ANOTACIONES

Curso de Spring Core 5

Contenedores

Posibles formas de configuración de metadatos

Spring nos permite configurar los metadatos a través de varias formas:

- XML
- Anotaciones
- Código Java (conocido como Java-Config).

A lo largo del curso cubriremos las 3 formas, si bien comenzaremos con XML.

Dependencia Maven

Cabe recordar que la dependencia *maven* que necesitamos para comenzar a usar el *Spring Ioc container* es:

```
<dependency>
  <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.0.8.RELEASE</version>
</dependency>
```

ClassPathXMLApplicationContext vs. FileSystemXMLApplicationContext

En esta lección hemos visto el uso de ambas clases. Durante el resto de lecciones utilizaremos la primera de ellas.

Cómo nombrar un bean

Un bean, normalmente, tiene un solo nombre; y este debe ser único en el contenedor donde esté registrado. Si necesitamos que tenga más de un nombre, deberíamos declarar un alias.

En XML, para indicar el nombre de un bean, podemos usar su propiedad id. Los nombres suelen seguir la notación *Camel Case*: myBean, emailService, … Si queremos declararle algún alias, podemos usar la propiedad name, indicando los nombre separados por comas, punto y comas o espacios.

Definición de un bean

Durante este ejemplo solo utilizaremos algunas de las propiedades necesarias de un bean, como el id, name o class. A continuación tenemos una lista más completa:

- class
- name
- scope
- lazy-init
- depends-on
- init-method
- destroy-method

Inyección dependencia: vía setter vs vía constructor

Inner beans (beans anidados)

En ocasiones, podemos crear beans anidados (o internos) a otros beans, en lugar de referenciarlos. **¿Cuál sería la ventaja de un *inner bean* frente a**

otro *referenciado*?. La respuesta no es difícil: el bean anidados no será accesible desde fuera el bean externo, mientras que el referenciado puede ser accedido por otros beans.

```
<bean id="outer" class="...">
    <!-- instead of using a reference to a target bean, simply define the target bean inline -->
    <property name="target">
        <bean class="com.example.Person"> <!-- this is the inner bean -->
            <property name="name" value="Fiona Apple"/>
            <property name="age" value="25"/>
        </bean>
    </property>
</bean>
```

En este ejemplo, el bean de tipo Person solo podría ser accedido por el bean outer.

Colecciones

Spring nos ofrece la posibilidad de inyectar valores dentro de una colección. Los tipos soportados son <list> (java.util.List), <set> (java.util.Set), <map> (java.util.Map), <props> (java.util.Properties).

```
<bean id="moreComplexObject" class="example.ComplexObject">
    <!-- results in a setAdminEmails(java.util.Properties) call -->
    <property name="adminEmails">
        <props>
            <prop key="administrator">administrator@example.org</prop>
            <prop key="support">support@example.org</prop>
            <prop key="development">development@example.org</prop>
        </props>
    </property>
    <!-- results in a setSomeList(java.util.List) call -->
    <property name="someList">
        <list>
```

```
        <value>a list element followed by a reference</value>
        <ref bean="myDataSource" />
      </list>
    </property>
    <!-- results in a setSomeMap(java.util.Map) call -->
    <property name="someMap">
      <map>
        <entry key="an entry" value="just some string"/>
        <entry key ="a ref" value-ref="myDataSource"/>
      </map>
    </property>
    <!-- results in a setSomeSet(java.util.Set) call -->
    <property name="someSet">
      <set>
        <value>just some string</value>
        <ref bean="myDataSource" />
      </set>
    </property>
</bean>
```

Exclusión de la inyección automática

Podemos excluir un bean de ser *candidato* a inyectarse automáticamente mediante el atributo autowired-candidate=false. De esa forma, solo podrá inyectarse de forma explícita.

También podemos limitar los candidatos a ser autoinyectados utilizando un patrón sobre los nombres de los beans y el atributo default-autowired-candidates ***(este es solo aplicable al elemento raiz <beans>)***. Por ejemplo, para que todos ellos incluyeran el nombre *repository*, podríamos usar el patrón *Repository.

Configuracion Vía XML vs Vía anotaciones

Las anotaciones son mejores que XML para configurar Spring??
La introducción de la configuración basada en anotaciones planteó la pregunta de si este enfoque es "mejor" que XML. La respuesta corta es que **depende**. La respuesta larga es que cada enfoque tiene sus pros y sus contras, y generalmente le corresponde al desarrollador decidir qué estrategia le conviene más. Debido a la forma en que se definen, las anotaciones proporcionan *mucha información* en su declaración, lo que lleva a una configuración más breve y concisa. Sin embargo, XML se destaca en el *cableado* de componentes sin tocar su código fuente o recompilarlos. Algunos desarrolladores prefieren tener el *cableado* cerca del código fuente, mientras que otros argumentan que las clases anotadas ya no son POJO (*Plain Old Java Object*) y, además, que la configuración se vuelve descentralizada y más difícil de controlar.

No importa la elección, Spring puede acomodar ambos estilos e incluso mezclarlos. Vale la pena señalar que a través de su opción JavaConfig, Spring permite que las anotaciones se utilicen de forma no invasiva, sin tocar el código fuente de los componentes objetivo.

La inyección por anotación se realiza *antes* de la inyección de XML, por lo tanto, la última configuración anulará la anterior para las propiedades inyectadas a través de ambos enfoques.

@Autowired

La especificación JSR 330 de Java define un conjunto de anotaciones *estándar* para la inyección de dependencias. **En nuestro caso, estamos usando las anotaciones propias de Spring, pero podríamos usar perfectamente las estándar con el mismo comportamiento**.

Para usar las dependencias estándar, necesitamos añadir la siguiente dependencia Maven:

```
<dependency>
    <groupId>javax.inject</groupId>
    <artifactId>javax.inject</artifactId>
    <version>1</version>
</dependency>
```

En lugar de usar @Autowired, podríamos usar la anotación @Inject:

```
import javax.inject.Inject;

public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Inject
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    public void listMovies() {
        this.movieFinder.findMovies(...);
        ...
    }
}
```

Uso de primary y Qualifier

Anotaciones estándar

Al igual que en la lección anterior, podemos utilizar las anotaciones estándar para *calificar* o *nombrar* un bean. Tenemos disponibles las anotaciones estándar

- @Qualifier: sirve para asignar un nombre a un bean

- @Named: sería la equivalente al uso de la anotación de Spring @Qualifier.

Extendiendo la anotación @Qualifier

Para extender la anotación @Qualifier debemos crear un interfaz como este:

```
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Epoca {

   String value();

}
```

De esta forma, allá donde se pueda usar @Qualifier, podremos usar nuestra anotación (en el ejemplo, @Epoca).

Aunque la creación de anotaciones propias (en general) queda fuera del ámbito de este curso, puedes consultar la siguiente documentación oficial de Oracle: https://docs.oracle.com/javase/tutorial/java/annotations/declaring.html

CURSO DE
SPRING CORE 5

MIS DATOS

- Luis Miguel López Magaña
- 15 años desarrollando aplicaciones Java (Java SE, Java EE, Spring, Hibernate, Android, …)
- Profesor de FP desde hace 10 años.

REQUISITOS PARA ESTE CURSO

- Java SE 8.
- Metodología de programación orientada a objetos.

Y mejor si además sabes

- Algo de Java EE 7.
- Maven.
- Patrones de diseño.

¿QUÉ ME VA A APORTAR SPRING?

- Conocer algunos de los patrones de diseño que más se utilizan en la programación de aplicaciones empresariales.
- Reconocer la utilidad de los mecanismos de inversión de control e inyección de dependencias.
- Aprender a manejar muchos de los elementos transversales de la tecnología Spring, que sustentan al resto de módulos.

CONTENIDOS

1. Introducción a Spring
2. Contenedor de Inversión de Control
3. Ámbito y ciclo de vida de un bean
4. Configuración basada en anotaciones
5. Configuración a través de Java

PRÁCTICAS

Practicaremos la sintaxis de cada una de las diferentes lecciones.

Además, para finalizar el curso realizaremos paso a paso un proyecto completo que integre gran parte de los conocimientos del curso.

¿QUÉ SERÉ CAPAZ AL FINAL DEL CURSO?

- Aprenderás algunos de los patrones de diseño más utilizados en programación
- Conocerás qué es Spring y cuales son los módulos que conforman esta tecnología.
- Reconocerás la importancia de utilizar los mecanismos de inversión de control e inyección de dependencias.

¿QUÉ SERÉ CAPAZ AL FINAL DEL CURSO?

- Serás capaz de instalar todo el entorno de trabajo necesario para empezar a trabajar con Spring.
- Conocerás cómo crear tus propios beans, utilizando diferentes mecanismos.
- Aprenderás a utilizar el contenedor de inversión de control de Spring.

¿QUÉ CURSOS PUEDO REALIZAR AL TERMINAR ESTE?

- Curso de Desarrollo Web Java EE
- Curso de Hibernate
- Curso de introducción a Thymeleaf
- Curso de SQL desde cero.
- ...

CURSO DE
SPRING CORE 5

INTRODUCCIÓN A SPRING

1.

¿A QUÉ LLAMAMOS SPRING?

“

Spring es un framework de código abierto para la creación de aplicaciones empresariales Java, con soporte para Groovy y Kotlin; tiene estructura modular y una gran flexibilidad para implementar diferentes tipos de arquitecturas según las necesidades de la aplicación.

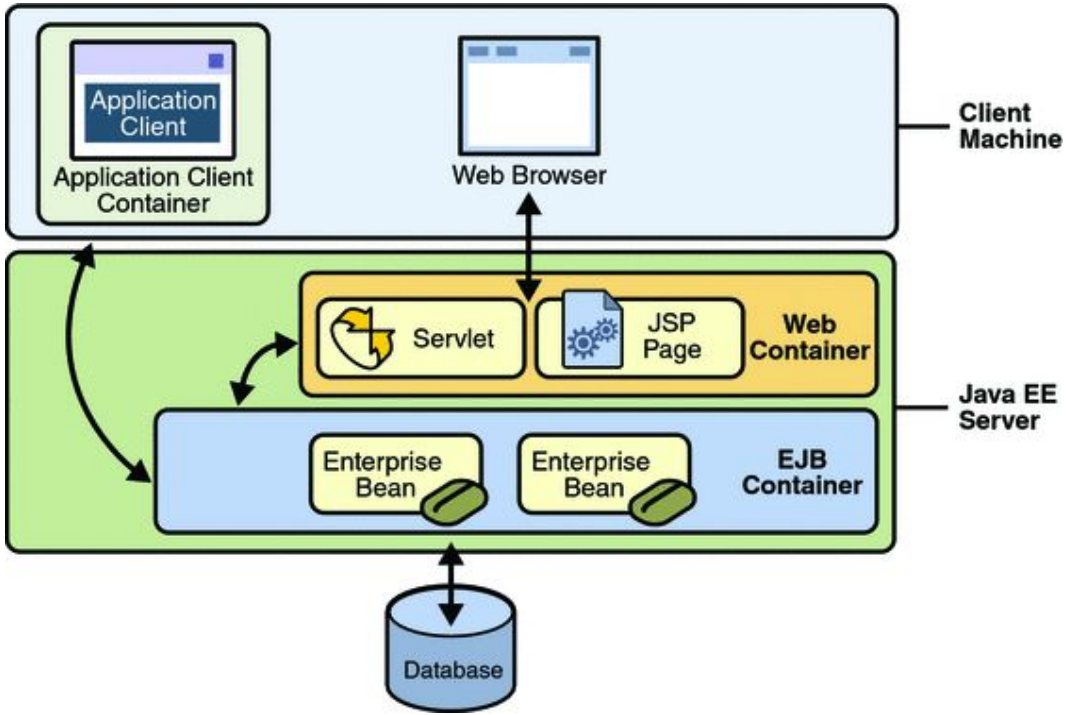
CUANDO DECIMOS SPRING QUEREMOS DECIR...

