AMS Bootcamp Fullstack Developer.

**Spring Core Notebook**

Índice

[1. Personal Notebook 2](#_Toc810438019)

[1.1 Datos alumno 3](#_Toc102121761)

[1.2 Personal Notebook 3](#_Toc235868087)

[2. Spring Framework 4](#_Toc1305482105)

[2.1 Objetivos de aprendizaje 5](#_Toc1544225856)

[2.2 Introducción a Spring e instalación inicial 6](#_Toc118243067)

[2.2.1 Objetivos 6](#_Toc289656652)

[2.2.2 Recursos formativos 6](#_Toc1847238221)

[2.2.3 Posibles preguntas 6](#_Toc2069778104)

[2.2.4 Tareas 6](#_Toc282322907)

[2.2.5 Dudas 6](#_Toc22852450)

[2.3 Qué es IoC e Inyección de dependencias 7](#_Toc888921285)

[2.3.1 Objetivos 7](#_Toc1521154812)

[2.3.2 Recursos formativos 7](#_Toc443216939)

[2.3.3 Posibles preguntas 7](#_Toc450159758)

[2.3.4 Tareas 7](#_Toc480609454)

[2.3.5 Dudas 7](#_Toc1598566544)

[2.4 Ejemplo de IoC e Inyección en Java 8](#_Toc1193314245)

[2.4.1 Objetivos 8](#_Toc21341433)

[2.4.2 Recursos formativos 8](#_Toc1565458720)

[2.4.3 Posibles preguntas 8](#_Toc455966070)

[2.4.4 Tareas 8](#_Toc2040347710)

[2.4.5 Dudas 8](#_Toc481280269)

[2.5 Ciclo de vida de los Beans 9](#_Toc2080245869)

[2.5.1 Objetivos 9](#_Toc1158382265)

[2.5.2 Recursos formativos 9](#_Toc253705643)

[2.5.3 Posibles preguntas 9](#_Toc729716350)

[2.5.4 Tareas 9](#_Toc2047045169)

[2.5.5 Dudas 9](#_Toc465475980)

[2.6 Configuración por Anotación 10](#_Toc275297976)

[2.6.1 Objetivos 10](#_Toc530609351)

[2.6.2 Recursos formativos 10](#_Toc1920774602)

[2.6.3 Posibles preguntas 10](#_Toc97747897)

[2.6.4 Tareas 10](#_Toc1181336253)

[2.6.5 Dudas 11](#_Toc1504396763)

[2.7 Ejemplo de IoC e Inyección en Spring 11](#_Toc1645289063)

[2.7.1 Objetivos 11](#_Toc1787453528)

[2.7.2 Recursos formativos 11](#_Toc1957008561)

[2.7.3 Posibles preguntas 11](#_Toc1863156334)

[2.7.4 Tareas 11](#_Toc1164858436)

[2.7.5 Dudas 11](#_Toc1070156885)

# Personal Notebook

## Datos alumno

* DAS : A926020
* Nombre Apellidos : Sandra Moriana Herrera
* Email : sandra.moriana@eviden.com
* URL de GitHub corporativo:

## Personal Notebook

Este es tu cuaderno de trabajo. En él tendrás que registrar tu aprendizaje, tus avances y tus dudas. ¡Vas a generar tu propio material!

Puede ser una gran herramienta si la utilizas correctamente.

El cuaderno estará compartido con los tutores y el resto de los colaboradores para ver tu progreso y ayudarte.

Al final del Bootcamp tendrás que entregarlo como evidencia de tu aprendizaje.



Instrucciones de uso:

El fichero está ubicado en tu carpeta del espacio que tenemos en Teams.

Siempre deberá ser editado en modo online y/o con guardado automático en la nube.

Responde a las preguntas con la información que necesites para dar una respuesta cualificada al formador o a tus compañeros. No es necesario copiar texto solo por “rellenar”. Cita las Fuentes en caso de utilizar otros recursos.

