# Spring Framework

**In the Spring Framework**, an annotation is a form of metadata that provides information about the program but is not part of the program itself. Annotations are used to simplify configuration and automate the handling of various tasks by marking classes, methods, fields, or other elements with specific metadata.

**Key Points:**

- **Simplifies Configuration:** Annotations reduce the need for extensive XML configurations.
- **Declarative Style:** You can declare the behaviors and dependencies of your beans directly in the code.
- **Metadata Tags:** Annotations act as tags that provide additional information to the Spring container.

Sure! Here is a comprehensive list of commonly used annotations in the Spring Framework along with their descriptions and simple usage examples to help you understand how and when to use them:

### Core Spring Annotations

**1. @Configuration**

**Description:** Defines a class as a source of bean definitions.

**Meaning**:Tells Spring, "This class contains methods to create and configure beans."

**Usage:**

```
@Configuration
public class AppConfig {
    @Bean
    public MyService myService() {
        return new MyServiceImpl();
    }
}
```

2. **@Bean**

**Description:** Indicates a method produces a bean to be managed by Spring.

**Meaning:** When you put the `@Bean` annotation on a method, it tells Spring that this method will return an object that should be registered as a bean in the Spring application context.

**Usage:**

```
 @Bean
 public MyService myService() {
    return new MyServiceImpl();
 }
```

3. **@Component**

**- Description:** Marks a class as a Spring component for auto-detection.

**Meaning:** Tells Spring, "This class should be managed by you."

**Usage:**

```
@Component
public class MyComponent {
   // Spring will manage this class and create an instance of it automatically
}
```

**4. @Service**

**Description:** Marks a class as a service layer component.

**Meaning:** By marking a class with `@Service`, you are indicating that this class performs some service, such as business logic, calculations, or other processes.

**Usage:**

```
    @Service
 public class MyService {
    // Class implementation
 }
```

5. **@Repository**

**-Description**: Indicates that a class is a repository.

**Meaning:** By annotating a class with `@Repository`, you are indicating that this class is a repository responsible for storing, retrieving, and managing data access operations.

**- Usage:**

```
 @Repository
 public class MyRepository {
    // Class implementation
 }
```

**6.@Controller**

- **Description**:Marks a class as a web controller.

**Meaning:**

- Web Request Handling: By annotating a class with @Controller, you are indicating that this class serves as a controller in a Spring MVC (Model-View-Controller) application.
- Handles HTTP Requests: Controllers handle incoming HTTP requests, execute appropriate business logic, and return a view or data to the client.

**- Usage:**

```
@Controller
public class MyController {
   // Class implementation
}
```

## Dependency Injection Annotations

Dependency Injection (DI) annotations in the Spring Framework are used to facilitate the automatic injection of dependencies into Spring-managed beans. These annotations help manage dependencies and promote loose coupling between components, enhancing the flexibility and maintainability of your application.

**1. @Autowired**

**- Description:** Marks a dependency to be injected by Spring.

**Meaning:** The `@Autowired` annotation in the Spring Framework is used to automatically inject dependencies into a Spring-managed bean.

- By annotating a field, constructor, or method with `@Autowired`, you are telling Spring to automatically inject the appropriate bean into that point when it creates the bean.
- **Reduces Boilerplate Code:** It reduces the boilerplate code required for manual dependency injection and allows for more concise and readable code.
-

**- Usage:**

```
@Autowired
private MyService myService;
```

**2. @Qualifier**

**- Description:** Specifies which bean to inject when multiple options are available.

**- Usage:**

```
@Autowired

@Qualifier("specificBean")

private MyService myService;
```

**3. @Value**

- **Description:** Injects values from properties files or environment variables.

**Meaning:** By annotating a field, method, or constructor parameter with `@Value`, you can inject values directly into your Spring beans from configuration properties files (`application.properties` or `application.yml`), environment variables, or other Spring beans.

**- Usage:**

```
@Value("${property.name}")

private String propertyName;
```

**@Inject:**

- **Description:** Provides dependency injection capability similar to @Autowired, but it is part of the Java Dependency Injection (JSR-330) standard.
- **Usage:** Can be applied to fields, constructors, or methods.

**Usage:**

```
@Inject

 private MyService myService;
```

## Aspect-Oriented Programming (AOP) Annotations

**Traditional Programming Approach:**

- In traditional programming, you write your application logic (like business logic) directly within your classes. This logic often includes not just the core functionality, but also other concerns like logging, security checks, transaction management, etc.

**Challenge with Traditional Approach:**

- As your application grows, these cross-cutting concerns (logging, security, etc.) tend to be repeated across multiple classes and methods. This can lead to code duplication and makes the codebase harder to maintain.

**What AOP Offers:**

- AOP provides a solution to this problem by separating these cross-cutting concerns from your application logic. Instead of scattering them throughout your codebase, you define them in a centralized way.

**How AOP Works:**

- **Aspect:** In AOP, you define aspects, which encapsulate cross-cutting concerns. An aspect is a module that encapsulates behaviors affecting multiple classes.

