## **1.1 ¿Qué es Spring?**

Spring es un framework de código abierto para el desarrollo de aplicaciones (empresariales) Java. Su origen está en el trabajo de [Rod Johnson](https://www.linkedin.com/in/johnsonroda), que trabajaba como consultor externo, y que plasmó en el libro [Expert One-to-One J2EE Design and Development](http://www.wrox.com/WileyCDA/WroxTitle/Expert-One-on-One-J2EE-Design-and-Development.productCd-0764543857.html)  (Octubre, 2002). Según Johnson, el modelo de los [Enterprise Java Beans](https://es.wikipedia.org/wiki/Enterprise_JavaBeans) era excesivamente tedioso y pesado para muchas aplicaciones desarrolladas hasta el momento. Por ello, condensó su experiencia y sus buenas prácticas en un conjunto de clases que fueron el origen del framework.

Algunas de las críticas de Johnson al uso de EJBs son (ver [J2EE development without EJB (2004)](http://www.wrox.com/WileyCDA/WroxTitle/Expert-One-on-One-J2EE-Development-without-EJB.productCd-0764558315.html)):

* Complejidad (descriptores, implementaciones de interfaces, …) y baja productividad del programador.
* Modelo remoto solamente basado en RMI.
* Muchas aplicaciones no necesitan componentes remotos.
* Difíciles de depurar (volver a hacer deployment y volver a arrancar).
* Mapeo O/R basado en entity beans limitado (por ejemplo, no existe la herencia)
* …

Spring popularizó desde su inicio ideas como la inyección de dependencias (que aprenderemos más adelante), el uso de POJOs (Plain old Java Object) como objetos de negocio, etc… que suponían un cambio radical con respecto al estándar; de esta forma, las aplicaciones podían ser más ligeras, y posibilitó que un framework que estaba inicialmente ideado para la capa de negocio se convirtiera en un stack de tecnologías para todas las capas de la aplicación.

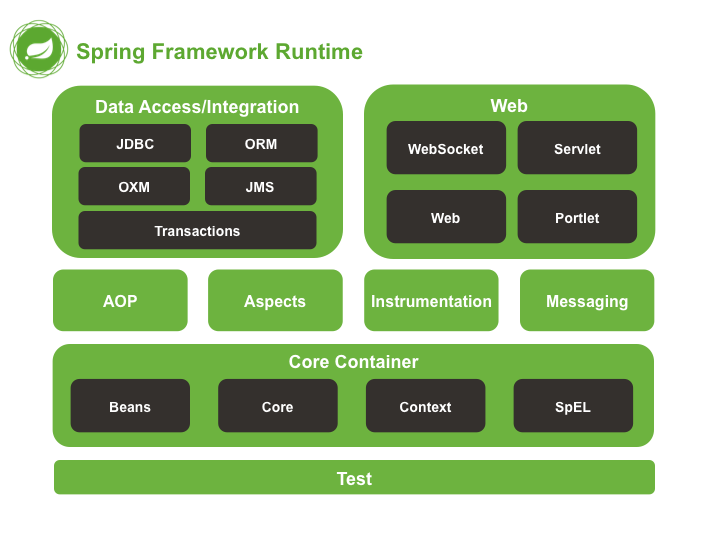
Spring está basado en los siguientes principios:

* El buen diseño es más importante que la tecnología subyacente.
* Los Java Beans ligados de una manera más libre entre interfaces es un buen modelo.
* El código debe ser fácil de probar.

Desde hace tiempo, estas ideas tan innovadoras popularizadas por Spring se han ido incorporando poco a poco al estándar, de forma que Spring y JavaEE han convergido mucho. Aun así, Spring ha conseguido crear una comunidad de desarrolladores en torno a sus diferentes tecnologías, siendo una alternativa necesaria de conocer, y muy utilizada en el entorno empresarial. En la actualidad, las aportaciones más novedosas de Spring se centran en campos de BigData, NoSQL, HTML5, Mobile, aplicaciones sociales, seguridad…

A día de hoy, una de las principales diferencias que podemos encontrar entre JavaEE y Spring es la posibilidad de usar un servidor web más convencional (estilo Tomcat) para desplegar la aplicación (con Spring Boot aprenderemos que nos podemos olvidar incluso de esta parte). JavaEE requiere el uso de un servidor de aplicaciones que, normalmente, requerirá de mayor conocimiento para su configuración y despliegue, y necesitará de unos recursos más potentes para correr cualquier aplicación.

## **1.2 Módulos de Spring**

En la siguiente imagen, podemos ver las diferentes tecnologías (en terminología Spring, proyectos) que conforman a día de hoy Spring:  


Algunos de los módulos más importantes de Spring

* **Core**: La parte fundamental de este framework es el módulo Core, y los adyacentes Bean y Context. Proveen toda la funcionalidad para la inyección de dependencias, permitiéndole administrar la funcionalidad del contenedor de Beans (trabajaremos sobre estos conceptos de forma más amplia en las próximas lecciones). Además, también proveen de los servicios Enteprise como JDNI, EJB, …
* **AOP**: Se trata de un módulo que nos permitirá utilizar el paradigma de Programación Orientada a Aspectos (Aspect Oriented Programming). Este paradigma permite mejorar la modularización y separar las responsabilidades. De esta forma, podemos separar las funcionalidades comunes, que se utilizan transversalmente a lo largo de toda la aplicación, de aquellas que son propias de cada módulo.
* **Data**: Se trata de un gran módulo, formado por múltiples submódulos, y que nos permite simplificar el acceso y persistencia de datos. Spring Data nos proporciona soporte para usar base de datos relacionales (JDBC), ORMs (como por ejemplo JPA, Hibernate, …) e incluso modelos de persistencia NoSQL (como por ejemplo, MongoDB).
* **Web**: Este módulo nos permitirá implementar el patrón Modelo-Vista-Controlador (MVC) de una manera sencilla y limpia, haciendo uso de forma transparente también de otros patrones de diseño, como FrontController. De esta forma, podemos separar limpiamente la lógica de negocio de la presentación de los datos y el acceso a los mismos. Además de aplicaciones que implican el uso de vistas y formularios, también podremos crear servicios web (por ejemplo, al estilo REST) de una forma sencilla y rápida.

La modularidad de Spring nos permitirá la posibilidad de usar solo algunas de las partes del framework, y poder combinar esta con otros frameworks diferentes:

* Un proyecto que implemente MVC mediante el uso de Struts podría incorporar el contenedor de IoC mediante el uso de Spring (sin necesidad de utilizar Spring Web MVC).
* Una aplicación web desarrollada con Spring Web MVC podría implementar su capa de datos mediante el uso de Hibernate (sin hacer uso de Spring Data).

## **1.3 Versiones de Spring**

La primera versión de Spring se publicó en marzo de 2004. Actualmente, la versión más estable en la actualidad de Spring es la 4.3.2, si bien ya podemos encontrar en fases preliminares la versión 5.0 (que nos permitirá trabajar con JDK 9).

| **VERSIÓN** | **DESCRIPCIÓN** |
| --- | --- |
| 1.0 | Primera versión de Spring |
| 2.0 | Espacios de nombres XML, soporte para AspectJ |
| 2.5 | Configuración a través de anotaciones |
| 3.0 | Actualización a Java5+, Configuración a través de Java (@Configuration) |

A continuación, podemos ver algunos de los elementos que se han incorporado en las diferentes versiones 4.X

| **VERSIÓN** | **CAMBIOS INTRODUCIDOS** |
| --- | --- |
| 4.0 | * Mejora de la documentación. * Eliminación de código deprecado. * Adaptación al uso de Java 8 * Basado en Java EE 6 (JPA 2.0, Servlet 3.0), con soporte para Java EE 7 (JPA 2.1, JTA 1.2, …) * Mejoras en el contenedor de inversión de control * Mejoras en el módulo web para la creación de servicios REST (@RestController, AsyncRestTemplate) |
| 4.1 | * Mejoras en las prestaciones de Java Message Services. * Soporte para anotaciones JCache (JSR-107) * Mejoras varias en el módulo web * Mejoras en el módulo de WebSocket. |
| 4.2 | * Varias mejoras en el módulo core (sobre todo a nivel de configuración a través de Java). * Mejoras en el módulo de acceso a datos (como por ejemplo, soporte para Hibernate 5.0). * Mejoras en JMS * Mejoras en el módulo web (integración de OkHTTP, soporte para CORS, anotaciones propias el mapeo, con @RequestMapping como metaanotación, …) |
| 4.3 | * Mejoras en el módulo core: métodos default de Java 8, mayor información en las excepciones, soporte de inyección por constructor para @Configuration… * Mejoras en el módulo web: anotaciones compuestas para @RequestMapping: @GetMapping, @PostMapping, @PutMapping, @DeleteMapping, y @PatchMapping; nuevas anotaciones compuestas para los ámbitos web: @RequestScope, @SessionScope, @ApplicationScope; otra serie de nuevas anotaciones… * Mejoras en los módulos de WebSocket y Testing |

## **2.1 Spring Tool Suite: nuestro entorno de trabajo**

STS (Spring Tool Suite), es un entorno de desarrollo basado en Eclipse que está preparado para el desarrollo de aplicaciones Spring. Proporciona un entorno ready-to-use para implementar, depurar, ejecutar y desplegar aplicaciones Spring, incluyendo integraciones con Git, Maven, AspectJ, así como todas las funcionalidades de las últimas versiones de eclipse.

La última versión disponible la podemos encontrar en la propia web de spring (<https://spring.io/tools/sts>). Si pinchamos sobre el enlace, podemos acceder a la descarga del entorno, que viene empaquetado como un fichero zip. Dentro del mismo, podemos encontrar una carpeta con todo el software, y el ejecutable, llamado STS.exe .

Sin embargo, en los primeros compases del curso, trabajaremos con Spring al estilo Maven, lo que nos proporcionará algunas ventajas:

* Podremos trabajar con proyectos legacy (heredados), que estén configurados como proyectos Maven.
* Nos permitirá implementar proyectos sin el uso de la herramienta específica STS, y así poder utilizar cualquier otro IDE de uso general (el propio Eclipse, Netbeans, JDeveloper…).

## **2.2 Una palabra más sobre Maven**

A día de hoy encontramos a muchos desarrolladores de Spring que utilizan Eclipse, y que comienzan los desarrollos de Spring como proyectos Maven. Maven nos permite gestionar las dependencias a través de un fichero de configuración, pom.xml . A continuación podemos encontrar las dependencias Maven más usuales (y para la versión actual) para el trabajo con Spring.

| **DEPENDENCIA** | **¿PARA QUÉ SIRVE?** |
| --- | --- |
| <dependency><groupId>org.springframework</groupId><artifactId>spring-core</artifactId><version>4.3.2.RELEASE</version></dependency> | Esta es una de las dependencias más básicas. Si queremos construir un proyecto básico con Spring, que nos permita usar el IoC container y definir nuestros propios beans, esta es la dependencia que debemos usar. |
| <dependency><groupId>org.springframework</groupId><artifactId>spring-webmvc</artifactId><version>4.3.2.RELEASE</version></dependency> | Esta dependencia nos permitirá trabajar con todos los componentes para implementar aplicaciones web siguiendo el patrón Modelo-Vista-Controlador |
| <dependency><groupId>org.springframework</groupId><artifactId>spring-aop</artifactId><version>4.3.2.RELEASE</version></dependency> | Esta dependencia sirve para poder utilizar la programación orientada a aspectos (AOP). |
| <dependency><groupId>org.springframework.data</groupId><artifactId>spring-data-jpa</artifactId></dependency> | Esta dependencia, algo más específica, nos permitirá trabajar con el submódulo de Spring Data para JPA |

A lo largo del curso trabajaremos con algunas dependencias más, que no son propias de Spring, pero sí de otros elementos que necesitaremos, como la interfaz de usuario, el proveedor de ORM (Object Relational Mapping ), la base de datos…:

| **DEPENDENCIA** | **¿PARA QUÉ SIRVE?** |
| --- | --- |
| <dependency><groupId>com.h2database</groupId><artifactId>h2</artifactId></dependency> | Base de datos relacional embebible en un proyecto, implementada íntegramente en Java. Es muy útil para la fase de desarrollo o para sistemas que no necesitan un sistema gestor de base de datos profesional de alto rendimiento. |
| <dependency><groupId>org.hibernate</groupId><artifactId>hibernate-entitymanager</artifactId></dependency> | Hibernate es uno de los motores de persistencia más utilizado a día de hoy en proyectos sofware. |
| <dependency><groupId>log4j</groupId><artifactId>log4j</artifactId><version>1.2.14</version></dependency> | Log 4 Java es uno de los múltiples sistemas de logging disponibles para Spring, que nos permitirá configurar rápidamente este aspecto de nuestros proyectos. |

Durante todas las lecciones, se proporcionarán las dependencias necesarias o el fichero pom.xmlcompleto necesarios para poder realizar los ejemplos.

## **3.1 ¿Qué es eso de la inversión del control?**

Aunque la plataforma Java provee de una gran cantidad de clases y funcionalidades para el desarrollo de aplicaciones, carece de los medios para organizar los bloques básicos en un todo coherente, dejando esta tarea para los arquitectos y desarrolladores. Si bien se pueden utilizar algunos patrones de diseño como Factory , Abstract Factory , Builder , Decorator , and Service Locator para componer las diferentes clases y objetos de la aplicación, estos patrones son simplemente eso: buenas prácticas .

Una forma sencilla de explicar el concepto de IoC ( Inversion of control ) es el **Principio de Hollywood**:

No nos llames; nosotros te llamaremos

La inversión de control es una forma de programar en la que el flujo de ejecución de un programa se invierte respecto a los métodos de programación tradicionales, en los que la interacción se expresa de forma imperativa realizando llamadas a procedimientos o funciones. De esta forma, para poder implementar la IoC , necesitamos un agente externo, normalmente llamado contenedor , que se encargará de realizar las conexiones necesarias entre las clases dependientes. Este agente externo será el encargado de controlar el flujo de la aplicación. De esa forma, podremos realiza la implementación de un código que esté débilmente acoplada.

Para entender este cambio, vamos a utilizar un ejemplo con pseudocódigo (para no enmarañar el concepto con mucho código fuente innecesario). Imaginemos que estamos escribiendo un programa que tomar alguna información del usuario y que estamos usando la línea de comandos. Deberíamos hacer algo así:

escribir "¿Cuál es tu nombre?"

nombre <- Introducir nombre por teclado

procesar\_nombre(nombre)

escribir "¿Cuál es tu pregunta?"

pregunta <- Introducir pregunta por teclado

procesa\_pregunta(pregunta)

A lo largo de la interacción existente en el programa, el código es el que decide el flujo: decide cuando preguntar, cuando leer las respuestas así como cuando procesarlas.

Supongamos ahora que queremos adaptar nuestro programa a un sistema gráfico de ventanas. Podríamos tener un código parecido al siguiente (se trata de otra simplificación en pseudocódigo, ya que este código implementado en Java necesitaría de bastantes líneas más):

importar ventanas, botones, campos\_texto

ventana\_principal <- crear nueva Ventana.

etiqueta\_nombre <- crear nueva etiqueta "¿Cuál es tu nombre?"

añadir a ventana\_principal << etiqueta\_nombre

nombre <- crear nuevo campo\_texto

añadir a ventana\_principal << nombre

en\_caso\_de\_evento('dejar foco') sobre nombre, procesar\_nombre(nombre)

etiqueta\_pregunta <- crear nueva etiqueta "¿Cuál es tu pregunta?"

añadir a ventana\_principal << etiqueta\_pregunta

pregunta <- crear nuevo campo\_texto

añadir a ventana\_principal << pregunta

en\_caso\_de\_evento('dejar foco') sobre pregunta, procesar\_pregunta(pregunta)

Ventanas.buclePrincipal

Hay una gran diferencia entre el flujo del primer programa y de este segundo, en concreto, en las llamadas a las funciones para procesar el nombre y la pregunta. En este segundo código, no se tiene control sobre cuando se llamarán a dichas funciones, sino que se le ha entregado al control al sistema de ventanas. El decidirá cuándo llamar a estas funciones, basándose en la vinculación que hemos hecho ( en\_caso\_de\_evento(...) ). El control se ha invertido (en lugar de invocar yo a una clase, es el framework el que me invoca a mí). Este ejemplo es de [Martin Fowler](http://martinfowler.com/), una de las personas que acuñaron el concepto de Inversión de Control.

La inversión de control es un término genérico que puede implementarse de diferentes maneras, si bien la implementación realizada por Spring es la de **inyección de dependencias**, que trabajaremos en la siguiente lección.

## **4.1 Inversión de control mediante la inyección de dependencias .**

La dependencia de un objeto respecto a otro ocurre cuando el primero necesita del segundo para completar alguna de sus tareas.

Un caso típico es el de un coche, que depende del motor para, por ejemplo, poder avanzar.

public class Motor

{

//...

public void acelerar() {

//...

}

public int getRevoluciones() {

return currentRPM;

}

//...

}

public class Vehiculo

{

private Motor m;

public Vehiculo() {

m = new Motor();

}

public int getRevolucionesMotor() {

return m.getRevoluciones();

}

}

Como podemos comprobar, la dependencia entre las clases Vehiculo y Motor queda patente dado que una instancia de la primera alberga dentro una instancia de la segunda.

Sin embargo, podemos comprobar que el acoplamiento existente en el código es alto. El motor está fuertemente ligado al vehículo, de forma que esta relación es poco flexible. Si quisiéramos realizar cualquier tipo de modificación en la clase Motor , esto supondría un alto impacto en la claseVehiculo (por ejemplo, si quisiéramos hacer una concreción en MotorDiesel oMotorGasolina ).

Como primer paso para desacoplar el motor del vehículo, podríamos hacer que la clase Vehiculodeje de encargarse de instanciar el objeto Motor , pasándoselo como parámetro al constructor. De esta forma, la clase Vehiculo quedaría de la siguiente manera:

public class Vehiculo

{

private Motor m;

public Vehiculo(Motor motorVehiculo) {

m = motorVehiculo;

}

public int getRevolucionesMotor() {

return m.getRevoluciones();

}

}

El **constructor de vehículo se encarga de inyectar la dependencia**dentro del objeto, eliminando esta responsabilidad de la propia clase. De esa forma, hemos dado un paso para desacoplar ambos objetos.

El siguiente paso que podríamos dar en aras de continuar con el desacoplamiento de ambos objetos es el uso de interfaces.

public interface IMotor

{

public void acelerar();

public int getRevoluciones();

}

public class MotorGasolina {

public void acelerar() {

realizarAdmision();

realizarCompresion();

realizarExplosion(); //Propio de los motores de gasolina

realizarEscape();

}

public int getRevoluciones() {

return currentRPM;

}

//...

