**What is Spring Framework?**

Spring is an open source development framework for Enterprise Java. The core features of the Spring Framework can be used in developing any Java application, but there are extensions for building web applications on top of the Java EE platform. Spring framework targets to make Java EE development easier to use and promote good programming practice by enabling a POJO-based programming model.

Spring is one of the most widely used Java EE frameworks. Spring framework core concepts are “Dependency Injection” and “Aspect Oriented Programming”.

Spring framework can be used in normal java applications also to achieve loose coupling between different components by implementing dependency injection and we can perform cross cutting tasks such as logging and authentication using spring support for aspect oriented programming.

I like spring because it provides a lot of features and different modules for specific tasks such as Spring MVC and Spring JDBC. Since it’s an open source framework with a lot of online resources and active community members, working with Spring framework is easy and fun at same time.

**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).

**Some of the features of spring framework are:**

•Lightweight and very little overhead of using framework for our development.

•Dependency Injection or Inversion of Control to write components that are independent of each other, spring container takes care of wiring them together to achieve our work.

•Spring IoC container manages Spring Bean life cycle and project specific configurations such as JNDI lookup.

•Spring MVC framework can be used to create web applications as well as restful web services capable of returning XML as well as JSON response.

•Support for transaction management, JDBC operations, File uploading, Exception Handling etc with very little configurations, either by using annotations or by spring bean configuration file.

**Name some of the important Spring Modules?**

Some of the important Spring Framework modules are:

•**Spring IOC** – for dependency injection.

•**Spring AOP** – for aspect oriented programming.

•**Spring DAO** – for database operations using DAO pattern

Spring DAO Support

Spring with JDBC

Spring with Hibernate

Spring with JPA

Spring with IBatis

**•Spring Web Module** – for creating web applications.

Spring with Struts1

Spring with Struts2

Spring with JSF

•**Spring MVC** – Model-View-Controller implementation for creating web applications.

**•Spring Security**

**•Spring Integrations**

Spring with Java Mail

Spring with RMI

Spring with EJB2

Spring with EJB3

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

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 is Spring IoC Container?**

**Inversion of Control** (IoC) is the mechanism to achieve loose-coupling between Objects dependencies. To achieve loose coupling and dynamic binding of the objects at runtime, the objects define their dependencies that are being injected by other assembler objects. Spring IoC container is the program that injects dependencies into an object and make it ready for our use.

Spring Framework IoC container classes are part of org.springframework.beans andorg.springframework.context packages and provides us different ways to decouple the object dependencies.

**There are basically two types of IOC Containers 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: The ApplicationContext interface is built on top of the BeanFactory interface.

**Difference between Inversion of Control vs Dependency Injection?**

IoC is a generic term meaning rather than having the application call the methods in a framework, the framework calls implementations provided by the application.

DI is a form of IoC, where implementations are passed into an object through constructors/setters/service look-ups, which the object will depend on in order to behave correctly.

IoC without using DI, for example would be the Template pattern because the implementation can only be changed through sub-classing.

DI Frameworks are designed to make use of DI and can define interfaces (or Annotations in Java) to make it easy to pass in implementations.

IoC Containers are DI frameworks that can work outside of the programming language. In some you can configure which implementations to use in metadata files (e.g. XML) which are less invasive. With some you can do IoC that would normally be impossible like inject implementation at pointcuts.

**Different Type Dependency Injection Types**

Dependency injection is classified into three categories namely,

1. Constructor Injection

2. Setter Injection

3. Field Injection (Using Annotations)

**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.

**Difference between Setter and Constructor Injection in Spring framework.**

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 constructor to inject dependency on any Spring-managed bean.

2) Because of using setter method, setter Injection in more readable than constructor injection in Spring configuration file usually applicationContext.xml . Since setter method has name e.g. 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 drawback 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 other hand constructor Injection does not allow you to construct 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) If 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.

**When to use Setter Injection over Constructor Injection in Spring.**

Setter Injection has upper hand over Constructor Injection in terms of readability. Since for configuring Spring we use XML files, readability is much bigger concern. Also drawback of setter Injection around ensuring mandatory dependency injected or not can be handled by configuring Spring to check dependency using "dependency-check" attribute of tag or tag. Another worth noting point to remember while comparing Setter Injection vs Constructor Injection is that, once number of dependency crossed a threshold e.g. 5 or 6 its handy manageable to passing dependency via constructor. Setter Injection is preferred choice when number of dependency to be injected is lot more than normal, if some of those arguments is optional than using Builder design pattern is also a good option.

