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.

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.

```
@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`:

```
<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:

```
@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

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
```

- 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:

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

@Configuration
@PropertySource("classpath:config.properties")
```

```java
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:

```java
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.

```java
@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:

```java
@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

Podemos usar esta anotación 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

@ConfigurationProperties

La anotación `@ConfigurationProperties` es una anotación en Spring Boot que permite mapear propiedades externas (generalmente de archivos `.properties` o `.yaml`) a un objeto Java. Esta anotación simplifica la configuración al proporcionar una manera **estructurada y tipada** para acceder a las propiedades en una aplicación Spring.

La anotación `@ConfigurationProperties` acepta un prefijo que indica qué propiedades del archivo se deben mapear a esta clase.

Para que Spring reconozca y cargue las propiedades en la clase, se debe **habilitar el soporte** para `@ConfigurationProperties` en tu configuración de Spring. Esto se hace normalmente 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
```

El prefijo "app" coincide con el prefijo de las claves. Esta sería la clase de propiedades:

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

@Component
@ConfigurationProperties(prefix = "app")
```

```java
public class AppProperties {

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

    // Getters y setters
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getVersion() {
        return version;
    }

    public void setVersion(String version) {
        this.version = version;
    }

    public boolean isFeaturesEnabled() {
        return featuresEnabled;
    }

    public void setFeaturesEnabled(boolean featuresEnabled) {
        this.featuresEnabled = featuresEnabled;
    }
}
```

Y su uso en un componente:

```java
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
- 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

@RestController

La anotación `@RestController` es una meta-anotación que combina las anotaciones `@Controller` y `@ResponseBody`.

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

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

- Javadoc

@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");
    }
}
```

Si marcamos una clase con él, se aplica a todos los métodos del controlador de solicitudes que contiene. Podemos ajustar el comportamiento de CORS con los argumentos de esta anotación.

Se puede habilitar CORS globalmente para todos los controladores usando una clase de configuración como esta:

```
@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);
    }
}
```

Si se utiliza Spring Security en la aplicación, hay que asegurarse de que la configuración de CORS no entre en conflicto con las políticas de seguridad definidas en Spring Security.

- Javadoc
- CORS with Spring
- CORS

Request Handling Annotations

@RequestMapping

En pocas palabras, `@RequestMapping` marca métodos manejadores de peticiones dentro de clases anotadas con `@Controller`; puede configurarse utilizando:

- **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` si aplicamos esta anotación a nivel de **clase**. La única excepción es la URL, que Spring no sobrescribirá con configuraciones a nivel de método, sino que **anexará las dos partes del path**.

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";
```

```
    }
}
```

Además, `@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping` y `@PatchMapping` son variantes de `@RequestMapping` donde el método HTTP ya está configurado específicamente como GET, POST, PUT, DELETE y PATCH respectivamente.

Estas anotaciones están disponibles desde la **versión 4.3 de Spring**.

- Javadoc
- Mapping Requests

@RequestBody

La anotación `@RequestBody` mapea el cuerpo de la solicitud HTTP a un objeto:

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

La deserialización es automática y depende del tipo de contenido de la solicitud.

- Javadoc

@PathVariable

Esta anotación indica que un argumento de método está vinculado a una variable de plantilla URI. Si el nombre de la parte en la plantilla 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;
    }
}
```

Sin embargo, se puede vincular un argumento (o varios argumentos) de método a una de las partes de la plantilla con `@PathVariable` si se indica como argumento de la anotación.

```
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;
    }
}
```

Además, se puede marcar una variable de ruta como opcional estableciendo el argumento `required = false`:

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

- Javadoc

@RequestParam

Se utilizas `@RequestParam` 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;
}
```

Tiene las mismas opciones de configuración que la anotación `@PathVariable`.

