**he Expertise and Skills**

* Deep knowledge and working experience with middleware Java stack, Spring frameworks and Kafka
* Experience with PL/SQL programming and ETL tools.
* Exposure to UI tech stack HTML, CSS and Angular is a plus
* Experience of AWS/Azure services and deployment
* Experience working in an Agile software delivery environment, with exposure to CI/CD.
* Passionate about engineering excellence with a strong understanding of object-orientated programming.
* Knack for writing clean, testable, readable, and easily maintainable code
* Proven understanding of fundamental design principles for building a scalable application
* You are committed to collaboration and communication in your work

**The Value You Deliver**

* You will lead design and implementation of software solutions that meet functional and non-functional requirements and that are aligned to the program’s strategic direction
* You will lead application design efforts, participate in design reviews and code reviews.
* Fully engaging in product quality plan efforts, including test case development, design and code reviews and regression testing.
* Working to improve engineering practice in the areas of ATDD and CI/CD.
* Engaging in working with global team of developers in leading standard methodologies for software engineering

**CORE JAVA**

1. What is Java? Java is a high-level, object-oriented programming language developed by Sun Microsystems (now owned by Oracle). It is widely used to create web applications, desktop applications, mobile applications, and other types of software.
2. What are the features of Java? Java has several features such as platform independence, object-oriented programming, memory management, exception handling, multithreading, etc.
3. What is a JVM? JVM stands for Java Virtual Machine. It is responsible for executing the Java bytecode generated by the Java compiler.
4. What is the difference between JDK and JRE? JDK stands for Java Development Kit, which contains tools for developing and compiling Java programs. JRE stands for Java Runtime Environment, which is required to run Java programs.
5. What is a constructor? A constructor is a special method that is used to initialize objects. It has the same name as the class and does not have a return type.
6. What is an object in Java? An object is an instance of a class. It has state (data) and behavior (methods).
7. What is inheritance? Inheritance is a mechanism in which one class acquires the properties (fields and methods) of another class.
8. What is the difference between inheritance and composition? Inheritance is a mechanism in which one class acquires the properties of another class. Composition is a mechanism in which an object contains another object as one of its instance variables.
9. What is method overloading? Method overloading is a feature in which a class has two or more methods with the same name but different parameters.
10. What is method overriding? Method overriding is a feature in which a subclass provides its own implementation of a method that is already provided by its parent class.
11. What is polymorphism? Polymorphism is the ability of an object to take on many forms. In Java, polymorphism can be achieved through method overloading and method overriding.
12. What is encapsulation? Encapsulation is a mechanism in which the data and methods that act on the data are bound together within a single unit.
13. What is a package? A package is a collection of related classes and interfaces.
14. What is an interface? An interface is a collection of abstract methods that can be implemented by a class.
15. What is an abstract class? An abstract class is a class that cannot be instantiated. It can only be used as a superclass for other classes.
16. What is a static method? A static method is a method that belongs to the class rather than to any particular object of the class.
17. What is a final keyword? The final keyword is used to declare a constant variable or to prevent a method or class from being overridden.
18. What is a static variable? A static variable is a variable that belongs to the class rather than to any particular object of the class.
19. What is a try-catch block? A try-catch block is used to handle exceptions in Java. The code that may throw an exception is placed in the try block, and the code that handles the exception is placed in the catch block.
20. What is an exception? An exception is an event that occurs during the execution of a program that disrupts the normal flow of the program's instructions.
21. What is the difference between checked and unchecked exceptions? Checked exceptions are exceptions that are checked at compile time. Unchecked exceptions are exceptions that are not checked at compile time.
22. What is a thread? A thread is a lightweight sub-process that can run concurrently with other threads in the same program.
23. What is synchronization? Synchronization is a mechanism that is used to control access to shared resources in a multithreaded environment. What is the difference between wait() and sleep()? wait() is a method that is used to make a thread wait until another thread signals that a particular condition has been met. sleep() is a method that is used to make a thread pause for a specified period of time.
24. What is a synchronized block? A synchronized block is a block of code that is synchronized on a particular object. It ensures that only one thread can execute the block at a time.
25. What is a collection? A collection is a group of objects that can be manipulated as a single unit.
26. What is an array? An array is a collection of elements of the same type.
27. What is a List? A List is an ordered collection of elements that can contain duplicates.
28. What is a Set? A Set is a collection of elements that cannot contain duplicates.
29. What is a Map? A Map is a collection of key-value pairs.
30. What is a Iterator? An Iterator is an interface that is used to iterate over the elements in a collection.
31. What is a Comparable interface? The Comparable interface is used to define the natural ordering of a class. It contains a single method, compareTo(), that is used to compare two objects.

*class Student implements Comparable<Student>{*

*int rollno;*

*String name;*

*int age;*

*Student(int rollno,String name,int age){*

*this.rollno=rollno;*

*this.name=name;*

*this.age=age;*

*}*

*public int compareTo(Student st){*

*if(age==st.age)*

*return 0;*

*else if(age>st.age)*

*return 1;*

*else*

*return -1;*

*}*

*Use:*

*List<Students> stds=new ArrayList(); stds.add(new Student……………….)*

*Collections.sort(stds); // This will sort based on the age , because it is natural comparison is defied on age.*

1. What is a Comparator interface? The Comparator interface is used to define a custom ordering of a class. It contains a single method, compare(), that is used to compare two objects.

import java.util.\*;

class AgeComparator implements Comparator{

public int compare(Object o1,Object o2){

Student s1=(Student)o1;

Student s2=(Student)o2;

if(s1.age==s2.age)

return 0;

else if(s1.age>s2.age)

return 1;

else

return -1;

}

}

1. What is serialization? Serialization is the process of converting an object into a stream of bytes so that it can be stored in a file or transmitted over a network.
2. What is deserialization? Deserialization is the process of converting a stream of bytes back into an object.
3. What is the difference between Serializable and Externalizable? Serializable is a marker interface that indicates that an object can be serialized. Externalizable is an interface that provides more control over the serialization process.
4. What is the difference between equals() and ==? equals() is a method that is used to compare the contents of two objects. == is an operator that is used to compare the references of two objects.
5. What is a StringBuilder? A StringBuilder is a mutable sequence of characters. It is similar to a String but can be modified.
6. What is a StringBuffer? A StringBuffer is a thread-safe, mutable sequence of characters. It is similar to a StringBuilder but can be accessed by multiple threads simultaneously.
7. What is a lambda expression? A lambda expression is a concise way of representing an anonymous function. It is a function that can be created without belonging to any class.
8. What is a functional interface? A functional interface is an interface that contains exactly one abstract method. It is used to represent lambda expressions.
9. What is the diamond operator? The diamond operator is a feature that was introduced in Java 7. It is used to infer the type arguments of a generic class.
10. What is var? var is a keyword that was introduced in Java 10. It is used to declare local variables with inferred types.

var x = 100;

1. What is the Stream API? The Stream API is a feature that was introduced in Java 8. It is used to process collections of data in a functional and parallel way.
2. What is the Optional class? The Optional class is a container object that may or may not contain a non-null value. It is used to avoid NullPointerExceptions.
3. What is the difference between an ArrayList and a LinkedList? An ArrayList is an ordered collection that is implemented as an array. A LinkedList is an ordered collection that is implemented as a linked list. Accessing elements in an ArrayList is faster than in a LinkedList, but inserting and deleting elements is slower.
4. What is the difference between a HashSet and a TreeSet? A HashSet is an unordered collection that is implemented using a hash table. A TreeSet is an ordered collection that is implemented using a red-black tree. HashSet provides constant-time performance for the basic operations (add, remove, contains), while TreeSet provides log(n) time performance.
5. What is the difference between checked and unchecked exceptions? Checked exceptions are exceptions that are checked at compile time. They must be caught or declared in the method signature eg., ClassNotFoundEx, FileNotFoundEx, SQLException. Unchecked exceptions are exceptions that are not checked at compile time. They do not need to be caught or declared ArthmeticEx,NullPointerEx,ArrayIndexOutOfBounds.
6. What is a try-with-resources statement? A try-with-resources statement is a statement that is used to automatically close resources that are opened in a try block. It is a convenient way of managing resources without needing to explicitly close them.
7. What is the difference between a static and non-static method? A static method is a method that belongs to a class, rather than an instance of a class. It can be called without creating an instance of the class. A non-static method is a method that belongs to an instance of a class. It can only be called on an instance of the class.
8. What is the difference between an interface and an abstract class?

An interface is a contract that specifies a set of methods that a class must implement. An abstract class is a class that cannot be instantiated and may contain abstract methods that must be implemented by subclasses. In Java, a class can implement multiple interfaces, but can only inherit from a single abstract class.

Example of an interface:

public interface Animal {

void speak();

}

public class Dog implements Animal {

@Override

public void speak() {

System.out.println("Woof!");

}

}

Example of an abstract class:

public abstract class Shape {

public abstract double area();

}

public class Rectangle extends Shape {

private double width;

private double height;

public Rectangle(double width, double height) {

this.width = width;

this.height = height;

}

@Override

public double area() {

return width \* height;

}

}

1. What is the purpose of the finalize() method?

The finalize() method is a method that is called by the garbage collector before an object is destroyed. It can be used to perform any necessary cleanup before the object is freed from memory.

Example:

public class MyClass {

private int id;

public MyClass(int id) {

this.id = id;

}

@Override

protected void finalize() throws Throwable {

System.out.println("Object with ID " + id + " is being destroyed.");

}

}

public class Main {

public static void main(String[] args) {

MyClass obj = new MyClass(123);

obj = null;

System.gc(); // request garbage collection

}

}

1. What is the difference between a private and a protected method?

A private method is a method that can only be accessed within the same class. A protected method is a method that can be accessed within the same class and its subclasses.

Example:

public class Animal {

private void breathe() {

System.out.println("I am breathing.");

}

protected void move() {

System.out.println("I am moving.");

}

}

public class Dog extends Animal {

public void doDogThings() {

// breathe(); // error: private method cannot be accessed

move(); // okay: protected method can be accessed

}

}

1. What is the difference between a static and non-static variable?

A static variable is a variable that belongs to the class, rather than an instance of the class. It is shared among all instances of the class. A non-static variable is a variable that belongs to an instance of the class. Each instance has its own copy of the variable.

Example:

public class MyClass {

private static int count = 0; // static variable

private int id; // non-static variable

public MyClass() {

count++;

id = count;

}

public void printId() {

System.out.println("My ID is " + id);

}

public static void printCount() {

System.out.println("There are " + count + " instances of MyClass.");

}

}

public class Main {

public static void main(String[] args) {

MyClass obj1 = new MyClass();

MyClass obj2 = new MyClass();

MyClass obj3 = new MyClass();

obj1.printId(); // My ID is 1

obj2.printId(); // My ID is 2

obj3.printId(); // My ID is 3

MyClass.printCount(); // There are 3 instances of MyClass.

}

}

1. What is the difference between a public and default access modifier?

A public access modifier makes a class, method or variable accessible to all classes in all packages. A default access modifier (also known as package-private) makes a class, method or variable accessible only to classes within the same package.

Example:

csharp

Copy code

package com.example;

public class MyClass {

public String name; // public variable

int age; // default variable

public void doSomething() { // public method

System.out.println("I am doing something.");

}

void doSomethingElse() { // default method

System.out.println("I am doing something else.");

}

}

package com.example2;

import com.example.MyClass;

public class AnotherClass {

public void doAnotherThing() {

MyClass obj = new MyClass();

obj.name = "John"; // okay: public variable can be accessed

// obj.age = 30; // error: default variable cannot be accessed

obj.doSomething(); // okay: public method can be accessed

// obj.doSomethingElse(); // error: default method cannot be accessed

}

}

1. What is the difference between a deep copy and a shallow copy?

A deep copy creates a new object with a new memory location, and copies the values of all fields (including nested objects) into the new object. A shallow copy creates a new object with the same memory location as the original object, and copies only the references of the fields (not the actual objects).

Example:

public class Person {

private String name;

private Address address;

public Person(String name, Address address) {

this.name = name;

this.address = address;

}

public Person(Person other) { // copy constructor

this.name = other.name;

this.address = new Address(other.address);

}

// getters and setters

}

public class Address {

private String city;

public Address(String city) {

this.city = city;

}

public Address(Address other) { // copy constructor

this.city = other.city;

}

// getters and setters

}

public class Main {

public static void main(String[] args) {

Address address = new Address("New York");

Person person1 = new Person("John", address);

Person person2 = new Person(person1); // deep copy

System.out.println(person1.getAddress().getCity()); // New York

System.out.println(person2.getAddress().getCity()); // New York

person1.getAddress().setCity("Boston");

System.out.println(person1.getAddress().getCity()); // Boston

System.out.println(person2.getAddress().getCity()); // New York (deep copy is not affected)

}

}

1. What is the difference between the equals() and == operators?

The == operator compares two objects for reference equality. It returns true if the two objects have the same memory location. The equals() method compares two objects for value equality. It returns true if the two objects have the same values for all their fields.

Example:

public class Person {

private String name;

private int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

// getters and setters

@Override

public boolean equals(Object other) {

if (this == other) return true;

if (!(other instanceof Person)) return false;

Person person = (Person) other;

return age == person.age && Objects.equals(name, person.name);

}

}

public class Main {

public static void main(String[] args) {

Person

person1 = new Person("John", 30);

Person person2 = new Person("John", 30);

Person person3 = person1;

System.out.println(person1 == person2); // false (different memory location)

System.out.println(person1.equals(person2)); // true (same values for all fields)

System.out.println(person1 == person3); // true (same memory location)

}

}

1. What is the difference between an abstract class and an interface?

An abstract class can have both abstract and non-abstract (concrete) methods, while an interface can only have abstract methods (prior to Java 8). An abstract class can also have instance variables, while an interface cannot. A class can only extend one abstract class, but can implement multiple interfaces.

Example:

public abstract class Shape {

protected int x;

protected int y;

public Shape(int x, int y) {

this.x = x;

this.y = y;

}

public abstract double getArea();

public void move(int deltaX, int deltaY) {

x += deltaX;

y += deltaY;

}

}

public interface Printable {

void print();

}

public class Rectangle extends Shape implements Printable {

private int width;

private int height;

public Rectangle(int x, int y, int width, int height) {

super(x, y);

this.width = width;

this.height = height;

}

@Override

public double getArea() {

return width \* height;

}

@Override

public void print() {

System.out.println("Printing rectangle...");

}

}

public class Main {

public static void main(String[] args) {

Shape shape = new Rectangle(0, 0, 5, 10);

shape.move(2, 3);

System.out.println(shape.getArea()); // 50.0

Printable printable = (Printable) shape;

printable.print(); // Printing rectangle...

}

}

1. What is the difference between a checked and an unchecked exception?

A checked exception is a type of exception that the compiler forces the programmer to handle explicitly, either by using a try-catch block or by declaring it in the method signature with the throws keyword. Examples include IOException and SQLException. An unchecked exception is a type of exception that the compiler does not force the programmer to handle explicitly. Examples include NullPointerException and ArrayIndexOutOfBoundsException.

