El Framework Spring

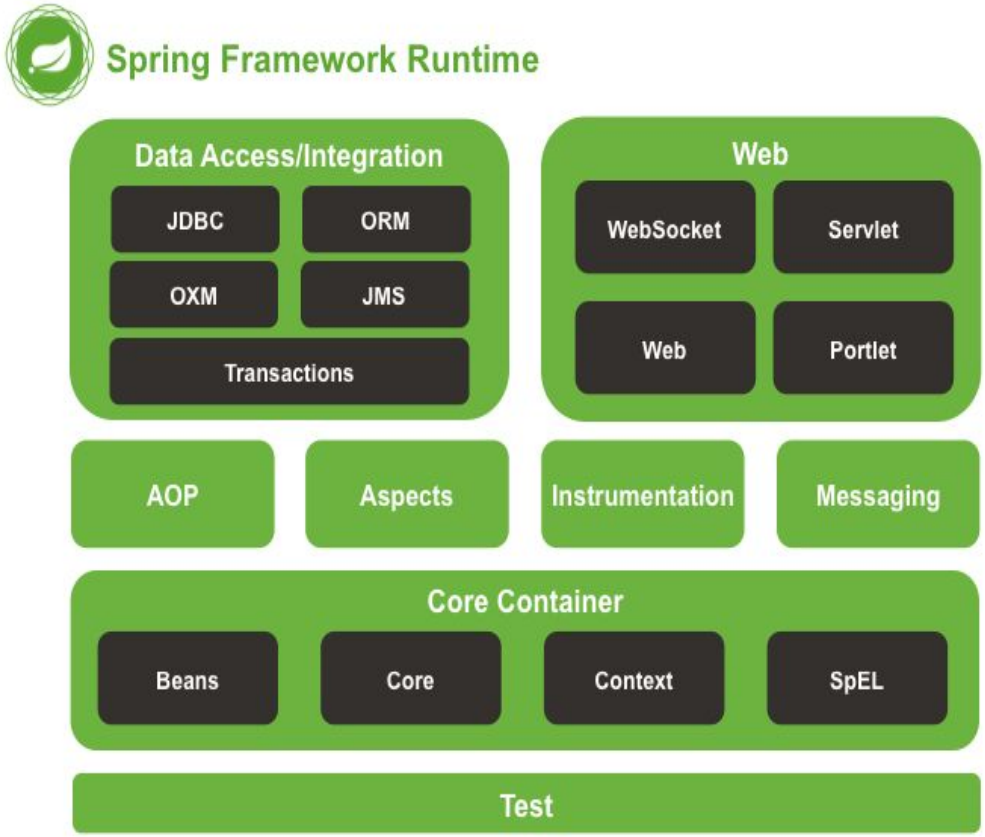
- Evolución del framework Spring
- Arquitectura del Framework
- Inversión de Control (IoC) e Inyección de Dependencias (DI)
- Librerías básicas necesarias
- Ejemplo práctico para entender los conceptos
- Configuración por XML, los tags **`<bean>`** y **`<beans>`**.
- Configuración por anotaciones
- Integración de Spring con JPA

¿Qué es Spring?

- Spring es un framework *open source*, creado por Rod Johnson, descripto en su libro: *Expert One-on-One: J2EE Design and Development*. Spring fue creado para abordar la complejidad del desarrollo de aplicaciones java empresariales, que 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 la misma.
- La primera versión fue lanzada formalmente bajo la licencia Apache 2.0 en 2004. A partir de ahí se sucedieron diferentes versiones con mejoras.
- Las ideas "innovadoras" que en su día popularizó Spring se han incorporado en la actualidad a las tecnologías y herramientas estándares.
- Spring Framework: `https://spring.io/projects/spring-framework`

