Contents

[Spring & Spring boot 2](#_Toc116756265)

[Introduction 2](#_Toc116756266)

[Different Types of Java Frameworks 4](#_Toc116756267)

[Different Modes of Spring application Development 4](#_Toc116756268)

[Annotation Driven Configuration 4](#_Toc116756269)

[@Required 4](#_Toc116756270)

[@Autowired 5](#_Toc116756271)

[Stereo Type annotation 7](#_Toc116756272)

[Method Replacer 8](#_Toc116756273)

[Points to remember while working with method replacer 8](#_Toc116756274)

[Factory Bean 8](#_Toc116756275)

[ServiceLocator Design Pattern 9](#_Toc116756276)

[Advantages of taking ServiceLocator as FactoryBean class in spring env 9](#_Toc116756277)

[Limitations of taking ServiceLocator as factory bean class in spring env 9](#_Toc116756278)

[Bean Post Processor 10](#_Toc116756279)

Spring & Spring boot

Introduction

Spring is a java framework that is developed on the top of Java, J2EE technologies to simplify the application development. Spring boot is a framework that is developed on the top of spring framework to simplify the application development. Spring boot is an extension to the spring framework to give another level of abstraction at configuration level. Spring boot also brings some level of automation for dependency management.



Framework provides an abstraction layer on various technologies to simplify the application development. It helps from broiler plate code re-write and improves the productivity of application development. Various level of pre-designed APIs helps, which again helps in application development.

We have different types of frameworks like:

1. Web application framework

It provides an abstraction layer on technologies like Servlet, JSP and simplifies MVC architecture-based java wen application development.

Some popular frameworks are: Spring, JSF (java Server Faces), ADF (Application Development Framework)

1. ORM framework

Provides the abstraction on technologies like JDBC and simplifies the object-based Database software independent Persistence logic with out using SQL query.

Some popular frameworks are: Hibernate, Spring JDBC, Eclipse Link, iBatis & etc…

Spring ORM, Spring data JPA modules provides abstraction on different frameworks like Hibernate.

Note: All ORM frameworks follows JPA specifications given by Sun MS. JPA is a theoretical documentation having guidelines to develop ORM framework.

1. JEE Framework/ Application framework

These frameworks provide abstraction on Java, JEE technologies and on small java frameworks to simplify all kinds of logics development and all kinds of application development.

Spring is a solid example for this, which itself provides an abstraction on different technologies as well as different frameworks.

1. Web services framework/ Distributed application framework

These frameworks are based web services methodology of programming or application development to develop distributed applications as the web based interoperable applications.

An Interoperable application is the client and server applications of distributed applications that can be developed either in same domain/ language/ technology or different domains/ languages/ technologies.

We have 2 types of web service:

1. SOAP based: Uses HTTP over SOAP as protocol for the communication between client and server applications.

Some examples are: jax-ws, axix, apache cfx & etc..

1. Restful web service: Uses http with JSON as protocol for the communication between client and server applications.

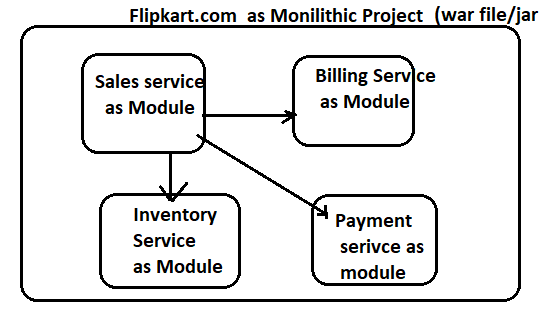
Some examples are: jax-rs, Jersy, Spring Rest

