**What Is Spring Framework?**

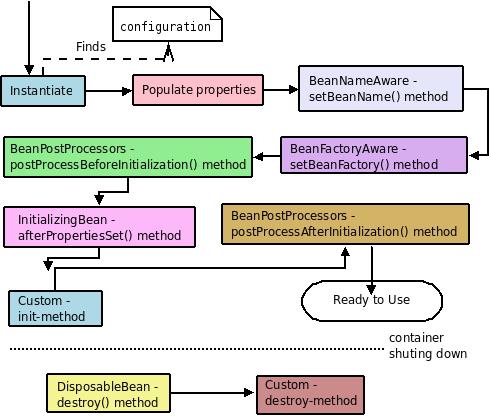
* Spring is a light weight and open source framework
* can be used for all layer implementations unlike struts [ only for front end related ] and hibernate [ only for database related ], but with spring we can develop all layers
* For testing spring application, server is not mandatory spring has its own container to run the applications.
* Spring has two patterns at its core. They are:

1. Inversion-of-Control (IOC)

2. Aspect Oriented Programming (AOP)

* **Inversion of control (IOC):** Loose coupling is achieved in spring using the technique Inversion of Control.
* **Aspect oriented (AOP):** Spring supports Aspect oriented programming.

**Explain Bean lifecycle in Spring framework?**



1) The spring container finds the bean’s definition from the XML file and instantiates the beanusing Java Reflection API.

2) If there are any properties associated with bean then spring injects those properties.

3) If the Bean class implements the BeanNameAware interface, then the spring will call setBeanName()method by passing the name of the Bean.  
4) If the Bean class implements the BeanFactoryAware interface, then the spring will callsetBeanFactory() method by passing an instance of BeanFactory object.  
5) If there are any BeanPostProcessors associated with the BeanFactory then the spring willcall postProcessBeforeInitialization() method before the properties for the Bean are injected.  
6) If the Bean class implements the InitializingBean interface, then the spring will callafterPropertiesSet() method 7) If there is any custom init-method declared in the configuration file, that method will be called.  
8) If there are any BeanPostProcessors associated with the BeanFactory then the spring will call postProcessAfterInitialization() method.  
9) Now bean is ready to use.  
10) If the Bean class implements the DisposableBean interface, then the spring calls destroy() method when the Application no longer needs the bean reference.  
11) If there is any custom destroy-method declared in the configuration file, that method will be called.

**What are the different modules in Spring framework?**

1. The Core container module
2. Application context module
3. AOP module
4. DAO module
5. ORM module
6. Web module
7. MVC module

## The IoC container

* The container will create the objects, wire them together, configure them, and manage their scope and complete lifecycle from creation till destruction.
* The spring container uses dependency injection (DI) to provide loose coupling between beans.
* org.springframework.beans  andorg.springframework.context packages provide the basis for IoC container.
* In IoC container all beans are managed by **BeanFactory** interface
* **ApplicationContext** interface extend **BeanFactory** interface.

**1)Bean Factory**

* + Bean instantiation/wiring

**2)Application Context**

* + Bean instantiation/wiring
  + Automatic BeanPostProcessor registration
  + Automatic BeanFactoryPostProcessor registration
  + Convenient MessageSource access (for i18n)
  + ApplicationEvent publication

## 1. Differentiate between BeanFactory and ApplicationContext in spring.

|  |  |
| --- | --- |
| **ApplicationContext.** | **BeanFactory** |
| Here we can have more than one config files possible | In this only one config file or .xml file |
| Application contexts can publish events to beans that are registered as listeners | Doesn’t support. |
| Support internationalization (I18N) messages | It’s not |
| Support application life-cycle events, and validation. | Doesn’t support. |
| Supports  many enterprise services such JNDI access, EJB integration, remoting | Doesn’t support. |

**Instantiating a container**

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

IUserMasterServiceuserService= (IUserMasterService)context.getBean("userService");

**Dependency Injection**

* IN SHORT: The basic principle behind **Dependency Injection (DI)**is that objects define their dependencies. Then, it is the job of the container to actually inject those dependencies when it creates the bean. So you do not have to create your objects but you can describe the way of how they should be created.
* If Class A contain the reference of another class B as a property then if class A want to perform his work then it must have to create the object of Class B so here is tight coupling between this two objects this is called Class A is dependent on Class B. if method of class B changed then we also have to change the Class A logic.

So springcontainer creates the object of B and inject into their dependent object (A). Like “Don’t call me I will call you”. This whole process is called dependency injection.

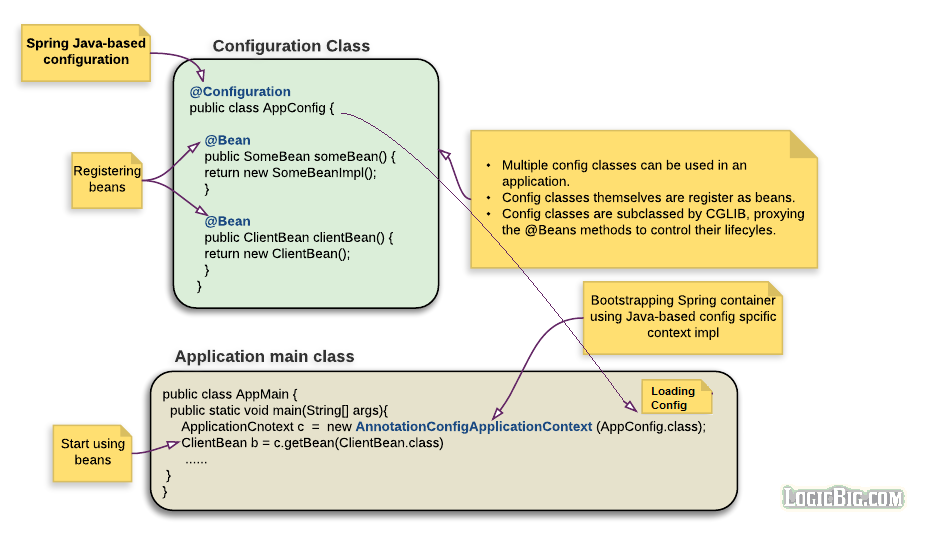
Advantages of Dependency Injection

* Loosely couple architecture.
* Separation of responsibility.
* Configuration and code is separate.
* Using configuration, a different implementation can be supplied without changing the dependent code.
* Testing can be performed using mock objects.