Arquitectura de Spring
Módulos que lo componen

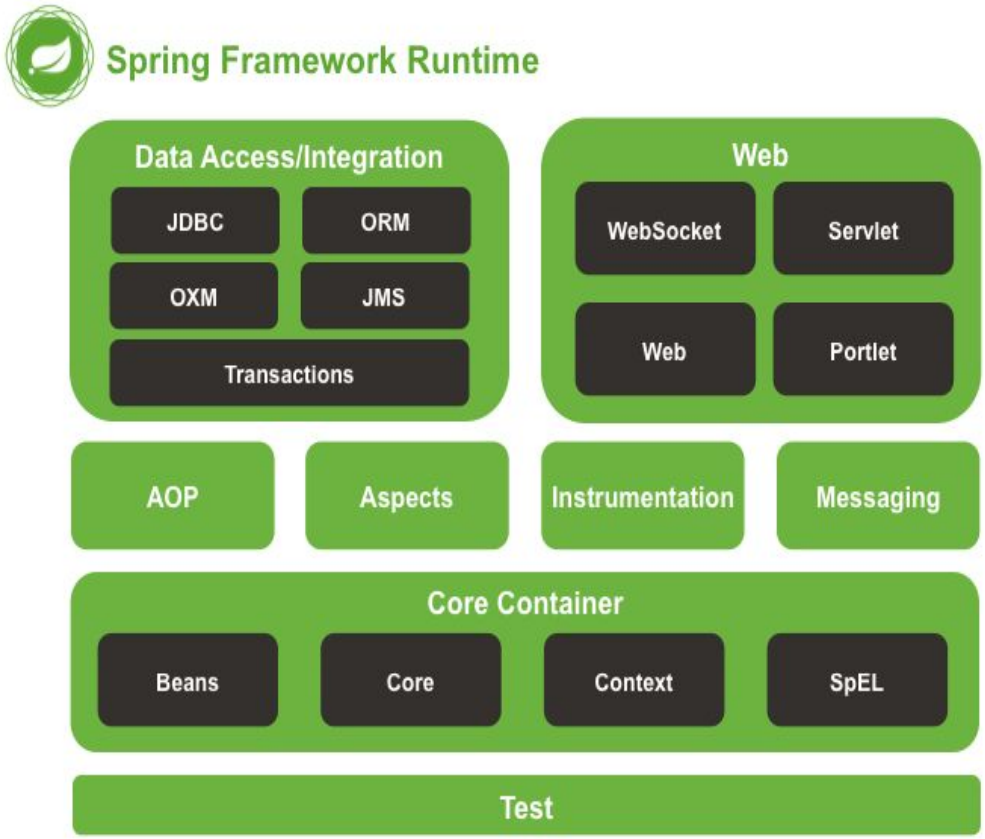
El framework Spring consta de funciones organizadas en más de 20 módulos. Estos módulos se agrupan en:



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



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

Programación Orientada a Aspectos (AOP): 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 objetos Mock que pueden ser usados para testear la aplicación.

Spring
Conceptos importantes

Inversión de Control e Inyección de Dependencias

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

- La Inversión de Control (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 inyección de dependencias (DI) es una forma especializada de IoC, mediante la cual los objetos definen sus dependencias (es decir, los otros objetos con los que trabajan) a través de argumentos de constructor, argumentos para un método setter o propiedades del objeto. El contenedor de IoC crea el bean y lo inyecta.

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 y de conectar sus dependencias a partir del código de la aplicación.**

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.

Se conoce como **Inyección de Dependencia**
al mecanismo de Spring para relacionar un bean con otro.

Spring

Contenedores de Inversión de Control y Beans

¿Qué es un Contenedor IoC Spring?

Un contenedor IoC (Inversión de Control) en Spring es el componente central del framework encargado de gestionar la creación, configuración y ciclo de vida de los objetos, así como de inyectar sus dependencias de manera automática, leyendo de metadatos de configuración de XML, anotaciones Java y código Java en archivos de configuración. Es la componente que gestiona los objetos de la aplicación.

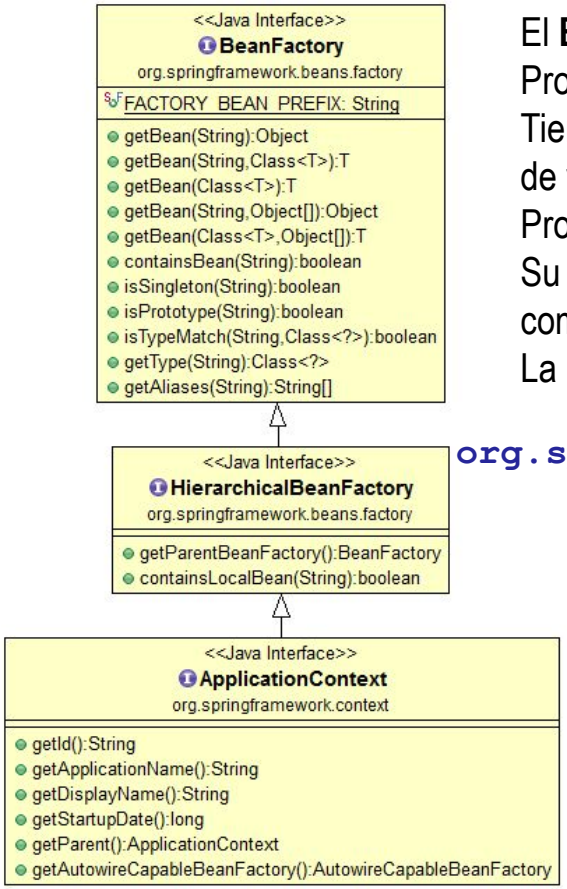
¿Qué es un Bean en Spring?