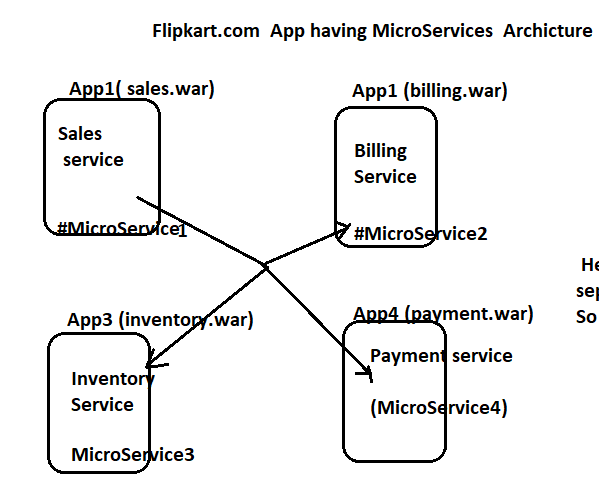
1. Big data frameworks & etc…

Data that is beyond the capacity of storing and processing from a regular Hard-disk / Database software is called Big Data. Big data framework has been given to store and process this big-data on a network.

Some examples are: Hadoop, Spark

A monolithic architecture is the traditional unified model for the design of a software program. Monolithic, in this context, means “Composed all in one piece”. It’s a combination of some tightly coupled modules, that cannot be extracted explicitly and difficult to make changes. In monolithic architecture, each component and its associated component must all be present for code to execute or compile and for the software to run.

 To avoid this tight coupling and maintenance overhead, we came up with Microservice architecture. It is a variant of Service-Oriented Application structural styled architecture pattern that arranges an application as a collection of loosely-coupled, fine grained services, communicating through lightweight protocols.



# Web Application Vs Distributed Application Vs Enterprise Application

Web application are client server applications where server is a Software application and client is fixed, that is web browser. These are thin clients or we can say them as Thin Server application. Interaction model of Web applications is request-response model. We use technologies like servlet, JSP and frameworks like struts, spring MVC, spring boot to develop web applications.

Distributed Application is a software executes or runs on multiple computers in a network. They interact in order to achieve a specific task and you have these applications running on both the client and server system. Basically, we have multiple users trying two access the system at once. The interactions here is between the client systems that access the data and the server that processes the data. These are the fat servers or we call them as Fat clients.

Enterprise applications can be a web application or a distributed applications or can be the combination of both. It is big business application, which is complex, scalable and deployed on a variety of platforms. Unlike distributed applications that are specifically designed for a specific requirement, the enterprise applications are meant to satisfy multiple requirements and these requirements are inter-dependent on each other.

# Different Types of Java Frameworks

1. Invasive Framework.

An invasive framework means that you are forced to extend/ implement one of their classes/ interfaces. These are tightly coupled frameworks.

Some examples are: Struts or EJB2

1. Non-Invasive Framework.

A Non-Invasive framework doesn’t force a programmer to extend/ Implement their class from any predefined class or interface given by the framework. These are loosely coupled frameworks. Because of loose coupling, these frameworks enhance the portability of application.

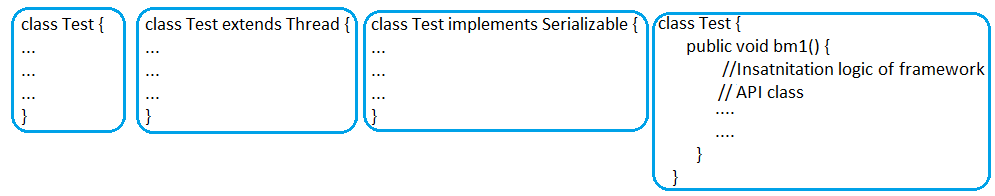
Some examples are: Spring

# Some technical terms of Java Frameworks

### POJO class (Plain Old Java Objects)

This is an ordinary Java class with out any specialities. These classes don’t implement technology/ framework API interfaces or does not extend from technology/ framework classes. Non-Invasive frameworks take support of POJO class. Classes designed with POJO class recommendation doesn’t restrict to implement/ extend any framework/ technology interface/ class.

