1-Internal Implementation of HashMap

Implementing a HashMap from scratch can be quite complex due to the various data structures and algorithms involved, but I can give you a simplified version to illustrate the key components and logic behind it. Here's a basic implementation of HashMap in Java:

**package** com;

**import** java.util.LinkedList;

**import** java.util.List;

**public** **class** HashMap<K, V> {

**private** **static** **final** **int** ***DEFAULT\_CAPACITY*** = 16;

**private** List<Entry<K, V>>[] table;

**private** **int** size;

**public** HashMap() {

**this**(***DEFAULT\_CAPACITY***);

}

**public** HashMap(**int** capacity) {

table = **new** LinkedList[capacity];

size = 0;

}

**private** **static** **class** Entry<K, V> {

K key;

V value;

Entry<K, V> next;

Entry(K key, V value) {

**this**.key = key;

**this**.value = value;

}

}

// Hash function to compute the bucket index

**private** **int** hash(K key) {

**return** Math.*abs*(key.hashCode() % table.length);

}

**public** **void** put(K key, V value) {

**int** bucketIndex = hash(key);

Entry<K, V> newEntry = **new** Entry<>(key, value);

**if** (table[bucketIndex] == **null**) {

table[bucketIndex] = **new** LinkedList<>();

}

**for** (Entry<K, V> entry : table[bucketIndex]) {

**if** (entry.key.equals(key)) {

entry.value = value;

**return**;

}

}

table[bucketIndex].add(newEntry);

size++;

}

**public** V get(K key) {

**int** bucketIndex = hash(key);

**if** (table[bucketIndex] != **null**) {

**for** (Entry<K, V> entry : table[bucketIndex]) {

**if** (entry.key.equals(key)) {

**return** entry.value;

}

}

}

**return** **null**;

}

**public** **void** remove(K key) {

**int** bucketIndex = hash(key);

**if** (table[bucketIndex] != **null**) {

table[bucketIndex].removeIf(entry -> entry.key.equals(key));

size--;

}

}

**public** **boolean** containsKey(K key) {

**int** bucketIndex = hash(key);

**if** (table[bucketIndex] != **null**) {

**for** (Entry<K, V> entry : table[bucketIndex]) {

**if** (entry.key.equals(key)) {

**return** **true**;

}

}

}

**return** **false**;

}

**public** **int** size() {

**return** size;

}

**public** **boolean** isEmplty() {

**return** size == 0;

}

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

HashMap<String, Integer> map = **new** HashMap<>();

map.put("apple", 1);

map.put("banana", 2);

map.put("cherry", 3);

System.***out***.println("Size of map: " + map.size());

System.***out***.println("Value for key 'banana': " + map.get("banana"));

map.remove("banana");

System.***out***.println("Size of map after removing 'banana': " + map.size());

System.***out***.println("Contains key 'banana'? " + map.containsKey("banana"));

}

}

// output

Size of map: 3

Value for key 'banana': 2

Size of map after removing 'banana': 2

Contains key 'banana'? false

**Explanation**

1. **Constants and Variables**:
   * DEFAULT\_CAPACITY: Default number of buckets in the hash table.
   * table: Array of lists (LinkedList) to store entries (buckets).
   * size: Current number of key-value mappings in the map.
2. **Entry Class**:
   * Entry<K, V>: Nested class to represent key-value pairs and handle collisions using chaining (next pointer).
3. **Constructors**:
   * HashMap(): Default constructor initializes table with default capacity.
   * HashMap(int capacity): Constructor initializes table with specified capacity.
4. **Hash Function**:
   * hash(K key): Computes the bucket index using the hash code of the key.
5. **Put Method**:
   * put(K key, V value): Adds or updates a key-value pair in the map.
   * Calculates the bucket index using hash(key).
   * Checks if the key already exists in the bucket, updates value if found.
   * Adds new entry to the bucket if key not found.
6. **Get Method**:
   * get(K key): Retrieves the value associated with the key.
   * Computes the bucket index using hash(key).
   * Searches for the key in the bucket, returns value if found.
7. **Remove Method**:
   * remove(K key): Deletes the key-value pair from the map.
   * Computes the bucket index using hash(key).
   * Removes the entry from the bucket if key found.
8. **Contains Key Method**:
   * containsKey(K key): Checks if the map contains the given key.
   * Computes the bucket index using hash(key).
   * Searches for the key in the bucket, returns true if found.
9. **Size and Empty Check Methods**:
   * size(): Returns the current size of the map.
   * isEmpty(): Checks if the map is empty.
10. **Main Method**:
    * main(String[] args): Test the HashMap implementation by adding, retrieving, and removing key-value pairs.

**Summary**

This basic HashMap implementation demonstrates how a hash table can be implemented in Java using an array of linked lists for collision handling. It covers fundamental operations such as put, get, remove, containsKey, and provides utility methods for size and emptiness checks. Real-world HashMap implementations in Java are more complex, optimized, and include features like resizing, load factor management, and concurrency handling.

2 – Internal implementation of HashSet

To understand how HashSet works internally using HashMap in Java, let's delve into the implementation details and provide an example to illustrate its usage.

