



El Framework Spring

- ① Evolución del framework Spring
- ② Arquitectura del Framework
- ③ Inversión de Control (IoC) e Inyección de Dependencias (DI)
- ④ Librerías básicas necesarias
- ⑤ Ejemplo práctico para entender los conceptos
- ⑥ Configuración por XML, los tags **<bean>** y **<beans>**.
- ⑦ Configuración por anotaciones
- ⑧ Ciclo de vida de un bean manejado por Spring

¿Qué es Spring?



Spring es un framework *open source*, creado por Rod Johnson, descrito en su libro: *Expert One-on-One: J2EE Design and Development*. Spring fue creado para abordar la complejidad del desarrollo de aplicaciones empresariales, y usando componentes simples JavaBeans logra cosas que antes sólo era posible con componentes complejas (como EJB).

Spring es una plataforma que nos proporciona soporte para desarrollar aplicaciones Java SE y aplicaciones Java EE. Spring maneja la infraestructura de la aplicación y así los programadores se pueden centrar en el desarrollo de las aplicaciones.

Spring Framework	
RELEASE	DOCUMENTATION
5.0.0 M2 <small>PRE</small>	Reference API
5.0.0 <small>SNAPSHOT</small>	Reference API
4.3.4 <small>SNAPSHOT</small>	Reference API
4.3.3 <small>CURRENT</small> <small>GA</small>	Reference API
4.2.8 <small>GA</small>	Reference API
3.2.17 <small>GA</small>	Reference API

La primera versión fue lanzada formalmente bajo la licencia Apache 2.0 en 2004. A partir de ahí se sucedieron diferentes versiones con mejoras. En enero de 2013 fue anunciado el inicio de la versión 4 y la versión actual es la 4.3.3.

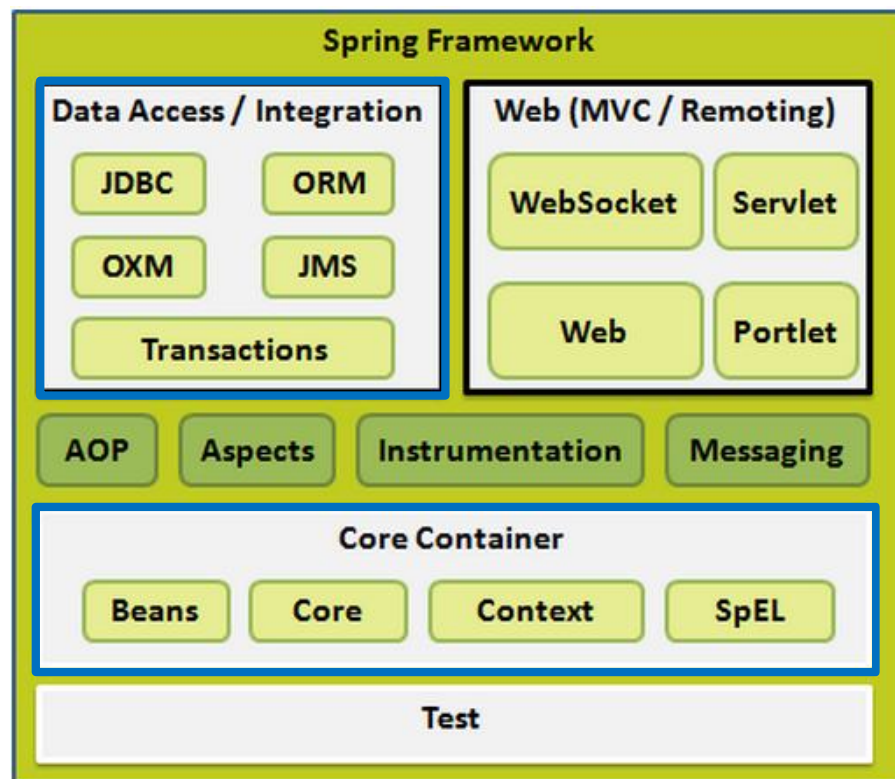
El sitio oficial es spring.io

Arquitectura de Spring

Módulos que lo componen



El framework Spring consta de funciones organizadas en aproximadamente 20 módulos. Estos módulos se agrupan en:

