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. Integración de Spring con JPA

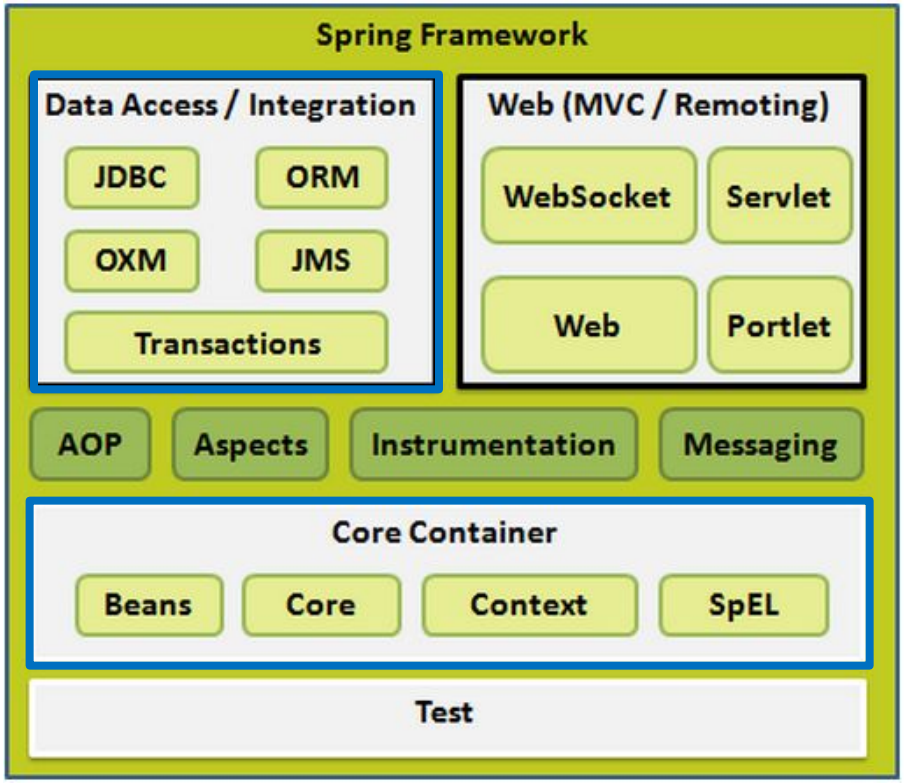
¿Qué es Spring?

- 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

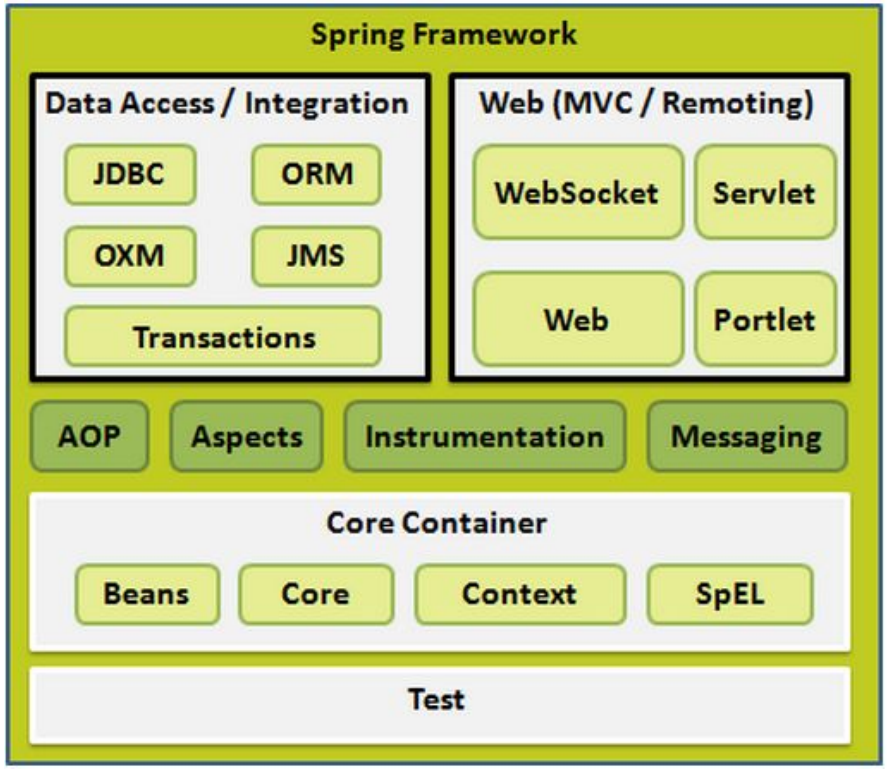
El framework Spring consta de funciones organizadas en aproximadamente 20 módulos. Estos módulos se agrupan en:



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.

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 de objetos Mock que pueden ser usados para testear la aplicación.

Spring
Conceptos importantes

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

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 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 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.

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

Spring

Contenedores de Inversión de Control y Beans

¿Qué es un Contenedor IoC Spring?

Los contenedores IoC de Spring, son responsables de **crear instancias, configurar y ensamblar beans leyendo de metadatos** de configuración de XML, anotaciones Java y/o código Java en los archivos de configuración.

¿Qué es un Bean en Spring?

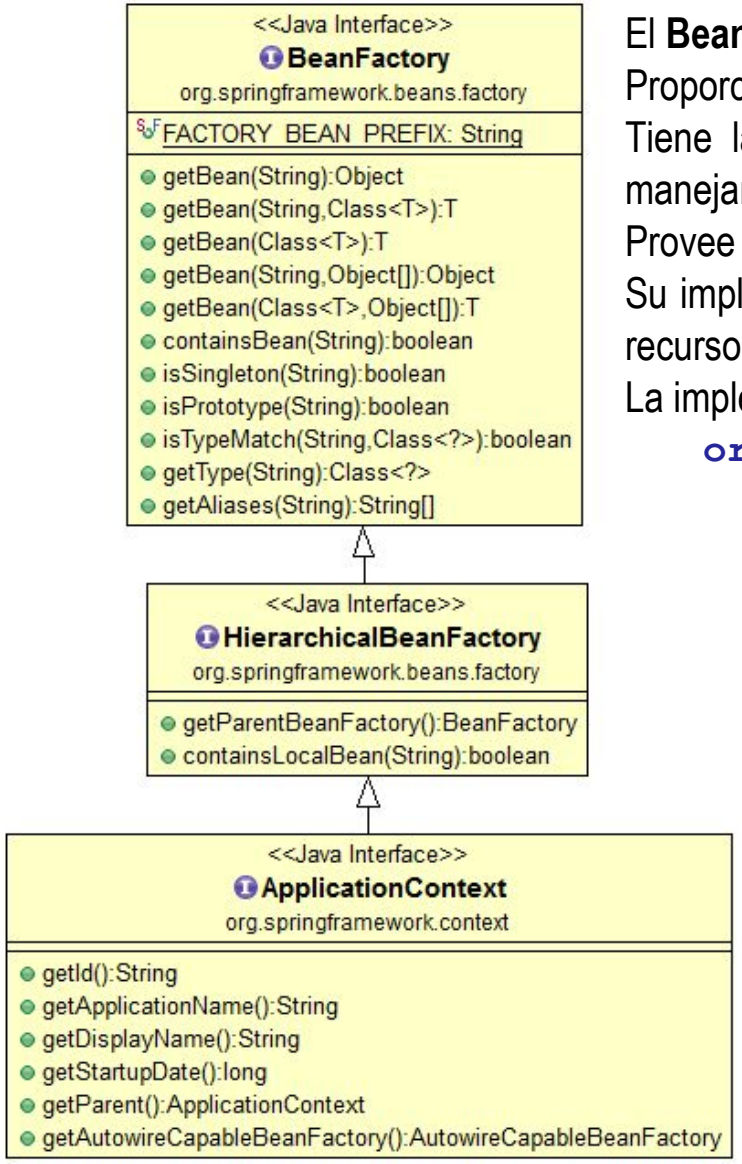
En Spring, los objetos que conforman la estructura principal de las aplicaciones, y que son administrados por el **Contenedor de IoC**, son llamados "beans". Un bean es un objeto que es declarado, para que sea instanciado, inicializado con todas sus dependencias y administrado por el contenedor de IoC.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

