### **Spring Framework**

Spring is an open-source framework created to address the complexity of an enterprise application development

One of the chief advantages of the Spring framework is its layered architecture, which allows developers to be selective about which of its components they can use while providing a cohesive framework for J2EE application development

**Spring framework provides support and integration to various technologies for e.g.:**

**>** Support for Transaction Management

**>** Support for interaction with the different databases

**>** Integration with the Object Relationship frameworks for e.g. Hibernate, iBatis etc

**>** Support for Dependency Injection which means all the required dependencies will be resolved with the help of containers

**>** Support for REST style web-services

**Spring Dependency Injection with Example :**

**What is Dependency Injection:**

Dependency Injection (DI) is a design pattern means injecting the dependency between the two objects as per the application’s requirement and helps to reduce the dependency to each other and provides independent unit testing of every object.

Dependency Injection is the main functionality provided by [Spring](https://www.geeksforgeeks.org/introduction-to-spring-framework/) IOC(Inversion of Control). The Spring-Core module is responsible for injecting dependencies through either Constructor or Setter methods.

Dependency Injection in [Spring](https://www.geeksforgeeks.org/introduction-to-spring-framework/) also ensures loose-coupling between the classes.

The design principle of Inversion of Control emphasizes keeping the Java classes independent of each other and the container frees them from object creation and maintenance.

Dependency injection (DI) is a design principle to makes your application:

> easier to develop

> your code less coupled

> easier to test your code

Spring provides a light-weight container, e.g. the Spring core container, for DI. The injection in Spring is either done via setter injection of via construction injection. These classes which are managed by Spring must conform to the JavaBean standard. In the context of Spring classes are also referred to as beans or as Spring beans.

**The Spring core container:**

> handles the configuration, generally based on annotations or on an XML file (XMLBeanFactory)

> manages the selected Java classes via the BeanFactory

The core container uses the so-called bean factory to create new objects. New objects are generally created as Singletons if not specified differently.

The injection in Spring is either done via setter, field or constructor injection. Classes which are managed by Spring DI must conform to the Java bean standard.

In the context of Spring classes are also referred to as beans or as spring beans.

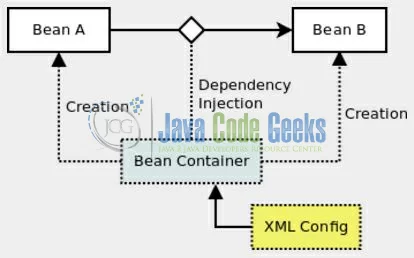
**Need for Dependency Injection:** Suppose class One needs the object of class Two to instantiate or operate a method, then class One is said to be dependent on class Two.

**[Spring](https://www.geeksforgeeks.org/introduction-to-spring-framework/) IOC** resolves such dependencies with Dependency Injection, which makes the code easier to test and reuse. [Loose coupling](https://www.geeksforgeeks.org/coupling-in-java/) between classes can be possible by defining [interfaces](https://www.geeksforgeeks.org/interfaces-in-java/) for common functionality and the injector will instantiate the objects of required implementation. The task of instantiating objects is done by the container according to the configurations.

**Types of Spring Dependency Injection:**There are two types of Spring Dependency Injection. They are:

**1.Setter Dependency Injection (SDI):**

**2.Constructor Dependency Injection (CDI)**

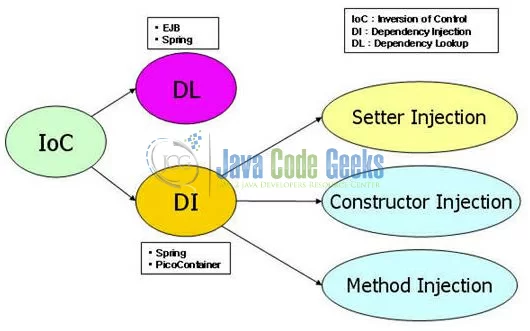


1. **Setter Dependency Injection (SDI):**

Spring framework called Setter-Based Dependency Injection which simply injects the dependent objects into the client using a setter method.

It is accomplished by the container calling the setter methods on the beans after invoking a no-argument constructor or a no-argument static factory method to instantiate the bean.

