Apunte Spring Boot - JPA - Lombok - Test

Gustavo García, 26 de septiembre de 2022.

# Resumen

En este apunte quiero reunir todos los materiales que están dispersos. A medida que revise cada documento de cada proyecto y materia, iré incorporando acá lo que corresponda, en el lugar indicado.

Esto me va a tomar varios meses, pero al final creo que el resultado valdrá la pena.

# application.properties

<https://docs.spring.io/spring-boot/docs/current/reference/html/howto.html#howto.data-access>

En este archivo configuramos la conexión a la base de datos.

spring.application.name=AAA

server.port=BBB

spring.datasource.driver-class-name =com.mysql.cj.jdbc.Driver

spring.datasource.url=jdbc:mysql://${MYSQL\_HOST:localhost}:3306/CCC

spring.datasource.username=DDD

spring.datasource.password=EEE

# https://stackoverflow.com/questions/50322550/hibernate-dialect-for-mysql-8

spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

spring.jpa.hibernate.ddl-auto=update

#spring.jpa.show-sql: true

logging.level.org.hibernate.SQL=debug

AAA: el nombre de la aplicación en el pom.xml.

BBB: el puerto por el que Tomcat va a servir la aplicación, normalmente 8080.

CCC: el nombre de la base de datos, tal como está en MySQL.

DDD: el nombre del usuario que tiene acceso a la base de datos, mejor que no sea el root.

EEE: la password del usuario anterior.

# Anotaciones

Vamos a listar todas las anotaciones que usamos. Algunas son de Spring Boot y otras de Lombok.

Este artículo de Baeldung cubre todas las anotaciones:

<https://www.baeldung.com/jpa-entities>

Este otro es más amigable:

<https://www.paradigmadigital.com/dev/proyecto-lombok-facilitame-la-vida/>

## @Data

La documentación oficial está en:

https://projectlombok.org/features/Data

y en:

https://projectlombok.org/api/lombok/Data.html

Los siguientes tutoriales explican cómo se usa:

https://javabydeveloper.com/lombok-data-annotation/

https://www.educative.io/answers/what-is-the-data-annotation-in-lombok

@Data agrupa las características de @ToString, @EqualsAndHashCode, @Getter, @Setter y @RequiredArgsConstructor juntas: en otras palabras, @Data genera todos los estándares que normalmente se asocian con POJOS y beans:

* Getters para todos los campos.
* Setters para todos los campos no finales y las implementaciones toString.
* Equals y hashCode que involucran los campos de la clase.
* Un constructor que inicializa todos los campos finales.
* Así como todos los campos no finales sin inicializador que haya sido marcado con @NonNull, para garantizar que el campo nunca sea nulo.

OJO: no se trata de un patrón de inyección de dependencias. Se puede usar combinada con las anotaciones de persistencia de JPA, pero hay que poner ambas, porque unas no reemplazan a las otras.

NOTAR que solo se declaran los campos. No declaramos explícitamente ningún método, y por supuesto no los implementamos. Todo eso lo hace por nosotros automáticamente la anotación @Data.

Tiene como ventajas que nos evita tener que escribir boiler plate code, y teminamos usando las implementaciones de esos métodos que han sido escritas por los mejores programadores. Tiene la desventaja de que es poco flexible, porque escapa a nuestro control.

## @Entity

<https://stackoverflow.com/a/29333628>

La anotación @Entity de JPA indica a la plataforma que inyecte a esta clase las dependencias necesarias para ser una entidad persistente. Para eso, la clase necesita por ejemplo un constructor, pero de eso se puede encargar la anotación @Data de Lombok, si es que la usamos.

La anotación @Entity define que una clase se puede asignar a una tabla. Y eso es todo, es solo un marcador, como por ejemplo la interfaz Serializable. Es necesario poner @Entity, porque esa es la forma en que está diseñada JPA.

Cuando crea una nueva entidad, debe hacer al menos dos cosas:

* anote la clase con @Entity,
* cree un campo de identificación y anótelo con @Id.

Cualquier otra cosa es opcional, por ejemplo, el nombre de la tabla se deriva del nombre de la clase de entidad (y, por lo tanto, la anotación @Table puede ser opcional), las columnas de la tabla se derivan de variables de entidades (y, por lo tanto, la anotación @Column puede ser opcional), y así sucesivamente. JPA trata de proporcionar un comienzo rápido y fácil para los desarrolladores que desean aprender.

Se puede usar combinada con @Data, de Lombok, pero hay que poner ambas, porque una no reemplaza a la otra.