Examples of POJO classes:



Examples of classes that cannot be called as POJO class:

### 

### POJI (Plain Old Java Interface)

This is an ordinary Java Interface with out any specialities. These interfaces doesn’t extend any technology/ framework API interfaces.

### Java bean class

Java bean is a java class that has been developed with some java standards, as stated below:

1. Classes must be declared as public
2. Recommends to implement java.io.Serializable(I) interface
3. Recommended to place 0-parameter constructors directly or indirectly.
4. This class should define all the states/ variables with private access specifier and should be non-static.
5. Public Getter and Setter method should be there to determine the state.

Java Bean is always used as data carrier. Helper class to bind multiple values into single object and to pass object data from one layer of application to other layer of same/ different application.

When we don’t use java bean, to send data from one layer to another layer, we use function parameters. While using function parameters, we need to remember the sequence of the parameter and type of parameter, otherwise it leads to a syntactic error. Using function parameter also reduces the readability of the code. Because of these reasons, it’s always good to use Java bean for data carrier.

### Bean class/ Component class

Java classes that are designed having both state (member variables) and behaviour (methods) and states being used inside the behaviour’s business logic is called Component class or bean class. Bean class is different from Java bean class. We use bean class as main class having business methods there as we use Java bean class as Model/ helper class to represent a model object.

Component class can be as POJO or Non-POJO class. All main classes of the project can develop as the component class. In most of the cases, Invasive frameworks take the component class as non-POJO class, whereas Non-Invasive framework tries to keep these classes as POJO class.

### Spring Bean

 Spring bean is a java class whose object is created and managed by spring container. It can be a user defined or library class or third party supplied class. It can be POJO, non-POJO or a component class. We cannot take Abstract class or interfaces as spring bean class because we cannot instantiate an object for this class. All the containers run on the top of JRE/JVM.

# Spring Core

## Introduction

This is the base module for all the modules. With the implementation of this module, we can only develop standalone applications. This module gives two containers:

1. Bean Factory container.
2. Application Context Container.

## Spring Containers (IoC Containers)

Containers are there to create and manage the life cycle of beans. In layered applications there are number of classes we do instantiate to complete our task and these classes belongs to a particular framework/ technology we work with. There can be a lot of dependent classes we might need to instantiate, before instantiating our required class. Like before instantiating a Connection class object, there must be some dependent class we need to initialize, that we are not aware of and if we know certain also, we don’t know how to inject them into our implementation hierarchy. In this case containers help us to instantiate those classes in automated way and injecting them to require places, before giving us a required class object.

Spring containers are lightweight containers and can be started in the middle of any standalone, web application. By instantiating the spring supplied API classes, we can build a spring container. Never like the Servlet/ JSP container, Spring container doesn’t require a heavy weight web/ application server to build a container. Because of above said nature, we call spring container as lightweight. These containers are being used for spring bean life cycle management and dependency management.

Why do we call the spring containers are called as IoC container?

IoC is also called as Dependency Injection. It is a process where object define their dependencies through constructor arguments, arguments to a factory methods or properties. These dependencies are set on the on-object instance after it is constructed or returned from a factory method. This process id fundamentally the inverse, hence the name Inversion of Control.

## Dependency Lookup Vs Dependency Injection

In Dependency lookup approach, while constructing the target we need to arrange all the dependencies. But in the Dependency Injection, a DI container works to gather the dependencies dynamically and inject them to the target class. Spring supports both the approach for dependency management. But in most of the scenarios, we use the dependency injection approach.

Different types of dependency injection mechanism used in spring are:

1. Setter Injection
2. Constructor Injection
3. Field Injection
4. Arbitrary Method Injection
5. Aware Injection/Interface injection
6. Lookup Method Injection
7. Method injection/Method Replacer.

