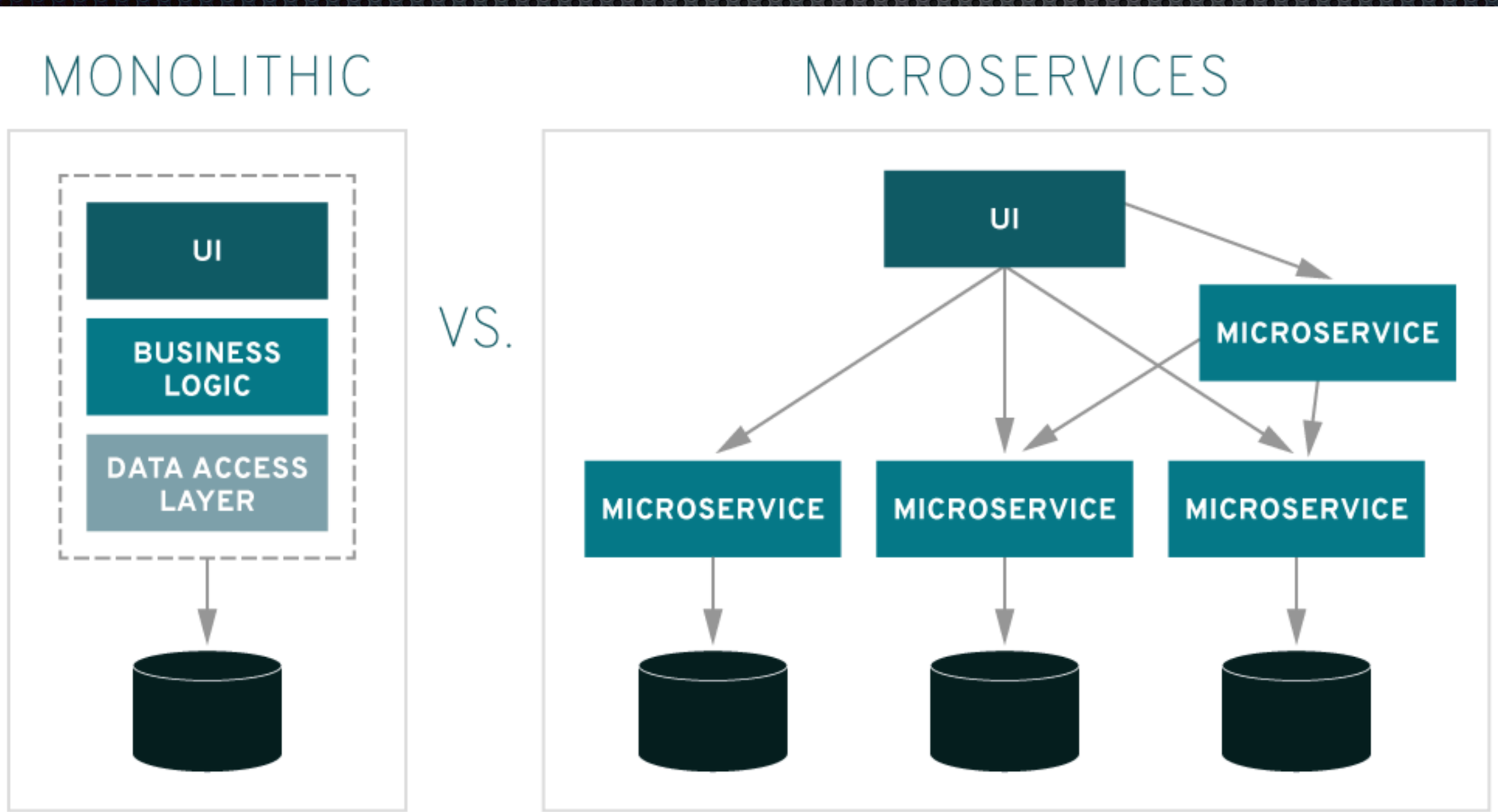
WHY CQRS and ES ?

