Saeed Aghaee

66 Followers About Follow

CQRS Pattern with Kafka Streams — Part 1

 Saeed Aghaee Feb 17 · 5 min read

CQRS stands for **Command Query Responsibility Segregation**. It promotes the idea of separating the "responsibilities" of "commands" and "queries". In this article, I will try to answer the following questions:

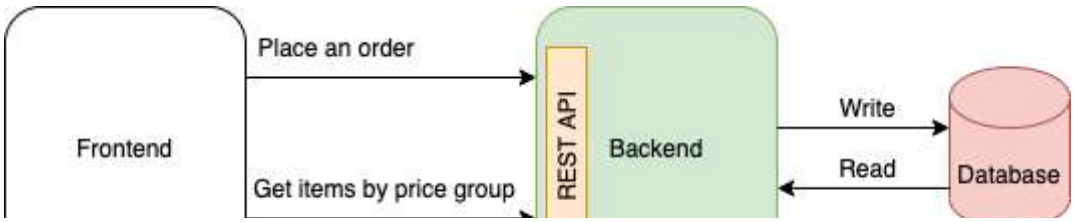
- What is CQRS?
- Why is Kafka Streams a natural choice for implementing CQRS?
- How to implement CQRS pattern using Kafka Streams?

Case study: Online Ordering System

Let's start with a classic example: an online ordering system for retail. It has two main use cases:

- Users can places orders for items.
- Users are able to view their ordered items in real time, grouped based on their price tag. We categorise items into three groups based on their price: *cheap* (less than £5), *affordable* (between £5 and £50), and *expensive* (more than £50).

A very high level design might look like this:



High level design for our simple CRUD-based ordering system

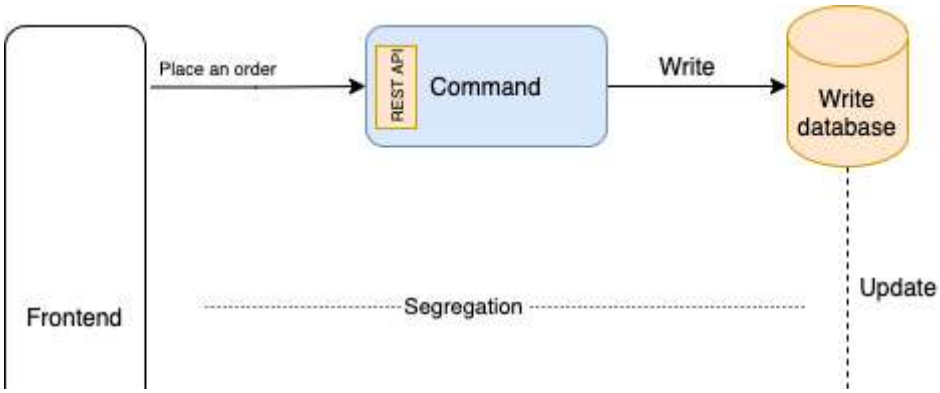
This is a simple CRUD application that has a REST API (backend) and a single-page application (frontend). The frontend uses a REST API to place an order. The backend then triggers some business logic and writes the new orders into the database. Similarly, to get a groups of items, the frontend calls an API that results in a set of read operations in the backend.

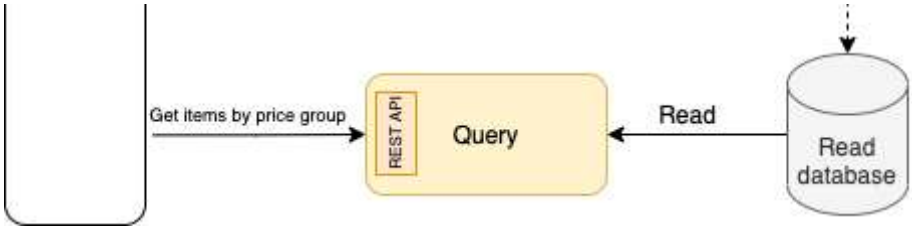
Applying CQRS pattern

In CQRS terms, read and write operations are referred to as, respectively, *query* and *command*. In general, CRUD systems can be all logically divided into command and query sub-systems. However, CQRS is best to apply on the CRUD applications that **(a)** require to use a different model to update information than the model they use to read information, **(b)** require time-consuming queries for reads, and **(c)** have considerably higher reads than writes. Otherwise, increased complexity, as a consequence, will follow.

In our case, we need different models for read and write and can probably expect a higher number of reads than writes (every time the page is loaded a read request is sent to the backend). For the sake of this article, let's apply and benefit from the CQRS pattern and embrace its complexity.

