End-Semester Report

Submitted in partial fulfillment of the course:

PRACTICE SCHOOL (BITS F412)

By

Mayank Jain - 2019A7PS0141P

Birla Institute of Technology and Science, Pilani Rajasthan - 333031

June 2023

Table Of Contents

EatStreet

Introduction

The overall architecture of the food ordering system follows a microservices-based approach, with each microservice responsible for a specific domain. The components are designed to communicate through events using Kafka, ensuring loose coupling and scalability. The architecture also incorporates database per service and event sourcing with CQRS patterns.

Components

1. Order Service:
    - Exposes a REST API endpoint that allows clients to create orders using POST /orders and track order Status using GET /orders/{trackingId} requests.
    - Communicates with a local Postgres database for data persistence, Publishes events to Kafka through the messaging component & Implements domain logic related to order processing.
2. Payment Service:
    - Handles the payment process for orders. Communicates with the local database for data access. It subscribes to relevant Kafka topics for event-driven communication & implements business logic related to payment operations.
3. Restaurant Service:
    - Responsible for approving orders based on business logic, such as product availability. It interacts with the local database for data access, subscribes to Kafka topics to consume events & Implements business logic for order approval.
4. Customer Service:
    - Uses a REST endpoint POST /customer and handles creation of customer data in the customer database.
    - Allows the Order Service to query customer data directly from the customers schema. It moves customer data to a table in the order database using event sourcing and CQRS.
5. Kafka:
    - Serves as the central messaging component for event-driven communication. Enables services to publish and consume events. Facilitates communication between the Order Service, Payment Service, and Restaurant Service.
    - Has 5 kafka topics - Namely payment-request-topic, payment-response-topic , restaurant-approval-request-topic, restaurant-approval-response-topic, customer-topic.

Microservices Design Patterns Used -

SAGA

Overview

- The SAGA pattern is used for handling long-running transactions across services. It enables coordination and completion of complex transactions by orchestrating a chain of local asset transactions.
- For this, the first service should fire an event after completing the local ACID transaction. And the second service should read this event and start its own transaction. Thus, a distributed transaction basically requires a publish/subscribe mechanism to handle events across services.
- The SAGA pattern provides a solution for managing distributed long-running transactions across services.
- The project uses the SAGA pattern for distributed transactions involving order, payment, and restaurant services. The order service acts as the SAGA coordinator, and Kafka serves as the event bus for inter-service communication. Services communicate through event emission and listening, following a choreography approach. The Saga pattern is implemented as a coordinated choreography, not a complete orchestration that mandates each service to start saga steps without orchestrator events.

Technical Decision: What is the reason for using a coordinator (Order Service) in the choreography approach of the Saga pattern?

- Saga pattern can be implemented using either Choreography or Orchestration. In Choreography approach, the Saga flow may not be as clear as in Orchestration. To address this issue, we use a coordinator for the Saga in Choreography approach. It provides more control and prevents potential issues. It ensures that the event, such as "Payment Completed," is consumed by the appropriate services, like the Order service and Restaurant service, in a coordinated manner. This helps manage rollbacks and maintain control over the order process.
- The coordinator helps in keeping the Saga step implementations in a single service. This allows for better visibility of the Saga flow and implementation details. While the processing logic remains in the individual services, the coordinator provides a centralized view of the Saga flow, thus simplifying the understanding of the Saga's overall process.

Data Flow

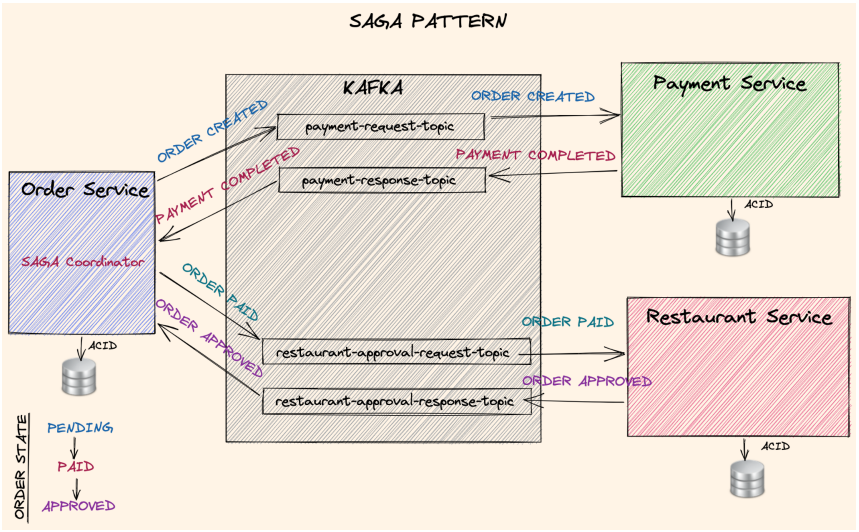
The order process follows the SAGA pattern, with the Order Service acting as the coordinator. The flow involves the following steps:

1. Order Creation:

   The Order Service creates an order in its local database, marking it as pending. Simultaneously, the Order Service sends an "order created" event to the payment service through the payment request topic.

2. Payment Processing:

   The Payment Service listens to the payment request topic and processes the payment in its payment database. Upon successful payment, the Payment Service generates a "payment completed" event, which it sends to the Order Service through the payment response topic. The Order Service consumes the payment response topic and updates its local database, indicating that the order has been paid.



3. Restaurant Approval:

   With the order paid, the Order Service sends an "order paid" event to the Restaurant Service through the restaurant approval request topic. The Restaurant Service processes the request, approving the order in its restaurant database. Upon approval, the Restaurant Service generates an "order approved" event and sends it to the Order Service through the restaurant approval response topic.

4. Order Completion:

   Upon receiving the "order approved" event, the Order Service updates its local database, marking the order as approved. Clients can query the Order Service through the GET orders/{trackingId} endpoint to obtain the approved state of the order.

SAGA Step Interface

1. Purpose:
   - To implement the SAGA pattern, a SAGA step interface is introduced.

    - Each step in the SAGA flow implements this interface and provides the necessary methods for processing and rolling back operations.
2. Methods:
    - The SAGA step interface includes two key methods: process and rollback.
    - The process method is responsible for executing the current step's operation.
    - The rollback method ensures consistency in the SAGA pattern by compensating for previous operations in case of failures. If a Saga step fails, the preceding step must undo its changes through a compensating transaction.

Failure Scenarios and Rollback

To handle failure scenarios in the SAGA pattern, each SAGA step implements a process and rollback method. If a failure occurs at any point, previous operations can be compensated using the rollback methods. The Order Service acts as the coordinator and ensures the integrity of the SAGA flow.

Outbox

Overview

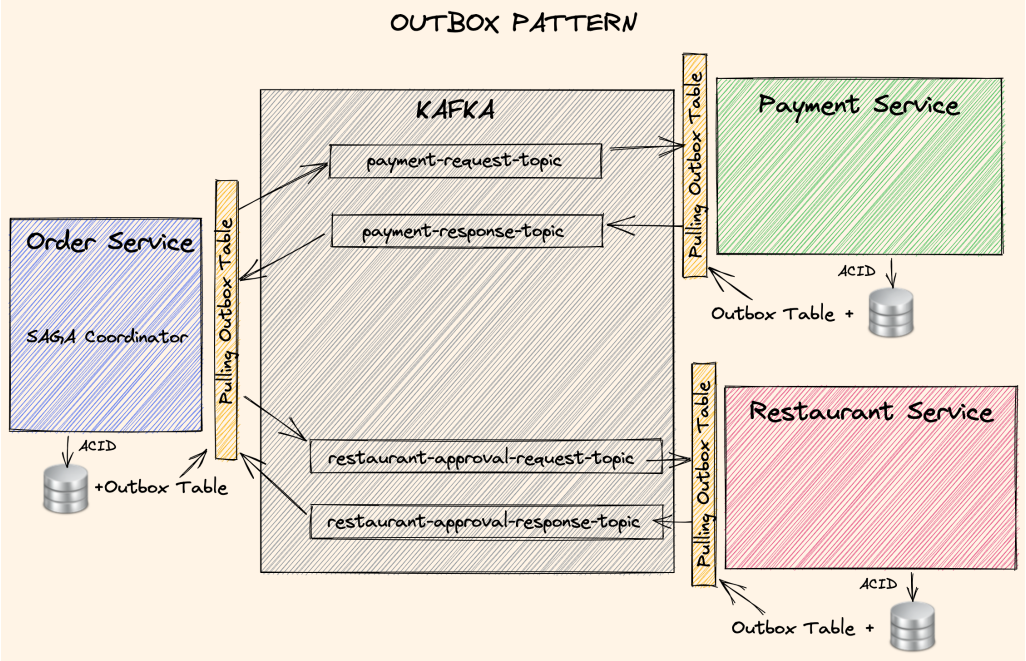
The Outbox Pattern is employed to achieve consistency in distributed transactions, specifically within the SAGA pattern. It addresses the challenge of maintaining consistency when combining local data store transactions and event publishing/consuming operations.

