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.

> Introducción 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 a menos que se haya declarado al menos dos constructores.

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

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

java` // Field injection class Car { @Autowired Engine engine; }

```
`@Autowired` tiene un argumento booleano llamado `required` con un valor predeterminado de `true`. Este argumento ajusta

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

- [Javadoc](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/annotati

- [Guide to Spring @Autowired](https://www.baeldung.com/spring-autowire)

- [Constructor Dependency Injection in Spring](https://www.baeldung.com/constructor-injection-in-spring)

#### @Bean

La anotación `@Bean` marca un _'factory method'_ que crea una instancia de un _bean_ de Spring:
```

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

```
**Spring llama a estos métodos** cuando se requiere una nueva instancia del tipo de retorno.

El bean resultante tiene el mismo nombre que el _'factory method'_. Si se requiere que tenga un nombre diferente, se pue
```

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

```
Hay que tener en cuenta que **todos los métodos anotados con `@Bean` deben estar en clases `@Configuration`**.

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

```
#### @Qualifier

Se usa la anotación `@Qualifier` junto con `@Autowired` para proporcionar la **identificación del bean** o el **nombre d
```

java class Bike implements Vehicle {}

class Car implements Vehicle {}

```
En caso de ambigüedad, se utiliza `@Qualifier` para indicar a Spring **el bean a inyectar**:
```

java // 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;

```
- [Javadoc](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/annotati

#### @Value

Se puede utilizar la anotación `@Value` para inyectar valores de propiedad en beans. Es compatible con constructorres, m
```

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

```
Por supuesto, inyectar valores estáticos no es útil. Por lo tanto, podemos usar **cadenas _'placeholder'_** en `@Value`

Por ejemplo, un valor en un fichero externo podría ser:
```

text engine.fuelType=petrol

```
Podemos inyectar el valor de esta forma:
```

java @Value("${engine.fuelType}") String fuelType;

```
- [Javadoc](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/annotati

- [A Quick Guide to Spring @Value](https://www.baeldung.com/spring-value-annotation)

#### @DependsOn

La anotación `@DependsOn` se puede utilizar para hacer que Spring **inicialice otros beans antes del anotado**. Normalme

Solo necesitamos esta anotación cuando **las dependencias están implícitas**, por ejemplo, carga del controlador JDBC o

Podemos usar `@DependsOn` en la clase dependiente especificando los nombres de los beans de dependencia. El argumento de
```

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

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

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

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

#### @Lazy

La anotación `@Lazy` se utiliza cuando queremos inicializar un bean de forma diferida. De forma predeterminada, Spring c

Sin embargo, hay casos en los que necesitamos crear un bean cuando se solicita solicitamos, no al iniciar la aplicación.

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

Por ejemplo, marcar beans para que se carguen inmediatamente cuando la configuración global es diferida, o configurar me
```

java @Configuration @Lazy class VehicleFactoryConfig {

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

}

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

- [A Quick Guide to the Spring @Lazy Annotation](https://www.baeldung.com/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 i

- [Javadoc](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/annotati

- [@Lookup Annotation in Spring](https://www.baeldung.com/spring-lookup)

#### @Primary

A veces necesitamos definir **múltiples beans del mismo tipo**. En estos casos, la inyección no tendrá éxito porque Spri

Ya vimos una opción para manejar este escenario: marcar todos los puntos de conexión con `@Qualifier` y especificar el n

Sin embargo, la mayoría de las veces necesitamos un bean específico y rara vez los otros. Podemos usar `@Primary` para s
```

java @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 c

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

```
#### @Scope

Usamos `@Scope` para definir el ámbito de una clase `@Component` o una definición de `@Bean`. Puede ser **_singleton_**
```

java @Component @Scope("prototype") class Engine {}

```
**El ámbito por defecto es 'singleton'**. Esto significa que Spring crea una única instancia del bean y la reutiliza en

