Spring Framework

> ⚠ EN DESARROLLO ⚠

Overview

Spring Framework es un poderoso y ampliamente utilizado marco de desarrollo de software para aplicaciones empresariales en Java. Diseñado para simplificar y acelerar el desarrollo de aplicaciones, Spring ofrece un enfoque integral que abarca desde la configuración hasta la implementación, abordando varios aspectos del desarrollo de software como la inversión de control, la inyección de dependencias, la gestión de transacciones, la seguridad y mucho más.

Una de las características distintivas de Spring es su **enfoque modular y extensible**, permitiendo a los desarrolladores elegir los módulos específicos que necesitan para sus proyectos. Además, fomenta las mejores prácticas de programación y sigue el principio de diseño de "Programación Orientada a Aspectos" (AOP), que facilita la separación de preocupaciones y mejora la modularidad del código.

Spring Framework se utiliza comúnmente para construir aplicaciones empresariales robustas y escalables, facilitando la creación de servicios web, aplicaciones basadas en la arquitectura Modelo-Vista-Controlador (MVC), integración con bases de datos, gestión de transacciones y mucho más. Con una comunidad activa y un ecosistema de proyectos relacionados, Spring ha evolucionado para adaptarse a las cambiantes demandas del desarrollo de software, convirtiéndose en una opción popular entre los desarrolladores Java.

Spring Boot es una extensión del popular Spring Framework que se centra en simplificar drásticamente el proceso de desarrollo de aplicaciones Java, especialmente aplicaciones basadas en Spring. Su objetivo principal es facilitar la creación de aplicaciones autónomas, autocontenidas y listas para la producción con la menor cantidad de configuración posible.

La relación entre Spring Boot y Spring Framework es fundamental, ya que Spring Boot se construye sobre la base sólida proporcionada por Spring. Spring Boot utiliza las características clave de Spring, como la inversión de control (IoC) y la inyección de dependencias, pero agrega una capa de convenciones y configuraciones por defecto para acelerar el desarrollo.

Lo más notable de Spring Boot es su enfoque de "opinión sobre la configuración", lo que significa que proporciona configuraciones predeterminadas sensatas para la mayoría de los casos de uso, permitiendo a los desarrolladores empezar rápidamente con sus proyectos sin tener que configurar extensamente. No obstante, sigue siendo altamente personalizable, permitiendo a los desarrolladores anular las configuraciones por defecto según sea necesario.

Con Spring Boot, el proceso de desarrollo se simplifica mediante la inclusión de un servidor embebido, como Tomcat o Jetty, lo que elimina la necesidad de desplegar la aplicación en un servidor externo. También facilita la gestión de dependencias mediante el uso de la herramienta Spring Initializr para generar proyectos con las dependencias necesarias.

En resumen, Spring Boot es una extensión de Spring Framework diseñada para hacer que el desarrollo de aplicaciones Java sea más rápido, sencillo y eficiente al proporcionar configuraciones por defecto y convenciones inteligentes sin sacrificar la flexibilidad y la potencia que ofrece Spring Framework.

⚠ **Sección introductoria generada por ChatGPT**

Spring Core Annotations

Estas anotaciones forman parte del paquete *'org.springframework.beans.factory.annotation'* y *'org.springframework.context.annotation'*.

- Annotation-based Container Configuration
- Spring Core Annotations - Baeldung

DI-Related Annotations

@Autowired

La anotación `@Autowired` se utiliza para marcar una dependencia que **el motor DI de Spring resolverá e inyectará**. Esta anotación se puede usar en un **constructor**, en un **método *'setter'*** o en un **campo**:

```
// Constructor injection
class Car {
  Engine engine;

  @Autowired
  Car(Engine engine) {
    this.engine = engine;
  }
}
```

A partir de la versión 4.3, no es necesario anotar constructores con `@Autowired` de forma explícita. Sin embargo, si hay dos o más contructores sigue siendo necesario para que Spring sepa cuál de ellos debe utilizar.

Si se utiliza la inyección del constructor, **todos los argumentos del constructor son obligatorios**.

```
// MyService.java
package com.example.demo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class MyService {

    // Inyección por campo
    @Autowired
    private Dependency1 dependency1;

    private Dependency2 dependency2;

    // Inyección por constructor
    @Autowired
    public MyService(Dependency2 dependency2) {
        this.dependency2 = dependency2;
    }

    // Inyección por setter
    @Autowired
    public void setDependency1(Dependency1 dependency1) {
        this.dependency1 = dependency1;
    }

    public void printMessages() {
        System.out.println("Dependency1: " + dependency1.getMessage());
        System.out.println("Dependency2: " + dependency2.getMessage());
    }
}
```

`@Autowired` tiene un argumento booleano llamado `required` con un valor predeterminado de `true`. Este argumento ajusta el comportamiento de Spring cuando no encuentra un *bean* adecuado para conectar. Cuando es verdadero, se lanzará una excepción si no encuentra el *bean*; de lo contrario, no se conecta nada. Sin embargo, se recomienda dejar el valor a `true` (predeterminado) para evitar excepciones de *'null pointer'* al no estar inicializado el *bean* y ser utilizado en el código.

```java
@Autowired(required=false)
public MyService(Dependency2 dependency2) {
    this.dependency2 = dependency2;
}
```

Puede utilizarse `@Inject` en lugar de `@Autowired` en Spring Framework. `@Inject` (y `@Named`) es parte de la especificación estándar JSR-330 de Jakarta EE/CDI, mientras que `@Autowired` es específico de Spring y ofrece funcionalidades adicionales. Para poder utilizar esta anotación, hay que añadir la dependencia ***Jakarta Dependency Injection*** al fichero `pom.xml` o al fichero `build.gradle`:

```xml
<dependency>
    <groupId>jakarta.inject</groupId>
    <artifactId>jakarta.inject-api</artifactId>
    <version>2.0.0</version>
</dependency>
```

La anotación `@Autowired` por sí sola no es suficiente para que Spring gestione la inyección de dependencias. Es crucial que Spring esté configurado para encontrar y manejar los *beans* en el contexto de la aplicación.

Los *beans* son aquellas clases anotadas con `@Component`, `@Service`, `@Repository` o `@Controller`.

Si Spring está configurado mediante clases Java, se utiliza la anotación `@ComponentScan` para indicar a Spring dónde buscar los beans. En una aplicación Spring Boot, la anotación `@SpringBootApplication` ya incluye `@ComponentScan` por defecto, escaneando el paquete donde se encuentra la clase de arranque y sus subpaquetes.

Una vez que los *beans* están registrados en el contexto de la aplicación, ya se puede utilizar `@Autowired` para inyectar estos *beans* en otras partes de la aplicación. La inyección se realiza en el momento en que Spring crea e inicializa los objetos de la clase que necesita las dependencias.

- Javadoc - @Autowired
- Guide to Spring @Autowired - Baeldung
- Constructor Dependency Injection in Spring - Baeldung
- Using @Autowired - Spring Framework
- Using JSR 330 Standard Annotations - Spring Framework

@Bean

La anotación `@Bean` se usa en métodos dentro de una clase anotada con `@Configuration` para registrar y configurar *beans* en el contexto de la aplicación. **Spring llamará a estos métodos** cuando se requiera de una instancia del tipo de retorno del método:

```java
@Configuration
public class AppConfiguration {
  @Bean
  public Engine engine() {
    return new Engine();
  }
}
```

Esta anotación se utiliza básicamente cuando se definen *beans* que no pueden ser anotados directamente, como por ejemplo, cuando se trabaja con **clases de librerías de terceros**.

El *bean* resultante por defecto tiene el mismo nombre que el *'factory method'*. Si se requiere que tenga un nombre diferente, se puede proporcionar un nombre explícito como argumento `name` de la anotación o mediante un alias. Además, se pueden indicar varios nombres que pueden ser utilizados de forma indistinta:

```
@Configuration
public class AppConfiguration {
  @Bean("engine")
  public Engine getEngine() {
    return new Engine();
  }

  @Bean(name = "vehicle")
  public Vehicle getVehicle() {
    return new Vehicle();
  }

  @Bean(name = {"car", "truck"})
  public Vehicle getVehicle() {
    return new Vehicle();
  }
}
```

> **Todos los métodos anotados con `@Bean` deben estar en clases `@Configuration`.**

- Javadoc - @Bean
- Bean Overview - Spring Framework
- Basic Concepts: @Bean and @Configuration - Spring Framework
- Using the @Configuration annotation - Spring Framework

@Qualifier

Se usa la anotación `@Qualifier` junto con `@Autowired` para proporcionar la **identificación** o el **nombre** del *bean* que Spring debe usar en situaciones ambiguas.

```
@Component
class Bike implements Vehicle {}

@Component
class Car implements Vehicle {}
```

En caso de ambigüedad, Spring lanzará la excepción ***NoUniqueBeanDefinitionException***. En el ejemplo anterior, Spring no sabrá que *bean* debe inyectar. En estos casos, se utiliza la anotación `@Qualifier` para indicar a Spring **el *bean* a inyectar**:

```
// Using constructor injection
@Autowired
Biker(@Qualifier("bike") Vehicle vehicle) {
    this.vehicle = vehicle;
}

// Using setter injection
@Autowired
void setVehicle(@Qualifier("bike") Vehicle vehicle) {
    this.vehicle = vehicle;
}

@Autowired
@Qualifier("bike")
```

```
void setVehicle(Vehicle vehicle) {
    this.vehicle = vehicle;
}

// Using field injection
@Autowired
@Qualifier("bike")
Vehicle vehicle;
```

Otra forma de gestionar las ambigüedades es utilizando la anotación `@Primary`.

- Javadoc - @Qualifier
- Fine-tuning Annotation-based Autowiring with Qualifiers - Spring Framework

@Value

Se puede utilizar la anotación `@Value` para inyectar valores de propiedad en *beans*. Es compatible con **constructores**, **métodos *'setter'*** y con **campos**.

```
// Constructor
Engine(@Value("8") int cylinderCount) {
    this.cylinderCount = cylinderCount;
}
```

Por supuesto, inyectar valores estáticos como en el ejemplo no tiene demasiada utilidad.

Podemos usar **cadenas *'placeholder'*** en `@Value` para conectar valores definidos en fuentes externas, por ejemplo, en archivos *".properties"* o *".yaml"*.

Por ejemplo, un valor en un fichero externo *"application.properties"* podría ser:

```
catalog.name=MovieCatalog
```

Podemos inyectar este valor de la siguiente forma:

```
@Component
public class MovieRecommender {

    private final String catalog;

    public MovieRecommender(@Value("${catalog.name}") String catalog) {
        this.catalog = catalog;
    }
}
```

Hay que tener en cuenta que además, se necesitaría la siguiente configuración:

```
@Configuration
@PropertySource("classpath:application.properties")
public class AppConfig { }
```

Si la propiedad no está definida, se puede proporcionar un **valor por defecto**:

```java
@Component
public class MovieRecommender {

    private final String catalog;

    public MovieRecommender(@Value("${catalog.name:defaultCatalog}") String catalog) {
        this.catalog = catalog;
    }
}
```

- Javadoc - @Value
- A Quick Guide to Spring @Value - Baeldung
- Using @Value - Spring Framework

Lenguaje de Expresiones SpEL

El **Lenguaje de Expresiones de Spring (SpEL)** permite evaluar expresiones en tiempo de ejecución dentro del contexto de Spring. Cuando `@Value("#{expression}")` contiene una expresión SpEL, el valor se calculará dinámicamente en tiempo de ejecución.

```java
// Literal expressions
"#{'Hello World'}"  //strings
"#{3.1415926}"      //numeric values (double)
"#{true}"           //boolean
"#{null}"           //null

// Inline list
"#{1,2,3,4}"                //list of number
"#{'a','b'},{'x','y'}}"    //list of list

// Inline maps
"#{name:'Nikola',dob:'10-July-1856'}"
"#{name:{first:'Nikola',last:'Tesla'},dob:{day:10,month:'July',year:1856}}" //map of maps

// Invoking Methods
"#{'abc'.length()}" //evaluates to 3
"#{say('hello')}"   //'say' is a method in the class and it has a string parameter

// Invoking Methods from another bean
"#{otherBean}"          // Bind a bean
"#{otherBean.property}" // Property from bean
"#{otherBean.method()}" // Invoke method from bean

// Seguridad de tipos con el operador ?.
"#{otherBean?.property}" // Antes de acceder a property se valida que otherBean != null
```

SpEL es útil para evaluar expresiones dinámicas en configuraciones de Spring, como valores de propiedades, condiciones, y más.

Se puede usar en diversas anotaciones y configuraciones de Spring, como `@Value`, `@ConditionalOnExpression`, entre otras.

- Spring Expression Language (SpEL) - Spring Framework

@DependsOn

La anotación `@DependsOn` se puede utilizar para hacer que Spring **inicialice otros *beans* antes del *bean* anotado**. Normalmente, este comportamiento de inicialización y carga es automático y se basa en las dependencias explícitas entre

beans. Spring, de forma predeterminada, gestiona el ciclo de vida de los *beans* y organiza su orden de inicialización según las necesidades.

Solo necesitamos esta anotación cuando **las dependencias están implícitas**, como por ejemplo, la carga de un controlador JDBC o la inicialización de variables estáticas.

Las **dependencias implícitas** son aquellas que no están declaradas directamente entre beans (por ejemplo, usando `@Autowired`) pero que son necesarias para el correcto funcionamiento de la aplicación.

Podemos usar `@DependsOn` en la clase dependiente especificando los nombres de los *beans* de dependencia. El argumento de valor de la anotación necesita una matriz que contenga los nombres de los *beans* de dependencia:

```
@DependsOn("engine")
class Car implements Vehicle {}
```

Alternativamente, si se define un bean con la anotación `@Bean`, el *'factory method'* debería anotarse con `@DependsOn`:

```
@Bean
@DependsOn("fuel")
Engine engine() {
    return new Engine();
}
```

- Javadoc - @DependsOn

@Lazy

La anotación `@Lazy` se utiliza a nivel de **clase** o **método** cuando queremos inicializar un *bean* de forma diferida o perezosa.

De forma predeterminada, Spring crea todos los *beans* con el alcance *"singleton"* por defecto al inicio del contexto de la aplicación de manera ansiosa o *'eager'*.

Sin embargo, hay casos en los que necesitamos crear un *bean* cuando se solicita, no al iniciar la aplicación. En estos casos se utilizará la anotación `@Lazy`.

Esta anotación tiene un argumento con el valor predeterminado de 'true'. Es útil para anular el comportamiento predeterminado.

Por ejemplo, marcar *beans* para que se carguen inmediatamente cuando la configuración global es diferida, o configurar métodos específicos de `@Bean` para carga inmediata en una clase `@Configuration` marcada con `@Lazy`:

```
@Configuration
@Lazy
class VehicleFactoryConfig {

    @Bean
    @Lazy(false)
    Engine engine() {
        return new Engine();
    }
}
```

- Javadoc - @Lazy
- A Quick Guide to the Spring @Lazy Annotation

@Lookup

Un método anotado con `@Lookup` le indica a Spring que devuelva una instancia del tipo de retorno del método cuando lo invoquemos.

