bean de conexión automática

Podemos seguir utilizando conjuntamente autowire y también inyectar una propiedad con <property>

Sin embargo no podemos combinar la conexión automática en los constructores con elementos <constructor-arg>

TRABAJAR CON VARIOS ARCHIVOS XML

Podemos separar los ficheros de configuración y que los lea automáticamente el contenedor utilizando la etiqueta <import>

```
<import resource="servicios.xml"/>
```

Gráfico 72. Importar otro archivo xml

Para cargar el contenedor de beans lo haremos a partir del XML principal.

```
ApplicationContext factoria =
            new ClassPathXmlApplicationContext("spring.xml");
```

Gráfico 73. Levantar el contexto de Spring

Otra opción es declarar múltiples archivos en la construcción del contexto de Spring. Por ejemplo, si en vez de hacer el import de applicationContext-dao desde applicationContext-business, se declaran ambos, resulta:

```
ApplicationContext ctx =        new

ClassPathXmlApplicationContext(

        "applicationContext-business.xml",

        "applicationContext-dao.xml");
```

4.- CONFIGURACION A TRAVES DE ANOTACIONES

ANOTACIONES PARA GENERAR BEANS

Para poder utilizar anotaciones necesitamos activarlo mediante el espacio de nombres context.

```
<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-4.3.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-4.3.xsd">

    <context:annotation-config />
```

Gráfico 74. Activar anotaciones

Otra forma de habilitar anotaciones sería crear un bean de la clase de la anotación a interpretar. Por ejemplo para la anotación @PostConstruct:

<bean class="org.springframework.context.annotation.CommonAnnotationBeanPostProcessor" />

DETECCION AUTOMATICA DE BEANS

Todas las clases que estén anotadas con:

- @Component; componente de Spring
- @Controler; controlador MVC
- @Repository; repositorio de datos
- @Service; servicio
- @Configuration; clase que define beans

```
@Service("seguro")
public class Aseguradora {
```

Gráfico 75. Clase anotada como bean

El bean se creará con el ID “seguro”, si no hubiésemos puesto nada el bean se crearía con el nombre de la clase en minúsculas.

La distinción entre anotaciones es solamente para facilitar su identificación y filtrado. Spring no valida que las clases sean efectivamente del tipo que se declara. Por ejemplo, se podría marcar con @Service una clase de acceso a datos, y el efecto es el mismo que utilizar @Repository o @Component.

Para que Spring genere los beans de las clases anotadas debemos incluir la siguiente etiqueta en el documento xml:

```
<context:component-scan base-package="com.grupoatrium" />
```

Gráfico 76. Detección automática de beans

Tenemos los siguientes tipos de filtros para component-scan:

- annotacion; clases con la anotación que indica el atributo expresión.
- assignable; clases que se podrían asignar a lo que indica el atributo expression.
- aspectj; clases que coinciden con la expresión Aspectj especificada en el atributo expression
- custom; usa una implementacion TypeFilter especificada en el atributo expression
- regex; clases cuyo nombre coincide con la expresión regular especificada en el atributo expression

Este fragmento indica que se registran todas las implementaciones de Persona excepto las que están anotadas con @Service

```
<context:component-scan base-package="paquete,paquete" >
    <context:include-filter type="assignable"
                expression="app.modelo.Persona" />
    <context:exclude-filter type="annotation"
                expression="org.springframework.stereotype.Service" />
</context:component-scan>
```

Gráfico 77. Filtro para component-scan

INYECCION DE PROPIEDADES

@RESOURCE

Esta anotación la utilizamos cuando queremos inyectar una propiedad por nombre y por tipo.

```
@Resource
private ItfzTaller taller;
```

Gráfico 78. Inyección por tipo

Si solo hay un bean de ese tipo no habría problema. Si hay varios le podemos poner el nombre del bean a inyectar.

```
@Resource(name="tallerPintura")
private ItfzTaller taller;
```

Gráfico 79. Inyección por nombre

Podemos situar esta anotación sobre la propiedad o sobre el método set. Funcionará correctamente si el nombre del método set se corresponde con el id del bean a inyectar.

```
@Resource
public void setTallerPintura(ItfzTaller taller) {
    this.taller = taller;
}
```

Gráfico 80. Inyección por nombre de método set

@AUTOWIRED

En el siguiente método Spring intenta la conexión automática por tipo, esto es, si encuentra un bean de tipo ItfzTaller lo conecta.

```
@Autowired
private ItfzTaller taller;
```

Gráfico 81. Inyección por tipo con Autowired

Podemos utilizar esta anotación en constructores, propiedades y cualquier método aunque sean privados.

