FRAMEWORK

Introducción

Módulo I

harriague+asociados

Emilio Rey | María Agustina Aliaga

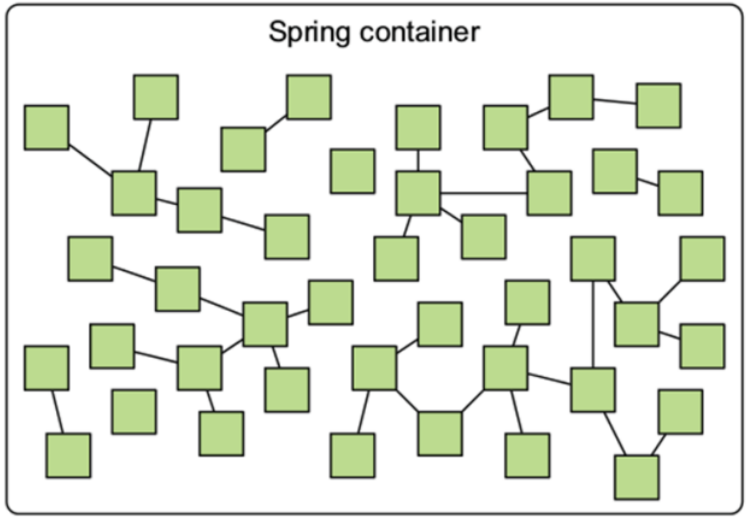
INCLUIT
empowering your business

Qué es Spring

Spring es un **framework** para el desarrollo de aplicaciones.

Nos referimos a Spring como una especie de **contenedor**, donde viven los objetos de la aplicación.

El framework se encarga de crear objetos, configurarlos y gestionar las dependencias dentro de la aplicación.



IoC Container

El núcleo de Spring es el **Contenedor de Inversión de Control**, que es altamente configurable.

Su trabajo es:

- **instanciar**
- **inicializar**
- **conectar**

entre ellos a los **objetos** de la aplicación.

Los componentes que son instanciados por el contenedor se llaman Beans.

A través del contenedor IoC es posible **configurar**:

- El número de **instancias** de un componente.
- Si el componente es **singleton** o no.
- En qué **momento** el componente es **creado** o **destruído** en la memoria.

Inicializar un **bean** con Spring, es el equivalente de utilizar la palabra clave **new** para crear un objeto en Java.

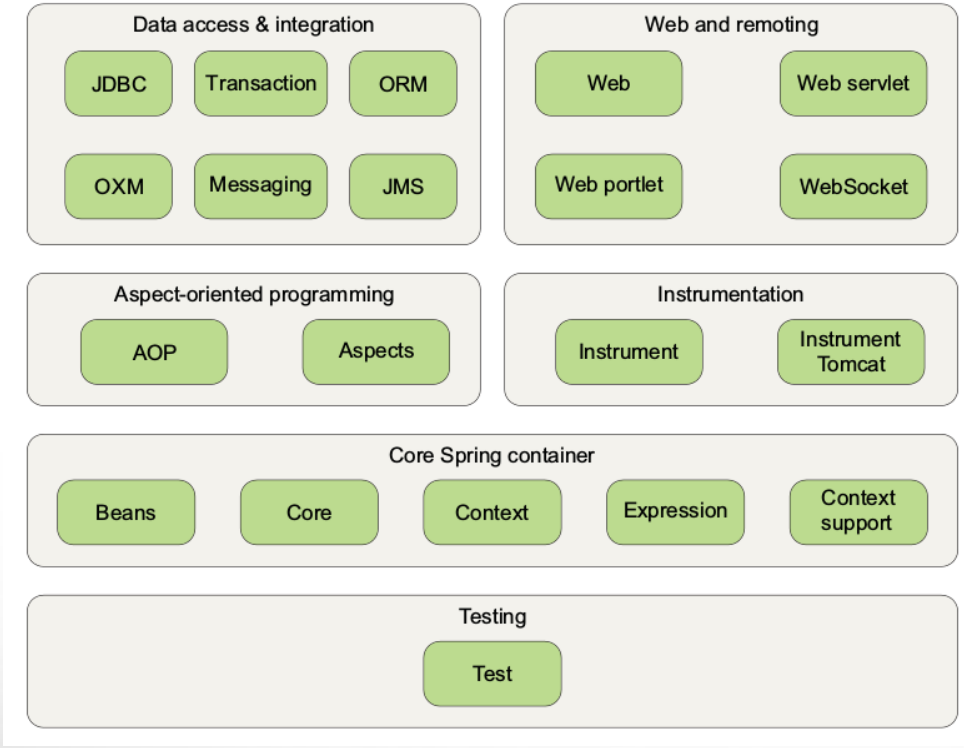
Spring provee dos tipos de **implementaciones** del contenedor:

- **Bean Factory: Es el contenedor más simple, permite la DI,** definidas por la interfaz org.springframework.beans.factory.BeanFactory aún se encuentra presente en Spring a los fines de compatibilidad.

- **Application Context:** definida por la sub-interfaz de Bean Factory org.springframework.context.ApplicationContext

Para enlazar los Beans, el contenedor utiliza la metadata que se encuentra en el archivo de configuración XML o en Anotaciones.

La arquitectura modular de Spring



Spring está compuesto por 20 módulos empaquetados en 20 archivos JAR.

Los módulos más importantes

Módulo	Funcionalidad
AOP	Clases necesarias para utilizar las funcionalidades de **programación orientada** a **aspectos** dentro de Spring, y transaction management declarativa.
aspects	Clases para integración avanzada con la librería de **AspectJ**.
beans	Clases que permiten **manipular los beans**. Aquí están contenidas las clases requeridas para procesar los archivos XML de configuración de Spring y las anotaciones de Java.
context	Clases que proveen muchas **extensiones al núcleo de Spring**, por ejemplo las que necesitan usar la funcionalidad del **Application Context.**
core	Este es el módulo necesario para cada aplicación de Spring, contiene **clases que son compartidas por otros módulos y clases utilitarias.** Aquí se encuentra la **Bean Factory y el Application Context**, es decir la parte del framework que permite la Inyección de Dependencias (DI) y la Inversión de Control (IoC). Todos los módulos de Spring se construyen usando el core container como base.
ASM	Es un framework de **manipulación de bytecode**. Spring depende de esta librería para analizar el bytecode de los beans, modificarlos dinámicamente, y generar nuevo bytecode en tiempo de ejecución. www.asm.ow2.org

Módulo	Funcionalidad
test	Spring provee un **conjunto de clases de mock para facilitar el testing** de las aplicaciones. Es posible mockear **HttpServletRequest** y **HttpServletResponse** en unit tests de aplicaciones web. Spring se encuentra integrado al framework Junit. Por ejemplo **SpringJUnit4ClassRunner** provee una manera simple de utilizar el Spring ApplicationContext en un ambiente de test unitario.
web	Clases para **utilizar Spring en las aplicaciones web**, incluyendo las clases para lanzar un ApplicationContext automáticamente. El modelo **MVC** (Model-View-Controller) es un abordaje ampliamente aceptado para construir aplicaciones web separando la interfaz de usuario de la lógica. Spring también permite exponer y consumir **REST** APIs.
JDBC y ORM	**Clases que ofrecen soporte a JDBC.** Es necesario para las aplicaciones que requieren acceso a BD. Trabajar con JDBC muchas veces supone una gran cantidad de **"boilerplate code"** (que debe repetirse muchas veces) para crear la conexión, un statement, procesar un result set, y cerrar la conexión. Spring JDBC y los módulos de Data Access permiten mantener el código limpio y previene los problemas que pueden resultar de transacciones fallidas. También crea una **capa de excepciones** basadas en los mensajes de error que devuelven varios servidores de BD. **ORM** extiende la funcionalidad estándar JDBC para permitir el uso de tools ORM como **Hibernate, JPA**, etc.

Principio de Inversión de Control (IoC)

IoC es un **principio** de la Programación Orientada a Objetos que **externaliza:**

- **creación**
- **gestión**

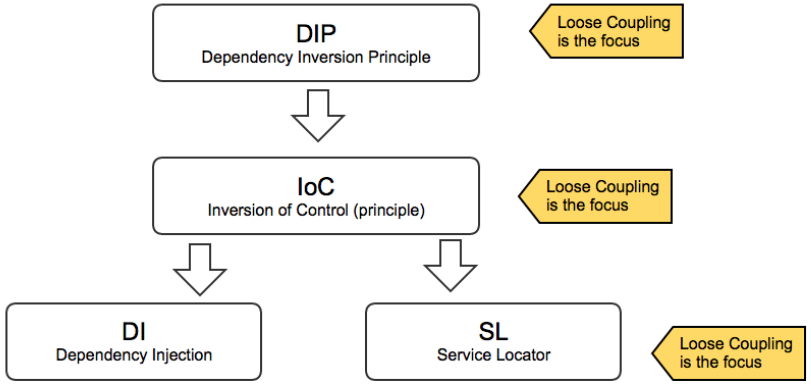
de las **dependencias** entre los componentes.