### HashSet using HashMap Implementation

Here's how you can implement a simplified version of HashSet using HashMap internally:

**package** com;

**import** java.util.HashMap;

**public** **class** HashSet<E> {

**private** **transient** HashMap<E, Object> map;

**private** **static** **final** Object ***PRESENT*** = **new** Object();

**public** HashSet() {

map = **new** HashMap<>();

}

**public** **boolean** add(E element) {

System.***out***.println(map.put(element, ***PRESENT***));

**return** map.put(element, ***PRESENT***) == **null**;

}

**public** **boolean** contains(E element) {

**return** map.containsKey(element);

}

**public** **boolean** remove(E element) {

**return** map.remove(element) == ***PRESENT***;

}

**public** **int** size() {

**return** map.size();

}

**public** **boolean** isEmpty() {

**return** map.isEmpty();

}

**public** **void** clear() {

map.clear();

}

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

HashSet<String> set = **new** HashSet<>();

// Adding elements to the set

set.add("apple");

set.add("banana");

set.add("cherry");

// Checking if elements exist in the set

System.***out***.println("Set contains 'apple': " + set.contains("apple"));

System.***out***.println("Set contains 'orange': " + set.contains("orange"));

// Removing elements from the set

set.remove("banana");

// Checking size and emptiness

System.***out***.println("Size of set: " + set.size());

System.***out***.println("Is set empty?: " + set.isEmpty());

// Clearing the set

set.clear();

System.***out***.println("Is set empty after clearing?: " + set.isEmpty());

}

}

// output

Set contains 'apple': true

Set contains 'orange': false

Size of set: 2

Is set empty?: false

Is set empty after clearing?: true

**Explanation**

1. **HashMap as Internal Storage**:
   * private transient HashMap<E, Object> map;
     + map is used internally to store elements of type E as keys, with a dummy Object (PRESENT) as the value. This dummy object is used because HashMap does not allow duplicate keys, so we leverage its properties to ensure uniqueness in our HashSet.
2. **Adding Elements (add method)**:
   * public boolean add(E element) { return map.put(element, PRESENT) == null; }
     + Adds the element to the map as a key. If the element was already present (i.e., put returns a non-null value), it returns false, indicating the element was not added. Otherwise, it returns true.
3. **Checking Element Existence (contains method)**:
   * public boolean contains(E element) { return map.containsKey(element); }
     + Checks if the element exists in the map using containsKey.
4. **Removing Elements (remove method)**:
   * public boolean remove(E element) { return map.remove(element) == PRESENT; }
     + Removes the element from the map and returns true if the element was present (i.e., removed successfully).
5. **Utility Methods**:
   * size(), isEmpty(), and clear() are standard methods to get the size of the set, check if it's empty, and clear all elements from the set, respectively.
6. **Example Usage (main method)**:
   * Demonstrates how to create a HashSet, add elements ("apple", "banana", "cherry"), check for existence (contains), remove an element (remove), check size and emptiness (size, isEmpty), and clear the set (clear).

**Summary**

This implementation of HashSet using HashMap internally leverages the properties of HashMap to provide efficient operations for adding, checking existence, and removing elements, ensuring uniqueness and performance similar to a traditional HashSet. This approach demonstrates how you can utilize existing data structures (HashMap in this case) to implement custom collections tailored to specific needs.

**explain this line return map.put(element, PRESENT) == null;**

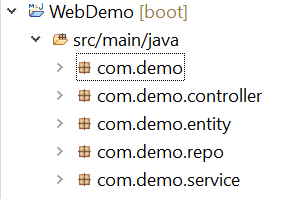
**Example Scenario**

Consider a scenario where you have a HashSet<String> set and you execute set.add("apple");:

* Initially, map.put("apple", PRESENT) will return null because "apple" was not present in the HashMap.
* Therefore, return true; will indicate that "apple" was successfully added to the HashSet.

If you later try to add "apple" again (set.add("apple");), the put method will return the previous value associated with "apple" (which is PRESENT), and thus return false; will indicate that "apple" was not added again because it already exists in the HashSet.

QUESTION 3 : api demo



1. Entity

**package** com.demo.entity;

**import** jakarta.persistence.Entity;

**import** jakarta.persistence.Id;

@Entity

**public** **class** Product {

@Id

**private** Long id;

**private** String name;

**private** String description;

**public** Product() {

}

**public** Product(Long id, String name, String description) {

**this**.id = id;

**this**.name = name;

**this**.description = description;

}

**public** Long getId() {

**return** id;

}

**public** **void** setId(Long id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getDescription() {

**return** description;

}

**public** **void** setDescription(String description) {

**this**.description = description;

}

@Override

**public** String toString() {

**return** "Product [id=" + id + ", name=" + name + ", description=" + description + "]";

}

}

2 – repo

**package** com.demo.repo;

**import** org.springframework.data.jpa.repository.JpaRepository;

**import** com.demo.entity.Product;

**public** **interface** ProductRepository **extends** JpaRepository<Product, Long>{

}

3----service

**package** com.demo.service;

**import** java.util.List;

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

**import** org.springframework.stereotype.Service;

**import** com.demo.entity.Product;

**import** com.demo.repo.ProductRepository;

@Service

**public** **class** ProductService {

@Autowired

**private** ProductRepository prodRepo;

**public** List<Product> getAllProducts() {

**return** prodRepo.findAll();

}

**public** Product getProductById(Long Id) {

**return** prodRepo.findById(Id).orElse(**null**);

}

**public** Product saveProduct(Product prod) {

**return** prodRepo.save(prod);

}

**public** **void** deleteProduct(Long Id) {

prodRepo.deleteById(Id);

}

}

4-----------------controller

For exception :

https://chatgpt.com/c/32185a11-0e64-40ca-b81c-0977841ef501

**package** com.demo.controller;

**import** java.util.List;

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

**import** org.springframework.http.HttpStatus;

**import** org.springframework.http.ResponseEntity;

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

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

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

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

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

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

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

**import** com.demo.entity.Product;

**import** com.demo.service.ProductService;

@RestController

**public** **class** ProductController {

@Autowired

**private** ProductService prodService;

@GetMapping("/api/products")

**public** ResponseEntity<List<Product>> getAllProducts() {

List<Product> allProducts = prodService.getAllProducts();

**return** **new** ResponseEntity<>(allProducts, HttpStatus.***OK***);

}

@GetMapping("/api/products/{id}")

