Spring Interview Questions and Answers

What is Spring Framework?
Spring is one of the most widely used Java EE framework. 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.

What are some of the important features and advantages of Spring Framework?
Spring Framework is built on top of two design concepts – Dependency Injection and Aspect Oriented Programming.

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.

Some of the advantages of using Spring Framework are:

- Reducing direct dependencies between different components of the application, usually Spring IoC container is responsible for initializing resources or beans and inject them as dependencies.
- Writing unit test cases are easy in Spring framework because our business logic doesn't have direct dependencies with actual resource implementation classes. We can easily write a test configuration and inject our mock beans for testing purposes.
- Reduces the amount of boiler-plate code, such as initializing objects, open/close resources. I like JdbcTemplate class a lot because it helps us in removing all the boiler-plate code that comes with JDBC programming.
- Spring framework is divided into several modules, it helps us in keeping our application lightweight. For example, if we don't need Spring transaction management features, we don't need to add that dependency in our project.
- Spring framework support most of the Java EE features and even much more. It's always on top of the new technologies, for example there is a Spring project for Android to help us write better code for native android applications. This makes spring framework a complete package and we don't need to look after different framework for different requirements.

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.

How do we implement DI in Spring Framework?
We can use Spring XML based as well as Annotation based configuration to implement DI in spring applications.

Name some of the important Spring Modules?
Some of the important Spring Framework modules are:
- Spring Context – for dependency injection.
- Spring AOP – for aspect oriented programming.
- Spring DAO – for database operations using DAO pattern
- Spring JDBC – for JDBC and DataSource support.
- Spring ORM – for ORM tools support such as Hibernate
- Spring Web Module – for creating web applications.
- Spring MVC – Model-View-Controller implementation for creating web applications, web services etc.

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**: 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 is 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 the specific point in the application such as method execution, exception handling, changing object variable values etc. In Spring AOP a join points 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 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 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 and org.springframework.context packages and provides us different ways to decouple the object dependencies.

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 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.

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.

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

2. 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);
```

3. 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" />
```

What are different scopes of Spring Bean?
There are five scopes defined for Spring Beans.

1. **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.
2. **prototype**: A new instance will be created every time the bean is requested.
3. **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.
4. **session**: A new bean will be created for each HTTP session by the container.
5. **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 scopes annotation
You can also use annotation to define your bean scope.

```
package com.mkyong.customer.services;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Service;

@Service
@Scope("prototype")
public class CustomerService
{
	String message;

	public String getMessage() {
		return message;
	}

	public void setMessage(String message) {
		this.message = message;
	}
}
```

Enable auto component scanning

```
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xmlns:context="http://www.springframework.org/schema/context"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
	http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
	http://www.springframework.org/schema/context
	http://www.springframework.org/schema/context/spring-context-2.5.xsd">

    <context:component-scan base-package="com.mkyong.customer" />

</beans>
```

What is Spring Bean life cycle?
Spring Beans are initialized by Spring Container and all the dependencies are also injected. When context is destroyed, it also destroys all the initialized beans. This works well in most of the cases but sometimes we want to initialize other resources or do some validation before making our beans ready to use. Spring framework provides support for post-initialization and pre-destroy methods in spring beans.

We can do this by two ways – by implementing InitializingBean and DisposableBean interfaces or using init-method and destroy-method attribute in spring bean configurations. For more details, please read Spring Bean Life Cycle Methods.

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, please read Spring 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.

What are different types of Spring Bean autowiring?
Following are the autowiring modes, which can be used to instruct the 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.

1. **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.

2. **byName**
   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.

3. **byType**
   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.

4. **constructor**
   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.

5. **autodetect**
   Spring first tries to wire using autowire by constructor, if it does not work, Spring tries to autowire by byType.

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 @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.

What is a MultipartResolver and when its used?
MultipartResolver interface is used for uploading files – CommonsMultipartResolver and StandardServletMultipartResolver 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.

1. **Controller Based** – We can define exception handler methods in our controller classes. All we need is to annotate these methods with **@ExceptionHandler** annotation.
2. **Global Exception Handle**r – 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.
3. **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.