1. Issue with Event Publishing:
    - While applying the Saga pattern, we have two operations at each step. The local ACID transaction for business logic, and the event publishing.
        - If events are published after committing the database transaction, inconsistencies can arise if the publish operation fails.
        - Similarly, if events are published before the database transaction, failures in the database transaction can lead to incorrect events being published.
- The Outbox Pattern introduces an Outbox Table in the local database. Instead of directly publishing events, events are stored in the Outbox Table. This table resides in the same database used for local database operations, enabling the use of a single ACID transaction to ensure consistency.

Implementation:

- The Outbox Pattern can be implemented using two approaches: pulling the Outbox Table data or change data capture (CDC). We use the pulling Outbox Table approach.
- The pulling approach involves periodically polling the data from the Outbox Table and publishing the events.

- The pulling is done by OutboxSchedulerclasses, while the messages in COMPLETED State are deleted everyday at Midnight through OutboxCleanerScheduler classes.



1. Outbox Table:
    - Create an Outbox Table in the local database, dedicated to storing events. This table enables the use of a single ACID transaction for both database operations and event insertion.
2. Event Storage:
    - Instead of directly publishing events, store them in the Outbox Table as part of the local transaction. This ensures that events are created within the local database using ACID transaction guarantees.
3. Event Publishing:
    - Periodically poll the Outbox Table for publishing events. Ensure reliable event publishing to avoid missing events or compromising data integrity.
4. Consistency and Idempotency:
    - Utilize the Outbox Table to track the state changes of the SAGA and order status for each operation.
    - Ensure idempotency by handling duplicate events, leveraging the information stored in the Outbox Table.
    - Apply optimistic locks and database constraints to prevent data corruption in concurrent operations.

Failure Scenarios

Concurrency & Race Condition Handling

When multiple threads simultaneously process SAGA messages, it is possible for race conditions to occur. Synchronizing the processing methods would impact performance, so optimistic locking is used instead.

Optimistic Locking

- The payment outbox message has a version field, which enables optimistic locking for the corresponding JPA entity. In JPA it is implemented by @Version annotation. When data is read from the database, a version column with a value is retrieved.
- After processing logic is run, the version field is checked again before updating the original data.
    - If the version is different, another transaction has updated the data, and the operation is rolled back.
    - If the versions are the same, the original transaction updates the data and increments the version by one.

Concurrency & Race Conditions Handled

- The goal is to protect against multiple threads updating the same data, as it can lead to unexpected behaviors and results.
- When two threads simultaneously try to update the same row, one will get the lock and update the data while the other waits. After the lock is released, the second thread will find that the version has been incremented and will roll back its operation. Optimistic locking is more efficient than pessimistic locking in most cases, but pessimistic locking is still recommended for monitor operations.
- In another scenario where the first thread retrieves the Outbox object but hasn't completed the method, the data is not committed.
- If a second thread tries to access the Outbox object from the database while the first thread is still processing, the behavior depends on the isolation level of the database.
    - If the database has an "uncommitted reads" isolation level, the second thread will not find the data because it has been changed but not committed. The data with the "started" status will no longer be available, and the second thread will return immediately.
    - If the isolation level is "read committed" (the default in Postgres), the second thread will wait for the first transaction to be committed before retrieving the updated data. Since the status will be updated, the result will be empty, and the second thread will also return immediately.
- Optimistic Locking ensures that all scenarios are covered, and there will be no double update with multiple threads.
- The insert operation for approval Outbox also has a unique index on the restaurant approval Outbox table, ensuring that duplicate records cannot be created.

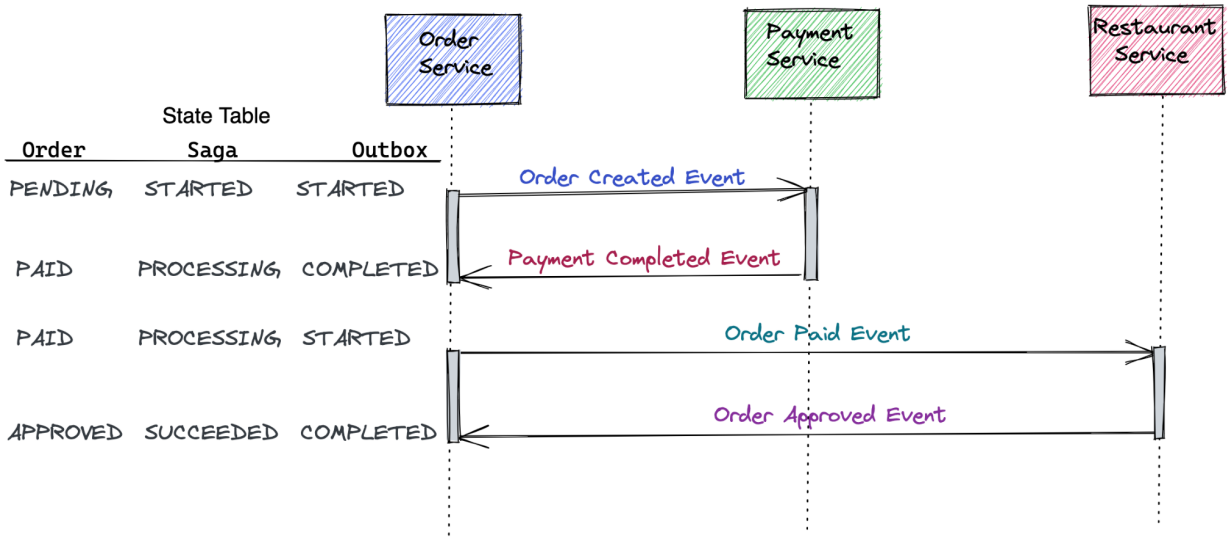
- The unique constraint exception serves as an additional check after optimistic locking. Optimistic Locking Exception does not require a retry. It is caused by a duplicate message so a log is sufficient.

Benefits of Using the Outbox Pattern

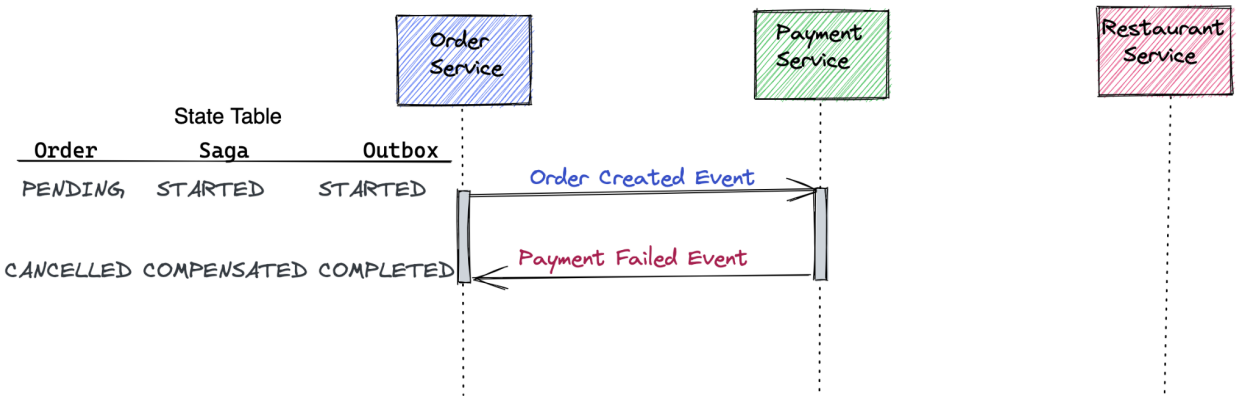
1. Consistent Distributed Transactions:
    - The Outbox Pattern ensures that distributed transactions are completed consistently, preventing inconsistencies between data store operations and event publishing.
2. Fault Tolerance:
    - By using the Outbox Table, failures during event publishing can be handled gracefully.
    - Events are reliably stored in the Outbox Table and can be published later, mitigating the risk of leaving the system in an inconsistent state.
3. Idempotency and Data Integrity:
    - The Outbox Table enables the application to enforce idempotency and maintain data integrity in the face of concurrent operations.
    - Optimistic locks and database constraints can be applied to prevent conflicts and data corruption.

