**Spring Framework :** Spring is an open-source framework. 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** is a fundamental aspect of the Spring framework, through which the Spring container "injects" objects into other objects or "dependencies".

this allows for loose coupling of components and moves the responsibility of managing components onto the container**.**

**Dependency Injection (DI)** is a design pattern means injecting the dependency between the two objects 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.

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.

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.

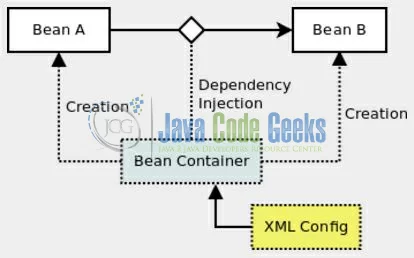
**Need for Dependency Injection:** Suppose class One needs the object of class Two to instantiate or operate a method, then class One is said to be dependent on class Two.

**[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)**

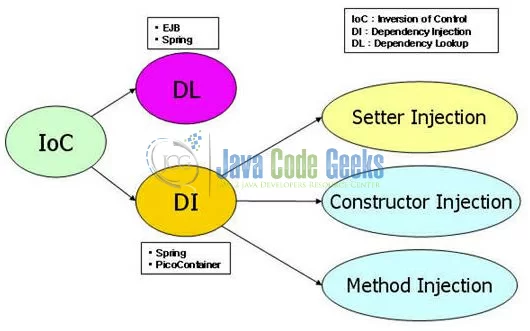


1. **Setter Dependency Injection (SDI):**

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.

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?**

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. 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.

### **[How autowiring works in Spring :](https://www.edureka.co/community/35377/how-autowiring-works-in-spring" \l ":~:text=Depends on whether you went,the bean XML definition route.&text=When it sees @Autowired, Spring,which one it should use.)**

<beans ...>

<bean id="userService" class="com.foo.UserServiceImpl"/>

<bean id="fooController" class="com.foo.FooController"/>

</beans>

The autowiring happens when the application starts up. So, in fooController, which for arguments , wants to use the UserServiceImpl class, you'd annotate it as follows:

public class FooController {

// You could also annotate the setUserService method instead of this

@Autowired

private UserService userService;

// rest of class goes here}

When it sees @Autowired, Spring will look for a class that matches the property in the applicationContext, and inject it automatically. If you have more than 1 UserService bean, then you'll have to qualify which one it should use.

First, and most important - all Spring beans are managed by inside a container, called "application context".

Second, each application has an entry point to that context. Web applications have a Servlet Also, there is a place where the application context is bootstrapped and all beans - autowired. In web applications this can be a startup listener.

Autowiring happens by placing an instance of one bean into the desired field in an instance of another bean. Both classes should be beans, i.e. they should be defined to live in the application context.

What is "living" in the application context? This means that the context instantiates the objects, not you. I.e. - you never make new UserServiceImpl() - the container finds each injection point and sets an instance there.

In your controllers, you just have the following:

@Controller // Defines that this class is a spring bean

@RequestMapping("/users")public class SomeController {

// Tells the application context to inject an instance of UserService here

@Autowired

private UserService userService;

@RequestMapping("/login")

public void login(@RequestParam("username") String username,

@RequestParam("password") String password) {

// The UserServiceImpl is already injected and you can use it

userService.login(username, password);

}}

A few notes:

> In your applicationContext.xml you should enable the <context:component-scan> so that classes are scanned for the @Controller, @Service, etc. annotations.

> The entry point for a Spring-MVC application is the DispatcherServlet, but it is hidden from you, and hence the direct interaction and bootstrapping of the application context happens behind the scene.

> UserServiceImpl should also be defined as bean - either using <bean id=".." class=".."> or using the @Service annotation. Since it will be the only implementor of UserService, it will be injected.

> Apart from the @Autowired annotation, Spring can use XML-configurable autowiring. In that case all fields that have a name or type that matches with an existing bean automatically get a bean injected. In fact, that was the initial idea of autowiring - to have fields injected with dependencies without any configuration. Other annotations like @Inject, @Resource can also be used.

**Autowiring In Spring :** Spring provides a way to automatically detect the relationships between various beans. This can be done by declaring all the bean dependencies in Spring configuration file.

The XML configuration based autowiring functionality has five modes – **no, byName, byType, constructor, and autodetect**. The default mode is **no**.

