



Published in Design Microservices Architecture with Patterns &amp; Principles



Mehmet Ozkaya

Follow

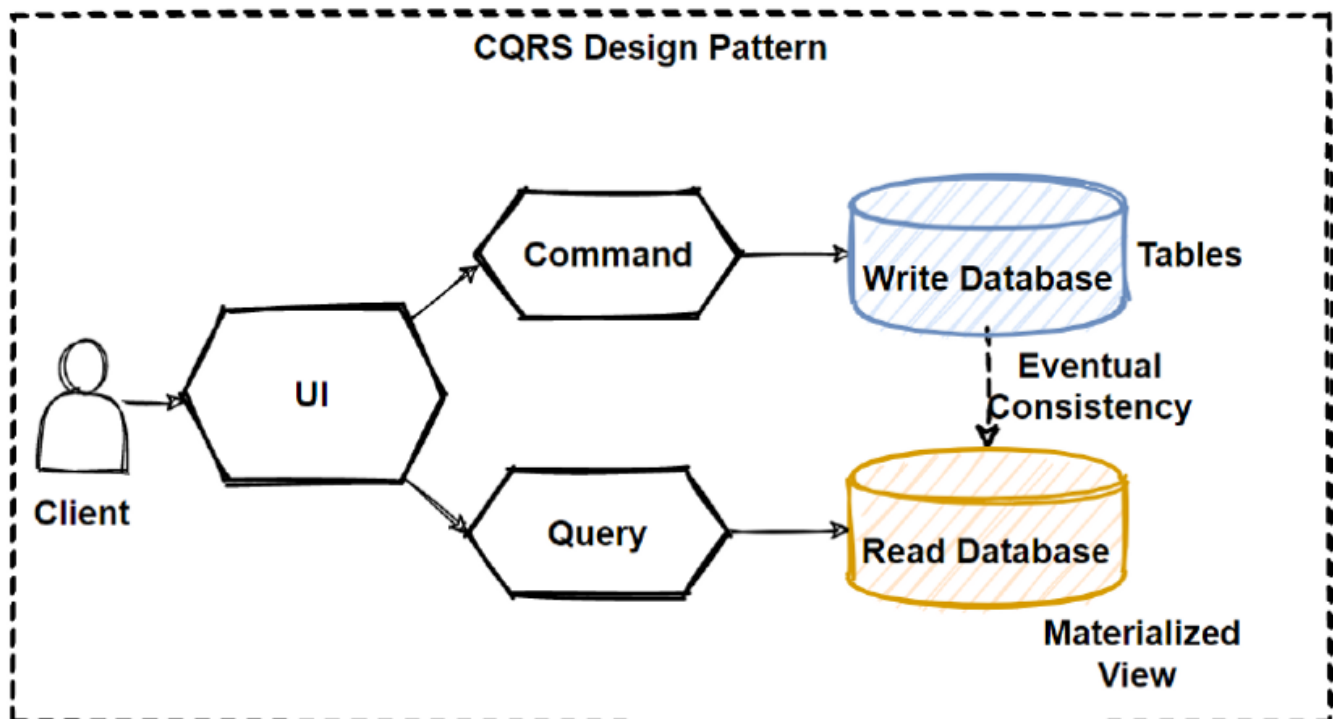
Sep 8, 2021 · 6 min read · [Listen](#)

Save



# CQRS Design Pattern in Microservices Architectures

In this article, we are going to talk about **Design Patterns** of Microservices architecture which is **The CQRS Design Pattern**. As you know that we learned **practices and patterns** and add them into our **design toolbox**. And we will use these **pattern and practices** when **designing e-commerce microservice architecture**.



By the end of the article, you will learn where and when to apply CQRS Design Pattern into Microservices Architecture with designing e-commerce application system.

## Step by Step Design Architectures w/ Course

### Design Microservices Architecture with Patterns & Principles

Handle millions of request with designing high scalable and high available systems on microservices architecture.