Además de esas configuraciones, con `@RequestParam` podemos especificar un valor inyectado cuando Spring no encuentra ningún valor o está vacío en la solicitud. Para lograr esto, tenemos que establecer el argumento *"defaultValue"*:

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

- Javadoc
- @RequestParam

@ModelAttribute

La anotación `@ModelAttribute` en Spring MVC es utilizada para enlazar un método o parámetro de método a un atributo de modelo. Esta anotación es útil en el contexto de controladores de Spring MVC, donde se manejan las solicitudes HTTP y se prepara la respuesta.

Cuando se usa `@ModelAttribute` en un **parámetro de método**, Spring intenta enlazar los datos de la solicitud HTTP a ese objeto. Es útil para la recepción de datos del formulario o de la solicitud HTTP.

Con esta anotación podemos acceder a elementos que ya están en el modelo de un controlador MVC:

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

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

Cuando se anota un método de un controlador con `@ModelAttribute`, ese método se invoca antes de que cualquier método anotado con `@RequestMapping` dentro del mismo controlador se ejecute. Esto permite preparar y agregar atributos al modelo que se utilizarán en la vista.

Spring agregará automáticamente el valor de retorno del 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.

Al usar `@ModelAttribute`, Spring realiza automáticamente el enlace de datos entre los parámetros de solicitud y el objeto del modelo definido. Esto incluye la conversión de tipos de datos y la validación de los campos según las anotaciones de validación de Spring, como `@NotNull`, `@Size`, etc.

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
- Spring MVC and the @ModelAttribute Annotation
- @ModelAttribute
- Model

@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 específica 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, se requiere la presencia de la cookie. En caso contrario se lanza la excepción `MissingCookieValueException`. Se puede indicar que no es requerida con el argumento `required=false` en la anotación.

Se puedes usar `@CookieValue` varias veces en un método de controlador para recuperar múltiples valores de cookies:

```
@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" respectivamer
    return ResponseEntity.ok("Value of 'cookie1': " + cookie1 + ", Value of 'cookie2': " + cookie2);
}
```

- Javadoc
- @CookieValue

@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, se requiere la presencia de la cabecera, lo que significa que Spring espera encontrar la cabecera especificada. Si no se encuentra, se generará una excepción `MissingRequestHeaderException` a menos que se especifique un valor predeterminado a falso.

- Javadoc
- @RequestHeader

Response Handling Annotations

@ResponseBody

Si marcamos un método de controlador de solicitudes con `@ResponseBody`, Spring trata el resultado del método como la respuesta misma.

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 anotamos una clase @Controller con esta anotación, todos los métodos del controlador de solicitudes la usarán. Este es el efecto de `@RestController`, que no es más que una metaanotación marcada con `@Controller` y `@ResponseBody`

- Javadoc
- @ResponseBody

@ExceptionHandler

Con esta anotación, podemos declarar un método de manejo de errores personalizado. Spring llama a este método cuando un método de controlador de solicitudes genera cualquiera de las excepciones especificadas.

La excepción detectada se puede pasar al método como argumento:

```
@ExceptionHandler(IllegalArgumentException.class)
void onIllegalArgumentException(IllegalArgumentException exception) {
    // ...
}
```

- Javadoc
- Exceptions

@ResponseStatus

Podemos especificar el código de estado HTTP deseado de la respuesta si anotamos un método manejador de solicitud con esta anotación. Podemos declarar el código de estado con el argumento `code`, o su alias, el argumento `value`.

```
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!";
    }
}
```

Además, podemos proporcionar un motivo utilizando el argumento `reason`:

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

También podemos usarlo junto con `@ExceptionHandler`:

```
@ExceptionHandler(IllegalArgumentException.class)
@ResponseStatus(HttpStatus.BAD_REQUEST)
void onIllegalArgumentException(IllegalArgumentException exception) {
    // ...
}
```

- Javadoc
- Returning Custom Status Codes from Spring Controllers

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.

