JavaScript Interview Questions

1. JavaScript is synchronous and single-threaded language => executes the single line at a time and once it gets over, it will switch to the next line.
2. Function Statement – Example

function a(){

    console.log("a called");

}

1. Function Expression – function used as values. It can be with or without name. With Function name is known as named function expression.

var b = function b(){

    console.log("b called");

}

1. Anonmyous Function – Function without any name and it can be used only as values as above, because it is not having any name.

function (){

    console.log("b called");

}

1. Named Function Expression – Function Expression with name is named function expression.

As in 2), b is the name and it is called named function expression.

1. Parameter – in the method, arguments are used in the call of the function
2. FirstClass Fucntion/FirstClassCitizens – The ablitiy in which function is used as values, passed as arguments and also return function

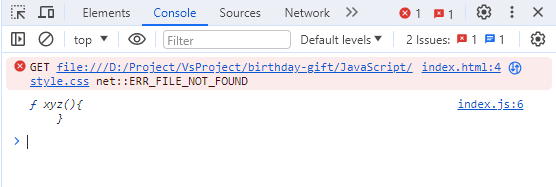
var b = function (param){

    return function xyz(){

    }

}

console.log(b());



1. CallBack Function – function which passed as arguments, used as async operations using setTimeout()

function x(y){

    console.log("x called");

    y();

}

x(function y(){

    console.log("y called");

});

1. Event Listeners are heavy. Nobody know when a button clicks, so it keeps in memory. It needs to be garbage collected.

function addClickEvent(){

    let count =0;

    document.getElementById("clickEvent").addEventListener("click",function(){

        console.log("Button Clicked", ++count);

    });

}

addClickEvent()

1. setTimeOut(function(){},5000) function, In the Web API, the call back method gets registered. When the timer got expired, the call back function goes to call back queue and the event loop checks if anything in the callback queue, if it is there then it pushes into CallStack
2. Event Loop: continuously listens the call stack and call back queue.
3. Microtask Queue: higher priority queue than callback queue. All the callback function from the promises and mutation observer goes to microtask queue. Mutation Observer observes continuously whether dom is mutated or not.
4. Higher Order Function – Function which is passed as an argument to another function or return function.
5. Map – to get new array out of original array
6. Filter – to get some array values from the list of array values
7. Reduce – takes all values of array and return only single value from it using some logic.
8. CallBack Hell – a callback function within callback function (pyramid of doom)

const cart = ["shoes", "shirts"]

api.createOrder(cart, function(){

    api.proceedPayment();

});

1. Inversion of Control – giving control to the program. In the above program, proceedPayment() is in the control of createOrder api. So this is issue with callback. It can be handled by **promises**.
2. Promise – empty object initially, after createOrder api returns some values and that is captured in promises.

A container for the future value.

Promise is an object representing the eventual completion or failure of an async operation.

const promise = createOrder(cart);

// Initially promise = {data: undefined}

// after some secs createOrder api returns orderID

// promise {data : orderId}

promise.then(function(){

    proceedPayment();

});

1. Fetch() – to call api to make external calls.
2. Promise contains two things

PromiseType – pending(initial step) , fulfilled(data returned), rejected(data not returned)

PromiseResult – contains the entire reponse as json object.

function createOrder(cart){

    const pr  = new Promise(function (resolve, reject){

        if(!isValidCart(cart)){

            const err =  new Error("Cart is not full");

            reject(err);

        }

        const orderId = "abc123";

        if(orderId){

            resolve(orderId);

        }

    });

    return pr;

}

1. Promise chain – promise contains another promise.
2. Promise Api’s,

* Promise.all() – It contains arrays of promises. If all promises are success, it will wait for the last one and return success , of all the arrays[val1,val2,val3]. If anyone fails, then it immediately return failure irrespective of others and won’t wait of other promises.
* Promise.allSettled() - Array of Promises. If all success, then it will return fulfilled of all arrays as promise.all(). If anyone fails, it will wait for other promises and return all the results. If array is of 3 promises, then output will be [val1, err2, val3]
* Promise.Race() - Array of promises. [p1,p2,p3] when all success. Whichever finishes first, it will return that value. If any promises settled with error , it will return the error , it considers for time only.
* Promise.Any() – It will wait for the first success and return it. If all fails, it will return aggregate error[err1,err2,err3].