# Spring Framework

## Objetivos de aprendizaje

Objetivo: **Adquirir los conocimientos básicos del framework Spring.**

Conceptos principales a adquirir:

* IoC
* ID
* Anotaciones
* Configuración de Spring
* Instalación de entornos (IDE) en Spring

Lecciones:

1. Introducción a Spring e instalación inicial
2. Qué es IoC e Inyección de dependencias
3. Ejemplo de IoC e Inyección en Java
4. Ciclo de vida de los Beans
5. Configuración por Anotación
6. Ejemplo de IoC e Inyección en Spring

## Introducción a Spring e instalación inicial

### Objetivos

1. **Conocer los motivos para usarlo y sus ventajas.**
2. **Componentes de Spring.**
3. **Tener un entorno de desarrollo (IDE+servidor) listo para trabajar**

### Recursos formativos

<https://www.youtube.com/watch?v=kFIvslQQZ9k&list=PLU8oAlHdN5Blq85GIxtKjIXdfHPksV_Hm>

(PildorasInformáticas videos del 1 al 5)

### Posibles preguntas

**¿Qué es Spring?**

Es un framework para el desarrollo de aplicaciones Java. Un framework es un "entorno de trabajo" que proporciona herramientas y reglas que facilitan el desarrollo de aplicaciones. Actualmente, Spring es el framework más utilizado para desarrollar aplicaciones Java, dado que es más ligero y fácil de usar frente a otras alternativas como JEE.

**Indica los motivos por los que usar Spring**

En general, el motivo por el cual utilizar Spring es para optimizar el desarrollo de código en aplicaciones java de forma fácil y rápida. Entre sus ventajas destacan las siguientes:

* Permite la inyección de dependencias, lo que favorece el “loose copling” (débil acoplamiento) lo que permite a su vez que los cambios que se requieran hacer en una parte del código no tengan que repetirse en otras partes o componentes, mejorando el rendimiento.
* Permite el desarrollo sencillo de POJOs (Plain Old Java Objects), que son clases sencillas que no heredan de ninguna otra ni implementan ninguna interfaz.
* Minimiza el código repetitivo (boilerplate code).
* Simplifica el acceso a datos.
* Incluye la Programación Orientada a Aspectos (AOP), lo que permite una alta modularización de las aplicaciones para separar o independizar tareas o funcionalidades transversales.

**Partes del framework (módulos y que hacen)**

Spring se estructura en los siguientes módulos y submódulos:

* Core Container: Permite la creación de beans (componentes reutilizables para construir componentes más complejos dentro de un programa), la lectura de archivos config, el manejo de propiedades y dependencias. Se divide a su vez en los siguientes módulos:
  + Core: Proporciona la base del framework. Permite inyectar beans y dependencias, gestionando el contenedor IoC (componente que gestiona las dependencias entre clases y/o objetos).
  + Beans: Permite la crear os beans e instanciarlos.
  + Context: Almacena los beans en memoria y simplifica la carga de recursos.
  + SpEL (Spring Expression Language): Implementa este lenguaje que permite manipular objetos en tiempo de ejecución.
* Infraestructura: Maneja todo lo referente a transacciones, logueos y seguridad de las aplicaciones. Incluye los siguientes submódulos:
  + AOP**:** Lenguaje Programado a Aspectos, que permite una alta modularización de las aplicaciones para separar o independizar funcionalidades o módulos transversales (denominadas aspectos).
  + Aspectos: Son funcionalidades encapsuladas que pueden utilizarse de forma transversal, aplicándose en determinados puntos de otros componentes (ejemplo: login, seguridad…).
  + Instrumentation: Proporciona soporte e instrumentos para trabajar con las clases cargadas la JVM (Máquina Virtual de Java).
  + Messaging: Proporciona soporte para la mensajería, implementando protocolos como JMS (API de Java que proporciona un protocolo para enviar, recibir y gestionar mensajes entre diferentes aplicaciones y componentes).
* Data Access: Permite optimizar el trabajo de acceso y manejo de datos. Se subdivide en:
  + JDBC: Permite acceder a datos (bases de datos) con menos código mediante clases Helper Classes.
  + ORM (Object Relational Maping): Permite usar la POO (Programación Orientada a Objetos), de forma que se pueden tratar las bases de datos como objetos. Proporciona soporte para el mapeo objeto-relación y se integra con otras tecnologías como JPA e Hibernate.
  + Transactions: Compuesto por:
    - OXM: Permite el mapeo de objetos Java y XML, convirtiendo atributos y estructuras de un objeto a un formato XML válido(y viceversa).
    - JMS: Java Messaging Service: Permite el manejo de mensajes Java.
* Web/MVC (Modelo Vista-Controlador): Módulo que permite el acceso web remoto, la programación distribuida de aplicaciones web y la integración con otras tecnologías. Se subdivide en:
  + Web: Proporciona funciones básicas para el desarrollo de aplicaciones web.
  + Web socket: Para establecer conexiones entre cliente y servidor.
  + Servlet: Optimiza el manejo de solicitudes HTTP y la generación de respuestas a dichas solicitudes.
  + Portlet: Proporcionan funcionalidades específicas dentro de una página web.
* Test: Facilita la integración de otras tecnologías para realizar pruebas sobre el software desarrollado:
  + Junit: Para ejecutar pruebas unitarias.
  + Mock: Con los Mock Objects se pueden simular los objetos y realizar pruebas sobre los mismos.