}

public class MotorDiesel {

public void acelerar() {

realizarAdmision();

realizarCompresion();

realizarCombustion(); //Propio de los motores diesel

realizarEscape();

}

public int getRevoluciones() {

return currentRPM;

}

//...

}

public class Vehiculo

{

private IMotor m;

public Vehiculo(IMotor motorVehiculo) {

m = motorVehiculo;

}

public int getRevolucionesMotor() {

return m.getRevoluciones();

}

}

Como podemos observar, la clase Vehiculo ya no está acoplada a la clase Motor , sino que bastará con un objeto que implemente la interfaz IMotor , como lo son MotorDiesel oMotorGasolina .

Por último, nos faltaría ver como se realizaría la inyección de dependencias propiamente dicha. Veamos la siguiente clase, Main :

// todos los import necesarios

public class Main {

public static void main(String[] args) {

/\*Al proporcionar un objeto que implementa la interfaz IMotor

como parámetro del constructor de Vehiculo, estamos inyectando

la dependencia.\*/

Vehiculo cocheDiesel = new Vehiculo(new MotorDiesel());

/\*En este caso, realizamos otra inyección de una dependencia,

pero con una instancia de MotorGasolina.\*/

Vehiculo cocheGasolina = new Vehiculo(new MotorGasolina());

//...

}

}

## **4.2 Diferentes formas de inyección de dependencias**

Como hemos visto en el ejemplo anterior, una de las formas de realizar la inyección de dependencias es mediante el uso de un constructor, pero no es la única. De hecho, a nivel formal podemos decir que existen algunas más:

* Inyección por constructor
* Inyección por método
* Inyección por propiedad (esta la encontramos en otros lenguajes de programación, con C#).

En el ejemplo anterior hemos podido apreciar que, para la **inyección por constructor**, la clase que tiene la dependencia sobre otra debe tener, al menor, un constructor, donde reciba un parámetro de ese tipo.

public class Vehiculo

{

private IMotor m;

public Vehiculo(IMotor motorVehiculo) {

m = motorVehiculo;

}

//...

}

En el caso de que quisiéramos implementar esta inyección de dependencia sin el uso de constructores con parámetros, podríamos hacer uso de la **inyección por método**(normalmente, uno de los métodos apodados como setter ). Veamos el siguiente código, modificado para realizar la inyección de dependencia de esta forma:

public class Vehiculo

{

private IMotor m;

public Vehiculo() {}

public void setMotor(IMotor motor) {

this.m = motor;

}

//...

}

// todos los import necesarios

public class Main {

public static void main(String[] args) {

/\* En este caso, inyectamos la dependencia

a través de un método\*/

Vehiculo cocheDiesel = new Vehiculo();

cocheDiesel.setMotor(new MotorDiesel());

Vehiculo cocheGasolina = new Vehiculo();

cocheGasolina.setMotor(new MotorGasolina());

//...

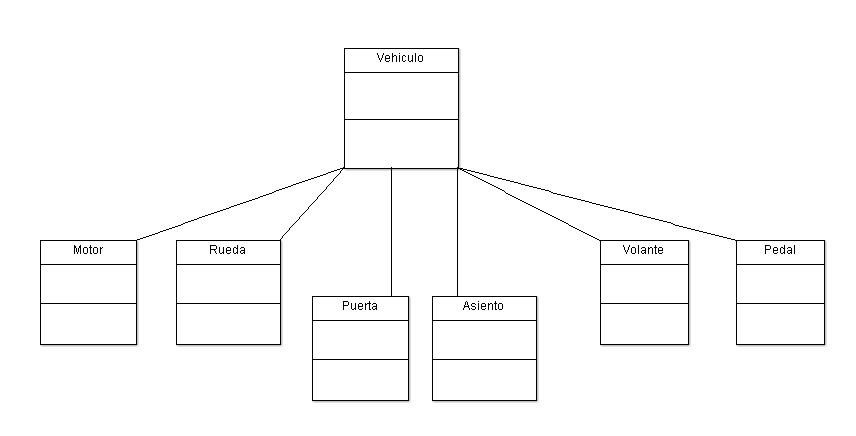
}

}

## **4.3 Inyección de dependencias con Spring**

Hasta ahora, hemos visto la conocida como inyección de dependencias para pobres , en la que solamente hemos utilizado elementos de Java SE para realizar dicha inyección.

Antes de continuar, hagámonos la siguiente pregunta con respecto al ejemplo: ¿qué otras dependencias tiene un vehículo?

  
Diagrama de clases simplificado para un Vehículo

Como podemos observar en el anterior diagrama de clases UML, la clase Vehiculo tiene una gran cantidad de dependencias. ¿Quién se hará cargo de todas ellas? La respuesta es **Spring IoC Container**, en el cual trabajaremos en la siguiente sección.

## **5.1 El contenedor de IoC de Spring**

Los paquetes org.springframework.beans y org.springframework.context son la base para el contenedor de IoC de Spring. El interface BeanFactory nos proporciona un mecanismo avanzado de configuración para gestionar cualquier tipo de objeto. ApplicationContext es un subinterface de BeanFactory , que añade una integración con Spring AOP, manejo de recursos, internacionalización, publicación de eventos y otros elementos específicos para su uso con aplicaciones web.

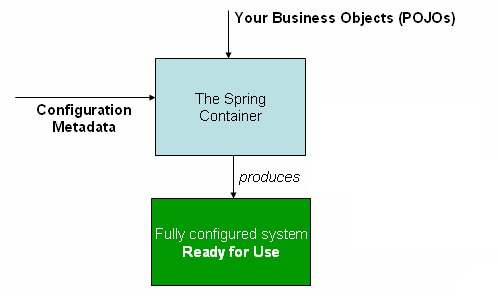
### **5.1.1 Pero, ¿qué es eso de Bean ?**

En Spring, los objetos que forman el nucleo de nuestra aplicación y que son gestionados por el contenedor de IoC se llaman beans . Un bean es un objeto que será instanciado, ensamblado y en definitiva gestionado por el IoC container . Dicho de otro modo, un bean no será más que uno de los muchos objetos que conformen nuestra aplicación. Los Beans, y las dependencias entre ellos, son reflejados en el fichero de configuración de metadatos usado por el contenedor.

### **5.1.2 Elementos del contenedor de IoC**

La interface org.springframework.context.ApplicationContext representa el contenedor de IoC y es responsable de instanciar, configurar y ensamblar los ya mencionados beans. El contenedor toma toda esta información leyendo los metadatos de configuración. Estos se pueden representar en XML, mediante anotaciones Java o incluso mediante código Java. Estos metadatos nos permitirán expresar los objetos que componen nuestra aplicación así como las interdependencias existentes entre ellos.

En la siguiente imagen, podemos ver, a alto nivel, como trabaja Spring. Nuestras clases son combinadas con los metadatos de configuración; por lo que después de crear e inicializar nuestroApplicationContext , tendremos un sistema totalmente configurado y listo para ser ejecutado.



El contenedor de inversión de control de Spring.

### **5.1.3 Metadatos de configuración**

Como hemos visto en la figura anterior, el contenedor de IoC necesita unos metadatos de configuración. Estos metadatos representan la forma en que serán instanciados, configurados y ensamblados los objetos de nuestra aplicación.

Tradicionalmente, esta configuración se ha proporcionado mediante XML, si bien podríamos usar otras alternativas:

* La configuración mediante anotaciones: fue introducida a partir de la versión 2.5 de Spring (la usaremos de forma combinada con XML).
* La configuración mediante código Java: comenzó con la versión 3.0 de Java.

A continuación podemos ver cual sería la estructura básica de un fichero XML de configuración:

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

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

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="..." class="...">

<!-- collaborators and configuration for this bean go here -->

</bean>

<bean id="..." class="...">

<!-- collaborators and configuration for this bean go here -->

</bean>

<!-- more bean definitions go here -->

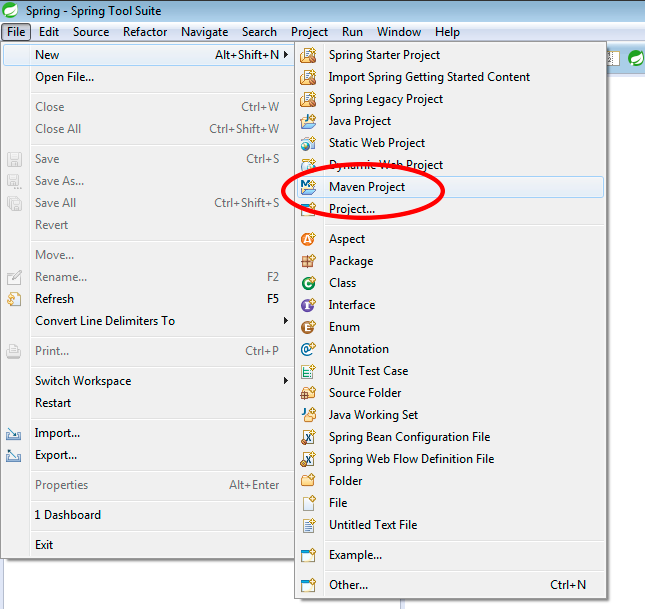
</beans>

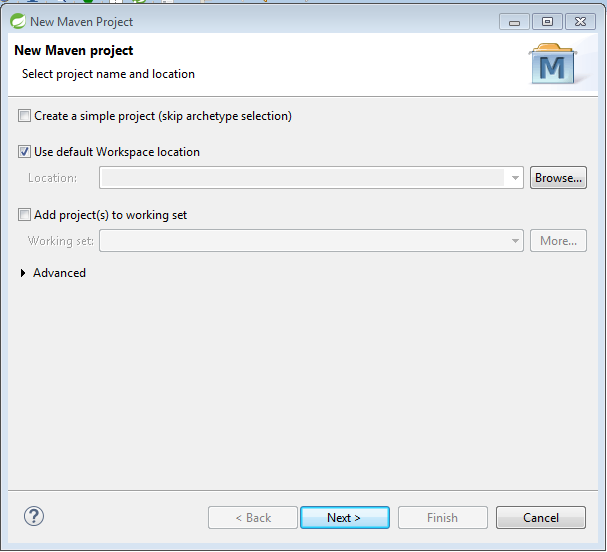
## **5.2 HolaMundo con Spring (con configuración XML)**

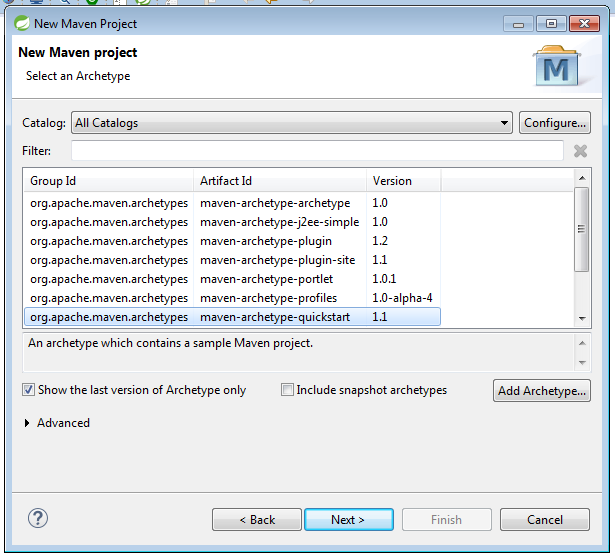
Hasta ahora hemos introducido alguno de los conceptos nucleares sobre el Spring IoC container . Sin embargo, cualquier tecnología o lenguaje de programación que se precie debe tener su**HolaMundo**. Vamos a crear el nuestro paso a paso.

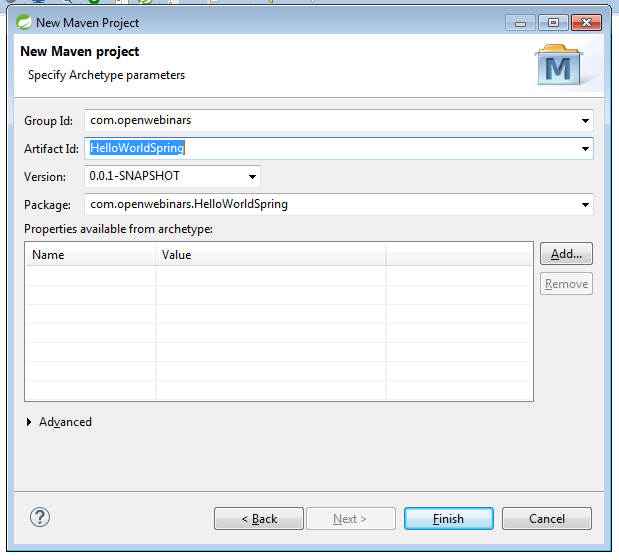
### **Paso 1: Crear un proyecto Maven.**

Ejecutamos nuestro entorno de desarrollo, Spring Tool Suite , y procedemos a la creación de un nuevo proyecto de tipo Maven. En este punto, seleccionamos el arquetipo maven-archetype-quickstart . Como metadatos del proyecto, escribimos un Group Id , que en nuestro caso serácom.openwebinars ; y por último como Artifact Id escribimos el nombre del proyecto, que seráHelloWorldSpring .



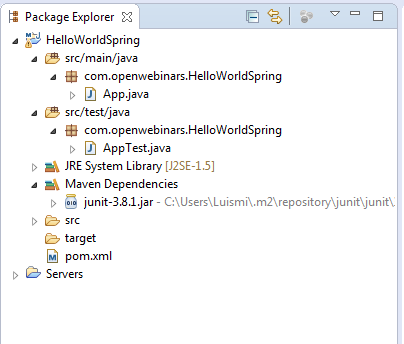






Creación de un proyecto de tipo Maven, selección de arquetipo e introducción de los metadatos básicos del proyecto.

Si seguimos estos pasos, obtendremos un proyecto que tendrá el siguiente aspecto en el explorador del IDE:



Estructura del proyecto recién creado.

Procedemos, por último, a eliminar los ficheros App.java y AppTest.java , ya que crearemos nuestras propias clases a través de los asistentes.

### **Paso 2: Añadir las dependencias necesarias.**

Abrimos el fichero pom.xml , al cual podemos acceder a través de nuestro proyecto, y seleccionamos la pestaña inferior pom.xml ; necesitamos añadir las dependencias necesarias para usar Spring. La siguiente, nos añadirá todos los ficheros necesarios para usar el módulo Core .

<!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>4.3.1.RELEASE</version>

</dependency>

De esta forma, el fichero pom.xml debería tener el siguiente contenido:

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.openwebinars</groupId>

<artifactId>HelloWorldSpring</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>HelloWorldSpring</name>

<url>http://maven.apache.org</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

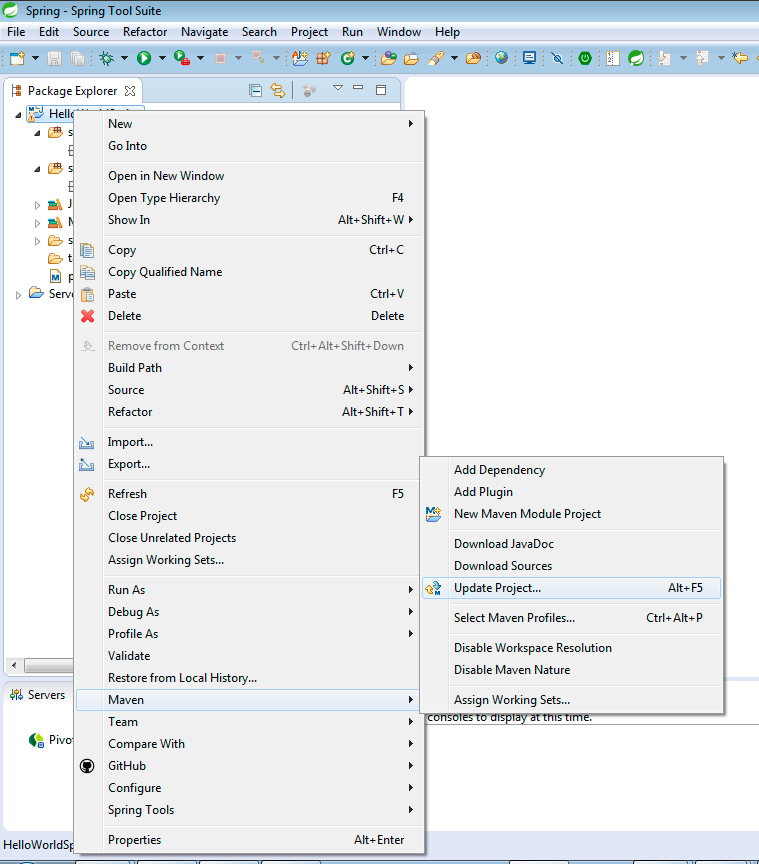
<version>4.3.1.RELEASE</version>

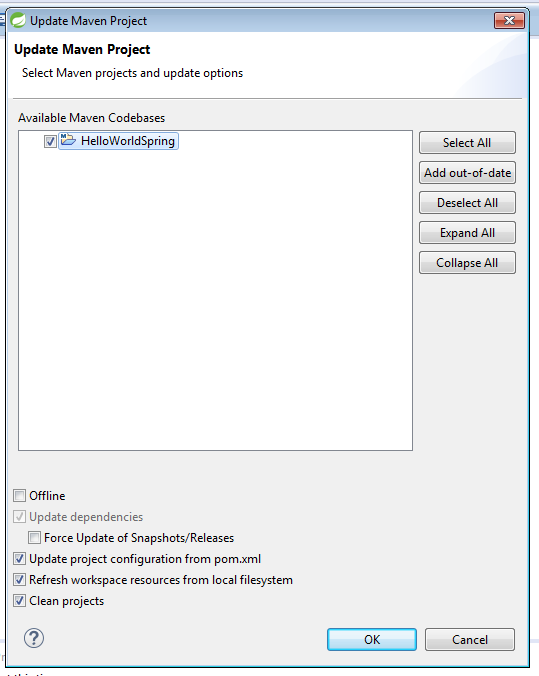
</dependency>

</dependencies>

</project>

Almacenamos el contenido del fichero, y procedemos a actualizar el proyecto Maven.





Actualización del repositorio local de Maven, para descargar todas las dependencias.

### **Paso 3: Añadir el código Java.**

Una vez configurado nuestro proyecto, vamos a dar paso a crear nuestro Hola Mundo .

En primer lugar, creamos un nuevo paquete, llamado com.openwebinars.beans , donde ubicaremos los beans de nuestro proyecto.