La IoC puede implementarse mediante el uso de diferentes **patrones de diseño**, como por ejemplo Strategy, Service Locator, Factory, y Dependency Injection.

En Spring el **contenedor IoC gestiona la instanciación y destrucción de los objetos**, y nos permite su configuración de una manera **centralizada**.

Las dependencias de los Beans, **los eventos del ciclo de vida y la configuración, residen fuera de los componentes.**

De esta manera se refuerza el patrón de inyección de dependencias y los componentes se encuentran desacoplados (loosely coupled).



Inyección de Dependencias (DI)

La Inyección de Dependencias, es un **patrón de diseño a través del que implementamos la Inversión de Control** y que consiste en **inyectar comportamientos** a los componentes.

Es la **Fábrica** quien **instancia** los objetos requeridos por una aplicación y los **inyecta** en sus objetos dependientes **en vez de hacerlo los objetos por sí mismos**.

De esta manera se extraen las **responsabilidades** de un componente para delegarlas a otros.

En el caso de Spring, los componentes se codifican contra la interfaz, pero **el contendor IoC gestiona la instanciación de los objetos de manera que la aplicación no deba hacerlo**, removiendo las dependencias con clases de más bajo nivel.

Tipos de Inyección de Dependencias

Tipo	Implementación
Constructor	Los argumentos del constructor son inyectados durante la instanciación de las clases.
Setter	Las dependencias son seteadas en los objetos a través de los métodos setter definidos en el archivo de configuración de Spring.
Field-based	Las dependencias pueden ser inyectadas marcándolas con la anotación @Autowired

Ejemplo de Inyección de dependencias por Constructor

```
public class ConstructorMessage {

    private String message = null;

    /**
     * Constructor
     */
    public ConstructorMessage(String message) {
        this.message = message;
    }

    /**
     * Gets message.
     */
    public String getMessage() {
        return message;
    }
```

```
<?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="message"
          class="org.springbyexample.di.xml.ConstructorMessage">
        <constructor-arg value="Spring is fun." />
    </bean>

</beans>
```

Tenemos una **clase** llamada **ConstructorMessage**.

En el **constructor** de la clase ConstructorMessage se le pasa como **argumento** un **String message**.

En el **context.xml** indicamos un nuevo bean con el **id=message**, y el elemento **constructor-arg** inyecta el mensaje en el bean utilizando el valor del atributo **value**, en este caso un mensaje que dice «Spring is fun»

Ejemplo de Inyección de dependencias por Setter

```
public class SetterMessage {

    private String message = null;

    /**
     * Gets message.
     */
    public String getMessage() {
        return message;
    }

    /**
     * Sets message.
     */
    public void setMessage(String message) {
        this.message = message;
    }

}
```

```
<?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="message"
          class="org.springbyexample.di.xml.SetterMessage">
        <property name="message" value="Spring is fun." />
    </bean>

</beans>
```

En este caso, en la **clase SetterMessage**, tenemos un método setter.

En el context.xml indicamos un nuevo Bean con el **id=message**, y dentro del mismo señalamos la **clase** que se corresponde con dicho Bean, así como el **property name**, y su **value** «Spring is fun»

Beans

Cualquier **recurso gestionado por el contenedor IoC** de Spring es llamado Bean.