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

### Context Configuration Annotations

#### @Profile

Si queremos que Spring use una clase `@Component` o un método `@Bean` solo cuando un perfil específico esté activo, pode
```

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

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

- [Spring Profiles](https://www.baeldung.com/spring-profiles)

#### @Import

La anotación `@Import` en Spring se utiliza para importar configuraciones adicionales a una configuración principal de I

Si tenemos una clase anotada con `@Configuration` que define beans y configuraciones específicas para la aplicación, se
```

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

```
Otro uso de esta antoación es que podemos utilizar **clases específicas anotadas con `@Configuration` sin escaneo de cor
```

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

```
Por último, esta anotación sirve para importar clases no relacionadas con `@Configuration`, es decir, además de importar
```

java @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.

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

#### @ImportResource

Podemos importar configuraciones XML con esta anotación. Podemos especificar las ubicaciones de los archivos XML utiliza
```

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

```
La anotación `@ImportResource` se utiliza exclusivamente para importar configuraciones desde archivos XML dentro del cor
```

```
En contraste, `@Import` se utiliza para importar configuraciones y componentes de otras clases de configuración, ya sea

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

#### @PropertySource

Con esta anotación, podemos definir archivos de propiedades ('.properties' o '.yml') para la configuración de la aplicac
```

java 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 contienen configuraciones como URLs de bases de datos, rutas de archivos, configuraciones de conexión, et

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

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");
    // Utilizar la propiedad dbUrl...
}
```

}

```
`@PropertySource` aprovecha la función de anotaciones repetidas de Java 8, lo que significa que podemos marcar una clase
```

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

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

#### @PropertySources

Podemos usar esta anotación para especificar múltiples configuraciones de `@PropertySource`:
```

java @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](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Prop

---

## [Spring Web Annotations](https://www.baeldung.com/spring-mvc-annotations)
```

```
Estas anotaciones forman parte del paquete [org.springframework.web.bind.annotation](https://docs.spring.io/spring-frame

- [Spring Web MVC](https://docs.spring.io/spring-framework/reference/web/webmvc.html)

### @RestController

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

java @Controller @ResponseBody class VehicleRestController { // … }

java @RestController class VehicleRestController { // … }

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

### @CrossOrigin

La anotación `@CrossOrigin` en Spring Framework es utilizada para configurar las políticas de intercambio de recursos en

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

java @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

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

java @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

- [Javadoc](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/Cro

- [CORS with Spring](https://www.baeldung.com/spring-cors)

- [CORS](https://docs.spring.io/spring-framework/reference/web/webmvc-cors.html)

### Request Handling Annotations

#### @RequestMapping

En pocas palabras, `@RequestMapping` marca métodos manejadores de peticiones dentro de clases anotadas con `@Controller`

- **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.
```

java @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 ap

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

java @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 `@RequestMappi

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

- [Javadoc](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/Req

- [Mapping Requests](https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-requestmapping.html

#### @RequestBody
```

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

java @PostMapping("/save") void saveVehicle(@RequestBody Vehicle vehicle) { // … }

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

- [Javadoc](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/Re

#### @PathVariable

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

java 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 `@Pa
```

java 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`:
```

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

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

#### @RequestParam

Se utilizas `@RequestParam` para acceder a los parámetros de solicitud HTTP:
```

java // "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 ni
```

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

```
- [Javadoc](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/Re
- [@RequestParam](https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-methods/requestparam.h
#### @ModelAttribute
La anotación `@ModelAttribute` en Spring MVC es utilizada para enlazar un método o parámetro de método a un atributo de
Cuando se usa `@ModelAttribute` en un **parámetro de método**, Spring intenta enlazar los datos de la solicitud HTTP a e
Con esta anotación podemos acceder a elementos que ya están en el modelo de un controlador MVC:
```

java // 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) { // … }