**¿Qué significa acoplamiento?**

Es el grado de dependencia o conexión que tienen los diferentes componentes o módulos de una aplicación. Un acoplamiento alto indica que los módulos están fuertemente conectados entre sí, mientras que un bajo acoplamiento (loose coupling) indica que los módulos trabajan de manera independiente. El bajo acoplamiento es deseable, puesto que facilita la realización de cambios sin afectar a todos los componentes.

**¿Si quiero enviar mensajería que tengo que usar?**

Con JMS (Java Messaing Service), API de Java que proporciona un protocolo para enviar, recibir y gestionar mensajes entre diferentes aplicaciones y componentes. En Spring, el módulo Messaging proporciona soporte para la mensajería implementando protocolos de JMS.

**¿Qué tipo de aplicaciones puede crear con Spring?**

Todo tipo de aplicaciones (aplicaciones y servicios web, aplicaciones de acceso a datos, de mensajería, de seguridad, en tiempo real…). Spring proporciona muchas funcionalidades y versatilidad, lo que permite el desarrollo de una gran variedad de aplicaciones (siempre empleando el lenguaje de programación Java)

.

### Tareas

**Montar un entorno de desarrollo para trabajar con Spring (IDE a usar Eclipse XXX, Tomcat 8, Maven, …). Describir los pasos a realizar (de forma general)**

***Nota: No usar STS hasta que no se indique en los cursos. La tarea es montar el entorno sobre un “Eclipse estándar” y añadir las librerías y servidores correspondientes.***

Para la instalación y configuración de un entorno de desarrollo listo para trabajar con Spring, se han realizado los siguientes pasos:

1. Descarga e instalación de Java (JDK, Java Development Kit y JRE, Java Runtime Enviroment). En este caso se ha instalado la última versión (JDK 22) disponible a través de la web oficial de Oracle. La instalación de JDK incluye la de JRE. Se sigue el asistente de instalación sin mayor personalización.
2. Descarga e instalación del servidor Java. En este caso se ha instalado Tomcat 10.1 a través de la web oficial de Apache Tomcat. Se sigue el asistente de instalación sin mayor personalización.
3. Descarga e instalación de un entorno de desarrollo (IDE). En este caso se ha instalado Eclipse JEE (para empresas) en su versión más reciente (2024-03 4.31.0). Se sigue el asistente de instalación, creando una carpeta que se almacenarán los proyectos desarrollados.
4. Añadir el servidor Java (Tomcat) en Eclipse. Para ello, desde Eclipse, en la pestaña “Servers” se puede añadir el servidor Tomcat indicando la versión instalada y la ruta de instalación del mismo. También hay que indicar la versión de Java que se utilizará (en este caso, la 22).
5. Descarga e importación de librerías Spring (.jar). Para ello, se busca la librería más reciente a través de internet. En este caso se ha descargado el paquete 6.1.5.jar.
   1. En Eclipse, en la perspectiva Java, crear un nuevo proyecto. Dentro del mismo, crear una carpeta en la que se debe copiar el/los archivos de la librería descargada.
   2. Hacer clic secundario sobre el nombre o raiz del proyecto, en el explorador de proyectos de Eclipse, y seleccionar Propiedades. En el menú que se abre, indicar “Java Buld Path” en el lateral izquierdo y la pestaña “Libraries”. Desde aquí, añadir los JARs indicando la ruta de la carpeta anterior en la que se han almacenado las librerías.
   3. Es posible que también se tenga que importar (correctamente) la librería de Java (el JRE/JDK 22 en este caso), indicando la ruta de su instalación.

### Dudas

Resueltas en la reunión (acoplamiento, inyección de dependencias).