Example:

public class Main {

public static void main(String[] args) {

// checked exception

try {

BufferedReader reader = new BufferedReader(new FileReader("file.txt"));

String line = reader.readLine();

System.out.println(line);

reader.close();

} catch (IOException e) {

e.printStackTrace();

}

// unchecked exception

int[] arr = {1, 2, 3};

System.out.println(arr[3]); // throws ArrayIndexOutOfBoundsException

}

}

1. What is the difference between a static and a non-static method?

A static method is a method that belongs to the class and not to any instance of the class. It can be called using the class name, without the need to create an object of the class. A non-static method, on the other hand, belongs to each instance of the class and can only be called using an object of the class.

Example:

public class MathUtils {

public static int add(int a, int b) {

return a + b;

}

public int multiply(int a, int b) {

return a \* b;

}

}

public class Main {

public static void main(String[] args) {

int sum = MathUtils.add(2, 3);

System.out.println(sum); // 5

MathUtils math = new MathUtils();

int product = math.multiply(2, 3);

System.out.println(product); // 6

}

}

1. What is the purpose of the final keyword?

The final keyword can be used to make a class, method, or variable immutable, i.e., not subject to change. A final class cannot be subclassed, a final method cannot be overridden, and a final variable cannot be reassigned.

Example:

public final class MathUtils {

public static final double PI = 3.14;

public final int add(final int a, final int b) {

final int sum = a + b;

return sum;

}

}

public class Main {

public static void main(String[] args) {

MathUtils math = new MathUtils();

int sum = math.add(2, 3);

System.out.println(sum); // 5

// math.PI = 3.14159; // error: cannot assign a value to final variable PI

}

}

1. What is a lambda expression?

A lambda expression is a concise way to represent a functional interface, i.e., an interface with only one abstract method. It allows you to pass a behavior (i.e., a method implementation) as a parameter to another method.

Example:

public interface MathOperation {

int operate(int a, int b);

}

public class Main {

public static void main(String[] args) {

MathOperation add = (a, b) -> a + b;

MathOperation subtract = (a, b) -> a - b;

int sum = operate(2, 3, add);

int difference = operate(2, 3, subtract);

System.out.println(sum); // 5

System.out.println(difference); // -1

}

public static int operate(int a, int b, MathOperation operation) {

return operation.operate(a, b);

}

}

1. What is a functional interface?

A functional interface is an interface with only one abstract method. It is used to represent a behavior (i.e., a method implementation) that can be passed as a parameter to another method, such as a lambda expression.

Example:

@FunctionalInterface

public interface MathOperation {

int operate(int a, int b);

}

public class Main {

public static void main(String[] args) {

MathOperation add = (a, b) -> a + b;

MathOperation subtract = (a, b) -> a - b;

int sum = operate(2, 3, add);

int difference = operate(2, 3, subtract);

System.out.println(sum); // 5

System.out.println(difference); //

**Java/Springboot tech check questions …**

Mostly clients ask to describe your work background and experience and as you describe projects they would follow up with a targeted question such as these below. These questions were compiled across multiple interviews so that candidates are assessed based on their breadth and depth of knowledge/experience.

1. **Describe the most latest project,multi layer app - front-end, middle, ware and backend.**
2. **What are principles of OOP**

Abstraction, encapsulation, inheritance, and polymorphism are four of the main principles of object-oriented programming

1. **What is run time polymorphism**

Whenever an object is bound with the functionality at run time, this is known as runtime polymorphism. The runtime polymorphism can be achieved by method overriding. Java virtual machine determines the proper method to call at the runtime, not at the compile time. It is also called dynamic or late binding.

*class SuperClass{void method(“This is actual method.”);}*

*class SubClass extends SuperClass{void method(sop(“This is derived method ”));}*

*SuperClass sup=new SubClass();*

*sup.method();// subclass method is invoked at* ***runtime***

1. **What is compile time polymorphism**

Whenever an object is bound with its functionality at the compile time, this is known as the compile-time polymorphism. At compile-time, java knows which method to call by checking the method signatures.

*add(int x,int y){...}*

*add(double x,double y){...}*

1. **Why you need Immutability and how you implement? What is the difference between a String and stringbuffer?**

Immutable objects are those objects whose state cannot be changed once created.

* Make your class final
* Declare all instance variable with private and final
* Say no to setter methods
* Initialize all variables in constructor
* Perform cloning of mutable objects while returning from getter method

Ex: String, Integer, Long, Double, Character, Boolean etc and much more. Date is not an immutable class.

*java.lang.String // it is immutable, The String objects are cached in the String pool, and it makes the String immutable*

*java.lang.StringBuffer // it is mutable and thread safe*

*java.lang.StringBuilder//it is mutable but not thread safe*

1. **When would you use abstract class vs a functional interface (SAM)**

**Abstract**: An abstract class is mostly used to provide a base for subclasses to extend and implement the abstract methods and override or use the implemented methods in abstract class

**FunctionalInterface**:

* Functional interfaces are included in Java SE 8 with Lambda expressions and Method references in order to make code more readable, clean, and straightforward. Functional interfaces are interfaces that ensure that they include precisely only Single abstract method (SAM).
* We can pass function as parameter

*Runnable, ActionListener and Comparable* are some of the examples of functional interfaces.

Thread **implements** Runnable

@FunctionalInterface

**public** **interface** Runnable { **public** **abstract** **void** run();}

Other Functional interfaces

void Consumer.accept(<T>)

boolean Predicate.test(<T>)

<E> BiFunction.apply(<T>,<T>,<E>)

<E> Supplier.get()

1. **Difference between abstract class and interface**

| **Abstract class** | **Interface** |
| --- | --- |
| 1) Abstract class can have **abstract and non-abstract methods.** | Interface can have only **abstract methods**. Since Java 8, it can have **default and static** methods also. |
| 2) Abstract class doesn't support multiple inheritance. | Interface supports multiple inheritance. |
| 3) Abstract class can have final, non-final, static and non-static variables. | Interface has only static and final variables. |
| 4) Abstract class can provide the implementation of interface. | Interface can't provide the implementation of abstract class. |
| 5) The abstract keyword is used to declare **abstract** class. | The **interface** keyword is used to declare interface. |
| 6) An abstract class can extend another Java class and implement multiple Java interfaces. | An interface can extend another Java interface only. |
| 7) An abstract class can be extended using keyword "**extends**". | An interface can be implemented using keyword "**implements**". |
| 8) A Java abstract class can have class members like **private**, **protected**, etc. | Members of a Java interface are **public** by default. |

1. **What is the time complexity of Arrays.sort?**

*Arrays.sort(Object[])*is based on the TimSort algorithm, giving us a average time complexity of***O(n log(n))*.** In short, TimSort makes use of the [Insertion sort](https://www.baeldung.com/java-insertion-sort) and the [MergeSort](https://www.baeldung.com/java-merge-sort) algorithms. Arrays.sort(Object[]) is based on the TimSort algorithm, giving us a time complexity of O(n log(n)). In short, TimSort makes use of the Insertion sort and the MergeSort algorithms. Though, in the worst case, its time complexity is *O(n2)*.

|  | **Best** | **Average** | **Worst** |
| --- | --- | --- | --- |
| **Merge Sort** | n(log(n)) | n(log(n)) | n(log(n)) |
| **Insertion sort** | Log(n) | Log(n) | N^2 |
| **Tim Sort** | N | n(log(n)) | n(log(n)) |

**Log(n)>n>n(log(n))>n^2**

1. **What is the difference between list and set? How do you remove duplicates in a list? How do you convert a list into a set?**
   1. Lists maintain the **order** of elements, while sets do not.
   2. Lists can contain **duplicates**, while sets cannot.
   3. Lists allow you to access elements by their **index**(get()), while sets do not.
   4. In general, **adding and removing** elements from a set is **faster** but **accessing** elements in a set is **slower** and vice versa

Remove duplicates from Java 8

List<String> myList = new ArrayList<>();

myList = myList.stream().distinct().collect(Collectors.toList());

**(or)**

You can convert the list into a Set, which automatically removes duplicates, and then convert it back to a list.

Set<String> mySet = new LinkedHashSet<>(myList);

1. **Difference between Hashmap and TreeMap**

In summary, if you need to store key-value pairs and don't care about the order of keys, and null keys are allowed, then ***HashMap*** is a good choice. On the other hand, if you need to maintain a sorted order of keys or need to iterate in sorted order, then ***TreeMap*** is a better choice, but it comes at a performance cost compared to ***HashMap***.

1. **What is the difference between Hashmap and concurrent Hashmap**

* **Thread Safety**: *HashMap* is not thread-safe and *ConcurrentHashMap* is designed to be thread-safe.
* **Locking Mechanism**: ConcurrentHashMap uses a locking mechanism known as "lock stripping" to ensure thread safety. This mechanism divides the map into multiple segments and allows multiple threads to update different segments concurrently, which reduces the locking contention and improves performance. `HashMap

1. **Private subnet but available public ?NAT gateways**

If you have resources in a private subnet that need to access the Internet, you can use a Network Address Translation (NAT) Gateway to enable outbound Internet connectivity. A NAT Gateway is a managed service provided by AWS that allows instances in a private subnet to communicate with the Internet or other AWS services, while hiding their private IP addresses.

1. **Security groups in AWS?**

In Amazon Web Services (AWS), a Security Group is a virtual firewall that controls the inbound and outbound traffic for an instance or a group of instances.

Security Groups act as a first line of defense for your resources by allowing you to specify which traffic is allowed to access your instances. You can create security groups for specific instances, and each security group can have rules that allow or deny traffic based on protocol, port number, and source or destination IP addresses.

1. **Have you configured swagger?**

Yes,

Swagger is a tool for documenting and testing RESTful APIs. It provides a user interface that allows developers to interact with the API and test its endpoints. Here are the steps to configure Swagger in a Java Spring Boot application:

1. Add Swagger Dependencies: Add the Swagger dependencies to your Spring Boot project by including the following in your **pom.xml** file:

Configure Swagger: Create a Swagger configuration class that defines the Swagger Docket, which is the primary interface for building and configuring Swagger. Here's an example configuration class:

@SpringBootApplication

@EnableSwagger2

**public** **class** SpringBootTutorialApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringBootTutorialApplication.**class**, args);

}

@Bean

**public** Docket productApi() {

**return** **new** Docket(DocumentationType.***SWAGGER\_2***).select()

.apis(RequestHandlerSelectors.*basePackage*("com.mk ")).build();

}

}

1. Test the API: Once the Swagger dependencies are added and the configuration is set up, you can test the API by navigating to **http://localhost:8080/swagger-ui.html** in a web browser. This will display the Swagger UI, which shows the available endpoints and allows you to interact with them.
2. **What is the difference between Spring and Spring Boot?**

Spring and Spring Boot are both frameworks for building Java applications, but there are some key differences between the two.

1. Configuration: In Spring, configuring an application involves a lot of manual setup, including defining beans, setting up data sources, and configuring logging. Spring Boot, on the other hand, provides a lot of these configurations out of the box, making it much easier to get up and running quickly.
2. Dependency Management: Spring Boot uses a dependency management system that automatically manages dependencies and resolves conflicts, which can be time-consuming in Spring. In Spring, you have to manually manage dependencies and resolve conflicts.
3. Auto-configuration: Spring Boot uses auto-configuration to automatically configure an application based on the dependencies that are included. This eliminates the need for a lot of manual configuration in Spring.
4. Embedded Server: Spring Boot comes with an embedded server (Tomcat, Jetty, or Undertow), which makes it easy to deploy applications without having to install a separate server. Spring does not come with an embedded server, so you have to install and configure a server separately.
5. Starter Dependencies: Spring Boot provides a wide range of starter dependencies that include everything you need to get started with a specific technology (e.g., Spring Data, Spring MVC, etc.). In Spring, you have to manually configure all of the dependencies.
6. **Name few Spring Annotations**

Here are some commonly used Spring annotations:

1. **@Component**: Indicates that a class is a Spring-managed component, which can be automatically detected and instantiated by Spring.
2. **@Autowired**: Injects a bean dependency automatically into a class constructor, field, or setter method.
3. **@Controller**: Indicates that a class is a Spring MVC controller, which is used to handle HTTP requests.
4. **@RestController**: A specialized version of **@Controller** that combines **@Controller** and **@ResponseBody**. This is used to create RESTful web services that return data in JSON or XML format.
5. **@RequestMapping**: Maps HTTP requests to a specific method in a controller.
6. **@Service**: Indicates that a class is a Spring-managed service, which typically contains business logic.
7. **@Repository**: Indicates that a class is a Spring-managed repository, which is used to perform database operations.
8. **@Transactional**: Defines a transaction boundary for a method, which ensures that the method is executed atomically.
9. **@Value**: Injects a value into a Spring-managed bean, typically from an external configuration file.
10. **@PostConstruct**: Indicates that a method should be executed after a bean has been initialized by Spring.
11. **Difference between @service and @Repository**

**@Service**: Indicates that a class is a Spring-managed service, which typically contains business logic.

**@Repository**: Indicates that a class is a Spring-managed repository, which is used to perform database operations.

1. **How and where did you use dependency injection ? Give couple of examples..**

By using annotations such as **@Autowired** or **@Inject**, Spring can automatically inject the necessary dependencies into a component at runtime.

There are several injections like Constructor injection, Setter injection

Dependency injection is commonly used in unit testing to mock or stub out dependencies that would otherwise be difficult to test

1. **what does an @springBoot Application mean? and @SpringBootApplicationConfirator equals to?**

In other words, when a Java class is annotated with @SpringBootApplication, it indicates that the class is the main entry point for a Spring Boot application. This annotation combines three other annotations: @Configuration, @EnableAutoConfiguration, and @ComponentScan.

* **@Configuration** indicates that the class provides Spring configuration.
* **@EnableAutoConfiguration** enables automatic configuration of the Spring application based on the classpath.
* **@ComponentScan** tells Spring to scan the current package and its sub-packages for components that can be injected.

1. **How are properties maintained for each env in Springboot and how they are brought in context in each env.**

By default, Spring Boot uses the application.properties or application.yml file to store configuration properties.

To specify which profile to activate, we can use the spring.profiles.active property. This property can be set in several ways, including:

1. Setting the **SPRING\_PROFILES\_ACTIVE** environment variable.
2. Adding the **--spring.profiles.active=dev** command line argument when running the application.
3. Adding **spring.profiles.active=dev** to the **application.properties** or **application.yml** file.
4. **What are spring Streams? How have you used it?**

Spring Streams is based on the concept of "continuous processing", where data is processed in real-time as it is received. This allows for faster, more efficient processing of large volumes of data.

@EnableBinding(MyChannels.class)

public class MyStreamProcessor {

@StreamListener(MyChannels.INPUT)

public void processMessage(String message) {

// Process the message

System.out.println("Received message: " + message);

}

}

1. **What is executors? where have you used in your application?**

**Yes,**

The **Executor** interface defines a single method, **execute(Runnable command)**, which takes a **Runnable** object as input and submits it for execution in a background thread. The actual thread creation and management is abstracted away by the **Executor** implementation, which allows for more efficient thread utilization and better control over thread creation.

import java.util.concurrent.Executor;

import java.util.concurrent.Executors;

public class ExecutorExample {

public static void main(String[] args) {

// Create a new executor with a fixed pool of 5 threads

Executor executor = Executors.newFixedThreadPool(5);

// Submit 10 tasks for execution

for (int i = 1; i <= 10; i++) {

executor.execute(new Task(i));

}

}

private static class Task implements Runnable {

private int taskId;

public Task(int taskId) {

this.taskId = taskId;

}

public void run() {

// Simulate some work

System.out.println("Task " + taskId + " is starting...");

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

// Handle exception

}

System.out.println("Task " + taskId + " is complete.");

}

}

}

1. **How do you print three print statements in main method? t1.join().t2.join().t3**

By calling **join()** on each thread in turn, we ensure that each thread completes its execution before the next one starts. This can be useful in cases where we need to coordinate the execution of multiple threads in a specific order. However, it's worth noting that this approach can lead to a potential deadlock if the threads are not designed to release the resources they hold. So it's important to use it with caution and ensure that the threads are properly synchronized.