Setter Injection is the preferable method of Dependency Injection (DI) in the spring framework. Setter-based DI is implemented by calling a setter method on the application’s bean. From the configuration point of view, Setter Injection is easier to understand because the property name being set is used as an attribute to the bean.



#### **Example of Setter Injection in Spring Framework :**

For e.g. In an ATM (Automated Teller Machine) system, the ATM class and Printer class can collaborate with each other to print the balance information for a bank account.

Collaboration between classes is usually expressed as the dependency where the reference of one class is held by another class. For e.g. the ATM class holds a reference to the Printer class i.e.

public class **ATM** {

private Printer printerObj;

….

}

In this example, spring will create an instance of the Printer class and associate that instance with the printerObj member in the ATM class. However as the printerObj member in the ATM class is private, the ATM class needs to expose its dependency to the spring framework for it to inject the Printer instance into the ATM class. For e.g. If the ATM class exposes its dependency on the Printer class as a setter method then the spring framework can inject the Printer object and is known as Setter Based Injection.

#### **Implementation of Model Class**

**public** **class** Employee {

**private** **int** emp\_id;

**private** String emp\_fname;

**private** String emp\_city;

    // Dependency Injection By Setter Methods

    // setter & getter method

**public** **void** showResult() {

        System.out.println("\nEmployee Details? Id= " + emp\_id + ", Name= " + emp\_fname + ", City= " + emp\_city);

}

}

**Implementation of Utility Class**

This class will get the bean definition from the context file (i.e. spring-beans.xml) and calls the showResult() method of the Employee bean class to display the values injected through the setter.

**import** org.springframework.context.ApplicationContext;

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

**public** **class** AppMain {

    @SuppressWarnings("resource")

**public** **static** **void** main(String[] args) {

        ApplicationContext contextObj = **new** ClassPathXmlApplicationContext("spring-beans.xml");

        Employee empObj = (Employee) contextObj.getBean("employeeBean");

        empObj.showResult();

    }

}

**Configuration File**

To configure the spring framework, we need to implement a bean configuration file i.e.**spring-beans.xml** which provides an interface between the basic Java class and the outside world.

<**beans**… >

    <!-- Definition For Employee Bean -->

    <**bean** id="employeeBean" class="com.jcg.spring.setter.injection.Employee">

        <!-- Injecting Primitive Values As Dependencies-->

        <**property** name="emp\_id">

            <**value**>101</**value**>

        </**property**>

        <**property** name="emp\_fname">

            <**value**>Daniel Atlas</**value**>

        </**property**>

        <**property** name="emp\_city">

            <**value**>Greece</**value**>

        </**property**>

    </**bean**>

</**beans**>

> We have set the bean id as: employeeBean for the Employee class which will act as a reference for calling the said class

> The employee details are set via the setter injection by using the <property /> tag. The name refers to the property names of the corresponding bean and the value sub-element of <property /> tag will assign the specified value

This is the simpler of the two DI methods. the DI will be injected with the help of setter and/or getter methods. Now to set the DI as SDI in the bean, it is done through the bean-configuration file For this, the property to be set with the SDI is declared under the <property> tag in the bean-config file.

**Example**: Let us say there is class GFG that uses SDI and sets the property geeks. The code for it is given below.

|  |
| --- |
| **import** com.geeksforgeeks.org.IGeek;    **public** **class** GFG {        // The object of the interface IGeek      IGeek geek;        // Setter method for property geek  **public** **void** setGeek(IGeek geek)     {  **this**.geek = geek;      }  } |

**Setting the SDI in the bean-config file:**

|  |
| --- |
| <**beans…..**>        <**bean** id="GFG" class="com.geeksforgeeks.org.GFG">          <**property** name="geek">              <**ref** bean="CsvGFG" />          </**property**>      </**bean**>    <**bean** id="CsvGFG" class="com.geeksforgeeks.org.impl.CsvGFG" />  <**bean** id="JsonGFG" class="com.geeksforgeeks.org.impl.JsonGFG" />    </**beans**> |

This injects the ‘CsvGFG’ bean into the ‘GFG’ object with the help of a setter method (‘setGeek’)

1. **Constructor Dependency Injection (CDI):** In this, the DI will be injected with the help of [contructors](https://www.geeksforgeeks.org/constructors-in-java/). Now to set the DI as CDI in bean, it is done through the bean-configuration file For this, the property to be set with the CDI is declared under the **<constructor-arg>** tag in the bean-config file.