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.**

Can we have multiple Spring configuration files?
For Spring MVC applications, we can define multiple spring context configuration files through **contextConfigLocation**. 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.
1. Add **spring-context** and **spring-webmvc** dependencies in the project.
2. Configure **DispatcherServlet** in the web.xml file to handle requests through spring container.
3. 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.
4. Controller class with request mappings defined to handle the client requests.

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 can we use Spring to create Restful Web Service returning JSON response?
We can use Spring Framework to create Restful web services that returns JSON data. Spring provides integration with Jackson JSON API that we can use to send JSON response in restful web service.

We would need to do following steps to configure our Spring MVC application to send JSON response:

Adding Jackson JSON dependencies, if you are using Maven it can be done with following code:

```
<!-- Jackson -->
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>${jackson.databind-version}</version>
</dependency>
```

Configure **RequestMappingHandlerAdapter** bean in the spring bean configuration file and set the **messageConverters** property to **MappingJackson2HttpMessageConverter** bean. Sample configuration will be:

```
<!-- Configure to plugin JSON as request and response in method handler -->
<beans:bean
class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdap-
ter">
    <beans:property name="messageConverters">
        <beans:list>
            <beans:ref bean="jsonMessageConverter"/>
        </beans:list>
    </beans:property>
</beans:bean>

<!-- Configure bean to convert JSON to POJO and vice versa -->
<beans:bean id="jsonMessageConverter"
class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter">
</beans:bean>
```

In the controller handler methods, return the Object as response using **@ResponseBody** annotation. Sample code:

```
    @RequestMapping(value = EmpRestURIConstants.GET_EMP, method =
RequestMethod.GET)
    public @ResponseBody Employee getEmployee(@PathVariable("id") int empId) {
        logger.info("Start getEmployee. ID="+empId);

        return empData.get(empId);
    }
```

You can invoke the rest service through any API, but if you want to use Spring then we can easily do it using **RestTemplate** class.

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.
- **@RequestMappin**g – 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.

Can we send an Object as the response of Controller handler method?
Yes we can, using @ResponseBody annotation. This is how we send JSON or XML based response in restful web services.

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 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 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.

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:** Creating beans with default scope.
- **Factory Pattern:** Bean Factory classes
- **Prototype Pattern**: Bean scopes
- **Adapter Pattern:** Spring Web and Spring MVC
- **Proxy Pattern:** Spring Aspect Oriented Programming support
- T**emplate Method Pattern:** 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.

Injecting Prototype Beans into a Singleton Instance in Spring

1. Overview

In this quick article, we're going to show different approaches of injecting prototype beans into a singleton instance. We'll discuss the use cases and the advantages/disadvantages of each scenario.

By default, Spring beans are singletons. The problem arises when we try to wire beans of different scopes. For example, a prototype bean into a singleton. This is known as the scoped bean injection problem.

2. Prototype Bean Injection Problem

In order to describe the problem, let's configure the following beans:

```
@Configuration
public class AppConfig {

    @Bean
    @Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
    public PrototypeBean prototypeBean() {
        return new PrototypeBean();
    }

    @Bean
    public SingletonBean singletonBean() {
        return new SingletonBean();
    }
}
```

Notice that the first bean has a prototype scope, the other one is a singleton.

Now, let's inject the prototype-scoped bean into the singleton – and then expose if via the getPrototypeBean() method:

```
public class SingletonBean {

    // ..

    @Autowired
    private PrototypeBean prototypeBean;

    public SingletonBean() {
        logger.info("Singleton instance created");
    }

    public PrototypeBean getPrototypeBean() {
        logger.info(String.valueOf(LocalTime.now()));
        return prototypeBean;
    }
}
```

Then, let's load up the ApplicationContext and get the singleton bean twice:

```
public static void main(String[] args) throws InterruptedException {
    AnnotationConfigApplicationContext context
      = new AnnotationConfigApplicationContext(AppConfig.class);

    SingletonBean firstSingleton = context.getBean(SingletonBean.class);
    PrototypeBean firstPrototype = firstSingleton.getPrototypeBean();

    // get singleton bean instance one more time
    SingletonBean secondSingleton = context.getBean(SingletonBean.class);
    PrototypeBean secondPrototype = secondSingleton.getPrototypeBean();

    isTrue(firstPrototype.equals(secondPrototype), "The same instance should be returned");
}
```

Here's the output from the console:

```
Singleton Bean created
Prototype Bean created
11:06:57.894
// should create another prototype bean instance here
11:06:58.895
```

Both beans were initialized only once, at the startup of the application context.

