Java Spring Q/A Spring Boot

Q1.What is Spring Boot?

Ans.1.Spring boot makes to easy create stand alone and production ready spring application with third party api with minimum effort.

And Spring boot application require little Spring configuration.

2. We dont need any container like (Tomcat and jetty). we run directly in IDE.

3.It ensure that our application get a list of version of library inside or outside of spring portfolio that are guaranteed work together without problems.

② mvnw and mvnw.cmd allows the project to build even if you don’t have Maven installed via the Maven Wrapper.

Q2.How can we chang the defualt port of server?

Ans.In application.properties file we add

server.port=8081

Common application properties of the Spring Boot reference documentation.

To change the port using the command line, do the following:

java -jar application.jar --server.port=8888

Q3.How you can enable and disable Certain part of application?

Ans.Using **Spring profiles**  we can selectively enable or disable parts of your application

Spring profiles "provide a way to segregate parts of your application configuration and make it only available in certain environments".

For example, you can have an EmailGateway interface with a LoggingEmailGateway, SendGridEmailGateway and SmtpEmailGateway implementations. You could then use the LoggingEmailGateway during development, SmtpEmailGateway in staging, and SendGridEmailGateway for production.

Q4.Which logging library spring boot use default?

Ans.Spring Boot uses Logback for logging by default, and logs everything to the console

Q25.What is Transaction Management ?

Ans.developer must know few key things mentioned below:

1. Database Transactions
2. Key Properties of the Transaction (ACID Properties)

A — Atomicity

C — Consistency

I — Isolation

D — Durability

3. Transaction boundaries

4. Concurrent Transactions

# What is Transaction

A Series of actions that are treated as a single unit of work considered as a **transaction.** Also, all actions in a transaction either complete as a group or fail as a group. A single action of the transaction failed, all transaction should be rolled back or all actions are completed then the transaction is permanently committed.

Key properties of the transaction are explained here:

**Atomicity** — All transactions must be complete of rolled backed if any of the actions failed. Means all actions acting together as a unit of work. Ther should be no partially committed transactions.

**Consistency** — Transaction committed only all the actions completed, this keeps the data consistency of the transaction.

**Isolation** — each transaction is isolated from other transactions, they are not mixing each other for any reason.

**Durability** — Once a transaction committed it cannot be undone with the system failure.

So transaction management ensures the data consistency and integrity. A properly managed transaction is essential to enterprise application development because all the enterprise application persist data to the database.

Example: Let's think about online ticket/seat reservation for the movie. So there are two actions must be completed.

1. Paying for the ticket
2. Reserving a seat or remove the seat from the inventory.

If there is no transaction management and think about the situation like only one action is completed other failed because of system failure. The system became an inconsistent state and not functioning as expected. So as a proper coding practice, the developer needs to define transaction boundaries as below code sample.

**try** {

connection = dataSource.getConnection();// start the transaction

connection.setAutoCommit(**false**);

ticket.buy();

seat.reserve();

connection.commit(); // this is where to commit the transaction

} **catch** (Exception e) {

connection.rollback(); // where the rollback transaction

}

There are two(2) transaction types.

1. Global Transactions — which used to manage multiple resources such as message queues, relational databases, etc.. and managed by the application server.
2. Local Transactions — which used to manage single/one transaction such as JDBC connection. Typically associated with the specific resource. and resource manages the transaction.

# Spring Framework Advantages

Spring framework allows

1. Transaction managed seamlessly.
2. Offering consistent transaction management/programming model over the global and local transactions.
3. Benefits from the different transaction management strategies. That means the code is written once, Spring support several strategies such as JPA, Hibernate.

# Transaction Management Types

1. Programmatic Transaction Management — Developer writes custom code to manage transactions and set transaction boundaries.
2. Declarative Transaction Management — Transaction is managed by the container. which allows the developer the separate transaction management from the business code.

# Spring Transaction Isolation Levels

Spring supports transaction isolation levels. That will help developers to overcome problems when using multiple transactions in the applications or operating concurrent actions.

DEFAULT — the transaction will be managed according to the user database.

READ\_UNCOMMITED — Read uncommitted changes.

READ\_COMMITED — Read only committed changes.

REPEATABLE\_READ — read identical values multiple times.

SERIALIZABLE — read identical rows multiple times.

**DEFAULT:** Use the default isolation level of the underlying database.

**READ\_COMMITTED:** A constant indicating that dirty reads are prevented; non-repeatable reads and phantom reads can occur.

**READ\_UNCOMMITTED:** This isolation level states that a transaction may read data that is still uncommitted by other transactions.

**REPEATABLE\_READ:** A constant indicating that dirty reads and non-repeatable reads are prevented; phantom reads can occur.

**SERIALIZABLE:** A constant indicating that dirty reads, non-repeatable reads, and phantom reads are prevented.



What do these Jargons dirty reads, phantom reads, or repeatable reads mean?

* **Dirty Reads**: Transaction "A" writes a record. Meanwhile, Transaction "B" reads that same record before Transaction A commits. Later, Transaction A decides to rollback and now we have changes in Transaction B that are inconsistent. This is a dirty read. Transaction B was running in READ\_UNCOMMITTED isolation level so it was able to read Transaction A changes before a commit occurred.
* **Non-Repeatable Reads**: Transaction "A" reads some record. Then Transaction "B" writes that same record and commits. Later Transaction A reads that same record again and may get different values because Transaction B made changes to that record and committed. This is a non-repeatable read.
* **Phantom Reads:** Transaction "A" reads a range of records. Meanwhile, Transaction "B" inserts a new record in the same range that Transaction A initially fetched and commits. Later Transaction A reads the same range again and will also get the record that Transaction B just inserted. This is a phantom read: a transaction fetched a range of records multiple times from the database and obtained different result sets (containing phantom records).