Spring supports the following **autowiring modes:**

**no:** It’s the default autowiring mode. It means no autowiring.

<bean id="department" class="guru.springframework.autowiringdemo.Department">

<property name="deptName" value="Information Technology" />

</bean>

<bean id="employee" class="guru.springframework.autowiringdemo.Employee"></bean>

**byName:** The byName mode injects the object dependency according to name of the bean. In such case, property name and bean name should be same. It internally calls setter method.

This option enables autowire based on bean names. Spring looks up the configuration file for a matching bean name. If found, this bean is injected in the property. However, if no such bean is found, an error is raised.

In this case, the name of the department bean is same as the employee bean’s property (Department), so Spring will be autowired to it via the setter method – setDepartment(Department department).

<bean id="department" class="guru.springframework.autowiringdemo.Department">

<property name="deptName" value="Information Technology" />

</bean>

<bean id="employee" class="guru.springframework.autowiringdemo.Employee" autowire="byName"></bean>

**byType:** The byType mode injects the object dependency according to type. So it can have different property name and bean name. It internally calls setter method.

<bean id="department" class="guru.springframework.autowiringdemo.Department">

<property name="deptName" value="Information Technology" />

</bean>

<bean id="employee" class="guru.springframework.autowiringdemo.Employee" autowire="byType"></bean>

**constructor:** The constructor mode injects the dependency by calling the constructor of the class. It calls the constructor having large number of parameters.

Autowiring by constructor is similar to byType but it applies to constructor arguments. It will look for the class type of constructor arguments, and then do an autowire byType on all constructor arguments. If exactly one bean of the constructor argument type is not present in the container, a fatal error will be raised.

The data type of department bean is same as the constructor argument data type in employee bean’s property (Department object). Therefore, Spring autowires it using the constructor method – public Employee(Department department).

<bean id="department" class="guru.springframework.autowiringdemo.Department">

<property name="deptName" value="Information Technology" />

</bean>

<bean id="employee" class="guru.springframework.autowiringdemo.Employee" autowire="constructor"></bean>

**autodetect:** In this mode, Spring first tries to autowire by constructor. If this fails, it tries to autowire by using byType.

Autowiring by autodetect uses two modes i.e. constructor or byType modes. First, it will look for valid constructor with arguments. If it is found then the constructor mode is chosen. If there is no constructor defined in a bean, the autowire byType mode is chosen.

In following case, since there is a Department object in the Employee class, Spring autowires it using byType via the setter method – setDepartment(Department department).

<bean id="department" class="guru.springframework.autowiringdemo.Department">

<property name="deptName" value="Information Technology" />

</bean>

<bean id="employee" class="guru.springframework.autowiringdemo.Employee" autowire="autodetect"></bean>

**Note :**Autodetect functionality will work with the 2.5 and 2.0 schemas. It will not work from 3.0+.

## **Example of Autowiring :** We’ll create a simple Java Bean, named Department. Department will have department name property with getter and setter methods. After that, we will initialize this property value in the Spring bean configuration file.

public class Department {

private String deptName;

public String getDeptName() {

return deptName;

}

public void setDeptName(String deptName) {

this.deptName = deptName;

}

}

Now let’s create our Employee class. In which we will inject Department bean through Spring autowiring.

public class Employee {

private int eid; private String ename; private Department department;

public int getEid() {

return eid;

}

public void setEid(int eid) {

this.eid = eid;

}

public String getEname() {

return ename;

}

public void setEname(String ename) {

this.ename = ename;

}

public Department getDepartment() {

return department;

}

public void setDepartment(Department department) {

this.department = department;

}

public void showEployeeDetails(){

System.out.println("Employee Id : " + eid);

System.out.println("Employee Name : " + ename);

System.out.println("Department : " + department.getDeptName());

}

}

Now, looking at Spring bean configuration file, it is the main part of any Spring application. So let’s see how our Spring bean configuration file looks.

<?xml version="1.0" encoding="UTF-8"?>

<beans .....>

<bean id="department" class="guru.springframework.autowiringdemo.Department">

<property name="deptName" value="Information Technology" />

</bean>

<bean id="emp" class="guru.springframework.autowiringdemo.Employee" autowire="byName"></bean>

</beans>

Now, our Spring application is ready with all types of Spring autowiring. So let’s write a simple test program to see if it works as expected.