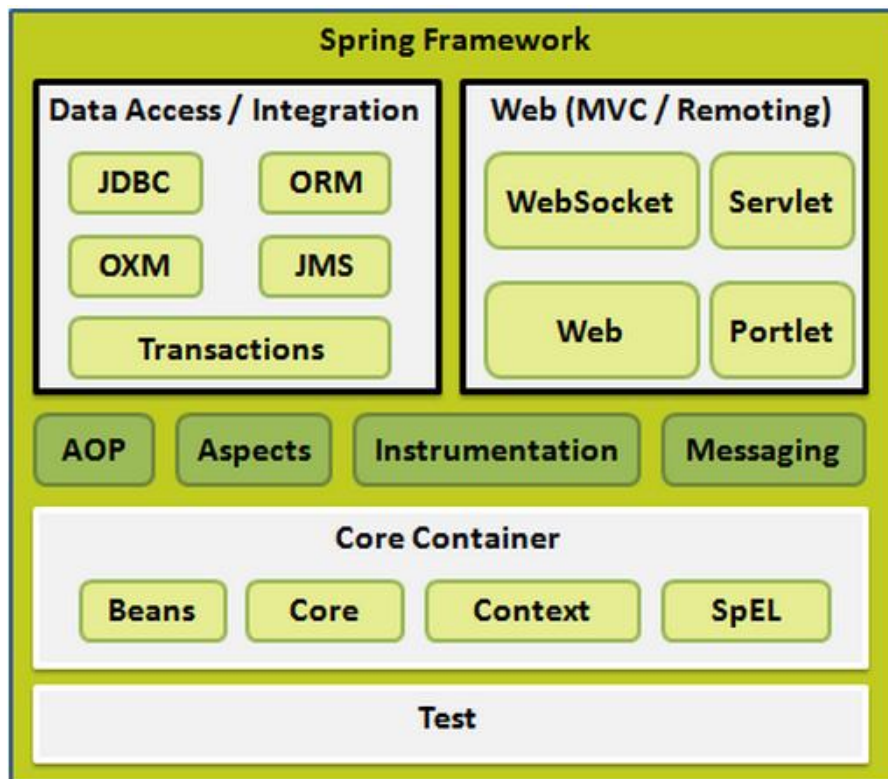


Core Container: Este módulo contiene las partes fundamentales del framework, incluyendo la inversión de control y la inyección de dependencias. Estos principios son los que permiten desacoplar la configuración y especificación de dependencias, de la lógica del sistema.

Data Access/Integration: El módulo provee soporte para facilitar y agilizar el uso de tecnologías como JDBC, ORM, JMS y Transacciones. Por ejemplo: En JDBC, elimina la necesidad de parsear y codificar los errores específicos de cada base de datos.

Arquitectura de Spring

Módulos que lo componen (cont.)



Web: Ofrece integración para trabajar con tecnologías Web, incluyendo un framework propio, Spring MVC.

AOP (Programación Orientada a Aspectos): Habilita la implementación de rutinas transversales. Es muy utilizado por el propio Framework para implementar características como transaccionalidad declarativa.

Test: Este módulo brinda soporte para testear aplicaciones/componentes que utilizan Spring como Contenedor de IoC, usando tecnologías como JUnit o TestNG. Entre otras cosas, da soporte para cargar el contexto de Spring en los Test, además de proveer de objetos Mock que pueden ser usados para testear la aplicación.

Spring

Conceptos importantes



Inversión de Control (IoC) - Inyección de Dependencias (ID)

Estos conceptos algunas veces se los usa de manera intercambiable, sin embargo:

La **Inversión de Control** es un concepto mucho más general, y puede estar manifestado de diferentes maneras. La **Inyección de Dependencias** es un ejemplo concreto de Inversión de Control.

La IoC cubre un amplio rango de técnicas que le permiten a un objeto volverse un participante “pasivo” en el sistema. Cuando la técnica de IoC es aplicada, un objeto delega el control sobre algunas características o aspectos al framework o ambiente donde se ejecuta.

La IoC utiliza por ejemplo **Inyección de Dependencias** y **Programación Orientada a Aspectos** (AOP) para hacerse cargo de ciertos controles sobre los objetos de la aplicación.

Spring



Inversión de Control (IoC) – Inyección de Dependencias (ID)

Inversión de control implica darle el control al Framework!!

Un ejemplo concreto de IoC es la **Inyección de Dependencias**, que se usa para vincular (*wire*) a los objetos de la aplicación. **Es posible darle al framework Spring la responsabilidad de crear objetos a partir del código de la aplicación y de conectar sus dependencias.**

El módulo **Core Container** incluye clases e interfaces que representan una sofisticada implementación del patrón *Factory*, que es el corazón de la inversión de control y consiste en delegar en el Contenedor de Spring la creación y la forma en que los beans deben componerse unos con otros.



Al mecanismo de Spring para relacionar un bean con otro se lo conoce como **Inyección de Dependencia**

Spring



Beans y Contenedores de Inversión de Control

¿Qué es un Bean en Spring?

En Spring, los objetos que conforman la estructura principal de las aplicaciones, y que son administrados por el **Contenedor de IoC**, son llamados "beans". Un bean es un objeto que es declarado, para que sea instanciado, inicializado con todas sus dependencias y administrado por el contenedor de IoC. Un bean es simplemente uno de muchos objetos de nuestra aplicación.

Las clases básicas del Contenedor de IoC de Spring están en dos paquetes: **org.springframework.beans** y **org.springframework.context**.