# Using Transaction with Spring

Spring provides support for automatically committing or rolling back the transaction if it is failed. There are several different API help developers to manage transactions.

1. Java Transaction API
2. Java Persistence API
3. Java Database Connectivity
4. Java Data Objects
5. Hibernate
6. Java Message Service

The Advantage of using Spring to manage transaction is that they provide consistent programming model across all above APIs.

**Manual/Programmatic Transaction Management**

Let's take below example of the manual transaction management. Which developer is responsible for begin and commit the transaction.

**public** **void** BuyTicket(Ticket ticket){

Session session = sessionFactory.getCurrentSession();

session.getTransaction().begin(); // begin transaction

session.save(ticket); // perform action

session.getTransaction().commit(); // commit the transaction

}

However, Spring introduced Transaction Template such as JDBCTemplate and Platform Transaction Manager to handle transactions across Hibernate, JPA, JMS, JDBC, etc.

**Spring Transaction Management**

Let’s consider below example which is Spring transaction management. which is very lightweight and flexible code.

@Transactional

**public** **void** BuyTicket(Ticket ticket){

Session session = sessionFactory.getCurrentSession().save(ticket);

}

Above code, **@Transactional,** Spring manages the commit and rollback and developer don't need to worry about it.

Let’s consider some complex transaction with Spring.

@Transactional

**public** **void** doComplexTransaction(Ticket ticket){

ticketDao.save(ticket);

sendEmail(ticket);

doOtherStuff(ticket);

}

Above code is doing few actions instead of one action. even if there are multiple actions, Spring @Transactional will take care of it. That means if there is any failure happens either one of the actions, the transaction will be rollback. (Automatic rollback happens during an exception). Spring provides this support for both programmatic and declarative transaction management.

Above the second example considered as declarative transaction management. Spring will intercept the request made by the application and do the necessary transaction management actions such as;

1. Begin
2. Suspend
3. Commit
4. Rollback
5. Transactional Parameters such as isolation level

Whenever possible, declarative transaction management is the better choice since container manages everything for the developer. It will save development time as well as reduce code complexity by separating business logic from the transaction management.

**@Transactional Annotation**

Using this means that developers do not need to think about direct transaction management and exception handling. Spring will do it for you.

Using @Transaction annotation,

1. Spring creates a proxy to hold the transaction management code.
2. Can be used in class, interface or method level.
3. Transaction propagation handled automatically.

Q24.How many instances created for singleton bean referring to a session bean/prototype bean?

Ans.**The bean configuration is like this**

<bean id="a" class="A"> <property name="b" ref="b"/> </bean> <bean id="b" class="B" scope="session"/> or <bean id="b" class="B" scope="prototype"/>

In Spring, most of the beans we work with are Singletons. If a singleton bean is wired with yet another singleton bean, there is absolutely no problem. But if it is wired with a bean which is of different scope, say prototype, how does it work? Here is the example:

public class RequestProcessor {

private RequestValidator validator;

public void handleRequest(String requestId){

validator.validate(requestId);

// Process the request and update

}

public RequestValidator getValidator() {

return validator;

}

public void setValidator(RequestValidator validator) {

this.validator= validator;

}

}

public class RequestValidator {

private List<String> errorMessages = new ArrayList<String>();

public RequestValidator() {

System.out.println("Validator instance created!");

}

// Validates the request and populates error messages

public void validate(String requestId){

}

public List<String> getErrorMessages() {

return errorMessages;

}

}

<bean id="requestProcessor" class="com.pramati.spring.RequestProcessor">

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

</bean>

<bean id="validator" scope="prototype" class="com.pramati.spring.RequestValidator"/>

With this configuration, it is expected that when ever I fetch *requestProcessor* from application context, it will be wired with a new validator as we declared the *validator*bean is of prototype scope. But this does not happen.

When the application context gets initialized, it sees that *requestProcessor* is a singleton bean and initializes it to the context after wiring it with all the dependencies set. So from then onwards when we request context for *requestProcessor*, it return the same bean every time. To solve this issue, we have 2 approaches:

**1. Lookup Method injection:** For this, we have to declare the beans as follows:

<bean id="requestProcessor" class="com.pramati.spring.RequestProcessor">

<lookup-method name="getValidator" bean="validator"/>

</bean>

<bean id="validator" scope="prototype" class="com.pramati.spring.RequestValidator"/>

**2. Scoped Proxies:** This can be implemented as:

<bean id="requestProcessor" class="com.pramati.spring.RequestProcessor">

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

</bean>

<bean id="validator" scope="prototype" class="com.pramati.spring.RequestValidator">

<!-- This instructs the container to proxy the current bean-->

<aop:scoped-proxy/>

</bean>

Remember, in case of look up method injection, proxy is created for singleton bean. But in case of scoped proxies, proxy is created for prototype bean and wired into the singleton bean during the process of registering the singleton bean in the context. The proxy thus created understands the scope and returns instances based on the requirements of the scope. So in our case, *requestProcessor* holds a reference to proxy in place of *validator*.

And in case of lookup method injection, when *requestProcessor* gets loaded into the context, *validator* will not be initialized at all. And when we call the look up method, it returns the prototype bean. But instead of calling the method, if you try to directly access the prototype bean(assuming it is accessible), it gives a Nullpointer Exception as it didn’t get initialized(We are not wiring it using *property* tag of bean)

**Few points to note:**

1. Both method injection and scoped proxies work not only for prototype beans. This works more generic. Whenever a bean of different scope is injected into a singleton bean, we can use any of these techniques to ensure that we get a corresponding scope object.