CQRS : Command and Query Responsibility Segregation
ES　　: Event Sourcing

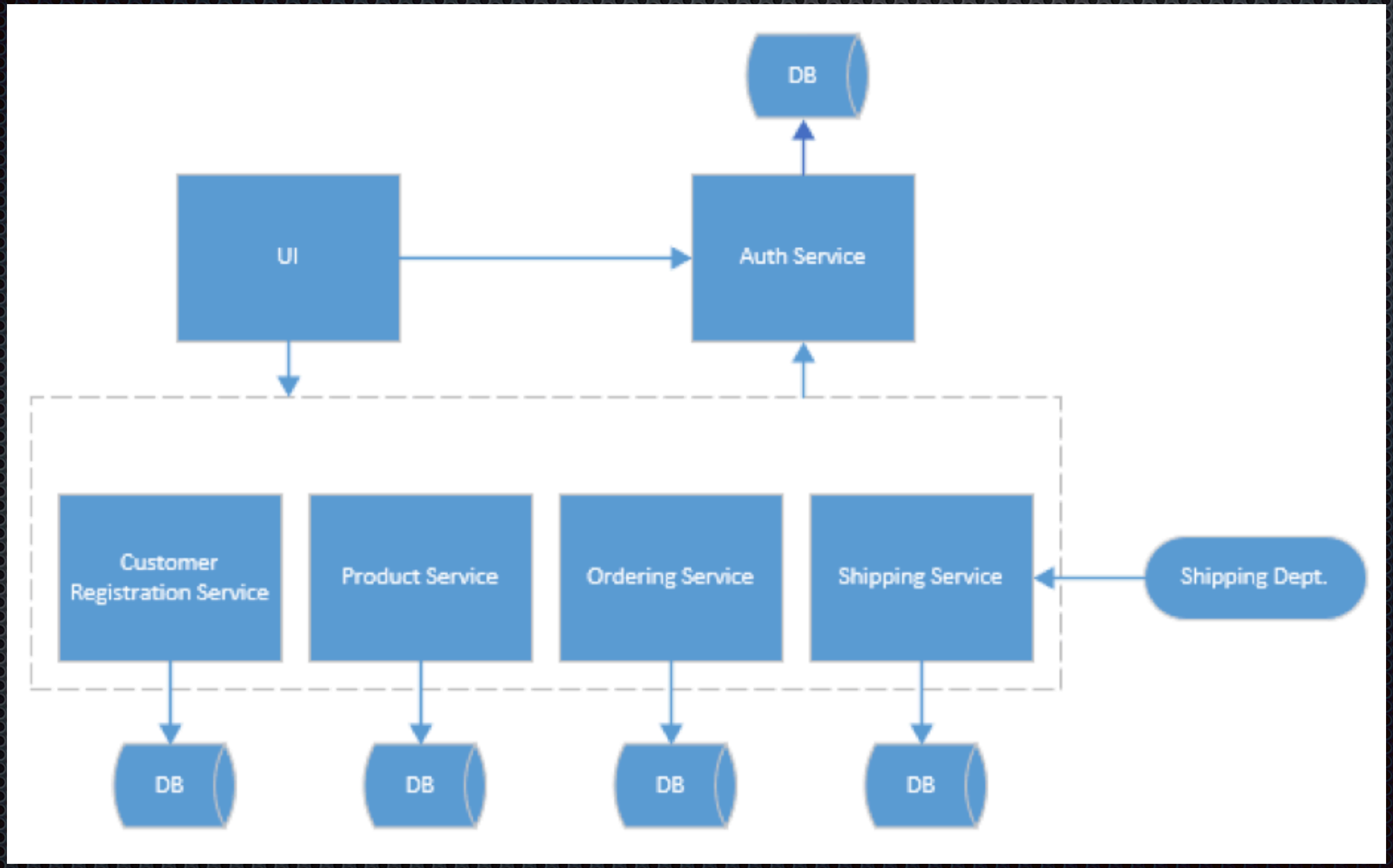
Agenda

- Brief Introduction why we are here
- Data consistency challenge in micro-services
- Domain Driven Design to Rescue
- How to implement DDD
- CQRS and ES Deep Dive
- Pros and Cons of CQRS-ES
- Does this to our ecosystem
- Questions

