Tools and Technologies Used

Development Environment

- Programming Language: Java 11
- Framework: Spring Boot 2.7

Databases

- Command-side Database: PostgreSQL
    - Database Management UI: pgAdmin4
- Query-side Database: MongoDB
    - Database Management UI: MongoDB Atlas

Event Streaming

- Event Streaming Platform: Apache Kafka

API Testing

- API Testing Tool: Postman

Architectural Choices

CQRS for Optimized Performance

Using the Command-Query Responsibility Segregation (CQRS) pattern was a non-negotiable requirement for this project, and for a good reason. The segregation of command and query operations allows for more granular optimization. Write operations require a strong focus on data integrity and are often transactional, while read operations demand speed and responsiveness. With CQRS, I could fine-tune these aspects individually, and the architecture could scale either the command or the query side based on demand.

PostgreSQL for Write-Heavy Operations

PostgreSQL came as a requirement, and it fit well with my architecture. Its ACID compliance and rich indexing capabilities make it well-suited for write-heavy operations in the Command Service. The built-in support for JSON data types also provides flexibility, aligning it well with modern application needs. The transactional integrity that PostgreSQL offers is essential for operations like order processing, where data consistency is crucial.

Event Sourcing through Kafka

The architecture leverages Apache Kafka for Event Sourcing, another project requisite. Kafka’s abilities to handle high throughput and provide low-latency data streaming are unparalleled. The durable storage and fault-tolerant capabilities ensure that the data is reliable, which is indispensable for maintaining the integrity of an e-commerce platform. Additionally, Kafka's native stream processing capabilities make it easier to implement real-time analytics and other reactive features.

MongoDB for Read-Heavy Scenarios

For the Query Service, MongoDB was my choice due to its document-oriented storage and flexibility in schema design, enabling quick and complex queries. Its natural support for JSON-like documents offers agility in handling semi-structured data. MongoDB also provides robust aggregation pipelines, which can be invaluable for generating real-time analytics.

Achieving Eventual Consistency

Kafka's role isn't limited to just Event Sourcing; it also serves as the backbone for ensuring eventual consistency between the PostgreSQL and MongoDB databases. Kafka’s durable nature and the asynchronous model allow the system to be more resilient, decoupled, and eventually consistent, matching the CQRS pattern beautifully.

Why Spring Boot?

I chose Spring Boot for its rapid development capabilities and its rich ecosystem. The framework dramatically simplifies building production-ready applications and offers out-of-the-box support for building microservices, working with JPA, and integrating with Kafka and MongoDB. I am quite familiar with Spring Boot, and its versatility perfectly complements my architectural choices, making it easier to implement CQRS and work seamlessly with both PostgreSQL and MongoDB.

The Importance of JPA and Database Connections

The use of JPA (Java Persistence API) for database operations in the Command Service streamlines the data interaction and transaction management with PostgreSQL. Spring Boot also offers seamless integration with MongoDB through its Spring Data MongoDB project, making it easier to perform CRUD operations in the Query Service.

Summary

To sum it up, each architectural choice and tool was either a project requirement or carefully picked to serve specific use-cases in the e-commerce domain. From the mandatory usage of CQRS, PostgreSQL, and Kafka to the selective adoption of MongoDB and Spring Boot, each element contributes to building a scalable, reliable, and maintainable system. The system is not just tailored to meet current demands but is also geared to adapt and scale for future needs.

Postgres Database Schema Design Command-Side

Entity Design and Justification

Customer

- customerId: Auto-generated unique identifier for each customer.
- name: Stores the name of the customer.
- email: Stores the email of the customer.
- phoneNumber: Stores the phone number of the customer with a maximum length of 15.

The Customer entity focuses on storing information that can identify and contact the customer. Each customer has a unique customerId, serving as a primary key and assisting in referencing orders related to them.

Order

- orderId: Auto-generated unique identifier for each order.
- orderDate: Timestamp marking the date when the order was placed will be generated automatically when the order is created.
- customer: A many-to-one relationship with the Customer entity.
- orderItems: A one-to-many relationship with OrderItem.
- totalPrice: Stores the total price of all items in the order, defined as a decimal with two decimal places and also it will be calculated based on the quantity and the products prices.

