**What is Spring Framework? \*\*\***

Spring Framework is a Java platform that provides comprehensive infrastructure support for developing Java applications. Spring handles the infrastructure so you can focus on your application.

Spring enables you to build applications from “plain old Java objects” (POJOs) and to apply enterprise services non-invasively to POJOs. This capability applies to the Java SE programming model and to full and partial Java EE.

## What is Inversion of Control? \*\*\*

Inversion of Control is a principle in software engineering by which the control of objects or portions of a program is transferred to a container or framework. It’s most often used in the context of object-oriented programming.

By contrast with traditional programming, in which our custom code makes calls to a library, IoC enables a framework to take control of the flow of a program and make calls to our custom code. To enable this, frameworks use abstractions with additional behavior built in. **If we want to add our own behavior, we need to extend the classes of the framework or plugin our own classes.**

The advantages of this architecture are:

* decoupling the execution of a task from its implementation
* making it easier to switch between different implementations
* greater modularity of a program
* greater ease in testing a program by isolating a component or mocking its dependencies and allowing components to communicate through contracts

Inversion of Control can be achieved through various mechanisms such as: Strategy design pattern, Service Locator pattern, Factory pattern, and Dependency Injection (DI).

IoC is also known as dependency injection (DI). It is a process whereby objects define their dependencies, that is, the other objects they work with, only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method. The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse, hence the name Inversion of Control (IoC), of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes, or a mechanism such as the Service Locator pattern.

**What do you understand by Dependency Injection? \*\*\***

Dependency Injection, an aspect of Inversion of Control (IoC), is a general concept stating that you do not create your objects manually but instead describe how they should be created. An IoC container will instantiate required classes if needed.

Dependency Injection design pattern allows us to remove the hard-coded dependencies and make our application loosely coupled, extendable and maintainable. We can implement dependency injection pattern to move the dependency resolution from compile-time to runtime.

Some of the benefits of using Dependency Injection are Separation of Concerns, Boilerplate Code reduction, Configurable components, and easy unit testing.

**What do you understand by Aspect Oriented Programming? \*\*\***

Enterprise applications have some common cross-cutting concerns that are applicable to different types of Objects and application modules, such as logging, transaction management, data validation, authentication etc. In Object Oriented Programming, modularity of application is achieved by Classes whereas in AOP application modularity is achieved by Aspects and they are configured to cut across different classes methods.

AOP takes out the direct dependency of cross-cutting tasks from classes that are not possible in normal object-oriented programming. For example, we can have a separate class for logging but again the classes will have to call these methods for logging the data.

**What is Aspect, Advice, Pointcut, JointPoint and Advice Arguments in AOP? \*\*\***

**Aspect**: Aspect is a **class** that implements cross-cutting concerns, such as transaction management. Aspects can be a normal class configured and then configured in Spring Bean configuration file or we can use Spring AspectJ support to declare a class as Aspect using @Aspect annotation.

**Advice**: Advice is the action taken for a particular join point. In terms of programming, they are methods that gets executed when a specific join point with matching pointcut is reached in the application. You can think of Advices as Spring interceptors or Servlet Filters.

**Pointcut**: Pointcut are regular expressions that are matched with join points to determine whether advice needs to be executed or not. Pointcut uses different kinds of expressions that are matched with the join points. Spring framework uses the AspectJ pointcut expression language for determining the join points where advice methods will be applied.

**Join Point**: A join point is a specific point in the application such as method execution, exception handling, changing object variable values etc. In Spring AOP a join point is always the execution of a method.

**Advice Arguments**: We can pass arguments in the advice methods. We can use args() expression in the pointcut to be applied to any method that matches the argument pattern. If we use this, then we need to use the same name in the advice method from where the argument type is determined.

**What is Spring IoC Container? \*\*\***

The **Spring IoC 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 dependency injection (DI) to manage the components that make up an application.