Migrate Monolith To Micro-services



Example Micro-Services Layout



Challenges Comparison In Short

	Monolithic	Microservice
Architecture	Built as a single logical executable (typically the server-side part of a three tier client-server-database architecture)	Built as a suite of small services, each running separately and communicating with lightweight mechanisms
Modularity	Based on language features	Based on business capabilities
Agility	Changes to the system involve building and deploying a new version of the entire application	Changes can be applied to each service independently
Scaling	Entire application scaled horizontally behind a load-balancer	Each service scaled independently when needed
Implementation	Typically written in one language	Each service implemented in the language that best fits the need
Maintainability	Large code base intimidating to new developers	Smaller code base easier to manage
Transaction	ACID	BASE

Data Consistency Challenge In Micro-Services

- **CAP Theorem**
  - **Consistency:** Every read receives the most recent write or an error
  - **Availability:** Every request receives a (non-error) response, without the guarantee that it contains the most recent write. Another way to state this—all working nodes in the distributed system return a valid response for any request, without exception
  - **Partition tolerance:** Partition tolerance means that the cluster must continue to work despite any number of communication breakdowns between nodes in the system.
- **All three cannot be achieved at the same time. So it either CP or AP**
- **ACID (A**tomic, **C**onsistent, **I**solated, **D**urable**)**
  - **JTA/2PC**
- **BASE (B**asic **A**vailability, **S**oft-state, **E**ventual consistency**)**
  - **Saga**
    - **Choreography**
    - **Orchestration**

Why Throughout ACID and 2PC Is NOT A Option ?

In Monolith

Begin Transaction

Select * from Order where customerId = ?

Select availableCount from ProductInventory where productId = ?

… do some business validation

Insert into Order ….

…

Commit Transaction / Rollback Transaction

Why Throughout ACID and 2PC Is NOT A Option ?

In Micro-Services

Private To Order Service

Private To Product Service

Begin Transaction

Select * from **Order** where customerId = ?

Select availableCount from **ProductInventory** where productId = ?

