**Spring container**

The **Spring container** is at the core of the **Spring** Framework. The **container** will create the objects, wire them together, configure them, and manage their complete life cycle from creation till destruction. The **Spring container** uses DI to manage the components that make up an application.

Spring IOC, also known as Inversion of Control, is a design pattern that allows objects to be created and managed without having to know about their dependencies. This is done by using a container, which is responsible for creating and managing the objects. The container injects the dependencies into the objects when they are created, so that the objects do not need to know about how to get their dependencies.

There are several benefits to using Spring IOC. First, it makes the code more modular and reusable. Second, it makes the code easier to test. Third, it makes the code more scalable.

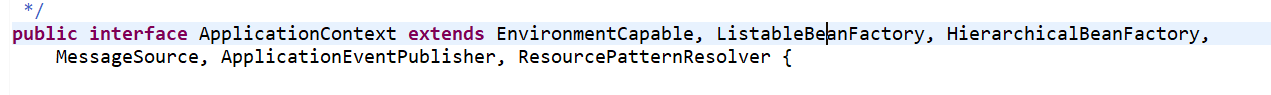
It is similar how servlet container does .Servlet contain reads the web.xml and instantiate the servlets defined in the web.xml file.

Spring uses the BeanFactory which uses the Factory design pattern to create spring beans from provided xml where it is defined how spring should create the objects and creates the object and provides it back.

BeanFactory factory=new XmlBeanFactory(new FileSystemResource(“Beans.xml”));

// ApplicationContex does everything that BeanFactory does and more than that.

ApplicationContex context=new ClassPathXmlApplicationContext(new FileSystemResource(“Beans.xml”));





A blue rectangle with white text

Description automatically generated

ApplicationContext is sub interface of BeanFactory



This is specifically for web applications.

* **BeanFactory:** Does not support the Annotation based dependency Injection.
* **ApplicationContext:** Support Annotation based dependency Injection. -@Autowired, @PreDestroy
* **BeanFactory:** Does not Support.
* **ApplicationContext:** Application contexts can publish events to beans that are registered as listeners.
* **BeanFactory:** Does not support way to access Message Bundle (internationalization (I18N)
* **ApplicationContext:** Support internationalization (I18N) messages.
* **BeanFactory:** Doesn’t support.
* **ApplicationContext:** Support many enterprise services such JNDI access, EJB integration, remoting.
* **BeanFactory:** By default, its support Lazy loading
* **ApplicationContext:** It's By default support Aggressive loading.
* Easier integration with Spring’s AOP features
* Message resource handling (for use in internationalization)
* Event publication
* Application-layer specific contexts such as the WebApplicationContext for use in web applications.

JNDI

JNDI stands for Java Naming and Directory Interface. It's an application programming interface (API) that provides directory and naming functionality for applications written using the Java programming language.

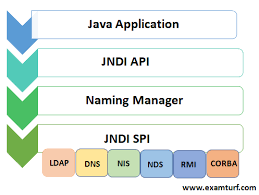
JNDI is still used in some Java technologies, including JDBC, EJB, and JMS. These technologies are used in many Java enterprise applications.

JNDI is mainly used to map resources to names, similar to storing configurations. It's not used as much in lightweight, containerized Java applications like Spring Boot.

A JNDI lookup is a way to find an object by name. JNDI organizes names into a hierarchy, where a name can be any string. For example, "com.example. ejb.MyBean" is a name.

Naming services, such as DNS and LDAP, maintain bindings that relate names to objects. These bindings allow objects to be looked up by name.

JNDI is the acronym for the Java Naming and Directory Interface API. By making calls to this API, applications locate resources and other program objects. A resource is a program object that provides connections to systems, such as database servers and messaging systems.



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | [enter image description here](https://i.stack.imgur.com/00grD.jpg)   |  |  |  | | --- | --- | --- | |  | XMLBeanFactory | ApplicationContext | | Annotation support | No | Yes | | BeanPostProcessor Registration | Manual | Automatic | | Implementation | XMLBeanFactory | ClassPath/FileSystem/WebXmlApplicationContext | | Internationalization | No | Yes | | Enterprise services | No | Yes(EJB , JNDI,JMS) | | ApplicationEvent publication | No | Yes | |  |  |  |  1. FileSystemXmlApplicationContext: Beans loaded through the full path. 2. ClassPathXmlApplicationContext: Beans loaded through the CLASSPATH 3. WebXmlApplicationContext: Beans loaded through the web application context.   **Lazy Loading vs. Eager Loading**  BeanFactory loads beans on-demand, while ApplicationContext loads all beans at startup. Thus, BeanFactory is lightweight as compared to ApplicationContext.  public class Student {  public static boolean isBeanInstantiated = false;  public void postConstruct() {  setBeanInstantiated(true);  }  //standard setters and getters  }  postConstruct() method as the init-method in our BeanFactory configuration file, ioc-container-difference-example.xml:  **ioc-container-difference-example.xml**  <bean id="student" class="com.baeldung.ioccontainer.bean.Student" init-method="postConstruct"/>  @Test  public void whenBFInitialized\_thenStudentNotInitialized() {  Resource res = new ClassPathResource("ioc-container-difference-example.xml");  BeanFactory factory = new XmlBeanFactory(res);  assertFalse(Student.isBeanInstantiated());  }  Student object is not initialized. In other words, only the BeanFactory is initialized. The beans defined in our BeanFactory will be loaded only when we explicitly call the getBean() method.  Let’s check the initialization of our Student bean where we’re manually calling the getBean() method.  @Test  public void whenBFInitialized\_thenStudentInitialized() {  Resource res = new ClassPathResource("ioc-container-difference-example.xml");  BeanFactory factory = new XmlBeanFactory(res);  Student student = (Student) factory.getBean("student");  assertTrue(Student.isBeanInstantiated());  }  ApplicationContext, and it will load all the beans instantly by using an eager-loading strategy:  @Test  public void whenAppContInitialized\_thenStudentInitialized() {  ApplicationContext context = new ClassPathXmlApplicationContext("ioc-container-difference-example.xml")  assertTrue(Student.isBeanInstantiated());  }  Here, the Student object is created even though we have not called the getBean() method.  ApplicationContext is considered a heavy IOC container because its eager-loading strategy loads all the beans at startup. BeanFactory is lightweight by comparison and could be handy in memory-constrained systems. |

