Section 1 : Interview

**1:- How do you make sure that all threats are completed and wait to complete?**

Behavior of thread.join():

If you call join() on a thread, the calling thread will block (pause its execution) until the thread it called join() on has completed its task.

Once the thread on which join() was called has finished, the calling thread resumes its execution.

public class JoinExample {

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

// Create a new thread

Thread thread = new Thread(() -> {

try {

// Simulate some work in the new thread

System.out.println("Thread is starting...");

Thread.sleep(2000); // Sleep for 2 seconds

System.out.println("Thread is finished.");

} catch (InterruptedException e) {

e.printStackTrace();

}

});

// Start the thread

thread.start();

// Main thread waits for the thread to complete

System.out.println("Main thread is waiting for the thread to finish.");

thread.join(); // Main thread waits for the above thread to finish

// After the thread finishes, main thread continues

System.out.println("Main thread is finished.");

}

}

//output

Main thread is waiting for the thread to finish.

Thread is starting...

Thread is finished.

Main thread is finished.

Key Points:

The main thread waits for the child thread to complete before proceeding further.

The join() method blocks the main thread until the child thread has finished executing.

=======================================================================================

**2:---how do you configure load balance how we communicate each other?**

Example Use Case for Both:

**Client-Side Load Balancing:**

If you have a microservices-based architecture and you're using Spring Cloud (with Eureka and Ribbon),

each service can directly manage load balancing for outgoing requests to other services.

Server-Side Load Balancing: If you use Spring Cloud Gateway or an API Gateway (with Eureka for service discovery and Spring Cloud LoadBalancer),

the gateway can act as a server-side load balancer, distributing incoming requests to various backend services.

In large-scale systems, you might combine both approaches—client-side load balancing for internal service-to-service communication

and server-side load balancing for managing external traffic.

==================================================================

**3------ explain EKS, what is the difference between deployment and pod?**

EKS (Elastic Kubernetes Service):

EKS is a managed Kubernetes service provided by Amazon Web Services (AWS). It simplifies running Kubernetes clusters on AWS,

handling tasks like cluster provisioning, scaling, and management. It allows users to deploy, manage, and scale containerized

applications using Kubernetes without needing to manage the underlying infrastructure.

Difference Between Deployment and Pod:

Pod:

The smallest and simplest unit in Kubernetes.

Represents a single instance of a running process or container.

Can contain one or more containers, which share the same network, storage, and namespace.

Pods are ephemeral, meaning they can be terminated and replaced by new ones.

Deployment:

A higher-level abstraction used to manage replicas of Pods.

Ensures that the specified number of replicas of a Pod are running at any time, even if Pods are terminated or fail.

Handles rollouts and updates to Pods, providing easy scaling, versioning, and self-healing of Pods.

A Deployment typically creates and manages multiple Pods.

In short, Pods are individual units of execution, while a Deployment manages and ensures the desired state and scaling of Pods in

a Kubernetes environment.

=====================================================================

**4:---- how do you do transaction in the below scenario? A message has to read parts and save the database**

Explanation:

@Transactional ensures that the whole processMessage method is executed within a transaction. If any part of the process fails

(e.g., saving parts to the database), the transaction will be rolled back, ensuring no partial data is committed to the database.

If an exception is thrown, Spring automatically rolls back the transaction.

The database operations (like saving part1 and part2) are handled in a single transactional context.

Key Points:

Atomicity: The whole operation (reading, processing, saving) is treated as a single unit of work.

Rollback: If any part fails (e.g., savePartToDatabase), the transaction is rolled back automatically, ensuring consistency.

This approach provides a clean and concise way to manage transactions for reading and saving data in a Spring application.

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

@Service

public class MessageService {

@Transactional

public void processMessage(String message) {

try {

// Step 1: Read parts from the message

String part1 = extractPart1(message);

String part2 = extractPart2(message);

// Step 2: Process parts (e.g., transform data, validate)

// (Processing logic can be here)

// Step 3: Save the parts to the database

savePartToDatabase(part1);

savePartToDatabase(part2);

} catch (Exception e) {

// Exception handling: Transaction will be rolled back automatically

throw new RuntimeException("Failed to process message", e);

}

}

private String extractPart1(String message) {

// Logic to extract part1 from the message

return message.substring(0, 5); // Example extraction logic

}

private String extractPart2(String message) {

// Logic to extract part2 from the message

return message.substring(5); // Example extraction logic

}

private void savePartToDatabase(String part) {

// Logic to save the part to the database (e.g., using a repository)

// For example, using JPA repository: messageRepository.save(new Message(part));

}

}

=====================================================

**5:--- how do you group by department in a person object ID name department ID and find second A salary**

import java.util.\*;

import java.util.stream.Collectors;

class Person {

int id;

String name;

int departmentId;

double salary;

public Person(int id, String name, int departmentId, double salary) {

this.id = id;

this.name = name;

this.departmentId = departmentId;

this.salary = salary;

}

public double getSalary() {

return salary;

}

public int getDepartmentId() {

return departmentId;

}

@Override

public String toString() {

return name + " (" + departmentId + "): " + salary;

}

}

public class Main {

public static void main(String[] args) {

List<Person> persons = Arrays.asList(

new Person(1, "Alice", 1, 50000),

new Person(2, "Bob", 1, 60000),

new Person(3, "Charlie", 2, 70000),

new Person(4, "David", 2, 80000),

new Person(5, "Eve", 1, 55000),

new Person(6, "Frank", 2, 75000)

);

Map<Integer, List<Person>> groupedByDepartment = persons.stream()

.collect(Collectors.groupingBy(Person::getDepartmentId));

// Finding the second highest salary in each department using skip(1)

for (Map.Entry<Integer, List<Person>> entry : groupedByDepartment.entrySet()) {

int departmentId = entry.getKey();

List<Person> departmentMembers = entry.getValue();

// Sort by salary in descending order

Person secondHighestSalaryPerson = departmentMembers.stream()

.sorted(Comparator.comparingDouble(Person::getSalary).reversed()) // Sort descending by salary

.skip(1) // Skip the highest salary

.findFirst() // Get the next person, which is the second highest salary

.orElse(null); // If no second salary exists, return null

if (secondHighestSalaryPerson != null) {

System.out.println("Second highest salary in department " + departmentId + ": " + secondHighestSalaryPerson);

} else {

System.out.println("Not enough people in department " + departmentId + " to find second highest salary.");

}

}

}

}

==========================================

using sql

SELECT department\_id, GROUP\_CONCAT(name) AS employees

FROM Person

GROUP BY department\_id;

SELECT department\_id, MAX(salary) AS second\_highest\_salary

FROM Person

WHERE salary < (

SELECT MAX(salary)

FROM Person

WHERE department\_id = Person.department\_id

)

GROUP BY department\_id;

=======================================================================================

**6---what is consumer group**

Summary:

Consumer Group is a set of consumers that read data from Kafka topics in a coordinated way.

It ensures load balancing, parallelism, and fault tolerance.

Each consumer in a group is assigned partitions, and each partition is read by only one consumer in the group.

Kafka tracks the offset for each consumer group, so that consumers know where they left off, providing reliability.

The consumer group concept is central to building scalable, fault-tolerant, and high-throughput systems in Kafka.

A Consumer Group in Apache Kafka is a mechanism that allows multiple consumers to read data from Kafka topics in a coordinated manner. It is a key part of Kafka's scalability and fault tolerance features.

Here’s a deeper explanation:

Key Concepts:

Kafka Topic:

A Kafka topic is a stream of messages. Kafka topics are divided into partitions, which allow for parallelism in consuming messages.

Consumer:

A consumer is an application or service that reads messages from Kafka topics. Each consumer is responsible for reading messages from

specific partitions of a topic.

Consumer Group:

A consumer group is a collection of consumers that work together to consume messages from one or more Kafka topics.