**public** **class** ThreadJoinDemo {

**public** **static** **void** main(String[] args) **throws** InterruptedException {

Thread t1 = **new** Thread(() -> {

System.***out***.println("Thread t1 started");

// some code

System.***out***.println("Thread t1 ended");

});

Thread t2 = **new** Thread(() -> {

System.***out***.println("Thread t2 started");

// some code

System.***out***.println("Thread t2 ended");

});

Thread t3 = **new** Thread(() -> {

System.***out***.println("Thread t3 started");

// some code

System.***out***.println("Thread t3 ended");

});

t1.start();

t2.start();

t3.start();

t1.join();

t2.join();

t3.join();

System.***out***.println("All threads completed");

}

}

1. **Would you choose property files vs YML in spring boot application?**

Spring Boot supports both property files and YAML (YAML Ain't Markup Language) files for configuring the application.

Here are some differences between the two:

1. Syntax: Property files use a key-value pair syntax, where the key is separated from the value by an equals sign, while YAML uses a hierarchical structure with indentation.
2. Readability: YAML files are generally more readable than property files, especially for complex configurations with nested properties.
3. Flexibility: YAML files provide more flexibility in terms of how properties are structured, and allow for comments and multiple document blocks. Property files, on the other hand, are more limited in their structure and functionality.
4. Type safety: YAML files are more type-safe than property files, as they allow for more explicit type declarations and data structures.
5. Integration: YAML files are commonly used in other technologies, such as Docker Compose and Kubernetes, so it may be easier to integrate your Spring Boot application with these technologies if you use YAML.

In general, the choice between using property files or YAML for configuring your Spring Boot application depends on your specific use case and personal preference. If you have a relatively simple configuration with mostly flat properties, property files may be sufficient. If you have a more complex configuration with nested properties, YAML may be a better choice for readability and maintainability.

1. **What are view resolvers in spring MVC? can you name one of them?**

View resolvers are responsible for resolving logical view names returned by a controller into an actual view object that can be rendered by the web browser. In other words, view resolvers help map the view name to the actual view file or template that will be rendered.

@Configuration

@EnableWebMvc

public class AppConfig implements WebMvcConfigurer {

@Override

public void configureViewResolvers(ViewResolverRegistry registry) {

InternalResourceViewResolver resolver = new InternalResourceViewResolver();

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

resolver.setSuffix(".jsp");

registry.viewResolver(resolver);

} }

In this example, the InternalResourceViewResolver is configured to resolve view names to JSP files located in the /WEB-INF/views/ directory, with a .jsp file extension. This means that if a controller returns the logical view name "home", the InternalResourceViewResolver will look for a file named home.jsp in the specified directory.

1. **How to convert JSON to XML and vice versa? JAXB**

JAXB (Java Architecture for XML Binding) is a Java technology that allows you to convert Java objects to XML and vice versa. It provides a simple way to map Java objects to XML elements and attributes, and vice versa.

import javax.xml.bind.\*;

import javax.xml.transform.stream.\*;

import org.json.\*;

public class JsonToXmlConverter {

public static void main(String[] args) throws JSONException, JAXBException {

// JSON string to be converted to XML

String jsonString = "{\"employee\":{\"id\":\"101\",\"name\":\"John\",\"age\":\"35\"}}";

// Parse JSON string to create a JSONObject

JSONObject jsonObject = new JSONObject(jsonString);

// Create JAXBContext and Marshaller

JAXBContext jaxbContext = JAXBContext.newInstance(Employee.class);

Marshaller marshaller = jaxbContext.createMarshaller();

// Convert JSONObject to Employee object

Employee employee = new Employee(jsonObject.getJSONObject("employee").getString("id"),

jsonObject.getJSONObject("employee").getString("name"),

jsonObject.getJSONObject("employee").getString("age"));

// Convert Employee object to XML

StringWriter stringWriter = new StringWriter();

marshaller.marshal(employee, new StreamResult(stringWriter));

String xmlString = stringWriter.toString();

System.out.println("JSON to XML conversion:\n" + xmlString);

}

}

@XmlRootElement

class Employee {

String id;

String name;

String age;

public Employee() {}

public Employee(String id, String name, String age) {

this.id = id;

this.name = name;

this.age = age;

}

@XmlElement

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

@XmlElement

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

@XmlElement

public String getAge() {

return age;

}

public void setAge(String age) {

this.age = age;

}

}

1. **What is Spring Actuator?**

Spring Actuator is a sub-project of Spring Boot that provides a set of tools and endpoints for monitoring and managing the health of a Spring Boot application. It includes features like health checks, metrics gathering, and auditing of the application. Spring Actuator provides endpoints that can be accessed using HTTP requests to gather information about the application's state, including information about the system, environment, and resources. It also provides a way to expose custom endpoints to monitor application-specific metrics. By default, Spring Actuator endpoints are secured and require authentication, but this can be customized as needed.

1. **Have you been involved in spring upgrade**

Yes, Update the Spring dependencies in the **pom.xml** file or build file to the latest version of Spring or we can also use a dependency management tool like Maven or Gradle to manage the version of Spring across all modules.

1. **How do you use a different web server than tomcat in spring boot**

* Add the Jetty dependency to the **pom.xml** file or **build.gradle** file:
* Update the application properties in **application.properties** or **application.yml** to configure the Jetty server:

1. **What are generics**

Generics in Java allow you to define a class or a method that can work with any data type. Generics are a way to make code more type-safe and reusable, by enabling classes, interfaces, and methods to be parameterized by one or more types.

Here's an example of how generics can be used:

public class MyList<T> {

private T[] items;

private int size;

public MyList() {

items = (T[]) new Object[10];

size = 0;

}

public void add(T item) {

if (size == items.length) {

T[] newItems = (T[]) new Object[2 \* items.length];

System.arraycopy(items, 0, newItems, 0, items.length);

items = newItems;

}

items[size++] = item;

}

public T get(int index) {

if (index < 0 || index >= size) {

throw new IndexOutOfBoundsException();

}

return items[index];

}

}

In this example, the **MyList** class is parameterized by a type **T**. This means that instances of the **MyList** class can be created to hold elements of any type. The **add** method can take any type of element and add it to the list, and the **get** method can return an element of any type.

1. **What is the difference between synchronous and asynchronous transactions?**

Synchronous and asynchronous transactions are two different ways of handling transactions in a computer system.

In a synchronous transaction, the application waits for the transaction to complete before moving on to the next task. This means that the application is blocked until the transaction is finished, and cannot continue processing other tasks until the transaction is completed.

public class SynchronousTransactionExample {

public void transferMoney(Account from, Account to, double amount) {

synchronized (from) {

synchronized (to) {

if (from.getBalance() >= amount) {

from.withdraw(amount);

to.deposit(amount);

System.out.println("Transfer successful");

} else {

System.out.println("Transfer failed: insufficient funds");

}

}

}

}

}

In contrast, in an asynchronous transaction, the application initiates the transaction and then continues processing other tasks without waiting for the transaction to complete. When the transaction is finished, the application is notified and can then process the results of the transaction.

public class AsynchronousTransactionExample {

public void transferMoney(Account from, Account to, double amount) {

CompletableFuture.runAsync(() -> {

if (from.getBalance() >= amount) {

from.withdraw(amount);

to.deposit(amount);

System.out.println("Transfer successful");

} else {

System.out.println("Transfer failed: insufficient funds");

}

});

System.out.println("Transaction initiated");

}

}

1. **How do you run synchronous blocks**

Synchronous blocks are executed in a sequential manner, meaning that one statement is executed at a time and the execution of the next statement waits for the completion of the previous statement. Here's an example of a synchronous block in Java:

public void doSynchronous() {

System.out.println("Starting synchronous block");

System.out.println("Executing statement 1");

System.out.println("Executing statement 2");

System.out.println("Executing statement 3");

System.out.println("Synchronous block completed");

}

1. **How do you implement a Singleton design pattern**

Singleton design pattern is a creational pattern that ensures that a class has only one instance and provides a global point of access to that instance. Here's an example of how to implement Singleton design pattern in Java:

public class Singleton {

private static Singleton instance;

private Singleton() {}

public static synchronized Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

In this example, the Singleton class has a private constructor to prevent the creation of multiple instances of the class. The class also has a static method called getInstance() that returns the single instance of the class. This method checks if an instance of the class already exists and creates a new instance if it doesn't.

The getInstance() method is declared as synchronized to ensure that only one thread can access it at a time, preventing multiple instances from being created simultaneously in a multi-threaded environment.

To use this Singleton class, you can call the getInstance() method:

Singleton singletonInstance = Singleton.getInstance();

1. **How would you print all your beans**

In Spring, you can print all the beans available in the application context using the ApplicationContext object. Here's an example of how to print all beans in Java:

@Autowired

private ApplicationContext context;

String[] beanNames = context.getBeanDefinitionNames();

Arrays.sort(beanNames);

for (String beanName : beanNames) {

System.out.println(beanName);}

1. **How would you look at your dependency tree**

You can use the Maven Dependency Plugin to generate a dependency tree for your Spring application. Here's an example of how to do this:

* Open a terminal window and navigate to your Spring project directory.
* Run the following Maven command to generate the dependency tree:

mvn dependency:tree

1. **What are design patterns you have used**

Spring framework follows several design patterns to implement its core functionality. Some of the common design patterns used in Spring applications are:

1. Dependency Injection (DI) pattern // Dependency Injection is the most widely used design pattern in Spring, which allows for loosely coupled components and enhances testability and maintainability of the code.
2. Singleton pattern// Singleton pattern is used to ensure that a class has only one instance.
3. Factory pattern// Factory pattern is used to create objects without specifying the exact class of object that will be created.
4. Proxy pattern// The Proxy pattern is used to create a wrapper around an object to add additional functionality
5. Template method pattern // Template method pattern is used to define a skeleton of an algorithm in a superclass, with specific steps deferred to subclasses.
6. Observer pattern //. The Observer pattern is used to notify interested objects of changes to a subject
7. Decorator pattern// Decorator pattern is used to add new behaviors or responsibilities to an object dynamically.
8. Adapter pattern //The Adapter pattern is used to make two incompatible interfaces work together by creating a middle layer.
9. **Have you built a REST API? What is the PATH variable in REST API? How do you define PATH variables in Spring? What is the difference between PATH variable and Request Param? In Spring can we make request param optional?**

Yes,

The PATH variable in REST API refers to a part of the URL that identifies a specific resource in the API. In a typical REST API, the URL consists of two parts: the base URL and the resource path. The base URL is the root endpoint of the API, while the resource path is the part that specifies the resource being accessed.

For example, consider a REST API for managing products. The base URL could be "<https://example.com/api>", while the resource path could be "/products". To retrieve a specific product, the path variable could be used to identify the product ID, like "/products/123".

In Spring, the PATH variables can be defined using the **@PathVariable** annotation. This annotation is used to bind a method parameter to a URI template variable, which can be used to extract values from the URL path.

@RestController

@RequestMapping("/products")

public class ProductController {

@GetMapping("/{productId}")

public Product getProduct(@PathVariable("productId") Long productId) {

// logic to retrieve product with given ID

}

@PostMapping("/{categoryId}")

public void createProduct(@PathVariable("categoryId") Long categoryId, @RequestBody Product product) {

// logic to create product with given category ID

}

}

1. **What is dependency injection**

In Spring, dependency injection is used to inject the dependencies of a class at runtime. Instead of creating the objects itself, Spring creates and injects the objects that a class depends on, thereby promoting a loosely coupled architecture.

1. There are three ways in which Spring can perform dependency injection:
2. Constructor Injection: Dependencies are passed to a class through its constructor. This is considered the most efficient and preferred way to perform dependency injection.
3. Setter Injection: Dependencies are passed to a class using its setter methods.
4. Field Injection: Dependencies are injected directly into the fields of a class.

By using dependency injection, we can achieve a number of benefits, such as code reuse, easier testing, and improved maintainability. It also helps in achieving a better separation of concerns, making the code more modular and easier to understand.

1. **What is auto wired**

**@Autowired** is a Spring framework annotation that allows automatic wiring of dependencies into a class. It eliminates the need for manual wiring of dependencies and makes the code more readable and maintainable.

@Autowired

private MyRepository myRepository;

1. **What background do you have with databases**

We are using postgres , Mysql and Oracle.

1. **How did you perform Integration Testing, challenges and automation**

Integration testing is a type of testing where individual software modules are combined and tested as a group to ensure that the entire system works as expected. It is a crucial step in the software development process to ensure that different components of a system work well together.

There are several challenges in integration testing such as:

* Environment setup: Setting up a testing environment that accurately reflects the production environment can be challenging.
* Integration of different components: Integrating different components of the system and ensuring that they work together seamlessly can be difficult.
* Data consistency: Ensuring that data is consistent across the different components of the system can be challenging.
* Test data management: Managing test data and ensuring that it is consistent across different tests can be difficult.

Automation of integration testing can help overcome some of these challenges. Some of the ways to automate integration testing are:

* Using test automation frameworks: Test automation frameworks can help automate the process of running tests and analyzing the results.
* Continuous Integration (CI): CI tools like Jenkins, Travis CI, etc. can be used to automate the process of building and testing code.
* Mocking: Mocking can be used to simulate the behavior of components that are not available during testing.
* Docker containers: Docker containers can be used to create a consistent and reproducible testing environment.

In Spring, integration testing can be performed using the Spring Test framework. The Spring Test framework provides several annotations such as @RunWith, @SpringBootTest, @WebMvcTest, etc. which can be used to write integration tests for different components of a Spring application.

1. **What are the scopes of beans**
2. Singleton: The default scope in Spring, which means that only one instance of the bean is created and shared across the entire application context.
3. Prototype: A new instance of the bean is created every time it is requested from the container.
4. Request: A new instance of the bean is created for each HTTP request. This scope is only valid in the context of a web-aware Spring ApplicationContext.
5. Session: A new instance of the bean is created for each HTTP session. This scope is only valid in the context of a web-aware Spring ApplicationContext.
6. Global session: Similar to the session scope, but applies to global HTTP sessions. This scope is only valid in the context of a web-aware Spring ApplicationContext.
7. Custom: You can define custom scopes by implementing the Scope interface.
8. **When would you use Asynchronous**

Asynchronous programming can be useful in situations where you need to perform long-running or blocking operations, such as making network requests or performing disk I/O, without blocking the main thread or application. By using asynchronous programming, you can free up the main thread to continue processing other tasks, which can improve the overall responsiveness and performance of your application.

