### **一、Java 基础与高级**

1. **Checked and unchecked exceptions in Java.**IOException SQLException  
   NullPointerExcepetion ArrayIndexOutOfBounds
2. **Benefits of abstraction Why are we hiding the implementation? What is the use of hiding? From whom we are hiding?**

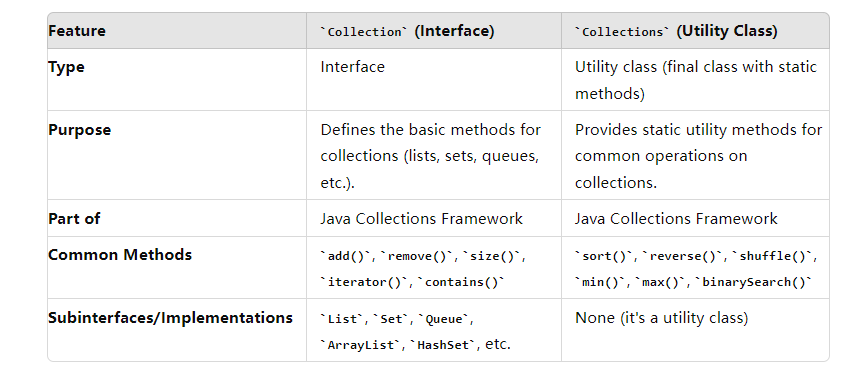
End user Developers Using Api Other Components or Systems

1. **Can a constructor be inherited in Java?**no
2. **Difference between abstract class and interface.**
3. **Why introduce default and static method?**
4. **Why decouple? Why is the interface not a tight couple?**
5. **Why is the service interface loosely coupled? Why loosely coupled?**
6. **What is the marker interface? What is the usage of marker interface? example of marker interface**
7. **Diamond problem occurs in the interface**
8. **What is method reference?**
9. **Explain deep copy and shallow copy in Java.**
10. **Why string immutable?**
11. **how to create immutable object? Use final keyword on class and perform deep copies of Mutable Fields.**
12. **Difference between Overloading and Overwriting in Java.**
13. **comparable and comparator**
14. **Thread life cycle**
15. **Explain the Singleton pattern and how to ensure it is thread-safe.**
16. **Explain the difference between throw and throws in Java.**
17. **Explain the differences between HashMap, HashTable, and ConcurrentHashMap in Java.**
18. **Why is ConcurrentHashMap faster than HashTable?**
19. **stream vs parallel stream**
20. **map vs flatMap in stream api**
21. **Real time use case of flatmap**
22. **volatile vs Synchronized**
23. **Explain the difference between ArrayList, LinkedList, and CopyOnWriteArrayList in Java.**
24. **What is the Callable interface and how is it different from Runnable?**
25. **What is the difference of future.thenAccept(); future.thenApply();** future.thenRunAsync();

future.thenAccept(); // take Consumer as input

future.thenApply(); // Function input

future.thenRunAsync() // Runnable input

1. **Collections vs Collection in Java.  
   **
2. **What is CompletableFuture? Rewrite the code to async execute it**
3. **What are the new features of Java 17?**
4. **What is a sealed class in Java? Can you provide a practical use case of a sealed class?**Permits keyword  
   public sealed class Shape permits Circle, Rectangle, Square
5. **Explain what is a command design pattern in Java.**
6. **What is the Observer design pattern in Java?**
7. **Explain the Open/Closed principle in Java with code.**
8. **What is the difference between sleep and wait in Java?**
9. **Is sleep a method of the Object class in Java?  
   No thread class**
10. **What is the difference between ArrayList's iterator and a for loop in Java?**
11. **Explain how to solve an out of memory exception in Java**

**JVM**

1. **How does Java internally work? How do linking, loading, and initialization happen in Java?**
2. **JAVA remove aot, jit, why**
3. **Type of Classloader**
4. **How Java internally work? JVM?**
5. **How memory located? What type of memory in Java?**
6. **Why Java 8 remove pattern generation?**
7. **Young generation -- `New`**
8. **Survival generation**
9. **Old generation**
10. **How GC work in those generation**
11. **Which gc is responsible for remove the garbage from one to another one?**
12. **How latency impact based on choice of gc?**
13. **Two types of minor gc, explain which situation use which gc?**
14. **What is dangling pointer problem? How to avoid or solve.**

### **二、Spring & Spring Security**

1. **How to solve the circular dependency problem in Spring?**@Lazy Setter injection @PostConstruct
2. **What’s JPA, What’s hibernate**
3. **Authorization and Authentication in Spring Security.**
4. **How to use OAuth 2 in Spring Security?**
5. **Use @Qualifier annotation in Spring with an example.**
6. **Use of Spring Actuator.**Monitoring health
7. **What are the design principles you used in Spring?**
8. **Explain the Factory design pattern in Java.**
9. **How to implement transactions in Spring?**
10. **What are the common design patterns used in Spring?**IOC Single Responsibility Open/Closed Principle
11. **Explain the Open/Closed principle in Java with code.**
12. **How to use @Autowired in Spring?**
13. **Difference between BeanFactory and ApplicationContext in Spring.**
14. **Explain the concept of circular dependency in Spring and how to resolve it.**
15. **What is Spring IoC container?**
16. **How to implement a RESTful controller in Spring, e.g., save user to database?**
17. **Write a unit test for a Spring controller.**
18. **What are the differences between BeanFactory and ApplicationContext in Spring?**
19. **Explain how to deal with circular dependencies in Spring.**
20. **How to use dependency injection in Spring?**
21. **Explain the Singleton pattern in Spring.**
22. **Explain how to use @Autowired in Spring.**
23. **What are annotations used in Spring?**
24. **Explain how to solve circular dependency problem in Spring.**
25. **Explain what is JWT and its authentication process in Spring Security.**
26. **Explain the differences between BeanFactory and ApplicationContext in Spring.**
27. **What’s SSL**
28. **What is Private key and public Key?**
29. **How do you handle bid data?**
30. **How to use Spring Batch if you have 5 millions of data from ElasticSearch and Oracle**
31. **Tell me the steps to set up Spring batch in your application**
32. **Write unit test to your service layer? What is Mockito? What is @InjectMock**
33. **Define a composite key class(@Embeddable)**
34. **Crud Repo vs JPA Repo**

### **三、Microservices & Cloud (AWS, Docker, Kubernetes)**

1. **How to connect a VPC to S3 in AWS?  
   To connect a VPC to S3 in AWS, you can use a VPC Endpoint:**

Create a VPC Endpoint:

* + Go to the AWS Management Console.
  + Navigate to VPC and then Endpoints.
  + Click Create Endpoint.
  + Choose the S3 service from the list of available services.
  + Select the VPC where you want to create the endpoint.
  + Choose the Route Table for your VPC to route traffic to the endpoint.
  + Configure the endpoint policy if you need to restrict access to specific S3 buckets.
  + Create the endpoint.

Route Traffic Through the VPC Endpoint:

* + Ensure that the route table associated with your subnets in the VPC has a route pointing to the VPC endpoint.

Access S3 from the VPC:

* + You can now access S3 directly from your VPC without going over the public internet.

1. **How to deal with batch jobs in AWS without using AWS Batch?**

**You can handle batch jobs in AWS without using AWS Batch by using services like AWS** Lambda, AWS Step Functions, or EC2 Spot Instances:

AWS Lambda:

* 1. Scheduled Execution: Use Amazon CloudWatch Events (now EventBridge) to schedule Lambda functions to run batch jobs periodically.
  2. Triggered Execution: Set up S3, SNS, or DynamoDB triggers to run Lambda functions as needed.

AWS Step Functions:

* 1. Orchestration: Use Step Functions to coordinate Lambda functions or ECS tasks for more complex batch workflows. You can define states, transitions, and parallel executions to manage your batch jobs.

EC2 Spot Instances:

* 1. Cost-Effective Processing: Use EC2 Spot Instances for running batch jobs, especially for workloads that are fault-tolerant and can handle interruptions. Use Auto Scaling groups to manage the fleet of Spot Instances.

Amazon SQS:

* 1. Queue-Based Processing: Use SQS to queue tasks and then process them using Lambda functions or EC2 instances.

1. **Explain the Elastic Block Store (EBS) in AWS.**Amazon Elastic Block Store (EBS) is a high-performance block storage service designed for use with Amazon EC2 instances. Key features include:

**Persistent Storage**:

* EBS provides persistent storage that can be attached to EC2 instances. Data on an EBS volume persists independently of the life of the instance.

**Types of EBS Volumes**:

* **General Purpose SSD (gp2, gp3)**: Balanced performance for most workloads.
* **Provisioned IOPS SSD (io1, io2)**: High-performance storage for I/O-intensive applications.
* **Throughput Optimized HDD (st1)**: Low-cost HDD for frequently accessed, throughput-intensive workloads.
* **Cold HDD (sc1)**: Low-cost HDD for less frequently accessed data.

**Snapshots**:

* EBS volumes can be backed up to Amazon S3 by creating snapshots, which are incremental backups.

**Encryption**:

* EBS supports encryption of data at rest, in transit, and during snapshot creation, using AWS KMS.

**Resizing**:

* EBS volumes can be resized on the fly without detaching them from the instance.

1. **What are the best practices in microservices architecture?**Decentralized Data Management:

* Each microservice should have its own database to avoid coupling. Use patterns like Event Sourcing and CQRS for complex data management scenarios.

APIs as Contracts:

* Define APIs as contracts and version them to ensure backward compatibility. Use API Gateway for routing and managing APIs.

Service Discovery:

* Implement service discovery to dynamically find the network location of microservices. Use tools like Eureka, Consul, or Zookeeper.

Resilience and Fault Tolerance:

* Implement circuit breakers, retries, timeouts, and bulkheads to make services resilient to failures. Use libraries like Hystrix or Resilience4j.

Monitoring and Logging:

* Implement centralized logging (e.g., using ELK Stack) and distributed tracing (e.g., Jaeger, Zipkin) to monitor microservices.

Security:

* Implement security at every layer. Use OAuth 2.0, JWT for authentication, and enforce authorization policies at the API Gateway.

CI/CD:

* Automate testing, deployment, and rollback with CI/CD pipelines to ensure fast and reliable delivery of changes.

1. **What are the best practices in AWS EC2?**Use the Right Instance Type:

* Choose instance types that best match your workload requirements. Use Spot Instances for cost savings where applicable.

Enable Auto Scaling:

* Set up Auto Scaling groups to automatically adjust the number of EC2 instances based on demand.

Monitor and Optimize Costs:

* Use CloudWatch to monitor performance and set up billing alerts. Optimize costs by using Reserved Instances or Spot Instances.

Security Best Practices:

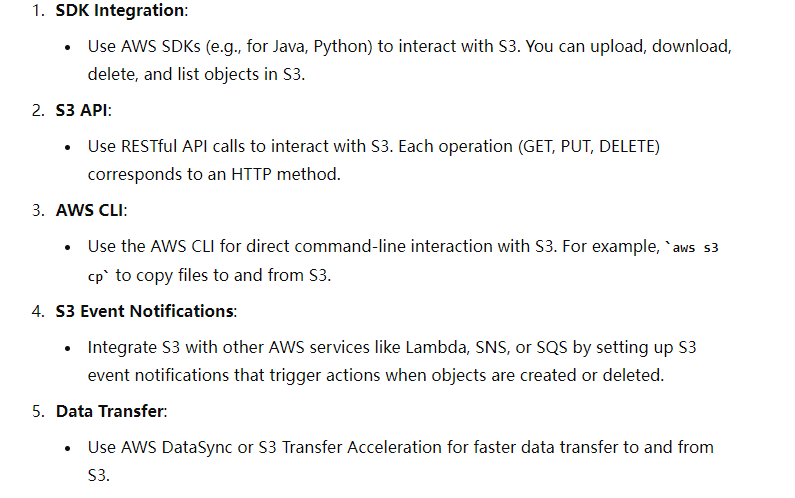
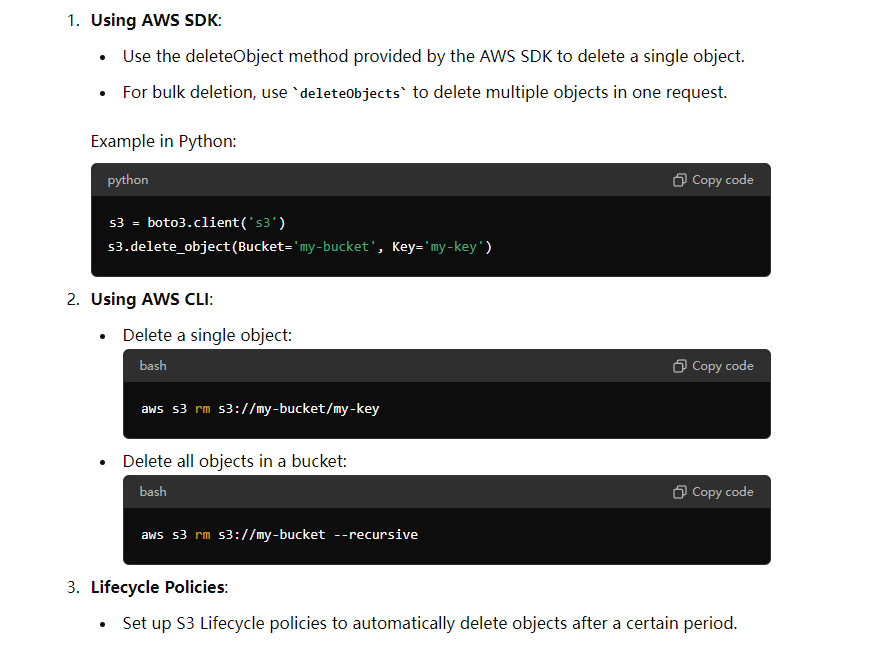
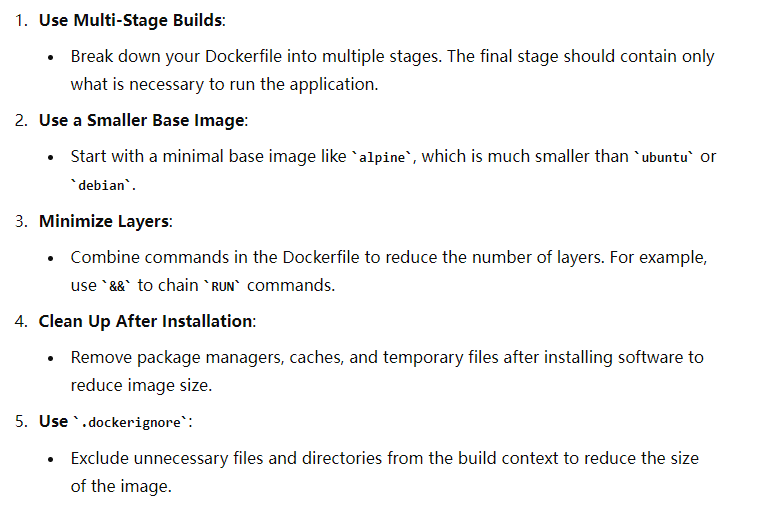
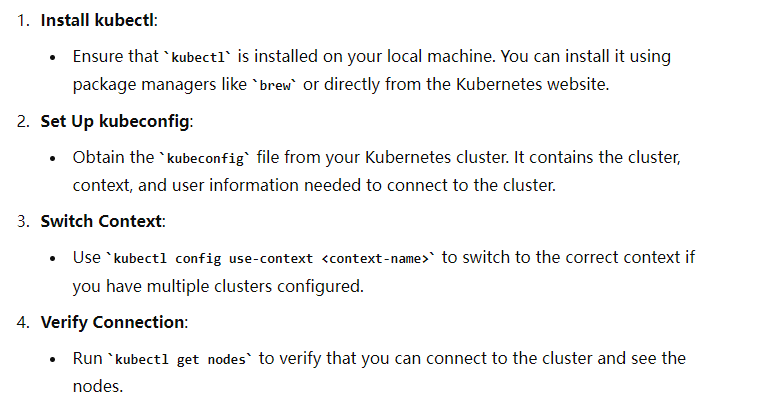
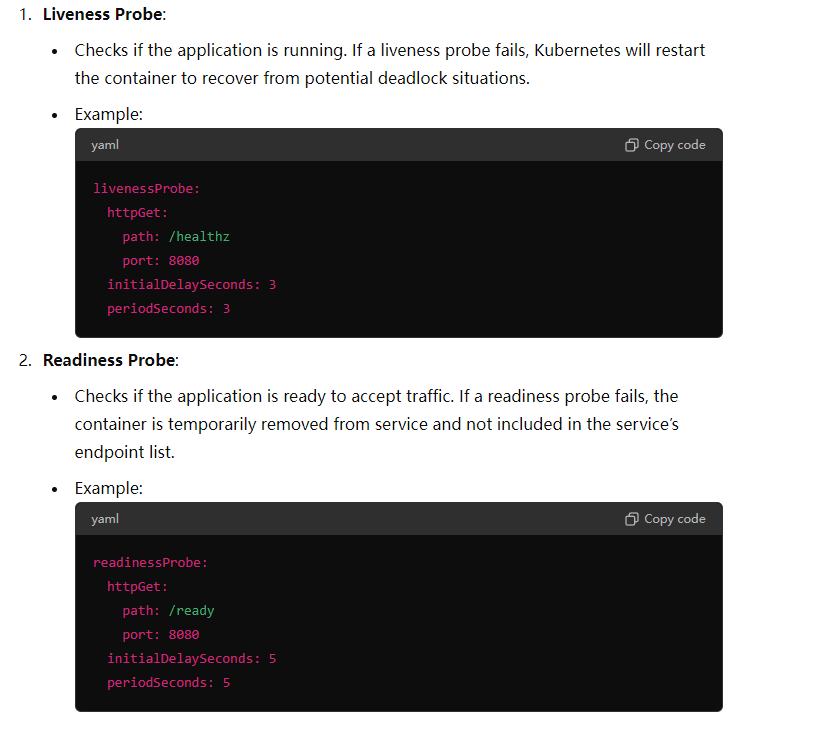
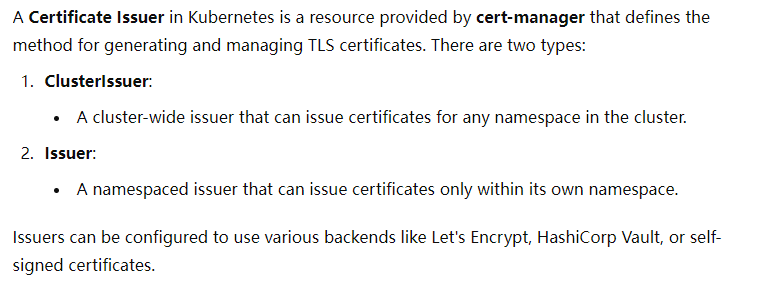
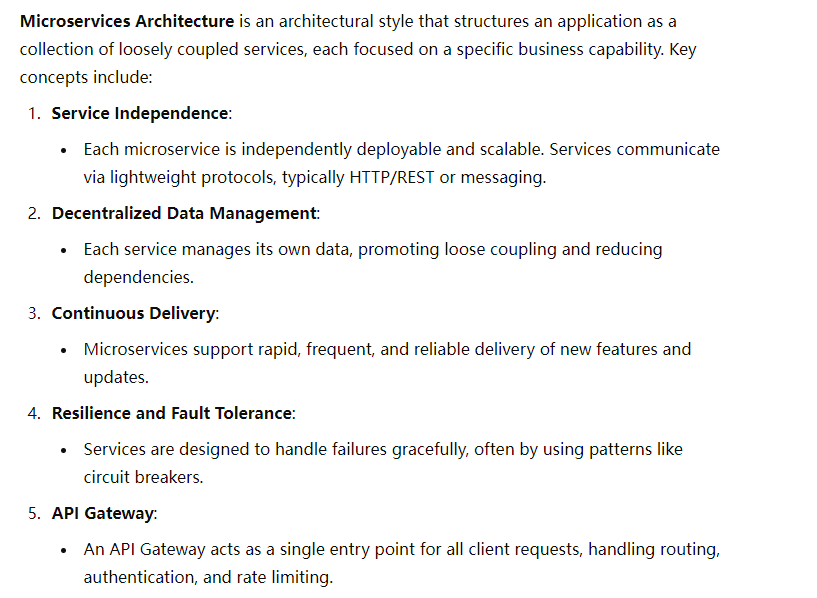
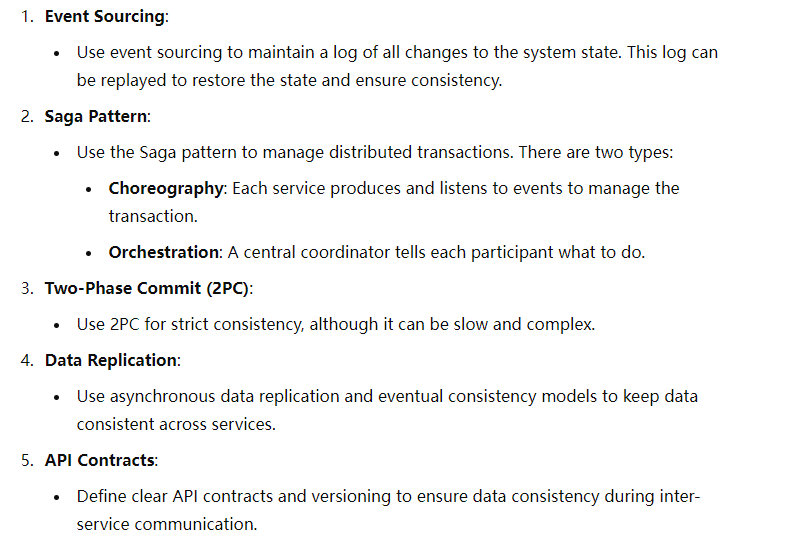
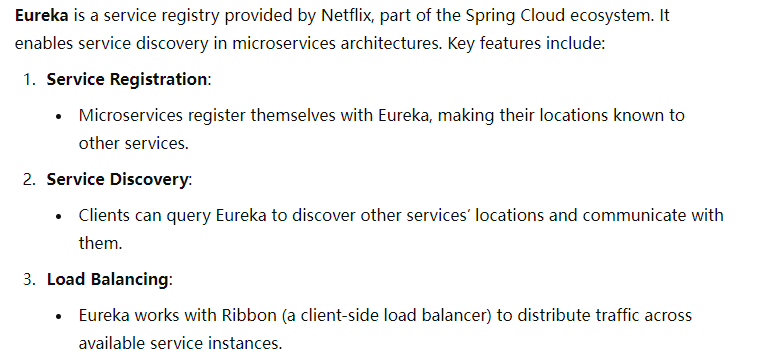
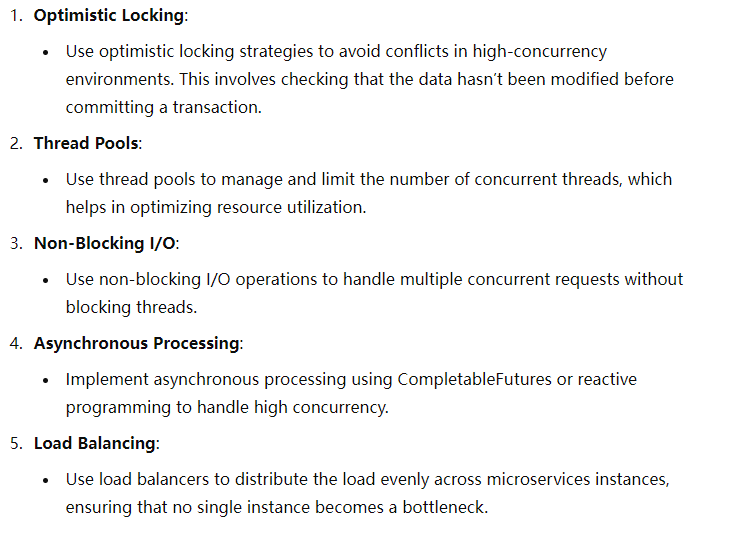
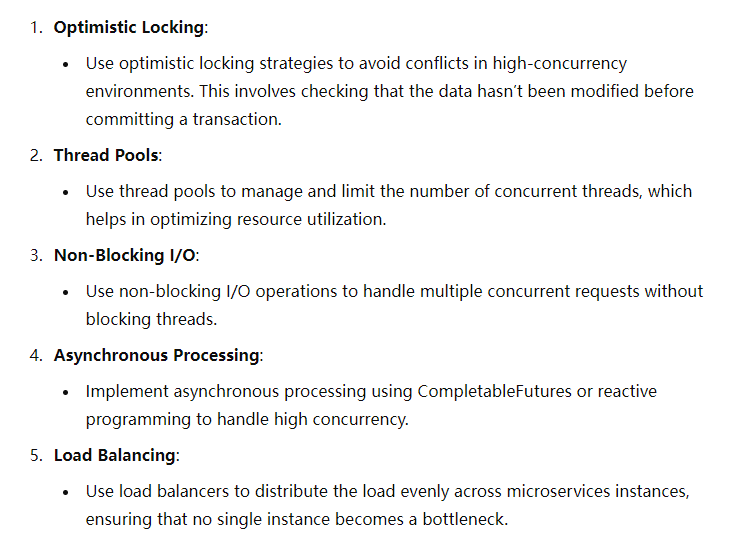
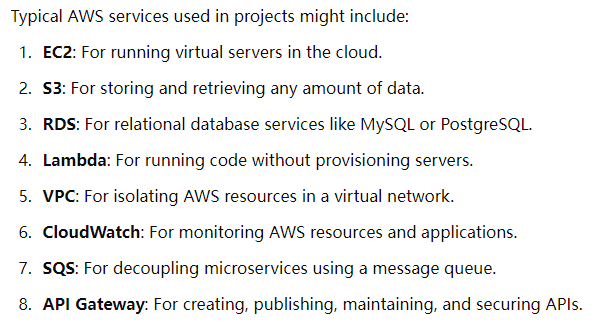
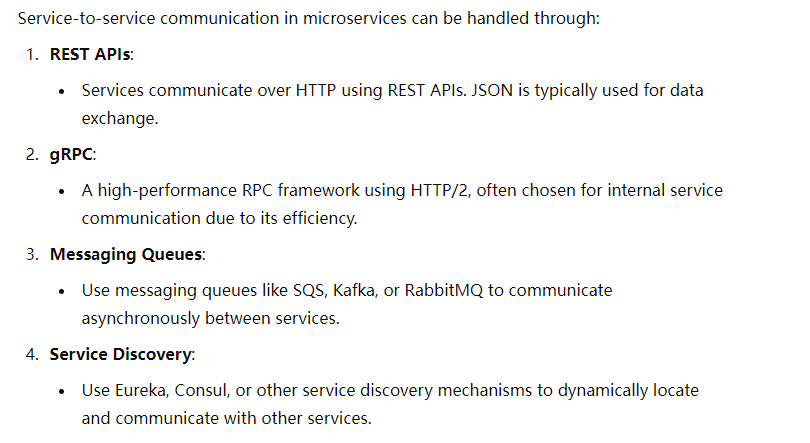
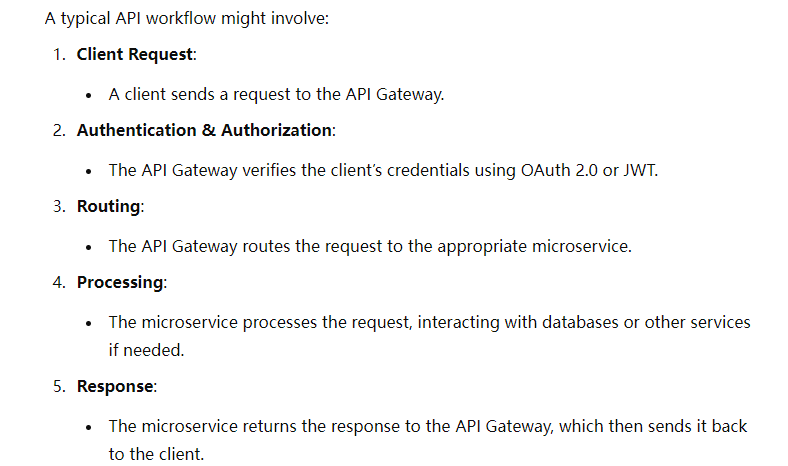
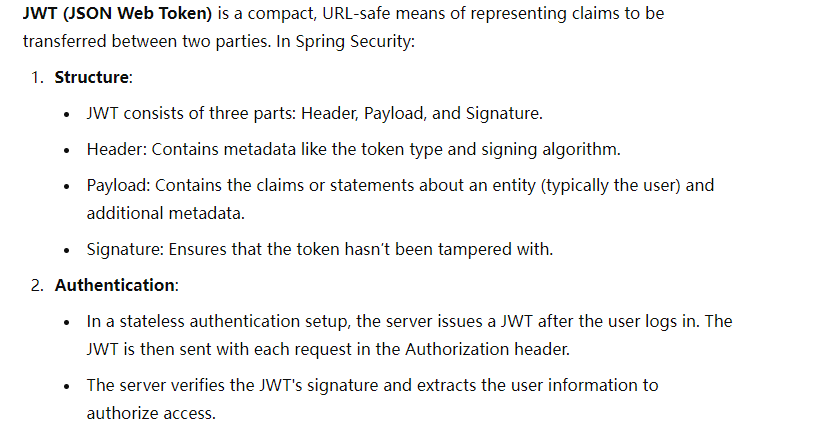
* Use IAM roles instead of hardcoding credentials.
* Enable VPC, security groups, and network ACLs to control inbound and outbound traffic.
* Encrypt EBS volumes and ensure all sensitive data is encrypted.

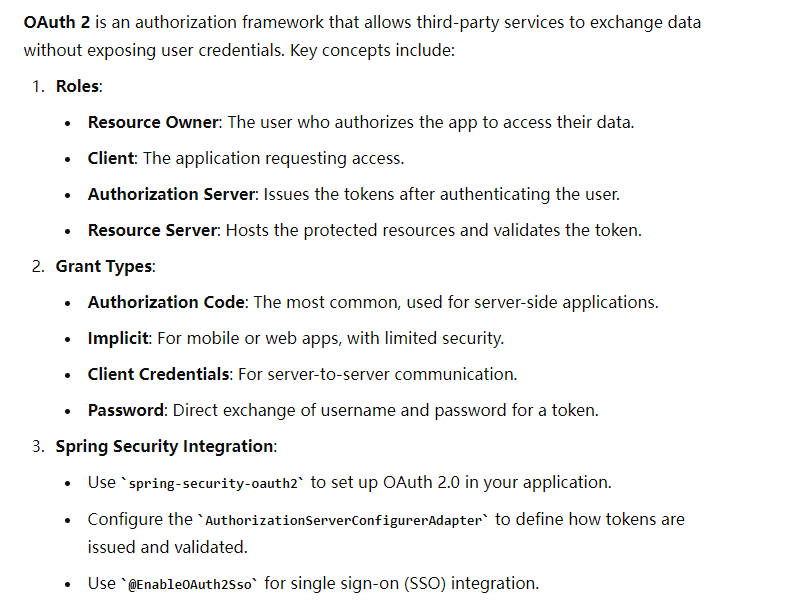
Backup and Recovery:

* Regularly back up your EBS volumes using snapshots.
* Use AMIs (Amazon Machine Images) to create backups of your EC2 instances.