Estos beans son **creados con la metadata del archivo de configuración** que le proporcionamos al contenedor.

```
<!-- A simple bean definition -->
<bean id = "..." class = "...">
   <!-- collaborators and configuration for this bean go here -->
</bean>

<!-- A bean definition with lazy init set on -->
<bean id = "..." class = "..." lazy-init = "true">
   <!-- collaborators and configuration for this bean go here -->
</bean>

<!-- A bean definition with initialization method -->
<bean id = "..." class = "..." init-method = "...">
   <!-- collaborators and configuration for this bean go here -->
</bean>

<!-- A bean definition with destruction method -->
<bean id = "..." class = "..." destroy-method = "...">
   <!-- collaborators and configuration for this bean go here -->
</bean>
```

Atributo	Descripción
class	Es imprescindible y **especifica la clase a la que hace referencia** el bean.
name/id	**Identifica al bean individualmente**. Se utiliza el atributo id o name. Spring utilizará para reconocer el bean. **Dentro de un ApplicationContext** cada bean debe tener al menos un nombre que es **único**.
scope	Especifica el **ámbito o alcance** de los objetos creados a partir de una definición de un bean. (**Default: Singleton**)
constructor-arg	Es utilizado para **inyectar** las dependencias a través del **constructor**.
properties	Utilizado para **inyectar** las dependencias a través de **setters**.
autowiring mode	Utilizado para **inyectar** las dependencias a través de **anotaciones**.
lazy-initialization mode	El IoC **creará una instancia de ese bean cuando es requerido por primera vez** en vez de instanciarlo en el momento en que se inicia la aplicación.

Instanciación de los Beans

Todos los Beans son **creados como instancias Singleton por defecto** y Spring utiliza las mismas instancias para satisfacer las llamadas a ese bean.

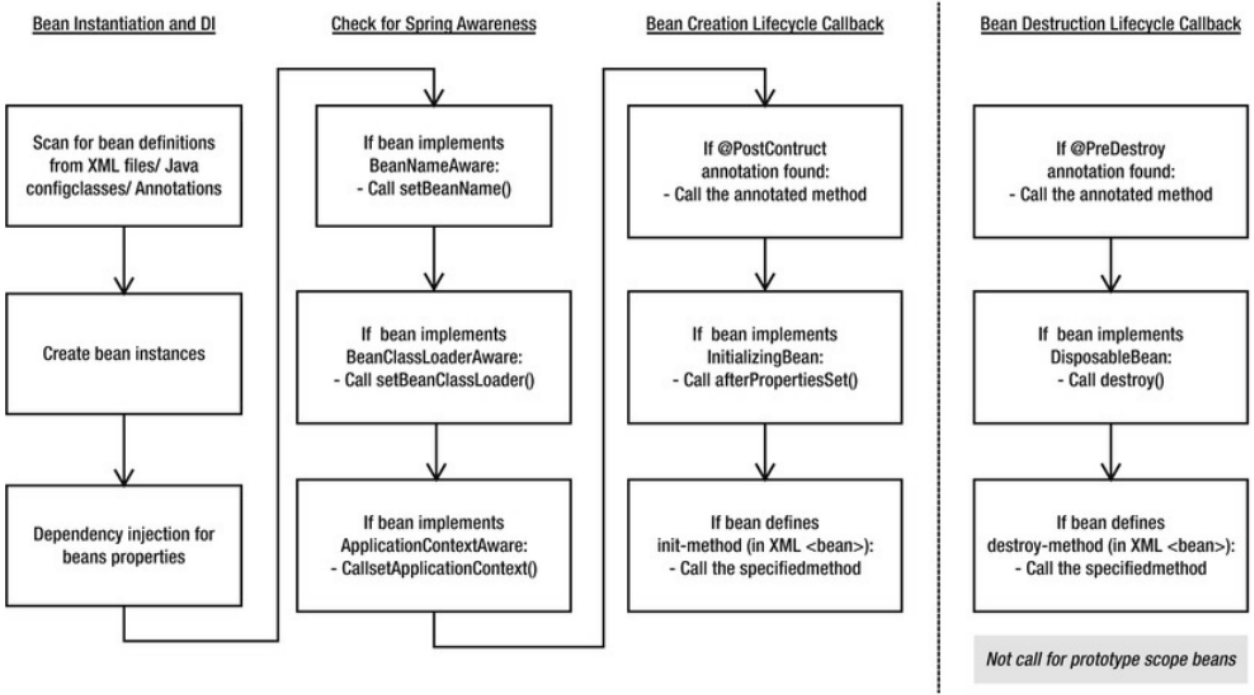
Una **clase singleton** garantiza que **sólo será instanciada una vez.** Usualmente, una clase Singleton tiene un método estático que retorna una instancia de la clase y un constructor privado por lo que sólo él mismo puede instanciarse.

Es posible **configurar** un Bean para que **sea instanciado de otra manera**, esto puede ser necesario por ejemplo en los casos de acceso **multihilo** o en casos de **sesiones** web.