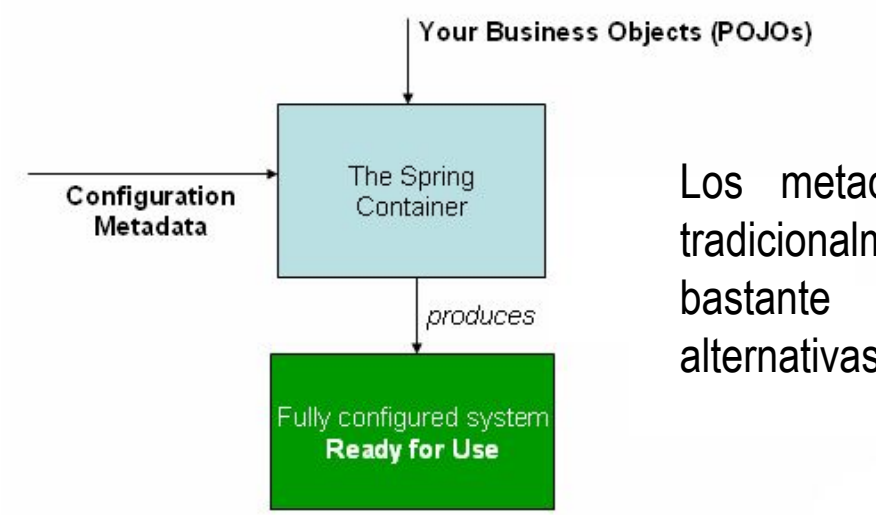
El **ApplicationContext** es una especialización de **BeanFactory**.
Los contenedores Spring que implementan esta interface son más avanzados que los que implementar BeanFactory.
Esta interface agrega al BeanFactory funcionalidades más específicas para aplicaciones empresariales cómo resolver mensajes textuales desde archivos de propiedades (i18n) o publicar eventos a los listener interesados.
Existen varias implementaciones de ApplicationContext como:

`FileSystemApplicationContext`
`ClassPathApplicationContext`
`WebApplicationContext`

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 en 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 bastante intuitivos 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 `@autowiring`, más algunos archivos XML.
- **Spring 3.0**, incorpora otras anotaciones como **@Configuration**, **@Bean**, **@Import** que permiten evitar el uso de archivos XML.

Spring

Contenedores de IoC - Configuración basada en XML

Existen varias implementaciones de **ApplicationContext**. Para aplicaciones de escritorio se puede utilizar 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));
 }
}
```

app-ctx.xml: se definen los beans de Servicios y DAOs

```
<?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>
 <bean id=“usuario" class="ar.com.ttps.modelo.Usuario"/>
</beans>
```

Spring

Contenedores de IoC - Configuración basada en XML

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>
  </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);
```

