Spring

### ****What is a Spring Framework?****

* Spring is a powerful open source, application framework created to reduce the complexity of enterprise application development.
* It is light-weighted and loosely coupled.
* It has layered architecture, which allows you to select the components to use, while also providing a cohesive framework for J2EE application development.
* Spring framework is also called the framework of frameworks as it provides support to various other frameworks such as Struts, Hibernate, Tapestry, EJB, JSF etc.

### ****What are the different features of Spring Framework?****

Following are some of the major features of Spring Framework :

* **Lightweight:** Spring is lightweight when it comes to size and transparency.
* **Inversion of control (IOC):** The objects give their dependencies instead of creating or looking for dependent objects. This is called Inversion Of Control.
* **Aspect oriented Programming (AOP):** Aspect oriented programming in Spring supports cohesive development by separating application business logic from system services.
* **Container:**Spring Framework creates and manages the life cycle and configuration of the application objects.
* **MVC Framework:** Spring Framework’s MVC web application framework is highly configurable. Other frameworks can also be used easily instead of Spring MVC Framework.
* **Transaction Management:** Generic abstraction layer for transaction management is provided by the Spring Framework. Spring’s transaction support can be also used in container less environments.
* **JDBC Exception Handling:** The JDBC abstraction layer of the Spring offers an exception hierarchy, which simplifies the error handling strategy.

### ****Modules Of Spring Framework****



**IOC Container** - As the name implies Inversion of the control means now we have inverted the control of creating the object from our own using new operator to container or framework.

Now it’s the responsibility of the container to create an object as required. We maintain one XML file where we configure our components, services, all the classes, and their property. We just need to mention which service is needed by which component and container will create the object for us.

This concept is known as dependency injection because all object dependency (resources) is injected into it by the framework.

**How many types of IOC containers are there in spring?**

**BeanFactory:** BeanFactory is like a factory class that contains a collection of beans. It instantiates the bean whenever asked for by clients.

**ApplicationContext:** It is an advanced version of IOC Container. It provides all the functionalities of Bean Factory and also provides things like AOP, Internationalization capabilities, web application context (request, session, etc).

**The ApplicationContext can be defined in two ways:**

1. using XML,
2. using @Configuration.

Once the configuration is done in any of the ways defined above, the ApplicationContext is created using new ClassPathXmlApplicationContext. The ClassPathXmlApplicationContext looks for the XML files, using this is one of the two ways. The other way is to use AnnotationConfigApplicationContext.

**Differentiate between BeanFactory and ApplicationContext.**

|  |  |
| --- | --- |
| **BeanFactory** | **ApplicationContext** |
| It is an interface defined in org.springframework.beans.factory.**BeanFactory** | It is an interface defined in org.springframework.context.**ApplicationContext** |
| It uses Lazy initialization | It uses Eager/ Aggressive initialization |
| It explicitly provides a resource object using the syntax | It creates and manages resource objects on its own |
| It doesn’t supports internationalization | It supports internationalization |
| It doesn’t supports annotation based dependency | It supports annotation based dependency |

**Some of the benefits of IoC are:**

It will minimize the amount of code in your application.

It will make your application easy to test because it doesn’t require any singletons or JNDI lookup mechanisms in your unit test cases.

It promotes loose coupling with minimal effort and least intrusive mechanism.

It supports eager instantiation and lazy loading of the services.

**Dependency Injection**- DI framework is easy to test with the mock object and more comfortable to maintain because object creation code is centralized in the framework and the client code is not littered with that.

**Dependency Injection reduces coupling between multiple objects as its dynamically injected by the framework.**

There are mainly two types of Dependency Injection: Setter Injection and Constructor Injection

**The recommended approach is to use constructor arguments for mandatory dependencies and setters for optional ones. Constructor injection allows injecting values to immutable fields and makes testing easier.**

1) The fundamental difference between setter and constructor injection, as their name implies, is How dependency is injected. Setter injection in Spring uses setter methods like setDependency() to inject dependency on any bean managed by Spring's IOC container. On the other hand, constructor injection uses the constructor to inject dependency on any Spring-managed bean.