**Injecting Bean Using BeanFactory**

BeanFactory factory=new XmlBeanFactory (new FileSystemResource(“Beans.xml”));

Triangle triangle =( Triangle)factory.getBean(“rectangle”);

On triangle object we can call the methods defined inside.

**Triangle Class**

class Triangle{

private String type;

//getters and setters

}

**Beans.xml**

<beans>

<bean id=”triangle” class=”com.xyz.Triangle”>

<property name=”type” value=”Equilateral Triangle” />

</bean>

</beans>

**Injecting Bean using ApplicationContex**

**Main Class**

ApplicationContex context=new ClassPathXmlApplicationContext(new FileSystemResource(“Beans.xml”));

Triangle triangle =(Triangle)context.getBean(“triangle”);

triangle.draw();

**Trangle Class**

Class Trangle{

priivate Point pontA;

priivate Point pontB;

priivate Point pontC;

//getters and setters

}

**Point Class**

class Point{

private int x;

private int y;

//getters and setters

}

// in this case Triangle bean is dependent on Point bean

**beans.xml**

<beans>

<bean id=”triangle” class=”com.xyz.Triangle”>

<property name=”pointA” ref=”point1” />

<property name=”pointB” ref=”point2” />

<property name=”pointC” ref=”point3” />

</bean>

<bean id=”point1” class=”com.xyz.Point”>

<property name=”x” value=”0”/>

<property name=”y” value=”0”/>

</bean>

<bean id=”point2” class=”com.xyz.Point”>

<property name=”x” value=”20”/>

<property name=”y” value=”30”/>

</bean>

<bean id=”point3” class=”com.xyz.Point”>

<property name=”x” value=”0”/>

<property name=”y” value=”40”/>

</bean>

</beans>

**Instantiating Bean using Constructor Way and Property way**

Class Trangle{

priivate int height;

priivate String type;

public Triangle(int height, int type){

this.height=height;

this.type=type;

}

//getters and setters

}

**beans.xml**

<beans>

<bean id=”triangle” class=”com.xyz.Triangle”>

<property name=”height” ref=”10” />

<property name=”type” ref=”Equilateral” />

</bean>

</beans>

In the above approach we are instantiating the class Trangle using property’s setter method.

**Constructor approch**

Class Trangle{

priivate int height;

priivate String type;

//getters and setters

}

<beans>

<bean id=”triangle” class=”com.xyz.Triangle”>

<construtor-arg type =” int” name=”height” ref=”10” />

<construtor-arg type=”lava.lang.String” name=”type” ref=”Equilateral” />

</bean>

</beans>

//for overloaded constructor we have to give type of argument

Another approch is index number

<beans>

<bean id=”triangle” class=”com.xyz.Triangle”>

<construtor-arg index =” 0” name=”height” ref=”10” />

<construtor-arg inex=”1” name=”type” ref=”Equilateral” />

</bean>

</beans>

**InnerBean**

Inner is used in those cases if the bean is going to be used only once and should be part of one parent bean.

In that case the inner bean does not have to have a id and the the parent property should not have a ref .

**beans.xml**

<beans>

<bean id=”triangle” class=”com.xyz.Triangle” name=”triangle-name”>

<property name=”pointA” >

<bean class=”com.xyz.Point”>

<property name=”x” value=”0”/>

<property name=”y” value=”0”/>

</bean>

</property>

<property name=”pointB” >

<bean class=”com.xyz.Point”>

<property name=”x” value=”20”/>

<property name=”y” value=”30”/>

</bean>

</property>

<property name=”pointC” ref=”point3” />

</bean>

<bean id=”point3” class=”com.xyz.Point”>

<property name=”x” value=”0”/>

<property name=”y” value=”40”/>

</bean>

<alias name=”triangle” alias=”triangle-alias” />

</beans>

BeanFactory factory=new XmlBeanFactory(new FileSystemResource(“Beans.xml”));

Triangle triangle =(Triangle)context.getBean(“triangle”);

Triangle triangle =(Triangle)context.getBean(“triangle-alias”);

Triangle triangle =(Triangle)context.getBean(“triangle-name”);

//both are same

**beans.xml**

<beans>

<bean id=”triangle” class=”com.xyz.Triangle” name=”triangle-name”>

<property name=”pointA” >

<bean class=”com.xyz.Point”>

<property name=”x” value=”0”/>

<property name=”y” value=”0”/>

</bean>

</property>

<property name=”pointB” >

<bean class=”com.xyz.Point”>

<property name=”x” value=”20”/>

<property name=”y” value=”30”/>

</bean>

</property>

<property name=”pointC” >

<idref= “point3”/> // this line will check any bean is there with id point3 in the xml