High Availability:

* Distribute instances across multiple Availability Zones (AZs) for fault tolerance.

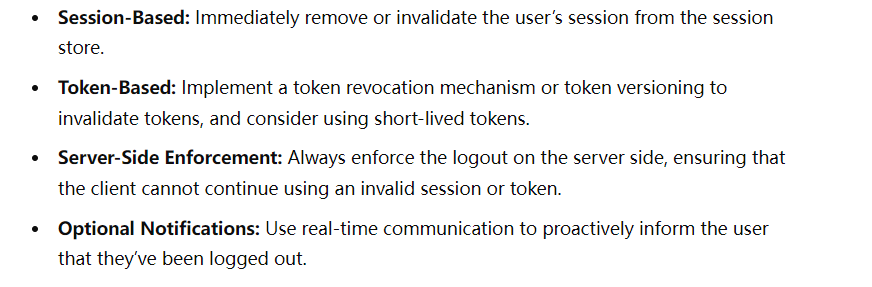
1. **How to integrate with S3 in AWS?  
   **
2. **How to delete data from S3 in AWS?  
   **
3. **How to optimize Docker images' size?  
   **
4. **How to connect to a Kubernetes cluster from local?  
   **
5. **Explain Liveness Probe vs Readiness Probe in Kubernetes.  
   **
6. **What is a Certificate Issuer in Kubernetes?  
   **
7. **How to explain Microservices architecture?  
   **
8. **Explain how to deal with data consistency between teams in a microservices environment.  
   **
9. **Explain what is Eureka in Spring Cloud.  
   **
10. **Explain best practices in microservices architecture.  
    **
11. **How to handle high concurrency in Java within a microservices architecture?  
    **
12. **What AWS services did you use in your project?  
    **
13. **How to communicate with other services in your project?  
    **
14. **Explain the workflow of your API.  
    **
15. **Explain what is JWT in Spring Security.  
    **
16. **Explain what is OAuth 2 and how to use it in Spring Security.**

****

1. **How nodes communicate in k8s**Kubelet   
   Kube-proxy  
   Pod-to-pod communication   
   service communication   
   node-to-node communication
2. **What is the server of k8s**Control plane  
   API server Etcd Controller manager Scheduler cloud Controller Manager
3. **Is Kafka used for one-to-one communication or broadcasting  
   Both**
4. **When an admin deletes a user, the user will not log out immediately. How to solve that? (This is a huge question)**
5. **What is a Circuit Breaker and how to implement it?**

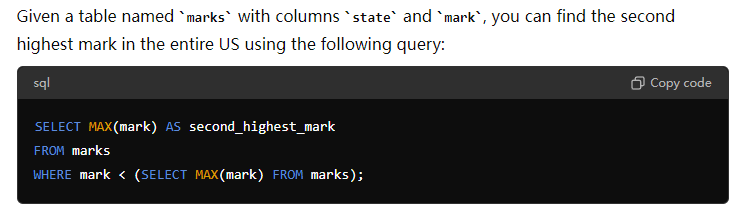
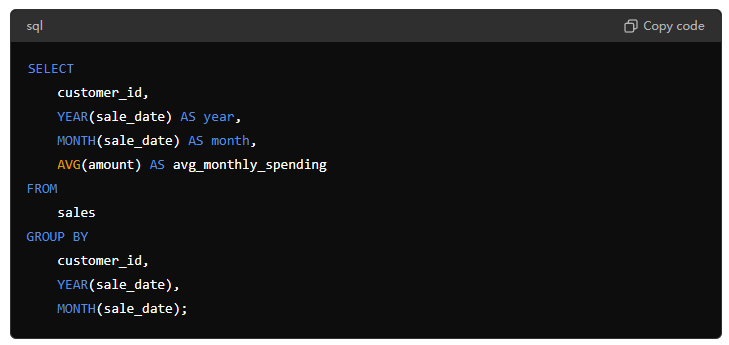
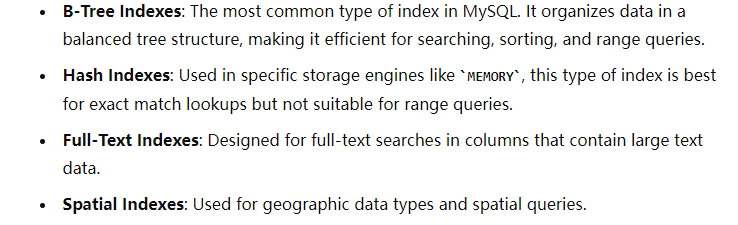
Detect failures and encapsulate the logic of preventing a failure   
Closed state open state half- open state

Resilience4J



1. **What is the Zuul Gateway API?**A open-source gateway by Netflix  
   Routing(reverse proxy) Load Balancing Security Filter Mechanism Rate Limiting Analytics and Monitoring
2. **how does service registry know if there is a new microservice got created**Service Startup-> Registration-> HeartBeat/Health Checks-> Service Discovery by other Services-> Deregistration

### **四、SQL & 数据库**

1. **Given a table with states and marks, how do you get the second highest mark in the entire US?  
   **
2. **SQL: Calculate the average monthly spending per customer from a sales table.  
   **
3. **SQL: Get the employees who joined more than 1 department.**
4. **Write a query to find the highest paid employee in each department.**
5. **Explain how to implement JDBC in Spring.**Add dependency -> configure connection application.properties or application.yml ->  
   Model Repository Service Controller
6. **Explain the use of indexes in MySQL and when to use them.  
   **
7. **Explain what is a Dead Letter Queue in a database context.**To store messages or records that cannot be processed or delivered successfully.

### **五、系统设计 & 架构**

1. **What are system design aspects to consider?**
2. **Explain design patterns in Spring.**
3. **Explain the Factory design pattern in Java.**
4. **Explain the Observer design pattern in Java.**
5. **What is the Command design pattern in Java?**
6. **Explain how to design a JPA application using a 3-tier model.**
7. **Explain how to deal with circular dependency in a large system design.**
8. **Explain design patterns in Microservices**
9. **How to design a system to invoke three different microservices for user registration with failure recovery?**

Design a system that invokes three different microservices (HR, Admin, Network) to register an employee. Once the employee is successfully registered in all three departments, the system should send a unique employee ID to the employee via email. If an API call fails at any point, the system should remember where it failed. The next time the same employee tries to onboard, the system should resume from the point of failure. For example, if the registration succeeds with Admin but fails with Network, the next attempt should start with Network, not Admin.

1. **What is Test-Driven Development(TTD)  
   Test-Driven Development (TDD)** is a software development approach in which tests are written before writing the actual code. TDD is part of the agile software development process and emphasizes writing automated test cases for new features or functionality before the code itself is implemented.
2. **What are the drawbacks of a system tracking the state of each record for microservice registration?**Tracking the state of each record for microservice registration in a service registry comes with several potential drawbacks. These drawbacks primarily relate to increased complexity, scalability challenges, performance issues, and potential reliability concerns. Here’s a detailed explanation:
3. **how to realize the rollback between 4 services in microservice**

**Service A**: Payment Service

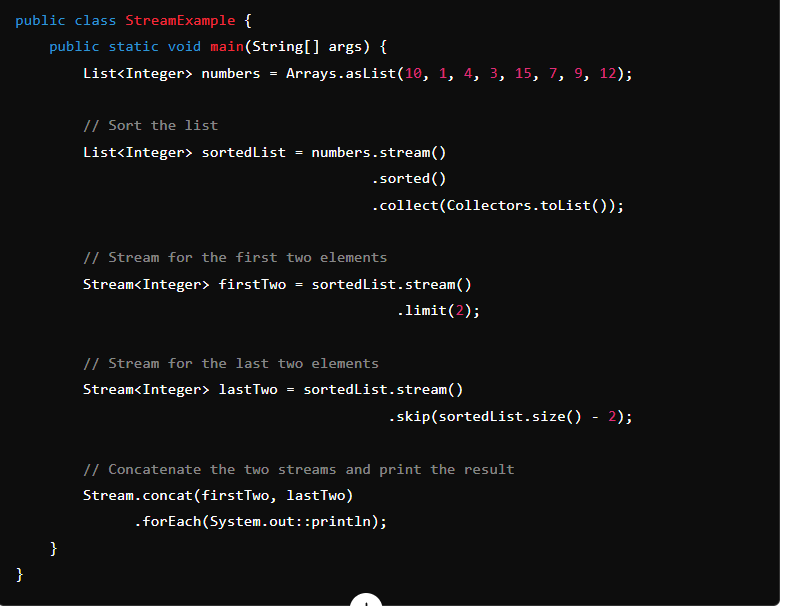
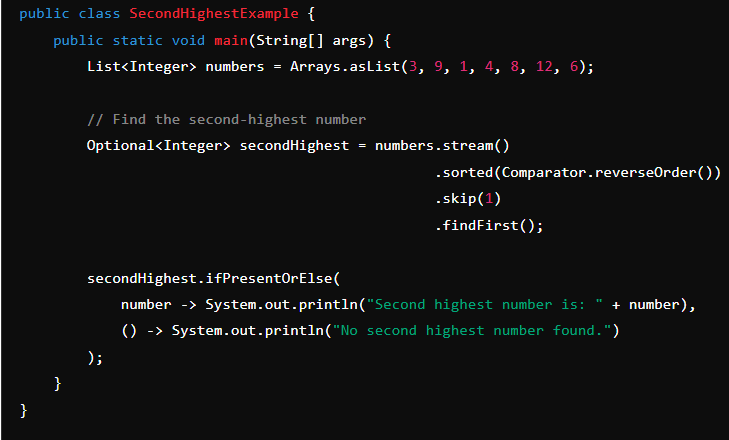
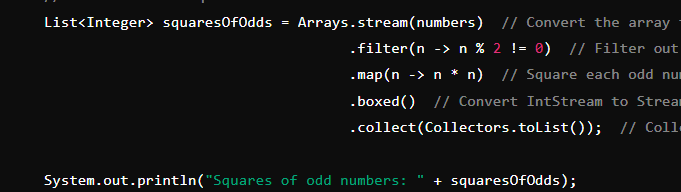
**Service B**: Inventory Service

**Service C**: Shipping Service

**Service D**: Notification Service

Saga pattern orchestration

### **六、编码 & 算法**

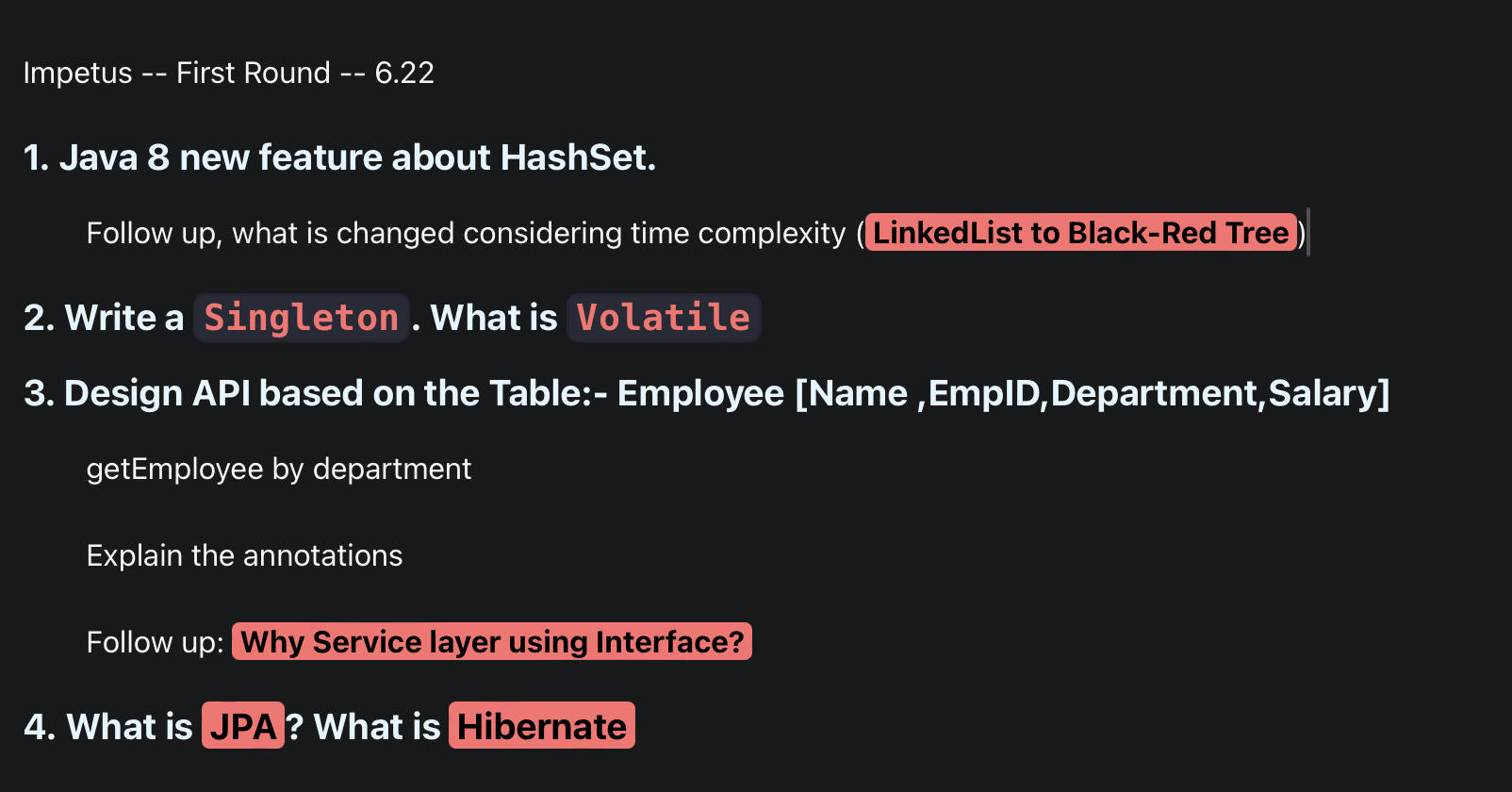
1. **Use Stream to count the number of words in a sentence.**java.util.Arrays.stream(sentence.split("\\s+")) // Split the sentence by whitespace .filter(word -> !word.isEmpty()) // Filter out any empty strings (in case of multiple spaces) .count();
2. **Use Stream to print the first two and last two elements in a sorted list.  
   **
3. **Use Streams to find the second-highest number in a list.  
   **
4. **use stream api and lambda to get the sum of all the odd numbers in an array**int sumOfOdds = Arrays.stream(numbers) // Convert the array to an IntStream  
    .filter(n -> n % 2 != 0) // Filter out even numbers, keeping only odd numbers   
   .sum(); // Sum the remaining (odd) numbers
5. **use stream api and lambda to get the square of all the odd numbers in an array  
   **
6. **Write a REST controller that consumes a User Payload and saves it to the database.**
7. **Write a method that takes two lists of integers and a lambda expression as parameters, and returns a new list containing the elements that are common to both lists, based on the condition specified by the lambda expression.  
   public static void main(String[] args) {**
8. **List<Integer> list1 = List.*of*(1, 2, 3, 4, 5);**
9. **List<Integer> list2 = List.*of*(3, 4, 5, 6, 7);**
10. **// Example lambda: checks if two numbers are equal**
11. **BiPredicate<Integer, Integer> condition = (a, b) -> a.equals(b);**
12. **List<Integer> commonElements = *findCommonElements*(list1, list2, condition);**
13. **System.*out*.println("Common elements: " + commonElements);**
14. **}**
15. **public static List<Integer> findCommonElements(List<Integer> list1, List<Integer> list2, BiPredicate<Integer, Integer> condition) {**
16. **return list1.stream()**
17. **.filter(element1 -> list2.stream().anyMatch(element2 -> condition.test(element1, element2)))**
18. **.collect(Collectors.*toList*());**
19. **}**
20. **Write a Java program to simulate a simple producer-consumer scenario using multithreading.**

