Spring - Dependency Injection by Setter Method

Last Updated : 23 Jul, 2025

Dependency Injection is one of the core features of the Spring Framework Inversion of Control (IOC) container. It reduces the need for classes to create their own objects by allowing the Spring IOC container to do it for them. This approach makes the code more flexible, easier to test, and simpler to maintain.

What is Dependency Injection?

It is a design pattern that allows the spring IOC Container to create and manage objects automatically by providing their required dependencies. In Spring, the IoC container handles this by injecting dependencies through constructors or setter methods.

Why Do We Need Dependency Injection?

In traditional programming, if class A requires an object of class B to perform its operations, class A directly creates an instance of class B. This creates a tight coupling between the two classes, making the code harder to test, maintain, and reuse.

```
// Tight coupling
class A {
    private B b = new B();
}
```

But with the help of Dependency Injection, the Spring IOC container injects the dependency (object of class B) into class A. This creates loose coupling between the two classes and makes the code more flexible and testable.

Types of Dependency Injection in Spring

There are two types of dependency injection in spring

- Constructor-Based Dependency Injection: Dependencies are injected using a constructor.
- Setter-Based Dependency Injection: Dependencies are injected using setter methods.

Setter-Based Dependency Injection

In Setter-Based Dependency Injection, dependencies are provided using setter methods. First, the Spring IoC container creates the object using a no-argument constructor, and then it calls the setter methods to set the required dependencies.

Example: This example demonstrates how to use Setter-Based Dependency Injection (DI) in Spring, where dependencies are injected using setter methods.

Step 1: Create a POJO Class (Student.java)

```java
package com.spring;

public class Student {
    private String studentName;
    private String studentCourse;

    // No-argument constructor (required for Setter-Based DI)
    public Student() {
    }

    // Setter methods for Dependency Injection
    public void setStudentName(String studentName) {
        this.studentName = studentName;
    }

    public void setStudentCourse(String studentCourse) {
        this.studentCourse = studentCourse;
    }

    @Override
    public String toString() {
        return "Student{studentName=" + studentName
        + ", studentCourse=" + studentCourse + "}";
    }
}
```

Step 2: Create the Spring Configuration File (config.xml)

```xml
<?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">

    <!-- Setter-Based Dependency Injection -->
    <bean id="stud" class="com.spring.Student">
        <property name="studentName" value="John" />
        <property name="studentCourse" value="Spring Framework" />
    </bean>

</beans>
```

Step 3: Create the Main Application (Main.java)

```java
package com.spring;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {
    public static void main(String[] args) {

        // Load the Spring configuration file
        ApplicationContext context
        = new ClassPathXmlApplicationContext("config.xml");

        // Retrieve the bean from the Spring container
        Student student = (Student) context.getBean("stud");

        // Print the student details
        System.out.println(student);
    }
}
```

Output:

```
Student{studentName=John, studentCourse=Spring Framework}
```

Annotation-Based Dependency Injection

XML based configuration is useful but modern Spring applications uses annotation for dependency injection. Some commonly used annotations are listed below:

- @Autowired: Automatically injects dependencies.
- @Component: Marks a class as a Spring bean.
- @Configuration: Indicates that the class contains Spring configuration.
- @Bean: Defines a bean in a configuration class.

Below is an example of how to use annotations for Setter-Based DI.

Step 1: Enable Component Scanning

Add the @ComponentScan annotation to a configuration class to enable component scanning.

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

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

Step 2: Annotate Classes with @Component

Annotate the Student class with @Component to mark it as a Spring bean.

```
import org.springframework.stereotype.Component;

@Component
public class Student {
    private String studentName;
    private String studentCourse;

    // Setter methods for Dependency Injection
    public void setStudentName(String studentName) {
        this.studentName = studentName;
    }
    public void setStudentCourse(String studentCourse) {
        this.studentCourse = studentCourse;
    }

    @Override
    public String toString() {
        return "Student{studentName=" + studentName + ", studentCourse=" +
studentCourse + "}";
    }
}
```

Step 3: Inject Dependencies Using @Autowired

Use the @Autowired annotation to inject dependencies via setter methods.

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class Student {
    private String studentName;
    private String studentCourse;

    @Autowired
    public void setStudentName(@Value("John") String studentName) {
        this.studentName = studentName;
    }

    @Autowired
    public void setStudentCourse(@Value("Spring Framework") String
studentCourse) {
        this.studentCourse = studentCourse;
    }

    @Override
    public String toString() {
        return "Student{studentName=" + studentName + ", studentCourse=" +
studentCourse + "}";
    }
}
```

Step 4: Create the Main Application Class

Update the Main class to use annotation-based configuration.

```
import org.springframework.context.ApplicationContext;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;
```

```
public class Main {
    public static void main(String[] args) {
        // Load the Spring application context using annotation-based
configuration
        ApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);

        // Retrieve the Student bean
        Student student = context.getBean(Student.class);

        // Print the student details
        System.out.println(student);
    }
}
```

Output:

```
Student{studentName=John, studentCourse=Spring Framework}
```

Java-Based Configuration

Modern Spring applications often use Java-based configuration instead of XML. Below is an example of how to configure the same example using Java-based configuration.

Step 1: Create a Configuration Class

Define a configuration class using the @Configuration annotation. Use the @Bean annotation to define beans.

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.beans.factory.annotation.Value;

@Configuration
public class AppConfig {

    @Bean
    public Student student(@Value("John") String studentName,
                           @Value("Spring Framework") String studentCourse) {
        Student student = new Student();
        student.setStudentName(studentName);
        student.setStudentCourse(studentCourse);
        return student;
    }
}
```

Step 2: Use the Configuration in the Main Class

Update the Main class to use Java-based configuration.

```
import org.springframework.context.ApplicationContext;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Main {
    public static void main(String[] args) {

        // Load the Spring application context
        // using Java-based configuration
        ApplicationContext context
        = new AnnotationConfigApplicationContext(AppConfig.class);

        // Retrieve the Student bean
        Student student = context.getBean(Student.class);

        // Print the student details
        System.out.println(student);
    }
}
```

Output:

```
Student{studentName=John, studentCourse=Spring Framework}
```

Advantages of Dependency Injection

- **Loose Coupling**: Classes work independently, making the system easier to manage and update.
- **Easier Testing**: Dependencies can be replaced with test versions, making unit testing simple.
- **Reusability:** Classes can be used in different projects without changes.
- **Centralized Configuration**: All dependencies are managed in one place (e.g., config.xml).