Scope/Ámbito de los Beans

Scope	Definición
Singleton	El contenedor de Spring **retorna una sola instancia.** (Valor por defecto).
Prototype	El contenedor retorna una **nueva instancia cada vez que es llamado**.
Request	El contenedor retorna una **nueva instancia en cada HTTP Request**.
Session	El contenedor retorna una **nueva instancia en cada HTTP session**.
Global	El contenedor retorna **una sola instancia por sesión HTTP global.**

Ciclo de vida de los Beans

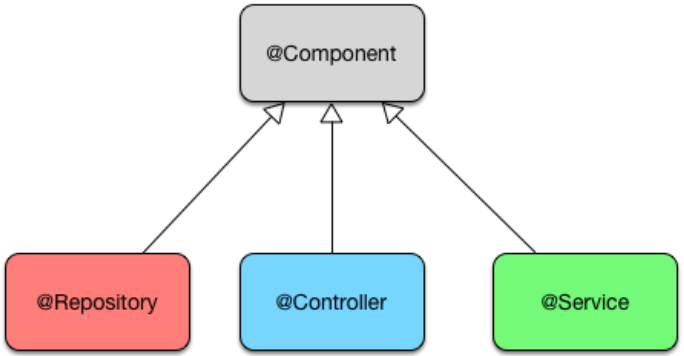


Los Beans pasan por **diferentes etapas** desde su **creación** hasta su **destrucción** en el contenedor de Spring.

Cada paso es una oportunidad para **customizar** cómo va a **gestionarse** ese bean dentro de Spring.

Usando @Anotaciones

Llamamos **estereotipos** de Spring a las anotaciones que permiten **categorizar a los componentes**, asociándoles una **responsabilidad** concreta.



Las anotaciones @Repository, @Controller y @Service son **especializaciones** del marcador genérico @Component.

Estereotipo	Función
@Component	Es el **marcador genérico** que Spring va a reconocer como un componente que debe gestionar.
@Repository	Está asociado a un **objeto de acceso a datos o DAO.** Las clases anotadas de esta manera pueden ser procesadas por otras herramientas o aspectos dentro del contenedor de Spring.
@Controller	Se utiliza para marcar las clases que realicen **tareas de controlador**, dentro un **contexto web.**
@Service	Se utiliza para marcar las clases que formen parte de la **capa de servicios.**

@Autowired

Mediante la anotación @Autowired Spring permite **controlar** dentro del código de una clase, **dónde se inyectan las dependencias.**

- **En el constructor de una clase:** se inyecta la dependencia en el momento en que el bean es creado, cumple la misma función que el tag <constructor-arg> en el archivo XML de configuración.

```
@Controller
public class MyController {

    private final MyBean myBean;

    @Autowired
    public MyController(MyBean myBean) {

        this.myBean = myBean;
    }
}
```

- **En un método setter:** Podemos obviar el tag <property> en el archivo XML de configuración. El método setter es llamado con una instancia de MyBean cuando MyController es creado.

```
@Controller
public class MyController {

    private MyBean myBean;

    @Autowired
    public void setMyBean(MyBean myBean) {

        this.myBean = myBean;
    }
}
```

- **En un atributo:** Spring **crea la instancia de la clase y una vez creada inyecta la dependencia.** Así es posible evitar los setters, ya que Spring asignará automáticamente esas propiedades con los valores que pasamos.

```
@Controller
public class MyController {

    @Autowired
    private MyBean myBean;
}
```

@Qualifier

La anotación **@Qualifier** es utilizada para **especificarle a Spring cuál bean es requerido.**

En el caso de que **haya más de un bean del mismo tipo disponible en el contenedor**, el framework va a lanzar una excepción del tipo *NoUniqueBeanDefinitionException*.

En el **ejemplo** tenemos una **interfaz** llamada **Formatter** que es **implementada** por dos clases: «FooFormatter» y «BarFormatter».

Cuando inyectamos el bean del tipo Formatter en la clase «FooService» Spring no sabe a cuál implementación de la clase nos estamos refiriendo.