Outbox Pattern Work Flows

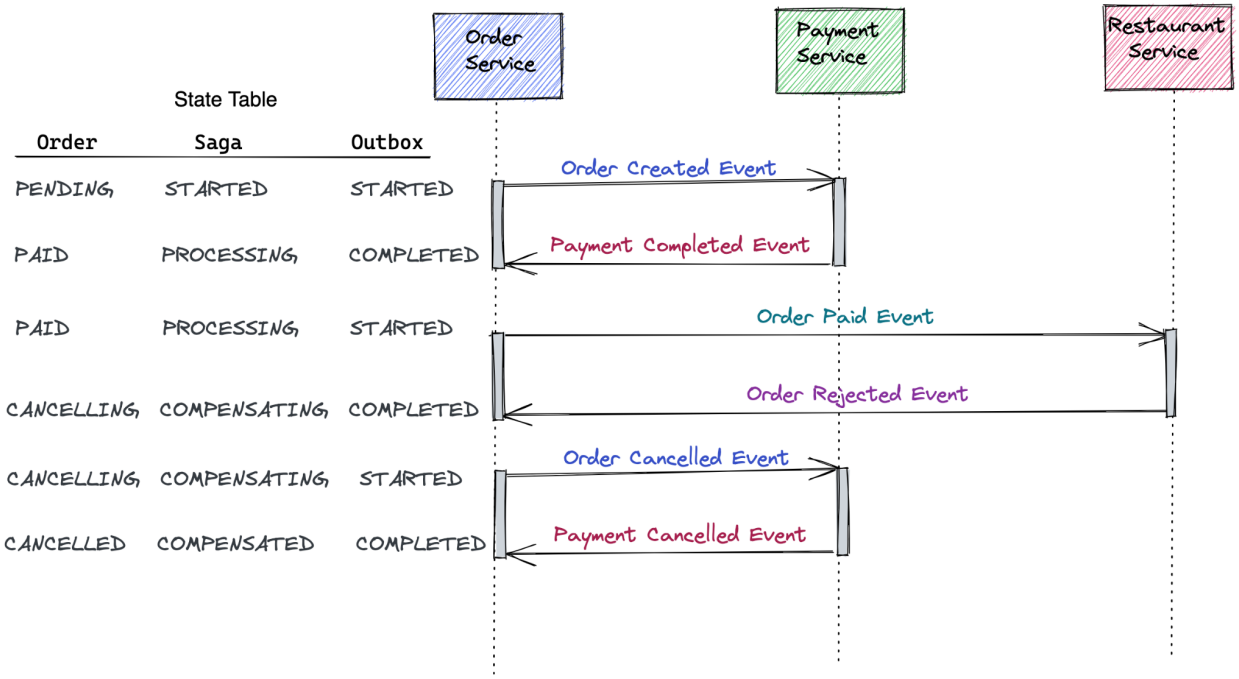
1. Happy Flow (Order Approved) -



2. Order Payment Failure



3. Order Approval Failure



Conclusion

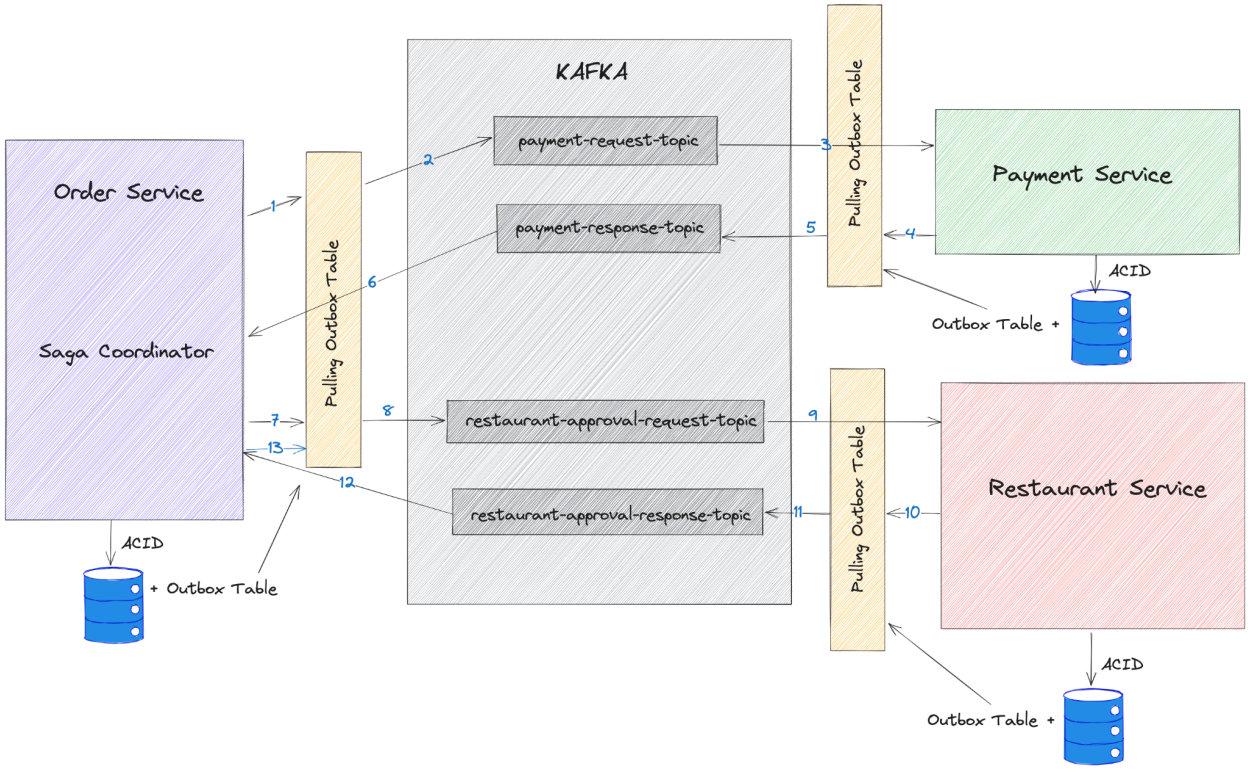
The Outbox Pattern provides a robust solution for achieving consistency in distributed transactions, particularly within the SAGA pattern. By storing events in the Outbox Table and utilizing a single ACID transaction for database operations and event insertion, the Outbox Pattern ensures a safe and consistent execution of distributed transactions. Implementing the Outbox Pattern involves creating the Outbox Table, storing events within the local database, and implementing event publishing mechanisms. The pattern offers benefits such as consistent transactions, fault tolerance, and improved idempotency and data integrity.

Combined Implementation Using SAGA and Outbox Pattern

The SAGA pattern is employed to manage the order process, ensuring consistency and handling compensating transactions. The Order Service acts as the coordinator of the SAGA flow. The Outbox pattern is combined with the SAGA pattern to achieve a consistent solution. Each microservice includes an Outbox table within its local database. The process involves:

1. Instead of directly publishing events, events are saved into the Outbox table within the same asset transaction used for local database operations. This ensures the creation of events is atomic and consistent.
2. Schedulers within each microservice read events from the Outbox table and send them to the message pass, in this case, Kafka. The events are marked as sent only if they are successfully transmitted to Kafka.

ARCHITECTURE DIAGRAM



The Outbox pattern helps address various scenarios, including ensuring idempotency, preventing concurrency issues with optimistic locks and database constraints, and updating SAGA and order status for each operation.

CQRS

Overview

1. Purpose:

   The CQRS (Command Query Responsibility Segregation) pattern segregates the responsibilities of command (write) and query (read) operations in a system. By separating these components, each part can be scaled independently and utilize appropriate technologies or data stores.

2. Command and Query Components:

   - The command/write component handles operations that modify data, such as creating, updating, or deleting entities.
   - The query/read component is responsible for serving queries and retrieving data from a dedicated query store.

3. Eventual Consistency

   As the read store is updated asynchronously at a later time, it will result in an Eventual consistent system. Eventual consistency provides high availability and better performance by sacrificing the strong consistency. CQRS will also eliminate the requirement for distributed transactions if eventual consistency can be accepted.

4. Architecture Components:

   - Command Side: Responsible for handling write operations (e.g., create, update, delete) and persisting changes in the write database. It also creates and sends events representing the changes to the event store.
   - Event Store: Stores events that capture the state changes made in the command side. These events serve as a log and can be replayed for various purposes.
   - Query Side: Consumes events from the event store and updates the query store to keep the read data up-to-date. It creates separate query stores tailored for efficient query operations.
   - Query Store: Maintains denormalized data optimized for read operations, which can be specific to certain use cases (e.g., Elasticsearch for search, analytics database for calculations).

Benefits of Using the CQRS Pattern

1. Scalability:

   The CQRS pattern allows for independent scaling of the write and read components. Write-heavy systems can scale the command side, while read-heavy systems can scale the query side, optimizing performance.