0.0 ★★★★★ (0 ratings) 74 students

Created by [Mehmet Özkaya](#)

#### I have just published a new course — Design Microservices Architecture with Patterns & Principles.

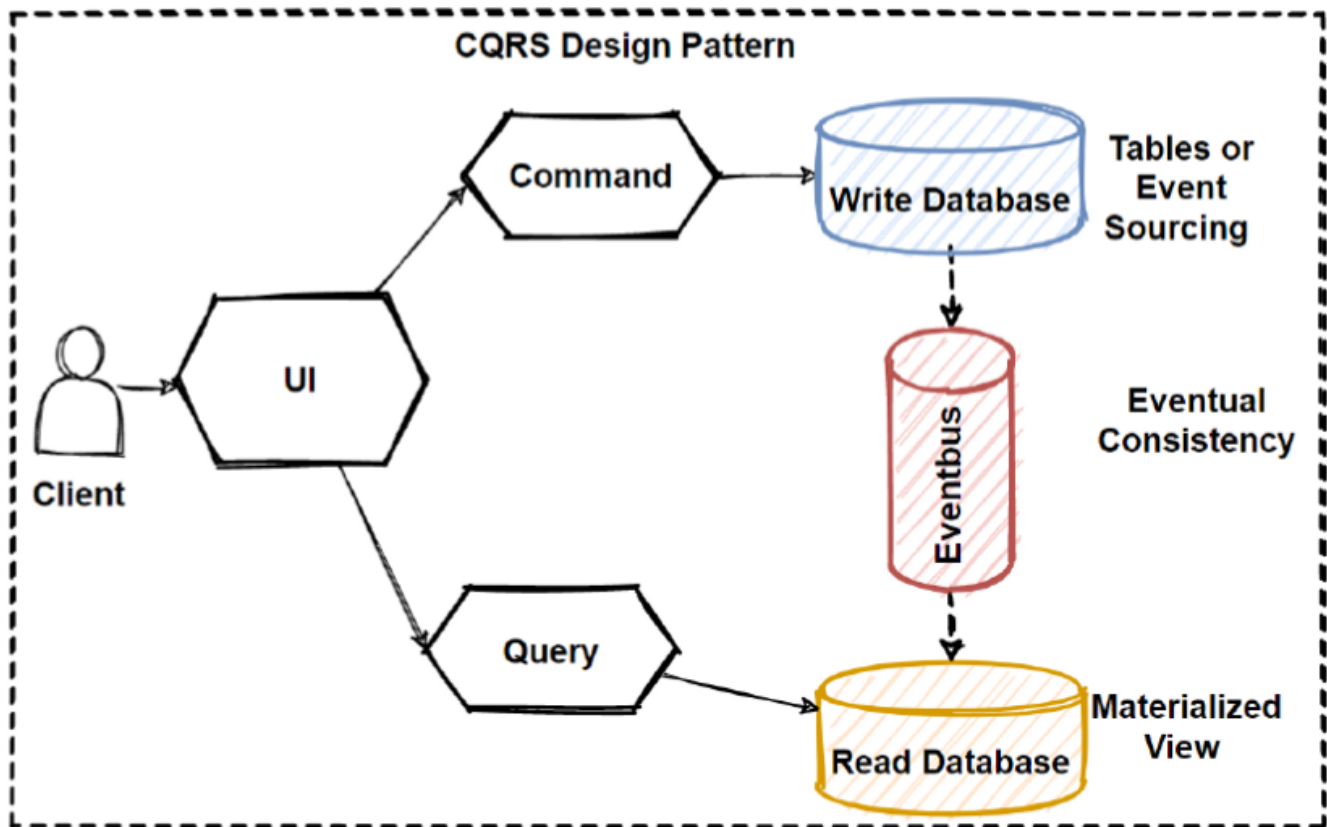
In this course, we're going to learn **how to Design Microservices Architecture** with using **Design Patterns, Principles** and the **Best Practices**. We will start with designing **Monolithic to Event-Driven Microservices** step by step and together using the right architecture design patterns and techniques.

