CQRS stands for **Command Query Responsibility Segregation**. It is an architectural pattern that separates the handling of read (query) and write (command) operations into different models. This approach can help optimize performance, scalability, and maintainability, especially in complex systems.

**Key Concepts:**

1. **Commands**: These represent the "write" operations in CQRS. Commands change the state of the system (e.g., creating, updating, or deleting data). Commands are typically handled by a command handler.
2. **Queries**: These represent the "read" operations. Queries fetch data without changing the state. Queries are typically handled by a query handler.
3. **Separation of Models**: In CQRS, the models used for reading data (queries) are separated from the models used for writing data (commands). This allows each model to be optimized for its specific use case, which can lead to more efficient database access patterns.
4. **Eventual Consistency**: When using CQRS, there can be eventual consistency between the write and read models. This means the data in the read model may not immediately reflect changes made in the write model, but it will eventually become consistent.

**Benefits of CQRS:**

* **Optimized for Performance**: By separating commands and queries, each model can be optimized independently. For example, the read model can be denormalized and indexed for fast querying, while the write model can be normalized for consistency.
* **Scalability**: CQRS can help scale the read and write sides independently, allowing the system to better handle load.
* **Maintainability**: It simplifies the codebase by enforcing clear separation of concerns between reading and writing, which makes the system easier to maintain and evolve.
* **Flexibility**: It allows different technologies to be used for the command and query sides, for example, a relational database for writes and a NoSQL database for reads.

**Use Cases:**

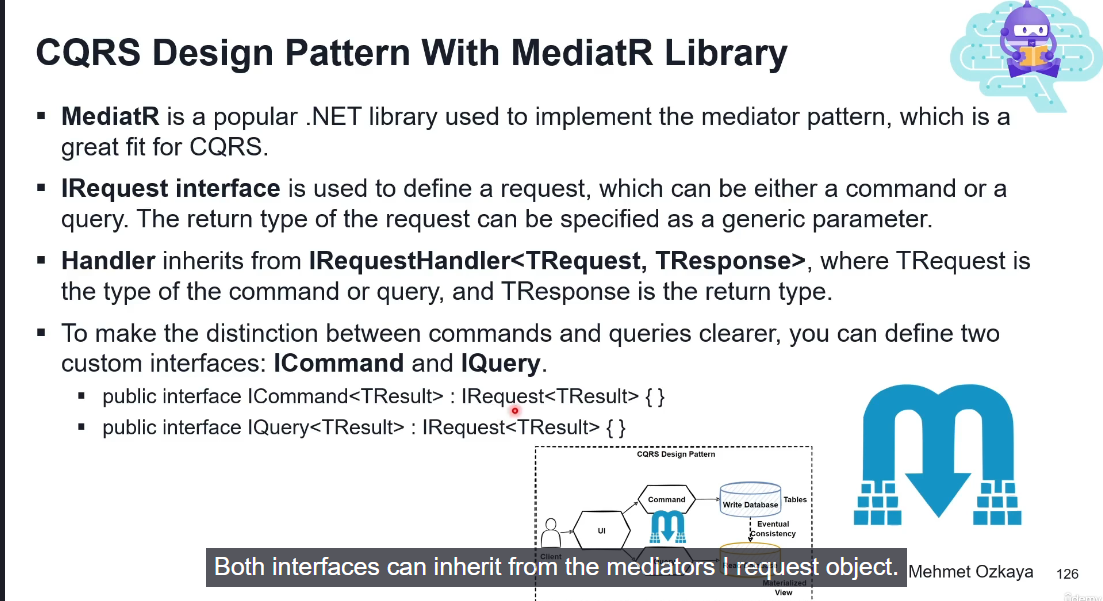
CQRS is particularly useful in systems where:

* There is a high read-to-write ratio.
* You need to scale the read and write sides independently.
* You are building a system that requires complex or evolving business logic.