</property>

</bean>

<bean id=”point3” class=”com.xyz.Point”>

<property name=”x” value=”0”/>

<property name=”y” value=”40”/>

</bean>

<alias name=”triangle” alias=”triangle-alias” />

</beans>

**Collection of Bean**

Class Trangle{

priivate List<Point> points;

//getters and setters

}

**Point Class**

class Point{

private int x;

private int y;

//getters and setters

}

// in this case Triangle bean is dependent on Point bean

**beans.xml**

<beans>

<bean id=”triangle” class=”com.xyz.Triangle”>

<property name=”points” >

<list>

<ref bean=” point1” />

<ref bean=” point2” />

<ref bean=” point3” />

</list>

</property>

</bean>

<bean id=”point1” class=”com.xyz.Point”>

<property name=”x” value=”0”/>

<property name=”y” value=”0”/>

</bean>

<bean id=”point2” class=”com.xyz.Point”>

<property name=”x” value=”20”/>

<property name=”y” value=”30”/>

</bean>

<bean id=”point3” class=”com.xyz.Point”>

<property name=”x” value=”0”/>

<property name=”y” value=”40”/>

</bean>

</beans>

// for set and map we can use similarly

**Customer Class**

public class Customer

{

private List<Object> lists;

private Set<Object> sets;

private Map<Object, Object> maps;

private Properties pros;

//...

}

**Beans.xml**

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

<bean id="CustomerBean" class="com.mkyong.common.Customer">

<!-- java.util.List -->

<property name="lists">

<list>

<value>1</value>

<ref bean="PersonBean" />

<bean class="com.mkyong.common.Person">

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

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

<property name="age" value="28" />

</bean>

</list>

</property>

<!-- java.util.Set -->

<property name="sets">

<set>

<value>1</value>

<ref bean="PersonBean" />

<bean class="com.mkyong.common.Person">

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

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

<property name="age" value="28" />

</bean>

</set>

</property>

<!-- java.util.Map -->

<property name="maps">

<map>

<entry key="Key 1" value="1" />

<entry key="Key 2" value-ref="PersonBean" />

<entry key="Key 3">

<bean class="com.mkyong.common.Person">

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

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

<property name="age" value="28" />

</bean>

</entry>

</map>

</property>

<!-- java.util.Properties -->

<property name="pros">

<props>

<prop key="admin">admin@nospam.com</prop>

<prop key="support">support@nospam.com</prop>

</props>

</property>

</bean>

<bean id="PersonBean" class="com.mkyong.common.Person">

<property name="name" value="mkyong1" />

<property name="address" value="address 1" />

<property name="age" value="28" />

</bean>

</beans>

**App Class**

public class App

{

public static void main( String[] args )

{

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

Customer cust = (Customer)context.getBean("CustomerBean");

System.out.println(cust);

}

}

**Autowiring Of Beans**

If we apply autowire attribute to byName in triangle bean then spring will search the bean.xml where any bean is declared with id same as property of Triangle class property name.

Not a good practice.

Other possible values

byName

byType 🡪 one type per bean (two property of same type not allowed)

constructor 🡪 one type per bean (two property of same type not allowed)

**Triangle Class**

Class Trangle{

priivate Point pontA;

priivate Point pontB;

priivate Point pontC;

//getters and setters

}

**Point Class**

class Point{

private int x;

private int y;

//getters and setters

}

**Beans.xml**

<beans>

<bean id=”triangle” class=”com.xyz.Triangle” autowire=”byName”>

</bean>

<bean id=”pointA” class=”com.xyz.Point”>

<property name=”x” value=”0”/>

<property name=”y” value=”0”/>

</bean>

<bean id=”pointB” class=”com.xyz.Point”>

<property name=”x” value=”20”/>

<property name=”y” value=”30”/>

</bean>

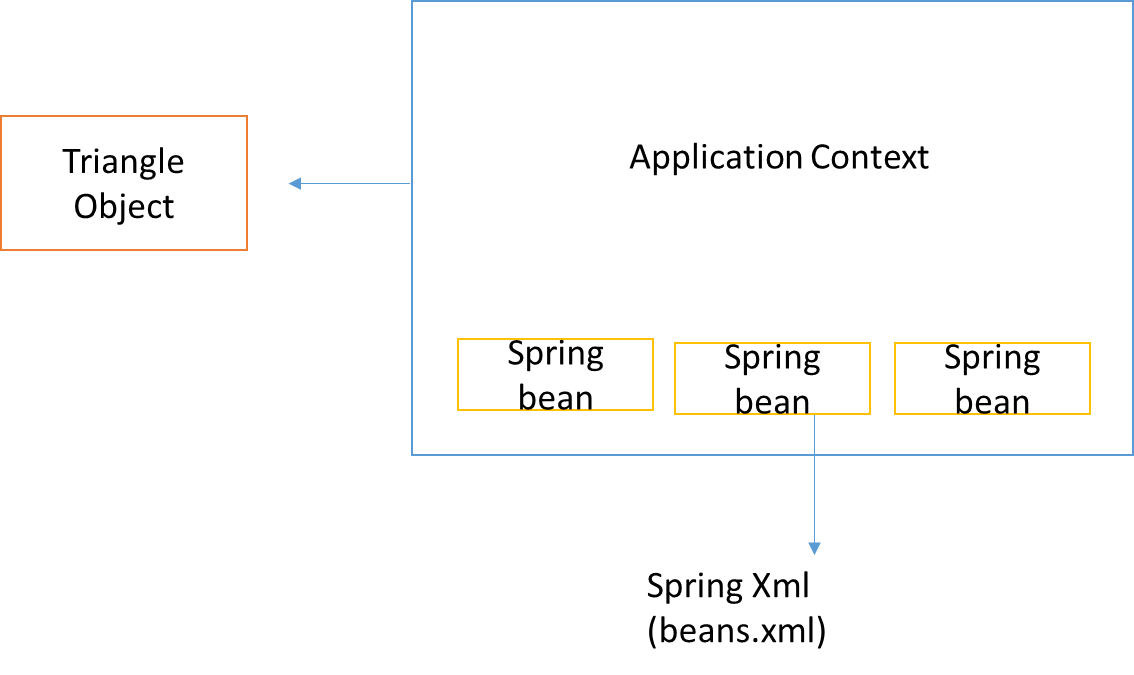
<bean id=”pointC” class=”com.xyz.Point”>