2. Note that in the proxy, the method returning the prototype bean is overridden to return a new instance for every single call

Q23.What is difference between YAML and properties?

Ans.YAML is a human friendly data serialization standard, which is mainly made for configuration files. YAML stands for YAML Ain't Markup Language

YAML supports Maps, lists and scalar types. YAML is hierarchical and may use consistent spaces to denote hierarchy. It will also support inline maps. YAML also supports multiple Spring profiles in single YAML config file. One limitation of YAML however is that it does not work with @PropertySource annotation

Example:

map1:

key: k

value: 9

map2: {b=true, date=2016-10-10}

Properties file is mainly used with Java, supports only String types and is non-hierarchical; we can have maps by denoting hierarchies as dots. Properties file will also work with @PropertySource annotation. One of the limitation of properties file is that we can only configure one Spring profile per properties file.

Example:

map1.key=k

map2.value=9

Q22.What is spring profile?

Ans.Spring Profiles provide a way to segregate parts of your application configuration and make it be available only in certain environments. Any @Component or @Configuration can be marked with @Profile to limit when it is loaded.

*@Configuration*  
*@Profile("production")*  
**public** **class** ProductionConfiguration {  
  
 *// ...*  
  
}

You can use a spring.profiles.active Environment property to specify which profiles are active. You can specify the property in any of the ways described earlier in this chapter. For example, you could include it in your application.properties

spring.profiles.active=dev,hsqldb

**@Component** Preferable for component scanning and automatic wiring.

*When should you use* ***@Bean****?*

Sometimes automatic configuration is not an option. **When?** Let's imagine that you want to wire components from 3rd-party libraries (you don't have the source code so you can't annotate its classes with @Component), so automatic configuration is not possible.

The **@Bean** annotation **returns an object** that spring should register as bean in application context. The **body of the method** bears the logic responsible for creating the instance.

1. @Component **auto detects** and configures the beans using classpath scanning whereas @Bean **explicitly declares** a single bean, rather than letting Spring do it automatically.
2. @Component **does not decouple** the declaration of the bean from the class definition where as @Bean **decouples** the declaration of the bean from the class definition.
3. @Component is a **class level annotation** where as @Bean is a **method level annotation** and name of the method serves as the bean name.
4. @Component **need not to be used with the @Configuration** annotation where as @Bean annotation has to be **used within the class which is annotated with @Configuration**.
5. We **cannot create a bean** of a class using @Component, if the class is outside spring container whereas we **can create a bean** of a class using @Bean even if the class is present **outside the spring container**.
6. @Component has **different specializations** like @Controller, @Repository and @Service whereas @Bean has **no specializations**.

Q21.What is IOC and DI IN SPRING ?

Ans. The IoC container is responsible to instantiate, configure and assemble the objects. The IoC container gets informations from the XML file and works accordingly. The main tasks performed by IoC container are:

* To instantiate the application class
* To configure the object
* To assemble the dependencies between the objects There are two types of IoC containers. They are:
* BeanFactory
* ApplicationContext

Dependency Injection (DI)

Dependency Injection (DI) is a design pattern that removes the dependency from the programming code so that it can be easy to manage and test the application. Dependency Injection makes our programming code loosely coupled.

**Spring framework provides two ways to inject dependency**

* By Constructor
* By Setter method

1. **Partial dependency**: can be injected using setter injection but it is not possible by constructor. Suppose there are 3 properties in a class, having 3 arg constructor and setters methods. In such case, if you want to pass information for only one property, it is possible by setter method only.
2. **Overriding**: Setter injection overrides the constructor injection. If we use both constructor and setter injection, IOC container will use the setter injection.
3. **Changes**: We can easily change the value by setter injection. It doesn't create a new bean instance always like constructor. So setter injection is flexible than constructor injection.

|  |  |  |
| --- | --- | --- |
| **4.** Setter injection makes bean class object as mutable [We can change ] |  | **4.** Constructor injection makes bean class object as immutable [We cannot change ] |

Q20.What is Spring bean life Cycle Explain it?

Ans. A ***Spring Bean*** represents a ***POJO component*** performing some useful operation. All ***Spring Beans*** reside within a ***Spring Container*** also known as ***IOC Container***. The Spring Framework is transparent and thereby hides most of the complex infrastructure and the communication that happens between the Spring Container and the Spring Beans

1. The Bean Container finds the definition of the Spring Bean in the Configuration file.
2. The Bean Container creates an instance of the Bean using Java Reflection API.
3. If any properties are mentioned, then they are also applied. If the property itself is a Bean, then it is resolved and set.
4. If the Bean class implements the BeanNameAware interface, then the setBeanName()method will be called by passing the name of the Bean.
5. If the Bean class implements the BeanClassLoaderAware interface, then the method setBeanClassLoader() method will be called by passing an instance of the ClassLoader object that loaded this bean.
6. If the Bean class implements the BeanFactoryAware interface, then the method setBeanFactory() will be called by passing an instance of BeanFactory object.
7. If there are any BeanPostProcessors object associated with the BeanFactory that loaded the Bean, then the method postProcessBeforeInitialization() will be called even before the properties for the Bean are set.
8. If the Bean class implements the InitializingBean interface, then the method afterPropertiesSet() will be called once all the Bean properties defined in the Configuration file are set.
9. If the Bean definition in the Configuration file contains a 'init-method' attribute, then the value for the attribute will be resolved to a method name in the Bean class and that method will be called.
10. The postProcessAfterInitialization() method will be called if there are any Bean Post Processors attached for the Bean Factory object.
11. If the Bean class implements the DisposableBean interface, then the method destroy() will be called when the Application no longer needs the bean reference.
12. If the Bean definition in the Configuration file contains a 'destroy-method'attribute, then the corresponding method definition in the Bean class will be called.
13. **Bean LifeCycle :**
14. When a bean is initialized it might require to perform some action before it can come into a usable state(State in which application can use it) and when a bean is getting destroyed there might be some cleanup activity required for given bean. These activities are known as bean Lifecycle

Standard bean lifecycle interfaces & the standard order of execution are given below.  
**1-**   IoC container will look for the *configuration metadata* of given Bean.  
**2-**   Once found, the container will create the instance of Bean(Using reflection API).  
**3-**   After instance, creation dependency will be injected(DI).

