1. **Why do we need spring ? I have an java directly connect hibernate then why I need spring ?**

Spring is an loosely coupled framework , it provides loose coupling between plain java component and hibernate component (framework). It is an framework that helps to wire the difference components together. We can use spring, When we have lot of components , and wish to combine it in such a way and when we want to replace the one component with another based on different settings and different environments . It is also useful to control the lifecycle of the java bean , via IOC container. It provides easy unit testing , since all the fields has the setters methods which can be easily mocked by using the mocking framework. It a good framework for web programming. Spring supports less code to write . It provides an good segregation on service layer , web layer and business layer. Basically we can choose spring for creating the distributes application in a loosely coupled way and it provides easy maintanence.

We can use spring only we have only J2EE server or Servlet Container but not having the Application Server.

1. **How to define spring bean using the annotations ?**

@Configuration

public class SpringConfiguration{

@Bean

public TransferService transferService(){

return new TransferService();

}

}

Public Class TransferService{

Private String serviceMessage;

//getters

//setters

}

@Configuration :- indicates that particular class acts as configuration class and source of bean definitions . Which is similar to <beans/> in xml based configuration. This is the class level annotation.

@Bean :- This is the method level annotation used to define an bean. The method which is annotated with @bean return the bean of specific type and registered with beanfactory or application context. Which similar to <bean/> in xml based configuration. Therese are the properties can also be used with @bean .(intiMethodName(init-method in xml), destoryMethodName(destroy-method in xml), Scope and AutoWire)

And it should be retrieved as

ApplicationContext context=AnnotationConfigApplicationContext();

Context.register(SpringConfiguration.class);

TransferService transferService=context.getBean(TransferService.class);

transferService.setServiceMessage(“test”);

transferService.getServiceMessage();

1. **How to use dependency Injection using Spring Annotations?**

Dependency injection is done by passing the one bean method to other bean definition.

public class SpringConfiguration{

@Bean

public TransferService transferService(){

return new TransferService(localTransafer());

}

@Bean LocalTransfer localTransafer(){

return new LocalTransfer();

}

}

Here transfer service bean getting reference of localtransafer bean via constructor injection.

1. **What is Annotation based configuration in Spring?**

From spring 2.5 , it is possible to do the dependency injection via annotation instead of wiring the beans in XML. Annotation injection is performed before than the XML injection , so if we use both ,the XML injection override the annotation injection.

Annotation configuration is not enabled by default in spring container. It can be enabled by,

<beans >

<context:annotation-config />

</beans>

Once it is enabled , spring can do the annotation wiring in constructors , fields and methods.

Annotations Used:

**@Required:**

It is used for the setter method of the property. If the setter method is specified with @Required then it should be set along with the bean definition in bean configuration file , other wise it throws BeanInitilizationException (applicationContext.xml)

Ex: public class Employee{

Private String name;

Private int age;

//getter

//setter

@Required

Public void SetName(String name){

}

//getter and setter for age property

}

In bean configuration file,

<beans>

<context:annotation-config/>

//bean definition

<bean id=”emp” class=”Employee”>

<property name=”name” value=”Vasantha” />

</bean>

</beans>

@Required property value must be supplied at configuration time in bean configuration file . The other property “age” does not need to be set at the time of bean initialization.

**@Qualifier:**

It is an field level annotation. When we need to create two beans of the same type and want to refer only one at a time for the specific property , we can use @Qualifier along with @Autowired annotation.

EX:

public class Employee{

Private String name;

Private int age;

@Autowired

@Qualifier(“homeAdd”)

Private Address address;

//getter

//setter

@Required

Public void SetName(String name){

}

//getter and setter for age property

}

Public class Address{

Private int doorNo;

Private String streetName;

//getters and setters

}

In bean definitions xml file(applicationcontext.xml),

<beans>

<context:annotation-config/>

//bean definition

<bean id=”emp” class=”Employee”>

<property name=”name” value=”Vasantha” />

</bean>

<bean id=”homeAdd” class=”Address”>

</bean>

<bean id=”officeAdd” class=”Address”>

</bean>

</beans>

@Qualifier specifies the bean id from the bean definitions file, in the above Employee class the address field autowired with the bean “homeAdd” eventhough we have “officeAdd” bean of same type.

**@Autowired:**

It can be used with constructors , fields(properties) and methods (setter). It can be used for the dependency injection. It can be used for automatic dependency injection.

