**🔧 Configuration & Bean Definition**

* @Configuration: Marks a class as a source of bean definitions.
* @Bean: Declares a bean method in a @Configuration class.
* @Import: Allows importing additional configuration classes.
* @ImportResource: Imports XML-based configuration.
* @PropertySource: Loads properties from a file into the Spring Environment.

**💉 Dependency Injection**

* @Autowired: Automatically injects a bean by type.
* @Qualifier: Specifies the exact bean to inject when multiple beans of the same type exist.
* @Value: Injects values from properties files.
* @Required: (Deprecated) Ensures a property is injected (older style).
* @DependsOn: Defines the dependency between beans.

## 🧾 Stereotype Annotations in Spring

Spring provides four stereotype annotations used to define Spring-managed components:

| **Annotation** | **Meaning & Role** |
| --- | --- |
| @Component | Generic annotation to mark a class as a Spring-managed bean (IoC container will detect and manage it). |
| @Service | Specialized version of @Component — used to mark **service layer classes**. Also provides **transaction support** (with Spring AOP). |
| @Repository | Specialized version of @Component — used for **DAO/data access layer**. Enables **exception translation** (e.g., JDBC SQLException to Spring's DataAccessException). |
| @Controller | Specialized version of @Component — used for **web controllers** in Spring MVC. Allows handling of **HTTP requests** (used with @RequestMapping, etc.). |

### Service , repository,controller classes pending

### 🛠️ To Enable Component Scanning (in XML):

You need this in your applicationContext.xml:

<context:component-scan base-package="your.base.package" />

This allows Spring to **automatically detect** classes annotated with @Component, @Service, @Repository, or @Controller.

Spring's **stereotype annotations** are:

* @Component
* @Service
* @Repository
* @Controller

These are used to indicate **what role a class plays** in your application, and they must be applied to the **class declaration**.

**🌀 Bean Lifecycle & Scope**

* @Scope: Defines the scope (singleton, prototype, etc.) of a bean.
* @Lazy: Delays bean creation until it is needed.
* @Primary: Specifies a default bean when multiple candidates exist.
* @Lookup: Used to inject a new bean instance (prototype) into a singleton.

**⚙️ Order and Execution**

* @Order: Defines the order of execution among components.

 **XML Driven**

* Uses applicationContext.xml files.
* Beans are defined and wired using XML tags.
* Example: <bean id="myBean" class="com.example.MyBean"/>

 **Annotation Driven Configuration**

* Uses annotations like @Component, @Autowired, @Configuration, and @Bean.
* Often combined with context:component-scan in XML or Java config to enable scanning.

 **100% Code Driven Configurations**

* Pure Java-based configuration using @Configuration and @Bean.
* No XML; all setup is in Java classes.
* Enables type safety and IDE support.

 **Spring Boot Driven Configuration**

* Uses @SpringBootApplication and auto-configuration.
* Minimizes boilerplate; relies heavily on **convention over configuration**.
* Embeds servers like Tomcat and simplifies setup with application.properties or application.yml.

 **XML Driven**

* Uses applicationContext.xml files.
* Beans are defined and wired using XML tags.
* Example: <bean id="myBean" class="com.example.MyBean"/>

 **Annotation Driven Configuration**

* Uses annotations like @Component, @Autowired, @Configuration, and @Bean.
* Often combined with context:component-scan in XML or Java config to enable scanning.

 **100% Code Driven Configurations**

* Pure Java-based configuration using @Configuration and @Bean.
* No XML; all setup is in Java classes.
* Enables type safety and IDE support.

 **Spring Boot Driven Configuration**

* Uses @SpringBootApplication and auto-configuration.
* Minimizes boilerplate; relies heavily on **convention over configuration**.
* Embeds servers like Tomcat and simplifies setup with application.properties or application.yml.

----------------------------------------------------------------------------------------------------------------------------

### @Autowired in Spring:

* **Autowiring Modes**:  
  @Autowired performs dependency injection by:
  + **byType** (default): Injects a bean by matching type.
  + **byName**: If ambiguity occurs, the bean name is used.
  + **constructor**: If applied to a constructor, Spring uses constructor injection.
* **Applicable To**:
  + **Field level** (instance variables)
  + **Constructor**
  + **Setter methods**
* **Limitation**:
  + @Autowired **cannot inject primitive or String values directly**.
  + It is used **only for object-type dependencies** (i.e., Spring-managed beans).