**Example**: Let us take the same example as of SDI

|  |
| --- |
| **public** **class** GFG {        // The object of the interface IGeek      IGeek geek;        // Constructor to set the CDI      GFG(IGeek geek) {  **this**.geek = geek;      }  } |

**Setting the CDI in the bean-config file:**

|  |
| --- |
| <**beans** …. >        <**bean** id="GFG" class="com.geeksforgeeks.org.GFG">          <**constructor-arg**>              <**bean** class="com.geeksforgeeks.org.impl.CsvGFG" />          </**constructor-arg**>      </**bean**>    <**bean** id="CsvGFG" class="com.geeksforgeeks.org.impl.CsvGFG" />  <**bean** id="JsonGFG" class="com.geeksforgeeks.org.impl.JsonGFG" />    </**beans**> |

This injects the ‘CsvGFG’ bean into the ‘GFG’ object with the help of a constructor.

**IoC (Inversion of Control)** and **DI (Dependency Injection) :**

**What Is Inversion of Control :**Inversion of Control is a principle in software engineering by which the control of objects .It's most often used in the context of object-oriented programming.

IoC enables a framework to take control of the flow of a program and make calls to our custom code.

The advantages of this **architecture** are:

* decoupling the execution of a task from its implementation.
* making it easier to switch between different implementations.
* greater modularity of a program.
* greater ease in testing a program by isolating a component or mocking its dependencies and allowing components to communicate through contracts.

Inversion of Control can be achieved through various mechanisms such as: Strategy design pattern, Service Locator pattern, Factory pattern, and **Dependency Injection (DI).**

## **What Is Dependency Injection?**

Dependency injection is a pattern through which to implement IoC, where the control being inverted is the setting of object's dependencies.

The act of connecting objects with other objects, or “injecting” objects into other objects, is done by an assembler rather than by the objects themselves.

how you would create an object dependency in traditional programming:

public class Store {

private Item item;

public Store() {

item = new ItemImpl1();

}}

In the example above, we need to instantiate an implementation of the Item interface within the Store class itself.

By using DI, we can rewrite the example without specifying the implementation of Item that we want:

public class Store {

private Item item;

public Store(Item item) {

this.item = item;

}}

## **The Spring IoC Container :**

An **IoC container** is a common characteristic of frameworks that implement IoC.

In the Spring framework, the IoC container is represented by the interface **ApplicationContext**. The Spring container is responsible for **instantiating**, **configuring** and **assembling objects known as beans**, as well as **managing their lifecycle**.

The Spring framework provides several implementations of the **ApplicationContext**interface — **ClassPathXmlApplicationContext**and **FileSystemXmlApplicationContext**for standalone applications, and **WebApplicationContext**for web applications.

In order to assemble beans, the container uses configuration metadata, which can be in the form of XML configuration or annotations.

**Here's one way to manually instantiate a container:**

**ApplicationContext context**

**= new ClassPathXmlApplicationContext("applicationContext.xml");**

To set the item attribute in the example above, we can use metadata. Then, the container will read this metadata and use it to assemble beans at runtime.

**Dependency Injection in Spring can be done through constructors, setters or fields.**

**Constructor-Based Dependency Injection :** the container will invoke a constructor with arguments each representing a dependency we want to set.

Spring resolves each argument primarily by type, followed by name of the attribute and index for disambiguation.

Let's see the configuration of a bean and its dependencies **using annotations:**

**@Configuration**

public class AppConfig {

**@Bean**

public Item item1() {

return new ItemImpl1();

}

**@Bean**

public Store store() {

return new Store(item1());

}

}

The **@Configuration** annotation indicates that the class is a source of bean definitions. Also, we can add it to multiple configuration classes.

The **@Bean** annotation is used on a method to define a bean. If we don't specify a custom name, the bean name will default to the method name.