#### **CQRS Design Pattern**

CQRS is one of the important pattern when querying between microservices. We can use CQRS design pattern in order to avoid complex queries to get rid of inefficient joins. CQRS stands for **Command and Query Responsibility Segregation**. Basically this pattern separates read and update operations for a database.

Normally, in monolithic applications, most of time we have 1 database and this database should respond both query and update operations. That means a database is both working for **complex join queries**, and also **perform CRUD** operations. But if the application goes more complex this query and crud operations will be also is going to be un-manageable situation.

In example of reading database, if your application required some query that needs to **join more than 10 table**, this will lock the database due to latency of **query computation**. Also if we give example of writing database, when performing crud operations we would need to make complex validations and process long business logics, so this will **cause to lock database** operations.



So reading and writing database has different approaches that we can define different strategy to handle that operation. In order to that **CQRS** offers to use “**separation of concerns**” principles and separate reading database and the writing database with 2 database. By this way we can even use different database for **reading** and **writing** database types like using no-sql for reading and using relational database for crud operations.

Another consideration is we should understand our application use case behaviors, if our application is mostly reading use cases and not writing so much, we can say our application is **read-incentive application**. So we should design our architecture as per our reading requirements with focusing reading databases.

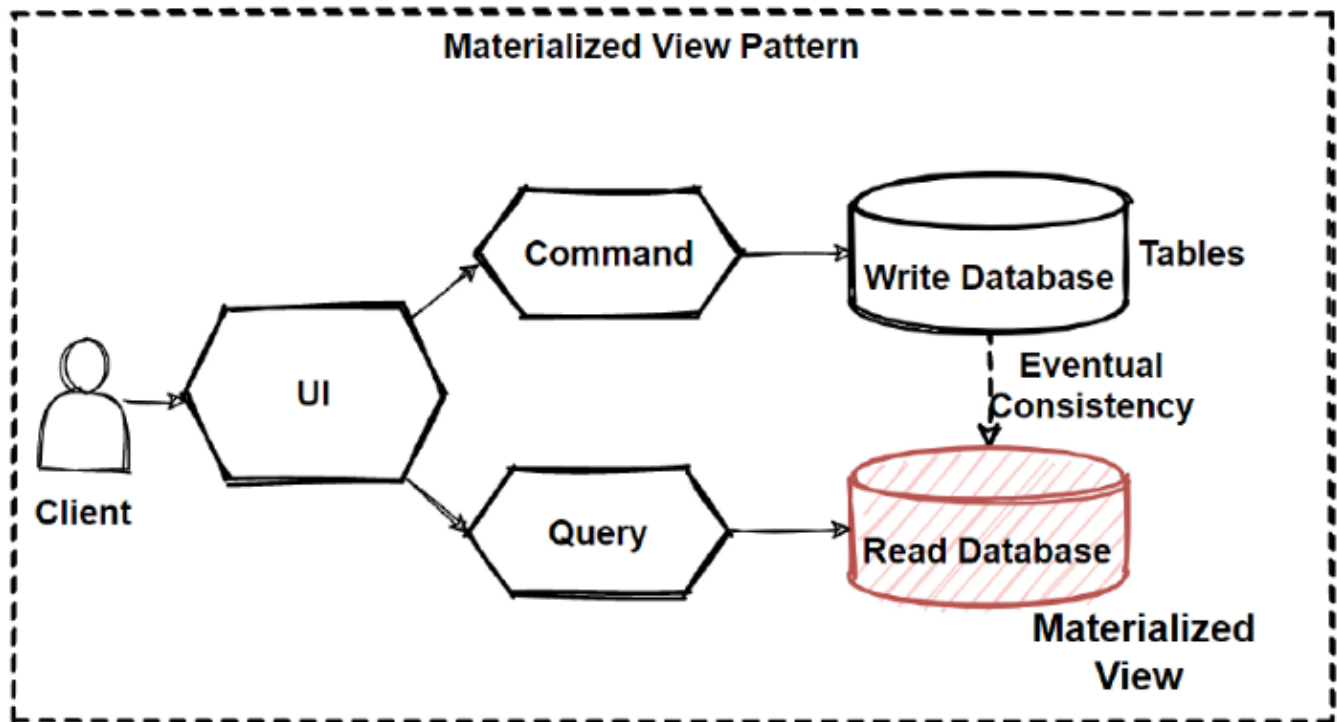
So we can say that **CQRS** separates reads and writes into **different databases**, Commands performs update data, Queries performs read data.

