# CQRS & Event Sourcing

Nieben van Sint Annaland
06-17-2024

# Introduction

CQRS is a design pattern that separates the read adn write operations of a data store into distinct models:

1. **Commands**: These are operations that change the state of the system. Commands are typically handled by the write model
2. **Queries:** These are operations that retrieve data without modifying it. Queries are handled by the read model.

## Benefits of CQRS

- **Seperation of concerns:** By separating read and writes, each model can be optimized independently. For example, the read model can use denormalized views for fast querying, while the write model can maintain a normalized data structure for transactional integrity.
- **Scalability:** The read and write sides can be scaled independently. For example, if the application has a high read load (which usually happens), you can scale the read side without affecting the write side.
- **Flexibility:** Different data storage mechanisms can be used for the read and write models. For instance, a relational database for the write model and a NoSQL database for the read model.

# Event Sourcing

Event Sourcing is a pattern where state changes are stored as a sequence of events rather than overwriting the current state. Each event represents a change that has occurred in the system.

## How Event Sourcing Works

- **Event Stream**: Instead of storing the current state of an entity, you store a log of all hte events that have modified the entity
- **Reconsitution:** To get the current state of an entity, you replay the events from the event stream

### Benefits of Event Sourcing

- **Auditabilty:** Every state change is logged as an event, providing a complete audit trail of what has happened in the system
- **Reconstruction:** The state of the system can be reconstructed at any point in time by replaying the events
- **Consistency**: Since events are immutable and represent a single point of truth, it is easier to maintain consistency in distributed systems.

# Using CQRS and Event Sourcing Together

When combined, CQRS and Event Sourcing offer powerful capabilities:

- **Write Model (Event Sourcing):** Commands result in events that are appended to an event store. The write model can be reconstructed by replaying the events.
- **Read Model (CQRS):** The read model can be built by projecting the events into a format optimzied for querying. This often involves creating and updating read-optmimized views in response to new events

### Pracitcal Considerations

- **Complexity:** Implementing CQRS and Event Sourcing can introduce significant complexity, especially in terms of eventual consistency and handling concurrent updates.
- **Tooling and Infrastructure:** Requires robust infrastructure to manage event stores and projections, as well as tooling for handling event versioning and replay
- **Domain Modeling**: Works best in domains with complex business logic and frequent state changes, where the benefits of auditability and flexibility outweigh the added complexity

# Advantages over a Tranditional Database

1. Seperation of Concerns
    - **Traditional Database:** Uses the same model for both reading and writing data, which can lead to complex queries and less optimized performance
    - **CQRS:** Separates the read and write models, allowing each to be optimized independently. The write model focuses on transactional integrity and domain logic, while the read model can be designed for fast querying and reporting.
2. Scalability
    - **Traditional Database:** Scaling typically involves replicating the entire database, which can be resource-intensive and complicated.
    - **CQRS:** Read and write sides can be scaled independently. The read model, which often experiences more load, can be replicated or distributed across

multiple nodes, while the write model can be managed separately to ensure consistency.

3. Performance Optimization
    - **Traditional Database:** Performance tuning often involves balancing the needs of read and write operations, which can conflict.
    - **CQRS:** Read models can be denormalized and optimized for specific queries, reducing the complexity and improving the performance of read operations without impacting the write operations
4. Flexibility and Evolvability
    - **Traditoinal Database:** Schema changes can be disruptive, requiring migrations and potentially causing downtime
    - **Event Sourcing:** Events are immutable and versioned. New business requirements can be addressed by adding new event types or projecting events into new read models, without altering existing data.
5. Auditability and Traceability
    - **Traditional Database:** Typically only maintains the current state, making it difficult to trace how and why the state evolved
    - **Event Sourcing:** Maintains a complete log of all changes (events), providing a full audit trail and the ability to reconstruct historical states.

# Contribution to Distributed Data Systems

1. Consistency and Fault Tolerance
    - **Event Sourcing:** Ensures strong consistency for writes as events are the single source of truth. In a distributed system, events can be reliably replicated across nodes.
    - **CQRS:** Helps manage consistency between the read and write sides. Even if read models are eventually consistent, the write models ensures that all changes are consistently recorded.
2. Scalability and Availability
    - **Read Model Scalability:** In a distributed system, read models can be easily replicated and distributed across multiple nodes, enhancing availability and load distribution.
    - **Write Model Scalability:** The write model, although more centralized to maintain consistency, can still benefit from horizontal scaling strategies and fault tolerance mechanisms like leader-follower replication
3. Decoupling and Microservices Architecture
    - **Microservices:** CQRS and Event Sourcing naturally decouple the system into distinct components. Each microservice can handle specific command or query responsibilities and publish event to communicate state changes
    - **Event-Driven Architecture:** Events can be published to an event bus, allowing other microservices to react to changes in a decoupled manner, promoting flexibility and scalability

## 4. Resilience and Recovery

- **Event Replay:** In the event of a failure, the system can recover by replaying events from the event store, ensuring no data loss and consistent state recovery
- **Snapshots:** Periodic snapshots of the current state can optimize recovery times while maintaining the ability to replay events for audit purposes