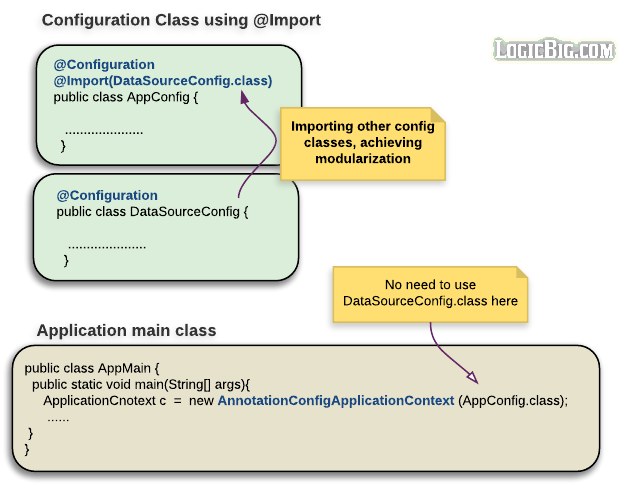
**Spring - Configuration Metadata**

Spring configuration metadata is to tell Spring container how to initiate, configure, wire and assemble the application specific objects

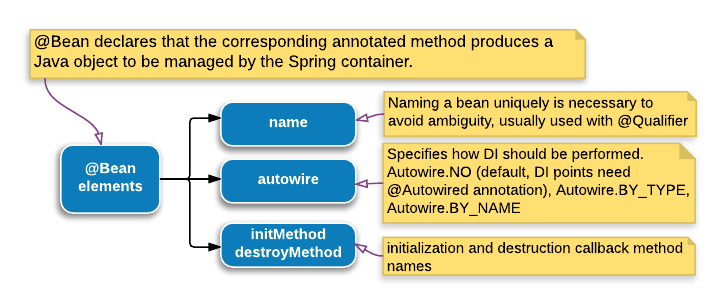
1. **XML-based Configuration :** All configurations are in one or multiple XML files. This is the most verbose way of configuration. Huge projects require tedious amount of XML which is difficult to manage.
2. **Annotation-based configuration :** Spring 2.5 introduces annotation-based configuration. We still have to write XML files but just to indicate "component-scan" on the packages of annotated classes.
3. **Java-based configuration (JavaConfig):**Starting with Spring 3.0, a pure-Java means of configuring container was provided. We don't need any XML with this method of configuration.



# Spring - Using @Import



# Spring - Using @Bean



The @Bean annotation is used in @configuration annotated class

**name :**

The optional bean name.

@Configuration

public class AppConfig {

@Bean(name = "myBean")

public MyBean createBean() {

......