```
Cuando se anota un método de un controlador con `@ModelAttribute`, ese método se invoca antes de que cualquier método ar
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 **e
Al usar `@ModelAttribute`, Spring realiza automáticamente el enlace de datos entre los parámetros de solicitud y el obje
En el siguiente ejemplo, se muestra un formulario en Spring:
```

html Name

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

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

```
En el controlador, se redirige a una vista JSP pero antes, se recuperan los datos enviados desde el formulario y se añad
```

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](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/Mo

- [Spring MVC and the @ModelAttribute Annotation](https://www.baeldung.com/spring-mvc-and-the-modelattribute-annotation

- [@ModelAttribute](https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-methods/modelattrib

- [Model](https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-modelattrib-methods.html)

#### @CookieValue

La anotación `@CookieValue` en Spring Framework se utiliza para enlazar el valor de una cookie HTTP específica a un para
```

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

```
    @GetMapping("/getCookie")
    public ResponseEntity<String> getCookieValue(@CookieValue(name = "myCookie", defaultValue = "default") String cookieValu
        // 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 `MissingCookieValueExceptio

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

java @GetMapping("/getCookies") public ResponseEntity 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" respectivamente return ResponseEntity.ok("Value of 'cookie1': " + cookie1 + ", Value of 'cookie2': " + cookie2); }

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

```
- [@CookieValue](https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-methods/cookievalue.ht

#### @RequestHeader

La anotación `@RequestHeader` en Spring Framework se utiliza para enlazar el valor de una cabecera HTTP específica a un
```

java @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 especifi

- [Javadoc](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/Re

- [@RequestHeader](https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-methods/requestheade

### Response Handling Annotations

#### @ResponseBody

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

Es decir, cuando un método de controlador anotado con `@ResponseBody` se invoca y retorna un objeto, Spring convierte au

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

java 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

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

```
- [@ResponseBody](https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-methods/responsebody.h

#### @ExceptionHandler

Con esta anotación, podemos declarar un método de manejo de errores personalizado. Spring llama a este método cuando un

La excepción detectada se puede pasar al método como argumento:
```

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

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

- [Exceptions](https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-exceptionhandler.html)

#### @ResponseStatus

Podemos especificar el código de estado HTTP deseado de la respuesta si anotamos un método manejador de solicitud con es
```

java 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`:
```

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

```
También podemos usarlo junto con `@ExceptionHandler`:
```

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

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

- [Returning Custom Status Codes from Spring Controllers](https://www.baeldung.com/spring-mvc-controller-custom-http-st

## [Spring Boot Annotations](https://www.baeldung.com/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](https://docs.spring.io/spring-boot/a

### @SpringBootApplication
```

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

java @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`

- [Javadoc](https://docs.spring.io/spring-boot/api/java/org/springframework/boot/autoconfigure/SpringBootApplication.ht

- [SpringApplication](https://docs.spring.io/spring-boot/reference/features/spring-application.html)

### @EnableAutoConfiguration

Esta anotación `@EnableAutoConfiguration`, como su nombre indica, permite la configuración automática. Significa que Spr

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

java @Configuration @EnableAutoConfiguration class VehicleFactoryConfig {}

```
- [Javadoc](https://docs.spring.io/spring-boot/api/java/org/springframework/boot/autoconfigure/EnableAutoConfiguration.

### Auto-Configuration Conditions

En el contexto de Spring Framework, las condiciones de auto-configuración (`@Conditional`) permiten condicionar la aplic

Estas condiciones se utilizan ampliamente en la auto-configuración automática de Spring Boot y en la configuración perso

Spring proporciona diversas condiciones predefinidas que listas para utilizar.

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

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

- [Create a Custom Auto-Configuration with Spring Boot](https://www.baeldung.com/spring-boot-custom-auto-configuration)

- [Creating Your Own Auto-configuration](https://docs.spring.io/spring-boot/reference/features/developing-auto-configur

#### @Conditional

La anotación `@Conditional` se utiliza para aplicar una condición a un componente de Spring, como un bean o una configur
```