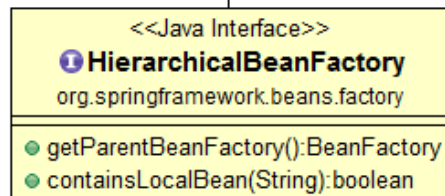
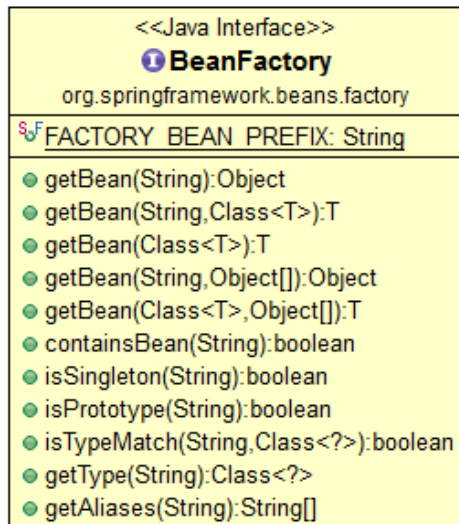
Las dos interfaces que realizan el manejo del ciclo de vida de los beans son:

- **org.springframework.beans.factory.BeanFactory**
- **org.springframework.context.ApplicationContext**

Ambas interfaces representan contenedores de beans, solo que de tipos distintos y existen múltiples implementaciones.

Spring

Contenedores de Inversión de Control



El **BeanFactory** es el tipo de contenedor más simple.

Proporciona el soporte básico para Inyección de Dependencias.

Tiene la capacidad para crear beans, vincularlos con sus dependencias, manejar su ciclo de vida, llamar a métodos de inicio y fin, etc.

Proveer un mecanismo de lookup para los beans.

Su implementación es más liviana y recomendada para ambientes de pocos recursos como podría ser una aplicación Móvil.

La implementación más conocida es:

`org.springframework.beans.factory.BeanFactory`

El **ApplicationContext** es una especialización de **BeanFactory**.

Los contenedores Spring que implementan esta interface son más avanzados que los que implementan **BeanFactory**.

Esta interface agrega al **BeanFactory** funcionalidades más específicas para aplicaciones empresariales como resolver mensajes textuales desde archivos de propiedades (i18n) o publicar eventos a los listeners interesados.

Existen varias implementaciones de **ApplicationContext** como:

`FileSystemApplicationContext`

`ClassPathApplicationContext`

`WebXmlApplicationContext`

Spring

Contenedores de IoC – Configuración



Existen varias implementaciones de **ApplicationContext**. Para aplicaciones de escritorio comunmente se utiliza la clase **ClassPathXmlApplicationContext**, a la que hay que pasarle el nombre del archivo XML con la definición de los **beans**. Esta clase buscará el archivo en cualquier directorio del *classpath*.

```
public class Main {  
    public static void main(String[] args) {  
        ApplicationContext context = new ClassPathXmlApplicationContext("app-ctx.xml");  
        AutenticaUsuarioService autUsuarioService=  
            context.getBean("autUsuarioService", AutenticaUsuarioService.class);  
        Usuario usuario = (Usuario) context.getBean("usuario");  
        System.out.println("La autenticacion fue: " + autUsuarioService.autenticarUsuario(usuario));  
    }  
}
```

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans" . . .  
    <bean id="autenticador" class="ar.com.ttps.spring.beans.AutenticarPorGuarani"/>  
    <bean id="autUsuarioService" class="ar.com.ttps.spring.beans.AutenticaUsuarioService">  
        <property name="autenticador" ref="autenticador"/>  
    </bean>  
</beans>
```

app-ctx.xml
donde se
definen los
beans

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans" . . .  
    <context:component-scan base-package="ar.com.ttps.spring.beans"/>  
</beans>
```

app-ctx.xml
para indicar
que se usará
anotaciones

Spring

Contenedores de IoC – Configuración



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="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
version="3.0">
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring/app-ctx.xml</param-value>
    classpath:*ar/com/tps/resources/app-ctx.xml
  </context-param>
  <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>
  .
  .
  .
</web-app>
```

Al igual que en las aplicaciones de escritorio, a través del **ApplicationContext de Spring** se puede acceder explícitamente a los beans así:

```
ApplicationContext context =
    WebApplicationContextUtils.getRequiredWebApplicationContext(getServletContext());
usuarioService = context.getBean("autUsuarioService", AutenticaUsuarioService.class);
```

Sin embargo la forma habitual de trabajar en aplicaciones web con Spring es usando inyección de dependencias.

Spring

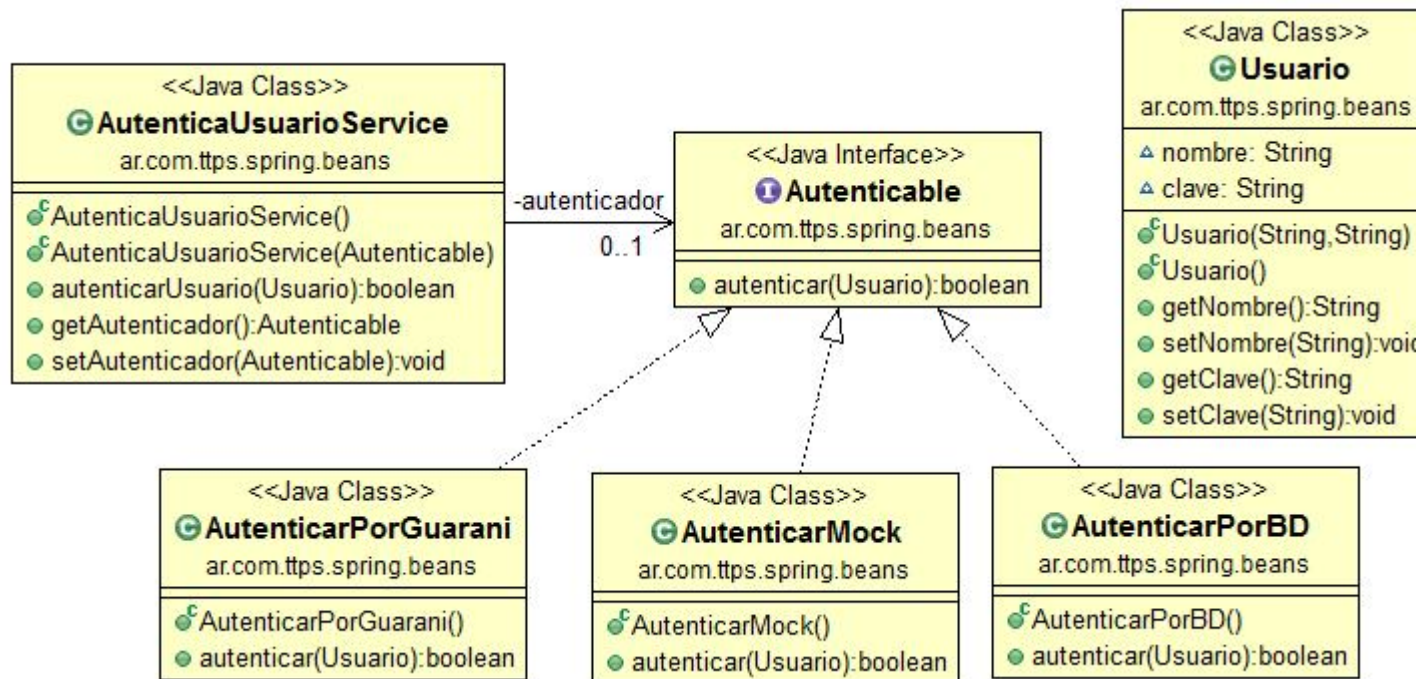
Inyección de Dependencias



Para ilustrar cómo definir los beans de nuestras aplicaciones y sus dependencias en Spring, vamos a resolver el siguiente ejercicio:

“Construya una clase `AutenticaUsuarioService`, que contenga un método `autenticarUsuario`, el cual recibe un objeto `Usuario` y retorna `true` si el usuario puede autenticarse o `false` en otro caso. El mecanismo por cual autenticar los datos debería ser fácilmente extensible.”

Modelo de clases



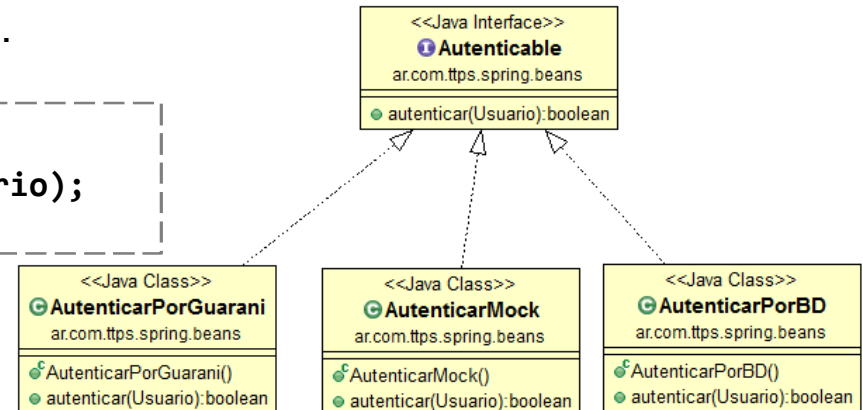
Spring

Inyección de Dependencias



La interface **Autenticable** sólo tiene un método para autenticar a un usuario. A modo de ejemplo se muestran dos implementaciones simples.

```
public interface Autenticable {  
    public abstract boolean autenticar(Usuario usuario);  
}
```



```
public class AutenticarPorGuarani implements Autenticable {  
    @Override  
    public boolean autenticar(Usuario usuario){  
        //Consume el servicio de SIU Guarani y autentica  
        System.out.println("... Se está autenticando a través de SIU Guarani...");  
        return true;  
    }  
}
```

```
public class AutenticarPorBD implements Autenticable {  
    @Override  
    public boolean autenticar(Usuario usuario){  
        //accede a la base de datos  
        System.out.println("... Se está autenticando por Base de Datos ...");  
        return false;  
    }  
}
```

Spring

Inyección de Dependencias



La clase `AutenticaUsuarioService` además del método `autenticarUsuario(Usuario usuario)` tiene una declaración de una variable de tipo `Autenticable`.