Solution are as follows

1. Injecting ApplicationContext

We can also inject the ApplicationContext directly into a bean.

To achieve this, either use the @Autowire annotation or implement the ApplicationContextAware interface:

```
public class SingletonAppContextBean implements ApplicationContextAware {
     private ApplicationContext applicationContext;

    public PrototypeBean getPrototypeBean() {
        return applicationContext.getBean(PrototypeBean.class);
    }

    @Override
    public void setApplicationContext(ApplicationContext applicationContext)
      throws BeansException {
        this.applicationContext = applicationContext;
    }
}
```

Every time the getPrototypeBean() method is called, a new instance of PrototypeBean will be returned from the ApplicationContext.

However, this approach has serious disadvantages. It contradicts the principle of inversion of control, as we request the dependencies from the container directly.

Also, we fetch the prototype bean from the applicationContext within the SingletonAppcontextBean class. This means coupling the code to the Spring Framework.

2. Method Injection
Another way to solve the problem is method injection with the @Lookup annotation:

```
@Component
public class SingletonLookupBean {

   @Lookup
   public PrototypeBean getPrototypeBean() {
      return null;
   }
}
```

Spring will override the getPrototypeBean() method annotated with @Lookup. It then registers the bean into the application context. Whenever we request the getPrototypeBean() method, it returns a new PrototypeBean instance.

It will use CGLIB to generate the bytecode responsible for fetching the PrototypeBean from the application context.

3. javax.inject API
The setup along with required dependencies are described in this Spring wiring article.

Here’s the singleton bean:

```
public class SingletonProviderBean {

   @Autowired
   private Provider<PrototypeBean> myPrototypeBeanProvider;

   public PrototypeBean getPrototypeInstance() {
      return myPrototypeBeanProvider.get();
   }
}
```

We use Provider interface to inject the prototype bean. For each getPrototypeInstance() method call, the myPrototypeBeanProvider.get() method returns a new instance of PrototypeBean.

4.Scoped Proxy
By default, Spring holds a reference to the real object to perform the injection. Here, we create a proxy object to wire the real object with the dependent one.

Each time the method on the proxy object is called, the proxy decides itself whether to create a new instance of the real object or reuse the existing one.

To set up this, we modify the Appconfig class to add a new @Scope annotation:

```
@Scope(
  value = ConfigurableBeanFactory.SCOPE_PROTOTYPE,
  proxyMode = ScopedProxyMode.TARGET_CLASS)
```

By default, Spring uses CGLIB library to directly subclass the objects. To avoid CGLIB usage, we can configure the proxy mode with ScopedProxyMode.INTERFACES, to use the JDK dynamic proxy instead.

5.ObjectFactory Interface
Spring provides the ObjectFactory<T> interface to produce on demand objects of the given type:

```
public class SingletonObjectFactoryBean {

    @Autowired
    private ObjectFactory<PrototypeBean> prototypeBeanObjectFactory;

    public PrototypeBean getPrototypeInstance() {
        return prototypeBeanObjectFactory.getObject();
    }
}
```

Let’s have a look at getPrototypeInstance() method; getObject() returns a brand new instance of PrototypeBean for each request. Here, we have more control over initialization of the prototype.

Also, the ObjectFactory is a part of the framework; this means avoiding additional setup in order to use this option.

AOP with Spring Framework

One of the key components of Spring Framework is the Aspect oriented programming (AOP) framework. Aspect-Oriented Programming entails breaking down program logic into distinct parts called so-called concerns. The functions that span multiple points of an application are called cross-cutting concerns and these cross-cutting concerns are conceptually separate from the application's business logic. There are various common good examples of aspects like logging, auditing, declarative transactions, security, caching, etc.

The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. Dependency Injection helps you decouple your application objects from each other and AOP helps you decouple cross-cutting concerns from the objects that they affect. AOP is like triggers in programming languages such as Perl, .NET, Java, and others.

Spring AOP module provides interceptors to intercept an application. For example, when a method is executed, you can add extra functionality before or after the method execution.
AOP Terminologies

Before we start working with AOP, let us become familiar with the AOP concepts and terminology. These terms are not specific to Spring, rather they are related to AOP.

