



# Java Technologies

## Lecture 10

# Data Management in Microservices

Fall, 2025

# Agenda

- The Challenges of Distributed Data
- CAP Theorem, PACELC Design Principle
- Strong/Weak Consistency Models
- ACID vs. BASE, Eventual Consistency
- Shared Database, Database-Per-Service, Hybrid Variants
- Distributed Transactions, Two-Phase Commit (2PC)
- The Saga Pattern
- Event Sourcing, Transactional Outbox
- API Composition
- Command Query Responsibility Segregation (CQRS)

# The Context

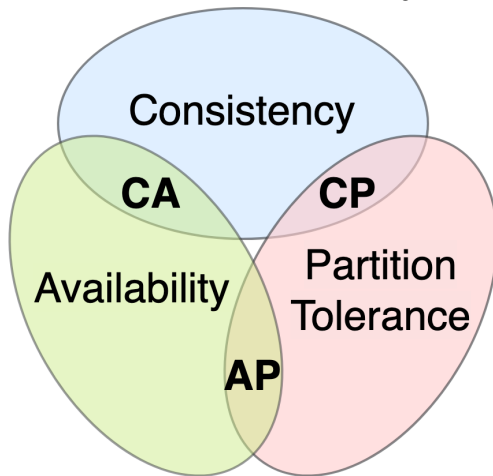
- What's the database architecture in a microservices app? 😬
- Each service should have its own database.
- Services must be loosely coupled.
- Some business transactions must enforce invariants that span multiple services or update data owned by multiple services.
- Some business transactions need to query data that is owned by multiple services.
- Some queries must join data that is owned by multiple services.
- Some databases must be replicated/sharded in order to scale.
- Different services may have different data storage requirements.

# Brewer's CAP Theorem

- A distributed system can have only two of the following: consistency, availability, partition tolerance.
- **Consistency**: All nodes see the same data at the same time.  
→ Every read receives the most recent write or an error.
- **Availability**: The system operates even if some nodes are down.  
→ Every request receives a response (nonerror), without the guarantee that it contains the most recent write.
- **Partition tolerance**: The system operates even if some nodes cannot communicate due to network failure.  
→ Messages can be dropped or delayed.
- **No network can guarantee the lack of failure**, so network partitioning is non-negotiable in distributed systems.

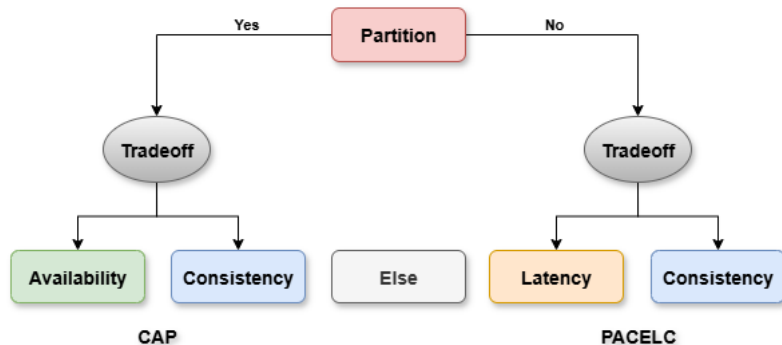
# CAP Theorem Implication

You must sacrifice either consistency or availability.



# PACELC Design Principle

CAP only talks about what happens during network partitions.  
But what about normal operation?



Consistency is "expensive" → Agreement → Coordination →  
Blocking → Latency / Reduced availability

# Consistency Models

The rules for how data is synchronized and perceived across multiple nodes, when read/write operations are executed concurrently.

- **Strong Consistency:** All nodes in a system have the same view of the data at any given moment.
  - Linearizability (or Atomic Consistency)
  - Sequential Consistency
- **Weak Consistency:** Higher availability and lower latency, particularly in the face of network partitions
  - Causal Consistency: "Causally related" operations will be seen in the same order by all nodes.
  - Eventual Consistency: The system will reach consistency at one point in time. However, there may be a period of time when the system is in an inconsistent state.