## @GeneratedValue

<https://stackoverflow.com/questions/20603638/what-is-the-use-of-annotations-id-and-generatedvaluestrategy-generationtype>

La anotación @Id se hereda de javax.persistence.Id. Indica que el campo de miembro a continuación es la clave principal de la entidad actual. Por lo tanto, Hibernate y Spring pueden hacer algunos trabajos de reflexión basados en esta anotación.

La anotación @GeneratedValue es para configurar la forma de incremento de la columna (campo) especificada. Por ejemplo, al usar Mysql, puede especificar auto\_increment en la definición de la tabla para que sea autoincremental y luego usar

@GeneratedValue(strategy = GenerationType.IDENTITY)

en el código Java para indicar que también quiere usar esta estrategia del lado del servidor de la base de datos. Además, puede cambiar el valor de esta anotación para adaptarse a diferentes requisitos.

## @GetMapping

Documentación oficial:

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

Asigna solicitudes HTTP GET a métodos de controlador específicos. Es decir, cuando en el browser se pone la URL indicada, el controlador llama al método que está inmediatamente pegado a esta anotación.

@GetMapping es una anotación compuesta que actúa como atajo para

@RequestMapping(method = RequestMethod.GET)

## @Id

<https://stackoverflow.com/questions/20603638/what-is-the-use-of-annotations-id-and-generatedvaluestrategy-generationtype>

La anotación @Id se hereda de javax.persistence.Id. Indica que el campo de miembro a continuación es la clave principal de la entidad actual. Por lo tanto, Hibernate y Spring pueden hacer algunos trabajos de reflexión basados en esta anotación.

El tipo Long del campo id viene forzado por la interfaz genérica CrudRepository.

## @Override

Indica que un método con un determinado nombre, signatura y tipo de retorno reemplaza al método con el mismo nombre, signatura y tipo de retorno del supertipo más próximo.

Manual Java 4.10.2 Subtyping among Class and Interface Types

En resumen, el supertipo directo de una clase o una interfaz es la clase que extiende o la interfaz que implementa.

Manual Java 8.4.8 Inheritance, Overriding, and Hiding

## @RequestMapping

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

La URL que vaya en la anotación habrá que agregarla detrás del puerto :8080 en todas las llamadas a esta aplicación. Se puede usar a nivel de clase o de método. Por ejemplo @RequestMapping("/user") resultaría en lo siguiente:

localhost:8080/user.... y detrás de esto habría que agregar el resto de la URL.

Si usamos @RequestMapping("") o @RequestMapping("/") no necesitamos nada, y queda simplemente localhost:8080.

## @RequestParam

Documentación oficial:

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

Anotación que indica que un parámetro del método debe vincularse a un parámetro de la solicitud (request) web. En Spring MVC, los "parámetros de la solicitud" se asignan a parámetros de query, datos de formulario y partes en solicitudes de multipartes. Esto se debe a que la API de servlet combina los parámetros de query y los datos de formulario en un solo mapa llamado "parámetros", y eso incluye el análisis automático del cuerpo de la solicitud.

## @Table

<https://stackoverflow.com/a/37733064>

<https://www.baeldung.com/jpa-entities#table>

La anotación @Table le permite especificar los detalles de la tabla que se utilizará para conservar la entidad en la base de datos. La anotación @Table proporciona cuatro atributos, lo que le permite cambiar el nombre de la tabla, su catálogo y su esquema, y aplicar restricciones únicas en las columnas de la tabla.

## @Transactional

<https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/Transactional.html>

Las transacciones se pueden manejar de modo programático o declarativo. El principal inconveniente del modo programático es que acopla fuertemente la lógica de negocios con la capa de persistencia. El modo preferido es el declarativo, y para esto se usa @Transactional.

Indica que un método individual o una clase funcionan de modo transaccional. Cuando se declara a nivel de clase, se aplica por defecto a todos los métodos de la clase declarante y sus subclases.

Si no se configuran reglas de rollback personalizadas, la transacción se revertirá si hay RuntimeException o Error, pero no en las excepciones chequeadas.

La anotación @Transactional incluye automáticamente el código necesario para deshabilitar auto\_commit() y para soportar Commit() y Rollback(). Esto ayuda a revertir la operación de la base de datos en caso de que surja alguna excepción o error, lo que garantiza la coherencia y la eficacia en el procesamiento de datos. En pocas palabras, está garantizado que ocurrirá una y solo una de dos cosas: o bien toda la transacción se completa exitosamente, en cuyo caso queda comprometida, o bien ocurre algún error, en cuyo caso todo se vuelve atrás (rollback) y el sistema completo queda en el estado original que tenía antes de iniciar la transacción, como si nada hubiera ocurrido.

