Gaurav Shukla

shuklagauravjn@gmail.com

Abstract

The 2 services being provided as a part of the assignment can be the foundation for a full-fledged micro services architecture in future.

Solution architecture

Implementation approach for the case study

Contents

[Solution Architecture 2](#_Toc195019309)

[Recommendations and Solution Outline 2](#_Toc195019310)

[(A) Transactional Boundaries in Service A 2](#_Toc195019311)

[Local Transactions: 2](#_Toc195019312)

[Distributed Transactions (if needed): 2](#_Toc195019313)

[(B) Threading Model in Service A (JEE or Spring Boot) 2](#_Toc195019314)

[Spring Boot/JEE Default Threading: 2](#_Toc195019315)

[(1) Network Communication Issues Between Service A and Service B 3](#_Toc195019316)

[a. Service B Temporarily Unreachable: 3](#_Toc195019317)

[b. Connection to Service B Timing Out or Being Lost: 4](#_Toc195019318)

[(2) Service A Crashing While Processing a User Request 4](#_Toc195019319)

[a. Possible Inconsistencies: 4](#_Toc195019320)

[b. Addressing Inconsistencies (Reconciliation): 4](#_Toc195019321)

[Key Considerations 4](#_Toc195019322)

# Solution Architecture

The system involves a User interacting with two RESTful web services, Service A and Service B. The workflow is:

@startuml

title Order Processing Sequence

actor User

User -> ServiceA : POST /order

activate ServiceA

ServiceA -> Database : Persist

activate Database

Database -->> ServiceA : OK

deactivate Database

ServiceA -> ServiceB : POST /notify

activate ServiceB

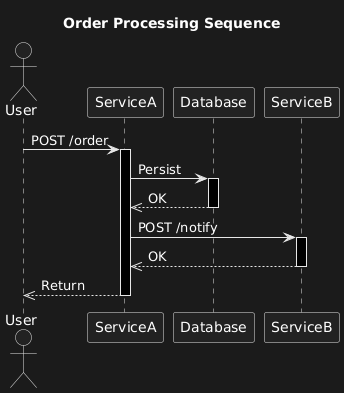
ServiceB -->> ServiceA : OK

deactivate ServiceB

ServiceA -->> User : Return

deactivate ServiceA

@enduml



# Recommendations and Solution Outline

Here's a recommended approach to address the questions, broken down by section:

## (A) Transactional Boundaries in Service A

Local Transactions: Use local transactions within Service A to ensure that the persistence operation is atomic. If the persistence fails, the entire operation should roll back. Essentially H2 Database which is RDBMS is used in code. So RDBMS have ACID transaction. Atomicity is single, one indivisible unit of work at a time. Consistency maintains the integrity of the data. It moves the database in one state or another. Isolated means that none other transaction should interfere, while Durability means post completion that data should be permanent. We do not need eventual consistency here.

Distributed Transactions (if needed): If Service A needs to interact with other services within the transactional boundary (which isn't explicitly stated, but could be implied), then we need to consider a distributed transaction management approach (e.g., using a two-phase commit protocol or a saga pattern). Here there are 2 services only, so introducing saga is an overkill. If there were more than 10 microservice, then it would be better to use temporal saga, rather than writing saga orchestrator or choreographer, as temporal does a good job in distributed transaction handling. However, for this simple scenario, local transactions are likely sufficient.

## (B) Threading Model in Service A (JEE or Spring Boot)

The question is about how Service A handles concurrent requests and how it manages threads.

Spring Boot/JEE Default Threading: Both Spring Boot and JEE application servers typically use a thread pool to handle incoming requests. Each request is assigned a thread from the pool.

Implications for Request Processing:

Statelessness: Service A should be designed to be as stateless as possible. By the way we are using REST which is stateless. Request-specific data should be stored within the request context (e.g., request attributes) rather than in instance variables. This avoids thread-safety issues.

Thread Safety: If Service A must use shared resources (e.g., caches), ensure they are thread-safe using appropriate synchronization mechanisms (e.g., locks, concurrent data structures). With experience, it is advisable to avoid usage of threads. Rather asynchronous processing, retries can help achieve a lot of things provided we are using eventually consistency instead of ACID.

Asynchronous Processing: For long-running tasks (e.g., sending the notification to Service B), consider using asynchronous processing (e.g., using Spring's @Async annotation or a message queue) to avoid blocking the request thread.

Sketch: The sketch should show the thread pool handling concurrent requests, with annotations indicating thread-safe components and asynchronous processing.