# Transactions

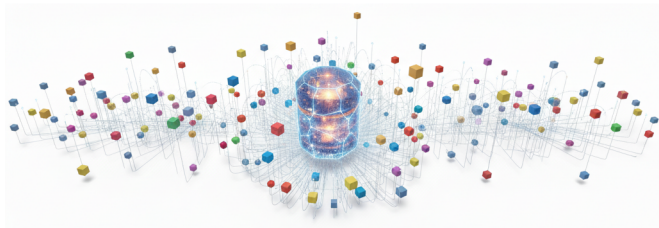
A **transaction** is a single, logical unit of work that must guarantee a set of properties, in order to maintain data integrity, consistency and/or availability of a (distributed) database.

- **ACID**: Consistency over Availability
  - Atomicity, Consistency, Isolation, Durability
- **BASE**: Availability over Consistency
  - **Basically Available**: The database appears to work most of the time; there will be a response to every request.
  - **Soft State**: Different node instances don't have to be mutually consistent all the time. State can change over time, even if there are no reads or writes.
  - **Eventual Consistency**: The system keeps changing the data to make it consistent. Eventually, it becomes consistent.

# Back to Our Microservices

- How many databases?
  - Shared Database
  - Database-Per-Service
  - Hybrid Variants
- How to execute distributed transactions?
  - Two Phase Commit
  - Sagas
- How to query data owned by multiple services?
  - API Composition
  - Command Query Responsibility Segregation (CQRS)
- How to update data owned by multiple services?
  - Domain Events
  - Event Sourcing

# Shared Database



- A **single database** shared by multiple services.
- The database acts as the **integration point** for all services.
- **Advantages:** Simplicity, ACID transaction, easy joins.
- **Disadvantages:**
  - Technology lock-in (lack of flexibility in DBMS choice)
  - Development time coupling (schema)
  - Runtime interferences (locking)
  - Lack of control regarding data invariants
  - Performance bottlenecks

# Database-Per-Service

- Each microservice is assigned **its own private database**.
- Data sharing is done through APIs or async communication.
- **Variants:**
  - Private-tables-per-service
  - Schema-per-service
  - Database-server-per-service
- **Advantages:** Loose coupling, high autonomy, fault isolation, technology flexibility, independent optimization & scaling.
- **Disadvantages:**
  - **Distributed transactions**
  - Increased management overhead
  - Complex queries are harder to implement
  - Performance "issues"

# Hybrid Variants: Shared & Per-Service

- **Shared Operational DB + Per-Service Read Models**  
→ Services write to a shared DB. Each service maintains its own read-optimized copy (via events or replication).
- **Shared DB for Reference Data + Per-Service DB for Core Data**  
→ Reference/master data (countries, currencies) lives in a shared DB. Each service owns its transactional data separately.
- **Per-Service DB + Shared Reporting/Analytics DB**  
→ Each service owns its own DB for operational use. Data is periodically replicated or streamed into a shared data warehouse/lake for cross-service queries, BI, and reports.
- **Polyglot Hybrid**  
→ Some use relational, some NoSql, some Neo4j.
- **Shared DB in Legacy**  
→ Gradual Migration to Per-Service DBs

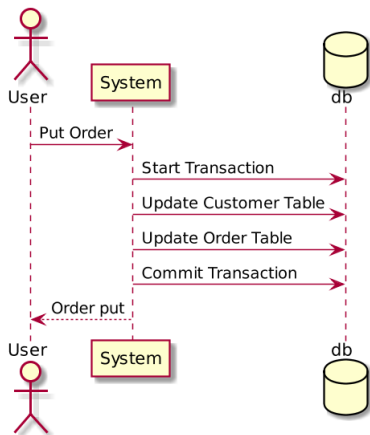
# Distributed Transactions

- A distributed transaction is an operation that **spans multiple systems, databases, or services** to ensure that all of the operations either succeed or fail as a single, atomic unit.
- **Participants:**
  - Resource managers (individual systems, databases, or services)
  - Coordinator (transaction manager)
- **Atomicity:** All participants must reach a uniform consensus on whether a successful commit is possible. If any part fails, the entire transaction must be rolled back.
- **Common Patterns and Protocols:**
  - **Two-Phase Commit (2PC)** → Strong consistency
  - **Saga Pattern** → Eventual consistency