For a bean with the default singleton scope, Spring first checks if a cached instance of the bean already exists and only creates a new one if it doesn't. If we're using the prototype scope, the container returns a new bean instance for each method call.

Another way to create the configuration of the beans is through **XML configuration:**

<bean id="item1" class="org.store.ItemImpl1" />

<bean id="store" class="org.store.Store">

<**constructor-arg** type="ItemImpl1" index="0" name="item" ref="item1" />

</bean>

## **Setter-Based Dependency Injection :**For setter-based DI, the container will call setter methods of our class, after invoking a no-argument constructor or no-argument static factory method to instantiate the bean. Let's create this configuration using annotations:

@Bean

public Store store() {

Store store = new Store();

store.setItem(item1());

return store;

}

We can also use **XML** for the same configuration of beans:

<bean id="store" class="org.store.Store">

<property name="item" ref="item1" />

</bean>

**Constructor-based** and **setter-based** types of injection can be combined for the same bean. The Spring documentation recommends using constructor-based injection for mandatory dependencies, and setter-based injection for optional ones.

## **Field-Based Dependency Injection :**In case of Field-Based DI, we can inject the dependencies by marking them with an **@Autowired** annotation.

public class Store {

**@Autowired**

private Item item;

}

While constructing the Store object, if there's no constructor or setter method to inject the Item bean, the container will use reflection to inject Item into Store.

This approach might look simpler and cleaner but is not recommended to use because it has a few **drawbacks** such as:

* This method uses reflection to inject the dependencies, which is costlier than constructor-based or setter-based injection
* It's really easy to keep adding multiple dependencies using this approach. If you were using constructor injection having multiple arguments would have made us think that the class does more than one thing which can violate the Single Responsibility Principle.

## **Autowiring Dependencies :**

**[Wiring](https://www.baeldung.com/spring-annotations-resource-inject-autowire)**allows the Spring container to automatically resolve dependencies between collaborating beans by inspecting the beans that have been defined.

There are **four modes of autowiring** a bean using an XML configuration:

* **no**: the default value – this means no autowiring is used for the bean and we have to explicitly name the dependencies
* **byName**: autowiring is done based on the name of the property, therefore Spring will look for a bean with the same name as the property that needs to be set
* **byType**: similar to the **byName autowiring,** only based on the type of the property. This means Spring will look for a bean with the same type of the property to set. If there's more than one bean of that type, the framework throws an exception.
* **constructor**: autowiring is done based on constructor arguments, meaning Spring will look for beans with the same type as the constructor arguments.

For example, let's autowire the item1 bean defined above by type into the store bean:

**@Bean(autowire = Autowire.BY\_TYPE)**

public class Store {

private Item item;

public setItem(Item item){

this.item = item;

}}

We can also inject beans using the @Autowired annotation for autowiring by type:

public class Store {

**@Autowired**

private Item item; }

**Note : If there's more than one bean of the same type, we can use the @Qualifier annotation to reference a bean by name:**

public class Store {

**@Autowired**

**@Qualifier("item1")**

private Item item;

}

Now, let's autowire beans by type through XML configuration:

**<bean id="store" class="org.store.Store" autowire="byType"> </bean>**

Next, let's inject a bean named item into the item property of store bean by name through XML:

**<bean id="item" class="org.store.ItemImpl1" />**

**<bean id="store" class="org.store.Store" autowire="byName">**

**</bean>**

We can also override the autowiring by defining dependencies explicitly through constructor arguments or setters.

**Lazy Initialized Beans**

By default, the container creates and configures all singleton beans during initialization. To avoid this, you can use the lazy-init attribute with value true on the bean configuration:

**<bean id="item1" class="org.store.ItemImpl1" lazy-init="true" />**

As a consequence, the item1 bean will be initialized only when it's first requested, and not at startup. The advantage of this is faster initialization time, but the trade-off is that configuration errors may be discovered only after the bean is requested, which could be several hours or even days after the application has already been running.