IoC containers for standalone applications

There are two IoC containers has been provided by spring”

1. BeanFactory
2. ApplicationContext

ApplicationContext is the child class of BeanFactory container. It means AppliacationContext container comes up with all the capabilities of BeanFactory container and some more newly added features.

## Configuring a spring bean with XML and annotation-based approach

Passing details or instructions to the underlying server, container or framework is called configuration. To configure a java class as a spring bean, we use <bean> (called as bean tag) in spring bean configuration XML file or @Component annotation at class level.

Though the XML based configuration is already out of date, still there are some use cases where we use XML for spring configuration. When we use a XML for spring bean configuration then it is recommended to use applicationContext as the name of spring bean configuration file.

Every spring bean is identified in the system with its bean id. Spring has its own mechanism to give a default bean id for all the beans. But spring also allows us to give a bean id to the beans. User specified bean names helps to avoid the ambiguity problem. Whether it’s a user defined class or a java library class, we need to configure the class as spring bean to be managed by spring IoC container.

Here is the example how can we define a user defined class(UserManagement.Java) as spring bean. Java library classes can also defined in the same way, but for java library classes we can’t use the annotation based bean declration.

### XML based Bean declaration

<bean id=”userMgmt” class = “org.luv2code.UserManagement” />

### Annotation Based Bean declaration

@Component

Public class UserManagement

### Code based Bean Declaration

@Bean

Public UserManagement getUserManagement() {

return new UserManagement();

}

## How spring IoC container create a java object and make that a spring bean

Spring IoC container uses 3 ways to create spring bean classes:

1. 0-param Constructor
2. Parameterized Constructor and
3. Factory Method

### 0-param Constructor

When we specify a bean with no injection or setter injection only then spring IoC container uses the 0-param constructor to create the java class.

Setter injection is meant to inject the dependency class to the target class. In setter injection after java class creates a object, it takes the setter methods of the states to inject the dependent class objects.

We can achieve the setter injection by using the property tag (<property>) in XML based configuration or we can use the @Autowired annotation in class states level or at setter method level.

### Parameterized Constructor

When we specify a bean with constructor injection and pass require parameter to the configuration details, then IoC container uses the parameterized constructor of the configured java class to create the object. In this injection, dependent objects are being instantiated at first then target class object gets created with dependent class objects.

We can specify the constructor injection by using constructor-arg tag(<constructor-arg>) or @Autowired annotation on the top of the constructor.

### Factory Method

In Basic java project, there are some classes whose object creation is handled by a library class method and we call that method as factory method. This factory method has ability to create same class object or different class object(relevant or irrelevant). Like Java.lang.Thread class. We need to call the factory method of Thread class itself to create same class object.

Thread thread = Thread.currentThread();

To make IoC container creates this object we need to pass factory-method attribute in bean tag like:

<bean id=”thread” class=”java.lang.Thread” factory-method=”currentThread”/>

To make

Different Modes of Spring application Development

We have different modes of application development in spring, they are something like this:

1. XML driven configuration
2. Annotation driven configuration
3. 100% code Driven configuration
4. Spring boot driven configuration

## Annotation Driven Configuration

Annotation support is introduced to spring from spring 2.0 and added more annotation in the later versions to enhance annotation driven configuration programming. Here are some annotations are there.

@Required

While using any parameterized constructor for constructor injection, we must configure all parameters of that constructors for injection, otherwise exception may come. This restriction is not there while working with setter injection. To bring such restriction on our choice bean properties through setter injection, we need to go for @Required annotation.

**class** PersonalInfo {

**private** **int** pid;

**private** String pName;

**private** String pAddrs;

@Required

**public** **void** setPid(**int** pid) {

**this**.pid = pid;

}

@Required

**public** **void** setpName(String pName) {

**this**.pName = pName;

}

**public** **void** setpAddrs(String pAddrs) {

**this**.pAddrs = pAddrs;

}

//toString

}

In this example, if we will not pass pid and pName while configuring bean, then we will get BeanInitializationException. But unfortunately, @Required annotation has been deprecated in spring 5.X in favour for using constructor injection for required settings (or custom InitializingBean implementation). RequiredAnnotationBeanPostProcessor has the implementation of @Required annotation.