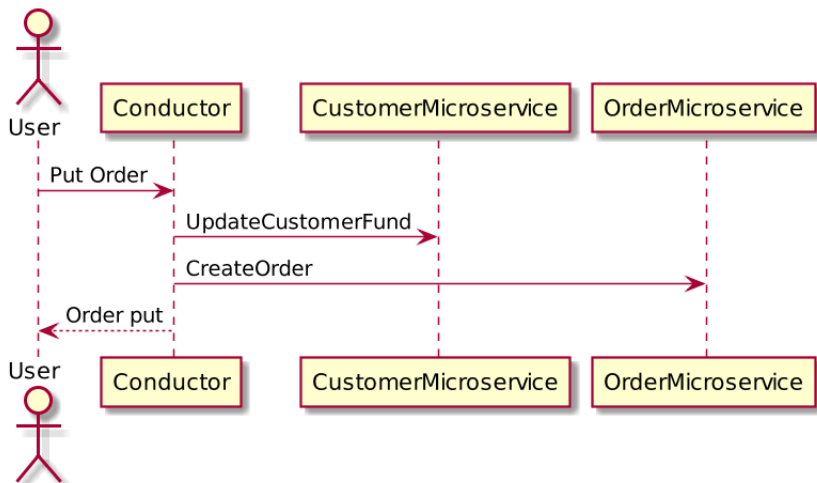
# A Simple Scenario - The Monolith Version

When a customer places an order:

- Deduct the order amount from the customer's fund.
- Create a new order record.



# A Simple Scenario - The Microservices Version



# Two-Phase Commit (2PC)

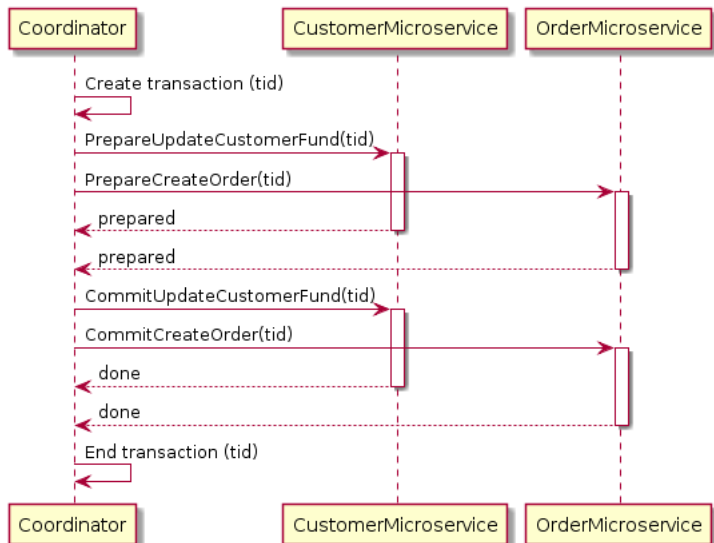
The classic protocol for achieving distributed atomicity. Widely used in relational database systems (XA/JTA).

- 1 **The voting (prepare) phase:** The coordinator asks all participants if they are ready to commit the transaction. Each participant responds with a "yes" or "no" after performing the necessary operations and locking resources.
- 2 **The commit phase:** If all participants vote "yes," the coordinator sends a "commit" message. If even one participant votes "no," the coordinator sends an "abort" message to all participants, and they roll back their changes.