* **Autowired on Constructors: (Constructor based Autowiring)**

It can be used with constructors. If the constructors is annotated with @Autowired , it should be included in the bean definition when creating the bean, even if no <constructor-arg> element is specified in the bean definition XML file.

EX:

Public class Employee{

Private Address address

@Autowired

Public Employee(Address address){

this.address=address

}

}

Public class Address{

Private int doorNo;

Private String streetName;

//getters and setters

}

In bean definitions xml file,

<beans>

<context:annotation-config/>

<bean id=”emp” class=”Employee” />

<bean id=”address” class=”Address” />

</beans>

* **Autowired on properties:**

It can be applied to fields or properties. It can be get rid of defining the setter method in the bean class.

Ex:

Public class Employee{

@Autowired

Private Address address ;

//getters method setters

}

Refer Address class definition on the previous section.

In the bean definition configuration file,

<beans>

<context:annotation-config/>

<bean id=”emp” class=”Employee”/>

<bean id=”address” class=”Address” />

</beans>

Though the Employee class doesnt have the setter method for the address property , it will automatically injected at the time of bean creation based on “byType” injection.

* **Autowired on settermethod: (setter based injection)**

It can be applied to the setter method on the property. The setter method annotated with @Autowired ,then it doesn’t require the <property> element as part of bean definition in the bean configuration file. It auomatically the wire the bean on byType autowiring method. It basically get rid(be freed ) of <property> element in the bean configuration file.

Ex:

Public class Employee{

Private Address address ;

//getter method

//setters

@Autowired

Public void setAddress(Address address){

this.address=address;

}

}

Refer Address class definition on the previous section.

In the bean definition configuration file,

<beans>

<context:annotation-config/>

<bean id=”emp” class=”Employee”/>

<bean id=”address” class=”Address” />

</beans>

The “emp” bean doesn’t have the <property> element defined for the property “address” . But once the bean is instantiated , address property or filed is injected with respective Address type bean “address” since the setter method is @Autowired.

By default , @Autowired annotation requires the dependency, we can disable the default autowiring by setting the @Autowired(required=false).

1. **What is JSR Annotations supported by Spring?**

JSR is Java Specification Request. Which common for all J2EE technologies. Spring also supports JSR annotations.

**@PostConstruct:**

This is method level annotation. The method with @PostConstruct is called once the bean is created and the properties are set(after all dependency injection is set). This is an alternate for the initialization call back method “init-method”. Only one method should be annotated with this annotation. The return type should be void and should not take any parameters. It should not throw any checked exception. The method cannot be static may be final.

**Ex:**

Public class Employee{

@PostConstruct

Public void initilizeEmployee(){

}

}

**@PreDestroy:**

This is also method level annotation. The method with this annotation is called before the bean is removed from the container. This is an alternate for the destruction call back method “destroy-method”. Only one method should be annotated with this annotation. The return type should be void and should not take any parameters. It should not throw any checked exception. The method cannot be static may be final.

**Ex:**

Public class Employee{

@PreDestroy

Public void destroyEmployee(){

}

}

**@Resource:**

This is applicable for a method, fields and component class . It is used to inject the bean based on its name(enables byname injection).

**Ex:**

Public class Employee{

Private Adresss address;

@Resource(name=”homeAdd”)

Public void setAddress(Address address){

This.address=address

}

}

@Resource = @Autowired + @ Qualifier

If the name attribute is not used , it inject the bean with the same name on the fields or parameter of the setter method . If this annotation is appliedon component class, it makes the class is auto detectable at runtime.

1. **What are the important spring annotations used?**

The most frequently used annotations are,

**@Component:-** Indicates that the annotated class is a component. These type of component available for auto detection when the annotation based configuration or class –path scanning is used.

**@Repository:-** It is an special type of @Component. Used in persistent framework layer.

**@Controller:-** It is an special type of @Component. Used in Presentation layer

**@Service:-** It is an special type of @Component. Used in Service Layer

**@Configuration:-** The class annotated with @Configuration act as configuration class as like old XML. It contains bean definitions annotated with @Bean. It is similar to <beans/> in XML. It is responsible for having the logic for instantiating , initializing and configuring objects that will be managed by spring IOC Container.