```
public class AutenticaUsuarioService {  
  
    private Autenticable autenticador;  
  
    public boolean autenticarUsuario(Usuario usuario){  
        System.out.println("Comienza el proceso de autenticación");  
        Boolean aut = this.getAutenticador().autenticar(usuario);  
        System.out.println("Finaliza el proceso de autenticación.");  
        return aut;  
    }  
    private Autenticable getAutenticador() {  
        return autenticador;  
    }  
    public void setAutenticador(Autenticable autenticar) {  
        this.autenticador = autenticar;  
    }  
}
```

Notar que el atributo `autenticador` no se instancia/inicializa en ninguna parte del código. Se inyecta un objeto automáticamente

Inyección por setter: el setter público del `autenticador` es requerido por Spring para realizar la inyección.

¿Cómo indicamos el objeto a inyectar?

Existen dos mecanismos para declarar beans e indicarle al Contenedor de Spring como manejar las dependencias: mediante XML o con anotaciones

Spring



Configuración basada en XML – Inyección de Dependencias

Comenzaremos usando un archivo XML para definir los beans y sus dependencias. Este archivo de configuración contiene el tag raíz **<beans>** dentro del cual se declaran cada uno de los beans usando el tag **<bean>**. El elemento **<bean>** es la unidad de configuración más básica en Spring, le dice que cree y administre un objeto.

applicatonContext.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:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.2.xsd">
    <!-- Definiciones de los beans aquí -->
</beans>
```

Ubicación

En un proyecto estándar, este archivo puede estar suelto en la raíz de los fuentes `/src` o adentro de un paquete, por ejemplo:

`ttps.spring`

Si usamos MAVEN este archivo se ubica en alguna de las carpetas cuyo contenido es enviado a la carpeta `/classes` en el WAR resultante.



Spring



Configuración basada en XML – Inyección de Dependencias

La siguiente configuración especifica dos beans para que el **ApplicationContext** cree y maneje. El primero es una instancia de **AutenticarPorGuarani**, el segundo es una instancia de **AutenticaUsuarioService**.

app-ctx.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" . . . >

  <bean id="autenticador" class="ar.com.ttps.spring.beans.AutenticarPorGuarani"/>

  <bean id="autUsuarioService" class="ar.com.ttps.spring.beans.AutenticaUsuarioService">
    <property name="autenticador" ref="autenticador"/>
  </bean>
</beans>
```

id: atributo con el cual se identifica al bean. Debe ser único en todo el Contenedor. Es opcional (beans internos)

property: elemento con el que se manejan los atributos en la clase. Se puede identificar el nombre y referenciar (**ref**) la dependencia o el valor (**value**) con el cual inicializar el atributo.

class: atributo que le indica a Spring de qué clase será el bean. Si no se especifica un constructor, Spring usará el constructor por defecto para instanciar la clase.

Le pedimos al **ApplicationContext** que encuentre un bean con el nombre (id) **autenticador** y que lo inyecte en el bean **autUsuarioService** usando su método setter.

Spring



Configuración basada en XML – Inyección por setter (1/3)

La inyección de dependencias por *setter* es un mecanismo de Spring para relacionar los *beans* unos con otros. Para que esto funcione, Spring invoca al método setter público de la propiedad, inmediatamente después de haber instanciado la clase a través de su constructor.

```
public class AutenticaUsuarioService {  
    private Autenticable autenticador;  
    public boolean autenticarUsuario(Usuario usuario){ }  
    public void setAutenticador(Autenticable autenticar) {  
        this.autenticador = autenticar;  
    }  
    . . .  
}
```

Si el método setter, no existe, Spring lanzará una excepción como la que se muestra a continuación y abortará el inicio de la aplicación.

Caused by: org.springframework.beans.NotWritablePropertyException: Invalid property 'mensaje' of bean class [ar.com.ttps.spring.beans.AutenticaUsuarioService]: Bean property 'mensaje' is not writable or has an invalid setter method. Does the parameter type of the setter match the return type of the getter?

app-ctx.xml

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
. . .
```

```
<bean id="autenticador" class="ar.com.ttps.spring.beans.AutenticarPorGuarani"/>
```

```
<bean id="autUsuarioService" class="ar.com.ttps.spring.beans.AutenticaUsuarioService">
```

```
    <property name="autenticador" ref="autenticador"/>
```

```
</bean>
```

```
</beans>
```

Implementa **Autenticable**

El objeto se contruye con el constructor por defecto y luego se hace **Inyección por Setter**

Spring



Configuración basada en XML – Inyección por setter (2/3)