Kafka assigns partitions of a topic to consumers in the group. Each partition is consumed by one consumer in the group,

ensuring that messages from a partition are processed in order and by only one consumer.

How It Works:

Partition Assignment:

Kafka topics are divided into partitions. When consumers are part of a consumer group, Kafka assigns each partition to a single consumer within

that group. If there are more partitions than consumers, some consumers will read from multiple partitions.

If there are more consumers than partitions, some consumers in the group will be idle because each partition can only be assigned to one

consumer within a group.

Message Distribution:

Every message published to a Kafka topic is stored in one of the topic’s partitions. Each consumer in the group reads messages from o

ne or more partitions.

Consumers in a group do not receive duplicate messages. Each partition is consumed by only one consumer in the group, so each message

is processed by one consumer.

Parallel Processing:

The main benefit of a consumer group is parallel processing. Each consumer in the group processes different partitions simultaneously,

which allows Kafka to scale horizontally. As a result, multiple consumers can consume from the same topic in parallel, leading to better

throughput and efficiency.

Fault Tolerance:

If one consumer in the group fails (due to a crash, network issue, etc.), Kafka will automatically rebalance the partitions.

Other consumers in the group will take over the partitions previously handled by the failed consumer, ensuring that the message

consumption continues without interruption.

This rebalancing process ensures fault tolerance.

Offset Tracking:

Kafka tracks the offsets for each consumer group. An offset is a unique identifier for a message in a partition,

which indicates the position of the consumer in the partition.

Each consumer group maintains its own offset for each partition it is consuming from. This means different consumer

groups can consume the same messages independently.

The offset is committed, meaning the consumer will know where it left off in case it restarts or crashes, avoiding reprocessing of messages.

==============================================================

**7---if kafa produces a msg within a consumer group then once messge will get mutiple time or same group**

To summarize:

Each message will be consumed only once within a consumer group.

If a consumer fails or is slow, Kafka will rebalance the messages and may send unprocessed messages to

other consumers, but the same message will not be processed by multiple consumers in the group simultaneously.

=============================================================

**8:- how many ways thread can be refined or managed**

1. Thread Priorities

Use setPriority() to adjust the priority of a thread.

Higher priority threads are more likely to be executed before lower priority ones.

2. Thread Pooling (Executor Framework)

Use the ExecutorService to manage a pool of threads.

Prevents the overhead of creating new threads by reusing threads from a pool.

Example: Executors.newFixedThreadPool(10)

3. Thread Synchronization

Use synchronized methods or blocks to control access to shared resources.

Ensures that only one thread accesses a critical section at a time.

Example: synchronized (object) { // critical code }

4. Daemon Threads

Daemon threads run in the background and do not prevent the JVM from exiting.

Set using setDaemon(true) before calling start().

5. Thread Communication (Wait, Notify, NotifyAll)

Use wait(), notify(), and notifyAll() for inter-thread communication.

Threads can wait for a condition to be met and notify other threads when it's met.

6. Using Future and Callable for Asynchronous Tasks

Use Callable and Future for tasks that return a result or can throw exceptions.

Helps manage asynchronous execution and retrieve results.

Example: Future<Integer> result = executor.submit(task)

7-Thread Local Storage (ThreadLocal)

Use ThreadLocal to store thread-specific data, preventing data sharing between threads.

Example: ThreadLocal<Integer> threadLocal = ThreadLocal.withInitial(() -> 0);

8. Handling Thread Interruption

Use Thread.interrupted() or isInterrupted() to handle thread interruptions.

Important for canceling long-running tasks or handling thread termination.

9. Graceful Shutdown of Threads

Use shutdown() or shutdownNow() in ExecutorService to gracefully stop threads.

Ensures that threads finish executing or are safely interrupted.

10. Thread Safe Collections

Use thread-safe collections like ConcurrentHashMap or CopyOnWriteArrayList for safe data access in multi-threaded environments.

============================================================================

9---what is multithreading application

A multithreading application is an application that uses multiple threads to execute tasks concurrently within a single process.

Each thread represents an independent unit of execution, and multiple threads can run in parallel or share resources within the same program.

This can significantly improve performance, especially on multi-core processors, by utilizing multiple CPU cores to execute tasks simultaneously.

========================================================================================================================

Section 2:-----------------

1-:-

Trade Class has below

trade id

trade insurument

trade data;

trade value

Write the code to return

By instruemnt find the trade object highest trades for that instrument.

import java.util.\*;

import java.util.function.Function;

import java.util.stream.Collectors;

class Trade {

private String tradeId;

private String tradeInstrument;

private String tradeDate;

private double tradeValue;

public Trade(String tradeId, String tradeInstrument, String tradeDate, double tradeValue) {

this.tradeId = tradeId;

this.tradeInstrument = tradeInstrument;

this.tradeDate = tradeDate;

this.tradeValue = tradeValue;

}

public String getTradeId() {

return tradeId;

}

public String getTradeInstrument() {

return tradeInstrument;

}

public String getTradeDate() {

return tradeDate;

}

public double getTradeValue() {

return tradeValue;

}

@Override

public String toString() {

return "Trade{id='" + tradeId + "', instrument='" + tradeInstrument + "', date='" + tradeDate + "', value=" + tradeValue + "}";

}

}

public class TradeAnalyzer {

public static void main(String[] args) {

List<Trade> trades = Arrays.asList(

new Trade("T1", "Stock", "2025-01-15", 5000),

new Trade("T2", "Stock", "2025-01-16", 7000),

new Trade("T3", "Forex", "2025-01-14", 15000),

new Trade("T4", "Forex", "2025-01-17", 20000),

new Trade("T5", "Bond", "2025-01-18", 8000)

);

// Directly finding the highest trade for each instrument

Map<String, Optional<Trade>> highestTrades = trades.stream()

.collect(Collectors.groupingBy(

Trade::getTradeInstrument, // Group by instrument

Collectors.maxBy(Comparator.comparingDouble(Trade::getTradeValue)) // Get max value for each group

));

// Printing out the results

highestTrades.forEach((instrument, trade) ->

System.out.println("Highest trade for " + instrument + ": " +

(trade.isPresent() ? trade.get() : "No trades found"))

);

}

}

==========================================================

2:--

**Reverse the string with keeping special characters intact ( means dont change position of special character)**

Approach:

Use two pointers: one starting from the beginning of the string and the other from the end.

Move the left pointer until an alphabetic character is found.

Move the right pointer until an alphabetic character is found.

Swap the characters at the two pointers.

Repeat this process until the pointers meet.

public class ReverseStringWithSpecialCharacters {

public static String reverseStringWithSpecialCharacters(String input) {

// Convert the string to a character array for easy manipulation

char[] chars = input.toCharArray();

int left = 0;

int right = chars.length - 1;

// Traverse the array with two pointers

while (left < right) {

// Move left pointer to the next alphabetic character

if (!Character.isAlphabetic(chars[left])) {

left++;

}

// Move right pointer to the previous alphabetic character

else if (!Character.isAlphabetic(chars[right])) {

right--;

} else {

// Swap the alphabetic characters at left and right pointers

char temp = chars[left];

chars[left] = chars[right];

chars[right] = temp;

// Move both pointers inward

left++;

right--;

}

}

// Convert the character array back to a string

return new String(chars);

}

public static void main(String[] args) {

String input = "a,bCde#fgh!ij";

String result = reverseStringWithSpecialCharacters(input);

System.out.println("Original: " + input);

System.out.println("Reversed: " + result);

}

}

=================================================

**3: -------filter and sort Employee objects based on the given criteria using Java Streams API.**

Filtering and Sorting Employees:

Now, let's assume we want to:

Filter employees who are older than 30 years.

Filter employees who have a salary greater than $50,000.