… do some business validation

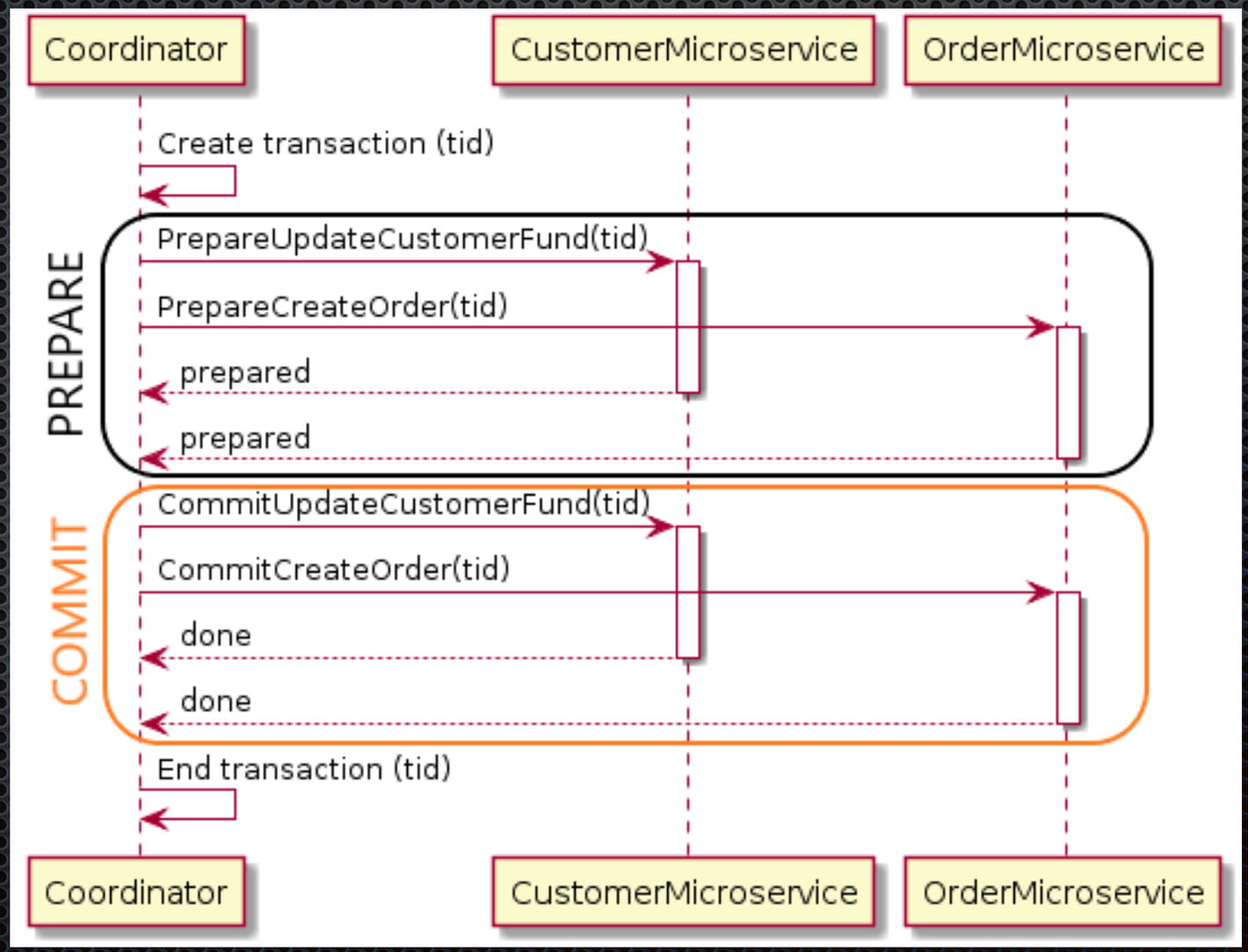
Insert into Order ….

…

Commit Transaction / Rollback Transaction

How do we maintain the consistency of data across multiple services?

Two-Phase Commit ?
Two phase commit (2PC) is a well known algorithm to achieve the benefits of ACID. Handling 2PC is not that simple. 2PC consists of two stages, one is “PREPARE” and “COMMIT”.