At its heart, CQRS is all about segregating command and query operations. To do that, we split up the backend into two *microservices,* command and query, so that they can be independently scaled and maintained. Moreover, in order to support different read and write models and ensure loose coupling, we decentralise the database and keep each microservice's persistent data private to that service and accessible only via its APIs. As the number of users grow and the amount of data increases, both microservices will be able to satisfy different storage and schema requirements.



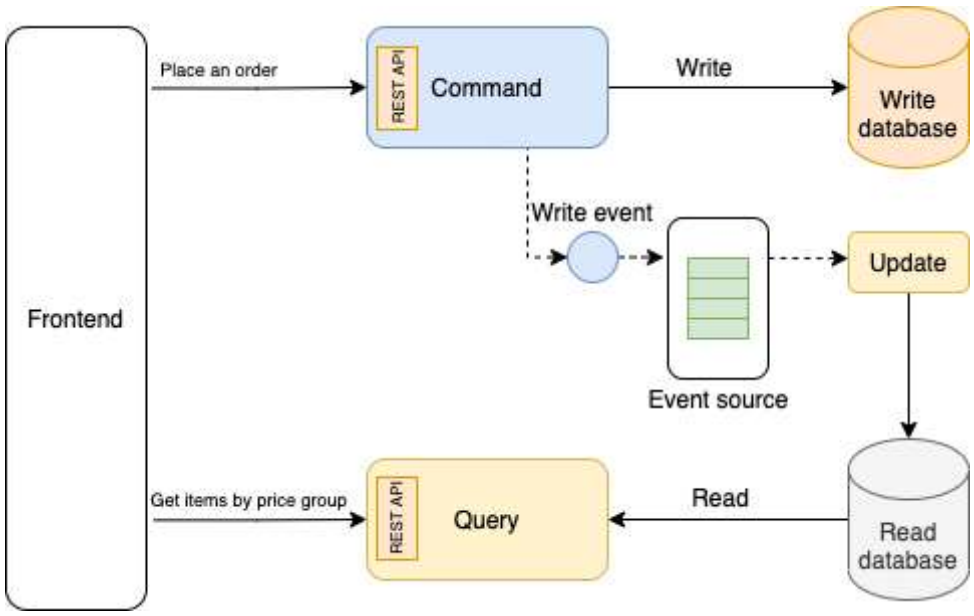


Applying Event Sourcing

So far, we have successfully segregated the command and query operations. However, there is a missing piece to the puzzle and that's how we are going to reliably *update* the read database as information is being written into the write database. For example, as soon as a user places a new order for an item, in addition to undertaking the write responsibilities, the backend needs to simultaneously identify the item as one of *cheap*, *affordable*, or *expensive* and store it in the read database.

One option is to call the query API from the command microservice and synchronously update the read database. This solution, however, introduces tight coupling and compromises reliability. If the query microservice is down, any incoming data will be lost.

A much better option is to apply *Event Sourcing*. Event sourcing involves two steps: **(1)** modeling the state changes made by an application as a set of immutable events, and **(2)** modeling the state changes as responses to the events. In simple words, event sourcing decouples the application change from the record of that change and uses the latter as the source of truth. Event sourcing will allow us to asynchronously (and reliably) update the read database to reflect changes to the write database.