**Commands** should be actions with task-based operations like “add item into shopping cart” or “checkout order”. So commands can be handle with message broker systems that provide to process commands in async way.

**Queries** is never modify the database. Queries always return the JSON data with DTO objects. By this way, we can isolate the Commands and Queries.

In order to isolate **Commands** and **Queries**, its best practice to separate read and write database with 2 database physically. By this way, if our application is read-intensive that means reading more than writing, we can define custom data schema to optimized for queries.

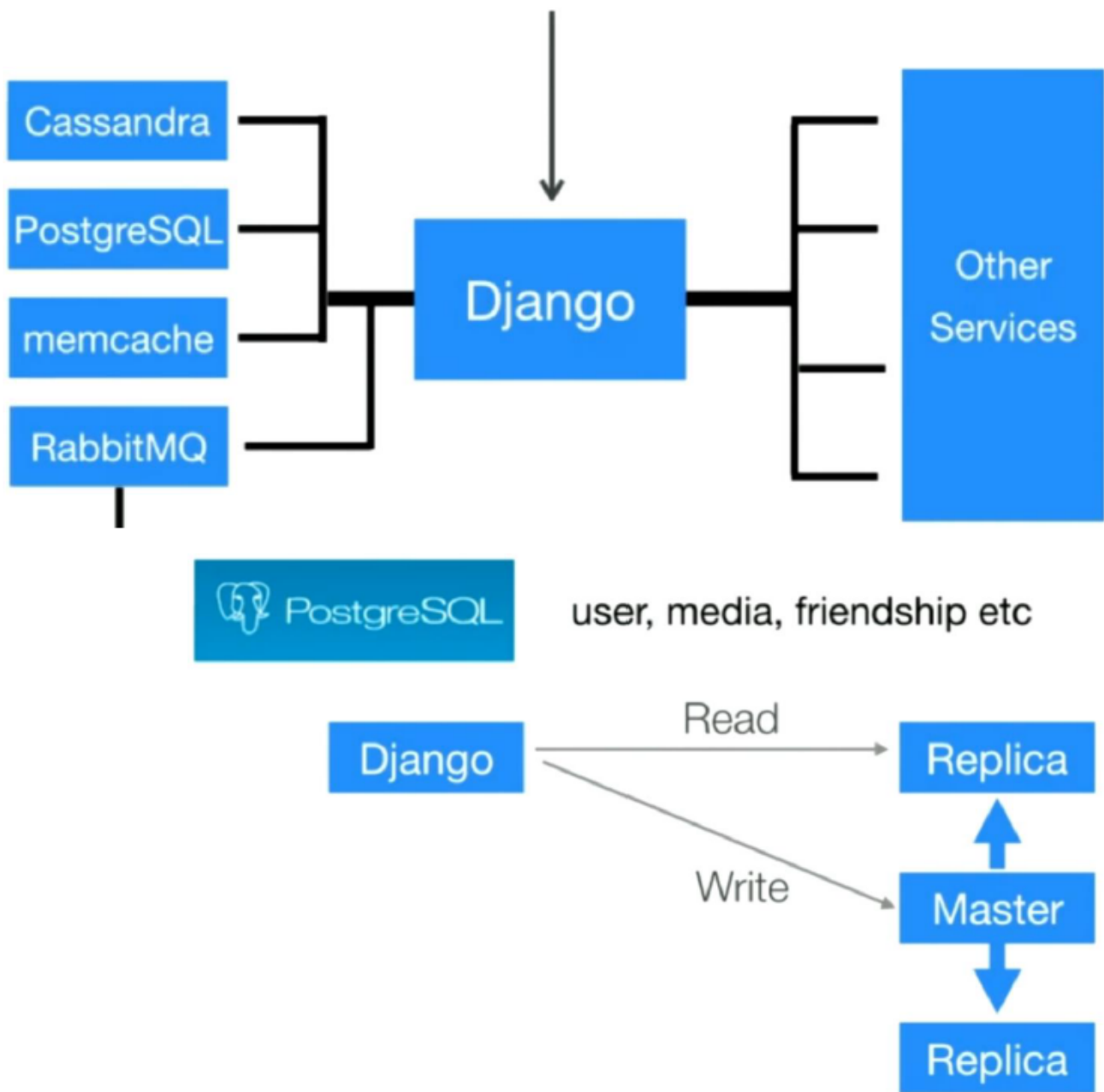
**Materialized view pattern** is good example to implement reading databases. Because by this way we can avoid complex joins and mappings with pre defined fine-grained data for query operations.