Note: Every Bean Processor will be activated automatically once it is configured as spring bean in spring bean configuration file. But configuring each bean post processor is a complex process, alternate to that we can add context namespace and can add <context:annotation-config/> tag, which activates some of the annotation post processor in application context container. From spring 5.1 <context:annotation-config/> is not working for deprecated annotation. If we are using any deprecated annotation then we need to configure deprecated annotation’s post processor explicitly.

### @Autowired

It performs byType, byName and constructor mode of auto-wiring (detecting the dependent beans dynamically without using <property>, <constructor-arg> args). It can be applied in filed level, constructor level, setter method level or in arbitrary method level. It Cannot be used to inject simple values; it can only inject value of Object/ref types.

While working with setter or constructor injection with XML bean configuration file support, we required setter, getter or constructors mandatorily, but while working with @Autowired annotation we can inject property values by using either field, setter method, constructor or arbitrary methods. In this situation it will access private property through reflection API and performs and injection on that property by seeing @Autowired annotation.

#### Internal Flow

IOC container creation 🡪 Loading of spring bean configuration file 🡪 Checking well formedness and validate spring configuration file 🡪 create in-memory meta data of spring bean configuration file 🡪 Pre-instantiation of all singleton scope 🡪 Activation of BeanPostProcessor based <context:annotation-config/> tag. 🡪 Users reflection API and detects in @Autowired on the top of the field in the class 🡪 get Access to that property and search filed variable type spring bean configuration in the in memory meta data of spring bean file 🡪 finds/creates(In case not available in meta data) the bean of required type and assigns the bean to the property of target class 🡪 Keeps all the beans if in-memory meta data of IOC container.

Sometimes we configure more than one dependent type bean in spring bean configuration file, in this case if we will configure the dependent bean in target with @Autowired Annotation then we will get NoUniqueBeanDefinitionException while configuring target bean. This is called ambiguity problem.

#### How to resolve ambiguity problem, that comes while working with @Autowired based dependency injection?

There are number of ways to do this.

1. Using @Qualifier with dependent bean id or using qualifier name of dependent bean.

<bean id =*"udt"* class =*"java.util.Date"*>

<qualifier value=*"u1"*/>

</bean>

<bean id =*"sdt"* class =*"java.sql.Date"*>

<qualifier value=*"s1"*/>

</bean>

@Autowired

@Qualifier("date")

// @Qualifier("u1")

**private** Date date;

1. By Matching target filed/property name with dependent class bean id.

<bean id =*"date"* class =*"java.util.Date"/*>

<bean id =*"sdt"* class =*"java.sql.Date"/*>

@Autowired

// @Qualifier("date")

// @Qualifier("u1")

**private** Date date;

1. Using primary*=”true”* of <bean> (in XML driven spring bean cfg file) or @Primary (In annotation driven spring bean cfg) for one of the dependent bean.

<bean id =*"udt"* class =*"java.util.Date"* primary*=”true”/*>

<bean id =*"sdt"* class =*"java.sql.Date"/*>

@Autowired

// @Qualifier("date")

// @Qualifier("dt")

**private** Date date;

1. By making only one dependent bean as Autowire candidate in spring bean configuration file, but here if we have “n” dependent bean of same kind then we need to set n-1 beans not as autowire candidate and the required one as autowire candidate bean.

<bean id =*"date"* class =*"java.util.Date"* autowrire-candidate*=”true”/*>

<bean id =*"date"* class =*"java.sql.Date"* autowrire-candidate*=”false”/*>

#### @Autowire at different level in target class

1. At setter method level. It does byType autowiring, in case of ambiguity if we define Qualifier annotation then it will do byName type of autowiring.