In Summary, both Setter Injection and Constructor Injection has there own advantage and disadvantage. The good thing about Spring is that it doesn't restrict you to use either Setter Injection or Constructor Injection and you are free to use both of them in one Spring configuration file. Use Setter injection when a number of dependencies are more or you need readability. Use Constructor Injection when Object must be created with all of its dependency.

**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.

**Bean Instance created by Spring Container can be one of the following Scope**

|  |  |
| --- | --- |
| **Scope** | **Description** |
| Singleton | This scopes the bean definition to a single instance per Spring IoC container (default). |
| Prototype | This scopes a single bean definition to have any number of object instances. |
| Request | This scopes a bean definition to an HTTP request. Only valid in the context of a web-aware Spring ApplicationContext. |
| Session | This scopes a bean definition to an HTTP session. Only valid in the context of a web-aware Spring ApplicationContext. |
| global-session | This scopes a bean definition to a global HTTP session. Only valid in the context of a web-aware Spring ApplicationContext. |

**What is the difference between singleton and prototype bean?**

**Prototype scope** = A new object is created each time it is injected/looked up. It will use new SomeClass() each time.

**Singleton scope** = *(Default)* The same object is returned each time it is injected/looked up. Here it will instantiate one instance of SomeClass and then return it each time.

**Bean Loading Type In Spring**

Loading the bean which is configured in the Spring-context.xml file, can be done in two ways

**1. Eager loading or Aggressive loading**

**2. Lazy Loading**

**Eager loading or Aggressive loading**

In the case of Aggressive loading all the beans will be loaded and initialized at container start-up. It is called as eager or aggressive loading because of loading all the bean , instance instantiated and initializing at the container start-up only. We just have to write some configuration in the XML file to indicate container for eager loading. This loading is at the bean level .

To load any bean aggressively we have to mention lazy-init="false" .

Note that this configuration is at the bean level that means if we want to load all the configured bean to load eagerly then have to mention in each bean tag. If not mention then also by default it load eagerly.

**Lazy loading**

In the case of lazy loading all the beans will be loaded, instantiated and initialized when the container try to use them by calling getBeans() method. We just have to mention lazy-init="true" in order to load the beans lazily.

So bean will be loaded according to the configured lazy-init attribute with its corresponding value , or if not mentioned then by default it load Eagerly.

Note : All bean having singleton scope will follow aggressive loading and all bean with prototype scope will follow lazy loading.

**What are the different Modes of Autowiring?**

**Autowiring Modes:**

There are following autowiring modes which can be used to instruct Spring container to use autowiring for dependency injection. You use the **autowire** attribute of the <bean/> element to specify autowire mode for a bean definition.

|  |  |
| --- | --- |
| **Mode** | **Description** |
| No | This is default setting which means no autowiring and you should use explicit bean reference for wiring. You have nothing to do special for this wiring. This is what you already have seen in Dependency Injection chapter. |
| [byName](http://www.tutorialspoint.com/spring/spring_autowiring_byname.htm) | Autowiring by property name. Spring container looks at the properties of the beans on which *autowire* attribute 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](http://www.tutorialspoint.com/spring/spring_autowiring_bytype.htm) | Autowiring by property datatype. Spring container looks at the properties of the beans on which *autowire* attribute 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 exists, a fatal exception is thrown. |
| [constructor](http://www.tutorialspoint.com/spring/spring_autowiring_byconstructor.htm) | 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 by *byType*. |

You can use **byType** or **constructor** autowiring mode to wire arrays and other typed-collections.

**Limitations with autowiring:**

Autowiring works best when it is used consistently across a project. If autowiring is not used in general, it might be confusing to developers to use it to wire only one or two bean definitions. Though, autowiring can significantly reduce the need to specify properties or constructor arguments but you should consider the limitations and disadvantages of autowiring before using them.

|  |  |
| --- | --- |
| **Limitations** | **Description** |
| Overriding possibility | You can still specify dependencies using <constructor-arg> and <property> settings which will always override autowiring. |
| Primitive data types | You cannot autowire so-called simple properties such as primitives, Strings, and Classes. |
| Confusing nature | Autowiring is less exact than explicit wiring, so if possible prefer using explict wiring. |

**What are the different Modes of Autowiring through Annotation?**

**@Autowired** (byType)

**@Resource** (byType)

**@inject**  (byType)

**@Autowired +@Qualifier**  (byName)

**@ inject +@Qualifier** (byName)

**@Resource + Name attribute** (byName)

**Inject and Resource and Autowired annotations**

@Autowired: spring propriety annotation (as opposed to @Inject and @Resource) that inject a resource by-type, i.e. by the class of by the interface of the annotated field or contractor. In case we have few implementation of an interface or a subclass we can narrow down the selection using the @Qualifier annotation to avoid ambiguity. For a fallback match, the bean name is considered a default qualifier value. Although you can use this convention to refer to specific beans by name, @Autowired is fundamentally about type-driven injection with optional semantic qualifiers.

