Inyección de dependencias en Spring Boot

Gustavo García, 24 de septiembre de 2022.

# Proyecto de ejemplo

En la carpeta inyedep hay un proyecto muy simple, que tiene lo mínimo imprescindible para ilustrar todos los puntos discutidos en este apunte. En el código fuente de ese proyecto hay abundantes comentarios, con referencias a la documentación pertinente.

# Inyección de dependencias

La inyección de dependencias es un patrón de diseño que delega la administración de dependencias en el framework. Spring inicializará e inyectará los objetos necesarios para otros objetos. La inyección de dependencias (DI) también se conoce como Inversion of Control (IoC). Es un proceso mediante el cual los objetos definen sus dependencias (es decir, los otros objetos con los que trabajan) solo a través de:

* argumentos del constructor,
* argumentos de un método factoría, o
* propiedades que se establecen en la instancia del objeto después de que se construye o se devuelve desde un método factoría.

Luego, el contenedor de Spring inyecta las dependencias requeridas cuando crea el bean que las necesita. Este proceso es fundamentalmente el inverso (de ahí el nombre, Inversión de Control) del proceso en el cual es el propio bean el que controla la creación de instancias o la ubicación de sus dependencias mediante la instanciación directa de clases o un mecanismo como el patrón del localizador de servicios.

Spring facilita la inyección de dependencias por medio de dos anotaciones que trabajan en conjunto:

* @SpringBootApplication, y
* @Autowired.

# @SpringBootApplication

Spring Boot introduce la anotación @SpringBootApplication. Esta única anotación es equivalente a usar:

* @Configuration,
* @EnableAutoConfiguration, y
* @ComponentScan.

Como resultado, el container de Spring Boot realizará automáticamente una cantidad de tareas de configuración, que no vamos a discutir ahora, y lo que realmente nos interesa en este apunte, escaneará automáticamente los componentes del paquete actual y sus subpaquetes. En la jerga de Spring a esto se le llaman scanning del classpath. Este mecanismo permite:

* identificar los lugares en los que queremos inyectar una dependencia,
* identificar la clase o el bean de esa dependencia,
* buscar en el proyecto para ver si encuentra el bean que necesita,
* si lo encuentra, inicializarlo apropiadamente,
* finalmente inyectarlo en el lugar donde se necesita.

Si no lo encuentra, arroja una excepción.

Usemos esta anotación en la clase principal de la aplicación:

@SpringBootApplication

public class App {

**public** **static** **void** **main**(String[] args) {

SpringApplication.run(App.class, args);

}

}

# Factoría de objetos

Spring funciona como una factoría de objetos. Una clase se registra para instanciar objetos con alguna de las anotaciones @Component, @Controller, @Repository, @RestController o @Service. Una vez que los objetos están creados la anotación @Autowired se encarga de construir las ligazones entre los distintos elementos, y esta es la inyección de dependencias.

# @Component

<https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Component.html>

Indica que la clase anotada es un "componente". La clase se considera candidata para la detección automática cuando se utiliza la configuración basada en anotaciones y el scanning de classpath. Otras anotaciones también se pueden usar a nivel de clase para identificar un componente, generalmente un tipo especial de componente: por ejemplo @Controller, @Repository o la anotación @Aspect de AspectJ.

# @Controller

<https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Controller.html>

Indica que la clase anotada es un "Controlador" (por ejemplo, un controlador web). Esta anotación es una especialización de @Component, lo que permite que las clases de implementación se detecten automáticamente a través del escaneo de classpath. Por lo general, se usa en combinación con métodos de controlador anotados basados en la anotación @RequestMapping.

# @Repository

<https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Repository.html>

Indica que una clase anotada es un "repositorio", definido originalmente por Domain-Driven Design (Evans, 2003) como "un mecanismo para encapsular el comportamiento de almacenamiento, recuperación y búsqueda que emula una colección de objetos".

Esta anotación también sirve como una especialización de @Component, lo que permite que las clases de implementación se detecten automáticamente a través del escaneo de classpath.