Sort employees by their name in ascending order.

import java.util.\*;

import java.util.stream.Collectors;

public class EmployeeStreamExample {

public static void main(String[] args) {

List<Employee> employees = Arrays.asList(

new Employee("John", 28, 45000),

new Employee("Alice", 35, 55000),

new Employee("Bob", 40, 70000),

new Employee("Charlie", 25, 48000),

new Employee("David", 45, 85000),

new Employee("Eve", 30, 60000)

);

// Filter employees who are older than 30 and have a salary > 50000

List<Employee> filteredAndSortedEmployees = employees.stream()

.filter(e -> e.getAge() > 30) // Filter by age

.filter(e -> e.getSalary() > 50000) // Filter by salary

.sorted(Comparator.comparing(Employee::getName)) // Sort by name in ascending order

.collect(Collectors.toList());

// Print the filtered and sorted list

filteredAndSortedEmployees.forEach(System.out::println);

}

}

==================================

**4--------------**

**create a solution that groups employees into a Map where the key is the first character of**

**their first name and the value is a list of employees whose names start with that character**

import java.util.\*;

import java.util.stream.Collectors;

class Employee {

private String firstName;

private int age;

private double salary;

// Constructor

public Employee(String firstName, int age, double salary) {

this.firstName = firstName;

this.age = age;

this.salary = salary;

}

// Getter for firstName

public String getFirstName() {

return firstName;

}

// toString for easy printing

@Override

public String toString() {

return "Employee{firstName='" + firstName + "', age=" + age + ", salary=" + salary + '}';

}

}

public class EmployeeGroupingExample {

public static void main(String[] args) {

List<Employee> employees = Arrays.asList(

new Employee("Alice", 30, 50000),

new Employee("Bob", 28, 60000),

new Employee("Anna", 35, 55000),

new Employee("Charlie", 40, 70000),

new Employee("David", 45, 85000)

);

// Group employees by the first letter of their first name

Map<Character, List<Employee>> groupedByFirstLetter = employees.stream()

.collect(Collectors.groupingBy(e -> e.getFirstName().charAt(0)));

// Print the result

groupedByFirstLetter.forEach((letter, employeeList) -> {

System.out.println(letter + " -> " + employeeList);

});

}

}

================================================================

**5------------Spring data JPA**

Spring Data JPA makes working with relational databases easy by abstracting common data access patterns. By extending interfaces

like JpaRepository or CrudRepository, you can quickly implement CRUD operations and customize queries using method naming

conventions or JPQL queries.

In this walkthrough, we demonstrated:

Setting up an entity class with JPA annotations.

Defining a repository interface that extends JpaRepository.

Implementing a service layer for business logic.

Exposing API endpoints via a controller.

Spring Data JPA significantly reduces the boilerplate code required to interact with databases and is a powerful tool for building robust

data-driven applications.

=================================================================

6:- Does spring helps in creating queries ? concise

Yes, Spring Data JPA helps in creating queries in several ways:

Query Methods:

You can create queries by simply defining method names in the repository interface.

Spring Data JPA generates the query for you based on the method name.

Example: findByLastName(String lastName) will generate a query like SELECT \* FROM Employee WHERE lastName = ?.

JPQL Queries:

You can use @Query annotation to write custom JPQL (Java Persistence Query Language) queries.

Example: @Query("SELECT e FROM Employee e WHERE e.department = :dept") creates a custom query.

Native SQL Queries:

You can also use @Query(nativeQuery = true) to execute native SQL queries directly on the database.

Pagination and Sorting:

Spring Data JPA supports pagination and sorting without writing custom queries by using Pageable and Sort arguments.

In summary,

Spring Data JPA allows for both automatic query generation and custom queries via method names or annotations,

reducing the amount of boilerplate code needed for database interactions.

===========================================================

**7----Then sql query for the same data fetching example**

Conclusion:

You can use native SQL queries in Spring Data JPA with the @Query annotation and the nativeQuery = true flag.

This allows you to write SQL queries directly, providing more control over the database interactions,

especially when JPQL might not be sufficient.

=====================================================================

8----------------

write a SQL query which would just give me the name of the person, and the name of the managers.

Given a employee table with employee id and manager id

[12:02, 6/1/2025] Angela:

Extension to above question add a dept table and write a query to find department name,

the employee name who draws the maximum salary, and the salary that is actually drawing.

SELECT e.employee\_name AS employee\_name, m.employee\_name AS manager\_name

FROM employee e

LEFT JOIN employee m ON e.manager\_id = m.employee\_id;

SELECT d.department\_name,

e.employee\_name,

e.salary

FROM department d

JOIN employee e ON d.dept\_id = e.dept\_id

WHERE e.salary = (

SELECT MAX(salary)

FROM employee

WHERE dept\_id = d.dept\_id

);

===========================================================================

9**:------Kubernetes, docker, communication between pods, containers, how communication happens if the ip is dynamic**

In a Kubernetes environment, the dynamic nature of pod IPs can make direct communication between pods more challenging. However,

Kubernetes provides several mechanisms to ensure that communication between pods (or containers) happens seamlessly, even when their IP addresses

change. Here's how Kubernetes handles communication in a dynamic IP scenario:

1. Pod IPs Are Dynamic

When a pod is created, it gets a unique IP address assigned to it. However, if the pod is deleted and recreated (for example,

during scaling or due to pod failures), the IP address of the new pod may be different from the old one.

These IP addresses are allocated dynamically from a pool and are ephemeral, meaning they can change frequently.

2. Service Abstraction and DNS for Communication

Rather than directly using dynamic pod IPs for communication, Kubernetes provides a service abstraction to simplify communication between pods:

Services in Kubernetes are designed to group together a set of pods and expose a stable, consistent interface (usually a DNS name)

for accessing them.

A Service (e.g., my-service) has its own ClusterIP, which remains constant even if the IPs of the underlying pods change. This ensures

that clients can communicate with a service reliably, without needing to worry about the dynamic IPs of individual pods.

Example:

Pod pod-1 may be assigned IP 10.0.0.1, and Pod pod-2 may be assigned IP 10.0.0.2.

Instead of directly referring to 10.0.0.1 or 10.0.0.2, a Service (e.g., my-service) provides a stable IP address (e.g., 10.96.0.1) and/or DNS

name (my-service.default.svc.cluster.local).

Pods that need to communicate can refer to my-service instead of the dynamic IPs of individual pods. Kubernetes ensures that traffic directed

to the Service IP is correctly routed to one of the available pods.

=====================

**10:-- why uses factory patterns**

Why Use the Factory Pattern?

Separation of Concerns: The factory pattern abstracts the object creation process from the client code,

allowing the code to focus on using the object without knowing its specific class.

Flexibility: It allows new product types to be added without changing the code that uses the factory, adhering to the Open/Closed Principle.

Encapsulation: The object creation logic is centralized in the factory, which simplifies object instantiation and improves maintainability.

===================================================================================

**section 3:-------------------**

**1. What are the wrapper classes in Java (Integer, Boolean etc.) and why they are required**

Wrapper classes in Java are part of the java.lang package and provide object representations for the primitive data types.

Each primitive type has a corresponding wrapper class:

byte → Byte

short → Short

int → Integer

long → Long

float → Float

double → Double

char → Character

boolean → Boolean

Why Wrapper Classes Are Required:

Object-Oriented Features: Java is an object-oriented language, and many of the standard library classes or APIs require objects.

Since primitive types are not objects, wrapper classes provide a way to use primitives in contexts that require objects,

such as in collections like ArrayList, HashMap, etc.

Autoboxing and Unboxing: Java introduced autoboxing and unboxing in Java 5. Autoboxing is the automatic conversion

of a primitive type to its corresponding wrapper class (e.g., converting int to Integer), and unboxing is the reverse process

(e.g., converting Integer to int). This allows seamless integration between primitive types and objects.

Nullability: Primitive types cannot be null, while wrapper objects can be. This is useful when working with databases,

serialization, or other systems where null values may be necessary to represent the absence of a value.

Utility Methods: Wrapper classes provide utility methods for converting between types, parsing values, and handling

specific operations. For example:

Integer.parseInt() to convert a String to an int.