Podemos probar que la aplicación ya está funcionando, para esto creamos una clase que tenga el método main. Dentro del main vamos a obtener el contexto de Spring y le vamos a pedir los beans para autenticar a un usuario.

```
public class Main {  
  
    public static void main(String[] args) {  
        ApplicationContext context = new ClassPathXmlApplicationContext("ar/com/ttps/resources/app-ctx.xml");  
        AutenticaUsuarioService autUsrService=context.getBean("autUsuarioService",AutenticaUsuarioService.class);  
        Usuario usuario = (Usuario)context.getBean("usuario");  
        System.out.println("La autenticacion fue: " + autUsrService.autenticarUsuario(usuario));  
    }  
}
```

Como lo tenemos configurado con una implementación para autenticador por Guarani (**AutenticarPorGuarani**), podemos ver que la salida por consola será la siguiente:

```
<terminated> Main (1) [Java Application] /usr/lib/jvm/java-7-openjdk-i386/bin/java (14/11/2013 01:10:29)  
INFO: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory  
Comienza el proceso de autenticación  
... Se esta autenticando a traves de SIU Guarani...  
Finaliza el proceso de autenticación.  
La autenticacion fue: true
```

Spring



Configuración basada en XML – Inyección por setter (3/3)

La interface **Autenticable** tiene otras implementaciones que podríamos intercambiar sin necesidad de tocar código Java ni de recompilar el código fuente, esto es una característica de Spring y de la configuración por XML. Sólo comentamos el bean anterior, y agregamos el **AutenticadorPorBD**.

app-ctx.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" . . . >

  <!-- bean id="autenticador" class="ar.com.ttps.spring.beans.AutenticarPorGuarani" / -->
  <bean id="autenticador" class="ar.com.ttps.spring.beans.AutenticarPorBD"/>

  <bean id="autUsuarioService" class="ar.com.ttps.spring.beans.AutenticaUsuarioService">
    <property name="autenticador" ref="autenticador"/>
  </bean>

</beans>
```

La salida será la siguiente:



```
<terminated> Main (1) [Java Application] /usr/lib/jvm/java-7-openjdk-i386/bin/java (14/11/2013 01:31:14)
INFO: Pre-instantiating Singletons in org.springframework.beans.factory.support.DefaultListableBean
Comienza el proceso de autenticación
... Se esta autenticando por BBDD ...
Finaliza el proceso de autenticación.
La autenticacion fue: false
```

Spring



Configuración basada XML – Inyección por constructor (1/3)

La Inyección de Dependencias por constructor es otro mecanismo que provee Spring para relacionar nuestros beans. Para demostrar esta técnica agregamos un constructor a la clase **AutenticaUsuarioService** como figura debajo:

```
public class AutenticaUsuarioService {  
    private Autenticable autenticador;  
    public AutenticaUsuarioService(Autenticable autenticador) {  
        this.autenticador = autenticador;  
    }  
    public boolean autenticarUsuario(Usuario usuario){ }  
    public void setAutenticador(Autenticable autenticar) {}  
}
```

Para que el bean **AutenticaUsuarioService** inicialice su atributo **autenticador** invocando un constructor, se debe utilizar tag **<constructor-arg>**

app-ctx.xml

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans" . . .  
    <bean id="autenticador" class="ar.com.ttps.spring.beans.AutenticarPorGuarani"/>  
    <bean id="autUsuarioService" class="ar.com.ttps.spring.beans.AutenticaUsuarioService">  
        <constructor-arg ref="autenticador"/>  
    </bean>  
</beans>
```

Con el atributo **ref**, referenciamos al bean que Spring pasará como parámetro al argumento del constructor. **Inyección por constructor**

Spring



Configuración basada XML – Inyección por constructor (2/3)

Ambigüedad en los parámetros

Supongamos que agregamos tres constructores mas a la clase `AutenticaUsuarioService`

```
public class AutenticaUsuarioService {  
    private Autenticable autenticador;  
    private String mensaje;  
    public AutenticaUsuarioService(Autenticable autenticar) {  
        this.autenticador = autenticar;  
    }  
    public AutenticaUsuarioService(Autenticable autenticar, String mensaje) {  
        this.autenticador = autenticar;  
        this.mensaje = mensaje;  
    }  
    public AutenticaUsuarioService(Autenticable autenticar, Integer token) {  
        this.autenticador = autenticar;  
    }  
    public AutenticaUsuarioService(Autenticable autenticar, Integer token, String mensaje) {  
        this.autenticador = autenticar;  
        this.mensaje = mensaje;  
    }  
    public boolean autenticarUsuario(Usuario usuario){ }  
    public void setAutenticador(Autenticable autenticar) {}  
}
```



Configuración basada XML – Inyección por constructor (3/3)

Ambigüedad en los parámetros

Dependiendo de la configuración de los beans en el ApplicationContext, se invocarán a diferentes constructores:

Se invoca al segundo constructor que recibe un objeto Autenticable y un String

```
<bean id="autUsuService" class="ar.com.ttps.spring.beans.AutenticaUsuarioService">
    <constructor-arg ref="autenticador" />
    <constructor-arg value="123456" />
</bean>
```