# @RestController

<https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html>

Una anotación de conveniencia que en sí misma está anotada con @Controller y @ResponseBody.

Los tipos que llevan esta anotación se tratan como controladores donde los métodos @RequestMapping asumen la semántica @ResponseBody de manera predeterminada.

# @Service

<https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Service.html>

Indica que una clase anotada es un "Servicio", definido como "una operación que se ofrece como una interfaz que se encuentra sola en el modelo, sin estado encapsulado".

Esta anotación sirve como una especialización de @Component, lo que permite que las clases de implementación se detecten automáticamente a través del escaneo de classpath.

# @Autowired

<https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/annotation/Autowired.html>

@Autowired es una anotación que permite inyectar dependencias dentro de Spring . Como ya se dijo, trabaja en conjunto con la anotación @SpringBootApplication.

La anotación @SpringBootApplication habilita, entre otras cosas, el scanning automático del classpath. Cuando Spring encuentra la anotación @Autowired, se fija de qué clase es la dependencia que se necesita, y busca por todo el proyecto. No mira todas las clases, sino solo las que están marcadas como candidatas por medio de las anotaciones @Component, @Controller, @Repository, @RestController o @Service. De entre todas las candidatas, elige la que sea más apropiada. Sigue unos determinados criterios, que no vamos a discutir ahora, pero son de sentido común.

Podemos usar @Autowired en constructores, campos y setters. Marca un constructor, un campo, un setter o un método de configuración para que las facilidades de inyección de dependencias de Spring lo conecten automáticamente.

# @Autowired en campos

Los campos se inyectan en un bean justo después de la construcción del bean, antes de que se invoque cualquier método de configuración. El campo no necesita ser público.

Veamos cómo podemos anotar una propiedad usando @Autowired. Esto elimina la necesidad de getters y setters. Primero, definamos un bean fooFormatter:

@Component("fooFormatter")

**public** **class** **FooFormatter** {

**public** String **format**() {

**return** "foo";

}

}

Este bean que acabamos de definir es el que será inyectado en otro bean; es la dependencia que será inyectada. La clase FooFormater está anotada como @Component. Por lo tanto, es candidata para inyección de dependencias. Inyectaremos el bean anterior en el bean FooService usando @Autowired en la definición del campo:

@Component

**public** **class** **FooService** {

@Autowired

**private** FooFormatter fooFormatter;

}

El campo fooFormatter está anotado como @Autowired. Por lo tanto Spring se va a fijar que es de clase FooFormater, y va a buscar por todo el proyecto la clase FooFormatter, y va a revisar si está marcada como candidata a la inyección de dependencias. Si no encuentra justo la clase FooFormatter, tiene unos criterios para buscar alternativas, pero ahora no los vamos a discutir. En este ejempo, sí la encuentra. Como resultado, Spring inyecta fooFormatter cuando se crea FooService. Esto significa que cuando se cree una instancia de la clase FooService, esa instancia (objeto, o bean) tendrá su campo fooFormatter, de clase FooFormatter, correctamente inicializado, aunque nosotros no escribimos ni una línea explícitamente. Todo lo hace Spring.

# @Autowired en Setters

Ahora intentemos agregar la anotación @Autowired en un método setter. En el siguiente ejemplo, se llama al método setter con la instancia de FooFormatter cuando se crea FooService:

**public** **class** **FooService** {

**private** FooFormatter fooFormatter;

@Autowired

**public** **void** **setFormatter**(FooFormatter fooFormatter) {

this.fooFormatter = fooFormatter;

}

}

# @Autowired en constructores

Solo un constructor de cualquier clase de bean determinada puede declarar esta anotación con el atributo required() establecido en verdadero, lo que indica que el constructor debe conectarse automáticamente cuando se usa como un bean Spring. Además, si el atributo requerido se establece en verdadero, solo se puede anotar un único constructor con @Autowired. Si varios constructores no requeridos declaran la anotación, se considerarán candidatos para el cableado automático. Se elegirá el constructor con el mayor número de dependencias que se puedan satisfacer haciendo coincidir beans en el contenedor Spring. Si no se puede satisfacer a ninguno de los candidatos, se utilizará un constructor principal/predeterminado (si está presente). De manera similar, si una clase declara múltiples constructores pero ninguno de ellos está anotado con @Autowired, entonces se usará un constructor primario/predeterminado (si está presente). Si una clase solo declara un único constructor para empezar, siempre se usará, incluso si no se anota. Un constructor anotado no tiene que ser público.