Integer.valueOf() to create an Integer from a String.

Boolean.parseBoolean() to convert a String to a boolean.

Generics: In Java, generics work with objects, not primitive types. Therefore, wrapper classes are used when working with

collections or other generic classes. For instance, List<Integer> is valid, but List<int> is not.

In summary, wrapper classes allow primitive types to be used as objects, enabling them to work in object-oriented contexts,

provide additional functionality, and support features like nullability and generics.

==========================================================

**2- What is immutable class?**

An immutable class in Java is a class whose objects cannot be modified after they are created.

Key Characteristics:

Final class: The class is declared final to prevent subclassing.

Final fields: All fields are final, ensuring they cannot be changed after initialization.

No setter methods: There are no methods that modify the object's state.

Proper initialization: Fields are initialized through the constructor, and their values cannot be changed.

Defensive copying: If the class contains mutable objects, those objects are copied rather than directly referenced.

Example:

public final class Person {

private final String name;

private final int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

public String getName() {

return name;

}

public int getAge() {

return age;

}

}

Why Immutable Classes?

Thread safety: Immutable objects can be shared across threads without synchronization.

Security: Prevents modification of the object's state after creation.

Reliability: Reduces errors due to unintended state changes.

===================================

**3:- Write an implementation of immutable class of Student with fields String name, int regnum, Map<String, String> metadata**

Here’s a concise implementation of an immutable class Student in Java:

import java.util.Collections;

import java.util.Map;

public final class Student {

private final String name;

private final int regnum;

private final Map<String, String> metadata;

// Constructor to initialize fields

public Student(String name, int regnum, Map<String, String> metadata) {

this.name = name;

this.regnum = regnum;

// Create a defensive copy of the metadata to ensure immutability

this.metadata = metadata != null ? Collections.unmodifiableMap(metadata) : null;

}

// Getter methods

public String getName() {

return name;

}

public int getRegnum() {

return regnum;

}

public Map<String, String> getMetadata() {

return metadata;

}

}

Key Points:

Final class: Student is declared as final to prevent subclassing.

Final fields: name, regnum, and metadata are final, ensuring they are immutable.

Defensive Copying: metadata is wrapped with Collections.unmodifiableMap() to prevent external modifications.

If metadata is mutable, this ensures immutability.

=========================================================

**4:- Write a program. Given a string, find the largest substring with no repeating character e.g. String s = "pwwekw" output: "wek"**

import java.util.HashSet;

public class Test {

public static void main(String[] args) {

String s = "abcdabcbb";

int start = 0;

int maxLength = 0;

int maxStart = 0; // To store the start index of the largest substring

HashSet<Character> set = new HashSet<>();

for (int end = 0; end < s.length(); end++) {

char currentChar = s.charAt(end);

// Move the start index to the right of the duplicate character

while (set.contains(currentChar)) {

set.remove(s.charAt(start));

start++;

}

set.add(currentChar);

// Update the maximum length and the start index of the largest substring

if (end - start + 1 > maxLength) {

maxLength = end - start + 1;

maxStart = start;

}

}

// Print the largest substring

String largestSubstring = s.substring(maxStart, maxStart + maxLength);

System.out.println("Largest substring: " + largestSubstring);

}

}

==================================================

**5:-What is spring boot?**

Spring Boot is a framework that simplifies the development of Java applications using the Spring framework.

It provides production-ready defaults for application setup, configuration, and deployment, reducing the need for boilerplate code

and configuration. Spring Boot allows you to create stand-alone, production-grade Spring-based applications with minimal effort.

Key Features:

Auto-Configuration: Automatically configures Spring application context based on dependencies in the classpath.

Embedded Web Servers: Comes with embedded servers like Tomcat, Jetty, or Undertow, which means you don't

need to deploy the application to a separate server.

Spring Initializr: Provides a web-based tool (start.spring.io) for quickly generating a project structure.

Microservices Ready: Designed to support microservices architecture with easy integration of Spring Cloud components.

Minimal Configuration: No need for web.xml or applicationContext.xml. Most configurations are auto-detected.

==================================

**6 What is the difference between @Controller and @RestController annotations?**

Key Differences:

Aspect @Controller @RestController

Purpose For MVC controllers (views) For RESTful web services (JSON/XML responses)

Return Type Returns view names (e.g., ModelAndView) Returns response data (e.g., JSON, XML)

@ResponseBody Must be explicitly added to methods for response Implicitly added to all methods

Use Case Traditional web applications with views APIs that return data, typically JSON/XML

In summary,

use @Controller when you're working with traditional web applications that return views,

and use @RestController when you're creating REST APIs that return data directly.

====================================================================

**7:- Explain SAGA design pattern in microservices**

SAGA Design Pattern in Microservices

The SAGA pattern is a design pattern used to manage distributed transactions in a microservices architecture.

In traditional monolithic applications, you might use a single database transaction to ensure consistency.

However, in a microservices architecture, each service has its own database, so distributed transactions can be problematic.

The SAGA pattern helps to ensure data consistency across multiple services without using a distributed transaction coordinator

like 2PC (Two-Phase Commit). Instead, it breaks the transaction into a series of smaller, isolated transactions that are managed

by the services themselves.

Types of SAGA

Choreography-Based SAGA:

Each service involved in the SAGA knows which service to call next.

There is no central coordinator, and services communicate by emitting events and listening to events.

This is event-driven and more decentralized.

Orchestration-Based SAGA:

A central service (called the orchestrator) controls the flow of the SAGA by explicitly telling each service what to do next.

SAGA in Microservices

The SAGA pattern helps in maintaining consistency in the system by using compensating actions to roll back changes in case of failure.

For example, if a service in the middle of a chain of services fails, the preceding services can call compensating transactions to

undo their changes.

Steps in SAGA

Start the Saga: A transaction is initiated from the first service.

Perform Local Transactions: Each service performs its own local transaction (such as creating an order, payment, or shipping).

Compensating Transactions: If any service fails, a compensating transaction is triggered to undo any changes made by the previous services

Choreography-Based SAGA Example

Let’s create a simple choreography-based SAGA pattern in a microservices setup for a Order service, Payment service, and Shipping service.

Step-by-Step Breakdown:

Order Service initiates the SAGA (order creation).

Payment Service processes the payment.

If Payment succeeds, the Shipping Service is called to ship the product.

If any service fails, a compensating transaction (like a refund) is triggered.

Example Code for Choreography-Based SAGA

1. Order Service (Order Service)

This service initiates the process and sends an event to start the payment process.

@RestController

public class OrderController {

private final ApplicationEventPublisher eventPublisher;

@Autowired

public OrderController(ApplicationEventPublisher eventPublisher) {

this.eventPublisher = eventPublisher;

}

@PostMapping("/order")

public ResponseEntity<String> createOrder(@RequestBody Order order) {

// Step 1: Create the order (local transaction)

Order createdOrder = orderService.createOrder(order);

// Step 2: Emit an event to trigger the payment process

eventPublisher.publishEvent(new PaymentInitiatedEvent(createdOrder.getId()));

return ResponseEntity.ok("Order created. Waiting for payment...");

}

}

2. Payment Service (Payment Service)

This service listens for the PaymentInitiatedEvent and processes the payment.

@Component

public class PaymentService {

@EventListener

public void onPaymentInitiated(PaymentInitiatedEvent event) {

// Step 3: Process the payment (local transaction)

boolean paymentSuccess = paymentGateway.processPayment(event.getOrderId());

if (paymentSuccess) {

// Step 4: If payment is successful, emit a Shipping event

eventPublisher.publishEvent(new ShippingInitiatedEvent(event.getOrderId()));

} else {

// Step 5: If payment fails, trigger a compensation event

eventPublisher.publishEvent(new PaymentFailedEvent(event.getOrderId()));

}

}

}

3. Shipping Service (Shipping Service)

This service listens for the ShippingInitiatedEvent and ships the product.