@Autowired

@Qualifier("udt")

**public** **void** setDate(Date date) {

System.***out***.println("WishMessageGenerator: setter method");

**this**.date = date;

}

1. At parameterized constructor level. It will do constructor injection

@Autowired

**public** WishMessageGenerator(@Qualifier("udt") Date date) {

System.***out***.println("1-param constructors");

**this**.date = date;

}

1. At arbitrary method level. It does byType autowiring, in case of ambiguity if we define Qualifier annotation then it will do byName type of autowiring. **The arbitrary method needs to be had a common signature (public void).**

@Autowired

@Qualifier("udt")

**public** **void** assign(Date date) {

System.***out***.println("WishMessageGenerator.assign()");

**this**.date=date;

}

1. At field level. It does byType autowiring, in case of ambiguity if we define Qualifier annotation then it will do byName type of autowiring. It gives flexibility to work with other field level annotation as well as; it improves the readability of code. So always it is recommended to work with field level annotation.

Note: If we are enabling autowiring in all level (field, constructor, setter, arbitrary method) then either setter or arbitrary method level will override the other. Order of injection will be Constructor level 🡪 field level 🡪 either setter/arbitrary method.

Important observation in @autowired annotation:

1. Applying @Autowired on 0-param constructor is meaningless.
2. If multiple overloaded constructors having @Autowired(required = **true**) then BeanCreationException will be raised.
3. If multiple overloaded constructors having @Autowired(required = **false**) then constructor having greatest number of dependency bean will be executed.
4. If multiple overloaded constructors having @Autowired(required = **false**) on few constructors and @Autowired(required = **true**) on few constructors then also BeanCreationException will be raised.
5. If multiple overloaded constructors having @Autowired(required = **false**) with same number of parameters then IOC container picks up the constructor randomly.

Note: We can apply @Autowired on multiple fields, setters and arbitrary methods either having required value to true or false. But only constructors can have @Autowired with required = **true**.

Stereo Type annotation

We have multiple annotation with similar behaviours having minor differences, these annotations are called stereo type annotations. Some of these annotations are:

@Component 🡪 To config java class as spring bean.

@Service 🡪 @Component + also makes as service class by giving transaction management support.

@Repository 🡪 @Component + also makes as DAO class by having exception translation ability.

@Controller 🡪 @Componenet + also makes as web Controller class by having ability to process http request.

And etc…

# Method Replacer

## Points to remember while working with method replacer

We can not take target class as final class, because the generated proxy class/in-memory class comes as the sub class of the target class and by defining the target class as final class contradict the process. If we still declare the target class as final class then we will get the following exception:

org.springframework.beans.factory.BeanCreationException: Error creating bean with name <beanId> defined in class path resource <beanConfig File>: Instantiation of bean failed; nested exception is java.lang.IllegalArgumentExceptionInstantiation of bean failed; nested exception is java.lang.IllegalArgumentException: Cannot subclass final class <className>

We should not declare targeted method as final, because again the final method cannot be overridden in the sub class. If we still declare the targeted method as final method, then it will always execute the targeted method, though method replacer has been enabled.

Though we can write new logics for multiple target methods in single reimplement(-,-,-) of a MethodReplacer class, it is recommended to take separate replacer class for every target method.

There is no annotation given for <replaced-method> tag, so while working with annotations also target class needs to be configure in XML file in order to work with Method Replacement feature.

The <arg-type> tags under <replaced-method> are useful to configure method replacer for only one form of target method, if multiple overloaded target methods are available in the target class.

The underlying IOC container generates proxy class/ in-memory sub class for the target class only when <replaced-method> tag is present under <bean> tag that configures target spring bean class.