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`**

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.
Provee 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:

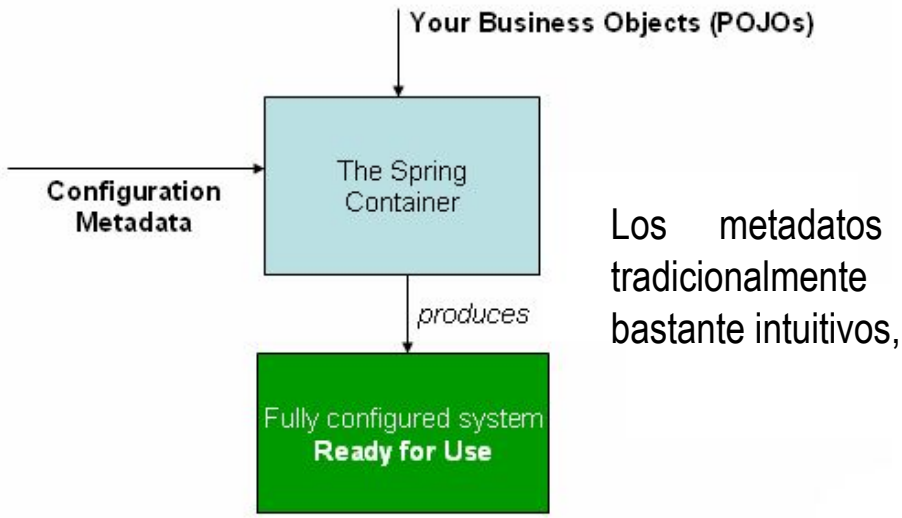
`org.springframework.beans.factory.ClassPathXmlApplicationContext`

El **ApplicationContext** es una especialización de **BeanFactory**.
Los contenedores Spring que implementan esta interfaz son más avanzados que los que implementar BeanFactory.
Esta interfaz agrega características de POA, manejo de mensajes textuales desde archivos de propiedades (i18n), integración con el entorno web.
Existen varias implementaciones de ApplicationContext como:

`ClassPathXmlApplicationContext`
`FileSystemXmlApplicationContext`
`XmlWebApplicationContext`
`AnnotationConfigWebApplicationContext`

Spring
Contenedores de IoC - Configuración

El contenedor de Spring hace uso de metadatos de configuración. Estos metadatos le especifican al contenedor de Spring cómo crear instancias, configurar y ensamblar los objetos de la aplicación.



Los metadatos de configuración han sido tradicionalmente especificados en archivos XML bastante intuitivos, pero actualmente hay alternativas.

Los metadatos de configuración pueden especificarse de diferentes maneras:

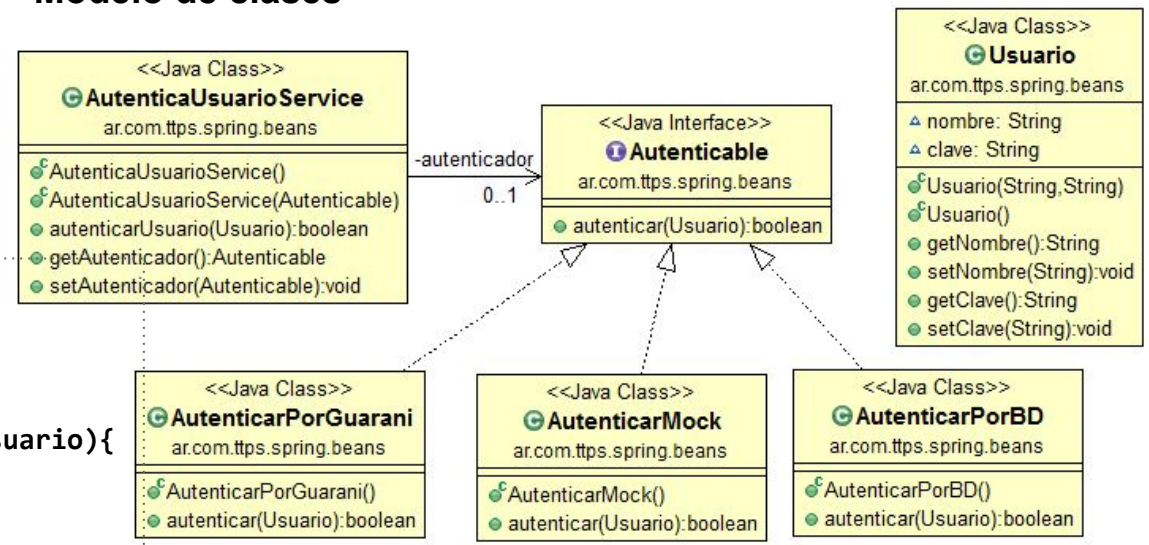
- **Configuración mediante archivos xml:** tradicionalmente se han utilizado archivos XML, los cuales brindan una manera bastante intuitiva para crear y vincular objetos de la aplicación.
- **Configuración basada en anotaciones:** a partir de la versión **Spring 2.5** se introdujo soporte para metadatos de configuración basado en anotaciones como **`@Component`** y **`@Autowired`**, más algunos archivos XML.
- **Spring 3+**, incorpora otras anotaciones como **`@Configuration`**, **`@Bean`**, **`@Import`** y **`@DependsOn`** que permiten evitar el uso de archivos XML.

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



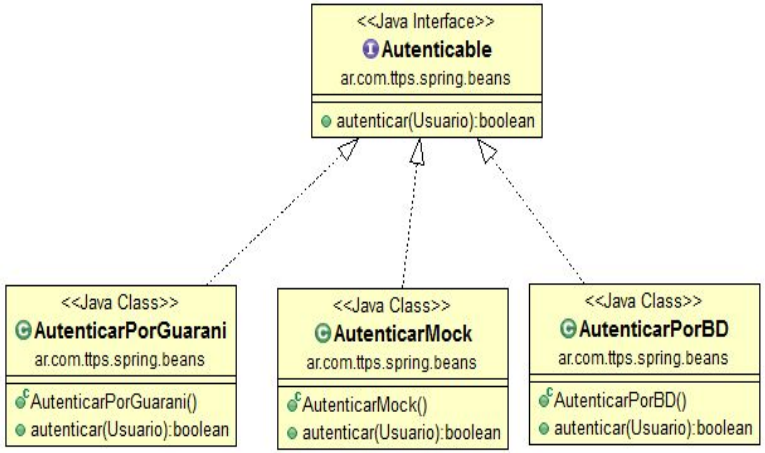

```
public class AutenticaUsuarioService {

  private Autenticable autenticador;

  public boolean autenticarUsuario(Usuario usuario){
       ...
      autenticador.autenticar(usuario);
   }
   ...
}
```

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 = autenticador.autenticar(usuario);
     System.out.println("Finaliza el proceso de autenticación.");
     return aut;
   }

   . . .
}
```

Notar que el atributo **autenticador** no se instancia/inicializa en ninguna parte del código. **Spring inyectará un objeto automáticamente**

¿Cómo indicamos el objeto a inyectar?

Por XML, anotaciones o basado en Java.

Spring

Inyección de Dependencias - Configuración basada en XML

La siguiente configuración define dos beans para que los cree y maneje el contenedor de Spring. 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.

`property`: elemento con el que se manejan los atributos en la clase. Se debe 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 a Spring 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 usando métodos setter. 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 'autenticador' is not writable** or has an invalid setter method. Does the parameter type of the setter match the return type of the getter?

```
<?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**

app-ctx.xml

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) {
   // crea y configura los beans
   ApplicationContext context = new ClassPathXmlApplicationContext("resources/app-ctx.xml");
   //recupera instancias configuradas
   AutenticaUsuarioServiceautUsrService =
                                  context.getBean("autUsuarioService",AutenticaUsuarioService.class);
   Usuario usuario = new Usuario("user01", "passUser01");
   //usa instancia recuperada
   System.out.println("La autenticacion fue: " + autUsrService.autenticarUsuario(usuario));
  }
 }
```

El String del path es a partir de la carpeta de fuentes /src

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

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.xm

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

Ahora lo tenemos configurado con una implementación para autenticador por base de datos (**AutenticarPorBD**, podemos ver que la salida por consola será la siguiente:

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.

Spring

Configuración basada en Anotaciones

Spring 2.5+

Spring
Configuración basada en Anotaciones

A partir de la versión 2.5 de Spring, se pueden configurar los beans y sus dependencias a través de anotaciones. 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 reemplaza la configuración `<property>` o `<constructor-arg>` en las declaraciones de los beans.