1. **When would you use abstract class, interface and why?**

Abstract class:

* An abstract class is a class that cannot be instantiated, but can be subclassed.
* An abstract class can contain both abstract and non-abstract methods.
* An abstract class can have instance variables, constructors, and static methods.
* An abstract class can implement an interface.
* An abstract class can be used when you want to provide a partial implementation of a class and leave some methods to be implemented by subclasses.

Interface:

* An interface is a contract that defines a set of methods that a class must implement.
* An interface cannot have any implementation.
* All the methods in an interface are abstract by default.
* An interface can only have static final variables, not instance variables.
* An interface can extend multiple interfaces but cannot extend a class.
* An interface can be used when you want to define a contract that a class must implement, regardless of its implementation details.

1. **What is DDD? What is the purpose?**

DDD stands for Domain-Driven Design, which is an approach to software development that emphasizes the importance of understanding the domain or business problem at hand and modeling it in the software design. The purpose of DDD is to help developers create software that closely models the business problem and is more maintainable, extensible, and scalable.

DDD uses a set of practices and patterns to help developers achieve this goal. Some of these practices include:

1. Ubiquitous Language: using a common language between the domain experts and developers to ensure a shared understanding of the problem domain.
2. Bounded Contexts: defining boundaries around parts of the domain to ensure that the domain model is clear and unambiguous.
3. Aggregates: defining groups of related entities and value objects that are treated as a single unit of work.
4. Domain Events: capturing important domain events that can be used to trigger other actions or updates.
5. **How Kafka consumers and offset are maintained?**
6. **Have you used Cucumber or any testing automation tools**
7. **Difference between Bean and Component in Spring**

In Spring Framework, both **@Bean** and **@Component** annotations are used for defining beans. The main difference between them is that **@Component** is a more generic annotation that is used to mark any class as a Spring-managed component, while **@Bean** is a method-level annotation that is used to explicitly declare a bean instance.

Here are some key differences between **@Bean** and **@Component**:

* **@Bean** is used on methods and returns an object that will be managed by the Spring container, whereas **@Component** is used on classes to mark them as Spring-managed components.
* **@Bean** is more powerful and provides more control over the configuration of the bean, as it allows you to specify the bean's name, scope, and lifecycle callbacks. **@Component**, on the other hand, does not offer this level of control and is mainly used for generic components.
* **@Bean** can be used in combination with other annotations such as **@Configuration**, **@Scope**, and **@Lazy**, whereas **@Component** cannot.
* **@Bean** methods can be defined in a configuration class that is separate from the bean's implementation class, while **@Component** is always tied to a specific class.

In general, use **@Bean** when you need more control over the configuration of the bean, and use **@Component** for general-purpose components that do not require explicit configuration.

1. **How do you handle exceptions in SpringBoot**

In Spring Boot, exceptions can be handled using several approaches. Here are some common ways to handle exceptions in Spring Boot:

* Using @ExceptionHandler annotation: This annotation is used to define a method to handle exceptions thrown from a controller method. The @ExceptionHandler method should be annotated with the exception class that it handles. This method can be defined in the same controller or in a separate controller advice.
* Using @ControllerAdvice annotation: This annotation is used to define global exception handling for all controllers. It allows you to define @ExceptionHandler methods for multiple controllers in a single class.
* Using @ResponseStatus annotation: This annotation is used to define the status code to be returned for a specific exception. You can annotate an exception class with @ResponseStatus to define the HTTP status code to be returned when that exception is thrown.
* Using ResponseEntity: This class allows you to customize the response returned for an exception. You can create an instance of ResponseEntity with the appropriate HTTP status code, headers, and body to be returned.

@ExceptionHandler(UserNotFoundException.class)

@ResponseStatus(HttpStatus.NOT\_FOUND)

public String handleUserNotFoundException(UserNotFoundException ex) {

return ex.getMessage();

}

1. **What is the difference between GET & POST? What are the limitations of Request set? Any on the size of the request body?**

GET requests are used to retrieve data from the server. They are idempotent, meaning that multiple identical GET requests should have the same effect as a single request. GET requests can have parameters in the URL query string, but they have limitations on the size of the request body, typically limited to a few kilobytes.

POST requests are used to submit data to the server. They are not idempotent, meaning that multiple identical POST requests can have different effects. POST requests can have a larger request body than GET requests, making them more suitable for sending large amounts of data, such as file uploads.

The limitations on request size can vary depending on the server and configuration, but they are typically in the range of a few megabytes. However, sending large amounts of data in a single request can impact the performance and reliability of the application, so it is recommended to split the data into smaller chunks or use other techniques such as compression.

Most common limitations of the HTTP request set are:

* Limited request size: Most web servers and web frameworks have a limit on the size of a request that they can handle. This is done to prevent malicious users from overloading the server or taking up too much bandwidth. The limit can vary depending on the server or framework being used.
* Limited request types: HTTP request methods are limited to GET, POST, PUT, DELETE, HEAD, OPTIONS, and TRACE. This means that more complex actions, such as file uploads or database queries, may require a more customized approach.
* Limited security: HTTP is a stateless protocol, meaning that each request is treated as a completely separate transaction. This can make it difficult to ensure that users are who they say they are and that the data being transmitted is secure.
* Limited caching: HTTP requests are often cached by web browsers or other intermediate devices, which can cause problems if the content being requested has changed since it was last cached.
* Limited flexibility: While HTTP is a very flexible protocol, there are still some limitations on what can be accomplished using the basic request set. For more complex tasks, such as real-time data processing or streaming, other protocols may be more appropriate.

1. **How would you handle a DELETE API call**

In Spring Boot, handling a DELETE API call can be done by creating a method that maps to the **DELETE** HTTP request method using the **@DeleteMapping** annotation.

@DeleteMapping("/users/{id}")

public ResponseEntity<Void> deleteUser(@PathVariable Long id) {

// delete the user with the given id

// return appropriate response status

}

1. **What are 40? Series of HTTP errors?**

Client Errors eg.,

|  |  |
| --- | --- |
| 400 | Bad Request |
| 401 | Unauthorized |
| 402 | Payment Required |
| 403 | Forbidden |
| 404 | Not Found |
| 405 | Method Not Allowe |

1. **what is the difference between 401 and 403?**

In summary, a 401 Unauthorized response should be used for missing or bad authentication, and a 403 Forbidden response should be used afterwards, when the user is authenticated but isn’t authorized to perform the requested operation on the given resource.

1. **When application errors arise what is the HTTP error thrown by the API, and how errors are mapped to HTTP errors in SpringBoot**

When application errors arise, the API can return various HTTP errors depending on the type of error. Generally, the HTTP status code 5xx is used for server-side errors, such as a database connection error or an internal server error, while the HTTP status code 4xx is used for client-side errors, such as invalid input or unauthorized access.

In Spring Boot, we can map application exceptions to specific HTTP error codes by using the **@ExceptionHandler** annotation.

@ExceptionHandler(UserNotFoundException.class)

@ResponseStatus(HttpStatus.NOT\_FOUND)

public String handleUserNotFoundException(UserNotFoundException ex) {

return ex.getMessage();}

1. **How is Strategy pattern implemented**

The Strategy pattern is a design pattern that allows you to encapsulate a family of related algorithms and make them interchangeable at runtime. This pattern helps you to select a specific algorithm at runtime from a set of algorithms to perform a specific task. The implementation of this pattern generally involves the following steps:

// Step 1: Create an interface for the strategy

public interface SortingStrategy {

public void sort(int[] numbers);

}

// Step 2: Create concrete classes implementing the strategy interface

public class BubbleSort implements SortingStrategy {

public void sort(int[] numbers) {

// Sorting algorithm implementation using bubble sort

}

}

public class QuickSort implements SortingStrategy {

public void sort(int[] numbers) {

// Sorting algorithm implementation using quick sort

}

}

// Step 3: Define a context class that will use the strategy

public class SortContext {

private SortingStrategy strategy;

public void setStrategy(SortingStrategy strategy) {

this.strategy = strategy;

}

public void sortNumbers(int[] numbers) {

strategy.sort(numbers);

}

}

// Step 4: Create setter methods in the context class to allow clients to set the strategy at runtime

public class Client {

public static void main(String[] args) {

SortContext context = new SortContext();

context.setStrategy(new BubbleSort());

context.sortNumbers(new int[]{5, 2, 9, 3, 6});

context.setStrategy(new QuickSort());

context.sortNumbers(new int[]{5, 2, 9, 3, 6});

} }

1. **How CompletableFuture keeps the context for each invocation?(>Java8)**

CompletableFuture is a class in Java that provides a way to perform asynchronous programming using the Future interface. It allows you to chain together a series of asynchronous tasks and specify what should happen when they complete.

1. **How do you vary the number of threads to be used in CompletableFuture**

CompletableFuture.supplyAsync(() -> {

// async operation

}, Executors.newFixedThreadPool(10));

1. **What java8 features have you used**

Java 8 introduced several new features and improvements over its predecessor, Java 7. Some of the key features of Java 8 are:

// 1. Lambda expressions: Lambda expressions provide a concise way to define anonymous functions. Here's an example of how you can use a lambda expression to sort a list of strings:

System.out.println("1. Lambda expressions:");

List<String> strings = Arrays.asList("foo", "bar", "baz");

Collections.sort(strings, (s1, s2) -> s1.compareTo(s2));

System.out.println(strings);

//2.Stream API: The Stream API allows you to perform operations on collections in a more functional and declarative way. Here's an example of how you can use the filter and map operations to transform a list of strings:

System.out.println("2.Stream API:");

List<String> strings2 = Arrays.asList("foo", "bar", "ba77777z","1234");

List<Integer> lengths = strings2.stream()

.filter(s -> s.length() > 3)

.map(String::length)

.collect(Collectors.toList());

System.out.println(lengths);

//3. Date and Time API: The Date and Time API provides a better way to work with dates, times, and time zones. Here's an example of how you can use the new API to parse and format dates:

System.out.println("3. Date and Time API: ");

LocalDate date = LocalDate.parse("2022-03-07");

DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy");

String formattedDate = date.format(formatter);

System.out.println(formattedDate);

//4.Default methods:

System.out.println("4.Default methods:");

new Java8Features().bar();

//5.Functional interfaces:

System.out.println("5.Functional interfaces: ");

MyFunction<Integer, String> function = (i) -> String.valueOf(i);

String result = function.apply(123);

System.out.println(result);

//6.Method references: Method references provide a way to refer to a method without invoking it. Here's an example of how you can use a method reference to sort a list of strings:

System.out.println("6.Method references");

List<String> strings6 = Arrays.asList("foo", "bar", "baz");

Collections.sort(strings6, String::compareTo);

System.out.println(strings6);

// 7.Optional:Optional is a new class that provides a way to handle null values in a more functional way. Here's an example of how you can use Optional to handle null values:

System.out.println("7. Optional");

Optional<String> optionalString = Optional.ofNullable(null);

System.out.println(optionalString);

List<String> listString = null;

System.out.println(listString);

8. Nashorn JavaScript engine: Java 8 includes a new JavaScript engine called Nashorn, which is faster and more efficient than the previous Rhino engine.

9. Parallel array sorting: Java 8 introduced a new way to sort arrays in parallel, which can significantly improve the performance of sorting large arrays.

10. Compact Profiles: Java 8 introduced compact profiles, which allow you to create smaller, more focused versions of the Java Runtime Environment (JRE) for use in embedded systems and other resource-constrained environments.

1. **What is map and flatMap**

**map** is used to apply a function to each element in a collection and return a new collection with the transformed values.

List<Integer> numbers = new ArrayList<>();

numbers.add(3); numbers.add(4); numbers.add(5); numbers.add(6); numbers.add(7);

List<Double> doubledNumbers = numbers.stream().map(x -> x \* 2.0).collect(Collectors.toList());

List<Integer> numbers1 = numbers.stream().map(x -> x \* 2).collect(Collectors.toList());

List<Integer> numbers2 = numbers1.stream().map(x -> x \* 2).collect(Collectors.toList());

List<Integer> numbers3 = numbers2.stream().map(x -> x \* 2).collect(Collectors.toList());

**flatMap** : is similar to map, but it's used when the transformation function returns a collection itself. In this case, flatMap will "**flatten**" the resulting **collections into a single** collection.

List<List<Integer>> numbersAll = new ArrayList<List<Integer>>();

numbersAll.add(numbers); numbersAll.add(numbers1);numbersAll.add(numbers2); numbersAll.add(numbers3);

System.out.println(numbersAll);

List<Integer> numbersAll2=numbersAll.stream().flatMap(v->v.stream()).collect(Collectors.toList());

System.out.println(numbersAll2);

1. **What design patterns**

Spring framework follows several design patterns to implement its core functionality. Some of the common design patterns used in Spring applications are:

1. Dependency Injection (DI) pattern // Dependency Injection is the most widely used design pattern in Spring, which allows for loosely coupled components and enhances testability and maintainability of the code.
2. Singleton pattern// Singleton pattern is used to ensure that a class has only one instance.
3. Factory pattern// Factory pattern is used to create objects without specifying the exact class of object that will be created.
4. Proxy pattern// The Proxy pattern is used to create a wrapper around an object to add additional functionality
5. Template method pattern // Template method pattern is used to define a skeleton of an algorithm in a superclass, with specific steps deferred to subclasses.
6. Observer pattern //. The Observer pattern is used to notify interested objects of changes to a subject
7. Decorator pattern// Decorator pattern is used to add new behaviors or responsibilities to an object dynamically.
8. Adapter pattern //The Adapter pattern is used to make two incompatible interfaces work together by creating a middle layer.
9. **How to secure APIs**

* Use Spring Security: Spring Security is a powerful framework that provides comprehensive support for authentication and authorization.
* Use HTTPS: Secure your APIs by using HTTPS instead of HTTP to encrypt communication between the client and the server.
* Implement Authorization: Define roles and permissions that determine which users can access which endpoints or resources.
* Validate Input: Validate input to prevent attacks like SQL injection, Cross-Site Scripting (XSS), and Cross-Site Request Forgery (CSRF).
* Use JWT Tokens: Use JSON Web Tokens (JWT) to store user authentication and authorization data instead of traditional session-based authentication.

1. **Aspect Oriented programming – cross cutting**

In Spring Framework, AOP is implemented using the AspectJ library. To use AOP in Spring, you can define an aspect using the @Aspect annotation and define the pointcut expression using the @Pointcut annotation. You can also define advice, which is the code executed when the aspect is applied, using the @Before, @After, @Around, or @AfterReturning annotations.

@Aspect

@Component

**public** **class** LoggingAspect {

@Before("execution(\* demo.ShoppingCart.checkout(..))||execution(\* demo.ShoppingCart.checkout1(..))")

**public** **void** beforeLogger(JoinPoint jp) {

System.***out***.println(jp.getSignature());

// System.out.println(jp.getArgs()[0].toString());

System.***out***.println("Before logging");

}

@After("execution(\* demo.ShoppingCart.checkout(..))")

**public** **void** afterLogger() {

System.***out***.println("After logging");

}

}

1. **What is a micro service and what is the benefit of MS architecture? What are the different factors you would consider while designing Micro services?**

A microservice is a small, independent, and modular service that performs a single business capability. Each microservice is self-contained and can be deployed and scaled independently of other services. Microservices architecture (MSA) is a software architecture style that structures an application as a collection of small services that communicate with each other over a network.