@SpringBootApplication

Esta anotación se utiliza para marcar la clase principal de una aplicación Spring Boot:

```
@SpringBootApplication
class VehicleFactoryApplication {

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

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

- Javadoc
- SpringApplication - Spring Boot

@EnableAutoConfiguration

Esta anotación `@EnableAutoConfiguration`, como su nombre indica, permite habilitar la configuración automática. Significa que Spring Boot busca *beans* de configuración automática en su classpath y los aplica automáticamente.

Hay que tener en cuenta que hay que usar esta anotación con `@Configuration`:

```
@Configuration
@EnableAutoConfiguration
class VehicleFactoryConfig {}
```

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

Por lo tanto, si se utiliza `@SpringBootApplication` no es necesario utilizar la anotación `@EnableAutoConfiguration`.

- Javadoc

- Auto-configuration - Spring Boot

Auto-Configuration Conditions

En el contexto de Spring Framework, las condiciones de auto-configuración (`@Conditional`) permiten condicionar la aplicación de configuraciones basadas en ciertas condiciones en tiempo de ejecución.

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

Spring proporciona **diversas condiciones predefinidas** listas para ser utilizadas.

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

- 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 solo se active si la condición especificada se cumple en tiempo de ejecución:

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

- Javadoc

@ConditionalOnClass and @ConditionalOnMissingClass

Usando estas condiciones, Spring solo usará el *bean* de configuración automática marcado si la clase en el argumento de la anotación está presente/ausente:

```
@Configuration
@ConditionalOnClass(DataSource.class)
class MySQLAutoconfiguration {
    //...
}
```

- Javadoc - @ConditionalOnClass
- Javadoc - @ConditionalOnMissingClass

@ConditionalOnBean and @ConditionalOnMissingBean

Podemos usar estas anotaciones cuando queramos definir condiciones basadas en la presencia o ausencia de un *bean* específico:

```
@Bean
@ConditionalOnBean(name = "dataSource")
```

```
LocalContainerEntityManagerFactoryBean entityManagerFactory() {
    // ...
}
```

- Javadoc - @ConditionalOnBean
- Javadoc - @ConditionalOnMissingBean

@ConditionalOnProperty

Con esta anotación, podemos poner condiciones sobre los valores de las propiedades:

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

- Javadoc - @ConditionalOnProperty

@ConditionalOnResource

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

```
@ConditionalOnResource(resources = "classpath:mysql.properties")
Properties additionalProperties() {
    // ...
}
```

- Javadoc - @ConditionalOnResource

@ConditionalOnWebApplication and @ConditionalOnNotWebApplication

Con estas anotaciones podemos crear condiciones en función de si la aplicación actual es o no una aplicación web:

```
@ConditionalOnWebApplication
HealthCheckController healthCheckController() {
    // ...
}
```

- 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() {
    // ...
}
```

- 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

@EnableAsync

Con esta anotación, podemos habilitar la funcionalidad asíncrona en Spring.

Debemos usarla junto con `@Configuration`:

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

Ahora que habilitamos las llamadas asincrónicas, podemos usar `@Async` para definir los métodos que las admiten.

- Javadoc

@EnableScheduling

Con esta anotación, podemos habilitar la programación en la aplicación.

Debemos usarla junto con `@Configuration`:

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

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

- Javadoc

@Async

La anotación `@Async` en Spring se utiliza para marcar un método como **asíncrono**, lo que permite que dicho método sea ejecutado en un hilo separado o en un pool de hilos dedicado 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 lograr esto, podemos anotar el método 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`)

Tenga en cuenta que debemos habilitar las llamadas asincrónicas para que funcione esta anotación, con `@EnableAsync` o 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;

@Service
public class MyService {

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

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

@Service public class MyService {

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

}

```
- [Javadoc](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/scheduling/annotation/A

- [How To Do @Async in Spring](https://www.baeldung.com/spring-async)

### @Scheduled

La anotación `@Scheduled` en Spring se utiliza para programar la ejecución de métodos a intervalos específicos, en momen

Si necesitamos que un método se ejecute periódicamente, podemos usar esta anotación:
```