- Proyecto Spring Framework
  - Núcleo de toda la familia Spring
- Familia de proyectos Spring
  - Diversos módulos que abarcan múltiples necesidades
- Entorno de desarrollo Spring Tool Suite
  - Basado en Eclipse

2.

ALGO DE HISTORIA

ORÍGENES DE SPRING: RESPUESTA AL MODELO EJB



ORÍGENES DE SPRING: RESPUESTA AL MODELO EJB

- Alta complejidad y baja productividad para el programador.
- Solo basado en RMI.
- No todas las aplicaciones necesitan componentes remotos.
- Difíciles de depurar
- Mapeo O/R basado en beans de entidad muy limitado.

SPRING: COMPLEMENTO AL MODELO JAVA EE

Spring no incluye la especificación Java EE. Integra especificaciones individuales de algunos componentes:

- Servlet API
- WebSocket API
- Concurrencia
- JSON Binding API
- Validación de Beans
- JPA
- ...

3.

PROYECTOS Y MÓDULOS DE SPRING

FAMILIA DE PROYECTOS SPRING

Spring Framework

Spring Boot

Spring Data

Spring Security

Spring Cloud

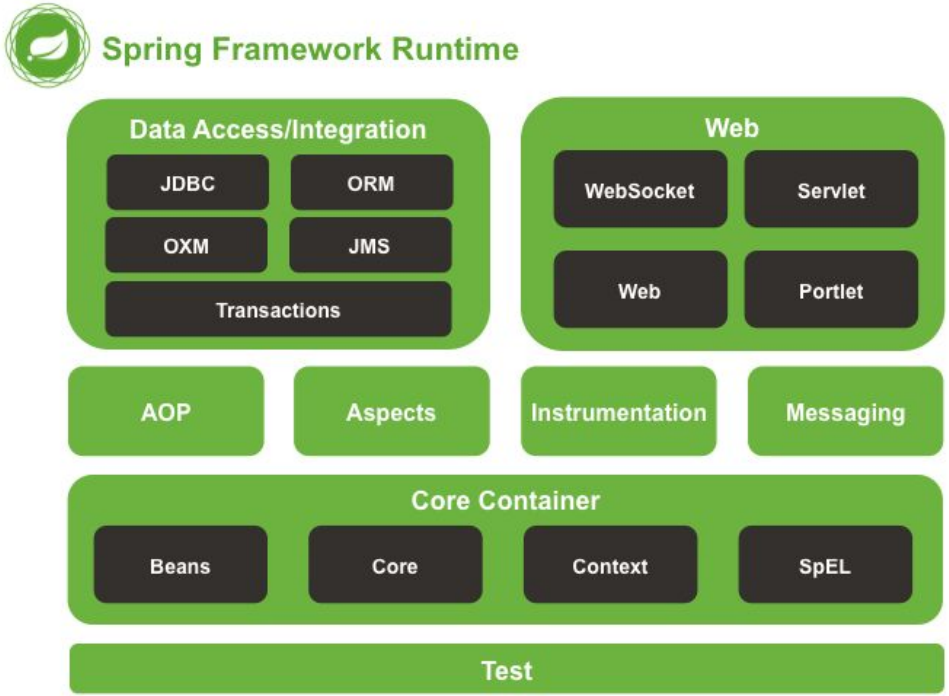
Spring HATEOAS

Spring Batch

Spring for Android

...

MÓDULOS MÁS USUALES DE SPRING



4.

VERSIONES DE SPRING

VERSIÓN ACTUAL: 5

- Versión actual: 5.0.8
- Para Spring 4, la versión es 4.3.18
- Próximamente, tendremos disponible la versión 5.1

NOVEDADES DE SPRING 5

- Uso de JDK 8 y Java EE 7.
- Compatibilidad con JDK 9 y Java EE 8.
- Versiones mínimas: Servlet 3.1, Bean Validation 1.1, JPA 2.1.
- Compatibilidad para Servlet 4.0, Bean Validation 2.0, JPA 2.2, JSON Binding API 1.0
- Compatibilidad para: Tomcat 8.5+, Jetty 9.4+, WildFly 10+
- Modulo para web reactiva: WebFlux
- Programación funcional con Kotlin

INSTALACIÓN DEL ENTORNO DE TRABAJO

1.

REQUISITOS PREVIOS

¿QUÉ TENGO QUE SABER PARA TRABAJAR CON SPRING?

- Metodología de programación y POO
- Java (SE/EE)
- Algunos patrones de diseño
- Maven (al menos como gestor de dependencias)

2.

ENTORNOS DE DESARROLLO

POSIBLES IDEs

Cualquiera que nos permita trabajar con Java y Maven (también Gradle).

- Eclipse
- Netbeans
- IntelliJ Idea
- Visual Studio Code
- ...

IDE PREFERIDO:
SPRING TOOL SUITE

- Plugin para eclipse (o Bundle)
- *Ready-to-use*
- Todas las ventajas de eclipse
- Soporte para Java SE, Java EE, ...
- Soporte para Cloud Foundry y Pivotal tc Server
- Soporte para Maven y Gradle

ÚLTIMA VERSIÓN:
SPRING TOOL SUITE 3.9.X

- Basado en Eclipse Oxygen (4.8)
- Pivotal tc Server 3.2.6
- Soporte para Java 9

https://spring.io/tools/sts

3.

INSTALACIÓN DEL ENTORNO DE DESARROLLO

PASO 1:
COMPROBAR JDK

- Desde la línea de comandos:

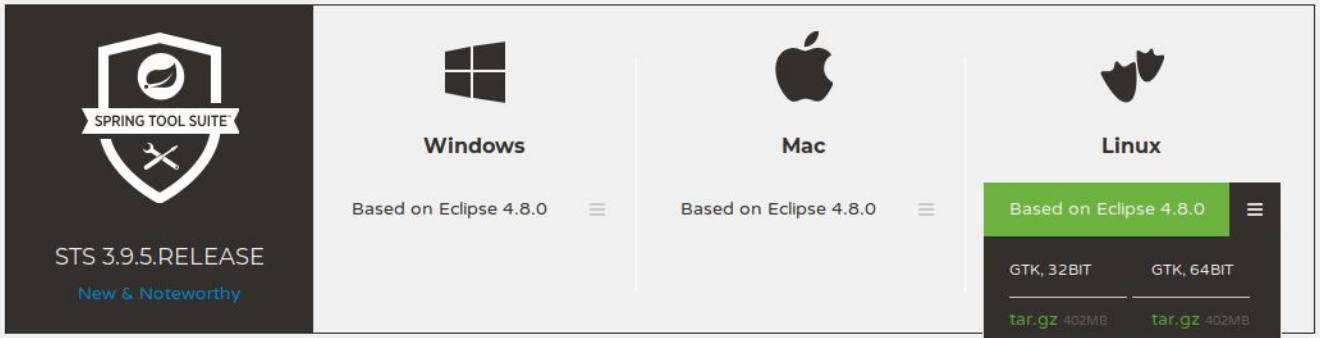
```
java -version
```

```
user@ubuntu:~$ java -version

No se ha encontrado la orden «java», pero se puede instalar con:

sudo apt install default-jre
sudo apt install openjdk-11-jre-headless
sudo apt install openjdk-8-jre-headless
```

```
user@ubuntu:~$ java -version
openjdk version "1.8.0_181"
OpenJDK Runtime Environment (build 1.8.0_181-8u181-b13-0ubuntu0.18.04.1-b13)
OpenJDK 64-Bit Server VM (build 25.181-b13, mixed mode)
```

PASO 2:
DESCARGAR SPRING TOOL SUITE

https://spring.io/tools/sts



PASO 3:
DESCOMPRIMIR Y ENLAZAR

- Desde la línea de comandos:

```
cd ~/Descargas
sudo mv spring-....tar.gz /opt
```

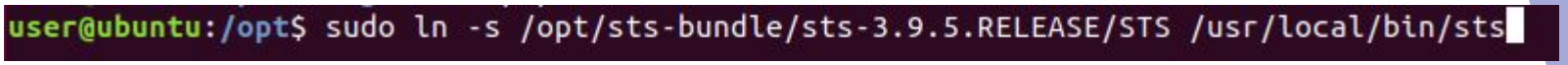
```
user@ubuntu:~$ cd ~/Descargas/
user@ubuntu:~/Descargas$ sudo mv spring-tool-suite-3.9.5.RELEASE-e4.8.0-linux-gtk-x86_64.tar.gz /opt
```

```
cd /opt
sudo tar zxvf spring-...tar.gz
```

```
user@ubuntu:~/Descargas$ cd /opt
user@ubuntu:/opt$ sudo tar zxvf spring-tool-suite-3.9.5.RELEASE-e4.8.0-linux-gtk-x86_64.tar.gz
```

PASO 3:
DESCOMPRIMIR Y ENLAZAR

```
sudo ln -s
    /opt/sts-bundle/sts-3.9.5.RELEASE/STS
    /usr/local/bin/sts
```



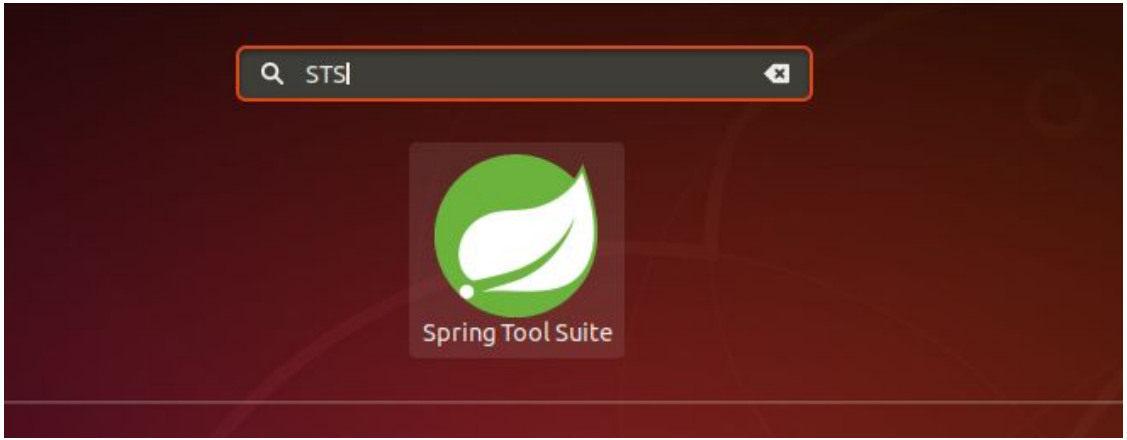
PASO 4 (OPCIONAL): CREAR ACCESO DIRECTO

- Desde la línea de comandos:

  sudo gedit /usr/share/applications/sts.desktop

```
[Desktop Entry]
Name=Spring Tool Suite
Comment=Spring Tool Suite 3.9.5
Exec=/usr/local/bin/sts
Icon=/opt/sts-bundle/sts-3.9.5.RELEASE/icon.xpm
StartupNotify=true
Terminal=false
Type=Application
Categories=IDE;Development;Java;
```

Y... LISTO



4.