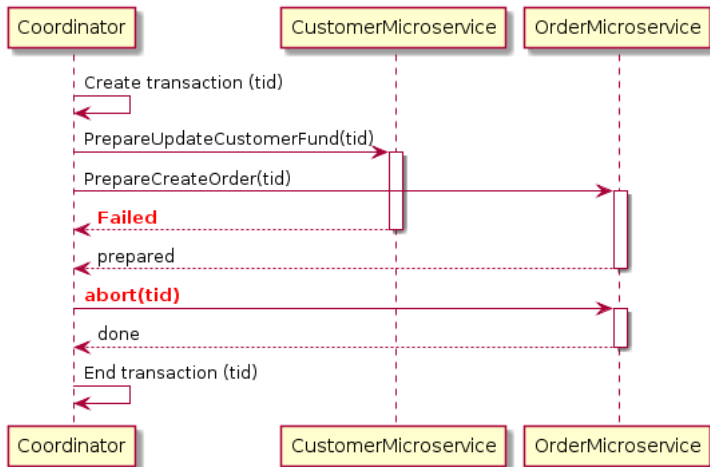
**Disadvantages:** Synchronous communication, blocking protocol, single point of failure (coordinator), resource locking.

→ Doesn't scale well in large microservice architectures.

# A Simple Scenario - 2PC (Success)



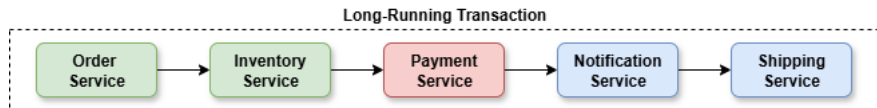
# A Simple Scenario - 2PC (Failure)



# The Saga Pattern

*A long story of heroic achievement, especially in medieval (scandinavian) prose.*

- Some business transactions span over multiple services.  
→ We need a mechanism to ensure data consistency.
- 2PC protocol affects availability by using synchronous communication → we don't want it.
- The Saga pattern is an architectural design pattern that breaks a single, complex business transaction into **a sequence of smaller, local transactions.**
- Each local transaction **updates the database and publishes a message** to trigger the next local transaction in the saga.



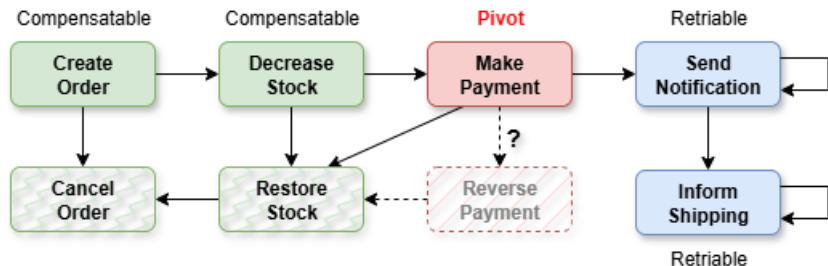
# Local Transactions

- If a local transaction fails because it violates a business rule then the saga executes **compensating transactions** that undo the changes that were made by the preceding local transactions.
- **Compensatable transactions:** Transactions that can be rolled back using a compensating transaction.

$$T_1 \rightarrow T_2 \cdots \rightarrow T_n \text{ (fails)} \rightarrow C_n \rightarrow \cdots \rightarrow C_2 \rightarrow C_1$$

- **Pivot transactions:** This transaction is "the point point of no return". If it succeeds so will the rest of the saga. It usually is the first transaction that is not compensatable.
- **Retriable transactions:** These are transactions that come after the pivot transaction and can be repeated until succeed.

# Compensatable - Pivot - Retriable



- 1 **Before the pivot** → use compensatable steps.
- 2 **At the pivot:** → commit to the process (no undo).
- 3 **After the pivot** → use retriable steps until everything finishes.