Utilizando la anotación **@Qualifier** le decimos que **nos referimos a la implementación «FooFormatter»** con el mismo nombre que utilizamos cuando declaramos el componente.

```java
@Component("fooFormatter")
public class FooFormatter implements Formatter {

    public String format() {
        return "foo";
    }
}
```

```java
@Component("barFormatter")
public class BarFormatter implements Formatter {

    public String format() {
        return "bar";
    }
}
```

```java
public class FooService {

    @Autowired
    @Qualifier("fooFormatter")
    private Formatter formatter;

}
```

¿XML o Anotaciones?

Spring ofrece dos formas de configuración.

- **Anotaciones**: Permiten **definir y ver la DI junto con el código**, dentro de las clases.
- **XML**: La ventaja de usarlo es que toda la **configuración de Spring está en un solo lugar** y es posible visualizarla en conjunto sin buscar clase por clase las anotaciones. Usualmente termina dividiéndose en varios archivos.
- **Mixto**: Es posible utilizar ambas opciones, suele definirse la infraestructura de la aplicación (data source, transaction manager JMS connection factory, JMX) en un archivo XML mientras que se define la configuración referida a la inyección de dependencias en anotaciones.

Ejemplo 1: Definiendo un Bean mediante **anotación**.

```
import org.springframework.context.annotation.*;

@Configuration
public class HelloWorldConfig {
   @Bean
   public HelloWorld helloWorld(){
      return new HelloWorld();
   }
}
```

Ejemplo 2: Definiendo un Bean en **XML**.

```
<beans>
   <bean id = "helloWorld" class = "com.tutorialspoint.HelloWorld" />
</beans>
```

De ambas formas puede definirse un bean con iguales resultados.

Namespaces

Para la configuración XML, es necesario declarar un **namespace** al principio del archivo **ApplicationContext**, aquí definimos los namespaces y **referenciamos** el **schema (XSD) que valida el XML**.

En el ejemplo vemos:

- **Beans**: Lo configuramos como el namespace por defecto. Es requerido para definir los beans para la aplicación.
- **Context**: posibilita configurar el ApplicationContext.
- **P**: Facilita la configuración de DI para Setter.
- **C**: Facilita la configuración de DI para Constructor.
- **Util**: Provee utilidades para configurar DI.

En el caso de querer habilitar el uso de **anotaciones** debemos agregar el tag **<content:annotation-config>** y el tag **<context:component-scan>** que le indica a Spring dónde buscar en el código los beans.

Ejemplo 1: Namespaces.

```
<?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:p="http://www.springframework.org/schema/p"
    xmlns:c="http://www.springframework.org/schema/c"
    xmlns:util="http://www.springframework.org/schema/util"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.1.xsd
        http://www.springframework.org/schema/util
        http://www.springframework.org/schema/util/spring-util-3.1.xsd">

</beans>
```

Ejemplo 2: Habilitar el uso de anotaciones.

```
// Namespace declarations skipped

<context:annotation-config/>

<context:component-scan base-package="com.apress.prospring3.ch4.annotation" />
```

Component Scan

En el ApplicationContext.xml podemos indicarle **dónde debe escanear** el código de la aplicación **buscando anotaciones** tales como @Component, @Repository y @Service.

<context:component-scan base package="com.prospring">

Pueden **añadirse filtros para incluir o excluir** ciertos componentes, basándose en el tipo, anotaciones, expresiones de AspectJ o expresiones regulares.

<context:exclude-filter type="annotation"
expression="org.springframework.stereotype.Controller" />

Este filtro excluye instanciar las clases marcadas como @Controller dado que, puede suceder que éstos deban ser instanciados por otro contexto, como el WebApplicationContext.xml.

Spring Expression Language

En el archivo ApplicationContext.xml pueden utilizarse **expresiones (SpEL) para referenciar beans**.
En el ejemplo, los campos «engine» y «horsePower» del bean «someCar» usan expresiones que son referencias al bean «engine» y a su campo «horsePower.»

```
public class Engine {
    private int capacity;
    private int horsePower;
    private int numberOfCylinders;

    // Getters and setters
}
```

```
public class Car {
    private String make;
    private int model;
    private Engine engine;
    private int horsePower;

    // Getters and setters
}
```

```
<bean id="engine" class="com.baeldung.spring.spel.Engine">
    <property name="capacity" value="3200"/>
    <property name="horsePower" value="250"/>
    <property name="numberOfCylinders" value="6"/>
</bean>
<bean id="someCar" class="com.baeldung.spring.spel.Car">
    <property name="make" value="Some make"/>
    <property name="model" value="Some model"/>
    <property name="engine" value="#{engine}"/>
    <property name="horsePower" value="#{engine.horsePower}"/>
</bean>
```

Maven y el POM

Maven es una de las herramientas más populares para

- **gestionar las dependencias**
- **construir** las aplicaciones
- **empaquetarlas**.

