Spring @Bean Annotation with Example

Last Updated : 23 Jul, 2025

The **@Bean annotation in Spring** is a powerful way to define and manage beans in a Spring application. Unlike @Component, which relies on class-level scanning, @Bean explicitly declares beans inside @Configuration classes, offering greater flexibility in object creation. In this article, we will explore how @Bean works, its advantages, and how to use it for dependency injection, manual bean creation, and Spring IoC container management with practical examples.

Prerequisites:

Before diving into the @Bean annotation, it is recommended to have a basic understanding of the following Spring annotations:

- **Spring @ComponentScan Annotation:** @ComponentScan annotation is used to automatically detect and register beans in the Spring context.
- **Spring @Configuration Annotation:** @Configuration annotation indicates that a class declares one or more @Bean methods and may be processed by the Spring container to generate bean definitions.

What is the @Bean Annotation?

The **@Bean Annotation in Spring is used to define a method that produces a bean to be managed by the Spring IOC (Inversion of Control) container.** When we annotate a method with @Bean, we are telling Spring to manage this object (The method will return an instance of a class), and Spring will treat that instance as a bean in its container. These beans can be injected into other components such as **service, controllers, or repositories** through dependency injection.

Key points about @Bean Annotation:

- The @Bean annotation tells Spring that the returned object from the method must be registered as a Spring bean.
- The @Bean annotation is typically used in a @Configuration class.
- The @Bean annotation can also be used with other annotations like **@Primary, @Qualifier, and @Scope** for advanced configuration.

Example: This example demonstrates how to define a bean (MyService) in a Spring configuration class using the @Bean annotation so Spring manages its creation and lifecycle.

```
@Configuration
public class AppConfig {

    @Bean
    public MyService myService() {
        // Spring will manage this instance
        return new MyServiceImpl();
    }
}
```

Explanation: In the above code, the myService() method is annotated with @Bean annotation that means it provides a bean of type MyService. Spring will call this method to create a object and then the object will then be added to the Spring container. The object can be injected and used in other components of the Spring application.

Let's now create a simple Spring project to demonstrate the use of the @Bean annotation.

Creating a Bean Using @Bean Annotation

Step 1: Create a Simple Class

Suppose we already have a Java project and all the Spring JAR files are imported into the project. First, we are going to create a simple class named as College and inside the class we have a simple method.

Here, the @Component annotation tells the Spring to treat college class as a bean, which then can be automatically managed by the Spring container.

College.java:

```
// Java Program to Illustrate College Class

package BeanAnnotation;
// Importing required classes
import org.springframework.stereotype.Component;

@Component("collegeBean")
public class College {
    public void test()
    {
        System.out.println("Test College Method");
    }
}
```

Step 2: Create a Configuration Class

Now let's create a Configuration class named CollegeConfig.

Configuration Class:

```
// Java Program to Illustrate Configuration Class

package BeanAnnotation;
// Importing required classes
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

// Annotation
@Configuration
@ComponentScan(basePackages = "BeanAnnotation")

// Class
public class CollegeConfig {
}
```

But we do not want to use the @Component and @ComponentScan annotations to create the beans. Let's discuss another way of doing the same task.

Using the @Bean Annotation

We can manually create the beans in a configuration class using the @Bean annotation. In the CollegeConfig class, we will use @Bean annotation to define a beans for the class. Please refer to the comments for a better understanding.

```
@Bean
// Here the method name is the
// bean id/bean name
public College collegeBean()
{
   // Returns the College object
   return new College();
}
```

Here, the @Configuration annotation indicates that this class is a configuration class and the @Bean annotation is used to define a bean.

CollegeConfig.java:

```
// Java Program to Illustrate
// Configuration of College Class

package BeanAnnotation;

// Importing required classes
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration

public class CollegeConfig {

    // Creating College class Bean
    // using Bean annotation
    @Bean

    // Here the method name is the
    // bean id/bean name
    public College collegeBean()
    {
```

```
        // Returns the College class object
        return new College();
    }
}
```

Note: Whenever you are using the @Bean annotation to create the bean you don't need to use the @ComponentScan annotation inside your configuration class.