Ex:

@Configuration

Public class SessionFactoryConfiguration{

@Bean

Public DataSource dataSource(){

DataSource dataSource=new DataSource();

dataSource.setURL(“”);

dataSource.setDriverClassName(“”);

dataSource.setUserName(“”);

dataSource.setPassWord(“”);

return dataSource;

}

@Bean

Public SessionFactory sessionFactory(){

SessionFactory sessionFactory=new SessionFactory(

//Setter injection

sessionFactory.setDataSource(dataSource()));

sessionFactory.setShowSQL(true);

return sessionFactory;

}

}

**@Bean:-** Method level annotation used to define the bean .It same as <bean> in XML.

**@Scope:-** By default all beans in spring IOC is Singletone. Scope can be defined for the bean using the @Scope.

Public class MyConfiguration{

@Bean

@scope(StandardScopes.PROTOTYPE)

Public TransferService transferService{

Return new TransferService();

}

}

In the above code , the bean default scope is overridden for the bean transferServcie. The new bean will be created for every client request.

**@AutoWired:-** This annotations is applicable for the constructors , fields , setters and config methods to provide the dependency injection provided by the spring. (constructors,fields,setters explained in previous question)

* Config methods: In java based Configuration ,One configuration class can directly refer the bean in another configuration class using @Autowired

Ex: @Configuration

Public Class ConfigOne{

@Bean

Public AccountRepository accountRepository(){

Return new AccountRepository();

}

}

@Configuration

Public class ConfigTwo{

@Autowired AccountRepository accountRepository;

@Bean

Public TransferService transaferService(){

return new TransferService();

}

}

Here, the AccountRepository bean is creatde in ConfigOne and will be Autowired in the AccountRepository field in ConfigTwo.

**@Import :-** Used at classlevel . Used to import one or more configuration classes. Which is equivalent to <import/> in XML.

For clarity and modularity , Bean definitions can be split into multiple @Configurations files. All these configuration files are aggregated using the @Import .

EX:

@Configuration

Public Class DataSourceConfig{

@Bean

Public DataSource dataSource(){

return new DriverManagerDataSource(…);

}

}

@Configuration

@Import(DataSourceConfig.class)

Public Class AppConfig(){

@Bean

Public transferService TransferService(){

return new TransaferService();

}

}

One or more classes can be impoerted by,

@Import(DatasourceConfig.class,DataSourceConfig1.class)

**@Lazy:-**

It is used along with @Configuration and @Bean. If The bean definition method annotated with @Lazy , spring container will not instantiate the bean at the time container loads. Instead it will be instantiated when it is used in the code at first time.

**Ex:**

@Configuration

Public class AppConfig{

@Bean

@Lazy(value=true)

Public TransferService transferService(){

Return new TransferService();

}

}

1. **What is the difference between init () and afterpropertiesSet() in spring ?**

Both are used to initializing the bean after the dependencies are injected. If the bean implements the InitilizingBean and overrides afterPropertiesSet then the order of execution will be @PostConstruct , afterPropertiesSet(), then init-method is called.

1. **What is the spring bean lifecycle ?**

BeanInstanitation->populate properties(dependency injection) - > BeanNameAware (setBeanName() ) -> BeanFactoryAware (setBeanFactory()) - > PreinitilizationBeanPostProcessor -> Initializing bean via afterProeprtiesSet() -> initializing bean via custom init-method -> PostInitilizationBeanPostProcessor -> **Bean is ready to Use** -> when the spring container shutdown , DisposableBeans(destroy()) -> custom-destroy method

Ex:

public class InitSequenceBean implements InitializingBean {

public InitSequenceBean() {

System.out.println("InitSequenceBean: constructor");

}

@PostConstruct

public void postConstruct() {

System.out.println("InitSequenceBean: postConstruct");

}

public void initMethod() {

System.out.println("InitSequenceBean: init-method");

}

@Override

public void afterPropertiesSet() throws Exception {

System.out.println("InitSequenceBean: afterPropertiesSet");

}

}

**Output :**

**InitSequenceBean: constructor**

**InitSequenceBean: postConstruct**

**InitSequenceBean: afterPropertiesSet**

**InitSequenceBean: init-method**

**Spring MVC:**

* it is an model view controller web framework for the web based applications. There is a center servlet called as Dispatcher Servlet which receives the request and dispatch to appropriate handlers and resolve the view and return the response.

**Annotations used in Spring MVC:**

**@Controller :** This annotation marks the class as spring bean and act as controller to handle the different http request based on the mapping specified in the class and individual methods.