**If Bean Class implements any of the below-highlighted interfaces then corresponding method will be invoked in below order(Point 4 – 13).**

**4-**   setBeanName method of ***BeanNameAware* I**nterface. It sets the name of the bean in the bean factory that created this bean.  
**5-**   setBeanClassLoader method of ***BeanClassLoaderAware*** Interface. Callback that supplies the bean to a bean instance.  
**6-** setBeanFactory method of ***BeanFactoryAware***Interface. Callback that supplies the owning factory to a bean instance.

**Below method execution will be applicable when running in an application context. (Points 7 – 11)**

**7-**   setResourceLoader  method of ***ResourceLoaderAware***Interface. It set the ResourceLoader that this object runs in.  
**8-** setApplicationEventPublisher  method of ***ApplicationEventPublisherAware***Interface. Set the ApplicationEventPublisher that this object runs in.  
**9-**   setMessageSource method of ***MessageSourceAware***Interface. Set the MessageSource that this object runs in.  
**10-**   setApplicationContext method of ***ApplicationContextAware***Interface. Set the ApplicationContext that this object runs in.  
**11-**   setServletContext method of ***ServletContextAware***Interface. Set the ServletContext that this object runs in.

**12-**   postProcessBeforeInitialization method of ***BeanPostProcessor***Interface. Apply this BeanPostProcessor to the given new bean instance before any bean initialization callbacks.  
**13-**   afterPropertiesSet method of ***InitializingBean***Interface. Invoked by a BeanFactory after it has set all bean properties supplied.

When Bean Factory is getting shut down following lifecycle methods will be executed.

**1-**   DisposableBean’s **destroy** method. Invoked by a BeanFactory on the destruction of a singleton.  
**2-**   **Custome destroy** method will be executed if there is any defined via destroy-method attributes

## Q1. Explain Spring Bean Autowiring?

* Ans. Autowiring of the Spring framework enables you to inject the object dependency implicitly. Autowiring needs significantly less specification with properties or constructor arguments.

## Q2. Explain different modes of bean autowiring?

There are **five auto wiring modes** in spring framework. Lets discuss them one by one.

1. **no**: This option is default for spring framework and it means that autowiring is OFF. You have to explicitly set the dependencies using tags in bean definitions.
2. **byName**: This option enables the dependency injection based on bean names. When autowiring a property in bean, property name is used for searching a matching bean definition in configuration file. If such bean is found, it is injected in property. If no such bean is found, a error is raised.
3. **byType**: This option enables the dependency injection based on bean types. When autowiring a property in bean, property’s class type is used for searching a matching bean definition in configuration file. If such bean is found, it is injected in property. If no such bean is found, a error is raised.
4. **constructor**: Autowiring by constructor is similar to byType, but applies to constructor arguments. In autowire enabled bean, it will look for class type of constructor arguments, and then do a autowire by type on all constructor arguments. Please note that if there isn’t exactly one bean of the constructor argument type in the container, a fatal error is raised.
5. **autodetect**: Autowiring by autodetect uses either of two modes i.e. constructor or byType modes. First it will try to look for valid constructor with arguments, If found the constructor mode is chosen. If there is no constructor defined in bean, or explicit default no-args constructor is present, the autowire byType mode is chosen.
6. **18) How do you turn on annotation based autowiring?**

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

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

2. Include AutowiredAnnotationBeanPostProcessor directly in bean configuration file.

|  |
| --- |
| <beans>      <beanclass="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor"/>  </beans>  Example-> |

<bean id="employeeDAO" class="com.howtodoinjava.EmployeeDAOImpl" autowire="byName" />

**Q21) Explain @Qualifier annotation with example?**

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

public class Customer

{

    @Autowired

    private Person person;

}

<bean id="customer" class="com.howtodoinjava.common.Customer" />

<bean id="personA" class="com.howtodoinjava.common.Person" >

    <property name="name" value="lokesh" />

</bean>

<bean id="personB" class="com.howtodoinjava.common.Person" >

    <property name="name" value="alex" />

</bean>

public class Customer

{

    @Autowired

    @Qualifier("personA")

    private Person person;

}

**Q2) Name some of the design patterns used in Spring Framework?**

**Ans.** There are loads of different design patterns used, but there are a few obvious ones:

* **Proxy** – used heavily in AOP, and remoting.
* **Singleton** – beans defined in spring config files are singletons by default.
* **Template method** – used extensively to deal with boilerplate repeated code e.g. [**RestTemplate**](https://howtodoinjava.com/spring/spring-restful/spring-restful-client-resttemplate-example/), JmsTemplate, JpaTemplate.
* **Front Controller** – Spring provides DispatcherServlet to ensure an incoming request gets dispatched to your controllers.
* **View Helper** – Spring has a number of custom JSP tags, and velocity macros, to assist in separating code from presentation in views.
* **Dependency injection** – Center to the whole BeanFactory / ApplicationContext concepts.
* **Factory pattern** – BeanFactory for creating instance of an object.