**Spring provides following two distinct types of containers.**

1. BeanFactory container
2. ApplicationContext container

**Some of the useful ApplicationContext implementations that we use are:**

1. [**FileSystemXmlApplicationContext**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/support/FileSystemXmlApplicationContext.html) – 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.
2. [**ClassPathXmlApplicationContext**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/support/ClassPathXmlApplicationContext.html) – 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.
3. [**WebXmlApplicationContext**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/context/support/XmlWebApplicationContext.html) – This container loads the XML file with definitions of all beans from within a web application.

**What is a Spring Bean? \*\*\***

The instance of any normal java class which form backbone of any application that is **initialized** **assembled**, and otherwise **managed** by a Spring **IoC container** is called Spring Bean. We use Spring ApplicationContext to get the Spring Bean instance.

Spring IoC container manages the life cycle of Spring Bean, bean scopes and injecting any required dependencies in the bean.

**Which is the best way of injecting beans and why? \*\*\***

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.

### Differentiate between constructor injection and setter injection.

#### Constructor Injection vs Setter Injection

|  |  |  |
| --- | --- | --- |
| Properties | Setter Injection | Constructor Injection |
| Injecting Mechanism | Setter injection in Spring uses setter methods like setDependency() to inject dependency on any bean managed by Spring's IOC container | [Constructor injection uses constructor to inject dependency on any Spring-managed bean.](http://javarevisited.blogspot.sg/2012/01/what-is-constructor-overloading-in-java.html) |
| Readability | User POJO's setter method thus it is more readable in ApplicationContecxt file | Not much readable as it uses index positions to inject dependencies. One needs to check the java class to verify if the dependencies are injected properly |
| Ensuring DI | There is no gaurantee that all the dependenies are set using setter injection as while injecting if a bean is not injected we would not know | Ensures DI as constructor Injection does not allow you to construct object, until your dependencies are ready. |
| Security | Setter Injection can override the previous dependency. | Constructor Injection is secure in these matter because every time you call the constructor, a new object is gets created. |
| When to use | When there are lot of dependencies | When there are less dependencies |

### Differentiate between BeanFactory and ApplicationContext. \*\*\*

#### BeanFactory vs 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 |

**What are different ways to configure a class as Spring Bean? \*\*\***

There are three different ways to configure Spring Bean.

1. **XML Configuration:** This is the most popular configuration and we can use bean element in context file to configure a Spring Bean. For example:

<bean name="myBean" class="com.gaurav.spring.beans.MyBean"></bean>

1. **Java Based Configuration:** If you are using only annotations, you can configure a Spring bean using @Bean annotation. This annotation is used with @Configuration classes to configure a spring bean. Sample configuration is:

@Configuration

@ComponentScan(value="com.gaurav.spring.main")

public class MyConfiguration {

@Bean

public MyService getService(){

return new MyService();

}

}

AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(MyConfiguration.class);

MyService service = ctx.getBean(MyService.class);

1. **Annotation Based Configuration:** We can also use **@Component, @Service, @Repository and @Controller** annotations with classes to configure them to be as spring bean. For these, we would need to provide base package location to scan for these classes. For example:

<context:component-scan base-package="com.gaurav.spring" />

**What are different scopes of Spring Bean? \*\*\***

There are five scopes defined for Spring Beans.

**Singleton**: Only one instance of the bean will be created for each container. This is the **default scope** for the spring beans. While using this scope, make sure spring bean doesn’t have shared instance variables otherwise it might lead to data inconsistency issues because it’s not thread-safe.

**Prototype**: A new instance will be created every time the bean is requested.

**Request**: This is same as prototype scope, however it’s meant to be used for web applications. A new instance of the bean will be created **for each HTTP request**.

**Session**: A new bean will be created for each HTTP session by the container.

**Global-Session**: This is used to create global session beans for Portlet applications.

Spring Framework is extendable and we can create our own scopes too, however most of the times we are good with the scopes provided by the framework.

To set spring bean scopes we can use “**scope**” attribute in bean element or **@Scope** annotation for annotation based configurations.