<property name=”x” value=”0”/>

<property name=”y” value=”40”/>

</bean>

</beans>



As soon as application context object is initialized the application context reads the file and creates the beans even before getBean method is called. This is default behavior of application context.

That’s why The default behavior of every bean is singleton.ie even if we call getBean multiple times it will return the same object that got created during initialization of application context.

It is possible multiple context running on same jvm in that case singleton means per context one object will created.

Beans can be of two types

1. Singleton 🡪 only once created
2. Prototype 🡪 every time getBean creates a new bean
3. Request 🡪 New Bean per servlet request
4. Session 🡪 New Bean per session.
5. Global Session 🡪New bean per global HTTP session (portlet context)

<beans>

<bean id=”triangle” class=”com.xyz.Triangle” scope=”singleton/prototype/request/session”>

-----------------------------

</bean>

</beans>

**Spring bean Definition Inheritance**

triangle1 and triangle2 inherit the pointA from the parent triangle

abstract=”true” will stop application creating bean for parenttriangle

**beans.xml**

<beans>

<bean id=”parenttriangle” class=”com.xyz.Triangle” abstract=”true”>

<property name=”pointA” ref=”point1” />

</bean>

<bean id=”triangle1” class=”com.xyz.Triangle” parent=”parenttriangle”>

<property name=”pointB” ref=”point2” />

<property name=”pointC” ref=”point3” />

</bean>

<bean id=”triangle2” class=”com.xyz.Triangle” parent=”parenttriangle”>

<property name=”pointB” ref=”point2” />

<property name=”pointC” ref=”point3” />

</bean>

<bean id=”point1” class=”com.xyz.Point”>

<property name=”x” value=”0”/>

<property name=”y” value=”0”/>

</bean>

<bean id=”point2” class=”com.xyz.Point”>

<property name=”x” value=”20”/>

<property name=”y” value=”30”/>

</bean>

<bean id=”point3” class=”com.xyz.Point”>

<property name=”x” value=”0”/>

<property name=”y” value=”40”/>

</bean>

</beans>

We can inherit the collection, we can override a collection defined in parent bean we can merge also.

**beans.xml**

<beans>

<bean id=”parenttriangle” class=”com.xyz.Triangle” abstract=”true”>

<property name=”points” > // points is a list type property in triangle bean

<list>

<ref bean=”point1”/>

</list>

</property>

</bean>

<bean id=”triangle1” class=”com.xyz.Triangle” parent=”parenttriangle”>

<property name=”points” >

<list merge=”true”> // if merge is not true child bean will override the parent points

<ref bean=”point1”/>

</list>

</property>

</bean>

<bean id=”point1” class=”com.xyz.Point”>

<property name=”x” value=”0”/>

<property name=”y” value=”0”/>

</bean>

<bean id=”point2” class=”com.xyz.Point”>

<property name=”x” value=”20”/>

<property name=”y” value=”30”/>

</bean>

<bean id=”point3” class=”com.xyz.Point”>

<property name=”x” value=”0”/>

<property name=”y” value=”40”/>

</bean>

</beans>

**ApplicationContextAware**

By implementing ApplicationContextAware we have implement below method in our bean

@override

public void setApplicationContext(ApplicationContext context) throws BeansException

{

this.context = context;

}

This mean we have context in our bean .So in case where triangle object is of type singleton mentioned in beans.xml file but pointA, pointB, pointC are declared as prototype. When triangle object will be created application context will return the same object again and again even the inner objects points will the same even if those are declared as prototype.

To created new pointA, pointB, pointC objects every time triangle bean is created we need the context inside the triangle object .we can read the bean. Xml file to create pointA, pointB, pointC.

**BeanNameAware**

If Bean class implements this interface we can get the bean name defined in the beans.xml file.

@override

Public void setBeanName (String name)

{

}

**Close Application Context**

If we are writing desktop application we need to close the application context.

We have to t

Public static void main{

AbstractApplicationContex context=new ClassPathXmlApplicationContext(new FileSystemResource(“Beans.xml”));

context.registerShutdownHook();

Triangle triangle =( Triangle)factory.getBean(“rectangle”);

Triangle.draw();

}

To know once object is initialized and when object is destroyed below interfaces are used

InitializedBean 🡪 afterPropertiesSet() method we need to override in bean

DisposableBean 🡪 destroy() method we need to override in bean

Without implementing InitializedBean interface we can call on user defined init method

By mentioning the below property in the bean’s declaration in xml and putting the mentioned method in the bean. Similarly for DisposableBean.

<bean id=”triangle” init-method=”myInit” destroy-method=”myDestroy”>

</bean>

To add init and destroy method at global level we need to use below property in <beans> tag of xml.These methods will be called for if it exists in the bean otherwise will be ignored.