The Order entity establishes a many-to-one relationship with the Customer entity. It also contains multiple OrderItems, justifying a one-to-many relationship with the OrderItem entity. The totalPrice is calculated server-side before being persisted, thereby ensuring data integrity.

OrderItem

- itemId: Auto-generated unique identifier for each item in an order.
- order: Many-to-one relationship with Order.
- product: Many-to-one relationship with Product.
- quantity: Number of units of the product.
- price: The price at which the product was sold, preserved for historical accuracy will be taken from the product.

Product

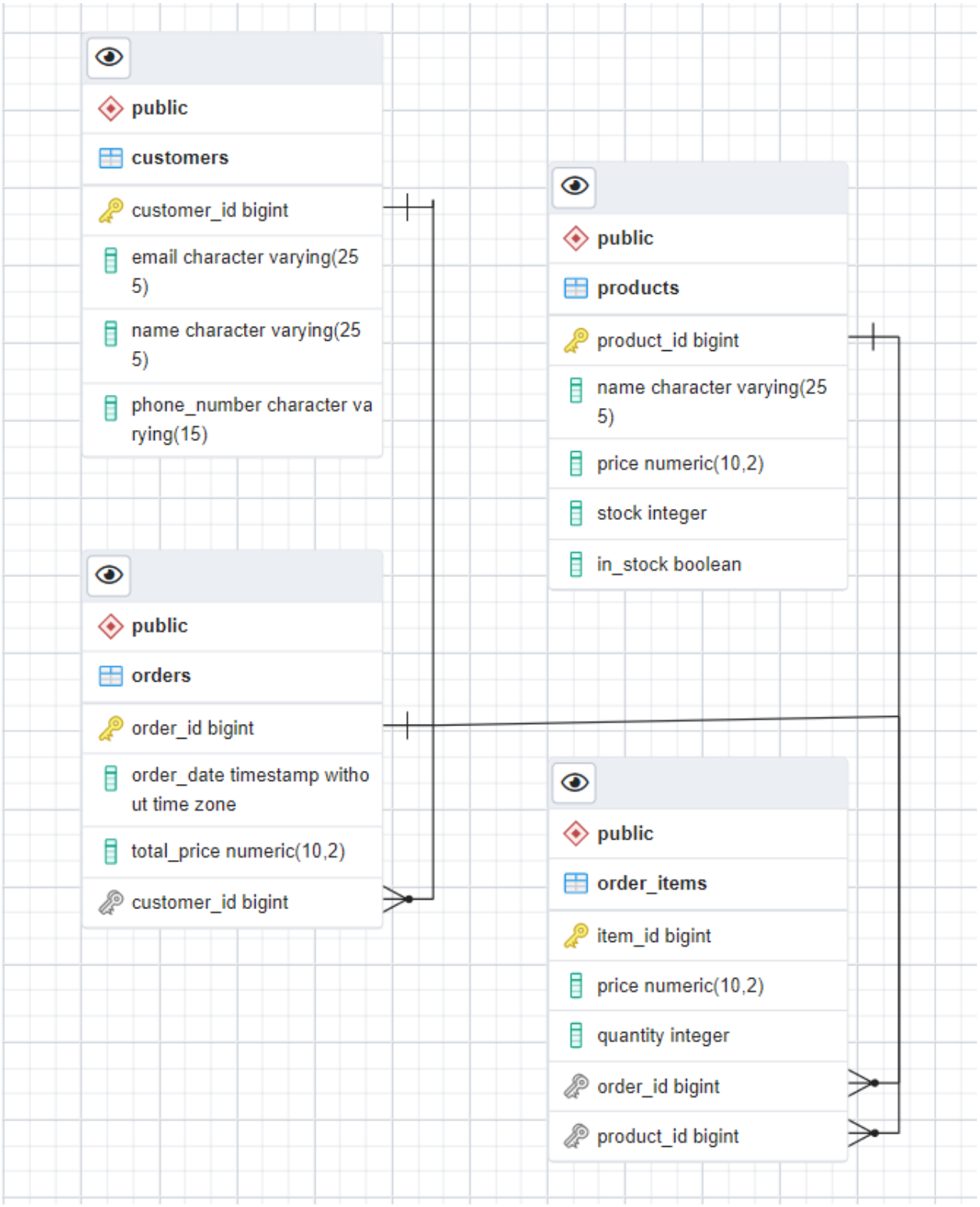
- productId: Auto-generated unique identifier for each product.
- name: Name of the product.
- price: Price of the product, stored as a decimal.
- stock: Number of items available in stock.
- stockStatus: A boolean that indicates whether the item is in stock or not will be automatically updated based on the stock.