La anotación `@Lookup` en Spring permite la **obtención dinámica de *beans*** desde el contenedor de Spring. Se utiliza para resolver y obtener instancias de *beans* de manera perezosa, especialmente útil en combinación con *beans 'prototype'* o cuando se requiere obtener una nueva instancia de un bean en cada solicitud. Spring reemplaza el método anotado con `@Lookup` para devolver una nueva instancia del bean solicitado cada vez que el método es invocado.

Esta anotación es útil en casos donde la inyección directa no es posible o no es adecuada, como por ejemplo cuando se inyecta un *bean 'prototype'* en un *bean 'singleton'*.

La explicación es que cuando se inyecta un *bean 'singleton'*, Spring crea una sola instancia de ese *bean* y la reutiliza en toda la aplicación. Si se intenta inyectar un *bean 'prototype'* en un *bean 'singleton'* mediante inyección directa, Spring solo inyectará una instancia única del *bean 'prototype'*, es decir, la misma instancia será utilizada en cada solicitud, lo cual no es el comportamiento deseado.

La anotación `@Lookup` permitirá obtener una nueva instancia del *bean 'prototype'* en cada invocación del método anotado.

```
// bean 'singleton'
@Component
public class MiBeanSingleton {

    @Lookup
    public MiBeanPrototype obtenerBeanPrototype() {
        // Spring reemplaza esta implementación para devolver una nueva instancia de 'MiBeanPrototype'
        return null;
    }

    public void usarBeanPrototype() {
        MiBeanPrototype beanPrototype = obtenerBeanPrototype();
        // Trabajar con la nueva instancia del bean prototype
    }
}

// bean 'prototype'
@Component
@Scope("prototype")
public class MiBeanPrototype {
    // Bean creado con alcance prototype
}
```

- Javadoc - @Lookup
- @Lookup Annotation in Spring

@Primary

A veces se necesita definir **múltiples *beans* del mismo tipo**. En estos casos, la inyección no tendrá éxito porque Spring no sabe qué *bean* necesitamos.

Una opción para manejar este escenario es marcar todos los puntos de conexión con `@Qualifier` y especificar el nombre del *bean* requerido.

Sin embargo, la mayoría de las veces se necesita un *bean* específico y rara vez los otros. Se puede emplear `@Primary` para simplificar este caso: si se marca el *bean* usado más **frecuentemente** con `@Primary`, será elegido en los puntos de inyección no calificados:

```
@Component
@Primary
class Car implements Vehicle {}

@Component
class Bike implements Vehicle {}

@Component
class Driver {
    @Autowired
    Vehicle vehicle;
}

@Component
class Biker {
    @Autowired
    @Qualifier("bike")
    Vehicle vehicle;
}
```

En el ejemplo anterior, *'Car'* es el vehículo principal. Por lo tanto, en la clase *'Driver'*, Spring inyecta un *bean* de tipo *'Car'*. Por supuesto, en el *bean 'Biker'*, el valor del campo '*vehicle'* será un objeto de tipo *'Bike'* porque está calificado.

- Javadoc - @Primary
- Fine-tuning Annotation-based Autowiring with Qualifiers - Spring Framework

@Scope

Usamos `@Scope` para definir el **ámbito o alcance** de una clase `@Component` o una definición de `@Bean`. Los ámbitos pueden ser:

- ***singleton*** - ConfigurableBeanFactory.SCOPE_SINGLETON
- ***prototype*** - ConfigurableBeanFactory.SCOPE_PROTOTYPE
- ***request*** - WebApplicationContext.SCOPE_REQUEST
- ***session*** - WebApplicationContext.SCOPE_SESSION
- ***application*** - WebApplicationContext.SCOPE_APPLICATION
- ***websocket***
- **ámbito personalizado**

```
@Component
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
class Engine {}
```

> El ámbito o alcance por **defecto** es ***'singleton'***.

El ámbito ***'singleton'*** significa que Spring crea **una única instancia** del *bean* en el contexto de la aplicación y la reutiliza en toda la aplicación. Por tanto, independientemente de cuántas veces se inyecte un *bean*, Spring inyectará la misma instancia.

- Javadoc - @Scope
- Bean Scopes - Spring Framework

Context Configuration Annotations

@Profile

La anotación `@Profile` es útil para cargar beans solo en entornos específicos (*'dev'*, *'test'*, *'prod'*) o bajo condiciones especiales. Podemos configurar el nombre del perfil como argumento de la anotación:

```
@Component
@Profile("sportDay")
class Bike implements Vehicle {}
```

Se puede aplicar a clases o métodos para controlar la configuración en función de los perfiles activos. Si se utiliza a nivel de clase en una clase de configuración por ejemplo, se cargarán todos los *beans* del perfil activo. Cuando un perfil no está activo, los *beans* anotados con `@Profile` correspondiente no se crean, lo que evita la carga innecesaria de componentes en entornos donde no son necesarios.

Además, esta anotación puede combinarse a nivel de clase y a nivel de método en una misma clase:

```
@Configuration
@Profile("dev")
public class MixedConfig {

    // Se carga con el perfil "dev"
    @Bean
    public MyBean myDevBean() {
        return new MyBean("Dev Bean");
    }

    // Se carga con el perfil "test"
    @Bean
    @Profile("test")
    public AnotherBean anotherTestBean() {
        return new AnotherBean("Another Test Bean");
    }
}
```

Por último, se pueden combinar perfiles mediante operadores lógicos como `AND`, `OR` y `NOT`:

```
// Indica que el bean se carga cuando dev está activo y prod no lo está.
@Configuration
@Profile({"dev", "!prod"})
public class MixedConfig {
    // ...
}
```

Para activar un perfil, se puede hacer de varias maneras. La más común sería usando las propiedades de configuración `spring.profiles.active` y/o `spring.profiles.default` en el fichero *"application.properties"* o *"application.yml"*. Otras formas sería con parámetros de línea de comandos, o mediante variables de entorno. Se pueden definir varios perfiles separándolos con comas:

```
// Determina los perfiles activos
spring.profiles.active=dev,feature-x

// Si no hay perfiles activos, Spring se fija en el perfil por defecto
spring.profiles.default=prod
```

Si no se establece ninguna de las dos propiedades anteriores, no habrá perfiles activos y **Spring sólo creará los *beans* que no tengan perfil**.

- Javadoc - @Profile
- Spring Profiles - Baeldung
- Environment Abstraction - Spring Framework
- Profiles - Spring Boot

@ActiveProfiles

Spring dispone de la anotación `@ActiveProfiles`, que es una anotación compatible con JUnit 5, para declarar los perfiles activos a usar cuando se carga un contexto de aplicación en las clases de prueba.

Por lo tanto, si anotamos una clase de prueba con la anotación `@ActiveProfiles` y establecemos el atributo `value` a `test`, todas las pruebas en la clase usarán el perfil `test`:

```
@SpringBootTest(classes = ActiveProfileApplication.class)
@ActiveProfiles(value = "test")
public class TestActiveProfileUnitTest {

    @Value("${profile.property.value}")
    private String propertyString;

    @Test
    void whenTestIsActive_thenValueShouldBeKeptFromApplicationTestYaml() {
        Assertions.assertEquals("This the the application-test.yaml file", propertyString);
    }
}
```

La anotación `@ActiveProfiles` permite definir perfiles activos específicos para pruebas. Se puede usar con un solo perfil o múltiples perfiles si se necesita combinar configuraciones, por ejemplo: `@ActiveProfiles({"test", "dev"})`.

Si se activan múltiples perfiles, el orden en que se especifiquen puede ser relevante en situaciones donde las propiedades se superpongan.

- Javadoc - @ActiveProfiles
- @ActiveProfiles - Spring Framework
- Execute Tests Based on Active Profile With JUnit 5 - Baeldung

@Import

La anotación `@Import` en Spring se utiliza para importar configuraciones adicionales a la configuración principal de la aplicación.

Es decir, si tenemos una clase anotada con `@Configuration` que define *beans* y/o configuraciones específicas para la aplicación, se puede usar `@Import` para incluir esta configuración en otra clase anotada con `@Configuration` y que corresponde con la configuración principal:

```
@Configuration
@Import(MyAdditionalConfig.class)
public class MainConfig {
    // Configuración principal de la aplicación
}
```

Además, `@Import` permite utilizar **clases específicas anotadas con `@Configuration` sin escaneo de componentes**, lo cual es útil para tener control explícito sobre qué configuraciones y *beans* están disponibles en la aplicación. Esas clases específicas se pueden proporcionar como argumento de la anotación:

```
@Configuration
@Import({DataSourceConfig.class, SecurityConfig.class})
public class MainConfig {
    // Configuración principal de la aplicación
}
```

Por último, `@Import` también permite importar clases que no están anotadas con `@Configuration`, asegurando que estas clases estén disponibles en el contexto de la aplicación. Importar clases no anotadas con `@Configuration` puede ser útil para asegurar que ciertas utilidades o servicios auxiliares estén disponibles en el contexto de la aplicación, incluso si no forman parte directa de la configuración de Spring.

```
@Configuration
@Import({MyUtilityClass.class, AnotherHelper.class})
public class MainConfig {
    // Configuración principal de la aplicación
}
```

Aquí, *'MyUtilityClass'* y *'AnotherHelper'* son clases normales que no necesariamente son configuraciones de Spring, pero que pueden ser útiles en el contexto de la aplicación.

- Javadoc - @Import

@ImportResource

Para importar **configuraciones XML** podemos utilizar la anotación `@ImportResource`. Se puede especificar la ubicación del archivo XML mediante el argumento de la anotación:

```
@Configuration
@ImportResource("classpath:/annotations.xml")
class VehicleFactoryConfig {}
```

Se pueden especificar múltiples ubicaciones de archivos XML usando `@ImportResource`, separando las ubicaciones con comas:

```
@Configuration
@ImportResource({"classpath:/annotations.xml", "classpath:/more-config.xml"})
class VehicleFactoryConfig { }
```

La anotación `@ImportResource` se utiliza **exclusivamente** para importar configuraciones desde archivos XML dentro del contexto de Spring. Esto puede ser útil en proyectos que aún dependen de configuraciones XML o cuando se integran con sistemas existentes.

En contraste, `@Import` se utiliza para importar configuraciones y componentes de **otras clases de configuración**, ya sean basadas en anotaciones o en XML, pero principalmente se usa con configuraciones basadas en anotaciones.

- Javadoc - @ImportResource

@PropertySource

La anotación `@PropertySource` se utiliza para definir archivos de propiedades ('.properties' o '.yml') para la configuración de la aplicación. Se debe indicar la ubicación del fichero de propiedades a cargar mediante un argumento de la anotación:

```
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;

@Configuration
@PropertySource("classpath:config.properties")
public class AppConfig {
    // Configuración adicional de la aplicación
}
```

Estos archivos pueden contener configuraciones como URLs de bases de datos, rutas de archivos, configuraciones de conexión, etc. Los archivos en `src/main/resources` se incluyen en el **classpath** durante la construcción del proyecto, además de todas las dependencias incluidas con Maven o Gradle.

Las propiedades cargadas se integran con el `Environment` de Spring, lo que permite acceder a ellas desde cualquier parte de la aplicación:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.env.Environment;
import org.springframework.stereotype.Component;

@Component
public class MyComponent {

    @Autowired
    private Environment env;

    public void someMethod() {
        String dbUrl = env.getProperty("database.url");
        // Default value
        String user = env.getProperty("database.user", "userByDefault");
        // Convertir un valor a int
        int connectionCount = env.getProperty("database.count", Integer.class, 30);
        // ...
    }
}
```

El objeto `Environment` está **disponible en el contexto de la aplicación** y contiene propiedades predeterminadas proporcionadas por Spring, como **valores de configuración** de la JVM (por ejemplo, `java.home`) o **propiedades del sistema**.

Mediante este objeto se pueden consultar los perfiles activos y el perfil actual del entorno utilizando este objeto. Esto es útil para ajustar el comportamiento de la aplicación según el perfil activo.

Otra forma de acceder a una propiedad podría ser utilizar directamente la anotación `@Value` . Esta anotación `@Value` es útil para inyectar valores directamente en campos de clase, especialmente para casos simples.

```
@Component
public class MyComponent {
    @Value("${database.url}")
    private String dbUrl;
}
```

La anotación `@PropertySource` aprovecha la función de anotaciones repetidas de Java 8, lo que significa que podemos marcar una clase con ella varias veces:

```
@Configuration
@PropertySource("classpath:/annotations.properties")
@PropertySource("classpath:/vehicle-factory.properties")
class VehicleFactoryConfig {}
```

En caso de **conflicto de claves**, prevalecerá el **último archivo cargado**; es decir, el último archivo sobreescribe el valor de la clave si hay una clave repetida en ambos archivos.

- Javadoc - @PropertySource
- Javadoc - Environment
- Resources

@PropertySources

La anotación `@PropertySources` es un contenedor que se utiliza para especificar múltiples configuraciones de `@PropertySource`:

```
@Configuration
@PropertySources({
    @PropertySource("classpath:/annotations.properties"),
    @PropertySource("classpath:/vehicle-factory.properties")
})
class VehicleFactoryConfig {}
```

Tenga en cuenta que desde Java 8 se puede lograr el mismo resultado con la función de anotaciones repetidas.

- Javadoc - @PropertySources

@ConfigurationProperties

La anotación `@ConfigurationProperties` en Spring Boot permite mapear propiedades externas (como propiedades en archivos `.properties` o `.yaml`) a una clase Java, facilitando el acceso a configuraciones de manera **estructurada y tipada** dentro de una aplicación Spring.

Esta anotación acepta un prefijo que indica qué propiedades del archivo se deben mapear a la clase. Por ejemplo, al utilizar el prefijo "app", todas las propiedades que comiencen con "app." se mapearán automáticamente.

Para que Spring reconozca y cargue las propiedades en la clase, se debe **habilitar el soporte** para `@ConfigurationProperties`. Esto se suele hacer con la anotación `@EnableConfigurationProperties` en una clase de configuración:

```
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.context.annotation.Configuration;

@Configuration
@EnableConfigurationProperties(AppProperties.class)
public class AppConfig {
    // Otras configuraciones
}
```

Por ejemplo, un archivo de propiedades llamado `application.properties`:

```
app.name=MyApp
app.version=1.0.0
```

```
app.features.enabled=true
```

Con el prefijo "app", las propiedades anteriores se mapean a la siguiente clase:

```
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;

@Component
@ConfigurationProperties(prefix = "app")
public class AppProperties {

    private String name;
    private String version;
    private boolean featuresEnabled;

    // Getters y setters
}
```

El objeto `AppProperties` puede luego inyectarse y utilizarse en cualquier componente de Spring:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class AppService {

    private final AppProperties appProperties;

    @Autowired
    public AppService(AppProperties appProperties) {
        this.appProperties = appProperties;
    }

    public void printProperties() {
        System.out.println("App Name: " + appProperties.getName());
        System.out.println("App Version: " + appProperties.getVersion());
        System.out.println("Features Enabled: " + appProperties.isFeaturesEnabled());
    }
}
```

- Javadoc - @ConfigurationProperties
- Using Environment Variables in Spring Boot's Properties Files - Baeldung
- Configuring System Environment Properties - Spring Boot

Spring Web Annotations

Estas anotaciones forman parte del paquete *'org.springframework.web.bind.annotation'*.