Sr.No	Terms & Description
1	**Aspect** This is a module which has a set of APIs providing cross-cutting requirements. For example, a logging module would be called AOP aspect for logging. An application can have any number of aspects depending on the requirement.
2	**Join point** This represents a point in your application where you can plug-in the AOP aspect. You can also say, it is the actual place in the application where an action will be taken using Spring AOP framework.
3	**Advice** This is the actual action to be taken either before or after the method execution. This is an actual piece of code that is invoked during the program execution by Spring AOP framework.
4	**Pointcut** This is a set of one or more join points where an advice should be executed. You can specify pointcuts using expressions or patterns as we will see in our AOP examples.
5	**Introduction** An introduction allows you to add new methods or attributes to the existing classes.
6	**Target object** The object being advised by one or more aspects. This object will always be a proxied object, also referred to as the advised object.
7	**Weaving** Weaving is the process of linking aspects with other application types or objects to create an advised object. This can be done at compile time, load time, or at runtime.

Types of Advice

Spring aspects can work with five kinds of advice mentioned as follows −

Sr.No	Advice & Description
1	**before** Run advice before the a method execution.
2	**after** Run advice after the method execution, regardless of its outcome.
3	**after-returning** Run advice after the a method execution only if method completes successfully.
4	**after-throwing** Run advice after the a method execution only if method exits by throwing an exception.
5	**around** Run advice before and after the advised method is invoked.

Here is the content of Logging.java file. This is actually a sample of aspect module which defines methods to be called at various points.

```
package com.tutorialspoint;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Around;

@Aspect
public class Logging {
  /** Following is the definition for a pointcut to select
    *  all the methods available. So advice will be called
    *  for all the methods.
  */
  @Pointcut("execution(* com.tutorialspoint.*.*(..))")
  private void selectAll(){}

  /**
    * This is the method which I would like to execute
    * before a selected method execution.
  */
  @Before("selectAll()")
  public void beforeAdvice(){
    System.out.println("Going to setup student profile.");
  }

  /**
```

```
    * This is the method which I would like to execute
    * after a selected method execution.
  */
  @After("selectAll()")
  public void afterAdvice(){
    System.out.println("Student profile has been setup.");
  }

  /**
    * This is the method which I would like to execute
    * when any method returns.
  */
  @AfterReturning(pointcut = "selectAll()", returning = "retVal")
  public void afterReturningAdvice(Object retVal){
    System.out.println("Returning:" + retVal.toString() );
  }

  /**
    * This is the method which I would like to execute
    * if there is an exception raised by any method.
  */
  @AfterThrowing(pointcut = "selectAll()", throwing = "ex")
  public void AfterThrowingAdvice(IllegalArgumentException ex){
    System.out.println("There has been an exception: " + ex.toString());
  }
}
```

Following is the content of the Student.java file

```
package com.tutorialspoint;

public class Student {
  private Integer age;
  private String name;

  public void setAge(Integer age) {
    this.age = age;
  }
  public Integer getAge() {
        System.out.println("Age : " + age );
    return age;
  }
  public void setName(String name) {
    this.name = name;
  }
  public String getName() {
    System.out.println("Name : " + name );
    return name;
  }
```

```
   public void printThrowException(){
      System.out.println("Exception raised");
      throw new IllegalArgumentException();
   }
}
```

Following is the content of the MainApp.java file

```
package com.tutorialspoint;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

      Student student = (Student) context.getBean("student");
      student.getName();
      student.getAge();

      student.printThrowException();
   }
}
```

Following is the configuration file **Beans.xml**

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xmlns:aop = "http://www.springframework.org/schema/aop"
   xsi:schemaLocation = "http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
   http://www.springframework.org/schema/aop
   http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">

   <aop:aspectj-autoproxy/>

   <!-- Definition for student bean -->
   <bean id = "student" class = "com.tutorialspoint.Student">
      <property name = "name" value = "Zara" />
      <property name = "age"  value = "11"/>
   </bean>

   <!-- Definition for logging aspect -->
   <bean id = "logging" class = "com.tutorialspoint.Logging"/>

</beans>
```

Once you are done creating the source and bean configuration files, let us run the application. If everything is fine with your application, it will print the following message −