<beans default-init-method=” myInit” default-destroy-method=” myDestroy”>

</beans>

**Bean post processors**

BeanPostProcessor is used to interact with newly created bean instances *before* and/or *after* their initialization method is invoked by the Spring container. You can use BeanPostProcessor to execute custom logic *before* and/or *after* bean’s initialization method is invoked by the Spring container.

BeanPostProcessor interface defines the following methods:

* Object postProcessBeforeInitialization(Object bean, String beanName) – this method is invoked *before* the initialization method of a bean instance is invoked
* Object postProcessAfterInitialization(Object bean, String beanName) – this method is invoked *after* the initialization method of a bean instance is invoked

BeanPostProcessor’s methods accept newly created bean instance and its name as arguments, and return the same or modified bean instance. You configure a BeanPostProcessor implementation in the application context XML file like any other Spring bean. Once the BeanPostProcessor beans are created, the Spring container invokes each BeanPostProcessor’s postProcessBeforeInitialization and postProcessAfterInitialization methods for each bean instance created by the Spring container.

### BeanPostProcessor example – Validating bean instances

In a Spring application, you may want to verify that a bean instance is configured correctly before it is injected into dependent beans or accessed by other objects in the application. Let’s see how we can use a BeanPostProcessor implementation to give an opportunity to each bean instance to validate its configuration before the bean instance is made available to dependent beans or other application objects.

The following example listing shows an InstanceValidator interface that must be implemented by beans whose configuration we want to validate using a BeanPostProcessor implementation:

**Example listing**– InstanceValidator interface

package sample.spring.chapter04.springbankapp.common;

public interface InstanceValidator {

  void validateInstance();

}

InstanceValidator interface defines a validateInstance method that verifies whether the bean instance was correctly initialized or not. We’ll soon see that the validateInstance method is invoked by a BeanPostProcessor implementation.

The following example listing shows the FixedDepositDaoImpl class that implements InstanceValidator interface:

**Example listing** – FixedDepositDaoImpl class

package sample.spring.chapter04.springbankapp.dao;

import org.apache.log4j.Logger;

import sample.spring.chapter04.springbankapp.common.InstanceValidator;

public class FixedDepositDaoImpl implements FixedDepositDao, InstanceValidator {

  private static Logger logger = Logger.getLogger(FixedDepositDaoImpl.class);

  private DatabaseConnection connection;

  public FixedDepositDaoImpl() {

    logger.info("FixedDepositDaoImpl's constructor invoked");

  }

  public void initializeDbConnection() {

  logger.info("FixedDepositDaoImpl's initializeDbConnection method invoked");

   connection = DatabaseConnection.getInstance();

  }

  @Override

  public void validateInstance() {

    logger.info("Validating FixedDepositDaoImpl instance");

     if(connection == null) {

     logger.error("Failed to obtain DatabaseConnection instance");

   }

  }

}

In the above example listing, the initializeDbConnection method is the initialization method that retrieves an instance of DatabaseConnection by calling getInstance *static* method of DatabaseConnection class. The connection attribute is null if FixedDepositDaoImpl instance fails to retrieve an instance of DatabaseConnection. If connection attribute is null, the validateInstance method logs an error message indicating that the FixedDepositDaoImpl instance is not correctly initialized. As the initializeDbConnection initialization method sets the value of connection attribute, the validateInstance method *must* be invoked *after* the initializeDbConnection method. In a real world application development scenario, if a bean instance is not configured correctly, the validateInstance method may take some corrective action or throw a runtime exception to stop the application from starting up. For simplicity, the validateInstance method logs an error message if a bean instance is not configured correctly.

The following example listing shows the InstanceValidationBeanPostProcessor class that implements Spring’s BeanPostProcessor interface, and is responsible for invoking validateInstance method of newly created beans:

**Example listing**– InstanceValidationBeanPostProcessor class

package sample.spring.chapter04.springbankapp.postprocessor;

import org.springframework.beans.BeansException;

import org.springframework.beans.factory.config.BeanPostProcessor;

import org.springframework.core.Ordered;

public class InstanceValidationBeanPostProcessor implements BeanPostProcessor, Ordered {

    private static Logger logger = Logger.getLogger(InstanceValidationBeanPostProcessor.class);

    private int order;

    public InstanceValidationBeanPostProcessor() {

        logger.info("Created InstanceValidationBeanPostProcessor instance");

    }

    @Override

    public Object postProcessBeforeInitialization(Object bean, String beanName)

            throws BeansException {

        logger.info("postProcessBeforeInitialization method invoked");

        return bean;

    }

    @Override

    public Object postProcessAfterInitialization(Object bean, String beanName)

            throws BeansException {

        logger.info("postProcessAfterInitialization method invoked");

        if (bean instanceof InstanceValidator) {

            ((InstanceValidator) bean).validateInstance();

        }

        return bean;

    }

    public void setOrder(int order) {

        this.order = order;

    }

    @Override

    public int getOrder() {

        return order;

    }

}

The above example listing shows that the InstanceValidationBeanPostProcessor class implements Spring’s BeanPostProcessor and Ordered interfaces. The postProcessBeforeInitialization method simply returns the bean instance passed to the method. In the postProcessAfterInitialization method, if the bean instance is found to be of type InstanceValidator, the bean instance’s validateInstance method is invoked. This means that if a bean implements InstanceValidator interface, InstanceValidationBeanPostProcessor calls validateInstance method of the bean instance *after* the initialization method of the bean instance is invoked by the Spring container.