**@RequestMapping:** This annotation is used to map the http request to appropriate handler class or handler methods. This takes several attributes.

1. **Value:** which specifies the path of the request
2. **Method:** which specifies the type of the http method that each method can serve it for.

**Spring MVC Controller Sample :**

**@Controller**

**@RequestMapping("/")**

public class HelloWorldController {

    @RequestMapping(method = RequestMethod.GET)

    public String sayHello(**ModelMap** model) {

        model.addAttribute("greeting", "Hello World from Spring 4 MVC");

        return "welcome";

    }

    @RequestMapping(value="/helloagain", method = RequestMethod.GET)

    public String sayHelloAgain(ModelMap model) {

        model.addAttribute("greeting", "Hello World Again, from Spring 4 MVC");

        return "welcome";

    }

}

In the above example , ModelMap is a map implementation which set and get attributes in the request or session. The return value of the method will be prefixed or suffixed with the values method in the spring-configuration file as like below.

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

xmlns:context="http://www.springframework.org/schema/context"

xmlns:mvc="http://www.springframework.org/schema/mvc"

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-4.0.xsd

http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd">

**<context:component-scan base-package="com.websystique.springmvc" />**

**<mvc:annotation-driven />**

<bean

class="org.springframework.web.servlet.view.**InternalResourceViewResolver"**>

<property name="prefix">

<value>/WEB-INF/views/</value>

</property>

<property name="suffix">

<value>.jsp</value>

</property>

</bean>

</beans>

The equivalent java configuration class for the above bean.xml file is as below,

@Configuration

@EnableWebMvc

@ComponentScan(basePackages = "com.websystique.springmvc")

public class HelloWorldConfiguration {

    @Bean

    public ViewResolver viewResolver() {

        InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();

        viewResolver.setViewClass(JstlView.class);

        viewResolver.setPrefix("/WEB-INF/views/");

        viewResolver.setSuffix(".jsp");

        return viewResolver;

    }

}

**@EnableWebMVC** is equivalent to **<mvc:annotation-driven/>**

**@Configuration** indicates that this class has one or more methods annotated with @Bean which produces the bean manageable by container.

**@ComponentScan** is equivalent to **<context:component-scan base-package=” “ />** indicates that where to look up the spring managed beans and classes.

**InternalResourceViewResolver : -** which resolves the view name by using the values specified in the property and help the controller to delegate response to appropriate view.

**<mvc:annotation-driven /> :** which enables the spring annotation configuration . Instead of specifying the spring beans in spring configuration file , we can annotate the dependencies.

**<context:component-scan base-package="com.websystique.springmvc" /> :-** which specifies the location where the spring container look for the @Component , @Service , @Controller , @Repository annotations and register the class in bean factory.

Specify the spring configuration file in the web.xml

<web-app id="WebApp\_ID" version="2.4"

xmlns="http://java.sun.com/xml/ns/j2ee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee

http://java.sun.com/xml/ns/j2ee/web-app\_2\_4.xsd">

<display-name>Spring4MVCHelloWorldDemo Web Application</display-name>

<servlet>

<servlet-name>dispatcher</servlet-name>

<servlet-class>

**org.springframework.web.servlet.DispatcherServlet**

</servlet-class>

**<init-param>**

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

**<param-value>/WEB-INF/spring-servlet.xml</param-value>**

**</init-param>**

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

</servlet>

<servlet-mapping>

<servlet-name>dispatcher</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

</web-app>

**DispatcherServlet :-** is the front controller which receives each request and redirect the request to appropriate controller. Also responsible for directing the response from controller to views.

The above web.xml configuration is equivalent to the below configuration class. The configuration class must implement the WebApplicationInitilizer class and onstartup() method will be invoked by the servlet container on startup.

public class **HelloWorldInitializer** implements **WebApplicationInitializer** {

    public void **onStartup**(ServletContext container) throws ServletException {

**AnnotationConfigWebApplicationContext** ctx = new AnnotationConfigWebApplicationContext();

        ctx.register(**HelloWorldConfiguration**.class);

        ctx.setServletContext(container);

        ServletRegistration.Dynamic servlet = container.addServlet("dispatcher", new **DispatcherServlet**(ctx));

        servlet.setLoadOnStartup(1);

        servlet.addMapping("/");

    }

}

Or the class can be written in more preferred way as below,

public class HelloWorldInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

@Override

protected Class<?>[] getRootConfigClasses() {

return new Class[] { HelloWorldConfiguration.class };

}

@Override

protected Class<?>[] getServletConfigClasses() {

return null;

}

@Override

protected String[] getServletMappings() {

return new String[] { "/" };

}

}

**Spring MVC + REST API:**

**@RestController :** It is introduced in the Spring MVC 4. It is the combination of @Controller and @ResponseBody

**@RequestBody:** if the method parameter is annotated with this , then spring will automatically bind the request body with this argument. Behind the scene , spring will use the HTTP Message converters to do this mapping based on the content type specified in the header.

