Agenda

Database Patterns

- Database Patterns
- Database Per Service Pattern
- Shared Database Pattern
- CQRS Pattern
- Saga Pattern

Agenda

- Database Patterns
- Database Per Service Pattern
- Shared Database Pattern
- CQRS Pattern
- Saga Pattern

Database Patterns

Problem: While defining database architecture for microservices, the following concerns must be addressed:

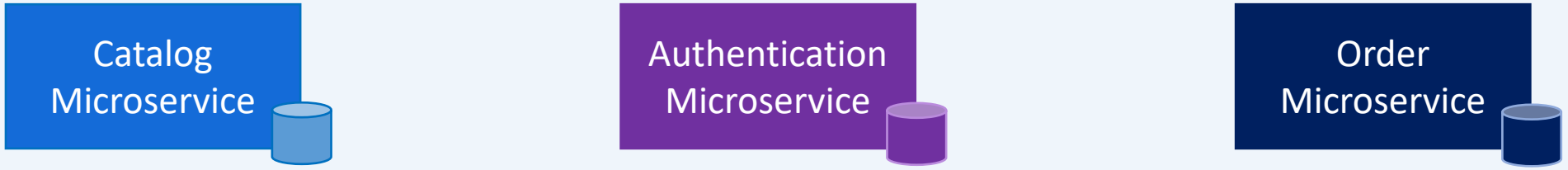
- Services must be loosely coupled and developed, deployed, & scaled independently.
- Business transactions may enforce invariants that span multiple services.
- Some business transactions need to query data that is owned by multiple services.

Solution: There are the following patterns to rescue the above concerns:

- Database Per Service
- Shared Database
- CQRS (Command Query Responsibility Segregation)
- Saga Pattern

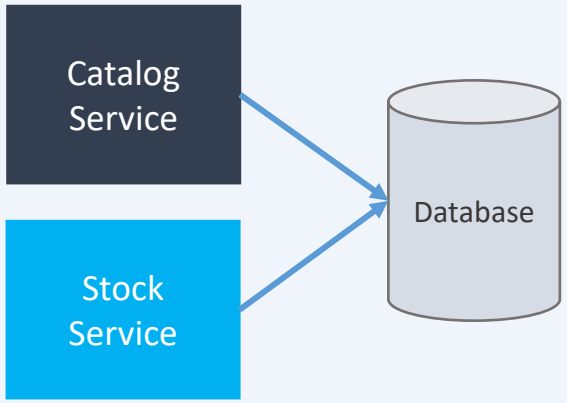
1. Database Per Service

- One database per microservice must be designed; it must be private to that service only. It should be accessed by the microservice API only.
- It cannot be accessed by other services directly.



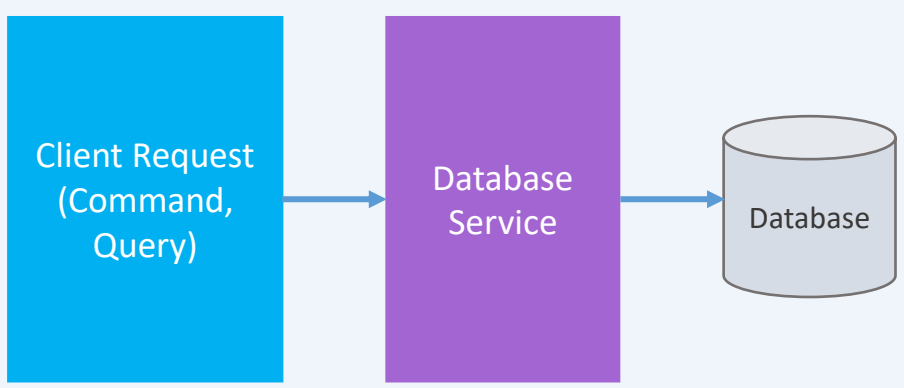
2. Shared Database

- In this pattern, one database can be aligned with more than one microservice, but it has to be restricted to 2-3 max, otherwise scaling, autonomy, & independence will be challenging to execute.
- An ideal solution for legacy applications.