**@Inject**: Annotation based on JSR-330 (Dependency Injection for Java) identifies injectable constructors, methods, and fields. This annotation is an almost complete drop-in replacement for Spring’s @Autowired annotation. So, instead of using the Spring-specific @Autowired annotation, you might choose to use @Inject. One of the differences between @Autowired and @Inject is that @Inject does not have the required field so in case we fail to find a suitable object to injected it will fail while @Autowired can used required=false and allow null able field (only if required!). Advantage of @Inject annotation is that rather than inject a reference directly, you could ask @Inject to inject a Provider. The Provider interface enables, among other things, lazy injection of bean references and injection of multiple instances of a bean. In case we have few implementation of an interface or a subclass we can narrow down the selection using the @Named annotation to avoid ambiguity. @Named annotation works much like Spring’s @Qualifier

**@Resource**: annotation based on JSR-250. @Resource is quite similar to @Autowired and @Inject, but the main difference is the execution paths taken to find out the required bean to inject. @Resource will narrow down the search first by name then by type and finally by Qualifiers (ignored if match is found by name). @Autowired and @Inject will narrow down the search first by type then by qualifier and finally by the name.

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

•**AnnotationConfigApplicationContext**: For standalone java applications using annotations based configuration.

•**ClassPathXmlApplicationContext**: For standalone java applications using XML based configuration.

•**FileSystemXmlApplicationContext**: Similar to ClassPathXmlApplicationContext except that the xml configuration file can be loaded from anywhere in the file system.

AnnotationConfigWebApplicationContext and XmlWebApplicationContext for web applications

**What is a Spring Bean?**

Any normal java class that is initialized by Spring IoC container is called Spring Bean. We use SpringApplicationContext 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.

**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](http://www.journaldev.com/3358/spring-mvc-requestmapping-annotation-example-with-controller-methods-headers-params-requestparam-pathvariable)

**@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.

**What are the difference between BeanFactory and ApplicationContext in spring.**

This one is very popular spring interview question and often asks in entry level interview. ApplicationContext is preferred way of using spring because of functionality provided by it and interviewer wanted to check whether you are familiar with it or not.

|  |  |
| --- | --- |
| **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. |
| Support  many enterprise services such JNDI access, EJB integration, remoting | Doesn’t support. |

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

Enterprise applications have some common cross-cutting concerns that is applicable for 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 is 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**: Commonly Required Middle Level services which you are implementing for Your Enterprise application are called as Aspects.

Security, Transaction, Logging etc are aspects.

**Advice**: Implementing of a Middle Level Service is called as Advice.

Implementation of an Aspect is called as Advice.

Advice is a class which contains code for Aspects like Security, Transaction, Logging etc.

**Pointcut**: Collection of Join Points is called as Pointcut.

By Default, Advices will be applied for all the business operations of all the Business Services.

When you want to apply the advices for some specified business operations of specified Business Services the you must define point-cut with the required Aspectj expression.

**Join Point**: A point in the program where you want to apply advices.

Join Point is a point in the Program where you want to run Middle level Services code.

**Advisor:** Advisor is the combination of Advice and Point cut.

**Target:** Target is an Object of Your business service before applying the Advices or Advisors.

**Proxy:** Proxy is an Object of your business service after applying the Advisors or Advices.

**Weaving:** It is the process of applying the Advices or Advisors to the Target Objects at given pointcuts to get the Proxy Objects.

**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 argument type is determined.

**What is the difference between Spring AOP and AspectJ AOP?**

AspectJ is the industry-standard implementation for Aspect Oriented Programming whereas Spring implements AOP for some cases. Main differences between Spring AOP and AspectJ are:

•Spring AOP is simpler to use than AspectJ because we don’t need to worry about the weaving process.

•Spring AOP supports AspectJ annotations, so if you are familiar with AspectJ then working with Spring AOP is easier.

•Spring AOP supports only proxy-based AOP, so it can be applied only to method execution join points. AspectJ support all kinds of pointcuts.

•One of the shortcoming of Spring AOP is that it can be applied only to the beans created through Spring Context.

**What is the importance of Spring bean configuration file?**

We use Spring Bean configuration file to define all the beans that will be initialized by Spring Context. When we create the instance of Spring ApplicationContext, it reads the spring bean xml file and initialize all of them. Once the context is initialized, we can use it to get different bean instances.