Posteriormente, creamos dentro de este nuevo paquete una nueva clase, llamada Mundo , y que tendrá un código muy sencillo (tan solo una propiedad, y sus correspondientes getter/setter ):

package com.openwebinars.beans;

public class Mundo {

private String saludo;

public String getSaludo() {

return saludo;

}

public void setSaludo(String saludo) {

this.saludo = saludo;

}

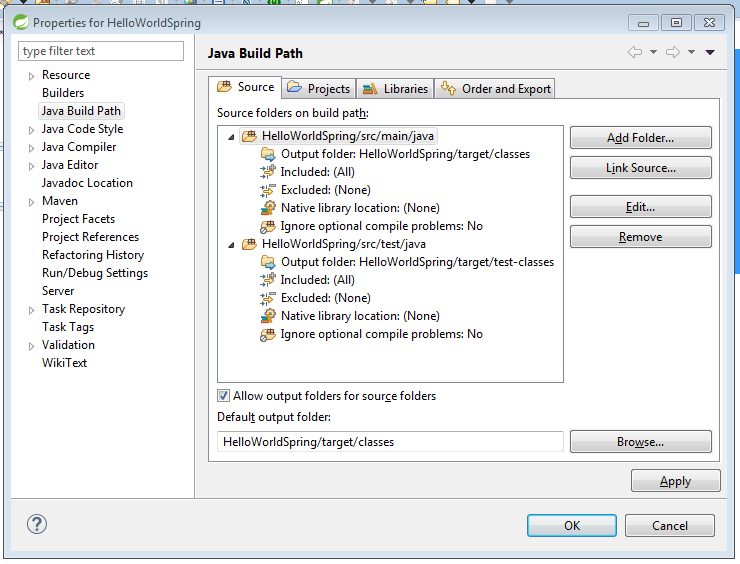
}

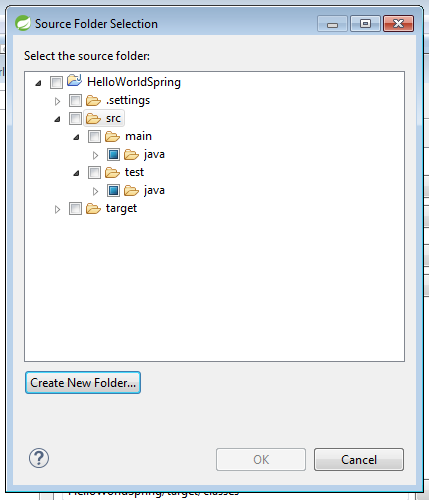
Ahora necesitamos definir este objeto, Mundo , en un contenedor , para que Spring lo pueda manejar. Esta operación la realizamos en el siguiente paso.

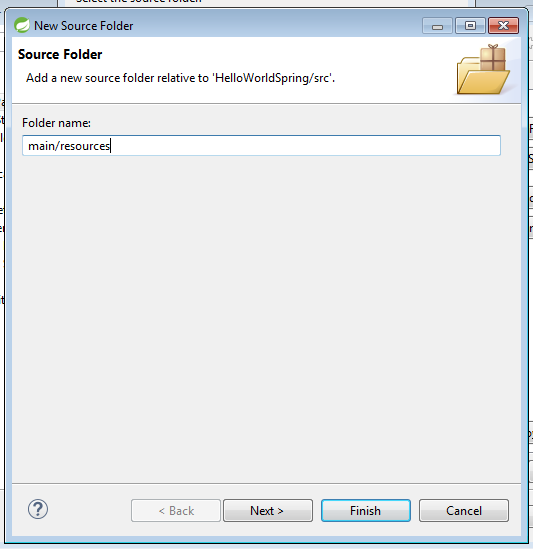
### **Paso 4: Definir el fichero de configuración XML.**

En primer lugar, vamos a crear la ubicación en la cual almacenaremos nuestro fichero de configuración de beans . Es una carpeta que llamaremos resources y que estará físicamente a la misma altura del árbol de directorios que la carpeta llamada java , que contiene el código fuente.

Mediante el botón derecho del ratón sobre el proyecto, o la combinación de teclas Alt+Enter , accedemos a las propiedades del proyecto, y de entre ellas accedemos a Java Build Path . Seleccionamos la pestaña Source , y pulsamos sobre el botón Add Folder. En el diálogo que nos aparece, seleccionamos la carpeta src y pulsamos el botón Create New Folder . La ruta de la carpeta será main/resources .

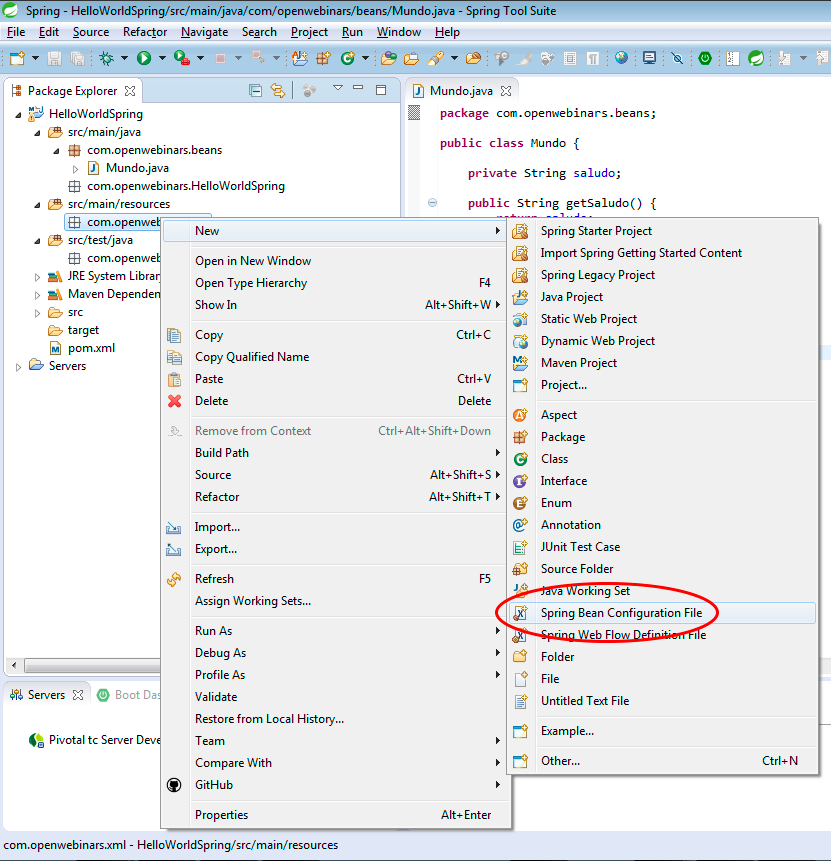


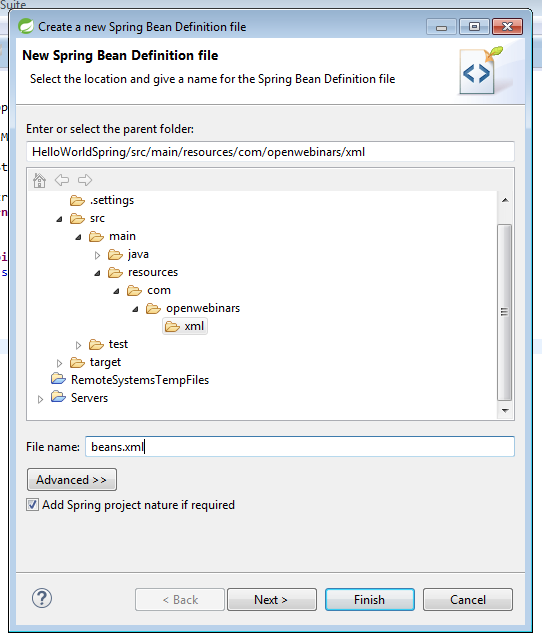




Creación de la nueva carpeta src/main/resources donde ubicaremos el fichero de configuración de beans .

Dentro, creamos un nuevo paquete llamado com.openwebinars.xml . Dentro de este paquete, vamos a ubicar el fichero de configuración. Para crearlo, seleccionamos sobre dicho paquete Botón derecho > New > Spring Bean Configuration File (si no aparece directamente, tendriamos que seleccionar New > Other y buscarlo dentro de los asistentes de Spring).





Creación del fichero de configuración beans.xml

Ahora es cuando añadiremos la magia de Spring. A través de la etiqueta bean , registraremos dentro del contenedor el objeto Mundo , para poder usarlo posteriormente dentro de nuestra aplicación. Veamos el código a añadir para poder comentarlo

...

<bean id="mundo" class="com.openwebinars.beans.Mundo">

<property name="saludo" value="Hola Mundo!"></property>

</bean>

Cada objeto deberá tener un identificador, un alias, que nos permitirá diferenciarlo de los demás a la hora de invocarlo. Dicho identificador lo establecemos a través del atributo id . En segundo lugar, el bean debe ir asociado a una clase ( POJO ) de nuestro proyecto. Dicha clase se establece mediante el atributo class ; hay que indicar la ruta completa de la clase (es decir,nombre\_paquete.nombre\_clase ).

Si queremos inicializar alguna de las propiedades del bean , no tenemos más que añadir un elemento property dentro del propio elemento bean , indicando el nombre de la propiedad, en nuestro caso saludo , y el valor que queremos establecerle inicialmente, que será Hola Mundo!.

### **Paso 5: Clase Main y ejecución.**

Por último, nuestra aplicación necesita un Main para poder ejecutarse. Creamos una nueva clase, llamada App , dentro del paquete principal, com.openwebinars.HelloWorldSpring .

El primer paso que debemos seguir es crear un ApplicationContext ; en particular, seleccionamos ClassPathXmlApplicationContext . Si el fichero XML estuviera en la misma ubicación que la clase, no necesitaríamos añadir ningún parámetro opcional; al no ser así, tenemos que indicar la ruta del paquete, de forma que la instanciación del objeto quedaría como sigue:

ApplicationContext appContext = new ClassPathXmlApplicationContext("com/openwebinars/xml/beans.xml");

Ahora, vamos instanciar el bean Mundo , a través del contexto . Para ello, usaremos el métodogetBean(...) de ApplicationContext , y utilizaremos alguna de las siguientes líneas de código:

* Usando el **id**

Mundo m = (Mundo) appContext.getBean("mundo");

* Usando el .class

Mundo m = (Mundo) appContext.getBean(Mundo.class);

En nuestro caso, preferimos la primera opción.

Para completar la primera versión de nuestro Main , añadimos una impresión por consola del saludo, quedando el código como sigue:

package com.openwebinars.HelloWorldSpring;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.openwebinars.beans.Mundo;

public class App {

public static void main(String[] args) {

ApplicationContext appContext = new ClassPathXmlApplicationContext("com/openwebinars/xml/beans.xml");

Mundo m = (Mundo) appContext.getBean("mundo");

System.out.println(m.getSaludo());

}

}

Si ahora ejecutamos este proyecto (como una aplicación java), podemos comprobar que la salida que obtenemos por la consola es la siguiente:

...

Hola Mundo!

Como podemos comprobar, hemos obtenido por consola Hola Mundo! , el mensaje que esperábamos.

Si revisamos el código, podemos comprobar que la variable appContext parece tener un warning: Resource leak: ‘appContext’ is never closed . Algunos programadores prefieren liberar el recurso asociado al contenedor de IoC una vez que no se va a utilizar más. Para ello, tendríamos que añadir la siguiente línea de código al final, quedando el código de la siguiente manera:

package com.openwebinars.HelloWorldSpring;

import org.springframework.context.ApplicationContext;

import org.springframework.context.ConfigurableApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.openwebinars.beans.Mundo;

public class App {

public static void main(String[] args) {

ApplicationContext appContext = new ClassPathXmlApplicationContext("com/openwebinars/xml/beans.xml");

Mundo m = (Mundo) appContext.getBean("mundo");

System.out.println(m.getSaludo());

((ConfigurableApplicationContext) appContext).close();

}

}

## **5.3 Otras formas de inyección de dependencia**

### **5.3.1 Inyección por constructor**

Si revisamos el código del ejemplo HelloWorldSpring , y en particular, el de la clase Mundo , podemos darnos cuenta de que ¡Oh! No tiene constructor .

package com.openwebinars.beans;

public class Mundo {

private String saludo;

public String getSaludo() {

return saludo;

}

public void setSaludo(String saludo) {

this.saludo = saludo;

}

}

¿Esto significa algo? En principio, no debería significar nada; Java, para aquellas clases que no definen un constructor de forma explícita, se encarga de definir un constructor (sin parámetros, obviamente) de forma implícita. Sin embargo, el hecho de no tener un constructor, nos dice que el tipo de inyección de dependencia que hemos hecho ha sido por método. ¿Y si quisiéramos realizarla por constructor? Los cambios a realizar serían francamente sencillos.

En primer lugar, definiríamos un constructor con parámetros para la clase Mundo , quedando la misma de la siguiente manera:

package com.openwebinars.beans;

public class Mundo {

private String saludo;

public Mundo(String saludo) {

this.saludo = saludo;

}

public String getSaludo() {

return saludo;

}

public void setSaludo(String saludo) {

this.saludo = saludo;

}

}

Teniendo ya un constructor explícito que recibe un parámetro con la dependencia que queremos inyectar, tendríamos que modificar el fichero de configuración, beans.xml . En este caso, en lugar de usar la etiqueta property , usaríamos constructor-arg para inicializar los valores, quedando el código de la siguiente manera:

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

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

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="mundo" class="com.openwebinars.beans.Mundo">

<!-- <property name="saludo" value="Hola Mundo!"></property>-->

<constructor-arg value="Hola Mundo!"></constructor-arg>

</bean>

</beans>

Si ejecutamos la aplicación, observaremos que el resultado es el mismo.

#### **5.3.1.1 Definición de los argumentos de un constructor si son más de uno**

Supongamos el escenario en el que la clase Mundo tuviera más de una propiedad:

package com.openwebinars.beans;

public class Mundo {

private String saludo;

private int radio;

public Mundo(String saludo) {

this.saludo = saludo;

}

public Mundo(int radio) {

this.radio = radio;

}

public Mundo(String saludo, int radio) {

this.saludo = saludo;

this.radio = radio;

}

public int getRadio() {

return radio;

}

public void setRadio(int radio) {

this.radio = radio;

}

public String getSaludo() {

return saludo;

}

public void setSaludo(String saludo) {

this.saludo = saludo;

}

}

Si el fichero de configuración lo modificamos para que tenga el siguiente aspecto:

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

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

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="mundo" class="com.openwebinars.beans.Mundo">

<!-- <property name="saludo" value="Hola Mundo!"></property>-->

<constructor-arg value="Hola Mundo!"></constructor-arg>

<constructor-arg value="6371"></constructor-arg>

</bean>

</beans>

Y el fichero App.java lo modificamos para poder imprimir por pantalla el valor correspondiente al radio:

package com.openwebinars.HelloWorldSpring;

import org.springframework.context.ApplicationContext;

import org.springframework.context.ConfigurableApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.openwebinars.beans.Mundo;

public class App {

public static void main(String[] args) {

ApplicationContext appContext = new ClassPathXmlApplicationContext("com/openwebinars/xml/beans.xml");

Mundo m = (Mundo) appContext.getBean("mundo");

System.out.println(m.getSaludo() + "(" + m.getRadio() + " km.)");

((ConfigurableApplicationContext) appContext).close();

}

}

Deberíamos obtener el siguiente resultado:

...

Hola Mundo!(6371 km.)

...

Como podemos comprobar, ha realizado la construcción del objeto de forma correcta. Pero, ¿y si queremos inicializar solamente un parámetro? O ¿y si cambiamos el orden de los argumentos?

Spring nos permite algunas introducir algunos atributos más a las etiquetas constructor-argpara esclarecer la inyección de dependencias:

* Establecimiento de los **tipos de datos**: de esta forma, la asignación de los argumentos se realiza en base al tipo de dato usado.

...

<bean id="mundo" class="com.openwebinars.beans.Mundo">

<constructor-arg class="int" value="6371"></constructor-arg>

<constructor-arg class="java.lang.String" value="Hola Mundo!"></constructor-arg>

</bean>

...

* Establecimiento en base al **índice del parámetro**en el constructor: de esta forma, se realiza la inyección en base al orden de los parámetros definidos en el constructor.

...

<bean id="mundo" class="com.openwebinars.beans.Mundo">

<constructor-arg index="1" value="6371"></constructor-arg>

<constructor-arg index="0" value="Hola Mundo!"></constructor-arg>

</bean>

...

* Establecimiento en base al **nombre del parámetro**en el constructor: de esta forma, la inyección se materializa usando el nombre de los parámetros del constructor.

...

<bean id="mundo" class="com.openwebinars.beans.Mundo">

<constructor-arg name="radio" value="6371"></constructor-arg>

<constructor-arg name="saludo" value="Hola Mundo!"></constructor-arg>

</bean>

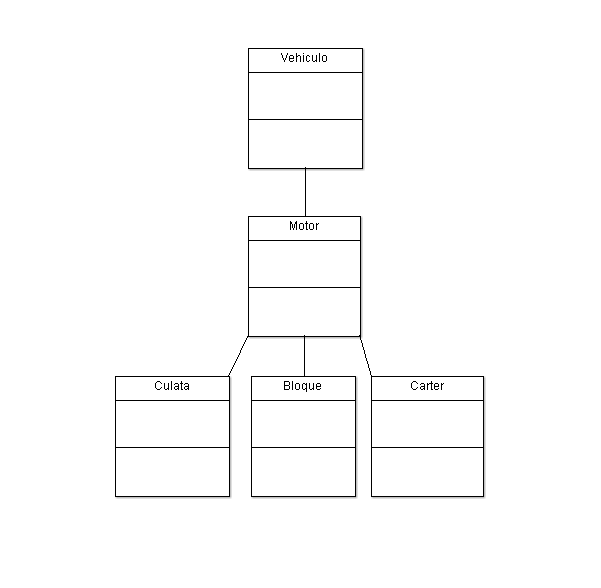
...

### **5.3.2 Inyección por constructor vs. inyección por método**

Ámbos mecanismos son posibles, y podemos mezclar su utilización como queramos. Sin embargo, algunos autores recomiendan utilizar la inyección por constructor para las dependenciasobligatorias , y por método aquellas que sean opcionales

## **5.4 Referencias entre beans**

Como podremos suponer, muy pocas apliaciones tendrán que manejar un solo bean. Normalmente, nuestro modelo y nuestra lógica de negocio tendrán que manejar múltiples beans . Hasta ahora, hemos visto como inyectar dependencias con tipos básicos java, pero ¿qué sucedería si necesitáramos inyectar como dependencia otro bean?