By this **isolation**, we can even use different database for reading and writing database types like using **no-sql document database** for reading and using relational database for crud operations.

## Instagram Database Architecture

This is so popular on microservices architecture also let me give an example of **Instagram architecture**. Instagram basically uses two database systems, one is relational database **PostgreSQL** and the other is no-sql database — **Cassandra**.



So that means Instagram uses **no-sql Cassandra database** for user stories that is **read-incentive data**. And using relational PostgreSQL database for User Information bio update. This is one of the example of CQRS approaches.

### How to Sync Databases with CQRS ?

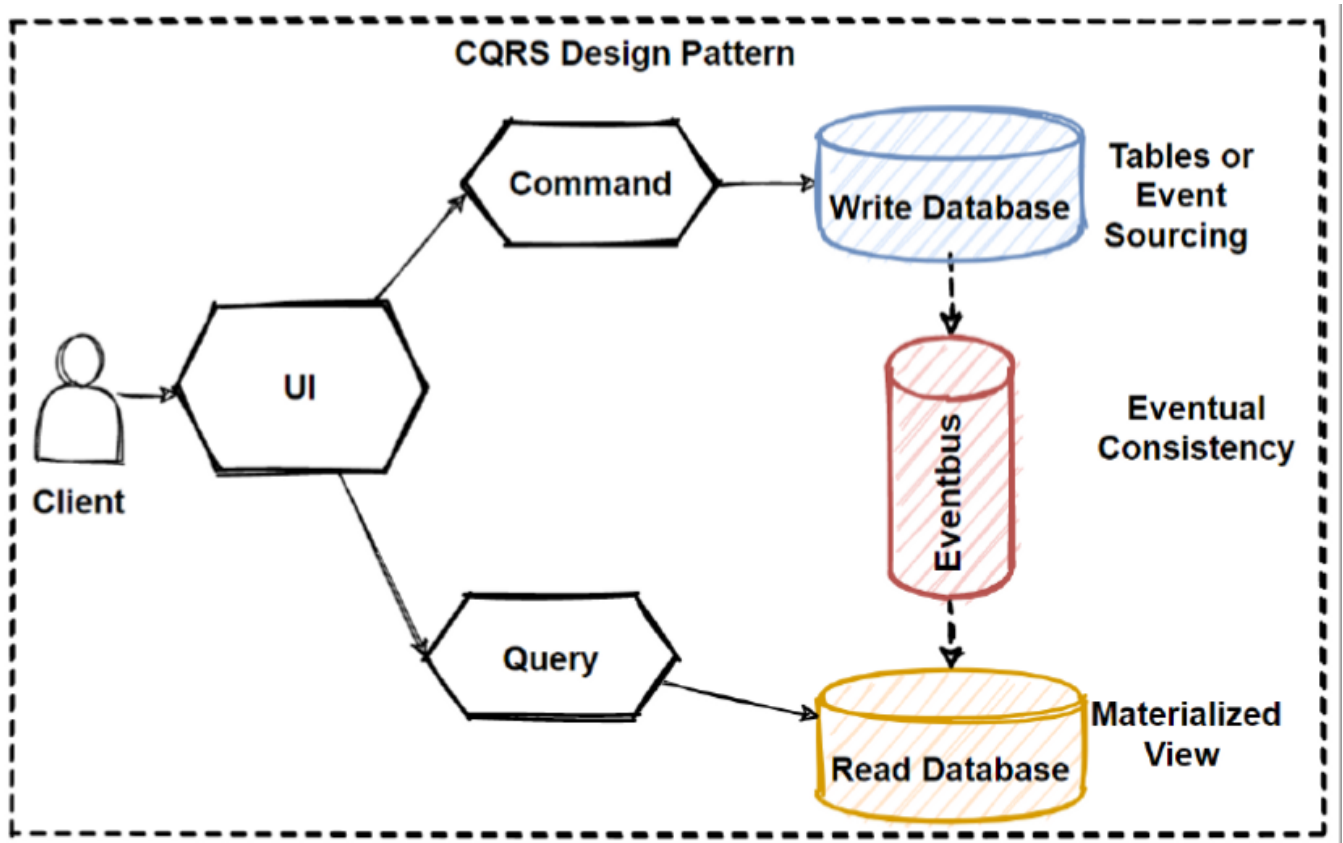
But when we separate read and write databases in 2 different database, the main consideration is sync these two database in a proper way.

