**SPRINGBOOT INTERVIEW QUESTIONS**

1. **Loose Coupling :**

loose coupling refers to the practice of designing components in a way that minimizes their dependencies on each other.

Dependency injection is a design pattern used in Spring where the dependencies of a class are provided from an external source rather than created within the class itself. This helps in decoupling the classes and makes them more reusable and easier to test.

1. **what is a dependency?**

Dependencies can be of different types, such as code libraries, frameworks, databases, external services, or other components within the same application.

1. **what is IOC in spring?**

Inversion of Control (IoC) is a design principle in software engineering where the control of object creation and lifecycle is transferred to an external entity (often a framework or container) rather than the objects (application code) themselves.

1. **what is auto wiring?**

@Autowired annotation is used for automatic dependency injection in Spring. This annotation allows Spring to automatically resolve and inject collaborating beans into your bean.

1. **Bean Factory v/s Application Context**

\*\*Bean Factory\*\*:

- Bean Factory is the simplest container in Spring that provides the basic support for dependency injection.

- It is the core interface for accessing and managing beans.

- Bean Factory provides the basic features of dependency injection and bean lifecycle management.

- Bean Factory is the foundation upon which more advanced container implementations, such as Application Context, are built.

\*\*Application Context\*\*:

- Application Context is a more advanced container than the Bean Factory and builds upon its capabilities.

- It is the preferred way to access Spring beans in a Spring application.

- Application Context includes all functionality of the Bean Factory and provides additional features such as event propagation, internationalization support, resource loading, etc.

- It is an interface representing the Spring IoC container and is responsible for instantiating, configuring, and assembling beans.

\*\*IOC Container\*\*:

- IOC Container is a general concept that refers to the mechanism of managing the lifecycle of objects and their dependencies.

- In the context of Spring, the IOC Container is implemented by Application Context (or Bean Factory).

- IOC Container is responsible for creating, initializing, configuring, and managing beans in a Spring application.

- It helps in decoupling the application components and promotes the principle of Inversion of Control.

In summary, Bean Factory is the basic container providing core features, Application Context is an advanced container with additional functionalities, and IOC Container is the overarching concept that refers to the mechanism of managing beans and their dependencies in a Spring application.

1. **Important roles of an IOC container?**

* Dependency Injection
* Lifecycle Management

It controls the creation, initialization, configuration, and destruction of objects according to the specified configuration

* Configuration Management

allows developers to configure the application components and their dependencies externally, typically using XML, Java annotations, or Java configuration classes.

* AOP Support

AOP allows developers to separate cross-cutting concerns such as logging, security, and transaction management from the core business logic

* Event Handling

IoC containers often provide event handling mechanisms that allow components to publish events and subscribe to events

1. **How do you create an application context with spring?**

Define a configuration class or XML configuration file where you specify the beans and their dependencies. This is where you define the components that make up your application context.

*configuration class example*:

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class MyApp {

public static void main(String[] args) {

AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

MyService myService = context.getBean(MyService.class);

myService.doSomething();

context.close(); // Don't forget to close the context when done

}

}

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

public class AppConfig {

@Bean

public MyService myService() {

return new MyService();

} }

*XML configuration file example*:

<?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.xsd">

<bean id="myService" class="com.example.MyService" />

</beans>

package com.example;

public class MyService {

public void doSomething() {

System.out.println("Doing something in MyService");

}

}

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MyApp {

public static void main(String[] args) {

ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

MyService myService = context.getBean("myService", MyService.class);

myService.doSomething();

context.close();

}

}

1. **how to do define component scan in xml or configuration class? and how is it done with springboot?**

Component scanning is the process of automatically discovering and registering Spring beans (components, services, repositories, etc.) based on certain criteria, such as annotations or package names.

When you annotate a configuration class with @ComponentScan, Spring will scan the specified packages and register any beans that meet the criteria specified by annotations such as @Component, @Service, @Repository, etc. This eliminates the need to manually define each bean in a configuration class, making your Spring application more scalable and maintainable.

1.In your XML configuration file (e.g., applicationContext.xml), include the following element to enable component scanning:

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

Specify the base package(s) to scan for components. In this example, Spring will scan the "com.example" package and register beans annotated with @Component, @Service, @Repository, etc.

2. Java Based Configuration

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

@Configuration

@ComponentScan(basePackages = "com.example")

public class AppConfig {

// Additional bean definitions can go here

}

1. In a Spring Boot application, component scanning is automatically enabled by default. Spring Boot uses the @SpringBootApplication annotation, which includes @ComponentScan with the base package of the application main class.