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

```
#### @ConditionalOnClass and @ConditionalOnMissingClass

Usando estas condiciones, Spring solo usará el bean de configuración automática marcado si la clase en el argumento de
```

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

```
#### @ConditionalOnBean and @ConditionalOnMissingBean

Podemos usar estas anotaciones cuando queramos definir condiciones basadas en la presencia o ausencia de un bean especí
```

java @Bean @ConditionalOnBean(name = "dataSource") LocalContainerEntityManagerFactoryBean entityManagerFactory() { // … }

```
#### @ConditionalOnProperty

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

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

```
#### @ConditionalOnResource

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

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

```
#### @ConditionalOnWebApplication and @ConditionalOnNotWebApplication

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

java @ConditionalOnWebApplication HealthCheckController healthCheckController() { // … }

```
#### @ConditionalExpression

Podemos utilizar esta anotación en situaciones más complejas. Spring usará la definición marcada cuando la expresión Sp
```

java @Bean @ConditionalOnExpression("${usemysql} && ${mysqlserver == 'local'}") DataSource dataSource() { // … }

```
## [Spring Scheduling Annotations](https://www.baeldung.com/spring-scheduling-annotations)

Cuando la ejecución de un solo hilo no es suficiente, podemos usar anotaciones del paquete [org.springframework.schedul
- [Task Execution and Scheduling](https://docs.spring.io/spring-framework/reference/integration/scheduling.html#schedul
### @EnableAsync

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

Debemos usarla junto con `@Configuration`:
```

java @Configuration @EnableAsync class VehicleFactoryConfig {}

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

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

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

```
Debemos usarla junto con `@Configuration`:
```

java @Configuration @EnableScheduling class VehicleFactoryConfig {}

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

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

### @Async

La anotación `@Async` en Spring se utiliza para marcar un método como **asíncrono**, lo que permite que dicho método sea

Para lograr esto, podemos anotar el método con `@Async`:
```

java @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 (norm

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

Un método anotado con `@Async` puede devolver un valor envuelto en un _"Future"_ si se necesita obtener el resultado de
```

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");
}
```

}

```
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
- How To Do @Async in Spring

@Scheduled

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

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

```
@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 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.

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

```
@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
- The @Scheduled Annotation in Spring

@Schedules

Podemos usar esta anotación para especificar múltiples reglas `@Scheduled`:

```
@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

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
- Data Access

Common Spring Data Annotations

@Transactional

La anotación `@Transactional` en Spring se utiliza para administrar transacciones en métodos o clases de manera declarativa. 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.

Cuando queramos configurar el comportamiento transaccional de un método, podemos hacerlo con:

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

Si aplicamos esta anotación a nivel de clase, funciona en todos los métodos dentro de la clase. Sin embargo, podemos anular sus efectos aplicándolo a un método específico.

- Javadoc
- Transactions with Spring and JPA

@NoRepositoryBean

La anotación `@NoRepositoryBean` en Spring se utiliza para indicar a Spring que una interfaz de repositorio específica no debe ser considerada como un repositorio gestionado por **Spring Data**. Esto significa que no se creará una instancia de Spring para esa interfaz de repositorio en particular, y por lo tanto, no se aplicarán las características de repositorio típicas 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**.

- Javadoc

@Param

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

```
@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
- Spring Data JPA @Query

@Id

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

```
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 almacenamiento de datos.

- Javadoc

@Transient

Podemos usar esta anotación para marcar un campo en una clase de modelo como **transitorio**. Por lo tanto, el motor del almacén de datos no leerá ni escribirá el valor de este campo:

```
class Person {

    // ...

    @Transient
    int age;

    // ...

}
```

Al igual que `@Id`, `@Transient` también es independiente de la implementación, lo que hace que sea conveniente utilizarlo con múltiples implementaciones de almacenes de datos.