2) Because of using the setter method, setter Injection in more readable than constructor injection in Spring configuration file usually applicationContext.xml . Since the setter method has name like setReporotService() by reading Spring XML config file you know which dependency you are setting. While in constructor injection, since it uses an index to inject the dependency, it's not as readable as setter injection and you need to refer either Java documentation or code to find which index corresponds to which property.

3) Another difference between setter vs constructor injection in Spring and one of the drawbacks of setter injection is that it does not ensures dependency Injection. You can not guarantee that certain dependency is injected or not, which means you may have an object with incomplete dependency. On the other hand, constructor Injection does not allow you to construct an object until your dependencies are ready.

4) One more drawback of setter Injection is Security. By using setter injection, you can override certain dependency which is not possible with constructor injection because every time you call the constructor, a new object is gets created.

5) there is a circular dependency between two object A and B.

If Object A and B are dependent each other i.e A is depends ob B and vice-versa. Spring throws ObjectCurrentlyInCreationException while creating objects of A and B bcz A object cannot be created until B is created and vice-versa. So spring can resolve circular dependencies through setter-injection. Objects constructed before setter methods invoked.

**What are the different components of a Spring application?**

**Interface:** It defines the functions.

**Bean class:** It contains properties, its setter and getter methods, functions etc.

**Spring Aspect Oriented Programming (AOP):** Provides the functionality of cross-cutting concerns.

**Bean Configuration File:** Contains the information of classes and how to configure them.

**User program:** It uses the function.

**Scope of a bean** - Scopes a single bean definition to any number of object instances.

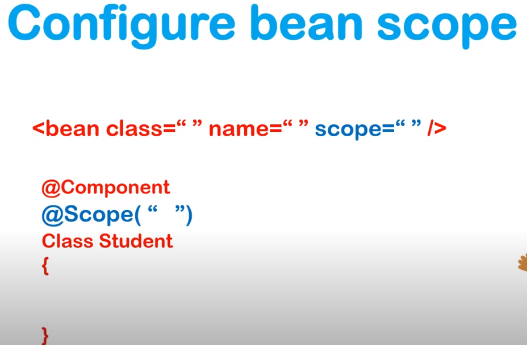
you should use the prototype scope for all beans that are stateful, while the singleton scope should be used for stateless beans.

**stateless** applications don't “store” data whereas **stateful** applications require backing storage.

**stateful**  ::

It's mainly due to synchronization issues that need to be kept in kind when using singleton beans in a multithreaded environment. With **prototype** beans each request for the **bean** will create a new instance of the **bean** and thus you don't need to care about synchronizing access to the **bean's** state across threads

|  |  |  |
| --- | --- | --- |
| **No.** | **Scope** | **Description** |
| 1) | singleton | The bean instance will be only once and same instance will be returned  by the IOC container. It is the default scope. |
| 2) | prototype | The bean instance will be created each time when requested. |
| 3) | request | The bean instance will be created per HTTP request. |
| 4) | session | The bean instance will be created per HTTP session. |
| 5) | globalsession | The bean instance will be created per HTTP global session.  It can be used in portlet context only. |



**What is a Spring configuration file?**

Spring configuration file is basically a XML file which mainly contains the classes information and describes how those classes are configured and linked to each other.

**3 ways of providing the configuration meta data to the spring container**

**XML-Based configuration:**

<bean id="interviewBitBean" class="org.intervuewBit.firstSpring.InterviewBitBean">

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

</bean>

**Annotation-Based configuration:**

The beans can be configured into the component class itself by using annotations on the relevant class, method, or field declaration.

@Configuration

public class AnnotationConfig

{

@Bean

public MyDemo myDemo()

{ return new MyDemoImpll(); }

}

Annotation wiring is not active in the Spring container by default. This has to be enabled in the Spring XML configuration file as shown below

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

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

<beans &hellip;&hellip;&hellip;&hellip; />

</beans>

**Java-based configuration:** This makes use of **the @Configuration** annotated classes and **@Bean** annotated methods. Note that:

@Bean annotation has the same role of the <bean/> element.