Se invoca al tercer constructor que recibe un objeto Autenticable y un Integer

```
<bean id="autUsuService" class="ar.com.ttps.spring.beans.AutenticaUsuarioService">
    <constructor-arg ref="autenticador" />
    <constructor-arg value="123456" type="java.lang.Integer" />
</bean>
```

Se invoca al último constructor que recibe un objeto Autenticable , un Integer y un String

```
<bean id="autUsuService" cla4="ar.com.ttps.spring.beans.AutenticaUsuarioService">
    <constructor-arg ref="autenticador" />
    <constructor-arg value="123456" type="java.lang.Integer"/>
    <constructor-arg value="Un Mensaje" />
</bean>
```



Inyección de Dependencias – Configuración por XML

El tag **<bean>** contiene toda la información para que el contenedor sepa cómo crear el bean, manejar su ciclo de vida y sus dependencias. La siguiente tabla detalla los atributos que soporta el tag **<bean>**:

Nombre	Tipo	Descripción
id	atributo	Identificador único de un Bean
class	atributo	El nombre completo de la clase del bean, excepto si sólo sirve como definición padre (<code>abstract=true</code>).
scope	atributo	El alcance del bean: por lo general es " singleton " (una instancia compartida, que será devuelta por todas las llamadas a <code>getBean</code> con el id indicado), o " prototype " (crea una instancia nueva por cada llamada a <code>getBean</code>). De forma predeterminada, un bean será un singleton.
init-method	atributo	Método que será invocado después de inicializar el bean. El método no debe tener argumentos.
destroy-method	atributo	Idem al anterior, pero al momento de eliminar el bean del contenedor.
abstract	atributo	Indica que el bean no debe instanciarse, y sirve para inicializar beans hijos.
parent	atributo	Especifica cuál es el bean padre del bean actual, a partir del cual se heredan datos de configuración.
property	elemento	Corresponde a los métodos <i>setter</i> públicos especificados por la clase Java. Spring soporta primitivos, referencia a otros beans, listas, arrays, mapas y props.
constructor-arg	elemento	Define un parámetro del constructor de la clase. Los argumentos se pueden indicar por índice (orden en el constructor), tipo o nombre. Se crea un elemento nuevo por cada argumento del constructor.

Spring

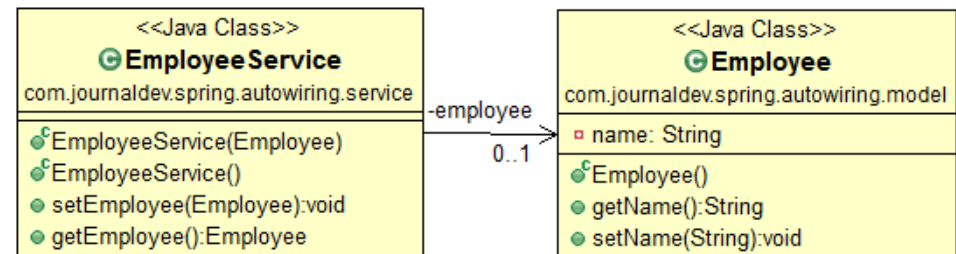


Configuración basada en Anotaciones

A partir de la versión 3 de Spring, se pueden configurar los beans y sus dependencias a través de anotaciones. De esta manera los beans y sus dependencias no se especifican en el archivo de configuración XML. Spring usa dos mecanismos basados en anotaciones para descubrir y vincular beans:

- **Autodiscovery** que evita la declaración de cada bean con el tag `<bean>`
- **Autowiring** que permite eliminar la necesidad de configurar `<property>` o `<constructor-arg>` en las declaraciones de los beans.

Para simplificar los ejemplos con anotaciones definimos dos clases simples: **EmployeeService** que tiene una referencia a un objeto **Employee**.



```
package ttps.spring.autowiring;

public class EmployeeService {
    private Employee employee;

    public EmployeeService(Employee empl) {
        System.out.println("Constructor con Employee en EmployeeService");
        this.employee = empl;
    }

    public EmployeeService() {
        System.out.println("Constructor Default EmployeeService");
    }

    public void setEmployee(Employee empl) {
        this.employee = empl;
    }

    public Employee getEmployee() {
        return this.employee;
    }
}
```

```
package ttps.spring.autowiring;

public class Employee {
    private String name;

    public Employee() {
        System.out.println("Constructor Default Emp");
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Spring

Configuración basada en Anotaciones



Spring pone en práctica dos mecanismos **Autodiscovery** y **Autowiring**. Para que esto suceda debemos indicarle a Spring qué clases debe usar para crear beans y cómo inyectar sus dependencias.

Mediante la anotación **@Component** le podemos indicar a Spring a partir de qué clases crear beans. El nombre del bean lo definimos usando **@Component("emple")** (antes **<bean="emple">**) o simplemente **@Component**.

A través de la anotación **@Autowired**, que puede ubicarse sobre un atributo, método o constructor, Spring determina un "bean candidato" para ser inyectado en la propiedad. Se puede especificar que el candidato sea determinado a partir del tipo de la propiedad (**byType**), del nombre (**byName**) o **automáticamente**.

