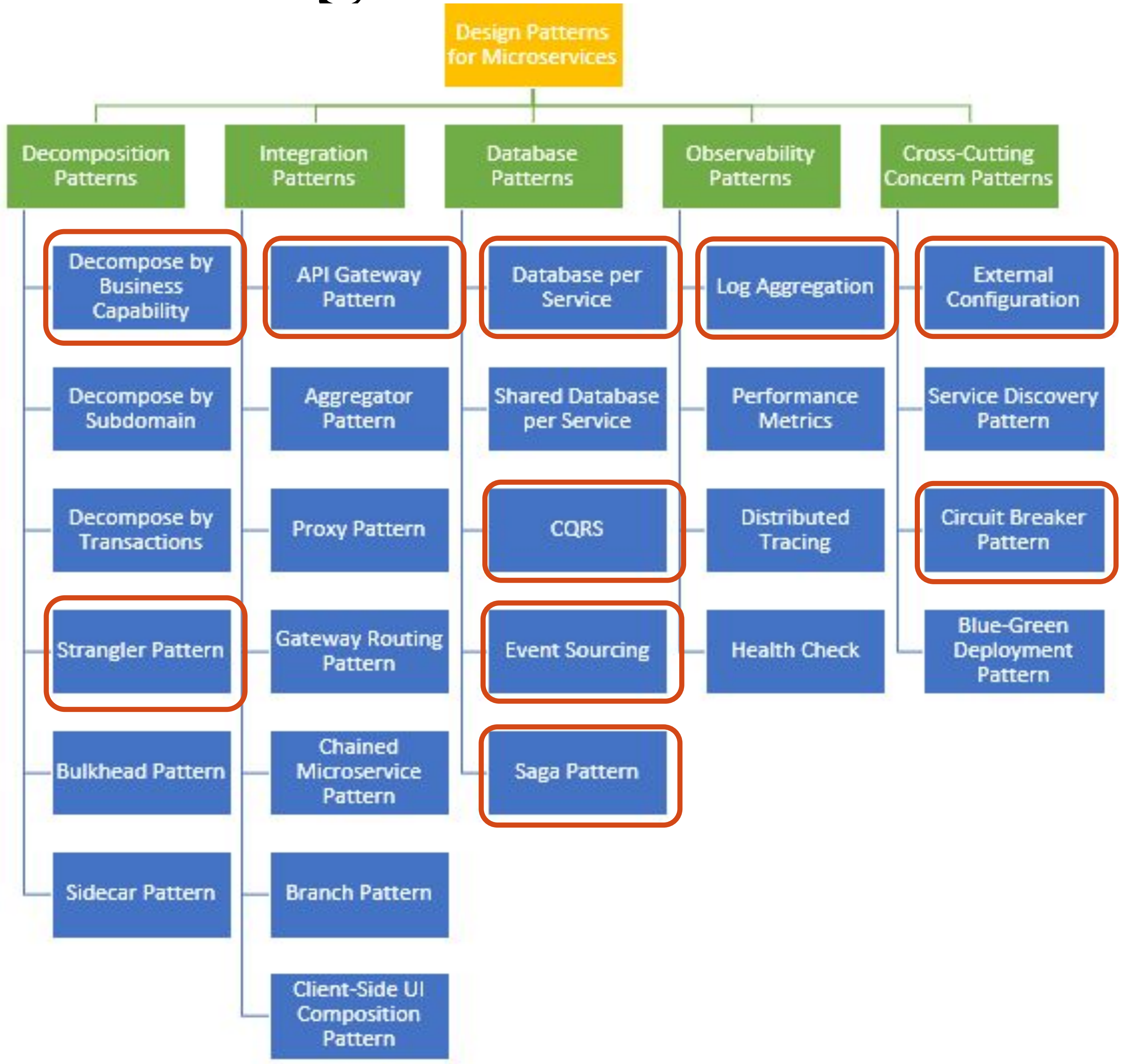
Microservice Design Patterns

For Software Design and Architecture (SWE 4601)

