Spring - Understanding Inversion of Control with Example

Last Updated : 04 Aug, 2025

Spring IoC (Inversion of Control) Container is the core of the Spring Framework. It creates and manages objects (beans), injects dependencies and manages their life cycles. It uses Dependency Injection (DI), based on configurations from XML files, Java-based configuration, annotations or POJOs. Since the container, not the developer, controls object creating and wiring, it's called Inversion of Control (IoC).

Types Ioc Containers

There are two types of IoC containers in Spring:

- BeanFactory
- ApplicationContext

1. BeanFactory

- The BeanFactory is the most basic version of the IoC container.
- It provides basic support for dependency injection and bean lifecycle management.
- It is suitable for lightweight applications where advanced features are not required.

2. ApplicationContext

- The ApplicationContext is an extension of BeanFactory with more enterprise features like event propagation, internationalization and

more.
- It is the preferred choice for most Spring applications due to its advanced capabilities.

Key Features of IoC Container

The key features of IoC Container are listed below

- **Dependency Injection:** Automatically injects dependencies into our classes.
- **Lifecycle Management:** Manages the lifecycle of beans, including instantiation, initialization and deletion.
- **Configuration Flexibility:** Supports both XML-based and annotation-based configurations.
- **Loose Coupling:** Promotes loose coupling by decoupling the implementation of objects from their usage.

Understanding IoC in Spring with a Practical Example

Step 1: Create the Sim Interface

So now let's understand what is IoC in Spring with an example. Suppose we have one interface named Sim and it has some abstract methods calling() and data().

```
public interface Sim
{
    void calling();
    void data();
}
```

Step 2: Implement the Sim interface

Now we have created another two classes, Airtel and Jio which implement the Sim interface and override its methods..

```
public class Airtel implements Sim {
```

```
    @Override
    public void calling() {
        System.out.println("Airtel Calling");
    }

    @Override
    public void data() {
        System.out.println("Airtel Data");
    }
}

public class Jio implements Sim {
    @Override
    public void calling() {
        System.out.println("Jio Calling");
    }

    @Override
    public void data() {
        System.out.println("Jio Data");
    }
}
```

Step 3: Calling Methods Without Spring IoC

Without Spring IoC, we would manually create instances of the Sim implementation in the main method. For example:

```
public class Mobile {

    // Main driver method
    public static void main(String[] args)
    {
        // Manually creating an instance of Jio
        Sim sim = new Jio();

        // Calling methods
        sim.calling();
        sim.data();
    }
}
```

This approach tightly couples the Mobile class to the Jio implementation. If we want to switch to Airtel, we need to modify the source code.

Step 4: Using Spring IoC with XML Configuration

To avoid tight coupling, we can use the Spring IoC container. First, we create an XML configuration file (beans.xml) to define the beans.

Example: beans.xml File

```
<?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">

  <!-- Define the Jio bean -->
  <bean id="sim" class="Jio"></bean>

</beans>
```

Explanation: In the beans.xml file, we define beans by giving each a unique id and specifying the class name. Later, in the main method, we can use these beans — which will be shown in the next example.

Step 5: Run the Code

In the Mobile class, we use the ApplicationContext to retrieve the bean:

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

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

        // Using ApplicationContext tom implement Spring IoC
        ApplicationContext applicationContext
        = new ClassPathXmlApplicationContext("beans.xml");

        // Get the bean
        Sim sim = applicationContext.getBean("sim", Sim.class);

        // Calling the methods
        sim.calling();
        sim.data();
    }
}
```

Output:

```
Jio Calling
Jio Data
```

And now if you want to use the Airtel sim so you have to change only inside the beans.xml file. The main method is going to be the same.

<bean id="sim" class="Airtel"></bean>

Implementation:

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

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

        // Using ApplicationContext tom implement Spring IoC
        ApplicationContext applicationContext
        = new ClassPathXmlApplicationContext("beans.xml");

        // Get the bean
        Sim sim = applicationContext.getBean("sim", Sim.class);

        // Calling the methods
        sim.calling();
        sim.data();
    }
}
```

Output:

```
Airtel Calling
Airtel Data
```

Step 6: Using Java-Based Configuration

Modern Spring applications often use Java-based configuration instead of XML. Let's see how to configure the same example using Java-based configuration.

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;
```

```
@Configuration
public class AppConfig {

    @Bean
    public Sim sim() {
        return new Jio(); // Change to new Airtel() to switch implementations
    }
}
```

2. Use the Configuration in the Mobile Class: Update the Mobile class to use the Java-based configuration.

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

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

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

        // Retrieve the bean
        Sim sim = context.getBean("sim", Sim.class);

        // Call methods
        sim.calling();
        sim.data();
    }
}
```

Output:

```
Jio Calling
Jio Data
```

To switch to Airtel, simply update the AppConfig class:

```
@Bean
public Sim sim() {
    return new Airtel();
}
```

Step 7: Using Annotations for Dependency Injection

Spring also supports annotation-based configuration, which is widely used in modern applications. Let's update the example to use annotations.

1. Enable Component Scanning: Add the @ComponentScan annotation to the configuration class to enable component scanning.

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

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

2. Annotate Classes with @Component: Annotate the Airtel and Jio classes with @Component

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

@Component
public class Airtel implements Sim {
    @Override
    public void calling() {
        System.out.println("Airtel Calling");
    }

    @Override
    public void data() {
        System.out.println("Airtel Data");
    }
}

@Component
public class Jio implements Sim {
    @Override
    public void calling() {
        System.out.println("Jio Calling");
    }

    @Override
    public void data() {
        System.out.println("Jio Data");
    }
}
```

3. Inject the Dependency Using @Autowired: Update the Mobile class to use @Autowired for dependency injection.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.stereotype.Component;

@Component
public class Mobile {

    @Autowired
    private Sim sim;

    public void useSim() {
        sim.calling();
        sim.data();
    }

    public static void main(String[] args) {
        ApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);
        Mobile mobile = context.getBean(Mobile.class);
        mobile.useSim();
    }
}
```

Output:

```
Jio Calling
Jio Data
```

Note: When we run the above code, Spring will throw a NoUniqueBeanDefinitionException because it finds two beans (Airtel and Jio) of type Sim. To resolve this, we need to specify which bean to inject.

Fixing The NoUniqueBeanDefinitionException

1. Use @Primary Annotation

Mark one of the beans as the primary bean using the @Primary annotation.

```
@Component
@Primary
public class Airtel implements Sim {
    // Methods implementation
}
```

2. Use @Qualifier Annotation

Use the @Qualifier annotation to specify which bean to inject

```
@Component
public class Mobile {

    @Autowired
    @Qualifier("jio") // Specify the bean name
    private Sim sim;

    public void useSim() {
        sim.calling();
        sim.data();
    }

    public static void main(String[] args) {
        ApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);
        Mobile mobile = context.getBean(Mobile.class);
        mobile.useSim();
    }
}
```

3. Explicit Bean Names

Explicitly name the beans and use those names in the @Qualifier annotation.

```
@Component("airtelBean")
public class Airtel implements Sim {
    // Methods implementation
}

@Component("jioBean")
public class Jio implements Sim {
    // Methods implementation
}
```

Then in the Mobile class,

```
@Component
public class Mobile {

    @Autowired
    @Qualifier("jioBean") // Use the explicit bean name
```

```
    private Sim sim;

    public void useSim() {
        sim.calling();
        sim.data();
    }

    public static void main(String[] args) {
        ApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);
        Mobile mobile = context.getBean(Mobile.class);
        mobile.useSim();
    }
}
```