@startuml

title Thread Pool and Concurrency in Service A

box "Service A" #LightBlue

participant "Thread Pool" as TP

participant "Request Handler" as RH

participant "Thread-Safe Component" as TSC

participant "Asynchronous Processor" as AP

TP -> RH : Incoming Request

activate RH

RH -> TSC : Access Shared Data (Thread-Safe)

activate TSC

TSC -->> RH : Data

deactivate TSC

RH -> AP : Delegate Long-Running Task (Asynchronous)

note right of RH : Non-Blocking Call

activate AP

AP -> Database : Persist Data

activate Database

Database -->> AP : OK

deactivate Database

AP -->> RH : Completion Signal

deactivate AP

RH -->> TP : Response

deactivate RH

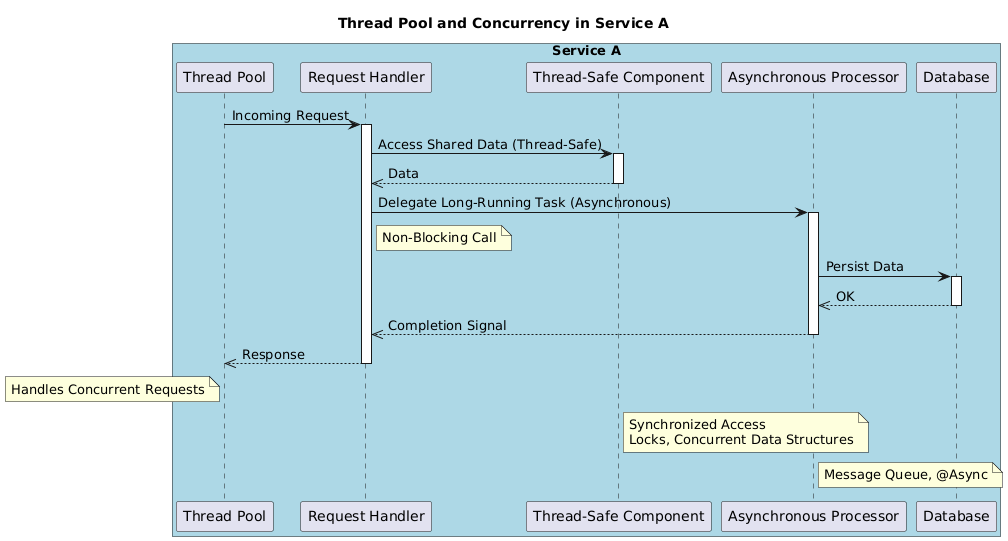
note left of TP : Handles Concurrent Requests

note right of TSC : Synchronized Access\nLocks, Concurrent Data Structures

note right of AP : Message Queue, @Async

end box

@enduml



# (1) Network Communication Issues Between Service A and Service B

## a. Service B Temporarily Unreachable:

Retry Mechanism: Implement a retry mechanism with exponential back off. If Service B is unreachable, Service A should retry the request after a delay, increasing the delay with each attempt.

Circuit Breaker with hysterix and turbine(deprecated) or Resilience4J or Temporal: Use a circuit breaker pattern. If Service B is consistently failing, the circuit breaker should "open," preventing Service A form even attempting to call Service B for a period of time. This prevents cascading failures and gives Service B time to recover. However temporal can own this responsibility and do retries.

Dead Letter Queue (DLQ): When we are using Event driven approach, we usually send events as messages. If retries fail after a certain number of attempts, move the notification message to a dead letter queue of the broker for later analysis and manual reprocessing. Apache Kafka or RabbitMQ can be of use.

## b. Connection to Service B Timing Out or Being Lost:

Timeouts: Configuring appropriate timeouts for the connection to Service B. The timeout should be long enough to allow for normal processing but short enough to detect failures quickly.

Connection Pooling: Usage of a connection pool to manage connections to Service B efficiently. This can help reduce the overhead of establishing new connections.

Heartbeats/Keep-Alive: Implement heartbeats or keep-alive messages to detect broken connections and re-establish them proactively.

# (2) Service A Crashing While Processing a User Request

## a. Possible Inconsistencies:

Partial Persistence: If the crash occurs after persisting some data but before completing the transaction, the database might be in an inconsistent state.

Lost Notification: If the crash occurs after persisting data but before notifying Service B, Service B will not receive the notification.

Incomplete Response: The user might not receive a response, or might receive an incomplete or error response.

## b. Addressing Inconsistencies (Reconciliation):

Transaction Log/Journaling: Use a transaction log or journaling mechanism to record all operations performed by Service A. Upon restart, Service A can replay the log to complete any interrupted transactions.

Idempotency: Design the notification endpoint in Service B to be idempotent. This means that if Service B receives the same notification multiple times, it will only process it once. This can be achieved by including a unique identifier in the notification message.

Database Recovery: Ensure that the database is configured for automatic recovery in case of a crash.

Monitoring and Alerting: Implement monitoring and alerting to detect crashes quickly and trigger recovery procedures.

# Key Considerations

Technology Stack: The specific technologies used (e.g., database, message queue) will influence the implementation details.

Scalability:  The scalability requirements of the system. If Service A needs to handle a large number of requests, consider using a distributed architecture with multiple instances of Service A.

Monitoring: Implement comprehensive monitoring to track the health and performance of Service A and Service B.