- Javadoc

Campos de auditoría

En Spring Data, las anotaciones de campos de auditoría son utilizadas 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 la entidad, el usuario que la modificó, y las fechas de creación y modificación:

- `@CreatedBy`: se utiliza para marcar 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`: se utiliza para anotar 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, se debe 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:

```
@Component
public class AuditorAwareImpl implements AuditorAware<String> {

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

Spring Data JPA Annotations

@Query

Con la anotación `@Query`, se proporciona proporcionar 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);
```

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

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

- 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:

```
@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`:

```
@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 bloqueo que se debe aplicar a una consulta JPA en las operaciones de acceso a la base de datos. Esta anotación es útil para controlar la concurrencia y evitar problemas como condiciones de carrera, especialmente en entornos donde múltiples transacciones pueden 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 y además incrementa la versión de la entidad. Asegura 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. Es útil para asegurar que cualquier otra transacción que quiera leer los datos tendrá que esperar hasta que la transacción actual complete.
- ** `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 accedan a los datos bloqueados.

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

- Javadoc
- Javadoc
- Enabling Transaction Locks in Spring Data JPA
- Locking

@Modifying

Se pueden modificar datos con un método de repositorio si se anota 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);
```

- 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`:

```
@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 {}
```

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

@Document

Esta anotación marca una clase como un objeto de dominio que queremos conservar en la base de datos:

```
@Document
class User {}
```

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

```
@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 documento:

```
@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:

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

@EnableMongoRepositories

Para utilizar repositorios de MongoDB, tenemos que indicárselo a Spring. Podemos hacer esto con `@EnableMongoRepositories`. Esta anotación se tiene que utilizar con la anotación `@Configuration`:

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

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

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

Spring Boot hace esto automáticamente si encuentra **Spring Data MongoDB en el classpath**.

Spring Bean Annotations

Hay varias formas de configurar *beans* en un contenedor Spring. En primer lugar, podemos declararlos usando la configuración XML. También podemos declarar *beans* usando la anotación `@Bean` en una clase de configuración.

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.

@ComponentScan

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

La anotación `@ComponentScan` configura qué paquetes escanear en busca de clases con anotación de configuración. Podemos especificar los nombres de los **paquetes** base directamente con uno de los argumentos "*basePackages*" o *"value"* (alias para *"basePackages"*):

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

Además, podemos señalar **clases** en los paquetes base con el argumento *"basePackageClasses"*:

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

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

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

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

```
@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 `@ComponentScan`:

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

- Javadoc - `@ComponentScan`
- Javadoc - `@ComponentScans`
- Spring Component Scanning

@Component

La anotación `@Component` es una anotación de **nivel de clase**. Durante el análisis de componentes, Spring Framework detecta automáticamente las clases anotadas con `@Component`:

```
@Component
class CarUtility {
```

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

De forma predeterminada, las instancias de *bean* de esta clase tienen el mismo nombre que el nombre de la clase con una inicial en minúscula. Además, podemos especificar un nombre diferente utilizando el argumento de *"value"* opcional de esta anotación.

Dado que `@Repository`, `@Service`, `@Configuration` y `@Controller` son **metanotaciones** de `@Component`, comparten el mismo comportamiento de denominación de *beans*. Spring también los detecta automáticamente durante el proceso de escaneo de componentes.

- Javadoc

@Repository

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

```
@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 anotadas con `@Repository` se traducirán automáticamente a subclases de `DataAccessExeption` de Spring.

Para habilitar la traducción de excepciones, necesitamos declarar nuestro propio bean *"PersistenceExceptionTranslationPostProcessor"*:

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

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

- Javadoc

@Service

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

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

- Javadoc

@Controller

La anotación `@Controller` es una anotación a **nivel de clase**, que le dice a Spring Framework que esta clase sirve como controlador en Spring MVC:

```
@Controller
public class VehicleController {
    // ...
}
```

- Javadoc

@Configuration

Las clases de configuración pueden contener **métodos de definición de *beans*** anotados con `@Bean`:

```
@Configuration
class VehicleFactoryConfig {

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

}
```

- Javadoc

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