@Component

public class ShippingService {

@EventListener

public void onShippingInitiated(ShippingInitiatedEvent event) {

// Step 6: Ship the product (local transaction)

boolean shipmentSuccess = shippingGateway.shipProduct(event.getOrderId());

if (!shipmentSuccess) {

// Step 7: If shipping fails, trigger compensating action to cancel the order

eventPublisher.publishEvent(new ShippingFailedEvent(event.getOrderId()));

}

}

}

4. Compensating Actions

In case of failure in any step, compensating transactions are triggered.

@Component

public class CompensationService {

@EventListener

public void onPaymentFailed(PaymentFailedEvent event) {

// Compensating action for failed payment: refund

refundService.refund(event.getOrderId());

}

@EventListener

public void onShippingFailed(ShippingFailedEvent event) {

// Compensating action for failed shipping: cancel order

orderService.cancelOrder(event.getOrderId());

}

}

Flow of Events

The Order Service creates an order and publishes the PaymentInitiatedEvent.

The Payment Service processes the payment. If successful, it emits the ShippingInitiatedEvent.

The Shipping Service ships the product. If any failure occurs, compensating actions are triggered (e.g., order cancellation or refund).

Advantages of SAGA Pattern

Scalability: Each service manages its own transaction, so there’s no single point of failure.

Flexibility: You can handle failure gracefully by performing compensating actions.

Decentralization: No need for a centralized transaction coordinator like 2PC.

Challenges of SAGA Pattern

Complexity: Managing and implementing compensating transactions for failures can be complex.

Eventual Consistency: Unlike traditional transactions, SAGA may lead to eventual consistency, meaning the system may be

in an inconsistent state for a short period of time.

Long-running Transactions: As each service executes asynchronously, the system may involve long-running transactions,

which require careful management.

Conclusion:

The SAGA pattern provides a way to handle distributed transactions in a microservices environment by breaking them down

into smaller local transactions.

With choreography, services communicate using events, and compensating transactions are used to handle failures and ensure consistency.

-------------------------------------------------------------------------------

using kafka example :------ orchestrator

Sure! Below is a simple example of how to implement the Saga Pattern using Kafka in Java.

In this example, we are creating a basic saga pattern with three services. The services are:

Order Service: This service will initiate the saga.

Payment Service: This service will handle the payment processing.

Shipping Service: This service will handle shipping after successful payment.

Each service will be represented by a Kafka consumer and producer to simulate a distributed transaction using Kafka topics.

Prerequisites:

Kafka running locally or in a distributed environment.

Kafka dependencies in pom.xml.

Java 8 or higher.

Maven Dependencies (pom.xml):

<dependencies>

<!-- Spring Boot dependencies for Kafka -->

<dependency>

<groupId>org.springframework.kafka</groupId>

<artifactId>spring-kafka</artifactId>

<version>2.8.0</version> <!-- Adjust as needed -->

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

<groupId>org.springframework.kafka</groupId>

<artifactId>spring-kafka-test</artifactId>

<version>2.8.0</version>

<scope>test</scope>

</dependency>

</dependencies>

1 Saga Orchestrator (Order Service):

This service orchestrates the saga by sending messages to different services based on the result of the transaction.

import org.springframework.kafka.core.KafkaTemplate;

import org.springframework.web.bind.annotation.\*;

@RestController

@RequestMapping("/order")

public class OrderService {

private final KafkaTemplate<String, String> kafkaTemplate;

public OrderService(KafkaTemplate<String, String> kafkaTemplate) {

this.kafkaTemplate = kafkaTemplate;

}

@PostMapping("/create")

public String createOrder(@RequestBody Order order) {

// Step 1: Initiating the saga

kafkaTemplate.send("order-created", order.getId());

return "Order created and saga initiated.";

}

}

2- Payment Service:

This service listens for an order-created message and processes payment.

import org.springframework.kafka.annotation.KafkaListener;

import org.springframework.kafka.core.KafkaTemplate;

import org.springframework.stereotype.Service;

@Service

public class PaymentService {

private final KafkaTemplate<String, String> kafkaTemplate;

public PaymentService(KafkaTemplate<String, String> kafkaTemplate) {

this.kafkaTemplate = kafkaTemplate;

}

@KafkaListener(topics = "order-created", groupId = "payment-service")

public void handleOrderCreated(String orderId) {

System.out.println("Processing payment for order: " + orderId);

// Simulate payment processing logic

boolean paymentSuccess = processPayment(orderId);

if (paymentSuccess) {

kafkaTemplate.send("payment-success", orderId);

} else {

kafkaTemplate.send("payment-failure", orderId);

}

}

private boolean processPayment(String orderId) {

// Simulate payment logic (for demonstration purposes)

return true; // Simulating successful payment

}

}

Shipping Service:

This service listens for a payment-success message and proceeds with the shipping.

import org.springframework.kafka.annotation.KafkaListener;

import org.springframework.kafka.core.KafkaTemplate;

import org.springframework.stereotype.Service;

@Service

public class ShippingService {

private final KafkaTemplate<String, String> kafkaTemplate;

public ShippingService(KafkaTemplate<String, String> kafkaTemplate) {

this.kafkaTemplate = kafkaTemplate;

}

@KafkaListener(topics = "payment-success", groupId = "shipping-service")

public void handlePaymentSuccess(String orderId) {

System.out.println("Processing shipping for order: " + orderId);

// Simulate shipping processing logic

shipOrder(orderId);

}

private void shipOrder(String orderId) {

// Simulate shipping logic

System.out.println("Shipping order: " + orderId);

}

}

Compensation in Case of Failure:

In case of any failure in the saga, compensation actions should be triggered. For example, if payment fails, the order can be canceled.

Order Service (Handling Payment Failure):

import org.springframework.kafka.annotation.KafkaListener;

@Service

public class OrderService {

private final KafkaTemplate<String, String> kafkaTemplate;

public OrderService(KafkaTemplate<String, String> kafkaTemplate) {

this.kafkaTemplate = kafkaTemplate;

}

@KafkaListener(topics = "payment-failure", groupId = "order-service")

public void handlePaymentFailure(String orderId) {

System.out.println("Payment failed for order: " + orderId);

// Compensate the order creation (cancel the order)

cancelOrder(orderId);

}

private void cancelOrder(String orderId) {

// Simulate canceling the order

System.out.println("Order " + orderId + " has been canceled due to payment failure.");

}

}

Application Configuration (application.properties):

properties

Copy

spring.kafka.bootstrap-servers=localhost:9092

spring.kafka.consumer.group-id=order-service

spring.kafka.consumer.auto-offset-reset=earliest

Kafka Topics:

You need to create Kafka topics for this example:

order-created

payment-success

payment-failure

You can create these topics using the Kafka command line or programmatically.

Running the Services:

Make sure Kafka is running.

Start each service as a Spring Boot application.

Use a tool like Postman or cURL to send a POST request to the /order/create endpoint to initiate the saga.

Example Request:

bash

Copy

curl -X POST http://localhost:8080/order/create -H "Content-Type: application/json" -d '{"id": "order123"}'

This should trigger the saga, and the Order Service will initiate the saga, the Payment Service will process the payment,

and the Shipping Service will handle shipping once the payment is successful.

Error Handling and Compensation:

In this example, if the payment fails (by sending a message to the payment-failure topic), the order will be canceled by listening

for that failure message and compensating for it.

Conclusion:

This basic example demonstrates how to implement the Saga pattern using Kafka in Java. The key idea behind the Saga pattern

is that instead of having a single monolithic transaction across services, you break the transaction into smaller steps,

with each service managing its part and compensating if something goes wrong. Kafka helps us coordinate these distributed services and events.

=====================================================================

2-------------------- choregraphy using kafka example in java

Certainly! In a choreography-based Saga pattern, there is no central orchestrator like in the previous example.

