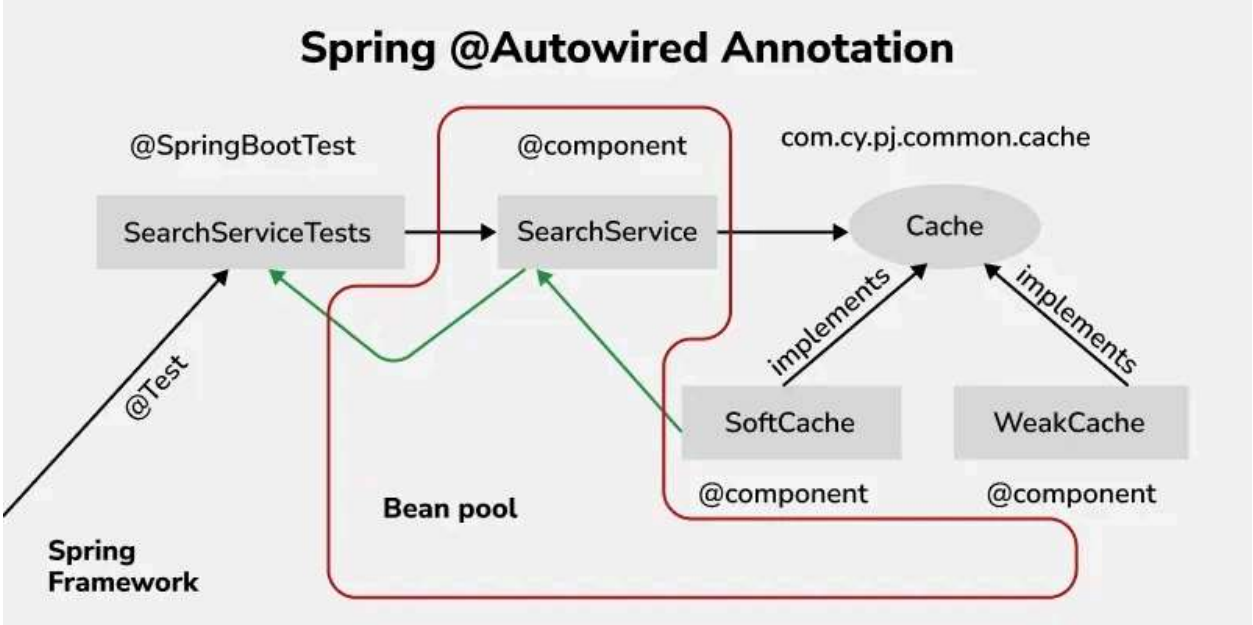
Spring @Autowired Annotation

Last Updated : 23 Jul, 2025

The **@Autowired annotation in Spring** marks a constructor, setter method, property, or configuration method to be autowired. This means that Spring will automatically inject the required dependencies (beans) at runtime using its Dependency Injection mechanism. The image below illustrates this concept:



Enabling @Autowired annotation

Spring beans can be declared either by **Java configuration** or **XML configuration**. By declaring beans, you provide metadata to the Spring Container which return the required dependency object at runtime. This is called **Spring Bean Autowiring**.

Java Based Configuration

In java based configuration, all the bean methods are defined in the class with **@configuration** annotation. At runtime, Spring will provide bean definitions by reading those methods. Using **@Autowired**, the right dependency is assigned by the Spring Container.

```
@Configuration
public class AppConfig {
    // Bean methods go here
}
```

XML-Based Configuration

In XML based configuration, if the beans are wired using **@Autowired annotation,** then **<context:annotation-config/>** has to be added to the XML file. Otherwise, you can include the **AutowiredAnnotationBeanPostProcessor** bean in the XML configuration file.

```
<bean
class="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor"/>
```

Using @Autowired with @SpringBootApplication

The **@SpringBootApplication annotation** is a combination of **@Configuration, @EnableAutoConfiguration and @ComponentScan.** It scans all the components or services and other configuration files included in the base and child packages and register them in Spring Context and inject the beans at runtime using @Autowired.

```
@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
```

```
    }
}
```

Using @Autowired in Different Scenarios

1. XML Configuration file

If XML configuration is used to wire the beans, then the configuration file looks like this

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans/"
        xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans/
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context/
        http://www.springframework.org/schema/context//spring-context.xsd"
        xmlns:context="http://www.springframework.org/schema/context/"
        >

    <context:annotation-config/>
    <bean id="customer" class="com.gfg.demo.domain.Customer">
        <property name="type" value="1" />
    </bean>

    <bean id="person" class="com.gfg.demo.domain.Person">
        <property name="name" value="ganesh" />
        <property name="age" value="21" />
    </bean>
</beans>
```

2. Java Configuration class

If Java Configuration is used to wire the beans, then the configuration class looks like this,

```
package com.gfg.demo.config;

import com.gfg.demo.domain.Customer;
import com.gfg.demo.domain.Person;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

```
@Configuration
public class AppConfig {
    @Bean
    public Person person(){
        Person person = new Person();
        person.setName("ganesh");
        person.setAge(21);
        return person;
    }
}
```

After enabling @Autowired annotation and deciding on what configuration to be used. The beans can be wired via constructor or properties or setter method. For example, there are two POJO classes Customer and Person. The Customer class has a dependency on the Person.

Customer.java:

```
@Component
public class Customer {

    private int type;
    private Person person;

    // Constructors
    // getters and setter
}
```

Person.java:

```
public class Person {
    private String name;
    private String age;

    // Constructors
    // getters and setters
```

}

3. Constructor-based Autowiring

@Autowired annotation is optional for constructor based injection. Here, the person object from the container is passed to the constructor while creating the Customer object.

```
@Component
public class Customer {

    private int type;
    private Person person;

    public Customer() {
    }

    @Autowired
    public Customer(Person person) {
        this.person = person;
    }
}
```

4. Property-based Autowiring

The person object will be injected into the property person at run time using **@Autowired** annotation

@Component
public class Customer {
private int type;

@Autowired
private Person person;
}

5. Setter based Autowiring

The setter method will be called with the Person object at runtime by the container.

```
@Autowired
public void setPerson(Person person) {
    this.person = person;
}
```

6. Optional Dependencies

If the bean of type **Person** is not defined then Spring will throw **NoSuchBeanDefinitionException**. It prevents the Spring Container **from** launching successfully by throwing the following exception.

```
org.springframework.beans.factory.NoSuchBeanDefinitionException:
No qualifying bean of type  'com.gfg.demo.Person' available:
expected at least 1 bean which qualifies as  autowire candidate.
Dependency annotations:
{@org.springframework.beans.factory.annotation.Autowired(required
=true)}
```

To fix this, we can set the **required** attribute of @Autowired as **false**,

```
@Autowired(required = false)
private Person person;
```