java @Scheduled(fixedRate = 10000) void checkVehicle() { // … }

```
Podemos usarlo para ejecutar un método a **intervalos fijos**, o podemos ajustarlo con **expresiones similares a cron**:

- **'fixedRate'**: ejecuta el método con una tasa fija en milisegundos, independientemente de cuánto tiempo haya tomado
```

```
- **'fixedDelay'**: ejecuta el método con un retraso fijo en milisegundos después de que se completa la ejecución anter
- **'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.
`@Scheduled` aprovecha la función de anotaciones repetidas de Java 8, lo que significa que podemos marcar un método con
```

java @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 c
Para que `@Scheduled` funcione correctamente, la clase que contiene el método anotado debe ser administrada por Spring
- [Javadoc](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/scheduling/annotation/S
- [The @Scheduled Annotation in Spring](https://www.baeldung.com/spring-scheduled-tasks)
### @Schedules
Podemos usar esta anotación para especificar múltiples reglas `@Scheduled`:
```

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

```
Hay que tener en cuenta que desde Java 8 se puede lograr lo mismo con la función de anotaciones repetidas.
## [Spring Data Annotations](https://www.baeldung.com/spring-data-annotations)
**Spring Data** proporciona una abstracción sobre las tecnologías de almacenamiento de datos. Por lo tanto, nuestro cód
- [Javadoc](https://docs.spring.io/spring-data/commons/docs/current/api/)
- [Data Access](https://docs.spring.io/spring-framework/reference/data-access.html)
### Common Spring Data Annotations
#### @Transactional
La anotación `@Transactional` en Spring se utiliza para administrar transacciones en métodos o clases de manera declarat
Cuando queramos configurar el comportamiento transaccional de un método, podemos hacerlo con:
```

java @Transactional void pay() {}

```
Si aplicamos esta anotación a nivel de clase, funciona en todos los métodos dentro de la clase. Sin embargo, podemos anu
- [Javadoc](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/
- [Transactions with Spring and JPA](https://www.baeldung.com/transaction-configuration-with-jpa-and-spring)
#### @NoRepositoryBean
La anotación `@NoRepositoryBean` en Spring se utiliza para indicar a Spring que una interfaz de repositorio específica r
```

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

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

@Repository public interface UserRepository extends MyBaseRepository { // 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 m

- [Javadoc](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/NoRepository

#### @Param

Podemos pasar parámetros con nombre a nuestras consultas usando `@Param`:
```

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

```
Tenga en cuenta que nos referimos al parámetro con la sintaxis `:name`.

- [Javadoc](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/query/Param.

- [Spring Data JPA @Query](https://www.baeldung.com/spring-data-jpa-query)

#### @Id

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

java class Person {

```
@Id
Long id;

// ...
```

}

```
Dado que es independiente de la implementación, hace que una clase de modelo sea fácil de usar con múltiples motores de

- [Javadoc](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/annotation/Id.html)

#### @Transient

Podemos usar esta anotación para marcar un campo en una clase de modelo como **transitorio**. Por lo tanto, el motor del
```

java class Person {

```
// ...

@Transient
int age;

// ...
```

}

```
Al igual que `@Id`, `@Transient` también es independiente de la implementación, lo que hace que sea conveniente utilizar

- [Javadoc](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/annotation/Transient.ht

#### Campos de auditoría

En Spring Data, las anotaciones de campos de auditoría son utilizadas para mantener un rastro automático de ciertas acci

- **`@CreatedBy`**: se utiliza para marcar un campo que debe contener el usuario que creó la entidad. Este campo se llen

- **`@LastModifiedBy`**: indica el usuario que realizó la última modificación a la entidad. Este campo se actualiza auto

- **`@CreatedDate`**: marca un campo para almacenar la fecha y hora en que la entidad fue creada. Este campo se llena au

- **`@LastModifiedDate`**: se utiliza para anotar un campo que debe contener la fecha y hora de la última modificación d
```

java public class Person {

```
// ...