Eg: AutowireAnnotationComponentScanFieldLevelByType

### ✅ Why Setter Was Not Included:

In the example I gave, I used **field-based injection**:

@Autowired

@Qualifier("dtdc")

private Courier courier;

In this style, Spring **injects the dependency directively into the field**, so **a setter is not required**.

| **Injection Type** | **Uses byType?** | **Uses byName?** | **Uses constructor?** | **Notes** |
| --- | --- | --- | --- | --- |
| **Field Injection** | ✅ Yes | ❌ No | ❌ No | Spring injects by **type** only. If multiple beans exist, use @Qualifier. |
| **Setter Injection** | ✅ Yes | ✅ Yes\* | ❌ No | Defaults to **byType**. **byName** is possible if method name matches bean name. |
| **Constructor Injection** | ✅ Yes | ❌ No | ✅ Yes | Pure **constructor**-based injection. Spring uses **byType** matching. |

### 🔧 Fix External Resource Download Error in Eclipse (Maven Projects)

#### ✅ Step 1: Force Maven to Download All Dependencies & Resources

1. **Right-click your project** in Project Explorer
2. Go to **Maven → Update Project**
3. Check ✅ **Force Update of Snapshots/Releases**
4. Click **OK**

This tells Maven to **re-fetch all dependencies**, including any missing .xsd files used by Spring’s XML configuration.

#### ✅ Step 2: Enable Auto-Download in Eclipse Maven Settings

1. Go to **Window → Preferences → Maven**
2. Under Maven preferences, ensure all these are ✅ checked:
   * Download Artifact Sources
   * Download Artifact JavaDoc
   * Download repository index updates on startup
   * (Optional): Enable Update Maven projects on startup
3. Click **Apply and Close**

### 🔁 Restart Eclipse

After completing the above steps, **restart Eclipse** to ensure all background indexing and downloads are refreshed.

Eg: AutowireAnnotationComponentScanSetterInjectionByType

Eg: AutowireAnnotationComponentScanSetterInjectionByName

Eg: AutowireAnnotationComponentScanConstructorInjectionByType

Eg: AutowireLazyValueAnnotation

## 🔍 What is @Configuration?

@Configuration

public class AppConfig {

@Bean

public Random random() {

return new Random();

}

}

can be written in legacy Spring XML as:

✅ Equivalent XML Configuration (applicationContext.xml)

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

<!-- Declare Random bean manually -->

<bean id="random" class="java.util.Random" />

</beans>

🔍 How It Works:

This creates and registers a Random object in the Spring container.

Now you can use @Autowired in your Java class:

@Autowired

private Random random;

Spring will inject the Random instance defined in XML.

* @Configuration is a **Spring annotation** introduced in **Spring 3.0**
* It tells Spring:  
  📢 “This class is a **source of bean definitions** — just like an XML file!”

## 🧠 What Happens When You Use @Configuration?

When Spring sees a class annotated with @Configuration, it does the following:

1. It **creates an object of this class**
2. It **calls all methods annotated with @Bean**
3. It **registers** the returned objects as **Spring-managed beans**

So, in essence, @Configuration is like the <beans> tag in XML.

## ✅ Example: Using @Configuration

@Configuration

public class AppConfig {

@Bean

public Random random() {

return new Random(); // This object becomes a Spring bean

}

}

Now the object returned by random() is available for @Autowired.

## 🧾 Behind the Scenes: What Spring Does Internally

When Spring loads the context:

1. It finds the class with @Configuration
2. It processes each @Bean method
3. Each method is run **only once**, and Spring **caches the return object**
4. Even if another @Bean calls random(), it gets the **same instance**

✅ So, it's not like ordinary Java — Spring **proxies** the @Configuration class to ensure beans are singleton (unless specified otherwise).

## 🔍 1. @ComponentScan(basePackages = "in.orcas.bean")

### ✅ What It Does:

* Tells Spring to **scan the specified package** (and its subpackages) for classes annotated with:
  + @Component
  + @Service
  + @Repository
  + @Controller

### 🔧 Why You Need It:

Without @ComponentScan, your Spring container **won’t detect** these annotated classes (like Flipkart, DTDC, BlueDart, etc.), and they **won’t be registered as beans**.

### 🔬 Internal Behavior:

* Spring loads AppConfig
* It reads @ComponentScan(basePackages = "in.orcas.bean")
* It scans:

/in/orcas/bean/

├── Flipkart.java (@Component)

├── DTDC.java (@Component)

├── BlueDart.java (@Component)

└── etc.

* It registers all these classes in the **Spring BeanFactory**

## ✅ 1. What does classpath: mean?

* classpath: tells Spring to **look for the file inside the application's classpath**.
* The **classpath** is essentially:
  + All directories and JARs that are available to your application during runtime
  + In a Maven/Gradle project, src/main/resources/ is automatically included in the classpath

**What is @Service in Spring?**

@Service is a **specialized component annotation** in Spring used to mark classes that contain **business logic**. It is part of the **service layer** in a typical multi-layered architecture.

**🔹 Why is @Service important?**

Because:

1. It **separates** your business logic from DAO and controller logic.
2. When combined with @Transactional, Spring **automatically manages**:
   * Opening a transaction
   * Committing it on success
   * Rolling it back on exception

**🔹 Where do we normally write transaction code manually (in Hibernate)?**

In pure Hibernate:

Transaction tx = session.beginTransaction();

userDAO.save(user);

tx.commit();

This is **manual and error-prone**.

**🔹 What Spring does with @Service + @Transactional:**

You write only **core logic** like:

userDAO.save(user);

Spring **internally handles** the transaction boundaries using AOP (Aspect Oriented Programming). You don’t write beginTransaction(), commit(), or rollback().