The benefits of microservices architecture include:

1. Scalability: Each microservice can be scaled independently, enabling the application to handle high loads more efficiently.
2. Agility: Microservices enable faster development cycles and continuous delivery, as each service can be developed, tested, and deployed independently.
3. Resilience: Microservices are designed to handle failures gracefully, ensuring that a single failure does not affect the entire system.
4. Maintainability: Each microservice is self-contained and can be developed, tested, and maintained independently, making it easier to update or replace individual services without impacting the rest of the system.
5. Technology Diversity: Microservices architecture allows the use of different programming languages, frameworks, and technologies for each service.

When designing microservices, there are several factors to consider, including:

1. Business Capabilities: Each microservice should perform a single business capability, such as user authentication, payment processing, or inventory management.
2. Service Boundaries: Services should be defined around clear and concise business capabilities and should have well-defined boundaries that encapsulate their functionality.
3. Communication Protocols: Communication between services should be defined using standard protocols such as HTTP/REST, gRPC, or message queues.
4. Data Management: Each microservice should manage its data independently, and the data should be isolated from other services to ensure scalability, security, and resilience.
5. Deployment: Each microservice should be independently deployable, and deployment should be automated to ensure consistency and reduce the risk of human error.
6. Monitoring and Logging: Each microservice should be instrumented for monitoring and logging to enable real-time monitoring, troubleshooting, and performance optimization.

In summary, microservices architecture is a software architecture style that structures an application as a collection of small, independent, and modular services. The benefits of microservices architecture include scalability, agility, resilience, maintainability, and technology diversity. When designing microservices, it is essential to consider business capabilities, service boundaries, communication protocols, data management, deployment, and monitoring and logging.

1. **Take a functional flow from you last project and explain the application architecture.**
2. **What is optional? How to check object not null? How do you return a default value? Did you use orelse, orelseget, orelse throw?**

In Java, the Optional class is a container object that may or may not contain a non-null value. It is designed to prevent NullPointerExceptions and make the code more robust and expressive.

To check if an object is not null, you can use the Optional.ofNullable() method. This method returns an Optional object that contains the specified value if it is not null; otherwise, it returns an empty Optional object.

For example:

String name= **null**;

Optional<String> opionalName=Optional.*ofNullable*(name);

// isPresent()

System.***out***.println(opionalName.isPresent()?opionalName.get():"No name");

// orElse

System.***out***.println(Optional.*ofNullable*(name).orElse("No name"));

// orElseGet

System.***out***.println(Optional.*ofNullable*(name).orElseGet(()->{ **return** "No Name"; }));

// orElseThrow

System.***out***.println(Optional.*ofNullable*(name).orElseThrow(()->{

// return new NullPointerException("Name cannot be null");

**return** **new** IllegalArgumentException("Name cannot be null");

}));

1. **What is a volatile variable?**

In Java, the volatile keyword is used to declare a variable whose value may be modified by different threads. The volatile keyword ensures that any changes made to the variable by one thread are immediately visible to other threads.

When a variable is marked as volatile, its value is stored in the main memory instead of the thread's local memory. This means that any thread that reads the value of a volatile variable will always get the most up-to-date value, rather than a stale value from its local memory cache.

**class** MyThread **extends** Thread {

**private** **volatile** **boolean** keepRunning = **true**;

**public** **void** run() {

**while** (keepRunning) {

// do something

}

}

**public** **void** stopRunning() {

keepRunning = **false**;

}}

1. **Are you familiar with design patterns?  Can you describe one and how you would implement it?**

**Talk about Observer, Template method and Singleton patterns**

**Observer pattern:** The observer pattern is a design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any changes to its state. This pattern is used to establish a one-to-many relationship between objects, where any change to the subject's state triggers a notification to all its observers.

The observer pattern consists of two types of objects: the subject and the observer. The subject maintains a list of its observers and provides methods for adding and removing observers from the list. The observer provides an update() method that the subject calls whenever its state changes.

**Template method pattern:** The template method pattern is a design pattern that defines the skeleton of an algorithm in a superclass but lets subclasses override specific steps of the algorithm without changing its structure. This pattern is used to create a standard framework for a family of related algorithms, where the individual steps may vary but the overall structure remains the same.

The template method pattern consists of two types of methods: abstract methods and template methods. Abstract methods are declared in the superclass but implemented in the subclasses, while template methods are defined in the superclass and provide a framework for the algorithm. The template method calls the abstract methods to perform the specific steps of the algorithm, which are implemented differently in each subclass.

**Singleton pattern**: The singleton pattern is a design pattern that ensures that a class has only one instance, and provides a global point of access to that instance. This pattern is used to ensure that there is only one instance of a class, and to provide a single point of access to that instance for other objects.

The singleton pattern involves a single class that is responsible for creating its own instance, and a static method that provides access to that instance. The class's constructor is made private to prevent other objects from creating instances of the class. The static method checks whether an instance of the class has already been created, and returns that instance if it exists, or creates a new instance if it does not.

1. **How would you implement integration testing for external services? How do you protect a resource in an endpoint? For Integration testing, What areas do you mock vs real test? Did you use any tools?**
2. **In a web service, there is a call on to another web service, while persisting data how do you design transaction for “all”or “none”?**

To design a transaction that adheres to the "all-or-none" principle, you can use a two-phase commit (2PC) protocol. In this protocol, the transaction coordinator communicates with all the services involved in the transaction, asking them to prepare for the transaction. If all services are able to prepare successfully, the coordinator then sends a commit request to all the services, which completes the transaction. However, if any service reports that it cannot prepare for the transaction, the coordinator sends a rollback request to all services to undo the changes made so far and abort the transaction.

1. **How did you call/use stored procedures in springboot app? How did you write unit tests?**

@Repository

public interface MyRepository extends JpaRepository<MyEntity, Long> {

@Procedure(name = "my\_stored\_procedure\_name")

void executeMyStoredProcedure(@Param("param1") String param1, @Param("param2") String param2);ram("param1) String param1, @P

;}

@Service

public class MyService {

@Autowired

private MyRepository myRepository;

public void callMyStoredProcedure(String param1, String param2) {

myRepository.executeMyStoredProcedure(param1, param2);

}}

// Testing

@Transactional

Public class MyTest{

@Test

public void testExecuteMyStoredProcedure() {

myRepository.executeMyStoredProcedure(param1, param2);

}

}

the **@Transactional** annotation ensures that the test is run in a transaction, which is rolled back at the end of the test, ensuring that the database state is not affected by the test.

1. **What is the difference between run time and compile time exceptions? How do you handle them?**

Compile-time exceptions are errors that are detected by the compiler when the program is being compiled. These errors typically relate to syntax or type mismatches in the code. If a compile-time exception is not resolved, the program will not compile and will not run.

To handle compile-time exceptions, you need to fix the issues in the code that are causing the errors.

Runtime exceptions, on the other hand, occur when a program is running and are caused by unexpected conditions or errors in the program logic. These types of exceptions are not detected by the compiler and can only be discovered when the program is executed.

In general, runtime exceptions are handled using exception handling mechanisms built into the programming language.

1. **Are you familiar with tools like Datadog? Splunk? How did you use them?**

No

Datadog and Splunk are two popular tools for monitoring and analyzing data across various IT systems, applications, and infrastructure.

1. **Describe your cloud experience. How do you use lambda function in AWS for security validation?**
2. **Describe your frontend experience, if any.**

I designed forms for web application using HTML, ES6, CSS with JSP and JSF. And also I develop functional and class components in ReactJs and other npm libraries for CSS,formic, axios, primereact, react-redux, DB etc.

1. **Describe couple of situations how you identified and improved performance.**

**In Backend**:

* For complex problems I will write algorithms and analyze code for various use cases. Then convert into code.
* Avoid unnecessary database hits.
* Avoid unnecessary iterations of entities (db records)
* Avoid dead lock on thread safe objects.

**InFrontend**:

* Minimize the user inputs.
* Use listeners instead of form submissions.
* Data Lazy loading.

**DataBase**:

* Write stored procedures/functions for complex queries.
* Run db queries on indexes.

1. **How to increment a counter variable in a thread safe way? Multiple Threads and how do you handle not to cross**

AtomicInteger counter = **new** AtomicInteger();

System.***out***.println(counter.incrementAndGet());

1. **In junit, what is the difference between @mock and @injectinmock**

The **@Mock** annotation is used to create a mock object for a given class or interface. It is typically used in conjunction with the **Mockito.mock()** method to create a mock object instance, which can then be used to replace a real object in the test code.

@Mock

private MyService myService;

On the other hand, the **@InjectMocks** annotation is used to inject the mock objects created using **@Mock** annotations into the object being tested. It is used to create the object under test and inject the mock objects into it automatically. For example:@InjectMocks

private MyController myController;

1. **Code challenge -   Integer Array[4,6,3,10, 2,8] , implement a logic find two entries from which total equals to 12, can use Java8 streams also, it should run for 13, etc**

// for loop i

System.***out***.println(IntStream.*range*(0, array.length).boxed().collect(Collectors.*toList*()));

//for loop j

System.***out***.println(IntStream.*range*(0, array.length).boxed().flatMap(i->IntStream.*range*(i+1, array.length).boxed()).collect(Collectors.*toList*()));

//make pair objects {array[i],array[j]} for all combinations

System.***out***.println(IntStream.*range*(0, array.length).boxed().flatMap(i->IntStream.*range*(i+1, array.length).mapToObj(j->**new** **int**[] {array[i],array[j]})).collect(Collectors.*toList*()));

// check pair total ==12

System.***out***.println(IntStream.*range*(0, array.length).boxed().flatMap(i->IntStream.*range*(i+1, array.length).mapToObj(j->**new** **int**[] {array[i],array[j]})).filter(pair->pair[0]+pair[1]==12).collect(Collectors.*toList*()));

//convert pair object to Intstream

System.***out***.println(IntStream.*range*(0, array.length).boxed().flatMap(i->IntStream.*range*(i+1, array.length).mapToObj(j->**new** **int**[] {array[i],array[j]})).filter(pair->pair[0]+pair[1]==12).flatMapToInt(pair->IntStream.*of*(pair)).boxed().collect(Collectors.*toList*()));

1. **Can we do static method overriding in java,**

In Java, it is not possible to override a static method from a superclass. When a method is declared static in a superclass, it becomes a class-level method, which means it is associated with the class itself rather than with instances of the class.

When a subclass defines a static method with the same signature as a static method in its superclass, it is not considered an override. Instead, it is simply hiding the static method in the superclass. This means that when the subclass's static method is called, the superclass's static method will not be called even if the subclass's method does not provide an implementation for the method.

class Superclass {

public static void printMessage() {

System.out.println("Hello from Superclass");

}

}

class Subclass extends Superclass {

public static void printMessage() {

System.out.println("Hello from Subclass");

}

}

public class Main {

public static void main(String[] args) {

Superclass.printMessage(); // Output: "Hello from Superclass"

Subclass.printMessage(); // Output: "Hello from Subclass"

}

}

Therefore, in Java, it is not possible to perform method overriding on static methods, as they are bound to the class itself, rather than the instance of the class.

1. **When will static members get initialized/executed, for example like compile time/run time?**

Run time.

In Java, static members (variables and methods) are associated with the class rather than with instances of the class. This means that they are initialized/executed when the class is loaded by the JVM, which typically happens at the time of the first reference to the class, either by creating an object of the class or by referring to a static member of the class.

public class MyClass {

static int x = 10;

static {

System.out.println("Static block 1");

}

static {

System.out.println("Static block 2");

}

static void myMethod() {

System.out.println("Static method");

}

public static void main(String[] args) {

System.out.println("Main method");

MyClass.myMethod();

}

}

**Output:**

Static block 1

Static block 2

Main method

Static method

1. **Have you implemented Multithreading concepts in your app, how you implemented them?**

Yes,

We process employee time/punch card in daily basis. Instead of putting all employees in one loop, we created multiple threads over plant code and process the data.

1. **Checked Exceptions vs Unchecked Exceptions, handling of it with examples?**

Exceptions in Java are divided into two categories: Checked Exceptions and Unchecked Exceptions.

**Checked Exceptions:**

Checked Exceptions are the exceptions that are checked at compile time. This means that the compiler checks if the method has a proper exception handling mechanism or not. If a method throws a checked exception, it must either handle it using a try-catch block or declare it in the method signature using the 'throws' keyword. Examples of Checked Exceptions are IOException, ClassNotFoundException, etc. ex: filereading

Unchecked exceptions, also known as runtime exceptions, are exceptions that are not checked by the compiler. Examples of unchecked exceptions include NullPointerException, ArrayIndexOutOfBoundsException, and IllegalArgumentException.

Example: accessing array item which is indexed more than lenth.

1. **Finally block and its benefits, what is improved in Java7 on I/O resource closing?**

The **finally** block is a crucial part of exception handling in Java. It is used to define a block of code that is executed after a **try** block completes, whether or not an exception is thrown. The primary benefit of the **finally** block is that it ensures that critical resources are always released

1. **Difference between List and a Set? Give example for right use?**

A List is an ordered collection of elements where each element is identified by its index. Elements can be added or removed from any position within the list. Lists can contain duplicate elements, and you can access an element in a List by its index.

A Set, on the other hand, is an unordered collection of unique elements. Sets do not allow duplicate elements and do not provide an index-based access method to retrieve elements. Instead, you can use the **contains()** method to check if an element exists in a Set.

1. **What is String immutability, why does java have it/benefits?**

In Java, a String is an object that represents a sequence of characters. One of the key characteristics of a String in Java is that it is immutable, which means that once a String object is created, its value cannot be changed.

String immutability is implemented in Java for a few reasons:

1. Security: Since Strings are often used to store sensitive information such as passwords or credit card numbers, making Strings immutable helps to ensure that the values of these Strings cannot be changed once they are created. This reduces the risk of security breaches.
2. Thread-safety: Immutable objects are inherently thread-safe since they cannot be changed by multiple threads simultaneously. This means that multiple threads can access and use the same String object without any synchronization issues.
3. Performance: Because String objects are immutable, they can be safely shared across multiple objects and threads without the risk of one thread modifying the value of a String while another thread is using it. This can improve performance in applications where Strings are frequently used.
4. Caching: Because String objects are immutable, they can be cached and reused in memory. This can improve the performance of applications that use a lot of String objects since the memory footprint is reduced.

For example, if we have a String **str** and we want to modify it by appending another string, we cannot modify the original String. Instead, we have to create a new String object that contains the concatenated value:

String str = "hello";

str += " world"; // creates a new String object

System.out.println(str); // prints "hello world"

1. **Hashcode() and Equals() overriding contract and why do we do this?**

In Java, **hashCode()** and **equals()** are methods defined in the **Object** class. The **hashCode()** method returns an integer value that represents the object's unique identifier, while the **equals()** method compares the current object with another object to determine if they are equal.

When working with custom classes, it is important to override these methods to ensure that they behave properly. The general contract for these methods is:

1. If two objects are equal according to the **equals()** method, then calling **hashCode()** on each of the objects must produce the same integer result.
2. If two objects are not equal according to the **equals()** method, then calling **hashCode()** on each of the objects does not necessarily produce distinct integer results. However, it is recommended that distinct objects have distinct hash codes for better performance in hash-based collections.