# Saga Coordination Styles

- **Choreography**

- There is **no central coordinator**.
- The flow is driven by events: each service publishes events after completing its local transaction in order to trigger the next step.
- **Pros:** Less coupling, no single point of failure.
- **Cons:** Difficult to manage and to debug.

- **Orchestration**

- There is a **dedicated "orchestrator" service** that manages the entire saga workflow. (e.g., OrderSagaManager)
- The orchestrator sends commands to the services, decides the next steps, either continuing the saga or triggering compensating transactions.
- **Pros:** Easier to debug, manage, and understand the workflow. Clear separation of concerns.
- **Cons:** Single point of failure, potential bottleneck.

# The Lack of Isolation

## Problems

- **Lost updates:** One saga updates a record, but another saga overwrites that update before the first one is complete.
- **Dirty reads:** A saga reads the incomplete changes (not yet fully committed) made by another saga.
- **Fuzzy/nonrepeatable reads:** A saga rereads the same data and finds that another saga has modified or deleted it.

## Solutions

- Semantic locks (business-level lock)
- Reread values (optimistic locking)
- Commutative updates (order up updates does not matter)
- Pessimistic view ("temporary" pessimistic records)
- By design (immutable data / custom logic to prevent conflicts)

# Implementing the Saga Pattern

- **Choose the saga type:**
  - Choreography: Use messaging (Kafka, RabbitMQ)
  - Orchestration: Create a new service manager
- **Implement "undo"** operations for each step that is compensatable.
- **Implement "retry"** logic and ensure idempotency for retries.
- **Log and trace** the saga for auditing and recovery.
- **Additional tools & libraries:**
  - Eclipse MicroProfile LRA (Long Running Actions)
  - Tempomatic Saga, Axon, Eventuate Tram, Temporal, Camunda/Zeebe, Seata
  - Spring State Machine
  - Spring Integration

## The Dual-Write Problem

- In the context of a saga, for each local transaction, there are two actions that must be done atomically:
  - ① Update the database (change the system state)
  - ② Publish the domain event
- How to make sure the two actions are executed **atomically**?

## Solutions

- **Event Sourcing**: Merge ① and ②, such that the state of the system changes only as a result of a successful published event.
- **Transactional Outbox**: Execute ① (change the state) and implement a mechanism that guarantees the execution of ② (publishing the event).

# Event Sourcing

- We check the bank account, the balance is zero. How did this happen? How much money was there last week? 😞
- **Event Sourcing** is an architectural pattern where the state of a system is not stored as a single, current snapshot in a database, but is instead derived from **log of all the events** that have occurred in the system.
- In a traditional approach, a record is updated or deleted.

```
UPDATE account SET balance = 0;
```

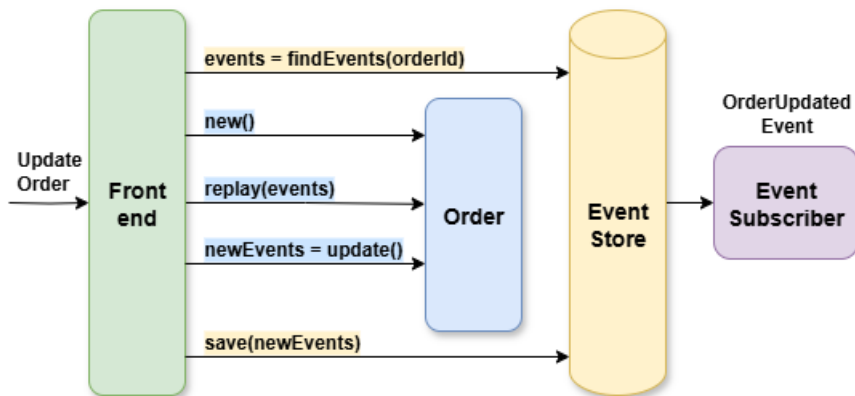
- An event-sourced system records every change as an "event", storing them in a database log, in the order in which they happened.

```
AccountCreated()  
MoneyDeposited(1000)  
MoneyWithdrawn(600)  
MoneyTransferred(400)
```