Cada **artefacto** Maven está identificado por un Group ID un Artifact ID un tipo de empaquetamiento y una versión.

/com/curso/spring/hello-world-app/1.0.0/helloworldapp-1.0.0.jar

GROUP ID ARTIFACT V. Jar file

Maven descarga las dependencias del proyecto de **repositorios centralizados**. Si es necesario cambiar la versión de una librería, sólo habrá que actualizar el archivo POM.

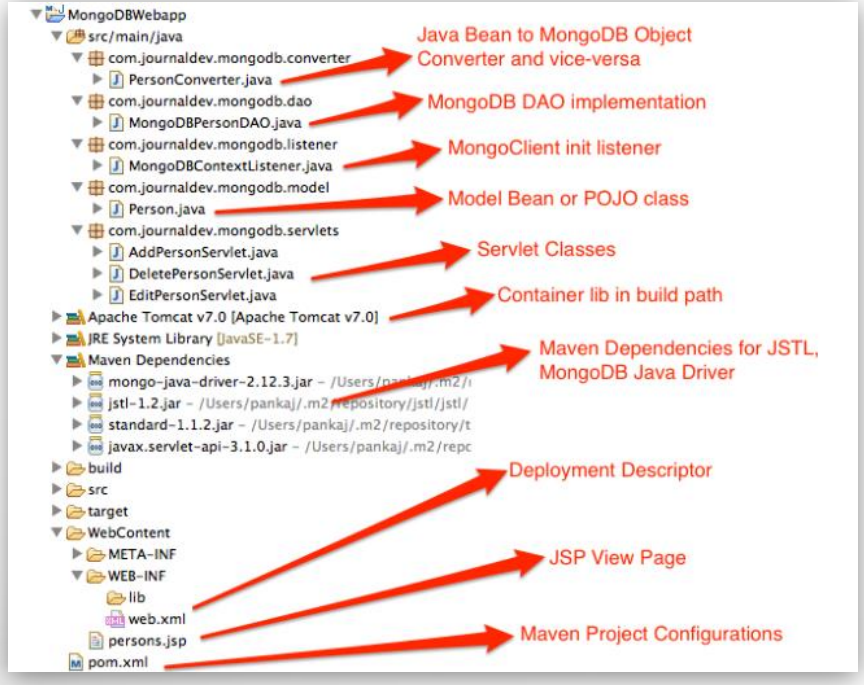
El **POM** (Project Object Model) Es una **representación XML de los recursos del proyecto**. En él especificamos todos los detalles para construir el proyecto.

```
<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/mavenv4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.prospinghibernate</groupId>
  <artifactId>gallery</artifactId>
  <version>1.0.0-SNAPSHOT</version>

  <packaging>war</packaging>

  <properties>
    <spring.version>3.0.2.RELEASE</spring.version>
    <hibernate.version>3.5.0-Final</hibernate.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>${spring.version}</version>
    </dependency>

    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>${hibernate.version}</version>
    </dependency>
```

Tags en el POM

Tag	Descripción
versionId	Se usa para declarar la **versión del proyecto.**
groupId	ID único, **se utiliza el nombre del proyecto como un fully qualified domain name** (FQDN).
artifactId	Es un **nombre único dentro de un groupId, con una versión única**. Dos proyectos no pueden tener la misma combinación de groupId:artifactId:version.
packaging	Describe el **tipo de empaquetado** que tendrá el proyecto. Por defecto es **JAR**, pero puede elegirse **WAR** en caso de tratarse de una aplicación web o **EAR**.
properties	Pueden definirse **variables** que se vayan a utilizar en el resto del proyecto.
dependencies	Son **archivos Jar externos , librerías java que usa el proyecto**. Se definen las librerías identificándolas con su groupId, artifactId y versión. Si no se encuentran en el repositorio maven local son descargadas del repositorio de maven.
plugins	Se pueden definir **plugins** para el proyecto como por ejemplo **Apache Tomcat** para correr una aplicación web.

Estructura de un proyecto WEB



Repositorio Maven