2. Technology Flexibility:

   Separating the write and read components enables the use of different technologies or data stores that best fit each part's requirements. This flexibility allows leveraging specialized tools or databases for specific tasks, enhancing overall system efficiency.

3. Event-driven Architecture:

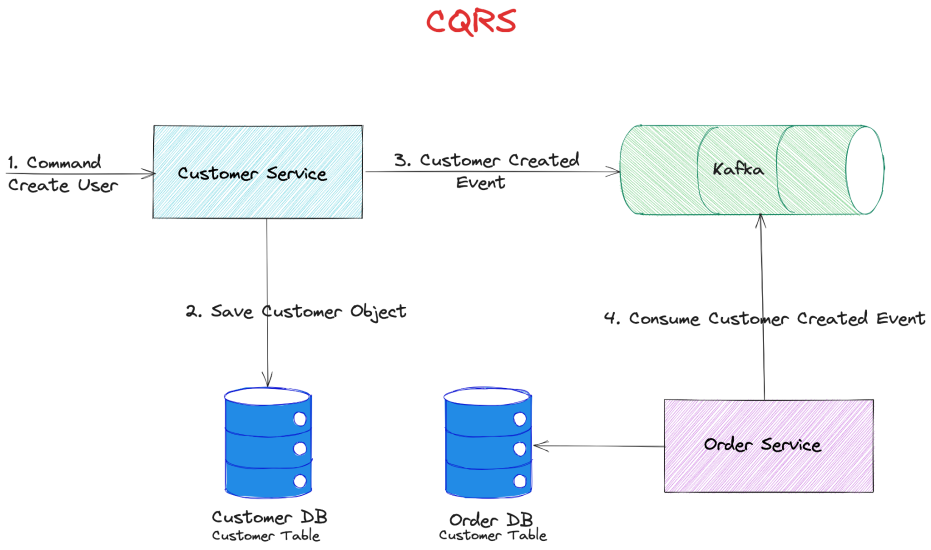
   CQRS promotes an event-driven architecture by capturing events in the event store and consuming them to update query stores. Events serve as a reliable source of data changes and can be replayed for various purposes, such as maintaining data consistency or creating additional query stores.

4. Eventual Consistency:

   - Embracing eventual consistency provides scalability and allows systems to handle large workloads without blocking operations. While read data might have a slight delay in reflecting the latest changes, eventual consistency ensures a highly scalable and responsive system.

Implementation in Customer Service

The Customer Service utilizes event sourcing, and creating a new table within the Order Service database to store customer data.



Conclusion

The CQRS pattern provides a powerful approach to handling write and read operations in a system, enabling scalability, flexibility in technology selection,

and eventual consistency. By segregating the responsibilities and utilizing event-driven architecture, the CQRS pattern empowers developers to build highly scalable and performant systems.

Designing the Microservices

Theory:

Hexagonal and Clean Architecture

The project adopts the hexagonal or clean architecture, which emphasizes isolating domain logic from infrastructure and external dependencies. The architecture consists of different deployable units, each representing a specific component. The components include Data Access, Messaging, REST API, and Business Logic. These components are deployed separately, enabling modular development and independent scalability.

Input and Output Ports

In the domain module, input and output ports are defined as interfaces, representing the entry points and exit points of the domain logic. These interfaces are implemented by adapters, which interact with external systems and infrastructure. For example, the Order Service has input and output ports, implemented by primary and secondary adapters. The primary adapter, an HTTP client, handles incoming requests from clients and invokes the domain logic. The secondary adapters, such as the data access and messaging components, implement output ports and enable communication between the domain layer and external systems.

Clean Architecture

Clean Architecture focuses on organizing the software in a way that keeps the business logic at the center and separates it from infrastructure and external dependencies. The architecture promotes a dependency rule that allows dependencies to point inward toward the domain layer. This is achieved by applying the Dependency Inversion Principle and utilizing polymorphism to invert dependency relations.

The key components of Clean Architecture include:

1. Domain Layer: This layer represents the core business logic and holds entities and use cases. Entities encapsulate critical business rules, while use cases handle application-specific business rules.
2. Interfaces: Interfaces are defined at the boundaries of the domain layer and are used for input and output interactions with external layers. Dependency Injection is employed to pass implementations of these interfaces to the domain layer and external layers.

3. Primary and Secondary Adapters: Primary adapters are components that use the core logic of the domain layer. They utilize input ports to call the implementations defined in the domain layer. Secondary adapters are implemented in external modules such as databases or message queues.

By isolating the domain layer and ensuring that all dependencies point toward it, Clean Architecture enables easily testable applications with independent business logic. It also allows for the separation of development and deployment of different layers and modules.

Hexagonal Architecture (Ports and Adapters)

Hexagonal Architecture, also known as Ports and Adapters, is a concept closely related to Clean Architecture. It divides the software into two main parts: the "inside" and the "outside." The "inside" represents the domain layer that contains the business logic, while the "outside" encompasses all the dependent layers, such as APIs, databases, message queues, and external services.

The key principles of Hexagonal Architecture include:

1. Isolating the Domain: The main principle of Hexagonal Architecture is to isolate the domain layer from any dependencies. This ensures that the business logic remains independent of infrastructure and data sources.
2. Adapters: Adapters are interchangeable components that facilitate communication between the domain layer and external layers. Primary adapters use the core logic of the domain layer, while secondary adapters are implemented in external modules.
3. Interfaces and Dependency Injection: Interfaces are used for input and output interactions at the boundaries of the domain layer. Dependency Injection is employed to pass the implementations of these interfaces to the core logic and external layers.

Hexagonal Architecture provides a flexible and adaptable structure, making it easier to replace adapter implementations without affecting the business layer. It also allows for separate development and deployment of different layers and provides natural isolation, making each layer easier to understand and maintain.

Domain-Driven Design (DDD) Principles

The architecture incorporates strategic and tactical domain-driven design (DDD) patterns. DDD patterns help organize and structure the domain logic effectively. The following DDD concepts are utilized:

1. Entities: Entities are core domain objects that contain critical business logic. They represent the concepts and behavior within the domain. Each Entity must have a unique identifier assigned during creation, which remains unchanged throughout its lifespan. Entities are responsible for running business logic and updating properties based on calculations. It is important to provide well-defined methods for state changes using appropriate verbs.

2. Aggregates: Aggregates are groups of logically related Entity objects. For example, an order process aggregate may include entities such as order, order item, and product. Aggregates should be retrieved and stored as a whole to ensure a consistent state. An Aggregate Root, which owns the aggregate, is responsible for maintaining the consistency of the aggregate and enforcing business invariants. All state-altering operations should go through the Aggregate Root, even if the changes are related to other entities within the aggregate.
3. Value Objects: Value Objects bring context to specific values within the domain. They are used to represent concepts like price, identifier, etc. Value Objects encapsulate the value and provide methods for business operations if necessary. They help to validate the values during construction and are immutable. Creating a new Value Object is required to change its value. Value Objects allow for interchangeability, meaning different Value Objects with the same value can be used for the same purpose. They encapsulate specific concepts and do not have unique identifiers. For instance, a Money object is used instead of direct Java types to represent currency values.
4. Domain Services: Domain services handle business logic that spans multiple aggregates or doesn't fit naturally within any specific entity. They encapsulate complex operations and orchestrate interactions between different components.
5. Application Services: Application services act as the entry point for external layers to communicate with the domain layer. Any layer outside the domain layer should use application services to interact with the domain. Application Services expose the required domain methods to the outside world. They act as an interface for external objects to interact with the domain. Application Services should implement the necessary transactions, security requirements, and data loading/saving using repositories. While the Application Service is responsible for obtaining data, it should not contain any business logic. The data is passed to the Domain Service or Entities for processing.
6. Domain Events: Domain Events facilitate the decoupling of bounded contexts within a system. They are used to notify other contexts about changes that have occurred in a domain. Domain Events enable eventual consistency in the system. Events can be triggered after running business logic on a domain through the Aggregate Root. Domain Event Listeners can subscribe to these events and execute relevant business logic in response. A message queue or event log, such as Kafka, can be used to distribute and consume events, providing a retry mechanism and history tracking. Domain events are used to notify other services running in different contexts about specific occurrences. They facilitate communication and ensure loose coupling between services. In the example, the Order Service creates order-related domain events and sends them to the Payment Service using Kafka topics.

Clean Architecture and Domain-Driven Design (DDD)

Clean Architecture and Domain-Driven Design share similar principles and complement each other in designing software systems. While Clean Architecture focuses on the organization of layers and dependencies, DDD provides concepts and patterns for modeling complex domains.

