sSpring Boot Annotations

1. @SpringBootApplication

📘 Definition:

A convenience annotation that combines:

- @Configuration: Declares the class as a configuration source.
- @EnableAutoConfiguration: Enables Spring Boot's auto-configuration mechanism.
- @ComponentScan: Scans the package for Spring components.

✅ Usage:

Place it on the main application class to enable all auto configurations and component scanning.

💡 Example:

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

🔧 How it works:

- When the application runs, Spring Boot scans the classpath.
- It auto-configures beans based on dependencies.
- It initializes the Spring context and starts the embedded server (e.g., Tomcat).
- This annotation serves as the starting point for the application's context.

---

2. @Configuration

📘 Definition:

Declares that the class contains one or more @Bean methods. The Spring IoC container processes this class to generate Spring Beans.

✅ Usage:

Used in Java-based configuration to define beans programmatically.

💡 **Example:**

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

@Configuration
public class AppConfig {

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

🔧 **How it works:**

- Spring scans the AppConfig class because it's annotated with @Configuration.
- It identifies the @Bean method myService().
- It executes this method and registers its return object (MyService) as a bean.
- Anywhere in the application, Spring can inject MyService using @Autowired.
- This allows fine-grained, programmatic control over bean creation.

---

3. @Bean

📘 **Definition:**

Declares a single bean that Spring should manage. Typically used within a @Configuration class.

✅ **Usage:**

When you want to manually define a bean.

💡 **Example:**

```
@Bean
public DataSource dataSource() {
   return new HikariDataSource();
}
```

🔧 **How it works:**

- The method is executed by Spring during startup.
- The return value is stored in the ApplicationContext as a bean.
- It's ideal for integrating third-party libraries or legacy components into Spring.

---

4. @ComponentScan

📘 **Definition:**

Directs Spring to scan specified packages for components (classes with @Component, @Service, etc.).

✅ **Usage:**

Used when your configuration class is outside the component classes.

💡 **Example:**

```
@ComponentScan(basePackages = "com.example")
```

🔧 **How it works:**

- Spring starts scanning the specified package.
- It picks up all classes annotated with @Component, @Service, @Repository, and @Controller.
- Registers them as beans in the container.
- Enables automatic detection and registration of beans.

5.Component, @Service, @Repository, @Controller

📘 Definition:

The @Component annotation is a generic stereotype used to define a class as a Spring-managed component (i.e., a bean). **It tells Spring that this class should be considered for dependency injection.**

✅ Usage:

Use @Component when creating a class that should be automatically detected and instantiated by Spring's component scanning mechanism.

💡 Example:

```
@Component

public class EmailSender {

   public void send(String message) {

      System.out.println("Sending: " + message);

   }

}
```

🔧 How it works:

- Spring detects classes annotated with @Component during the classpath scanning phase.
- It creates and registers an instance of the class in the ApplicationContext.
- The component becomes eligible for @Autowired injection anywhere it's needed.

🔎 Types of @Component

@Component is the base stereotype, and Spring provides specializations for better semantic clarity:

@Service: Used for service layer components that contain business logic.
@Service

- public class UserService { }

@Repository: Used for DAO (Data Access Object) components interacting with the database. Also enables automatic exception translation.
@Repository

- public interface UserRepository extends JpaRepository<User, Long> {}

@Controller: Used in Spring MVC to define controller classes handling HTTP requests.
@Controller

```
public class UserController {

   @GetMapping("/hello")

   public String hello() {

      return "hello";

   }
```

- }

🧠 Additional Notes:

- These annotations improve readability and maintainability by clarifying the role of each component in the application.
- All these annotations are picked up via @ComponentScan.
- You can specify bean names using @Component("beanName").

---

6. @Autowired

📘 **Definition:**

Inject a bean automatically into the required location.

✅ **Usage:**

On fields, constructors, or setters.

💡 **Example:**

```
@Autowired
private UserService userService;
```

🔧 **How it works:**

- Spring looks for a matching bean type.
- It injects the dependency into the field/constructor.
- If more than one matching type exists, it may throw an error unless resolved using @Qualifier.