Two classes producer consumer using queue  
Producer has a parameter called capacity

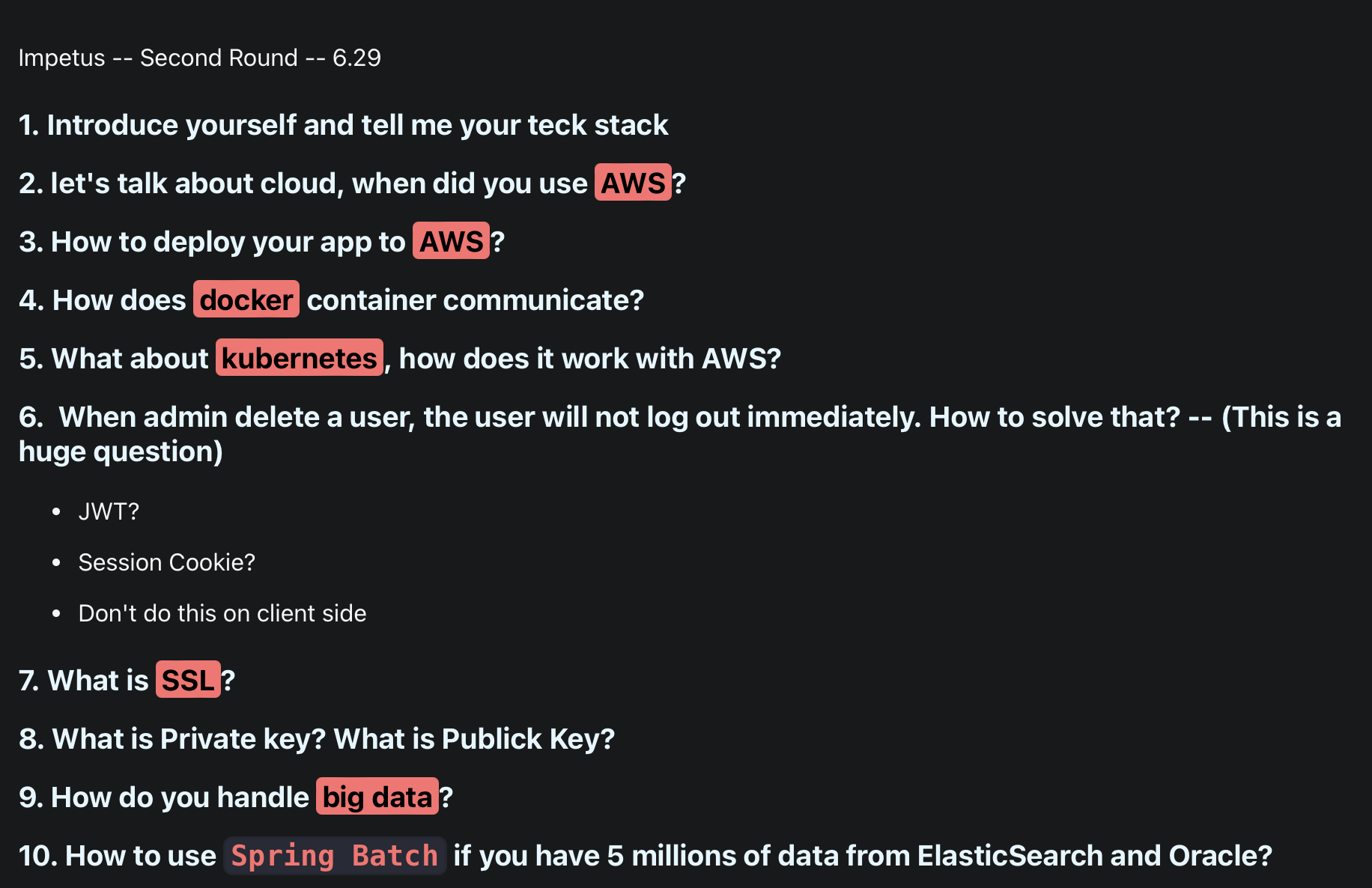
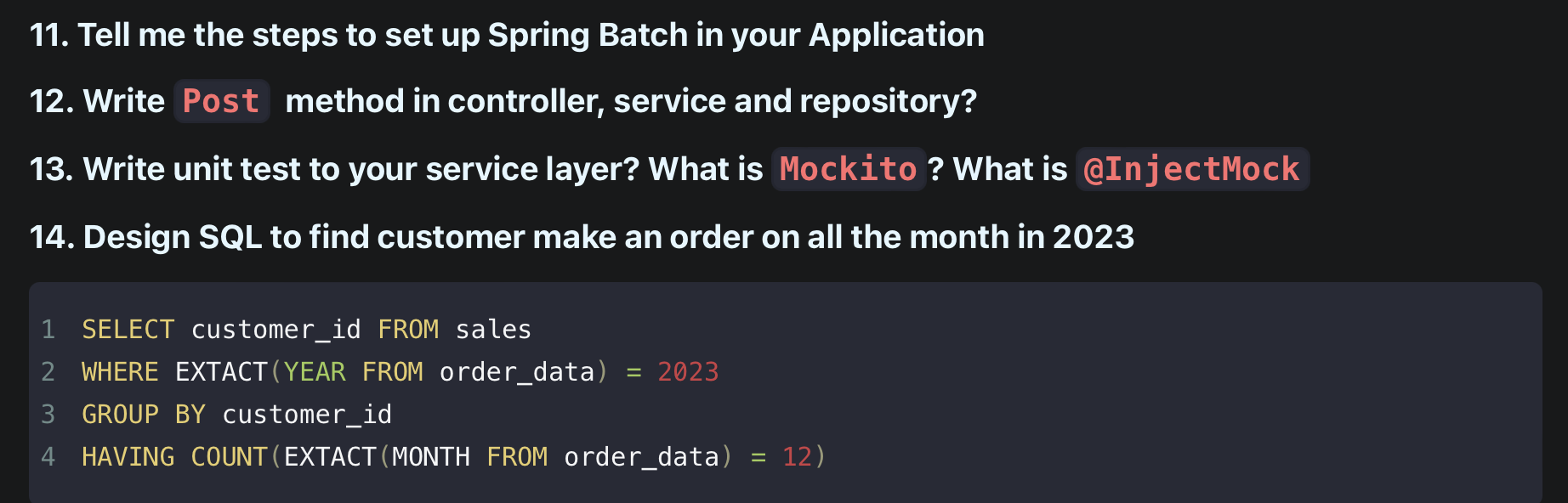
1. **Write code to rotate an array to the right.**
2. **Write a method to left rotate an integer number. For example, 123456 left rotated 2 times -> 345612.**
3. **Write code for the Singleton pattern in Java.**
4. **Explain the use of recursion to generate Fibonacci sequence in Java.**
5. **How to use Streams to count the number of words in a sentence in Java?**
6. **Given a String of Integer, sort it, and print it out(Stream/ Non-Stream way)  
   String input = "43215";**String sortedString = input.chars() // Convert the string to an IntStream of character .mapToObj(c -> (char) c) // Convert each character code back to a character   
   .sorted() // Sort the stream of characters   
   .map(String::valueOf) // Convert each character  
   .collect(Collectors.joining()); // Join all strings back into one sorted string
7. **how to use flatmap to get the numbers >= 40 in an array of array. What is boxed() in your code?**
8. **Longest Palindromic Substring. Implement Brutal-force approach. Give Dynamic Programming Idea**
9. **Design API based on the Table:Employee[Name, EmpId, department, Salary]  
   getEmployee by department explain the annotations   
   Follow up: Why Service layer using Interface**MODEL Repository Service Controller
10. **Java program to convert file to byte array**readAllBytes()
11. **Setting Up Spark for Real-Time Processing**
12. **Storing Data to a Database (Using JDBC with PostgreSQL)**
13. **given a string array {"abcd", "abc”, "ab", "a"}, return a4b3c2d1 with O(1).  
    Int[] 26**
14. **Coding:**

**System.out.println("Call me async"); print it concurrently 5 times**

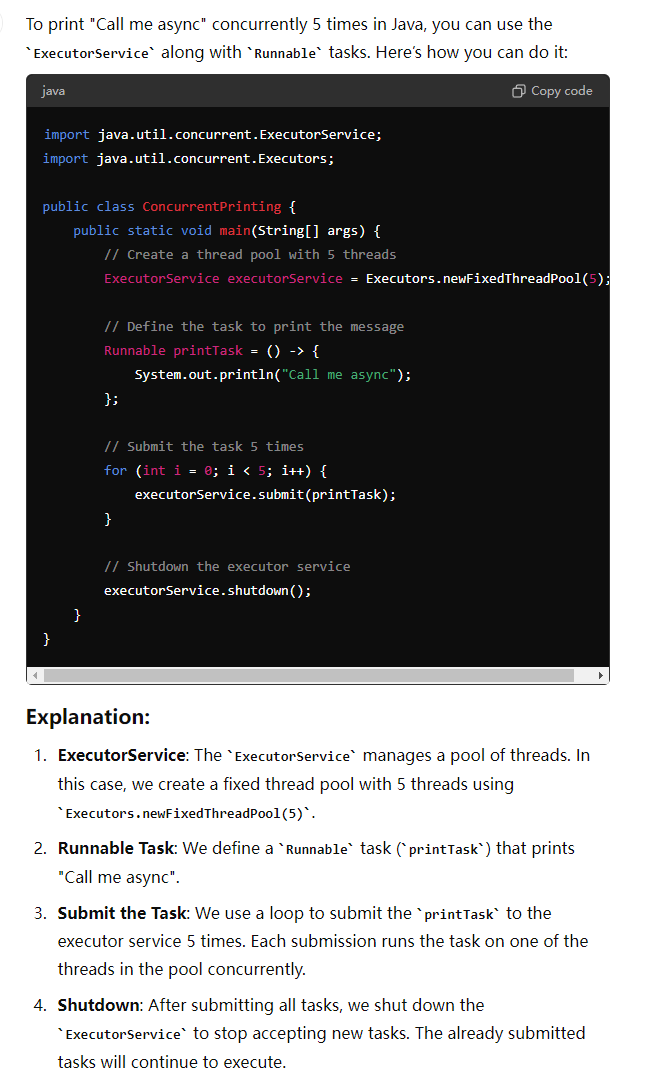
### **七、详细补充答案**

1. **What kind of security have you adopted in your project?**
2. **What is JWT?**
3. **What is a functional interface? Write a calculator as a example**
4. **What is CompletableFuture? Rewrite the code to async execute it**
5. **Benefits of abstraction**
6. **Why are we hiding the implementation? What is the use of hiding? From whom we are hiding?**
7. **Why decouple? Why is the interface in not tight couple?**
8. **why service interface is loosely coupled? Why loosely couple?**
9. **Java 17 new features**
10. **Java 8 new features**
11. **Why introduce default and static method?**
12. **What is method reference?**
13. **What is functional interface? Why only one abstract method?**
14. **map vs flatMap in stream api**
15. **Real time use case of flatmap**
16. **What is DP?**
17. **How hashmap works internally? W  
    what is equals() what is hashcode() ?**
18. **Thread life cycle**
19. **Why string immutable?**
20. **how to create immutable object? Use final keyword on class and perform deep copies of Mutable Fields.**
21. **Checked Exception vs Unchecked Exception? Compiler knows the Checked exception at compile time?**  
    **Introduce yourself, the recent project, your responsibility**
22. **Java 17 new feature**
23. **JAVA remove aot, jit, why**
24. **How Java internaly work? JVM?**
25. **How memory located? What type of memory in Java?**
26. **Why Java 8 remove pattern generation?**
27. **Young generation -- `New`**
28. **Survival generation**
29. **Old generation**
30. **How GC work in those generation**
31. **Which gc is responsible for remove the garbage from one to another one?**
32. **How latency impact based on choice of gc?**
33. **Two types of minor gc, explain which situation use which gc?**
34. **What is dangling pointer problem? How to avoid or solve.**
35. **Why string immutable? Advantage of that?**
36. **How create yourself immutable object?**

**e.g. employee object, has a reference of address, they can change address, how to stop immutability breakable**

1. **how iternally hashmap works in java**
2. **Lifecycle of thread  
   Three coding:**
3. **Java program to convert file to byte array**
4. **Setting Up Spark for Real-Time Processing**
5. **Storing Data to a Database (Using JDBC with PostgreSQL)**
6. **`notify` vs `notifyall`**  
   
7. **Coding:**

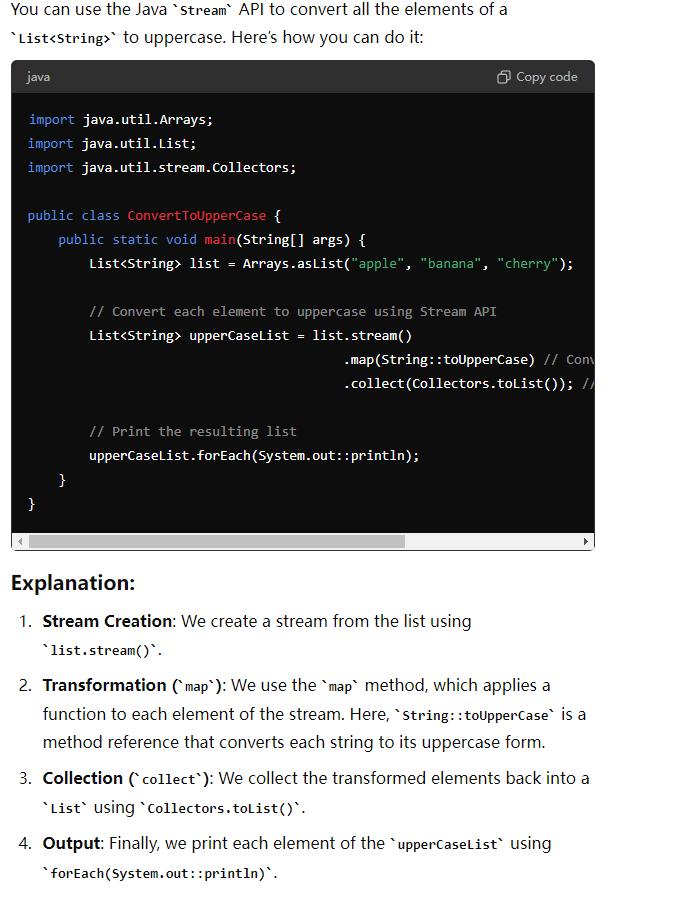
**System.out.println("Call me async"); print it concurrently 5 times**



1. **Usage of Future**
2. **Coding:**

Use Stream, to make the following list uppercase

List<String>list=Arrays.asList("apple","banana","cherry");



1. **stream vs parallel stream**  
   In Java, the Stream API provides a powerful way to process collections of data in a functional style. There are two main types of streams: **sequential streams** and **parallel streams**. Here's a comparison between the two:

### **1. Sequential Stream**

* **Execution**: A sequential stream processes elements in a single thread, one after the other, in the order they appear in the source (e.g., a list or an array).
* **Performance**: Sequential streams are straightforward and often perform better for smaller datasets or when the operation is inherently sequential (e.g., when the order of processing is important).