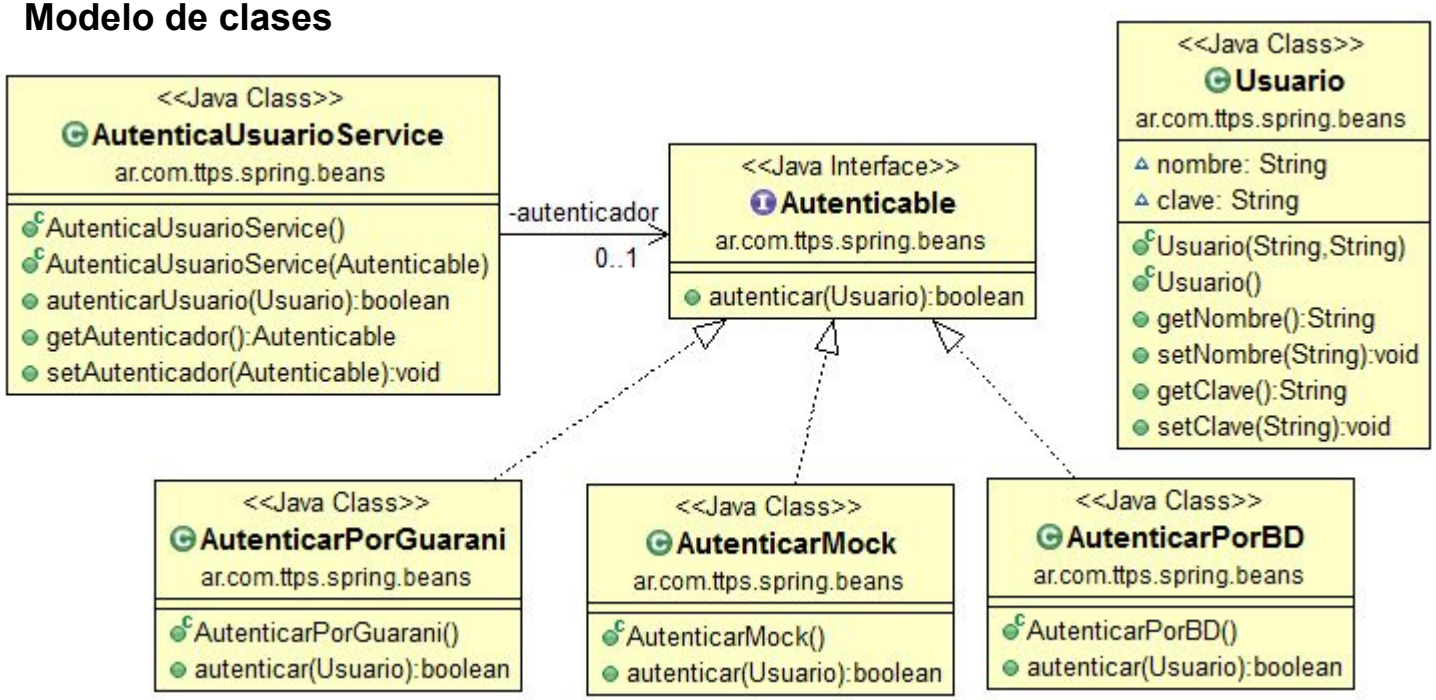
Sin embargo la forma habitual de trabajar en aplicaciones web con Spring es usando inyección de dependencias, NO de esta forma.

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

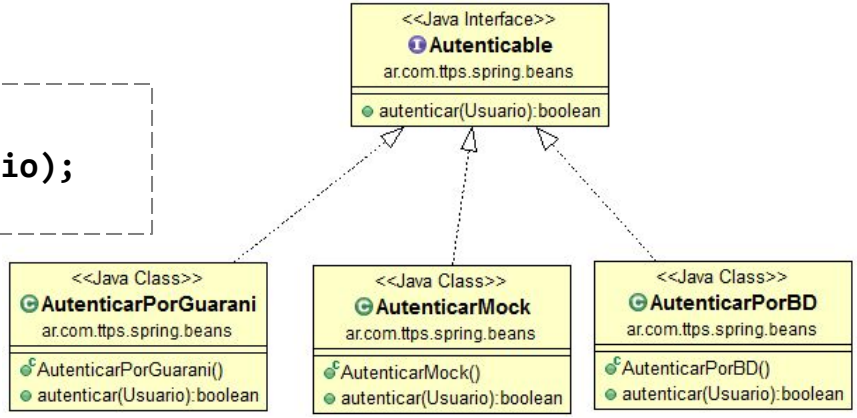


Spring

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

Inyección de Dependencias - Configuración basada en XML

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 **`ApplicationConext`** 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 'autenticador' 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:

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:

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 más 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) {}
}
```

Spring

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="autUsuarioService" 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="autUsuarioService" 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="autUsuarioService" 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>
```

Spring

Configuración de JPA con XML (1/3)

En el **`ApplicationContext`** se suelen registrar los bean relacionados con la capa del medio (middle-tier) y con la capa de acceso a datos (data-tier). A continuación vamos a definir beans relacionados con la configuración para acceso a datos:

app-ctx.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xmlns="http://www.springframework.org/schema/beans" xmlns:tx="http://www.springframework.org/schema/tx"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

<!-- para que reconozca los Servicios, DAOs, etc anotados -->
<context:component-scan base-package="ttps.daosjpa"/>