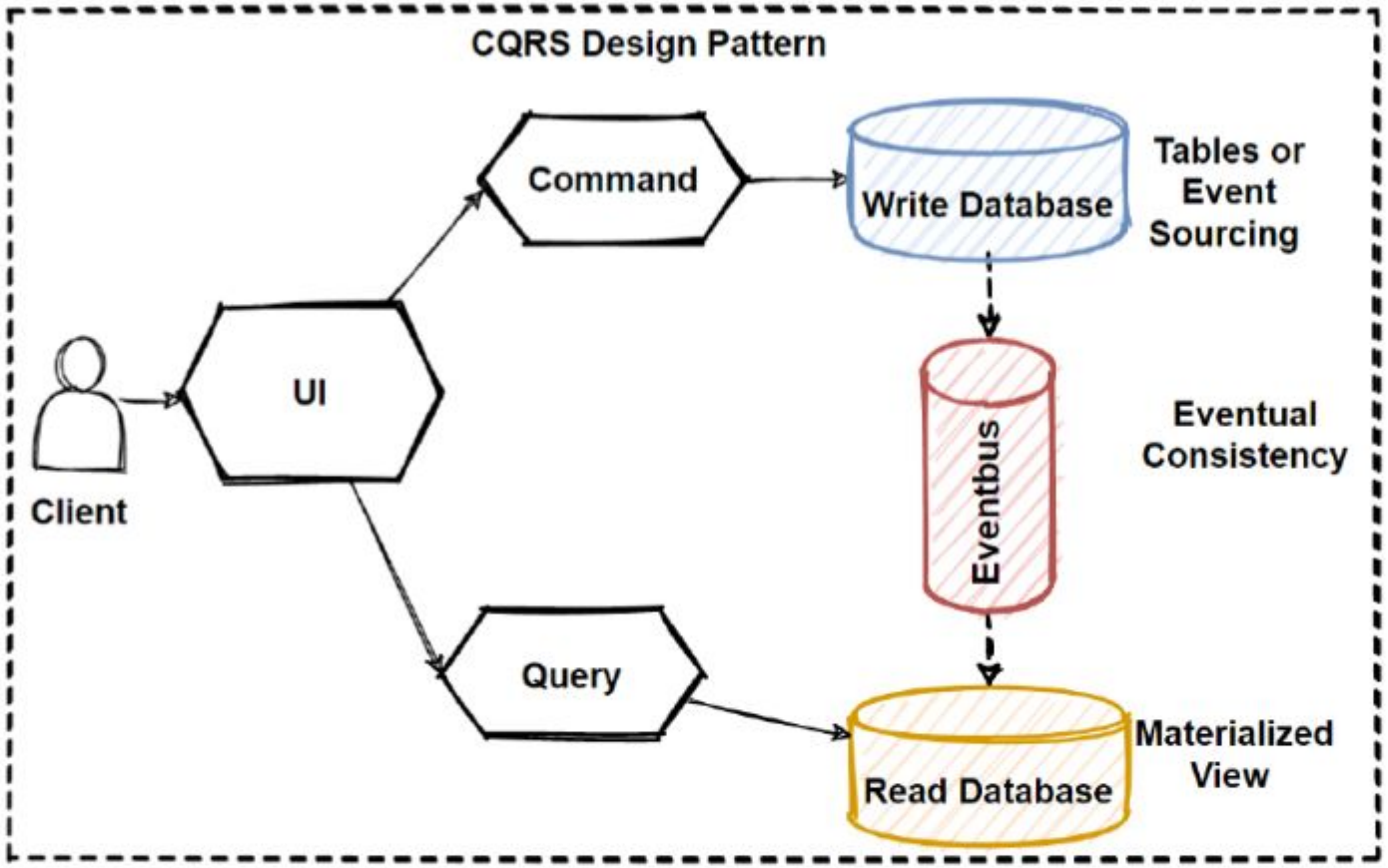
Microservice Design Patterns



CQRS Pattern (Database)

- Command Query Responsibility Segregation Pattern
- Context
  - You have applied the Microservices architecture pattern and the Database per service pattern.
  - As a result, it is no longer straightforward to implement queries that join data from multiple services.
- Problem
  - How to implement a query that retrieves data from multiple services in a microservice architecture?
- Solution
  - Define a view database, which is a read-only replica that is designed to support that query. It indicates the reporting database.