1. Async() function – always return promise. If string is returned manually, it will automatically wraps into Promise and return it.

const pro = new Promise(function(resolve, reject){

    resolve("Resolved");

});

async function getData(){

    return pro;

}

const dataPromise = getData();

console.log(dataPromise);

dataPromise.then(function(resp){

    console.log(resp);

});

1. Await() function – needs to use await keyword infront of promise. Await keyword only inside async function().

const pro = new Promise(function(resolve, reject){

    resolve("Resolved");

});

async function getData(){

    const usingAwait = await pro;

    console.log(usingAwait);

}

getData();

// const dataPromise = getData();

// console.log(dataPromise);

// dataPromise.then(function(resp){

//     console.log(resp);

// });

1. Fetch() – gives to return promise. When promise is resolved, it will return response object. To convert this response (readable stream) to json.

const API\_URL = "https://api.github.com/users/Mukeshraj94";

async function getData(){

   const data = await fetch(API\_URL);

   const jsonVal = await data.json();

   //fetch => reponse.json => jsonVal

}

getData();

1. Exception handling –

async function getData(){

    try {

        const data = await fetch(API\_URL);

        const jsonVal = await data.json();

        //fetch => reponse.json => jsonVal

        console.log(jsonVal);

    } catch (error) {

        console.log(error);

    }

}

getData();

1. Async await Vs Promise.then/catch – async internally uses then/catch. If async is used, only after promises return something, then only next line will execute.
2. This keyword scopes.

* In Global Space – “this” keyword returns global object as “window” in browser, “global” in nodejs.
* Inside function () – In strict mode, it is undefined. In non-strict mode, “this” substitution happens and became global object.
* This inside object’s method – value of “this” keyword is object.
* In arrow function – enclosing lexical context.
* In DOM elements – it reference to particular html dom element.

Java Interview Question

1. For Setting environment variables, copy the path “C:\Program Files\Java\jdk-17\bin” where java compiler installed.
2. JDK = JDK + JRE(contains compiler and support libraries) + JVM(byte code will execute)
3. JRE = JRE + JVM
4. JVM = Stack Memory + Heap Memory(has String constant pool)
5. Stack Memory = contains local variable and methods.
6. Heap Memory = contains object and global variables.
7. Array – Collection of objects

    int arr[] = new int[3]; // fixed array

    int arr1[][] = new int[4][5]; //multi dimensional array

    int arr2[][] = new int[4][]; //jagged array - column can be different

    arr2[0] = new int[1];   //First row with 1 column

    arr2[1] = new int[2];   //Second row with 2 columns

    arr2[2] = new int[3];   //Third row with 3 columns

    arr2[3] = new int[4];   //Fourth row with 4 columns

1. Static variable – common for all the objects
2. Static block – called only once and it will be the first one to call before constructor.
3. Anonymous object – object created without any reference variable.

new Demo();

1. For Inheritance, even if .java is not present, the extended .java will inherit. Only, .class file is enough.
2. Ambiguity Problem – Multiple inheritance is not supported in java. Two parents may have same method name. In that case, child class get confused which to choose. For this, interface is used.
3. By default, every constructor has super.

class Demo{

    public Demo(){

        super();//by default

    }

}

class B extends A{

    public B{

        System.out.println("B called");

    }

}

1. Every class in java extends Object class.
2. this(), will call constructor of the current class.

class Demo{

    public Demo(){

        super();//by default

    }

}

class B extends Demo{

    public B(){

        System.out.println("B called");

    }

    public B(int a){

        this();// it will call B()

        System.out.println("B parametrized called");

    }

}

1. Dynamic method dispatch happens in polymorphism (many forms )

    A a= new A();

    a.show();

    a = new B();    //different forms

    b.show();

1. By default, .equals() compares two objects with hashcode
2. Equals(), will check only the reference object.
3. UpCasting and Downcasting

A obj= new B(); // implicitly Upcasting

    a.show1();

    B b = (B) obj;    // Downcasting

    b.show2();

1. Wrapper class – Integer, Double, Character. All primitive type has wrapper class. In int-> Integer.

    int num = 7;