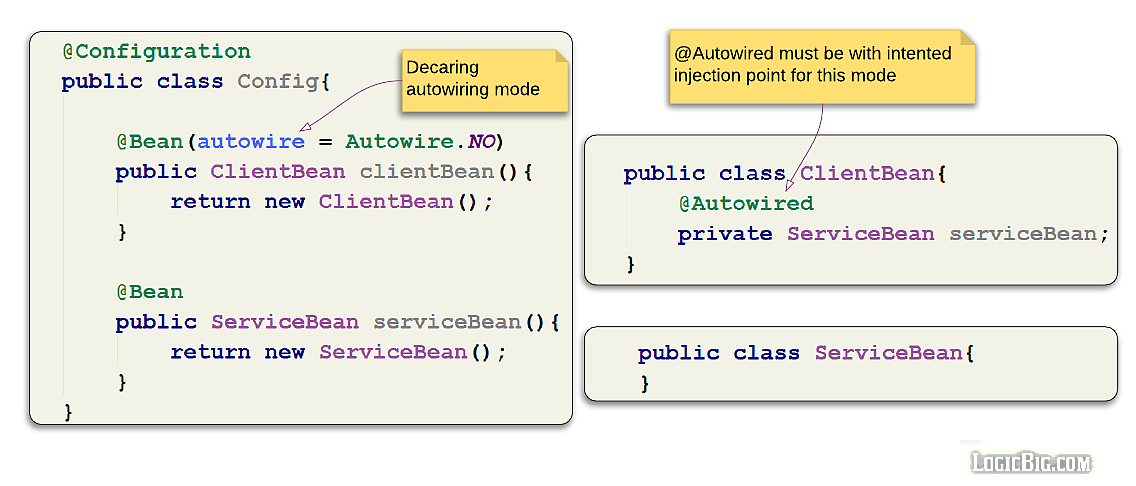
}

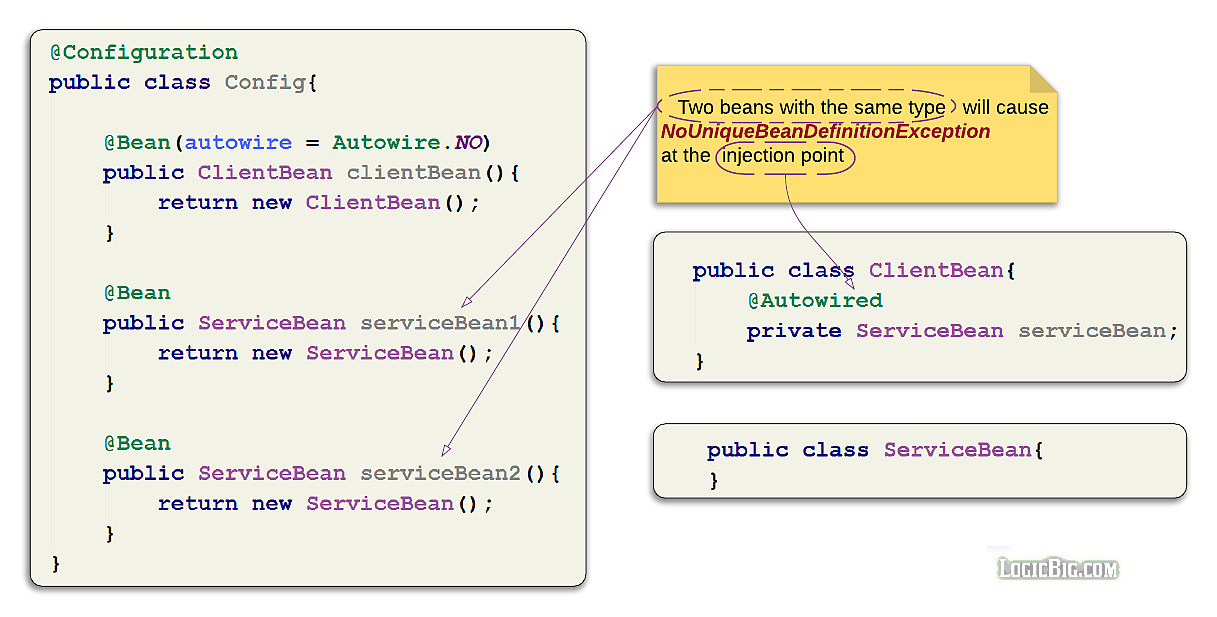
}

**autowire :**

The autowiring mode.

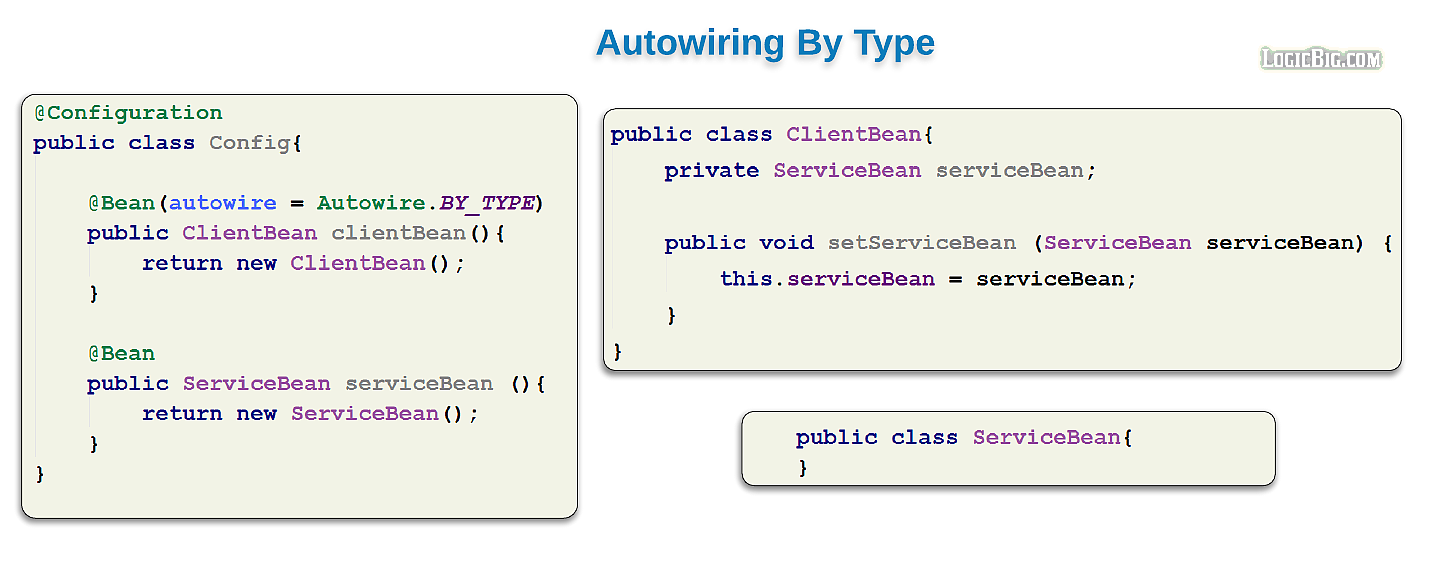
Autowire.NO : This is the default. In this case, we have to explicitly use @Autowired at injection point.