- Spring Web MVC - Spring Framework
- Spring Web Annotations - Baeldung

@RestController

La anotación `@RestController` es una meta-anotación que combina las anotaciones `@Controller` y `@ResponseBody` para todos los métodos de la clase, lo que significa que los valores retornados por estos métodos se serializan directamente en la respuesta HTTP (generalmente en formato JSON o XML).

```
@Controller
@ResponseBody
class VehicleRestController {
    // ...
}
```

```
@RestController
class VehicleRestController {
    // ...
}
```

La principal diferencia entre `@Controller` y `@RestController` es que:

- `@Controller`: Se usa para los controladores tradicionales en aplicaciones web, donde los métodos de los controladores retornan vistas o datos que deben ser procesados por una vista (como un archivo HTML).
- `@RestController`: Se usa para servicios RESTful y APIs, donde los métodos retornan directamente datos que se serializan en el formato adecuado (como JSON) sin necesidad de una vista intermedia.
- Javadoc - @RestController
- Annotated Controllers - Spring Framework

@CrossOrigin

La anotación `@CrossOrigin` en Spring Framework es utilizada para configurar las políticas de intercambio de recursos entre diferentes dominios (**CORS**, por sus siglas en inglés - *"Cross-Origin Resource Sharing"*). CORS es un mecanismo de seguridad implementado por los navegadores web para restringir las solicitudes HTTP que se originan desde un dominio diferente al del servidor de destino.

La anotación `@CrossOrigin` habilita la comunicación entre dominios para los métodos del controlador de solicitudes anotados:

```
@RestController
@RequestMapping("/api")
public class MyController {

    @CrossOrigin(origins = "http://allowed-origin.com", methods = {RequestMethod.GET, RequestMethod.POST})
    @GetMapping("/data")
    public ResponseEntity<String> getData() {
        // Lógica para obtener datos
        return ResponseEntity.ok("Data fetched successfully");
    }
}
```

En este ejemplo, se permite el acceso a http://allowed-origin.com para los métodos GET y POST del endpoint `/data`. Si la anotación se coloca en una clase, se aplica a todos los métodos del controlador que contiene.

También se puede habilitar CORS globalmente para todos los controladores mediante una configuración global en una clase que implemente `WebMvcConfigurer`:

```
@Configuration
public class CorsConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**")
                .allowedOrigins("http://domain1.com", "https://domain2.com")
                .allowedMethods("GET", "POST", "PUT", "DELETE")
                .allowedHeaders("header1", "header2")
                .exposedHeaders("header3", "header4")
                .allowCredentials(true)
                .maxAge(3600);
    }
}
```

Aquí, se permiten solicitudes desde http://domain1.com y https://domain2.com con los métodos HTTP especificados. Los encabezados permitidos y expuestos se pueden definir, y se puede configurar el uso de credenciales.

Si se está utilizando Spring Security en la aplicación, es fundamental asegurarse de que la configuración de **CORS** no entre en conflicto con las políticas de seguridad definidas en Spring Security. La configuración de **CORS** en Spring Security se realiza a través de la clase `WebSecurityConfigurerAdapter`, y se puede integrar con la configuración global de **CORS** para asegurar una política coherente.

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.cors()
            .and()
            .authorizeRequests()
            .anyRequest().authenticated();
    }

    @Bean
    public CorsConfigurationSource corsConfigurationSource() {
        CorsConfiguration configuration = new CorsConfiguration();
        configuration.setAllowedOrigins(Arrays.asList("http://domain1.com", "https://domain2.com"));
        configuration.setAllowedMethods(Arrays.asList("GET", "POST", "PUT", "DELETE"));
        configuration.setAllowedHeaders(Arrays.asList("header1", "header2"));
        configuration.setExposedHeaders(Arrays.asList("header3", "header4"));
        configuration.setAllowCredentials(true);
        configuration.setMaxAge(3600L);

        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration("/**", configuration);
        return source;
    }
}
```

Este ejemplo configura **CORS** en la seguridad de Spring para que coincida con la configuración global y permita solicitudes desde los dominios especificados con los métodos y encabezados adecuados.

- Javadoc - @CrossOrigin
- CORS with Spring - Baeldung
- CORS - Spring Framework

Request Handling Annotations

@RequestMapping

En Spring Framework, la anotación `@RequestMapping` se utiliza para marcar **métodos manejadores de peticiones** dentro de clases anotadas con `@Controller` o `@RestController`.

Permite mapear solicitudes HTTP a métodos de controlador específicos en función de varios criterios. Los elementos clave de `@RequestMapping` son:

- `path` (o sus alias `name` y `value`): indica a qué URL está mapeado el método.
- `method`: define los métodos HTTP compatibles.
- `params`: filtra las peticiones basándose en la presencia, ausencia o valor de parámetros HTTP.
- `headers`: filtra las peticiones basándose en la presencia, ausencia o valor de cabeceras HTTP.
- `consumes`: especifica los tipos de medios que el método puede consumir en el cuerpo de la petición HTTP.
- `produces`: especifica los tipos de medios que el método puede producir en el cuerpo de la respuesta HTTP.

```
@Controller
class VehicleController {

    @RequestMapping(value = "/vehicles/home", method = RequestMethod.GET)
    String home() {
        return "home";
    }
}
```

Podemos proporcionar configuraciones predeterminadas para todos los métodos manejadores en una clase `@Controller` aplicando `@RequestMapping` a nivel de clase. La URL base especificada a nivel de clase se combinará con los *paths* definidos en los métodos individuales:

Por ejemplo, la siguiente configuración tiene el mismo efecto que la configuración del ejemplo anterior:

```
@Controller
@RequestMapping(value = "/vehicles", method = RequestMethod.GET)
class VehicleController {

    @RequestMapping("/home")
    String home() {
        return "home";
    }
}
```

Desde la **versión 4.3** de Spring, se introdujeron las anotaciones específicas para métodos HTTP como:

- `@GetMapping`
- `@PostMapping`
- `@PutMapping`
- `@DeleteMapping`
- `@PatchMapping`

Estas meta-anotaciones son variantes de `@RequestMapping` y permiten una configuración más concisa y específica para cada tipo de solicitud HTTP:

```
@GetMapping("/vehicles/home")
public String home() {
    return "home";
}
```

- Javadoc - @RequestMapping
- Mapping Requests - Spring Framework

@HttpExchange

Es una anotación introducida en Spring Framework 6 que nos permite definir interfaces HTTP de **forma declarativa**. Si bien el objetivo principal de `@HttpExchange` es abstraer el código del cliente HTTP con un proxy generado, la interfaz HTTP sobre la que se colocan estas anotaciones es un contrato neutral para el uso del cliente frente al servidor

```
@HttpExchange("/persons")
interface PersonService {

  @GetExchange("/{id}")
  Person getPerson(@PathVariable Long id);

  @PostExchange
  void add(@RequestBody Person person);
}
```

El controlador implementa la interfaz declarada:

```
@RestController
class PersonController implements PersonService {

  public Person getPerson(@PathVariable Long id) {
    // ...
  }

  @ResponseStatus(HttpStatus.CREATED)
  public void add(@RequestBody Person person) {
    // ...
  }
}
```

- Javadoc - @HttpExchange
- @HttpExchange - Spring Framework

@RequestBody

La anotación `@RequestBody` en Spring Framework se utiliza para **mapear el cuerpo de una solicitud HTTP a un objeto Java**. Esta anotación es particularmente útil en los métodos de controladores que manejan solicitudes POST, PUT, o PATCH, donde los datos se envían en el cuerpo de la solicitud y no en la URL.

```
@PostMapping("/save")
void saveVehicle(@RequestBody Vehicle vehicle) {
```

```
    // ...
}
```

En el ejemplo anterior, el cuerpo de la solicitud HTTP se deserializa automáticamente en una instancia de la clase `Vehicle`. El tipo de deserialización se basa en el tipo de contenido de la solicitud, como `application/json`, `application/xml`, etc.

Spring utiliza un convertidor adecuado, como Jackson para JSON, para realizar esta conversión.

- Javadoc - @RequestBody
- @RequestBody - Spring Framework

@PathVariable

La anotación `@PathVariable` se utiliza para **vincular un argumento de método a una variable de ruta** en una URI. Si el nombre de la variable en la ruta coincide con el nombre del argumento del método, no es necesario especificarlo en la anotación:

```
import org.springframework.web.bind.annotation.*;

// "http://example.com/users/{userId}"
@RestController
@RequestMapping("/users")
public class UserController {

    @GetMapping("/{userId}")
    public String getUserById(@PathVariable Long userId) {
        return "Obteniendo usuario con ID: " + userId;
    }
}
```

Si los nombres no coinciden o si se requiere especificarlo explícitamente, se puede hacer usando `@PathVariable` con el nombre de la variable de ruta:

```
import org.springframework.web.bind.annotation.*;

// "http://example.com/products/{category}/{productId}"
@RestController
@RequestMapping("/products")
public class ProductController {

    @GetMapping("/{category}/{productId}")
    public String getProductDetails(
            @PathVariable("category") String category,
            @PathVariable("productId") Long productId) {
        return "Detalles del producto: categoría = " + category + ", ID = " + productId;
    }
}
```

Si una variable de ruta es opcional, se puede indicar estableciendo el argumento de la anotación como `'required = false'`.

Cuando se establece `'required = false'`, la variable de ruta puede ser omitida en la URI. Si se omite, el valor del argumento será `null` (en el caso de objetos) o el valor por defecto para tipos primitivos, como 0 para `long`:

```
@RequestMapping("/{id}")
Vehicle getVehicle(@PathVariable(required = false) long id) {
```

```
    // ...
}
```

- Javadoc - @PathVariable

@RequestParam

La anotación `@RequestParam` se utiliza para **acceder a los parámetros de solicitud HTTP**:

```
// "http://example.com/users?id=123"
@GetMapping("/users")
public String getUser(@RequestParam String id) {
    // El valor de id será "123"
    return "User ID: " + id;
}
```

La anotación `@RequestParam` ofrece opciones de configuración similares a las de `@PathVariable`.

Además, permite especificar un valor predeterminado para el parámetro en caso de que no se encuentre en la solicitud o esté vacío. Esto se logra mediante el atributo `defaultValue`:

```
@RequestMapping("/buy")
Car buyCar(@RequestParam(defaultValue = "5") int seatCount) {
    // ...
}
```

- Javadoc - @RequestParam
- @RequestParam - Spring Framework

@ModelAttribute

La anotación `@ModelAttribute` en Spring MVC se utiliza para **vincular un método o parámetro de método a un atributo del modelo**. Esto es especialmente útil en controladores de Spring MVC, donde se manejan solicitudes HTTP y se preparan datos que serán utilizados en la vista.

Cuando se usa `@ModelAttribute` en un **parámetro de método**, Spring enlaza automáticamente los datos de la solicitud HTTP a ese objeto, facilitando la recepción de datos de formularios o solicitudes HTTP. Con esta anotación, también podemos acceder a elementos que ya están presentes en el modelo del controlador MVC:

```
// Se indica la clave del modelo en la anotación '@ModelAttribute'
@PostMapping("/assemble")
void assembleVehicle(@ModelAttribute("vehicle") Vehicle vehicleInModel) {
    // ...
}

// El nombre del argumento del método coincide con la clave del modelo
@PostMapping("/paint")
void paintVehicle(@ModelAttribute Vehicle vehicle) {
    // ...
}
```

Si un método de un controlador se anota con `@ModelAttribute`, este método se ejecuta **antes** que cualquier método anotado con `@RequestMapping` en el mismo controlador. Esto permite preparar y agregar atributos al modelo que serán utilizados en la vista. Spring agrega automáticamente el valor de retorno de este método al modelo:

```java
// Se indica la clave del modelo en la anotación '@ModelAttribute'
@ModelAttribute("vehicle")
Vehicle getVehicle() {
    // ...
}

// El nombre del método coincide con la clave del modelo
@ModelAttribute
Vehicle vehicle() {
    // ...
}
```

En Spring MVC, para procesar correctamente los datos enviados desde un formulario HTML o una solicitud HTTP POST, es **esencial enlazar** esos datos a un objeto del modelo utilizando `@ModelAttribute`. Si no se realiza este enlace explícitamente, Spring no podrá asignar los datos del formulario a ningún objeto y, por lo tanto, no estarán disponibles para su uso en el controlador.

El uso de `@ModelAttribute` también permite la validación automática de los datos de entrada mediante anotaciones como `@Valid` o `@Validated`, lo que asegura que los datos recibidos cumplen con las reglas de validación antes de procesarse.

En el siguiente ejemplo, se muestra un formulario en Spring:

```jsp
<form:form method="POST" action="/spring-mvc-basics/addEmployee"
  modelAttribute="employee">
    <form:label path="name">Name</form:label>
    <form:input path="name" />

    <form:label path="id">Id</form:label>
    <form:input path="id" />

    <input type="submit" value="Submit" />
</form:form>
```

En el controlador, se redirige a una vista JSP pero antes, se recuperan los datos enviados desde el formulario y se añaden al modelo:

```java
@Controller
@ControllerAdvice
public class EmployeeController {

    private Map<Long, Employee> employeeMap = new HashMap<>();

    @RequestMapping(value = "/addEmployee", method = RequestMethod.POST)
    public String submit(
      @ModelAttribute("employee") Employee employee,
      BindingResult result, ModelMap model) {
        if (result.hasErrors()) {
            return "error";
        }
        model.addAttribute("name", employee.getName());
        model.addAttribute("id", employee.getId());

        employeeMap.put(employee.getId(), employee);

        return "employeeView";
    }

    @ModelAttribute
    public void addAttributes(Model model) {
        model.addAttribute("msg", "Welcome to the Netherlands!");
```

```
    }
}
```

- Javadoc - @ModelAttribute
- Spring MVC and the @ModelAttribute Annotation - Baeldung
- @ModelAttribute - Spring Framework
- Model - Spring Framework
- Validation - Spring Framework

@CookieValue

La anotación `@CookieValue` en Spring Framework se utiliza para **enlazar el valor de una *cookie* HTTP específica a un parámetro de método** en un controlador de Spring MVC. Esto es útil cuando se necesita acceder al valor de una *cookie* enviada por el cliente en una solicitud HTTP.