**Advantages of Autowiring**

* Autowiring requires less code because we don’t need to write the code to inject the dependency explicitly.
* It reduces develop time by removing the necessity of specifying properties and constructor arguments.
* **Disadvantages of Autowiring**
* Wiring information may not be available to tools that generate documentation from a Spring container.

Q1-1) What is spring framework? Why Java programmer should use Spring framework  
  
Ans. Spring is a framework which helps Java programmer in development. Spring provides Dependency Injection and IOC container, Spring MVC flow and several useful API for Java programmer.  
  
Q2. 2) What is default scope of bean in Spring framework?  
Ans. The default scope of a Spring bean is the Singleton scope and in the web application default scope of a spring bean is request scope. Singleton bean means the same instance of a bean is shared with all other beans, while request scope means a bean is alive only for a request.  
  
In spring framework bean declared in ApplicationContext.xml can reside in five scopes:

1) Singleton (default scope)

2) prototype

3) request

4) session

5) global-session

Singleton and prototype are two common bean scope which is available on all Spring Application Context while request,session and global session bean scope are only available on Web aware application Context like WebApplicationContext.

1.Singleton Bean->**Singleton bean scope is default scope for bean** declared in Spring and applicable when you don't specify scope attribute while specifying bean details in ApplicationContext.xml or Spring configuration file. Singleton bean scope is like [Singleton pattern in Java](http://javarevisited.blogspot.com/2011/03/10-interview-questions-on-singleton.html) where only one instance of bean is created per Spring container. So no matter how many times you call getBean() method, **same bean instance** will be returned if its bean scope is declared as Singleton

2. prototype bean scope, every getBean() call creates a new instance of Spring bean

Q3**: What is the difference between singleton and prototype bean?**

**ANS. Singleton:** means single bean definition to a single object instance per Spring IOC container.  
**Prototype**: means a single bean definition to any number of object instances.

Whatever beans we defined in spring framework are singleton beans. There is an attribute in bean tag named ‘singleton’ if specified true then bean becomes singleton and if set to false then the bean becomes a prototype bean. By default, it is set to true. So, all the beans in spring framework are by default singleton beans.

  <bean id="createNewStock"     class="springexample.stockMarket.CreateNewStockAccont" **singleton=”false”**>  
        <property name="newBid"/>   
  </bean>

## Q 3) Does Spring singleton beans are thread-safe IS yes Why or not why? No, Spring singleton beans are not thread-safe. Singleton doesn't mean bean would be[thread-safe](http://javarevisited.blogspot.sg/2012/01/how-to-write-thread-safe-code-in-java.html).

2.  singleton beans will not have any state (stateless). Singleton bean scope ensures that single instance per BeanFactory. So in multi threading environment it will not assure the single instance even with singleton bean scope.

So to fix this, we need to change the bean scope from **singleton** scope to **prototype** scope as it is the developers responsibility to ensure the thread safety.

## 

Singletons are about **creation**. This design pattern ensures that only one instance of a class is created.

Thread safety is about **execution**

## 4) What is Dependency Injection? Dependency Injection is one of the design pattern, which allows injecting dependency on Object, instead of object resolving the dependency. 5) What is Inversion of Control concept, how does Spring support IOC? ([answer](http://javarevisited.blogspot.sg/2012/12/inversion-of-control-dependency-injection-design-pattern-spring-example-tutorial.html)) The simple meaning of inversion of the control means that now the framework, Spring is responsible for creating objects, wiring dependencies and managing their life-cycle instead of a developer, which was the case before. That's where control is inverted from developer to framework. Inversion of Control and Dependency Injection design pattern

Any way let’s back to core *concept of Inversion of Control and dependency Injection*design pattern. Look at below implementation of an AuditService whose job is to store every audit messages into database. This is one of the simplest kind of auditing Service required in Enterprise Java application.

**public** **class** AuditServiceImpl **implements** AuditService{  
  
    **private** AuditDAO auditDao = **new** AuditDAO();  
        
    @**Override**  
    **public** **boolean** audit (**String** message) {  
       **return** auditDao.store(message);  
    }  
    
}

there are three major problem with this implementation:

1) Every AuditServiceImpl has its **own copy of AuditDAO** which is an **expensive** object as it wraps a [database connection](http://javarevisited.blogspot.sg/2012/06/jdbc-database-connection-pool-in-spring.html)with in. It make no sense to create separate instances of AuditDAO, if you can share one between multiple AuditService.

2) AuditServiceImpl is **closely coupled with AuditDAO** as its creating instance of AuditDAO using new() operator. If you change the [constructor](http://javarevisited.blogspot.sg/2012/12/what-is-constructor-in-java-example-chainning-overloading.html) of AuditDAO this code will be broken. Because of this coupling its difficult to replace AuditDAO with better implementation.

3) There is **no easy way to test audit()** method which is **dependent on auditDAO**. Since you can not mock AuditDAO you have to rely on actual implementation and if AuditDAO is an environmental dependent object which it is as it connect to different database on different environment, your [Junit test](http://javarevisited.blogspot.sg/2012/08/best-practices-to-write-junit-test.html) case may pass in some environment and may fail in other environment.