So we should **sync these 2 databases** and keep sync always.

This can be solve by using **Event-Driven Architecture**. According to **Event Driven Architecture**, when something update in write database, it will publish an update

event with using **message broker** systems and this will consume by the read database and **sync data** according to latest changes.

But this solution creates a **consistency** issue, because since we have implemented **async communication** with message brokers, the data would not be reflected immediately.



This will operate the principle of “**eventual consistency**”. The read database eventually synchronizes with the **write database**, and it can take some time to update the read database in the **async** process. We discuss eventual consistency in the next sections.

So if we come back to our read and write databases in **CQRS pattern**, When starting your design, you can take the read database from replicas of the write database. By this way we can use different **read-only replicas** with applying the **Materialized view pattern** can significantly increase query performance.

Also when we **separated read and write databases**, it means we can scale them independently. That means if our application is read-intensive, I mean if it has much more query than write, then we can scale only reading databases very fast.

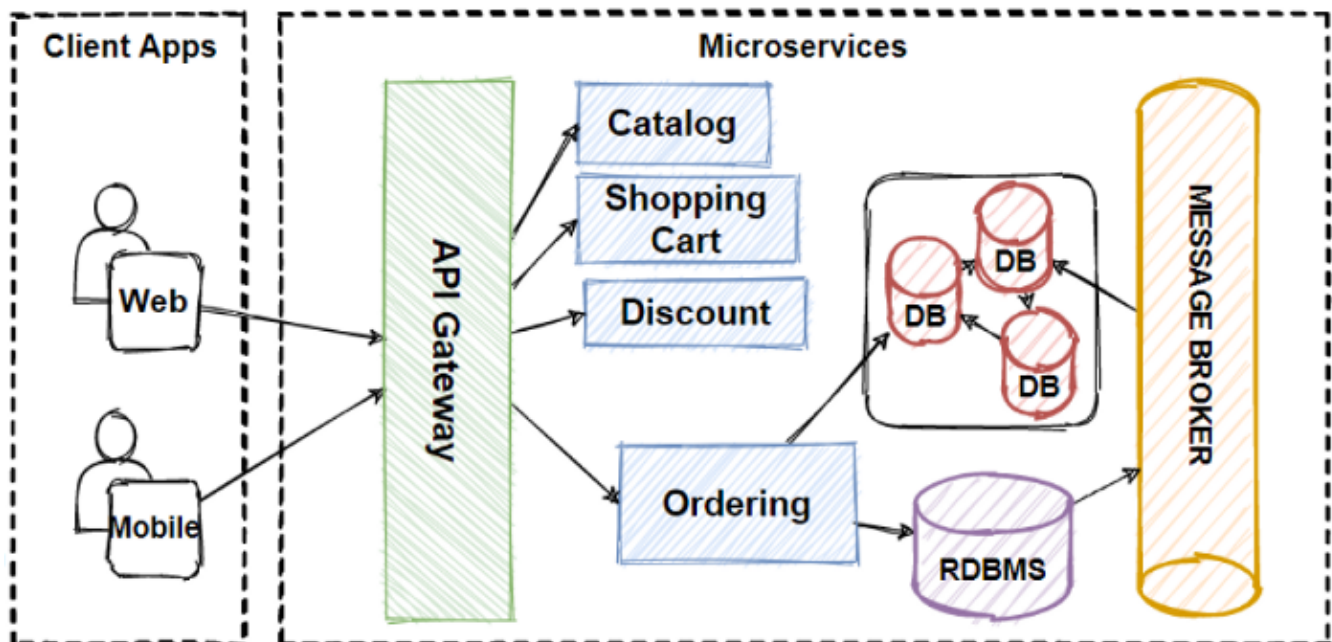


CQRS comes with **separating commands and query databases**. So this required to sync both 2 databases with offering event-driven architectures. And with **Event-driven architectures** there are some new patterns and practices should be consider when applying CQRS.

Event Sourcing pattern is the first pattern we should consider to use with CQRS. Mostly CQRS is using with “**Event Sourcing pattern**” in Event-Driven Architectures. So after we have learned the CQRS we should learn “**Event Sourcing pattern**”, because CQRS and “**Event Sourcing pattern**” is the best practice to use both of them.

## Design the Architecture — CQRS, Event Sourcing, Eventual Consistency, Materialized View

We are going to Design our e-commerce Architecture with applying CQRS Pattern.



