**What are the benefits of using Spring Framework?**

Following is the list of few of the great benefits of using Spring Framework --

* With the Dependency Injection(DI) approach, dependencies are explicit and evident in constructor or JavaBean properties.
* IoC containers tend to be lightweight, especially when compared to EJB containers, for example. This is beneficial for developing and deploying applications on computers with limited memory and CPU resources.
* Spring does not reinvent the wheel instead, it truly makes use of some of the existing technologies like several ORM frameworks, logging frameworks, JEE, Quartz and JDK timers, other view technologies.
* Spring is organized in a modular fashion. Even though the number of packages and classes are substantial, you have to worry only about ones you need and ignore the rest.
* Testing an application written with Spring is simple because environment-dependent code is moved into this framework. Furthermore, by using JavaBean-style POJOs, it becomes easier to use dependency injection for injecting test data.
* Spring’s web framework is a well-designed web MVC framework, which provides a great alternative to web frameworks such as Struts or other over engineered or less popular web frameworks.
* Spring provides a consistent transaction management interface that can scale down to a local transaction (using a single database, for example) and scale up to global transactions (using JTA, for example).

**What is Inversion of Control (IoC) and Dependency Injection?**

 Inversion of control (IoC) is a programming technique in which object coupling is bound at run time. **Inversion of control is a design paradigm with the goal of giving more control to the targeted components of your application, the ones that are actually doing the work.**

Dependency injection is a pattern used to create instances of objects that other objects rely on without knowing at compile time which class will be used to provide that functionality. Inversion of control relies on dependency injection because a mechanism is needed in order to activate the components providing the specific functionality.

**In Java, dependency injection may happen through 3 ways:**

A constructor injection

A setter injection

An interface injection (Not supported by Spring)

**How to inject a java.util.Properties into a Spring Bean?**

First way is to use <props> tag as below.

|  |
| --- |
| <bean id="adminUser" class="com.app.common.Customer">        <!-- java.util.Properties -->      <property name="emails">          <props>              <prop key="admin">admin@nospam.com</prop>              <prop key="support">support@nospam.com</prop>          </props>      </property>    </bean> |

You can use “util:” namespace as well to create properties bean from properties file, and use bean reference for setter injection.

|  |
| --- |
| <util:properties id="emails" location="com/app/emails.properties" /> |

**How do you turn on annotation based autowiring?**

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

1. Include <context:annotation-config > in bean configuration file.

|  |
| --- |
| <beans>      <context:annotation-config />  </beans> |

2. Include AutowiredAnnotationBeanPostProcessor directly in bean configuration file.

|  |
| --- |
| <beans>      <bean class="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor"/>  </beans> |

**Explain @Required annotation with example?**

we can set “dependency-check” attribute of <bean> and set one of four attributes i.e. none, simple, objects or all (none is default option).

In case one application, we will not be interested in checking all the bean properties configured in your context files. Rather you would like to check if particular set of properties have been set or not in some specific beans only. Spring’s dependency checking feature using “dependency-check” attribute, will not able to help you in this case. So solve this problem, you can use @Required annotation.

To Use the @Required annotation over setter method of bean property in class file as below:

|  |
| --- |
| public class EmployeeFactoryBean {  @Required  private String designation;        public String getDesignation() {          return designation;      }            public void setDesignation(String designation) {          this.designation = designation;      }        //more code here  } |

**Explain @Autowired annotation with example?**

The @Autowired annotation provides more fine-grained control over where and how autowiring should be accomplished. The @Autowired annotation can be used to autowire bean on the setter method just like@Required annotation, constructor, a property or methods with arbitrary names and/or multiple arguments.

E.g. You can use @Autowired annotation on setter methods to get rid of the <property> element in XML configuration file. When Spring finds an @Autowired annotation used with setter methods, it tries to performbyType autowiring on the method.

You can apply @Autowired to constructors as well. A constructor @Autowired annotation indicates that the constructor should be autowired when creating the bean, even if no <constructor-arg> elements are used while configuring the bean in XML file.

|  |
| --- |
| public class TextEditor {     private SpellChecker spellChecker;       @Autowired     public TextEditor(SpellChecker spellChecker){        System.out.println("Inside TextEditor constructor." );        this.spellChecker = spellChecker;     }       public void spellCheck(){        spellChecker.checkSpelling();     }  } |

And it’s configuration without constructor arguments.

|  |
| --- |
| <beans>       <context:annotation-config/>       <!-- Definition for textEditor bean without constructor-arg  -->     <bean id="textEditor" class="com.howtodoinjava.TextEditor">     </bean>       <!-- Definition for spellChecker bean -->     <bean id="spellChecker" class="com.howtodoinjava.SpellChecker">     </bean>    </beans> |

**Explain @Qualifier annotation with example?**

@Qualifier means, which bean is qualify to autowired on a field. The qualifier annotation helps disambiguate bean references when Spring would otherwise not be able to do so.