CQRS Pattern (Database)



CQRS Pattern (Database)

- Resulting context
  - Supports multiple denormalized views that are scalable and performant
  - Improved separation of concerns = simpler command and query models
  - Necessary in an event sourced architecture
- This pattern has the following drawbacks:
  - Increased complexity
  - Potential code duplication
  - Replication lag/eventually consistent views
- Related patterns
- The **Database per Service** pattern creates the need for this pattern.
- The **API Composition** pattern is an alternative solution (API Gateway often does API Composition)
- CQRS is often used with **Event sourcing**

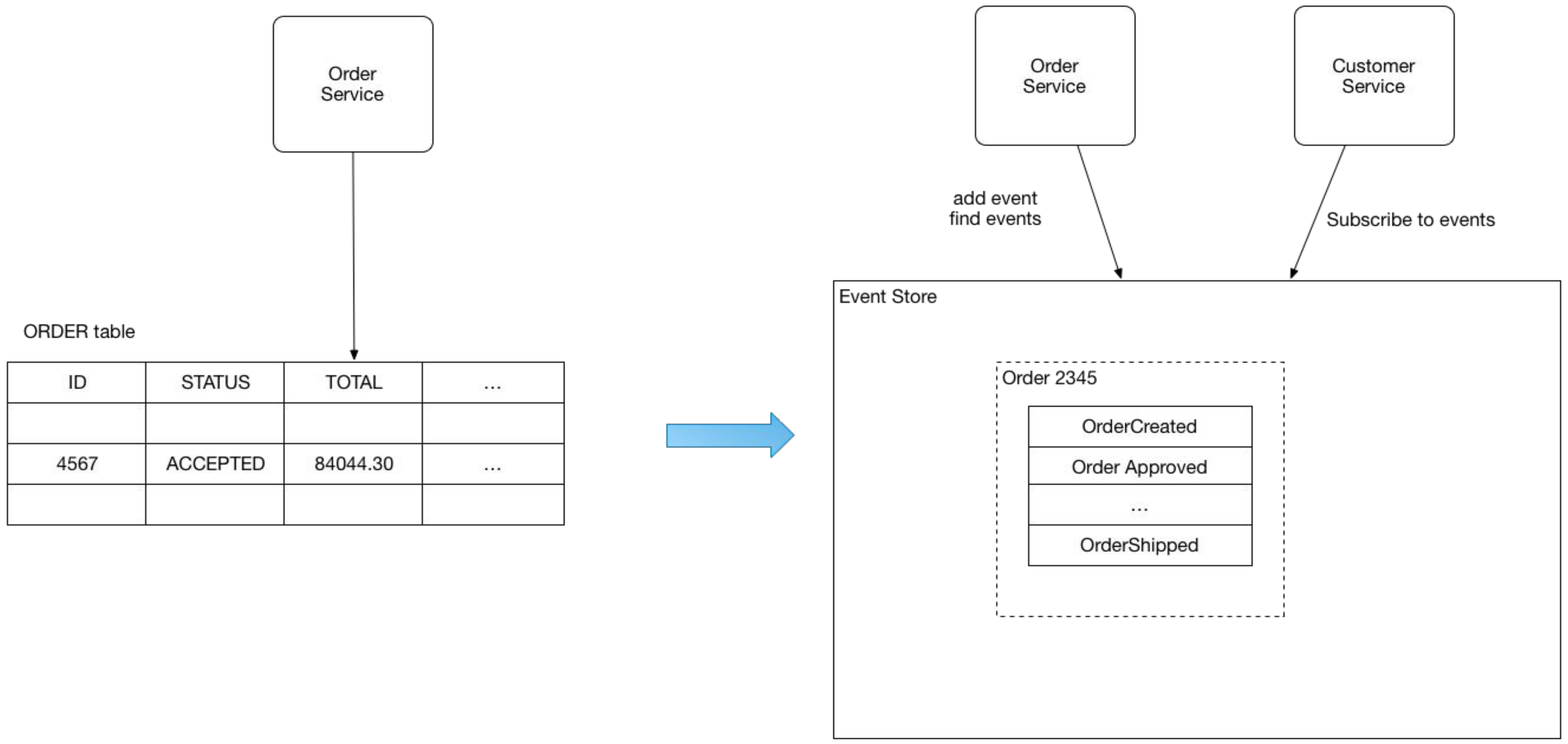
Event Sourcing Pattern (Database)

- Context
  - A service command typically needs to update the database **and** send messages/events.
  - A service must atomically update the database and send messages in order to avoid data inconsistencies and bugs.
- Problem
  - How to reliably/atomically update the database and send messages/events?
- Forces
  - 2PC is not an option
  - If the database transaction commits messages must be sent. Conversely, if the database rolls back, the messages must not be sent
  - Messages must be sent to the message broker in the order they were sent by the service. This ordering must be preserved across multiple service instances that update the same aggregate.