    Integer num1 = num; // AutoBoxing - primitive type to wrapper class

    int num2 = num1;    //Auto-unboxing - wrapper class to primitive type

1. For abstract class, not able to create object. But object can be created with the help of Anonymous Inner Class(class with no name).

abstract class A{

    public abstract void show();

}

public class Main{

    public static void main(String[] args) {

       A a = new A() {

        public void show(){

            System.out.println("new show");

        }

       };

       a.show();

    }

}

1. Types of Interface: Normal, Functional interface or SAM(Single Abstract Method) – FI contains only one abstract method and Marker interface – no abstract method (useful for resuming videos)
2. FI – only one abstract method

@FunctionalInterface

interface A{

    void show();

}

public class Main{

    public static void main(String[] args) {

       A a = new A() {

        public void show(){

            System.out.println("new show");

        }

       };

       a.show();

    }

}

1. Lambda Expression – useful for FI

@FunctionalInterface

interface A{

    void show();

}

public class Main{

    public static void main(String[] args) {

       A a = () -> System.out.println("new show");  //lambda expression

       a.show();

    }

}

See 23 for better understanding.

1. Exceptions are runtime error.
2. explicity throw Exception using throw keyword
3. throws means that particular class not responsible but the parent has to handle the exception.
4. Buffered reader works with resources and it needs to be closed while after using it.

InputStreamReader in = new InputStreamReader(System.in);

      BufferedReader bf = new BufferedReader(in); // we can read data from user or read file,etc and other resources

      Scanner sc = new Scanner(System.in);

      try {

        System.out.println("Enter input:");

        int s = Integer.parseInt(bf.readLine());

        System.out.println("Entered Value:"+s);

      } catch (IOException e) {

        // TODO Auto-generated catch block

        e.printStackTrace();

      }

1. Finally is used for closing resources.it will execute even if there is no catch block. Whatever it may throw exception, finally will execute.

finally{

bf.close();

      }

1. For Optimal usage of closing resources, we use try with resources,

try (BufferedReader bf = new BufferedReader(new InputStreamReader(System.in)))

      {

        System.out.println("Enter input:");

        int s = Integer.parseInt(bf.readLine());

        System.out.println("Entered Value:"+s);

      } catch (IOException e) {

        e.printStackTrace();

      }

1. For class to use thread, it can be done by extends Thread or implements Runnable interface.
2. For thread extends, we can call directly using start() with class reference object

class A extends Thread{

  public void run(){

    System.out.println("A called");

  }

}

public class Main{

    public static void main(String[] args) {

      A a = new A();

      a.start();

    }

}

1. For Runnable interface, needs to create object of thread and call them.

class A implements Runnable{

  public void run(){

    System.out.println("A called");

  }

}

public class Main{

    public static void main(String[] args) {

      Runnable a = new A();

      Thread t = new Thread(a);

      t.start();

    }

}

1. Race condition can be eliminated by synchronize keyword in the method, where only one thread can execute that particular method at a time.
2. Collection API (concept),1) Collection(interface), 2) Collections(class) 3) Map
3. Collections – Comparable uses natural sorting order. compareTo(), uses one sort order
4. Collections – Comparator uses our own sorting order. Compare() , for comparing two different objects
5. Parallel Stream will be faster than stream but, don’t use parallel stream when sorting is performed.

Spring Interview Questions

https://github.com/navinreddy20/spring6-course/tree/main/3.1%20Working%20with%20Java-Based%20Config%20(3.1%20to%203.9)

1. IoC- Inversion Of Control principle (giving control to spring container)
2. DI – Dependency Injection is the design pattern to implement Ioc.
3. Spring Core – Dependency injection, Spring MVC – FrontEnd Controller,

Spring REST – Spring ORM -

1. Before Springboot, spring framework will create objects as below,

ApplicationContext context = new ClassPathXmlApplicationContext("spring.xml");

Alien al = (Alien)context.getBean("alien");

al.code();

In Spring.xml, <bean id=”alien” class=”com.mukesh.Alien”></bean>

1. Dependency Injection/ Inversion Of Control – Bean creation and injection will be taken care of the spring core.
2. By default, Spring container will create only one object, it same class object is created once again(Singleton pattern)
3. Prototype, separate object will be created for every new reference object.