In Clean Architecture, the core business logic is implemented in entities that represent critical business rules. Use cases orchestrate these entities and handle application-specific business rules. In DDD, entities are still present and encapsulate core business rules. Additionally, DDD introduces the concepts of aggregates and aggregate roots, which ensure consistency within groups of business objects.

DDD also introduces domain services, which handle business logic that spans multiple aggregates or doesn't fit naturally within any specific entity. Application services act as the entry point to the business layer and expose the required business methods to the outside world.

When implementing the domain logic, Clean Architecture focuses on entities and use cases, while DDD incorporates entities, aggregates, aggregate roots, domain services, and application services. Both approaches emphasize isolating the domain layer and attaching lower-level dependencies such as user interfaces, databases, and external services.

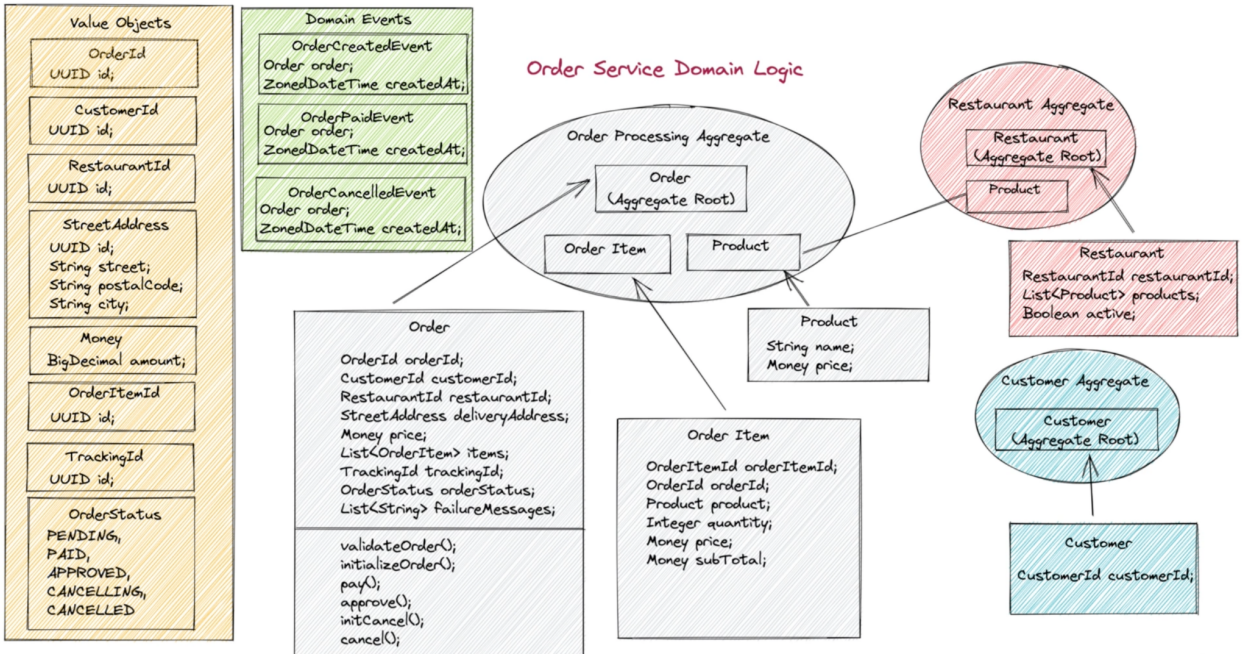
Advantages and Considerations

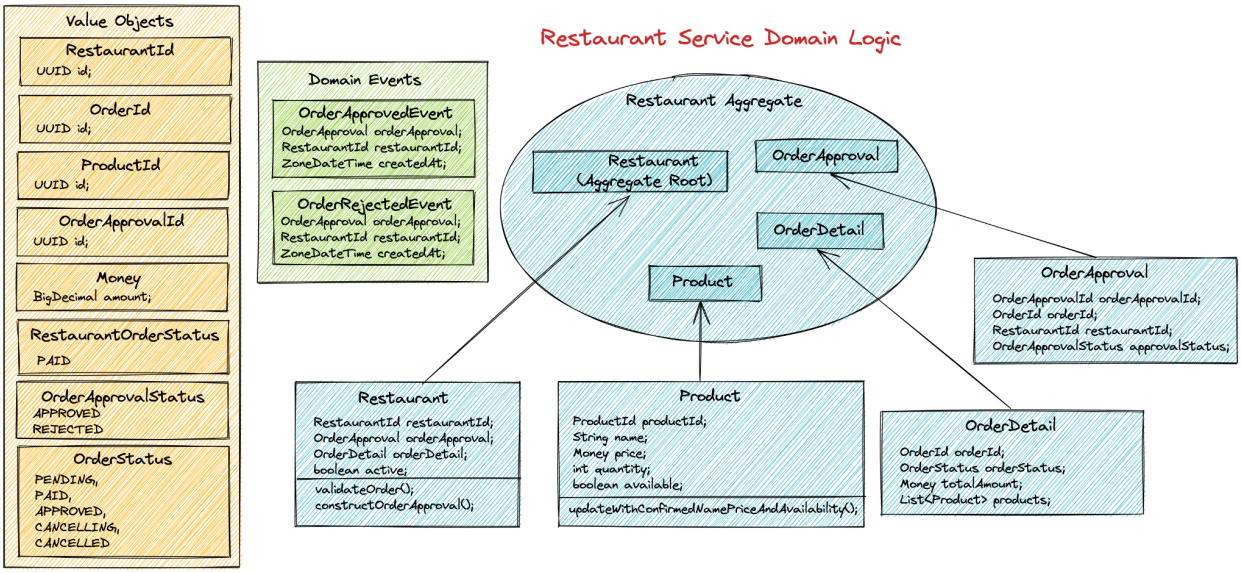
Clean and Hexagonal Architectures offer several advantages:

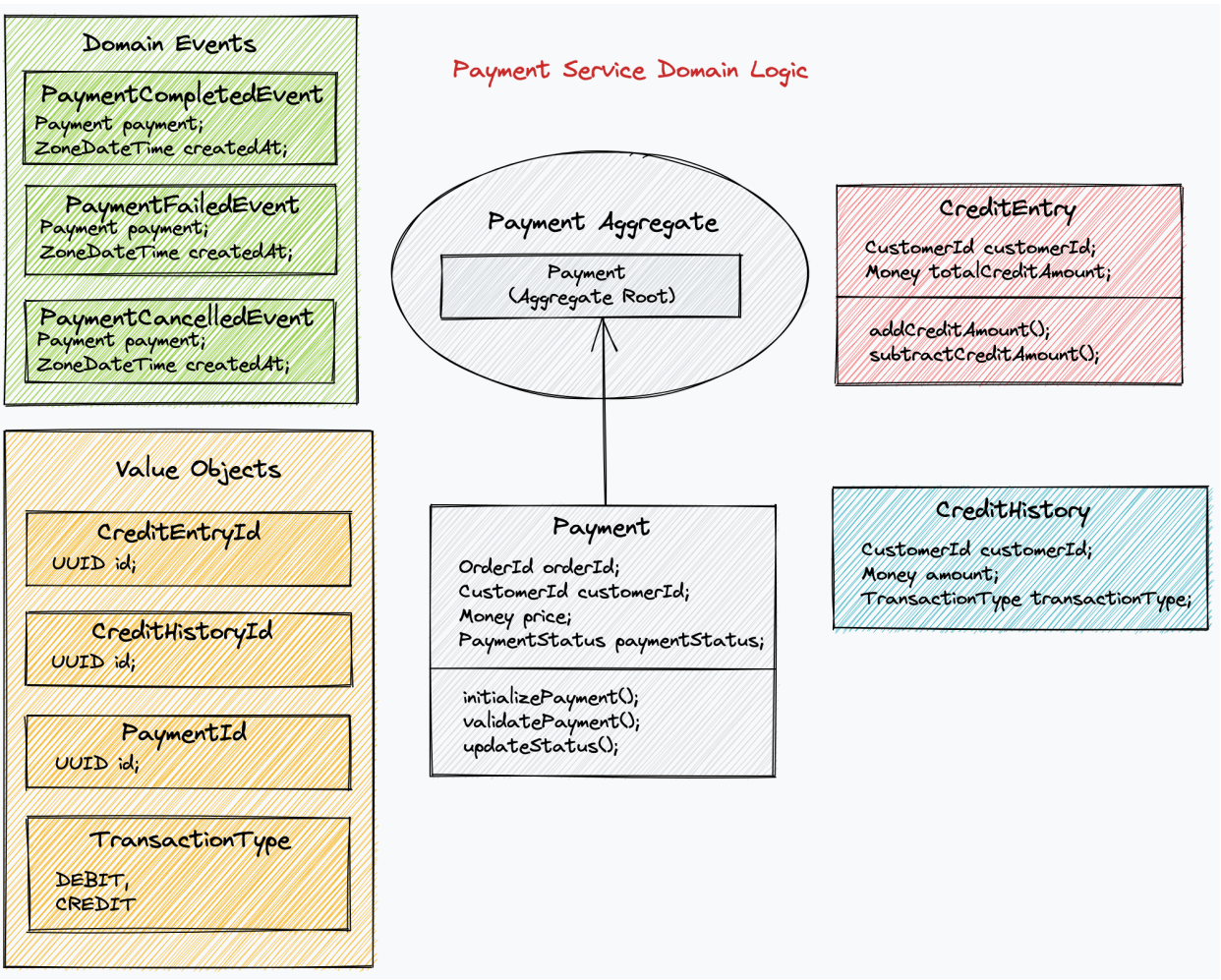
1. Modularity and Testability: The architectures promote modularity by isolating the business logic and separating layers, making the codebase easier to understand, maintain, and test.
2. Independence from External Elements: By separating the domain layer from external dependencies, the architectures allow for easy replacement of adapters and data sources without affecting the core business logic.
3. Flexibility and Adaptability: The architectures enable the delay of implementation decisions, allowing for the adoption of new technologies and frameworks while keeping the business logic intact.