@CreatedBy
User creator;

@LastModifiedBy
User modifier;

@CreatedDate
Date createdAt;

@LastModifiedDate
Date modifiedAt;

// ...
```

}

```
Para utilizar estas anotaciones, se debe seguir algunos pasos adicionales para habilitar la auditoría en una aplicación

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

java @Configuration @EnableJpaAuditing(auditorAwareRef = "auditorProvider") public class AuditConfig { }

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

java @Component public class AuditorAwareImpl implements AuditorAware {

```
@Override
public Optional<String> getCurrentAuditor() {
    // Lógica para obtener el usuario actual, por ejemplo, del contexto de seguridad de Spring
    return Optional.of(SecurityContextHolder.getContext().getAuthentication().getName());
}
```

}

```
- [Javadoc](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/annotation/package-sum
- [Auditing with JPA, Hibernate, and Spring Data JPA](https://www.baeldung.com/database-auditing-jpa)
- [Auditing](https://docs.spring.io/spring-data/jpa/reference/auditing.html)
### Spring Data JPA Annotations
#### @Query
Con la anotación `@Query`, se proporciona proporcionar una **implementación JPQL** para un método de repositorio:
```

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

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

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

```
Spring se pueden utilizar consultas SQL nativas, si se configura el argumento `nativeQuery = true`:
```

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

```
- [Spring Data JPA @Query](https://www.baeldung.com/spring-data-jpa-query)
#### @Procedure
Con Spring Data JPA podemos llamar fácilmente a procedimientos almacenados desde repositorios.
Primero, se necesita declarar el repositorio en la clase de entidad usando anotaciones JPA estándar:
```

java @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 puedes hacer referencia a él en el repositorio con el nombre que se declaramos en el argumento `name
```

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

```
#### @Lock
La anotación `@Lock` en Spring es parte del módulo **Spring Data JPA** y se utiliza para especificar el nivel de bloque
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
- **`LockModeType.PESSIMISTIC_READ`: este bloqueo permite que otras transacciones lean los datos, pero no permite actua
- **`LockModeType.PESSIMISTIC_WRITE`**: este bloqueo evita que otras transacciones lean o actualicen los datos mientras
- **`LockModeType.PESSIMISTIC_FORCE_INCREMENT`**: adquiere un bloqueo de escritura pesimista y además incrementa la ver
```

```
- **`LockModeType.OPTIMISTIC`**: utiliza el bloqueo optimista, que se basa en versiones. Permite que múltiples transacci
- **`LockModeType.OPTIMISTIC_FORCE_INCREMENT`**: similar al bloqueo optimista, pero además incrementa la versión de la e
- **`LockModeType.READ`**: sinónimo de `OPTIMISTIC`. Poco usado en la práctica.
- **`LockModeType.WRITE`**: sinónimo de `OPTIMISTIC_FORCE_INCREMENT`. Poco usado en la práctica.
Los **bloqueos pesimistas** pueden afectar el rendimiento de la base de datos, ya que impiden que otras transacciones ac
Los **bloqueos optimistas** son más apropiados para sistemas con alta concurrencia, ya que permiten una mayor escalabili
- [Javadoc](https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/Lock.html)
- [Javadoc](https://jakarta.ee/specifications/persistence/2.2/apidocs/javax/persistence/lockmodetype)
- [Enabling Transaction Locks in Spring Data JPA](https://www.baeldung.com/java-jpa-transaction-locks)
- [Locking](https://docs.spring.io/spring-data/jpa/reference/jpa/locking.htm)
#### @Modifying
Se pueden modificar datos con un método de repositorio si se anota con `@Modifying`:
```

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

```
- [Spring Data JPA @Query](https://www.baeldung.com/spring-data-jpa-query)
#### @EnableJpaRepositories
Para utilizar repositorios JPA, se le tiene que indicar a Spring mediante la anotación `@EnableJpaRepositories`.
Hay que tener en cuenta que se debe usar esta anotación con la anotación `@Configuration`:
```

java @Configuration @EnableJpaRepositories class PersistenceJPAConfig {}

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

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

```
También hay que tener en cuenta que Spring Boot hace esto automáticamente si encuentra **Spring Data JPA en el classpath
### Spring Data Mongo Annotations
Spring Data hace que trabajar con MongoDB sea mucho más fácil.
- [Introduction to Spring Data MongoDB](https://www.baeldung.com/spring-data-mongodb-tutorial)
#### @Document
Esta anotación marca una clase como un objeto de dominio que queremos conservar en la base de datos:
```

java @Document class User {}

```
También nos permite elegir el nombre de la colección que queremos utilizar:
```

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

```
Esta anotación `@Document` es el equivalente en Mongo de `@Entity` en JPA.