Relationships and Normalization

- Customer to Order: One-to-Many
  A single customer can place multiple orders, while each order is associated with only one customer.
- Order to OrderItem: One-to-Many
  An order can contain multiple items, but each item belongs to only one order.
- Product to OrderItem: Many-to-One
  A single product can be part of multiple order items across different orders, but each order item is associated with only one product.

The schema design aims for Third Normal Form (3NF), ensuring that all data is factually dependent only on the primary key. This has several benefits:

- Elimination of Duplicate Data: Ensuring that each piece of data is stored only once, thereby reducing storage costs and improving data integrity.
- Data Integrity: The use of foreign keys and relationships ensures that the data is consistent and reliable.
- Logical Data Storage: Each piece of information is stored in its most logical place. For example, price is stored in OrderItem to maintain historical data even if the current price of the product changes.


public
customers
customer_id bigint
email character varying(255)
name character varying(255)
phone_number character varying(15)
public
products
product_id bigint
name character varying(255)
price numeric(10,2)
stock integer
in_stock boolean
public
orders
order_id bigint
order_date timestamp without time zone
total_price numeric(10,2)
customer_id bigint
public
order_items
item_id bigint
price numeric(10,2)
quantity integer
order_id bigint
product_id bigint

Benefits of Schema Design

- Scalability: The modular design allows each entity to be independent, offering more straightforward scalability.
- Read and Write Segregation: Adhering to the CQRS architecture, the schema is optimized for write operations, thereby allowing performance fine-tuning specific to this use case.

This database schema design fulfills the requirements outlined in the assessment. It ensures data integrity, scalability, and operational efficiency by making judicious use of PostgreSQL capabilities and features.

Event Sourcing Architecture: Ensuring Data Integrity and Scalability (Event Producing)

Introduction to Event Sourcing

Event Sourcing is an architectural pattern that emphasizes storing the state of a system as a sequence of state-changing events. By logging events instead of only current state data, you capture more nuanced historical information. This approach provides strong audit trailing and enables more straightforward debugging and understanding of system behaviors. It is crucial for ensuring data integrity and system scalability.

The Role of Kafka in Event Sourcing

Apache Kafka serves as the backbone for event sourcing in this application. Kafka is a distributed streaming platform designed for building real-time data pipelines and streaming applications. It provides native capabilities to store, process, and disseminate data events. Kafka is known for its fault tolerance, scalability, and ability to handle high throughput with low latency.

Utilization of KafkaTemplate

The Spring Kafka library integrates with Spring Boot applications to provide seamless Kafka capabilities. One of the key classes is KafkaTemplate, which is used in the EventProducer class. KafkaTemplate simplifies sending messages to Kafka topics. It abstracts the complexities involved with Kafka producers, making it more straightforward to send domain events to specified Kafka topics.

- KafkaTemplate<String, Object>: It's a templated Kafka producer where the key is a string, and the value can be an arbitrary object. It serializes the object before sending it to Kafka.
- Sending Events: Events such as ProductCreated, ProductUpdated, ProductSold, etc., are produced and sent to Kafka topics using KafkaTemplate.send() method. Callbacks are also registered to log the success or failure of the event sending operation.

Different Topics and Events

In the application, each domain, like Product, Customer, and Order, has its own set of events and corresponding Kafka topics. This segregation aids in the isolation of events, ensuring that services can listen to only those events that are relevant to them.

- ProductCreated, ProductUpdated, ProductSold, ProductDeleted: These topics are tied to the lifecycle events of products in the application.
- CustomerCreated, CustomerUpdated, CustomerDeleted: Similarly, these are topics for customer-related events.
- OrderCreated, OrderDeleted: Topics for order-related events.

Each of these topics is created using Kafka's command-line tools, after initializing the Zookeeper and Kafka servers.

Interactions with the Command Service

The ProductService class in the command-side microservice triggers various events depending on the operation performed—be it creating, updating, or deleting a product. These events are then produced to Kafka by invoking the relevant methods on the ProductProducer class. The producer uses the EventProducer to encapsulate the event data and send it to the appropriate Kafka topic.