In a **Spring Boot application**, component scanning is automatically enabled by default. Spring Boot uses the @SpringBootApplication annotation, which includes @ComponentScan with the base package of the application main class.

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class MySpringBootApplication {

public static void main(String[] args) {

SpringApplication.run(MySpringBootApplication.class, args);

}

}

9. **explain @component and @autowired annotations**

1. **@Component Annotation**:
   * The @Component annotation is used to indicate that a class is a Spring component. Spring components are Java classes that are managed by the Spring IoC container, allowing them to be automatically discovered and registered as Spring beans.
   * By annotating a class with @Component, you are essentially telling Spring to manage that class as a bean, making it available for dependency injection and other Spring features.
   * There are also specialized stereotypes of @Component such as @Service, @Repository, and @Controller which are used to define specific types of components in the application.
2. **@Autowired Annotation**:
   * The @Autowired annotation is used to inject dependencies into Spring-managed beans automatically. When a bean is annotated with @Autowired, Spring will attempt to find a matching bean to inject into that dependency at runtime.
   * It allows you to achieve loose coupling by letting Spring handle the creation and injection of dependencies, rather than manually instantiating and passing dependencies.
   * @Autowired can be used to inject dependencies into other Spring components, such as services, controllers, and repositories.

import org.springframework.stereotype.Component;

import org.springframework.beans.factory.annotation.Autowired;

@Component

public class MyService {

private MyRepository myRepository;

@Autowired

public MyService(MyRepository myRepository) {

this.myRepository = myRepository;

}

// Other methods using myRepository

}

1. **@controller , @service and @repository annotations**

In Spring Framework, @Controller, @Service, and @Repository are three specialized stereotypes of the @Component annotation, each serving a specific purpose in defining and categorizing Spring-managed beans:

@Controller:

The @Controller annotation is used to mark a class as a Spring MVC controller. It is typically used in web applications to handle HTTP requests, process user input, and generate responses.

Controllers in Spring MVC are responsible for processing user requests, invoking business logic, and returning the appropriate view or data to the client.

@Controller classes are often used to implement the presentation layer in a Spring MVC web application.

@Service:

The @Service annotation is used to mark a class as a service component in the business logic layer of an application. It typically encapsulates business logic, data manipulation, and other services.

Service classes in Spring are commonly used to implement the business logic of an application, separating it from the presentation and data access layers.

@Service classes are meant to contain and execute complex business logic and are often used to coordinate operations between multiple @Repository components.

@Repository:

The @Repository annotation is used to mark a class as a data access component or repository. It is typically used to interact with a database, perform CRUD (Create, Read, Update, Delete) operations, and manage data access logic.

Repository classes in Spring are responsible for data access and persistence, abstracting the interaction with the underlying data source (e.g., database, file system).

@Repository classes often use Spring's Data Access Object (DAO) pattern to provide a consistent and reusableway to access and manipulate data.

1. **Default scope of a bean? other scopes available? spring beans are thread safe?**

The default scope of a Spring bean is singleton ( 1 per AC ), GOF ( 1 per JVM)

OTHER SCOPES:

1. **Singleton (default)**:
   * Only one instance of the bean is created and shared across the Spring container.
2. **Prototype**:
   * A new instance of the bean is created every time it is requested. This scope is useful for beans that require a new instance for each request or usage.
3. **Request**:
   * A new instance of the bean is created for each HTTP request in a web application context.
4. **Session**:
   * A new instance of the bean is created for each HTTP session in a web application context

**Thread Safety of Spring Beans:**

* By default, singleton beans in Spring are not thread-safe. If a singleton bean contains mutable state (e.g., instance variables that can be modified), it can lead to concurrency issues when accessed by multiple threads concurrently.
* Prototype beans, on the other hand, are thread-safe by nature since a new instance is created for each request, avoiding shared mutable state between threads.
* To ensure thread safety for singleton beans with mutable state, you can use synchronization, thread-safe data structures, or design patterns such as immutable objects or thread-local variables.

1. **How is spring's singleton bean different from gang of four singleton pattern?**

**Spring Singleton Bean**:

* In Spring, a singleton bean is a single shared instance of a bean within the Spring container.

**Gang of Four Singleton Pattern**:

* The Gang of Four Singleton design pattern is a creational pattern that ensures a class has only one instance and provides a global point of access to that instance. It is typically implemented by defining a static method (e.g., getInstance()) in the class that returns the single instance.