```
@RestController
@RequestMapping("/api")
public class MyController {

    @GetMapping("/getCookie")
    public ResponseEntity<String> getCookieValue(@CookieValue(name = "myCookie", defaultValue = "default") String cookie
        // Aquí puedes usar cookieValue, que contendrá el valor de la cookie "myCookie"
        return ResponseEntity.ok("Value of 'myCookie': " + cookieValue);
    }
}
```

Por defecto, la presencia de la *cookie* es **requerida**. Si la *cookie* no está presente, Spring lanzará una excepción `MissingCookieValueException`. Para indicar que la *cookie* no es obligatoria, puedes usar el argumento `'required = false'` en la anotación:

```
@GetMapping("/optionalCookie")
public ResponseEntity<String> getOptionalCookie(@CookieValue(name = "optionalCookie", required = false) String optionalC
    // Si la cookie "optionalCookie" no está presente, optionalCookie será null
    return ResponseEntity.ok("Value of 'optionalCookie': " + optionalCookie);
}
```

También es posible recuperar múltiples *cookies* en un solo método del controlador utilizando `@CookieValue` varias veces:

```
@GetMapping("/getCookies")
public ResponseEntity<String> getCookies(@CookieValue(name = "cookie1", defaultValue = "default1") String cookie1,
                                         @CookieValue(name = "cookie2", defaultValue = "default2") String cookie2) {
    // Aquí puedes usar cookie1 y cookie2, que contendrán los valores de las cookies "cookie1" y "cookie2" respectivamen
    return ResponseEntity.ok("Value of 'cookie1': " + cookie1 + ", Value of 'cookie2': " + cookie2);
}
```

- Javadoc - @CookieValue
- @CookieValue - Spring Framework

@RequestHeader

La anotación `@RequestHeader` en Spring Framework se utiliza para **enlazar el valor de una cabecera HTTP específica a un parámetro de método** en un controlador de Spring MVC. Esto es útil cuando necesitas acceder a valores específicos de las cabeceras HTTP enviadas por el cliente en una solicitud.

```
@RestController
@RequestMapping("/api")
public class MyController {

    @GetMapping("/getUserAgent")
    public ResponseEntity<String> getUserAgent(@RequestHeader("User-Agent") String userAgent) {
        // Aquí puedes usar userAgent, que contendrá el valor de la cabecera "User-Agent"
        return ResponseEntity.ok("User-Agent header value: " + userAgent);
    }
}
```

Por defecto, Spring espera que **la cabecera esté presente** en la solicitud. Si la cabecera no se encuentra, se generará una excepción `MissingRequestHeaderException`. Para manejar este caso y proporcionar un valor predeterminado si la cabecera está ausente, puedes usar el atributo `defaultValue` de la anotación:

```
@GetMapping("/getCustomHeader")
public ResponseEntity<String> getCustomHeader(@RequestHeader(value = "X-Custom-Header", defaultValue = "defaultValue") S
    // Aquí puedes usar customHeader, que contendrá el valor de la cabecera "X-Custom-Header" o "defaultValue" si no est
    return ResponseEntity.ok("X-Custom-Header value: " + customHeader);
}
```

En este ejemplo, si la cabecera `X-Custom-Header` no está presente en la solicitud, la variable `customHeader` tomará el valor "defaultValue" en lugar de generar una excepción.

- Javadoc - @RequestHeader
- @RequestHeader - Spring Framework

Response Handling Annotations

@ResponseBody

La anotación `@ResponseBody` en Spring Framework se utiliza para indicar que **el valor retornado por un método de controlador debe ser vinculado directamente a la respuesta HTTP**. En otras palabras, Spring convierte automáticamente el objeto retornado en un formato específico (como JSON o XML) usando convertidores de medios configurados.

Es decir, cuando un método de controlador anotado con `@ResponseBody` se invoca y retorna un objeto, Spring convierte automáticamente ese objeto en un formato específico para la respuesta HTTP. La conversión del objeto a formato se realiza a través de un convertidor de medios (*'media converter'*), que depende de la configuración de Spring y de las bibliotecas disponibles en el classpath.

Es útil cuando se está construyendo una API RESTful y se desea retornar objetos como JSON o XML en lugar de vistas HTML.

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class ExampleController {

    @GetMapping("/hello")
    @ResponseBody
```

```
    public String helloWorld() {
        return "Hello, World!";
    }

    @GetMapping("/user")
    @ResponseBody
    public User getUser(@RequestParam String username) {
        // Supongamos que aquí se obtiene un usuario de una base de datos
        User user = userRepository.findByUsername(username);
        return user;
    }
}
```

Si se anota una clase con `@ResponseBody`, todos los métodos de la clase también se verán afectados. Sin embargo, `@RestController` combina `@Controller` y `@ResponseBody`, aplicando `@ResponseBody` a todos los métodos de la clase automáticamente.

Para que la conversión funcione correctamente, asegúrate de tener las bibliotecas adecuadas en el *classpath*, como Jackson para JSON o JAXB para XML. En caso de errores, se pueden usar mecanismos como `@ExceptionHandler` para manejar excepciones y devolver respuestas adecuadas.

Además, si necesitas una conversión personalizada, puedes configurar convertidores de medios en la configuración de Spring.

- Javadoc - @ResponseBody
- @ResponseBody - Spring Framework

@ExceptionHandler

La anotación `@ExceptionHandler` en Spring Framework se utiliza para **manejar excepciones generadas por métodos de controladores** de solicitudes. Puedes declarar un método de manejo de errores personalizado utilizando esta anotación, y Spring invocará este método cuando se detecte una excepción especificada.

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class ExampleController {

    @GetMapping("/example")
    public String exampleMethod() {
        // Simulamos una excepción
        throw new IllegalArgumentException("Invalid argument");
    }

    @ExceptionHandler(IllegalArgumentException.class)
    public ResponseEntity<String> onIllegalArgumentException(IllegalArgumentException exception) {
        // Personaliza la respuesta de error
        return new ResponseEntity<>("Error: " + exception.getMessage(), HttpStatus.BAD_REQUEST);
    }
}
```

Puedes manejar múltiples excepciones en un solo método usando una lista de excepciones:

```
@ExceptionHandler({IllegalArgumentException.class, NullPointerException.class})
public ResponseEntity<String> handleMultipleExceptions(Exception exception) {
```

```
    return new ResponseEntity<>("Error: " + exception.getMessage(), HttpStatus.INTERNAL_SERVER_ERROR);
}
```

El método anotado con `@ExceptionHandler` puede devolver diferentes tipos de respuestas, como `ResponseEntity`, para proporcionar una respuesta más completa y personalizada:

```
@ExceptionHandler(IllegalArgumentException.class)
public ResponseEntity<ErrorResponse> handleIllegalArgumentException(IllegalArgumentException exception) {
    ErrorResponse errorResponse = new ErrorResponse("Invalid argument", exception.getMessage());
    return new ResponseEntity<>(errorResponse, HttpStatus.BAD_REQUEST);
}
```

Donde `ErrorResponse` es una clase que se podría definir para encapsular detalles del error:

```
public class ErrorResponse {
    private String error;
    private String message;

    // Constructor, getters y setters
}
```

- Javadoc - @ExceptionHandler
- Exceptions - Spring Framework

@ResponseStatus

La anotación `@ResponseStatus` en Spring Framework se utiliza para **especificar el código de estado HTTP** de la respuesta generada por un método manejador de solicitudes. Esta anotación puede declararse en métodos de controladores de solicitudes y también en métodos de manejo de excepciones para establecer el código de estado HTTP adecuado.

Puedes usar `@ResponseStatus` para asignar un código de estado HTTP a una respuesta de un método:

```
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class ExampleController {

    @GetMapping("/hello")
    @ResponseStatus(HttpStatus.OK)
    public String helloWorld() {
        return "Hello, World!";
    }
}
```

En este ejemplo, el método `helloWorld` devuelve un código de estado '200 OK' con la respuesta.

Además, podemos proporcionar un motivo para el código de estado utilizando el argumento `reason`:

```
@GetMapping("/notfound")
@ResponseStatus(value = HttpStatus.NOT_FOUND, reason = "Resource not found")
public void resourceNotFound() {
```

```
    // Método vacío con código de estado 404 y motivo "Resource not found"
}
```

El argumento `reason` es opcional y se utiliza para proporcionar una descripción adicional que puede aparecer en los registros del servidor o en los detalles de la respuesta, dependiendo del cliente y del servidor.

La anotación `@ResponseStatus` también puede usarse en métodos de manejo de excepciones para devolver un código de estado HTTP específico cuando se produce una excepción:

```
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class ExampleController {

    @ExceptionHandler(IllegalArgumentException.class)
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    void onIllegalArgumentException(IllegalArgumentException exception) {
        // Manejo de IllegalArgumentException con código de estado 400 Bad Request
    }
}
```

En este caso, cuando se lanza una `IllegalArgumentException`, el código de estado de la respuesta será '400 Bad Request'.

- Javadoc - @ResponseStatus
- Returning Custom Status Codes from Spring Controllers - Baeldung

Spring Boot Annotations

Spring Boot facilita la configuración de Spring con su función de configuración automática.

Estas anotaciones forman parte del paquete *'org.springframework.boot.autoconfigure'* y *'org.springframework.boot.autoconfigure.condition'*.

- Spring Boot Annotations - Baeldung

@SpringBootApplication

La anotación `@SpringBootApplication` se utiliza para marcar la **clase principal de una aplicación Spring Boot**. Esta anotación es esencial en las aplicaciones Spring Boot, ya que combina varias configuraciones clave en una sola anotación, simplificando y automatizadno el proceso de configuración.

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class VehicleFactoryApplication {
    public static void main(String[] args) {
        SpringApplication.run(VehicleFactoryApplication.class, args);
    }
}
```

Esta anotación es una **meta-anotación** de las anotaciones `@SpringBootConfiguration` (**meta-anotación** de `@Configuration`), `@EnableAutoConfiguration` y `@ComponentScan` con sus atributos predeterminados.

- La anotación `@Configuration` marca la clase como una **clase de configuración** de Spring. Esto indica que la clase puede contener métodos anotados con `@Bean`, los cuales definen los beans del contexto de aplicación de Spring.
- La anotación `@EnableAutoConfiguration` habilita la **configuración automática** de Spring Boot. Spring Boot intenta configurar automáticamente la aplicación basándose en las dependencias presentes en el *classpath*. Esto simplifica la configuración inicial y reduce la necesidad de especificar manualmente los beans de configuración.
- La anotación `@ComponentScan` activa el **escaneo de componentes** en el paquete en el que se encuentra la clase principal y en los sub-paquetes. Esto permite que Spring detecte y registre componentes, servicios, controladores y otros beans definidos en el proyecto.

> El método `SpringApplication.run` arranca la aplicación Spring Boot, configurando el contexto de aplicación y lanzando el servidor embebido (como Tomcat, si se está utilizando).

- Javadoc - @SpringBootApplication
- Using the @SpringBootApplication Annotation - Spring Boot
- SpringApplication - Spring Boot

@EnableAutoConfiguration

La anotación `@EnableAutoConfiguration` en Spring Boot habilita la **configuración automática** de la aplicación. Spring Boot busca *beans* de configuración automática en el *classpath* y los aplica según las dependencias presentes.

Para utilizar `@EnableAutoConfiguration` directamente, debe ser combinada con la anotación `@Configuration`:

```
@Configuration
@EnableAutoConfiguration
public class VehicleFactoryConfig {
    // Configuración de la aplicación
}
```

Por otro lado, la anotación `@SpringBootApplication` es una meta-anotación de las anotaciones `@Configuration`, `@EnableAutoConfiguration` y `@ComponentScan` con sus atributos predeterminados.

Eso significa que si se utiliza `@SpringBootApplication`, no es necesario utilizar la anotación `@EnableAutoConfiguration` por separado.

- Javadoc - @EnableAutoConfiguration
- Auto-configuration - Spring Boot

Auto-Configuration Conditions

En el contexto de Spring Framework, las condiciones de auto-configuración (`@Conditional`) permiten aplicar configuraciones basadas en condiciones específicas durante la ejecución.

Estas condiciones se utilizan ampliamente en la auto-configuración de Spring Boot y en la configuración personalizada de aplicaciones Spring, permitiendo una configuración modular y flexible que se adapta dinámicamente según el entorno y las condiciones del sistema.

Spring proporciona **diversas condiciones predefinidas** listas para ser utilizadas, como `@ConditionalOnProperty` y `@ConditionalOnClass`.

Estas anotaciones de configuración se pueden colocar en clases `@Configuration` o métodos `@Bean`.

```
@Configuration
@ConditionalOnProperty(name = "feature.enabled", havingValue = "true")
public class FeatureConfig {
    @Bean
    public MyFeature myFeature() {
        return new MyFeature();
    }
}
```

- Creating Your Own Auto-configuration - Spring Boot
- Create a Custom Auto-Configuration with Spring Boot - Baeldung

@Conditional

La anotación `@Conditional` se utiliza para aplicar una condición a un componente de Spring, como un *bean* o una configuración, para que se active únicamente si la condición especificada se cumple en tiempo de ejecución:

```
@Configuration
@Conditional(OnProductionEnvironmentCondition.class)
public class ProductionConfiguration {
    // Configuración específica para el entorno de producción
}
```

La clase `OnProductionEnvironmentCondition` podría verificar si la aplicación se está ejecutando en un entorno de producción. Esto permite que la configuración específica solo se aplique en ese entorno, asegurando que las configuraciones adecuadas se carguen según el contexto:

```
public class OnProductionEnvironmentCondition implements Condition {
    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
        String env = context.getEnvironment().getProperty("env");
        return "production".equalsIgnoreCase(env);
    }
}
```

- Javadoc - @Conditional

@ConditionalOnClass and @ConditionalOnMissingClass

Usando estas condiciones, Spring solo aplicará el *bean* de configuración automática marcado si la clase especificada en el argumento de la anotación está presente (`@ConditionalOnClass`) o ausente (`@ConditionalOnMissingClass`) en el classpath.

```
@Configuration
@ConditionalOnClass(DataSource.class)
class MySQLAutoconfiguration {
    // Configuración específica para MySQL si DataSource está presente
}
```

```
@Configuration
@ConditionalOnMissingClass("com.example.SomeClass")
class FallbackConfiguration {
```

```
    // Configuración alternativa si SomeClass no está presente
}
```

- Javadoc - @ConditionalOnClass
- Javadoc - @ConditionalOnMissingClass

@ConditionalOnBean and @ConditionalOnMissingBean

Podemos usar estas anotaciones cuando queramos definir condiciones basadas en la presencia (`@ConditionalOnBean`) o ausencia (`@ConditionalOnMissingBean`) de un *bean* específico en el contexto de Spring.

```
@Bean
@ConditionalOnBean(name = "dataSource")
LocalContainerEntityManagerFactoryBean entityManagerFactory() {
    // Configuración del EntityManagerFactory si dataSource está presente
}
```

```
@Bean
@ConditionalOnMissingBean(name = "dataSource")
DataSource fallbackDataSource() {
    // Configuración alternativa si dataSource no está presente
}
```

- Javadoc - @ConditionalOnBean
- Javadoc - @ConditionalOnMissingBean

@ConditionalOnProperty

Con esta anotación, podemos poner condiciones sobre los valores de las propiedades en el archivo de configuración de la aplicación, como `application.properties` o `application.yml` .

```
@Bean
@ConditionalOnProperty(name = "usemysql", havingValue = "local")
DataSource dataSource() {
    // ...
}
```

También podemos usar `@ConditionalOnProperty` para condicionar la configuración basada en la presencia o ausencia de una propiedad utilizando el atributo `matchIfMissing` :

```
@Bean
@ConditionalOnProperty(name = "usemysql", matchIfMissing = true)
DataSource defaultDataSource() {
    // Configuración del DataSource por defecto si 'usemysql' no está presente
}
```

- Javadoc - @ConditionalOnProperty

@ConditionalOnResource

Podemos hacer que Spring use una definición solo cuando un recurso específico esté presente en el classpath:

```
@ConditionalOnResource(resources = "classpath:mysql.properties")
Properties additionalProperties() {
    // Configuración adicional si 'mysql.properties' está presente en el classpath
}
```

- Javadoc - @ConditionalOnResource

@ConditionalOnWebApplication and @ConditionalOnNotWebApplication

Con estas anotaciones podemos crear condiciones en función de si la aplicación actual es una aplicación web (`@ConditionalOnWebApplication`) o no es una aplicación web (`@ConditionalOnNotWebApplication`):

```
@Bean
@ConditionalOnWebApplication
HealthCheckController healthCheckController() {
    // Configuración del controlador de verificación de salud si es una aplicación web
}
```

- Javadoc - @ConditionalOnWebApplication
- Javadoc - @ConditionalOnNotWebApplication

@ConditionalOnExpression

Podemos utilizar esta anotación en situaciones más complejas. Spring usará la definición marcada cuando la expresión SpEL se evalúe como verdadera:

```
@Bean
@ConditionalOnExpression("${usemysql} && ${mysqlserver == 'local'}")
DataSource dataSource() {
    // Configuración del DataSource si usemysql es true y mysqlserver es 'local'
}
```

- Javadoc - @ConditionalOnExpression

Spring Scheduling Annotations

Cuando la ejecución de un solo hilo no es suficiente, podemos usar anotaciones del paquete *'org.springframework.scheduling.annotation'*.