DEPENDENCIAS MAVEN

DEPENDENCIA MAVEN

- https://mvnrepository.com
- Spring Context 5.0.8.RELEASE
- pom.xml

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.0.8.RELEASE</version>
</dependency>
```

ESTRUCTURA DE UNA APLICACIÓN EMPRESARIAL Y PATRONES DE DISEÑO

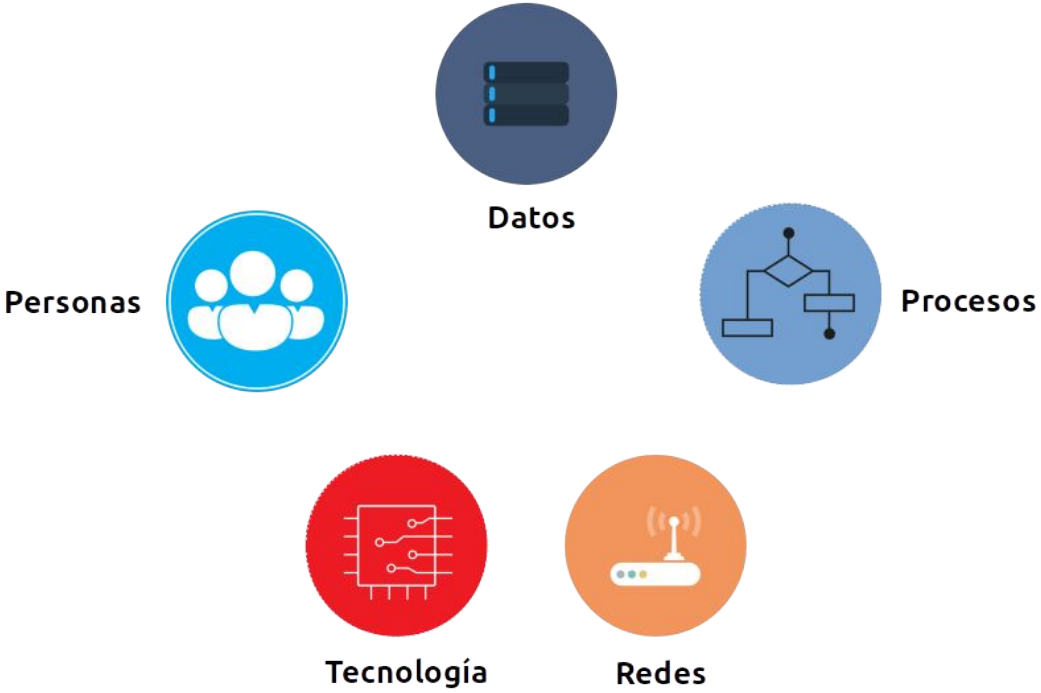
1.

APLICACIÓN EMPRESARIAL

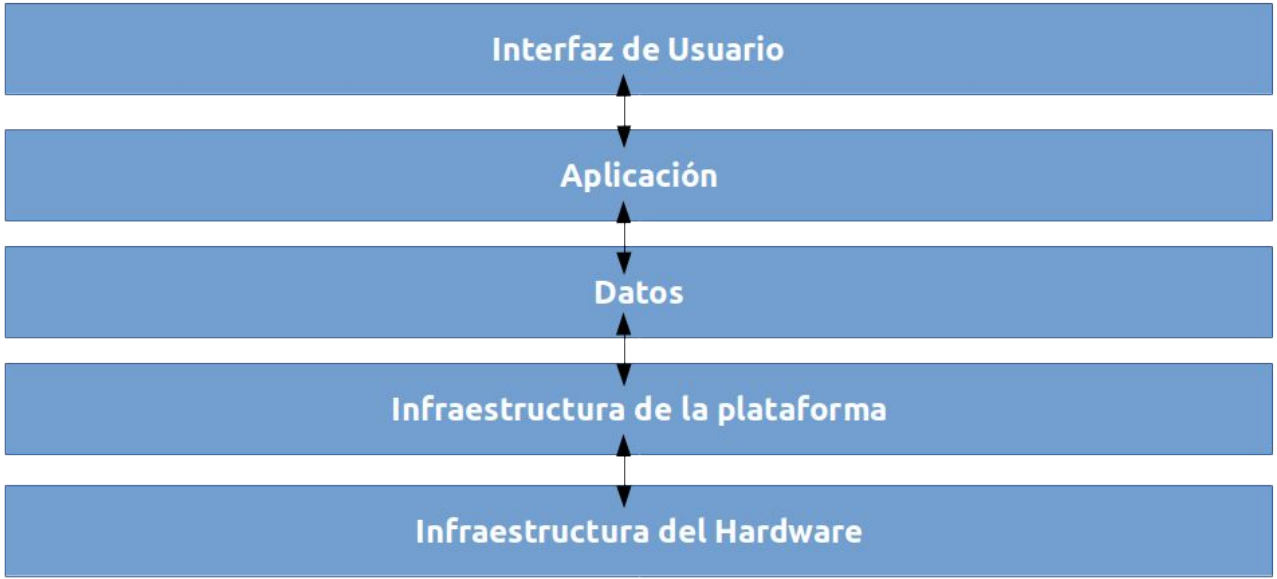
¿QUÉ ES UNA APLICACIÓN EMPRESARIAL?

- *Gran* aplicación comercial
- Compleja, escalable, distribuida, crítica
- Despliegue en redes corporativas o internet
- Centrada en los datos.
- Intuitiva, de uso fácil.
- Requisitos de seguridad y mantenibilidad.

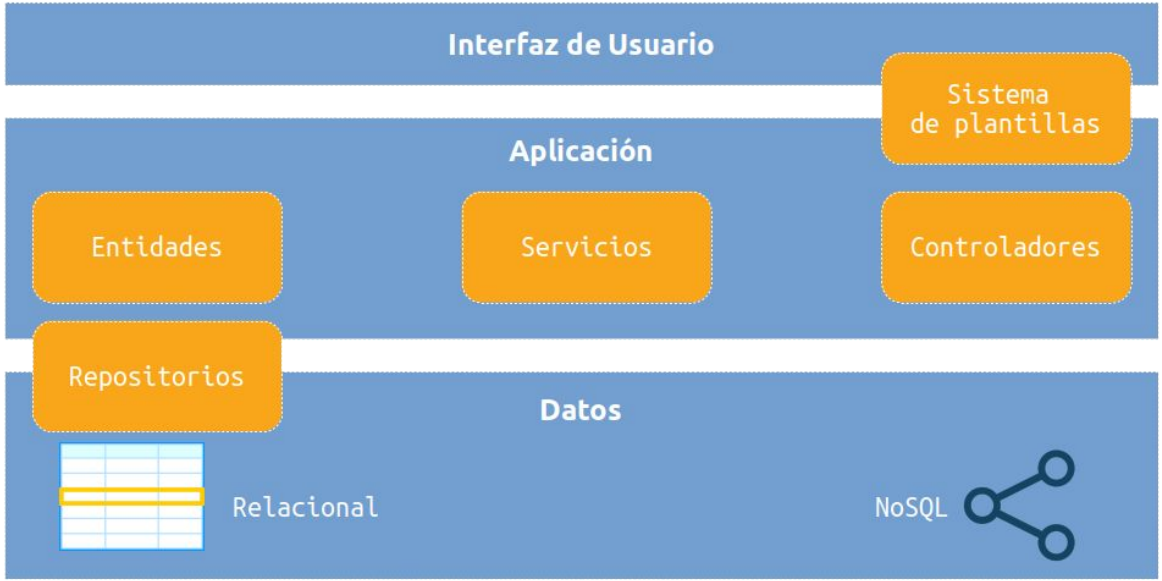
ESTRUCTURA DE UNA APLICACIÓN EMPRESARIAL



ESTRUCTURA DE UNA APLICACIÓN EMPRESARIAL



SPRING EN UNA APLICACIÓN EMPRESARIAL



2.

PATRONES DE DISEÑO

PATRÓN DE DISEÑO SOFTWARE

- Tengo un problema al diseñar mi aplicación.
- ¿Seguro que no le ha pasado a nadie más?
- En un alto porcentaje de situaciones, alguien ya tuvo ese problema y lo resolvió.

Un patrón de diseño es una solución a un problema, que ya ha sido utilizada, y cuya efectividad ha sido probada; además, es reutilizable en problemas con circunstancias similares.

TIPOS DE PATRONES DE DISEÑO

Propuestos por *Gang of Four*.

		PROPÓSITO		
		Creacional	*Estructural*	*Comportamiento*
ÁMBITO	*Clase*	Factory Method	Adapter	Intepreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Bridge Composite Decorator Facade Flyweight Proxy	Chain of reponsibilty Command Iterator Mediator Memento Observer State Strategy Visitor

TIPOS DE PATRONES DE DISEÑO

Propuestos por otros autores:

- Inversión de control
- Inyección de dependencias
- *View Helper*
- Modelo-Vista-Controlador
- *Data Access Object*
- *Front-Controller*
- ...

¿QUÉ PATRONES DE DISEÑO USAREMOS CON SPRING?

- **Inyección de dependencias**
- *Singleton*: restringe la creación de objetos de un tipo a uno solo
- *Prototype*: permite la creación de nuevos objetos duplicándolos.
- Proxy: proporciona un sustituto o representante de un objeto, para controlar el acceso a este

Ámbitos de un bean

Spring AOP (Programación orientada a aspectos)

¿QUÉ PATRONES DE DISEÑO USAREMOS CON SPRING?

- Modelo-Vista-Controlador: separa la lógica y los datos de la interfaz y el control de peticiones.
- Front Controller: un solo controlador maneja todas las peticiones.
- *Factory*: centralización de la construcción de objetos.

Spring Web MVC

Spring IoC Container

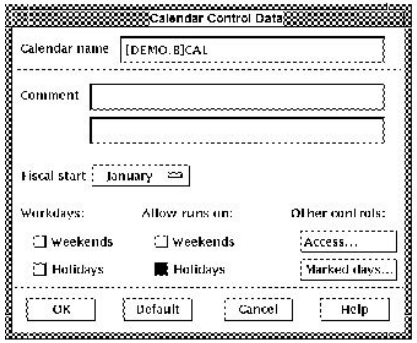
INVERSIÓN DE CONTROL E INYECCIÓN DE DEPENDENCIAS

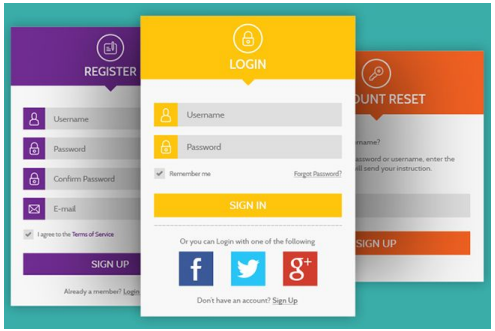
1.

INVERSIÓN DE CONTROL

INVERSIÓN DE CONTROL (IoC)

- Principio de diseño (o patrón)
- El objetivo es conseguir *desacoplar* objetos.





“

No nos llames.

Nosotros te llamaremos a tí.

Principio de Hollywood

INVERSIÓN DE CONTROL (IoC)

- Martin Fowler
- *Dejar que sea otro el que controle el flujo del programa* (por ejemplo, un framework)