## Qué es IoC e Inyección de dependencias

### Objetivos

* **Comprender los conceptos de Ioc e ID**
* **motivos por los cuales se usán**
* **ventajas e incovenientes…**
* **conceptos de "Contenedor", "Patrón", etc..**

### Recursos formativos

* (Recomendado) <https://www.youtube.com/watch?v=-Cs1HN6pEg4>

(pildorasInformáticas) (videos del 6 al 12)

* (complementario) <https://www.youtube.com/watch?v=jURvf96Qaq8>
* (complementario) <https://www.youtube.com/watch?v=UXeoZ6YA6LM&list=PLF6521742961719FC&index=3&t=0s>

### Posibles preguntas

* **¿Qué es IoC? ¿e ID?**

IoC (Inversión of Control) es un principio que consiste en invertir el flujo del programa de forma que la construcción y el manejo o gestión de objetos se externaliza a un agente externo, como puede ser el framework Spring. Por lo tanto, no es la propia aplicación la que tiene el control sobre la creación y manejo de los objetos, sino que realiza solicitudes al agente externo y éste le proporciona los objetos o funcionalidades. Por tanto, es el framework externo el que controla el flujo de ejecución del programa, lo que facilita la modularización (bajo acoplamiento y baja dependencia) del software y la flexibilidad en su desarrollo y configuración.

ID (Inyección de Dependencias) es un patrón o forma de diseñar el software que implica que las dependencias de un objeto respecto a otro no son creadas ni gestionadas por el propio objeto, sino por un agente externo (framework) que atiende a la llamada del objeto. Esta estrategia propicia la Inversión de Control (IoC).

* **¿Es lo mismo IoC que ID?**

Aunque tanto IoC como ID forman parte del mismo concepto, no son lo mismo. IoC es un principio de diseño de software según el cual se pretende invertir el control de flujo de un programa para que sea asumido por un agente externo en lugar de por el propio programa. La Inyección de Dependencias es la técnica que se emplea para cumplir con dicho principio. IoC es el concepto general o la filosofía, mientras que ID es la herramienta que implementa dicho concepto.

* **¿Qué es el principio Hollywood?**

Es un principio en el que se basa la Inversión de Control (IoC). Se explica con una analogía de un casting de Hollywood, en el que los aspirantes pretenden llamar, todos a la vez, al responsable del casting para saber el resultado de las pruebas. Para evitar esto, el responsable se anticipa diciéndoles “No nos llames, nosotros te llamaremos”, de manera que invierte el flujo de la comunicación y asume el control de esta. Por tanto, es un principio que establece que un programa no llame directamente a sus componentes u objetos, sino que delegue el control a un agente externo que quien proporcione mecanismos para crear o invocar objetos o componentes. Esto se traduce finalmente en un bajo acoplamiento de los componentes del software y una mayor flexibilidad en su diseño y desarrollo.

* **¿IoC/ID están dentro de los principios SOLID?**

No son principios SOLID como tal pero sí están relacionados con uno de ellos: Con el principio de Inversión de Dependencia. Este principio establece que los módulos o funcionalidades de alto nivel no deben depender de módulos de bajo nivel, promoviendo desacoplar componentes y favorecer la flexibilidad y extensión del software.

La Inversión de Control (IoC) es un principio que externaliza el control de la ejecución de un programa y la Inyección de Dependencias (DI) es la técnica que materializa este principio. Con lo cual, ambas contribuyen a desacoplar componentes de manera interna.

* **¿y qué son los SOLID?**

Son un conjunto de 5 principios de diseño de software que tienen la finalidad de simplificar el código, facilitando su modularidad y mantenimiento. Los principios son:

**S** – Principio de responsabilidad única (**S**ingle Responsability Principle): Una clase debe tener una única razón para cambiar, o lo que es lo mismo, una única responsabilidad dentro del sistema. De esta forma las clases se vuelven más independientes (bajo acoplamiento).

**O** – Principio de abierto/cerrado (**O**pen/Close Principle): Los elementos o entidades del software (clases, módulos, funciones) deben estar abiertas para su extensión pero cerradas para su modificación. El código debe diseñarse de forma que una extensión no suponga una modificación del código existente.

**L** – Principio de sustitución de Liskov (**L**iskov Substitution Principle): Los objetos de las subclases pueden sustituirse por objetos de sus superclases, y viceversa, sin afectar al funcionamiento del programa.

**I** – Principio de segregación de la interfaz (**I**nterface Segregation Principle): Las interfaces generales deben dividirse en interfaces más pequeñas y específicas, para que cada cliente implemente solo los métodos que necesite.

**D** – Principio de inversión de dependencia (Dependency Inversión Principle): Establece que los módulos o funcionalidades de alto nivel no deben depender de módulos de bajo nivel, promoviendo desacoplar componentes y favorecer la flexibilidad y extensión del software.

