

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](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org.springframework.beans.factory.annotation/) y [org.springframework.context.annotation](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org.springframework.context.annotation/).

@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.annotation.Autowired.html)
- [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 AppConfig { @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 puede usar `@Qualifier`.

```
java @Configuration public class AppConfig { @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.Bean.html)
- ### @Qualifier

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

```
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/annotation/>)

@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/annotation/>)

- [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/DependsOn.html)

### @Lazy

La anotación `@Lazy` se utiliza cuando queremos inicializar un bean de forma diferida. De forma predeterminada, Spring crea los beans al iniciar la aplicación. 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 predeterminado. Por ejemplo, marcar beans para que se carguen inmediatamente cuando la configuración global es diferida, o configurar métodos de inicialización.
```

```
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.html)
- [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 solicite.

- [Javadoc](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/annotation/LookupMethod.html)
- [@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 Spring no sabe cuál de los beans debe inyectar. Ya vimos una opción para manejar este escenario: marcar todos los puntos de conexión con `@Qualifier` y especificar el nombre del bean. Sin embargo, la mayoría de las veces necesitamos un bean específico y rara vez los otros. Podemos usar `@Primary` para marcar un bean como el principal.
```

```
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 de tipo _'Vehicle'_ .

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

### @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/Scope.html>)

@Profile

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

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

- [Javadoc](<https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Profile.html>)
- [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 la aplicación.

Si tenemos una clase anotada con `@Configuration` que define beans y configuraciones específicas para la aplicación, se puede importar en la configuración principal.

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

Otro uso de esta anotación es que podemos utilizar ****clases específicas anotadas con '@Configuration' sin escaneo de clases** para importar configuraciones.

```
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 configuraciones, se pueden importar clases de utilidad.

```
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/ImportResource.html>)

@ImportResource

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

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

La anotación `@ImportResource` se utiliza exclusivamente para importar configuraciones desde archivos XML dentro del contexto de la aplicación.

En contraste, `@Import` se utiliza para importar configuraciones y componentes de otras clases de configuración, ya sean `@Configuration` o `@Component`.

- [Javadoc](<https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/ImportResource.html>)

@PropertySource

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

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

```
}
```

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

```
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/PropertySource.html>)

@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/PropertySources.html>)

[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-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/package-summary.html>)

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

@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 así lo queremos.

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 `@RequestMapping`.

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/RequestMapping.html>)
- [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/RequestBody.html>)

@PathVariable

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

```
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/PathVariable.html>)

@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/RequestMapping.html)
 - [RequestParam](https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-methods/requestparam.html)
- ### ### @ResponseBody

Si marcamos un método de controlador de solicitudes con `@ResponseBody`, Spring trata el resultado del método como la respuesta. Es decir, cuando un método de controlador anotado con `@ResponseBody` se invoca y retorna un objeto, Spring convierte automáticamente el objeto a una cadena de texto. 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 es el caso de los siguientes ejemplos:

- [Javadoc](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestMapping.html)
 - [ResponseBody](https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-methods/responsebody.html)
- ### ### @ExceptionHandler

Con esta anotación, podemos declarar un método de manejo de errores personalizado. Spring llama a este método cuando una 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/ExceptionHandler.html)
 - [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 esta anotación:

```
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.RestController.html>)
- [Returning Custom Status Codes from Spring Controllers](<https://www.baeldung.com/spring-mvc-controller-custom-http-status-codes>)

@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.ResponseBody.html>)

@ModelAttribute

La anotación `@ModelAttribute` en Spring MVC es utilizada para enlazar un método o parámetro de método a un atributo de modelo.

Cuando se usa `@ModelAttribute` en un **parámetro de método**, Spring intenta enlazar los datos de la solicitud HTTP a un atributo de modelo.

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 de 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 necesario que el controlador devuelva un objeto de tipo `Model` o `ModelAndView`. Al usar `@ModelAttribute`, Spring realiza automáticamente el enlace de datos entre los parámetros de solicitud y el objeto de destino. 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 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](<https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/ModelAttribute.html>)
- [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/modelattribute.html>)

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

@CrossOrigin

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

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

```
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 ajustarlo para que solo se aplique a ciertos métodos.

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 con la de Spring Security.

- [Javadoc](<https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/CrossOrigin.html>)

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

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

@CookieValue

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

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

```
    @GetMapping("/getCookie")
    public ResponseEntity<String> getCookieValue(@CookieValue(name = "myCookie", defaultValue = "default") String cookieValue) {
        // 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 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/CookieValue.html>)
- [`@CookieValue`](<https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-methods/cookievalue.html>)

@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 específica.

- [Javadoc](<https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestHeader.html>)
- [`@RequestHeader`](<https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-methods/requestheader.html>)

[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/docs/current/api/org.springframework.boot.autoconfigure/>)

@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.html>)
- [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 Spring Boot intenta encontrar y configurar automáticamente los componentes necesarios para que la aplicación funcione.

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.html>)

Auto-Configuration Conditions

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

Estas condiciones se utilizan ampliamente en la auto-configuración automática de Spring Boot y en la configuración personalizada de la aplicación.

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/Conditional.html>)
- [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-configuration.html>)

@Conditional

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

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

@ConditionalOnClass and @ConditionalOnMissingClass

TODO

@ConditionalOnBean and @ConditionalOnMissingBean

TODO

@ConditionalOnProperty

TODO

@ConditionalOnResource

TODO

@ConditionalOnWebApplication and @ConditionalOnNotWebApplication

TODO

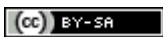
@ConditionalExpression

TODO

Referencias

- <https://spring.io/>
- <https://docs.spring.io/spring-framework/reference/>
- <https://docs.spring.io/spring-boot/docs/current/reference/html>
- <https://www.baeldung.com/spring-core-annotations>
- <https://www.baeldung.com/spring-boot-start>

Licencia



Esta obra está bajo una [licencia de Creative Commons Reconocimiento-Compartir Igual 4.0 Internacional](https://creativecommons.org/licenses/by-sa/4.0/).