```
#ruby
puts 'What is your name?'
name = gets
process_name(name)
puts 'What is your quest?'
quest = gets
process_quest(quest)
```

```
require 'tk'
root = TkRoot.new()
name_label = TkLabel.new() {text "What is Your Name?"}
name_label.pack
name = TkEntry.new(root).pack
name.bind("FocusOut") {process_name(name)}
quest_label = TkLabel.new() {text "What is Your Quest?"}
quest_label.pack
quest = TkEntry.new(root).pack
quest.bind("FocusOut") {process_quest(quest)}
Tk.mainloop()
```

Ejemplos propuestos por Martin Fowler

Ralph Johnson and Brian Foote

Journal of Object-Oriented Programming

Junio/Julio 1988

*Una característica importante de un **framework** es que los **métodos definidos por el usuario** para adaptar el mismo a menudo **serán llamados desde el framework**, en lugar de desde el código de aplicación del usuario. El framework a veces **desempeña el papel de programa principal** en la coordinación y secuenciación de actividad de la aplicación. Esta **inversión de control** proporciona al framework la posibilidad de servir como un **esqueleto extensible**. El usuario proporciona métodos que adaptan los algoritmos genéricos.*

ALGUNOS EJEMPLOS DE INVERSIÓN DE CONTROL

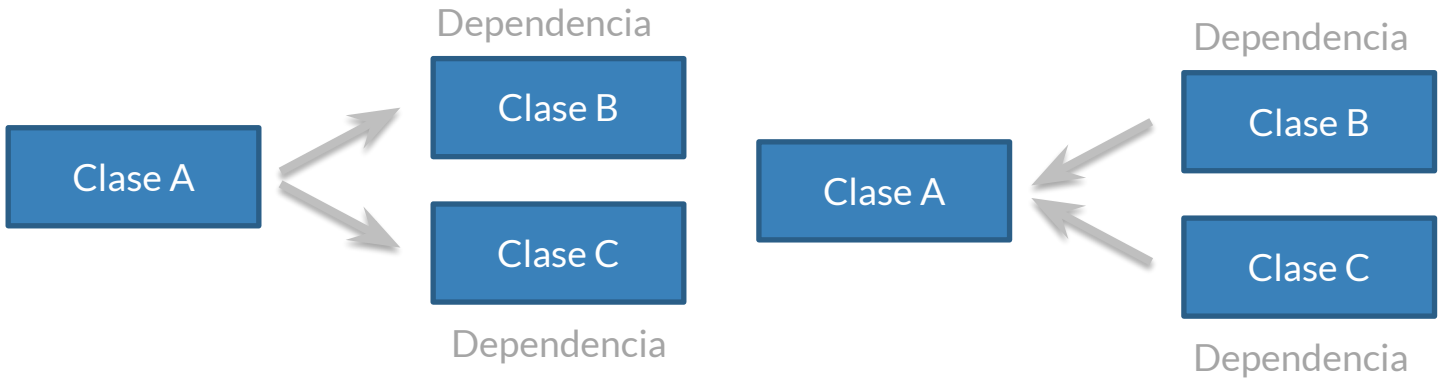
- Suscripción o manejo de eventos (.NET, Java, ...)
- Session Bean (EJB): `ejbRemove`, `ejbPassivate`, `ejbActivate`, ...
- JUnit: `setUp`, `tearDown`, ...
- Inyección de dependencias: es solo una forma de inversión de control.
- ....

2.

INYECCIÓN DE DEPENDENCIAS

INYECCIÓN DE DEPENDENCIAS

- Es una forma de inversión de control.



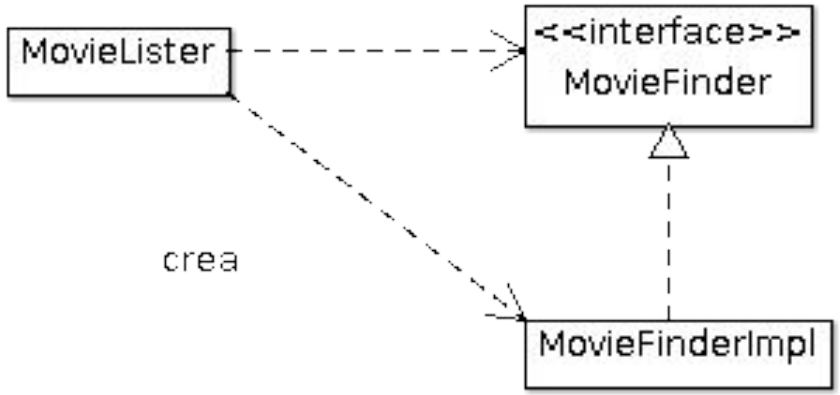
MOTIVACIÓN PARA EL USO DE INYECCIÓN DE DEPENDENCIAS

```
public class MovieLister {
  public Movie[] moviesDirectedBy(String arg)
  {
      List<Movie> allMovies = finder.findAll();

      for (Iterator it = allMovies.iterator(); it.hasNext();)
      {
        Movie movie = (Movie) it.next();
        if (!movie.getDirector().equals(arg)) it.remove();
      }

      return (Movie[]) allMovies.toArray(new
                                        Movie[allMovies.size()])
  }
}
```

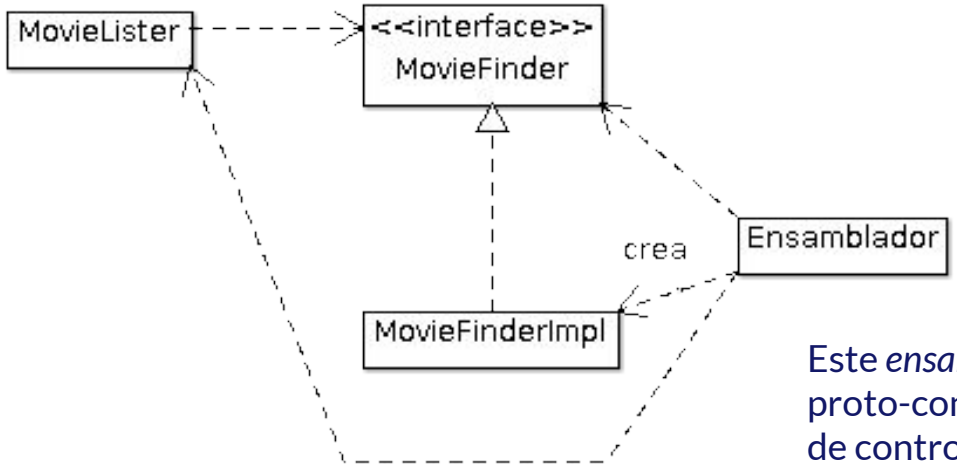
MOTIVACIÓN PARA EL USO DE INYECCIÓN DE DEPENDENCIAS

```
public interface MovieFinder
{
   List<Movie> findAll();
}

public class MovieLister {

  private MovieFinder finder;

  public MovieLister() {
    finder = new CSVMovieFinder("movies.txt");
  }
  //...
}
```

MOTIVACIÓN PARA EL USO DE INYECCIÓN DE DEPENDENCIAS



MOTIVACIÓN PARA EL USO DE INYECCIÓN DE DEPENDENCIAS



Este *ensamblador* es nuestro proto-contenedor de inversión de control.

EJEMPLO DE INYECCIÓN DE DEPENDENCIAS CON SPRING

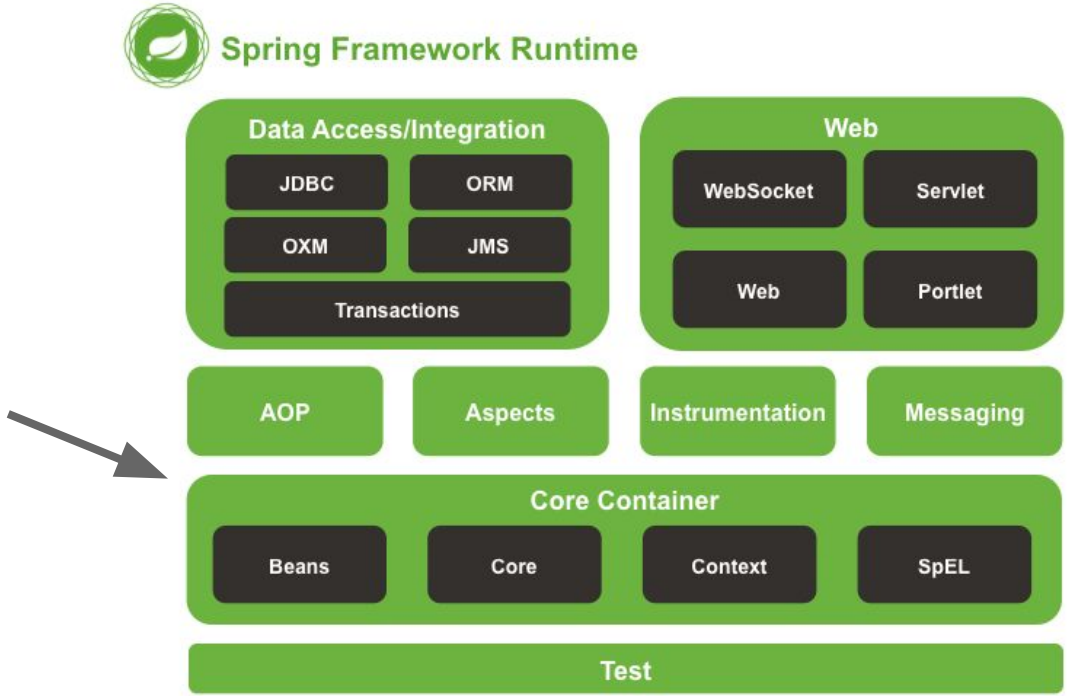
```
class MovieLister {
    private MovieFinder finder;

    public void setFinder(MovieFinder finder) {
       this.finder = finder;
    }
}

class CSVMovieFinder implements MovieFinder {

    public void setFilename(String filename) {
       this.filename = filename;
    }
}
```

EJEMPLO DE INYECCIÓN DE DEPENDENCIAS CON SPRING