@SpringBootApplication

public class AutowiringdemoApplication {

public static void main(String[] args) {

SpringApplication.run(AutowiringdemoApplication.class, args);

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

Employee emp = context.getBean("employee", Employee.class);

emp.setEid(101);

emp.setEname("Spring Framework Guru");

emp.showEployeeDetails();

}

}

In the above program, we are just creating the Spring application context and using it to get different beans and printing the employee details.

**@Autowired Annotation :** In Spring, you can use **@Autowired** annotation to auto wire bean on the setter method, constructor or a field. Moreover, it can autowire property in a particular bean. We must first enable the annotation using below configuration in configuration file.

If you are using **Java based configuration**, you can enable annotation-driven injection by using below spring configuration:

@Configuration

@ComponentScan("guru.springframework.autowiringdemo")

public class AppConfig {}

As an alternative, we can use below XML based configuration in Spring:

**<context:annotation-config />**

We have enabled annotation injection. After that, it can be used on modes like properties, setters, and constructors.

**Let’s discuss them one by one.**

### **@Autowired on Properties :** when the annotation is directly used on properties, Spring looks for and injects Department when Employee is created. This is how it eliminates the need for getters and setters.

import org.springframework.stereotype.Component;

@Component

public class Department {

private String deptName;

public String getDeptName() {

return deptName;

}

public void setDeptName(String deptName) {

this.deptName = deptName;

}

}

import org.springframework.beans.factory.annotation.Autowired;

public class Employee {

private int eid;

private String ename;

@Autowired

private Department department;

//setter & getter

public void showEployeeDetails(){

System.out.println("Employee Id : " + eid);

System.out.println("Employee Name : " + ename);

department.setDeptName("Information Technology");

System.out.println("Department : " + department.getDeptName());

}

}

**@Autowired on Setters :** In the below example, when the annotation is used on the setter method, the setter method is called with the instance of Department when Employee is created.

@Component

public class Employee {

private int eid;

private String ename;

private Department department;

// setter & getter method

@Autowired

public void setDepartment(Department department) {

this.department = department;

}

public void showEployeeDetails(){

System.out.println("Employee Id : " + eid);

System.out.println("Employee Name : " + ename);

department.setDeptName("Information Technology");

System.out.println("Department : " + department.getDeptName());

}}

**@Autowired on Constructors :**In the below example, the annotation is used on a constructor, an instance of Department is injected as an argument to the constructor when Employee is created.

@Component

public class Employee {

private int eid;

private String ename;

private Department department;

@Autowired

public EmployeeBean(DepartmentBean deptBean) {

System.out.println("\*\*\* Autowiring by using @Autowire annotation on constructor \*\*\*");

this.deptBean = deptBean;

}

// setter & getter

public void showEployeeDetails(){

System.out.println("Employee Id : " + eid);

System.out.println("Employee Name : " + ename);

department.setDeptName("Information Technology");

System.out.println("Department : " + department.getDeptName());

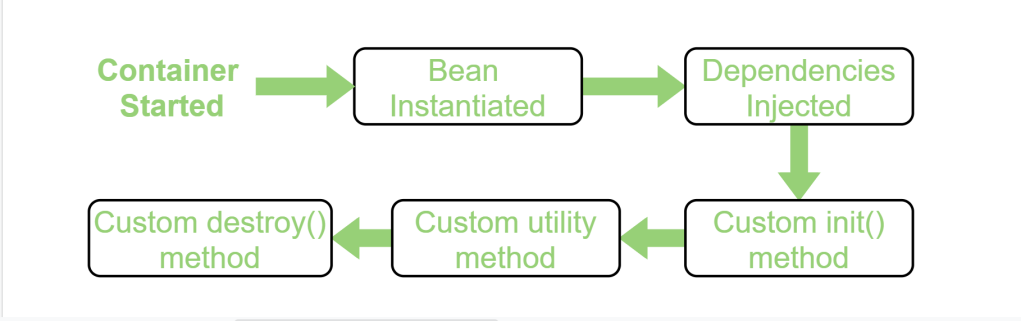
}

}

**Autowire Conflict Resolution :** By default, Spring resolves @Autowired entries byType. If more than one beans of the same type are available in the container, the framework will throw NoUniqueBeanDefinitionException exception indicating that more than one bean is available for autowiring.