See below example, it will autowired a “person” bean into customer’s person property.

|  |
| --- |
| public class Customer  {      @Autowired      private Person person;  } |

And we have two bean definitions for Person class.

|  |
| --- |
| <bean id="customer" class="com.howtodoinjava.common.Customer" />    <bean id="personA" class="com.howtodoinjava.common.Person" >      <property name="name" value="lokesh" />  </bean>    <bean id="personB" class="com.howtodoinjava.common.Person" >      <property name="name" value="alex" />  </bean> |

Will Spring know which person bean should autowired? NO. When you run above example, it hits below exception :

|  |
| --- |
| Caused by: org.springframework.beans.factory.NoSuchBeanDefinitionException:      No unique bean of type [com.howtodoinjava.common.Person] is defined:          expected single matching bean but found 2: [personA, personB] |

To fix above problem, you need @Quanlifier to tell Spring about which bean should autowired.

|  |
| --- |
| public class Customer  {      @Autowired      @Qualifier("personA")      private Person person;  } |

**Difference between constructor injection and setter injection?**

In Setter Injection, partial injection of dependencies can possible, means if we have 3 dependencies like int, string, long, then its not necessary to inject all values if we use setter injection. If you are not inject it will takes default values for those primitives. In constructor injection, partial injection of dependencies is not possible, because for calling constructor we must pass all the arguments right, if not so we may get error.

Setter Injection will overrides the constructor injection value, provided if we write setter and constructor injection for the same property. But, constructor injection cannot overrides the setter injected values. It’s obvious because constructors are called to first to create the instance.

Using setter injection you can not guarantee that certain dependency is injected or not, which means you may have an object with incomplete dependency. On other hand constructor Injection does not allow you to construct object, until your dependencies are ready.

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

**How do you define the scope of a bean?**

When defining a <bean> in Spring, we can also declare a scope for the bean. It can be defined through the scope attribute in the bean definition. For example, when Spring has to produce a new bean instance each time one is needed, the bean’sscope attribute to be prototype. On the other hand, when the same instance of a bean must be returned by Spring every time it is needed, the the bean scope attribute must be set to singleton.

**Explain the bean scopes supported by Spring**

There are five scoped provided by the Spring Framework supports following five scopes:

In singleton scope, Spring scopes the bean definition to a single instance per Spring IoC container.

In prototype scope, a single bean definition has any number of object instances.

In request scope, a bean is defined to an HTTP request. This scope is valid only in a web-aware Spring ApplicationContext.

In session scope, a bean definition is scoped to an HTTP session. This scope is also valid only in a web-aware Spring ApplicationContext.

In global-session scope, a bean definition is scoped to a global HTTP session. This is also a case used in a web-aware Spring ApplicationContext.

The default scope of a Spring Bean is Singleton.

**What are inner beans in Spring?**

When a bean is only used as a property of another bean it can be declared as an inner bean. Spring’s XML-based configuration metadata provides the use of <bean/> element inside the <property/> or <constructor-arg/> elements of a bean definition, in order to define the so-called inner bean. Inner beans are always anonymous and they are always scoped as prototypes.

**Explain different modes of auto wiring?**

The autowiring functionality has five modes which can be used to instruct Spring container to use autowiring for dependency injection:

no: This is default setting. Explicit bean reference should be used for wiring.

byName: When autowiring byName, the Spring container looks at the properties of the beans on which autowireattribute is set to byName in the XML configuration file. It then tries to match and wire its properties with the beans defined by the same names in the configuration file.

byType: When autowiring by datatype, the Spring container looks at the properties of the beans on which autowireattribute is set to byType in the XML configuration file. It then tries to match and wire a property if its type matches with exactly one of the beans name in configuration file. If more than one such beans exist, a fatal exception is thrown.

constructor: This mode is similar to byType, but type applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised.

autodetect: Spring first tries to wire using autowire by constructor, if it does not work, Spring tries to autowire bybyType.