**Constructor Dependency Injection in Spring Example :**

**Annotation Based Configuration**

Java configuration file looks pretty much like a plain-old Java object with some additional annotations:

@Configuration

@ComponentScan("com.constructordi")

public class Config {

@Bean

public Engine engine() {

return new Engine("v8", 5);

}

@Bean

public Transmission transmission() {

return new Transmission("sliding");

}

}

Here we are using annotations to notify Spring runtime that this class is a provider of bean definitions (@Bean annotation) and that a context scan for additional beans needs to be performed in package com.baeldung.spring. Next, we define a Car class:

@Component

public class Car {

@Autowired

public Car(Engine engine, Transmission transmission) {

this.engine = engine;

this.transmission = transmission;

}

}

Spring will encounter our Car class while doing a package scan and will initialize its instance by calling the @Autowired annotated constructor.

Instances of Engine and Transmission will be obtained by calling @Bean annotated methods of the Config class. Finally, we need to bootstrap an ApplicationContext using our POJO configuration:

ApplicationContext context = new AnnotationConfigApplicationContext(Config.class);

Car car = context.getBean(Car.class);

**Implicit Constructor Injection**

As of Spring 4.3, classes with a single constructor can omit the @Autowired annotation. A nice little bit of convenience and boilerplate removal!

On top of that, also starting with 4.3, the constructor-based injection can be leveraged in @Configuration annotated classes. And yes, if such a class has only one constructor the @Autowired annotation can be omitted as well.

**XML Based Configuration**

Another way to configure Spring runtime with constructor-based dependency injection is to use an XML configuration file:

<bean id="toyota" class="com.baeldung.constructordi.domain.Car">

<constructor-arg index="0" ref="engine"/>

<constructor-arg index="1" ref="transmission"/>

</bean>

<bean id="engine" class="com.baeldung.constructordi.domain.Engine">

<constructor-arg index="0" value="v4"/>

<constructor-arg index="1" value="2"/>

</bean>

<bean id="transmission" class="com.baeldung.constructordi.domain.Transmission">

<constructor-arg value="sliding"/>

</bean>

Note that constructor-arg can accept a literal value or a reference to another bean and that an optional explicit index and type can be provided. Type and index attributes can be used to resolve ambiguity (for example if a constructor takes multiple arguments of the same type).

name attribute could also be used for xml to java variable matching, but then your code must be compiled with debug flag on.

A Spring application context, in this case, needs to be bootstrapped using ClassPathXmlApplicationContext:

ApplicationContext context = new ClassPathXmlApplicationContext("baeldung.xml");

Car car = context.getBean(Car.class);

**Pros and Cons**

Constructor injection has a few advantages compared to field injection.

The first benefit is testability. Suppose we're going to unit test a Spring bean that uses field injection:

public class UserService {

@Autowired

private UserRepository userRepository;

}

During the construction of a UserService instance, we can't initialize the userRepository state. The only way to achieve this is through the Reflection API, which completely breaks encapsulation. Also, the resulting code will be less safe compared to a simple constructor call.

Additionally, with field injection, we can't enforce class-level invariants. So it's possible to have a UserService instance without a properly initialized userRepository. Therefore, we may experience random NullPointerExceptions here and there. Also, with constructor injection, it's easier to build immutable components.

Moreover, using constructors to create object instances is more natural from the OOP standpoint.

On the other hand, the main disadvantage of constructor injection is its verbosity especially when a bean has a handful of dependencies. Sometimes it can be a blessing in disguise, as we may try harder to keep the number of dependencies minimal.

# **[How does dependency injection work in Spring?](https://stackoverflow.com/questions/3725515/how-does-dependency-injection-work-in-spring)**

IoC is also known as dependency injection (DI).It is a process whereby objects define their dependencies, that is, the other objects they work with.

Dependency Injection is a fundamental aspect of the Spring framework, through which the Spring container "injects" objects into other objects or "dependencies".

Simply put, this allows for loose coupling of components and moves the responsibility of managing components onto the container.

The **org.springframework.beans** and **org.springframework.context** packages are the basis for Spring Framework's IoC container. The **[BeanFactory](http://static.springsource.org/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/BeanFactory.html" \t "https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/_top)**interface provides an advanced configuration mechanism capable of managing any type of object. **[ApplicationContext](http://static.springsource.org/spring-framework/docs/current/javadoc-api/org/springframework/context/ApplicationContext.html" \t "https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/_top)**is a sub-interface of BeanFactory. It adds easier integration with Spring's AOP features; message resource handling (for use in internationalization), event publication; and application-layer specific contexts such as the WebApplicationContext for use in web applications.