<bean id=”alien” class=”com.mukesh.Alien” scope=”prototype”></bean>

1. Setter Injection: using set method, we are setting values

<bean id=”alien” class=”com.mukesh.Alien” scope=”prototype”>

<property id=”age” value=”10”></property>

</bean>

The above code works for only primitive datatype.

For other class reference object, we need to use as follow,

<bean id=”alien” class=”com.mukesh.Alien” scope=”prototype”>

<property id=”age” value=”10”></property>

<property id=”laptop” ref=”laptop”></property>

</bean>

<bean id=”laptop” class=”com.mukesh.Laptop”></bean>

1. Constructor injection- using constructor –arg tag in the xml for setting.

<bean id=”alien” class=”com.mukesh.Alien” scope=”prototype”>

<constructor-arg value=”12”></constructor-arg>

</bean>

1. Autowire – byname will search with “name” of the variable is present or not in that pojo

byType will search with the type of the variables in that pojo.

<bean id=”alien” class=”com.mukesh.Alien” autowire=”byType”>

<property id=”age” value=”10”></property>

<property id=”laptop” ref=”laptop”></property>

</bean>

<bean id=”laptop” class=”com.mukesh.Laptop”></bean>

1. Primary bean – Spring container will give priority to the one which is given with the keyword primary and consider for it, if there are two reference objects are there.

<bean id=”alien” class=”com.mukesh.Alien” autowire=”byType”>

<property id=”age” value=”10”></property>

<property id=”laptop” ref=”laptop”></property>

</bean>

<bean id=”laptop” class=”com.mukesh.Laptop” primary=”true”></bean>

<bean id=”desktop” class=”com.mukesh.Desktop”></bean>

1. Lazy-init – object will be created whenever bean gets called and required and spring container will not do it at the start of the project.

<bean id=”desktop” class=”com.mukesh.Desktop”></bean>

1. Java based configuration is similar to spring.xml based configuration

@Configuration

public class AppConfig {

// @Bean(name = {"Beast","desktop","com1"})

//multiple words can be used for a bean to call

@Bean

public Desktop desktop() {

return new Desktop();

}

}

public class App {

public static void main(String[] args) {

ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

Desktop dt = context.getBean("desktop",Desktop.class);

dt.compile();

1. @Scope(“prototype”) will be used after @Bean,
2. @Component – seterotype of annotation and has to be mentioned in the class. No need to write the code in 13) like @Bean and other stuffs
3. If we use @Component, then AppConfig there is no code and spring will not understand where to create beans, hence we need to use @Component-Scan to scan the beans

@Configuration

@Component-Scan(“com.mukesh”)

public class AppConfig {

1. @Qualifier has higher precedence than @Primary
2. @Value(“21”) private int age; we are injecting age value with @Value annotations.
3. Lombak jar will take care of getter and setter in pojo class.

@Data – for getter and setter

@NoArgsConstructor – for creating default constructor

@AllArgConstructor – for creating all variables parametrized constructor.

1. Hibernate config dependency jars: hibernate core(for hibernate connection), spring orm(for spring to connect with orm), Spring transaction(for handling transaction), mysql connector(for mysql connection), c3p0 (for handling multiple pooling)
2. Spring.jpa.hibernate.ddl-auto = create/update. Create uses to create new table, update will update the table if present or create new table
3. Spring.jpa.hibernate.show-sql= true. No need to write sql, hibernate will take care.
4. Query DSL (Domain Specific Language)- Query annotation to be used in the Repo class, when our own find logic will be used.

@Query(“select s from Student s where s.name=?1”)

List<Student> findByName(String name);

findByName- method needs to be overridden in the repo class.

SpringBoot Questions

1. Client sends request to server, inside server Controller takes care of request and sends back to request, that is the only job of Controller. After that, Service layer , where business logic will be returned. After that Repository/DAO layer where database related comes into picture. DAO sends data back to service, and service sends back to Controller.
2. @Service – for service layer. It internally calls @Component.
3. @Repository – for Db related function use this. It internally calls @Component.
4. Update() – for insert, update db

private JdbcTemplate jdbc;

public JdbcTemplate getJdbc() {

return jdbc;

}

@Autowired

public void setJdbc(JdbcTemplate jdbc) {

this.jdbc = jdbc;