We override these methods to ensure that objects of our custom classes can be properly compared and used in hash-based collections such as **HashMap**, **HashSet**, and **Hashtable**. If we don't override these methods, the default implementation provided by **Object** class will be used, which compares object references rather than object contents. This can lead to unexpected behavior and incorrect results when using these collections.

By overriding **equals()** and **hashCode()** methods, we can make sure that objects with the same state are treated as equal, even if they are different instances. This enables us to properly compare and store objects in hash-based collections, and provides better performance and efficiency.

1. **How do you do Junit testing, why do we need it?**

JUnit is a unit testing framework for Java programming language that provides support for writing and running automated tests. It allows developers to write test cases to check that the code behaves as expected, and to detect and fix issues in the code before they are deployed to production.

JUnit tests are written in Java and are typically organized into test suites, which group related test cases together. Each test case consists of a series of assertions that check whether the code under test behaves correctly in various scenarios.

To write JUnit tests, follow these steps:

1. Add the JUnit dependency to your project's build file (e.g. **pom.xml** for Maven or **build.gradle** for Gradle).
2. Create a test class for the class or method you want to test, using the **@Test** annotation to indicate which methods are test methods.
3. Write test methods that create test cases and assert the expected behavior of the code under test.
4. Run the tests using a testing framework such as JUnit.

JUnit provides a number of assertions and annotations that make it easy to write and run tests. Some of the most commonly used annotations include:

* **@Test**: indicates that the method is a test method.
* **@Before**: indicates that the method should be run before each test method to set up test data.
* **@After**: indicates that the method should be run after each test method to clean up test data.
* **@BeforeClass**: indicates that the method should be run once before all test methods to set up any static data needed for the tests.
* **@AfterClass**: indicates that the method should be run once after all test methods to clean up any static data needed for the tests.

JUnit tests are important for several reasons:

1. They help ensure that the code behaves correctly: JUnit tests provide a safety net that helps catch bugs early in the development process, before they cause problems in production.
2. They improve code quality: Writing testable code requires careful thought about design and architecture, which can lead to better overall code quality.
3. They support continuous integration and delivery: Automated tests, including JUnit tests, can be run automatically as part of a continuous integration and delivery (CI/CD) pipeline, which helps ensure that changes to the codebase don't introduce new bugs or regressions.
4. They help document the code: Test cases serve as a form of documentation, providing examples of how the code should behave under different conditions.
5. **How do you write test cases for  end to -end API? What framework did you use?**

Writing test cases for an end-to-end API involves testing the complete flow of the API, from the client sending a request to the server to receiving a response back from the server. To write test cases for an end-to-end API, you can follow these steps:

1. Identify the endpoints that you want to test, along with their expected inputs and outputs.
2. Set up any necessary test data, such as test users or test database records.
3. Write test cases for each endpoint, using a testing framework such as JUnit, TestNG, or RestAssured.
4. For each test case, send a request to the API endpoint and verify that the response meets your expected results.
5. Clean up any test data that was created during the test.

The exact details of how to write the test cases will depend on the API and the testing framework that you are using. Here's an example of how you could write an end-to-end API test using RestAssured:  
import io.restassured.RestAssured;

import io.restassured.response.Response;

import org.junit.jupiter.api.Test;

import static io.restassured.RestAssured.given;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class MyApiEndToEndTest {

@Test

public void testMyApiEndpoint() {

// Set up the request parameters

String endpointUrl = "http://localhost:8080/api/my-endpoint";

String requestPayload = "{\"key\":\"value\"}";

// Send the request and get the response

Response response = given()

.contentType("application/json")

.body(requestPayload)

.when()

.post(endpointUrl)

.then()

.extract().response();

// Verify the response

assertEquals(200, response.statusCode());

assertEquals("expectedResult", response.path("result"));

}

}

In this example, we are using RestAssured to send a POST request to the API endpoint, passing in a JSON request payload. We then verify that the response status code is 200 and that the "result" field in the response matches our expected result.

Overall, the approach to writing end-to-end API tests involves writing test cases that cover the entire flow of the API, from the client sending a request to the server to receiving a response back from the server. The exact implementation details will depend on the API and the testing framework being used.

1. **Have you done integration testing, which tools/framework are used? Why do we do this?**
2. **Code coverage tools used, if code coverage is 100%, means there won’t be any bugs?**

There are several code coverage tools that are commonly used in Java development, including JaCoCo, Cobertura, and Emma.

Code coverage tools help measure the extent to which your code is being tested by your test suite. They work by instrumenting the code, running the tests, and then reporting on which lines of code were executed during the test run.

While having a high code coverage percentage is generally a good thing, it does not guarantee that there are no bugs in the code. Code coverage measures the amount of code that is executed by the tests, but it doesn't necessarily measure the quality of the tests themselves.

For example, it's possible to write tests that cover every line of code in a program, but still miss important edge cases or interactions between different parts of the code. Additionally, code coverage tools may not detect errors in the logic of the tests themselves.

Therefore, while code coverage is an important metric for evaluating the quality of a test suite, it should be used in conjunction with other testing techniques such as manual testing, exploratory testing, and code reviews.

1. **How does CI/CD work for your application?**

CI/CD (Continuous Integration/Continuous Delivery or Deployment) is a software development practice that involves automating the process of building, testing, and deploying software changes. Here's how CI/CD typically works for a web application:

1. Code changes are committed to a version control system, such as CSVN.
2. A build server, such as Jenkins or CircleCI, monitors the version control system for new changes.
3. When a new change is detected, the build server automatically checks out the code and builds the application.
4. Automated tests are run on the built application to verify that the changes haven't introduced any regressions or bugs.
5. If the tests pass, the built application is deployed to a staging environment for further testing.
6. Manual or automated tests are run in the staging environment to verify that the changes are working correctly.
7. If the changes pass testing in the staging environment, they are deployed to the production environment.

The goal of this process is to minimize the time between writing code and getting it into the hands of users, while also minimizing the risk of introducing regressions or bugs into the application. By automating the build, testing, and deployment process, developers can focus on writing code and rely on the automation to ensure that the application remains stable and functional.

In summary, CI/CD works by automating the process of building, testing, and deploying software changes, which helps to reduce the time between writing code and getting it into the hands of users, while also minimizing the risk of introducing bugs into the application.

1. **Where is the application deployed, how does it get deployed?**

It is deployed in JBOSS server. It will be deployed using Jenkins.

1. **In code reviews, what do you review?**

In a code review, there are several areas that you should review to ensure that the code is of high quality and meets the requirements of the project. Here are some common areas to focus on during a code review:

1. **Functionality**: Check that the code implements the required functionality and meets the project requirements.
2. **Code structure and organization**: Check that the code is well-organized, easy to understand, and follows best practices for code structure.
3. **Readability**: Check that the code is readable and easy to understand, with descriptive variable and function names, appropriate comments, and clear logic.
4. **Efficiency**: Check that the code is efficient and doesn't waste resources, such as memory or processing power.
5. **Security**: Check that the code is secure and doesn't contain any vulnerabilities or potential exploits.
6. **Testing**: Check that the code has appropriate unit tests, and that the tests cover all required functionality and edge cases.
7. **Maintainability**: Check that the code is maintainable, meaning that it can be easily modified and extended in the future.
8. **Style consistency**: Check that the code follows a consistent coding style and adheres to any style guidelines set out by the project or organization.

When reviewing code, it's important to provide constructive feedback to the developer to help them improve their code. It's also important to keep an open mind and be willing to learn from the developer, as they may have insights or perspectives that you haven't considered.When reviewing code, it's important to provide constructive feedback to the developer to help them improve their code. It's also important to keep an open mind and be willing to learn from the developer, as they may have insights or perspectives that you haven't considered.

1. **What is Spring Bean dependency injection?**

Spring Bean dependency injection is a key feature of the Spring Framework, which allows developers to manage dependencies between different components of their application. In this model, dependencies between objects are managed by the Spring container, which creates, configures, and injects beans into other beans as needed.

In the Spring framework, a bean is simply an object that is instantiated, assembled, and managed by the Spring IoC container. When a bean is created, it can be configured to have dependencies on other beans in the application, which are automatically resolved by the container using one of several mechanisms, including constructor injection, setter injection, or field injection.

It can be achieved by using **@Autowired** or **@Inject**

1. **What are different types of bean injections?**

In Spring Framework, there are three types of dependency injection, also known as bean injections:

**1.Constructor Injection:** In this type of injection, dependencies are injected via constructor parameters. When a bean is created, Spring looks at the bean's constructor arguments and tries to match them with other beans in the container. If a match is found, Spring injects the dependency into the constructor parameter. Here's an example:

public class MyController {

private final MyService myService;

public MyController(MyService myService) {

this.myService = myService;

}

}

**2. Setter Injection:** In this type of injection, dependencies are injected via setter methods. When a bean is created, Spring looks for setter methods that match the type of the dependency being injected. If a match is found, Spring calls the setter method and injects the dependency. Here's an example:

public class MyController {

private MyService myService;

public void setMyService(MyService myService) {

this.myService = myService;

}

}

**3. Field Injection:** In this type of injection, dependencies are injected directly into class fields using Java reflection. When a bean is created, Spring looks for fields that match the type of the dependency being injected. If a match is found, Spring injects the dependency into the field. Here's an example:

public class MyController {

@Autowired

private MyService myService;

}

1. **What are different Springboot Annotations used?**

Spring Boot provides a wide range of annotations to help developers build robust and scalable applications. Here are some of the most commonly used Spring Boot annotations:

1. @SpringBootApplication: This annotation is used to mark the main class of a Spring Boot application. It enables component scanning and auto-configuration features of Spring Boot.
2. @RestController: This annotation is used to mark a class as a RESTful controller. It combines @Controller and @ResponseBody annotations.
3. @RequestMapping: This annotation is used to map HTTP requests to handler methods. It can be used at the class level and method level.
4. @Autowired: This annotation is used to inject dependencies into a Spring managed bean.
5. @Component: This annotation is used to mark a class as a Spring component. It enables automatic component scanning and registration.
6. @Configuration: This annotation is used to mark a class as a configuration class. It can contain methods that return beans to be registered with the Spring application context.
7. @EnableAutoConfiguration: This annotation is used to enable auto-configuration of the Spring application context. It automatically configures beans based on the dependencies and classpath.
8. @Value: This annotation is used to inject values from application.properties or application.yml files into Spring managed beans.
9. @Profile: This annotation is used to activate a specific profile in a Spring Boot application.
10. @Conditional: This annotation is used to conditionally create a bean or configure a class based on a condition.

These are just a few of the many annotations provided by Spring Boot. By leveraging these annotations, developers can quickly and easily build powerful and scalable applications.

1. **How did you decompose your monolith application to microservices, difficulties faced?**

Decomposing a monolithic application into microservices is a complex and challenging process that requires careful planning and execution. Here are some general steps that can be followed to decompose a monolith application into microservices:

1. Identify Business Capabilities: Identify the business capabilities of the monolith application and group them based on their function.

2. Define Microservices: Define the microservices that will be created based on the identified business capabilities. Each microservice should have a single responsibility and be independently deployable.

3. Design Communication Protocols: Design communication protocols between the microservices to ensure that they can communicate with each other effectively.

4. Implement Microservices: Implement the microservices and deploy them into a container-based platform like Kubernetes or Docker Swarm.

5. Migrate Data: Migrate data from the monolith application to the microservices, ensuring that data consistency is maintained throughout the migration process.

6. Test and Deploy: Test the microservices to ensure that they are functioning as expected and deploy them into production.

7. Monitor and Manage: Monitor the microservices in production to ensure that they are performing well and manage them as needed.

Some of the difficulties faced during the process of decomposing a monolith application into microservices are:

Organizational Challenges: Decomposing a monolith application into microservices can also create organizational challenges. It requires collaboration between different teams and a change in the way software is developed and deployed. It is important to ensure that everyone involved is aligned and committed to the process.

1. Complexities in Communication: As microservices are distributed, there is a need to design communication protocols that allow for effective communication between the different services. This can be complex and requires careful planning and execution.

2. Data Consistency: When data is migrated from the monolith application to the microservices, ensuring data consistency can be challenging. It is important to carefully plan and execute the data migration process to ensure that data consistency is maintained.

3. Monitoring and Management: As the number of microservices increases, monitoring and managing them becomes more complex. Tools like Kubernetes can help to simplify this process, but it still requires careful attention to ensure that the microservices are performing as expected.

**DATABASE:**

Spring boot developers need to know how to call/use Oracle stored procedures PL/SQL but not required to be a master in PL/SQL. Please find the questions from couple of PL/SQL interview. Sharing these as optional item so that you can prep as needed.

1. What is the difference between procedure and function in PL/SQL.

In PL/SQL, procedures and functions are both subprograms that can be used to encapsulate a set of SQL statements and logic. However, there are some differences between the two:

1. Return Type: A procedure doesn't return any value, whereas a function returns a value. A function must have a return type defined, whereas a procedure does not.

2. Usage: A procedure is used to perform an action, such as inserting data into a database or updating a record, whereas a function is used to return a value based on some computation.

3. Calling: A procedure is called using the "call" statement or as a part of a SQL statement, whereas a function is called as part of an expression or as a value in a SQL statement.

4. Parameters: Both procedures and functions can take parameters, but functions must return a value based on the parameters passed, whereas a procedure can perform an action based on the parameters passed.

5. Modifying Data: A procedure can modify data in the database, whereas a function is not allowed to modify data in the database.

In summary, procedures are used to perform an action without returning a value, whereas functions are used to return a value based on some computation. Procedures can modify data in the database, whereas functions are not allowed to do so. Both procedures and functions can take parameters.

1. What are triggers, their types and use

In PL/SQL, a trigger is a special type of stored procedure that is automatically executed in response to certain events, such as insertions, updates, or deletions in a table. There are two main types of triggers:

1. Row-level Triggers: These triggers are fired for each row that is affected by the triggering event. They can be used to enforce complex business rules or perform complex calculations based on the data in the affected rows.

2. Statement-level Triggers: These triggers are fired only once for each triggering event, regardless of the number of rows that are affected by the event. They can be used to enforce database-level constraints or perform database-level operations.

Triggers can be defined to execute either before or after an event. Before triggers are executed before the triggering event is processed, whereas after triggers are executed after the triggering event is processed. Additionally, triggers can be defined to execute either once per row (for row-level triggers) or once per statement (for statement-level triggers).

Triggers are typically used to enforce business rules or data integrity constraints that cannot be enforced by simple database constraints, such as ensuring that certain data is entered in a particular format, or calculating certain values based on the data in a table.

For example, a trigger can be used to ensure that every order placed by a customer is assigned a unique order number, or to automatically calculate the total cost of an order based on the quantity and price of the items ordered.

Overall, triggers can be a powerful tool for enforcing complex business rules or data integrity constraints in a PL/SQL application. However, they should be used with caution, as they can add complexity and overhead to the database, and can be difficult to debug and maintain.

1. What are Autonomous transactions

Autonomous transactions refer to a database transaction that is independent and self-contained, meaning that it can be executed independently of the main transaction in which it was called.

In other words, when an autonomous transaction is executed, it creates a separate transaction that can be committed or rolled back independently of the main transaction. This can be useful in certain scenarios, such as when a procedure needs to perform some logging or auditing activities that should not be rolled back if the main transaction fails.