In short, the **BeanFactory**provides the configuration framework and basic functionality, and the ApplicationContext adds more enterprise-specific functionality.

In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. Otherwise, a bean is simply one of many objects in your application. Beans, and the dependencies among them, are reflected in the configuration metadata used by a container.

**Container overview :**

The interface org.springframework.context.**ApplicationContext**represents the Spring IoC container and is responsible for instantiating, configuring, and assembling the beans. The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata. The configuration metadata is represented in XML, Java annotations, or Java code.

The following diagram is a high-level view of how Spring works. Your application classes are combined with configuration metadata so that after the ApplicationContext is created and initialized, you have a fully configured and executable system or application.



the Spring IoC container consumes a form of configuration metadata; this configuration metadata represents how you as an application developer tell the Spring container to instantiate, configure, and assemble the objects in your application.

For information about using other forms of metadata with the Spring container :

Configuration metadata is traditionally supplied in a simple and intuitive **XML format**,

**[Annotation-based configuration](https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/beans.html" \l "beans-annotation-config" \o "5.9 Annotation-based container configuration):** Spring 2.5 introduced support for annotation-based configuration metadata.

**[Java-based configuration](https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/beans.html" \l "beans-java" \o "5.12 Java-based container configuration):** Starting with Spring 3.0, many features provided by the [Spring JavaConfig project](http://www.springsource.org/javaconfig" \t "https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/_top) became part of the core Spring Framework. Thus you can define beans external to your application classes by using Java rather than XML files. To use these new features, see the @Configuration, @Bean, @Import and @DependsOn annotations.

Spring configuration consists of at least one and typically more than one bean definition that the container must manage. XML-based configuration metadata shows these beans configured as **<bean/>**elements inside a top-level **<beans/>** element.

<bean id="..." class="...">

*<!-- collaborators and configuration for this bean go here -->*

</bean>

The **id**attribute is a string that you use to identify the individual bean definition. The **class**attribute defines the type of the bean and uses the fully qualified classname. The value of the id attribute refers to collaborating objects.

**What is Bean ?**

In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. Otherwise, a bean is simply one of many objects in your application.

**What is qualifier annotation?**

There may be a situation when you create more than one bean of the same type and want to wire only one of them with a property. In such cases, you can use the **@Qualifier** annotation along with **@Autowired** to remove the confusion by specifying which exact bean will be wired.

beans >

<context:annotation-config/>

<!-- Definition for profile bean -->

<bean id = "profile" class = "com.tutorialspoint.Profile"></bean>

<!-- Definition for student1 bean -->

<bean id = "student1" class = "com.tutorialspoint.Student">

<property name = "name" value = "Zara" />

<property name = "age" value = "11"/>

</bean>

<!-- Definition for student2 bean -->

<bean id = "student2" class = "com.tutorialspoint.Student">

<property name = "name" value = "Nuha" />

<property name = "age" value = "2"/>

</bean>

</beans>

public class MainApp {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

Profile profile = (Profile) context.getBean("profile");

profile.printAge();

profile.printName();

}

public class Profile {

@Autowired

@Qualifier("student1")

private Student student;

public class Student {

private Integer age;

private String name;

**// setter & getter**

The @Autowired annotation is a great way of making the need to inject a dependency in Spring explicit. And although it's useful, there are use cases for which this annotation alone isn't enough for Spring to understand which bean to inject.

By default, Spring resolves autowired entries by type.

If more than one bean of the same type is available in the container, the framework will throw NoUniqueBeanDefinitionException, indicating that more than one bean is available for autowiring.

**Let's imagine a situation :**

@Component("fooFormatter")

public class FooFormatter implements Formatter {

public String format() {

return "foo";

}

}

@Component("barFormatter")

public class BarFormatter implements Formatter {

public String format() {

return "bar";

}

}

@Component

public class FooService {

@Autowired

private Formatter formatter;

}

If we try to load FooService into our context, the Spring framework will throw a **NoUniqueBeanDefinitionException**. This is because Spring doesn't know which bean to inject. To avoid this problem, there are several solutions. The **@Qualifier** annotation is one of them.