}

public void save(Student s) {

String sql="insert into student(rollno,name,marks) values(?,?,?)";

int rows=jdbc.update(sql,s.getRollNo(),s.getName(),s.getMarks());

System.out.println(rows);

}

1. For fetching result using spring jdbc,

public List<Student> findAll() {

String sql="select \* from student";

RowMapper<Student> mapper=new RowMapper<Student>() {

@Override

public Student mapRow(ResultSet rs, int rowNum) throws SQLException

{ Student s=new Student(); s.setRollNo(rs.getInt("rollno"));

s.setName(rs.getString("name")); s.setMarks(rs.getInt("marks")); return s; }

};

return jdbc.query(sql, mapper);

}

1. @RequestMapping – It will tell the controller to pick the method in which client is requested.
2. Tomcat jasper jar – parses jsp files to compile them into java code as servlets
3. @RequestParam() – for identifying parameter which has been passed from jsp to controller

public ModelAndView add(*@RequestParam*("num1") int i, *@RequestParam*("num2")int j)

1. ModelAndView – contains data and jsp page details.

*@RequestMapping*("/add")

public ModelAndView add(*@RequestParam*("num1") int i, *@RequestParam*("num2")int j) {

ModelAndView mv = new ModelAndView("result");

int num3 = i+j;

mv.addObject("num3",num3);

return mv;

}

1. If we need to configure jsp in different folders, we can configure in the application.properties files. (.jsp) can be made customisable

spring.mvc.view.prefix=/views/

spring.mvc.view.suffix=.jsp

1. Model or ModeAndView can be used , based on our choice.

*@RequestMapping*("/add")

public String add(*@RequestParam*("num1") int i, *@RequestParam*("num2")int j, Model m) {

// ModelAndView mv = new ModelAndView("result");

int num3 = i+j;

// mv.addObject("num3",num3);

m.addAttribute("num3",num3);

return "result";

}

1. ModelAttribute – before using that annotation

public String addAlien(*@RequestParam*("aid") int i, *@RequestParam*("aname")String j, Model m) {

Alien a= new Alien();

a.setAid(i);

a.setAname(j);

m.addAttribute("alien",a);

return "result";

}

After using @modelattribute, it reduces all codes,

*@RequestMapping*("/addAlien")

public String addAlien(*@ModelAttribute*("aliens") Alien a, Model m) {

// public String addAlien(@RequestParam("aid") int i, @RequestParam("aname")String j, Model m) {

// Alien a= new Alien();

// a.setAid(i);

// a.setAname(j);

// m.addAttribute("alien",a);

return "result";

}

1. @ModelAttribute – at method level. Spring will call this method before controller and model gets created and value is assigned.

*@ModelAttribute*

public void modelData(Model m) {

m.addAttribute("name","Alien");

}

Result.jsp

Welcome Back ${name}

</**body**>

1. Whenever, we are sending data to the server, we will use POST. Whenever we are getting data from the server, then we will use GET.

<**form** action=*"addAlien"* method=*"post"*>

*@PostMapping*("/addAlien")

public String addAlien(*@ModelAttribute*("aliens") Alien a) { return "result";

}

1. ORM – Object Relational Mapping. ORM tool internally uses the JDBC Api to interact with the database.
2. Hibernate is a framework that simplifies the development of java application to interact with the database.
3. @Transactional – Inorder start and end the database transaction, need to use it.
4. In order to connect with database, spring data jpa jar required and dao needs to extend Jpa repo. In application.properties, some config is required.

public interface AlienRepo extends JpaRepository<Alien, Integer>{}

spring.datasource.url=jdbc:mysql://localhost:3306/mukesh

spring.datasource.username=root

spring.datasource.password=12345678

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect

1. @ResponseBody – at the method level, tells that the method will return json/xml type
2. @Controller + @ResponseBody = @RestController, it tells that the controller will return data in the form of json/xml
3. @Controller , will return the view resolver.
4. @RequestBody – at the parameter level of the method, used to tell that the input from postman(client) is of some data and it needs to be converted into json/xml.
5. Below are for server side to mention that the data is of type json or xml.