Now We can Design our Ordering microservices databases

I am going to **split 2 databases** for **Ordering** microservices

1 for the **write** database for **relational** concerns

2 for the **read** database for **querying** concerns

So when user create or update an order, I am going to use relational write database, and when user **query order** or order history, I am going to use no-sql read database. and make consistent them when syncing 2 databases with using message broker system with applying **publish/subscribe pattern**.

Now we can consider **tech stack** of these databases,

I am going to use **SQL Server** for relational writing database, and using **Cassandra** for no-sql read database. Of course we will use **Kafka** for syncing these 2 database with pub/sub **Kafka** topic exchanges.

So we should **evolve our architecture** with applying **other Microservices Data Patterns** in order to **accommodate business adaptations** faster time-to-market and handle larger requests.

## Step by Step Design Architectures w/ Course

### Design Microservices Architecture with Patterns & Principles

Handle millions of request with designing high scalable and high available systems on microservices architecture.

0.0 ★★★★★ (0 ratings) 74 students

Created by [Mehmet Özkaya](#)

I have just published a new course — Design Microservices Architecture with Patterns & Principles.

In this course, we're going to learn **how to Design Microservices Architecture** with using **Design Patterns, Principles** and the **Best Practices**. We will start with designing **Monolithic to Event-Driven Microservices** step by step and together using the right architecture design patterns and techniques.


---

**Get an email whenever Mehmet Ozkaya publishes.**

Your email

---



Subscribe

By signing up, you will create a Medium account if you don't already have one. Review our [Privacy Policy](#) for more information about our privacy practices.



About

Help

Terms

Privacy

Get the Medium app

Download on the  
App Store

GET IT ON  
Google Play