Event Sourcing Pattern (Database)

- Solution
  - A good solution to this problem is to use event sourcing.
  - Event sourcing persists the state of a business entity such an Order or a Customer as a sequence of state-changing events.
  - Whenever the state of a business entity changes, a new event is appended to the list of events.
  - Since saving an event is a single operation, it is inherently atomic. The application reconstructs an entity's current state by replaying the events.
  - Applications persist events in an event store, which is a database of events

Event Sourcing Pattern (Database)



Event Sourcing Pattern (Database)

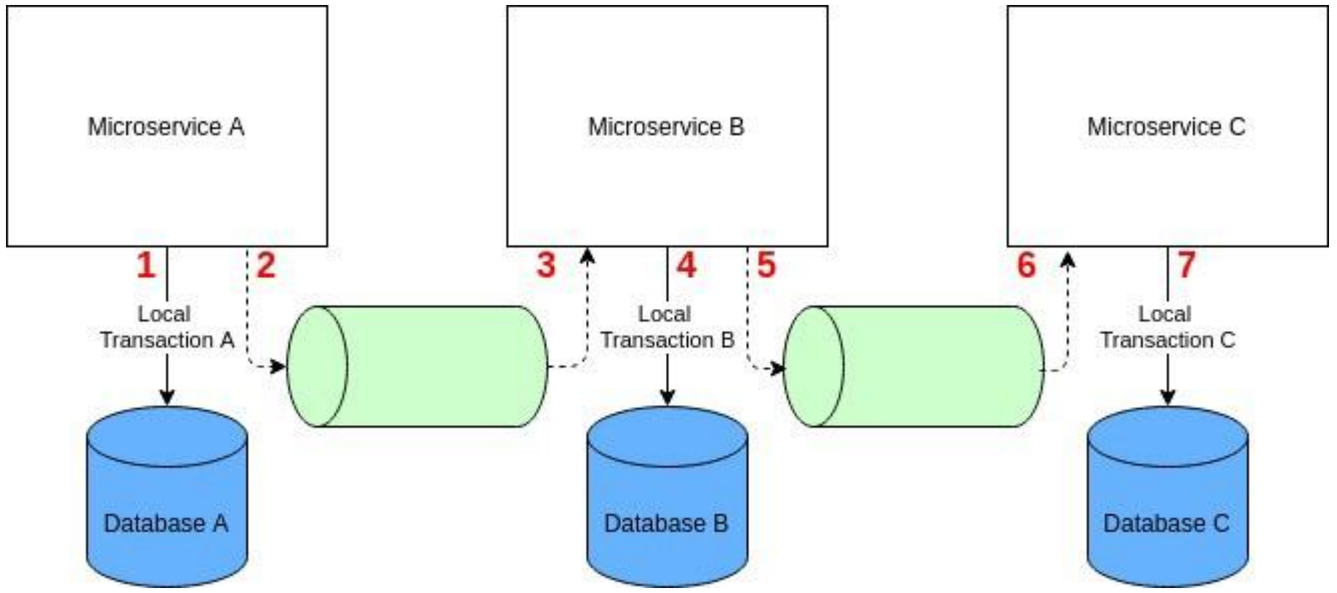
- Resulting context
- Event sourcing has several benefits:
  - It solves one of the key problems in implementing an event-driven architecture and makes it possible to reliably publish events whenever state changes.
  - It provides a 100% reliable audit log of the changes made to a business entity
  - It makes it possible to implement temporal queries that determine the state of an entity at any point in time.
  - Event sourcing-based business logic consists of loosely coupled business entities that exchange events. This makes it a lot easier to migrate from a monolithic application to a microservice architecture.
- Event sourcing also has several drawbacks:
  - It is a different and unfamiliar style of programming and so there is a learning curve.
  - The event store is difficult to query since it requires typical queries to reconstruct the state of the business entities. That is likely to be complex and inefficient
- Related patterns
  - The **Saga patterns** create the need for this pattern.
  - The **CQRS** must often be used with event sourcing.
  - Event sourcing implements the **Audit logging** pattern.

Saga Pattern (Database)

- SAGA
  - saga refers to Long Lived Transactions (LLT). a long, detailed story of connected events
  - Segregated Access of Global Atomicity
- Context
  - You have applied the **Database per Service** pattern. Each service has its own database. Some business transactions, however, span multiple service so you need a mechanism to implement transactions that span services.
  - For example: Order and customer service to check customer credit limit
  - Since Orders and Customers are in different databases owned by different services the application cannot simply use a local ACID transaction.
- Problem
  - How to implement transactions that span services?