```
Going to setup student profile.
Name : Zara
Student profile has been setup.
Returning:Zara
Going to setup student profile.
Age : 11
Student profile has been setup.
Returning:11
Going to setup student profile.
Exception raised
Student profile has been setup.
There has been an exception: java.lang.IllegalArgumentException
.....
other exception content
```

Spring - Transaction Management

A database transaction is a sequence of actions that are treated as a single unit of work. These actions should either complete entirely or take no effect at all. Transaction management is an important part of RDBMS-oriented enterprise application to ensure data integrity and consistency. The concept of transactions can be described with the following four key properties described as ACID −

- **Atomicity** − A transaction should be treated as a single unit of operation, which means either the entire sequence of operations is successful or unsuccessful.
- **Consistency** − This represents the consistency of the referential integrity of the database, unique primary keys in tables, etc.
- **Isolation** − There may be many transaction processing with the same data set at the same time. Each transaction should be isolated from others to prevent data corruption.
- **Durability** − Once a transaction has completed, the results of this transaction have to be made permanent and cannot be erased from the database due to system failure.

A real RDBMS database system will guarantee all four properties for each transaction. The simplistic view of a transaction issued to the database using SQL is as follows −

- Begin the transaction using begin transaction command.
- Perform various deleted, update or insert operations using SQL queries.
- If all the operation are successful then perform commit otherwise rollback all the operations.

Spring framework provides an abstract layer on top of different underlying transaction management APIs. Spring's transaction support aims to provide an alternative to EJB transactions by adding transaction capabilities to POJOs. Spring supports both programmatic and declarative transaction management. **EJBs require an application server, but Spring transaction management can be implemented without the need of an application server.**

Local vs. Global Transactions

Local transactions are specific to a single transactional resource like a JDBC connection, whereas global transactions can span multiple transactional resources like transaction in a distributed system.

Local transaction management can be useful in a centralized computing environment where application components and resources are located at a single site, and transaction management only involves a local data manager running on a single machine. Local transactions are easier to be implemented.

Global transaction management is required in a distributed computing environment where all the resources are distributed across multiple systems. In such a case, transaction management needs to be done both at local and global levels. A distributed or a global transaction is executed across multiple systems, and its execution requires coordination between the global transaction management system and all the local data managers of all the involved systems.

Programmatic vs. Declarative

Spring supports two types of transaction management −
Programmatic transaction management − This means that you have to manage the transaction with the help of programming. That gives you extreme flexibility, but it is difficult to maintain.

Let us use **PlatformTransactionManager** directly to implement the programmatic approach to implement transactions. To start a new transaction, you need to have a instance of TransactionDefinition with the appropriate transaction attributes. For this example, we will simply create an instance ofDefault-TransactionDefinition to use the default transaction attributes.

Once the TransactionDefinition is created, you can start your transaction by calling getTransaction() method, which returns an instance of TransactionStatus. The TransactionStatus objects helps in tracking the current status of the transaction and finally, if everything goes fine, you can use commit() method of PlatformTransactionManager to commit the transaction, otherwise you can use rollback() to rollback the complete operation.

```
package com.tutorialspoint;

import java.util.List;
import javax.sql.DataSource;

import org.springframework.dao.DataAccessException;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.TransactionDefinition;
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.DefaultTransactionDefinition;

public class StudentJDBCTemplate implements StudentDAO {
  private DataSource dataSource;
  private JdbcTemplate jdbcTemplateObject;
  private PlatformTransactionManager transactionManager;

  public void setDataSource(DataSource dataSource) {
    this.dataSource = dataSource;
    this.jdbcTemplateObject = new JdbcTemplate(dataSource);
  }
  public void setTransactionManager(PlatformTransactionManager transactionManager) {
    this.transactionManager = transactionManager;
  }
  public void create(String name, Integer age, Integer marks, Integer year){
    TransactionDefinition def = new DefaultTransactionDefinition();
    TransactionStatus status = transactionManager.getTransaction(def);

    try {
      String SQL1 = "insert into Student (name, age) values (?, ?)";
      jdbcTemplateObject.update( SQL1, name, age);

      // Get the latest student id to be used in Marks table
      String SQL2 = "select max(id) from Student";
      int sid = jdbcTemplateObject.queryForInt( SQL2 );

      String SQL3 = "insert into Marks(sid, marks, year) " + "values (?, ?, ?)";
      jdbcTemplateObject.update( SQL3, sid, marks, year);