Consumption in Query Microservice with MongoDB

Although this discussion focuses on event production and the command-side, it's worth noting that these produced events will eventually be consumed by a query-side microservice. This query-side service will have consumers listening to these Kafka topics to update a MongoDB database, which serves as the read-optimized data store.

Benefits

1. Data Integrity: Event Sourcing ensures that all changes to application state are stored as a sequence of events. Not just can we query these events, but we can also use them to reconstruct past states.
2. Scalability: Kafka's ability to handle high throughput ensures that the system can grow to accommodate an increasing number of events.
3. Flexibility: Having different topics for different types of events makes the system highly modular and easier to maintain and extend.
4. Auditability and Debugging: The immutable log of domain events makes it easy to understand the system's state changes, thereby simplifying debugging and providing a robust audit trail.

By implementing Event Sourcing with Kafka, the architecture achieves data integrity, scalability, and a high level of flexibility, paving the way for robust, enterprise-scale applications.

MongoDB Database Schema Design in Query Microservice

The database schema is designed to match the domain's needs and ensure that it can answer the questions posed by the system's read operations effectively. Below is an analysis of the MongoDB schema based on the code snippets you provided.

Collections and Documents

Customer Collection

- Collection Name: customers
- Fields:
    - customer_id: Unique ID of the customer.
    - name: Name of the customer.
    - email: Email address of the customer.
    - phoneNumber: Phone number of the customer.

Order Collection

- Collection Name: orders
- Fields:
    - order_id: Unique ID of the order.
    - orderDate: Date when the order was made.
    - customer: A reference to the Customer collection.
    - orderItems: List of items in the order.
    - totalPrice: Total price of the order.

Product Collection

- Collection Name: products
- Fields:
    - product_id: Unique ID for the product.
    - name: Name of the product.
    - price: Price of the product.
    - stock: Stock level of the product.
    - stockStatus: A boolean indicating if the product is in stock.

ProductSold Collection

- Collection Name: product_sold
- Fields:
    - id: Unique ID (usually generated by MongoDB).
    - productId: ID of the product that was sold.
    - productName: Name of the product sold.
    - productPrice: Price at which the product was sold.
    - quantitySold: Quantity of the product sold.
    - remainingStock: Remaining stock after the sale.
    - stockStatus: A boolean indicating stock status after the sale.
    - orderId: ID of the order in which the product was sold.
    - customerId: ID of the customer who bought the product.

- productTotalPrice: Total price of the product sold (usually productPrice * quantitySold).

Key Points and Considerations

Data Relationships and Normalization:

- The Customer is related to the Order through a MongoDB DBRef, allowing for easy traversal and aggregation.
- The OrderItem is embedded within the Order, enabling quick access to item-level details without additional queries.

Data Integrity and Redundancy:

- ProductSold contains some redundant information (e.g., productName, productPrice) to avoid frequent joins and to maintain a historical record of prices and names at the time of sale.

Scalability:

- MongoDB allows horizontal scaling through sharding, which could be particularly useful if any of the collections grow significantly in size.

Consumption in Query Microservice:

- The product_sold collection would be particularly useful for analytics and can be populated by consuming events like ProductSold from the Kafka topics.

Data Access Patterns:

- Given this schema design, various queries can be optimized for reading data, such as fetching all orders for a customer, all sales of a product, etc.

Event Sourcing:

- This schema is geared for optimal query performance, as it should be, being part of a CQRS (Command Query Responsibility Segregation) and event-sourced
- architecture. Changes in these collections are typically triggered by consuming events from Kafka topics.

Event Consumption and Ensuring Data Consistency (Consuming Event)

Introduction to Event Consumption

Event consumption is the counterpart to event production in an Event Sourcing architecture. While event production captures and publishes state changes, event consumption is responsible for listening to these events and updating the system's state accordingly. In this e-commerce platform, Kafka serves not just as the event publisher but also as the event subscriber for the read-side or the Query microservice.

Consumption in Query Microservice

In our architecture, the query microservice is responsible for the read-side of the application, and it updates its MongoDB database based on the events it consumes from Kafka topics. This ensures that the read-side database is eventually consistent with the write-side database, conforming to the CQRS pattern.