1. **Usage**: A sequential stream is the default stream type in Java. You create a sequential stream with stream().  
   java  
   Copy code  
   List<String> list = Arrays.asList("apple", "banana", "cherry");

list.stream()

.map(String::toUpperCase)

.forEach(System.out::println);

* **When to Use**: Use sequential streams when touerhe tasks are simple, the dataset is small, or the operations need to be executed in order.

### **2. Parallel Stream**

* **Execution**: A parallel stream divides the stream into multiple substreams and processes them in parallel using multiple threads. Java's ForkJoinPool is used internally to achieve parallelism.
* **Performance**: Parallel streams can improve performance for large datasets or computationally intensive operations by leveraging multiple CPU cores. However, there is an overhead associated with parallelism, so for small datasets, the performance might be worse than a sequential stream.

1. **Usage**: You create a parallel stream with parallelStream() or by calling parallel() on a sequential stream.  
   java  
   Copy code  
   List<String> list = Arrays.asList("apple", "banana", "cherry");
2. list.parallelStream()
3. .map(String::toUpperCase)
4. .forEach(System.out::println);
5. **Order of Execution**: Parallel streams do not guarantee the order of execution, so the results might appear in a different order than the input. However, methods like forEachOrdered can be used to maintain order if necessary.  
   java  
   Copy code  
   list.parallelStream()
6. .map(String::toUpperCase)
7. .forEachOrdered(System.out::println);

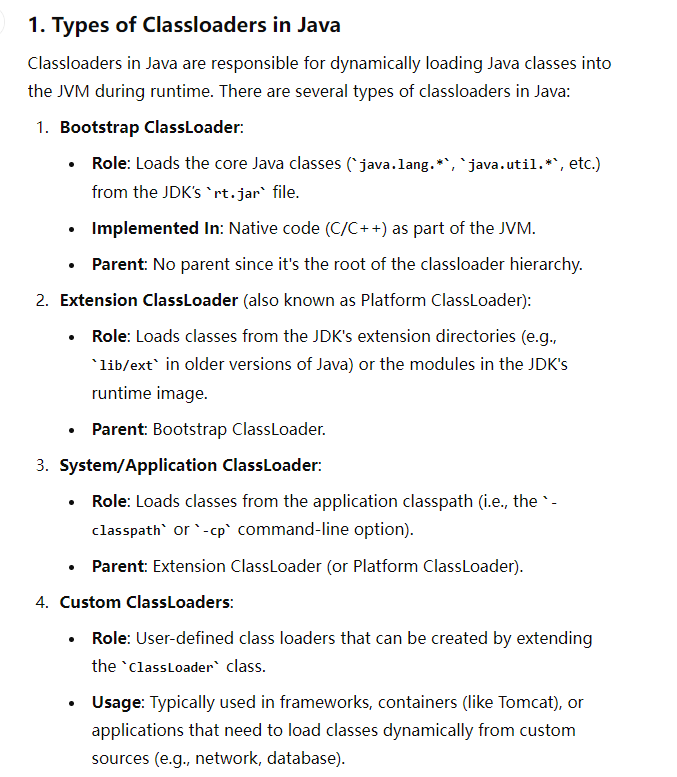
* **When to Use**: Use parallel streams when working with large datasets, when operations are independent of the order, or when the operations are computationally expensive and can benefit from parallel processing.

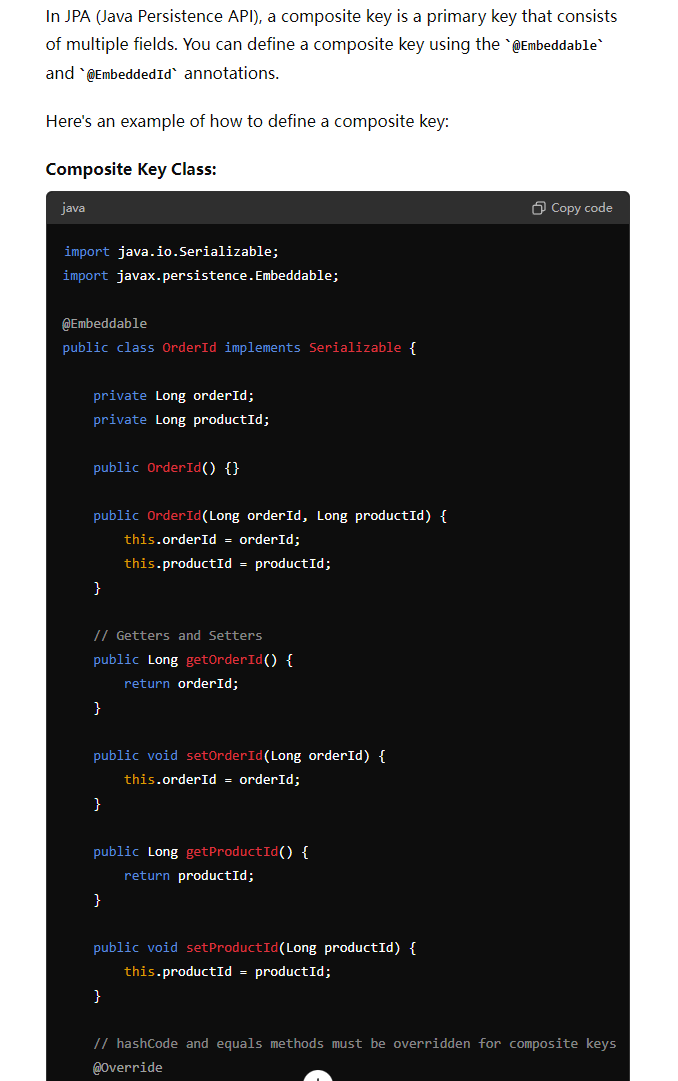
### **Key Considerations:**

* **Thread Safety**: Operations on parallel streams must be thread-safe since multiple threads are involved. If the operation modifies shared state, it could lead to race conditions.
* **Overhead**: Parallel streams introduce overhead due to the need to split tasks and manage threads. For small datasets or lightweight operations, this overhead can outweigh the benefits.
* **Order**: If the order of operations matters, sequential streams are usually a safer choice. Parallel streams can lead to out-of-order processing unless explicitly controlled.
* **Complexity**: Parallel streams are easier to implement than manual multithreading but require careful consideration to avoid pitfalls like deadlocks, race conditions, or improper use of shared resources.

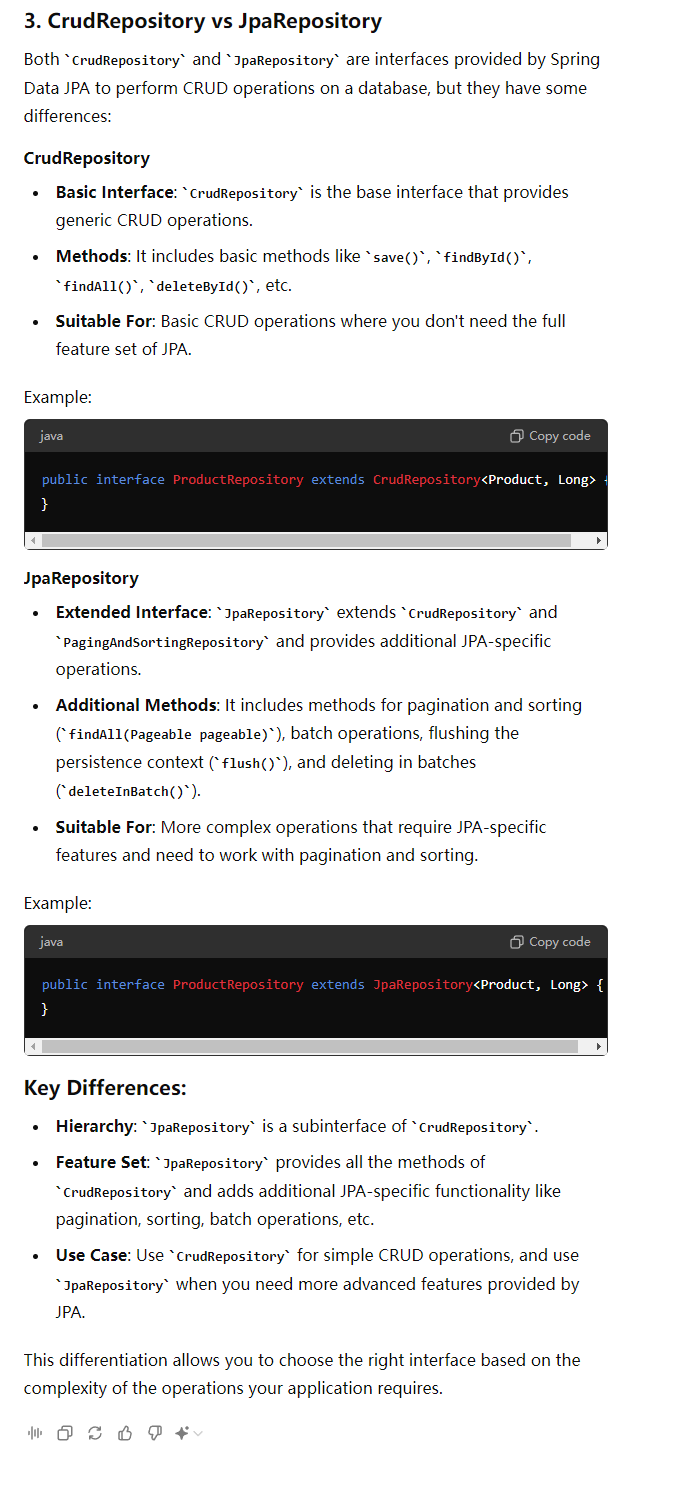
### **Summary**

* **Sequential Stream**: Use for smaller datasets, when order matters, or when operations are simple.
* **Parallel Stream**: Use for large datasets, computationally heavy operations, or when order does not matter and parallelism can provide performance benefits.

1. **map vs flatmap**
2. **Create a springboot project, including payload, entity, Repository interface, Service, RestController**
3. **Create a test class for the service**
4. **What is the Command Pattern and Observer Pattern?**
5. **SQL: Given a `sales` table with columns `order\_id`, `product\_id`, `customer\_id`, and `order\_date`, write a query to find customers who have placed orders in every month of the year 2023.  
   Springboot and SpringCloud version you use**
6. **Hashmap, hard hashmap, soft hashmap**
7. **wait vs sleep, how to use notify()**
8. **what is interrupt**
9. **pesudo coding：merge two sorted list**
10. **arraylist and auto scaling mechanism, vs linked list**
11. **comparable and comparator**
12. **marker interface**
13. **interface vs abstract class**
14. **java 11 new feature**
15. **Type of Garbage Collector**
16. **When we want an object as a key in hashmap, what should the object have?(the obj must have equal and hashcode method)**
17. **Using stream to to group the employee into different department**
18. **Using stream to sort the employee based on salary**
19. **Executor framework, callable and future**
20. **Concurrent hashmap vs Synchronized hashmap**
21. **Type of Classloader**
22. 
23. Define a composite key class(@Embeddable)





1. Crud Repo vs JPA Repo  
     
   
2. DI
3. Bean Scope
4. Is Singleton bean scope thread safe? how to make it thread safe?  
   In Spring, a bean defined with a singleton scope means that there is only one instance of that bean per Spring IoC container. While singleton beans are shared across multiple threads, they are **not** inherently thread-safe. Thread safety is not guaranteed by the Spring framework itself.

### **Why Singleton Beans May Not Be Thread-Safe**

If your singleton bean holds mutable state (e.g., instance variables that can be modified), multiple threads accessing this shared state simultaneously could lead to concurrency issues such as race conditions, inconsistent data, and unpredictable behavior.

For example, if your singleton bean has a counter that multiple threads increment, without proper synchronization, the counter could end up with an incorrect value due to race conditions.

### **How to Make Singleton Beans Thread-Safe**

1. **Avoid Mutable State**:
   * The simplest way to ensure thread safety is to avoid mutable state in singleton beans. If all the fields of your singleton bean are immutable (e.g., final), and the bean is stateless, it is inherently thread-safe.
2. **Use Thread-Local Variables**:
   * If the state needs to be specific to each thread, consider using ThreadLocal variables. This ensures that each thread has its own instance of the variable.

java  
Copy code  
public class MySingletonBean {

private ThreadLocal<Integer> threadLocalCounter = ThreadLocal.withInitial(() -> 0);

public void incrementCounter() {

threadLocalCounter.set(threadLocalCounter.get() + 1);

}

public int getCounter() {

return threadLocalCounter.get();

}

}

1. **Synchronized Methods/Blocks**:
   * You can synchronize access to critical sections of code where shared mutable state is modified.
2. java  
   Copy code  
   public class MySingletonBean {

private int counter = 0;

public synchronized void incrementCounter() {

counter++;

}

public synchronized int getCounter() {

return counter;

}

}

1. **Note**: Synchronized blocks or methods can lead to performance bottlenecks if used excessively, as they serialize access to the critical section.
2. **Use Atomic Classes**:
   * For simple operations on mutable state, Java's atomic classes (e.g., AtomicInteger, AtomicReference, etc.) from the java.util.concurrent.atomic package can be used.
3. java  
   Copy code  
   import java.util.concurrent.atomic.AtomicInteger;

public class MySingletonBean {

private AtomicInteger counter = new AtomicInteger(0);

public void incrementCounter() {

counter.incrementAndGet();

}

public int getCounter() {

return counter.get();

}

}

1. **Advantages**: Atomic classes provide lock-free thread-safe operations on single variables, which can be more performant than synchronized blocks.
2. **Using Locks**:
   * For more complex scenarios, where atomic operations are not sufficient, you can use explicit locks, like ReentrantLock.

java  
Copy code  
import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

public class MySingletonBean {

private int counter = 0;

private final Lock lock = new ReentrantLock();

public void incrementCounter() {

lock.lock();

try {

counter++;

} finally {

lock.unlock();

}

}

public int getCounter() {

lock.lock();

try {

return counter;

} finally {

lock.unlock();

}

}

}

### **Summary**

* **Singleton beans are not inherently thread-safe** in Spring.
* To make a singleton bean thread-safe, you should:
  + Avoid mutable shared state.
  + Use ThreadLocal for thread-specific data.
  + Synchronize access to critical sections.
  + Use atomic classes for simple state management.
  + Use explicit locks for complex concurrency control.