Usemos @Autowired en un constructor. Veremos que Spring inyecta una instancia de FooFormatter como un argumento para el constructor FooService:

**public** **class** **FooService** {

**private** FooFormatter fooFormatter;

@Autowired

**public** **FooService**(FooFormatter fooFormatter) {

this.fooFormatter = fooFormatter;

}

}

# Beneficios de usar la inyección de constructor

Con la inyección del constructor, podemos escribir código para verificar el tipo de instancia del bean inyectado o hacer lo que podamos con la codificación. como iniciar sesión, lanzar una excepción, etc.

# Desventajas de usar la inyección de constructor

Se pueden requerir muchos constructores para inyectar las dependencias necesarias en algún caso especial. A veces se necesita un constructor con muchos argumentos, que puede ser bastante feo.

# @Autowired y dependencias opcionales

Cuando se construye un bean, las dependencias de @Autowired deberían estar disponibles. De lo contrario, si Spring no puede resolver el cableado de un bean, lanzará una excepción. En consecuencia, evita que el contenedor Spring se inicie con éxito con la excepción del formulario:

Causado por: org.springframework.beans.factory.NoSuchBeanDefinitionException:

No se encontró ningún bean calificador de tipo [com.autowire.sample.FooDAO] para la dependencia:

se esperaba al menos 1 bean que califique como candidato de conexión automática para esta dependencia.

Anotaciones de dependencia:

{@org.springframework.beans.factory.annotation.Autowired(required=true)}

Caused by: org.springframework.beans.factory.NoSuchBeanDefinitionException:

No qualifying bean of type [com.autowire.sample.FooDAO] found for dependency:

expected at least 1 bean which qualifies as autowire candidate for this dependency.

Dependency annotations:

{@org.springframework.beans.factory.annotation.Autowired(required=true)}

Para arreglar esto, necesitamos declarar un bean del tipo requerido:

**public** **class** **FooService** {

@Autowired(required = false)

**private** FooDAO dataAccessor;

}

# Desambiguación de Autowired

De forma predeterminada, Spring resuelve las entradas de @Autowired por tipo. Si hay más de un bean del mismo tipo disponible en el contenedor, el marco generará una excepción fatal. Para resolver este conflicto, debemos decirle a Spring explícitamente qué bean queremos inyectar. Vamos a ver dos posibles soluciones: una es cambiar el nombre del campo para que sea simplemente el nombre de la clase que queremos elegir, solo que con inicial minúscula, y el otro es usando la anotación @Qualifier.

# Desambiguación de Autowired por nombre de bean

Spring usa el nombre del bean como valor predeterminado del Qualifier. Inspeccionará el contenedor y buscará un bean con el nombre exacto de la propiedad para autoconectarlo. Por lo tanto, en nuestro ejemplo, Spring hace coincidir el nombre de la propiedad fooFormatter con la implementación de FooFormatter. Por lo tanto, inyecta esa implementación específica al construir FooService:

**public** **class** **FooService** {

@Autowired

**private** Formatter fooFormatter;

}

# Desambiguación de Autowired por @Qualifier

Por ejemplo, veamos cómo podemos usar la anotación @Qualifier para indicar el bean requerido. Primero, definiremos 2 beans de tipo Formatter:

@Component("fooFormatter")

**public** **class** **FooFormatter** **implements** **Formatter** {

**public** String **format**() {

**return** "foo";

}

}

@Component("barFormatter")