```
<beans>
    <bean id="MovieLister" class="spring.MovieLister">
        <property name="finder">
            <ref local="MovieFinder"/>
        </property>
    </bean>
    <bean id="MovieFinder" class="spring.CSVMovieFinder">
        <property name="filename">
            <value>movies.txt</value>
        </property>
    </bean>
</beans>
```

CONTENEDOR DE INVERSIÓN DE CONTROL

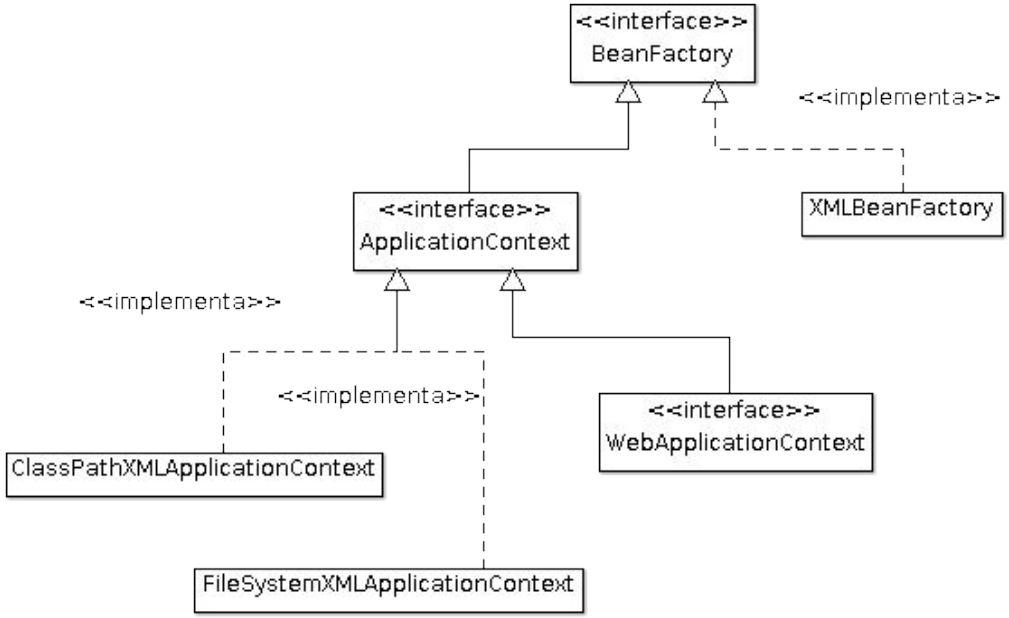
NOS SITUAMOS



LA BASE DEL *IoC CONTAINER*

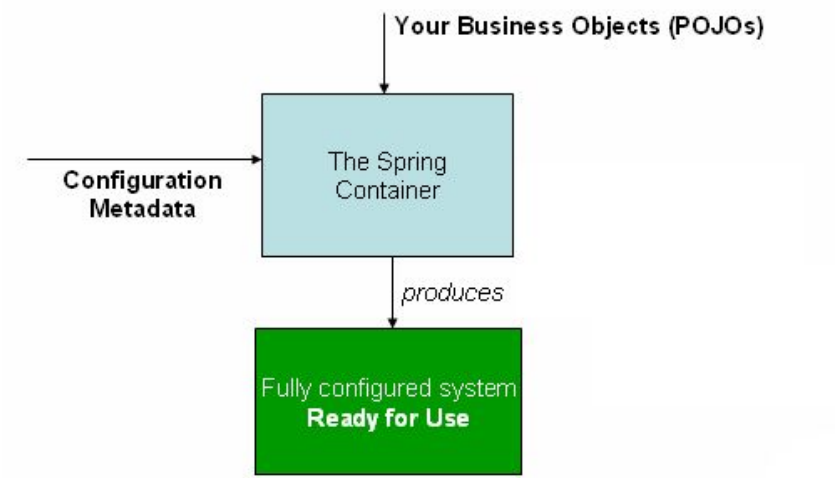
- Paquetes
  - `org.springframework.beans`
  - `org.springframework.context`
- Los elementos más básicos
  - `BeanFactory`: lo elemental para poder manejar cualquier ~~bean~~ objeto.
  - `ApplicationContext`: superset del anterior. Añade AOP, manejo de recursos, internacionalización, contextos específicos, ...

LA BASE DEL IoC CONTAINER



BEANS

- Se trata de un objeto (cualquiera) gestionado por nuestro contenedor de inversión de control.



Podríamos decir que es como un objeto *empoderado*.

¿CÓMO PUEDO USAR EL CONTENEDOR DE IoC?

- org.springframework.context.ApplicationContext
- Responsable de crear instancias, configurar y ensamblar los beans.
- Se consigue a través de los metadatos
  - XML
  - Anotaciones
  - Java

¿CÓMO PUEDO USAR EL CONTENEDOR DE IoC?

```java
public class App {

    public static void main(String[] args) {

        // Iniciamos el contexto
        ApplicationContext appContext = new ClassPathXmlApplicationContext("beans01.xml");

        // TODO
        // Utilizamos los beans

        // Cerramos el contexto
        ((ClassPathXmlApplicationContext) appContext).close();

    }

}
```

MI PRIMER BEAN

BEANS

- Objetos manejados por el contenedor *ioc*
- Se crean con los metadatos que nosotros proporcionamos (por ejemplo, xml).
- Debemos darle un id único dentro del contenedor.
- También debemos indicar la clase sobre la que definimos el bean.

BEANS

```
<beans xmlns="http://www.springframework.org/schema/beans" ...>

     <bean id="saludator"
                    class="com.openwebinars.beans.Saludator">
     </bean>

</beans>
```

Nombre completo de la clase
(incluyendo paquete y subpaquetes)

Debe ser único en el contexto

FORMAS DE OBTENER UN BEAN DEL CONTENEDOR

`appContext.getBean(id)`

- Nos obliga a hacer un casting

`appContext.getBean(id, class)`

- Dos argumentos

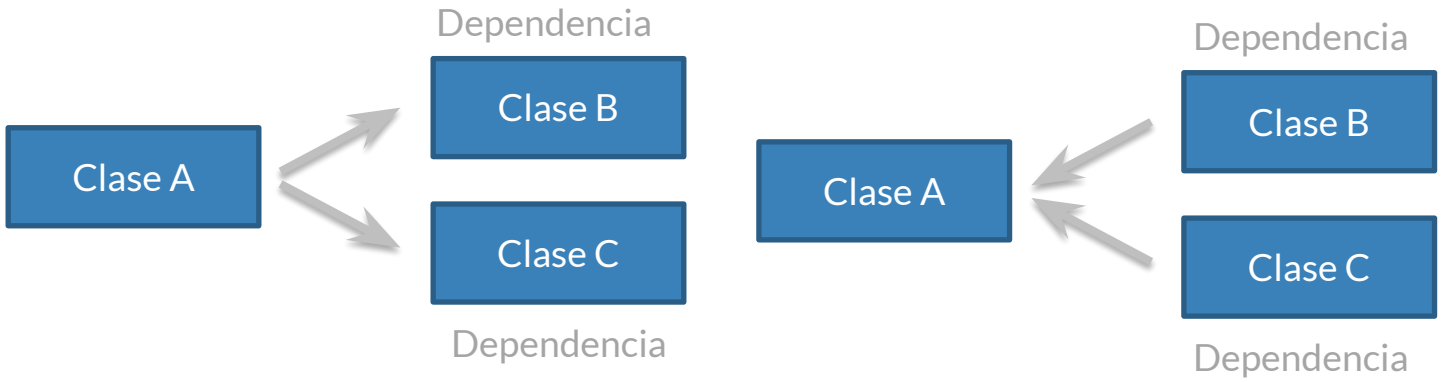
`appContext.getBean(class)`

- Excepción si repetido

INYECCIÓN DE DEPENDENCIAS

INYECCIÓN DE DEPENDENCIAS

- Es una forma de inversión de control.



TIPOS DE INYECCIÓN DE DEPENDENCIAS

- Vía *setter* (o *property*)
- Vía constructor
- Referencias entre beans

INYECCIÓN VÍA SETTER

```
public class Saludator {
     private String mensaje;
     public void setMensaje(String str) {
          this.mensaje = str;
     }
     //...
}
```

```
<beans xmlns="http://www.springframework.org/schema/beans"... >
     <bean id="saludator"
          class="com.openwebinars.beans.Saludator">
          <property name="mensaje"
               value="Hola alumnos de openwebinars"> </property>
     </bean>
</beans>
```

INYECCIÓN VÍA CONSTRUCTOR

```
public class Saludator {
     private String mensaje;
     public Saludator(String str) {
          this.mensaje = str;
     }
     //...
}
```

```
<beans xmlns="http://www.springframework.org/schema/beans"... >
     <bean id="saludator"
          class="com.openwebinars.beans.Saludator">
          <constructor-arg name="str"
           value="Hola alumnos de openwebinars"></constructor-arg>
     </bean>
</beans>
```

UN BEAN QUE REFERENCIA A OTRO BEAN

- Como cabe suponer, se puede inyectar dependencias más allá de tipos primitivos.