#### @Field

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

java @Document class User {

```
// ...

@Field("email")
String emailAddress;

// ...
```

}

```
Esta anotación `@Field` es el equivalente en Mongo de `@Column` en JPA.

#### @Query

Con la anotación `@Query`, podemos proporcionar una consulta de buscador en un método de repositorio de MongoDB:
```

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

```
#### @EnableMongoRepositories

Para utilizar repositorios de MongoDB, tenemos que indicárselo a Spring. Podemos hacer esto con `@EnableMongoRepositorie
```

java @Configuration @EnableMongoRepositories class MongoConfig {}

```
Spring buscará repositorios en los subpaquetes de esta clase `@Configuration`. Podemos alterar este comportamiento con e
```

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

```
Spring Boot hace esto automáticamente si encuentra **Spring Data MongoDB en el classpath**, es decir, si se ha incluido

## [Spring Bean Annotations](https://www.baeldung.com/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 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

- **Análisis y detección implícita de beans y conexión automática de beans**: finalmente, podemos marcar la clase como

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

La recomendación es usar la **conexión automática** ya que requiere menos configuración explícita y además usar la **co
```

```
### @ComponentScan

Spring puede **escanear automáticamente** un paquete en busca de _beans_ si el [escaneo de componentes está habilitado](

Esta anotación `@ComponentScan` se utiliza junto a la anotación [`@Configuration`](#configuration).

La anotación `@ComponentScan` configura qué paquetes escanear en busca de _beans_. Podemos especificar los nombres de lo
```

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

```
Además, podemos indicar **clases** en los paquetes base con el argumento _"basePackageClasses"_, lo que mejora la seguri
```

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

```
Ambos argumentos son matrices, por lo que podemos proporcionar varios paquetes o clases para cada uno.
```

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

```
Si no se especifica ningún argumento, el escaneo se realiza desde el mismo paquete donde está presente la clase anotada

En este ejemplo, la clase principal se ubica en _"com.example.myapplication"_ por lo que ese se considera el paquete ra
```

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

```
La anotación `@ComponentScan` aprovecha la función de anotaciones repetidas de Java 8, lo que significa que podemos mar
```

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

```
Como alternativa, se puede usar la anotación `@ComponentScans` para especificar múltiples configuraciones de `@Component
```

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

```
- [Javadoc - `@ComponentScan`](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/con

- [Javadoc - `@ComponentScans`](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/co

- [Spring Component Scanning - Baeldung](https://www.baeldung.com/spring-component-scanning)

- [Structuring Your Code - Spring Boot](https://docs.spring.io/spring-boot/reference/using/structuring-your-code.html)

### @Component

La anotación `@Component` es una anotación de **nivel de clase**. Durante el [análisis de componentes con `@ComponentSca
```

java @Component class CarUtility { // … }

```
De forma predeterminada, las instancias de _bean_ de esta clase tienen **el mismo nombre que el nombre de la clase** cor
```

java 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_ implica descartar la anotación `@Component` y, en su lugar, usar la anotación
```

java import javax.inject.Named;

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

```
Dado que `@Repository`, `@Service`, `@Configuration` y `@Controller` son **metanotaciones** de `@Component`, comparten c

- [Javadoc](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Component.ht

- [Using JSR 330 Standard Annotations - Spring Framework](https://docs.spring.io/spring-framework/reference/core/beans/s

### @Repository

Las clases DAO o _Repository_ generalmente representan la capa de acceso a la base de datos en una aplicación y deben ar
```

java @Repository class VehicleRepository { // … }

```
Una ventaja de utilizar esta anotación es que tiene habilitada la **traducción automática de excepciones de persistencia

Cuando se utiliza un marco de persistencia, como Hibernate, las excepciones nativas lanzadas dentro de las clases anotad

Para habilitar la traducción de excepciones, necesitamos declarar nuestro propio bean _"PersistenceExceptionTranslationP
```

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

```
Tenga en cuenta que en la mayoría de los casos, Spring realiza el paso anterior automáticamente.