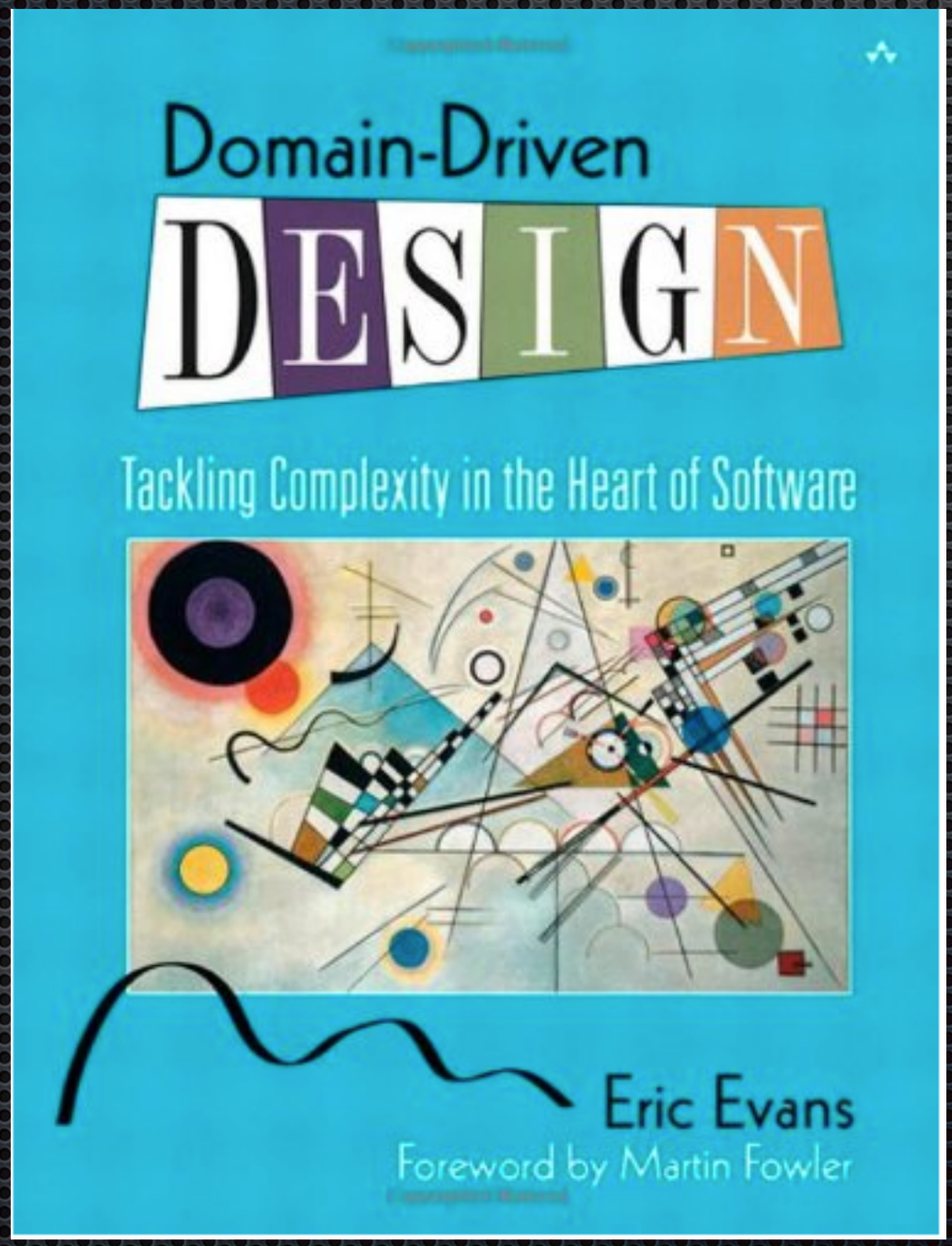


This solves the problem up to a certain extend, but still unable to solve below.

- There is no mechanism to rollback the other transaction if one micro service goes unavailable in commit phase.
- Others have to wait until the slowest resource finish its confirmation.