Instead, each service interacts directly with others by listening for events and taking actions accordingly.

Services communicate through events, and each service performs its part of the transaction.

In this pattern, services are more loosely coupled and independently handle the success or failure of the saga.

Scenario

We'll use a simple example with three services:

Order Service: Initiates the saga by creating an order.

Payment Service: Processes payments.

Shipping Service: Handles shipping after successful payment.

Each service will react to events by publishing events to Kafka and consuming events from other services.

Kafka Topics:

order-created: Order service sends an event when an order is created.

payment-success: Payment service sends an event when payment is successful.

payment-failure: Payment service sends an event when payment fails.

shipping-ready: Shipping service sends an event when the order is ready to ship.

Step-by-Step Implementation:

Maven Dependencies (pom.xml):

xml

Copy

<dependencies>

<!-- Spring Boot Kafka dependencies -->

<dependency>

<groupId>org.springframework.kafka</groupId>

<artifactId>spring-kafka</artifactId>

<version>2.8.0</version> <!-- Adjust based on the Spring Kafka version -->

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

</dependencies>

1-----Order Service (initiates saga and listens for payment result):

import org.springframework.kafka.core.KafkaTemplate;

import org.springframework.kafka.annotation.KafkaListener;

import org.springframework.web.bind.annotation.\*;

@RestController

@RequestMapping("/order")

public class OrderService {

private final KafkaTemplate<String, String> kafkaTemplate;

public OrderService(KafkaTemplate<String, String> kafkaTemplate) {

this.kafkaTemplate = kafkaTemplate;

}

// Step 1: Create order and initiate the saga

@PostMapping("/create")

public String createOrder(@RequestBody Order order) {

kafkaTemplate.send("order-created", order.getId()); // Send order-created event

return "Order created and saga initiated.";

}

// Step 2: Listen for payment failure and handle compensation

@KafkaListener(topics = "payment-failure", groupId = "order-service")

public void handlePaymentFailure(String orderId) {

System.out.println("Payment failed for order: " + orderId);

// Compensate for the failure (e.g., cancel the order)

cancelOrder(orderId);

}

private void cancelOrder(String orderId) {

System.out.println("Order " + orderId + " has been canceled due to payment failure.");

}

}

2-----------Payment Service (processes payment and sends events based on success/failure):

import org.springframework.kafka.annotation.KafkaListener;

import org.springframework.kafka.core.KafkaTemplate;

import org.springframework.stereotype.Service;

@Service

public class PaymentService {

private final KafkaTemplate<String, String> kafkaTemplate;

public PaymentService(KafkaTemplate<String, String> kafkaTemplate) {

this.kafkaTemplate = kafkaTemplate;

}

// Step 3: Listen for order-created event and process payment

@KafkaListener(topics = "order-created", groupId = "payment-service")

public void handleOrderCreated(String orderId) {

System.out.println("Processing payment for order: " + orderId);

// Simulate payment processing

boolean paymentSuccess = processPayment(orderId);

if (paymentSuccess) {

kafkaTemplate.send("payment-success", orderId); // Send payment-success event

} else {

kafkaTemplate.send("payment-failure", orderId); // Send payment-failure event

}

}

private boolean processPayment(String orderId) {

// Simulate payment logic (for this example, always return true for success)

return true; // Simulating successful payment

}

}

3----------Shipping Service (reacts to payment success and starts shipping):

import org.springframework.kafka.annotation.KafkaListener;

import org.springframework.kafka.core.KafkaTemplate;

import org.springframework.stereotype.Service;

@Service

public class ShippingService {

private final KafkaTemplate<String, String> kafkaTemplate;

public ShippingService(KafkaTemplate<String, String> kafkaTemplate) {

this.kafkaTemplate = kafkaTemplate;

}

// Step 4: Listen for payment-success event and start shipping

@KafkaListener(topics = "payment-success", groupId = "shipping-service")

public void handlePaymentSuccess(String orderId) {

System.out.println("Processing shipping for order: " + orderId);

shipOrder(orderId);

kafkaTemplate.send("shipping-ready", orderId); // Send shipping-ready event

}

private void shipOrder(String orderId) {

// Simulate shipping logic

System.out.println("Shipping order: " + orderId);

}

}

Kafka Topics: We need to create Kafka topics to handle the events:

order-created

payment-success

payment-failure

shipping-ready

You can create these topics manually via Kafka CLI:

bash

Copy

kafka-topics --create --topic order-created --partitions 1 --replication-factor 1 --bootstrap-server localhost:9092

kafka-topics --create --topic payment-success --partitions 1 --replication-factor 1 --bootstrap-server localhost:9092

kafka-topics --create --topic payment-failure --partitions 1 --replication-factor 1 --bootstrap-server localhost:9092

kafka-topics --create --topic shipping-ready --partitions 1 --replication-factor 1 --bootstrap-server localhost:9092

Application Configuration (application.properties):

properties

Copy

spring.kafka.bootstrap-servers=localhost:9092

spring.kafka.consumer.group-id=order-service

spring.kafka.consumer.auto-offset-reset=earliest

Main Class:

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class SagaChoreographyApplication {

public static void main(String[] args) {

SpringApplication.run(SagaChoreographyApplication.class, args);

}

}

Order Model (Order.java):

java

Copy

public class Order {

private String id;

// Getters and Setters

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

}

Flow of the Saga (Choreography):

Order Creation:

A POST request is sent to /order/create to create an order.

The OrderService sends an order-created event to Kafka.

Payment Processing:

The PaymentService listens for the order-created event.

The PaymentService processes the payment and sends either a payment-success or payment-failure event based on the payment outcome.

Shipping Processing:

If payment is successful, the ShippingService listens for the payment-success event.

It processes the shipping and sends a shipping-ready event once the shipping is done.

Failure Handling (Compensation):

If payment fails, the OrderService listens for the payment-failure event and compensates by canceling the order.

Example cURL Requests:

Create an order:

bash

Copy

curl -X POST http://localhost:8080/order/create -H "Content-Type: application/json" -d '{"id": "order123"}'

The services will respond with logs and process the events accordingly:

OrderService will initiate the saga.

PaymentService will process payment.

ShippingService will ship the order upon successful payment.

Conclusion:

In this example, we have implemented a choreography-based Saga using Kafka.

Each service reacts to events from other services, and there is no central orchestrator.

This decentralized approach ensures that each service handles its part of the business logic and compensates for failures independently.

Kafka acts as the medium for event-based communication between services.

=========================================================================================================================================

Section ------4

need to study :---

**1:- Having a list of transaction having attribute like instrumentId, transaction date, value of transaction**

Write code to group transactions by instrumentID and tradeDate and filter and out of those find transactions

with maximum transaction amount on each trade date from those transactions.

import java.time.LocalDate;

import java.util.\*;

import java.util.stream.Collectors;

// Define the Transaction class

class Transaction {

private String instrumentId;

private LocalDate tradeDate;

private double tradeValue;

// Constructor

public Transaction(String instrumentId, LocalDate tradeDate, double tradeValue) {

this.instrumentId = instrumentId;

this.tradeDate = tradeDate;

this.tradeValue = tradeValue;

}

// Getters

public String getInstrumentId() {

return instrumentId;

}

public LocalDate getTradeDate() {

return tradeDate;

}

public double getTradeValue() {

return tradeValue;

}

@Override

public String toString() {

return "InstrumentId: " + instrumentId + ", Date: " + tradeDate + ", Value: " + tradeValue;

}

}

public class TransactionReport {

public static void main(String[] args) {

// Sample transactions

List<Transaction> transactions = Arrays.asList(

new Transaction("A123", LocalDate.of(2025, 1, 19), 1000.50),

new Transaction("A123", LocalDate.of(2025, 1, 19), 1500.75),

new Transaction("B456", LocalDate.of(2025, 1, 19), 2000.00),

new Transaction("A123", LocalDate.of(2025, 1, 20), 1200.00),

new Transaction("A123", LocalDate.of(2025, 1, 20), 1100.00),

new Transaction("B456", LocalDate.of(2025, 1, 20), 2500.00),

new Transaction("A123", LocalDate.of(2025, 1, 21), 800.00),

new Transaction("B456", LocalDate.of(2025, 1, 21), 1200.00)