---

7. @Qualifier

📘 **Definition:**

Helps to specify which bean to inject when multiple beans of the same type are available.

💡 **Example:**

```
@Autowired
@Qualifier("mysqlRepo")
private UserRepository userRepository;
```

🔧 **How it works:**

- Resolves ambiguity by name-matching the bean.
- Must match the bean name explicitly.

---

8. @Value

📘 **Definition:**

Injects values from application properties or expressions.

💡 **Example:**

```
@Value("${app.name}")
private String appName;
```

🔧 **How it works:**

- Parses the property and assigns it to the field.
- Useful for configuration values such as URLs, secrets, and flags.

---

9. @RestController

📘 **Definition:**

Combines @Controller and @ResponseBody.

✅ **Usage:**

Used in REST APIs to return JSON/XML directly.

💡 **Example:**

```
@RestController
public class HelloController {
    @GetMapping("/hello")
    public String hello() {
        return "Hello World";
    }
}
```

🔧 **How it works:**

- Spring calls the method.
- The return value is written directly to the response body.
- No need for view resolution.

---

10. @RequestMapping, @GetMapping, @PostMapping, etc.

📘 **Definition:**

Maps HTTP requests to handler methods.

💡 **Example:**

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

🔧 **How it works:**

- Spring MVC routes incoming requests to these methods based on path and method type.
- Supports parameter binding and response handling.

---

11. @RequestBody, @ResponseBody, @PathVariable, @RequestParam

📘 **Purpose:**

- @RequestBody: Binds request JSON to method parameter.
- @ResponseBody: Returns method result directly as HTTP response.
- @PathVariable: Binds URL template variable.
- @RequestParam: Binds query string parameter.

💡 **Example:**

```
@GetMapping("/users/{id}")
public User getUser(@PathVariable int id) { ... }
```

---

12. @Entity, @Table, @Id, @GeneratedValue

📘 **Definition:**

Used for ORM with JPA to map Java classes to database tables.

💡 **Example:**

```
@Entity
@Table(name = "users")
public class User {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private String name;
}
```

---

13. @SpringBootTest

📘 **Definition:**

Used for integration testing. Loads the complete application context.

💡 **Example:**

```
@SpringBootTest
public class MyAppTests { ... }
```

🔧 **How it works:**

- Boots up the application context.
- Makes it possible to test real service layers and beans.

- Useful for end-to-end testing with full configurations.

---

14. @MockBean

📘 **Definition:**

Creates a Mockito mock and adds it to the Spring context.

💡 **Example:**

```
@MockBean
private UserService userService;
```

🔧 **How it works:**

- Replaces the actual bean with a mock object in the test context.
- Helpful for isolating units during integration tests.

---

15. @DataJpaTest

📘 **Definition:**

Configures JPA repositories and tests only the persistence layer.

✅ **Usage:**

Use this to test repositories in isolation with H2 or an embedded DB.

🔧 **How it works:**

- Loads only repository-related components.
- Disables full application context startup.
- Uses in-memory DB for faster and isolated tests.

---

16. @EnableAutoConfiguration

📘 **Definition:**

Enables Spring Boot's auto-configuration.

🔧 **How it works:**

- Looks at the classpath and application properties.
- Configures beans automatically if matching classes are found.
- It’s the core magic behind Spring Boot’s opinionated defaults.

---

17. @EnableConfigurationProperties

📘 **Definition:**

Enables support for @ConfigurationProperties beans.

💡 **Example:**

```
@EnableConfigurationProperties(AppProperties.class)
```

🔧 **How it works:**

- Scans the class and binds configuration values.
- Needed for using @ConfigurationProperties outside main app class.

---

18. @ConfigurationProperties

📘 **Definition:**

Maps properties from application.properties or application.yml to a Java object.

💡 **Example:**

```
@ConfigurationProperties(prefix = "app")
public class AppProperties {
   private String name;
   private String version;
   // getters and setters
}
```

🔧 **How it works:**

- Binds hierarchical properties from config files.
- Supports type-safe configuration for complex property groups.