**public** ResponseEntity<Product> getProductById(@PathVariable("id") Long id) {

Product product = prodService.getProductById(id);

**if** (product != **null**) {

**return** **new** ResponseEntity<>(product, HttpStatus.***OK***);

} **else** {

**return** **new** ResponseEntity<>(HttpStatus.***NOT\_FOUND***);

}

}

@PostMapping("/api/products")

**public** ResponseEntity<Product> createProduct(@RequestBody Product prod) {

Product product = prodService.saveProduct(prod);

**return** **new** ResponseEntity<>(product, HttpStatus.***CREATED***);

}

@PutMapping("/api/products/{id}")

**public** ResponseEntity<Product> updateProduct(@PathVariable("id") Long id, @RequestBody Product prod) {

Product product = prodService.getProductById(id);

**if** (product != **null**) {

product.setId(id);

Product updatedProduct = prodService.saveProduct(product);

**return** **new** ResponseEntity<>(updatedProduct, HttpStatus.***OK***);

} **else** {

**return** **new** ResponseEntity<>(HttpStatus.***NOT\_FOUND***);

}

}

@DeleteMapping("/api/products/{id}")

**public** ResponseEntity<Void> deleteProduct(@PathVariable("id") Long id) {

Product product = prodService.getProductById(id);

**if** (product != **null**) {

prodService.deleteProduct(id);

**return** **new** ResponseEntity<>(HttpStatus.***NO\_CONTENT***);

} **else** {

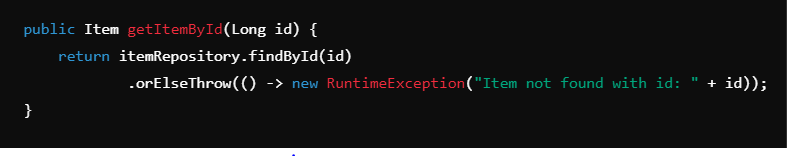
**return** **new** ResponseEntity<>(HttpStatus.***NOT\_FOUND***);