Apart from Spring Bean configuration, this file also contains spring MVC interceptors, view resolvers and other elements to support annotations based configurations.

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

There are three different ways to configure Spring Bean.

**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.journaldev.spring.beans.MyBean"></bean> |

**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.journaldev.spring.main")  public class MyConfiguration {      @Bean      public MyService getService(){          return new MyService();      }  } |

**To get this bean from spring context, we need to use following code snippet:**

AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(

MyConfiguration.class);

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

**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.journaldev.spring" /> |

**How to get ServletContext and ServletConfig object in a Spring Bean?**

There are two ways to get Container specific objects in the spring bean.

Implementing Spring \*Aware interfaces, for these ServletContextAware and ServletConfigAware interfaces, for complete example of these aware interfaces.

Using @Autowired annotation with bean variable of type ServletContext and ServletConfig. They will work only in servlet container specific environment only though.

|  |  |
| --- | --- |
|  | @Autowired  ServletContext servletContext; |

**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 spring framework also supports autowiring. 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.

**Does Spring Bean provide thread safety?**

The default scope of Spring bean is singleton, so there will be only one instance per context. That means that all the having a class level variable that any thread can update will lead to inconsistent data. Hence in default mode spring beans are not thread-safe.

However we can change spring bean scope to request, prototype or session to achieve thread-safety at the cost of performance. It’s a design decision and based on the project requirements.

**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 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 @Serviceannotations.

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 theServletContext 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*.

**What is a MultipartResolver and when its used?**

MultipartResolver interface is used for uploading files – CommonsMultipartResolver andStandardServletMultipartResolver are two implementations provided by spring framework for file uploading. By default there are no multipart resolvers configured but to use them for uploading files, all we need to define a bean named “multipartResolver” with type as MultipartResolver in spring bean configurations.

Once configured, any multipart request will be resolved by the configured MultipartResolver and pass on a wrapped HttpServletRequest. Then it’s used in the controller class to get the file and process it.

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

Spring MVC Framework provides 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.

**Can we have multiple Spring configuration files?**

For Spring MVC applications, we can define multiple spring context configuration files throughcontextConfigLocation. This location string can consist of multiple locations separated by any number of commas and spaces. For example;

<servlet>

<servlet-name>appServlet</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

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

<param-value>/WEB-INF/spring/appServlet/servlet-context.xml,/WEB-INF/spring/appServlet/servlet-jdbc.xml</param-value>

</init-param>

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

</servlet>

|  |  |
| --- | --- |
|  |  |

We can also define multiple root level spring configurations and load it through context-param. For example;

<context-param>

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

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

</context-param>

Another option is to use import element in the context configuration file to import other configurations, for example:

|  |  |
| --- | --- |
|  | <beans:import resource="spring-jdbc.xml"/> |

**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 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.

**Explain Flow of Spring MVC ?**

Client performs a request to an URL

The application server gets the URL and passes the handling to the web application.

The web application using Spring MVC will handle the URL processing to the Controller: DispatchServlet, which is a Servlet.

The DispatchServlet will try handle the URL. If there's an URL mapping, then it will pass it to the class (mapped in the spring.xml config or decorated with @Controller annotation).

This controller (which in fact is part of the model) will handle the request. It will call services, daos, etc (Model) and return the necessary data to complete the response to the DispatchServlet.

The DispatchServlet will finish the request handling and, in the end, will generate the results e.g. a text/json response, or it will forward to a JSP file (View).

**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.

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.

**How to achieve localization in Spring MVC applications?**

Spring provides excellent support for localization or i18n through resource bundles. Basis steps needed to make our application localized are:

Creating message resource bundles for different locales, such as messages\_en.properties, messages\_fr.properties etc.

Defining messageSource bean in the spring bean configuration file of typeResourceBundleMessageSource or ReloadableResourceBundleMessageSource.

For change of locale support, define localeResolver bean of type CookieLocaleResolver and configure LocaleChangeInterceptor interceptor. Example configuration can be like below:

<beans:bean id="messageSource" class="org.springframework.context.support.ReloadableResourceBundleMessageSource">

<beans:property name="basename" value="classpath:messages" />

<beans:property name="defaultEncoding" value="UTF-8" />

</beans:bean>

<beans:bean id="localeResolver"

class="org.springframework.web.servlet.i18n.CookieLocaleResolver">

<beans:property name="defaultLocale" value="en" />

<beans:property name="cookieName" value="myAppLocaleCookie"></beans:property>

<beans:property name="cookieMaxAge" value="3600"></beans:property>

</beans:bean>

<interceptors>

<beans:bean

class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">

<beans:property name="paramName" value="locale" />

</beans:bean>

</interceptors>

Use spring:message element in the view pages with key names, DispatcherServlet picks the corresponding value and renders the page in corresponding locale and return as response.