```
ttps.demo;

public class EmployeeService {
 private EmployeeDAO employeeDAO;

 public EmployeeService(EmployeeDAO empl){
   this.employeeDAO = empl;
 }
 public Employee findEmployee(..) {
     return this.employeeDAO.findById(..)
 }

}
```

```
ttps.demo;
public interface EmployeeDAO {
    Employee findById();
    Employee create(Employee emp);
    . . .
}
```

```
ttps.demo;
public class EmployeeDAOImpl implements
                                EmployeeDAO {
   @Override
   Employee findById(){...}

   @Override
   Employee create(Employee emp){...}
}
```

Spring
Configuración basada en Anotaciones

Para que Spring ponga en práctica los mecanismos de Autodiscovery y Autowiring se definieron dos anotaciones:

- **@Component** para indicarle a Spring a partir de qué clases crear beans. Opcionalmente se puede pasar el nombre del bean **@Component("emple")** (antes <bean=“emple”>).
- **@Autowired**, para indicarle a Spring que determine el ”bean candidato” para ser inyectado. Por configuración, se puede especificar que el candidato sea determinado a partir del tipo de la propiedad (**byType**), del nombre (**byName**). La anotación `@Inject` de JSR 330 se puede usar en lugar de la anotación `@Autowired` de Spring.

Cuando se usa anotaciones como @Autowired en Spring para la inyección de dependencias, **no se necesita obligatoriamente tener un setter o un constructor explícito**, ya que Spring puede inyectar las dependencias directamente con los campos (field injection).

Hay 3 tipos de inyección:

1. Field injection: inyección de Dependencias en Campos
2. Constructor Injection: inyección de Dependencias con Constructor
3. Setter injection: inyección de Dependencias por método Setter

Spring

Configuración basada en Anotaciones

1. Inyección de Dependencias en campos (Field Injection): Esta es la forma más simple de inyección, la anotación va directamente sobre el campo donde se desea inyectar la dependencia.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@component
public class EmployeeService {
 @Autowired
 private EmployeeDAO employeeDAO;


 public Employee findEmployee(..) {
     return this.employeeDAO.findById(..)
 }
 . . .
}
```

```
@component
public class EmployeeDAOImpl implements
                                EmployeeDAO {
   @Override
   Employee findById(){...}

   @Override
   Employee create(Employee emp){...}
}
```

- No se necesita ni constructor ni setter. Spring inyectará por reflection la dependencia directamente en el campo marcado con `@Autowired`.
- Es la forma más simple, pero algunos la consideran menos deseable porque no es fácilmente testeable.

Spring

Configuración basada en Anotaciones

2. Inyección de Dependencias con Constructor (constructor Injection): Esta es la forma más simple de inyección, la anotación va directamente sobre el campo donde se desea inyectar la dependencia.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@component
public class EmployeeService {
 private EmployeeDAO employeeDAO;
 @Autowired
 public EmployeeService(EmployeeDAO empl){
   this.employeeDAO = empl;
 }
 public Employee findEmployee(..) {
     return this.employeeDAO.findById(..)
 }
}
```

```
@component
public class EmployeeDAOImpl implements
                                EmployeeDAO {
    @Override
    Employee findById(){...}

    @Override
    Employee create(Employee emp){...}
}
```

A partir de Spring Framework 4.3, ya no es necesario usar la anotación `@Autowired` en un constructor si el bean tiene solo un constructor. Sin embargo, si hay varios constructores disponibles y no hay un constructor principal o por defecto, al menos uno de ellos debe estar anotado con `@Autowired` para indicar al contenedor cuál debe utilizar.

Spring
Configuración basada en Anotaciones

3. Inyección de Dependencias por método Setter (Setter Injection): Se debe definir un método setter y anotarlo con @Autowired. Spring invocará este método para inyectar la dependencia.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@component
public class EmployeeService {
 private EmployeeDAO employeeDAO;

 public EmployeeService(EmployeeDAO empl){
   this.employeeDAO = empl;
 }

 @Autowired
 public void setEmployeeDAO(EmployeeDAO empl){
   this.employeeDAO = empl;
 }

  . . .
}
```

```
@component
public class EmployeeDAOImpl implements
                                EmployeeDAO {
   @Override
   Employee findById(){...}

   @Override
   Employee create(Employee emp){...}
}
```

Es útil si se necesitan cambiar las dependencias durante la vida del objeto o si se trabaja con frameworks de pruebas que permiten modificar dependencias después de la creación del objeto. Es menos usada que la inyección por constructor, pero es válido si existen razones para cambiar la dependencia dinámicamente.