The Ordered interface defines a getOrder method which returns an integer value. The integer value returned by the getOrder method determines the priority of a BeanPostProcessor implementation with respect to other BeanPostProcessor implementations configured in the application context XML file. A BeanPostProcessor with *higher* order value is considered at a *lower* priority, and is executed *after* the BeanPostProcessor implementations with *lower* order values are executed.

The following example listing shows bean definitions for InstanceValidationBeanPostProcessor class:

**Example listing**– InstanceValidationBeanPostProcessor bean definition

<bean class="…...springbankapp.postprocessor.InstanceValidationBeanPostProcessor">

  <property name="order" value="1" />

</bean>

In the above bean definition, <bean> element’s id attribute is *not* specified because we typically don’t want InstanceValidationBeanPostProcessor to be a dependency of any other bean. The <property> element sets the value of order property to 1.

**BeanFactoryPostProcessor**

* [PropertyOverrideConfigurer](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/config/PropertyOverrideConfigurer.html) for "beanName.property=value" style overriding (*pushing* values from a properties file into bean definitions)
* [PropertyPlaceholderConfigurer](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/config/PropertyPlaceholderConfigurer.html) for replacing "${...}" placeholders (*pulling* values from a properties file into bean definitions)

BeanFactoryPostProcessor is an interface and beans that implement it are actually beans that undergo the Spring lifecycle (Example below) but these beans don't take part of the other declared beans' lifecycle.

public class CustomBeanFactory implements BeanFactoryPostProcessor {

@Override

public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException {

for (String beanName : beanFactory.getBeanDefinitionNames()) {

BeanDefinition beanDefinition = beanFactory.getBeanDefinition(beanName);

// Manipulate the beanDefiniton or whatever you need to do

}

}

}

**The differences about BeanFactoryPostProcessor and BeanPostProcessor:**

* A bean implementing BeanFactoryPostProcessor is called when all bean definitions will have been loaded, but no beans will have been instantiated yet. This allows for overriding or adding properties even to eager-initializing beans. This will let you have access to all the beans that you have defined in XML or that are annotated (scanned via component-scan).
* A bean implementing BeanPostProcessor operate on bean (or object) instances which means that when the Spring IoC container instantiates a bean instance then BeanPostProcessor interfaces do their work.
* BeanFactoryPostProcessor implementations are "called" during startup of the Spring context after all bean definitions will have been loaded while BeanPostProcessor are "called" when the Spring IoC container instantiates a bean (i.e. during the startup for all the singleton and on demand for the proptotypes one)

**Spring Beans Default Scope**

Yes, by default Spring beans are singleton. This means that only one instance of the bean is created and cached in memory, and all the requests for that bean will return a shared reference to the same bean.

This can be beneficial for performance reasons, as it avoids the overhead of creating and destroying new objects each time they are needed. It can also help to ensure that data is consistent, as all requests are using the same instance of the bean.

However, there are some cases where it may not be desirable to have a singleton bean. For example, if the bean is stateful, then changes made to the bean by one request may be visible to other requests. This can lead to unexpected behaviour and data corruption.

In these cases, you can use one of the other bean scopes provided by Spring. The most common alternative to singleton is prototype scope, which creates a new instance of the bean each time it is requested.

To change the scope of a bean, you can use the @Scope annotation. For example, to create a prototype bean, you would use the following annotation:

@Scope("prototype")  
public class MyBean {  
 *// ...*  
}

You can also configure the scope of a bean in the Spring configuration file. For example, to create a prototype bean, you would add the following element to the configuration file:

<bean id="myBean" class="com.example.MyBean" scope="prototype">  
 *// ...*  
</bean>

By default, Spring beans are singleton, but you can change the scope of a bean to prototype or another scope if needed.

In Spring, you can have multiple beans of the same type by either using annotations or XML.

Here are the steps on how to have multiple beans in Spring using annotations:

1. Create a Java class for each bean.
2. Annotate each class with the @Service or @Component annotation.
3. In the Spring configuration file, add the following annotation to each bean:

@Bean(name = "beanName")

where "beanName" is the name of the bean.

**Here are the steps on how to have multiple beans in Spring using XML:**

1. Create an XML file for each bean.
2. In the XML file, add the following element for each bean:

<bean id="beanName" class="beanClassName"/>

where "beanName" is the name of the bean and "beanClassName" is the name of the Java class for the bean.

Once you have created the beans, you can inject them into other beans using the @Autowired annotation.

Here are some additional things to keep in mind when working with multiple beans in Spring:

* If you have multiple beans of the same type, Spring will automatically inject the bean that is named "primary".
* If you want to inject a specific bean, you can use the @Qualifier annotation to specify the name of the bean.
* You can also use the @Autowired annotation to inject a list of beans of the same type.

**Using BeanDefinitionRegistryPostProcessor**

Here is an example of how to use BeanDefinitionRegistryPostProcessor to create multiple beans of the same type:

@Configuration  
public class MyConfiguration {  
  
 @Bean  
 public static BeanDefinitionRegistryPostProcessor myBeanDefinitionRegistryPostProcessor() {  
 return new BeanDefinitionRegistryPostProcessor() {  
 @Override  
 public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) {  
 *// Register the first bean*  
 BeanDefinition beanDefinition1 = new BeanDefinition();  
 beanDefinition1.setBeanClassName("com.example.MyBean");  
 beanDefinition1.setScope(BeanDefinition.SCOPE\_PROTOTYPE);  
 registry.registerBeanDefinition("myBean1", beanDefinition1);  
  
 *// Register the second bean*  
 BeanDefinition beanDefinition2 = new BeanDefinition();  
 beanDefinition2.setBeanClassName("com.example.MyBean");  
 beanDefinition2.setScope(BeanDefinition.SCOPE\_PROTOTYPE);  
 registry.registerBeanDefinition("myBean2", beanDefinition2);  
 }  
 };  
 }  
}

In this example, we create two beans of the same type, com.example.MyBean. We register the first bean with the alias myBean1 and the second bean with the alias myBean2. We also set the scope of both beans to prototype, which means that a new instance of the bean will be created each time it is requested.