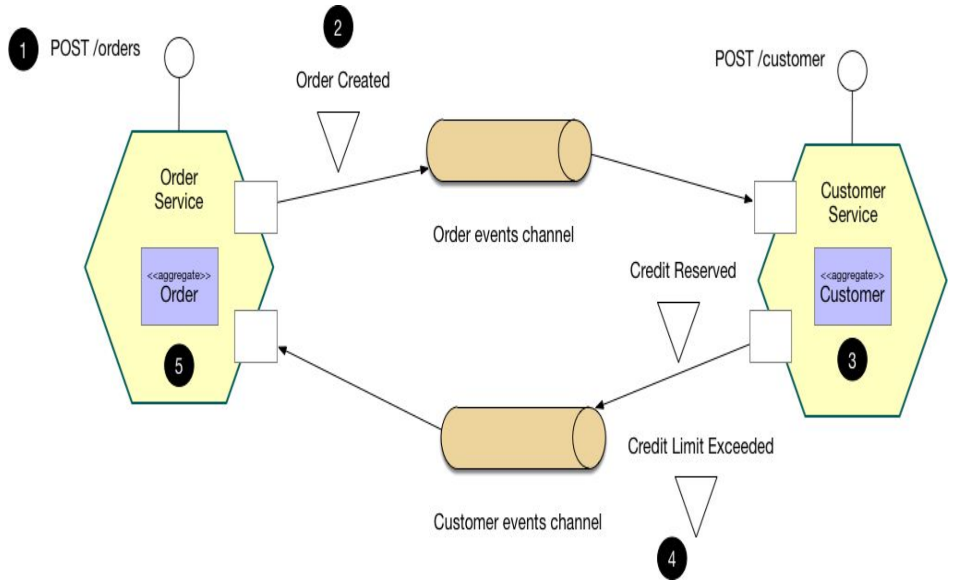
Saga Pattern (Database)

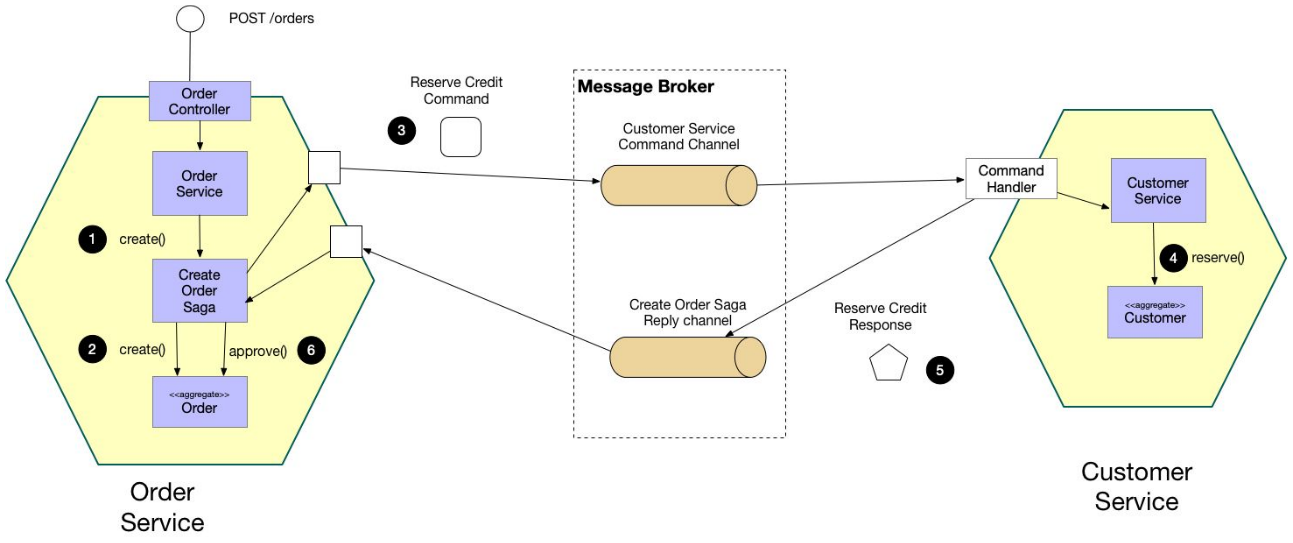
- Solution
  - Implement each business transaction that spans multiple services is a saga.
  - A saga is a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga.
  - If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.