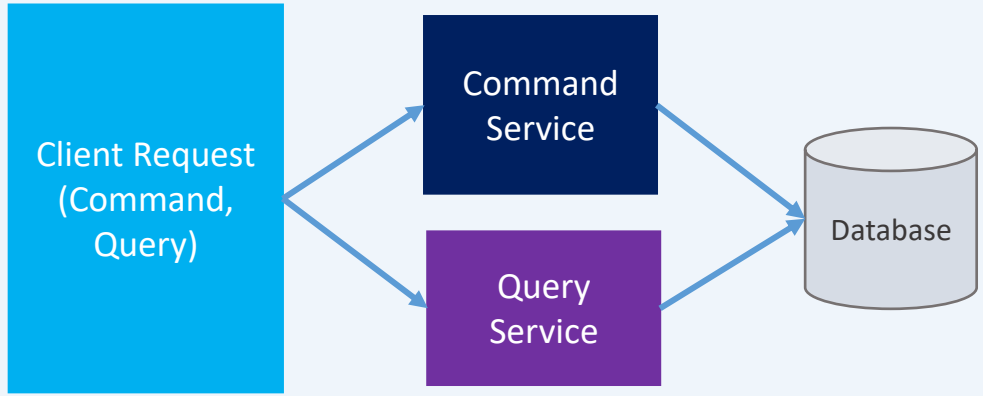


3. Command Query Responsibility Segregation (CQRS)

- Once we implement database-per-service, there is a requirement to query, which requires joint data from multiple services.
- CQRS suggests splitting the application into two parts :
  - Command side - Handles the Create, Update, and Delete requests.
  - Query side - Handles the query part by using the materialized views.