*@GetMapping*(path="/aliens",produces= {"application/json"})

public List<Alien> getAliens() {}

*@PostMapping*(path="/alien",consumes= {MediaType.***APPLICATION\_JSON\_VALUE***})

public Alien postAlien(*@RequestBody* Alien aln) {}

1. By default, java objects from client will be converted into Json with internal jar named as Jackson. But for xml, we need to import Jackson-dataformatter jar to be added.
2. If we have to use that conversion from the produce/consume
3. PathVariable – to mention that the client is getting data with some parameter,

*@GetMapping*("/alien/{aid}")

public Alien getAlien(*@PathVariable* int aid) {}

1. AOP – Aspect Oriented Programming. To achieve modularaisation by logging, transaction, etc.

*@Aspect*

*@Component*

public class LoggingAspect {

private static final Logger ***log*** = LoggerFactory.*getLogger*(LoggingAspect.class);

*@Before*("execution(public \* com.example.demo.HomeControllerForRest.getAliens())")

public void getLogBefore() {

***log***.info("Before GetAlien method called");

}

*@After*("execution(public \* com.example.demo.HomeControllerForRest.getAliens())")

public void getLogAfter() {

***log***.info("After finally GetAlien method called");

}

*@AfterThrowing*("execution(public \* com.example.demo.HomeControllerForRest.getAliens())")

public void getLogAfterThrow() {

***log***.info("After Throw GetAlien method called");

}

*@AfterReturning*("execution(public \* com.example.demo.HomeControllerForRest.getAliens())")

public void getLogAfterSuccess() {

***log***.info("After Success GetAlien method called");

}

}

1. DTO Data Transfer Objects – Data will be transferred from controller to service to repo by calling method name with object parameters
2. REST – Representational State Transfer. Just transferring data to the client. Rest is stateless, means everytime client hits server it will be new request. It will not remember, that it has been called early. Rest will deal with resources
3. @CrossOrigin(“http://localhost/jobPosts) - Inorder to connect with different framework like React.
4. Rest Repositories Web – It will create controller and service layer. In project only repo and model class is enough and nothing requires. While getting data from postman api, it will return json along with the “link” which will be used for retrieving.

Java MicroServices:

1. Microservices, handle each service differently.
2. Each service needs to register in the Eureka server.
3. In our case **Service-Registry**, StudentEurekaRegistry is the eureka server, only two dependencies are required,

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>

</dependency>

1. Below application.properties to be mentioned,

spring.application.name=StudentEurekaRegistry

server.port=8791

eureka.instance.hostname=localhost

eureka.client.fetch-registry=false

eureka.client.register-with-eureka=false

1. Annotation for Eureka server,

*@*SpringBootApplication

*@*EnableEurekaServer

public class StudentEurekaRegistryApplication {

1. **API Gateway**, has to be there in MS , where it will redirect the each service communication and it handles all the service request , similar to dispatchServlet,

spring.application.name=StudentApiGateway

server.port=9000

spring.cloud.gateway.discovery.locator.enabled=true

spring.cloud.gateway.discovery.locator.lower-case-service-id=true

eureka.client.service-url.defaultZone=http://localhost:8791/eureka

1. Two dependencies, eureka client and gateway to be added

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-gateway</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

1. StudentApplication microservice, below client dependency to be added. Feign dependency is for accessing the other microservice(Address)

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-openfeign</artifactId>

</dependency>

1. Feign Client annotation to be added,

*@SpringBootApplication*

*@EnableFeignClients*

public class StudentApplication {

1. For to access Address microservice, that can be done by adding that method signature as separate interface,

*@FeignClient*("StudentAddress")

public interface FeignInterface {

*@GetMapping*("/getAddressId/{id}")

public ResponseEntity<Address> getAddressId(*@PathVariable* Integer id);

}

*@*Autowired

FeignInterface fDao;

public ResponseEntity<Address> getAddressId(Integer id) {

// Address addr= aDao.findById(id).get();

Address addr= fDao.getAddressId(id).getBody();

return new ResponseEntity<Address>(addr,HttpStatus.OK);

}

1. Another Microservice Address,

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

1. Application.properties,

spring.application.name=StudentAddress

server.port=8090

eureka.client.service-url.defaultZone=http://localhost:8791/eureka

1. Incase if any failure happens in any service, we can use CircuitBreaker which will not stop the service. But it can be used for less risk only, not for payment service.
2. Below dependency to be added for circuit breaker,