<!-- 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"/>
      <property name="acquireIncrement" value="2"/>
      <property name="minPoolSize" value="20"/>
      <property name="maxPoolSize" value="50"/>
      <property name="maxIdleTime" value="600"/>
  </bean>
  . . .
</beans>
```

Spring

Configuración de JPA con XML (2/3)

Continuando con la definición de beans, ahora definimos un **`EntityManagerFactory`** manejado por el contendor. Recordemos que para la persistencia usando JPA se necesita una instancia de **`EntityManager`**.

app-ctx.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      . . .
      http://www.springframework.org/schema/context/spring-context-4.2.xsd“>
 . . .
<!-- Configuración JPA -->
<bean id="emf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
     <!- Estos tag haceN que /META-INF/persistence.xml ya no sea necesario -->
    <property name="dataSource" ref="dataSource" />
    <property name="packagesToScan" value="ttps.springmvc.entities" />
    <!- Seteo la implementación del EntityManager (JPA) de Hibernate -->
    <property name="jpaVendorAdapter">
      <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
    </property>
    <property name="jpaProperties">
         <props>
            <prop key="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</prop>
            <prop key="hibernate.hbm2ddl.auto">update</prop>
            <prop key="hibernate.format_sql">true</prop>
            <prop key="hibernate.show_sql">true</prop>
         </props>
      </property>
 </bean>
</beans>
```

Spring
Configuración de JPA con XML (3/3)

Finalmente queda definir el manejador de Transacciones. El manejador de transacciones **`JPATransactionManager`** es apropiado para aplicaciones que usan un único **`EntityMangerFactory`** para acceso transaccional a los datos.

`app-ctx.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
     . . .
     http://www.springframework.org/schema/context/spring-context-4.2.xsd“>
          . . .
<!-- Manejador de Transacciones -->

<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
     <property name="entityManagerFactory" ref= "emf" />
</bean>

<tx:annotation-driven transaction-manager="transactionManager"/>

</beans>
```

El **`JPATransactionManager`** necesita ser vinculado con un **`EntityMangerFactory`** (cualquier implementación de javax.persistence.EntityManagerFactory

Indicamos que para este **`TransactionManager`** vamos a usar anotaciones

Spring

Configuración basada en Anotaciones

Spring 2.5+

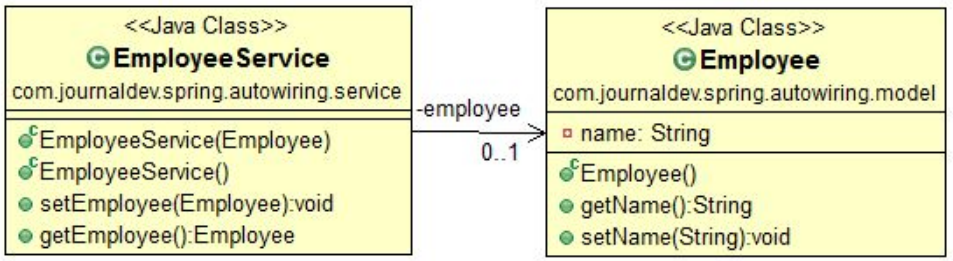
Spring

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. 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 **`<contructor-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("Constrctor con Employee en EmployeeService");
   this.employee = empl;
 }

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

 public void setEmployee(Employee empl) {
   this.employee = empl;
   System.out.println("Setter en EmployeeService");
}

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

constructor

setter

```
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**.

```
package ttps.spring.autowiring;
@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**.

```
package ttps.spring.autowiring;
@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 pasarle 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 igual necesita de un archivo XML de configuración mínima. Cuando usamos configuración 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="byType”>
  <!-- Habilitamos configuración basada en anotaciones -->
  <context:annotation-config/>
  <context:component-scan base-package=“ttps.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.

```
package ttps.spring.autowiring;
@Component
public class EmployeeService {
 private Employee emple;     //nombre emple

 // usado por autowiring x constructor
 public EmployeeService(Employee empl) {
   System.out.println("Constrctor con Employee en EmployeeService");
   this.employee = 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.