CQRS with Event Sourcing

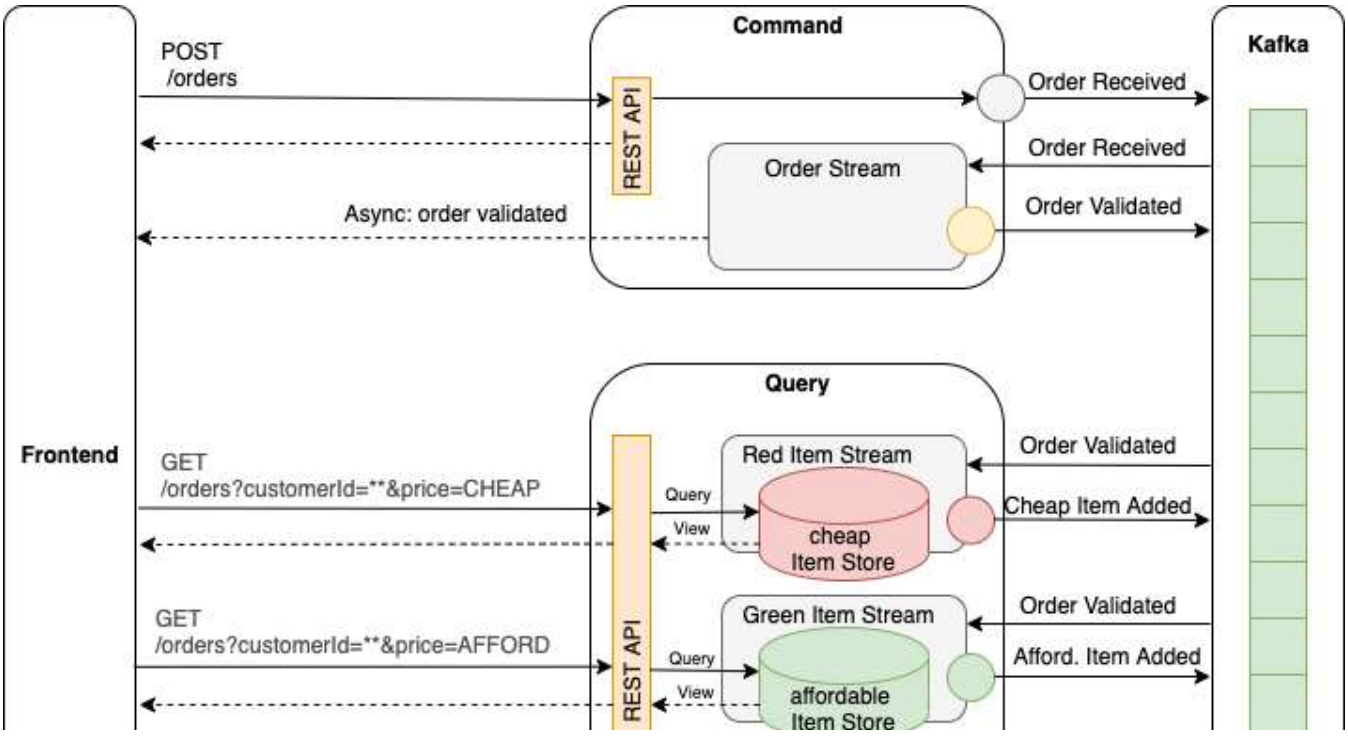
CQRS using Apache Kafka Streams

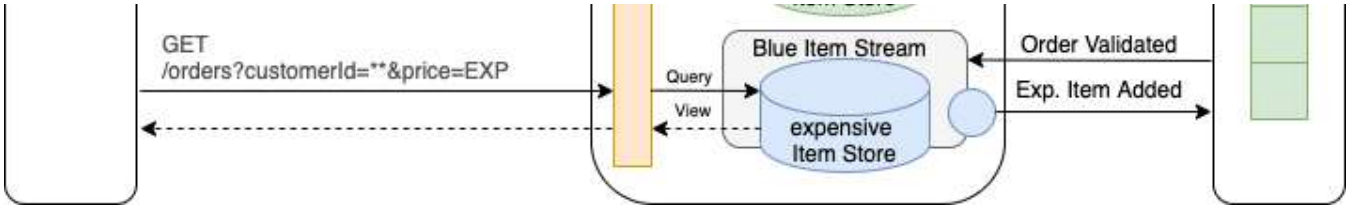
Kafka is already one of the best options for Event Sourcing. You can write events to Kafka (write model), read them back, and then push them into a database or a different topic (read model). In doing so, Kafka maps the read model onto the write model asynchronously, decoupling the two. Kafka is, therefore, a natural choice for implementing CQRS.

In an event sourcing architecture, events are the first class citizens. This is different from traditional architectures, in which database is the primary source of truth. The stream-table duality concept, coupled with fault tolerance and high availability of Kafka, enable us to replace database with events as the primary source of data.

What distinguishes an event-sourced view from a typical database or cache is that, while it can represent data in any form required, its data is sourced directly from the events log and can be regenerated at any time. Writes go into Kafka on the command side and generate a stream of events. We can transform the stream in a way that suits our use case, typically using Kafka Streams, then materialise it as a precomputed query.

A materialised view in Kafka is a table that contains the results of some predefined query. The view gets updated every time any of the underlying tables change. But unlike with materialized views in a relational database, the underlying events are decoupled from the view. This means **(a)** they can be scaled independently, and **(b)** the writing process doesn't have to wait for the view to be computed before it returns.





CQRS pattern using Kafka: Events are the source of truth. Materialized views provide a pre-computed in-memory cache for the query microservice.

Implementation

The complete implementation can be found on github.

In the next part, I'll explain the implementation process in detail.

Event Driven Architecture Kafka Kafka Streams Architecture Cqrs