Detalle del diagrama de clases de Vehiculo

El atributo ref nos permite referenciar otro bean del contenedor. De esta forma, el ficherobeans.xml quedaría como sigue:

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

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

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="vehiculo" class="com.openwebinars.Beans.Vehiculo">

<property name="marca" value="Mercedes Benz"></property>

<property name="modelo" value="Clase C"></property>

<property name="motor" ref="motor"></property>

</bean>

<bean id="motor" class="com.openwebinars.Beans.Motor">

<property name="culata" ref="culata"></property>

<property name="bloque" ref="bloque"></property>

<property name="carter" ref="carter"></property>

<property name="cilindrada" value="1.8"></property>

</bean>

<bean id="culata" class="com.openwebinars.Beans.Culata">

<property name="cantidadBujias" value="4"></property>

</bean>

<bean id="bloque" class="com.openwebinars.Beans.Bloque">

<property name="cantidadCilindros" value="4"></property>

</bean>

<bean id="carter" class="com.openwebinars.Beans.Carter">

<property name="volumenAceite" value="6.5"></property>

</bean>

</beans>

### **5.4.1 ¿Y si en lugar de property quiero usar constructor-arg ?**

Siguiendo la flexible línea que nos ofrece Spring, podemos referenciar beans haciendo uso de la etiqueta <constructor-arg> en lugar de <property> , para hacer una inyección basada en constructor, en lugar de basada en los métodos setter . Tan solo tendríamos que cambiar el fichero de configuración para que quedara como sigue:

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

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

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="vehiculo" class="com.openwebinars.Beans.Vehiculo">

<constructor-arg name="marca" value="Mercedes Benz"></constructor-arg>

<constructor-arg name="modelo" value="Clase C"></constructor-arg>

<constructor-arg name="motor" ref="motor"></constructor-arg>

</bean>

<bean id="motor" class="com.openwebinars.Beans.Motor">

<constructor-arg name="culata" ref="culata"></constructor-arg>

<constructor-arg name="bloque" ref="bloque"></constructor-arg>

<constructor-arg name="carter" ref="carter"></constructor-arg>

<constructor-arg name="cilindrada" value="1.8"></constructor-arg>

</bean>

<bean id="culata" class="com.openwebinars.Beans.Culata">

<constructor-arg name="cantidadBujias" value="4"></constructor-arg>

</bean>

<bean id="bloque" class="com.openwebinars.Beans.Bloque">

<constructor-arg name="cantidadCilindros" value="4"></constructor-arg>

</bean>

<bean id="carter" class="com.openwebinars.Beans.Carter">

<constructor-arg name="volumenAceite" value="6.5"></constructor-arg>

</bean>

</beans>

## **5.5 Beans anidados**

Una de las maneras que tenemos para inyectar un bean dentro de otro es mediante el atributo ref, tanto si hacemos una inyección por setter como si la hacemos por constructor.

Sin embargo, existe otro método para poder inyectar un bean dentro de otro bean, y se conoce como inner beans o beans anidados.

<bean id="outer" class="...">

<!-- en lugar de utilizar una referencia a un bean, simplemente definimos dicho bean en línea -->

<property name="target">

<bean class="com.example.Class">

<property name="prop1" value="Asdfg"/>

<property name="prop2" value="12"/>

</bean>

</property>

</bean>

El ejemplo del apartado 5.4 quedaría de la siguiente forma:

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

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

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="vehiculo" class="com.openwebinars.Beans.Vehiculo">

<property name="marca" value="Mercedes Benz"></property>

<property name="modelo" value="Clase C"></property>

<property name="motor">

<bean class="com.openwebinars.Beans.Motor">

<property name="culata">

<bean class="com.openwebinars.Beans.Culata">

<property name="cantidadBujias" value="4"></property>

</bean>

</property>

<property name="bloque">

<bean class="com.openwebinars.Beans.Bloque">

<property name="cantidadCilindros" value="4"></property>

</bean>

</property>

<property name="carter">

<bean class="com.openwebinars.Beans.Carter">

<property name="volumenAceite" value="6.5"></property>

</bean>

</property>

<property name="cilindrada" value="1.8"></property>

</bean>

</property>

</bean>

</beans>

### **5.5.1 Anidación por constructor**

Análogamente, también puede realizarse la anidación por constructor:

<bean id="outterBean" class="outerClass">

<constructor-arg>

<bean class="innerClass">

...

</bean>

</constructor-arg>

</bean>

La versión del fichero beans.xml con la inyección por constructor sería la siguiente:

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

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

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="vehiculo" class="com.openwebinars.Beans.Vehiculo">

<constructor-arg name="marca" value="Mercedes Benz"></constructor-arg>

<constructor-arg name="modelo" value="Clase C"></constructor-arg>

<constructor-arg name="motor">

<bean class="com.openwebinars.Beans.Motor">

<constructor-arg name="culata">

<bean class="com.openwebinars.Beans.Culata">

<constructor-arg name="cantidadBujias" value="4"></constructor-arg>

</bean>

</constructor-arg>

<constructor-arg name="bloque">

<bean class="com.openwebinars.Beans.Bloque">

<constructor-arg name="cantidadCilindros" value="4"></constructor-arg>

</bean>

</constructor-arg>

<constructor-arg name="carter">

<bean class="com.openwebinars.Beans.Carter">

<constructor-arg name="volumenAceite" value="6.5"></constructor-arg>

</bean>

</constructor-arg>

<constructor-arg name="cilindrada" value="1.8"></constructor-arg>

</bean>

</constructor-arg>

</bean>

</beans>

### **5.5.2 Anidación vs. Referencia**

¿Cuál de las dos opciones es mejor? La respuesta es que depende

| **BEANS REFERENCIADOS** | **BEANS ANIDADOS** |
| --- | --- |
| Evitamos tener grandes bloques de código, que son difíciles de leer | El código estará más cohesionado (los beans están definidos donde se van inyectar). |
| Podemos obtener instancias de cualquiera de los beans que tengan propiedad id . | Todos los inner beans son anónimos. Si un bean se utiliza solamente comoparte de otro, y no debe ser invocado como un objeto fuera de este, la mejor opción es anidarlo. |
| Permiten la separación de la definición de beans en más de un fichero | Se puede combinar la referencia de beans con la anidación de otros. |

## **5.6 Colecciones**

Hasta ahora, todas las referencias entre beans que hemos manejado podríamos llamarlas simples, es decir, siempre hacía referencia a un objeto, nunca a una colección de ellos. De sobra sabemos que las colecciones son elementos de la Programación Orientada a Objetos que se usan constantemente. Veamos como nos permitirá manejar esto Spring.

### **5.6.1. Tipos de colecciones manejadas por Spring**

Spring nos va a permitir trabajar con los siguientes tipos de colecciones. Podemos ver en la siguiente tabla la etiqueta que necesitaremos en el fichero de configuración, y el tipo de colección (dentro del API de Java) a la que corresponden:

| **ETIQUETA** | **TIPO DE COLECCIÓN** |
| --- | --- |
| <list/> | [java.util.List](https://docs.oracle.com/javase/8/docs/api/java/util/List.html) |
| <set/> | [java.util.Set](https://docs.oracle.com/javase/8/docs/api/java/util/Set.html) |
| <map/> | [java.util.Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html) |
| <props/> | [java.util.Properties](https://docs.oracle.com/javase/8/docs/api/java/util/Properties.html) |

### **5.6.2 Uso de una colección <list>**

El esqueleto de la definición de un bean que utilizara una colección de tipo list sería el siguiente:

<bean id="beanId" class="com.openwebinars.colecciones.ClassName">

<property name="name" value="value"></property>

<property name="collection">

<list>

...

</list>

</property>

</bean>

Dentro del elemento <list> podemos definir:

* Una referencia ( <ref ...> ) a otro bean
* Un bean en linea

Al igual que en anteriores situaciónes, también podemos inyectar la dependencia de una lista usando un constructor.

### **5.6.3 Uso de una colección <set>**

El esqueleto de la definición de un bean que utilizara una colección de tipo set sería el siguiente:

<bean id="beanId" class="com.openwebinars.colecciones.ClassName">

<property name="name" value="value"></property>

<property name="collection">

<set>

...

</set>

</property>

</bean>

### **5.6.4 Uso de una colección <map>**

El esqueleto de la definición de un bean que utilizara una colección de tipo map sería el siguiente:

<bean id="beanId" class="com.openwebinars.colecciones.ClassName">

<property name="name" value="value"></property>

<property name="collection">

<map>

<entry key="thekey">

...

</entry>

...

</map>

</property>

</bean>

Como podemos observar, esta sintaxis difiere de las anteriores debido a la naturaleza clave -> valorde las colecciones de tipo map . Por ello, cada bean de la colección, tendrá que definirse en un elemento <entry ...> , asignándole a cada uno una clave ( key ).

### **5.6.5 Uso de una colección <props>**

El esqueleto de la definición de un bean que utilizara una colección de tipo props sería el siguiente:

<bean id="beanId" class="com.openwebinars.colecciones.ClassName">

<property name="name" value="value"></property>

<property name="collection">

<props>

<prop key="thekey">value</prop>

...

</props>

</property>

</bean>

Las colecciones props , que es la abreviatura de properties , nos sirven como los maps, pero para definir una serie de pares clave -> valor donde el valor será un literal: un número, una cadena de caracteres, un booleano.

## **5.7 Carga tardía**

Por defecto, ApplicationContext crea y configura todos los beans (de ámbito singleton ) como parte del proceso de inicialización. Cuando esta opción no es deseable, podemos evitar esta preinstanciación marcándolo como lazy-init=true . De esta forma, le estamos indicando al contenedor que realice la instanciación la primera vez que el bean sea solicitado, en lugar de en la inicialización del contexto.

<bean id="lazy" class="com.foo.ExpensiveToCreateBean" lazy-init="true"/>

<bean name="not.lazy" class="com.foo.AnotherBean"/>

Se puede gestionar todo este proceso de manera global para el fichero de configuración, mediante el atributo default-lazy-init del elemento <beans> .

<beans default-lazy-init="true">

<!-- no se preinstanciará ningún bean -->

</beans>

### **5.7.1 Uso de log4j para el ejemplo de carga tardía**

Para hacer nuestro ejemplo de carga tardía, vamos a añadir una nueva dependencia en el ficheropom.xml , que nos permitirá gestionar mejor el logging (registro o bitácora) de Spring.

Añadimos esta nueva dependencia:

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.14</version>

</dependency>

Con esta, indicamos que el sistema de logging que usaremos será [Log4j](http://logging.apache.org/log4j/2.x/), que está ampliamente extendido entre los desarrolladores de Java y Spring.

Hay que añadir información adicional de configuración. Para ello, creamos un nuevo fichero (usando el asistente de creación de ficheros), llamado log4j.properties dentro del directorsrc/main/resources , y añadimos el siguiente contenido:

log4j.rootCategory=INFO, stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %t %c{2}:%L - %m%n

log4j.category.org.springframework.beans.factory=DEBUG

## **6.1 Introducción**

El contenedor de Spring puedes gestionar de forma automática las dependencias entre beans que colaboran. Se puede permitir a Spring resolver automáticamente inspeccionando dentro del contenido del ApplicationContext . A este proceso se le llama autowiring .

## **6.2 Modos de autowiring**

Al usar la configuración del contenedor basada en XML, podemos establecer la inyección automática usando el atributo autowire del elemento <bean> . Esta funcionalidad tiene cuatro modos de trabajo. Hay que especificar, como valor del atributo, cual de ellos queremos usar:

| **MODO** | **EXPLICACIÓN** |
| --- | --- |
| no | Es el modo por defecto (es decir, autowire=no es equivalente a no escribir nada con respecto aautowiring ). Las referencias entre beans serán establecidas de alguna de las diferentes formas que hemos aprendido en las lecciones anteriores. |
| byName | La inyección automática se realiza en base al nombre de la propiedad. Spring busca un bean que tenga el mismo nombre que la propiedad que tiene la dependencia. |
| byType | Permite satisfacer la dependencia si Spring encuentra un bean que tenga el mismo tipo. |
| constructor | Análogo a byType , pero aplicando a argumentos de constructores. |

## **6.3 Limitaciones y desventajas del autowiring**

El autowiring funciona bien cuando su uso es consistente a lo largo de todo un proyecto; si no es así, podemos encontrarnos en alguna situación de confusión.

Algunas de las limitaciones son:

* Las dependencias explícitas en <property> y <constructor-arg> sobrescriben el comportamiento automático. No se pueden inyectar automáticamente propiedades de tipos de datos básicos.
* La inyección automática es menos exacta que la inyección explícita; con todo, Spring pone mucho cuidado en evitar la ambigüedad.
* La inyección automática no permite generar documentación para el contenedor de Spring.
* Una definición de varios beans de un mismo tipo que el que necesitamos para inyectar una dependencia pueden producir un problema. Si se trata de una colección, no pasa nada; pero en el caso de ser una dependencia que espera un valor, podemos obtener por respuesta una excepción.

## **6.4 Ejemplo de uso de autowire**

Vamos a intentar trabajar, de una forma sencilla, con un ejemplo que tenga tres clases: Cliente ,Persona , Ciudad .

package com.openwebinars.autowire;

public class Ciudad {

private String nombre;

public Ciudad() { }

public Ciudad(String nombre) {

this.nombre = nombre;

}

public String getNombre() {

return nombre;

}

public void setNombre(String nombre) {

this.nombre = nombre;

}

@Override

public String toString() {

return "Ciudad [nombre=" + nombre + "]";

}

}

package com.openwebinars.autowire;

public class Persona {

private String nombre;

private Ciudad ciudad;

public Persona() { }

public Persona(String nombre, Ciudad ciudad) {

this.nombre = nombre;

this.ciudad = ciudad;

}

public String getNombre() {

return nombre;

}

public void setNombre(String nombre) {

this.nombre = nombre;

}

public Ciudad getCiudad() {

return ciudad;

}

public void setCiudad(Ciudad ciudad) {

this.ciudad = ciudad;

}

@Override

public String toString() {

return "Persona [nombre=" + nombre + ", ciudad=" + ciudad + "]";

}

}

package com.openwebinars.autowire;

public class Cliente {

private long id;

private Persona persona;

public Cliente() { }

public Cliente(long id, Persona persona) {

this.id = id;

this.persona = persona;

}

public long getId() {

return id;

}

public void setId(long id) {

this.id = id;

}

public Persona getPersona() {

return persona;

}

public void setPersona(Persona persona) {

this.persona = persona;

}

@Override

public String toString() {

return "Cliente [id=" + id + ", persona=" + persona + "]";

}

}

Como hemos visto en los apartados superiores, para realizar el autowiring , necesitamos usar el atributo autowire de la etiqueta <bean> .

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

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

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="cliente" class="com.openwebinars.autowire.Cliente" autowire="byName">

<property name="id" value="1"></property>

<!-- <property name="persona" ref="persona"></property> -->

</bean>

<bean id="persona" class="com.openwebinars.autowire.Persona" autowire="byName">

<property name="nombre" value="Luismi"></property>

<!-- <property name="ciudad" ref="ciudad"></property>-->

</bean>

<bean id="ciudad" class="com.openwebinars.autowire.Ciudad">

<property name="nombre" value="Sevilla"></property>

</bean>

</beans>

También podríamos haber hecho la inyección automática de la dependencia por constructor , con el siguiente código de configuración:

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

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

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="cliente" class="com.openwebinars.autowire.Cliente"

autowire="constructor">

<constructor-arg name="id" value="1"></constructor-arg>

</bean>

<bean id="persona" class="com.openwebinars.autowire.Persona"

autowire="constructor">

<constructor-arg name="nombre" value="Luismi">

</constructor-arg>

</bean>

<bean id="ciudad" class="com.openwebinars.autowire.Ciudad">

<constructor-arg name="nombre" value="Sevilla"></constructor-arg>

</bean>

</beans>

## **7.1 Introducción**

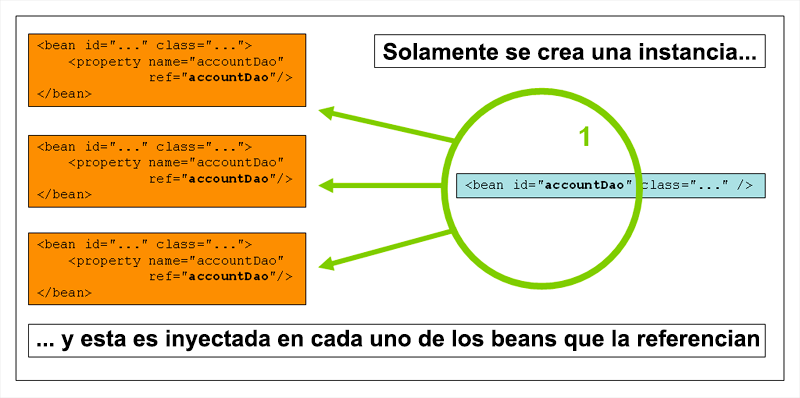
En Spring, no solo podemos controlar la inyección de varias dependencias en un solo objeto registrado como un bean, sino que además podemos controlar el ámbito o alcance (en inglés, scope) de los objetos creados. Este enfoque es potente y flexible, y permite controlar el ámbito a nivel de configuración, y así tener que evitar hacerlo directamente en Java.

En este apartado, conoceremos dos ámbitos: singleton y prototype (cuando trabajemos con Spring MVC veremos que existen algunos más: request , session , globalSession , application y WebSocket).

## **7.2 El ámbito singleton**

Cuando definimos un bean como singleton , estamos indicando que solamente vamos a tener una instancia de dicho bean, que será compartida para todas las referencias que existan del mismo.

Dicho de otro modo, al definir un bean como singleton , estamos indicando al contenedor de Spring que debe crear exactamente una instanacia del objeto. Dicha instancia será almacenada en caché, y todas las siguientes llamadas o peticiones a dicho bean, tendrán como resultado la instancia cacheada.



Explicación de un bean de ámbito singleton

En Spring, podemos hablar de singleton describiéndolo como una instancia por contenedor . Esto significa que si definimos un bean de una clase particular en un solo contenedor, entonces, el contenedor crea una y solo una instancia de la clase definida por ese bean.