```
package ttps.spring.autowiring;
@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("Constrctor con Employee en EmployeeServ");
   this.employee = 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.

Spring

Nuevas anotaciones para evitar el archivo de configuración XML

Spring 3

Spring

Configuración basada en Anotaciones

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.

```
package ttps.misbenas;
//imports
import javax.sql.DataSource;
. . .
@Configuration
public class PersistenceConfig {
      private static final String MODEL_PACKAGE = "ttps.spring.model";
      @Bean
       public DataSource dataSource() {
          DriverManagerDataSource driverManagerDataSource = new DriverManagerDataSource();
          driverManagerDataSource.setUsername("root");
          driverManagerDataSource.setPassword("root");
          driverManagerDataSource.setUrl("jdbc:mysql://localhost:3306/mibd");
          driverManagerDataSource.setDriverClassName("com.mysql.jdbc.Driver");
          return driverManagerDataSource;
       }
        . . .
}
```

Esto sería el equivalente en el archivo de configuración XML

```
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
    <property name="driverClass" value="com.mysql.jdbc.Driver"/>
     . . .
</bean>
```

Spring

Configuración basada en Anotaciones

@Component vs @Bean

@Component (al igual que @Service, @Repository, @Controller) se utilizan 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, en lugar de dejar que Spring lo haga automáticamente como se indicó anteriormente. 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

Configuración del Contexto con Anotaciones (1/3)

La configuración del contexto de Spring, a partir de la versión Servlet 3 puede hacerse con java y anotaciones. Para ello se crea una clase (como **`AppConfig)`** donde se registrarán todos los bean relacionados con Spring

```
package ttps.spring.config;

import java.util.List;

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "ttps.spring")
public class AppConfig implements WebMvcConfigurer {

    @Override
    public void extendMessageConverters(List<HttpMessageConverter<?>> converters) {
        converters.add(new MappingJackson2HttpMessageConverter());
    }
}
```

pueden ir paquetes separados por ","

Tag XML	Anotación	Descripción
<context:component-scan/>	@ComponentScan	Escanea por controllers, repositorios, servicios, beans, etc.
<mvc:annotation-driven/>	@EnableWebMvc	Habilita anotaciones como @Controller de SpringMVC
spring_mvc-xml	@Configuration	Trata a la clase como un archivo de configuracion para app String MVC

Spring

Configuración del Contenedor con Anotaciones (2/3)

Ahora crearemos una clase JAVA para reemplazar a nuestro **`web.xml`**. Esta clase debe implementar la interface **`WebApplicationInitializer`**.

```java
package ttps.spring.config;

import javax.servlet.ServletContext;

public class SpringWebApp implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext container) throws ServletException {
        // Create the 'root' Spring application context
        AnnotationConfigWebApplicationContext rootContext = new AnnotationConfigWebApplicationContext();
        rootContext.register(AppConfig.class);

        //ContextLoaderListener - Manage the lifecycle of the root application context
        container.addListener(new ContextLoaderListener(rootContext));

        //DispatcherServlet - Register and map the dispatcher servlet
        ServletRegistration.Dynamic dispatcher = container.addServlet("DispatcherServlet", new DispatcherServlet(rootContext));
        dispatcher.setLoadOnStartup(1);
        dispatcher.addMapping("/");
    }

}
```

A partir de Spring 3, ContextLoaderListener admite la inyección del contexto de la aplicación web raíz a través del constructor **ContextLoaderListener (WebApplicationContext)**, lo que permite esta configuración programática en entornos Servlet 3.0+.

En esta clase se registra la clase **`AppConfig`** como la que configura el contexto y registramos y configuramos el DispatcherServlet, el cual actúa como **`FrontController`** para las aplicaciones Spring MVC.