**Defining the beans extending Thread and annotating them with scope prototype**

Here is an example of how to define the beans extending Thread and annotating them with scope prototype:

@Configuration  
public class MyConfiguration {  
  
 @Bean  
 @Scope(BeanDefinition.SCOPE\_PROTOTYPE)  
 public MyBean myBean1() {  
 return new MyBean();  
 }  
c  
 @Bean  
 @Scope(BeanDefinition.SCOPE\_PROTOTYPE)  
 public MyBean myBean2() {  
 return new MyBean();  
 }  
}

Use code with caution.

[Learn more](https://support.google.com/legal/answer/13505487?hl=en-IN)

In this example, we create two beans of the same type, MyBean. We annotate both beans with the @Scope annotation and set the scope to prototype. This means that a new instance of the bean will be created each time it is requested.

Which method you choose to use will depend on your specific needs. If you need to create multiple beans of the same type at runtime, then you should use the BeanDefinitionRegistryPostProcessor method. If you need to create multiple beans of the same type that are managed by Spring, then you should use the defining the beans extending Thread and annotating them with scope prototype method.

# **@Autowired, @Resource and @Inject**

### ****Let us understand @Autowired,@Resource and @Inject****

All these annotations are used to inject dependent objects in spring.

##### **Lets see what is their differences**

**@Resource** -> javax.annotation

**@Inject** -> javax.inject

**@Autowired** -> org.springframework.beans.factory.annotation

##### **Let’s start with the below example**

**Create Vehicle.java**

[Copy this code](javascript:;)

1. **package** com.kb.autowire\_resource\_inject;
3. **public** **interface** Vehicle {
5. }

**Create TwoWheeler.java**

[Copy this code](javascript:;)

1. **package** com.kb.autowire\_resource\_inject;
3. **import** org.springframework.stereotype.Component;
5. @Component
6. **public** **class** TwoWheeler **implements** Vehicle {
8. }

**Create FourWheeler.java**

[Copy this code](javascript:;)

1. **package** com.kb.autowire\_resource\_inject;
3. **import** org.springframework.stereotype.Component;
5. @Component
6. **public** **class** FourWheeler **implements** Vehicle {
8. }

Let’s see some scenarios to understand it clearly

###### **Scenario 1 -> Inject using Interface type**

[Copy this code](javascript:;)

1. @Resource
2. Vehicle vehicle;
4. @Autowired
5. Vehicle vehicle;
7. @Inject
8. Vehicle vehicle;

All the above annotations are doing the same thing that is trying to inject the **dependent object** by Type.

Type is Vehicle but it has 2 implementations (**TwoWheeler**and **FourWheeler**)

Hence stuck with **ambiguity**exception as below

No qualifying bean of type [com.kb.autowire\_resource\_inject.Vehicle] is defined: expected single matching bean but found 2: fourWheeler,twoWheeler

###### **Scenario 2 -> Inject using field type as concrete class**

[Copy this code](javascript:;)

1. @Resource
2. TwoWheeler vehicle;
4. @Autowired
5. TwoWheeler vehicle;
7. @Inject
8. TwoWheeler vehicle;

Now all the above annotations get successful injection of dependent objects.

**Reason**– All are trying to inject by type and **type of the vehicle** is concrete class TwoWheeler hence **no ambiguity** for injecting.

###### **Scenario 3 -> injecting using field name**

[Copy this code](javascript:;)

1. @Resource
2. Vehicle twoWheeler;
4. @Autowired
5. Vehicle twoWheeler;
7. @Inject
8. Vehicle twoWheeler;

Now all the above annotations will **inject**the **dependent**object successfully.

**Reason**: It’s injecting by name , whenever we use **@Component** on the class, automatically class name itself is registered as a spring bean.

**TwoWheeler**class register as twoWheeler with container and twoWheeler bean is only one available inside container so **no ambiguity**.

###### **Scenario 4 -> Qualifier with default bean name**

[Copy this code](javascript:;)

1. @Resource
2. @Qualifier("twoWheeler")
3. **private** Vehicle vehicle;
5. @Autowired
6. @Qualifier("twoWheeler")
7. **private** Vehicle vehicle;
9. @Inject
10. @Qualifier("twoWheeler")
11. **private** Vehicle vehicle;

All the above annotation **injects**the **dependent**bean successfully.

**Reason**: All are injecting by using **qualifier**name which is thoWheeler and we have only one bean with this name in the container.

###### **Scenario 5 -> Injecting list of beans**

[Copy this code](javascript:;)

1. @Resource
2. **private** List<Vehicle> vehicles;
4. @Autowired
5. **private** List<Vehicle> vehicles;
7. @Inject
8. **private** List<Vehicle> vehicles;

all the above annotations injects the dependencies to the List of supertype class Vehicle.

###### **Scenario 6 -> special behavior of @Resource during beans conflict while injecting**

[Copy this code](javascript:;)

1. @Resource
2. @Qualifier("noSuchBean")
3. **private** Vehicle twoWheeler;
5. @Autowired
6. @Qualifier("noSuchBean")
7. **private** Vehicle twoWheeler;
9. @Inject
10. @Qualifier("noSuchBean")
11. **private** Vehicle twoWheeler;

In above scenario twoWheeler is not injected using **@Autowired** and **@Inject**

But twoWheeler is injected using @Resource.