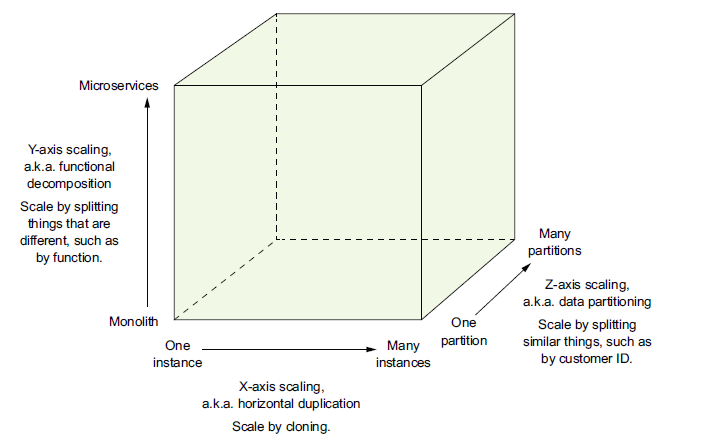
**Example in Practice:**

* **Write Model**: Imagine a system where you store orders. The command side would handle actions like creating, updating, or deleting orders.
* **Read Model**: The query side would handle fetching the details of an order, including potentially complex queries, without affecting the write side.

This pattern is especially popular in event-driven architectures and microservices. However, it can add complexity to the system, especially when you need to handle eventual consistency.



MediatR library



A *service* is a mini application that implements narrowly focused functionality, such as order management, customer management, and so on.

Benefits of Microservices

 It enables the continuous delivery and deployment of large, complex applications.

 Services are small and easily maintained.

 Services are independently deployable.

 Services are independently scalable.

 The microservice architecture enables teams to be autonomous.

 It allows easy experimenting and adoption of new technologies.

 It has better fault isolation.

The patterns are also divided into three layers:

 *Infrastructure patterns* —These solve problems that are mostly infrastructure issues

outside of development.

 *Application infrastructure* —These are for infrastructure issues that also impact

development.

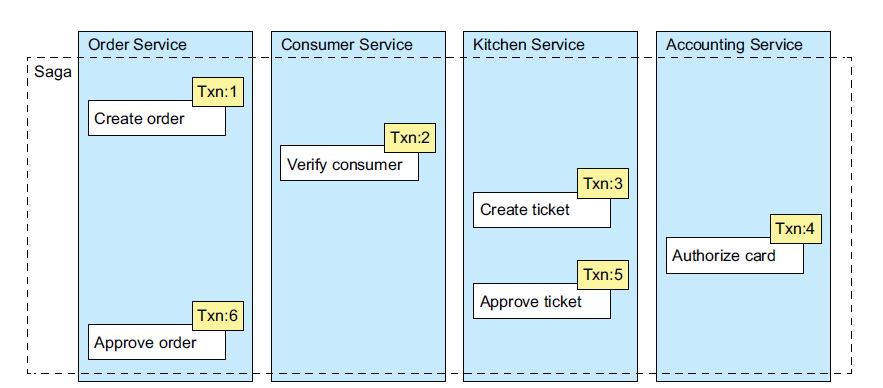
 *Application patterns*—These solve problems faced by developers.

Decomposition Pattern

One problem is that many modern technologies, including NoSQL databases

such as MongoDB and Cassandra, don’t support them. Also, distributed transactions aren’t supported by modern message brokers such as RabbitMQ and Apache Kafka.

*Sagas* are mechanisms to maintain data consistency in a microservice architecture without having to use distributed transactions. You define a saga for each system command that needs to update data in multiple services. A saga is a sequence of local transactions. Each local transaction updates data within a single service using the familiar ACID transaction frameworks and libraries mentioned earlier.



This saga consists of the following local transactions:

1 Order Service—Create an Order in an APPROVAL\_PENDING state.

2 Consumer Service—Verify that the consumer can place an order.

3 Kitchen Service—Validate order details and create a Ticket in the CREATE

\_PENDING.

4 Accounting Service—Authorize consumer’s credit card.

5 Kitchen Service—Change the state of the Ticket to AWAITING\_ACCEPTANCE.

6 Order Service—Change the state of the Order to APPROVED.

This means, for example, that if the authorization of the credit card

fails in the fourth step of the Create Order Saga, the FTGO application must explicitly undo the changes made by the first three steps. You must write what are known as *compensating* *transactions*

A screenshot of a computer

AI-generated content may be incorrect.