Standard, non-CQRS architecture



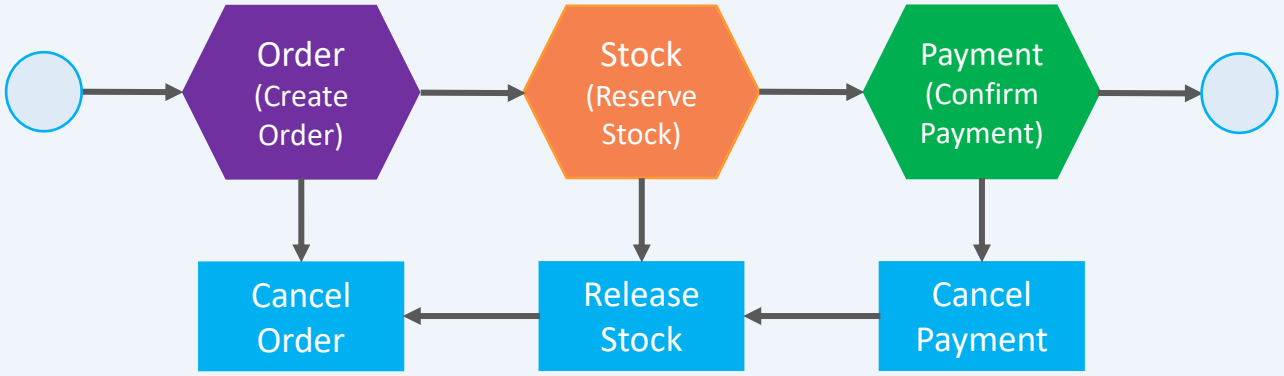
CQRS architecture

ScholarHat

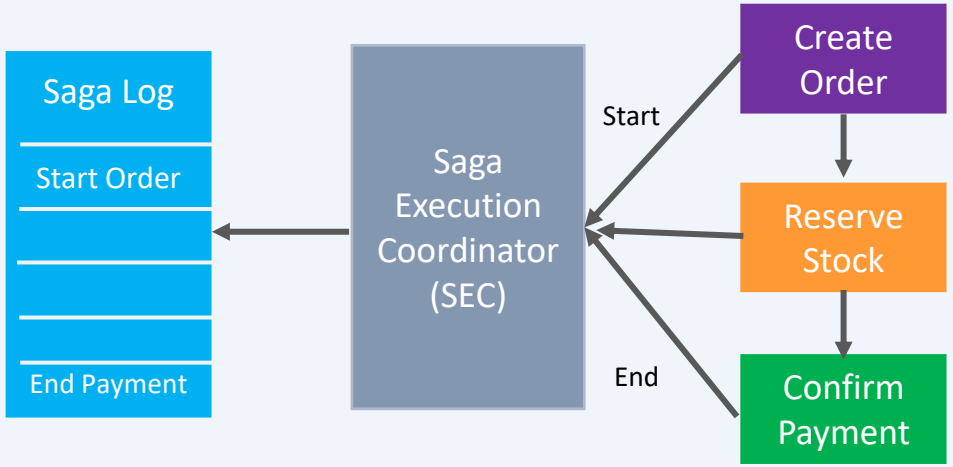
4. Saga Pattern

- **Problem:** When each service has its own database and a business transaction spans multiple services, how do we ensure data consistency across services?
- **Solution:** Saga pattern is the solution.
- A Saga is a sequence of local transactions where each local transaction updates the database and publishes a message or event to trigger the next local transaction. If a local transaction fails then the saga executes a series of **compensating transactions** that undo the changes made by the preceding local transactions.

4.1 Saga Pattern Implementation



Saga Flow



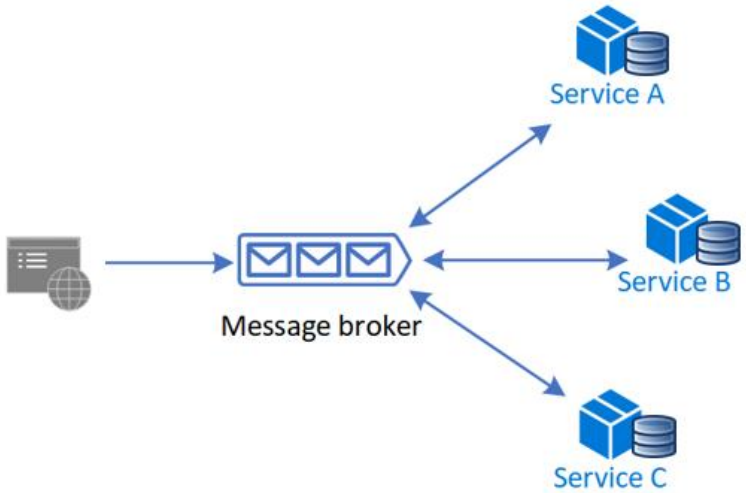
Saga Execution Coordinator

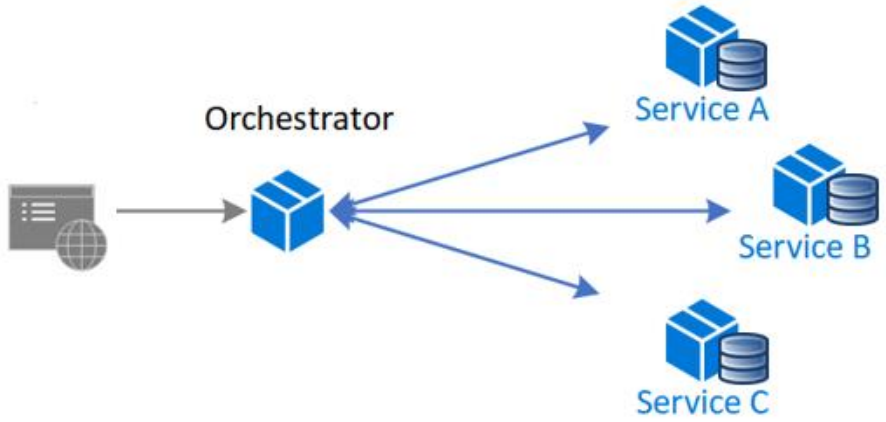
4.1 Saga Execution Coordinator

- The Saga Execution Coordinator (SEC) maintains a Saga log that contains the sequence of events of a particular flow.
- If a failure occurs within any of the components, the SEC queries the logs and helps identify which components are impacted and in which sequence the compensating transactions must be executed.
- If the SEC component itself fails, it can read the SEC logs when coming back, to identify which of the components are successfully rolled back, identify which ones were pending, and start calling them in reverse chronological order.

4.2 Saga Pattern Implementation

- Choreography – each local transaction publishes domain events to message broker that trigger local transaction in other service.





- Orchestration – an orchestrator (object) tells the participants what local transaction to execute.