However, it's important to consider that Clean and Hexagonal Architectures may require writing additional code compared to traditional layered architectures. This additional effort includes creating separate data transfer objects and handling data mappings. Despite this overhead, the architectures provide long-term maintainability and testability benefits.

Domain Driven Design of Our Microservices


Value Objects
OrderId
UUID id;
CustomerId
UUID id;
RestaurantId
UUID id;
StreetAddress
UUID id;
String street;
String postalCode;
String city;
Money
BigDecimal amount;
OrderItemId
UUID id;
TrackingId
UUID id;
OrderStatus
PENDING,
PAID,
APPROVED,
CANCELLING,
CANCELLED
Domain Events
OrderCreatedEvent
Order order;
ZonedDateTime createdAt;
OrderPaidEvent
Order order;
ZonedDateTime createdAt;
OrderCancelledEvent
Order order;
ZonedDateTime createdAt;
Order Service Domain Logic
Order Processing Aggregate
Order
(Aggregate Root)
Order Item
Product
Restaurant Aggregate
Restaurant
(Aggregate Root)
Product
Restaurant
RestaurantId restaurantId;
List<Product> products;
Boolean active;
Order
OrderId orderId;
CustomerId customerId;
RestaurantId restaurantId;
StreetAddress deliveryAddress;
Money price;
List<OrderItem> items;
TrackingId trackingId;
OrderStatus orderStatus;
List<String> failureMessages;
validateOrder();
initializeOrder();
pay();
approve();
initCancel();
cancel();
Order Item
OrderItemId orderItemId;
OrderId orderId;
Product product;
Integer quantity;
Money price;
Money subTotal;
Product
String name;
Money price;
Customer Aggregate
Customer
(Aggregate Root)
Customer
CustomerId customerId;


Value Objects
RestaurantId
UUID id;
OrderId
UUID id;
ProductId
UUID id;
OrderApprovalId
UUID id;
Money
BigDecimal amount;
RestaurantOrderStatus
PAID
OrderApprovalStatus
APPROVED
REJECTED
OrderStatus
PENDING,
PAID,
APPROVED,
CANCELLING,
CANCELLED
Domain Events
OrderApprovedEvent
OrderApproval orderApproval;
RestaurantId restaurantId;
ZoneDateTime createdAt;
OrderRejectedEvent
OrderApproval orderApproval;
RestaurantId restaurantId;
ZoneDateTime createdAt;
Restaurant Service Domain Logic
Restaurant Aggregate
Restaurant
(Aggregate Root)
OrderApproval
OrderDetail
Product
Restaurant
RestaurantId restaurantId;
OrderApproval orderApproval;
OrderDetail orderDetail;
boolean active;
validateOrder();
constructOrderApproval();
Product
ProductId productId;
String name;
Money price;
int quantity;
boolean available;
updateWithConfirmedNamePriceAndAvailability();
OrderDetail
OrderId orderId;
OrderStatus orderStatus;
Money totalAmount;
List<Product> products;
OrderApproval
OrderApprovalId orderApprovalId;
OrderId orderId;
RestaurantId restaurantId;
OrderApprovalStatus approvalStatus;


Domain Events
PaymentCompletedEvent
Payment payment;
ZoneDateTime createdAt;
PaymentFailedEvent
Payment payment;
ZoneDateTime createdAt;
PaymentCancelledEvent
Payment payment;
ZoneDateTime createdAt;
Payment Service Domain Logic
Payment Aggregate
Payment
(Aggregate Root)
CreditEntry
CustomerId customerId;
Money totalCreditAmount;
addCreditAmount();
subtractCreditAmount();
Value Objects
CreditEntryId
UUID id;
CreditHistoryId
UUID id;
PaymentId
UUID id;
TransactionType
DEBIT,
CREDIT
Payment
OrderId orderId;
CustomerId customerId;
Money price;
PaymentStatus paymentStatus;
initializePayment();
validatePayment();
updateStatus();
CreditHistory
CustomerId customerId;
Money amount;
TransactionType transactionType;

Implementation of Clean and Hexagonal Architecture using Domain Driven Design in Order Service

Each service is structured in the following manner -

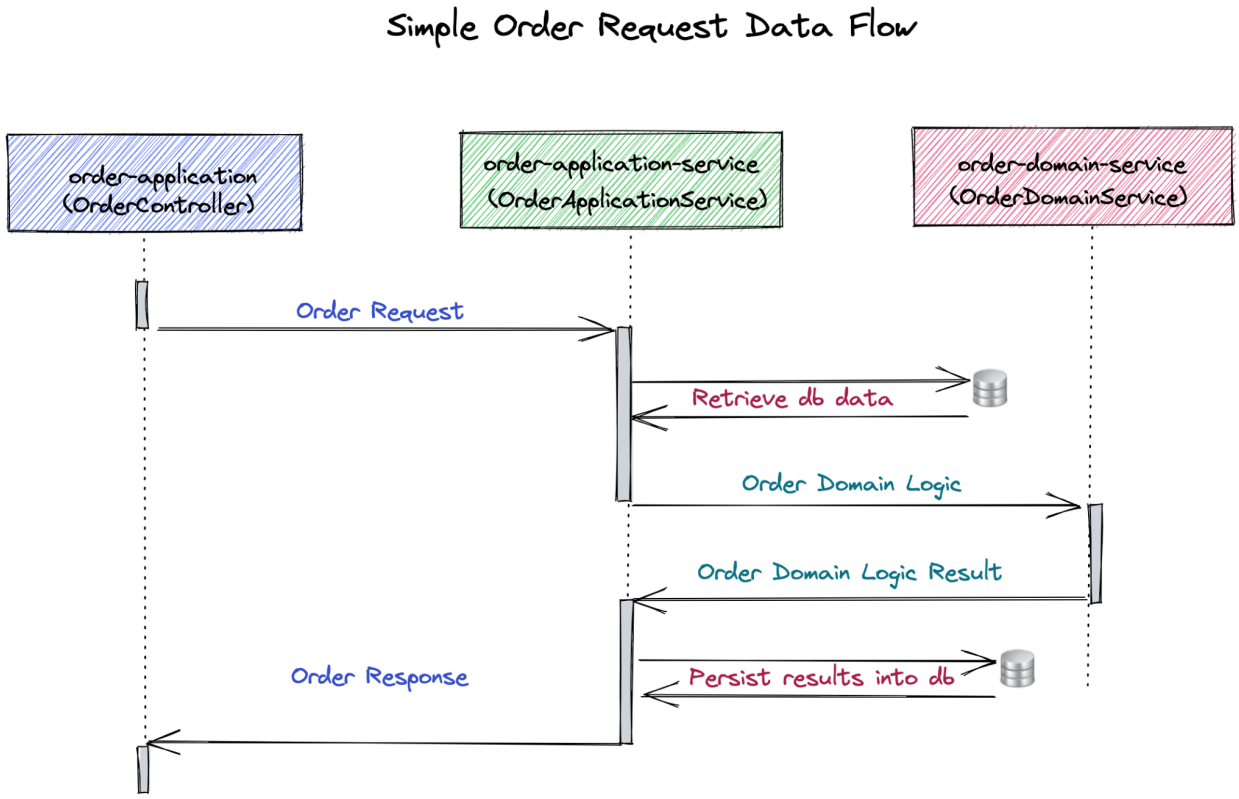
```
order-service
├── order-application (Entrypoint)
│   └── com.food.ordering.system.order.service.application
│                       ├── exception.handler
│                       └── rest (REST Controllers)
├── order-container (Runnable Jar to package the service)
│   └── com.food.ordering.system.order.service.domain
│                     ├── BeanConfiguration.java (Bean decl. for Domain-Service)
│                     └── OrderServiceApplication.java
├── order-dataaccess (Data Layer)
│   └── com.food.ordering.system.order.service.dataaccess
│                             └── table_name
│                                 ├── adapter (Repository implentation)
│                                 ├── entity (DB entity definitions)
│                                 ├── mapper (DB entity <-> Domain entity)
│                                 └── repository (Repository definition interface)
├── order-domain (Domain Layer)
│   ├──order-application-service (Adapters of input ports of target domain layer)
│   │   └── com.food.ordering.system.order.service.domain
│   │                   ├── config (Kafka topic names)
│   │                   ├── dto (REST Endpoint objects)
│   │                   │   └── endpoint
│   │                   ├── mapper (Request/Response <-> Domain entities)
│   │                   ├── outbox
│   │                   │   ├── model (Kafka message payloads)
│   │                   │   │   └── kafka_topic_name
│   │                   │   └── scheduler (Pulling schedulers, cleaners)
│   │                   │       └── kafka_topic_name
│   │                   ├── saga (Saga Step Handlers/Adapters for ports)
│   │                   └── ports (Interfaces)
│   │                       ├── input
│   │                       │   ├── message.listener
│   │                       │   │   └── input_kafka_topic_name
│   │                       │   └── service (REST Endpoint commandHandlers)
│   │                       └── output
│   │                           ├── message.publisher
│   │                           │   └── output_kafka_topic_name
│   │                           └── repository (DB Operations)
│   └── order-domain-core (Core business logic)
│       └── com.food.ordering.system.order.service.domain
│            ├── entity (Object definitions)
│            ├── event (Domain Events)
│            ├── exception (Domain exceptions)
│            └── valueobject (Domain Valueobjects)
└── order-messaging (Messaging Layer)
```

```
└── com.food.ordering.system.order.service.messaging
                                        ├── listener.kafka
                                        ├── mapper (Avro models <-> Domain dtos)
                                        └── publisher.kafka
```