**Bean life cycle in Java Spring :** The lifecycle of any object means when & how it is born, how it behaves throughout its life, and when & how it dies. Similarly, the bean life cycle refers to when & how the bean is instantiated, what action it performs until it lives, and when & how it is destroyed.

**Bean life cycle** is managed by the spring container. When we run the program first of all, the spring container gets started. After that, the container creates the instance of a bean as per the request and then dependencies are injected. And finally, the bean is destroyed when the spring container is closed. Therefore, if we want to execute some code on the bean instantiation and just after closing the spring container, then we can write that code inside the custom init() method and the destroy() method.



**Note:**We can choose custom method name instead of init() and destroy(). Here, we will use init() method to execute all its code as the spring container starts up and the bean is instantiated, and destroy() method to execute all its code on closing the container.

**Ways to implement the life cycle of a bean :**Spring provides three ways to implement the life cycle of a bean. In order to understand these three ways, let’s take an example. In this example, we will write and activate init() and destroy() method for our bean (HelloWorld.java) to print some message on start and close of Spring container. Therefore, the three ways to implement this are:

**By XML: In this approach,** in order to avail custom init() and destroy() method for a bean we have to register these two methods inside Spring XML configuration file while defining a bean.

1. Firstly, we need to create a bean HelloWorld.java in this case and write the init() and destroy() methods in the class.

public class HelloWorld {

    // This method executes

    // automatically as the bean

    // is instantiated

    public void init() throws Exception

    {

        System.out.println(

            "Bean HelloWorld has been "

            + "instantiated and I'm "

            + "the init() method");

    }

    // This method executes

    // when the spring container

    // is closed

    public void destroy() throws Exception

    {

        System.out.println(

            "Conatiner has been closed "

            + "and I'm the destroy() method");

    }

}

1. Now, we need to configure the spring XML file spring.xml and need to register the init() and destroy() methods in it.

<beans>

    <bean id="hw" class="beans.HelloWorld"

            init-method="init" destroy-method="destroy"/>

</beans>

1. Finally, we need to create a driver class to run this bean.

ConfigurableApplicationContext cap

            = new ClassPathXmlApplicationContext(

                "resources/spring.xml");

        // It will close the spring container

        // and as a result invokes the

        // destroy() method

        cap.close();

Bean HelloWorld has been instantiated and I’m the init() method  
Conatiner has been closed and I’m the destroy() method

**By Programmatic Approach:**To provide the facility to the created bean to invoke custom init() method on the startup of a spring container and to invoke the custom destroy() method on closing the container, we need to implement our bean with two interfaces namely InitializingBean, DisposableBean and will have to override afterPropertiesSet() and destroy() method. afterPropertiesSet() method is invoked as the container starts and the bean is instantiated whereas, the destroy() method is invoked just after container is closed.

Note: To invoke destroy method we have to call close() method of ConfigurableApplicationContext.

public class HelloWorld     implements InitializingBean,  DisposableBean {

      @Override

    // It is the init() method of our bean and it gets invoked on bean instantiation

    public void afterPropertiesSet()  throws Exception     {

        System.out.println(  "Bean HelloWorld has been "   + "instantiated and I'm the “ + "init() method");

    }

      @Override

    // This method is invoked just after the container is closed

    public void destroy() throws Exception     {

        System.out.println( "Conatiner has been closed "     + "and I'm the destroy() method");

    }

}

**Using Annotation:** To provide the facility to the created bean to invoke custom init() method on the startup of a spring container and to invoke the custom destroy() method on closing the container, we need annotate init() method by @PostConstruct annotation and destroy() method by @PreDestroy annotation.

Note: To invoke the destroy() method we have to call the close() method of ConfigurableApplicationContext.

public class HelloWorld {

    // Annotate this method to execute it automatically as the bean is instantiated

    @PostConstruct

    public void init() throws Exception     {

        System.out.println( "Bean HelloWorld has been " + "instantiated and I'm the " + "init() method");

    }

    // Annotate this method to execute it when Spring container is closed

    @PreDestroy

    public void destroy() throws Exception     {

        System.out.println(  "Conatiner has been closed "   + "and I'm the destroy() method");

    }

}

**There are five types of [spring bean](https://www.journaldev.com/2461/spring-ioc-bean-example-tutorial" \t "https://www.journaldev.com/21039/_blank) scopes:**