## What is Dependency Injection concept:

Dependency Injection is a design pattern on which dependency of object (in this caseAuditDAO is a dependency for AuditServiceImpl Object) is injected by framework rather than created by [Object](http://javarevisited.blogspot.ca/2012/12/what-is-object-in-java-or-oops-example.html) itself. Dependency Injection reduces coupling between multiple object as its dynamically injected by framework. One of the implementation of DI is Inversion of Control (IOC) on which framework like Spring controls object’s dependency. There are mainly two types of Dependency Injection: [Constructor Injection and Setter Injection](http://javarevisited.blogspot.sg/2012/11/difference-between-setter-injection-vs-constructor-injection-spring-framework.html)

1.In Constructor Injection, dependency of Object is injected using [constructor](http://javarevisited.blogspot.sg/2012/12/what-is-constructor-in-java-example-chainning-overloading.html), while in Setter Injection, Dependency is provided by[setter method](http://javarevisited.blogspot.sg/2012/12/getter-and-setter-method-vs-public-modifier-field-java.html)

2. Constructor DI allows object to be created in complete state and follows principle of fully functional object while Setter DI allows object to be created without its dependency. which may result in **incomplete object** if dependency is not available

3. Another benefit of Setter Dependency Injection is readability, since Spring is configured with **xml configuration file** and setter injection is provided with bean property which is much easier to read and understand than constructor injection which doesn't state the property.

**AuditServiceImpl written using Dependency Injection**

**public** **class** AuditServiceImpl **implements** AuditService{  
  
    **private** AuditDAO auditDao;  
  
    **public** **void** setAuditDao(AuditDAO AuditDao) {  
        **this**.AuditDao = AuditDao;  
    }  
    
    @**Override**  
    **public** **boolean** audit (**String** message) {  
       **return** auditDao.store(message);  
    }  
    
}

1. Since AuditDAO is injected here its possible to share single AuditDAO (an expensive object) between multipleAuditService.

2. Since AuditServiceImpl is not creating instance of AuditDAO its no more coupled with AuditDAO and work with any implementation of AuditDAO, thanks to another famous object oriented design principle [“program for interface than implementation"](http://javarevisited.blogspot.de/2012/03/10-object-oriented-design-principles.html).

3. Because AuditDAO is injected by DI at runtime its easy to test audit() method by providing a mock AuditDAO class. This not only makes testing easier but also independent of environmental changes as you are not using actual implementation ofAuditService.

**1) Reduce coupling**

both constructor and setter dependency injection reduce coupling. like in above example coupling between AuditService andAuditDAO is reduced by using Dependency Injection.

**2) Improves testability**

Dependency Injection allows to replace actual object with mock object which improves testability by writing simple JUnit tests which uses mock object.

**3) Flexibility**

**Q6) What is Spring MVC? Can you explain How one request is processed?** ([answer](http://javarevisited.blogspot.sg/2017/06/how-spring-mvc-framework-works-web-flow.html))  
ANS->Spring MVC is a framework to develop Java web application. It provides an implementation of MVC or Model View Controller architecture which is built on separation of concerns and makes the development of Java web application easy.

**flow of request in Spring MVC->**

1. It all starts with the client, which sends a request to a specific URL. When that request hit the web container e.g. Tomcat it look into web.xml and find the Servlet or Filter which is mapped to that particular URL. It the delegate that Servlet or Filter to process the request.
2. Web container e.g. Tomcat is responsible for creating Servlet and Filter instances and invoking their various life-cycle methods e.g. [init()](http://javarevisited.blogspot.sg/2015/02/constructor-vs-init-method-in-servlet.html), service(), destroy(). In the case of HTTP request, HttpServlet handles that and depending upon the HTTP request method various doXXX() method is invoked by container e.g. doGet() to process GET request and doPost() to process POST request.
3. to enable Spring MVC, we need to declare the **DispatcherServlet** from Spring MVC jar into web.xml. This [Servlet](http://javarevisited.blogspot.sg/2017/05/difference-between-servlet-and-jsp.html)listens for a URL pattern \* as shown in below web.xml, which means all request is mapped to DispatcherServlet
4. <web-app>
5. <!-- The front controller of this Spring Web application, responsible
6. for handling all application requests -->
7. <servlet>
8. <servlet-name>Spring MVC [Dispatcher Servlet](http://javarevisited.blogspot.sg/2015/09/eclipse-javalangclassnotfoundexception.html#axzz4jWEJmi6S)</servlet-name>
9. <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
10. <init-param>
11. <param-name>contextConfigLocation</param-name>
12. <param-value>/WEB-INF/config/web-application-config.xml</param-value>
13. </init-param>
14. <load-on-startup>1</load-on-startup>
15. </servlet>
16. <servlet-mapping>
17. <servlet-name>example</servlet-name>
18. <url-pattern>\*</url-pattern>
19. </servlet-mapping>
20. </web-app>

Q6.1) How does DispatcherServlet know which request needs to be passed to which controller?

Ans->it uses the **@RequestMapping** annotation or Spring MVC configuration file to find out mapping of request URL to different controllers. It can also use specific request processing annotations e.g. @GetMapping or @PostMapping. Controller classes are also identified using @Controller and @RestController

@Controller

@RequestMapping("/appointments")

public class AppointmentsController {

@GetMapping

public Map get() {

return appointmentBook.getAppointmentsForToday();

}

@PostMapping

public String add(@Valid AppointmentForm appointment, BindingResult result) {

if (result.hasErrors()) {

return "appointments/new";

}

appointmentBook.addAppointment(appointment);

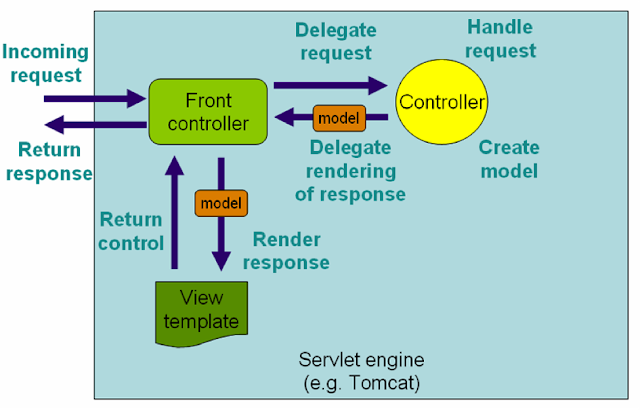
return "redirect:/appointments";

}

}

3. After processing the request, Controller returns a **logical view name** and model to [DispatcherServlet](http://javarevisited.blogspot.sg/2016/01/solving-javalangclassnotfoundexception-org.springframework.web.servlet.DispatcherServlet.html)and it consults view resolvers until an actual View is determined to render the output. DispatcherServlet then contacts the chosen view e.g. Freemarker or JSP with model data and it renders the output depending on the model data.