**Notar:**

* La anotación @Transactional funcionará solo para los métodos públicos de la clase anotada por la anotación @Service.
* Es necesario que la clase de servicio sea @Autowired, de lo contrario, la anotación @Transactional no funcionará.

# Favicon

Si no se tiene un archivo index.html, un modo simple es pegar favicon.ico en la carpeta src\main\resources\static.

# index.html

Crear un archivo src\main\resources\static\index.html

Si existe, la app arranca por acá. Este archivo puede ser el template que genera VS Code, o simplemente un html con un texto plano que diga hola, nada más. o algo más complejo.

# Interfaces marcadoras para CRUD

Discutimos algunos conceptos que aparecen en las aplicaciones CRUD.

## Marker Interfaces

<https://www.baeldung.com/java-marker-interfaces>

Una interfaz marcadora es una interfaz que no tiene métodos ni constantes. Proporciona información de tipo de tiempo de ejecución sobre los objetos, por lo que el compilador y la JVM tienen información adicional sobre el objeto. Una marker interface también se denomina tagging interface.

Aunque las interfaces marcadoras todavía están en uso, es muy probable que apunten a un olor a código y deben usarse con cuidado. La razón principal de esto es que borran las líneas sobre lo que representa una interfaz, ya que los marcadores no definen ningún comportamiento. El desarrollo más nuevo favorece las anotaciones para resolver algunos de los mismos problemas.

## Domain type

Un domain type es una de las clases que modelan a las entidades del dominio del problema. Estas clases normalmente estarán en el paquete models.

## Domain type's id type

Es el tipo del campo id de la clase. Como esta clase será implementada en una tabla de la base de datos, necesita una clave primaria, que es justamente esta id.

## Domain repositories

Estas clases están normalmente en el paquete repositories, y son las que conectan las clases de las entidades (que están el paquete models) con el servidor de la base de datos.

Si lo que queremos es nada más que un CRUD, estas clases implementarán la interfaz CrudRepository.

## Interface CrudRepository<T,ID>

<https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html>

En el paquete repositories hay una o más interfaces. Cada una de estas interfaces está pensada para una determinada tabla de la base de datos. Esta interfaz no implementa ningún método. No publica ni métodos ni propiedades. Se limita a extender CrudRepository, que es también una interfaz: publica pero no implementa. En otras palabras, no implementamos ni métodos ni propiedades. La magia ocurre en la clase de implementación, que probablemente está en el paquere services. Tiene un campo de tipo de esta interfaz de repositorio, y ese campo tiene la anotación @Autowired. Esa anotación es la que se ocupa de la inyección de dependencias. La interfaz CrudRepository es genérica en el tipo que estemos queriendo administrar. La clase administrada está en el paquete models.

Esta interfaz extiende la interfaz Repository.

Interfaz para operaciones CRUD genéricas en un repositorio para un tipo específico.

The CrudRepository provides sophisticated CRUD functionality for the entity class that is being managed.

|  |  |
| --- | --- |
| 1 | Saves the given entity. |
| 2 | Returns the entity identified by the given id. |
| 3 | Returns all entities. |
| 4 | Returns the number of entities. |
| 5 | Deletes the given entity. |
| 6 | Indicates whether an entity with the given id exists. |

|  |  |
| --- | --- |
| **Modifier and Type** | **Method and Description** |
| long | [**count**](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html#count--)()  Returns the number of entities available. |
| void | [**delete**](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html#delete-T-)([**T**](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html) entity)  Deletes a given entity. |
| void | [**deleteAll**](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html#deleteAll--)()  Deletes all entities managed by the repository. |
| void | [**deleteAll**](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html#deleteAll-java.lang.Iterable-)([**Iterable**](https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html?is-external=true)<? extends [**T**](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html)> entities)  Deletes the given entities. |
| void | [**deleteAllById**](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html#deleteAllById-java.lang.Iterable-)([**Iterable**](https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html?is-external=true)<? extends [**ID**](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html)> ids)  Deletes all instances of the type T with the given IDs. |
| void | [**deleteById**](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html#deleteById-ID-)([**ID**](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html) id)  Deletes the entity with the given id. |
| boolean | [**existsById**](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html#existsById-ID-)([**ID**](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html) id)  Returns whether an entity with the given id exists. |
| [**Iterable**](https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html?is-external=true)<[**T**](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html)> | [**findAll**](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html#findAll--)()  Returns all instances of the type. |
| [**Iterable**](https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html?is-external=true)<[**T**](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html)> | [**findAllById**](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html#findAllById-java.lang.Iterable-)([**Iterable**](https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html?is-external=true)<[**ID**](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html)> ids)  Returns all instances of the type T with the given IDs. |
| [**Optional**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html?is-external=true)<[**T**](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html)> | [**findById**](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html#findById-ID-)([**ID**](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html) id)  Retrieves an entity by its id. |
| <S extends [**T**](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html)> S | [**save**](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html#save-S-)(S entity)  Saves a given entity. |
| <S extends [**T**](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html)> [**Iterable**](https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html?is-external=true)<S> | [**saveAll**](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html#saveAll-java.lang.Iterable-)([**Iterable**](https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html?is-external=true)<S> entities)  Saves all given entities. |