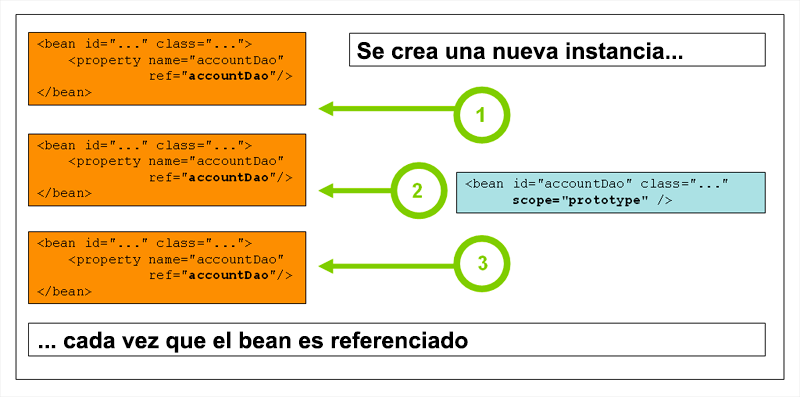
**El ámbito singleton es el ámbito por defecto en Spring**. Para definir un bean como singleton en XML, deberíamos escribir lo siguiente:

<bean id="accountService" class="com.foo.DefaultAccountService"/>

<!-- el siguiente modo es equivalente y redundante, ya que el ámbito singleton es el ámbito por defecto. --><bean id="accountService" class="com.foo.DefaultAccountService" scope="singleton"/>

## **7.3 El ámbito prototype**

De manera opuesta, tenemos el ámbito prototype , en el cual, para cada petición de un bean, tendremos como resultado una nueva instancia.



Explicación de un bean de ámbito prototype

En el siguiente ejemplo, definiremos un bean como prototype :

<bean id="accountService" class="com.foo.DefaultAccountService" scope="prototype"/>

A diferencia de otros ámbitos, Spring no gestiona el ciclo de vida completo de un bean definido como prototype : el contenedor instancia, configura y devuelve el objeto al cliente, sin realizar ninguna operación más. Tanto es así, que no toma ningún tipo de responsabilidad en la destrucción del objeto: es el cliente el encargado de liberar los recursos, en caso de que estos fueran costosos de mantener.

## **8.1 Introducción**

Spring nos permite interactuar con el contenedor, para realizar determinadas operaciones durante el ciclo de vida del mismo. Podemos ejecutar algún metodo justo después de instanciar un bean, y también justamente antes de destruirlo. Esto podemos hacerlo de 3 formas diferentes:

1. Implementando las interfaces InitializingBean y DisposableBean
2. Mediante el uso de las propiedades init-method y destroy-method en la configuración XML.
3. Usando las anotaciones @PostConstruct y @PreDestroy para decorar algún metodo del bean.

## **8.2 Uso de las interfaces InitializingBean yDisposableBean**

Spring provee de dos interfaces en el paquete org.springframework.beans.factory , llamadas InitializingBean y DisposableBean , siendo su nombre muy autodescriptivo .

* InitializingBean nos obliga a darle cuerpo al método afterPropertiesSet , y nos permitirá realizar algún tipo de lógica tras la inicialización del bean.
* DisposableBean nos obliga a implementar el método destroy , que nos permitirá ejecutar alguna funcionalidad antes de que el bean sea destruido.

Podemos ver ambas interfaces en funcionamiento en el siguiente ejemplo. A continuación, podemos ver el único bean que vamos a crear por ahora, llamado ClientesServicio :

package com.openwebinars.lifecycle;

import org.springframework.beans.factory.DisposableBean;import org.springframework.beans.factory.InitializingBean;

public class ClientesServicio implements InitializingBean, DisposableBean {

String mensaje;

public ClientesServicio() { }

public ClientesServicio(String mensaje) {

this.mensaje = mensaje;

}

public String getMensaje() {

return mensaje;

}

public void setMensaje(String mensaje) {

this.mensaje = mensaje;

}

public void afterPropertiesSet() throws Exception {

System.out.println("DESPUÉS DE INICIO");

}

public void destroy() throws Exception {

System.out.println("ANTES DEL FIN");

}

}

Como podemos apreciar, este bean implementa las ya citadas interfaces, InitializingBean yDisposableBean , lo cual nos obliga a darle cuerpo a los métodos afterPropertiesSet ydestroy . En nuestro caso, al funcionalidad será didática, y tan solo imprimiremos por consola dos mensajes: uno que nos indica el final del inicio, y otro el principio del fin.

Si revisamos la ejecución, Spring ha ejecutado los métodos en el orden que cabía esperar:

1. En primer lugar, podemos ver la ejecución del método afterPropertiesSet , cuyo resultado es la línea DESPUÉS DEL INICIO .
2. Posteriormente, encontramos la impresión del propio mensaje.
3. Por último, como parte del mecanismo del cierre del contexto, se ha invocado el métododestroy , dando como resultado el mensaje ANTES DEL FIN .

Ambos interfaces no tienen porque implementarse de forma conjunta; si necesitamos añadir algún tipo de funcionalidad solámente al inicio o al fin, podemos implmentar el interface necesario.

De las tres formas de implementar el manejo del ciclo de vida, esta es la ****menos adecuada****, ya que acopla nuestro código a Spring. Cualquiera de las dos siguientes es más recomendable.

## **8.3 Uso de de las propiedades init-method ydestroy-method**

También podemos conseguir esta mismo funcionalidad sin necesidad de implementar ninguna interfaz adicional, y haciendo uso de la configuración XML.

Ambas propiedades esperan como valor el nombre de un método del bean (normalmente, definido como void ). Dicho método será el que se ejecute, correspondientemente, tras la inicialización o antes de la destrucción del objeto.

Con tan solo unas pequeñas modificaciones, podemos comprobar el uso de estas propiedades con el código anterior:

* Modificamos la clase ClientesServicio , y eliminamos la implementación de los interfaces, así como los correspondientes métodos, de forma que nos quede el siguiente código:

package com.openwebinars.lifecycle;

public class ClientesServicio {

String mensaje;

public ClientesServicio() { }

public ClientesServicio(String mensaje) {

this.mensaje = mensaje;

}

public String getMensaje() {

return mensaje;

}

public void setMensaje(String mensaje) {

this.mensaje = mensaje;

}

}

Añadimos al mismo los métodos que queremos que se ejecuten durante el ciclo de vida. Podemos llamarlos initBean y destroyBean (el nombre puede ser arbitrario). En cada uno de ellos, tan solo escribiremos un mensaje por consola:

package com.openwebinars.lifecycle;

public class ClientesServicio {

String mensaje;

public ClientesServicio() { }

public ClientesServicio(String mensaje) {

this.mensaje = mensaje;

}

public String getMensaje() {

return mensaje;

}

public void setMensaje(String mensaje) {

this.mensaje = mensaje;

}

public void initBean() {

System.out.println("DESPUÉS DE INICIO");

}

public void destroyBean() {

System.out.println("ANTES DEL FIN");

}

}

Modificamos el fichero XML para añadir los atributos init-method y destroy-method , asignándole los valores correspondientes:

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

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="clientesServicio" class="com.openwebinars.lifecycle.ClientesServicio"

init-method="initBean" destroy-method="destroyBean">

<property name="mensaje" value="Un mensaje"></property>

</bean></beans>

Ejecutamos, y comprobamos que el resultado es el mismo.

## **8.4 Uso de de las anotaciones @PostConstruct y@PreDestroy**

En este caso, el uso de las anotaciones y el código XML es fácilmente intercambiable, ya que tan solo tenemos que sustituir la propiedad XML por la anotación correspondiente sobre el método adecuado. Como podemos imaginar:

* @PostConstruct se ejecutará tras la inicialización y
* @PreDestroy se ejecutará antes de finalizar.

Estas anotaciones no son propias de Spring, sino que pertenecen al estándar JSR-250 de Java.

En segundo lugar, añadimos las anotaciones sobre el código Java:

package com.openwebinars.lifecycle;

import javax.annotation.PostConstruct;import javax.annotation.PreDestroy;

public class ClientesServicio {

String mensaje;

public ClientesServicio() { }

public ClientesServicio(String mensaje) {

this.mensaje = mensaje;

}

public String getMensaje() {

return mensaje;

}

public void setMensaje(String mensaje) {

this.mensaje = mensaje;

}

@PostConstruct

public void initBean() {

System.out.println("DESPUÉS DE INICIO");

}

@PreDestroy

public void destroyBean() {

System.out.println("ANTES DEL FIN");

}

}

Si al tratar de importar las depedencias de javax.annotation.PostConstruct yjavax.annotation.PreDestroy comprobamos que STS no las localiza, es posible que tengamos que revisar con que versión de Java está configurado el proyecto. A partir de Java 6 lo tenemos disponible. Para cambiar la versión del JDK asociada a nuestro proyecto, podemos seguir los siguientes pasos:

* Botón derecho sobre el proyecto > Properties
* En la lista de la izquierda, seleccionamos Java Build Path , y de entre las pestañas de la derecha, seleccionamos Libraries .
* Hacemos clic una vez sobre JRE System Library y pulsamos el botón Edit.
* Seleccionamos, en Execution environment , alguna versión de Java superior o igual a la 6.

Java 8 es muy recomendable, puesto que ha incorporado novedades muy chulas, como el uso de funciones lambda, métodos default en las interfaces, mejoras en el tratamiento de fechas, nuevas excepciones…

Si ejecutamos ahora nos daremos cuenta de que… OUCH! ¿Y mis mensajes? Pues ha sucedido lo siguiente: para que Spring se encargue de escanear los beans en búsqueda de las anotaciones@PostConstruct y @PreDestroy , tenemos que decírslo explícitamente . Esto lo hacemos declarando un nuevo bean en nuestro fichero XML.

<bean class="org.springframework.context.annotation.CommonAnnotationBeanPostProcessor"></bean>

Se trata de un bean especial (hay que indicar que no le establecemos la propiedad id ) definido por Spring y que se encargará de escanear nuestros demás beans en la búsqueda de anotaciones definidas por la JSR-250, entre ellas, @PostConstruct y @PreDestroy .

## **8.5 Uso de BeanPostProcessor**

El interfaz BeanPostProcessor define métodos callback que pueden ser implementados para añadir nuestra propia lógica de instanciación, de resolución de dependencias, etc. Si necesitas implementar alguna lógica después de que Spring container finalice la instanciación, configuración e inicialización del bean, se pueden conectar una o más implementaciones de BeanPostProcessor.

* Object postProcessAfterInitialization(Object bean, String beanName) : se ejecutará este código después de la inicialización de cualquier bean del contenedor.
* Object postProcessBeforeInitialization(Object bean, String beanName) : se ejecutará este código antes de la inicialización de cualquier bean del contenedor.

Para usarlo, tenemos que crear una clase propia, que implemente la interfazorg.springframework.beans.factory.config.BeanPostProcessor :

package com.openwebinars.beanpostprocessorsample;

import org.springframework.beans.BeansException;import org.springframework.beans.factory.config.BeanPostProcessor;

public class CustomBeanPostProcessor implements BeanPostProcessor {

public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {

System.out.println(String.format("Bean %s inicializado satisfactoriamente".toUpperCase(), beanName));

return bean;

}

public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {

System.out.println(String.format("Inicializando bean %s".toUpperCase(),beanName));

return bean;

}

}

Además, lo tenemos que registrar en el fichero de configuración del contenedor, para que Spring lo pueda gestionar:

...

<bean class="com.openwebinars.beanpostprocessorsample.CustomBeanPostProcessor"></bean>

...

Se pueden añadir ****tantos BeanPostProcessor como estimemos necesarios****, para realizar diferentes tareas: loggin , validaciones, etc..

## **9.1 JavaConfig**

Ya que esta va a ser la forma de configurar que ****menos****vamos a usar por ahora, la aprenderemos primero.

Spring soporta el uso de la configuración mediante anotaciones sin tener que modificar (si no se desea) el código fuente. Para ello, hace uso, básicamente, de dos anotaciones: @Configuration y@Bean .

La anotación @Bean se usa para indicar a un metodo que va a instanciar, configurar e inicializar un nuevo objeto gestionado por Spring IoC container . Podemos decir, sin ningún lugar a dudas, que la anotación @Bean realiza la misma tarea que el elemento <bean> en la configuración por XML.

Si anotamos una clase con @Configuration , estamos diciendo que su principal propósito será la definición de una serie de beans.

Un ejemplo de fichero de configuración simple sería el siguiente:

@Configurationpublic class AppConfig {

@Bean

public MyService myService() {

return new MyServiceImpl();

}

}

La clase AppConfig realiza las mismas tareas que el siguiente XML:

<beans> <bean id="myService" class="com.acme.services.MyServiceImpl"/>

</beans>

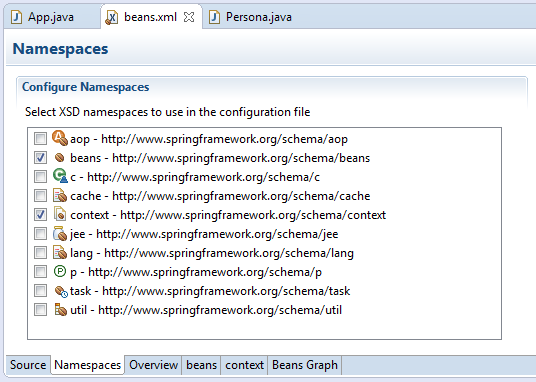
## **9.2 Anotaciones**

También vamos a poder maridar el uso de anotaciones de forma conjunta con la configuración XML (y así evitar la creación de un fichero JavaConfig ).

#### **Configuración de los beans de escaneo de anotaciones**

Para no tener que añadir a nuestro fichero de configuración referencias complejas a beans que define Spring, y que sirven, independientemente, para usar las diferentes anotaciones, podemos uasr un elemento que es capaz de resumir todas ellas en una línea de código, de forma que no tendremos que recordar si tenemos que registrar este u otro bean. El código es el siguiente:

<context:annotation-config />

Si lo añadimos en nuestro fichero, podremos comprobar que STS nos lanza un error, ya que no tenemos definido el namespace con prefijo context . Para ello, vamos a pulsar sobre la pestaña Namespaces del fichero bean.xml y seleccionaremos context tal y como podemos ver en la siguiente figura:   
  
Selección del namespace context   
  
  
Además, para que Spring escanee automáticamente un determinado paquete en busca de clases que estén anotadas como beans necesitamos otra anotación más:

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

El valor de la propiedad base-package lo marcará la ruta del paquete donde tengamos ubicados los beans.

@Component es la anotación más básica, e indica que una clase será tratada como un bean. Estudiaremos más en profundidad esta anotación en la lección sobre estereotipos. Si la acompañamos de la anotación @Value , podemos inyectar algunos valores iniciales en nuestro bean.

@Componentpublic class Mundo {

@Value("Hola Mundo!")

private String saludo;

public Mundo() { }

//...

}

A continuación, trabajaremos con las siguientes anotaciones

* @Required : indicará que tenemos que satisfacer la dependencia en la configuración.
* @Autowired : realiza la misma función que la propiedad autowire del elemento <bean> .
* @Qualifier : que usada junto a @Autowired nos permite afinar más en la inyección automática.

### **9.2.1 Anotación @Required**

La anotación @Required va a indicar que una determinada propiedad debe ser establecida en tiempo de configuración, a través de alguno de los mecanismos que ya conocemos:

* Un elemento <property>
* Un elemento <constructor-arg>
* Autowiring

El contenedor lanzará una excepción si la propiedad afectada del bean no ha sido seteada . Esto nos permitirá evitar, desde una etapa muy temprana, excepciones del tipo NullPointerException .

@Required

private Provincia provincia;

### **9.2.2 Anotaciones @Autowire y @Qualifier**

Estas anotaciones nos van a permitir realizar la inyección automática de beans, al igual que aprendimos a realizarlo en las lecciones anteriores a través de XML.

#### **9.2.2.1 @Autowired**

Podemos sustituir la anotación @Autowired , que es propia de Spring, con la anotación@Inject , que es propia del JSR330. Estas anotaciones son escaneadas de la misma forma que las anotaciones Spring. Tan solo hay que añadir los jar necesarios a nuestro classpath, mediante la siguiente dependencia:

<dependency>

<groupId>javax.inject</groupId>

<artifactId>javax.inject</artifactId>

<version>1</version></dependency>

Podemos usar la anotación @Autowired tanto para constructores, como para métodos setter:

public class MovieRecommender {

private final CustomerPreferenceDao customerPreferenceDao;

@Autowired

public MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {

this.customerPreferenceDao = customerPreferenceDao;

}

// ...

}

public class SimpleMovieLister {

private MovieFinder movieFinder;

@Autowired

public void setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

// ...

}

Por defecto, el autowiring convierte las propiedades a las que afecta en @Required . Si queremos cambiar dicho comportamiento por defecto, lo podemos hacer mediante la siguiente propiedad de la anotación:

public class SimpleMovieLister {

private MovieFinder movieFinder;

@Autowired(required=false)

public void setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

// ...

}

#### **9.2.2.2 @Qualifier**

Esta anotación nos permitirá afinar el trabajo a la hora de realizar la inyección de dependencias automática.

Supongamos que modificamos el fichero de configuración, beans.xml , de forma que registramos dos poblaciones diferentes:

<?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.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.3.xsd">

<context:annotation-config />

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

<bean id="Sevilla" class="com.openwebinars.anotaciones.autowire.Provincia">

<property name="nombre" value="Sevilla"></property>

</bean>

<bean id="SevillaCapital" class="com.openwebinars.anotaciones.autowire.Poblacion">

<property name="nombre" value="Sevilla"></property>

</bean>

<bean id="MairenaAljarafe" class="com.openwebinars.anotaciones.autowire.Poblacion">

<property name="nombre" value="Mairena del Aljarafe"></property>

</bean>

<bean id="cliente" class="com.openwebinars.anotaciones.autowire.Cliente">

<property name="id" value="1"></property>

</bean>

<bean id="persona" class="com.openwebinars.anotaciones.autowire.Persona">

<property name="nombre" value="Luismi"></property>

</bean>

</beans>

Para indicarle, explícitamente, a Spring, cual es el bean que debe inyectar como dependencia, podemos usar la anotación @Qualifier sobre el código Java:

package com.openwebinars.anotaciones.autowire;

import org.springframework.beans.factory.annotation.Autowired;import org.springframework.beans.factory.annotation.Qualifier;

public class Persona {

private String nombre;

@Autowired

@Qualifier("MairenaAljarafe")

private Poblacion poblacion;

public Persona() {

}

public Persona(String nombre, Poblacion poblacion) {

this.nombre = nombre;

this.poblacion = poblacion;

}

//...

}

De esta forma, la inyección del bean MairenaAljarafe se realiza correctamente en el beanPersona .