Application service is the point of contact to the outside of domain. So any other layer that wants to communicate with domain layer, needs to use application service. It will expose the required business layer methods to the outside using input ports, and these ports will be implemented in the domain layer itself. In the application service we can also have the data mappings, validations and transaction management. Application service is responsible to make any data adapter calls and gather the data to pass into domain service and entities.

Domain service coordinates the business logic that spans multiple aggregates. Domain service and entities has the core business logic and they don't deal with things like gathering data, mapping the data or validating the data. Those are the responsibility of application service.Also we can put business logic methods into business service, if the method does not logically fit into any entity. Domain service can also communicates with other domain services if necessary. Note that the domain service is still in the core of the domain logic, so it cannot be reached from outside.

Order Service Internal Data Flow -



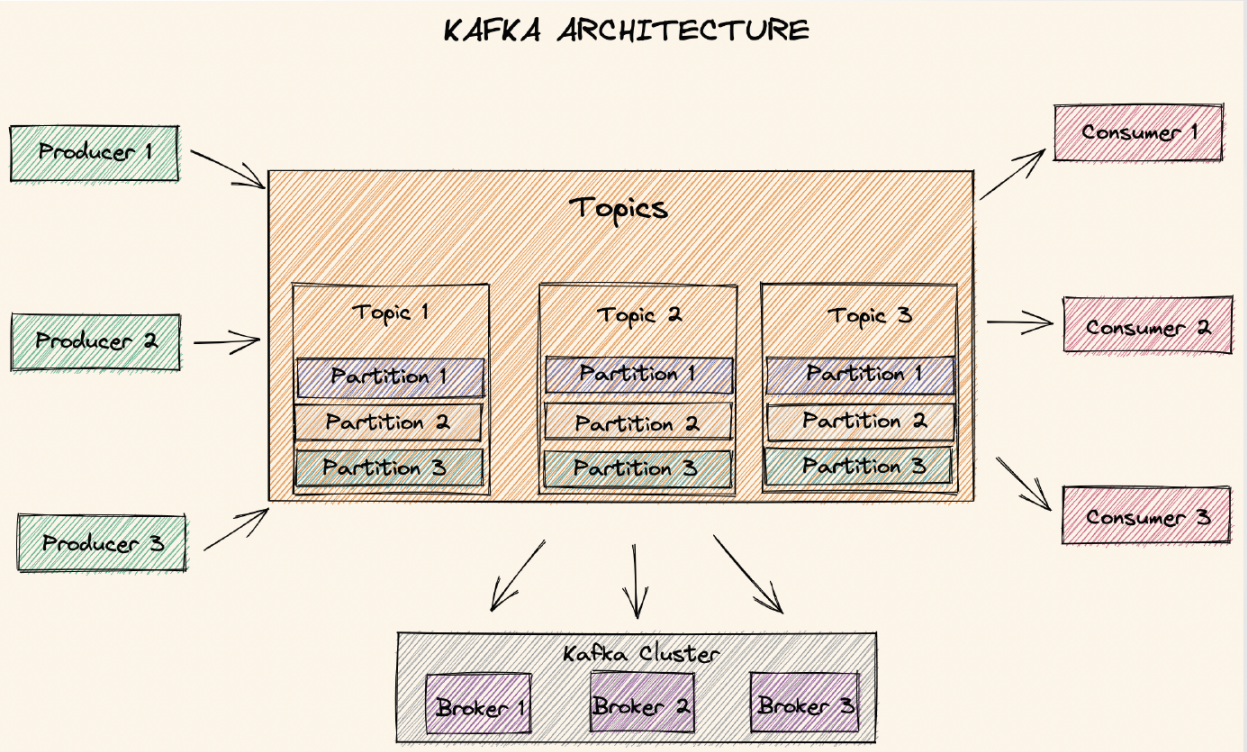
Microservice Communication

Kafka

Architecture Components

1. Brokers:
    - Kafka Brokers: Servers that form a cluster and serve producers and consumers by handling data insertion and retrieval. Multiple brokers in a cluster provide high availability and resiliency by replicating data across different nodes.
2. Topics and Partitions:
    - Topics: Logical data structures that consist of partitions. Topics represent categories or streams of records in Kafka.
    - Partitions: Smallest physical storage units that hold data. They allow for parallel processing and scalability. Data can be replicated across partitions on different brokers for increased resiliency using the replication factor property.

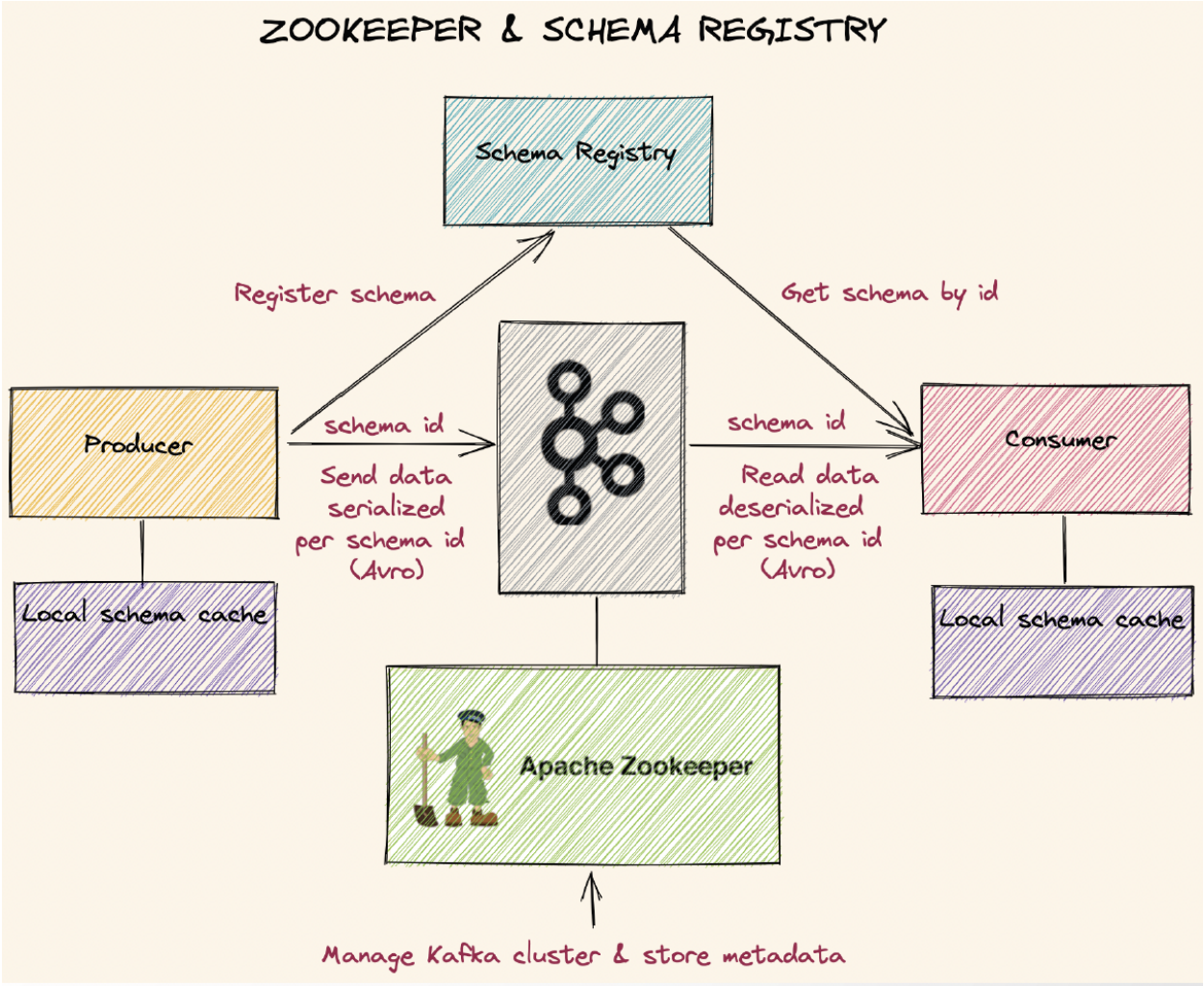
3. Producers and Consumers:
    - Producers: Write data to the end of a partition.
    - Consumers: Read data from the start of a partition and maintain an offset (ID) to continue reading from the last consumed position. Consumers can be organized into consumer groups, where each consumer group reads the data only once to achieve parallel processing and avoid duplication.