<dependency>

<groupId>io.github.resilience4j</groupId>

<artifactId>resilience4j-circuitbreaker</artifactId>

<version>2.1.0</version>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-aop</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

1. Below application.properties to be added,

SlidingWindowSize, is last failure call happened

Failure rate threshold is in %

Duration in open state is in millisecond

If any service failed in closed state (for 10 slidingwindow size) in our case, then it will go into open state for some ms(30000) and the call will not happen during that time. Once after, 30000 ms, it will go into Half-Open state and there is some permitted number of call only will happen(5) . If the failure is < =failure threshold then it will enter into closed state and normal flow happens. If the failure >= failure threshold, then it will again enter into open state.

#resilience properties

resilience4j.circuitbreaker.instances.addressService.sliding-window-size=10

resilience4j.circuitbreaker.instances.addressService.failure-rate-threshold=50

resilience4j.circuitbreaker.instances.addressService.wait-duration-in-open-state=30000

resilience4j.circuitbreaker.instances.addressService.automatic-transition-from-open-to-half-open-enabled=true

resilience4j.circuitbreaker.instances.addressService.permitted-number-of-calls-in-half-open-state=5

#to see the status of the service , need actuator properties along with resilience

resilience4j.circuitbreaker.instances.addressService.allow-health-indicator-to-fail=true

resilience4j.circuitbreaker.instances.addressService.register-health-indicator=true

#properties applicable only to actuator

management.health.circuitbreakers.enabled=true

management.endpoints.web.exposure.include=health

management.endpoint.health.show-details=always

#circuit closed - "UP" in actuator

#circuit open - "DOWN" in actuator

#circuit half-open - "UNKNOWN" in actuator

1. CircuitBreaker is AOP, and It needs to be added separately in the different class as below,

*@Service*

public class AddressAop {

*@Autowired*

FeignInterface fDao;

*@*CircuitBreaker(name="addressService", fallBackMethod="fallBack")

public Address getAddressById(Integer id) {

return fDao.getAddressId(id).getBody();

}

public Address fallBack(Integer id) {

return new Address();

}

}

TCS Interview Questions

1. How to set dev, qa and prod in springboot.
2. How to eliminate some config in springboot
3. Convert json to xml in springboot
4. Controller vs restcontroller
5. What controller returns
6. How to perform parallel operaions in springboot
7. @restcontroller returns?
8. Flatmap in java8
9. Connect db with springboot

Java 8

1. Multiple Inheritance is achieved in JAVA 8 by default methods.

Rule 1- Classes take higher precedence than interfaces –

Rule 2 — Derived interfaces or sub-interfaces take higher precedence than the interfaces higher-up in the inheritance hierarchy-

Rule 3 — In case Rule 1 and Rule 2 are not able to resolve the conflict then the implementing class has to specifically override and provide a method with the same method definition-

1. Lambada is a function without a name. Eg: ()->{}
2. Lambda expression used for implementing FI /SAM
3. Functional Interfaces are only one abstract methods. In java8, below are the FI,

Consumer, Predicate, Function and Supplier.

1. Consumer FI – accept input and perform some operation of the input. It won’t return any values. It will perform some operations
2. Bi Consumer FI – accept two inputs.
3. Predicate – return Boolean and used for the resuse of code.
4. BiPredicate – return Boolean and accepts two parameter.
5. Function – accepts one input and returns one output.
6. Unary Operator – accepts one input and returns one output but both has to be of same return type.
7. Binary Operator – accepts two input and returns one output, Both input and output has to be of same return type.
8. Supplier – it won’t accept input but returns output.
9. Method reference – ClassName::methodName
10. Constructor reference – ClassName::new //not possible for A a = new A();
11. Effectively Final – In Lambda’s, allowed to use local variable but they are not able to change the value of it. Hence, it is known as Effectively Final.
12. Stream API – to perform operations on Collections
13. In stream, cannot add or modify elements and follows sequence retrieval. It will be invoked when the terminal operations are invoked.
14. Stream can be used only once.
15. Peek() – for debugging filter
16. Map () – convert one type to another. This map is different from Collection Map. Input is Function.
17. flatMap – similar to map but it will handle operation with Stream of List/arrays, etc. Whereas map will be Stream of single object.
18. Distinct – return streams of values without duplication
19. Count() – return streams with long values.
20. Sorted() – return streams in natural sorting.
21. Sorted(Comparator) – return stream with our own sorting order.
22. Filter() – filters the stream of elements. Input is predicate
23. Collect() – collect streams as input and Collectors as parameter. It is terminal operation where it is used at the end of the stream.
24. Reduce() – it is also terminal operation. Used to reduce stream of data’s into single value.