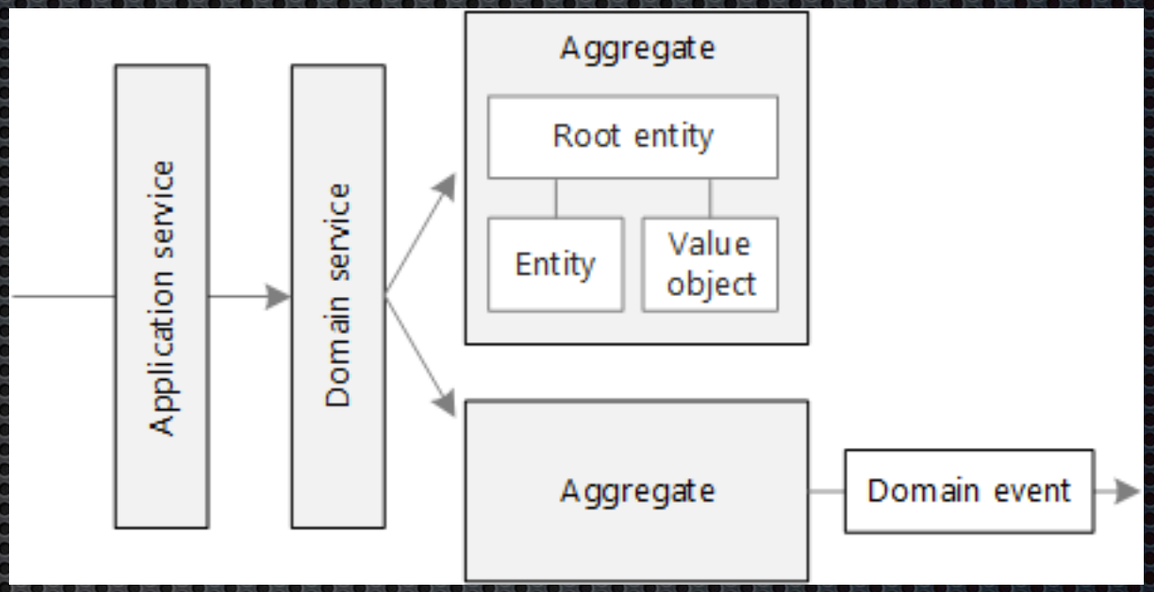
2PC is not a viable option

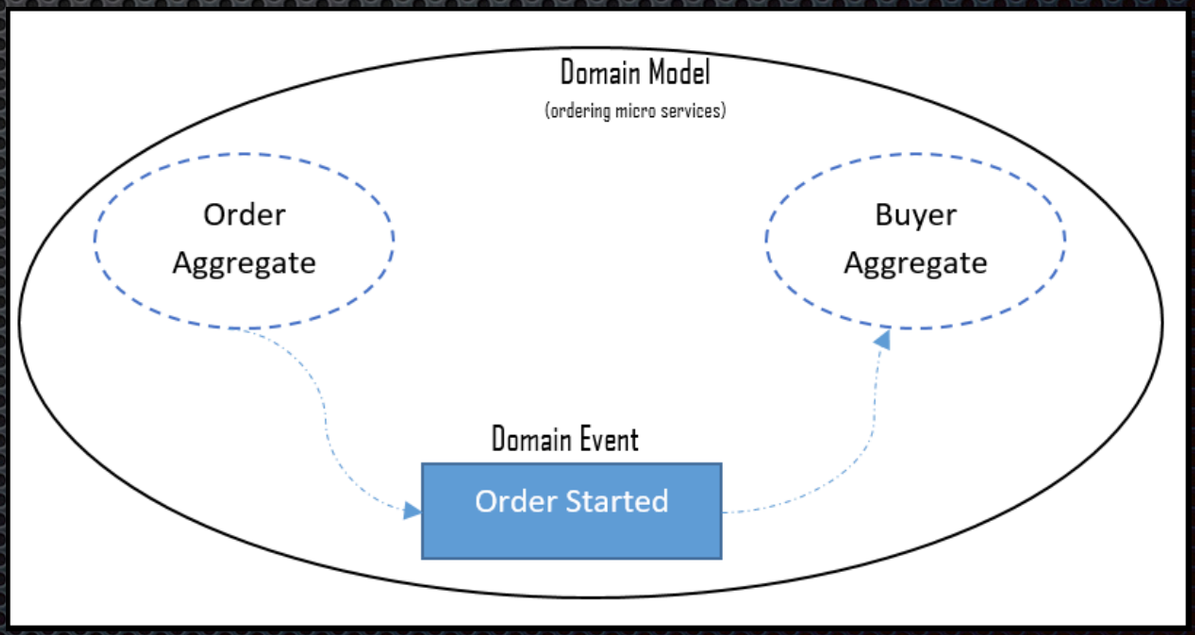
Domain Driven Design To Rescue



Domain Driven Design Concepts

- Business Domain
- Domain Services
- Bounded Context
- Aggregates
  - One or More Entities
  - Value Object
- Domain Events