Si no existe un bean que poder conectar o hay varios, Spring genera NoSuchBeanDefinitionException.

El atributo required indica que la propiedad se puede conectar o no, en este segundo caso tomará el valor null. Spring intenta realizar la conexión automática, pero si no hay un bean que coincida, lo dejará sin conectar.

```
@Autowired(required=false)
private Instrument instrument;
```

Gráfico 82. Inyección no obligatoria

Con sobrecarga de constructores, solo un constructor se puede configurar como required=true el resto deben ser false, Spring selecciona el constructor que tenga más argumentos que puedan cumplirse.

@QUALIFIED

Utilizamos esta anotación para resolver dependencias ambiguas.

Spring intenta conectar de forma automática con un bean cuyo id sea tallerPintura.

```
@Autowired
@Qualifier("tallerPintura")
private ItfzTaller taller;
```

Gráfico 83. Uso de Qualified

@INJECT

Otro modelo DI es el propuesto por JSR.330 (también conocido como at inject).

La anotación @Inject es igual que @Autowired de Spring con la excepción que no tiene un atributo required.

```
@Inject
private Instrument instrument;
```

Gráfico 84. Anotación Inject

@NAMED

Al igual que en Spring teníamos @Qualifier, JSR.330 nos proporciona la anotación @Named para resolver dependencias ambiguas

```
@Inject
@Named("guitar")
private Instrument instrument;
```

Gráfico 85. Anotación Named

@VALUE

Con la anotación @Value (nueva en Spring 3.0) podemos conectar valores a una propiedad.

```
@Value("Opel Astra")
private String modelo;
```

Gráfico 86. Anotación Value

Incluso podemos utilizar SpEL (Lenguaje de expresiones en Spring)

@Value(“#{cliente.nombre}”)

private String nombre;

VALIDACIONES CON @REQUIRED

Se podría validar que una dependencia esté inyectada en un bean utilizando un Assert en conjunto con algún control de inicialización, como InitializingBean.

Sin embargo, existe una anotación especializada para este tipo de verificaciones, llamada @Required. En este caso, si se quiere validar que una propiedad esté inyectada en un bean, se coloca:

```
public class ClientesBS {

    private ItfzDAO dao;

    @Required
    public void setDao(ItfzDAO dao) {
        this.dao = dao;
    }
```

Gráfico 87. Inyeccion requerida

Si no se inyecta, lanza BeanInitializationException: Property ‘dao’ is required for bean 'clientesBS'

Recordemos que las anotaciones deben estar habilitadas, con <context:annotation-config /> o declarando un bean de la clase "org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor".

5.- CONFIGURACION BASADA EN JAVA

CLASES DE CONFIGURACION

La tercera forma que nos queda por ver es como configuramos los beans de Spring utilizando clases Java.

Creamos una clase en un paquete aparte, esto se recomienda para no mezclar la configuración con el resto de clases de la aplicación.

Anotamos la clase con la anotación @Configuration.

```
@Configuration
public class ClaseConfig {
```

Gráfico 88. Clase de configuración Java

DECLARAR BEANS

Crearemos un método anotado como @Bean para generar el bean.

```
@Bean
public ClientesDAO clientesDAO(){
    return new ClientesDAO();
}

@Bean
public ClientesDAOTest clientesDAOTest(){
    return new ClientesDAOTest();
}
```

Gráfico 89. Declaración de beans

El identificador será el nombre del método.

Si quisiéramos cambiarlo se lo asignaríamos en el atributo name de la anotación @Bean.

@Bean(name=”otroIdentificador”)

INYECCION DE DEPENDENCIAS

Invocamos directamente a los métodos set de las propiedades a inyectar pasando como argumento la llamada al método que devuelve el bean.

```
@Bean
public ClientesBS clientesBS(){
    ClientesBS clientesBS = new ClientesBS();
    clientesBS.setDao(clientesDAOTest());
    return clientesBS;
}
```

Gráfico 90. Inyección de propiedades

DETECCION AUTOMATICA

Si anotamos nuestras clases para que Spring genere automáticamente un bean de ellas, tan solo necesitamos poner la anotación @ComponentScan sobre el paquete o paquetes a escanear.