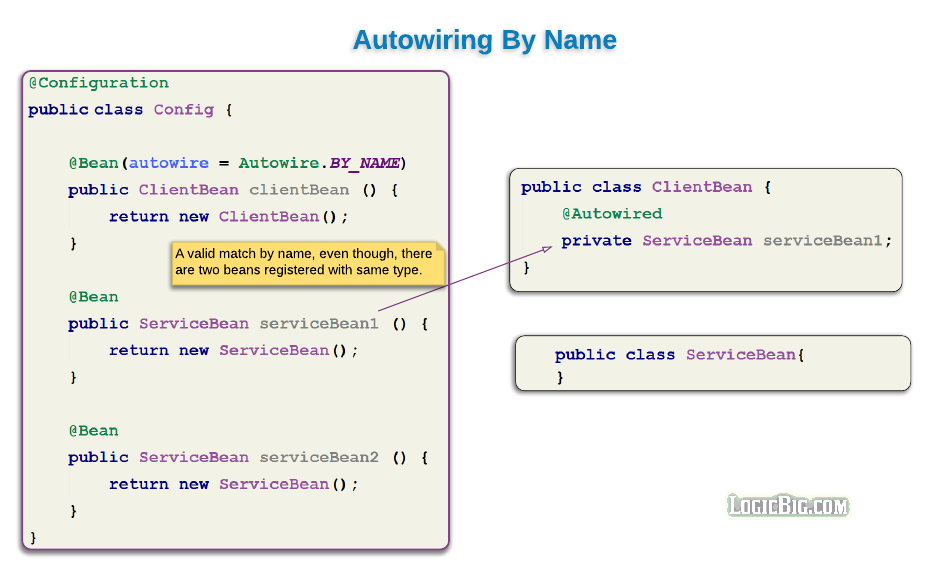


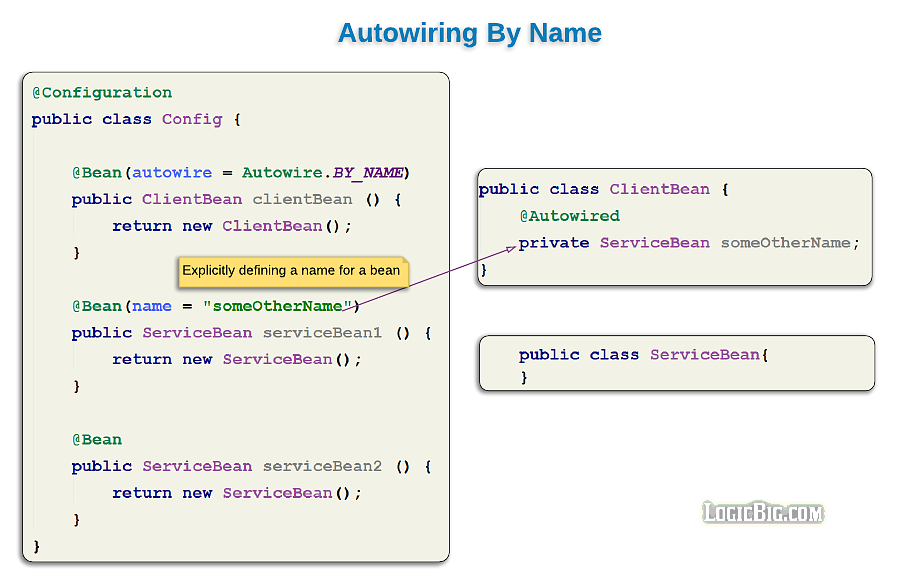
To avoid exception, we have to [use @Qualifier](https://www.logicbig.com/tutorials/spring-framework/spring-core/inject-bean-by-name.html).

Autowire.BY\_TYPE : we don't need @Autowired at the injection point, given that there is only one bean available for the injection. In this mode of autowiring, the field injection doesn't work. There must be a setter.

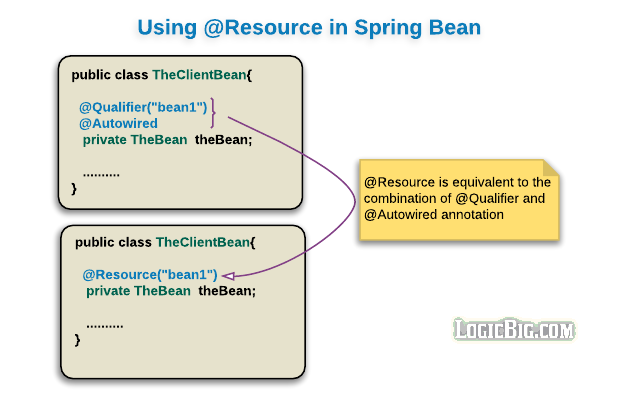


Autowire.BY\_NAME : If this mode of autowiring is specified and injection provider bean has specified name element with the some value in it's @Bean annotation, we have to use @Qualifier along with @Autowiredat injection point.

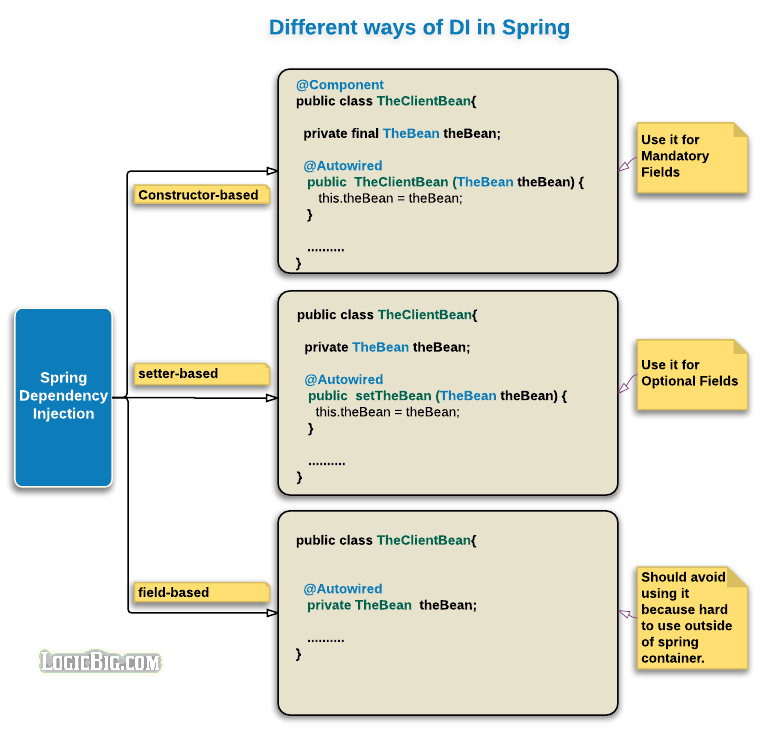




# Spring - Inject Bean By Name using @Resource



# Spring - Different ways of injecting dependencies



Type of auto-wiring means how to inject other beans in this class (no,Bytype or Byname)