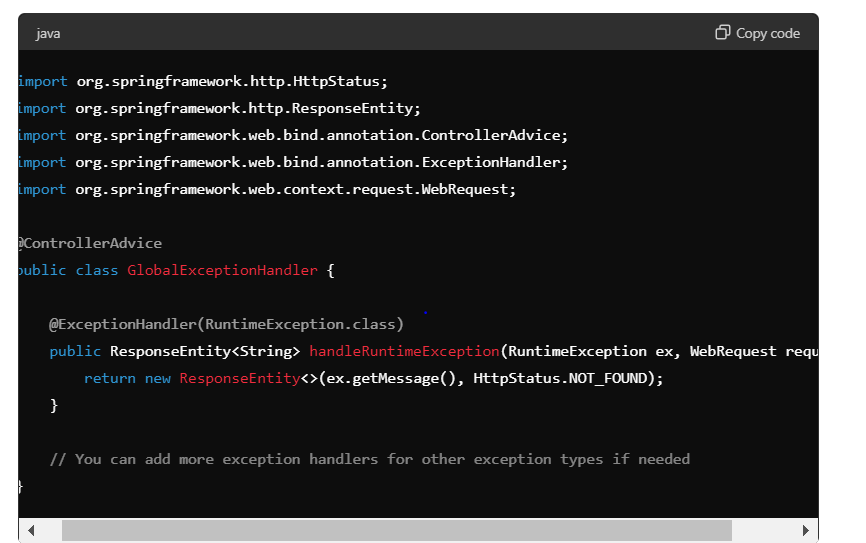
}

}

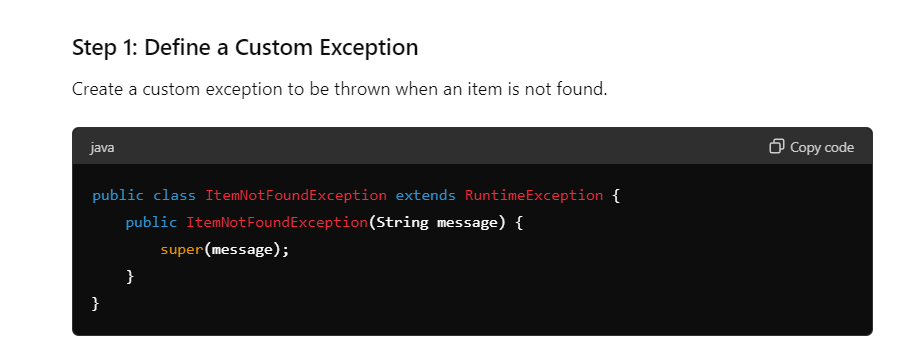
}

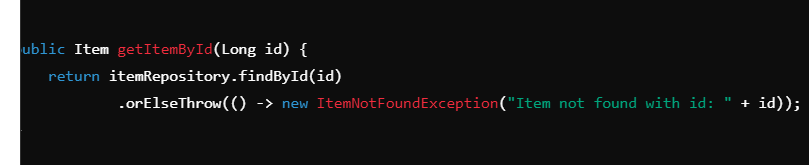
Exception

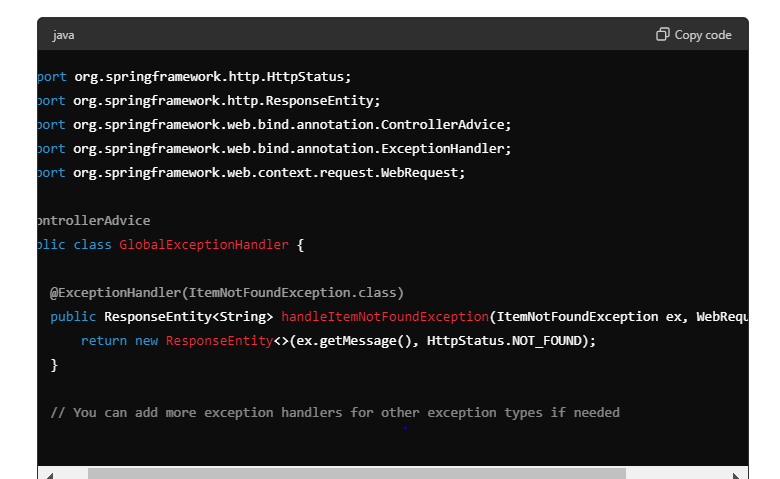




**we can create custom exception and throw also**

****





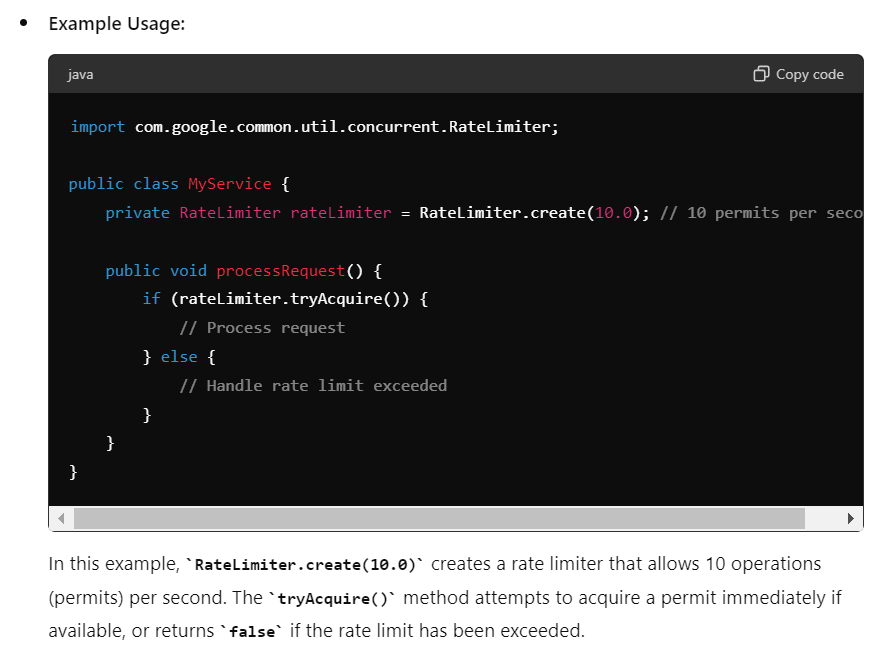
1. How to apply rate limit

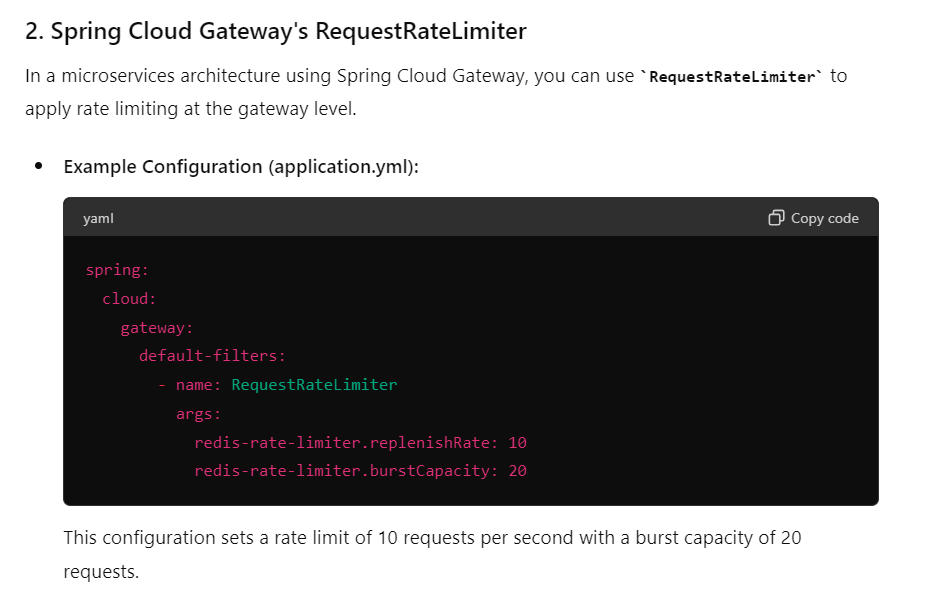
In Spring Boot, RateLimiter typically refers to a mechanism used to control the rate at which certain operations or requests are allowed to be performed or processed. This helps in managing resources, preventing abuse, and maintaining system stability under heavy load conditions.

There are different contexts in which RateLimiter can be used in Spring Boot:

### 1. Guava RateLimiter

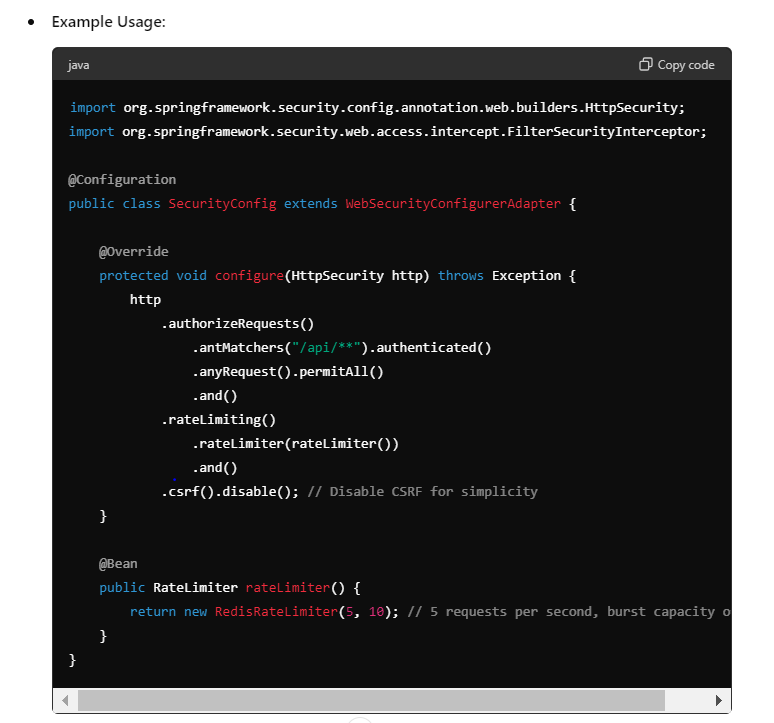
Guava provides a RateLimiter implementation as part of its library (com.google.common.util.concurrent.RateLimiter). It's a simple and efficient way to throttle operations based on a specified rate.





### 3. Spring Security Rate Limiting

Spring Security provides rate limiting capabilities that can be applied to specific endpoints or operations.



* In this example, RedisRateLimiter is used to configure a rate limiter that allows 5 requests per second with a burst capacity of 10 requests.

### Summary

In Spring Boot, RateLimiter implementations such as Guava's RateLimiter and Spring Cloud Gateway's RequestRateLimiter are used to control the rate of operations or requests, preventing overload and ensuring fair usage of resources. These mechanisms are essential for maintaining system stability and optimizing performance, especially in distributed and microservices-based applications.

5-what happens if one of the local transaction failed in saga design pattern

In the Saga design pattern, handling failures of local transactions is critical to ensuring consistency and reliability across distributed systems. Here's how failures in local transactions are typically managed within the Saga pattern:

### Understanding Local Transaction Failures

In a Saga, which orchestrates a sequence of local transactions to achieve a business goal (e.g., placing an order, reserving inventory, processing payment), failure in any of these transactions can occur due to various reasons such as network issues, service unavailability, or business rule violations.

### Handling Local Transaction Failures

When a local transaction fails, the Saga pattern employs compensating actions to undo or revert the effects of the preceding successful transactions. This ensures that the system reaches a consistent state despite partial failures.

#### Steps to Handle Local Transaction Failure:

1. **Compensating Actions:**
   * Each local transaction has a corresponding compensating action designed to reverse the effects of the transaction.
   * Compensating actions are idempotent and aim to restore the system to a consistent state by undoing any changes made by the failed transaction.
2. **Publishing Compensation Events:**
   * Upon detecting a failure in a local transaction, the Saga triggers compensating actions by publishing compensation events.
   * These events are handled by other components or sagas responsible for executing the compensating actions.
3. **Reverting State Changes:**
   * Compensating actions typically involve reverting state changes in downstream systems or services affected by the failed transaction.
   * For example, if a payment processing transaction fails, the compensating action would involve canceling or rolling back the payment.
4. **Handling Retry and Recovery:**
   * Sagas may implement retry mechanisms for local transactions that fail due to transient errors (e.g., network timeouts).
   * Recovery mechanisms ensure that failed sagas are resumed or retried from the point of failure to achieve eventual consistency.

### Example Scenario

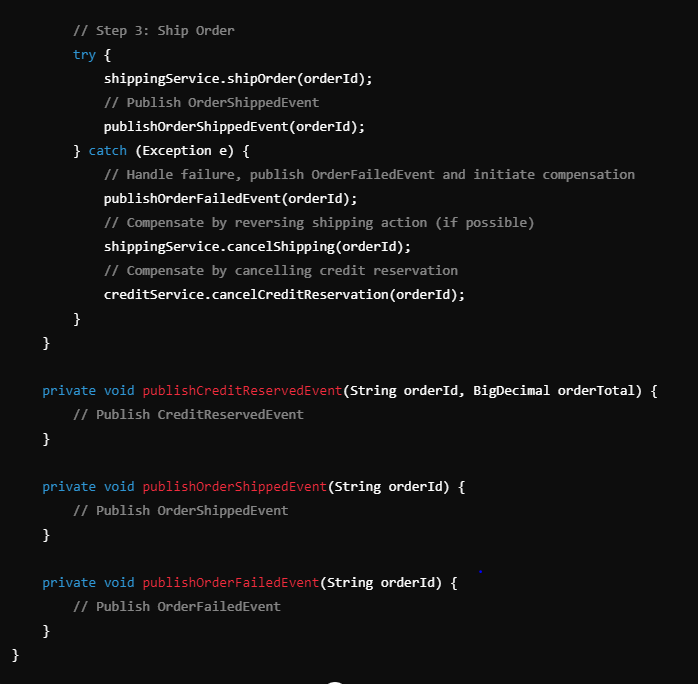
Consider an Order Management Saga:

* **Steps:**
  1. **Create Order**: Initiates the order creation process.
  2. **Reserve Credit**: Reserves credit for the order amount.
  3. **Ship Order**: Ships the order to the customer.
* **Failure Scenario:**
  1. If the "Reserve Credit" transaction fails due to insufficient funds, a compensating action (CancelCreditReservationCommand) is triggered to release the reserved credit and notify relevant parties.

### Code Example

Here's a simplified example of handling a local transaction failure and initiating compensating actions in a Saga-like structure (without using Axon Framework):





In this example, if either the reserveCredit or shipOrder methods throw an exception, the catch blocks handle the failure by publishing an OrderFailedEvent. Compensating actions (cancelCreditReservation and cancelShipping) are then initiated to revert any successful transactions preceding the failure.