* **¿Qué siginifica acoplamiento?**

Es el grado de dependencia o conexión que tienen los diferentes componentes o módulos de una aplicación. Un acoplamiento alto indica que los módulos están fuertemente conectados entre sí, mientras que un bajo acoplamiento (loose coupling) indica que los módulos trabajan de manera independiente. El bajo acoplamiento es deseable, puesto que facilita la realización de cambios sin afectar a todos los componentes.

* **¿Con spring tenemos un alto o bajo acoplamiento?**

Bajo acoplamiento, como ya se ha explicado en la definición de IoC y DI. Spring permite la inyección de dependencias (a través de los beans), externalizando la creación de objetos de forma que no dependen de la clase que los solicita. El control lo asume Spring, por tanto (IoC). Ello genera una independencia entre clases y por ello un bajo acoplamiento.

* **¿Qué es un POJO?**

POJO es el acrónimo de Plain Old Java Object (Objeto Java Sencillo y Antiguo). Se trata de una clase simple que no implementa interfaces ni extiende otras clases (no hereda de otras clases). Contiene campos o atributos privados con métodos públicos de acceso (setters y getters).

### Tareas

NA

### Dudas

Me costaba comprender el concepto de IoC y lo que supone una Inyección de Dependencias, pero realizando los ejemplos prácticos se va aclarando.

## Ejemplo de IoC e Inyección en Java

### Objetivos

* **Comprender los conceptos de IoC y DI utilizando únicamente el lenguaje Java**

### Recursos formativos

* <https://www.youtube.com/playlist?list=PLG1t8jaLbxA8NfCZwlHcrXhB_Ih1WTCyS>

(video 4 y 16)

### Posibles preguntas

NA

### Tareas

Entregar un ejemplo operativo y funcional donde se explique el funcionamiento de IoC e ID usando únicamente clases java (sin usar ningún framework tipo Spring o EJB3.0)

**Dirección repositorio:** <https://github.com/Sanmohe/EjerciciosSpring.git>

Archivos de clase en SRC – paquete Leccion3tarea

### Dudas

* ¿?

## Ciclo de vida de los Beans

### Objetivos

* + **comprender el concepto de Bean.**
  + **cómo se pueden añadir propiedades y valores a los Bean mediante configuraciones adicionales (anotaciones).**
  + **trabajar con beans con ambas configuraciones**

### Recursos formativos

* (Recomendado) <https://www.youtube.com/watch?v=-Cs1HN6pEg4>

(pildorasInformáticas) (videos del 9 al 14)

* <https://www.youtube.com/playlist?list=PLG1t8jaLbxA8NfCZwlHcrXhB_Ih1WTCyS>

### Posibles preguntas

* **¿Qué es un Bean?**

Un bean es un componente u objeto que es creado y gestionado por el contenedor (en este caso, el contenedor de Spring), el cual puede ser inyectado en las clases que lo necesiten o lo soliciten (aplicando el principio de Inversión de Control, mediante el cual las clases de un programa no crean sus propios objetos sino que los solicitan al contenedor, es decir, solicitan el bean al contenedor). El uso y manejo de beans de esta forma materializa la inyección de dependencias en las clases.

* **¿Cuál es la diferencia entre crear un Bean por constructor y por un setter?**

Al crear un Bean por constructor, la inyección de dependencias se realiza de forma distinta al crear un Bean por un Setter:

* Al crearlo mediante un **constructor**, las dependencias se inyectan en el constructor en el momento de la creación del bean. De esta forma, las dependencias pasan a estar disponibles en el momento de instanciar el bean. Una vez construido el bean, sus dependencias permanecen “inmutables”. En XML, la sintaxis para crear el bean mediante constructor es:

<bean id=*"nuevoBean"* class=*"paquete.Clase"*>

<constructor-arg ref=*"idBeanDependencia"*></constructor-arg>

<!-- En REF se inyecta la dependencia de otro Bean mediante su ID -->

</bean>

* Al crearlo mediante un **Setter**, las dependencias se inyectan a través de los métodos Setter una vez se ha creado el bean. En este caso, las dependencias son opcionales y pueden ser modificadas, ofreciendo mayor flexibilidad. En XML, la sintaxis para crear el bean mediante Setter es:

<bean id=*"nuevoBean"* class=*"paquete.Clase "*>

<property name=*"nombreSetter"* ref=*"idBeanDependencia"*></property>

<!-- En NAME se introduce el nombre del setter (declarado en la clase en cuestión) sin la palabra "set" y con la primera letra minúscula.