Types of dependency injection means where to inject beans (on constructor, on field or on setter method)

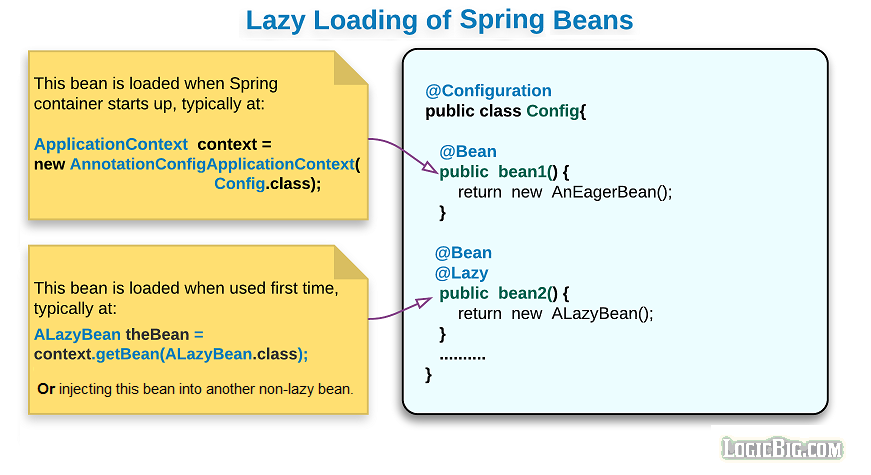
# Spring - Initialization and destruction lifecycle callbacks

There are many ways to handle lifecycle of bean

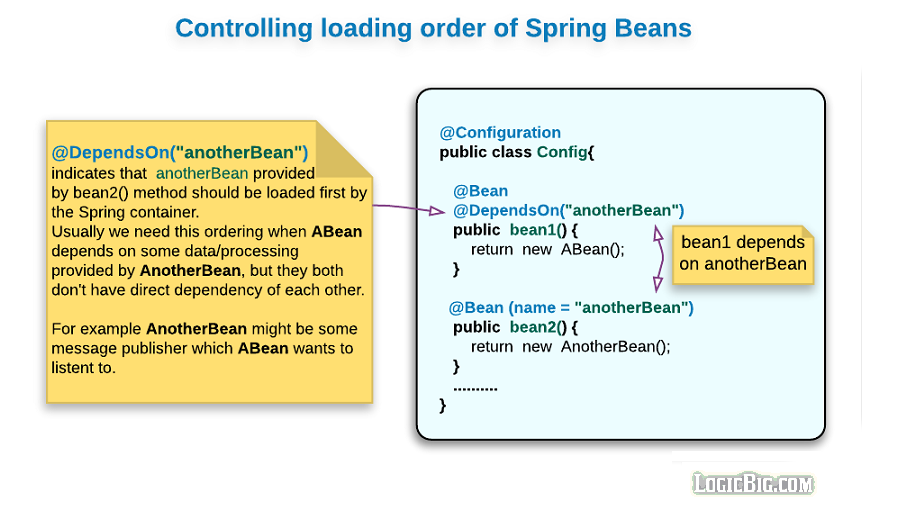
1. @Bean(initMethod = "init", destroyMethod = "destroy")
2. Using @PostConstruct and @PreDestroy
3. Implementing InitializingBean and DisposableBean

# Spring - Lazy Initialization, using @Lazy

By default Spring container instantiates all configured beans at startup (eager loading). In some situations, however, beans might rarely be used during application life cycle. Loading them at startup will, specially, be not good if they are going to use considerable memory to get initialized. In those sort of situations we may decide to initialize such beans only when they are first accessed by application code (i.e. initialize on demand). We can achieve that by using @Lazy on bean configuration method.



# Spring - Controlling Beans Loading Order, using @DependsOn



**Spring - Bean Scopes**

**singleton :** Only one instance of bean per Spring container (here container mean per org.springframework.context.ApplicationContext). That means regardless of how many times, we access/inject the bean there will be only one instance provided by the container. This is the default one.

1. **prototype :** A new instance of a bean is created, each time it's injected to some other bean or accessed via the container (springContext.getBean(...)). In what situations should we use this scope? The answer is, whenever calling code wants to set some personalized session information among multiple method calls on that bean. On the other hand a singleton is not and should not be aware of a caller specific session. In that sense we can say singletons are stateless (suitable for a service, DAO or controller etc) , whereas prototypes are stateful with respect to a particular calling session (for example shopping cart, wizard like steps etc). Why don't we create such beans ourselves using **new** operator rather than registering it to the Spring container? Yes we should if we can, but what if we want to conveniently have Spring to do some DI for us. Well, then of course we should use this scope.

[**Using on bean factory methods of @Configuration class**](https://www.logicbig.com/tutorials/spring-framework/spring-core/java-config.html)**:**

@Scope is used in @Configuration annotated class's method. These methods should primarily [be annotated with @Bean](https://www.logicbig.com/tutorials/spring-framework/spring-core/using-bean-annotation.html).

[**Using on classes annotated with @Component**](https://www.logicbig.com/tutorials/spring-framework/spring-core/javaconfig-with-componnet-scan.html)**:**

@Scope is used on component classes. This classes should be scanned by Spring at startup if @ComponentScan along with packages to scan is defined on @Configuration class.