**@ResponseBody:** if the method is annotated with this , spring will bind the return value to the Response body based on the content type specified in the header by using the HTTP method converters.

**ResponseEntity :** It is alternate of the @ResponseBody. It containes the entire response. We can set the http response method, body and headers.

**@PathVariable:** if the method parameter is annotated with this , spring will bind the path variable to the method parameter. The path variable comes with the uri and mentioned as {}.

**MediaType :** it is one of the parameter come along with @RequestMapping annotation . It specifies that message format to be produced or consumed with that method.

Ex:

//-------------------Create a User--------------------------------------------------------

@RequestMapping(value = "/user/", method = RequestMethod.POST)

public ResponseEntity<Void> createUser(@RequestBody User user, UriComponentsBuilder ucBuilder) {

System.out.println("Creating User " + user.getName());

if (userService.isUserExist(user)) {

System.out.println("A User with name " + user.getName() + " already exist");

return new ResponseEntity<Void>(HttpStatus.CONFLICT);

}

userService.saveUser(user);

HttpHeaders headers = new HttpHeaders();

headers.setLocation(ucBuilder.path("/user/{id}").buildAndExpand(user.getId()).toUri());

return new ResponseEntity<Void>(headers, HttpStatus.CREATED);

}

//------------------- Update a User --------------------------------------------------------

@RequestMapping(value = "/user/{id}", method = RequestMethod.PUT)

public ResponseEntity<User> updateUser(@PathVariable("id") long id, @RequestBody User user) {

System.out.println("Updating User " + id);

User currentUser = userService.findById(id);

if (currentUser==null) {

System.out.println("User with id " + id + " not found");

return new ResponseEntity<User>(HttpStatus.NOT\_FOUND);

}

currentUser.setName(user.getName());

currentUser.setAge(user.getAge());

currentUser.setSalary(user.getSalary());

userService.updateUser(currentUser);

return new ResponseEntity<User>(currentUser, HttpStatus.OK);

}

//------------------- Delete a User --------------------------------------------------------

@RequestMapping(value = "/user/{id}", method = RequestMethod.DELETE)

public ResponseEntity<User> deleteUser(@PathVariable("id") long id) {

System.out.println("Fetching & Deleting User with id " + id);

User user = userService.findById(id);

if (user == null) {

System.out.println("Unable to delete. User with id " + id + " not found");

return new ResponseEntity<User>(HttpStatus.NOT\_FOUND);

}

userService.deleteUserById(id);

return new ResponseEntity<User>(HttpStatus.NO\_CONTENT);

}

//------------------- Delete All Users --------------------------------------------------------

@RequestMapping(value = "/user/", method = RequestMethod.DELETE)

public ResponseEntity<User> deleteAllUsers() {

System.out.println("Deleting All Users");

userService.deleteAllUsers();

return new ResponseEntity<User>(HttpStatus.NO\_CONTENT);

}

}

The below dependency is responsible to convert the response into JSON format which is mentioned as media type in the above method.

<dependency>

    <groupId>com.fasterxml.jackson.core</groupId>

    <artifactId>jackson-databind</artifactId>

    <version>2.5.3</version>

</dependency>

**Spring Security :**

**CSRF :-** Cross Site Request Forgery

Spring security framework provides CSRF support. CSRF token is a randomly generated numbers used to validate the request coming from actual site or from malicious site.

* Csrf protection is needed for any request that are processed by browser.
* CSRF is enabled by default in spring 4.0 in both xml and java configuration

**XML :**

<http>

<csrf disabled=true />

</http>

**Java Configuration:**

Public class WebSecurityConfig extends WebSecurityConfigurerAdaptor{

Protected void configure(HttpSecurity http) throws Exception{

http.csrf().disable();

}

}