## **9.3 ¿Qué configuración es mejor? Anotaciones vs. XML**

La introducción del enfoque de configuración basado en anotaciones planteó la pregunta de si dicho enfoque es mejor que XML. La respuesta inmedianta es que depende . Si profundiamos algo más en la respuesta, podemos decir que cada uno tiene sus ventajas, y por lo general será el desarrollador el responsable de elegir cual le conviene. Por como están definidas, las anotaciones proporcionan una gran cantidad de información in situ , de forma que la configuración es más concisa y breve. Sin embargo, XML es la mejor opción para conectar componentes sin tocar su código fuente y recompilarlos. Unos desarrolladores prefieren tener la información sobre la configuración cerca del código al que afecta, y otros argumentan que de esa forma está descentralizada y es más complicada de controlar.

No importa la elección, puesto que Spring soporta todas sus opciones con ambas; incluso utilizar ambas a la vez. Vale la pena destacar que el uso de anotaciones junto con la configuración de los beans a través de código java ( JavaConfig ) permite usar las anotaciones de forma no invasiva , sin necesidad de tocar el código fuente de las clases afectadas.   
Spring Tool Suite soporta el uso de todos los métodos.

## **10.1 @Component y otras anotaciones de estereotipos**

Spring nos provee de varias anotaciones de estereotipos: @Component , @Service ,@Controller y @Repository (de esta última hablaremos en el bloque de Spring Data).@Component es un estereotipo genérico para cualquier componente manejado por Spring.@Repository , @Service y @Controller son especializaciones de @Component para casos de uso más específicos, de forma que las clases anotadas con las mismas se adaptarán mejor a esos casos (por ejemplo, @Service para la capa de servicios o lógica de negocio, o @Repository para la capa DAO o de acceso a datos).

## **10.2 Detección automática de clases y registro de las definiciones de los beans**

Spring puede detectar automáticamente clases estereotipadas y registrar sus correspondientesBeanDefinition en nuestro contenedor. Sea el siguiente ejemplo:

package org.example;

@Servicepublic class SimpleMovieLister {

private MovieFinder movieFinder;

@Autowired

public SimpleMovieLister(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

}

package org.example;

@Repositorypublic class JpaMovieFinder implements MovieFinder {

// omitimos la implementación para poner el acento en la definición de la clase.

}

Para autodetectar estas clases y registrarlas en los correspondientes beans, podríamos usar el siguiente código de configuración 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.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd">

<context:component-scan base-package="org.example"/>

</beans>

Esto provocará que Spring escanee todas las clases definidas en el paquete org.example en busca de beans definidos con @Component o alguna de sus especializaciones, para registrarlos como beans .

En la lección anterior, descubrimos que existía un nuevo elemento, perteneciente alnamespace context , que era <context:annotation-config /> ; este servía para escanear las diferentes anotaciones que podíamos haber añadido a los beans que estuvieran definidos en nuestro XML. El uso de la anotación <context:component-scan ... />incluye implícitamente a <context:annotation-config /> , con lo que solamente tendríamos que utilizar <context:component-scan ... /> .

### **10.2.1 Filtrado y personalización del escaneo de beans**

Por defecto, las clases anotadas con @Component , @Repository , @Service , @Controllerson las candidatas a ser detectadas. Aun así, se puede modificar esta configuración añadiendo algunos filtros. Estos pueden ser de inclusión o de exclusión mediante los elementos<context:include-filter> y <context:exclude-filter> .

| **TIPO DE FILTRO** | **EXPRESIÓN DE EJEMPLO** | **DESCRIPCIÓN** |
| --- | --- | --- |
| Anotación (por defecto) | org.example.SomeAnnotation | Una anotación que debe estar presente en los componentes |
| Asignable | org.example.SomeClass | Una clase (o interfaz) que los componentes deben extender/implementar |
| Expresión regular | org\.example\.Default.\* | Una expresión regular que debe ser cumplida por los nombres de los componentes. |
| Customizado | org.example.MyFilter | Una implementación propia del interfazorg.springframework.core.type.TypeFilter . |

A continuación podemos ver un ejemplo de escaneo con filtrado, en el que se incluirán todas las clases de dos paquetes que cumplan una expresión regular, y excluiremos todas aquellas clases anotadas con @Repository :

<beans>

<context:component-scan base-package="org.example">

<context:include-filter class="regex"

expression="org.example.dao.\*DAO.\*" />

<context:include-filter class="regex"

expression="org.example.services.\*Service.\*" />

<context:exclude-filter class="annotation"

expression="org.springframework.stereotype.Repository"/>

</context:component-scan>

</beans>

Como podemos observar:

* Se incluyen todas las clases del paquete org.example.dao que contengan DAO como parte de su nombre.
* Se incluyen todas las clases del paquete org.example.service que contengan Servicecomo parte de su nombre.
* Se excluyen todas las clases anotadas con @Repository .

## **10.3 Ámbitos mediante anotaciones**

Al igual que con los objetos gestionados por Spring en general, el ámbito por defecto y más usado es Singleton . De todas formas, habrá ocasiones (en el bloque de Spring MVC lo trabajaremos en profundidad) que necesitemos modificar esta configuración con la anotación @Scope . Veamos un ejemplo

@Scope("prototype")@Repository

public class MovieFinderImpl implements MovieFinder {

// ...

}

## **10.4 Uso de la anotación @Qualifier con estereotipos**

La anotación @Qualifier , que trabajamos en la lección anterior, nos permitía afinar a la hora de inyectar una dependencia con autowiring . En este ejemplo podemos ver como realizar toda la configuración con anotaciones, y no usando XML para la definición, como en la lección pasada.

@Servicepublic class SimpleMovieLister {

private MovieFinder movieFinder;

@Autowired

@Qualifier("hibernate")

public SimpleMovieLister(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

}

@Repository@Qualifier("hibernate")

public class JpaMovieFinder implements MovieFinder {

// ...

}

窗体顶端

## **11.1 Introducción**

El desarrollo de software es una tarea complicada, la cual depende en gran medida de la experiencia de las personas involucradas, en particular de los desarrolladores.

El 80% de los aportes vienen del 20% del personal.

La comprensión del software es uno de los problemas más complicados en la tarea de mantenimiento y evolución.

El 80% del esfuerzo esta en el 20% del código desarrollado.

Uno de los pilares básicos de la Orientación a Objetos en el desarrollo de software es la búsqueda de la ****reutilización****, que tendría las siguientes ventajas:

1. Reducción de tiempos.
2. Disminución del esfuerzo de mantenimiento.
3. Eficiencia.
4. Consistencia.
5. Fiabilidad.
6. Protección de la inversión en desarrollos.

Entre los diferentes mecanismos de reutilización están:

* ****Componentes****: elemento de software suficientemente pequeño para crearse y   
  mantenerse pero suficientemente grande para poder utilizarse.
* ****Frameworks****: bibliotecas de clases preparadas para la reutilización que pueden utilizar a su vez componentes.
* ****Objetos distribuidos****: paradigma que distribuye los objetos de cooperación a través de una red heterogénea y permite que los objetos interoperen como un todo unificado.
* ****Patrones de diseño****: sobre este último abundaremos un poco más.

Los patrones son una forma literaria para resolver problemas de   
ingeniería del software, que tienen sus raíces en los patrones de la arquitectura. Los diseñadores y analistas de software más experimentados aplican de forma intuitiva algunos criterios que solucionan los problemas de manera elegante y efectiva. La ingeniería del software se enfrenta a problemas variados que hay que identificar para poder utilizar la misma solución (aunque matizada) con problemas similares.

Según Christopher Alexander, “Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, para describir después el núcleo de la solución a ese problema, de tal manera que esa solución pueda ser usada más de un millón de veces, sin hacerlo ni siquiera dos veces de la misma forma” .

Sobre patrones de diseño existe una gran cantidad de literatura, si bien uno de los documentos más básico es el libro “Design Patterns: Elements of Reusable Object-Oriented Software” (Gang of Four, [GoF]) .

## **11.2 Patrón MVC (Modelo-Vista-Controlador)**

El patrón de arquitectura MVC (Modelo Vista Controlador) es un patrón que define la organización independiente del Modelo (Objetos de Negocio), la Vista (interfaz con el usuario u otro sistema) y elControlador (controlador del flujo o workflow de la aplicación).

De esta forma, dividimos el sistema en tres capas donde, como explicaremos más adelante, tenemos la encapsulación de los datos, la interfaz o vista por otro y por último la lógica interna o controlador.

El patrón de arquitectura “modelo vista controlador”, es una filosofía de diseño de aplicaciones, compuesta por:

* Modelo
  + Contiene el núcleo de la funcionalidad (dominio) de la aplicación.
  + Encapsula el estado de la aplicación.
  + No sabe nada / independiente del Controlador y la Vista.
* Vista
  + Es la presentación del Modelo.
  + Puede acceder al Modelo pero nunca cambiar su estado.
  + Puede ser notificada cuando hay un cambio de estado en el Modelo.
* Controlador
  + Reacciona a la petición del Cliente, ejecutando la acción adecuada y creando el modelo pertinente.

Para entender cómo funciona nuestro patrón Modelo vista controlador, se debe entender la división a través del conjunto de estos tres elementos y como estos componentes se comunican unos con los otros y con otras vistas y controladores externos a el modelo principal. Para ello, es importante saber que el controlador interpreta las entradas del usuario (tanto teclado como el ratón), enviado el mensaje de acción al modelo y a la vista para que se proceda con los cambios que se consideren adecuados

A continuación, podemos ver el patrón expresado en una imagen:

1. El usuario solicita, usando algún dispositivo, algún recurso. Lo puede hacer escribiendo una URL en el navegador, pulsando un botón o desencadenando otro tipo de evento. Esta solicitud es recibida por el controlador.
2. El controlador, una vez recibida la solicitud, pasa a procesarla. Es posible que haya una parte del código que sirva para decidir quien será el método concreto que haga este procesamiento, y si para ello, hay que extraer algún dato recibido a través de la petición. A partir de ahí, invoca al modelo, para realizar la operación solicitada.
3. El modelo, que es la representación de los objetos de nuestro dominio, posiblemente necesite interaccionar con el repositorio donde tengamos almacenados los datos (ficheros, base de datos relacional, NoSQL, API REST, …), pidiendo unos datos y recibiendo los mismos.
4. Los datos obtenidos en el paso (3) son enviados al controlador, para que estos puedan ser, en última instancia, mostrados al usuario.
5. Antes de ser enviados, los datos pueden ser procesados. Cuando lo hayan sido, si fuera necesario, son inyectados en la vista.
6. La vista, con los datos necesarios a mostrar, se entrega al usuario.

### **11.2.1 Ventajas y desventajas del uso de MVC**

* Ventajas:
  + Soporte para múltiples vistas . Dado que la vista se haya separada del modelo, y no hay dependencia directa entre el modelo y la vista.
  + Adaptación al cambio . Los requisitos de la IU tienden a cambiar más rápidamente que las reglas de negocio.
* Inconvenientes:
  + Complejidad . El patrón introduce nuevos niveles de indirección y por lo tanto aumenta ligeramente la complejidad de la solución (Spring MVC nos permitirá mitigar esto). Complejidad también en la depuración.
  + Costo de actualizaciones frecuentes . Desacoplar el modelo y la vista no significa que los desarrolladores del modelo puedan ignorar la naturaleza de las vistas.

## **11.3 Front Controller (Controlador frontal o de fachada)**

El patrón Front Controller es usado en aplicaciones web para tener un mecanismo centralizado de recepción de peticiones, que será gestionadas por un solo objeto manejador. De esta forma, este manejador puede incorporar autenticación, autorización, logging , … antes de pasar la petición a su correspondiente manejador concreto.

En este patrón intervienen las siguientes entidades:

* Controlador Frontal ( Front Controller ): Manejador único que gestionará todas las peticiones.
* Despachador ( Dispatcher ): El controlador frontal deberá usar un despachador, que enviará cada petición a su correspondiente manejador.
* Vista ( View ): el objeto esperado tras la petición.

A continuación, podemos ver el patrón expresado en una imagen:

Spring MVC implementa este patrón de una forma transparente para nosotros, a través del objetoorg.springframework.web.servlet.DispatcherServlet , como veremos en las próximas lecciones.

### **11.3.1 Ventajas y desventajas del uso de MVC**

* Ventajas:
  + Tenemos centralizado en un único punto la gestión de las peticiones.
  + Aumentamos la reusabilidad de código.
  + Mejoramos la gestión de la seguridad.
* Desventajas:
  + La velocidad de respuesta disminuye al tener que ser procesadas las peticiones primero por el controlador.

## **11.4 Patrón DAO ( Data Access Object )**

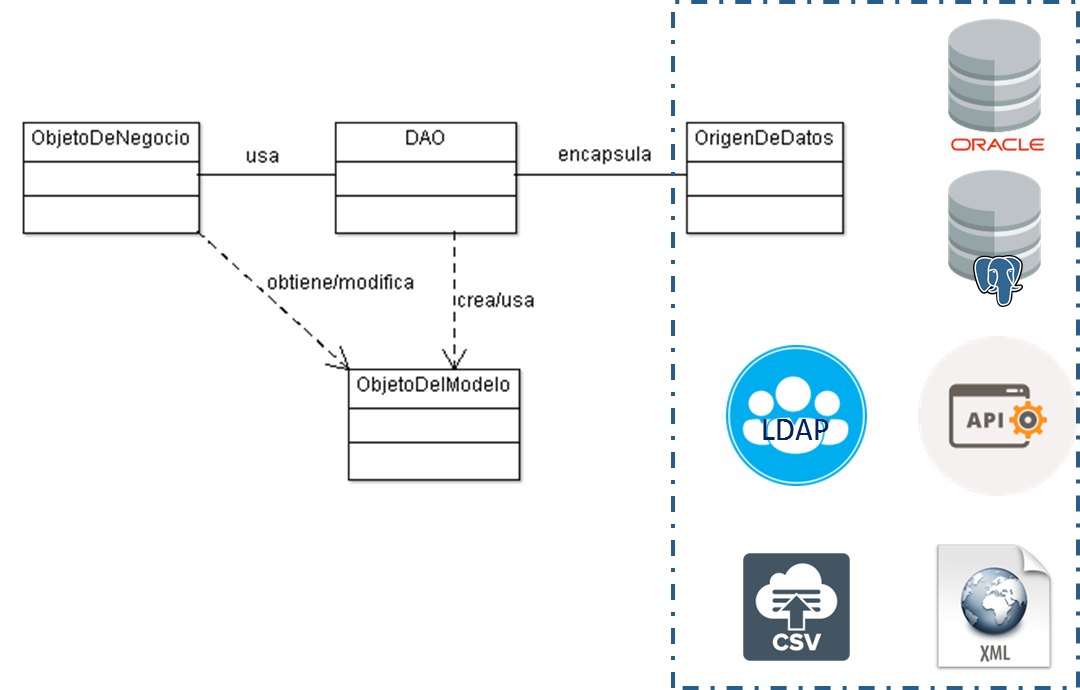
El acceso a los datos varía dependiendo de la fuente de los datos. El acceso al almacenamiento persistente, como una base de datos, varía en gran medida dependiendo del tipo de almacenamiento (bases de datos relacionales, bases de datos NoSQL, ficheros planos, etc.) y de la implementación del vendedor.

Toda aplicación web en el mundo real necesitan utilizar datos persistentes en algún momento. Para muchas de ellas, este almacenamiento persistente se implementa utilizando diferentes mecanismos. Por ejemplo, los datos podrían residir en sitemas heredados, repositorios LDAP, etc. Otro ejemplo es donde los datos los proporcionan servicios a través de sistemas externos como los sistemas de integración negocio-a-negocio (B2B), servicios de tarjetas de crédito, APIs REST, etc.

La solución a este problema nos la proporciona el patrón DAO: ****utilizar un Data Access Object (DAO) para abstraer y encapsular todos los accesos a la fuente de datos. El DAO maneja la conexión con la fuente de datos para obtener y almacenar datos****.

El DAO implementa el mecanismo de acceso requerido para trabajar con la fuente de datos. Los componentes de negocio que tratan con el DAO utilizan un interface simple expuesto por el DAO para sus clientes. El DAO oculta completamente los detalles de implementación de la fuente de datos a sus clientes. Como el interface expuesto por el DAO no cambia cuando cambia la implementación de la fuente de datos subyacente, este patrón permite al DAO adaptarse a diferentes esquemas de almacenamiento sin que esto afecte a sus clientes o componentes de negocio. Esencialmente, el DAO actúa como un adaptador entre el componente y la fuente de datos.

A continuación, podemos ver el patrón expresado en un diagrama de clases:



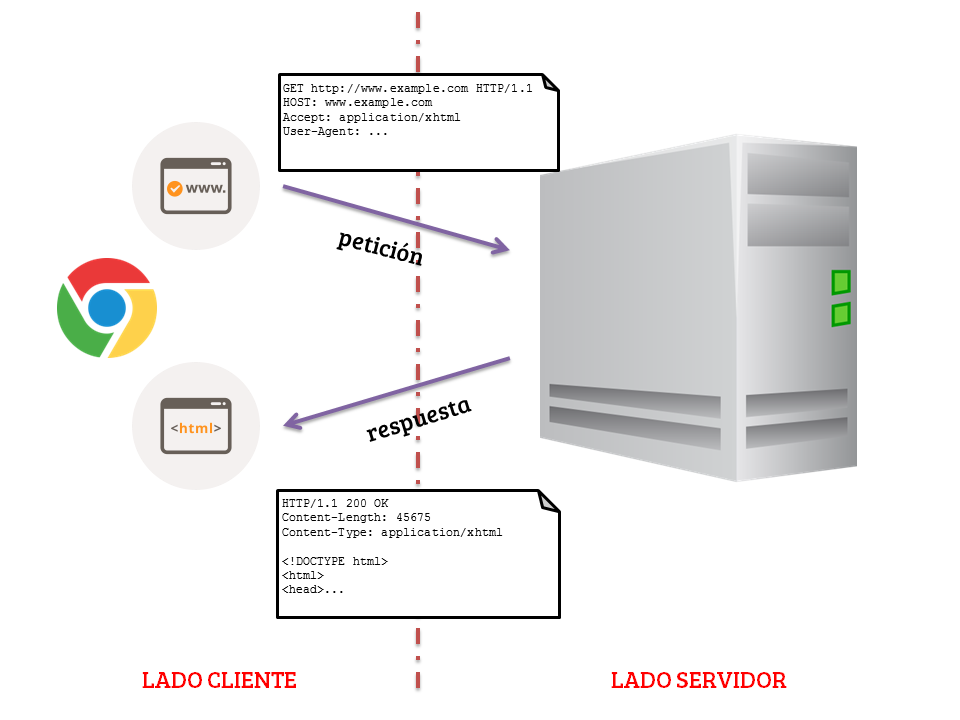
* ObjetoDeNegocio representa los datos del cliente. Es el objeto que requiere el acceso a la fuente de datos para obtener y almacenar datos.
* DAO es el objeto principal de este patrón. Abstrae la implementación del acceso a datos subyacente, y permite un acceso transparente a la fuente de datos.
* OrigenDeDatos representa la implementación de la fuente de datos, que podría ser un SGBDR, un ORM, un repositorio XML, un API REST, …
* ObjetoDelModelo representa el objeto (o colección) que queríamos obtener o modificar.