- Task Execution and Scheduling - Spring Framework
- Task Execution and Scheduling - Spring Boot
- Spring Scheduling Annotations - Baeldung

@EnableAsync

Esta anotación habilita la funcionalidad asíncrona en Spring. Para habilitarla, debemos usar `@EnableAsync` junto con `@Configuration` :

```
@Configuration
@EnableAsync
class VehicleFactoryConfig {}
```

Ahora que hemos habilitado las llamadas asíncronas, podemos usar `@Async` para definir los métodos que las admiten:

```
@Async
public void processOrder() {
    // Código para procesar la orden de manera asíncrona
}
```

El método `processOrder` se ejecutará en un hilo separado, permitiendo que el hilo principal continúe su ejecución sin esperar a que este método termine.

Además, es posible configurar un `Executor` personalizado si se necesita un control más fino sobre los hilos utilizados para las tareas asíncronas:

```
@Configuration
@EnableAsync
public class AsyncConfig {

    @Bean(name = "taskExecutor")
    public Executor taskExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(5);
        executor.setMaxPoolSize(10);
        executor.setQueueCapacity(25);
        executor.initialize();
        return executor;
    }

}
```

Configurar un `Executor` personalizado permite ajustar el número de hilos y la capacidad de la cola para las tareas asíncronas. Con esta configuración, cualquier método anotado con `@Async` utilizará este `taskExecutor` definido.

- Javadoc - @EnableAsync

@EnableScheduling

Esta anotación habilita la programación en la aplicación. Para habilitarla, debemos usar `@EnableScheduling` junto con `@Configuration`:

```
@Configuration
@EnableScheduling
class VehicleFactoryConfig {}
```

Como resultado, ahora podemos ejecutar métodos periódicamente con `@Scheduled`:

```
@Scheduled(fixedRate = 5000)
public void reportCurrentTime() {
    System.out.println("The time is now " + new Date());
}
```

El método `reportCurrentTime` se ejecutará cada 5 segundos, imprimiendo la hora actual. `@Scheduled` admite varios parámetros como `fixedRate`, `fixedDelay` y `cron` para definir la frecuencia de ejecución de los métodos.

- Javadoc - @EnableScheduling

@Async

La anotación `@Async` en Spring se utiliza para marcar un método como **asíncrono**, permitiendo que el método se ejecute en un hilo separado o en un pool de hilos administrado por Spring. Esto es útil para operaciones que no necesitan bloquear el hilo principal de ejecución, como tareas de larga duración, operaciones de E/S intensivas, o llamadas a servicios externos.

Para marcar un método como asíncrono, simplemente lo anotamos con `@Async`:

```
@Async
void repairCar() {
    // ...
}
```

Si aplicamos esta anotación a una clase, todos los métodos se llamarán de forma asincrónica.

Para que `@Async` funcione correctamente, la clase que contiene el método anotado debe ser administrada por Spring (normalmente usando `@Service`, `@Component`, o `@Repository`)

Es importante habilitar las llamadas asíncronas con `@EnableAsync` o mediante configuración XML.

Un método anotado con `@Async` puede devolver un valor envuelto en un `Future` si se necesita obtener el resultado de la ejecución asíncrona:

```
import org.springframework.scheduling.annotation.Async;
import org.springframework.stereotype.Service;
import java.util.concurrent.Future;
import java.util.concurrent.CompletableFuture;

@Service
public class MyService {

    @Async
    public Future<String> asyncMethodWithReturn() {
        // Método que retorna un resultado de manera asíncrona
        return new AsyncResult<>("Resultado asíncrono");
    }

    @Async
    public CompletableFuture<String> asyncMethodWithCompletableFuture() {
        // Método que retorna un resultado de manera asíncrona usando CompletableFuture
        return CompletableFuture.completedFuture("Resultado asíncrono con CompletableFuture");
    }
}
```

El uso de `CompletableFuture` permite manejar operaciones más complejas y no bloqueantes.

- Javadoc - @Async
- Javadoc - CompletableFuture
- How To Do @Async in Spring - Baeldung

@Scheduled

La anotación `@Scheduled` en Spring se utiliza para programar la ejecución de métodos a intervalos específicos, en momentos específicos o según expresiones cron. Esta anotación es útil para tareas recurrentes y automatización de procesos dentro de una aplicación Spring.

Para ejecutar un método periódicamente, podemos usar esta anotación:

```
@Scheduled(fixedRate = 10000)
void checkVehicle() {
    // ...
}
```

Podemos usarla para ejecutar un método a **intervalos fijos** o con **expresiones cron**:

- `fixedRate`: ejecuta el método con una tasa fija en milisegundos, independientemente de cuánto tiempo haya tomado la ejecución anterior.
- `fixedDelay`: ejecuta el método con un retraso fijo en milisegundos después de que se completa la ejecución anterior.
- `initialDelay`: especifica un retraso inicial en milisegundos antes de la primera ejecución del método.
- `cron`: permite una expresión cron para definir horarios más complejos y específicos.

La anotación `@Scheduled` aprovecha la función de anotaciones repetidas de Java 8, lo que significa que podemos marcar un método con ella varias veces:

```
// Se ejecutará cada 10 segundos y también a la hora en punto de lunes a viernes.
@Scheduled(fixedRate = 10000)
@Scheduled(cron = "0 * * * * MON-FRI")
void checkVehicle() {
    // ...
}
```

Tenga en cuenta que el método anotado con `@Scheduled` debe tener un tipo de retorno nulo.

Además, debemos habilitar la programación para que esta anotación funcione, por ejemplo, con `@EnableScheduling` o la configuración XML.

Para que `@Scheduled` funcione correctamente, la clase que contiene el método anotado debe ser administrada por Spring (normalmente utilizando `@Component`, `@Service`, o similar).

- Javadoc - @Scheduled
- The @Scheduled Annotation in Spring - Baeldung

@Schedules

La anotación `@Schedules` permite especificar múltiples reglas `@Scheduled`:

```
@Schedules({
  @Scheduled(fixedRate = 10000),
  @Scheduled(cron = "0 * * * * MON-FRI")
})
void checkVehicle() {
```

```
    // ...
}
```

Desde Java 8, se puede lograr lo mismo utilizando anotaciones repetidas:

```
@Scheduled(fixedRate = 10000)
@Scheduled(cron = "0 * * * * MON-FRI")
void checkVehicle() {
    // ...
}
```

Las anotaciones repetidas simplifican el uso de múltiples reglas `@Scheduled`, haciendo que el código sea más limpio y fácil de mantener.

- Javadoc - @Schedules

Spring Data Annotations

Spring Data proporciona una abstracción sobre las tecnologías de almacenamiento de datos. Por lo tanto, nuestro código de lógica de negocios puede ser mucho más independiente de la implementación de persistencia subyacente. Además, Spring simplifica el manejo de los detalles del almacenamiento de datos que dependen de la implementación.

- Javadoc - Spring Data Core
- Data Access - Spring Framework
- Spring Data Annotations - Baeldung

Common Spring Data Annotations

@Transactional

La anotación `@Transactional` en Spring administra transacciones de manera declarativa en métodos o clases. Esta anotación permite que los métodos anotados se ejecuten dentro de una transacción gestionada por Spring, asegurando la atomicidad, consistencia, aislamiento y durabilidad (ACID) de las operaciones realizadas en la base de datos u otros recursos transaccionales.

Para configurar el comportamiento transaccional de un método, podemos usar:

```
@Transactional
void pay() {
    // ...
}
```

La anotación `@Transactional` puede aplicarse a interfaces y métodos de interfaces, no solo a clases y métodos de clases.

Si aplicamos esta anotación a nivel de clase, afecta a todos los métodos dentro de la clase. Podemos anular su comportamiento aplicando una configuración diferente a un método específico:

```
@Transactional
public class PaymentService {

    @Transactional
```

```
    public void pay() {
        // ...
    }

    @Transactional(readOnly = true)
    public void checkPaymentStatus() {
        // ...
    }
}
```

La anotación `@Transactional` admite varios atributos para personalizar el comportamiento transaccional:

- `propagation`: define cómo debe comportarse la transacción actual en relación con las transacciones existentes.
- `isolation`: especifica el nivel de aislamiento de la transacción.
- `timeout`: define el tiempo máximo que puede durar la transacción antes de ser abortada.
- `readOnly`: indica si la transacción es solo de lectura.
- `rollbackFor`: especifica las excepciones que deben provocar un rollback de la transacción.

Además, `@Transactional` es compatible con varios gestores de transacciones, como JPA, JDBC, y JMS.

Para que `@Transactional` funcione correctamente, la clase que contiene el método anotado debe ser administrada por Spring (normalmente utilizando `@Service`, `@Component`, o similar).

- Javadoc - @Transactional
- Transactions with Spring and JPA - Baeldung
- Using @Transactional - Spring Framework

@NoRepositoryBean

La anotación `@NoRepositoryBean` en Spring se utiliza para indicar que una interfaz de repositorio específica no debe ser considerada como un repositorio gestionado por Spring Data. Esto significa que Spring no creará una instancia de esa interfaz de repositorio, y no se aplicarán las características típicas de repositorio, como la generación automática de consultas y métodos CRUD.

```
import org.springframework.data.repository.NoRepositoryBean;
import org.springframework.data.repository.Repository;

@NoRepositoryBean
public interface MyBaseRepository<T, ID> extends Repository<T, ID> {
    // Métodos personalizados que no son generados automáticamente por Spring Data
}

@Repository
public interface UserRepository extends MyBaseRepository<User, Long> {
    // Spring Data generará automáticamente métodos CRUD para la entidad User
}
```

La anotación `@NoRepositoryBean` es útil cuando se requiere definir una interfaz base para repositorios que contengan métodos comunes o personalizados, pero que no represente un repositorio que deba ser instanciado directamente por **Spring Data**.

La anotación `@NoRepositoryBean` es útil cuando se requiere definir una interfaz base para repositorios que contengan métodos comunes o personalizados, pero que no represente un repositorio que deba ser instanciado directamente por Spring Data. Esto

es especialmente útil en proyectos grandes donde se necesita una estructura de repositorios más compleja y reutilizable.

- Javadoc - @NoRepositoryBean

@Param

La anotación `@Param` permite pasar parámetros con nombre a nuestras consultas:

```
@Query("FROM Person p WHERE p.name = :name")
Person findByName(@Param("name") String name);
```

Es importante referirse al parámetro con la sintaxis `:name`. Usar parámetros con nombre mejora la legibilidad y evita errores en las consultas.

`@Param` es útil tanto en consultas JPQL como en consultas SQL nativas.

También se puede usar `@Param` en consultas con múltiples parámetros:

```
@Query("FROM Person p WHERE p.name = :name AND p.age = :age")
Person findByNameAndAge(@Param("name") String name, @Param("age") int age);
```

- Javadoc - @Param
- Spring Data JPA @Query - Baeldung

@Id

La anotación `@Id` marca un campo en una clase de modelo como la **clave principal**:

```
class Person {

    @Id
    Long id;

    // ...

}
```

Al ser independiente de la implementación, permite que una clase de modelo sea compatible con múltiples motores de almacenamiento de datos.

La clave principal es fundamental en el contexto de las bases de datos, ya que **identifica de manera única cada entidad**.

- Javadoc - @Id

@Transient

Podemos usar esta anotación para marcar un campo en una clase de modelo como **transitorio**. Esto significa que el motor de almacenamiento de datos no leerá ni escribirá el valor de este campo:

```
class Person {

    // ...
```

```
    @Transient
    int age;

    // ...

}
```

Al igual que `@Id`, la anotación `@Transient` también es independiente de la implementación, lo que facilita su uso con múltiples implementaciones de almacenamiento de datos.

Esta anotación es útil para campos **calculados o temporales** que no necesitan ser persistidos en la base de datos.