## No Pre Destroy callback for Prototype

Spring does not manage the complete lifecycle of a prototype bean. The container instantiates, configures, a prototype bean instance, and hands it to the client, with no further record of the instance. That's the reason, the prototype bean's method annotated with PreDestroy will never be called. The initialization lifecycle callback methods (@PostConstruct) are always called on all objects regardless of scope.

Following are the three scopes available, if spring application loaded using WebApplicationContext

1. **request:**One instance per HTTP request, i.e. every new HTTP request will have its own instance of the bean.
2. **session:**One instance per HTTP session.
3. **globalSession:**One instance per global HTTP session. Typically only valid when used in a portlet context.

**Using *@Lookup***

### ****Injecting prototype-scoped bean into a singleton bean****

@Component

@Scope("prototype")

public class SchoolNotification {

    // ... prototype-scoped state

}

@Component

public class StudentServices {

    @Lookup

    public SchoolNotification getNotification() {

        return null;

    }

}

Using *@Lookup*, we can get an instance of *SchoolNotification* through our singleton bean:

@Test

public void whenLookupMethodCalled\_thenNewInstanceReturned() {

    // ... initialize context

    StudentServices first = this.context.getBean(StudentServices.class);

    StudentServices second = this.context.getBean(StudentServices.class);

    assertEquals(first, second);

    assertNotEquals(first.getNotification(), second.getNotification());

}

Note that in *StudentServices*, we left the *getNotification* method as a stub.

This is because Spring overrides the method with a call to *beanFactory.getBean(StudentNotification.class)*, so we can leave it empty.

**Request vs Prototype scope beans**

Prototype creates a brand new instance everytime you call getBean on the ApplicationContext. Whereas for Request, only one instance is created for an HttpRequest. So in a single HttpRequest, I can call getBean twice on Application and there will only ever be one bean instantiated, whereas that same bean scoped to Prototype in that same single HttpRequest would get 2 different instances.

**HttpRequest scope**

**Mark mark1 = context.getBean("mark");**

**Mark mark2 = context.getBean("mark");**

**mark1 == mark2; //This will return true**

**Prototype scope**

**Mark mark1 = context.getBean("mark");**

**Mark mark2 = context.getBean("mark");**

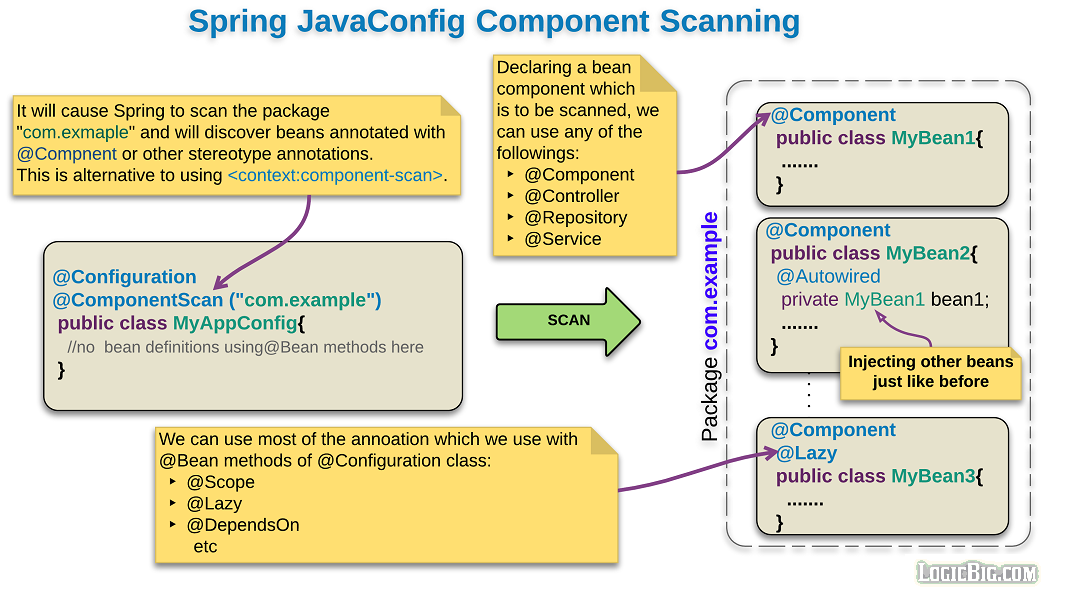
**mark1 == mark2; //This will return false**

**Spring - JavaConfig with Component Scan**

For component scanning to work we must annotate our beans with one of the stereotype annotations

1. Component
2. Controller
3. Repository
4. Service

Classes annotated with one of the above are candidate for spring container registration when using scanning.



**Spring - Implicit constructor Injection**

Starting Spring 4.3, it is no longer necessary to specify the @Autowired annotation if the target bean only defines one constructor.

public class FooService {

private final FooRepository repository;

public FooService(FooRepository repository) {

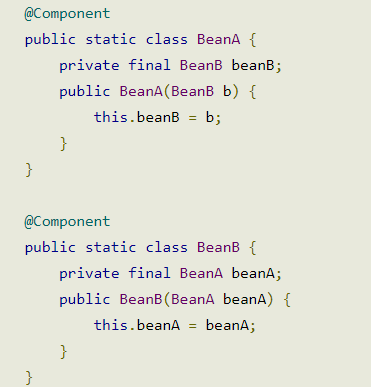
this.repository = repository

}

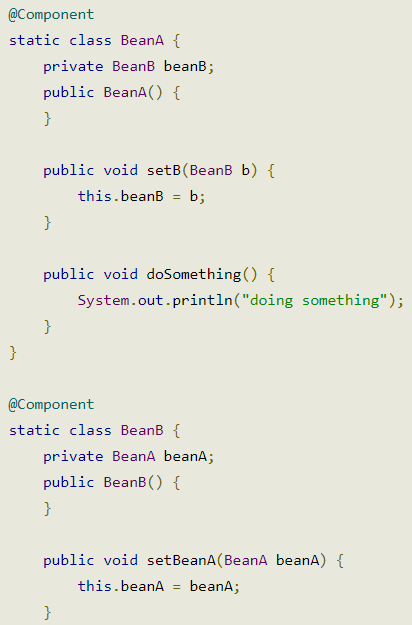
}

**Spring - Circular Dependencies**

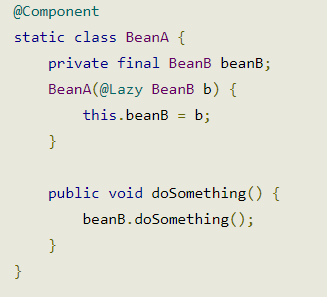
Circular dependencies is the scenario when two or more beans try to inject each other via constructor.