1. **Difference type of dependency injections? how do you choose between constructor and setter injections?**

In Spring Framework, there are three main types of dependency injection: constructor injection, setter injection, and field injection. Each type has its own advantages and use cases:

1. **Constructor Injection**:
   * Constructor injection involves passing dependencies to a class through its constructor. The dependencies are provided at the time of object creation.
   * Use constructor injection when:
     + The dependencies are mandatory for the object to function correctly.
     + You want to ensure that the object is fully initialized before it is used.
     + You want to create immutable objects.
2. **Setter Injection:**

Setter injection involves setting dependencies on a class using setter methods. The dependencies can be set after the object is created.

Use setter injection when:

* The dependencies are optional or can be changed at runtime.
* The class has a large number of dependencies that are not always needed.
* You want to decouple the object creation and dependency injection process.

3.**Field Injection**:

* Field injection involves injecting dependencies directly into fields of a class using annotations like @Autowired

**How to Choose Between Constructor and Setter Injections**:

* Use constructor injection when the dependencies are mandatory, and the object cannot function without them.
* Use setter injection when the dependencies are optional or can change at runtime.
* Prefer constructor injection for required dependencies to ensure the object is fully initialized.
* Use setter injection for optional dependencies or when you want to provide flexibility in changing dependencies.

14**. How do you choose between xml and java configuration in spring?**

Choosing between XML and Java configuration in Spring depends on various factors and preferences. Here are some considerations to help you decide which configuration approach to use:

**XML Configuration:**

1. **Historical Legacy**: XML configuration has been a traditional way of configuring Spring applications and may be preferred in projects with existing XML-based configurations.
2. **Separation of Concerns**: XML configuration can help separate configuration details from business logic, making it easier to understand the application's configuration at a glance.
3. **Externalization**: XML configuration allows for externalization of configuration details, which can be beneficial for environments where configuration changes need to be made without recompiling the code.
4. **IDE Support**: Some developers may find XML configuration easier to work with in certain integrated development environments (IDEs) that provide better support for XML editing and validation.

**Java Configuration:**

1. **Type Safety**: Java configuration provides type safety and compile-time checking, reducing the likelihood of configuration errors that might go unnoticed in XML configurations.
2. **Programmatic Flexibility**: Java configuration allows for programmatic logic and conditional bean definitions, enabling more dynamic and flexible configuration options.
3. **Annotation Support**: Java configuration works well with annotations like @Component, @Autowired, and others, providing a more modern and concise way to define beans and dependencies.
4. **Refactoring and Testing**: Java configuration can make refactoring and testing easier, as configuration details are directly visible in the codebase and can benefit from tools like code analysis and refactoring.

**15. How does spring do autowiring?**

1. **Types of Autowiring Modes**:
   * **No Autowiring** (autowire="no"): In this mode, autowiring is disabled, and you must wire dependencies explicitly using constructor injection, setter injection, or field injection.
   * **Autowiring by Type** (autowire="byType"): Spring automatically wires a property by matching the data type of the property with a bean in the container. If there is exactly one bean of the required type, it will be injected. If multiple beans of the same type exist, an exception will be thrown.
   * **Autowiring by Name** (autowire="byName"): Spring automatically wires a property by matching the property name with a bean name in the container. The property will be injected with the bean having the same name as the property.

**16. @qualifier and @primary**

If multiple beans of the same type are available for autowiring, you can use the @Qualifier annotation to specify which bean should be injected.

import org.springframework.stereotype.Repository;

@Repository("userDAOImpl1")

public class UserDAOImpl1 implements UserDAO {

@Override

public void saveUser(User user) {

// Implementation for saving user in database

}

}

@Service

public class UserService {

@Autowired

@Qualifier("userDAOImpl1")

private UserDAO userDAO;

public void saveUser(User user) {

userDAO.saveUser(user);

}

}

You can use the @Primary annotation on a bean to indicate that it is the primary candidate for autowiring when multiple beans of the same type exist.

@Component

@Primary

public class UserDAOImpl implements UserDAO {

// Implementation of UserDAO interface }

**17. what is CDI? does spring support CDI? would you recommend to use CDI or spring annotations?**

CDI stands for Contexts and Dependency Injection, which is a Java EE specification that defines a set of services for dependency injection and contextual lifecycle management in Java EE applications. CDI provides a powerful and flexible way to manage dependencies and context in Java EE applications.