### Summary

Handling local transaction failures in the Saga pattern involves designing compensating actions to maintain system integrity despite partial failures. This approach ensures that complex business processes can recover from errors and maintain consistency across distributed systems. Each compensating action is carefully designed to undo the effects of failed transactions, ultimately achieving the desired consistent state within the system.

6:- how to secure two microservice talk with each other with code in java

Securing communication between microservices involves ensuring confidentiality, integrity, authentication, and authorization of messages exchanged between them. Here's how you can achieve this using Java with Spring Boot, leveraging TLS/SSL for encryption and mutual authentication:

### Secure Communication between Microservices

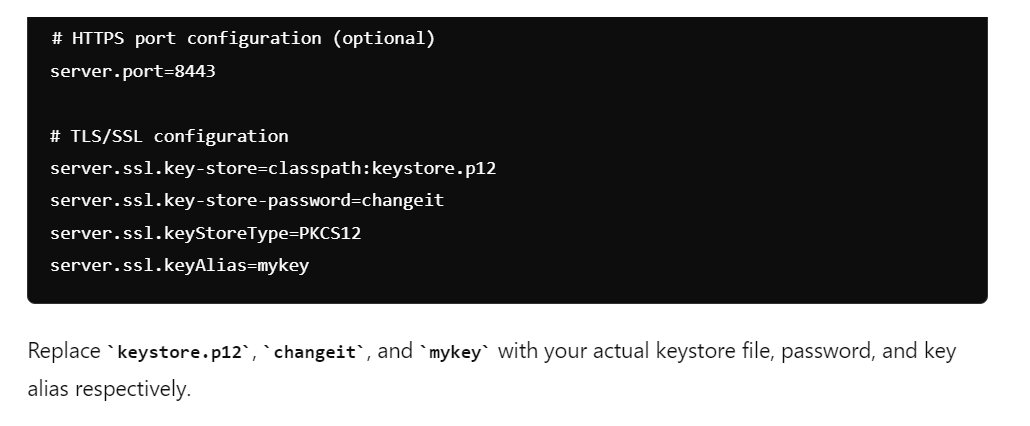
#### 1. Enable TLS/SSL in Spring Boot

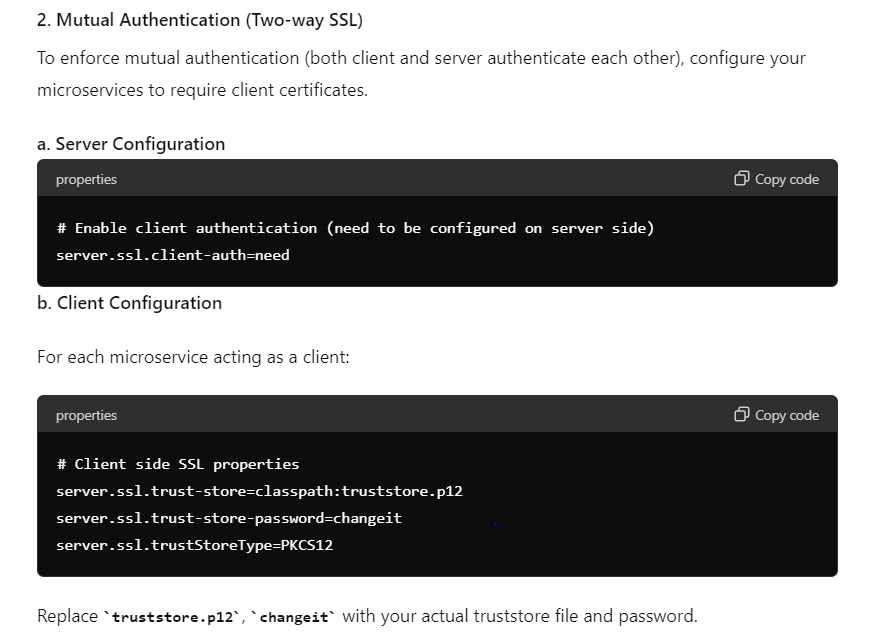
Spring Boot simplifies enabling TLS/SSL with embedded servers (like Tomcat) using configuration properties. Here’s how to configure it:

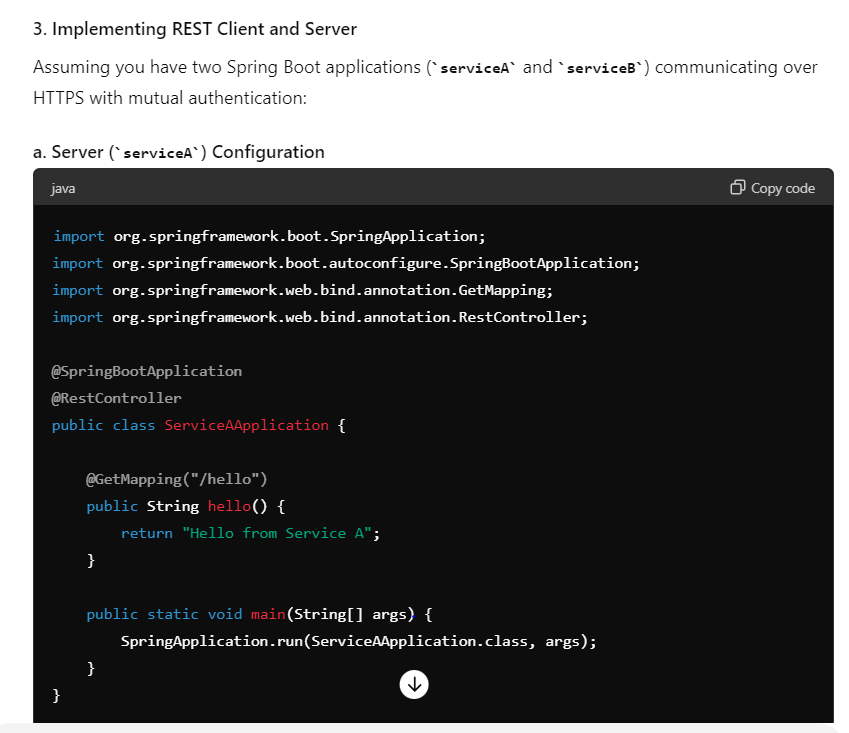
##### a. Generate SSL Certificates (if not already available)

You can generate self-signed certificates for development purposes using tools like keytool or use certificates from a trusted Certificate Authority (CA) for production.