```
@Component
public class EmployeeService {
    private Employee employee;

    @Autowired
    public EmployeeService(Employee empl) {
        System.out.println("Constr. con Employee en EmployeeService");
        this.employee = empl;
    }

    public EmployeeService() {}

    public void setEmployee(Employee empl) {...}

    public Employee getEmployee() {...}
}
```

Inyección por constructor

Según esta definición, estamos indicando **inyección por constructor**. También podría cambiarse la anotación arriba del método setter para indicar **inyección por setter**.

```
@Component
public class Employee {
    private String name;
    public Employee() {
        System.out.println("Constr. Default Emp");
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name=name;
    }
}
```

Spring instanciará un objeto de tipo **Employee** para pasarlo al constructor que fue anotado con **@Autowired**. Si anotamos el método setter con **@Autowired** entonces usará el constructor por defecto, instanciará un objeto de tipo **Employee** y se lo pasará al setter

Spring

Configuración basada en Anotaciones



Para que todo funcione, Spring aun necesita de un archivo XML de configuración mínimo. Cuando usamos configuracion basada en XML definimos todo adentro de este archivo, cuando se definen usando anotaciones en el fuente Java, un XML de configuración mínimo sigue siendo necesario.

Para que Spring tome la configuración por anotaciones debemos incorporar el tag `<context:annotation-config/>` e indicarle los paquetes en donde buscar clases anotadas usando el tag `<context:component-scan base-package= "nombre del paquete">`

app-ctx.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:context=http://www.springframework.org/schema/context
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.2.xsd"
  default-autowire="byName">
  <!-- Habilitamos configuración basada en anotaciones -->
  <context:annotation-config/>
  <context:component-scan base-package="tps.spring.autowiring" />
</beans>
```

Spring

Configuración basada en Anotaciones



Autowiring por nombre (*default-autowire="byName"*)

El autowiring por nombre trabaja buscando el **id** o **name** del bean manejado por Spring con nombres de atributos/métodos setters expuestos por objetos del framework.

```
@Component
public class EmployeeService {
    private Employee emple; //nombre emple
}
// usado por autowiring x constructor
public EmployeeService(Employee empl) {
    System.out.println("Constructor con Employee en EmployeeService");
    this.emple = empl;
}
// usado por autowiring x setter (ByName o ByType)
public EmployeeService() {
    System.out.println("Constructor Default EmployeeService");
}
// usado por autowiring x setter (ByName o ByType)
@Autowired
public void setEmple(Employee empl) {
    this.emple = empl;
    System.out.println("Setter en EmployeeService");
}
public Employee getEmployee() {
    return this.emple;
}
}
```

Si no se pone nombre **@Component**, Spring le pondrá como nombre el de la clase.

```
@Component("emple")
public class Employee {
    private String name;
    public Employee() {
        System.out.println("Constructor Default Emp");
    }
    public String getName() { return name; }
    public void setName(String name) {this.name=name;}
}
```

Con estas definiciones, Spring buscará en su contexto algún bean con **name** o **id** **"emple"**, si lo encuentra lo inyecta en **EmployeeService**.

La inyección **byName** trata de coincidir el nombre del atributo/del setter con un bean con ese nombre o id en el contexto.

Spring

Configuración basada en Anotaciones



Autowiring por tipo (*default-autowire="byType"*)

El autowiring por tipo trabaja buscando el tipo del objeto en alguno de los beans definidos dentro del contexto. Si lo encuentra lo usa. Si hay más de uno o no lo encuentra, da un error.

```
@Component
public class EmployeeService {
    private Employee emple; //tipo Employee
    // usado por autowiring x constructor
    @Autowired
    public EmployeeService(Employee empl) {
        System.out.println("Constructor con Employee en EmployeeServ");
        this.emple = empl;
    }

    // usado por autowiring x setter (ByName o ByType)
    public EmployeeService() {
        System.out.println("Constructor Default EmployeeService");
    }

    // usado por autowiring x setter (ByName o ByType)
    public void setEmple(Employee empl) {
        this.emple = empl;
        System.out.println("Setter en EmployeeService");
    }

    public Employee getEmployee() {
        return this.emple;
    }
}
```

```
@Component("emple")
public class Employee {
    private String name;
    public Employee() {
        System.out.println("Constructor Default Emp");
    }
    public String getName() { return name; }
    public void setName(String name) {this.name=name;}
}
```

Solo cambiando en el archivo de configuración ***default-autowire="byType"*** este ejemplo sigue funcionando con otra lógica.

La inyección ***byType*** trata de coincidir el tipo del atributo/o del parámetro del setter o el parámetro del constructor con el tipo de un bean en el contexto.