4. Resiliency and Scalability:
    - Resiliency: Achieved through data replication using the replication factor property, which duplicates data across partitions on different broker nodes. This ensures that the same data is stored on multiple brokers, enhancing system resiliency.
    - Easy Scaling: Kafka supports parallel consumer threads, allowing for concurrent consumption of data. Each partition in a topic can be consumed by a separate consumer thread, enabling easy scaling by creating new partitions and corresponding consumers.
5. Immutability and Speed:
    - Immutability: Data inserted into Kafka partitions is treated as an append-only log and cannot be changed or updated. This immutability ensures data consistency and simplicity.
    - Speed: Kafka achieves fast data insertion by appending data to the end of partitions without the need for search operations. This enables high throughput and low latency.
6. Other Components:
    - ZooKeeper: A centralized service for maintaining configuration information, providing distributed synchronization, and facilitating coordination across a Kafka Cluster. ZooKeeper is

essential for Kafka's operation and helps maintain the overall system's health and reliability. Currently used in production applications to manage the Kafka cluster and hold metadata.

- Schema Registry: A component that enforces strict schema compatibility between producers and consumers. It allows producers to register schemas and obtain an ID, which is used for serialization. Consumers use the ID to retrieve the schema from the Schema Registry and deserialize the data. Local caches on the producer and consumer sides prevent repeated network calls to the registry.



Technical Decisions

1. Replication Factor: The decision to configure the replication factor determines the level of resiliency and fault tolerance in the Kafka cluster. A higher replication factor provides stronger resiliency by replicating data across more broker nodes. Three topics are defined with three partitions and a

replication factor of three. This configuration enables concurrent data insertion and provides resiliency by replicating data across multiple brokers.

2. Parallel Consumer Threads: Utilizing parallel consumer threads enables efficient and concurrent consumption of data, allowing for easy scaling as new consumers can be added to handle additional partitions.
3. Immutability for Data Consistency: Kafka's immutable data model ensures that once data is inserted, it cannot be changed or updated. This simplicity guarantees consistency and eliminates the need for complex data modification operations.
4. Schema Registry for Schema Management: The inclusion of the Schema Registry component enables strict schema enforcement and ensures compatibility between producers and consumers. It allows for efficient serialization and deserialization of data by caching schemas locally.
5. Volume Mappings: The use of volume mappings allows data persistence between container restarts. By mapping data folders within the containers to local folders on the host machine, Kafka topics, ZooKeeper data, and transactions are preserved even when restarting the Docker Compose setup.
6. Network Configuration: The "reach" network driver is used to enable containers within the same network to communicate with each other using their container names. This simplifies the connectivity between containers and facilitates the inter-container communication required by the Kafka Cluster components.
7. Kafka Topics and Replication Factor: The Docker Compose setup includes an initialization step where Kafka topics are created.

Implementation

For implementation of Kafka in my project, I have used the following modules; All of them are under the infrastructure folder

- Kafka Config Data Generic Module: The module aims to centralize and simplify the management of Kafka-related configuration settings, allowing microservices to easily connect to Kafka clusters and configure producer and consumer behavior.

- Kafka Model Generic Module: This focuses on creating Kafka model classes using Avro schema types. The module leverages the Avro-maven-plugin to generate Java classes from Avro schema files. The Avro schema files define the structure and types of the data exchanged through Kafka topics. This module enables the usage of standardized and Avro-serialized messages in Kafka communication, enhancing interoperability and maintainability in a microservices architecture.

    1. Use of Avro Schemas and generated classes - Creating and registering a schema using Avro or Json will make the message exchange more reliable and prevent failures by disallowing incorrect messages. It will also create a good description and documentation for each field of schema.

Last but not least, performance wise, it has a compact binary structure and it takes less space on disk and on transit. That means we can exchange more messages in the same time interval which will increase throughput and decrease latency.

2. Avro Schema Files:
    - The Avro schema files define the data structure and types for the corresponding Kafka topics' messages.
    - Each schema file includes a namespace declaration, specifying the Java package name for the generated Avro classes.
    - The fields in the schema files are defined with specific types and logical types, such as UUID, decimal, long (for timestamps), enum, and array.
    - The logical types provide additional attributes or annotations to represent derived or specialized types, such as UUID and decimal.

- Kafka Producer Module: It enables the sending of data to Kafka brokers.

- Kafka Consumer Module: It enables the receiving the data from Kafka brokers.

The complete code can be viewed on Github at https://github.com/mayankjain0141/EatStreet

Future Work

The project can be further expanded upon by the following features

- Developing a cart service for a complete marketplace functionality.
- A central gateway authentication service for handling Users and restricting access.
- Create a dashboard for failed outbox messages, possibly with alerting using Grafana & Prometheus.
- Extend the payment service with a wallet functionality.

Technologies Stack

- Java -

Java is a popular and widely-used programming language known for its versatility, scalability, and platform independence. It offers a robust and extensive library of reusable components and frameworks, making it suitable for a wide range of applications, from desktop to web and mobile development.

- Spring Boot -

Spring Boot is a Java-based framework that simplifies the development of stand-alone, production-grade applications. It provides a convention-over-configuration approach, allowing developers to quickly set up and configure Spring applications with minimal boilerplate code. Spring Boot also integrates seamlessly with other Spring frameworks, offering a comprehensive ecosystem for building enterprise-grade applications.
libraries and tools. With Kotlin's concise syntax, null safety, and functional programming features, developers can write efficient and reliable code for their backend server applications. By using Kotlin with Ktor, developers can build high-performance, scalable, and reliable backend servers, making it an increasingly popular choice for many developers and organizations.

- Apache Kafka -

Apache Kafka is a distributed event streaming platform designed for high-throughput, fault-tolerant, and real-time data streaming. It provides a publish-subscribe messaging system that enables reliable data transfer between applications and microservices. Kafka's key features include scalability, fault-tolerance, and support for event-driven architectures, making it ideal for building data-intensive, event-based applications.

- PostgreSQL -

PostgreSQL is a powerful open-source relational database management system (RDBMS) known for its robustness, reliability, and adherence to SQL standards. It offers a wide range of advanced features, including support for complex queries, concurrency control, and extensibility. PostgreSQL is widely used in enterprise applications and provides excellent data integrity and scalability.

- Git -

Git is a popular version control system that is widely used by software developers to manage and track changes to their codebase. Git provides a robust and flexible platform for version control, allowing developers to work collaboratively and efficiently on codebase development. When combined with Gitlab, a web-based Git repository management system, developers can take advantage of additional features and tools to enhance their development workflows. Gitlab provides a comprehensive set of features, including code review, issue tracking, continuous integration, and continuous deployment, all in one platform, making it an ideal choice for software development teams. With Git and Gitlab, developers can work together seamlessly, track changes to their codebase, and ensure the code is of high quality, making it a valuable asset to any software development project.