**What does the Spring bean lifecycle look like? \*\*\***

First, a Spring bean needs to be instantiated, based on Java or XML bean definition. It may also be required to perform some initialization to get it into a usable state. After that, when the bean is no longer required, it will be removed from the IoC container.



Spring framework provides following 4 ways for controlling life cycle events of a bean:

1. **InitializingBean** and **DisposableBean** callback interfaces
2. \*Aware interfaces for specific behavior
3. Custom **init**() and **destroy**() methods in bean configuration file
4. @**PostConstruct** and @**PreDestroy** annotations

**What is Bean wiring and @Autowired annotation? \*\*\***

The process of injection spring bean dependencies while initializing it called Spring Bean Wiring.

Usually, it’s best practice to do the explicit wiring of all the bean dependencies, but the spring framework also supports auto-wiring. We can use @Autowired annotation with fields or methods for autowiring byType. For this annotation to work, we also need to enable annotation-based configuration in spring bean configuration file. This can be done by context:annotation-config element.

**What are different types of Spring Bean autowiring? \*\*\***

There are four types of autowiring in Spring framework.

1. autowire **byName** – For this type of autowiring, setter method is used for dependency injection. Also the variable name should be same in the class where we will inject the dependency and in the spring bean configuration file.
2. autowire **byType** – For this type of autowiring, class type is used. So there should be only one bean configured for this type in the spring bean configuration file.
3. autowire by **constructor** – This is almost similar to autowire byType, the only difference is that constructor is used to inject the dependency.
4. autowire by **autodetect** – If you are on Spring 3.0 or older versions, this is one of the autowire options available. This option was used for autowire by constructor or byType, as determined by Spring container. Since we already have so many options, this option is deprecated.
5. **@Autowired** annotation – We can use Spring @Autowired annotation for spring bean autowiring. @Autowired annotation can be applied on variables and methods for autowiring byType. We can also use @Autowired annotation on constructor for constructor based spring autowiring.

For @Autowired annotation to work, we also need to enable annotation based configuration in spring bean configuration file. This can be done by **context:annotation-config** element or by defining a bean of type oor.

1. **@Qualifier** annotation – This annotation is used to avoid conflicts in bean mapping and we need to provide the bean name that will be used for autowiring. This way we can avoid issues where multiple beans are defined for same type. This annotation usually works with the @Autowired annotation. For constructors with multiple arguments, we can use this annotation with the argument names in the method.

**What are the limitations with auto wiring? \*\*\***

Following are some of the limitations you might face with auto wiring:

* Overriding possibility: You can always specify dependencies using <constructor-arg> and <property> settings which will override autowiring.
* Primitive data type: Simple properties such as primitives, Strings and Classes can’t be autowired.

Confusing nature: Always prefer using explicit wiring because autowiring is less precise.

**What is a Controller in Spring MVC? \*\*\***

Just like MVC design pattern, Controller is the class that takes care of all the client requests and send them to the configured resources to handle it. In Spring MVC, org.springframework.web.servlet.**DispatcherServlet** is the front controller class that initializes the context based on the spring beans configurations.

A Controller class is responsible to handle a different kind of client requests based on the request mappings. We can create a controller class by using **@Controller** annotation. Usually, it’s used with **@RequestMapping** annotation to define handler methods for specific URI mapping.

**What’s the difference between @Component, @Controller, @Repository & @Service annotations in Spring? \*\*\***

**@Component** is used to indicate that a class is a component. These classes are used for auto-detection and configured as bean when annotation based configurations are used.

**@Controller** is a specific type of component, used in MVC applications and mostly used with RequestMapping annotation.

**@Repository** annotation is used to indicate that a component is used as repository and a mechanism to store/retrieve/search data. We can apply this annotation with DAO pattern implementation classes.

**@Service** is used to indicate that a class is a Service. Usually, the business facade classes that provide some services are annotated with this.

We can use any of the above annotations for a class for auto-detection but different types are provided so that you can easily distinguish the purpose of the annotated classes.