Classes annotated with @Configuration allows to define inter-bean dependencies by simply calling other @Bean methods in the same class.

**Explain Bean life cycle in Spring Bean Factory Container.**

* The IoC container instantiates the bean from the bean’s definition in the XML file.

Spring then populates all of the properties using the dependency injection as specified in the bean definition.

* The bean factory container calls setBeanName() which takes the bean ID and the corresponding bean has to implement BeanNameAware interface.
* The factory then calls setBeanFactory() by passing an instance of itself (if BeanFactoryAware interface is implemented in the bean).
* If BeanPostProcessors is associated with a bean, then the preProcessBeforeInitialization() methods are invoked.
* If an init-method is specified, then it will be called.
* Lastly, postProcessAfterInitialization() methods will be called if there are any BeanPostProcessors associated with the bean needs to be run post the creation.

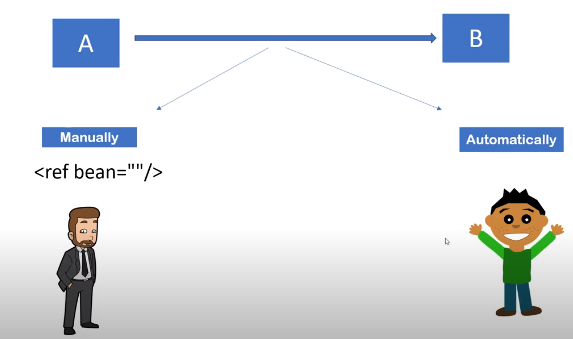
**What do you understand by Bean Wiring.**

When beans are combined together within the Spring container, it’s called wiring or bean wiring.

The Spring container needs to know what beans are needed and how the container should use dependency injection to tie the beans together, while wiring beans.

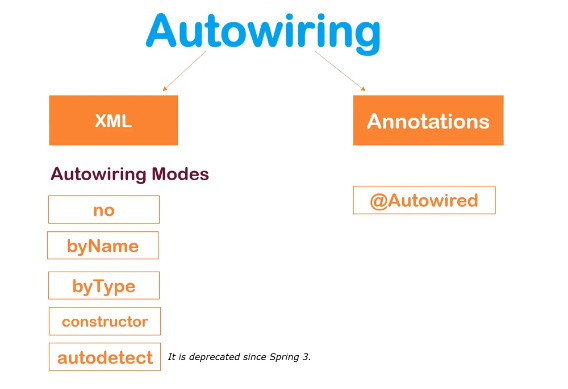
**Auto Wiring** - Autowiring enables the programmer to inject the bean automatically. We don't need to write explicit injection logic.

Autowiring can not be used to inject primitive and string values. It works with reference only.



Let's see the code to inject bean using dependency injection.

1. <bean id="emp" class="com.javatpoint.Employee" autowire="byName" />



**What are the limitations of auto wiring?**

**Overriding possibility:** Dependencies are specified using <constructor-arg> and <property> settings which overrides autowiring.

**Data types restriction:** Primitive data types, Strings and Classes can’t be autowired.

**Explain Component Scan.**

Component Scan is one method of asking Spring to detect Spring-managed components, the input for this search is the packages. Two methods are available to define a Component Scan-

(i) **Java Configuration**; wherein, we use the @Component annotation to which we specify all the packages, for which Spring does the search.

(ii**) XML Configuration**- we use <context:component-scan base-package=”com.demo.compscanex”/>

**Explain inner beans in Spring.**

A bean can be declared as an inner bean only **when it is used as a property of another bean**. For defining a bean, the Spring’s XML based configuration metadata provides the use of <bean> element inside the <property> or <constructor-arg>. **Inner beans are always anonymous and they are always scoped as prototypes.**

**Student.java**

public class Student

{

private Person person;

//Setters and Getters

}

public class Person

{

private String name;

private String address;

//Setters and Getters

}

**studentbean.xml**

<bean id=&ldquo;StudentBean" class="com.edureka.Student">

<property name="person">

<!--This is inner bean -->

<bean class="com.edureka.Person">

<property name="name" value=&ldquo;Scott"></property>

<property name="address" value=&ldquo;Bangalore"></property>

</bean>

</property>

</bean>

**Difference between @Component, @Controller, @Repository & @Service annotations in Spring.**



**@Component:** This marks a java class as a bean. It is a generic stereotype for any Spring-managed component. The component-scanning mechanism of spring now can pick it up and pull it into the application context.