Spring does not directly support CDI as it is a separate specification within the Java EE ecosystem

CDI provides features like interceptor bindings, event handling, decorators, and more, which may be useful for certain types of applications.

**18. Major features in different versions of Spring? 4.0 features and 5.0 features?**

Spring 4.0 Features:

1. Support for Java 8: Spring 4.0 introduced support for Java 8 features such as lambda expressions and the new Date and Time API.
2. WebSocket support: Spring 4.0 added support for WebSocket communication, allowing for real-time, bidirectional communication between clients and servers.
3. Spring MVC improvements: Spring 4.0 included enhancements to the Spring MVC framework, making it more flexible and easier to use.
4. HTML5 support: Spring 4.0 added support for HTML5 features and APIs, making it easier to develop modern web application

Spring 5.0 Features:

1. Reactive programming support: Spring 5.0 introduced a new reactive programming framework, allowing developers to build reactive applications that can handle large numbers of concurrent users.
2. Functional endpoints: Spring 5.0 introduced functional-style endpoints, allowing developers to define RESTful APIs using a more concise and expressive syntax.
3. Improved testing support: Spring 5.0 included enhancements to the testing framework, making it easier to write and run tests for Spring applications.

**19. Important Spring Modules and Projects?**

Some important Spring modules and projects include:

1. Spring Core Container: Provides the core functionality of the Spring framework, including Inversion of Control (IoC) and Dependency Injection (DI).
2. Spring Data: Simplifies data access using Spring and provides a consistent programming model for interacting with different data sources.
3. Spring Security: Provides comprehensive security services for Java applications, including authentication, authorization, and protection against common security vulnerabilities.
4. Spring MVC: A web framework that simplifies the development of web applications by providing a model-view-controller architecture.
5. Spring Boot: Simplifies the setup and configuration of Spring applications by providing a convention-over-configuration approach and embedded server support.
6. Spring Cloud: Provides tools and frameworks for building distributed systems and microservices using Spring.
7. Spring Batch: Supports the development of batch processing applications, allowing for the processing of large volumes of data.
8. Spring Integration: Facilitates the integration of disparate systems and applications by providing messaging and integration patterns.
9. Spring Web Services: Simplifies the development of SOAP-based web services using Spring.
10. Spring Test: Provides support for writing unit and integration tests for Spring applications.

**20. Design patterns used in the Spring framework**

Some common design patterns used in the Spring framework include:

1. Dependency Injection (DI) pattern
2. Singleton pattern
3. Factory pattern
4. Proxy pattern
5. Template method pattern
6. Observer pattern

**21. What is an embedded server ? why is it important? default embedded server in spring boot ? what other embedded servers supported by springboot?**

An embedded server is a web server that is bundled within an application or framework, allowing the application to run as a standalone executable without the need for external server software to be installed separately.

The default embedded server is Apache Tomcat.

Spring Boot also supports other embedded servers such as Jetty and Undertow.

**22. What are Starter Projects? Examples of Important Starter Projects?**

pre-configured templates that help developers quickly bootstrap their applications with essential dependencies and configurations

1. **spring-boot-starter-web**: This starter project includes all the necessary dependencies to build web applications using Spring MVC, embedded servers, and other web-related components.
2. **spring-boot-starter-data-jpa**: This starter project provides dependencies for integrating Spring Data JPA for working with relational databases in Spring Boot applications.
3. **spring-boot-starter-security**: This starter project includes dependencies for adding security features such as authentication and authorization to Spring Boot applications.
4. **spring-boot-starter-test**: This starter project provides dependencies for writing unit and integration tests in Spring Boot applications using popular testing frameworks like JUnit and Mockito.
5. **spring-boot-starter-actuator**: This starter project includes dependencies for monitoring and managing Spring Boot applications using built-in management endpoints.

**23**. **What is Starter Parent? Different things that are defined in Starter Parent?**

Spring Boot Starter Parent inherits all of these properties from spring-boot-dependencies.