1. **singleton –** only one instance of the spring bean will be created for the spring container. This is the default spring bean scope. While using this scope, make sure bean doesn’t have shared instance variables otherwise it might lead to data inconsistency issues.
2. A single bean instance is created per IOC container and this is the default scope. **Real world example:** connection to a database
3. **prototype –** A new instance will be created every time the bean is requested from the spring container. If the scope is declared prototype, then spring IOC container will create a new instance of that bean every time a request is made for that specific bean.
4. A new bean instance is created each time the bean is requested from the IOC container.  **Real world example:** declare configured form elements (a textbox configured to validate names, e-mail addresses for example) and get "living" instances of them for every form being created

The non-singleton, prototype scope of bean deployment results in the creation of a new bean instance every time a request for that specific bean is made (that is, it is injected into another bean or it is requested via a programmatic getBean() method call on the container). As a rule of thumb, **you should use the prototype scope for all beans that are stateful, while the singleton scope should be used for stateless beans.**

The following diagram illustrates the Spring prototype scope. Please note that a DAO would not typically be configured as a prototype, since a typical DAO would not hold any conversational state; it was just easier for this author to reuse the core of the singleton diagram.

Prototype bean is created at the time of usage. So when you would like to have statefull beans there is strong need sometimes to have prototypes scope or when you don't wont to cache any values in beans. Prototype bean can be associated with one session or some call.

**Example:**A data access object (DAO) is not typically configured as a prototype, because a typical DAO does not hold any conversational state; it was just easier for this author to reuse the core of the singleton diagram.

<!-- using spring-beans-2.0.dtd -->

<bean id="accountService" class="com.foo.DefaultAccountService" scope="prototype"/>

<!-- the following is equivalent and preserved for backward compatibility in spring-beans.dtd -->

<bean id="accountService" class="com.foo.DefaultAccountService" singleton="false"/>

1. **request –** This is same as prototype scope, however it’s meant to be used for web applications. A new instance of the bean will be created for each HTTP request. A single bean instance is created and available during the lifecycle of the HTTP request. Only valid with a web-aware spring ApplicationContext container. **Real world example:** information that should only be valid on one page like the result of a search or the confirmation of an order. The bean will be valid until the page is reloaded.
2. **session –** A new bean will be created for each HTTP session by the container.A single bean instance is created and available during the lifecycle of the HTTP session. Only valid with a web-aware spring ApplicationContext container. Real world example: to hold authentication information getting invalidated when the session is closed (by timeout or logout). You can store other user information that you don't want to reload with every request here as well.
3. **global-session –** This is used to create global session beans for Portlet applications.A single bean instance is created and available during the lifecycle of the global HTTP session (i.e. for portlet environments). Only valid with a web-aware spring ApplicationContext container

@Configuration public class MyConfiguration {

@Bean

@Scope(value="singleton")

public MyBean myBean() {

return new MyBean();

} }

|  |  |
| --- | --- |
| **singleton** | **prototype** |
| Only one instance is created for a single bean definition per Spring IoC container | A new instance is created for a single bean definition every time a request is made for that bean. |
| Same object is shared for each request made for that bean. i.e. The same object is returned each time it is injected. | For each new request a new instance is created. i.e. A new object is created each time it is injected. |
| By default scope of a bean is singleton. So we don’t need to declare a been as singleton explicitly. | By default scope is not prototype so you have to decalre the scope of a been as prototype explicitly. |
| Singleton scope should be used for stateless beans. | While prototype scope is used for all beans that are stateful |
|  | Prototype scope in the spring framework creates a new instance of a bean, every time; a request for that specific bean is made. The Prototype scope is preferred for the stateful beans, and the spring container does not manage the complete lifecycle of a prototype bean i.e. destruction lifecycle methods are uncalled. Like so, a developer is responsible for cleaning up the prototype-scoped bean instances and any resources it holds. |

**What is stateful and stateless bean?**

**Stateless session beans** do not maintain state associated with any client. Each stateless session bean can server multiple clients.

**Stateful session beans** maintain the state associated with a client. Each stateful session bean serves exactly one client.

Stateless session beans are intended to be simple and lightweight; that is, they are easy to develop with low runtime resource requirements on the server. If required, any state is maintained by the client, and thereby makes the server highly scalable. Because no state is maintained in this enterprise bean type, stateless session beans aren't tied to any specific client. Therefore, any available instance of a stateless session bean can be used to service another client.

