**✅ Full Form of @SpringBootApplication:**

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Inherited

@SpringBootConfiguration

@EnableAutoConfiguration

@ComponentScan

public @interface SpringBootApplication {

}

 @SpringBootConfiguration: A specialization of @Configuration, marks the class as a configuration class.

 @EnableAutoConfiguration: Tells Spring Boot to automatically configure beans based on the classpath and your properties.

 @ComponentScan: Enables scanning for @Component, @Service, @Repository, @Controller classes in the current package and sub-packages.

 @Configuration marks the class as a **configuration class**.

 @Bean tells Spring to manage the return value as a **bean** in the application context.

@Configuration

public class AppConfig {

@Bean

public Repository repository() {

return new RepositoryImpl();

}

@Bean

public Service service() {

return new ServiceImpl(repository());

}

}

**🆚 Difference between @Component and @Configuration:**

| **Annotation** | **Purpose** | **Bean methods** |
| --- | --- | --- |
| @Component | Marks a class as a Spring component | No |
| @Configuration | Configuration + bean methods | Yes, with @Bean |

The @ComponentScan annotation in Spring is used to tell the framework **where to look** for components (like @Component, @Service, @Repository, @Controller, etc.) to **auto-detect and register beans**.

**✅ Basic Usage:**

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

@Configuration

public class AppConfig {

}

This tells Spring to scan the com.example.app package and all its sub-packages for Spring-managed components.

## 🟡 @Bean Annotation

### ✅ Used in:

* A class annotated with @Configuration
* To define **manual bean creation logic**

### ✅ Example:

@Configuration

public class AppConfig {

@Bean

public MyService myService() {

return new MyServiceImpl();

}

}

### 🧠 Meaning:

* You're telling Spring: **"Create and manage this object as a bean."**
* Gives full control over instantiation logic.

## 🟢 @Component Annotation

### ✅ Used on:

* Classes you want Spring to **auto-detect and register as beans**
* Works only when class is in a @ComponentScan path

### ✅ Example:

@Component

public class MyServiceImpl implements MyService {

...

}

### 🧠 Meaning:

* You're telling Spring: **"This class is a Spring-managed component."**
* Spring will detect it via @ComponentScan and instantiate it automatically.

## 🔄 Key Differences

| **Feature** | **@Bean** | **@Component** |
| --- | --- | --- |
| Placement | On a method inside @Configuration | On a class |
| Instantiation | Manual (you write the creation logic) | Automatic via classpath scanning |
| Custom logic allowed | Yes (you write the logic in the method) | No (just simple class instantiation) |
| Flexibility | High (good for 3rd-party classes) | Easy for your own classes |

## 👀 Example Use Case

* Use @Component for **your own classes**, like services, repositories, controllers.
* Use @Bean when:
  + You need **custom initialization logic**
  + You’re wiring **external/third-party libraries** that you can’t annotate

### 🔧 @Autowired in Spring — Explained Simply

@Autowired is used to **automatically inject dependencies** into a Spring-managed class (i.e., wiring beans together).

## ✅ Basic Usage

java

CopyEdit

@Component

public class OrderService {

@Autowired

private PaymentService paymentService;

public void placeOrder() {

paymentService.processPayment();

}

}

### 💡 What happens:

Spring sees @Autowired and automatically injects an instance of PaymentService from the application context.

## 🧠 Where you can use @Autowired:

1. **Fields** (most common)

java

CopyEdit

@Autowired

private UserService userService;

1. **Constructors**

java

CopyEdit

@Autowired

public OrderService(UserService userService) {

this.userService = userService;

}

💡 From Spring 4.3+, if a class has only one constructor, you can **omit** @Autowired.

1. **Setters**

java

CopyEdit

@Autowired

public void setUserService(UserService userService) {

this.userService = userService;

}

## ✅ Example with @Component and @Service

java

CopyEdit

@Service

public class UserService {

public String getUser() {

return "John Doe";

}

}

@Component

public class AccountController {

@Autowired

private UserService userService;

public void printUser() {

System.out.println(userService.getUser());

}

}

## 🔄 Optional vs Required Injection

By default, Spring **throws an error** if it can’t find a bean to inject.

### To make it optional:

java

CopyEdit

@Autowired(required = false)

private DiscountService discountService;

## ✅ Best Practice:

* Prefer **constructor injection** → easier to test, immutable dependencies, and clearer contract.

## 🛠 Behind the Scenes:

Spring uses **Reflection** and **Dependency Injection (DI)** to inject beans at runtime, based on what's available in the context.

## 🔁 @Autowired vs @Inject

| **Feature** | **@Autowired** | **@Inject** |
| --- | --- | --- |
| Provided by | Spring Framework | Javax (JSR-330 standard) |
| Optional dependency | required = false | Needs @Nullable or Optional |
| Qualifier support | @Qualifier | @Named |
| Preferred in Spring | ✅ Yes | Sometimes (for standardization) |

### ✅ Example: @Autowired

@Autowired

private MyService myService;

Make optional:

@Autowired(required = false)

private MyService myService;

### ✅ Example: @Inject

import javax.inject.Inject;

@Inject

private MyService myService;

Optional with:

@Inject

@Nullable

private MyService myService;

OR

@Inject

private Optional<MyService> myService;

🔸 For @Inject to work, Spring still needs to manage the beans. So Spring scans them the same way, just uses a different annotation.

## 🎯 Handling Multiple Beans with @Qualifier

### Situation:

You have two beans of the same type — Spring won't know which one to inject unless you specify.

### ✅ Example:

@Component("smsService")

public class SmsNotificationService implements NotificationService {}

@Component("emailService")

public class EmailNotificationService implements NotificationService {}

Now inject with qualifier:

@Autowired

@Qualifier("emailService")

private NotificationService notificationService;

## ✅ TL;DR:

* @Component tells Spring: **"Create and manage this class as a bean."**
* @Autowired tells Spring: **"Inject the bean into this variable (field, constructor, or setter)."**

So:  
🟨 @Component = **register the class as a bean**  
🟩 @Autowired = **inject that bean where needed**

### 🧠 Think of it like this:

1. @Component creates the bean.
2. @Autowired pulls that bean **into another class** that needs it.

### ✅ Example to explain:

@Component

public class Engine {

public void start() {

System.out.println("Engine started.");

}

}

@Component

public class Car {

@Autowired

private Engine engine;

public void drive() {

engine.start(); // This will work because engine is injected

System.out.println("Car is moving...");

}

}

### What happens here?

* Spring sees @Component on Engine → registers it as a bean.
* Spring sees @Component on Car → registers it too.
* Spring sees @Autowired on the engine field in Car → injects the Engine bean automatically.

### ❓ So why not just use @Component alone?

Because:

* @Component **creates** the bean.
* But Spring **doesn't know** where to **inject** it unless you tell it — that’s where @Autowired comes in.

### ✅ Summary:

| **Annotation** | **Purpose** |
| --- | --- |
| @Component | Registers a class as a Spring bean |
| @Autowired | Injects a Spring bean where needed |