En REF se inyecta la dependencia de otro Bean mediante su ID -->

</bean>

En definitiva, crear un Bean mediante constructor hace más sólida o rígida la inyección de dependencias (siempre se inyecta), mientras que crear un Bean mediante Setter ofrece opcionalidad y flexibilidad en la inyección de dependencias hacia las clases (la dependencia es una propiedad y no una cualidad intrínseca en el constructor).

* **¿Qué significa el concepto de convención sobre configuración? (pon ejemplos)**

Es un paradigma de desarrollo de software según el cual se establecen convenciones (criterios predefinidos) para la configuración de componentes de software, en lugar de que cada componente requiera una configuración específica para cada caso. Se trata de una especie de estandarización de configuraciones. La finalidad de este paradigma es reducir la cantidad de código repetitivo y el número de decisiones que tiene que tomar el programador, ya que este podrá seguir las convenciones por defecto para configurar ciertos componentes genéricos, centrándose solo en las partes únicas o esenciales de su aplicación.

Por ejemplo:

Para mejorar la conexión con bases de datos, si una clase se llama Usuario, la tabla correspondiente en base de datos debería llamarse Usuarios, para que las conexiones puedan realizarse automáticamente con una configuración predefinida basada en una convención por nombre. Si se cambiara el nombre de la tabla a Altas\_Servicio, habría que configurar explícitamente que la clase Usuario está conectada a dicha tabla.

Otro ejemplo:

En Spring se utilizan convenciones para la inyección de dependencias. Si existe una clase llamada *ServicioUsuario* y otra llamada *RepositorioUsuario*, Spring buscará automáticamente una implementación de *RepositorioUsuario* para inyectar en *ServicioUsuario*, basándose en convenciones como el nombre de la clase o los tipos de campos.

Un ejemplo más:

Se pueden seguir convenciones para la configuración de la estructura de directorios de un determinado programa o aplicación. Por ejemplo, al crear una web en HTML es común establecer un árbol de directorios del tipo css/, js/, la web principal se denomina index.html… Si se emplea algún framework de apoyo al diseño, automáticamente detectará estos elementos “por defecto”. Si cambiamos el nombre convencional de ficheros o directorios, o sus rutas, tendríamos que realizar una nueva configuración para que todo funcionara.

* **¿Cuáles son los distintos scopes (ámbitos) de un Spring Bean? ¿Cuál es el scope por defecto?**

Un scope es un patrón de diseño a seguir en el desarrollo de aplicaciones. Aunque Spring puede manejar diferentes scopes, principalmente se utilizan los siguientes:

* **Singleton**: Es el scope por defecto de Spring. Genera una única instancia (objeto) de un mismo Bean en todo el contenedor. Cualquier clase que solicite dicho Bean apuntará al mismo objeto. Es un patrón de diseño adecuado cuando se quiere tener acceso a un único dato global o recurso compartido (por ejemplo, un fichero abierto). Permite un mayor control sobre el acceso a las instancias, optimiza la herencia (al trabajar con menor cantidad de objetos), lo que conlleva a un espacio de nombres más reducido y un código más claro.
* **Prototye**: Es el concepto opuesto a Singleton. Permite que cada vez que una clase solicite el mismo Bean, se cree una nueva instancia (un nuevo objeto independiente). Realmente se crea un primer objeto o prototipo y el resto de solicitudes generan clonaciones de dicho prototipo, que pueden tener variaciones a posteriori. Es un enfoque útil cuando se necesita trabajar con varios objetos que tienen atributos repetidos o comunes. Permite crear y eliminar objetos en tiempo de ejecución y ahorra tiempo y recursos al clonar objetos en lugar de crearlos desde cero.

Otros scopes son:

* **Request**: Crea una instancia única de bean por cada solicitud HTTP. La instancia del bean está disponible solo durante la duración de una solicitud HTTP y se destruye al final de la solicitud.
* **Session**: Crea una instancia única de bean por cada sesión de usuario. La instancia del bean está disponible durante toda la duración de la sesión del usuario y se destruye cuando la sesión expira o se invalida.
* **Application**: Este scope solo está disponible solo para aplicaciones web que utilizan Spring MVC. Crea una instancia única de bean por toda la aplicación web. La instancia del bean está disponible durante toda la duración de la aplicación y se destruye cuando la aplicación se detiene.
* **WebSocket**: Este ámbito está disponible solo para aplicaciones web que utilizan WebSocket. Crea una instancia única de bean por cada sesión WebSocket. La instancia del bean está disponible durante toda la duración de la sesión WebSocket y se destruye cuando la sesión WebSocket se cierra.