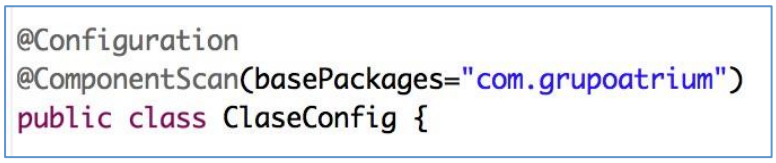


Gráfico 91. Detección automática de beans

TRABAJAR CON VARIAS CLASES DE CONFIGURACION

De la misma forma que podíamos trabajar con varios documentos xml también nos podemos encontrar con varias clases de configuración.

La anotación @import me permite importar otras clases:

@Configuration

@Import({ClaseConfig1.class,ClaseConfig2.class})

public class ClaseConfig{

}

IMPORTAR CONFIGURACION XML

Podemos combinar la configuración basada en Java con la configuración xml.

Si queremos levantar el contexto de Spring desde la clase de configuración entonces incluiremos la anotación @ImportResource para cargar el documento xml.

@Configuration

@ImportResource(“classpath:applicationContext.xml”) public

class ClaseConfig{

}

IMPORTAR CLASE CONFIG EN UN ARCHIVO XML

Si por el contrario queremos levantar el contexto de Spring desde el archivo applicationContext.xml incluiremos la declaración del bean de la clase config.

<bean class=”paquete.ClaseConfig” />

6.- CONFIGURACION AVANZADA

PERFILES

Que ocurre si queremos tener conexiones a BBDD diferentes si trabajamos en modo prueba o en producción.

Sería interesante tener un bean con los datos de conexión a la BBDD para pruebas en un bean que solo se generase si se necesita.

Desde la versión 3.1. de Spring podemos utilizar la anotación @Profile para definir el perfil asociado a un bean.

Hay que tener en cuenta que en la versión 3.1. solo se podía utilizar a nivel de clase pero en la versión 3.2 ya se puede anotar un método.

CREAR LOS BEANS PARA UN DETERMINADO PERFIL

```
<beans profile="real">
    <bean id="clientesDAO" class="com.grupoatrium.persistencia.ClientesDAO" />
</beans>

<beans profile="test">
    <bean id="clientesDAO" class="com.grupoatrium.persistencia.ClientesDAOTest" />
</beans>
```

Gráfico 92. Agrupar declaración de beans por perfil

ACTIVAR EL PERFIL

Spring dispone de dos propiedades para determinar que perfiles se han activado:

- spring.profiles.active
- spring.profiles.default.

Si se establece spring.profiles.active, su valor determina que perfiles están activos. En caso contrario Spring se fija en spring.profiles.default. Si no se establece ninguna de las dos propiedades, no habrá perfiles activos y solo se crean los beans que no tengan perfil.

Existen varias formas de establecer estas propiedades:

- Como parámetros iniciales de inicialización en DispatcherServlet.
- Como parámetros de contexto de una aplicación Web.
- Como entradas JNDI
- Como variables de entorno.
- Como propiedades del sistema JVM
- Con la anotación @ActiveProfiles en una clase de pruebas de integración.

Cuando veamos Spring MVC mostraremos como activar perfiles en el DispatcherServlet.

PRUEBAS CON PERFILES

Spring nos ofrece la anotación @ActiveProfiles para especificar el perfil que debe activarse al ejecutar una prueba.

En el siguiente ejemplo activamos el perfil test.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(name="classpath:applicationContext.xml")
@ActiveProfiles("test")
public class ClientesTest {
```

Gráfico 93. Activar perfil de prueba

BEANS CONDICIONALES

Con la llegada de Spring 4 tenemos la anotación @Conditional que nos permite evaluar una condición y dependiendo del valor obtenido el bean se creará o no.

Los pasos para su utilización sería crear una clase que implemente la interface Condition y en el método matches crear la condición.

