Spring Framework

… EN DESARROLLO …

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

@Autowired

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

### @Bean

TODO

### @Qualifier

TODO

### @Value

TODO

### @DependsOn

Spring, by default, manages beans’ lifecycle and arranges their initialization order.

But, we can still customize it based on our needs. We can choose either the SmartLifeCycle interface or the `@DependsOn`

We can use the `@DependsOn` annotation and its behavior in case of a missing bean or circular dependency. Or in case of
```

java @Configuration @ComponentScan("com.baeldung.dependson") public class Config {

```
@Bean
@DependsOn({"fileReader","fileWriter"})
public FileProcessor fileProcessor(){
    return new FileProcessor();
}

@Bean("fileReader")
public FileReader fileReader() {
    return new FileReader();
}

@Bean("fileWriter")
public FileWriter fileWriter() {
    return new FileWriter();
}
```

}

```
Using `@DependsOn` at the class level has no effect unless component-scanning is being used. If a DependsOn-annotated cl

[Más información](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotati

### @Lazy

TODO

### @Lookup

TODO

### @Primary

TODO

### @Scope

TODO

### @Profile

TODO

### @Import

TODO

### @ImportResource

TODO

### @PropertySource

TODO

### @PropertySources

TODO

---

### @Configuration (Class Level Annotation)

### @ComponentScan (Class Level Annotation)

### @PostConstruct & @PreDestroy (Method Level Annotation)

Spring calls the methods annotated with `@PostConstruct` only once, just after the initialization of bean properties. Ke

The method annotated with `@PostConstruct` can have any access level, but it can't be static. Annotated methods can have
```

```
One possible use of @PostConstruct is populating a database:
```

java @Component public class DbInit {

```
@Autowired
private UserRepository userRepository;

@PostConstruct
private void postConstruct() {
    User admin = new User("admin", "admin password");
    User normalUser = new User("user", "user password");
    userRepository.save(admin, normalUser);
}
```

}

```
A method annotated with `@PreDestroy` runs only once, just before Spring removes our bean from the application context.

Same as with @PostConstruct, the methods annotated with @PreDestroy can have any access level, but can’t be static.
```

java @Component public class UserRepository {

```
private DbConnection dbConnection;
@PreDestroy
public void preDestroy() {
    dbConnection.close();
}
```

}

```
NOTE: PreDestroy methods called if application shuts down normally. Not if the process dies or is killed.
```

java ConfigurableApplicationContext context = SpringApplication.run(…);

// Trigger call of all @PreDestroy annotated methods context.close();

```
Alternatively, `@Bean` has options to define these life-cycle methods:
```

java @Bean(initMethod="populateCache", destroyMethod="flushCache") public AccountRepository accountRepository () { //… }

```
So, which scheme to use?

- Use `@PostConstruct` and/or `@PreDestroy` for your own classes
- Use Lifecycle Method attributes of `@Bean` annotation for classes you didn't write and can't annotate, like third-part

Note that both the `@PostConstruct` and `@PreDestroy` annotations are part of Java EE. Since Java EE was deprecated in
```

java javax.annotation javax.annotation-api 1.3.2

```
- [Más información](https://docs.spring.io/spring-framework/reference/core/beans/annotation-config/postconstruct-and-pre
- [Más información](https://www.baeldung.com/spring-postconstruct-predestroy)

### @Required (deprecated)

The `@Required` annotation is method-level annotation. It applies to the **bean setter method**. It indicates that the a
```

java public class Machine {
private Integer cost;

@Required public void setCost(Integer cost) { this.cost = cost;
}

public Integer getCost() { return cost; } }

```
## Spring Framework Stereotype Annotations

### @Component (Class Level Annotation)

It is a class-level annotation. It is used to mark a Java class as a bean. A Java class annotated with `@Component` is f

`@Component` is a **generic** stereotype for any Spring-managed component. `@Repository`, `@Service`, and `@Controller`

![Diagram](https://www.techferry.com/articles/images/SpringComponentAnnotations.jpg)
```

java @Component public class ContactResource { //... }

```
### @Controller (Class Level Annotation)