The appropriate approach depends on your specific use case and the complexity of the operations involved.

1. CI/CD Process (Using Jenkins)

Quaid Ali Bohra - 题意不明确；一直说no

第二轮：

1. What is the marker interface? What is the usage of marker interface? example of marker interface  
   A **marker interface** in Java is an interface that does not contain any methods or fields; it is empty. The primary purpose of a marker interface is to convey metadata to the Java runtime or other components about the nature of the implementing class.

### **Usage of Marker Interface**

Marker interfaces are used to indicate or "mark" a class with a special property or behavior. The runtime system, libraries, or frameworks can check whether a class implements a marker interface and then apply specific logic based on that.

### **Common Example of Marker Interface**

One of the most well-known examples of a marker interface in Java is the Serializable interface.

#### **Example: Serializable**

The Serializable interface is used to mark a class as capable of being serialized. Serialization is the process of converting an object into a byte stream, which can then be written to a file, sent over a network, or stored in a database. Deserialization is the reverse process, where the byte stream is converted back into an object.

Here’s an example:

java

Copy code

1. import java.io.Serializable;
2. public class Employee implements Serializable {
3. private static final long serialVersionUID = 1L;
5. private String name;
6. private int id;
7. // Constructor, getters, setters, etc.
9. public Employee(String name, int id) {
10. this.name = name;
11. this.id = id;
12. }
13. public String getName() {
14. return name;
15. }
16. public void setName(String name) {
17. this.name = name;
18. }
19. public int getId() {
20. return id;
21. }
22. public void setId(int id) {
23. this.id = id;
24. }
25. }

In the above example, by implementing Serializable, the Employee class is marked as being serializable. This means that instances of Employee can be serialized and deserialized. The actual logic for serialization is handled by the Java runtime, and the implementation of the Serializable interface serves as a marker to indicate that the class supports this capability.

### **How Marker Interfaces Work**

When a class implements a marker interface, the Java runtime (or a third-party framework) can check if an object is an instance of a class that implements that marker interface using the instanceof operator:

java

Copy code

1. Employee emp = new Employee("John Doe", 101);
2. if (emp instanceof Serializable) {
3. // This object can be serialized
4. }

### **Other Examples of Marker Interfaces**

1. **Cloneable**: Used to indicate that a class allows its objects to be cloned. If a class implements Cloneable, the Object.clone() method can be used to create a shallow copy of instances of the class. If Cloneable is not implemented, calling clone() will result in a CloneNotSupportedException.  
   java  
   Copy code  
   public class Employee implements Cloneable {
2. private String name;
3. private int id;
4. public Employee(String name, int id) {
5. this.name = name;
6. this.id = id;
7. }
8. @Override
9. protected Object clone() throws CloneNotSupportedException {
10. return super.clone();
11. }
12. }

* **Remote**: Used in Java RMI (Remote Method Invocation) to mark an interface as one whose methods can be called remotely.

### **Why Use Marker Interfaces?**

Marker interfaces are a way to give metadata to a class. They are simple to use and are often more readable and less error-prone than using annotations or other methods. While modern Java development often favors annotations for this kind of metadata, marker interfaces are still useful in certain contexts:

* **Legacy systems**: Older Java frameworks and libraries might rely on marker interfaces.
* **Semantic clarity**: Implementing a marker interface can make the purpose or capabilities of a class immediately obvious to other developers, purely from its type.

### **Summary**

* A **marker interface** is an empty interface used to indicate that a class has a certain property or capability.
* **Usage**: Provides metadata to the runtime or other systems that a class can perform certain operations (like serialization).
* **Examples**: Serializable, Cloneable, Remote.

Marker interfaces are a simple yet powerful concept in Java that enable you to mark classes with specific characteristics, and while newer techniques like annotations are more flexible, marker interfaces remain a key part of Java's type system.

1. What diamond problem occurs in the interface?  
   In Java, the diamond problem can occur when an interface inherits from multiple other interfaces, and those parent interfaces have methods with the same signature. If a class implements the child interface, it can be unclear which version of the method should be used, leading to ambiguity.  
     
   **What happens in the above code?**
2. **Interfaces A, B, and C**:
   * A declares a display() method with a default implementation.
   * Both B and C extend A and provide their own default implementation of the display() method.
3. **Class D**:
   * D implements both B and C.
   * D inherits the display() method from both B and C, leading to a potential conflict.

### **Resolving the Diamond Problem in Java:**

Java requires that you explicitly resolve this conflict. In the example above, the class D must override the display() method to resolve the ambiguity. The implementation in D can choose to call either B's or C's version of display() by using B.super.display() or C.super.display().

### **Why the Diamond Problem is Less of an Issue in Java:**

* **Default Methods**: Java introduced default methods in interfaces starting from Java 8. Default methods allow interfaces to have method implementations, but if a class implements multiple interfaces that have the same default method, the class must override the method to resolve ambiguity.
* **No Multiple Inheritance of Classes**: Java avoids the diamond problem in classes by disallowing multiple inheritance of classes. A class can only extend one other class, so there’s no ambiguity about which method implementation to inherit.

### **Summary**

* The diamond problem in Java can occur when a class implements multiple interfaces that have the same default method.
* Java requires the implementing class to override the conflicting method to resolve the ambiguity.
* The problem is less severe in Java because multiple inheritance of classes is not allowed, and interface conflicts can be resolved using explicit method calls like B.super.method().

1. How to use redis in a project?
2. Write to backend vs Write to Cache
3. How to iterate over a hashmap?
4. ConcurrentModificationException
5. Write a Immutable Class
6. Usage of Immutable Class
7. Why is String immutable?
8. talk about the recent project
9. what is the result of following program

public class CompletableFutureCombine {

public static void main(String[] args) {

CompletableFuture<Integer> future1 = CompletableFuture.supplyAsync(() -> {

return 5;

});

CompletableFuture<Integer> future2 = CompletableFuture.supplyAsync(() -> {

return 10;

});

CompletableFuture<Integer> resultFuture = future1.thenCombine(future2, (result1, result2) -> result1 + result2);

try {

System.out.println("Combined result: " + resultFuture.get());

} catch (InterruptedException | ExecutionException e) {

e.printStackTrace();

}

}

}

1. merge two maps
   1. if there are duplicate keys just sum up the values

Map<String, Integer> map1 = new HashMap<>();

map1.put("a", 1);

map1.put("b", 2);

map1.put("c", 3);

Map<String, Integer> map2 = new HashMap<>();

map2.put("b", 3);

map2.put("c", 4);

map2.put("d", 5);

map2.forEach((key, val) -> map1.merge(key, val, Integer::sum));

1. write a Restful API to save a user payload
   1. write a test for User service (simulate when a user call this API)
2. Given a table `employees` with columns `id`, `name`, `manager\_id`, and `salary`, ‘department\_id’ write a query to find the highest paid employee in each department
3. What is the Zuul Gateway API?
4. What is Netflix Eureka -> Service Discovery
5. What is Singleton?  
   1. Recent project you used Spark. What is the use case you have worked on with that libraries?  
   In a recent project where I used Apache Spark, the primary use case involved processing large datasets for real-time analytics. The project required efficient handling and transformation of streaming data, which was ingested from various sources like Kafka and then processed to extract meaningful insights in near real-time.

### **Specific Use Case:**

The project involved a large e-commerce platform where user behavior data (such as clicks, searches, and purchases) was streamed in real-time. The goals were to:

1. **Real-Time Analytics**: Provide real-time dashboards and reports on user activity to the marketing and product teams.
2. **Recommendation Engine**: Process user interactions to feed into a recommendation engine that could suggest products based on real-time behavior.
3. **Anomaly Detection**: Identify and flag any unusual patterns, such as sudden spikes in traffic or suspicious activities, to enhance security and fraud detection.

### **How Spark Was Used:**

* **Spark Streaming**: We used Spark Streaming to process the real-time data streams. The streaming data was ingested from Kafka topics, and Spark was responsible for windowing operations, aggregations, and joining with historical data stored in a distributed file system.
* **Spark SQL**: To perform complex queries and aggregations on the data in real-time, Spark SQL was employed. This allowed us to leverage SQL-like syntax to filter, group, and analyze the streaming data efficiently.
* **DataFrames and Datasets**: We used DataFrames and Datasets to manage structured data, enabling type safety and optimization through Spark's Catalyst optimizer.
* **Machine Learning with Spark MLlib**: For the recommendation engine, Spark MLlib was used to apply collaborative filtering algorithms (such as ALS) on user-item interaction data. This helped in generating real-time product recommendations for users.

### **Outcome:**

The implementation of Spark in this project allowed for scalable, efficient processing of real-time data, enabling the e-commerce platform to deliver personalized experiences to users and respond to market trends swiftly. The use of Spark also ensured that the system could handle the large volume of data with low latency, providing valuable insights and recommendations almost instantaneously.

2. Can a constructor be inherited in Java?

3. Write a Java function that takes a list and returns a map with the count of each element in the list.

You have called elementCountMap.getOrDefault. What does this function do?

List<String> list = Arrays.asList("aaa", "bbb", "aaa");

Map<T, Integer> elementCountMap = new HashMap<>();

for (T element:list){

elementCountMap.put(element, elementCountMap.getOrDefault(element, 0)+1);

}

System.out.println(elementCountMap);

output: {aaa=2, bbb=1}

4. What is the Callable interface and how is it different from Runnable?

5. Output

public class StreamExample {

public static void main(String[] args) {

List<String> list = Arrays.asList("apple", "banana", "apricot", "avocado");

List<String> result = list.stream()

.filter(s -> s.startsWith("a"))

.sorted((s1, s2) -> s2.compareTo(s1))

.collect(Collectors.toList());

System.out.println(result);

}

}

avocado apricot, apple

6. What is the Eureka Registry in Spring Boot?

**Eureka** is a service registry developed by Netflix as part of the Netflix OSS suite and is widely used in Spring Boot applications as part of the Spring Cloud ecosystem.

#### **Key Concepts of Eureka:**

* **Service Registry**: Eureka acts as a service registry where microservices can register themselves at runtime. Once registered, other services can discover and communicate with them.
* **Discovery Client**: Each microservice that registers with Eureka is known as a Eureka client. These clients periodically send heartbeats to the Eureka server to indicate that they are still available.
* **Eureka Server**: The central server where all microservices register. It maintains a registry of all available service instances and their statuses.
* **Service Discovery**: Microservices can use the Eureka client to query the Eureka server for the locations of other services. This enables dynamic discovery and communication between microservices without hardcoding service URLs.

7. What is Zuul?  
**Zuul** is an edge service (API Gateway) developed by Netflix and part of the Netflix OSS suite. Zuul is often used in Spring Cloud to provide dynamic routing, monitoring, resiliency, security, and other cross-cutting concerns.

#### **Key Concepts of Zuul:**

* **API Gateway**: Zuul acts as an API gateway, handling all requests from clients and routing them to the appropriate microservices.
* **Routing**: Zuul dynamically routes requests to different backend services based on the URL patterns or request parameters.
* **Filters**: Zuul supports pre-routing, post-routing, error, and route filters, which can be used to implement custom logic, such as authentication, logging, or rate-limiting.
* **Load Balancing**: Zuul integrates with Ribbon (another Netflix OSS component) to provide client-side load balancing.

8. What is OAuth2?  
**OAuth 2.0** is an open standard for access delegation, commonly used to grant third-party applications limited access to HTTP services on behalf of a user. OAuth 2.0 provides a "secure delegated access" to server resources on behalf of a resource owner.

#### **Key Concepts of OAuth 2.0:**

* **Resource Owner**: The user who authorizes an application to access their account.
* **Client**: The application that requests access to the resource owner's account.
* **Authorization Server**: The server that authenticates the resource owner and issues access tokens to the client.
* **Resource Server**: The server that hosts the protected resources and accepts access tokens to grant access to them.

#### **OAuth 2.0 Flow:**

1. **Authorization Code Grant**:
   * **Step 1**: The client application directs the resource owner (user) to the authorization server, where the user authenticates and authorizes the client.
   * **Step 2**: The authorization server redirects the user back to the client with an authorization code.
   * **Step 3**: The client exchanges the authorization code for an access token.
   * **Step 4**: The client uses the access token to request resources from the resource server.
2. **Implicit Grant**:
   * Similar to the authorization code grant but used primarily for public clients, such as single-page applications (SPAs), where the access token is returned directly without an authorization code.
3. **Client Credentials Grant**:
   * Used when the client is also the resource owner, such as in machine-to-machine communication, where the client authenticates directly with the authorization server to obtain an access token.
4. **Refresh Token**:
   * A refresh token is issued alongside an access token and allows the client to request a new access token without requiring the resource owner to authenticate again.

9. What is the difference between @Controller and @RestController in Spring Framework?

10. What is Test-Driven Development (TDD)?  
**Test-Driven Development (TDD)** is a software development approach in which tests are written before writing the actual code. TDD is part of the agile software development process and emphasizes writing automated test cases for new features or functionality before the code itself is implemented.

### **Key Principles of TDD:**

1. **Red-Green-Refactor Cycle**:
   * **Red**: Start by writing a test for a new feature or functionality. Initially, the test will fail because the corresponding code hasn't been written yet. This is the "red" phase, indicating a failing test.
   * **Green**: Write the minimum amount of code necessary to make the test pass. The goal is to write just enough code to pass the test without worrying about design or optimization at this stage. Once the test passes, you've reached the "green" phase.
   * **Refactor**: After the test passes, review and clean up the code. Refactor the code to improve its structure, readability, and maintainability while ensuring that all tests continue to pass. This is the "refactor" phase.
2. **Write Small, Focused Tests**:
   * In TDD, tests are written to cover small units of functionality, typically focusing on a single method or function. These tests should be fast, isolated, and independent of each other.
3. **Frequent Testing**:
   * TDD encourages continuous and frequent testing. Developers write tests before each small piece of functionality, ensuring that every part of the codebase is covered by automated tests.
4. **Documentation**:
   * The tests themselves serve as documentation for the code. They describe how the code is expected to behave and provide examples of its usage.

11. What is a Singleton Design Pattern and how do you ensure the creation of a single object with it?

12. Given a `sales` table with columns `order\_id`, `product\_id`, `customer\_id`, and `order\_date`, write a query to find customers who have placed orders in every month of the year 2023.

select customer\_id from sales where EXTRACT(year from order\_date) = 2023

group by customer\_id having count(distinct extract(month from order\_date))=12;

Vendor round 2 (6/15) - 40 mins:

1. What are the Microservice Design Patterns you know?
2. How to design a system to invoke three different microservices for user registration with failure recovery?

Design a system that invokes three different microservices (HR, Admin, Network) to register an employee. Once the employee is successfully registered in all three departments, the system should send a unique employee ID to the employee via email. If an API call fails at any point, the system should remember where it failed. The next time the same employee tries to onboard, the system should resume from the point of failure. For example, if the registration succeeds with Admin but fails with Network, the next attempt should start with Network, not Admin.

1. **What are the drawbacks of a system tracking the state of each record for microservice registration?**  
   Tracking the state of each record for microservice registration in a service registry comes with several potential drawbacks. These drawbacks primarily relate to increased complexity, scalability challenges, performance issues, and potential reliability concerns. Here’s a detailed explanation:

### **1. Increased System Complexity**

* **State Management**: Maintaining the state of each microservice instance adds significant complexity to the system. The service registry must implement mechanisms to monitor, update, and store the state of each service instance, which can include health checks, status updates (e.g., up, down, degraded), and handling instances joining or leaving the network.
* **Consistency Issues**: Ensuring that the state information is consistent across distributed systems can be challenging. Network partitions, service failures, or delayed updates can lead to situations where the registry has stale or incorrect information about the state of services.