By using the @Qualifier annotation, we can eliminate the issue of which bean needs to be injected.

Let's revisit our previous example and see how we solve the problem by including the @Qualifier annotation to indicate which bean we want to use:

public class FooService {

@Autowired

@Qualifier("fooFormatter")

private Formatter formatter;

}

Note that we could've also used the @Qualifier annotation on the Formatter implementing classes, instead of specifying the names in their @Component annotations, to obtain the same effect:

@Component

@Qualifier("fooFormatter")

public class FooFormatter implements Formatter {

//...

}

@Component

@Qualifier("barFormatter")

public class BarFormatter implements Formatter {

//...

}

There's another annotation called **@Primary** that we can use to decide which bean to inject when ambiguity is present regarding dependency injection.

This annotation defines a preference when multiple beans of the same type are present. The bean associated with the @Primary annotation will be used unless otherwise indicated.

@Configuration

public class Config {

@Bean

public Employee johnEmployee() {

return new Employee("John");

}

@Bean

@Primary

public Employee tonyEmployee() {

return new Employee("Tony");

}

}

In this example, both methods return the same Employee type. The bean that Spring will inject is the one returned by the method tonyEmployee. This is because it contains the @Primary annotation. This annotation is useful when we want to specify which bean of a certain type should be injected by default.

And in case we require the other bean at some injection point, we would need to specifically indicate it. We can do that via the @Qualifier annotation. For instance, we could specify that we want to use the bean returned by the johnEmployee method by using the @Qualifier annotation.

It's worth noting that if both the @Qualifier and @Primary annotations are present, then the @Qualifier annotation will have precedence. Basically, @Primary defines a default, while @Qualifier is very specific.

**Another way** to decide between multiple beans when autowiring is by using the name of the field to inject. This is the default in case there are no other hints for Spring.

Let's see some code based on our initial example:

public class FooService {

@Autowired

private Formatter fooFormatter;

}

In this case, Spring will determine that the bean to inject is the FooFormatter one since the field name is matched to the value that we used in the @Component annotation for that bean.

**Note**:

@Autowired can be used alone . If it is used alone , it will be wired by type . So problems arises if more than one bean of the same type are declared in the container as @Autowired does not know which beans to use to inject. As a result , use @Qualifier together with @Autowired to clarify which beans to be actually wired by specifying the bean name (wired by name)

@Resource is wired by name too . So if @Autowired is used together with @Qualifier , it is the same as the @Resource.

The difference are that @Autowired and @Qualifier are the spring annotation while @Resource is the standard java annotation (from JSR-250) . Besides , @Resource only supports for fields and setter injection while @Autowired supports fields , setter ,constructors and multi-argument methods injection.

It is suggested to use @Resource for fields and setter injection. Stick with @Qualifier and @Autowired for constructor or a multi-argument method injection.

**what is @resource in spring**

**How many objects of a servlet are created for multiple requests?**

Only one object is created during the lifetime of a servlet . When you start the server it reads your web.xml file or your annotations and loads the class into memory and creates the object for that servlet class

Now when a request comes to that see let url it creates a thread and that thread will process your request using the object which was created in your earlier step. Let’s say there are 30–40 requests coming to the servlet url server creates those many threads to process your requests. Again how many threads are created to handle those many requests is depending on the server your are using. Each server has its own threadpool from which it’s threads are used for handling the incoming requests. Once the request is processed the thread will go back to the thread pool.

If you want to test this print a System.out.println statement in the servlet init method and in doGet print the thread ID. Write a client program to access the servlet and fire 100 requests you will see the the system out statement is excuted only during the starting of the application and system out statement in the doGet might see some repeated thread ids

Servlets are managed resources in the servlet container, whenever a request comes for the servlet for the first time, servlet is loaded and instantiated and used for request processing.

A Servlet is instantiated only once in the container, and this Servelt object is used for any further request processing, be it another 15-20 different requests, this servlet object is shared among the different requests.

per-request new thread is created to handle the request not the Servlet object.