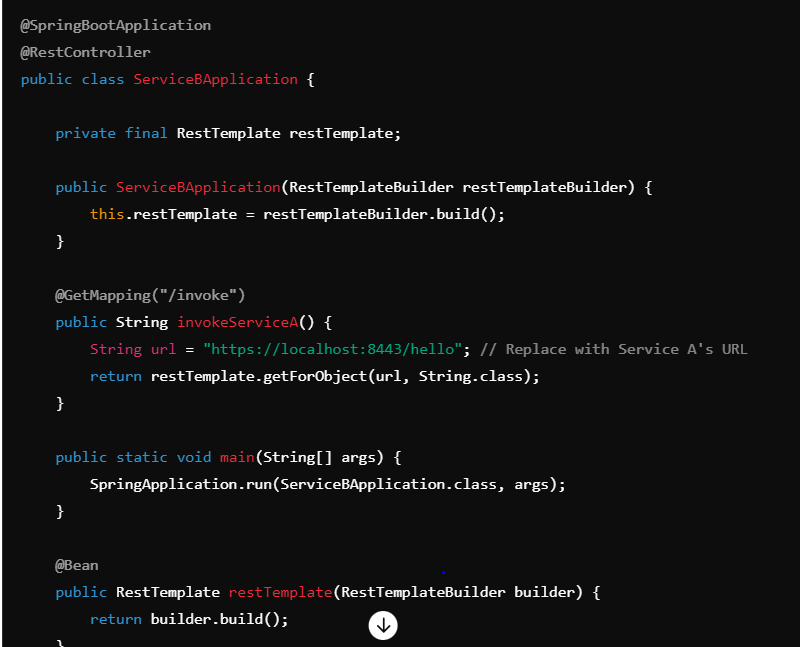
##### b. Configure TLS/SSL in application.properties or application.yml







b. Client (serviceB) Configuration



#### 4. Testing the Setup

Ensure both microservices are running and accessible over HTTPS. You can use tools like Postman or cURL to verify secure communication between them. The setup ensures that:

* Communication is encrypted using TLS/SSL.
* Mutual authentication ensures both microservices verify each other’s identity using certificates.

### Summary

Securing communication between microservices involves configuring TLS/SSL with mutual authentication. Spring Boot simplifies this setup using configuration properties for embedded servers and RestTemplate. Ensure proper certificate management and configuration to maintain security across microservices in your architecture. Adjust configurations based on your deployment environment (development, staging, production) and security requirements.

7-explain complete code in java with getting jwt and validating jwt and microservice interact with other microservice how other microservice validate token explain all

Certainly! Let's break down the process of using JWT for authentication and authorization between two microservices in Java. This example will involve two microservices: **Microservice A** (which generates and sends JWT) and **Microservice B** (which receives requests and validates JWT).

### Microservice A: Generating and Sending JWT

In Microservice A, we will generate a JWT upon user authentication and then send it to Microservice B in subsequent requests.

#### Dependencies

For handling JWT in Java, we typically use libraries like jjwt (Java JWT: JSON Web Token for Java and Android).



### Microservice B: Validating JWT

In Microservice B, we will receive requests containing JWT and validate them before processing further.

#### Dependencies

Similarly, for Microservice B, we need the jjwt library to handle JWT validation.

#### Example Code for Microservice B



### Explanation:

#### 1. **Microservice A** (Generating and Sending JWT):

* **JWT Generation**: Upon successful user authentication (/login endpoint), Microservice A generates a JWT using jjwt.
* **Sending JWT**: The JWT is then sent to Microservice B in subsequent requests, typically in the Authorization header.

#### 2. **Microservice B** (Validating JWT):

* **JWT Validation**: Microservice B receives requests (/resource endpoint) containing JWT in the request body.
* **Verification**: It verifies the JWT using the same SECRET\_KEY used by Microservice A to sign the JWT.
* **Claims Extraction**: If the JWT is valid, Microservice B extracts and uses the claims (like userId) for processing the request.
* **Error Handling**: Proper error handling is crucial, especially for handling SignatureException if the JWT signature is invalid.

#### Security Considerations:

* **Secure Transmission**: Ensure JWTs are transmitted over HTTPS to prevent interception.
* **Secret Management**: Keep SECRET\_KEY secure and rotate it periodically for enhanced security.
* **Token Expiry**: Optionally, include an expiration time (setExpiration()) when generating JWTs to limit their lifespan.

This setup illustrates a basic implementation of JWT-based authentication and authorization between two microservices in a Java environment. Adjustments may be needed based on specific frameworks (like Spring Security) or additional security requirements.