```
<beans xmlns="http://www.springframework.org/schema/beans"... >
     <bean id="emailService"
           class="com.openwebinars.beans.EmailService">
          <property name="saludator" ref="saludator"></property>
     </bean>

     <bean id="saludator" class="com.openwebinars.beans.Saludator">
          <property name="mensaje"
                    value="Hola alumnos de openwebinars"></property>
     </bean>
</beans>
```

BEANS QUE IMPLEMENTAN INTERFACES

```
public interface IEmailService {
     public void enviarEmailSaludo(String str);
}

public class EmailService implements IEmailService{
     //...
}

public class App {
     public static void main(String[] args) {
          //...
          IEmailService emailService = null;
          emailService = appContext.getBean(IEmailService.class);
          emailService.enviarEmailSaludo("luismi@openwebinars.net");
          //...
     }
}
```

OTROS ASPECTOS DE LOS BEANS QUE NOS DEJAREMOS EN EL TINTERO

- Inner beans (beans anidados)
  - Definimos un bean dentro de otro.
  - Similar a las referencias.
  - Ámbito más restrictivo.
- Colecciones
  - Posibilidad de inyectar valores de una colección
  - <list>, <set>, <map>, <props>

INYECCIÓN AUTOMÁTICA O AUTOCABLEADO

INYECCIÓN AUTOMÁTICA

- Spring permite la inyección *automática* entre beans que *se necesitan*.
- Busca candidatos dentro del contexto.
- Ventajas
  - Reduce la configuración necesaria
  - Útil durante el desarrollo. Permite requerir objetos sin configurarlo explícitamente.

INYECCIÓN AUTOMÁTICA
AUTOWIRED

```
<beans xmlns="http://www.springframework.org/schema/beans" ...>

     <bean id="emailService"
           class="com.openwebinars.beans.EmailService"
           autowire="byType">

     </bean>

     <bean id="saludator"
           class="com.openwebinars.beans.Saludator" >
          <property name="mensaje"
             value="Hola alumnos de openwebinars"></property>
     </bean>
</beans>
```

TIPOS DE AUTOWIRED

- **`no`**: sin autocableado
- **`byName`**: en función del nombre de la propiedad requerida.
- **`byType`**: en función del tipo de la propiedad requerida. Si hay más de un bean de este tipo, se produce excepción.
- **`constructor`**: análogo a *byType*, pero para argumentos del constructor.

INCONVENIENTES DEL AUTOCABLEADO

- Es útil si se usa siempre en un proyecto.
- En otro caso, puede ser confuso.
- No se pueden *autoinyectar* tipos primitivos o String.
- Menos exacto que la inyección explícita
- Posible ambigüedad en inyección *byType*.

INCONVENIENTES DEL AUTOCABLEADO: ¿QUÉ HACER?

- No usar el autocableado :(
- Manejar el autocableado a través de anotaciones (lo estudiaremos más adelante).
- Utilizar `autowired-candidate=false` en los beans más conflictivos.
- Utilizar `primary=true` en las opciones principales.

AUTOWIRED-CANDIDATE

- Nos permite excluir a un bean de ser autoinyectado.
- Se sigue permitiendo que se pueda inyectar de forma explícita.

```
<beans xmlns="http://www.springframework.org/schema/beans" ...>

    <bean id="saludator"
          class="com.openwebinars.beans.Saludator"
          autowire-candidate="false" >
        ...
    </bean>
</beans>
```

PRIMARY

- Nos permite indicar que un bean de un tipo tendrá preferencia sobre los demás del mismo tipo.

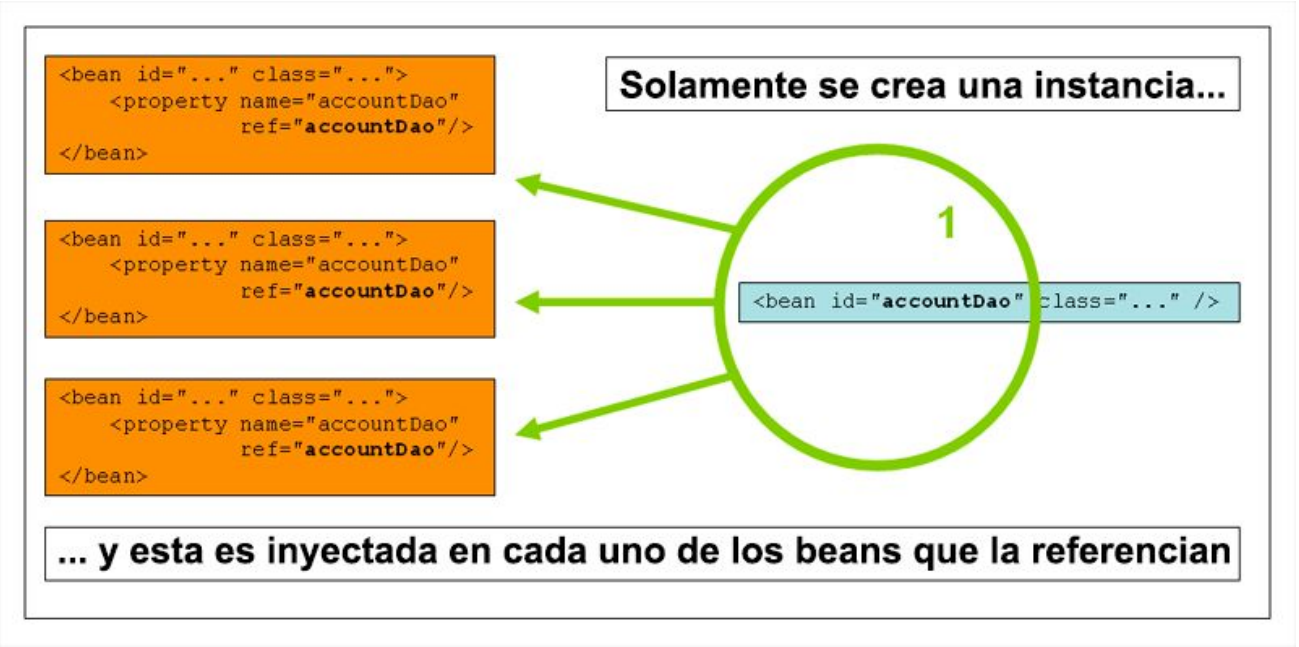
```
<beans xmlns="http://www.springframework.org/schema/beans" ...>
     <bean id="emailService"
           class="com.openwebinars.beans.EmailService"
           autowire="byType">
     </bean>
     <bean id="saludator"
            class="com.openwebinars.beans.Saludator"
            primary="true" >
            ...
     </bean>
     <bean id="englishSaludator"
           class="com.openwebinars.beans.Saludator">
           ...
     </bean>
</beans>
```

ÁMBITOS DE UN BEAN: SINGLETON Y PROTOTYPE

ÁMBITO DE UN BEAN

- Cuando definimos un bean (por ejemplo, en XML) creamos una **receta**.
- Esta nos permite crear instancias reales de la clase sobre la que definimos el bean.
- A partir de esa *receta*, es posible crear muchas instancias.
- Podemos configurar el ámbito de un bean a nivel de configuración (sin tener que hacerlo al *viejo estilo*).

ÁMBITO SINGLETON

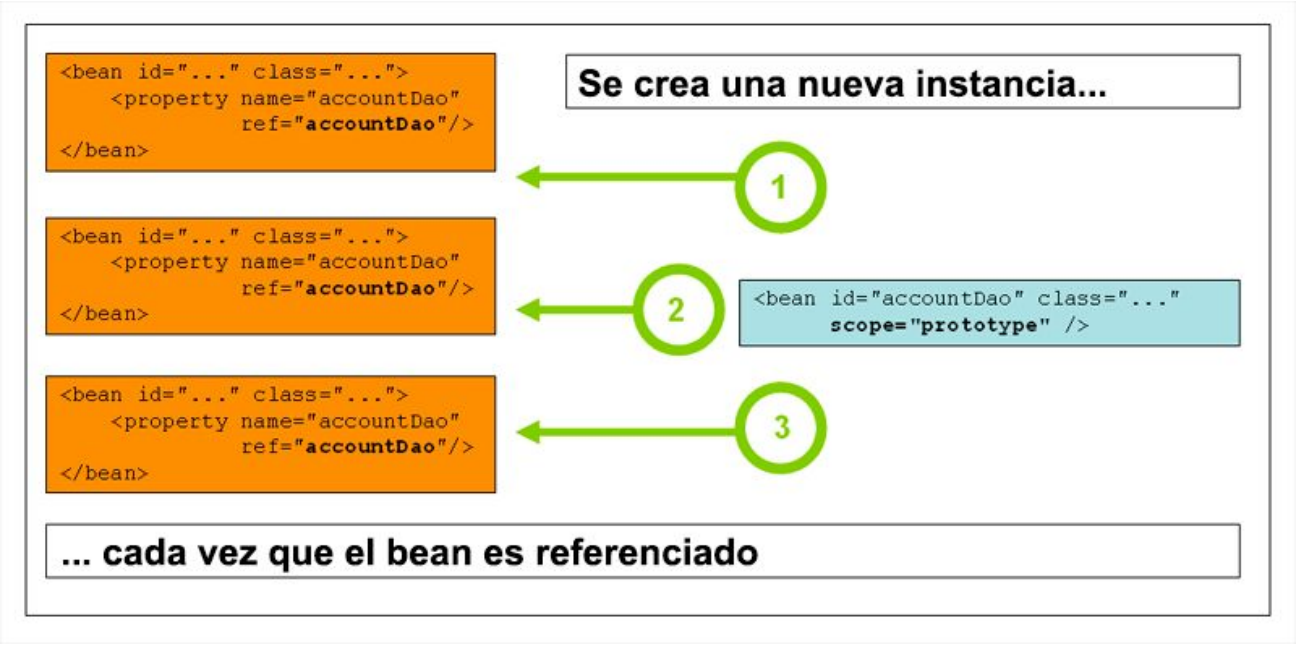


ÁMBITO SINGLETON

- Solamente se crea una instancia compartida de esa clase.
- Toda las referencias que obtengamos a ese bean serán el mismo objeto en memoria.
- Es el ámbito por defecto.

```
<bean id="emailService"
      class="com.openwebinars.beans.EmailService"
      scope="singleton">
      ....
</bean>
```

ÁMBITO PROTOTYPE



ÁMBITO PROTOTYPE

- Se crea una instancia cada vez que se le requiere.
- Estas instancias se crean en tiempo de ejecución.

```
<bean id="emailService"
      class="com.openwebinars.beans.EmailService"
      scope="prototype">
      ....
</bean>
```

OTROS ÁMBITOS

OpenWebinars

ÁMBITOS WEB

- Solo disponibles con un contexto web (por ejemplo, `XmlWebApplicationContext`).
- En caso de usar otros no web (por ejemplo `ClassPathXmlApplicationContext`) producirá excepción.
- Ámbitos
  - Request, Session, Application, WebSocket.

ÁMBITO REQUEST

- Se creará un objeto por cada petición HTTP.
- Cuando la petición termina de procesarse, se descarta el objeto.

```
<bean id="loginAction"
      class="com.foo.LoginAction"
      scope="request"/>
```

ÁMBITO SESSION

- Se creará un objeto por cada sesión HTTP que se cree.
- Cuando la sesión se destruye, se descarta el objeto.

```
<bean id="userPreferences"
      class="com.foo.UserPreferences"
      scope="session"/>
```

ÁMBITO APPLICATION

- Se creará un objeto por cada `ServletContext`.
- En la práctica, se trata de un objeto por cada instancia de la aplicación.

```
<bean id="appPreferences"
      class="com.foo.AppPreferences"
      scope="application"/>
```

CICLO DE VIDA DE UN BEAN

CICLO DE VIDA DE UN BEAN

- Podemos interactuar mediante *callbacks*
- Después de que se instancie/Antes de que se destruya.
- Interfaces `InitializingBean` y `DisposableBean`.
- También a través de XML.

INTERFACES InitializingBean Y DisposableBean.

- El bean debe implementar los interfaces.
- Desventaja: configuración e implementación van unidos.
- Ventaja: la interfaz es un contrato; nos da la firma del método a implementar.

INTERFACES InitializingBean Y DisposableBean.