### Tareas

NA

### Dudas

¿?

## Configuración por Anotación

### Objetivos

* **configurar Spring mediante anotaciones**
* **Conocer que existe la opción de configuración por XML**

### Recursos formativos

* (Recomendado) <https://www.youtube.com/watch?v=-Cs1HN6pEg4>

(pildorasInformáticas) (videos del 15 al 25)

* <https://www.youtube.com/playlist?list=PLG1t8jaLbxA8NfCZwlHcrXhB_Ih1WTCyS>
* [CONFIGURACIÓN DE SPRING: XML vs. ANOTACIONES vs. JAVACONFIG - YouTube](https://www.youtube.com/watch?v=yWDxgzDZzwo)
* lecturas: https://www.adictosaltrabajo.com/2013/10/30/tutorial-basico-spring-3/

### Posibles preguntas

* **¿en qué fichero se configura?**

Para que Spring utilice las anotaciones sin necesidad de implementar código de forma manual, en el **fichero configuración de beans (XML),** debe introducirse la siguiente etiqueta, dentro del elemento raiz <beans>:

<context:component-scan base-package=*"paqueteAescanear"*></context:component-scan>

También se puede realizar a través de un **fichero de configuración con lenguaje JAVA (.class)**, para lo que hay que crear una nueva clase y emplear las anotaciones:

@Configuration

@ComponentScan("paqueteAEscanear")

**public** **class** Config {

(…)

}

Con ello se exige a Spring que escanee todas las clases existentes dentro del “paqueteAescanear” en busca de anotaciones.

A su vez, en la **clase MAIN** hay que cargar el archivo de configuración:

//Fichero XML

ClassPathXmlApplicationContext contexto = new ClassPathXmlApplicationContext("applicationContext.xml");

//Fiechero JAVA

AnnotationConfigApplicationContext contexto =

**new** AnnotationConfigApplicationContext(Config.**class**);

* **¿Cómo se identifican la clases/bean que estarán bajo Spring? ¿Qué significa la propiedad autowiring?**

Las clases que están bajo el control de Spring se identifican con las Spring Annotations (anotaciones de Spring). La anotación base de Spring es:

* **@Component:** Se aplica sobre una clase para crear su bean correspondiente en el contenedor de Spring. De esta forma, será Spring quien administre dicha clase (IoC).

La anotación debe introducirse justo encima del código sobre el que aplica, ya sea una clase, un constructor, un método o un campo de clase. Cada anotación tiene una función diferente. Otras anotaciones principales son:

* **@Autowired:** Para inyectar la dependencia en un constructor, método SETTER o cualquier otro método que llame a la interfaz de la dependencia o en un campo de clase que apunta a la dependencia. Hace que Spring escanee todo el proyecto en busca de clases que implementen la interfaz de la dependencia. No es necesaria esta anotación si solo hay un constructor en el bean que se pide a Spring.
* **@Qualifier(“*nombreBean*”)**: Para especificar el ID de un bean. Se usa cuando existen varias clases que implementen la interfaz de la dependencia (varios bean que pueden inyectar la dependencia). La anotación se coloca debajo de @Autowired.
* **@Scope(“*nombrePatron*”)**:: Permite establecer el patrón de diseño (Singleton – Prototype) sin entrar en el código del HTML. El nombre de patron se introduce en minúsculas (singleton / prototype).
* **@PostConstruct**: Indica la parte de código o instrucciones que se ejecutarán antes de la construcción del bean (misma función que el método Init).
* **@PostDestroy:** Indica la parte de código o instrucciones que se ejecutarán después de la ejecución del bean y del apagado del contenedor (misma función que el método Destroy).
* **@Configuration:** Permite configurar el contenedor Spring con código Java emplear un archivo de configuración XML. En este caso se genera un archivo de configuración .CLASS.
* **@ComponentScan(“*paqueteAescanear*”):** Se coloca a continuación de la anotación @Configuration para indicar el paquete que Spring debe escanear en busca de anotaciones.
* **@Bean:** Permite la creación o definición de un BEAN dentro del fichero de configuración .CLASS mediante código Java.
* **@PropertySource(“*classpath:archivoPropiedades*”):** Permite especificar la ruta del fichero externo de propiedades dentro del fichero de configuración .CLASS.
* **@Value(“${*nombrePropiedad*}”):** Permite inyectar el valor de las propiedades desde un fichero externo.