The **spring-boot-starter-parent**is one of the starters that Spring Boot provides. It provides default configurations for our applications. It also provides dependency and plugin management for applications built in using Maven

**24. What is application.properties?** **Important things that can be customized in application.properties?**

Some important things that can be customized in application.properties include:

1. **Server Configuration**: You can configure properties related to the embedded server, such as server port, context path, and server-specific settings.
2. **Database Configuration**: Properties for configuring data sources, database connection settings, and dialects for different database systems can be specified in application.properties.
3. **Logging Configuration**: You can customize logging levels, log file locations, log formats, and other logging-related settings using properties in the configuration file.
4. **Spring Profiles**: application.properties can define profiles for different environments (e.g., development, production) and specify profile-specific configurations.
5. **Security Configuration**: Properties related to security settings, authentication mechanisms, and authorization rules can be customized in the configuration file.
6. **Thymeleaf and View Configuration**: Settings for Thymeleaf templates, view resolver configurations, and other view-related properties can be specified in application.properties.
7. **External Service Configuration**: You can configure properties for integrating with external services, such as API endpoints, credentials, timeouts, and connection settings.

**25. @ ConfigurationProperties (custom.property =value)**

@Component

@ConfigurationProperties(prefix = "custom")

public class CustomProperties { private String property; …}

**26.** **What is profile? how do you define beans for a specific profile? how do you have different configuration for different environments?**

**Define Profile-Specific Configuration Classes:**

@Configuration

@Profile("development")

public class DevelopmentConfig {

// Define beans specific to the development profile

}

@Configuration

@Profile("production")

public class ProductionConfig {

// Define beans specific to the production profile

}

**Different Configuration for Different Environments**: To have different configuration for different environments, you can create separate property files for each environment (e.g., **application-dev.properties, application-prod.properties**) and specify profile-specific properties in these files

**Activate Profiles**: You can activate profiles in Spring Boot applications by setting the spring.profiles.active property in the application.properties

**27. CommandLineRunner**

The CommandLineRunner is an interface in Spring Boot. When a class implements this interface, Spring Boot will automatically run its run method after loading the application context. Usually, we use this CommandLineRunner to perform startup tasks like user or database initialization, seeding, or other startup activities.

Here’s a simplified sequence to run CommandLineRunner. in Spring Boot.

1. Application starts.
2. Spring Boot initializes and configures beans, properties, and the application context.
3. CommandLineRunner (or ApplicationRunner) methods are executed.
4. The application is now ready to serve connections or requests.

**28. SpringBoot Actuator**

Spring Boot Actuator is a set of production-ready features that allow you to monitor and manage your Spring Boot application. It provides endpoints that give you insights into various aspects of your application, such as health status, metrics, environment details, and more.

Application.properties

management.endpoints.web.exposure.include=\*

management.endpoint.health.show-details=always

* To check the health status of your application, you can access the /actuator/health endpoint.
* To view metrics about your application, you can access the /actuator/metrics endpoint.
* To get information about your application's configuration properties, you can access the /actuator/configprops endpoint.

**JDBC,Spring JDBC & JPA**

JdbcTemplate is a class in the Spring Framework that simplifies database access using JDBC (Java Database Connectivity). It provides methods to execute SQL queries, update statements, and stored procedures without needing to write boilerplate code for handling database connections, statements, result sets, etc.

RowMapper is an interface in the Spring Framework used with JdbcTemplate to map a row of a ResultSet to an object. It is used to convert the rows of a ResultSet into Java objects. By implementing the RowMapper interface, you can define how the data from the ResultSet should be mapped to your custom Java object.A computer screen shot of a code

Description automatically generated

1. **What is JPA ? What is Hibernate?**

A JPA (Java Persistence API) is a specification of Java which is used to access, manage, and persist data between Java object and relational database. It is considered as a standard approach for Object Relational Mapping.

JPA doesn't perform any operation by itself. Thus, it requires implementation. So, ORM tools like Hibernate, TopLink, and iBatis implements JPA specifications for data persistence.

**3. how do you define an entity in JPA? what is an Entity Manager? what is a persistence context?**

An entity represents a persistent data entity, and instances of an entity class correspond to records in the database table. To define an entity in JPA, you annotate a Java class with the @Entity annotation

A screen shot of a computer code

Description automatically generated

In JPA, the EntityManager interface is used to interact with the database. It provides methods for persisting, merging, removing, and querying entities. Here is an example of using EntityManager to persist an entity:

A screenshot of a computer code

Description automatically generated

The **persistence context** in JPA is an EntityManager's cache of managed entities. It represents a set of managed entity instances that are currently in scope for a particular EntityManager.

**RESTful Web Services**

**1.What is REST? What are the key concepts in designing RESTful API?**

* Every object has some state(data) and behaviour(methods).In order to transfer state of object on server at particular instance of time to client, some sort of representation is needed like JSON or xml or any other format.
* So REST is about creating representation of object's current state and transferring that representation over network.