```
public class PersonaDAOImplMemory implements
          PersonaDAO, InitializingBean, DisposableBean {

     @Override
     public void afterPropertiesSet() throws Exception {
          insert(new Persona("Luismi", 35));
          insert(new Persona("Ana", 32));
          insert(new Persona("Pepe", 34));
          insert(new Persona("Julia", 39));
     }
     @Override
     public void destroy() throws Exception {
          System.out
             .println("Limpiando los datos de la lista");
          personas.clear();
     }
}
```

CONFIGURACIÓN VÍA XML

- Propiedades `init-method` y `destroy-method`
- El valor de la propiedad es un String.
- Debe ser el nombre de un método del bean.
  - void
  - Sin parámetros
  - Throws Exception.
- Ventaja: bajo acoplamiento

CONFIGURACIÓN VÍA XML

```
<beans xmlns="http://www.springframework.org/schema/beans"...>

   <bean id="personaDao"
        class="com.openwebinars.lifecycle.PersonaDAOImplMemory"
        init-method="init" destroy-method="destroy" />
</beans>

public class PersonaDAOImplMemory implements PersonaDAO {

  public void init() throws Exception { … }

  public void destroy() throws Exception { … }

}
```

CONFIGURACIÓN VÍA XML

- Propiedades `default-init-method` y `default-destroy-method`
- Propiedad de `<beans>`
- Permite definir un nombre de método de inicialización/destrucción para todos los beans de un contexto.

```
<beans xmlns="http://www.springframework.org/schema/beans"...
  default-init-method="init" default-destroy-method="destroy">
   <bean id="personaDao"
    class="com.openwebinars.lifecycle.PersonaDAOImplMemory"/>
</beans>
```

CONFIGURACIÓN A TRAVÉS DE ANOTACIONES. USO DE @REQUIRED

1.

CONFIGURACIÓN A TRAVÉS DE ANOTACIONES

CONFIGURACIÓN A TRAVÉS DE ANOTACIONES

- No configuramos los beans con XML.
- La configuración está más cerca del código.
- Más cerca significa más acoplada.
- Desde Spring 2.5
- Necesitamos configuración XML mínima (o JavaConfig)

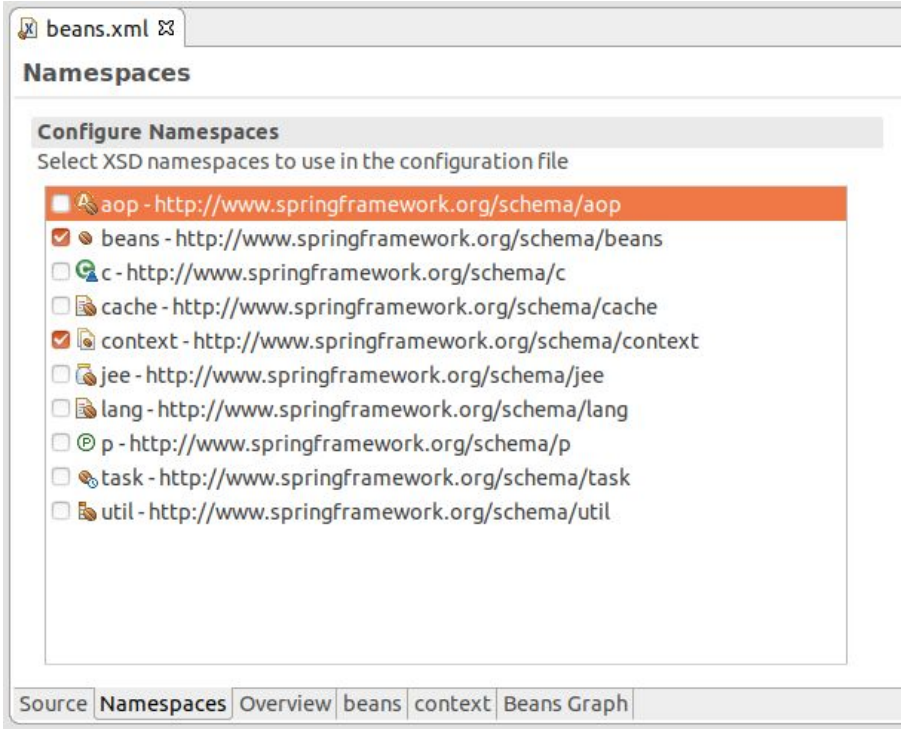
UNA PALABRA SOBRE BEANPOSTPROCESSOR

- Spring permite extender el contenedor de IoC.
- `BeanPostProcessor` es un interfaz que nos permite gestionar:
  - Instanciación
  - Configuración
  - Inyección de dependencias
- Podemos definir cuantos procesadores necesitemos.

UNA PALABRA SOBRE BEANPOSTPROCESSOR

- Spring define algunos `BeanPostProcessor` útiles:
  - `AutowiredAnnotationBeanPostProcessor.`
  - `RequiredAnnotationBeanPostProcessor.`
  - `CommonAnnotationBeanPostProcessor.`
  - `PersistenceAnnotationBeanPostProcessor.`
  - `...`

CONFIGURACIÓN A TRAVÉS DE ANOTACIONES



Debemos añadir algunos espacios de nombres y esquemas adicionales al encabezado de nuestro XML (en particular, `context`).

CONFIGURACIÓN A TRAVÉS DE ANOTACIONES

- La anotación `<context:annotation-config />` nos permite registrar los `BeanPostProcessor` necesarios para usar las anotaciones.

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

2.

@REQUIRED

USO DE @REQUIRED

- Nos permite indicar que una propiedad debe ser necesariamente inyectada.
- No indica cómo cómo debe realizarse la inyección
  - Explícita
  - Autowired
  - ...
- Si no se satisface, produce una excepción.
- Permite evitar NPE

USO DE @AUTOWIRED

USO DE @AUTOWIRED

- Tiene el mismo efecto que la configuración vía XML.
- Busca un bean adecuado y lo inyecta en la dependencia.
- Se realiza un autocableado *byType*

¿DÓNDE PUEDO USAR @AUTOWIRED?

- Método setter

```
@Autowired
public void setPeliculaDao(PeliculaDao peliculaDao) {...}
```

- Definición de la propiedad

```
@Autowired
private PeliculaDao peliculaDao;
```

- Constructor

```
@Autowired
public PeliculaService(PeliculaDao peliculaDao) {...}
```

USO DE @AUTOWIRED

- Se pueden mezclar los 3 tipos de uso de autowired.
  - En la propiedad es muy cómodo.
  - Si el método setter tiene alguna “lógica especial”, sería adecuado.
  - Para atributos *final*, usamos el constructor.

@AUTOWIRED DE VARIOS OBJETOS DE DIFERENTE TIPO.

- No hay limitación en el número de argumentos de un método anotadio con @Autowired.

```
public class MovieRecommender {
    private MovieCatalog movieCatalog;
    private CustomerPreferenceDao customerPreferenceDao;

    @Autowired
    public void prepare(MovieCatalog movieCatalog,
            CustomerPreferenceDao customerPreferenceDao) {
        this.movieCatalog = movieCatalog;
        this.customerPreferenceDao = customerPreferenceDao;
    }
    // ...
}
```

USO DE @AUTOWIRED PARA VARIOS OBJETOS DEL MISMO TIPO

- Podemos obtener todos los beans de un mismo tipo
  - Array
  - Colección: List, Set, Map

```
public class PeliculaDaoImplMemory implements PeliculaDao {
     @Autowired
     private Set<CatalogoPeliculas> catalogosPeliculas;
}

<bean id="catalogoClasicas"
class="com.openwebinars.annotation.CatalogoPeliculasClasicas" />
<bean id="catalogoActuales"
class="com.openwebinars.annotation.CatalogoPeliculasActuales" />
```

@AUTOWIRED NO SATISFECHO

- Si @Autowired no encuentra ningún bean candidato produce excepción.
- Podemos modificar este comportamiento para que deje la dependencia sin satisfacer, pero sin excepción:
  - @Autowired(required=false)
  - @Nullable (Spring 5)
  - Optional<?> (Java 8)

USO DE PRIMARY Y @QUALIFIER

PROBLEMAS CUANDO HAY MÁS DE UN BEAN DE UN TIPO

- @Autowired no funciona correctamente cuando hay más de un candidato de un mismo tipo.
- Necesitamos un mecanismo que nos permita indicar qué bean es el más adecuado o el seleccionado.

PROPIEDAD PRIMARY

- A nivel XML
- Permite indicar un bean *primus inter pares*
- Si hay más de un bean de un tipo, y uno de ellos marcado como `primary`, @Autowired inyectará dicho bean.
- Si hay más de un bean `primary`, se lanza excepción.
- Valor por defecto `primary=false`

USO DE @QUALIFIER

- Nos permite afinar mucho más el autocableado.
- Podemos *seleccionar* que bean específico (de entre varios de un tipo) queremos inyectar.
- Mecanismo extensible.

USO DE @QUALIFIER

- El mecanismo más sencillo es usar el nombre del bean.

```
public class PeliculaDaoImplMemory
                                implements PeliculaDao {

    @Autowired
    @Qualifier("catalogoClasicas")
    private CatalogoPeliculas catalogoPeliculas;

    //...

}
```

USO DE @QUALIFIER

- También podemos usar @Qualifier a nivel de argumento de un método.

```
public class MovieRecommender {

    @Autowired
    public void prepare(
       @Qualifier("main") MovieCatalog movieCatalog,
       CustomerPreferenceDao customerPreferenceDao) {
        this.movieCatalog = movieCatalog;
        this.customerPreferenceDao = customerPreferenceDao;
    }
    // ...
}
```

USO DE @QUALIFIER

- Podemos indicar explícitamente en XML un valor de *qualifier*

```
<bean class="example.SimpleMovieCatalog">
     <qualifier value="main"/>
</bean>

<bean class="example.SimpleMovieCatalog">
     <qualifier value="action"/>
</bean>
```

EXTENDIENDO @QUALIFIER

- Podemos crear nuestras propias anotaciones para extender @Qualifier

```
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Epoca {

     String value();

}
```

EXTENDIENDO @QUALIFIER

- De esta forma, el código es más legible

```
<bean id="catalogoClasicas"
  class="com.openwebinars.annotation.CatalogoPeliculasClasicas">
  <qualifier type="Epoca" value="clasicas" />
</bean>

public class PeliculaDaoImplMemory implements PeliculaDao {

     @Autowired
     @Epoca("clasicas")
     private CatalogoPeliculas catalogoPeliculas;

}
```

USO DE @POSTCONSTRUCT Y @PREDESTROY

CICLO DE VIDA DE UN BEAN

- Ya hemos visto cómo manejar el ciclo de vida vía XML.