Step 3: Create the Main Application Class

Now, let's create a Main class to test our application.

Application Class:

```
// Java Program to Illustrate Application Class

package BeanAnnotation;

// Importing required classes
import org.springframework.context.ApplicationContext;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;

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

        // Using AnnotationConfigApplicationContext
        // instead of ClassPathXmlApplicationContext
        // because we are not using XML Configuration
        ApplicationContext context
            = new AnnotationConfigApplicationContext(
                CollegeConfig.class);

        // Getting the bean
        College college
            = context.getBean("collegeBean", College.class);

        // Invoking the method
        // inside main() method
        college.test();
    }
}
```

Output:

```
Test College Method
```

Note: Now let's remove the @Bean annotation before the collegeBean() method and run the program again we are going to get the **NoSuchBeanDefinitionException**

Exception in thread "main" org.springframework.beans.factory.NoSuchBeanDefinitionException: No bean named 'collegeBean' available

at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBeanDefinition(DefaultListableBeanFactory.java:863)

at org.springframework.beans.factory.support.AbstractBeanFactory.getMergedLocalBeanDefinition(AbstractBeanFactory.java:1344)

at org.springframework.beans.factory.support.AbstractBeanFactory.doGetBean(AbstractBeanFactory.java:309)

at org.springframework.beans.factory.support.AbstractBeanFactory.getBean(AbstractBeanFactory.java:213)

at org.springframework.context.support.AbstractApplicationContext.getBean(AbstractApplicationContext.java:1160)

at BeanAnnotation.Main.main(Main.java:15)

So the point is to make the collegeBean() method work like a bean you need to define the @Bean annotation before that particular method.

Customizing Bean Names

Now the question arises can we give a different Bean ID for the collegeBean() method? The answer is YES, we can. We can modify our code something like this

```
// Annotation
@Bean(name = "myCollegeBean")
// class
public College collegeBean()
```

```
{
    return new College();
}
```

So, in order to test our application, we also need to do changes in the Main.java file

Application class:

```
// Java Program to Illustrate Application Class

package BeanAnnotation;

// Importing required classes
import org.springframework.context.ApplicationContext;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Main {

    public static void main(String[] args)
    {

        // Using AnnotationConfigApplicationContext
        // instead of ClassPathXmlApplicationContext
        // because we are not using XML Configuration
        ApplicationContext context
            = new AnnotationConfigApplicationContext(
                CollegeConfig.class);

        // Getting the bean
        College college = context.getBean("myCollegeBean",
                                          College.class);

        // Invoking the method
        // inside main() method
        college.test();
    }
}
```

Giving Multiple Names to the Same Bean

One more interesting thing is we can give multiple names to this particular collegeBean() method. So further we can modify our code something like this.

@Bean(name = {"myCollegeBean", "yourCollegeBean"})
public College collegeBean()

```
{
    return new College();
}
```

Similary, in order to run the application, we need to modify Main.java file

Dependency Injection with @Bean

Now let's discuss another scenario. Suppose we have a dependency class named Principal inside our College class then what to do? So the scenario is like this. We have a class named Principal.java and we have defined a simple method inside this.

Example: Here, we have created a simple Principle class with a method principalInfo() that prints a message.

```
// Java Program to Illustrate Principal Class

package BeanAnnotation;
public class Principal {

    public void principalInfo()
    {
        System.out.println("Hi, I am your principal");
    }
}
```

The College class looks something like below:

College.java:

```
// Java Program to Illustrate College Class

package BeanAnnotation;

public class College {

    // Class data member
    private Principal principal;

    // Method
    public void test()
    {
        principal.principalInfo();

        // Print statement
        System.out.println("Test College Method");
```

```
    }
}
```