- [Javadoc](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Repository.h

### @Service

La **lógica de negocio** de una aplicación normalmente reside dentro de **la capa de servicio**, por lo que usaremos la
```

java @Service public class VehicleService { // …
}

```
- [Javadoc](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Service.html

### @Controller

La anotación `@Controller` es una anotación a **nivel de clase**, que le dice a Spring Framework que esta clase sirve co
```

java @Controller public class VehicleController { // … }

```
- [Javadoc](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Controller.h

### @Configuration

Las clases de configuración contienen **métodos de definición de _beans_** anotados con `@Bean`:
```

java @Configuration class VehicleFactoryConfig {

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

}

```
Esta anotación `@Configuration` es una **meta-anotación** o especialización de `@Component` que se utiliza para definir

- [Javadoc](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Conf

- [Basic Concepts: @Bean and @Configuration - Spring Framework](https://docs.spring.io/spring-framework/reference/core/b

- [Using the @Configuration annotation - Spring Framework](https://docs.spring.io/spring-framework/reference/core/beans/

## [Spring AOP](https://docs.spring.io/spring-framework/reference/core/aop.html)

La **Programación Orientada a Aspectos (AOP)** es un paradigma de programación que se enfoca en separar las preocupacion

Una preocupación transversal pueden definirse como cualquier funcionalidad que afecta a varios puntos de una aplicación,

La AOP es muy útil para **modularizar estas funcionalidades que se repiten** en diferentes partes de la aplicación y des

- [Aspect Oriented Programming with Spring - Spring Framework](https://docs.spring.io/spring-framework/reference/core/ao

- [Introduction to Spring AOP - Baeldung](https://www.baeldung.com/spring-aop)

### Conceptos de AOP

- **Aspecto (Aspect)**

  Un aspecto es un módulo que encapsula una preocupación transversal. En términos simples, es **una clase que contiene l

- **Consejo (Advice)**

  Un _advice_ es **la acción que un aspecto realiza en un punto específico** en el programa. Es el código que se ejecuta

  Además de describir el trabajo que un aspecto debe llevar a cabo, los _advice_ deben responder a la pregunta de **cúan

  - **`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 lla

- **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

- **Corte Transversal (Pointcut)**
```

```
  Un pointcut es una **expresión que selecciona uno o más puntos de unión** donde un advice debería ejecutarse. Define e
- **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 (Target Object)**
  El _"target object"_ es el objeto cuyo método está siendo interceptado. Es el **objeto en el que se aplican los aspect
### Implementación de AOP en Spring
**Spring AOP** es el framework de Spring para la programación orientada a aspectos. Utiliza **proxies dinámicos** para i
Spring AOP es una implementación de AOP más **simple y centrada** en las necesidades más comunes de los desarrolladores
Algunos elementos clave de Spring AOP:
- Spring AOP se puede usar con AspectJ. Este es un framework de AOP completo para Java que permite la programación orien
- Utiliza **proxies dinámicos** para aplicar aspectos en tiempo de ejecución. Esto significa que solo puede aplicar aspe
- Se centra en aspectos que se aplican **antes, después o alrededor** de la ejecución de métodos.
- Se pueden utilizar varias **anotaciones** para definir aspectos y _advice_, como `@Aspect`, `@Before`, `@After`, `@Arc
- Los aspectos pueden configurarse utilizando XML o la configuración basada en anotaciones.
Definición de un aspecto de ejemplo en **Spring AOP**:
```

java 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");
}
```