The `@Controller` annotation is used to indicate the class is a Spring controller. This annotation is simply a speciali
```

java @Controller @RequestMapping("/api/brands") public class BrandsController{ @GetMapping("/getall") public Employee getAll(){ return brandService.getAll(); } }

```
### @Service (Class Level Annotation)

`@Service` marks a Java class that performs some service, such as executing business logic, performing calculations, and
```

java @Service public class TestService{ public void service1(){ //business code } }

```
### @Repository (Class Level Annotation)

This annotation is used on Java classes that directly access the database. The `@Repository` annotation works as a marke

This annotation has an automatic translation feature. For example, when an exception occurs in the `@Repository`, there

This annotation is a specialized form of the `@Component` annotation.
```

java @Repository public class TestRepo{ public void add(){ System.out.println("Added"); } }

```
## Spring Boot Annotations

### @EnableAutoConfiguration (Class Level Annotation)
### @SpringBootApplication (Class Level Annotation)

@SpringBootApplication is a convenience annotation that adds all of the following:

- @Configuration: Tags the class as a source of bean definitions for the application context.

- @EnableAutoConfiguration: Tells Spring Boot to start adding beans based on classpath settings, other beans, and variou

- @ComponentScan: Tells Spring to look for other components, configurations, and services in the com/example package, le

## Spring MVC and REST Annotations

### @RequestMapping (Method Level Annotation)

- Method Level Annotation
- It is used to map the HTTP request with specific method.
```

java @Controller @RequestMapping("/api/brands") public class BrandsController{ @GetMapping("/getall") public Employee getAll(){ return brandService.getAll(); } }

```
### @GetMapping (Method Level Annotation)

- Method Level Annotation
- It is used to map the HTTP GET request with specific method.
- It is used to get the data.
- It is used to read the data.
```

java @GetMapping("/getall") public Employee getAll(){ return brandService.getAll(); }

```
### @PostMapping (Method Level Annotation)

- Method Level Annotation
- It is used to map the HTTP POST request with specific method.
- It is used to add the data.
- It is used to create the data.
```

java @PostMapping("/add") public void add(@RequestBody Brand brand){ brandService.add(brand); }

```
### @PutMapping (Method Level Annotation)

- Method Level Annotation
- It is used to map the HTTP PUT request with specific method.
- It is used to update the data.
```

java @PutMapping("/update") public void update(@RequestBody Brand brand){ brandService.update(brand); }

```
### @DeleteMapping (Method Level Annotation)

- Method Level Annotation
- It is used to map the HTTP DELETE request with specific method.
- It is used to delete the data.
```

java @DeleteMapping("/delete") public void delete(@RequestBody Brand brand){ brandService.delete(brand); }

```
### @PatchMapping

### @RequestBody

- It is used to get the data from the request body.
- It is used to get the data from the HTTP request.
- It is used to get the data from the HTTP request body.
```

java @PostMapping("/add") public void add(@RequestBody Brand brand){ brandService.add(brand); }

```
### @ResponseBody

### @PathVariable (Method Level Annotation)

- Method Level Annotation
- It is used to get the data from the URL.
- It is the most suitable for RESTful web service that contains a path variable.
```

java @GetMapping("/getbyid/{id}") public Brand getById(@PathVariable int id){ return brandService.getById(id); }

```
### @RequestParam

- It is used to get the data from the URL.
- It is used to get the data from the URL query parameters.
- It is also known as query parameter.
```

java @GetMapping("/getbyid") public Brand getById(@RequestParam int id){ return brandService.getById(id); }

```
### @RequestHeader

### @RestController (Class Level Annotation)

- Class Level Annotation
- It is a marker interface.
- It is a controller layer.
- It is used to create a controller layer.
- It use with @RequestMapping annotation.
- It is a combination of @Controller and @ResponseBody annotations.
- @RestController annotation is explained with @ResponseBody annotation.
- @ResponseBody eliminates the need to add a comment to every method.
```

java @RestController @RequestMapping("/api/brands") public class BrandsController{ @GetMapping("/getall") public Employee getAll(){ return brandService.getAll(); } } ```

@RequestAttribute

TODO

Enlaces de interés

- 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

Licencia