1. Types of Exceptions and Raise application error in stored procedures
   1. Predefined exceptions: These are system-defined exceptions that are raised when specific errors occur during the execution of a program. Examples of predefined exceptions include NO\_DATA\_FOUND, TOO\_MANY\_ROWS, and INVALID\_NUMBER.
   2. User-defined exceptions: These are exceptions that are defined by the programmer to handle specific errors in their code. User-defined exceptions can be raised using the RAISE statement.

The RAISE\_APPLICATION\_ERROR procedure is a specific way to raise a user-defined exception.

1. Difference between IN and Exists in a select clause

In a SQL select statement, the IN operator and the EXISTS operator are used to filter data based on a condition. Here is the difference between the two:

The IN operator is used to compare a value to a list of values. It returns true if the value matches any value in the list. For example, the following query returns all the employees who work in the sales or marketing department:

SELECT \* FROM employees

WHERE department IN ('Sales', 'Marketing');

The EXISTS operator is used to check whether a subquery returns any rows. It returns true if the subquery returns at least one row. For example, the following query returns all the customers who have placed an order:

SELECT \* FROM customers

WHERE EXISTS (

SELECT \* FROM orders

WHERE orders.customer\_id = customers.id

);

In summary, the IN operator is used to compare a **value** to a **list of values**, while the **EXISTS** operator is used to check whether a **subquery** returns any rows.

1. Describe PL/ SQL collections

PL/SQL collections are composite data types that allow you to store multiple values in a single variable. There are three types of collections in PL/SQL:

**1. Associative Arrays (also known as Index-by Tables):** Associative arrays are similar to arrays in other programming languages but do not require you to declare the size of the array. Instead, you can add elements to the array dynamically at runtime. Associative arrays are indexed by a unique key value, which can be a number or a string. The syntax for declaring an associative array is as follows:

**Example:**

DECLARE

TYPE emp\_names IS TABLE OF VARCHAR2(50) INDEX BY NUMBER;

emp\_list emp\_names;

BEGIN

emp\_list(100) := 'John';

emp\_list(200) := 'Jane';

emp\_list(300) := 'Bob';

END;

**2. Nested Tables:** Nested tables are similar to associative arrays, but they can be stored in a database column and can be queried using SQL. Unlike associative arrays, nested tables can be declared as standalone objects or as attributes of an object type. The syntax for declaring a nested table is as follows:

DECLARE

TYPE int\_table IS TABLE OF NUMBER;

my\_table int\_table := int\_table(1, 2, 3, 4, 5);

BEGIN

-- do something with my\_table

END;

**3. Varrays (Variable-size Arrays):** Varrays are similar to nested tables, but they have a fixed size that is specified at the time of declaration. The syntax for declaring a varray is as follows:

DECLARE

TYPE emp\_names IS VARRAY(3) OF VARCHAR2(50);

emp\_list emp\_names := emp\_names('John', 'Jane', 'Bob');

BEGIN

-- do something with emp\_list

END;

In summary, PL/SQL collections are composite data types that allow you to store multiple values in a single variable. There are three types of collections: associative arrays, nested tables, and varrays. Each type has its own unique characteristics and use cases.

1. What is Listagg function

The LISTAGG function is an aggregate function in SQL that concatenates the values in a column of a table or a result set into a single string. You can use this function to group values from multiple rows into a single column, separated by a specified delimiter.

The syntax for the LISTAGG function is as follows:

SELECT department, LISTAGG(name, ',') WITHIN GROUP (ORDER BY name) as employees

FROM employees

GROUP BY department;

1. Define TABLE function

In SQL, a TABLE function is a user-defined function that returns a set of rows as a table. It is also known as a table-valued function. You can use a TABLE function in a SQL query just like a regular table or view.

CREATE FUNCTION get\_employees\_by\_department (dept\_name VARCHAR2)

RETURN TABLE

AS

employees\_table employees%ROWTYPE;

BEGIN

SELECT \* INTO employees\_table

FROM employees

WHERE department = dept\_name;

RETURN employees\_table;

END;

1. What is a materialized view and why do we use them

In SQL, a materialized view is a database object that stores the results of a query as a precomputed table. Unlike a regular view, which is simply a saved SQL query that is executed on the fly when it is queried, a materialized view is an actual table that contains the results of the query. The results of the query are refreshed periodically, either on a schedule or manually, to keep the data up to date.

Here are some common use cases for materialized views:

**1. Aggregation:** Materialized views can be used to precompute aggregate values such as sums, averages, and counts. This is useful when performing queries that involve large datasets or complex aggregations.

**2. Join optimization:** Materialized views can be used to precompute the results of a join operation between two or more tables. This can be especially useful when the join operation is expensive or when the data in the underlying tables changes infrequently.

**3. Data warehousing:** Materialized views can be used to store precomputed data for use in a data warehouse. This allows users to query the data warehouse quickly and efficiently without the need to compute the results on the fly.

1. What is table partition and partitioned index

In SQL, table partitioning is a database feature that allows a large table to be split into smaller, more manageable parts, called partitions. Each partition is stored as a separate table, but is managed as a single entity by the database system. The partitions can be defined based on a range of values in a column (e.g. by date), a list of discrete values, or a hashing algorithm.

Partitioning can provide several benefits, including improved query performance, simplified data management, and reduced backup and restore times. By dividing a large table into smaller partitions, queries that only access a subset of the data can be executed more quickly, as the database system can eliminate irrelevant partitions during query processing. Additionally, partitioning can simplify data management tasks such as adding or removing data, as these tasks can be performed on a partition-by-partition basis.

A partitioned index is an index that is built on a partitioned table. Like the table itself, a partitioned index is divided into separate parts, or partitions, that are managed as a single entity by the database system. Partitioned indexes can improve query performance by allowing the database system to more efficiently locate the desired data.

The partitioning scheme used for a partitioned index can be different from the partitioning scheme used for the table itself. For example, a table may be partitioned by date, while its associated index may be partitioned by a different column.

Partitioning can be a complex feature to implement, and may require additional planning and configuration to use effectively. However, for very large tables or databases with high levels of concurrent access, partitioning can be a valuable tool for improving performance and simplifying data management.

1. Can you override optimizer to pick an index using hints

Yes, in SQL it is possible to override the query optimizer's choice of index using hints. A hint is a special instruction that is added to a SQL query to provide additional information to the optimizer about how to execute the query.

SELECT /\*+ INDEX (employees emp\_name\_idx) \*/ \*

FROM employees

WHERE last\_name = 'Smith';

1. What is NVL and NVL2

**NVL(expr1, expr2) :** In SQL, NVL() converts a null value to an actual value. Data types that can be used are date, character and number. Data type must match with each other i.e. expr1 and expr2 must of same data type.  
Syntax –

NVL (expr1, expr2)

**NVL2(expr1, expr2, expr3) :** The NVL2 function examines the first expression. If the first expression is not null, then the NVL2 function returns the second expression. If the first expression is null, then the third expression is returned i.e. If expr1 is not null, NVL2 returns expr2. If expr1 is null, NVL2 returns expr3. The argument expr1 can have any data type.

Syntax –

NVL2 (expr1, expr2, expr3)

**DECODE() :** Facilitates conditional inquiries by doing the work of a CASE or IF-THEN-ELSE statement.

The DECODE function decodes an expression in a way similar to the IF-THEN-ELSE logic used in various languages. The DECODE function decodes expression after comparing it to each search value. If the expression is the same as search, result is returned.

If the default value is omitted, a null value is returned where a search value does not match any of the result values.

Syntax –

DECODE(col|expression, search1, result1

[, search2, result2,...,][, default])

**COALESCE() :** The COALESCE() function examines the first expression, if the first expression is not null, it returns that expression; Otherwise, it does a COALESCE of the remaining expressions.

The advantage of the COALESCE() function over the NVL() function is that the COALESCE function can take multiple alternate values. In simple words COALESCE() function returns the first non-null expression in the list.

Syntax –

COALESCE (expr\_1, expr\_2, ... expr\_n)

**NULLIF() :** The NULLIF function compares two expressions. If they are equal, the function returns null. If they are not equal, the function returns the first expression. You cannot specify the literal NULL for first expression.

Syntax –

NULLIF (expr\_1, expr\_2)

**LNNVL() :** LNNVL evaluate a condition when one or both operands of the condition may be null. The function can be used only in the WHERE clause of a query. It takes as an argument a condition and returns TRUE if the condition is FALSE or UNKNOWN and FALSE if the condition is TRUE.

Syntax –

LNNVL( condition(s) )

**NANVL() :** The NANVL function is useful only for floating-point numbers of type BINARY\_FLOAT or BINARY\_DOUBLE. It instructs the Database to return an alternative value n2 if the input value n1 is NaN (not a number). If n1 is not NaN, then database returns n1. This function is useful for mapping NaN values to NULL.

Syntax –

NANVL( n1 , n2 )

1. What is Case and decode

There is one big difference between DECODE and CASE and it has to do with how NULLs are compared. DECODE will return "true" if you compare NULL to NULL. CASE will not. For example:

DECODE(NULL, NULL, 1, 0)

will return '1'.

CASE NULL

WHEN NULL THEN 1

ELSE 0

END

will return '0'. You would have to write it as:

CASE

WHEN NULL IS NULL THEN 1

ELSE 0

END

1. What are few techniques used in Sql Query tuning

SQL query tuning is the process of optimizing SQL queries to improve their performance and reduce their execution time. Here are a few techniques commonly used in SQL query tuning:

1. Indexing: Proper indexing of tables can significantly improve the performance of SQL queries. Indexes provide fast access to the data, which can speed up query execution. Adding indexes to columns that are frequently used in queries can improve performance.
2. Query optimization: The way queries are written can have a significant impact on their performance. Rewriting queries to use more efficient SQL statements or removing unnecessary joins, filters, and subqueries can often improve performance.
3. Caching: Caching query results can help to reduce the time required to execute a query. Once a query has been executed, the results can be stored in a cache, so that subsequent requests for the same data can be served from the cache.
4. Partitioning: Partitioning is a technique that can be used to split a large table into smaller, more manageable pieces. By dividing the data into smaller chunks, queries can be executed more efficiently, as only the relevant data needs to be processed.
5. Analyzing query execution plans: Query execution plans provide valuable information about how SQL queries are executed. Analyzing these plans can help to identify performance bottlenecks and suggest improvements that can be made to the query or underlying database schema.
6. Normalization: Normalization is the process of organizing data in a database to minimize redundancy. By eliminating redundant data, queries can be executed more efficiently, as fewer data needs to be processed.
7. Hardware optimization: Hardware optimization involves configuring hardware components such as the CPU, memory, and disk drives to ensure that they are operating efficiently. This can include increasing memory, using solid-state drives, or upgrading to faster processors to improve query performance.
8. What are the restrictions on calling a function from select query.

When calling a function from a SELECT query in SQL, there are a few restrictions to keep in mind:

1. Function must exist: The function being called must already exist in the database and be accessible to the user executing the SELECT query.
2. Function must return a value: The function being called must return a value that can be used in the SELECT query. If the function does not return a value, or returns a value of the wrong data type, the query will fail.
3. Function must have the correct number of parameters: If the function requires parameters, they must be provided in the correct order and with the correct data types. If the function does not require parameters, none should be provided.
4. Function must be deterministic: Functions used in SELECT queries must be deterministic, meaning they always return the same value given the same inputs. If a function is non-deterministic, the result of the query may be inconsistent.
5. Function must be compatible with the database engine: Functions must be compatible with the database engine being used. For example, a function that is designed for one database engine may not work in a different engine.
6. Job scheduler , Autosys, batch jobs

We use linux cron jobs for such kind of batch processes.

A job scheduler is a software tool that enables the automation of batch processing tasks or jobs. It allows the scheduling and execution of a sequence of jobs, typically in the background, without requiring user intervention. Autosys is an enterprise-class job scheduler used by many organizations to automate and manage batch jobs.

Batch jobs are tasks that are executed in the background, usually at a specific time or on a recurring basis. They can include tasks such as data backups, data transfers, or report generation. Batch jobs can be scheduled to run at a specific time, such as during off-peak hours when system usage is low, to reduce the impact on other users and applications.

Autosys is a job scheduling and management tool that is used to manage and automate batch jobs in an enterprise environment. It provides a centralized platform for managing and monitoring batch jobs across multiple servers and applications. Autosys can be used to schedule and manage a variety of batch jobs, including shell scripts, Java applications, database queries, and more.

Some of the features of Autosys include:

1. Scheduling: Autosys allows users to schedule jobs based on specific time and date, as well as event triggers.
2. Dependencies: Autosys supports dependencies between jobs, allowing users to ensure that a job is not started until all of its dependent jobs have completed successfully.
3. Monitoring: Autosys provides real-time monitoring of jobs, allowing users to track job progress and status, and receive alerts if a job fails or runs longer than expected.
4. Job Streams: Autosys supports the concept of job streams, which are sequences of related jobs that are executed in a specific order.
5. Centralized Management: Autosys provides a centralized platform for managing and monitoring batch jobs, making it easier to manage jobs across multiple servers and applications.

In summary, Autosys is a popular job scheduler used to manage and automate batch jobs in an enterprise environment. It provides a variety of features and capabilities to make job scheduling and management easier and more efficient.

1. CI/CD integration of packages, functions, procedures

CI/CD (Continuous Integration and Continuous Deployment) is a software development practice that involves regularly integrating code changes into a shared repository and automatically deploying those changes to production environments. In the context of packages, functions, and procedures in a database environment, the integration and deployment process may involve the following steps:

1. Version Control: The code for packages, functions, and procedures should be stored in a version control system (VCS) such as Git. This enables developers to work on the code in a collaborative manner, and provides a history of changes that can be tracked and rolled back if necessary.
2. Build and Test: Once the code changes have been committed to the VCS, a build process can be initiated to compile the code and generate any necessary artifacts such as binary packages or SQL scripts. Automated tests can also be run to verify that the code changes are functioning correctly.
3. Artifact Repository: The generated artifacts can be stored in an artifact repository such as Artifactory or Nexus. This provides a central location for storing and distributing the packages, functions, and procedures to different environments.
4. Integration Testing: Once the artifacts are generated and stored, they can be deployed to a test environment for integration testing. This involves testing the code changes in the context of other components in the system to ensure that they are functioning correctly and do not introduce any new issues.
5. Staging and Deployment: Once the integration testing is complete and the changes have been approved, they can be deployed to a staging environment. This provides an additional level of testing in an environment that is more similar to the production environment. Once the changes have been verified in the staging environment, they can be deployed to the production environment.
6. Monitoring and Feedback: Continuous monitoring of the production environment is necessary to ensure that the changes are functioning correctly and to detect any issues that may arise. User feedback can also be gathered to identify areas for improvement and to guide future development efforts.

In summary, the CI/CD integration and deployment of packages, functions, and procedures in a database environment involves a series of steps including version control, build and test, artifact repository, integration testing, staging and deployment, and monitoring and feedback. These steps help to ensure that changes are introduced into the production environment in a controlled and consistent manner, while minimizing the risk of errors and downtime.