The container creates an implicit identity for a stateful session bean to manage its passivation and activation phases. On the other hand, the container doesn't create any identity for a stateless session bean.

The number of stateful session beans is equal to the number of active clients, whereas a small number of stateless session beans can be used to satisfy a large number of clients.

Stateful session beans provide easy and transparent state management on the server side. Because state is maintained in this enterprise bean type, the application server manages client-bean pairs. In other words, each instance of a given enterprise bean is created on behalf of a client, and is intended to be a private resource to that client (although it could be shared across clients using the enterprise bean instance's handle). In essence, a stateful session bean is a logical extension of the client, except that some of the client's load is distributed between itself and the enterprise bean on the server. Any conversational state-related data in the object's variables doesn't survive a server shutdown or crash, although a vendor could provide an enhanced implementation to make shutdowns and crashes transparent to the client by maintaining the enterprise bean's state.

**Explain the request flow and its lifecycle in Spring MVC :**

1. The request will be received by DispatcherServlet as the first step.
2. DispatcherServlet gets the help of HandlerMapping and maps the @Controller class associated with the given request to delegate the request.
3. So request gets transferred to the @Controller, and then @Controller will process the request by executing appropriate methods and returns ModeAndView object (contains both Model data and View name) back to the DispatcherServlet.
4. Now DispatcherServlet send the model object to the ViewResolver to resolve and retrieve the actual view page.
5. DispatcherServlet will pass the Model object to the View page to display the result and create the Response.
6. Finally DispatcherServlet sends the response back to the browser.\

**How does spring work internally?**

**The Spring IoC container** - It is the heart of the Spring Framework. The IoC container receives metadata from either an XML file, Java annotations, or Java code. The container gets its instructions on what objects to instantiate, configure, and assemble from simple Plain Old Java Objects (POJO) by reading the configuration metadata provided. These created objects through this process called **Spring Beans**.

We know that spring container or IOC container is at the core of spring framework.

**The Spring Container is responsible to:**

1. create the objects (bean)
2. wire the created objects together
3. configure the objects
4. manage the objects complete life cycle from creation till destruction.

Spring IOC container accepts two components as input :

1. Java POJO (Learn about pojo).
2. Configuration metadata in form of xml file, java based configurations, annotation or mixture of any of the three techniques.

The configuration metadata contains instruction for container about:

* Which bean objects to create?
* How to configure the beans?
* What is the scope for each beans?
* How to configure the created beans?

So we can see that spring container performs responsible tasks mentioned at the beginning by reading the instructions in configuration metadata.

In Spring, objects do not have responsibility of finding or creating their dependency that need to complete their task. Rather the Spring container takes the responsibility of giving the reference to the objects with whom they need to collaborate with.

# **POJO class in java :** POJO full form in java in 'Plain Old Java Object'.

The term POJO was first used by Martin Fowler, Josh Mackenzie and Rebecca Parsons.

POJO simply means a class that is not forced to implement any interface, extend any specific class, contain any pre-described annotation or follow any pattern due to forced restriction.

 Any POJO class has following properties:

* A *java POJO class* *should be accessible to all the classes*. It should be declared public.
* The variables declared in *java POJO class should be private*.
* A *java POJO class should contain* a default constructor.
* A*java POJO class must provide getter and setters*which should be declared public to let outer world interact with it easily.

**Design Patterns used in Spring Framework :** Following are the design patterns used in Spring Framework .

**MVC Pattern :** MVC Design Pattern is a software design that separates the following components of a system or subsystem:

* **Model -** Data about the state of the application or its components. May include routines for modification or access.
* **View -** An interpretation of the data (model). This is only limited to a visual representation, but could be audio, derived information (e.g. statistics piped into another model object), etc. Furthermore, a single model may have multiple views.
* **Controller -** Handles external input to the system invoking modifications on the model. The control/view may be closely related (in the case of a UI). However, other external input (such as network commands), may be processed which are completely independent of the view.