Todos los métodos anteriores son automáticamente implementados por Spring Boot con solo incluir la correspondiente anotación.

## Interface Repository<T,ID>

<https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/Repository.html>

Interfaz de marcador de repositorio central. Captura el tipo de dominio para administrar, así como el tipo de identificación del tipo de dominio. El propósito general es mantener información de tipo, así como también poder descubrir interfaces que amplíen esta durante el escaneo de classpath para facilitar la creación de Spring Bean.

Los repositorios de dominio que amplían esta interfaz pueden exponer selectivamente métodos CRUD simplemente declarando métodos de la misma firma que los declarados en CrudRepository.

# 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.

## @Autowired

@Autowired sirve para inyectar en una clase destino una propiedad perteneciente a otra clase origen. En este sentido, Spring funciona como una factoría[[1]](#footnote-1) de objetos, es decir que implementa el patrón factoría. Este caso es habitual cuando tenemos un servicio y un repositorio.

### Spring @AutowiredClase origen

Una clase se registra como origen de inyección con alguna de estas anotaciones:

* @Controller,
* @Service,
* @Repository, o
* @RestController.

En la clase origen tenemos la propiedad que deseamos inyectar.

### Clase destino

La clase destino es la clase en la cual deseamos inyectar la propiedad. Para eso, necesitamos declarar la propiedad, y anotarla con @Autowired. SpringBoot se encarga de implementar todo lo necesario, es decir que inyecta las dependencias.

## @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>

La anotación @RestController es la combinación de @Controller y @ResponseBody.

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

Si está anotando la clase todos los métodos la heredan y no es necesario anotar cada uno de ellos. Todos tendrán la semántica de @ResponseBody.

## @RequestBody

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

Indica un parámetro de un método debe vincularse al cuerpo de la solicitud web (web request). El cuerpo de la solicitud se pasa a través de HttpMessageConverter para resolver el argumento del método según el tipo de contenido de la solicitud. Opcionalmente, se puede aplicar la validación automática anotando el argumento con @Valid. Compatible con métodos de controlador anotados.

## @ResponseBody

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

<https://stackoverflow.com/a/28647129/2740402>

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

Esta anotación indica que el valor de retorno de un método es el response body, o sea el cuerpo de la respuesta web, no una vista. Lógicamente, la response corresponde a un request. Se puede usar en métodos de controlador anotados. Esta anotación también se puede agregar en el nivel de tipo, en cuyo caso se hereda y no es necesario agregarla en el nivel de método.

Un método podría retornar, o intentar retornar, objetos Java. Pero una respuesta HTTP, por supuesto, no puede contener objetos Java.Entonces, si el valor de retorno incluye objetos Java, esos objetos se transforman a un formato adecuado para aplicaciones REST, generalmente JSON o XML. Esto es lo que se llama la semántica de la anotación.

La elección del formato depende de los convertidores de mensajes instalados, de los valores del atributo produces de la anotación @RequestMapping y del tipo de contenido que acepta el cliente (que está disponible en los encabezados de solicitud HTTP). Por ejemplo, si la solicitud dice que acepta XML, pero no JSON, y hay un convertidor de mensajes instalado que puede transformar la lista a XML, se devolverá XML.

## @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.

En una aplicación, la lógica de negocio está en el paquete services, por lo que usamos la anotación @Service para indicar que una clase pertenece a ese paquete. También es una especialización de la anotación @Component como la anotación @Repository. Una de las cosas más importantes de la anotación @Service es que solo se puede aplicar a clases. Se utiliza para marcar la clase como proveedor de servicios. Por lo tanto, la anotación general de @Service se usa con clases que brindan algunas funcionalidades de lógica de negocios.

## @Autowired

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

@Autowired inyecta dependencias dentro de Spring . Como ya se dijo, trabaja en conjunto con la anotación @SpringBootApplication. Para entender cómo trabaja hay que explicar desde el principio. 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.

# 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>

1. <https://www.arquitecturajava.com/usando-el-patron-factory/> [↑](#footnote-ref-1)