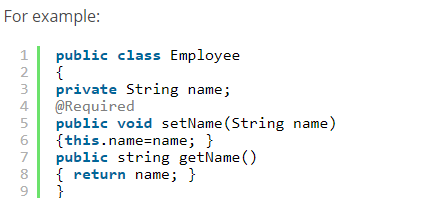
“A stereotype allows a framework developer to identify such a role and declare some common metadata for beans with that role in a central place. A bean may declare zero, one or multiple stereotypes, by applying the stereotype annotation to the bean class or producer method or field.”

**@Controller:** This marks a class as a Spring Web MVC controller. Beans marked with it are automatically imported into the Dependency Injection container.

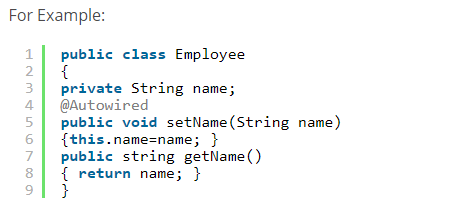
**@Service:** This annotation is a specialization of the component annotation. It doesn’t provide any additional behavior over the @Component annotation. You can use @Service over @Component in service-layer classes as it specifies intent in a better way.

**@Repository:** This annotation is a specialization of the @Component annotation with similar use and functionality. It provides additional benefits specifically for DAOs. It imports the DAOs into the DI container and makes the unchecked exceptions eligible for translation into Spring DataAccessException.

**@Required annotation:** @Required is applied to bean property setter methods. This annotation simply indicates that the affected bean property must be populated at the configuration time with the help of an explicit property value in a bean definition or with autowiring. If the affected bean property has not been populated, the container will throw BeanInitializationException.

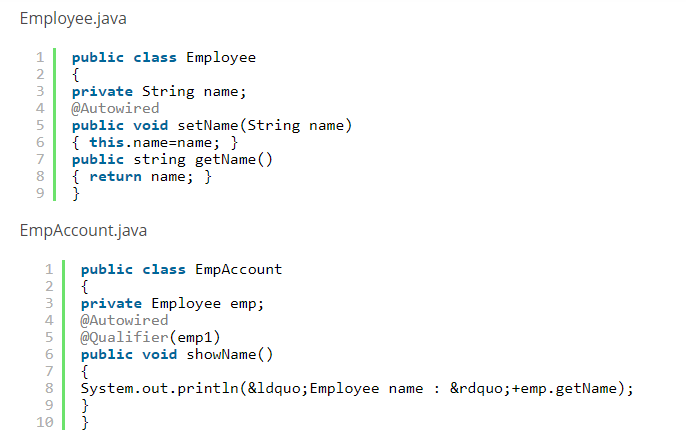


**@Autowired annotation:** The @Autowired annotation provides more accurate control over where and how autowiring should be done. This annotation is used to autowire bean on the setter methods, constructor, a property or methods with arbitrary names or multiple arguments. By default, it is a type driven injection.



**@Qualifier annotation:** When you create more than one bean of the same type and want to wire only one of them with a property you can use the @Qualifier annotation along with @Autowired to remove the ambiguity by specifying which exact bean should be wired.

For example, here we have two classes, Employee and EmpAccount respectively. In EmpAccount, using @Qualifier its specified that bean with id emp1 must be wired.



**@RequestMapping annotation:** @RequestMapping annotation is used for mapping a particular HTTP request method to a specific class/ method in controller that will be handling the respective request. This annotation can be applied at both levels:

**Class level :** Maps the URL of the request

**Method level:** Maps the URL as well as HTTP request method