);

// Step 1: Group by instrumentId and tradeDate

Map<String, Map<LocalDate, List<Transaction>>> groupedTransactions = transactions.stream()

.collect(Collectors.groupingBy(

Transaction::getInstrumentId,

Collectors.groupingBy(Transaction::getTradeDate)

));

// Step 2: For each group (instrumentId and tradeDate), find the transaction with the maximum tradeValue

groupedTransactions.forEach((instrumentId, dateMap) -> {

System.out.println("Instrument ID: " + instrumentId);

dateMap.forEach((date, tradeList) -> {

// Find the transaction with the maximum tradeValue for that tradeDate

Optional<Transaction> maxTransaction = tradeList.stream()

.max(Comparator.comparingDouble(Transaction::getTradeValue));

maxTransaction.ifPresent(transaction ->

System.out.println(" Date: " + date + ", Max Trade: " + transaction)

);

});

});

}

}

Explanation:

Transaction Class:

The Transaction class contains instrumentId, tradeDate, and tradeValue.

We have getters for these fields and a toString() method to output the transaction details.

Grouping Transactions:

We use Collectors.groupingBy() to group the transactions first by instrumentId and then by tradeDate.

The result is a Map<String, Map<LocalDate, List<Transaction>>>.

Finding Maximum Trade:

For each group of transactions (grouped by instrumentId and tradeDate), we use stream() and max()

with a Comparator.comparingDouble() to find the transaction with the highest tradeValue for that date.

We use Optional<Transaction> to handle the case when there are no transactions (although this won't happen in this example,

as the list is never empty).

Output:

The program prints the maximum transaction for each tradeDate within each instrumentId.

Sample Output:

Instrument ID: A123

Date: 2025-01-19, Max Trade: InstrumentId: A123, Date: 2025-01-19, Value: 1500.75

Date: 2025-01-20, Max Trade: InstrumentId: A123, Date: 2025-01-20, Value: 1200.0

Date: 2025-01-21, Max Trade: InstrumentId: A123, Date: 2025-01-21, Value: 800.0

Instrument ID: B456

Date: 2025-01-19, Max Trade: InstrumentId: B456, Date: 2025-01-19, Value: 2000.0

Date: 2025-01-20, Max Trade: InstrumentId: B456, Date: 2025-01-20, Value: 2500.0

Date: 2025-01-21, Max Trade: InstrumentId: B456, Date: 2025-01-21, Value: 1200.0

=============================================================

2----

Trade Class has below

trade id

trade instrument

trade data;

trade value

Write the stream api code to return a map

Certainly! There are other ways you can approach collecting data into a Map using the Stream API,

depending on what you're trying to achieve. Below are a few different approaches you can take to return a map,

each focusing on different aspects:

1. Mapping Trade ID to Trade Instrument

If you want a map where the tradeId is the key and the tradeInstrument is the value,

you can use a method reference for the value as well. Here's an example:

import java.util.\*;

import java.util.stream.\*;

import java.math.BigDecimal;

import java.time.LocalDate;

public class Main {

public static void main(String[] args) {

List<Trade> trades = Arrays.asList(

new Trade("T1", "Stock", LocalDate.of(2025, 1, 15), new BigDecimal("1000")),

new Trade("T2", "Bond", LocalDate.of(2025, 1, 16), new BigDecimal("2000")),

new Trade("T3", "Option", LocalDate.of(2025, 1, 17), new BigDecimal("3000"))

);

// Collecting into a map with tradeId as key and tradeInstrument as value

Map<String, String> tradeInstrumentMap = trades.stream()

.collect(Collectors.toMap(Trade::getTradeId, Trade::getTradeInstrument));

// Printing the result

tradeInstrumentMap.forEach((key, value) -> System.out.println(key + " -> " + value));

}

}

class Trade {

private String tradeId;

private String tradeInstrument;

private LocalDate tradeDate;

private BigDecimal tradeValue;

public Trade(String tradeId, String tradeInstrument, LocalDate tradeDate, BigDecimal tradeValue) {

this.tradeId = tradeId;

this.tradeInstrument = tradeInstrument;

this.tradeDate = tradeDate;

this.tradeValue = tradeValue;

}

public String getTradeId() {

return tradeId;

}

public String getTradeInstrument() {

return tradeInstrument;

}

public LocalDate getTradeDate() {

return tradeDate;

}

public BigDecimal getTradeValue() {

return tradeValue;

}

}

b part :---- By tradeinstrument find the trade object with highest trade value for that instrument.

import java.util.\*;

import java.util.function.\*;

import java.util.stream.\*;

import java.math.BigDecimal;

import java.time.LocalDate;

public class Main {

public static void main(String[] args) {

List<Trade> trades = Arrays.asList(

new Trade("T1", "Stock", LocalDate.of(2025, 1, 15), new BigDecimal("1000")),

new Trade("T2", "Bond", LocalDate.of(2025, 1, 16), new BigDecimal("2000")),

new Trade("T3", "Stock", LocalDate.of(2025, 1, 17), new BigDecimal("3000")),

new Trade("T4", "Bond", LocalDate.of(2025, 1, 18), new BigDecimal("1500")),

new Trade("T5", "Option", LocalDate.of(2025, 1, 19), new BigDecimal("5000"))

);

// Group trades by tradeInstrument and find the trade with the highest tradeValue for each instrument

Map<String, Optional<Trade>> highestTradeByInstrument = trades.stream()

.collect(Collectors.groupingBy(

Trade::getTradeInstrument,

Collectors.maxBy(Comparator.comparing(Trade::getTradeValue))

));

// Print the result

highestTradeByInstrument.forEach((instrument, trade) -> {

trade.ifPresent(t -> System.out.println(instrument + " -> " + t.getTradeId() + " with value " + t.getTradeValue()));

});

}

}

class Trade {

private String tradeId;

private String tradeInstrument;

private LocalDate tradeDate;

private BigDecimal tradeValue;

public Trade(String tradeId, String tradeInstrument, LocalDate tradeDate, BigDecimal tradeValue) {

this.tradeId = tradeId;

this.tradeInstrument = tradeInstrument;

this.tradeDate = tradeDate;

this.tradeValue = tradeValue;

}

public String getTradeId() {

return tradeId;

}

public String getTradeInstrument() {

return tradeInstrument;

}

public LocalDate getTradeDate() {

return tradeDate;

}

public BigDecimal getTradeValue() {

return tradeValue;

}

}

c------------

**4:---------Find the combination of instrumentId and date with 2nd maximum price for the below code :-**

import java.time.LocalDate;

import java.util.\*;

import java.util.stream.Collectors;

// Define the Trade class

class Trade {

private String tradeId;

private String tradeInstrument;

private LocalDate tradeDate;

private double tradeValue;

// Constructor

public Trade(String tradeId, String tradeInstrument, LocalDate tradeDate, double tradeValue) {

this.tradeId = tradeId;

this.tradeInstrument = tradeInstrument;

this.tradeDate = tradeDate;

this.tradeValue = tradeValue;

}

// Getters

public String getTradeId() {

return tradeId;

}

public String getTradeInstrument() {

return tradeInstrument;

}

public LocalDate getTradeDate() {

return tradeDate;

}

public double getTradeValue() {

return tradeValue;

}

@Override

public String toString() {

return "TradeId: " + tradeId + ", Instrument: " + tradeInstrument + ", Date: " + tradeDate + ", Value: " + tradeValue;