### **2. Scalability Challenges**

* **Resource Overhead**: As the number of microservices and their instances grows, the overhead of tracking each instance's state increases. This can strain the service registry's resources, including memory, CPU, and network bandwidth, especially if there are frequent state changes.
* **Performance Bottlenecks**: Continuously monitoring and updating the state of many microservices can become a performance bottleneck. The service registry may struggle to keep up with the load, leading to slower response times or even failures in state updates.

### **3. Latency and Delays**

* **Propagation Delays**: There may be delays in propagating state changes (e.g., when a service goes down) across the system. During this period, other services may attempt to communicate with a service that is no longer available, leading to failed requests and degraded performance.
* **Health Check Overhead**: Frequent health checks to determine the state of each service instance can introduce latency. These checks can consume network resources and processing power, especially in large-scale systems.

### **4. Reliability and Fault Tolerance**

* **Single Point of Failure**: The service registry itself becomes a critical component of the system. If it fails or becomes unavailable, other services may not be able to discover or communicate with each other. This increases the importance of ensuring the registry is highly available and fault-tolerant.
* **Stale Data**: In distributed systems, there’s always a risk of having stale or inconsistent state data due to network issues or service crashes. If the service registry is not updated in real-time, it may lead to incorrect routing of requests to unhealthy or unavailable services.

### **5. Complex Recovery and Reconciliation**

* **Reconciliation Challenges**: After events like network partitions or registry failures, reconciling the state of services can be complex. The registry must ensure that its records accurately reflect the current state of all service instances, which may require complex synchronization mechanisms.
* **Complicated Failover Handling**: Managing failover scenarios can be more complicated when each service’s state must be tracked. The registry needs to quickly identify and route traffic to healthy instances while avoiding unhealthy ones, which requires reliable and up-to-date state information.

### **6. Operational Overhead**

* **Maintenance**: The operational overhead of maintaining a service registry that tracks state can be significant. This includes ensuring the registry itself is robust, updating it as new services are added or removed, and monitoring its performance and health.
* **Coordination Between Services**: As the number of services grows, coordinating updates to the state of each service instance can become cumbersome. This might involve sophisticated mechanisms to manage the lifecycle of services, including deployment, scaling, and decommissioning.

### **7. Security Concerns**

* **Sensitive Data**: If the state information includes sensitive details about the services, such as their health status or configuration, securing this data becomes critical. Unauthorized access or tampering with the registry could lead to severe security breaches or operational issues.

### **Summary**

Tracking the state of each record for microservice registration brings several challenges:

* **Increased complexity** in managing state consistency and updating service status.
* **Scalability issues** as the system grows, leading to potential performance bottlenecks.
* **Latency and reliability concerns** due to propagation delays and the risk of stale data.
* **Operational overhead** in maintaining and securing the service registry.
* **Complex recovery processes** in case of failures or inconsistencies.

1. **What is the complexity in a system?**  
   **System complexity** refers to the various factors that contribute to the difficulty of understanding, designing, implementing, testing, maintaining, and scaling a software system. Complexity can arise from multiple sources, including the architecture, codebase, data handling, and interactions between components.

#### **Types of Complexity:**

* **Structural Complexity**:
  + **Component Interactions**: The more components a system has and the more they interact with each other, the more complex the system becomes. High coupling and dependencies between modules or microservices can lead to intricate and hard-to-manage systems.
  + **Code Complexity**: Deeply nested code, unclear logic, and a lack of modularity can make a system difficult to understand and maintain. This is often measured using metrics like cyclomatic complexity.
* **Behavioral Complexity**:
  + **Concurrency and Parallelism**: Systems that involve multiple concurrent or parallel processes introduce complexity in managing synchronization, shared resources, and avoiding issues like race conditions or deadlocks.
  + **State Management**: Managing the state, especially in distributed systems, adds complexity due to the need to ensure consistency, handle state transitions, and recover from failures.
* **Operational Complexity**:
  + **Deployment and Scaling**: Managing the deployment of services, especially in a microservices architecture, adds complexity. Ensuring that services are correctly deployed, monitored, and scaled can be challenging.
  + **Fault Tolerance and Recovery**: Designing a system that can handle failures gracefully and recover quickly adds to the complexity. This includes implementing redundancy, failover mechanisms, and disaster recovery plans.
* **Data Complexity**:
  + **Data Consistency**: Ensuring that data remains consistent across distributed systems or microservices, especially when dealing with eventual consistency models, can be complex.
  + **Data Handling**: Managing large volumes of data, ensuring data integrity, and optimizing data access patterns can contribute to system complexity.
* **Security Complexity**:
  + **Authentication and Authorization**: Implementing secure authentication and authorization mechanisms across different parts of the system adds complexity.
  + **Data Protection**: Ensuring that data is protected both at rest and in transit, and managing access controls, adds additional layers of complexity.

1. **What is the complexity in using a database or Redis cache for tracking and retrying failed registration stages in a microservice system?**

**When tracking and retrying failed registration stages in a microservices system, choosing between a database and a Redis cache introduces different complexities:**

#### **Using a Database:**

* **Advantages:**
  + **Durability: Databases provide persistent storage, ensuring that the state of failed registrations is not lost even in the case of a system crash.**
  + **Advanced Querying: Databases offer complex querying capabilities, allowing for more sophisticated logic when tracking and retrying failed stages.**
  + **Consistency: Databases, especially those with ACID properties, provide strong consistency guarantees, which can be crucial for ensuring that the state is accurately tracked.**
* **Complexity:**
  + **Performance Overhead: Databases might introduce latency due to disk I/O operations, especially if there are frequent writes and reads to track and retry failed registrations.**
  + **Scalability: Scaling databases can be challenging, especially if the system needs to handle a large volume of failed registrations. You might need to implement sharding, replication, or other scaling techniques.**
  + **Operational Complexity: Managing and maintaining a database, including backups, schema migrations, and query optimization, adds to the operational complexity.**

#### **Using Redis Cache:**

1. **Advantages:**
   1. **Speed: Redis is an in-memory data store, which provides very low latency for both reads and writes, making it ideal for tracking and retrying failed registrations in real-time.**
   2. **Ease of Use: Redis is relatively easy to set up and use, with straightforward data structures that are well-suited for caching scenarios.**
2. **Complexity:**
   1. **Durability: Redis is primarily an in-memory store, which means data can be lost if the system crashes unless you configure Redis with persistence (e.g., RDB snapshots or AOF). This adds complexity if durability is a requirement.**
   2. **Consistency: Redis provides eventual consistency, which may lead to scenarios where stale data is used if multiple Redis instances are involved. Ensuring data consistency across distributed Redis instances can be complex.**
   3. **Data Management: Redis is well-suited for simple caching, but for complex querying and relational data handling, it may require additional effort to manage and use effectively.**
3. **What is a Circuit Breaker and how to implement it?**

#### **Circuit Breaker Pattern:**

The **Circuit Breaker** pattern is a design pattern used to detect failures and encapsulate the logic of preventing a failure from continually recurring during maintenance, temporary external system failure, or unexpected system difficulties. The idea is to prevent an application from repeatedly trying to execute an operation that is likely to fail, thereby avoiding system overload and improving overall stability and resilience.

#### **How Circuit Breaker Works:**

1. **Closed State**: In this state, the circuit breaker allows requests to pass through and monitors the success and failure rates. If the failure rate exceeds a certain threshold, the circuit breaker moves to the Open state.
2. **Open State**: In the Open state, the circuit breaker short-circuits the requests and immediately returns a failure response or a fallback response. This prevents the application from making calls that are likely to fail.
3. **Half-Open State**: After a certain period, the circuit breaker transitions to the Half-Open state. In this state, it allows a limited number of requests to pass through and monitors their outcome. If the requests are successful, the circuit breaker transitions back to the Closed state. If they fail, it moves back to the Open state.

#### **Implementing Circuit Breaker in Spring Boot with Resilience4j:**

Resilience4j is a popular library for implementing the Circuit Breaker pattern in Java applications, especially in Spring Boot.

1. How do you implement security in your microservices? How do you manage authentication and authorization?
2. **How does Java internally work? How do linking, loading, and initialization happen in Java?**

### **Class Loading**

### **The class loading process in Java is dynamic, meaning classes are loaded into memory as needed, rather than all at once. This process is handled by the ClassLoader.**

### **Bootstrap ClassLoader: The lowest-level class loader, responsible for loading the core Java libraries (e.g., classes in rt.jar).**

### **Extension ClassLoader: Loads classes from the extension directories (lib/ext).**

### **Application ClassLoader: Loads classes from the application’s classpath (e.g., classes in your project’s src directory or external libraries).**

### **Linking**

**Linking in Java involves three steps: Verification, Preparation, and Resolution.**

* **Verification:**
  + **The JVM verifies that the bytecode is valid and doesn’t violate any Java language specifications. This step ensures that the bytecode adheres to Java’s security constraints, preventing issues like stack overflows or memory corruption.**
  + **Bytecode verification checks for things like proper use of data types, correct method invocation, and adherence to access control rules.**
* **Preparation:**
  + **During preparation, the JVM allocates memory for class variables and sets them to default values. This step initializes static fields to their default values (e.g., 0 for integers, null for object references).**
  + **Memory is allocated in the method area of the JVM.**
* **Resolution:**
  + **Resolution involves converting symbolic references in the bytecode (e.g., method names, class names) into direct references to memory locations. This allows the JVM to quickly access these references during execution.**
  + **This step may involve loading additional classes if they are referenced but not yet loaded.**

**Compilation**: Java source code is compiled into bytecode by javac.

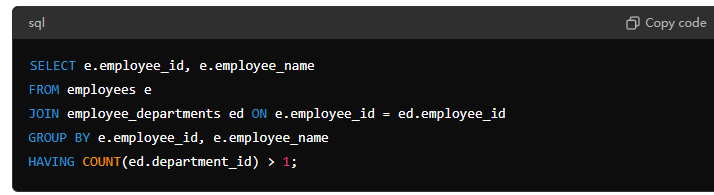
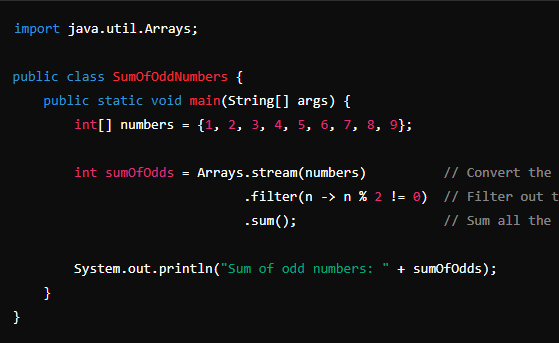
**Class Loading**: The JVM dynamically loads classes into memory using ClassLoaders.

**Linking**: The JVM verifies, prepares, and resolves classes, ensuring they are valid and ready for execution.

**Initialization**: The JVM initializes static and instance variables and runs constructors before using the class.

**Execution**: The JVM interprets or JIT compiles bytecode into native machine code for execution.

**Garbage Collection**: The JVM automatically reclaims memory by collecting and removing unused objects.

1. write a simple sum function and the related test cases  
   @Test  
   assertEquals
2. given a string array {"abcd", "abc”, "ab", "a"}, return a4b3c2d1 with O(1).
3. explain how to use Hibernate @OneToMany
4. write Sql query to get all the employees with multiple departments from two tables  
   
5. use stream api and lambda to get the sum of all the odd numbers in an array  
   
6. how to realize the rollback between 4 services in microservice  
   **Service A**: Payment Service

**Service B**: Inventory Service

**Service C**: Shipping Service

**Service D**: Notification Service

Saga pattern orchestration

1. what is the design pattern in microservice (circuit breaker)
2. what is singleton design pattern and singleton bean scope
3. java oop
4. what is checked exception, why is FileNotFound exception the checked exception
5. JIT
6. default method
7. composition, association and aggregation

05/22 Vendor round 2 - 30 mins:

1. what is service registry
2. how does service registry know if there is a new microservice got created
3. what is k8s
4. how nodes communicate in k8s
5. what is the server of k8s
6. Is Kafka used for one-to-one communication or broadcasting

both

1. what design pattern
2. Have you written singleton yourself or just used it
3. lazy loading and eager loading in hibernates
4. Any Cloud experience?  
   Tell about multi-thread

Tell about kafka, how to increase the thoughput / what to do if there's delay in consumer parts

**Tell me about vert.x, how to different comparing to Spring**

**Vert.x** is an event-driven, non-blocking, reactive framework designed for building high-performance, scalable applications. It is modular, polyglot, and ideal for scenarios requiring low latency and high concurrency.

**Spring** is a comprehensive, enterprise-grade framework that offers a wide range of features for building robust, secure, and maintainable applications. It traditionally follows a synchronous model but has added support for reactive programming with Spring WebFlux.

**Key Differences** include the programming model (blocking vs. non-blocking), scalability, modularity, and ease of use, with Vert.x being more suited for reactive, high-performance scenarios, and Spring being the go-to for traditional enterprise applications with extensive ecosystem support.

Tell about the relational and non-relational database you used, and what it used for, why

**Difference between Where and Having in SQL  
WHERE Clause**:

* Filters individual rows.
* Applied before grouping and aggregation.
* Used with SELECT, UPDATE, DELETE, etc.

**HAVING Clause**:

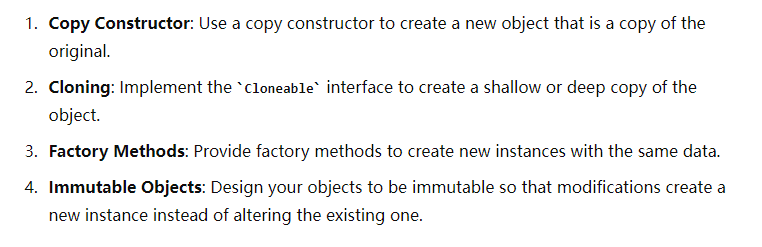
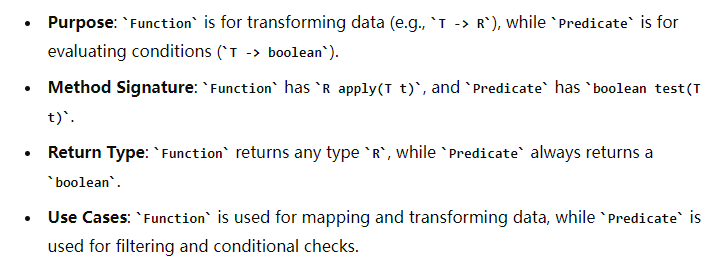
* Filters groups of rows after aggregation.
* Applied after grouping and aggregation.
* Used only with GROUP BY.