```
<beans xmlns="http://www.springframework.org/schema/beans"...>
   <bean id="personaDao"
        class="com.openwebinars.lifecycle.PersonaDAOImplMemory"
        init-method="init" destroy-method="destroy" />
</beans>

public class PersonaDAOImplMemory implements PersonaDAO {

  public void init() throws Exception { … }
  public void destroy() throws Exception { … }

}
```

CICLO DE VIDA DE UN BEAN

- Podemos usar las anotaciones @PostConstruct y @PreDestroy

```
<beans xmlns="http://www.springframework.org/schema/beans"...>
   <bean id="personaDao"
     class="com.openwebinars.lifecycle.PersonaDAOImplMemory"/>
</beans>

public class PersonaDAOImplMemory implements PersonaDAO {

  @PostConstruct
  public void init() { … }
  @PreDestroy
  public void destroy() { … }

}
```

USO DE ESTEREOTIPOS

1.

ESCANEO DE COMPONENTES

ESCANEO DE COMPONENTES

- Hasta ahora hemos tenido que declarar en XML todos nuestros beans.
- Las anotaciones permiten ahorrarnos mucho xml *verboso*.
  - Aun así, es necesaria la declaración xml.
- Spring proporciona la posibilidad de *detectar candidatos* a ser beans gestionados por el contenedor.

ESCANEO DE COMPONENTES

- Los candidatos serán clases específicas (incluso según un criterio de búsqueda) y que tengan la metainformación necesaria.
- Declaramos un paquete base sobre el que hacer el escaneo:

```
<beans ...>
     <context:component-scan
      base-package="com.openwebinars.stereotypes" />
</beans>
```

ESCANEO DE COMPONENTES

- Los candidatos estarán anotados con algún estereotipo. El más básico es @Component.

```
@Component
public class PeliculaService {

    @Autowired
    private PeliculaDao peliculaDao;

    //..
}
```

2.

ESTEREOTIPOS

ESTEREOTIPOS

- **@Component**
  - Estereotipo básico
  - Los demás son derivados de él.
- **@Service**: orientado a las clases servicio, lógica de negocio, ...
- **@Repository**: clases de acceso a datos (DAO)
- **@Controller**: clases que sirven para gestionar las peticiones recibidas.

CONFIGURACIÓN USANDO JAVA

JAVACONFIG

- Spring soporta la configuración vía código Java.
- Nos permite prescindir por completo de XML.
- Podemos combinar el uso de JavaConfig con las anotaciones trabajadas en el bloque anterior.

ANOTACIONES CLAVE

- @Configuration
  - A nivel de clase
  - Indica que una clase va a definir uno o más @Bean
- @Bean.
  - A nivel de método
  - Equivalente a <bean … />

JAVACONFIG BÁSICO

```
@Configuration
public class AppConfig {

    @Bean
    public Saludator saludator() {
        return new Saludator();
    }

}
```

INSTANCIACIÓN DEL CONTENEDOR

- Ahora usamos `AnnotationConfigApplicationContext`
- Recibe como argumento la/s clase/s que tienen alguna configuración.

```
public static void main(String[] args) {
   ApplicationContext appContext = new
            AnnotationConfigApplicationContext(AppConfig.class);
   Saludator saludator = appContext.getBean(Saludator.class);
   //…
}
```

INSTANCIACIÓN DEL CONTENEDOR

- Podemos usar el constructor vacío y registrar las clases.

```
public static void main(String[] args) {
    AnnotationConfigApplicationContext ctx =
                        new AnnotationConfigApplicationContext();
    ctx.register(AppConfig.class, OtherConfig.class);
    ctx.register(AdditionalConfig.class);
    ctx.refresh();
    MyService myService = ctx.getBean(MyService.class);
    myService.doStuff();
}
```

ESCANEO DE COMPONENTES

- Idéntico comportamiento que en XML
- @ComponentScan(basePackages=...)
- También programáticamente

```
public static void main(String[] args) {
    AnnotationConfigApplicationContext ctx =
                        new AnnotationConfigApplicationContext();
    ctx.scan("com.acme");
    ctx.refresh();
    MyService myService = ctx.getBean(MyService.class);
}
```

USO DE @BEAN

@BEAN

- Anotación a nivel de método
- Define un bean <bean … />
- En clases
  - **@Configuration: preferible**
  - @Component (y derivados): lite mode
- Atributos: *name*, *init-method*, *destroy-mehtod*, *autowiring*
  - Con este enfoque, seguramente esto se configure vía anotaciones

DECLARACIÓN DE UN BEAN

- Un método anotado @Bean
- El tipo de retorno es el tipo del bean
- El nombre por defecto es el nombre del método.

```
@Bean
public TransferServiceImpl transferService() {
    return new TransferServiceImpl();
}
```

```
<bean id="transferService"
                  class="com.acme.TransferServiceImpl"/>
```

DECLARACIÓN DE UN BEAN

- El tipo de retorno puede ser un interfaz o supertipo de la clase instanciada.

```
@Bean
public TransferService transferService() {
    return new TransferServiceImpl();
}
```

DEPENDENCIAS DE UN BEAN

- Un método @Bean puede recibir cero o más parámetros.
- Los objetos son dependencias del bean definido.
- El contenedor inyectará las mismas (al estilo de la inyección por constructor)

DEPENDENCIAS DE UN BEAN

```
@Configuration
public class AppConfig {

  @Bean
  public TransferService transferService(
                    AccountRepository accountRepository) {
      return new TransferServiceImpl(accountRepository);
    }
}
```

El contenedor se encargará de inyectar la dependencia de tipo *AccountRepository*.

ÁMBITO DE UN @BEAN

- Podemos definir su ámbito a través de anotaciones.
  - *Scope("singleton")*: por defecto
  - *Scope("prototype")*
  - *@RequestScope*
  - *@SessionScope*
  - *@ApplicationScope*

En un contexto web

USO DE @PRIMARY

- Ante varios beans de un tipo, es el primer candidato (*primus inter pares*)
- A nivel de clase (@Component y derivados)
- A nivel de método (@Bean)

USO DE @PRIMARY

```
@Primary
@Component
public class HibernateFooRepository extends FooRepository
{
     public HibernateFooRepository(
                         SessionFactory sessionFactory) {
          // ...
     }
 }
```

USO DE @PRIMARY

```
@Configuration
public class MovieConfiguration {

    @Bean
    @Primary
    public MovieCatalog firstMovieCatalog() { ... }

    @Bean
    public MovieCatalog secondMovieCatalog() { ... }

    // ...
}
```

INYECCIÓN DE VALORES
@VALUE

- ¿Cómo podemos inyectar valores de tipo primitivo (por ejemplo, String)?
- @Value
  - Uso de ficheros de properties
  - Variables de entorno
  - Valores por defecto
- En realidad, podemos inyectar valores en otros tipos: wrapper, List, ...

INYECCIÓN DE VALORES @VALUE

```
mensaje=Hola a todos desde un fichero de propiedades!
```

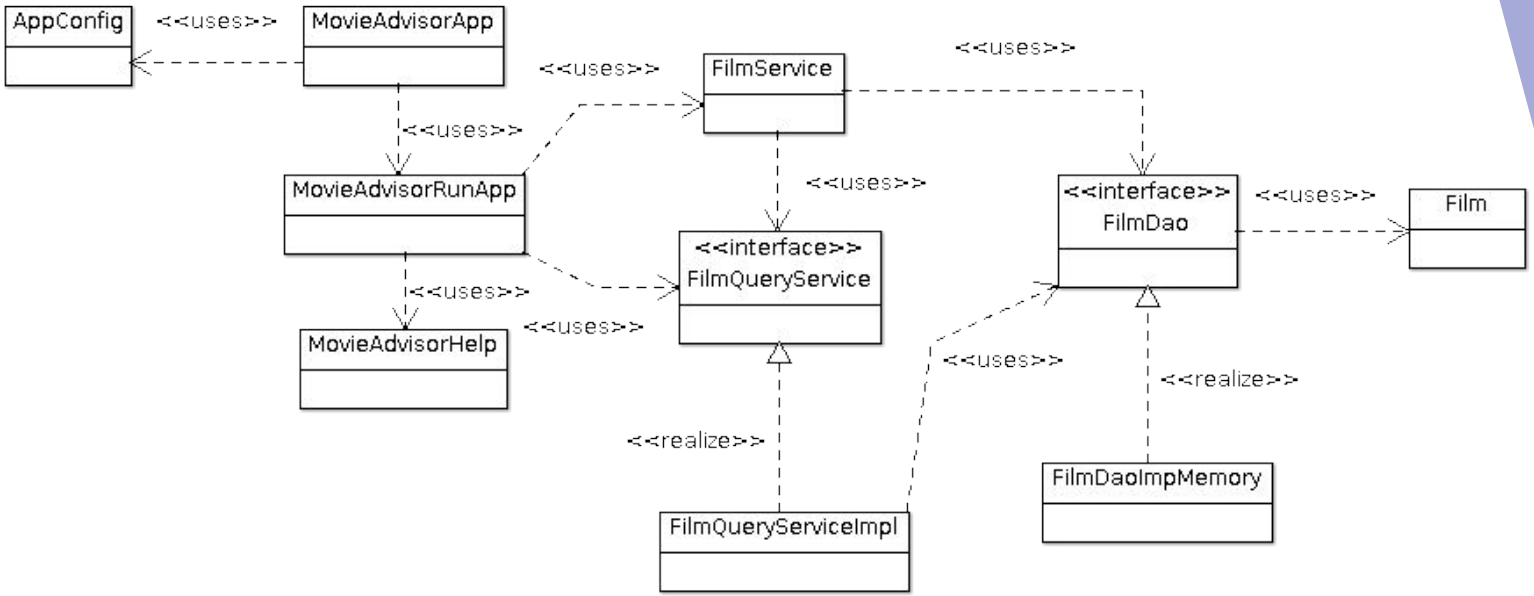
```
@Component
public class Saludator {

     @Value("${mensaje}")
     private String mensaje;

     public String saludo() {
          return mensaje;
     }

}
```

PROYECTO DE EJEMPLO: MOVIEADVISOR

PROYECTO DE EJEMPLO

- MovieAdvisor
- Sencillo recomendador de películas
- Datos de (casi) todas las películas de la historia (gracias IMDB).
- Herramienta CLI

DIAGRAMA DE CLASES



SINTAXIS DE LA APLICACIÓN

java -jar movieadvisor.jar [OPCIONES]

- -lg
  - listar los géneros
- -ag genero1,genero2,genero…
  - Películas que pertenezcan a algún genero del listado
- -tg genero1,genero2,genero…
  - Películas que pertenezcan a todos los generos del listado

SINTAXIS DE LA APLICACIÓN

- -y año
  - Películas estrenadas en dicho año
- -b desde,hasta
  - Películas estrenadas en el intervalo [desde,hasta]
- -t titulo
  - Películas cuyo título contega dicha cadena
- -h
  - Muestra el mensaje de ayuda.