```java
import org.springframework.context.annotation.Condition;
import org.springframework.context.annotation.ConditionContext;
import org.springframework.core.type.AnnotatedTypeMetadata;

public class MyCondition implements Condition{

    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
        // Aqui se evalua una expresion que devolverá un valor boleano
        // Si es true hara que el bean se genere
        // Si es false no se creará
        return true;
    }

}
```

Gráfico 94. Clase que implementa la condición

Luego añadimos la anotación @Conditional sobre el bean y le indicaremos la clase que contiene la condición.

```java
@Bean
@Conditional(MyCondition.class)
public ClientesDAOTest clientesDAOTest(){
    return new ClientesDAOTest();
}
```

Gráfico 95. Bean condicional

BEAN PRINCIPAL

A la hora de declarar bean, puede evitar las ambigüedades si designa a uno de los bean candidatos como el principal. En caso de producirse una ambigüedad, Spring elegirá al principal entre todos los candidatos.

Veamos como hacerlo en sus 3 formas de configuración:

```xml
<bean id="clientesDAO"
    class="com.grupoatrium.persistencia.ClientesDAO"
    primary="true" />
```

Gráfico 96. Declaración en XML de bean principal

```
@Repository
@Primary
public class ClientesDAO implements ItfzDAO{
```

Gráfico 97. Declaración con anotaciones de bean principal

```
@Bean
@Primary
public ClientesDAO clientesDAO(){
    return new ClientesDAO();
}
```

Gráfico 98. Declaración en clases config de bean principal

7.- LENGUAJE DE EXPRESSIONES SPEL

El lenguaje de expresiones SpEL (Spring Expression Language) es nuevo a partir de la versión 3 de Spring.

La sintaxis de este lenguaje es la siguiente:

#{expresión}

- Podemos inyectar valores literales como: – Números:
- Enteros #{8}
- Decimales #{8.37}
- Notación científica #{1e5}

– Cadenas

- Comillas simples #{'texto'}
- Comillas dobles #{"texto"}

– Booleanos

- #{true} o #{false}
- Referenciar un bean:

<property name="direccion" value="#{direccion}" />

- Acceder a una propiedad de otro bean:

<property name="direccion" value="#{cliente.direccion}" />

- Método de otro bean:

<property name="direccion" value="#{cliente.direccion()}" />

- Acceso a variables estáticas:

<property name="numero" value="#{T(java.lang.Math).PI}" />

- Acceso a métodos estáticos: <property name="numero" value="#{T(java.lang.Math).random()}" />

- Operadores:

– Aritméticos: +, -, *, /, %, ^(potencia)

– Relacionales: <, >, ==, <=, >=, lt, gt, eq, le, ge – Lógicos: and, or, not o !

– Condicionales:

- Ternario ?:

<property name="numero"

value="#{ condicion ? Valor_true : valor_false}" /> •

Elvis ?:

<property name="numero"

value="#{ expresion ?: valor_si_expresion_igual_null}" />

– Expresiones regulares**:** matches

- Acceder a un miembro de una colección:

<property name="numero" value="#{numeros[3]}" />

- Acceder a un elemento de un mapa por su clave:

<property name="propiedad" value="#{mapa['clave']}" />

- systemEnvironment contiene las variables de entorno del equipo:
  <property name="homePath" value="#{systemEnvironment['HOME']}" />

- systemProperties contiene las propiedades introducidas con –D:
  <property name="homePath" value="#{systemProperties['autor']}" />

- Seleccionar miembros de la colección en base a una condición: (**Selección**)
  <property name="prop" value="#{colección.?[condicion]}" />

- Primer elemento coincidente de la colección:
  <property name="prop" value="#{colección.^[condicion]}" />

- Ultimo elemento coincidente de la colección:
  <property name="prop" value="#{colección.$[condicion]}" />

- Recopilar elementos de una colección para incluirla en otra con una sola propiedad: (**Proyección**)
  <property name="nuevaColeccion" value="#{colección.![propiedad]}" />

- Recopilar elementos de una colección para incluirla en otra con varias propiedades:

<property name="nuevaColeccion" value="#{colección.![prop1 + ' , ' + prop2}" />

- Combinar proyección y selección

```
<property name="prueba"
          value="#{coleccion.?[condicion].![prop1 + ', ' + prop2]}" />
```

Gráfico 99. Proyección y selección

INDICE DE GRÁFICOS