1. What are Aggregate functions

Aggregate functions in SQL are functions that operate on a set of values and return a single value. They are commonly used to summarize data in queries by performing calculations such as counting, summing, averaging, or finding the maximum or minimum value in a set of data. Some of the common aggregate functions in SQL include:

1. COUNT(): This function counts the number of rows in a table or the number of non-null values in a specific column.
2. SUM(): This function calculates the sum of all the values in a specific column.
3. AVG(): This function calculates the average value of all the values in a specific column.
4. MAX(): This function returns the maximum value in a specific column.
5. MIN(): This function returns the minimum value in a specific column.
6. Describe Analytic functions (rank, dense\_rank, lag, lead)

Analytic functions in SQL are functions that operate on a set of rows and return a single value for each row based on a specified window or partition of data. Analytic functions are different from aggregate functions in that they do not collapse multiple rows into a single row, but instead calculate a value for each row based on a specified window of data.

Here are some common examples of analytic functions in SQL:

1. RANK(): This function assigns a rank to each row within a specified window or partition of data, based on the value of a specified column. Rows with the same value in the specified column will receive the same rank, and the next rank will be skipped. For example, if there are two rows with the same value and they receive a rank of 1, the next row will receive a rank of 3.
2. DENSE\_RANK(): This function assigns a rank to each row within a specified window or partition of data, based on the value of a specified column. Rows with the same value in the specified column will receive the same rank, and the next rank will be consecutive. For example, if there are two rows with the same value and they receive a rank of 1, the next row will receive a rank of 2.
3. LAG(): This function returns the value of a specified column from a previous row within a specified window or partition of data. The number of rows to look back is specified as an argument to the function.
4. LEAD(): This function returns the value of a specified column from a subsequent row within a specified window or partition of data. The number of rows to look ahead is specified as an argument to the function.

Here is an example of how to use the RANK() function:

SELECT name, score, RANK() OVER (ORDER BY score DESC) AS rank

FROM scores;

This statement will assign a rank to each row in the scores table based on the value of the score column, with the highest score receiving a rank of 1. The result set will include the name, score, and rank for each row.

In summary, analytic functions in SQL are functions that operate on a set of rows and return a single value for each row based on a specified window or partition of data. Common examples include RANK(), DENSE\_RANK(), LAG(), and LEAD(). These functions can be useful for performing complex calculations and analysis on large datasets.

1. When do you use Having clause

To complement a GROUP BY clause, use a HAVING clause to apply one or more qualifying conditions to groups after they are formed. The effect of the HAVING clause on groups is similar to the way the WHERE clause qualifies individual rows.

**React ~~Angular~~ Questions:**

* Most recent work experience with ~~Angular~~ React
* How was the code organized
* What are the benefits you get from feature modules
* How do you manage performance and optimization in Angular
* How do you handle routing
* How do you consume REST API
* Observables
* Services
* Examples of directives
* Have you ever worked with state management like Ngrx in Angular or Redux in React.
* Talk about a feature that you worked on for your most recent project.
* What version(s) of angular have you worked with
* Experience with Unit Testing and Automation

**QA Questions:**

* Tell me about yourself – Describe as a Automation test engineer
* What are OOP languages principles - Java
* What is locator?
* What is xpath?
* Cucumber and TestNG frameworks concepts and what are benefits out of using them.
* What are Challenges faced in automating testing…
* API automation frameworks
* How do you perform integration testing? Can you get an error after having 100% test coverage?
* What are the tools used for api testing… do you know Rest Assured?
* how do you check If jenkins fails
* API testing, rest apis, how do you configure BDD with APIs?
* Java, let's say you have several versions in text file and how do you get the latest version
* Singleton design pattern in your framework, and how do you achieve
* What is Scenario Outline and DDT in Cucumber
* How do you run test suites in Jenkins, what kind of configurations do you do
* How do you use Selenium Grid? Why do you need it
* What is after before methods class

Scrum Master:

Please find below the list of interview questions asked for Project, Role: Scrum Master

* Talk about scrum, Kanban and SAFe
* Talk about PI planning and what a SAFe environment looks like
* Other than the Sprint Retrospective What other ways can the Team collaborate?
* What is the biggest challenge you faced as a Scrum Master?
* What is Velocity? What are effects due to fluctuation in Velocity? How can you improve the Velocity?
* How do you recommend following up on Action items?
* What is your view of leadership as a Scrum Master?
* Talk about working with teams are new to agile
* What is the quality a Scrum Master should have to build a great Scrum Team?
* As a Scrum Master if you have a list of Open Impediments, How are you going to resolve them?
* How you handle changing priorities

**React JS**

1. What is React JS? Answer: React JS is a JavaScript library that is used for building user interfaces. It allows developers to create reusable UI components that can be used across different parts of the application.
2. What is a React component? Answer: A React component is a building block of a React application. It is a reusable piece of UI that can be rendered to the DOM. A component can be defined as a function or a class. Here is an example of a functional component:

import React from 'react';

function MyComponent() {

return <h1>Hello, world!</h1>;

}

1. What is JSX in React JS? Answer: JSX is a syntax extension to JavaScript that allows developers to write HTML-like code in their JavaScript files. It is used to define the structure and content of the UI components in React.
2. What is the Virtual DOM in React JS? Answer: The Virtual DOM is a lightweight copy of the actual DOM that React uses to keep track of changes to the UI. When there is a change in the data, React compares the current Virtual DOM with the previous one to find out what has changed and then updates the actual DOM accordingly.
3. What is the difference between state and props in React JS? Answer: State and props are both used to store data in React. The key difference is that state is used to store data that can change over time, while props are used to pass data from one component to another. State is managed within a component, while props are passed down from a parent component.

//STATE

import React, { useState } from 'react';

function MyComponent() {

const [count, setCount] = useState(0);

return (

<div>

<p>You clicked {count} times.</p>

<button onClick={() => setCount(count + 1)}>

Click me

</button>

</div>

);

}

In this example, we define a state variable called **count** and a function called **setCount** using the **useState** hook. We use these to manage the state of the component and update the count value when the button is clicked.

// **PROPS**

import React from 'react';

function ParentComponent() {

return <ChildComponent name="John" />;

}

function ChildComponent(props) {

return <p>Hello, {props.name}!</p>;

}

In this example, we define a parent component called **ParentComponent** that renders a child component called **ChildComponent** and passes a prop called **name**. The **ChildComponent** accesses this prop using **props.name** and renders a paragraph containing the name value.

1. What is the purpose of the render() method in React JS? Answer: The render() method is responsible for rendering the UI components on the screen. It is called whenever there is a change in the data or state of the component, and it returns a JSX code that is used to update the UI.
2. What are the lifecycle methods in React JS? Answer: The lifecycle methods in React are methods that are called at specific points during the life cycle of a component. These methods include componentDidMount, componentDidUpdate, componentWillUnmount, and shouldComponentUpdate.
3. What is Redux in React JS? Answer: Redux is a state management library that is used in React applications. It provides a centralized store for all the state data of the application and allows components to access and update this data.
4. What is the significance of keys in React JS? Answer: Keys are used to help React identify which items in a list have changed, been added or removed. They help improve the performance of the application by allowing React to only update the necessary parts of the DOM.
5. What is the difference between controlled and uncontrolled components in React JS? Answer: Controlled components are those where the value of the input field is controlled by React, while in uncontrolled components, the value of the input field is controlled by the DOM. Controlled components are generally preferred because they allow for more control over the input data.
6. What are higher-order components (HOCs) in React JS? Answer: Higher-order components (HOCs) are functions that take a component and return a new component with enhanced functionality. They are used to share common functionality between components and reduce code duplication.
7. What are React hooks? Answer: React hooks are functions that allow you to use state and other React features in functional components. They were introduced in React 16.8 as a way to use state and lifecycle methods in functional components.
8. What is the difference between useEffect and componentDidMount in React JS? Answer: The componentDidMount lifecycle method is called when a component is mounted for the first time, while useEffect is called after every render of the component. useEffect can be used to handle both component mount and update logic, while componentDidMount is only called once during the component mount.
9. What is the purpose of the shouldComponentUpdate lifecycle method in React JS? Answer: The shouldComponentUpdate method is used to optimize the performance of the application. It is called before the component is updated and allows you to check if the component should be re-rendered or not. By default, React re-renders the component whenever there is a change in the state or props, but shouldComponentUpdate can be used to prevent unnecessary re-renders.
10. What is the significance of React Router? Answer: React Router is a library that allows you to handle routing in a React application. It allows you to create a single-page application with multiple views by defining different routes for different components.
11. What is the difference between stateful and stateless components in React JS? Answer: Stateful components are those that have a state, while stateless components are those that do not have a state. Stateful components are generally used for more complex logic and state management, while stateless components are used for simpler UI rendering.
12. What is the difference between React and Angular? Answer: React is a JavaScript library for building user interfaces, while Angular is a complete framework for building web applications. React is more focused on the view layer, while Angular includes features such as routing, forms, and HTTP services.
13. What is Redux Thunk? Answer: Redux Thunk is a middleware for Redux that allows you to write asynchronous actions in a Redux application. It allows you to dispatch async actions that can make API calls and return data to the store.
14. What is the significance of React Developer Tools?

Answer: React Developer Tools is a browser extension that allows developers to inspect and debug React components in the browser. It provides a detailed view of the component hierarchy, state, and props, and allows you to see how changes to the data affect the UI.

1. What is the significance of the useContext hook in React JS?

Answer: The useContext hook allows you to use context in a functional component. Context is a way to share data between components without passing props down the component tree. useContext provides a simpler way to use context in functional components.

1. What is the significance of React Native?

Answer: React Native is a framework for building mobile applications using React. It allows developers to build native iOS and Android applications using the same codebase, and provides access to native features such as camera, GPS, and push notifications.

1. What is the difference between a class component and a functional component in React JS?

Answer: A class component is a component that extends the React.Component class and has a state object and lifecycle methods. A functional component is a component that is written as a JavaScript function and has no state or lifecycle methods. Functional components are simpler and easier to test, while class components are more powerful and flexible.

1. What is the significance of React Fiber?

Answer: React Fiber is a reimplementation of the React core algorithm that was introduced in React 16. It allows React to perform asynchronous rendering, which improves the performance of the application by breaking the rendering process into smaller chunks.

1. What is the difference between props and state in React JS?

Answer: Props are used to pass data from a parent component to a child component, while state is used to store data that can change over time within a component. Props are read-only and should not be modified, while state can be modified using the setState method.

1. What is the significance of the React DevTools Profiler?

Answer: The React DevTools Profiler is a tool that allows developers to profile the performance of their React components. It shows how long each component takes to render, and provides information about the component hierarchy and rendering times. This information can be used to optimize the performance of the application.

1. What is the purpose of the React Fragments feature?

Answer: React Fragments allow developers to group a list of children without adding extra nodes to the DOM. Fragments are useful when you need to render a list of components without adding a wrapper element around them.

1. What is the difference between React and Vue.js?

Answer: React is a JavaScript library for building user interfaces, while Vue.js is a progressive JavaScript framework for building user interfaces. Vue.js is more opinionated than React and includes more features out of the box, while React is more flexible and can be used with other libraries and frameworks.

1. What is the significance of React.memo?

Answer: React.memo is a higher-order component that allows you to memoize a component and improve its performance. It compares the props of the component with the previous props and only re-renders the component if the props have changed.

1. What is the significance of the React Server Side Rendering (SSR) feature?

Answer: React Server Side Rendering allows you to render the initial HTML on the server and send it to the client, which improves the performance and SEO of the application. It allows search engines to crawl the content of the application and provides a better user experience for users with slow internet connections.

1. What is the difference between shallow rendering and full rendering in React JS?

Answer: Shallow rendering is a technique for testing React components that only renders the component and its immediate children. Full rendering, on the other hand, renders the entire component tree and allows you to test the interactions between different components.

1. What is the purpose of the React Error Boundaries feature?

Answer: React Error Boundaries are components that catch JavaScript errors that occur during rendering and provide a fallback UI. They allow the application to continue rendering and prevent the entire UI from crashing.

1. What is JSX in React JS?

Answer: JSX is a syntax extension for JavaScript that allows you to write HTML-like code in your JavaScript files. It is used in React to define the structure and content of components. Here's an example:

import React from 'react';

function MyComponent() {

return (

<div>

<h1>Hello, world!</h1>

<p>This is a JSX component.</p>

</div>

);

}

In this example, we define a functional component called MyComponent that returns a JSX element containing a heading and a paragraph.

1. What is the purpose of the React createContext API? Answer: The React createContext API is used to create a context object that can be used to share data between components without passing props down the component tree. Here's an example:

import React, { createContext } from 'react';

const MyContext = createContext();

function MyComponent() {

return (

<MyContext.Provider value={{ name: 'John' }}>

<ChildComponent />

</MyContext.Provider>

);

}

function ChildComponent() {

return (

<MyContext.Consumer>

{value => <p>Hello, {value.name}!</p>}

</MyContext.Consumer>

);

}

In this example, we define a context object called MyContext and provide a value of **{ name: 'John' }** to it using the Provider component. We then consume the value of MyContext in the ChildComponent using the Consumer component and render a paragraph containing the name value.

1. What is the purpose of the React useEffect hook? Answer: The React useEffect hook is used to perform side effects in functional components. It replaces the lifecycle methods that are used in class components. Here's an example:

import React, { useState, useEffect } from 'react';

function MyComponent() {

const [count, setCount] = useState(0);

useEffect(() => {

document.title = `You clicked ${count} times`;

});

return (

<div>

<p>You clicked {count} times.</p>

<button onClick={() => setCount(count + 1)}>

Click me

</button>

</div>

);

}

In this example, we define a functional component called MyComponent that uses the useState hook to define a count state variable and a setCount function to update the state. We also use the useEffect hook to update the title of the document whenever the count state changes.

1. What is the purpose of the React useCallback hook? Answer: The React useCallback hook is used to memoize a function so that it is not re-created on each render. It is useful when passing functions down the component tree as props. Here's an example:

import React, { useState, useCallback } from 'react';

function MyComponent() {

const [count, setCount] = useState(0);

const incrementCount = useCallback(() => {

setCount(count + 1);

}, [count]);

return (

<div>

<p>You clicked {count} times.</p>

<Button onClick={incrementCount}>Click me</Button>

</div>

);

}

function Button({ onClick, children }) {

return (

<button onClick={onClick}>

{children}

</button>

);

}

In this example, we define a functional component called MyComponent that uses the useState hook to define a count state variable and a setCount function to update the state. We also define a function called incrementCount using the useCallback hook to memoize the function. We pass this function0

1. What is the difference between a class component and a functional component? Answer: A class component is a component that is defined as a class and extends the **React.Component** class. It can have state and lifecycle methods. A functional component is a component that is defined as a function and can use hooks to manage state and perform side effects.

Here is an example of a class component:

*import React, { Component } from 'react';*

*class MyComponent extends Component {*

*constructor(props) {*

*super(props);*

*this.state = { count: 0 };*

*}*

*render() {*

*return (*

*<div>*

*<p>You clicked {this.state.count} times.</p>*

*<button onClick={() => this.setState({ count: this.state.count + 1 })}>*

*Click me*

*</button>*

*</div>*

*);*

*}*

*}*

And here is an example of a functional component using the useState hook:

*import React, { useState } from 'react';*

*function MyComponent() {*

*const [count, setCount] = useState(0);*

*return (*

*<div>*

*<p>You clicked {count} times.</p>*

*<button onClick={() => setCount(count + 1)}>*

*Click me*

*</button>*

*</div>*

*);*

*}*

In both examples, we define a component that manages a count state and updates it