- Javadoc - @Transient

Campos de auditoría

En Spring Data, las anotaciones de campos de auditoría se utilizan para mantener un rastro automático de ciertas acciones comunes, como la creación y modificación de entidades. Estas anotaciones permiten que Spring Data gestione automáticamente campos como el usuario que creó o modificó la entidad, y las fechas de creación y modificación:

- `@CreatedBy`: marca un campo que debe contener el usuario que creó la entidad. Este campo se llena automáticamente cuando la entidad se persiste por primera vez.
- `@LastModifiedBy`: indica el usuario que realizó la última modificación a la entidad. Este campo se actualiza automáticamente cada vez que la entidad se modifica.
- `@CreatedDate`: marca un campo para almacenar la fecha y hora en que la entidad fue creada. Este campo se llena automáticamente cuando la entidad se persiste por primera vez.
- `@LastModifiedDate`: anota un campo que debe contener la fecha y hora de la última modificación de la entidad. Este campo se actualiza automáticamente cada vez que la entidad se modifica.

```
public class Person {

    // ...

    @CreatedBy
    User creator;

    @LastModifiedBy
    User modifier;

    @CreatedDate
    Date createdAt;

    @LastModifiedDate
    Date modifiedAt;

    // ...

}
```

Para utilizar estas anotaciones, es necesario seguir algunos pasos adicionales para habilitar la auditoría en una aplicación Spring:

- Añadir la anotación `@EnableJpaAuditing` en una clase de configuración de Spring para habilitar la auditoría:

```
@Configuration
@EnableJpaAuditing(auditorAwareRef = "auditorProvider")
```

```
public class AuditConfig {
}
```

- Implementar la interfaz `AuditorAware` para especificar cómo se obtiene el usuario actual:

```
import org.springframework.data.domain.AuditorAware;
import org.springframework.stereotype.Component;

@Component("auditorProvider")
public class AuditorAwareImpl implements AuditorAware<User> {

    @Override
    public Optional<User> getCurrentAuditor() {
        // Lógica para obtener el usuario actual
        return Optional.of(new User("currentUser"));
    }
}
```

- Auditing with JPA, Hibernate, and Spring Data JPA - Baeldung
- Auditing - Spring Data

Spring Data JPA Annotations

- Javadoc - Spring Data JPA

@Query

Con la anotación `@Query`, se proporciona una **implementación JPQL** para un método de repositorio:

```
@Query("SELECT COUNT(*) FROM Person p")
long getPersonCount();
```

Además, se puede utilizar parámetros con nombre con la anotación `@Param`:

```
@Query("FROM Person p WHERE p.name = :name")
Person findByName(@Param("name") String name);
```

También se pueden utilizar **consultas SQL nativas** configurando el argumento `nativeQuery = true`:

```
@Query(value = "SELECT AVG(p.age) FROM person p", nativeQuery = true)
int getAverageAge();
```

La anotación `@Query` es útil para consultas complejas que no se pueden expresar fácilmente con los métodos de consulta derivados. Además, permite el uso de expresiones SpEL (*Spring Expression Language*) para consultas dinámicas.

- Javadoc - @Query
- JPA Query Methods - Spring Data
- Spring Data JPA @Query - Baeldung

@Procedure

Con Spring Data JPA, podemos llamar fácilmente a procedimientos almacenados desde repositorios.

Primero, es necesario declarar el procedimiento almacenado en la clase de entidad usando anotaciones JPA estándar. La anotación `@NamedStoredProcedureQuery` se utiliza para definir procedimientos almacenados en la entidad:

```
@NamedStoredProcedureQueries({
    @NamedStoredProcedureQuery(
        name = "count_by_name",
        procedureName = "person.count_by_name",
        parameters = {
            @StoredProcedureParameter(
                mode = ParameterMode.IN,
                name = "name",
                type = String.class),
            @StoredProcedureParameter(
                mode = ParameterMode.OUT,
                name = "count",
                type = Long.class)
            }
    )
})

class Person {}
```

Después de esto, se puede hacer referencia a él en el repositorio usando el nombre declarado en el argumento `name`:

```
@Procedure(name = "count_by_name")
long getCountByName(@Param("name") String name);
```

La anotación `@Procedure` se utiliza para llamar a procedimientos almacenados definidos en la entidad, así como a otros procedimientos almacenados que estén definidos en otras entidades o directamente en la base de datos:

```
@Procedure(procedureName = "person.count_by_name")
long getCountByNameDirect(@Param("name") String name);
```

Los procedimientos almacenados son útiles para encapsular lógica compleja en la base de datos y mejorar el rendimiento.

- Javadoc - @Procedure

@Lock

La anotación `@Lock` en Spring Data JPA se utiliza para especificar el nivel de bloqueo que debe aplicarse a una consulta JPA en las operaciones de acceso a la base de datos. Esta anotación es fundamental para controlar la concurrencia y evitar problemas como las condiciones de carrera, especialmente en entornos donde múltiples transacciones podrían intentar acceder o modificar los mismos datos simultáneamente.

La anotación `@Lock` admite varios tipos principales de bloqueo:

- `LockModeType.NONE` : no se aplica ningún bloqueo. Este es el **modo predeterminado** si no se especifica un tipo de bloqueo.
- ** `LockModeType.PESSIMISTIC_READ` : este bloqueo permite que otras transacciones lean los datos, pero no permite actualizaciones. Es útil para evitar que los datos cambien mientras se está leyendo.

- `LockModeType.PESSIMISTIC_WRITE` : este bloqueo evita que otras transacciones lean o actualicen los datos mientras se mantiene el bloqueo. Es útil cuando se necesita asegurarse de que nadie más pueda leer o modificar los datos hasta que la transacción actual termine.

- `LockModeType.PESSIMISTIC_FORCE_INCREMENT` : adquiere un bloqueo de escritura pesimista e incrementa la versión de la entidadasegurando que las transacciones concurrentes vean la nueva versión de la entidad.

- `LockModeType.OPTIMISTIC` : utiliza el bloqueo optimista, que se basa en versiones. Permite que múltiples transacciones lean y actualicen los datos, pero verifica al final de la transacción si los datos han cambiado desde la última lectura. Si es así, lanza una excepción `OptimisticLockException` .

- `LockModeType.OPTIMISTIC_FORCE_INCREMENT` : similar al bloqueo optimista, pero además incrementa la versión de la entidad, asegurando que cualquier otra transacción que quiera leer los datos tendrá que esperar hasta que la transacción actual complete.

- ** `LockModeType.READ` : sinónimo de `OPTIMISTIC` , aunque se utiliza raramente en la práctica.

- ** `LockModeType.WRITE` : sinónimo de `OPTIMISTIC_FORCE_INCREMENT` también poco utilizado en la práctica.

Es importante considerar que los **bloqueos pesimistas** pueden afectar el rendimiento de la base de datos, ya que impiden que otras transacciones accedan a los datos bloqueados.

En cambio, los **bloqueos optimistas** son más adecuados para sistemas con alta concurrencia, ya que permiten una mayor escalabilidad al no bloquear los datos de manera exclusiva.

- Javadoc - @Lock

- Javadoc - LockModeType

- Enabling Transaction Locks in Spring Data JPA - Baeldung

- Locking - Spring Data

@Modifying

La anotación `@Modifying` se utiliza en Spring Data JPA para indicar que un método de repositorio modifica datos en la base de datos. Específicamente, se emplea en combinación con la anotación @Query cuando la consulta realiza operaciones de modificación como UPDATE, DELETE o INSERT.

Por ejemplo, se puede modificar el nombre de una persona en la base de datos con un método de repositorio anotado con `@Modifying` :

```
@Modifying
@Query("UPDATE Person p SET p.name = :name WHERE p.id = :id")
void changeName(@Param("id") long id, @Param("name") String name);
```

Es importante destacar que, al usar `@Modifying` , es recomendable también gestionar las transacciones para asegurarse de que los cambios se persistan correctamente. Si el método no está en una transacción activa, Spring iniciará una automáticamente.

- Javadoc - @Modifying

- Spring Data JPA @Query - Baeldung

@EnableJpaRepositories

Para utilizar repositorios JPA, se le tiene que indicar a Spring mediante la anotación `@EnableJpaRepositories`.

Hay que tener en cuenta que esta anotación se debe usar con la anotación `@Configuration`:

```
@Configuration
@EnableJpaRepositories
class PersistenceJPAConfig {}
```

Spring buscará repositorios en los subpaquetes de esta clase `@Configuration`. Se puede alterar este comportamiento con el argumento `basePackages`:

```
@Configuration
@EnableJpaRepositories(basePackages = "com.baeldung.persistence.dao")
class PersistenceJPAConfig {}
```

En aplicaciones Spring Boot no es necesario usar esta anotación explícitamente, ya que Spring Boot lo configura **automáticamente** si detecta la dependencia de Spring Data JPA en el classpath.

Sin embargo, en configuraciones avanzadas, como cuando se trabaja con múltiples fuentes de datos o configuraciones de repositorios personalizados, puede ser necesario.

- Javadoc - @EnableJpaRepositories

Spring Data Mongo Annotations

- Javadoc - Spring Data MongoDB
- Introduction to Spring Data MongoDB - Baeldung

@Document

La anotación `@Document` marca una clase como un objeto de dominio que queremos conservar en una base de datos MongoDB:

```
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;
import org.springframework.data.mongodb.core.mapping.Field;

@Document(collection = "user")
class User {

    @Id
    private String id;

    @Field("username")
    private String name;

    // Getters and setters...
}
```

Esta anotación también nos permite elegir el nombre de la colección que queremos utilizar a través del atributo `collection`.

Esta anotación `@Document` es el equivalente en MongoDB de `@Entity` en JPA. Al igual que `@Entity`, se utiliza junto con otras anotaciones como `@Id` para definir el campo que actúa como clave primaria, y `@Field` para mapear campos de la clase a nombres específicos en la colección.

- Javadoc - @Document

@Field

Con la anotación `@Field`, podemos configurar el nombre de un campo que queremos usar cuando MongoDB persiste el documento:

La anotación `@Field` en Spring Data MongoDB se utiliza para especificar el nombre de un campo en la base de datos que es diferente del nombre del campo en la clase Java. Esto es útil cuando se necesita que los nombres de los campos en MongoDB sigan una convención diferente a la de las propiedades en el código Java:

```
import org.springframework.data.mongodb.core.mapping.Document;
import org.springframework.data.mongodb.core.mapping.Field;

@Document(collection = "user")
class User {

    @Field("email")
    private String emailAddress;

    @Field("first_name")
    private String firstName;

    @Field("last_name")
    private String lastName;

    // Getters and setters...
}
```

Este ejemplo muestra cómo el campo `emailAddress` en Java se mapea al campo `email` en la colección MongoDB.

La anotación `@Field` es el equivalente en MongoDB de `@Column` en JPA. Al igual que `@Column`, se utiliza para mapear un nombre de campo en el código a un nombre específico en la base de datos.

Destacar que `@Field` es opcional si el nombre del campo en Java es el mismo que en MongoDB. Se usa principalmente cuando se necesita una diferenciación en los nombres.

- Javadoc - @Field

@Query

La anotación `@Query` en Spring Data MongoDB se utiliza para definir consultas personalizadas directamente en los métodos del repositorio. Esto es útil cuando necesitas realizar consultas que no pueden expresarse fácilmente con los métodos de consulta derivada proporcionados por Spring Data.

```
import org.springframework.data.mongodb.repository.MongoRepository;
import org.springframework.data.mongodb.repository.Query;
import java.util.List;

public interface UserRepository extends MongoRepository<User, String> {

    @Query("{ 'name' : ?0 }")
    List<User> findUsersByName(String name);

    @Query("{ 'age' : { $gt: ?0, $lt: ?1 } }")
    List<User> findUsersByAgeRange(int ageStart, int ageEnd);
}
```

En este ejemplo, la consulta `@Query("{ 'name' : ?0 }")` busca documentos en la colección donde el campo `name` coincide con el valor proporcionado por el parámetro `name` del método.

La consulta se define en formato JSON de MongoDB. El placeholder `?0` se refiere al primer argumento del método (en este caso, `String name`).

Puedes crear consultas más complejas utilizando operadores MongoDB. Por ejemplo, la siguiente consulta busca usuarios cuya edad está entre dos valores:

```
@Query("{ 'age' : { $gt: ?0, $lt: ?1 } }")
List<User> findUsersByAgeRange(int ageStart, int ageEnd);
```

También es posible definir proyecciones para seleccionar solo ciertos campos de los documentos, optimizando así las operaciones de lectura.

- Javadoc - @Query

@EnableMongoRepositories

La anotación `@EnableMongoRepositories` en Spring Data MongoDB se utiliza para habilitar la creación de repositorios basados en MongoDB. Esta anotación indica a Spring que busque interfaces de repositorio y las implemente automáticamente.

Para usar esta anotación, debes aplicarla a una clase de configuración marcada con `@Configuration`. Spring buscará las interfaces de repositorio en los subpaquetes de la clase configurada:

```
@Configuration
@EnableMongoRepositories
class MongoConfig {}
```

Por defecto, Spring escaneará todos los subpaquetes del paquete en el que se encuentra esta clase `@Configuration`. Si deseas especificar un paquete concreto donde Spring debería buscar los repositorios, puedes utilizar el argumento `basePackages`:

```
@Configuration
@EnableMongoRepositories(basePackages = "com.baeldung.repository")
class MongoConfig {}
```

Alternativamente, puedes usar `basePackageClasses` para especificar clases de un paquete, lo que también guiará a Spring a escanear los repositorios en esos paquetes:

```
@Configuration
@EnableMongoRepositories(basePackageClasses = MyRepository.class)
class MongoConfig {}
```

En aplicaciones Spring Boot, esta anotación no es necesaria explícitamente. Spring Boot configurará automáticamente los repositorios si detecta que Spring Data MongoDB está presente en el classpath, es decir, si se ha incluido como dependencia en el proyecto.

- Javadoc - @EnableMongoRepositories

Spring Bean Annotations

El contenedor de Spring se encarga de crear los beans en la aplicación y de coordinar las relaciones entre estos objetos a través de la inyección de dependencias o DI.

A la hora de expresar una especificación de conexión de bean, Spring ofrece tres mecanismos principales:

- **Configuración explícita en XML**: los *beans* se declaran en un fichero de configuración `.xml`.
- **Configuración explícita en Java (JavaConfig)**: los *beans* se declaran usando la anotación `@Bean` en una clase de configuración anotada con `@Configuration`.
- **Análisis y detección implícita de beans y conexión automática de beans**: finalmente, podemos marcar la clase como componentes gestionados por el contenedor de Spring con una de las anotaciones del paquete *'org.springframework.stereotype'* y dejar el trabajo para el escaneo de componentes mediante `@ComponentScan`.

Todos los mecanismos son compatibles entre sí y no excluyentes, es decir, dos mecanismos pueden ser utilizados en la misma aplicación.

La recomendación es usar la **conexión automática** ya que requiere menos configuración explícita y además usar la **configuración explícita** en el caso de usar librerías y frameworks de terceros para conectar sus componentes porque en estos casos **no se dispone de acceso al código fuente** para anotar las clases con `@Component` o `@Service` o cualquier otra.

- Spring Bean Annotations - Baeldung

@ComponentScan

Spring puede **escanear automáticamente** un paquete en busca de *beans* si el escaneo de componentes está habilitado.

La anotación `@ComponentScan` se utiliza junto a la anotación `@Configuration` para especificarle a Spring qué paquetes debe escanear en busca de componentes anotados con `@Component`, `@Service`, `@Repository`, entre otros. Esta anotación configura los paquetes base que deben ser escaneados.

Podemos especificar los nombres de los paquetes base directamente con los argumentos `basePackages` (o su alias `value`):

```
@Configuration
@ComponentScan(basePackages = "com.baeldung.annotations")
class VehicleFactoryConfig {}
```

Además, podemos indicar **clases** en los paquetes base con el argumento `basePackageClasses`, que proporciona seguridad de tipos. Esta forma es útil para evitar inconsistencias durante la refactorización de nombres de paquetes o clases:

```
@Configuration
@ComponentScan(basePackageClasses = VehicleFactoryConfig.class)
class VehicleFactoryConfig {}
```

Ambos argumentos son matrices, por lo que podemos proporcionar varios paquetes o clases para escanear:

```
@Configuration
@ComponentScan(basePackageClasses = {VehicleFactoryConfig.class, OtherFactoryConfig.class})
class VehicleFactoryConfig {}
```

Si no se especifica ningún argumento, Spring escaneará automáticamente el mismo paquete donde está presente la clase anotada con `@ComponentScan`. Se recomienda que la clase de configuración o la clase principal de la aplicación esté en un paquete raíz para que todos los subpaquetes se incluyan en el escaneo.