**What is DispatcherServlet and ContextLoaderListener? \*\*\***

DispatcherServlet is the **front controller** in the Spring MVC application and it loads the spring bean configuration file and initialize all the beans that are configured. If annotations are enabled, it also scans the packages and configure any bean annotated with @Component, @Controller, @Repository or @Service annotations.

ContextLoaderListener is the listener **to start up and shut down Spring’s root WebApplicationContext**. It’s important functions are to tie up the lifecycle of ApplicationContext to the lifecycle of the ServletContext and to automate the creation of ApplicationContext. We can use it to define shared beans that can be used across different spring contexts.

**What is ViewResolver in Spring? \*\*\***

ViewResolver implementations are used to resolve the view pages by name. Usually we configure it in the spring bean configuration file. For example:

<!-- Resolves views selected for rendering by @Controllers to .jsp resources in the /WEB-INF/views directory -->

<beans:bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<beans:property name="prefix" value="/WEB-INF/views/" />

<beans:property name="suffix" value=".jsp" />

</beans:bean>

**InternalResourceViewResolver** is one of the implementation of ViewResolver interface and we are providing the view pages directory and suffix location through the bean properties. So if a controller handler method returns “home”, view resolver will use view page located at /WEB-INF/views/home.jsp.

**How to handle exceptions in Spring MVC Framework? \*\*\***

Spring MVC Framework provides the following ways to help us achieving robust exception handling.

**Controller Based** – We can define exception handler methods in our controller classes. All we need is to annotate these methods with **@ExceptionHandler** annotation.

**Global Exception Handler** – Exception Handling is a cross-cutting concern and Spring provides **@ControllerAdvice** annotation that we can use with any class to define our global exception handler.

**HandlerExceptionResolver implementation** – For generic exceptions, most of the times we serve static pages. Spring Framework provides HandlerExceptionResolver interface that we can implement to create global exception handler. The reason behind this additional way to define global exception handler is that Spring framework also provides default implementation classes that we can define in our spring bean configuration file to get spring framework exception handling benefits.

For a complete example, please read Spring Exception Handling Example.

**How to create ApplicationContext in a Java Program? \*\*\***

There are following ways to create spring context in a standalone java program.

**AnnotationConfigApplicationContext**: If we are using Spring in standalone java applications and using annotations for Configuration, then we can use this to initialize the container and get the bean objects.

**ClassPathXmlApplicationContext**: If we have spring bean configuration xml file in standalone application, then we can use this class to load the file and get the container object.

**FileSystemXmlApplicationContext**: This is similar to ClassPathXmlApplicationContext except that the xml configuration file can be loaded from anywhere in the file system.

**What is ContextLoaderListener? \*\*\***

ContextLoaderListener is the listener class used to load root context and define spring bean configurations that will be visible to all other contexts. It’s configured in web.xml file as:

<context-param>

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

<param-value>/WEB-INF/spring/root-context.xml</param-value>

</context-param>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

**What are the minimum configurations needed to create Spring MVC application? \*\*\***

For creating a simple Spring MVC application, we would need to do the following tasks.

Add **spring-context and spring-webmvc** dependencies in the project.

Configure **DispatcherServlet** in the web.xml file to handle requests through spring container.

Spring bean configuration **file** to define beans, if using annotations then it has to be configured here. Also we need to configure view resolver for view pages.

Controller class with request mappings defined to handle the client requests.

Above steps should be enough to create a simple Spring MVC Hello World application.

**How would you relate Spring MVC Framework to MVC architecture? \*\*\***

As the name suggests Spring MVC is built on top of Model-View-Controller architecture. DispatcherServlet is the Front Controller in the Spring MVC application that takes care of all the incoming requests and delegate it to different controller handler methods.

The model can be any Java Bean in the Spring Framework, just like any other MVC framework Spring provides automatic binding of form data to java beans. We can set model beans as attributes to be used in the view pages.

View Pages can be JSP, static HTMLs etc. and view resolvers are responsible for finding the correct view page. Once the view page is identified, control is given back to the DispatcherServlet controller. DispatcherServlet is responsible for rendering the view and returning the final response to the client.