Saga Pattern (Database)

- There are two ways of coordination sagas:
  - Choreography - each local transaction publishes domain events that trigger local transactions in other services
  - Orchestration - an orchestrator (object) tells the participants what local transactions to execute





Saga Pattern (Database)

- Resulting context
- This pattern has the following benefits:
  - It enables an application to maintain data consistency across multiple services without using distributed transactions
- This solution has the following drawbacks:
  - The programming model is more complex. For example, a developer must design compensating transactions that explicitly undo changes made earlier in a saga.
- Related patterns
  - The **Database per Service** creates the need for this pattern
  - **Event sourcing** pattern is a way to *atomically* update state *and* publish messages/events

External Configuration Pattern (Cross Cutting Concern)

- Context
  - An application typically uses one or more infrastructure and 3rd party services. Examples of infrastructure services include: a Service Registry, a message broker and a database server.
  - Examples of 3rd party services include: payment processing, email and messaging, etc.
- Problem
  - How to enable a service to run in multiple environments without modification?
- Forces
  - A service must be provided with configuration data that tells it how to connect to the external/3rd party services. For example, the database network location and credentials
  - A service must run in multiple environments - dev, test, qa, staging, production - without modification and/or recompilation
  - Different environments have different instances of the external/3rd party services, e.g. QA database vs. production database, test credit card processing account vs. production credit card processing account
- Solution
  - Externalize all application configuration including the database credentials and network location. On startup, a service reads the configuration from an external source, e.g. OS environment variables, etc.

Circuit Breaker Pattern (Cross Cutting Concern)

- Context
  - You have applied the Microservice architecture. Services sometimes collaborate when handling requests.
  - When one service synchronously invokes another there is always the possibility that the other service is unavailable or is exhibiting such high latency it is essentially unusable.
  - Precious resources such as threads might be consumed in the caller while waiting for the other service to respond. This might lead to resource exhaustion, which would make the calling service unable to handle other requests. The failure of one service can potentially cascade to other services throughout the application.
- Problem
  - How to prevent a network or service failure from cascading to other services?
- Solution
  - A service client should invoke a remote service via a proxy that functions in a similar fashion to an electrical circuit breaker.

Circuit Breaker Pattern (Cross Cutting Concern)

  - When the number of consecutive failures crosses a threshold, the circuit breaker trips, and for the duration of a timeout period all attempts to invoke the remote service will fail immediately. After the timeout expires the circuit breaker allows a limited number of test requests to pass through. If those requests succeed the circuit breaker resumes normal operation.
  - Otherwise, if there is a failure the timeout period begins again
  - **Closed** – When everything is normal, the circuit breaker remains in the closed state and all calls pass through to the services. When the number of failures exceeds a predetermined threshold the breaker trips, and it goes into the Open state.
  - **Open** – The circuit breaker returns an error for calls without executing the function.
  - **Half-Open** – After a timeout period, the circuit switches to a half-open state to test if the underlying problem still exists. If a single call fails in this half-open state, the breaker is once again tripped. If it succeeds, the circuit breaker resets back to the normal closed state.

- Resulting Context
- This pattern has the following benefits:
  - Services handle the failure of the services that they invoke
- This pattern has the following issues:
  - It is challenging to choose timeout values without creating false positives or introducing excessive latency.

Log aggregation Pattern (Observability)

- Context
  - You have applied the Microservice architecture pattern. The application consists of multiple services and service instances that are running on multiple machines.
  - Each service instance generates writes information about what it is doing to a log file in a standardized format. The log file contains errors, warnings, information and debug messages.
- Problem
  - How to understand the behavior of an application and troubleshoot problems?
- Solution
  - Use a centralized logging service that aggregates logs from each service instance.
  - The users can search and analyze the logs. They can configure alerts that are triggered when certain messages appear in the logs.
- Resulting Issues:
  - handling a large volume of logs requires substantial infrastructure

References

- https://microservices.io/
- https://towardsdatascience.com/microservice-architecture-and-its-10-most-important-design-patterns-824952d7fa41
- https://medium.com/@madhukaudantha/microservice-architecture-and-design-patterns-for-microservices-e0e5013fd58a
- https://techblog.constantcontact.com/software-development/circuit-breakers-and-microservices/