4. This Rendered output is returned to the client as HTTP response. On it's way back it can pass to any configured Filter as well e.g. [Spring Security filter chain](http://javarevisited.blogspot.sg/2017/05/how-to-enable-spring-security-in-java-web-application.html#axzz4gg59c400) or Filters configured to convert the response to JSON or XML.  
  
The DispatcherServlet from Spring MVC framework is an implementation of Front Controller Pattern (see [Patterns of Enterprise Application Architecture](https://www.amazon.com/Patterns-Enterprise-Application-Architecture-Martin/dp/0321127420?tag=javamysqlanta-20)) and it's also a Single point of entry - handle all incoming requests, but again that depends upon your URL pattern mapping and your application.  
  
It delegates requests for further processing to additional components e.g. Controllers, Views, View Resolvers, handler mappers, exception handlers etc. It can also map directly to /, but then the exception for handling static resources needs to be configured. If you look at the web.xml configuration it also pre-loaded using the [load-on-startup](http://javarevisited.blogspot.sg/2011/12/load-on-startup-servlet-webxml-example.html#axzz4jWEcwHFD) tag



# Q7>[What is load-on-startup servlet element in web.xml with Example?](https://javarevisited.blogspot.in/2011/12/load-on-startup-servlet-webxml-example.html)

Ans->As stated earlier load-on-startup is a tag element which appear inside <servlet> tag in web.xml.load-on-startup tells the web container about loading of a particular servlet. if you don't specify load-on-startup then container will load a particular servlet when it feels necessary most likely when first request for that servlet will come, this may lead to longer response time for that query if Servlet is making [database connections](http://javarevisited.blogspot.com/2011/11/database-transaction-tutorial-example.html) or performing [ldap authentication](http://javarevisited.blogspot.com/2011/09/spring-interview-questions-answers-j2ee.html) which contribute network latency or any other time consuming job, to avoid this, web container provides you a mean to specify certain servlet to be loaded during deployment time of application by using load-on-startup parameter.

## Important points on load-on-startup element

1. If <load-on-startup> value is same for two servlet than they will be loaded in an order on which they are declared inside web.xml file.

2. if <load-on-startup> is 0 or negative integer than Servlet will be loaded when Container feels to load them.

3. <load-on-startup> guarantees loading, initialization and call to init() method of servlet by web container.

4. If there is no <load-on-startup> element for any servlet than they will be loaded when web container decides to load them.

**When to use <load-on-startup> in web.xml**

<load-on-startup> is suitable for those servlet which performs time consuming jobs e.g. Creating Database Connection pool, downloading files or data from network or prepare environment ready for servicing client in terms of initializing cache , clearing pipelines and loading important data in memory. If any of your servlet performs these jobs then declare them using <load-on-startup> element and specify order as per your business logic or what suites your application. **Remember lower the value of <load-on-startup>, servlet will be loaded first**

**Q8>7) How do you create a controller in Spring? @Controller vs @RestController?**  
A controller is nothing but a class, also known as a bean in Spring terminology. If you are using annotation then you can create controller by using @Controller annotation. If you are developing RESTful web service then you can also create REST controllers by using @RestController annotation

Q9>**17) What is the difference in Spring MVC and Spring core?**  
The Spring MVC is part of Spring framework which helps you to develop Java web application using model web controller pattern, while Spring Core provides the Dependency injection and Inversion of Control. The Spring Container is part of Spring core

## Both functionalities come in different JAR files. If you are developing just a core Java application using Spring then you just need Spring Core but if you are developing Web application then you need spring-mvc.jar as well Q10>BeanFactory vs ApplicationContext in Spring

Ans.- Spring provides two kinds of IOC container, one is BeanFactory and other is ApplicationContext. Syntactically BeanFactory and ApplicationContext both are [Java interfaces](http://javarevisited.blogspot.in/2012/04/10-points-on-interface-in-java-with.html) and ApplicationContext extends BeanFactory. Both of them are configuration using [XML configuration file](http://javarevisited.blogspot.in/2012/03/how-to-read-properties-file-in-java-xml.html). In short BeanFactory provides basic IOC and DI features while ApplicationContext provides advanced feature

1) BeanFactory doesn't provide support for internationalization i.e. i18n but ApplicationContext provides support for it.

2) Another difference between BeanFactory vs ApplicationContext is ability to publish event to beans that are registered as listener.

3) One of the popular implementation of BeanFactory interface is XMLBeanFactory while one of the popular implementation of ApplicationContext interface is ClassPathXmlApplicationContext. On [Java web application](http://javarevisited.blogspot.sg/2012/08/what-is-jsessionid-in-j2ee-web.html) we useWebApplicationContext  which extends ApplicationContext interface and adds getServletContext method.