Takes two parameter (First param- default or initial value, Second param-Binary operator)

1. Reduce(single param) – it will return Optional class
2. Limit(n) – limit ‘n’ number, even if 10 or many elements are present
3. Skip(n) – skip the particular ‘n’ element
4. anyMatch() – returns true if anyone of the elements matches predicate

allMatch() – return true if all of the elements matches predicate

noneMatch() – return true if none of the elements matches predicate and return false if anyone match is found

1. findFirst() – return the first element as of type Optional
2. findAny() – return the first encountered element as of type Optional.
3. ShortCircuiting – Functions does not have to iterate the whole stream to evaluate the result.

[Limit, (findFirst, findAny), (anyMatch,allMatch,noneMatch)]

1. Stream Operations – Of() – create stream of certain values passed to this method.

Iterate() and generate() – create infinite streams

1. Numeric Streams – for number related operations. In normal stream, Wrapper Class will unbox from Integer to int but numeric stream will do it in the primitive types directly.
2. Range(1,50) – it won’t include last element in that range.

RangeClosed(1,50) – It will include last element in that range

1. Boxing() – convert primitive to wrapper class ex: int to Integer

UnBoxing – convert wrapper class to primitive ex: Integer to int.

1. Terminal operations – used at the last of the stream. And such are,

Foreach, reduce, min,max, collect, etc

1. Groupingby() collector is similar to SQL group by operation. Based on property it will group. Output will be Map<K,V>
2. partioningBy() collector is similar to groupingBy(). Accepts predicate as input and return is Map<K,V> where Key has to be Boolean.
3. Multiple Inheritance is possible in java after java8.
4. LocalDate, LocalTime and LocalDateTime part of java.time package. This is alternate to joda-time library. All these are immutable.
5. Period used for finding difference between two dates. Not able to perform difference operations in Time.
6. Duration is used in the case of time related logics.
7. Instant – represent time in machine readable format. Instant ins= Instant.now();
8. Jan 1,1970 – Epoch
9. Time Zones – Date/Time with its time zone.

**System Design Principles:**

1. Vertical Scaling
2. Preprocessing and Cron job (consider pizza making during non-peak hours)
3. Backups
4. Horizontal Scaling
5. MicroServices architecture
6. Distributed System
7. Load Balancer

* In load balancing, earlier if we use hasing of the request id it will generate some integer and mod of no. of servers, then it will be mapped the request to some place.

Ex: hash(request id 1) = 10 % 5(no of server) =0

* Here request id 1 will be served in Server (s0). Here load balance is 1/N , N-> no of servers
* Likewise other request are handled, but if new servers are added, the denominator to be changed and cache of each server will be lost.
* In order to overcome this scenario, Consistent Hashing gets used.
* Here, it will follow circular way to handle request, the nearer server will handle the request.
* Here load balance is also 1/N but the load will not be shifted to the all other servers, load will be handled to the next server.
* But, if that the case, only one server will handle too much load if there is some nearer server gets crashed.
* So here, K hash function to be used, means if there are 4 servers, then K=3 , then 12 servers will be handled circularly, in which each server will have 3 hashing function and handle the request efficiently.

1. Decoupling
2. Logging and Metrics.
3. Extensibility – no need to code again and again for different resources(Ex: Today delivery agent will deliver pizza, tomorrow they will deliver burger)
4. Scalability – how many server and how big a server can handle request
5. Horizontal Scaling – many resources are available there to handle.

Load balancing required.

Resilient – if one server not available, then other server will take care.

Network calls RPC- each server will communicate between them. Because of this, data inconsistency will be there.

Scales well, when users are more.

1. Vertical Scaling – One big server handles all the request. Due to this,

No load balance is required,

Inter process communication will be faster.

Data will be consistent.

But there is hardware limit.