**What are some of the important Spring annotations you have used? \*\*\***

Some of the Spring annotations that I have used in my project are:

@Controller – for controller classes in Spring MVC project.

@RequestMapping – for configuring URI mapping in controller handler methods. This is a very important annotation, so you should go through Spring MVC RequestMapping Annotation Examples

@ResponseBody – for sending Object as response, usually for sending XML or JSON data as response.

@PathVariable – for mapping dynamic values from the URI to handler method arguments.

@Autowired – for autowiring dependencies in spring beans.

@Qualifier – with @Autowired annotation to avoid confusion when multiple instances of bean type is present.

@Service – for service classes.

@Scope – for configuring scope of the spring bean.

@Configuration, @ComponentScan and @Bean – for java based configurations.

AspectJ annotations for configuring aspects and advices, @Aspect, @Before, @After, @Around, @Pointcut etc.

**How to validate form data in Spring Web MVC Framework? \*\*\***

Spring supports **JSR-303** annotation based validations as well as provide Validator interface that we can implement to create our own custom validator. For using JSR-303 based validation, we need to annotate bean variables with the required validations.

For custom validator implementation, we need to configure it in the controller class

**What is Spring MVC Interceptor and how to use it? \*\*\***

Spring MVC Interceptors are like Servlet Filters and allow us to intercept client request and process it. We can intercept client request at three places – **preHandle**, **postHandle** and **afterCompletion**.

We can create spring interceptor by implementing **HandlerInterceptor** interface or by extending abstract class **HandlerInterceptorAdapter**.

We need to configure interceptors in the spring bean configuration file. We can define an interceptor to intercept all the client requests or we can configure it for specific URI mapping too.

**How would you achieve Transaction Management in Spring? \*\*\***

Spring framework provides transaction management support through Declarative Transaction Management as well as programmatic transaction management. Declarative transaction management is most widely used because it’s easy to use and works in most of the cases.

We use annotate a method with **@Transactional** annotation for Declarative transaction management. We need to configure the transaction manager for the DataSource in the spring bean configuration file.

<bean id="transactionManager"

class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource" />

</bean>

**What is Spring DAO? \*\*\***

Spring DAO support is provided to work with data access technologies like JDBC, Hibernate in a consistent and easy way. For example we have JdbcDaoSupport, HibernateDaoSupport, JdoDaoSupport and JpaDaoSupport for respective technologies.

Spring DAO also provides consistency in exception hierarchy and we don’t need to catch specific exceptions.

**What is Spring Security?**

Spring security framework focuses on providing both authentication and authorization in java applications. It also takes care of most of the common security vulnerabilities such as CSRF attack.

It’s very beneficial and easy to use Spring security in web applications, through the use of annotations such as @EnableWebSecurity.

**What are some of the best practices for Spring Framework? \*\*\***

Some of the best practices for Spring Framework are:

Avoid version numbers in schema reference, to make sure we have the latest configs.

Divide spring bean configurations based on their concerns such as spring-jdbc.xml, spring-security.xml.

For spring beans that are used in multiple contexts in Spring MVC, create them in the root context and initialize with listener.

Configure bean dependencies as much as possible, try to avoid autowiring as much as possible.

For application-level properties, the best approach is to create a property file and read it in the spring bean configuration file.

For smaller applications, annotations are useful but for larger applications, annotations can become a pain. If we have all the configuration in XML files, maintaining it will be easier.

Use correct annotations for components for understanding the purpose easily. For services use @Service and for DAO beans use @Repository.

Spring framework has a lot of modules, use what you need. Remove all the extra dependencies that get usually added when you create projects through Spring Tool Suite templates.

If you are using Aspects, make sure to keep the join pint as narrow as possible to avoid advice on unwanted methods. Consider custom annotations that are easier to use and avoid any issues.

Use dependency injection when there is an actual benefit, just for the sake of loose-coupling doesn’t use it because it’s harder to maintain.