## Fixing circular dependencies by using setter injection



## Fixing circular dependencies by using @Lazy at constructor injection point



## Annotation-based Dependency injection

| **Annotation** |  |  | Description |
| --- | --- | --- | --- |
| **@Autowired** |  |  | If the AutowiredAnnotationBeanPostProcessor is registered with the IoCconatiner as the bean is processed, the constructor, field, or method is autowired by type. |
| **@Configurable** |  |  | Indicates a class is eligible for Spring configuration, but is typically only used with AspectJ and along with *context:spring-configured*. |
| **@Qualifier** |  |  |  |
|  |  |  |  |
| **@Resource** |  |  | If the CommonAnnotationBeanPostProcessor is installed as the bean is processed, the field or method is autowired by name. Part of JSR-250 (Commons Annotations). |

you can use **@Autowired** annotation to auto wire bean on the setter method, constructor or a field.

To enable **@Autowired**, you have to register ‘**AutowiredAnnotationBeanPostProcessor**‘, and you can do it in two ways :

##### 1. Include <context:annotation-config />

##### 2. Include AutowiredAnnotationBeanPostProcessor by below formate

<bean

class="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor"/>

#### Dependency checking

By default, the @Autowired will perform the dependency checking to make sure the property has been wired properly. When Spring can’t find a matching bean to wire, it will throw an exception. To fix it, you can disable this checking feature by setting the “**required**” attribute of @Autowired to false.

@Qualifier

The @Qualifier annotation us used to control which bean should be autowire on a field. For example, bean configuration file with two similar person beans.

Will Spring know which bean should wire?

To fix it, you can use **@Qualifier** to auto wire a particular bean, for example,

publicclass Customer

{

@Autowired

@Qualifier("PersonBean1")

private Person person;

privateint type;

privateString action;

*//getter and setter methods*

}

| **Annotation** | Since Version | Target | Description |
| --- | --- | --- | --- |
| **@Component** | Spring 2.5 | Class | Indicates this class is eligible for registration as a Spring bean when detected by a classpath scanner (*context:component-scan*). |
| **@Controller** | Spring 2.5 | Class | Indicates this class is a Web Controller eligible for registration as a Spring bean when detected by a classpath scanner (*context:component-scan*). It is a specialized form of@Component. |
| **@Repository** | Spring 2.0 | Class | Indicates this class is a Repository (DAO) eligible for registration as a Spring bean when detected by a classpath scanner (*context:component-scan*). It is a specialized form of@Component. |
| **@Service** | Spring 2.5 | Class | Indicates this class is a Service (business service facade) eligible for registration as a Spring bean when detected by a classpath scanner (*context:component-scan*). It is a specialized form of @Component. |

**The contextnamespace**

has three main elements to help with annotation-based configuration.

* component-scan*,*
* annotation-config*,*
* spring-configured*.*

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

<context:annotation-config> is used to activate annotations**(@Required, @Autowired , @Component, @Repository, @Service etc)**  in beans already registered in the application context (no matter if they were defined with XML or by package scanning).

***OR***

**activates many different annotations in beans**, whether they are **defined in XML** or through **component scanning**.

***<context:component-scan>***

<context:component-scan> is for defining beans without using XML

It  can also do what <context:annotation-config> does but it  also scans packages to find and register beans within the application

<context:component-scan/> logically **extends** <context:annotation-config/>

**What is the Core container module?**

This module is provides the fundamental functionality of the spring framework. In this module**BeanFactory** is the heart of any spring-based application. The entire framework was built on the top of this module. This module makes the ***Spring container***.

**What is a BeanFactory?**

A BeanFactory is an implementation of the factory pattern that applies Inversion of Control to separate the application’s configuration and dependencies from the actual application code.

**Q: Give an example of BeanFactory implementation.**

**A:** The most commonly used BeanFactory implementation is the **XmlBeanFactory** class. This container reads the configuration metadata from an XML file and uses it to create a fully configured system or application.

**Q: What are the common implementations of the ApplicationContext?**

**A:** The three commonly used implementation of 'Application Context' are:

* **FileSystemXmlApplicationContext:** This container loads the definitions of the beans from an XML file. Here you need to provide the full path of the XML bean configuration file to the constructor.
* **ClassPathXmlApplicationContext:** This container loads the definitions of the beans from an XML file. Here you do not need to provide the full path of the XML file but you need to set CLASSPATH properly because this container will look bean configuration XML file in CLASSPATH.
* **WebXmlApplicationContext:** This container loads the XML file with definitions of all beans from within a web application.

**What is default scope of bean in Spring framework?**

**A:** The default scope of bean is Singleton for Spring framework.

**What do you mean by Bean wiring ?**

|  |
| --- |
|  |

The act of creating associations between application components (beans) within the Spring container is reffered to as Bean wiring.

**Q: What is bean auto wiring?**

**A:** The Spring container is able to autowire relationships between collaborating beans. This means that it is possible to automatically let Spring resolve collaborators (other beans) for your bean by inspecting the contents of the BeanFactory without using <constructor-arg> and <property> elements.