La propiedad **autowiring** es una característica de Spring que permite que sea el propio contenedor de Spring el que resuelva automáticamente las dependencias que debe inyectar cuando una clase solicita un determinado bean. Spring escanea todo el proyecto en busca de clases que implementen las interfaces de la dependencia. De esta forma, no es necesaria la introducción de código de forma manual en el archivo de configuración de beans.

* **¿Cómo funciona (el autowiring)? (enumerar los distintos tipos)**

En primer lugar, hay que configurar el archivo XML o el archivo de configuración .CLASS (Java) para establecer que Spring escanee todas las clases existentes dentro del paquete en busca de anotaciones.

Autowiring tiene cuatro tipos para referenciar o enlazar objetos:

* **byName**, tiene que coincidir el nombre o id con la propiedad.
* **byType**, tiene que coincidir el mismo tipo de objeto con la propiedad a la que estamos inyectando.
* **constructor**, intenta asignar a los parámetros del constructor las beans que encajen en tipo.
* **autodetect**, intenta aplicar el tipo por constructor, si falla, lo intenta por el tipo byType.
* **¿Qué anotaciones tenemos para el autowiring?**
* **@Autowired:**  Es la anotación más común. Como se ha explicado antes, puede aplicarse a constructores, métodos setters y campos de clase, para que Spring inyecte automáticamente la dependencia.
* **@Qualifier(“nombreBean):** Se suele combinar con @Autowired para especificar el ID o nombre del Bean que debe aplicarse cuando existen varios beans o clases que implementen la interfaz de la dependencia.
* **@Resource(name = “nombreRepositorio”):** Inyecta la dependencia por nombre en lugar de por tipo. Es decir, Spring buscará el del bean independientemente del tipo o clase. Tiene una función similar a @Qualifier.
* **@Inject:** Similar a @Autowired. Puede ser utilizada en otros frameworks que implementen JSR-330 (una especificación o estándar de Java que define un conjunto de anotaciones para la inyección de dependencias).
* **¿Qué diferencias tienen?**

**@Autowired** y **@Inject** tienen la misma funcionalidad: inyectar dependencias. La diferencia es que @Autowired es una anotación específica del framework Spring mientras que @Inject forma parte de un estándar de anotaciones de Java, por lo que tiene mayor compatibilidad con otros frameworks.

De igual forma **@Qualifier** y **@Resource** indican el ID o el nombre del Bean a inyectar en el caso de que existan varios beans o clases que implementen la interfaz de la dependencia. La diferencia, igualmente, es que @Qualifier es propia de Spring y se usa en conjunto con @Autowiring, mientras que @Resource forma parte del estándar de anotaciones de Java, por lo que se puede utilizar en otros frameworks o tecnologías que implementen dicho estándar.

* **¿Puedes indicar otro tipo de anotaciones de spring? (respuesta esperada: Component, Controller, Repository,Service)**

Otras anotaciones muy utilizadas y propias de Spring son (@Component se explica en la pregunta 1):

* **@Service**: Se usa para marcar clases de servicio (clases que encapsula una determinada funcionalidad en una aplicación).
* **@Repository**: Se usa para marcar clases de repositorio (clases que proporcionan métodos para operaciones de lectura, escritura, actualización y eliminación de datos en una base de datos u otro tipo de almacenamiento).
* **@Controller**: Se usa para marcar clases controladoras en una aplicación web. Las clases controladoras manejan las solicitudes de un cliente, las procesan y devuelven una respuesta adecuada.
* **@RestController**: Similar a @Controller, pero específicamente diseñado para construir servicios RESTful (un conjunto de principios de diseño para que las aplicaciones web puedan ser escalables, flexibles y fáciles de entender).

### Tareas

### Dudas

¿?

## Ejemplo de IoC e Inyección en Spring

### Objetivos

* **comprender los conceptos de IoC e ID utilizando únicamente el framework Spring**

### Recursos formativos

* lecturas: https://www.adictosaltrabajo.com/2013/07/25/spring-container-inyeccion-dependencias/

### Posibles preguntas

NA

### Tareas

* Realiza un ejercicio usando la versión actual de Spring [6.1.2](https://mvnrepository.com/artifact/org.springframework/spring-core/6.1.2) (sin usar Spring Boot) parecido a los vistos en los recursos formativos donde se ve el uso de las distintas anotaciones, configuraciones… de Spring (se valorará la originalidad y que el ejercicio no sea igual al del vídeo).

**Dirección repositorio**: <https://github.com/Sanmohe/EjerciciosSpring.git>

Archivos de clase en SRC – paquete Leccion6tareaFinal

### Dudas

¿?