Configuración basada en Anotaciones
autowire “byType”

Inyección por Tipo (ByType) → Es el comportamiento por defecto

En Spring, cuando se realiza la inyección de dependencias mediante @Autowired, el **modo por defecto** es la **inyección por tipo**. Esto significa que Spring intenta encontrar un bean en el contexto que coincida con el tipo de la dependencia que se va a inyectar.
Busca un bean que coincida con el **tipo** de la variable, parámetro del constructor o método setter a inyectar.

- Si hay **exactamente un bean** del tipo requerido, Spring lo inyecta sin problemas.
- Si hay **más de un bean** del mismo tipo, Spring lanza una excepción a menos que utilices un mecanismo para desambiguar (como `@Qualifier`).
- Si no encuentra ningún bean del tipo requerido, también lanzará una excepción, a menos que hayas indicado que es opcional con `required=false`.

```
@component
public class EmployeeService {
 private EmployeeDAO employeeDAO;
  . . .
}
```

Inyecta por tipo EmployeeDAO

Configuración basada en Anotaciones
autowire y ambigüedades

El autowiring por tipo trabaja buscando el tipo del objeto en alguno de los beans definidos dentro del contexto. ¿Qué pasa si encuentra más de un bean candidato?

```
public interface EmployeeDAO {
    Employee findById();
    Employee create(Employee emp);
    . . .
}
```

```
@Component(“employeeDAOJPA”)
public class EmployeeDAOJPA implements EmployeeDAO{
    @Override
    Employee findById(){...}
    @Override
    Employee create(Employee emp){...}
}
```

```
@Component(“employeeDAOJDBC”)
public class EmployeeDAOJDBC implements EmployeeDAO{
    @Override
    Employee findById(){...}
    @Override
    Employee create(Employee emp){...}
}
```

¿Hay dos beans de tipo EmployeeDAO, cuál inyecta?

Este código dispara la excepción NoUniqueBeanDefinitionException

```
@Component
public class EmployeeService {

    @Autowired
    private EmployeeDAO empDAO;

    public EmployeeService(EmployeeDAO empl) {
      this.empDAO = empl;
    }
    ...
}
```

La anotación @Qualifier

```
@Component
public class EmployeeService {

    @Autowired
    @Qualifier("employeeDAOJPA")
    private EmployeeDAO empDAO;
    ...
}
```

Spring
@Scopes - Alcance de lo beans

Singleton vs Prototype

Singleton
Cuando definimos un bean con scope "*singleton*", Spring creará una instancia única de la clase. Esa misma instancia será reutilizada y compartida para todos los demás beans que la requieran.

```
Component
@Scope(scopeName = ConfigurableBeanFactory.SCOPE_SINGLETON)
public class EmployeeDAOImpl implements EmployeeDAO {. . }
```

Prototype
Cuando definimos un bean con scope "*prototype*", Spring creará una nueva instancia de la clase cada vez que tenga que inyectar el bean como dependencia de otro bean.

```
Component
@Scope(scopeName = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public class EmployeeDAOImpl implements EmployeeDAO {. . }
```

Tambien existen los alcances request, session y application con la misma semántica de los alcances de JEE. Mas detalles: https://docs.spring.io/spring-framework/reference/core/beans/factory-scopes.html

Spring

Configuración basada en Anotaciones

En las versiones anteriores a Spring 3, el autowiring por defecto era: NO. Por ello era necesario configurar byType o ByName para que se efectivice el *autowiring*. La única manera de hacerlo era por XML.

Para que Spring tome la configuración por anotaciones se debía 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-2.5.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context-2.5.xsd“
  default-autowire="byType">
  <!-- Habilitamos configuración basada en anotaciones -->
  <context:annotation-config/>
  <context:component-scan base-package=“ttps.spring.autowiring" />
  …
 <!-- DataSource -->
 <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource" destroy-method="close">
      <property name="driverClass" value="com.mysql.jdbc.Driver"/>
      <property name="jdbcUrl" value=“jdbc:mysql://localhost:3306/cartelera"/>
      <property name="user" value="root"/>
      <property name="password" value="root"/>
  </bean>
  . . .
</beans>
```

Spring

Configuración basada en Anotaciones

Para probar la generación de Beans y el cableado se lo puede hacer creando un contexto de Spring para aplicaciones de escritorio como **AnnotationConfigApplicationContext.** Esta clase acepta como entrada, paquetes en donde buscar clases anotadas; también se pueden registrar beans con register().