**🔹 Real-time Example**

@Service

@Transactional

public class ProductService {

@Autowired

private ProductDAO productDAO;

public void addProduct(Product p) {

productDAO.save(p); // Only core logic here

}

}

**Spring does this internally:**

try {

openTransaction();

productDAO.save(p);

commitTransaction();

} catch(Exception e) {

rollbackTransaction();

}

**🔹 Benefit:**

You **focus only on the logic**, not boilerplate transaction code.

**🔹 Analogy:**

Think of @Service like a manager who handles the core task (business), and asks an assistant (Spring) to handle logistics like starting and finishing paperwork (transaction).

### ✅ Explanation of @Repository Annotation in Spring (Refined from Your Sir's Words):

#### 🔹 What is @Repository?

* @Repository is a **Spring stereotype annotation** used to mark **DAO (Data Access Object)** classes.
* It tells Spring, **"This class deals with persistence logic"** (like interacting with a database using JDBC or Hibernate).

#### 🔹 Why is it useful?

1. **Exception Translation (main point your sir explained):**
   * In traditional Java, data access code (e.g., JDBC) throws **checked exceptions** like SQLException.
   * These checked exceptions must be handled using try-catch or declared using throws.
   * When you annotate your DAO class with @Repository, Spring will automatically catch **any database-related exceptions** and re-throw them as **unchecked exceptions** (specifically, subclasses of DataAccessException).

✅ **This means**:

* + You don’t need to write try-catch blocks for every DB call in the DAO layer.
  + Spring handles it and allows upper layers (like Service or Controller) to handle exceptions **if needed**.

1. **Component Scanning:**
   * When you use @Repository, Spring automatically detects and registers this class as a **bean**, just like @Component.
2. **Separation of Concerns:**
   * You focus only on persistence logic (like save(), update()).
   * Let Spring handle the exception translation and bean management.

### Without @Repository:

You have to write try-catch and handle SQLException.

### With @Repository:

Spring catches SQLException, converts it to DataAccessException, and throws it **unchecked**, which can be optionally handled in the service layer.

**🔸 What is @Controller?**

It’s an annotation in Spring used to **handle HTTP web requests**.

You write a simple class and put @Controller on top. Spring will:

* Detect the class
* Create an object (bean)
* Connect incoming URLs to methods inside that class

**🧠 Real-world meaning**

Imagine you're at a hospital:

* The **receptionist** accepts your request and sends you to the right doctor.
* They don’t treat you—they just **coordinate**.

Similarly:

* The **Controller** accepts browser requests (like /home, /login)
* It doesn’t contain business logic
* It **calls the service layer**, which handles the actual processing

**🔄 How it works behind the scenes**

1. Browser sends a request: http://localhost:8080/home
2. Spring Boot runs an internal **Tomcat server**
3. **DispatcherServlet** (Spring's central front controller) catches that request
4. It looks for a class with @Controller
5. Finds the method mapped with @GetMapping("/home")
6. Executes the method
7. Sends response back to the browser

**💡 Why is @Controller useful?**

✅ You don't need to write:

* HttpServlet classes
* doGet(), doPost() methods
* Mapping logic for URLs

✅ Spring does it for you. You just write simple methods.

**🔧 Code Example (Easy)**

@Controller

public class HelloController {

@GetMapping("/hello")

public String sayHello(Model model) {

model.addAttribute("message", "Welcome to Spring!");

return "hello"; // View name → hello.html or hello.jsp

}

}

🟡 This method will be executed when you visit:  
http://localhost:8080/hello

Eg: BeanLifeCycle

### Java Class Lifecycle — Core Understanding

When you create and use an object in plain Java (without Spring), the following lifecycle steps are involved:

#### 1. **Static Block**

* **When?** Executed only once when the class is loaded by JVM.
* **Purpose:** Load static data/configurations.

static {

System.out.println("Static block executed");

}

#### 2. **Instance Block**

* **When?** Executes before the constructor **every time** you create an object.
* **Purpose:** Common setup code for all constructors.

{

System.out.println("Instance block executed");

}

#### 3. **Constructor**

* **When?** After instance block during object creation.
* **Purpose:** Initialize the object.

public MyBean() {

System.out.println("Constructor executed");

}

#### 4. **Setter Methods**

* **When?** Called manually in Java, or automatically in frameworks like Spring (for dependency injection).
* **Purpose:** Inject values or dependencies.

public void setValue(String value) {

this.value = value;

System.out.println("Setter executed");

}

#### 5. **Business Logic Execution**

* **When?** After object is fully ready.
* **Purpose:** You call the business methods on the object.

bean.processData();

* In your real project, this could be any method like:
  + sendEmail()
  + generateReport()
  + doWork() ← the one you actually used
  + calculate()

#### 6. **Destroy Phase**

* **When?** When object is no longer in use (eligible for GC).
* **In Java:** You may clean up manually via finalize() (deprecated now).
* **In Spring:** You can define custom destroy methods.