# Event Sourcing Key Concepts

- **Event:** A state changing **immutable fact** that something happened (e.g, UserSignedUp, MoneyDeposited).
- **Event Store:** An **append-only** log where events are stored. Immutability + append only = tamper-proof history.
- **Aggregate:** An entity **reconstructed** by replaying all events related to it (e.g., a User, an Account).
- **Projection / Materialized View:** Pre-built, read-optimized views of the data that are updated asynchronously as new events are added to the event store (e.g., current account balance)
- **Snapshot:** Intermediate state stored at a given time. To rebuild the current state, you load the latest snapshot and only replay events that occurred after it.

# Event Sourcing Flow



# Benefits & Challenges of Event Sourcing

## Benefits

- **Auditability:** Full history of everything that happened.
- **Time travel:** Temporal queries, debugging by replaying events.
- **Flexibility:** Create new projections anytime.
- **Event-driven integration:** Other services can react to events.

## Challenges

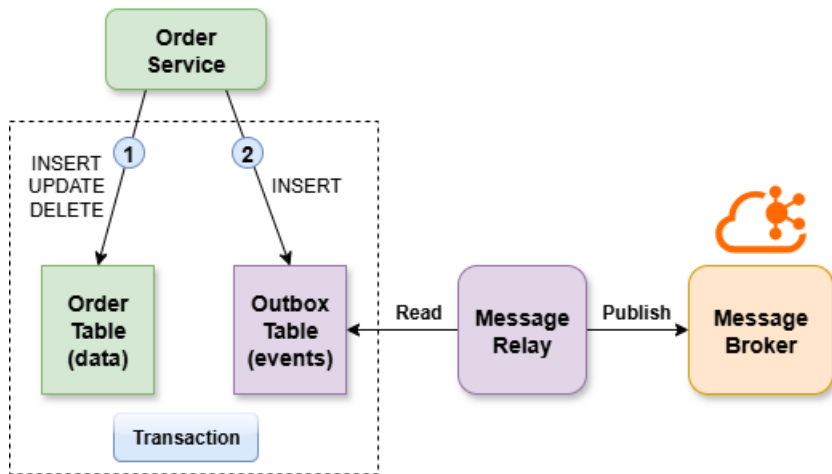
- **Complexity:** It can introduce significant architectural overhead.
- **Event evolution :** Changing event schema over time is tricky.
- **Storage growth:** Event store can grow very large.
- **Performance:** Replaying events may be time consuming.

**Implementations:** Axon, EventStoreDB, Eventuate, etc.

# Transactional Outbox

- A design pattern that ensures that a database change and an event publication are **completed atomically**.
- **The Outbox table:** A dedicated database table is used to store messages that need to be published.
- **Atomic write:** When a business operation occurs, the service performs two writes within a single, local database transaction:
  - 1 It updates the business data.
  - 2 It inserts a new record into the Outbox table: the event.
- **Guaranteed consistency:** Both writes are in the same tx.
- **The message relay:** An "Outbox Processor" is responsible for reading the events from the Outbox table.
- **Publishing the event:** The Message Relay reads the new events from the Outbox and publishes them to the broker.

# Transactional Outbox Flow



# Implementing the Outbox Pattern

## Benefits

- **Atomicity:** DB and event log are always consistent.
- **Reliability:** No lost messages.
- **Simplicity:** Avoids distributed transactions or 2PC.
- **Traceability:** Outbox contains all published events.

## Trade-offs

- **Extra storage:** Outbox table can grow large if not managed.
- **Latency:** Delay depends on poller frequency.
- **Complexity:** Requires a background job.

## Implementations

- **Polling publisher:** Periodically scans the outbox table.
- **Transaction log tailing:** Change data capture (CDC)
- **Frameworks:** Axon Framework, Eventuate, Micronaut Data.