## What is Spring MVC framework?

**The Spring web MVC framework provides**[**model-view-controller**](http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller)**architecture and provide the loose coupling between model, view and controller.**

**Advantages:**

1. **provide the loose coupling between model, view and controller**
2. Spring offers better integration with view technologies other than JSP (Velocity / XSLT / FreeMarker / XL etc.)
3. No ActionForms. Bind directly to domain objects
4. struts force the controller to extend Struts classes but in spring mvc we don’t have to extend anything
5. For request mapping we don’t need xml files
6. Spring mvc has built in support of JPA and ORM framework

**Configuration for web.xml file**

<servlet>

<servlet-name>dispatcher</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>dispatcher</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/dispatcher-servlet.xml</param-value>

</context-param>

<listener>

<listenerclass>

org.springframework.web.context.ContextLoaderListener

</listener-class>

</listener>

**Configuration for dispatcher-servlet.xml**

<mvc:annotation-driven/>

<context:component-scanbase-package=*"com.programcreek.helloworld"*/>

<!-- Specifying the Resource location to load JS, CSS, Images etc -->

<mvc:resourcesmapping=*"/resources/\*\*"*location=*"/WEB-INF/resources/"*/>

<beanclass=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>

<propertyname=*"prefix"*>

<value>/WEB-INF/views/</value>

</property>

<propertyname=*"suffix"*>

<value>.jsp</value>

</property>

</bean>

By default, DispatcherServlet loads its configuration file using <servlet\_name>-servlet.xml.

E.g. with above web.xml file, DispatcherServlet will try to find dispatcher-servlet.xml file in classpath.

ContextLoaderListener reads the spring configuration file (with value given against “**contextConfigLocation**” in web.xml), parse it and loads the beans defined in that config file.

# Difference between <mvc:annotation-driven /> and <context:annotation-config />

<context:annotation-config> declares support for core annotations such as @Required, @Autowired

<mvc:annotation-driven /> declares support for annotation-driven MVC controllers @RequestMapping, @Controller, @RequestBody @ ResponseBody.

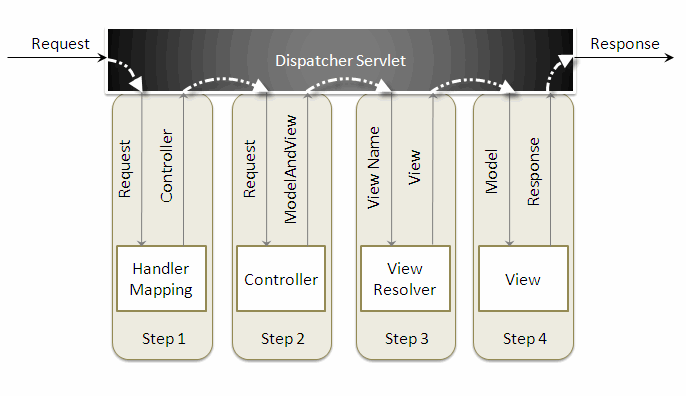
## What is the front controller class of Spring MVC?

**DispatcherServlet (actually a servlet) is the front controller in Spring MVC that intercepts every request and then dispatches/forwards requests to an appropriate controller.**

**What is view Resolver pattern? How it work in Spring MVC**

View Resolver pattern is a J2EE pattern which allows a web application to dynamically choose it's view technology e.g. HTML, JSP, Tapestry, JSF, XSLT or any other view technology. In this pattern, View resolver holds mapping of different views, controller return name of the view, which is then passed to View Resolver for selecting an appropriate view. Spring MVC framework supplies inbuilt view resolver for selecting views.

The figure below shows the flow of request in the Spring MVC Framework.



When a request is sent to the Spring MVC Framework the following sequence of events happen.

* The DispatcherServlet first receives the request.
* The DispatcherServlet consults the HandlerMapping and invokes the Controller associated with the request.
* The Controller process the request by calling the appropriate service methods and returns a ModeAndView object to the DispatcherServlet. TheModeAndView object contains the model data and the view name.
* The DispatcherServlet sends the view name to a ViewResolver to find the actual View to invoke.
* Now the DispatcherServlet will pass the model object to the View to render the result.
* The View with the help of the model data will render the result back to the user.

1. **ApplicationContext (default name for this file is applicationContext.xml)**
2. **WebApplicationContext (default name for this file is xxx-servlet.xml where xxx is the DispatcherServlet name in web.xml)**

## ApplicationContext

* applicationContext.xml is the root context configuration for every web application.
* Spring loads applicationContext.xml file and creates the ApplicationContext for the whole application.
* There will be only one application context per web application.
* If you are not explicitly declaring the context configuration file name in web.xml using the contextConfigLocation param, Spring will search for the applicationContext.xml under WEB-INF folder and throw [FileNotFoundException](https://javabeat.net/filenotfoundexception-java/) if it could not find this file.

## WebApplicationContext

* Apart from **ApplicationContext**, there can be multiple **WebApplicationContext** in a single web application.
* In simple words, each DispatcherServlet associated with single WebApplicationContext.
* xxx-servlet.xml file is specific to the [DispatcherServlet](https://javabeat.net/spring-mvc-dispatcherservlet-example/) and a web application can have more than one DispatcherServlet configured to handle the requests.
* In such scenarios, each DispatcherServlet would have a separate xxx-servlet.xml configured. But, applicationContext.xml will be common for all the [servlet](https://javabeat.net/servlets-tutorials/)configuration files.
* Spring will by default load file named “xxx-servlet.xml” from your webapps WEB-INF folder where xxx is the servlet name in web.xml.
* If you want to change the name of that file name or change the location, add initi-param with contextConfigLocation as param name.