      System.out.println("Created Name = " + name + ", Age = " + age);
```

```
         **transactionManager.commit(status);**
      }
      catch (DataAccessException e) {
         System.out.println("Error in creating record, rolling back");
         **transactionManager.rollback(status);**
         throw e;
      }
      return;
   }
   public List<StudentMarks> listStudents() {
      String SQL = "select * from Student, Marks where Student.id=Marks.sid";
      List <StudentMarks> studentMarks = jdbcTemplateObject.query(SQL,
         new StudentMarksMapper());

      return studentMarks;
   }
}
```

Following is the configuration file Beans.xml

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation = "http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd ">

   <!-- Initialization for data source -->
   <bean id = "dataSource"
      class = "org.springframework.jdbc.datasource.DriverManagerDataSource">
      <property name = "driverClassName" value = "com.mysql.jdbc.Driver"/>
      <property name = "url" value = "jdbc:mysql://localhost:3306/TEST"/>
      <property name = "username" value = "root"/>
      <property name = "password" value = "password"/>
   </bean>

   <!-- Initialization for TransactionManager -->
   <bean id = "transactionManager"
      class = "org.springframework.jdbc.datasource.DataSourceTransactionManager">
      <property name = "dataSource"  ref = "dataSource" />
   </bean>

   <!-- Definition for studentJDBCTemplate bean -->
   <bean id = "studentJDBCTemplate"
      class = "com.tutorialspoint.StudentJDBCTemplate">
      <property name = "dataSource" ref = "dataSource" />
      <property name = "transactionManager" ref = "transactionManager" />
   </bean>

</beans>
```

Declarative transaction management − This means you separate transaction management from the business code. You only use annotations or XML-based configuration to manage the transactions.

Declarative transaction management approach allows you to manage the transaction with the help of configuration instead of hard coding in your source code. This means that you can separate transaction management from the business code. You only use annotations or XML-based configuration to manage the transactions. The bean configuration will specify the methods to be transactional. Here are the steps associated with declarative transaction −

- We use <tx:advice /> tag, which creates a transaction-handling advice and at the same time we define a pointcut that matches all methods we wish to make transaction and reference the transactional advice.
- If a method name has been included in the transactional configuration, then the created advice will begin the transaction before calling the method.
- Target method will be executed in a try / catch block.
- If the method finishes normally, the AOP advice commits the transaction successfully otherwise it performs a rollback.

Following is the configuration file **Beans.xml**

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
  xmlns:tx = "http://www.springframework.org/schema/tx"
  xmlns:aop = "http://www.springframework.org/schema/aop"
  xsi:schemaLocation = "http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
  http://www.springframework.org/schema/tx
  http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
  http://www.springframework.org/schema/aop
  http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

  <!-- Initialization for data source -->
  <bean id="dataSource"
    class = "org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name = "driverClassName" value = "com.mysql.jdbc.Driver"/>
    <property name = "url" value = "jdbc:mysql://localhost:3306/TEST"/>
    <property name = "username" value = "root"/>
    <property name = "password" value = "cohondob"/>
  </bean>

  <tx:advice id = "txAdvice" transaction-manager = "transactionManager">
    <tx:attributes>
    <tx:method name = "create"/>
    </tx:attributes>
  </tx:advice>

  <aop:config>
    <aop:pointcut id = "createOperation"
```

```
      expression = "execution(* com.tutorialspoint.StudentJDBCTemplate.create(..))"/>

     <aop:advisor advice-ref = "txAdvice" pointcut-ref = "createOperation"/>
   </aop:config>

   <!-- Initialization for TransactionManager -->
   <bean id = "transactionManager"
     class = "org.springframework.jdbc.datasource.DataSourceTransactionManager">

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

   <!-- Definition for studentJDBCTemplate bean -->
   <bean id = "studentJDBCTemplate"
     class = "com.tutorialspoint.StudentJDBCTemplate">
     <property name = "dataSource" ref = "dataSource"/>
   </bean>

</beans>
```

Declarative transaction management is preferable over programmatic transaction management though it is less flexible than programmatic transaction management, which allows you to control transactions through your code. But as a kind of crosscutting concern, declarative transaction management can be modularized with the AOP approach. Spring supports declarative transaction management through the Spring AOP framework.