**How to upload file in Spring MVC Application?**

Spring provides built-in support for uploading files through **MultipartResolver** interface implementations. It’s very easy to use and requires only configuration changes to get it working. Obviously we would need to write controller handler method to handle the incoming file and process it. For a complete example, please refer [Spring File Upload Example](http://www.journaldev.com/2573/spring-mvc-file-upload-example-tutorial-single-and-multiple-files).

**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.

**What is Spring JdbcTemplate class and how to use it?**

Spring Framework provides excellent integration with JDBC API and provides JdbcTemplate utility class that we can use to avoid bolier-plate code from our database operations logic such as Opening/Closing Connection, ResultSet, PreparedStatement etc.\

**How to use Tomcat JNDI DataSource in Spring Web Application?**

For using servlet container configured JNDI DataSource, we need to configure it in the spring bean configuration file and then inject it to spring beans as dependencies. Then we can use it withJdbcTemplate to perform database operations.

Sample configuration would be:

<beans:bean id="dbDataSource" class="org.springframework.jndi.JndiObjectFactoryBean">

    <beans:property name="jndiName" value="java:comp/env/jdbc/MyLocalDB"/>

</beans:bean>

|  |  |
| --- | --- |
|  |  |

**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 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 andJpaDaoSupport for respective technologies.

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

**How to integrate Spring and Hibernate Frameworks?**

We can use Spring ORM module to integrate Spring and Hibernate frameworks, if you are using Hibernate 3+ where SessionFactory provides current session, then you should avoid usingHibernateTemplate or HibernateDaoSupport classes and better to use DAO pattern with dependency injection for the integration.

Also Spring ORM provides support for using Spring declarative transaction management, so you should utilize that rather than going for hibernate boiler-plate code for transaction management.

For better understanding you should go through following tutorials:

* + [Spring Hibernate Integration Example](http://www.journaldev.com/3524/spring-hibernate-integration-example-tutorial-spring-4-hibernate-3-and-hibernate-4)
  + [Spring MVC Hibernate Integration Example](http://www.journaldev.com/3531/spring-mvc-hibernate-mysql-integration-crud-example-tutorial)

**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. You should go through following posts to learn how to use Spring Security framework.

* + [Spring Security in Servlet Web Application](http://www.journaldev.com/2715/spring-security-in-servlet-web-application-using-dao-jdbc-in-memory-authentication)
  + [Spring MVC and Spring Security Integration Example](http://www.journaldev.com/2736/spring-mvc-security-example-using-in-memory-userdetailsservice-and-jdbc-authentication)

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

We need to define propertyConfigurer bean that will load the properties from the given property file. Then we can use Spring EL support to inject properties into other bean dependencies. For example;

|  |  |
| --- | --- |
|  | <bean id="propertyConfigurer"    class="org.springframework.context.support.PropertySourcesPlaceholderConfigurer">      <property name="location" value="/WEB-INF/application.properties" />  </bean>  <bean class="com.journaldev.spring.EmployeeDaoImpl">      <property name="maxReadResults" value="${results.read.max}"/>  </bean> |

If you are using annotation to configure the spring bean, then you can inject property like below.

@Value("${maxReadResults}")

private int maxReadResults;

**Name some of the design patterns used in Spring Framework?**

Spring Framework is using a lot of design patterns, some of the common ones are:

[Singleton Pattern](http://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-with-examples): Creating beans with default scope.

[Factory Pattern](http://www.journaldev.com/1392/factory-design-pattern-in-java): Bean Factory classes

[Prototype Pattern](http://www.journaldev.com/1440/prototype-pattern-in-java): Bean scopes

[Adapter Pattern](http://www.journaldev.com/1487/adapter-design-pattern-in-java-example-tutorial): Spring Web and Spring MVC

[Proxy Pattern](http://www.journaldev.com/1572/proxy-design-pattern-in-java-example-tutorial): Spring Aspect Oriented Programming support

[Template Method Pattern](http://www.journaldev.com/1763/template-method-design-pattern-in-java): JdbcTemplate, HibernateTemplate etc

Front Controller: Spring MVC DispatcherServlet

Data Access Object: Spring DAO support

Dependency Injection and Aspect Oriented Programming.

**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, 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 gets 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 actual benefit, just for the sake of loose-coupling don’t use it because it’s harder to maintain.