**Proxy Pattern :** Spring uses either JDK proxies (preferred wheneven the proxied target implements at least one interface) or CGLIB proxies (if the target object does not implement any interfaces) to create the proxy for a given target bean. Unless configured to do otherwise, Spring AOP performs run-time weaving Suppose we want to log every method entry and exit. This can be achieved by writing log statements in every method at the start and end. But this will require lot of code work. There are various such tasks like Security which need to be applied across all methods or classes. These are known as cross cutting concerns.AOP addresses the problem of cross-cutting concerns, which would be any kind of code that is repeated in different methods and cannot normally be completely refactored into its own module, like with logging or verification.

**Factory Pattern :** This patterns is used by spring to load beans using BeanFactory and Application context.

**Singleton Pattern :** Beans defined in spring config files are singletons by default. A singleton bean in Spring and the singleton pattern are quite different. Singleton pattern says that one and only one instance of a particular class will ever be created per classloader. The scope of a Spring singleton is described as "per container per bean". It is the scope of bean definition to a single object instance per Spring IoC container. The default scope in Spring is Singleton.

**Template method Pattern :** Template method design pattern is to define an algorithm as skeleton of operations and leave the details to be implemented by the child classes. The overall structure and sequence of the algorithm is preserved by the parent class. These are used extensively to deal with boilerplate repeated code.

**FrontController Pattern :** Front Controller is a controller pattern which provides a centralized controller for managing requests. Each client request must go through and be processed by the Front Controller first, no exceptions. All incoming data is delegated to front controller first. Useful for when your application has multiple entry points which you want to centralize through a single point for standardized processing.  
Spring implements this design pattern using DispatcherServlet, to dispatch incoming requests to the correct controllers.

**View Helper Pattern :** View Helper arranges view components for the user and delegates processing to other business components so the view component doesn't have to contain any processing logic other than logic to present views.  
Spring makes use of custom JSP tags etc to separate code from presentation in views.

**Prototype Pattern :** The Prototype pattern is known as a creational pattern,as it is used to construct objects such that they can be decoupled from their implementing systems.  
It creates objects based on a template of an exsiting object through cloning.

**DI/IOC Pattern :** Dependency Injection/Inversion of Control design pattern allows us to remove the hard-coded dependencies and make our application loosely coupled, extendable and maintainable.  
We can implement dependency injection in java to move the dependency resolution from compile-time to runtime.

**@Component vs @Repository vs @Service in Spring :**With Spring’s auto-scanning feature, it automatically detects various beans defined in our application. We usually annotate our beans using one of the available Spring annotations – @Component, @Repository, @Service, @Controller.On detecting the bean, Spring simply registers it into the ApplicationContext.

**@Component:** We can use @Component annotation to mark a bean as a Spring-managed component. In other words, it’s a generic stereotype for any Spring-managed component.

We can enable an auto-scan using <context:component-scan> tag. During auto-scan, Spring will scan and register all beans marked with a @Component annotation:

@Component

public class Employee {

   ...

}

**@Repository:**@Repository annotation is a specialization over @Component annotation:

@Component

public @interface Repository {

}

Since @Repository is a type of @Component, Spring also auto-scans and registers them.

**@Repository** is a stereotype for the persistence layer. Its job is to catch all persistence related exceptions and rethrow them as a Spring DataAccessException.

For this, we should configure PersistenceExceptionTranslationPostProcessor in our application context:

**<bean class=**

**"org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor"/>**

This bean post processor adds an advisor to all beans marked with @Repository. The advisor’s responsibility is to translate the platform-specific exceptions to the Spring’s unified unchecked exceptions.

**@Service:**

Just like @Repository, @Service is another specialization of @Component:

@Component

public @interface Service {

}

Just like @Repository, @Service is also a type of @Component. That means Spring will also automatically detect such beans.

The @Service annotation represents that our bean holds some business logic. Till date, it doesn’t provide any specific behavior over @Component.

Still, we should annotate the service-layer beans with the @Service annotation to make our intent clear. Additionally, we never know if someday Spring chooses to add some specific functionality to it.

* @Component is the most generic stereotype and marks a bean as a Spring-managed component
* Both @Service and @Repository annotations are the specializations over the @Component annotation
* @Repository is a stereotype used for persistence layer. It translates any persistence related exceptions into a Spring’s DataAccessException
* @Service is used for the beans at the service layer. Currently, it doesn’t offer any additional functionality over @Component
* It’s always preferable to use @Repository and @Service annotations over @Component, wherever applicable. It communicates the bean’s intent more clearly

<https://www.javacodegeeks.com/2017/11/difference-component-service-controller-repository-spring.html>