Esto es común en aplicaciones Spring Boot, donde la clase principal anotada con `@SpringBootApplication` actúa como punto de partida para el escaneo. La anotación `@SpringBootApplication` es una meta-anotación, que incluye, entre otras, la anotación `@ComponentScan` por lo que el escaneo se hace automáticamente:

```
package com.example.myapplication;

@SpringBootApplication
public class MyApplication { ... }
```

En este caso, Spring escaneará todos los subpaquetes, como *'com.example.myapplication.customer'* o *'com.example.myapplication.order'*.

```
com
 +- example
     +- myapplication
         +- MyApplication.java
         |
         +- customer
         |   +- Customer.java
         |   +- CustomerController.java
         |   +- CustomerService.java
         |   +- CustomerRepository.java
         |
         +- order
             +- Order.java
             +- OrderController.java
             +- OrderService.java
             +- OrderRepository.java
```

En Java 8, gracias a las anotaciones repetidas, se pueden utilizar varias anotaciones `@ComponentScan` en una clase de configuración, lo que permite especificar varias configuraciones de escaneo:

```
@Configuration
@ComponentScan(basePackages = "com.baeldung.annotations")
@ComponentScan(basePackageClasses = VehicleFactoryConfig.class)
class VehicleFactoryConfig {}
```

Alternativamente, se puede usar la anotación `@ComponentScans` para especificar múltiples configuraciones de `@ComponentScan`:

```
@Configuration
@ComponentScans({
  @ComponentScan(basePackages = "com.baeldung.annotations"),
  @ComponentScan(basePackageClasses = VehicleFactoryConfig.class)
})
class VehicleFactoryConfig {}
```

Spring también permite aplicar filtros para incluir o excluir determinados beans durante el escaneo. Esto se puede lograr con los atributos `includeFilters` y `excludeFilters`:

```
@ComponentScan(
  basePackages = "com.baeldung.annotations",
  includeFilters = @ComponentScan.Filter(MyCustomAnnotation.class),
  excludeFilters = @ComponentScan.Filter(AnotherCustomAnnotation.class)
)
class VehicleFactoryConfig {}
```

- Javadoc - @ComponentScan
- Javadoc - @ComponentScans
- Spring Component Scanning - Baeldung
- Classpath Scanning and Managed Components - Spring Framework
- Structuring Your Code - Spring Boot

@Component

La anotación `@Component` es una anotación a **nivel de clase** en Spring que marca una clase como un *bean* gestionado por el contenedor de Spring. Durante el análisis de componentes con `@ComponentScan`, Spring Framework detecta automáticamente las clases anotadas con `@Component` y las registra en el contexto de aplicación como *beans*:

```
import org.springframework.stereotype.Component;

@Component
class CarUtility {
    // ...
}
```

De forma predeterminada, el nombre del *bean* registrado será el mismo que el nombre de la clase, pero con la inicial en minúscula (`carUtility` en este caso). Sin embargo, podemos especificar un nombre personalizado para el *bean* utilizando el argumento `value` opcional:

```
import org.springframework.stereotype.Component;

@Component("customServiceName")
public class MyService {
    public void performService() {
        System.out.println("Service is being performed");
    }
}
```

Otra forma de asignar nombres a un *bean* es utilizando la anotación `@Named` de la **especificación de inyección de dependencias de Java JSR-330**. Esta anotación es equivalente a `@Component`, pero es parte de un estándar de inyección de dependencias más amplio:

```
import javax.inject.Named;

@Named("customServiceName")
public class MyService {
    public void performService() {
        System.out.println("Service is being performed");
    }
}
```

Anotaciones como `@Repository`, `@Service`, `@Controller` y `@Configuration` son **meta-anotaciones** de `@Component`, lo que significa que heredan el comportamiento de `@Component`. Además de marcar clases como *beans*, estas anotaciones proporcionan una semántica más específica sobre el propósito del *bean*:

- `@Repository` ➔ Usada para clases que interactúan directamente con la base de datos. Ofrece funcionalidades adicionales, como la conversión automática de excepciones de persistencia.

- `@Service` ➔ Indica que una clase contiene lógica de negocio. Aunque funcionalmente igual a `@Component`, su uso mejora la claridad semántica en el código.
- `@Controller` ➔ Se utiliza para clases que gestionan solicitudes HTTP en aplicaciones web. Además de registrarlas como *beans*, Spring las trata como controladores de peticiones.
- `@Configuration` ➔ Indica que la clase proporciona métodos de configuración que producen *beans*.

Estas anotaciones también permiten a Spring aplicar comportamientos específicos. Por ejemplo, `@Repository` está asociada con la traducción de excepciones de bases de datos en Spring, mientras que `@Service` no tiene ese comportamiento por defecto.

- Javadoc - @Component
- Using JSR 330 Standard Annotations - Spring Framework

@Repository

Las clases DAO o *Repository* representan la capa de acceso a la base de datos en una aplicación y deben anotarse con `@Repository`:

```
@Repository
class VehicleRepository {
    // ...
}
```

Una de las ventajas clave de utilizar la anotación `@Repository` es que habilita la **traducción automática de excepciones de persistencia**. Esto significa que cualquier excepción lanzada por el proveedor de persistencia, como Hibernate o JPA, será capturada y traducida automáticamente por Spring en una subclase de `DataAccessException`, que es una excepción más general y manejable.

Por ejemplo, si Hibernate lanza una `ConstraintViolationException`, Spring la traducirá en una `DataIntegrityViolationException`, lo que facilita el manejo coherente de errores en la capa de acceso a datos.

Para asegurarse de que Spring traduzca automáticamente estas excepciones, es posible declarar explícitamente un bean de tipo `PersistenceExceptionTranslationPostProcessor`, aunque en la mayoría de los casos, Spring lo configura automáticamente.

```
@Bean
public PersistenceExceptionTranslationPostProcessor exceptionTranslation() {
    return new PersistenceExceptionTranslationPostProcessor();
}
```

Este *bean* permite que las excepciones nativas lanzadas dentro de las clases anotadas con `@Repository` se traduzcan correctamente a subclases de `DataAccessException`.

En aplicaciones basadas en Spring Boot, esta funcionalidad suele estar habilitada automáticamente si se utiliza un marco de persistencia como JPA o Hibernate, sin necesidad de declarar explícitamente el bean `PersistenceExceptionTranslationPostProcessor`.

- Javadoc - @Repository

@Service

La **lógica de negocio** de una aplicación normalmente reside dentro de la **capa de servicio**, por lo que usamos la anotación `@Service` para indicar que una clase pertenece a esa capa:

```java
@Service
public class VehicleService {
    // ...
}
```

La anotación `@Service` es una especialización de `@Component`, por lo que las clases anotadas con `@Service` son detectadas automáticamente por el escaneo de componentes de Spring. Esto permite que Spring las registre como *beans* y las gestione dentro del contenedor de aplicación.

A diferencia de `@Component`, que es una anotación genérica para cualquier componente de Spring, `@Service` se utiliza para marcar **servicios que contienen la lógica de negocio**.

Aunque `@Service` no tiene un efecto funcional adicional en comparación con `@Component`, su uso mejora la claridad y semántica del código al indicar que la clase realiza tareas relacionadas con la lógica de negocio.

- Javadoc - @Service

@Controller

La anotación `@Controller` es una anotación a **nivel de clase** que le indica a Spring Framework que esta clase sirve como un controlador en el contexto de Spring MVC:

```java
@Controller
public class VehicleController {

    @GetMapping("/vehicles")
    public String listVehicles(Model model) {
        List<Vehicle> vehicles = vehicleService.findAll();
        model.addAttribute("vehicles", vehicles);
        return "vehicleList"; // nombre de la vista (ej. un archivo Thymeleaf o JSP)
    }
}
```

En este ejemplo, el método `listVehicles` maneja las solicitudes GET a la ruta , obtiene una lista de vehículos del servicio y los añade al modelo. Luego, retorna el nombre de la vista que se debe renderizar.

La anotación `@Controller` es parte del framework Spring MVC y se utiliza para manejar solicitudes web. Las clases anotadas con `@Controller` contienen métodos que gestionan las solicitudes HTTP y retornan vistas o datos.

Los métodos en una clase anotada con `@Controller` están destinados a manejar **solicitudes web**. Estos métodos suelen estar anotados con `@RequestMapping`, `@GetMapping`, `@PostMapping`, etc., para mapear las solicitudes a métodos específicos.

Las clases `@Controller` a menudo trabajan con modelos (`@ModelAttribute`) y vistas (*"ModelAndView"*). En una arquitectura MVC (Modelo-Vista-Controlador), `@Controller` se encarga de preparar datos del modelo y decidir qué vista debe ser renderizada.

Aunque `@Controller` se usa en aplicaciones Spring MVC, en aplicaciones Spring Boot, que suelen retornar JSON o XML, es común usar `@RestController`, que combina `@Controller` y `@ResponseBody`.

- Javadoc - @Controller

@Configuration

La anotación `@Configuration` se utiliza para definir clases que contienen métodos de configuración de *beans* en Spring. Estas clases son utilizadas para crear y configurar *beans* dentro del contexto de la aplicación.

Las clases anotadas con `@Configuration` pueden contener métodos anotados con `@Bean`. Estos métodos se encargan de instanciar, configurar y devolver los objetos que se registrarán como *beans* en el contexto de la aplicación.

```
@Configuration
public class AppConfig {

    @Bean
    public Engine engine() {
        return new Engine();
    }

    @Bean
    public Vehicle vehicle() {
        return new Vehicle(engine());
    }
}
```

En este ejemplo, la clase `AppConfig` define dos *beans*: `engine` y `vehicle`. El *bean* `vehicle` depende del *bean* `engine`, y Spring se encargará de la inyección de dependencias cuando se cree una instancia de `Vehicle`.

Esta anotación `@Configuration` es una **meta-anotación** o especialización de `@Component`, lo que significa que también se considera un componente de Spring y, por lo tanto, puede ser detectada automáticamente durante el escaneo de componentes si se usa `@ComponentScan` o `@SpringBootApplication`.

Las clases `@Configuration` pueden ser utilizadas para definir varias configuraciones dentro de una aplicación. Es común usar esta anotación para definir configuraciones específicas de diferentes módulos o áreas de la aplicación.

- Javadoc - @Configuration
- Basic Concepts: @Bean and @Configuration - Spring Framework
- Using the @Configuration annotation - Spring Framework

Spring AOP

La **Programación Orientada a Aspectos (AOP)** es un paradigma de programación que se enfoca en separar las preocupaciones transversales del núcleo de la lógica del negocio.

Una preocupación transversal pueden definirse como cualquier **funcionalidad que afecta a varios puntos de una aplicación**, como los inicios de sesión, la gestión de transacciones, el registro (logging), la seguridad, y el manejo de excepciones.

La AOP es muy útil para modularizar estas funcionalidades que se repiten en diferentes partes de la aplicación y desacoplar estas preocupaciones de la lógica de negocio.

- Aspect Oriented Programming with Spring - Spring Framework
- Spring AOP - Baeldung
- Introduction to Spring AOP - Baeldung

Conceptos de AOP

- **Aspecto (*Aspect*)**

  Un aspecto es un módulo que encapsula una **preocupación transversal**. En términos simples, es **una clase que contiene la lógica** para una funcionalidad transversal, como el registro de auditoría, la seguridad o el manejo de transacciones.

- **Consejo (*Advice*)**

  Un *advice* es **la acción que un aspecto realiza en un punto específico** en el programa. Es el código que se ejecuta cuando se encuentra un punto de unión. Define lo que se debe hacer y cuándo debe hacerse. Los tipos comunes de *advice* son:

  - `Before` : Se ejecuta antes de que el método objetivo sea llamado.
  - `After` : Se ejecuta después de que el método objetivo ha sido llamado, independientemente de su resultado.
  - `After Returning` : Se ejecuta después de que el método objetivo retorna un resultado.
  - `After Throwing` : Se ejecuta si el método objetivo lanza una excepción.
  - `Around` : Envuelve la ejecución del método objetivo, permitiendo realizar acciones antes y después de que se llame al método, y también puede controlar si se llama o no al método objetivo.

- **Punto de Unión (*Join Point*)**

  Un punto de unión es **un punto en la ejecución del programa donde se podría aplicar el aspecto**, como la invocación de un método o la ejecución de un bloque de código. En Spring AOP, los puntos de unión son principalmente las invocaciones de métodos.

- **Corte Transversal (Pointcut)**

  Un pointcut es una **expresión que selecciona uno o más puntos de unión** donde un *advice* debería ejecutarse. Define el **lugar exacto donde se aplica un aspecto** en el código. Por ejemplo, se puede definir un *pointcut* para todos los métodos en una clase específica o para métodos con un nombre particular.

- **Objeto de Intercepción (Proxy)**

  En AOP, un proxy es **un objeto creado en tiempo de ejecución que intercepta las llamadas** a los métodos de un objeto objetivo para aplicar el *advice*. Los proxies permiten que los aspectos se apliquen sin modificar el código original del objeto.

- **Objetivo (*Target Object*)**

  El objetivo es el objeto cuyo método está siendo interceptado. Es el **objeto en el que se aplican los aspectos**.

Estos términos no son específicos de Spring.

- AOP Concepts - Spring Framework

Implementación de AOP en Spring

Spring AOP es el framework de Spring para la programación orientada a aspectos. Utiliza **proxies dinámicos** para aplicar aspectos en tiempo de ejecución y se integra de manera efectiva con otros componentes de Spring. Spring AOP está implementado en Java puro.

El enfoque de Spring AOP hacia AOP difiere del enfoque de la mayoría de otros frameworks de AOP. El objetivo no es proporcionar la implementación de AOP más completa si no más bien proporcionar una implementación de AOP más **simple y centrada** en las necesidades más comunes de los desarrolladores de aplicaciones empresariales.

Algunas características y elementos clave de Spring AOP son:

- Spring AOP se puede usar con **AspectJ**, un framework completo de AOP para Java que ofrece más funcionalidades avanzadas. AspectJ permite una programación orientada a aspectos más extensa, pero Spring AOP es suficiente para

muchas necesidades de desarrollo.

- Spring AOP utiliza **proxies dinámicos** para aplicar aspectos en tiempo de ejecución. Esto significa que puede aplicar aspectos solo a los *beans* de Spring y únicamente a métodos públicos.
- Spring AOP se centra en aspectos que se aplican **antes, después o alrededor de la ejecución de métodos**. Esto cubre los casos de uso más comunes en aplicaciones empresariales.
- Spring AOP utiliza varias **anotaciones** para definir aspectos y *advice*, como `@Aspect`, `@Before`, `@After`, `@Around`, etc. Estas anotaciones son parte de AspectJ, pero Spring interpreta estas anotaciones utilizando su propia implementación de AOP.
- Los aspectos en Spring AOP pueden configurarse utilizando XML o la configuración basada en anotaciones.