ConsumerEvent Class and KafkaListener

- ConsumerEvent Class: Similar to the ObjectEvent class in the command service, the ConsumerEvent class in the query service defines the structure of an event consumed by Kafka, consisting of eventType and payload.
- KafkaListener Annotation: This annotation marks a method to be the target of a Kafka message listener within a class annotated with @Service. It listens to specified topics and executes the method whenever an event is published to those topics.

Event Consumers: ProductConsumer, and others for Customer and Order

- ProductConsumer: It has multiple methods marked with @KafkaListener, listening to various product-related topics like PRODUCT_CREATED, PRODUCT_UPDATED, etc.
- Customer and Order Consumers: Similar consumers are implemented for customer and order events.
- Deserialization: The raw JSON message consumed from Kafka is deserialized into a ConsumerEvent<T> object, and its payload is further used to update the MongoDB database.
- Repository Operations: Post deserialization, MongoDB's ProductRepository, ProductSoldRepository etc., execute the appropriate CRUD operation, thereby ensuring that the read database is in sync with the write database.

application.properties for Kafka Configuration

This file contains essential Kafka consumer configurations including bootstrap-servers, group-id, and key and value deserializers. These settings are auto-wired by Spring Boot, providing seamless integration with Kafka.

Benefits of This Approach

1. Data Consistency: By consuming every single event published to Kafka, the query microservice ensures that its MongoDB database is eventually consistent with the write-side PostgreSQL database.
2. Scalability: Kafka’s distributed nature allows the query service to scale horizontally, adding more consumers as the system grows.
3. Fault-Tolerance: Kafka ensures that events are not lost, providing a robust mechanism for maintaining data consistency across multiple services.
4. Decoupling: The producers and consumers are decoupled, allowing them to evolve, develop, deploy, and scale independently.

5. Real-Time Updates: Kafka's low latency enables real-time updates in the read-optimized database, thereby providing real-time analytics capabilities.
6.

By judiciously using Kafka for both event production and consumption, this architecture ensures a high level of data integrity and system scalability. It fully leverages the CQRS and Event Sourcing patterns to build a robust, scalable, and maintainable e-commerce platform.

Scalability and Performance: My Architectural Approach

Microservices for Segregated Operations

One of the first decisions I made was to divide the system into two distinct microservices— one for writing operations (Command Service) and another for reading operations (Query Service). This separation aligns with the Command Query Responsibility Segregation (CQRS) pattern, a key architectural decision that has proven invaluable in several ways:

- Optimized Operations: By decoupling read and write operations, I was able to fine-tune each service to be highly specialized. The write operations are transactional and require robust validation, while the read operations are optimized for speed and performance.
- Independent Scalability: With two separate services, scaling became far more straightforward. If the platform experiences a surge in read operations (like during a flash sale), I can easily scale out the Query Service without affecting the Command Service.

Event-Driven Architecture with Kafka

Another cornerstone of the architecture is Kafka, employed to implement Event Sourcing. Kafka is not just a message broker but a distributed event-streaming platform. By design, it's made to handle high throughput with low latency, which aligns perfectly with my goal for a scalable system.

- Parallel Processing: Kafka topics are divided into partitions, allowing multiple consumers to read a topic in parallel. This feature can be a lifesaver during high-traffic scenarios, and I leveraged it in the Query Service.
- Event Sourcing for Integrity and Rebuilding State: Having an immutable log of all changes gives us the power to rebuild the application state at any time, enhancing both data integrity and system resilience.

Data Storage and Management

For data storage, I employed PostgreSQL for the Command Service and MongoDB for the Query Service. I've normalized the data in PostgreSQL to remove any redundancies, ensuring data integrity and optimized storage. MongoDB's document-oriented nature makes it perfect for fast, flexible queries, serving the read-heavy Query Service exceptionally well.

Final Thoughts

The decision to use Spring Boot for building the microservices simplified many complexities, allowing me to focus on business logic. Combining this with Kafka and the CQRS pattern has set the stage for a highly scalable architecture.
Overall, the architecture isn't just built to solve today's problems but is ready for tomorrow's challenges. I've aimed for a balance of scalability and data integrity by making carefully thought-out architectural choices. And as the platform grows, I'm confident that this architecture is robust and flexible enough to grow with it.