**1. @Aspect**

- **Description:** Declares a class as an aspect.
- **Usage:**

```
@Aspect

public class MyAspect {

   // Aspect implementation

}
```

**2. @Before**

**- Description**: Executes before a matched method execution.

**Meaning**: `@Before` is an advice type in Spring AOP that specifies an action to be taken before a method execution.

**- Usage:**

```
@Before("execution(* com.example.MyService.*(..))")
public void logBefore(JoinPoint joinPoint) {
   // Log method execution
}
```

**3. @After**

**- Description:** Executes after a matched method execution.

**- Usage:**

```
@After("execution(* com.example.MyService.*(..))")
public void logAfter(JoinPoint joinPoint) {
   // Log method execution
}
```

**4. @Around**

**- Description:** Surrounds a matched method execution.

**Meaning:** The `@Around` annotation in Spring AOP is used to define advice that surrounds a method execution. It allows you to perform custom behavior before and after the method invocation, including modifying the method's return value or handling exceptions.

**- Usage:**

```
@Around("execution(* com.example.MyService.*(..))")
public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {
        // Log before and after method execution

        Object result = joinPoint.proceed();

        return result;

    }
```

**5. @AfterReturning**

**- Description:** Executes after a matched method execution if it returns normally.

**Meaning:** The `@AfterReturning` annotation in Spring AOP is used to define advice that executes after a matched method successfully returns a result, without throwing an exception.

**- Usage:**

```
    @AfterReturning(pointcut = "execution(* com.example.MyService.*(..))", returning = "result")

    public void logAfterReturning(JoinPoint joinPoint, Object result) {

        // Log method result

    }
```

**6. @AfterThrowing**

**- Description:** Executes if a matched method execution throws an exception.

**- Usage:**

```
    @AfterThrowing(pointcut = "execution(* com.example.MyService.*(..))", throwing = "error")

    public void logAfterThrowing(JoinPoint joinPoint, Throwable error) {

        // Log exception

    }
```

## Spring MVC Annotations

**1. @RequestMapping**

-**Description:** Maps HTTP requests to handler methods of MVC and REST controllers.

**- Usage:**

```
@Controller
@RequestMapping("/home")
public class HomeController {
   @RequestMapping
   public String home() {
      return "home";
   }
}
```

**2. @GetMapping**

**- Description:** Maps HTTP GET requests to handler methods.

**- Usage:**

```
@GetMapping("/users")
public List<User> getUsers() {
   return userService.getUsers();
}
```

**3. @PostMapping**

- **Description:** Maps HTTP POST requests to handler methods.

Meaning: The @PostMapping annotation in Spring Boot is used to map HTTP POST requests to specific handler methods in your controller classes.

- @PostMapping maps HTTP POST requests to specific handler methods.
- It is used to handle data submission from forms, JSON payloads, or other data sent in the body of an HTTP POST request.

**- Usage:**

```
@PostMapping("/users")

public void createUser(@RequestBody User user) {

   userService.createUser(user);

}
```

**4. @PutMapping**

**- Description:** Maps HTTP PUT requests to handler methods.

**- Usage:**

```
@PutMapping("/users/{id}")

public void updateUser(@PathVariable Long id, @RequestBody User user) {

   userService.updateUser(id, user);

}
```

**5. @DeleteMapping**

**- Description:** Maps HTTP DELETE requests to handler methods.

**- Usage:**

```
@DeleteMapping("/users/{id}")
 public void deleteUser(@PathVariable Long id) {
     userService.deleteUser(id);
 }
```

6. **@PatchMapping**

**- Description:** Maps HTTP PATCH requests to handler methods.

**- Usage:**

```
@PatchMapping("/users/{id}")
public void patchUser(@PathVariable Long id, @RequestBody Map<String, Object> updates) {
    userService.patchUser(id, updates);
}
```

**7. @PathVariable**

- Description: Binds a method parameter to a URI template variable.

- Usage:

```
@GetMapping("/users/{id}")
public User getUserById(@PathVariable Long id) {
    return userService.getUserById(id);
}
```

**8. @RequestParam**

**- Description:** Binds a method parameter to a web request parameter.

**- Usage:**

```
@GetMapping("/search")
public List<User> searchUsers(@RequestParam String name) {
    return userService.searchUsers(name);
}
```

**9. @RequestBody**

**- Description:** Binds the body of the web request to a method parameter.

**Meaning:** The @RequestBody annotation in Spring Boot is used to bind the body of an HTTP request to a method parameter. This is particularly useful for handling data sent in the body of a request, such as JSON or XML data, and mapping it directly to a Java object.

**- Usage:**

```
@PostMapping("/users")
public void createUser(@RequestBody User user) {
    userService.createUser(user);
}
```

**10. @ResponseBody**

**- Description:** Binds the return value of a method to the web response body.

**- Usage:**

```
@GetMapping("/users")
@ResponseBody
public List<User> getUsers() {
    return userService.getUsers();
}
```

**1. @RestController**

Description: A convenience annotation that is itself annotated with `@Controller` and `@ResponseBody`.