```
public class Main {

public static void main(String[] args) {
   ApplicationContext context = new AnnotationConfigApplicationContext("ttps.demo");
   EmployeeService employeeService = context.getBean(EmployeeService.class);
   EmployeeService.getEmployeeDAO();
      . . .
  }
}
```

Spring

Nuevas anotaciones para evitar el archivo de configuración XML

Spring 3+

Spring
Configuración basada en Anotaciones (1/2)

A partir de la versión 3 de Spring, se cuenta con nuevas anotaciones para facilitar la configuración del contexto de Spring y de los beans.
La anotación de Spring **@Configuration** es parte del framework Spring core e indica que es una clase de configuración. La anotación **@Bean** sobre un método le indica a Spring que el método retornará un objeto que debe ser registrado como un **bean** en el contexto de la aplicación Spring.

```
import javax.sql.DataSource;
import jakarta.persistence.EntityManagerFactory;
import org.springframework.context.annotation.Bean;
```

Cuando una clase se anota con **@Configuration** indica que la clase puede ser usada por el contenedor Spring IoC como fuente de definiciones de beans.

```
@Configuration   // marca la clase como de configuracion
@EnableTransactionManagement   //habilita la anotacion @Transactional
@ComponentScan(basePackages="ttps.spring")
public class PersistenceConfig {
     @Bean
     public DataSource dataSource() {
          DriverManagerDataSource driverManagerDataSource = new DriverManagerDataSource();
          driverManagerDataSource.setUsername("root");
          driverManagerDataSource.setPassword("r@@t");
          driverManagerDataSource.setUrl("jdbc:mysql://localhost:3306/buffet");
          driverManagerDataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
          return driverManagerDataSource;
     }
     . . .
}
```

Spring
Configuración basada en Anotaciones (2/2)

```
import javax.sql.DataSource;
import jakarta.persistence.EntityManagerFactory;
import org.springframework.context.annotation.Bean;

@Configuration   // marca la clase como de configuracion
@EnableTransactionManagement   //habilita la anotacion @Transactional
@ComponentScan(basePackages="ttps.spring")
public class PersistenceConfig {
    . . .
 @Bean
 public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
   LocalContainerEntityManagerFactoryBean emf = new LocalContainerEntityManagerFactoryBean();
   emf.setDataSource(dataSource());
   emf.setPackagesToScan("ttps.spring");
   emf.setEntityManagerFactoryInterface(jakarta.persistence.EntityManagerFactory.class);
   JpaVendorAdapter jpaVendorAdapter = new HibernateJpaVendorAdapter();
   emf.setJpaVendorAdapter(jpaVendorAdapter);
   return emf;
 }
  @Bean
  public JpaTransactionManager transactionManager(EntityManagerFactory emf) {
     JpaTransactionManager transactionManager = new JpaTransactionManager();
     transactionManager.setEntityManagerFactory(emf);
     return transactionManager;
  }
}
```

Configuración basada en Anotaciones

`@Component` vs `@Bean`

`@Component` se utiliza para detectar y configurar automáticamente beans mediante el escaneo de classpath. Hay una asignación implícita entre la clase anotada y el bean, es una anotación a nivel de clase. Preferible para escaneo de componentes y vinculación/cableado automático.

`@Bean` se usa para declarar explícitamente un bean. Desacopla la declaración del bean de la definición de clase, es una anotación a nivel de método para crear y configurar beans de otra forma.

A veces, la configuración automática no es una opción. Por ejemplo cuando se desea conectar componentes de bibliotecas de terceros (no se tiene el código fuente, por lo que no puede anotar con @Component) y no es posible la configuración automática.

Spring
Para probar los DAO con Spring desktop

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import main.java.ttps.spring.model.MiPrimerEntity;
import main.java.ttps.spring.config.PersistenceConfig;
import javax.sql.DataSource;

public class SpringTestMain {
  public static void main(String[] args) {
     // Create a new AnnotationConfigApplicationContext
     AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
     // Registra la clase de configuration (PersistenceConfig)
     ctx.register(PersistenceConfig.class);
     // Refresca para actualizar la creacion de beans
     ctx.refresh();
     // recupera beans
     MiPrimerEntity e = ctx.getBean("prueba", MiPrimerEntity.class);
     System.out.println(e.hola());
     DataSource dataSource = ctx.getBean(DataSource.class);
     System.out.println("DataSource: " + dataSource);
     // cierra el contexto para liberar recursos
     ctx.close();
  }
}
```