# Aggregating Data from Multiple Microservices

- The data needed to answer a query might be **scattered across different machines**, and even different types of databases.
- In a monolith connecting to a single relational database, you'd simply use a **JOIN** on some tables.
- In a microservice architecture, each service should be independent and own its **private database**.
- How to execute a query that needs data from multiple microservices?
- **Challenges:**
  - **Distributed data:** Data is scattered across network.
  - **Latency:** Every service call is a network hop.
  - **Over-fetching:** Retrieving more than a service needs.
- **Solutions:** Composition, Duplication, Caching

# API Composition

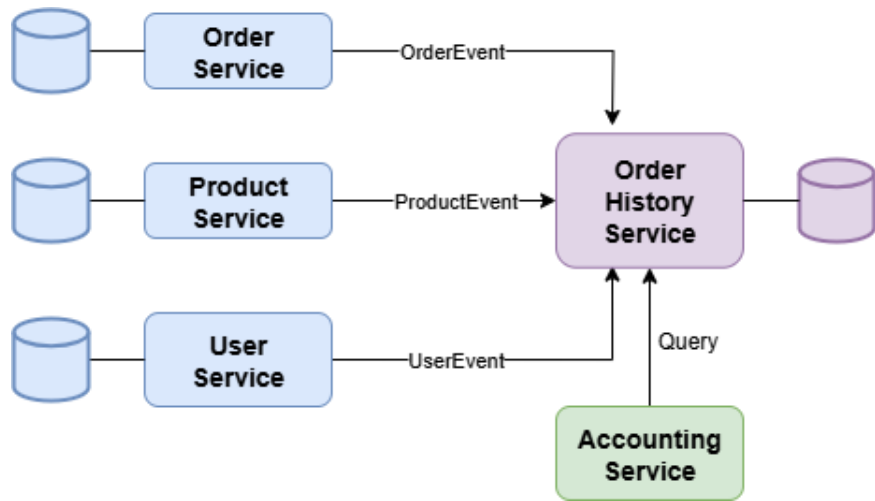
- **Client-Side:** The client itself makes multiple API calls to the services that own the data, and merges the results.
  - ✓ Simple, keeps services decoupled.
  - ✗ More network round-trips; hard to optimize.
- **Server-Side**
  - **API Gateway Aggregation:** It orchestrates calls to multiple services, merges the data, and returns a single response.
    - ✓ Centralized, hides complexity from clients.
    - ✗ Can become a bottleneck; tight coupling
  - **Aggregator Service:** A separate dedicated orchestrator microservice is responsible for aggregation.
    - ✓ Encapsulates aggregation logic; Backend-for-Frontend (BFF)
    - ✗ Adds another service to maintain.
- In both cases, API composition may lead to inefficient, in-memory joins of large datasets.

# CQRS Command Query Responsibility Segregation

Architectural pattern that separates the data model for reading (queries) from the data model for writing (commands).

- 1 Each microservice owns its data and processes write operations (commands) independently.
- 2 When a service's data changes, it publishes a **domain event**.
- 3 A separate "**Read Model Service**" or "Aggregator Service" subscribes to these domain events.
- 4 Upon receiving an event, the Read Model Service **updates its own denormalized, read-optimized database**.
- 5 Clients **query the dedicated Read Model Service**, which provides a fast and pre-aggregated response.

# CQRS Flow



# CQRS Pros & Cons

- **Pros**

- Performance – read queries are extremely fast.
- Avoids complex joins and multiple service calls at query time.
- Enables polyglot persistence. 💡
- Read and write operations scale independently.
- Works well with Event Sourcing pattern.

- **Cons**

- Eventual consistency.
- Increased architectural complexity.
- More infrastructure – event streaming.

- **Implementations:** Axon Framework, Eventuate Tram, Spring Boot with Event Sourcing support.