Spring nos proporciona un amplio soporte para el uso de este patrón dentro de Spring Data.

## **12.1 Introducción**

El desarrollo de aplicaciones web implica tener un conocimiento, al menos elemental, del funcionamiento del protocolo subyacente, HTTP. El Hypertext Transfer Protocol es un protocolo orientado a transacciones y que sigue un esquema de petición-respuesta entre un cliente y un servidor. Normalmente, un cliente (por ejemplo, un navegador web) realiza la petición de un recurso, localizable mediante su URL, frente a un servidor; este, al recibir dicha petición, devolverá una respuesta.

El protocolo no tiene estado. Esto significa que durante una conexión, no tenemos información sobre que ha sucedido en conexiones anteriores. Si un cliente realiza dos veces la misma petición a un servidor, atenderá la segunda petición exactamente igual que la primera. El hecho de no almacenar estados supone ventajas e inconvenientes frente al hecho de sí hacerlo, aunque algunos autores apuntan a que ha sido parte de su éxito. A día de hoy existen mecanismos auxiliares que nos permiten almacenar información entre peticiones (como por ejemplo las cookies ).



Esquema de petición - respuesta de HTTP

## **12.2 URLs ( Unified Resource Locator )**

Una URL es una forma de localizar a un recurso en internet. El formato genérico es el siguiente:

protocolo://maquina:puerto/camino/al/recurso

Usualmente, el protocolo es HTTP o HTTPS, aunque se pueden utilizar otros (como por ejemplo FTP). La máquina es el nombre de la misma o su IP. Habitualmente se utiliza un nombre (por ejemplo, openwebinars.net ), que es traducido a una IP por el servicio DNS. A continuación, se indica el puerto al cual nos queremos conectar; por defecto, HTTP tiene asignado el 80, y HTTPS el 443. Por último, camino/al/recurso es la ruta en el sistema de ficheros de la máquina remota; dicha ruta será relativa al directorio marcado como raiz de la web.

## **12.3 Peticiones y respuetas HTTP**

Una petición HTTP no es más que un mensaje que un cliente enviará a un servidor HTTP, con un formato muy específico. Este mensaje tiene siempre una estructura dividida en dos partes: la cabecera (HEADER) y el cuerpo (BODY).

Dentro de la cabecera de una petición HTTP se envía siempre una serie de datos que son necesarios para que el servidor pueda responder, a saber:

* Tipo de petición: GET, POST, … (los estudiaremos más adelante).
* URL del recurso solicitado
* Versión del protocolo
* Quién realiza la petición (User-Agent)
* ….

Además, podemos añadir a la cabecera algunos datos específicos para una petición determinada, y que el servidor puede procesar para realizar alguna tarea auxiliar (por ejemplo, los datos de autenticación si solicitamos un recurso protegido).

Por otro lado, el cuerpo de la petición puede que esté vacío (o no) dependiendo del tipo de la misma. Un ejemplo de contenido del cuerpo de una petición sería los campos rellenados en un formulario, un fichero adjunto, etc…

### **12.3.1 Métodos de peticiones HTTP**

El estándar del protocolo HTTP define una serie de métodos (también conocidos como verbos), que nos ayudarán a indicar qué estamos pidiendo exactamente al servidor con respecto a un recurso:

* ****GET****: solicita al servidor que envie el recurso identificado por la URL
* ****HEAD****: pide al servidor que envíe una respuesta idéntica a la que enviaría con ****GET****, pero sin el cuerpo de la respuesta.
* ****POST****: envía datos al servidor para que sean procesados por el recurso identificado por la URL. Los datos se deben incluir en el cuerpo de la petición.
* ****PUT****: envía un recurso determinado (un archivo) al servidor. A diferencia que POST, este método crea una nueva conexión (socket) y la emplea para enviar el recurso, lo cual resulta más eficiente que enviarlo dentro del cuerpo del mensaje.
* ****DELETE****: solicita la eliminación el recurso especificado.
* ****TRACE****: solicita al servidor que envíe un mensaje de respuesta. Se utiliza para diagnosticar problemas de conexión.
* ****OPTIONS****: pide al servidor que le indique los métodos HTTP que soporta para una determinada URL.
* ****PATCH****: se emplea para modificar parcialmente un recurso ya existente en el servidor.

Existen varios tipos de petición más (CONNECT, SEARCH, MOVE, MERGE) definidos en sus respectivos RFC, y que quedan fuera del alcance de este curso.

Las peticiones más usuales suelen ser GET y POST, aunque el auge de los servicios web de tipo RESTful, íntimamente ligados al protocolo HTTP, han retomado el uso de otras, como DELETE, PUT o PATCH.

### **12.3.2 Respuestas HTTP**

Una respuesta del servidor tiene una estructura parecida a la de una petición, también con una cabecera y un cuerpo. Como parte de la cabecera, normalmente recibiremos:

* La versión de HTTP con la que trabaja el servidor
* El código de estado o de respuesta, con su correspondiente frase de explicación .
* Algunas cabeceras adicionales, como el tipo de contenido que recibidos, su longitud en bytes, etc…

Como cuerpo de la respuesta, como era de esperar, recibidos el recurso localizado.

El código de estado HTTP de la respuesta se utiliza para explicar que ha sucedido a la hora de realizar la petición. Están formados por tres dígitos, y pueden ser alguno de los siguientes:

* Códigos 1XX: Mensajes
  + 100-111 Conexión rechazada
* ****Códigos 2XX: Operación realizada con éxito****
  + **200 OK**
  + 201-203 Información no oficial
  + 204 Sin contenido
  + 205 Contenido para recargar
  + 206 Contenido parcial
* Códigos 3XX: Redirección
  + 301 Mudado permanentemente
  + 302 Encontrado
  + 303 Vea otros
  + 304 No modificado
  + 305 Utilice un proxy
  + 307 Redirección temporal
* ****Códigos 4XX: Error por parte del cliente****
  + ****400 Solicitud incorrecta****
  + 402 Pago requerido
  + 403 Prohibido
  + ****404 No encontrado****
  + 409 Conflicto
  + 410 Ya no disponible
  + 412 Falló precondición
* ****Códigos 5xx: Error del servidor****
  + ****500 Error interno****
  + 501 No implementado
  + 502 Pasarela incorrecta
  + 503 Servicio no disponible
  + 504 Tiempo de espera de la pasarela agotado
  + 505 Versión de HTTP no soportada

Hemos resaltado los códigos de respuesta más usuales que podemos encontrar en el día a día con HTTP.

## **12.4 Lenguajes, tecnologías y estándares de la WWW**

Como indicábamos más arriba, HTTP es un protocolo de transferencia de hipertexto. Pero, ¿qué es eso del hipertexto? No es más que documentos de texto que pueden hacer referencia a otros documentos de texto. A día de hoy los conocemos como páginas web . La estructura de estos documentos viene definida en el ya archiconocido lenguaje HTML, si bien no es el único utilizado a día de hoy para crear contenido en la web.

* ****HTML****: Lenguaje de marcado de hipertexto. Sirve para la definición de la estructura y el contenido de documentos de hipertexto. Al ser un lenguaje de marcado, se puede escribir en texto plano y no necesita transformarse en código objeto ni nada por el estilo. Actualmente está vigente la versión 5.
* ****CSS****: lenguaje de estilos que permite, en interacción con HTML, modificar la forma en la que se presenta el contenido definido dentro de un documento web . La versión actual es la 3.0 y permite funcionalidades increibles, como gradientes, animaciones, transformaciones…
* ****Javascript****: es un lenguaje de scripting diseñado para usarse dentro de las páginas web. Inicialmente, nos permitía interaccionar con el Modelo de Objetos del Documento (DOM), si bien se ha convertido en una tecnología de mayor importancia y envergadura de lo que sus creadores soñaron inicialmente.
* ****XML****: se trata de un metalenguaje (un lenguaje para definir lenguajes) que nos permite estructurar información mediante un lenguaje de marcado, lo cual la hace interoperable entre sistemas heterogéneos.
* ****AJAX****: La unión de XML y las peticiones asíncronas con Javascript (AJAX) han revolucionado la web hasta transformarla en la conocida como Web 2.0, permitiendo crear servicios como Google Maps, Facebook, …

Todos estos lenguajes y tecnologías que hemos descrito anteriormente son de los que, normalmente, se utilizan en el conocido como lado cliente. Al otro lado, en el servidor, también existen una serie de tecnologías muy utilizadas en la actualidad y que también deben sonarnos .

* ****Java****: esta tecnología es omnipresente, y el campo de la web no podía ser menos. Java comenzó a mitad de los 90s con la incursión de los applets , pequeños programas embebibles dentro de una página web, y que permitieron crear las primeras aplicaciones web. Posteriormente, surgieron los Servlets y la tecnología JSP , que permitió la creación de aplicaciones web empresariales orientadas a la web.
* ****PHP****: Se trata de un popular lenguaje de programación que surgió para la creación de webs de contenido dinámico. Se trata de uno de los lenguajes más flexibles y potentes conocidos a día de hoy. Algunos de los servicios más utilizados del mundo, como Facebook, están implementados con él.
* ****Python****: Es otro popular lenguaje de programación multiparadigma y de código abierto. Tiene una gran cantidad de librerías que lo hacen un gran candidato para casi cualquier proyecto. En el caso de la web, el framework más utilizado es Django.
* ****ASP.NET****: Microsoft también tiene su tecnología del lado del servidor, llamada ASP.NET. Se trata de un framework para el desarrollo de aplicaciones web. Uno de los más extendidos es el que implementa aplicaciones a través del patrón MVC.
* ****Ruby****: se trata de un lenguaje de programación multiparadigma que se creó para que fuera altamente sencillo de programar, siguiendo el “principio de la menor sorpresa”. El framework más utilizado para el desarrollo web con este lengauje es Ruby on Rails.

## **13.1 Software a utilizar**

* Java SDK 1.7 o superior (preferiblemente 1.8)
* Spring Tool Suite 3.8 (la última versión liberada hasta ahora 3.8.1)
* Maven 2

## **13.2 Implementación del proyecto**

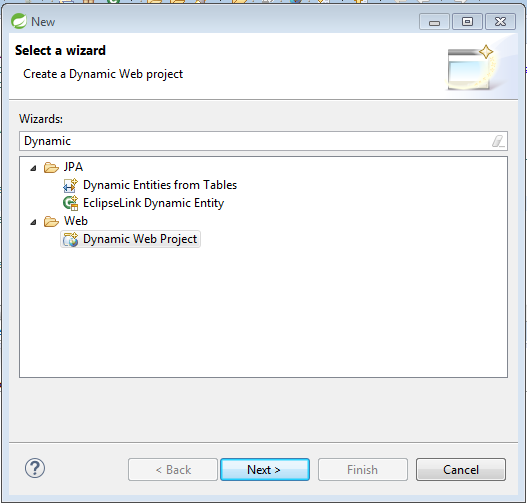
### **13.2.1 Paso 1: Aplicación base y configuración del entorno**

Para la creación de nuestro proyectos utilizaremos Maven, si bien antes de nada tendremos que matizar un poco esto.

Lo ideal sería que pudiéramos utilizar Maven como en el primer bloque del curso, es decir, utilizando un arquetipo para comenzar nuestra aplicación web. De hecho, en su catálogooficial , Maven ofrece uno, llamado maven-archetype-webapp , que nos permitiría comenzar a trabajar. Sin embargo, este arquetipo incluye la limitación de que la versión de servlets que soporta por defecto es la 2.3 (bastante antigua). Si tratamos de cambiar esto para poder usar una versión posterior (al menos la 2.5, o idealmente la 3.0 o 3.1), podemos comprobar como entramos en un callejón sin salida. Por ello, afrontaremos los pasos iniciales mediante una alternativa, que nos permitirá usar Maven y la versión de servlets que queramos.

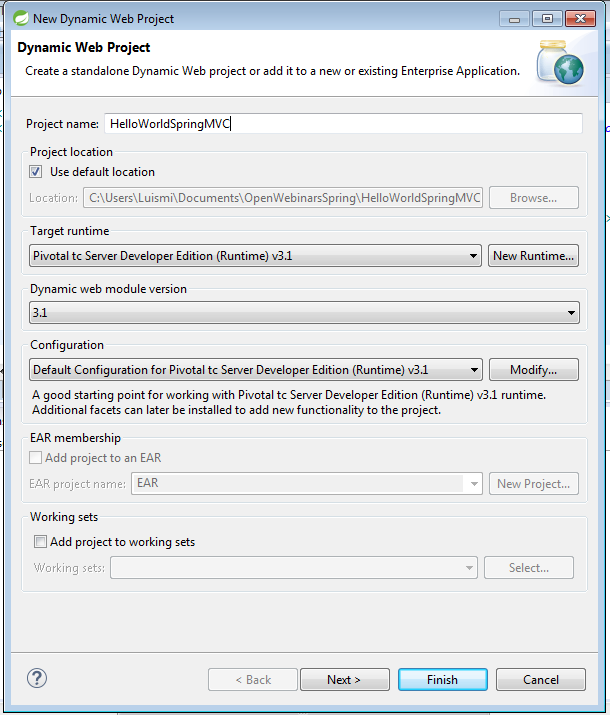
#### **13.2.1.1. Paso 1.1: Creación del proyecto**

Para comenzar, vamos a crear un nuevo Dynamic Web Project . Se trata de un tipo de proyecto, proporcionado por STS (Eclipse), y orientado a la creación de una aplicación web dinámica con JavaEE.



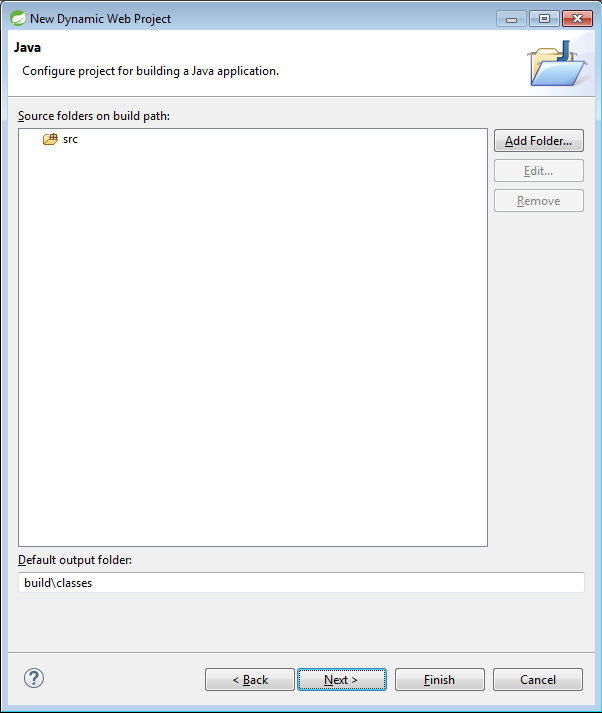
Creación de un Dynamic Web Project

Tras seleccionarlo, pulsamos Next , y en la siguiente ventana del asistente, tan escribimos el nombre del proyecto ( HelloWorldSpringMVC o el nombre que deseemos). Seleccionamos también el servidor donde se ejecutará la aplicación (por defecto, la versión 3.8 de STS incluye una instancia del servidor Pivotal tc Server , que no es más que un Tomcat tuneado ). Para la versión del módulo dinámico web ( Dynamic Web module version ) elegimos 3.1 (esto nos permitirá usar la versión 3.1 de los servlets ).



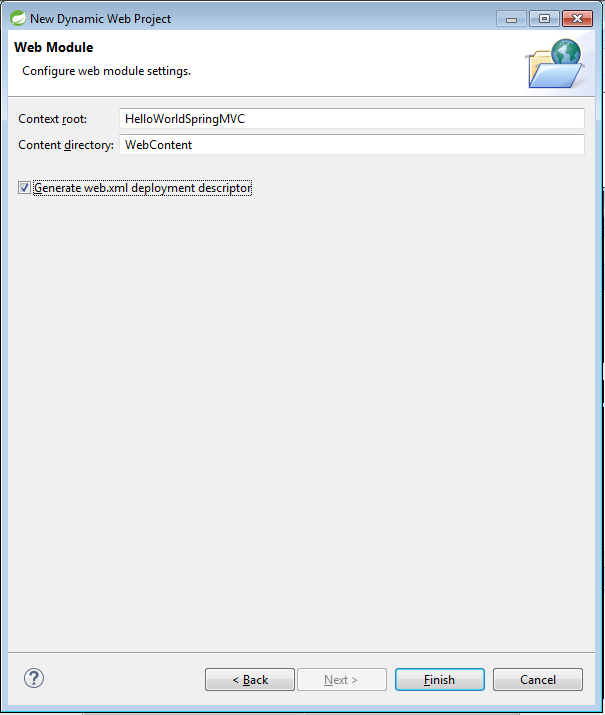
Datos iniciales del proyecto

En el siguiente paso, no tenemos que hacer nada, y tan solo pulsamos Next .



Configuración de las carpetas que se incluirán en el Build Path .

Por último, en la ventana final del asistente, marcamos la opción de generación del descriptor del proyecto ( Generate web.xml deployment descriptor ) y pulsamos Finish .



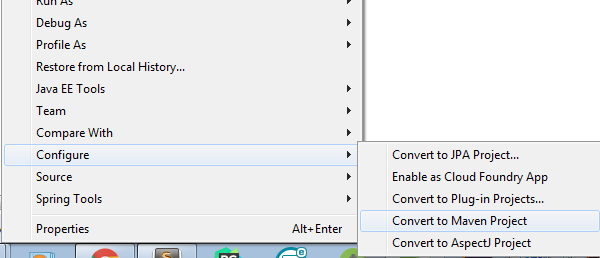
Generación del proyecto incluyendo el descriptor de despliegue.

Al tratarse de un proyecto Java EE, nos invitará a cambiar la perspectiva de Eclipse (la forma en que se organizan las ventanas, cuantas se visualizan inicialmente…), pero podemos declinar dicha invitación y continuar con la perspectiva Spring.