}

```
### [@AspectJ support](https://docs.spring.io/spring-framework/reference/core/aop/ataspectj.html)
**@AspectJ** se refiere a un estilo de declaración de aspectos como clases Java normales anotadas con anotaciones. El es
#### [Enabling @AspectJ Support](https://docs.spring.io/spring-framework/reference/core/aop/ataspectj/aspectj-support.ht
Para activar el soporte de AspectJ con Java, agregue la anotación `@EnableAspectJAutoProxy`, como muestra el siguiente e
```

java @Configuration @EnableAspectJAutoProxy public class AppConfig { }

```
#### [Declaring an Aspect](https://docs.spring.io/spring-framework/reference/core/aop/ataspectj/at-aspectj.html)
Con el soporte de @AspectJ habilitado, cualquier _bean_ definido en el contexto de aplicación con una clase que sea un a
```

java package com.xyz;

import org.aspectj.lang.annotation.Aspect;

@Aspect public class NotVeryUsefulAspect { // … }

```
Los aspectos (clases anotadas con `@Aspect`) pueden tener métodos y campos, al igual que cualquier otra clase.

Las clases de aspecto se pueden registrar como _beans_ normales mediante configuración XML de Spring, a través de métod

Sin embargo, hay que tener en cuenta que la anotación `@Aspect` no es suficiente para la detección automática en el clas
```

java package com.xyz;

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

@Aspect @Component public class NotVeryUsefulAspect { // … }

```
#### [Declaring a Pointcut](https://docs.spring.io/spring-framework/reference/core/aop/ataspectj/pointcuts.html)

Los _pointcuts_ determinan los puntos de unión de interés y, por lo tanto, nos permiten controlar cuándo se ejecutan los

Una declaración de pointcut tiene dos partes: una firma que comprende un nombre y cualquier parámetro, y una expresión d

En el estilo de anotación @AspectJ de AOP, una firma de _pointcut_ se proporciona mediante una definición de un método
```

java @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**: Para coincidir con puntos de unión de ejecución de métodos. Este es el principal designador de pointcut

- **within**: Limita la coincidencia a los puntos de unión dentro de ciertos tipos (la ejecución de un método declarado

- **this**: Limita la coincidencia a los puntos de unión (la ejecución de métodos cuando usas Spring AOP) donde la refer

- **target**: Limita la coincidencia a los puntos de unión (la ejecución de métodos cuando usas Spring AOP) donde el obj

- **args**: Limita la coincidencia a los puntos de unión (la ejecución de métodos cuando usas Spring AOP) donde los argu

- **@target**: Limita la coincidencia a los puntos de unión (la ejecución de métodos cuando usas Spring AOP) donde la cl
```

```
- **@args**: Limita la coincidencia a los puntos de unión (la ejecución de métodos cuando usas Spring AOP) donde el tipo

- **@within**: Limita la coincidencia a los puntos de unión dentro de tipos que tienen la anotación dada (la ejecución d

- **@annotation**: Limita la coincidencia a los puntos de unión donde el sujeto del punto de unión (el método que se eje

Spring AOP añade un designador de pointcut adicional llamado **bean** y que no forma parte del estándar PCD de AspectJ.
```

java @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_
```

java 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() {}

}

```
Al trabajar con aplicaciones empresariales, los desarrolladores a menudo necesitan referirse a módulos de la aplicación
```

java package com.xyz;

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() {}

}

```
- [The AspectJ Programming Guide - Eclipse](https://eclipse.dev/aspectj/doc/released/progguide/index.html)

- [The AspectJ 5 Development Kit Developer's Notebook - Eclipse](https://eclipse.dev/aspectj/doc/released/adk15notebook/

#### [Declaring Advice](https://docs.spring.io/spring-framework/reference/core/aop/ataspectj/advice.html)

El _advice_ está asociado con una expresión de _pointcut_ y se ejecuta antes, después o alrededor de las ejecuciones de

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

##### Before Advice

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

java 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`](https://eclipse.dev/aspect
```

java 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 detalla

- **`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
```

java 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 `@Afte
```

java 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 uti
```

java 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
```

java 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

Siempre se deb utilizarla forma de _advice_ menos más específica y menos invasiva que satisfaga los requisitos. Por ejer
```

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