**Meaning:** The @RestController annotation is a convenience annotation in Spring Boot that combines the functionalities of @Controller and @ResponseBody. This annotation is used to create RESTful web services.

- **Usage:**

```
@RestController
@RequestMapping("/api")
public class ApiController {
   @GetMapping("/users")
   public List<User> getUsers() {
      return userService.getUsers();
   }
}
```

**2. @RequestScope**

- Description: Scopes a single bean definition to the lifecycle of a single HTTP request.

**- Usage:**

```
@Component
@RequestScope
public class RequestScopedBean {
   // Bean implementation
}
```

**3. @SessionScope**

- **Description:** Scopes a single bean definition to the lifecycle of an HTTP session.

**- Usage:**

```
@Component
@SessionScope
public class SessionScopedBean {
   // Bean implementation
}
```

**4. @ComponentScan**

**- Description:** Configures component scanning directives for use with `@Configuration` classes.

**- Usage:**

```
@Configuration
@ComponentScan(basePackages = "com.example")
public class AppConfig {
   // Configuration implementation
}
```

**5. @EnableAutoConfiguration**

- **Description:** Enables Spring Boot’s auto-configuration mechanism.

**- Usage:**

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

**6. @SpringBootApplication**

**- Description**:A convenience annotation that combines `@Configuration`, `@EnableAutoConfiguration`, and `@ComponentScan`.

- **Usage:**

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

## Spring Boot Annotations:

Spring Boot annotations are special markers or tags in the Spring Boot framework that provide metadata about the program. These annotations are used to simplify the configuration and development of Spring-based applications. They help the Spring framework understand how to configure, manage, and integrate different parts of your application automatically.

1. **@SpringBootApplication**
   - **Description:** A convenience annotation that combines @Configuration, @EnableAutoConfiguration, and @ComponentScan with their default attributes.

2. **@EnableAutoConfiguration**
   - **Description:** Enables Spring Boot's auto-configuration mechanism, automatically configuring Spring application based on the dependencies present on the classpath.

3. **@Configuration**
   - **Description:** Indicates that a class declares one or more @Bean methods and can be processed by the Spring container to generate bean definitions.

4. **@ComponentScan**
   - **Description:** Configures component scanning directives for use with @Configuration classes.

## Stereotype Annotations

5. **@Component**
   - **Description:** Indicates that an annotated class is a "component". Such classes are considered as candidates for auto-detection when using annotation-based configuration and classpath scanning.

6. **@Service**
   - **Description:** Indicates that an annotated class is a "Service". It is a specialization of @Component and is used in the service layer.

7. **@Repository**
   - **Description:** Indicates that an annotated class is a "Repository", originally defined by Domain-Driven Design (DDD) as "a mechanism for encapsulating storage, retrieval, and search behavior which emulates a collection of objects".

8. **@Controller**
   - **Description:** Indicates that an annotated class is a "Controller" (e.g. a web controller).

9. **@RestController**
   - **Description:** A convenience annotation that is itself annotated with @Controller and @ResponseBody. It is used to create RESTful web services.

## Dependency Injection Annotations

10. **@Autowired**
    - **Description:** Marks a constructor, field, setter method, or config method as to be autowired by Spring's dependency injection facilities.

11. **@Qualifier**
    - **Description:** When there are multiple beans of the same type, use this annotation to specify which one should be autowired.

12. **@Value**
    - **Description:** Indicates a default value expression for the annotated element.

## Configuration Annotations

13. **@Bean**
    - **Description:** Indicates that a method produces a bean to be managed by Spring.

14. **@PropertySource**
    - **Description:** Provides a convenient and declarative mechanism for adding a PropertySource to Spring’s Environment.

## Spring MVC Annotations

15. **@RequestMapping**
    - **Description:** Provides routing information and is used to map web requests to specific handler classes or handler methods.

16. **@GetMapping**
    - **Description:** A shortcut for @RequestMapping(method = RequestMethod.GET).

17. **@PostMapping**
    - **Description:** A shortcut for @RequestMapping(method = RequestMethod.POST).

18. **@PutMapping**
    - **Description:** A shortcut for @RequestMapping(method = RequestMethod.PUT).

19. **@DeleteMapping**
    - **Description:** A shortcut for @RequestMapping(method = RequestMethod.DELETE).

20. **@PatchMapping**
    - **Description:** A shortcut for @RequestMapping(method = RequestMethod.PATCH).

21. **@PathVariable**
    - **Description:** Indicates that a method parameter should be bound to a URI template variable.

22. **@RequestParam**
    - **Description:** Binds a web request parameter to a method parameter.

23. **@RequestBody**
    - **Description:** Indicates a method parameter should be bound to the body of the web request.

24. **@ResponseBody**
    - **Description:** Indicates that the return value of a method should be bound to the web response body.

25. **@CrossOrigin**
    - **Description:** Enables cross-origin resource sharing (CORS) on a controller or specific handler methods.