#### **13.2.1.2 Paso 1.2: Conversión del Dynamic Web Project a un proyectoMaven**

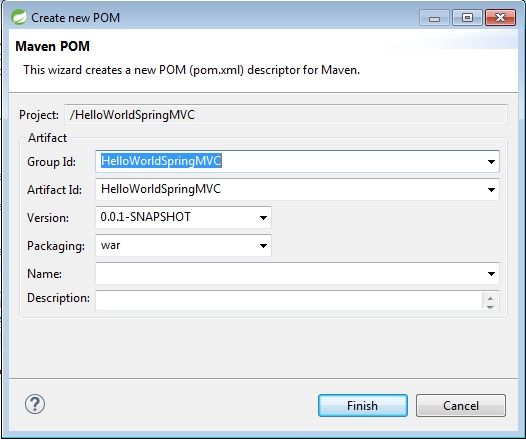
Por defecto, un proyecto de tipo Dynamic Web Project no está preparado para poder usar Maven. Sin embargo, el entorno de desarrolo nos permite transformar un proyecto de cualquier tipo en un proyecto Maven, de forma que modifica su estructura en todos los elementos necesarios, y prepara un fichero pom.xml para poder añadir las dependencias necesarias.

Para realizar la transformación, pulsamos con el botón derecho sobre el proyecto, y seleccionamosConfigure > Convert to Maven project .



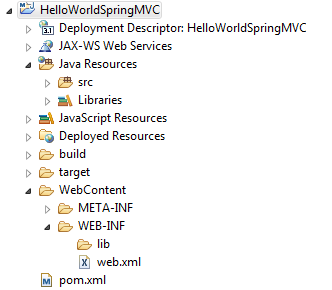
Conversión a un proyecto Maven.

En la siguiente ventana, no tenemos que modificar ningún dato, y tan solo pulsamos Finish .



Finalización de la conversión del proyecto Maven.

Tras finalizar la conversión, tenemos la siguiente estructura de proyecto:



Estructura del proyecto

#### **13.2.1.3 Paso 1.3: Añadimos las dependencias**

Como ya hiciéramos en el bloque pasado, vamos a trabajar con una serie de dependencias para incorporar las librerías necesarias. Maven es muy generoso, y podemos hacerlo todo a través de una sola dependencia, que es la siguiente:

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>${org.springframework.version}</version></dependency>

Si echamos un vistazo en el apartado <version> , podemos observar que, en lugar de un valor concreto, estamos usando una variable. Esto nos servirá en el caso de que utilicemos más de una dependencia Spring, para que siempre utilicemos la misma versión; y también, para que en caso de querer modificar la versión, podamos hacerlo en un lugar centralizado , y dicho cambio se propague a todas las dependencias. La definición de la propiedad se realiza fuera del elemento<dependencies> , de la siguiente forma:

<properties>

<org.springframework.version>4.3.2.RELEASE</org.springframework.version>

</properties>

También vamos a usar los servlets; de hecho, en el paso anterior, elegimos la versión 3.1. Esa es la versión que tendremos que añadir como dependencia:

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<version>3.1.0</version></dependency>

Si además queremos usar la Java Standard Tag Library , es decir, el conjunto oficial de etiquetas para JSP, tenemos que añadir su dependencia:

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version></dependency>

Por último, si queremos utilizar Log4j como sistema de logging, añadimos la dependencia correspondiente:

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.17</version></dependency>

### **13.2.2 Paso 2: Configuración de Spring MVC**

Si bien profundizaremos en la estructura de Spring MVC en las lecciones posteriores, si podemos decir que un proyecto de Spring MVC no es más que un proyecto Java EE.

#### **13.2.2.1 Paso 2.1: Configuración del descriptor de despliegue**

Como proyecto Java EE que es, necesita una configuración de los servlets que vamos a utilizar (en nuestro caso, solamente uno, que se encargará de despachar todas las peticiones de la aplicación, a modo de Front Controller ). Todo ello lo haremos en el fichero Webcontent/WEB-INF/web.xml :

<?xml version="1.0" encoding="UTF-8"?><web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd" id="WebApp\_ID" version="3.1">

<display-name>HelloWorldSpringMVC</display-name>

<servlet>

<servlet-name>dispatcher</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>dispatcher</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping></web-app>

De esta forma, como decíamos anteriormente, indicamos que todas las peticiones sean atendidas por el controlador frontal.

#### **13.2.2.2 Paso 2.2: Configuración del contexto**

Spring MVC necesita una definición de su contexto, y por defecto, debe estar un fichero que se llame <nombredelservlet>-servlet.xml . Es decir, si el nombre de nuestro servlet esdispatcher , el fichero de configuración, que ubicaremos en el mismo directorio, debe llamarsedispatcher-servlet.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"

xmlns:mvc="http://www.springframework.org/schema/mvc"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.3.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.3.xsd

http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd">

<context:component-scan base-package="com.openwebinars.helloworld" />

<mvc:annotation-driven />

<bean id="viewResolver"

class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="viewClass"

value="org.springframework.web.servlet.view.JstlView" />

<property name="prefix" value="/WEB-INF/jsp/" />

<property name="suffix" value=".jsp" />

</bean>

</beans>

#### **13.2.2.3 Paso 2.3: Configuración de Log4j**

Como en otras ocasiones, vamos a configurar Log4j para utilizarlo como sistema de logging. Para ubicar el fichero de properties, tenemos que crear una nueva carpeta, llamada resources , dentro de Java Resources . Para ello, tenemos que hacerlo desde la configuración del Build Path(botón derecho sobre el proyecto > Propiedades > Java Build Path).

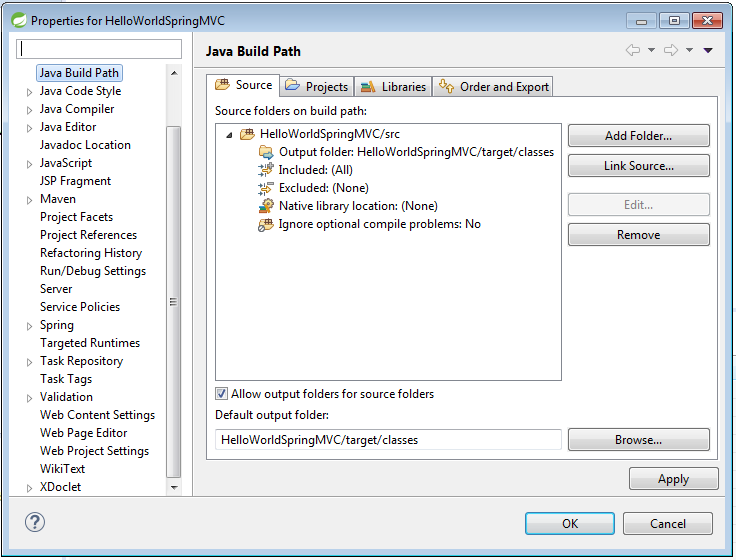
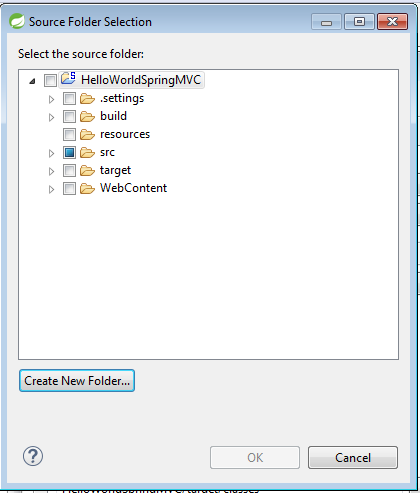
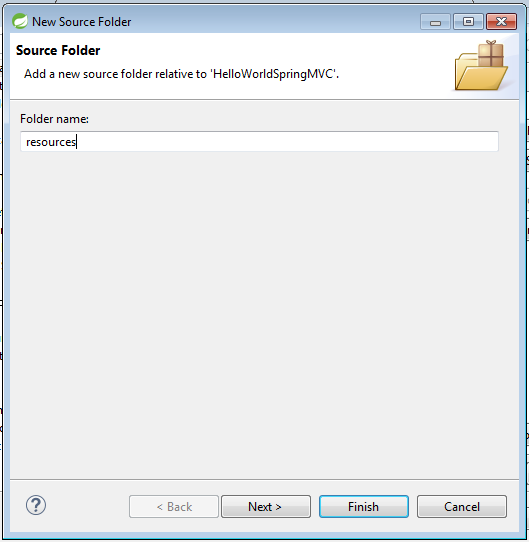


Figura XX. Configuración del Build Path .

Pulsamos sobre el botón Add Folder , y en diálogo que nos aparece, pulsamos sobre el botón Create New Folder . Damos como nombre a la carpeta resources .   
Pulsamos Finish , Ok , …



Creación de la carpeta (1)



Creación de la carpeta (2)

Ahora, dentro de esta carpeta, es donde podemos ubicar el fichero de propiedades de log4j, llamado log4j.properties , y cuyo contenido puede ser el siguiente:

log4j.rootCategory=INFO, stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %t %c{2}:%L - %m%n

log4j.category.org.springframework.beans.factory=DEBUG

A partir de ahora, vamos a definir nuestro primer controlador y una vista a mostrar.

### **13.2.3 Paso 3: La vista y el controlador**

#### **13.2.3.1 Paso 3.1: El controlador, una clase muy simple**

Spring nos facilita mucho la creación de controladores, que podrán realizar toda la lógica de negocio de nuestra aplicación antes de enviar los datos a la vista. De hecho, como podremos comprobar, se tratará de una clase POJO ( Plain Old Java Object ), que no debe heredar de ninguna clase especial. Creamos una nueva clase, llamada HelloController , dentro del paquetecom.openwebinars.helloworld . Veamos el código y desgranémoslo:

package com.openwebinars.helloworld;

import org.springframework.stereotype.Controller;import org.springframework.ui.Model;import org.springframework.web.bind.annotation.RequestMapping;

@Controllerpublic class HelloController {

@RequestMapping("/hello")

public String sayHello(Model model) {

model.addAttribute("saludo", "Hola Mundo!!!");

model.addAttribute("mensaje","Me llena de orgullo y satisfacción saludaros en este primer ejemplo de Spring MVC");

model.addAttribute("url", "http://www.openwebinars.net");

return "hello";

}

}

Lo primero que llama la atención es la simplicidad de la clase: es java en estado puro. Se trata una simple clase java, anotada con @Controller . Eso, unido a la configuración declarada en el ficherodispatcher-servlet.xml , harán que Spring se encargue del resto.

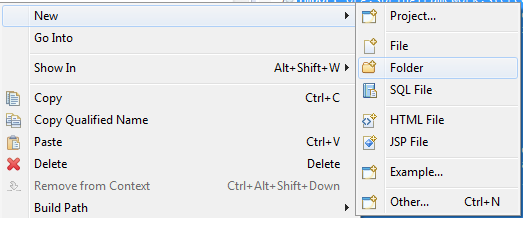
La clase tiene un solo método, cuya firma es public String sayHello(Model model) :

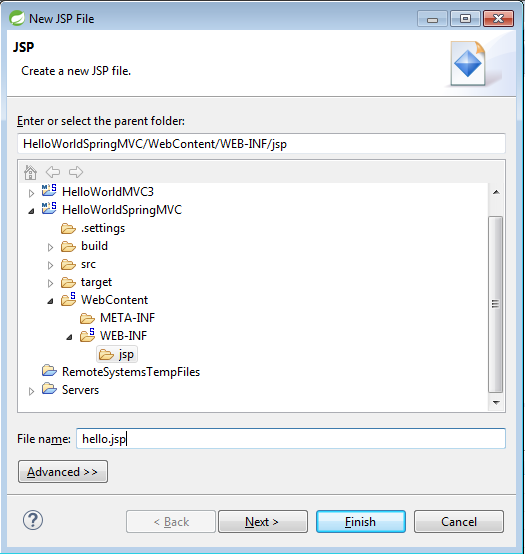
* Como decíamos más arriba, el tipo de retorno es String , ya que devolverá el nombre de una vista.
* Como argumento, recibe un elemento de tipo Model . Sera el contenedor que nos permita añadir datos que serán transportados a la vista.
* Además, está anotado con RequestMapping("/hello") . De esta forma indicamos que cualquier petición hacia la url http://máquina:puerto/HelloWorldSpringMVC/helloserá procesada por este método.

Ahora, nos toca crear la vista.

#### **13.2.3.2 Paso 3.2: Creación de la vista**

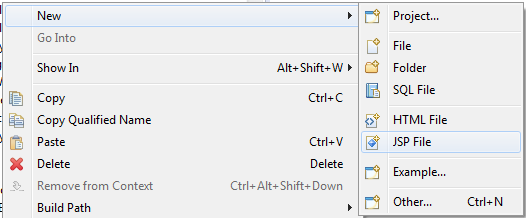
Para crear la vista, vamos a crear previamente una carpeta que las almacene todas, dentro de WEB-INF , y que se llamará jsp . (También podemos eliminar la carpeta WEB-INF/lib , ya que al usar Maven, no sirve para nada).

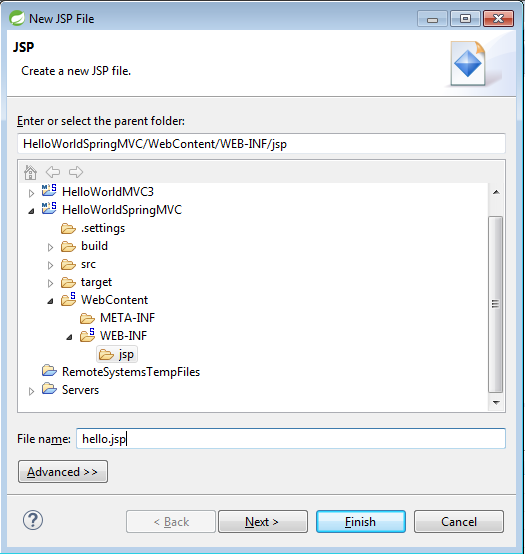




Creación de la carpeta WEB-INF/jsp, que almacenará todas las vistas.

Ahora, ya podemos proceder a la creación de nuestro fichero de vista, que se llamará hello.jsp .





Creación del fichero JSP

Inicialmente, la plantilla que tiene definida STS para una página JSP está pensada para usar HTML4. Vamos a realizar algunos cambios:

1. Cambiaremos la codificación de ISO-8859-1 a UTF-8.
2. Vamos a usar HTML5 en lugar de HTML4.
3. Además, usaremos como base bootstrap, lo cual nos permitirá tener una web de apariencia profesional y responsiva con muy poco esfuerzo por nuestra parte.

Abrimos el navegador, y entramos en http://getbootstrap.com . Accedemos a Getting startedy encontramos la plantilla base. Modificamos los enlaces de bootstrap, para no tener que descargar los ficheros y que los tome directamente el respositorio oficial:

<!DOCTYPE html><html>

<head>

<meta charset="utf-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content="width=device-width, initial-scale=1">

<!-- The above 3 meta tags \*must\* come first in the head; any other head content must come \*after\* these tags -->

<title>Bootstrap 101 Template</title>

<!-- Bootstrap -->

<link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" rel="stylesheet">

<!-- HTML5 shim and Respond.js for IE8 support of HTML5 elements and media queries -->

<!-- WARNING: Respond.js doesn't work if you view the page via file:// -->

<!--[if lt IE 9]>

<script src="https://oss.maxcdn.com/html5shiv/3.7.3/html5shiv.min.js"></script>

<script src="https://oss.maxcdn.com/respond/1.4.2/respond.min.js"></script>

<![endif]-->

</head>

<body>

<!-- jQuery (necessary for Bootstrap's JavaScript plugins) -->

<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.4/jquery.min.js"></script>

<!-- Include all compiled plugins (below), or include individual files as needed -->

<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></script>

</body></html>

Ahora, vamos a definir la información que queremos visualizar. Si revisamos el código del controlador, hemos añadido al modelo:

* Un saludo, que mostraremos en grande.
* Un mensaje, que mostraremos debajo.
* Un enlace, que usaremos para que los usuarios puedan visitar otra web.

Podemos utilizar una de las plantillas base de bootstrap para copiar el código, que es bastante fácil:

<div class="jumbotron">

<div class="container">

<h1>${saludo}</h1>

<p>${mensaje}</p>

<p><a class="btn btn-primary btn-lg" target="\_blank" href="${url}" role="button">Learn more &raquo;</a></p>

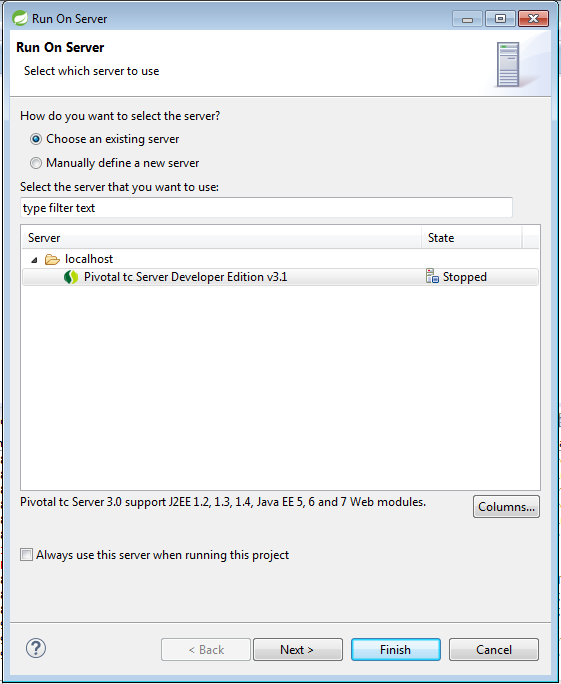
</div>

</div>

Además, encontramos que hemos usado Expression Language , una característica que JSP que nos permite utilizar un lenguaje sencillo para usar un JavaBean dentro de una página JSP. ${saludo}buscará dentro del contexto una variable llamada saludo , y mostrará su valor. Lo mismo sucede con ${mensaje} . Como vemos, ${url} nos sirve para definir el elemento href del enlace.

### **13.2.4 Paso 4: Ejecución**

Una vez que tenemos definido nuestro proyecto, lo hemos configurado, y hemos añadido una vista y un controlador, vamos a pasar a ejecutarlo. Para ello, tan solo tenemos que pulsar sobre el proyecto con el botón derecho > Run As > Run on Server . Ello hará que nos aparezca el siguiente diálogo:



Diálogo de ejecución del proyecto

Tan solo pulsamos sobre Finish , y el servidor comenzará a ejecutarse, desplegandose nuestra aplicación y lanzando el navegador que esté configurado. Y ¡sorpresa! Nos encontramos con el error 404. Esto se debe a que no hemos definido una página de bienvenida. Sin embargo, si recordamos, nuestro controlador iba a escuchar la urlhttp://maquina:puerto/nombredelproyecto/hello . Si la escribimos en el navegador, veremos que aparece nuestra esperada vista.