Tell about the sonarqube

Coding:

Output longest non-repeat substring from a given string

* Tell the time complexity

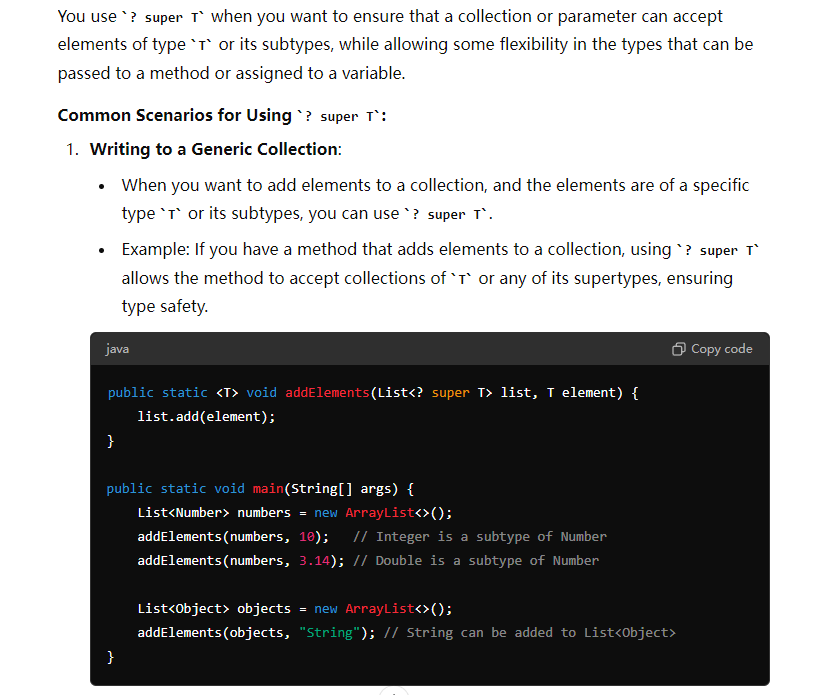
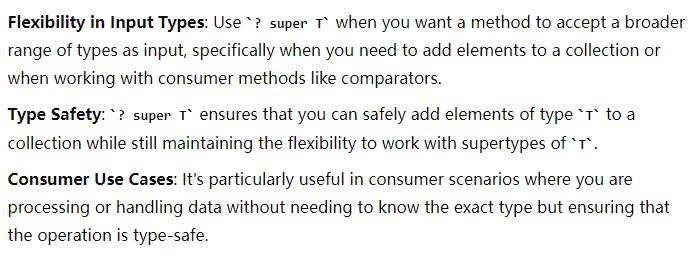
1. self-intro
2. Tell me about the Abstraction
   1. - Hiding those details from whom
   2. - Meaning of hiding the implementation details
3. Design Pattern you used in project
4. How to keep instances been noticed if you have 4 in different location
5. TreeMap vs. HashMap
6. Tell me about the JVM and JIT Compiler
7. Why String is immutable and how to make a class immutable
8. How to avoid effect the original object when modifying it   
   
9. Differences between compile-time exception and run-time exception
   1. - why don't add try/catch block to run-time exception
10. What's the default method of interface
    1. - And Why
11. Differences between Function and Predicate  
    
12. What's the functional interface before Java8 - anonymous class
13. What's SpringBoot's advantages
14. Bean scopes in Spring
15. Tell me about the IOC container
16. Differences between Singleton Pattern and Singleton Bean

Coding -

1. Given String and output the frequency of each character
   1. - follow up: output them based on the frequency
2. Find the longest and last occur palindrome from given String
   1. 要求口头dry run
3. SQL - Output the employee with multiple department

第二轮1h：Vendor技术面

换了个印度面试官

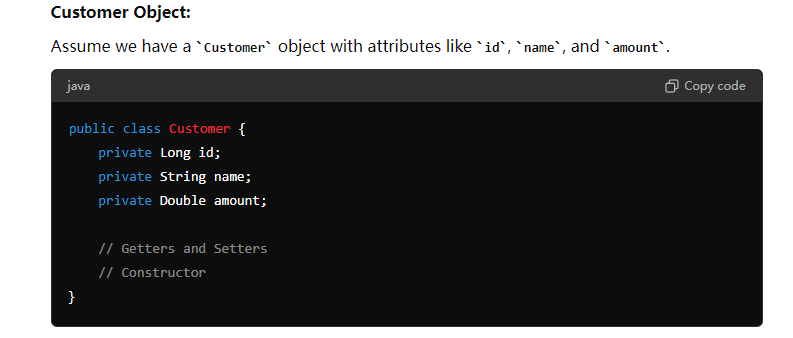
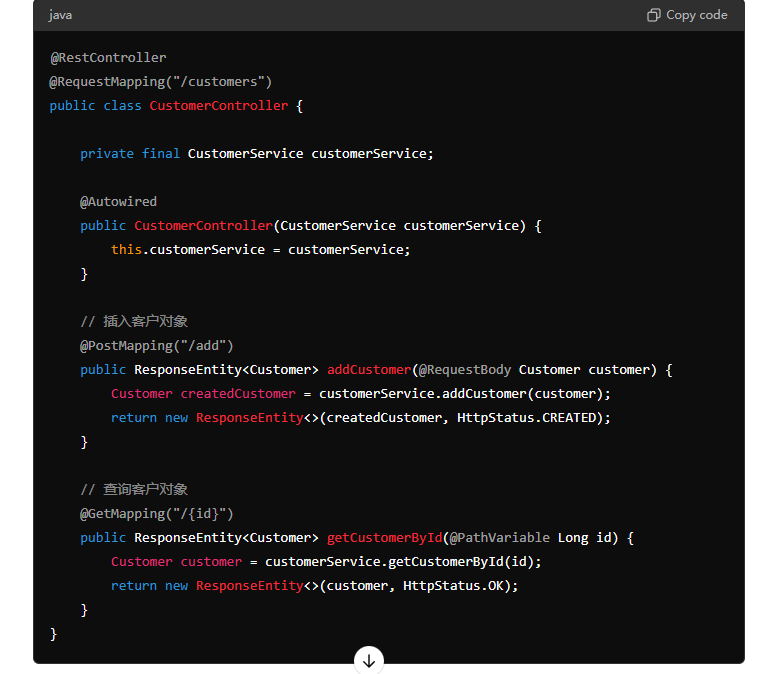
1. write code for functional interface
2. write code for convert (1, 2, 3, 4) to ({1, 2}, {2, 4}, {3, 6}, {4, 8})
3. how to pass parameters to SpringApplication
4. Tell me about IoC and DI
5. What's the advantages of GraphQL
6. Tell me about JPA and GraphQL
7. Compare about REST and GraphQL
8. Differences between hashMap and ConcurrentHashMap
9. - if we can make HashMap thread safe, choose which one for better performance
10. What kind of data do you stored using Cassandra/MongoDB
11. Tell me about Kafka
12. how message been consumed in the topic/ what offset is
13. What's pod in k8s
14. Tell me about the deployment process
15. - CICD
16. Tell me about Comparable and Comparator
17. Tell me about HashMap and LinkedHashMap
18. When would you use 'super' with generics  
      
    
19. Tell me about thread pool

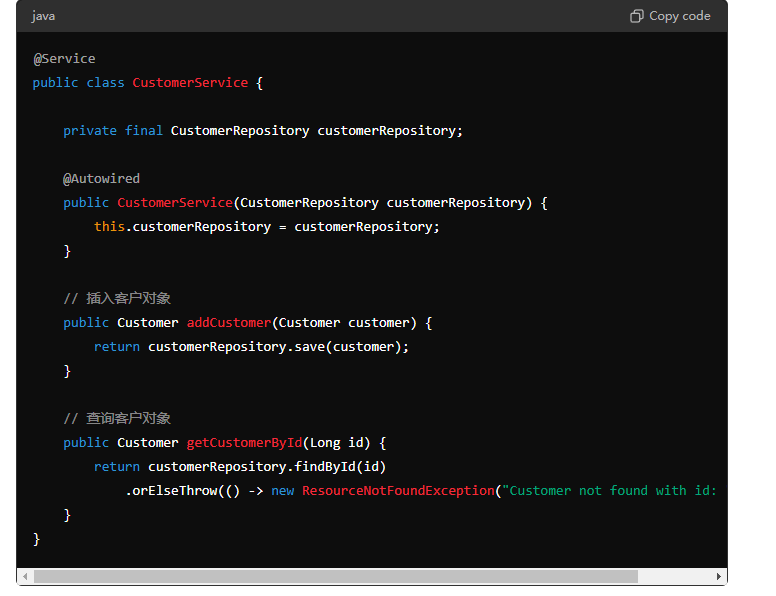
第三轮30min：HR面

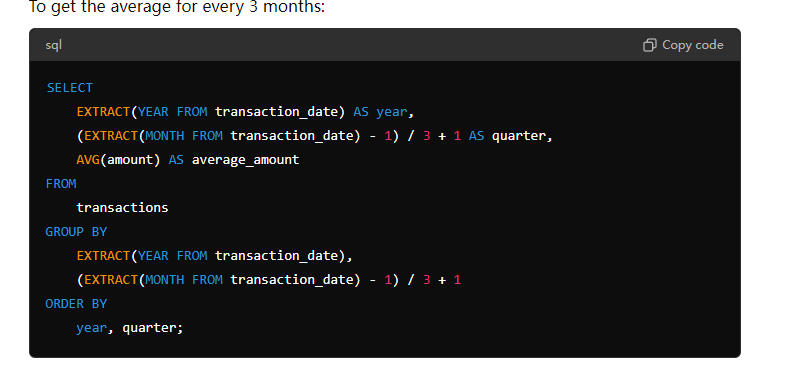
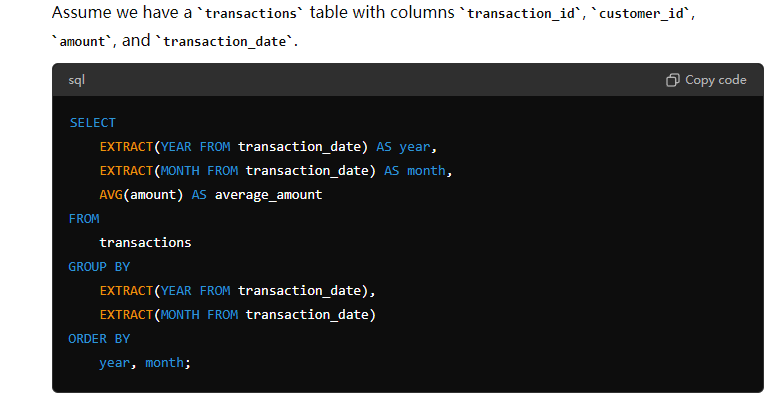
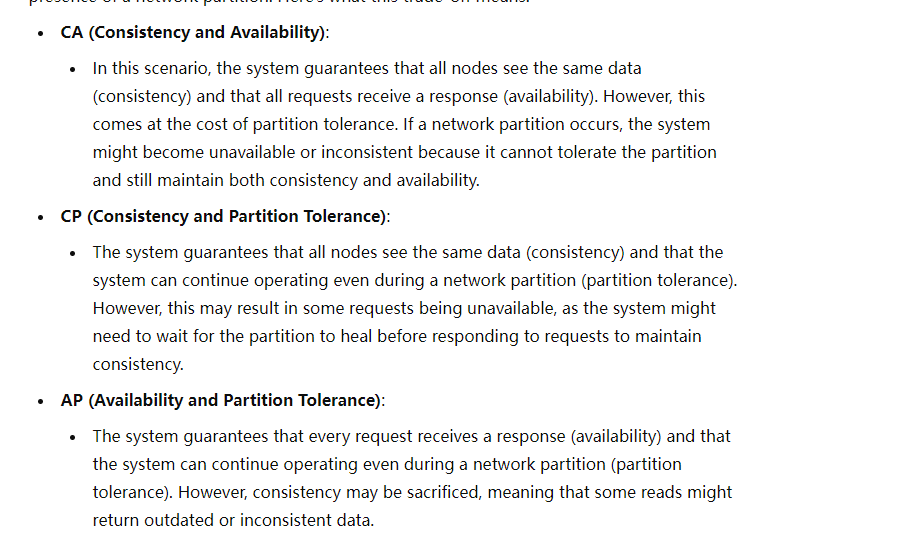
又换了个印度面试官

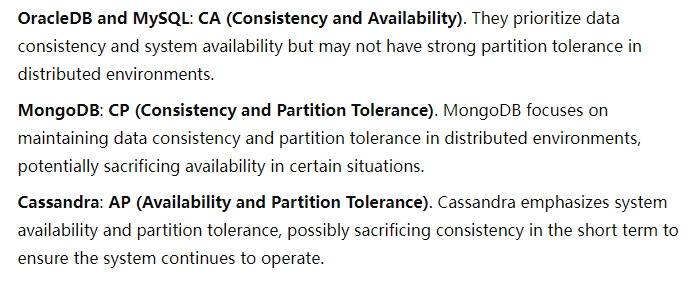
HR面，就是简历上的时间以及地点，没有技术

第四轮1h：Client面  
中国人面的

1. Write down two api(Controller, Service only) for insert & query customer object  
     
   



1. One SQL that request you to output the average amount of each calendar month
   1. follow up - for every 3 month  
      
2. One coding question to accept 4 customer object, and then output the customer with amount greater than 100  
   
3. One coding question to output the fewest number of coins to match the amount   
   DP problem
4. tell me about CAP   
   **Consistency**, **Availability**, and **Partition Tolerance**.  
   

OracleDB MySQL是CA   
MongoDB CP  
Cassandra AP  
  
**Write a Java program to simulate a simple producer-consumer scenario using multithreading.  
 public static void main(String[] args) {**

**// Shared resource (a queue) with a capacity of 5 items**

**Queue<Integer> queue = new LinkedList<>();**

**int capacity = 5;**

**// Create producer and consumer threads**

**Thread producerThread = new Thread(new Producer(queue, capacity), "Producer");**

**Thread consumerThread = new Thread(new Consumer(queue), "Consumer");**

**// Start the threads**

**producerThread.start();**

**consumerThread.start();**

**}**

**}**

**// Producer class**

**class Producer implements Runnable {**

**private final Queue<Integer> queue;**

**private final int capacity;**

**public Producer(Queue<Integer> queue, int capacity) {**

**this.queue = queue;**

**this.capacity = capacity;**

**}**

**@Override**

**public void run() {**

**int value = 0;**

**while (true) {**

**synchronized (queue) {**

**// Wait if the queue is full**

**while (queue.size() == capacity) {**

**try {**

**System.*out*.println("Queue is full, producer is waiting...");**

**queue.wait();**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**}**

**// Produce an item and add it to the queue**

**System.*out*.println("Produced: " + value);**

**queue.add(value++);**

**// Notify the consumer that an item is available**

**queue.notify();**

**// Sleep to simulate time taken to produce an item**

**try {**

**Thread.*sleep*(1000);**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**}**

**}**

**}**

**}**

**// Consumer class**

**class Consumer implements Runnable {**

**private final Queue<Integer> queue;**

**public Consumer(Queue<Integer> queue) {**

**this.queue = queue;**

**}**

**@Override**

**public void run() {**

**while (true) {**

**synchronized (queue) {**

**// Wait if the queue is empty**

**while (queue.isEmpty()) {**

**try {**

**System.*out*.println("Queue is empty, consumer is waiting...");**

**queue.wait();**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**}**

**// Consume an item from the queue**

**int value = queue.poll();**

**System.*out*.println("Consumed: " + value);**

**// Notify the producer that space is available**

**queue.notify();**

**// Sleep to simulate time taken to consume an item**

**try {**

**Thread.*sleep*(1000);**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**}**

**}**

**}**