So now we want to do the dependency injection. So we can do it in 2 ways as listed later implemented as shown below:

1. Constructor Dependency Injection (CDI)
2. Setter Dependency Injection (SDI)

1. Constructor Dependency Injection (CDI)

In this approach, we inject the dependency using a constructor. So our modified College.java file is,

College.java:

```
// Java Program to Illustrate College Class

package BeanAnnotation;

// Class
public class College {

    private Principal principal;

    public College(Principal principal)
    {
        this.principal = principal;
    }

    public void test()
    {
        principal.principalInfo();
        System.out.println("Test College Method");
    }
}
```

Now come to the CollegeConfig.java file and the modified CollegeConfig.java is given below. Refer to the comments for better understanding.

CollegeConfig.java:

```
// Java Program to Illustrate Configuration Class
```

```
// Java Program to Illustrate Configuration Class

package BeanAnnotation;

// Importing required classes
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

// Annotation
@Configuration
public class CollegeConfig {

    // Creating the Bean for Principal Class
    @Bean public Principal principalBean()
    {
        return new Principal();
    }

    @Bean public College collegeBean()
    {
        // Constructor Injection
        return new College(principalBean());
    }
}
```

And finally Below is the code for the Main.java file.

Application class:

```
// Java Program to Illustrate Application Class

package BeanAnnotation;

// Importing required classes
import org.springframework.context.ApplicationContext;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;

// Main class
public class Main {

    // Main driver method
    public static void main(String[] args)
    {

        // Using AnnotationConfigApplicationContext
        // instead of ClassPathXmlApplicationContext
        // because we are not using XML Configuration
        ApplicationContext context
            = new AnnotationConfigApplicationContext(
                CollegeConfig.class);
```

```
        // Getting the bean
        College college
            = context.getBean("collegeBean", College.class);

        // Invoking the method
        // inside main() method
        college.test();
    }
}
```

Output:

```
Hi, I am your principal
Test College Method
```

2. Setter Dependency Injection

In this approach, we inject the dependency using a setter method. So our modified College.java file is as follows:

College.java:

```
// Java Program to Illustrate College Class

package BeanAnnotation;

// Class
public class College {

    // Class data members
    private Principal principal;

    // Setter
    public void setPrincipal(Principal principal)
    {

        // this keywords refers to current instance itself
        this.principal = principal;
    }

    // Method
    public void test()
    {
        principal.principalInfo();

        // Print statement
        System.out.println("Test College Method");
    }
}
```

Now come to the CollegeConfig.java file and the modified CollegeConfig.java is given below as follows:

CollegeConfig.java:

```
// Java Program to Illustrate Configuration Class

package BeanAnnotation;

// Importing required classes
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

// Annotation
@Configuration

// Class
public class CollegeConfig {

    // Creating the Bean for Principal Class
    @Bean public Principal principalBean()
    {

        return new Principal();
    }

    @Bean public College collegeBean()
    {

        // Setter Injection
        College college = new College();
        college.setPrincipal(principalBean());

        return college;
    }
}
```

And finally Below is the code for the Main.java file.

Application Class:

```
// Java Program to Illustrate Application (Main) Class

package BeanAnnotation;
```

```
// Importing required classes
import org.springframework.context.ApplicationContext;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;

// Application (Main) class
public class Main {

    // Main driver method
    public static void main(String[] args)
    {

        // Using AnnotationConfigApplicationContext
        // instead of ClassPathXmlApplicationContext
        // because we are not using XML Configuration
        ApplicationContext context
            = new AnnotationConfigApplicationContext(
                CollegeConfig.class);

        // Getting the bean
        College college
            = context.getBean("collegeBean", College.class);

        // Invoking the method
        // inside main() method
        college.test();
    }
}
```

Output:

```
Hi, I am your principal
Test College Method
```

When to Use @Component vs @Bean

- Prefer @Component when you have control over the source code and want automatic scanning.
- Use @Bean when dealing with third-party classes or custom initialization logic is needed.