Definición de un aspecto de ejemplo en **Spring AOP**:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class LoggingAspect {

    @Before("execution(* com.example.service.*.*(..))")
    public void logBeforeMethod() {
        System.out.println("Logging before method execution");
    }
}
```

En este ejemplo, `LoggingAspect` es un aspecto que se aplica antes de la ejecución de cualquier método en la capa de servicio (`com.example.service`). El método `logBeforeMethod()` es el *advice*. La anotación `@Before` indica que el método `logBeforeMethod` debe ejecutarse antes de la ejecución de los métodos especificados por el *pointcut*.

@AspectJ support

@AspectJ se refiere a un estilo de declaración de aspectos utilizando clases Java normales anotadas con anotaciones específicas. Este estilo fue introducido por el proyecto AspectJ en la versión AspectJ 5.

Spring interpreta las anotaciones de AspectJ, utilizando una biblioteca proporcionada por AspectJ para el análisis y coincidencia de *pointcuts*. Sin embargo, el tiempo de ejecución de AOP sigue siendo puramente Spring AOP, y no hay dependencia del compilador o tejedor de AspectJ.

Esto significa que mientras que el estilo de definición de aspectos y los *pointcuts* se basan en las anotaciones de AspectJ, la implementación en tiempo de ejecución y la aplicación de los aspectos se realizan a través de Spring AOP.

Este enfoque permite que Spring use un estilo de declaración de aspectos más moderno y expresivo sin necesidad de integrar completamente el compilador AspectJ en el proceso de construcción de la aplicación.

- @AspectJ support - Spring Framework

Enabling @AspectJ Support

Para activar el soporte de AspectJ con Java, agregue la anotación `@EnableAspectJAutoProxy`, como muestra el siguiente ejemplo:

Para activar el soporte de AspectJ en una aplicación Spring, se debe agregar la anotación `@EnableAspectJAutoProxy` en una clase de configuración. Esta anotación habilita el uso de *proxies* basados en AspectJ, permitiendo que los aspectos definidos se

apliquen a los beans gestionados por Spring:

```
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.EnableAspectJAutoProxy;

@Configuration
@EnableAspectJAutoProxy
public class AppConfig {
    // Puedes definir otros beans aquí si es necesario
}
```

La anotación `@Configuration` indica que la clase contiene definiciones de *beans* y configuraciones de Spring, mientras que `@EnableAspectJAutoProxy` activa la capacidad de AOP utilizando *proxies*, facilitando la integración de aspectos en la aplicación.

- Enabling @AspectJ Support - Spring Framework

Declaring an Aspect

Con el soporte de @AspectJ habilitado, cualquier *bean* definido en el contexto de aplicación que sea una clase anotada con `@Aspect` es automáticamente detectado por Spring y utilizado para configurar Spring AOP. A continuación se muestra un ejemplo de cómo declarar un aspecto:

```
import org.aspectj.lang.annotation.Aspect;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class NotVeryUsefulAspect {
  // Métodos del aspecto se pueden definir aquí
}
```

Los aspectos, que son clases anotadas con `@Aspect`, pueden tener **métodos y campos**, al igual que cualquier otra clase en Java. Estos métodos pueden ser utilizados para definir el comportamiento del *advice* en puntos de unión específicos.

Las clases de aspecto se pueden registrar como *beans* normales mediante configuración XML de Spring, a través de métodos `@Bean` en clases anotadas con `@Configuration`, o hacer que Spring las detecte automáticamente mediante el escaneo del *classpath*, de la misma manera que cualquier otro *bean* administrado por Spring, siempre y cuando esté anotada con `@Component`, o cualquier otra anotación de estereotipo como `@Service`, etcétera..

- Declaring an Aspect - Spring Framework

Declaring a Pointcut

Los *pointcuts* determinan los puntos de unión de interés y, por lo tanto, nos permiten controlar **cuándo se ejecutan los *advice***. Spring AOP sólo admite puntos de unión de ejecución de métodos para *beans* de Spring.

Una declaración de *pointcut* tiene dos partes: una firma que comprende un nombre y cualquier parámetro, y una expresión de *pointcut* que determina exactamente qué ejecuciones de métodos nos interesan.

En el estilo de anotación `@AspectJ` de AOP, una firma de *pointcut* se proporciona mediante una definición de un método (este método debe tener un tipo de retorno `void`), y la expresión de *pointcut* se indica mediante el uso de la anotación `@Pointcut`:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.After;
```

```
@Aspect
public class MyAspect {

    // Definir el pointcut con una firma de método
    @Pointcut("execution(* com.example.service.*.*(..))")
    private void myPointcut() {
        // Método de marcador, el cuerpo está vacío
    }

    // Usar el pointcut en un consejo @Before
    @Before("myPointcut()")
    public void beforeAdvice() {
        System.out.println("Advice ejecutado antes del método objetivo");
    }

    // Otro consejo que usa el mismo pointcut
    @After("myPointcut()")
    public void afterAdvice() {
        System.out.println("Advice ejecutado después del método objetivo");
    }
}
```

Spring AOP soporta los siguientes **designadores de *pointcut*** de AspectJ (PCD) para su uso en expresiones:

- **execution**: Coincide con puntos de unión de ejecución de métodos. Este es el **designador principal** para trabajar con Spring AOP.
- **within**: Limita la coincidencia a los puntos de unión dentro de ciertos tipos.
- **this**: Limita la coincidencia a los puntos de unión donde la referencia del *bean* es una instancia del tipo dado.
- **target**: Limita la coincidencia a los puntos de unión donde el objeto objetivo es una instancia del tipo dado.
- **args**: Limita la coincidencia a los puntos de unión donde los argumentos son instancias de los tipos dados.
- **@target**: Limita la coincidencia a los puntos de unión donde la clase del objeto que ejecuta tiene una anotación del tipo dado.
- **@args**: Limita la coincidencia a los puntos de unión donde el tipo en tiempo de ejecución de los argumentos reales pasados tiene anotaciones de los tipos dados.
- **@within**: Limita la coincidencia a los puntos de unión dentro de tipos que tienen la anotación dada.
- **@annotation**: Limita la coincidencia a los puntos de unión donde el método que se ejecuta tiene la anotación dada.

Spring AOP añade un designador de *pointcut* adicional llamado **bean** y que no forma parte del estándar PCD de AspectJ. Este PCD te permite limitar la coincidencia de puntos de unión a un *bean* de Spring con un nombre específico o a un conjunto de *beans* de Spring con nombres específicos (cuando usas comodines). El PCD **bean** tiene la siguiente forma:

```
@Aspect
public class MyAspect {

    // Definir el pointcut usando el designador de bean
    @Pointcut("bean(myBean)")
    public void myBeanPointcut() {
        // Método de marcador, el cuerpo está vacío
    }

    // Definir el pointcut usando el designador de bean combinado con una expresión normal
    @Pointcut("bean(myServiceBean) && execution(* com.example.service.MyService.*(..))")
    public void myCombinedPointcut() {
```

```
        // Método de marcador, el cuerpo está vacío
    }

    // Definir el pointcut usando el designador de bean con comodines
    // limita la coincidencia a los beans cuyos nombres terminan en Service.
    @Pointcut("bean(*Service)")
    public void serviceBeansPointcut() {
        // Método de marcador, el cuerpo está vacío
    }

    @Before("myCombinedPointcut()")
    public void beforeAdvice() {
        System.out.println("Advice ejecutado");
    }
}
```

Puedes combinar expresiones de *pointcut* usando `&&`, `||` y `!`. También puedes referenciar expresiones de *pointcut* por nombre:

```
package com.xyz;

public class Pointcuts {

  @Pointcut("execution(public * *(..))")
  public void publicMethod() {}

  @Pointcut("within(com.xyz.trading..*)")
  public void inTrading() {}

  @Pointcut("publicMethod() && inTrading()")
  public void tradingOperation() {}

}
```

En aplicaciones empresariales, a menudo necesitas referirte a módulos y conjuntos particulares de operaciones desde varios aspectos. La recomendación es definir una **clase dedicada** para capturar **expresiones de *pointcut* con nombre comúnmente usadas**, como en el siguiente ejemplo de `CommonPointcuts`:

```
import org.aspectj.lang.annotation.Pointcut;

public class CommonPointcuts {

  /**
   * A join point is in the web layer if the method is defined
   * in a type in the com.xyz.web package or any sub-package
   * under that.
   */
  @Pointcut("within(com.xyz.web..*)")
  public void inWebLayer() {}

  /**
   * A join point is in the service layer if the method is defined
   * in a type in the com.xyz.service package or any sub-package
   * under that.
   */
  @Pointcut("within(com.xyz.service..*)")
  public void inServiceLayer() {}

  /**
   * A join point is in the data access layer if the method is defined
   * in a type in the com.xyz.dao package or any sub-package
   * under that.
   */
  @Pointcut("within(com.xyz.dao..*)")
  public void inDataAccessLayer() {}
```

```
  /**
   * A business service is the execution of any method defined on a service
   * interface. This definition assumes that interfaces are placed in the
   * "service" package, and that implementation types are in sub-packages.
   *
   * If you group service interfaces by functional area (for example,
   * in packages com.xyz.abc.service and com.xyz.def.service) then
   * the pointcut expression "execution(* com.xyz..service.*.*(..))"
   * could be used instead.
   *
   * Alternatively, you can write the expression using the 'bean'
   * PCD, like so "bean(*Service)". (This assumes that you have
   * named your Spring service beans in a consistent fashion.)
   */
  @Pointcut("execution(* com.xyz..service.*.*(..))")
  public void businessService() {}

  /**
   * A data access operation is the execution of any method defined on a
   * DAO interface. This definition assumes that interfaces are placed in the
   * "dao" package, and that implementation types are in sub-packages.
   */
  @Pointcut("execution(* com.xyz.dao.*.*(..))")
  public void dataAccessOperation() {}

}
```

- Declaring a Pointcut - Spring Framework
- The AspectJ Programming Guide - Eclipse
- The AspectJ 5 Development Kit Developer's Notebook - Eclipse

Declaring Advice

El *advice* está asociado con una expresión de *pointcut* y se ejecuta antes, después o alrededor de las ejecuciones de métodos que coinciden con el *pointcut*.

La expresión puede ser una expresión de *pointcut* en línea o una referencia a un *pointcut* con nombre.

- Declaring Advice - Spring Framework

Before Advice

Puedes declarar un *advice* que se ejecute antes en un aspecto utilizando la anotación `@Before`.

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class BeforeExample {

  // Expresión de pointcut en línea
  @Before("execution(* com.xyz.dao.*.*(..))")
  public void doAccessCheck() {
    // ...
  }

  // Pointcut con nombre
  @Before("com.xyz.CommonPointcuts.dataAccessOperation()")
  public void doAccessCheck() {
    // ...
  }
```

```
}
```

Este *advice* no tiene parámetros específicos, pero puede recibir un parámetro `JointPoint`

```
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class MyAspect {

    @Pointcut("execution(* com.example.service.*.*(..))")
    public void serviceMethods() {
        // Método de marcador, el cuerpo está vacío
    }

    @Before("serviceMethods()")
    public void beforeAdvice(JoinPoint joinPoint) {
        System.out.println("Antes de la ejecución del método: " + joinPoint.getSignature());
    }
}
```

La interfaz `JoinPoint` en Spring AOP proporciona una serie de métodos útiles que permiten acceder a información detallada sobre el método que está siendo interceptado.

- `getArgs()` : accede a los argumentos del método interceptado.
- `getThis()` : obtiene el objeto proxy de Spring AOP.
- `getTarget()` : obtiene el objeto objetivo que está siendo llamado.
- `getSignature()` : proporciona una descripción del método siendo aconsejado
- `toString()` : imprime una descripción legible del join point`

After Returning Advice

El *advice* se ejecuta cuando la ejecución del método coincidente finaliza con normalidad. Se puede declarar utilizando la anotación `@AfterReturning` .

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterReturning;

@Aspect
public class AfterReturningExample {

  @AfterReturning("execution(* com.xyz.dao.*.*(..))")
  public void doAccessCheck() {
    // ...
  }

}
```

A veces, se necesita acceso en el cuerpo del *advice* al valor real que se ha devuelto. Se puede usar la forma de `@AfterReturning` que vincula el valor de retorno para obtener ese acceso con el parámetro `returning` , como muestra el siguiente ejemplo:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterReturning;

@Aspect
public class AfterReturningExample {

  @AfterReturning(
  pointcut="execution(* com.xyz.dao.*.*(..))",
    returning="retVal")
  public void doAccessCheck(Object retVal) {
    // ...
  }

}
```

After Throwing Advice

El *advice* se ejecuta cuando la ejecución del método coincidente finaliza lanzando una excepción. Se puede declarar utilizando la anotación `@AfterThrowing`, como muestra el siguiente ejemplo:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterThrowing;

@Aspect
public class AfterThrowingExample {

  @AfterThrowing("execution(* com.xyz.dao.*.*(..))")
  public void doRecoveryActions() {
    // ...
  }

}
```

After (Finally) Advice

El *advice* se ejecuta cuando una ejecución de método coincidente termina, ya sea que haya terminado normalmente o haya lanzado una excepción. Se declara utilizando la anotación `@After`. Se usa para realizar tareas de limpieza o liberación de recursos que deben ejecutarse sin importar si el método termina correctamente o con error.

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.After;

@Aspect
public class AfterFinallyExample {

  @After("execution(* com.xyz.dao.*.*(..))")
  public void doReleaseLock() {
    // ...
  }
}
```

Around Advice

Este *advice* se ejecuta "alrededor" de la ejecución de un método coincidente. Tiene la oportunidad de realizar trabajo tanto antes como después de que el método se ejecute y puede determinar cuándo, cómo, e incluso si el método debe ejecutarse en absoluto.

Siempre se deb utilizarla forma de *advice* menos más específica y menos invasiva que satisfaga los requisitos. Por ejemplo, no uses *"around advice"* si *"before advice"* es suficiente.

```java
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.ProceedingJoinPoint;

@Aspect
public class AroundExample {

  @Around("execution(* com.xyz..service.*.*(..))")
  public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {
    // start stopwatch
    Object retVal = pjp.proceed();
    // stop stopwatch
    return retVal;
  }

}
```

En Spring AOP, el *around advice* requiere que se declare un primer parámetro de tipo `ProceedingJoinPoint`, que es una subclase de `JoinPoint`.

Referencias

- https://spring.io/
- https://docs.spring.io/spring-framework/reference/
- https://docs.spring.io/spring-boot/

Guías (Baeldung)

- Spring Framework Introduction
- Learn Spring Boot
- Spring Dependency Injection
- Spring MVC Guides
- REST with Spring Tutorial
- Security with Spring
- All Spring Data Guides

Licencia