**public** **static** **void** main(**String** args[]){  
    ApplicationContext ctx =**new** ClassPathXmlApplicationContext("beans.xml");  
    Hello hello =(Hello) ctx.getBean("hello");  
    hello.sayHello("John");  
}

Q5) when do you use Setter injection and Constructor Injection in Spring?

Q7. What happens, if we return NULL object for ModelAndView in Spring MVC request flow?

**Spring MVC request flow:**

1. ***DispatcherServlet*** is the central servlet; which handles every incoming HTTP requests in application
2. DispatcherServlet consults ***handler mapping*** before invoking associated ***@Controller*** with the incoming requests
3. One of the associated @Controller method with respective mapping –> processes the HTTP requests
4. After processing, ModelAndView is constructed and given back to DispatcherServlet again
5. DispatcherServlet sends view-name from ModelAndView object to ***view resolver***
6. View-name along with model in ModelAndView is ***rendered*** and finally result shown to end-users

 @Override

    protected ModelAndView handleRequestInternal(HttpServletRequest httpServletRequest,

            HttpServletResponse httpServletResponse) throws Exception {

        ModelAndView modelAndView = new ModelAndView("showMessage");

        modelAndView.addObject("message", "My First Spring MVC web application");

        return null;

**Conclusion:** When we access the URL,

* It is perfectly alright to return **NULL** for ModelAndView object
* There are no ***errors*** or ***exceptions*** thrown at server
* There is no **404 page not** found error
* But response rendered is empty or blank as you can see in the above screen capture

Q8. CAN WE have multiple ***DispatcherServlet*** configured in ***web.xml?***

***Ans.*** Absolutely, it is fine to configure ***2 different DispatcherServlet*** in the same ***web.xml*** for any Spring MVC based application

<?xml version="1.0" encoding="ISO-8859-1"?>

<web-app xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>" xmlns="<http://java.sun.com/xml/ns/javaee>" xsi:schemaLocation="<http://java.sun.com/xml/ns/javaee> <http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd>" id="WebApp\_ID" version="2.5">

    <!-- Spring MVC DispatcherServlet: dispatches HTTP requests to registered controllers -->

    <servlet>

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

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

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

    </servlet>

    <servlet>

        <servlet-name>mvc-dispatcher-2</servlet-name>

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

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

    </servlet>

    <servlet-mapping>

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

        <url-pattern>/first/\*</url-pattern>

    </servlet-mapping>

    <servlet-mapping>

        <servlet-name>mvc-dispatcher-2</servlet-name>

        <url-pattern>/services/\*</url-pattern>

    </servlet-mapping>

    <!-- location of the root application context xml file -->

    <context-param>

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

        <param-value>/WEB-INF/mvc-dispatcher-servlet.xml, /WEB-INF/mvc-dispatcher-2-servlet.xml</param-value>

    </context-param>

    <!-- context loader listener -->

    <listener>

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

    </listener>

</web-app>

# Difference between constructor and setter injection

1. **Partial dependency**: can be injected using setter injection but it is not possible by constructor. Suppose there are 3 properties in a class, having 3 arg constructor and setters methods. In such case, if you want to pass information for only one property, it is possible by setter method only.
2. **Overriding**: Setter injection overrides the constructor injection. If we use both constructor and setter injection, IOC container will use the setter injection.
3. **Changes**: We can easily change the value by setter injection. It doesn't create a new bean instance always like constructor. So setter injection is flexible than constructor injection.

# Autowiring in Spring

Autowiring feature of spring framework enables you to inject the object dependency implicitly. It internally uses setter or constructor injection.

Autowiring can't be used to inject primitive and string values. It works with reference only.

#### **Q5). Name some popular and mostly used Spring Modules.**

* Spring Context- Used for Dependency Injection
* Spring AOP- Used to implement aspect-oriented programming
* Spring DAO- Used for database operations with the help of DAO patterns
* Spring JDBC- uSed for DataSource and JDBC support
* Spring ORM- Used for creating web applications
* Spring Web Module- Used to create the web applications
* Spring MVC- It is also known as Model-View-Controller used to create and implement the web applications

**Q10). Define Spring Bean.**

When a normal java class starts with IoC container it is called Spring Bean. To get the Spring Bean instance we can use Spring Application Context. The life cycle of Spring Bean is managed by IoC container, it also manages the scope of bean and the dependencies required in case of bean injection in any module.

ContextLoaderListener reads the spring configuration file (with value given against “**contextConfigLocation**” in web.xml), parse it and loads the beans defined in that config file

**Difference between <context:annotation-config> vs <context:component-scan>?**

**<context:annotation-config> = Scanning and activating annotations in “already registered beans”.**

**<context:component-scan> = Bean Registration + Scanning and activating annotations**

**Difference between @Component, @Controller, @Repository & @Service annotations?**

**Ans.** 1) The @Component annotation marks a java class as a bean so the component-scanning mechanism of spring can pick it up and pull it into the application context.

@Component

public class EmployeeDAOImpl implements EmployeeDAO {

    ...

}

2) The @Repository annotation is a specialization of the @Component annotation with similar use and functionality. In addition to importing the DAOs into the DI container, it also makes the unchecked exceptions (thrown from DAO methods) eligible for translation into Spring DataAccessException.

3) The @Service annotation is also a specialization of the component annotation. It doesn’t currently provide any additional behavior over the @Component annotation, but it’s a good idea to use @Service over @Component in service-layer classes because it specifies intent better

4) @Controller annotation marks a class as a Spring Web MVC controller. It too is a @Componentspecialization, so beans marked with it are automatically imported into the DI container. When you add the @Controller annotation to a class, you can use another annotation i.e. @RequestMapping; to map URLs to instance methods of a class..