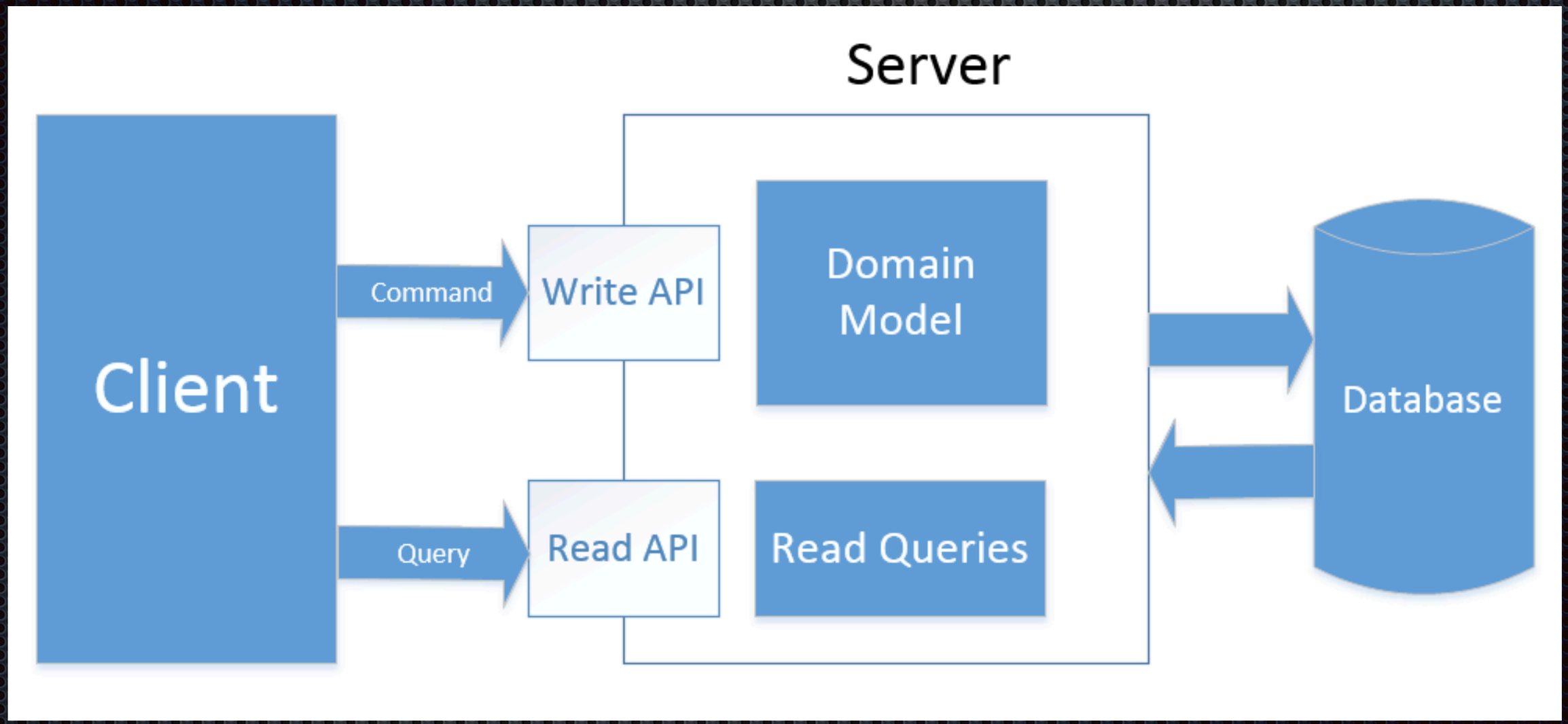




How To Maintain Data Consistency Between Aggregates

Event Driven Architecture To Rescue

What Is CQRS ?


Server
Client
Command
Write API
Domain Model
Database
Query
Read API
Read Queries

Three Building Block Of Message Types

- **Command** : A Command is a request for changing the internal state of the system (e.g. ConfirmOrder)
- **Query** : A query ask for a snapshot of the state (e.g. OrderConfirmed)
- **Event** : Events are the “atoms” of System state change(e.g. GetOrderDetails)

What Is Event Sourcing ?

- An architectural pattern that advocates the Design and Development of the applications by treating it as a **System Of Events** rather as **System Of State**

- Event Sourcing mandates the state of the application should **not be explicitly stored in a database** but as a **series of state-changing events**

- The state of an application is represented by the state of its various Aggregates and any operation on an Aggregate results in an event that describes the state change of the Aggregate.