8--------- what is generics ? explain with example

 **Type Safety**: Generics make your code safer by ensuring that you can only use the types you expect.

 **Reusability**: You can write code that works with different types, making your code more reusable.

 **Elimination of Casts**: Generics reduce the need for casting when retrieving elements from a collection.

Custom generic class

**package** com;

**class** Gen<T> {

**private** T obj;

**public** Gen(T obj) {

**this**.obj = obj;

}

@Override

**public** String toString() {

**return** "Gen [obj=" + getObj() + "]";

}

**public** T getObj() {

**return** obj;

}

**public** **void** setObj(T obj) {

**this**.obj = obj;

}

}

**public** **class** TestProg {

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

Gen<Integer> intValue = **new** Gen<>(123);

Gen<String> str = **new** Gen<>("saurabh");

System.***out***.println(intValue.getObj());

System.***out***.println(str.getObj());

}

}

9---------------- stereotype annotation

**These annotations help Spring to automatically detect and configure classes as beans within the Spring application context,**

**Automatic Detection**: They enable Spring to automatically detect and register these classes as beans during component scanning.

**Common Stereotype Annotations in Spring**

1. **@Component**: The base stereotype annotation for any Spring-managed component.
2. **@Service**: Indicates that a class performs service tasks (business logic).
3. **@Repository**: Indicates that a class is a Data Access Object (DAO) and integrates with the database.
4. **@Controller**: Indicates that a class is a web controller handling HTTP requests.
5. **@RestController**: A specialized version of @Controller for RESTful web services, combining @Controller and @ResponseBody.

10 :- how to optimize query in jpa

Optimizing queries in Java Persistence API (JPA) is essential for improving the performance of your application. Here are several strategies to optimize your JPA queries:

**1. Use JPQL and Native Queries Wisely**

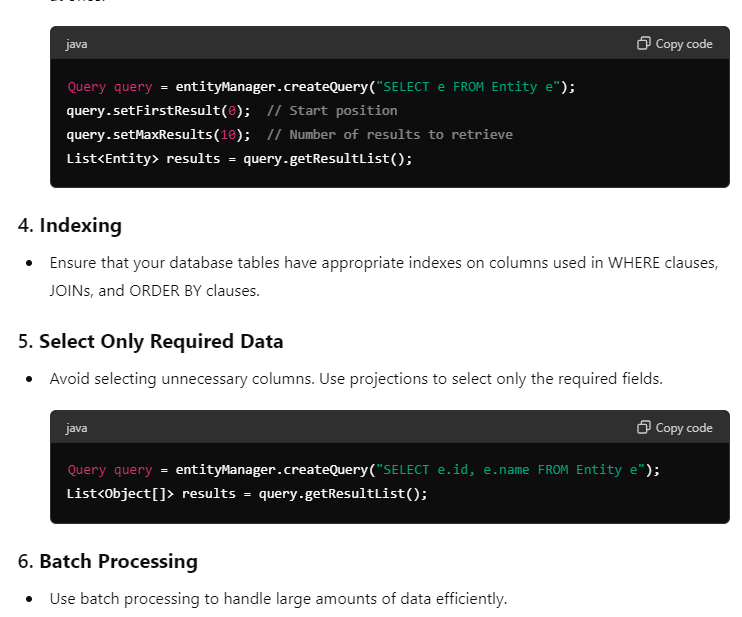
* **JPQL (Java Persistence Query Language)**: Use JPQL for complex queries involving entity relationships.
* **Native Queries**: Use native SQL queries for complex or performance-critical operations where JPQL might not be efficient.

**2. Fetch Strategies**

* **Eager Loading vs. Lazy Loading**: Use lazy loading (FetchType.LAZY) for collections and associations by default. Use eager loading (FetchType.EAGER) only when necessary to avoid fetching unnecessary data.
* **Join Fetching**: Use JPQL's JOIN FETCH clause to reduce the number of queries executed by fetching associated entities in a single query.

**3. Pagination**

* Use pagination for queries that return large result sets to avoid loading all records into memory at once.





**7. Cache Usage**

* Use second-level caching to reduce database access by caching entity data.
  + Configure caching using annotations or XML.
  + Use query caching for frequently executed queries.

**8. Optimize Entity Relationships**

* Avoid bi-directional relationships if not necessary to reduce complexity.
* Use @ManyToOne and @OneToMany appropriately based on the cardinality and access patterns.

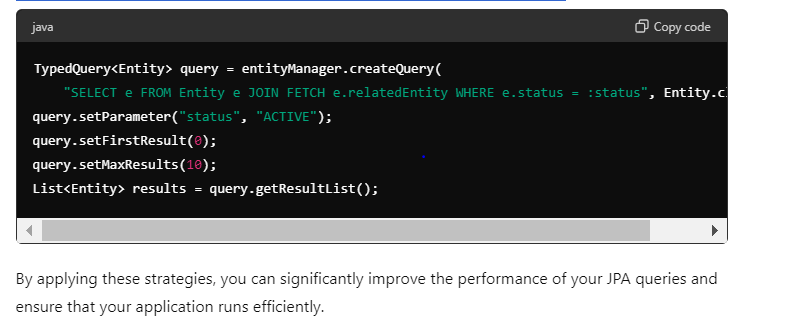
**9. Profiling and Monitoring**

* Use tools like JPA Performance Profiler or Hibernate Statistics to monitor query performance and identify bottlenecks.
* Analyze slow queries and optimize them accordingly.

**10. Database-Specific Optimizations**

* Leverage database-specific features like stored procedures, functions, and advanced indexing options for performance-critical operations.

**Example Optimized Query with Join Fetching and Pagination**



Question 11 : what is entityManager ?

The EntityManager is a key interface in Java Persistence API (JPA) that is used for interacting with the persistence context. It provides methods to create, read, update, and delete entities, as well as to query the database.

### Key Responsibilities of EntityManager

1. **Persisting Entities**: Saving entities to the database.
2. **Retrieving Entities**: Finding entities by their primary key or using JPQL/Criteria queries.
3. **Updating Entities**: Merging detached entities back into the persistence context.
4. **Removing Entities**: Deleting entities from the database.
5. **Transaction Management**: Beginning, committing, and rolling back transactions.