**Reason**-> **@Resource**uses the variable name which is twoWheeler to make dependency injection and ignores the @Qualifier.

But **@Autowired**and @Inject fails as they try to find a bean name specified in @Qualifier.

So **@Autowired and @Inject** works very **similar**but **@Resource** has bit different feature.

The order of choosing injection type

###### **@Autowired and @Inject**

1.**Match by Type** -> bean with same Data type of the variable should be available in spring container

2.**Restricts by Qualifier** -> If bean of variable’s data type not found or many implementation of the type available then it looks for any qualifier defined and if defined it uses Qualifier and wont go for 3rd option

3.**Matches by Name** –> searches the bean in the spring whose id should be same as variable name defined while autowiring.

###### **@Resource**

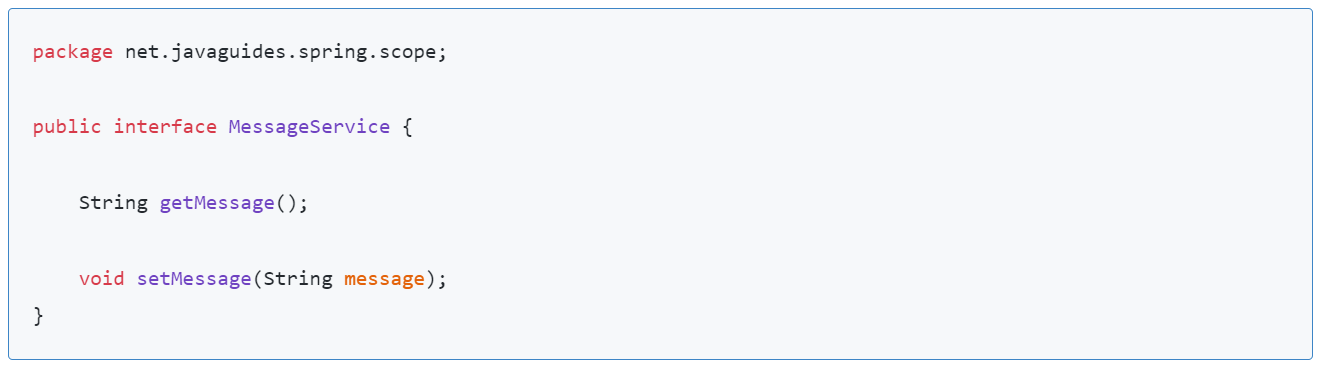
1.**Match by Name** -> first it searches for the bean in spring whose id should be same as **variable name declared.**

2.**Match by Type** -> bean with same Data type of the variable should be available in spring container

3.**Restricts by Qualifier**(ignores if 1st attempt said above by name matches)

**Prototype Bean Configuration**

* 1. **Annotation based config**



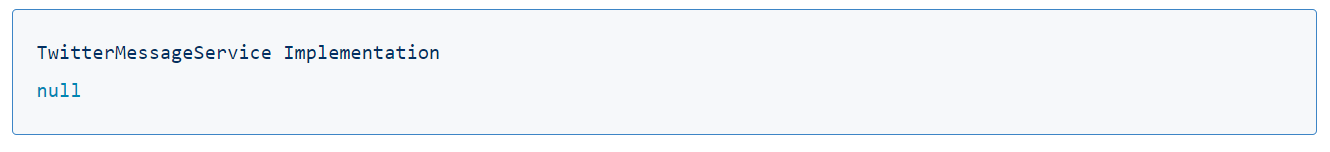
A screenshot of a computer program

Description automatically generated

A computer screen shot of a computer code

Description automatically generated

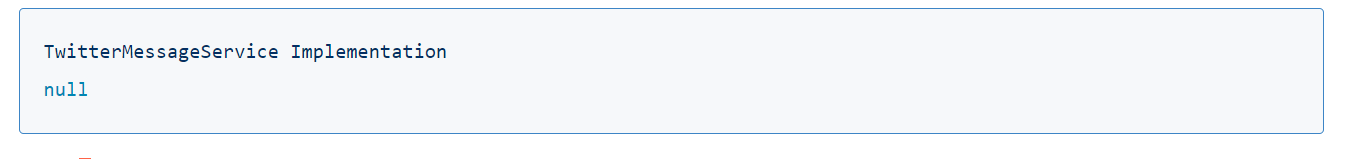




**Java Based configuration**







**All Aware interfaces available in Spring**

**1) ApplicationContextAware**  
Bean implementing this interface can get the **current application context** and this can be used to call any service from the application context

**2) BeanFactoryAware**  
Bean implementing this interface can get the **current bean factory** and this can be used to call any service from the bean factory

**3) BeanNameAware**  
Bean implementing this interface can get its **name**defined in the Spring container.

**4) MessageSourceAware**  
Bean implementing this interface can get the access to**message source** object which is used to achieve **internationalization**

**5) ServletContextAware**  
Bean implementing this interface can get the access to **ServeltContext**which is used to access servlet context parameters and attributes

**6) ServletConfigAware**  
Bean implementing this interface can get the access to **ServletConfig**object which is used to get the servlet config parameters

**7) ApplicationEventPublisherAware**  
Bean implementing this interface can publish the **application events** and we need to create listener which listen this event.

**8) ResourceLoaderAware**  
Bean implementing this interface can load the resources from the classpath or any external file.