Spring
Mas anotaciones

@PersistenceContext

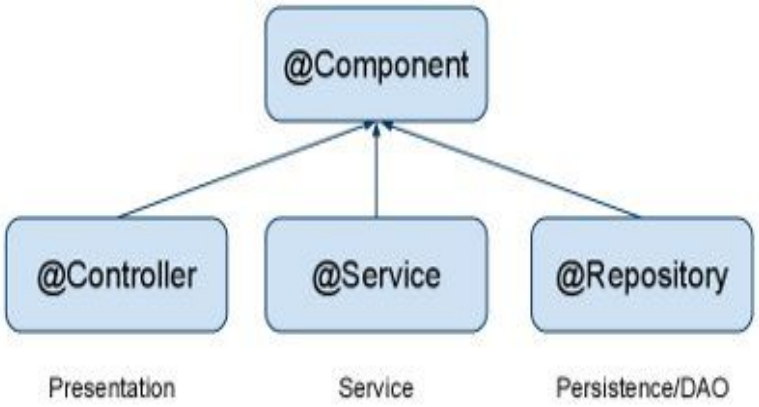
Esta anotación se usa para requerir Inyección de Dependencias del **EntityManagerFactory** default. Con esta anotación le indicamos a Spring que inyecte y maneje un EntityManager.

@Transaccional

Spring se encarga de realizar el manejo transaccional a través de la anotación **@Transaccional**. Con esta anotación en la clase, indicamos que todos los métodos son transaccionales.

@Component

Es un estereotipo de uso general, indica que la clase es un bean de Spring.



@Controller

Es un estereotipo que tiene el rol de Controller.

@Service

Mantiene la lógica de negocios e invoca a métodos de la capa de datos.

@Repository

Para implementaciones de los DAO. Entre los usos de este marcador está la traducción automática de excepciones.

Spring

¿Cómo afectan estas anotaciones en nuestro DAO?

```
public class GenericDAOHibernateJPA<T>
                        implements GenericDAO<T> {
. . .
@Override
public T persistir(T entity) {
    EntityManager em=EMF.getEMF().createEntityManager();
    EntityTransaction tx = null;
    try {
     tx = em.getTransaction();
     tx.begin();
     em.persist(entity);
     tx.commit();
    } catch (RuntimeException e) {
       if (tx != null && tx.isActive() )
         tx.rollback();
         throw e;
    } finally {
         em.close();
    }
  return entity;
 }
 . . .
}
```

```
@Transactional
public class GenericDAOHibernateJPA<T>
                                  implements GenericDAO<T> {

 @PersistenceContext
 private EntityManager entityManager;

 public void setEntityManager(EntityManager em){
    this.entityManager = em;
 }
 public EntityManager getEntityManager() {
    return entityManager;
 }

 @Override
 public T persistir(T entity) {
   this.getEntityManager().persist(entity);
   return entity;
 }

 . . .

}
```

`@Transaccional` por defecto abre y cierra una transacción por método. Pero!!! si desde ese método se invoca a un método de otra capa que también tiene solo `@Transaccional`, se usa la misma transacción y todo sigue funcionando.

Spring

¿Cómo afectan estas anotaciones en nuestro DAO?

```
@Transactional
public class GenericDAOHibernateJPA<T>
                                    implements GenericDAO<T>{

 @PersistenceContext
 private EntityManager entityManager;

 public void setEntityManager(EntityManager em){
    this.entityManager = em;
 }
 public EntityManager getEntityManager() {
    return entityManager;
 }

 protected Class<T> persistentClass;
 public GenericDAOHibernateJPA(Class<T> persistentClass) {
    this.setPersistentClass(persistentClass);
 }

 @Override
 public T persistir(T entity) {
   this.getEntityManager().persist(entity);
   return entity;
 }
  …
}
```

```
@Repository
public class PersonaDAOHibernateJPA
   extends GenericDAOHibernateJPA<Persona>
   implements PersonaDAO {

   public PersonaDAOHibernateJPA() {
     super(Persona.class);
   }
   . . .

}
```

Además para Spring es necesario que se definan cuales son las clases DAO usando la anotación `@Repository`
Spring también sugiere anotaciones como: `@Controller/@RestController, @Service` para estereotipar componentes de diferentes capas.