## Factory Bean

It is a bean which always injects a resultant object to target class. It is a selfless bean, which never injects the direct bean class, rather than it always injects a resultant object. Java class acts as a factory bean, when it implements FactoryBean interface. FactoryBean is a interface designed with java 8 module. Some methods are there inside this interface are:

1. T getObject(): This is an abstract method. We can place logic to gather/create resultant object.
2. Class<?> getObjectType(): This is an abstract method. We can place logic return Object class [java](eclipse-javadoc:%E2%98%82=factory-bean/C:%5C/Program%20Files%5C/Java%5C/jdk1.8.0_31%5C/jre%5C/lib%5C/rt.jar=/maven.pomderived=/true=/=/javadoc_location=/https:%5C/%5C/docs.oracle.com%5C/javase%5C/8%5C/docs%5C/api%5C/=/%3Cjava).[lang](eclipse-javadoc:%E2%98%82=factory-bean/C:%5C/Program%20Files%5C/Java%5C/jdk1.8.0_31%5C/jre%5C/lib%5C/rt.jar=/maven.pomderived=/true=/=/javadoc_location=/https:%5C/%5C/docs.oracle.com%5C/javase%5C/8%5C/docs%5C/api%5C/=/%3Cjava.lang).Class having resultant Object class name.
3. **boolean** isSingleton(): This is an default method, where we can specifies whether the resultant object is singleton or not. False means it is in prototype scope.

While working with FactoryBean dependency, IOC internal cache maintains resultant object in the target bean id mapping. Referring factory-bean project reference, IOC container will maintain cache in this way:

|  |  |
| --- | --- |
| Bean Id | Object Reference |
| Dfg | java.time.LocalDate Reference |
| Reminder | learn.luv2coode.beans.ScheduleReminder Reference |

applicationContext.xml for this IOC container can be found here.

While scanning the bean configuration file, if IOC container finds FactoryBean implemented class as bean, then it will not keep current FactoryBean object in the internal cache, more over it calls getObject() method on the current FactoryBean object to get Resultant object and keeps that Resultant object in the IOC container in internal cache.

<bean id=*"dfg"* class=*"learn.luv2coode.beans.DateFactoryBean"*>

<constructor-arg value=*"2020"*/>

<constructor-arg value=*"09"*/>

<constructor-arg value=*"19"*/>

</bean>

<bean id=*"reminder"* class=*"learn.luv2coode.beans.ScheduleReminder"*>

<constructor-arg ref=*"dfg"*></constructor-arg>

</bean>

# ServiceLocator Design Pattern

If we develop Client APP/Project as Spring based application, then we need to take Local service class as target and service locator as dependent spring bean. If we observe very carefully, local service class not interested in ServiceLocator object. It is actually interested in ServiceLocator supplied external; Component reference (Resultant Object), so we need to take serviceLocator as Factory Bean always.

## Advantages of taking ServiceLocator as FactoryBean class in spring env

1. No need of developing it as singleton java class, just singleton scope is sufficient.
2. No need of developing cache separately having external component reference, because the internal cache of IOC container itself maintains External component reference.

## Limitations of taking ServiceLocator as factory bean class in spring env

ServiceLocator becomes invasive, because to make it factory bean we need to implement spring API supplied FactorryBean(I)

What is the difference between service class and external component/ distributed app/ external service?

Local service is an ordinary java class having business logic of client APP/ client project (like service class id flipkart. Where as External component/service is a distributed application that can have type of remote or local applications accessing its services. External components will be developed by using distributed technologies like RMI, EJB, WebServices..

If Servlet, jsp based MVC web application is using Spring environment for developing Model layer service, Dao ServiceLocator classes. Can you tell me where should we create IOC container that is required model layer classes and how to get local service class object to the controller servlet?

Create IOC container in the init(-) method of Controller Servlet by enabling “load-on-startup” on ControllerServlet. Close IOC container in the destroy method and get service class object to ControllerServlet in service(-,-)/doGet(-,-)/doPost(-,-) by calling ctx.getBean(-,-) method. In this setup, ControllerServlet pre-instantiation and spring beans related pre-instantiation either during deployment of web application or at server start up.

# Bean Post Processor