**public** **class** **BarFormatter** **implements** **Formatter** {

**public** String **format**() {

**return** "bar";

}

}

Ahora intentemos inyectar un bean Formatter en la clase FooService:

**public** **class** **FooService** {

@Autowired

**private** Formatter formatter;

}

En nuestro ejemplo, hay dos implementaciones concretas de Formatter disponibles para el contenedor Spring. Como resultado, Spring generará una excepción NoUniqueBeanDefinitionException al construir FooService:

Causado por: org.springframework.beans.factory.NoUniqueBeanDefinitionException:

No se define ningún bean calificador de tipo [com.autowire.sample.Formatter]:

Se esperaba un bean coincidente único pero se encontraron 2: barFormatter,fooFormatter

Caused by: org.springframework.beans.factory.NoUniqueBeanDefinitionException:

No qualifying bean of type [com.autowire.sample.Formatter] is defined:

expected single matching bean but found 2: barFormatter,fooFormatter

Podemos evitar esto refinando la implementación mediante una anotación @Qualifier:

**public** **class** **FooService** {

@Autowired

@Qualifier("fooFormatter")

**private** Formatter formatter;

}

Cuando hay múltiples beans del mismo tipo, es una buena idea usar @Qualifier para evitar ambigüedades. Tenga en cuenta que el valor de la anotación @Qualifier coincide con el nombre declarado en la anotación @Component de nuestra implementación FooFormatter.

# Desambiguación de Autowired por @Qualifier personalizado

Spring también nos permite crear nuestra propia anotación @Qualifier personalizada. Para hacerlo, debemos agregar a la anotación @Qualifier la definición:

@Qualifier

@Target({

ElementType.FIELD, ElementType.METHOD, ElementType.TYPE, ElementType.PARAMETER})

@Retention(RetentionPolicy.RUNTIME)

**public** @interface FormatterType {

String **value**();

}

Entonces podemos usar FormatterType dentro de varias implementaciones para especificar un valor personalizado:

@FormatterType("Foo")

@Component

**public** **class** **FooFormatter** **implements** **Formatter** {

**public** String **format**() {

**return** "foo";

}

}

@FormatterType("Bar")

@Component

**public** **class** **BarFormatter** **implements** **Formatter** {

**public** String **format**() {

**return** "bar";

}

}

Finalmente, nuestra anotación de @Qualifier personalizada está lista para usar para el autowiring:

@Component

**public** **class** **FooService** {

@Autowired

@FormatterType("Foo")

**private** Formatter formatter;

}

El valor especificado en la metaanotación @Target restringe dónde aplicar el calificador, que en nuestro ejemplo son campos, métodos, tipos y parámetros.

# Conclusión

En este artículo, discutimos el autowiring y las diferentes formas de usarlo. También examinamos formas de resolver dos excepciones comunes de cableado automático causadas por un bean faltante o una inyección de bean ambigua.

El código fuente de este artículo está disponible en [el proyecto GitHub](https://github.com/eugenp/tutorials/tree/master/spring-core-2).

Los comandos usados para descargar esta carpeta son:

mkdir spring-core-2

cd spring-core-2

git init

git remote add -f origin https://github.com/eugenp/tutorials.git

git config core.sparsecheckout true

echo "spring-core-2/" >> .git/info/sparse-checkout

git pull origin master

El proyecto está en la carpeta spring-core-2. Es demasiado complejo para este curso.

# Fuentes

**Guide to Spring @Autowired**

<https://www.baeldung.com/spring-autowire>

**Spring @Autowired y la inyección de dependencias**

[https://www.arquitecturajava.com/spring-autowired-y-la-inyeccion-de-depehyndencias/](https://www.arquitecturajava.com/spring-autowired-y-la-inyeccion-de-dependencias/)

**Autowiring in Spring Application**

<https://programmingsharing.com/autowiring-in-spring-application-f168719b93ad>

**1.1. Introduction to the Spring IoC Container and Beans**

<https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-introduction>