}

}

public class TradeReport {

public static void main(String[] args) {

// Sample trades

List<Trade> trades = Arrays.asList(

new Trade("T1", "AAPL", LocalDate.of(2025, 1, 19), 1000.50),

new Trade("T2", "AAPL", LocalDate.of(2025, 1, 20), 1500.75),

new Trade("T3", "GOOG", LocalDate.of(2025, 1, 19), 2000.00),

new Trade("T4", "AAPL", LocalDate.of(2025, 1, 21), 1200.00),

new Trade("T5", "GOOG", LocalDate.of(2025, 1, 20), 3000.00),

new Trade("T6", "AAPL", LocalDate.of(2025, 1, 21), 1800.00),

new Trade("T7", "AAPL", LocalDate.of(2025, 1, 19), 1200.00),

new Trade("T8", "GOOG", LocalDate.of(2025, 1, 21), 2500.00)

);

// Step 1: Group by instrument and date

Map<String, Map<LocalDate, List<Trade>>> groupedTrades = trades.stream()

.collect(Collectors.groupingBy(

Trade::getTradeInstrument,

Collectors.groupingBy(Trade::getTradeDate)

));

// Step 2: Find the second maximum price for each group (instrumentId + date combination)

groupedTrades.forEach((instrumentId, dateMap) -> {

System.out.println("Instrument: " + instrumentId);

dateMap.forEach((date, tradeList) -> {

// Step 3: Sort trades by tradeValue in descending order and get the second highest using skip(1) and limit(1)

Optional<Trade> secondMaxTrade = tradeList.stream()

.sorted((trade1, trade2) -> Double.compare(trade2.getTradeValue(), trade1.getTradeValue())) // Sort descending

.skip(1) // Skip the highest trade

.limit(1) // Take the second trade

.findFirst(); // Find the second highest trade

if (secondMaxTrade.isPresent()) {

System.out.println(" Date: " + date + ", Second Max Trade: " + secondMaxTrade.get());

} else {

System.out.println(" Date: " + date + ", No second maximum trade (only one trade available).");

}

});

});

}

}

======================================

**3:-----;---- . write a java program to**

**show a report having list of data instrumentIdument and transaction date order and sum of each value of transaction for each day.**

import java.time.LocalDate;

import java.util.\*;

import java.util.stream.Collectors;

// Define the Transaction class

class Transaction {

private String instrumentId;

private LocalDate transactionDate;

private double orderAmount;

// Constructor

public Transaction(String instrumentId, LocalDate transactionDate, double orderAmount) {

this.instrumentId = instrumentId;

this.transactionDate = transactionDate;

this.orderAmount = orderAmount;

}

// Getters

public String getInstrumentId() {

return instrumentId;

}

public LocalDate getTransactionDate() {

return transactionDate;

}

public double getOrderAmount() {

return orderAmount;

}

@Override

public String toString() {

return "InstrumentId: " + instrumentId + ", Date: " + transactionDate + ", Order Amount: " + orderAmount;

}

}

public class TransactionReport {

public static void main(String[] args) {

// Sample transactions

List<Transaction> transactions = Arrays.asList(

new Transaction("A123", LocalDate.of(2025, 1, 19), 100.50),

new Transaction("A123", LocalDate.of(2025, 1, 19), 150.75),

new Transaction("B456", LocalDate.of(2025, 1, 19), 200.00),

new Transaction("A123", LocalDate.of(2025, 1, 20), 300.00),

new Transaction("B456", LocalDate.of(2025, 1, 20), 50.50),

new Transaction("A123", LocalDate.of(2025, 1, 21), 100.00)

);

// Group by transaction date and instrumentId, then sum order amounts

Map<String, Map<LocalDate, Double>> report = transactions.stream()

.collect(Collectors.groupingBy(

Transaction::getInstrumentId,

Collectors.groupingBy(

Transaction::getTransactionDate,

Collectors.summingDouble(Transaction::getOrderAmount)

)

));

// Print the report

report.forEach((instrumentId, dateMap) -> {

System.out.println("Instrument ID: " + instrumentId);

dateMap.forEach((date, totalAmount) -> {

System.out.println(" Date: " + date + ", Total Order Amount: " + totalAmount);

});

});

}

}

Explanation:

Transaction Class: This class has three attributes:

instrumentId for identifying the instrument.

transactionDate for the date of the transaction.

orderAmount for the value of the transaction.

We also provide a constructor to initialize these attributes and a toString() method for easy printing.

Stream Processing:

Grouping: We use the Collectors.groupingBy() method twice:

First, we group by instrumentId.

Then, within each instrument group, we group by transactionDate.

Summing: The Collectors.summingDouble() method is used to sum the orderAmount for each group.

Result: After grouping and summing, we print the report, which shows the instrument ID, transaction date, and total order amount for each date.

Sample Output:

yaml

Copy

Instrument ID: A123

Date: 2025-01-19, Total Order Amount: 251.25

Date: 2025-01-20, Total Order Amount: 300.0

Date: 2025-01-21, Total Order Amount: 100.0

Instrument ID: B456

Date: 2025-01-19, Total Order Amount: 200.0

Date: 2025-01-20, Total Order Amount: 50.5

This approach ensures that you can easily generate reports by leveraging the power of Java 8 streams and lambda expressions.

=======================================================================================================

**4:--- Find top 2 salary from employees table using sql,**

SELECT DISTINCT salary

FROM employees

ORDER BY salary DESC

LIMIT 2;

================================================================

**5:----------------Filter non duplicate student by name from the list**

import java.util.\*;

import java.util.stream.Collectors;

class Student {

private String name;

private int age;

// Constructor

public Student(String name, int age) {

this.name = name;

this.age = age;

}

// Getters

public String getName() {

return name;

}

public int getAge() {

return age;

}

@Override

public String toString() {

return "Student{name='" + name + "', age=" + age + "}";

}

}

public class Main {

public static void main(String[] args) {

// Sample student list

List<Student> students = Arrays.asList(

new Student("Alice", 20),

new Student("Bob", 22),

new Student("Alice", 21),

new Student("Charlie", 23),

new Student("David", 24)

);

// Filter out non-duplicate students by name

List<Student> nonDuplicateStudents = students.stream()

.collect(Collectors.groupingBy(Student::getName)) // Group by name

.entrySet().stream() // Convert map to stream

.filter(entry -> entry.getValue().size() == 1) // Keep only names with a single student

.flatMap(entry -> entry.getValue().stream()) // Flatten the list of students

.collect(Collectors.toList()); // Collect into a list

// Output the result

nonDuplicateStudents.forEach(System.out::println);

}

}

===================================================================

**6:-------------Kubernative commands related to deployment**

To get nodes: kubectl get nodes

To run image:-kubectl run ngnix --image=ngnix-------------------it will create pod also

To get pods : kubectl get pods

To get more info abt pods:- kubectl describe pods

To get node info also:- kubectl get pods -o wide

Kubectl create -f deployment.yaml

Kubectl get replicaset

Kubectl describe replicaset

Kubectl get pods ( three replica set will get created)

Note: replicaset use to monitor the pod and if one of the pod is deleted replicaset use to create new pod

kubectl delete pod myapp-replicaset-thhrt

NOTE: replica set will not allow to create pod with same label

kubectl get pods -o wide ( will give info about node)

If we updata replicaset

kubectl replace -f deployment.yaml

Deployment

To check what is up: kubectl get all

Create deployment: kubectl create -f deployment.yaml --record

Check status:kubectl rollout status deployment/myapp-deployment (as soon as deployment is created

rollout is the process to see deploying container in the background)

Will create 1 revision : kubectl rollout history deployment/myapp-deployment---------will tell about revision

Upgrade: kubectl apply -f deployment.yaml

Check: kubectl describe deployment --record

Will create revision1 and revision 2 : kubectl rollout status deployment/myapp-deployment

Rollout: kubectl rollout undo deployment/myapp-deployment

NOTE:

When we rollout revision 2 nd revision 3 will display not revision 1