Spring

Configuración de JPA con Anotaciones (3/3)

Para usar JPA en un proyecto Spring, es necesario configurar el EntityManager. Programáticamente se realiza a través de la clase **`LocalContainerEntityManagerFactoryBean.`**

```
package ttps.spring.config;
import java.util.Properties;

@Configuration
@EnableTransactionManagement
public class PersistenceConfig {

    private static final String MODEL_PACKAGE = "ttps.spring.model";

    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
        LocalContainerEntityManagerFactoryBean emf = new LocalContainerEntityManagerFactoryBean();
        emf.setDataSource(dataSource());
        emf.setPackagesToScan(new String[] {MODEL_PACKAGE});
        JpaVendorAdapter jpaVendorAdapter = new HibernateJpaVendorAdapter();
        emf.setJpaVendorAdapter(jpaVendorAdapter);
        emf.setJpaProperties(additionalProperties());
        return emf;
    }

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource driverManagerDataSource = new DriverManagerDataSource();
        driverManagerDataSource.setUsername("root");
        driverManagerDataSource.setPassword("root");
        driverManagerDataSource.setUrl("jdbc:mysql://localhost:3306/mibd?serverTimezone=UTC");
        driverManagerDataSource.setDriverClassName("com.mysql.jdbc.Driver");
        return driverManagerDataSource;
    }

    @Bean
    public PlatformTransactionManager transactionManager(EntityManagerFactory emf){
        JpaTransactionManager jpaTransactionManager = new JpaTransactionManager();
        jpaTransactionManager.setEntityManagerFactory(emf);
        return jpaTransactionManager;
    }
...
```

Spring
Mas anotaciones

@PersistenceContext

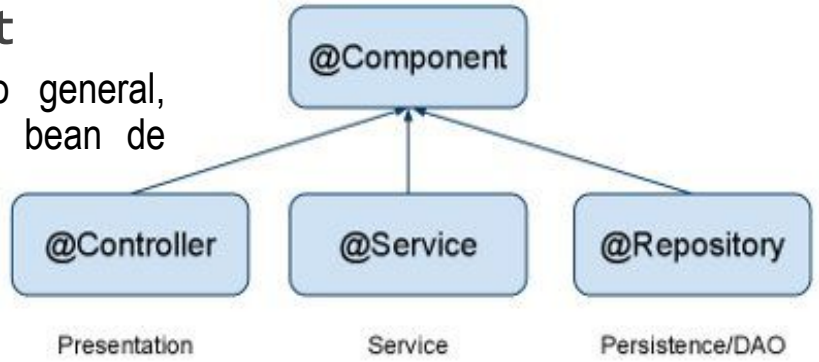
Esta anotación se usa para requerir Inyección de Dependencias del **EntityManagerFactory** default. Con esta anotación le indicamos a Spring que inyecte y maneje un EntityManager.

@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 DAOs. Entre los usos de este marcador está la traducción automática de excepciones.

Spring

¿Cómo afectan estas anotaciones en nuestros DAOs?

```
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> {
 private EntityManager entityManager;

                                    La injección es aplicada automáticamente por
                                    el contenedor de Spring y el EntityManager es
                                    manejado por Spring
 @PersistenceContext
 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 nuestros DAOs?

```
@Transactional
public class GenericDAOHibernateJPA<T>
                    implements GenericDAO<T> {
 private EntityManager entityManager;
 @PersistenceContext
 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.

Spring
Para probar los tests con Spring desktop

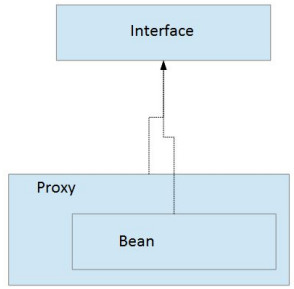
Contexto Spring standalone. Administra componentes, en particular las clases anotadas con @Configuración, pero también tipos @Component simples

```
public class TestSpringSimple {
   public static void main(String[] args) {
     AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();

     //registra una o más componentes que serán procesadas
     ctx.register(ttps.spring.config.PersistenceConfig.class);
     //Busca clases anotadas en el paquete base pasado por parámetro
     ctx.scan("ttps");
     ctx.refresh();

     UsuarioDAO usrDAO = ctx.getBean("usuarioDAOImpl", UsuarioDAO.class);
     System.out.println(usrDAO.listAll());
   }
}
```

El nombre por defecto del bean generado, es el nombre de la clase con la primer letra en minúscula.



Se utiliza la interface, ya que Spring crea e inyecta proxies (que la implementan), en lugar de inyectar directamente una instancia del DAO