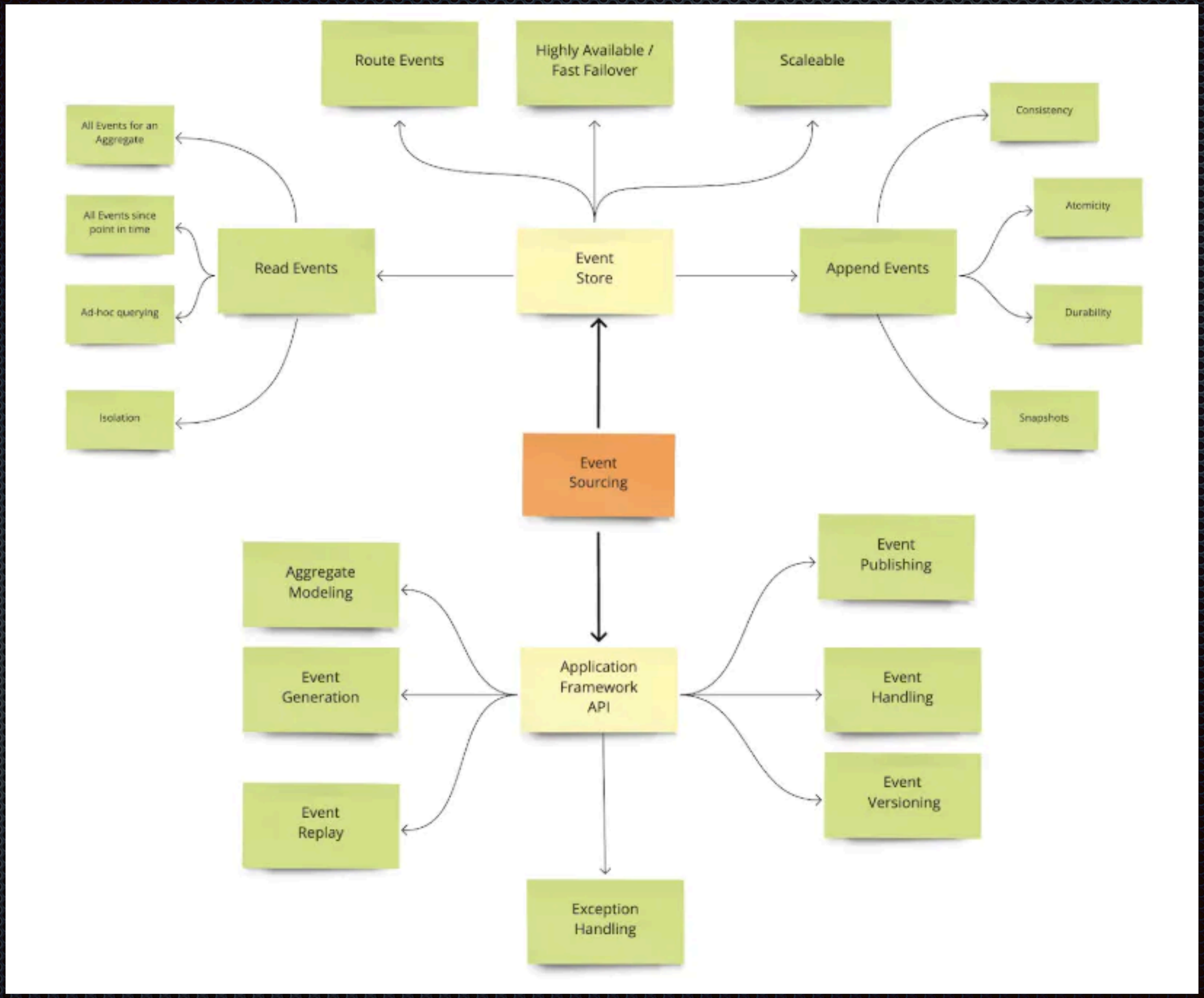
- This event is stored in a Database in an append-only fashion to set the state of Events that might have already occurred for that Aggregate. The current state of an Aggregate is then reconstructed by loading the full history of events from the Database and replaying it.

- Almost always used with CQRS pattern

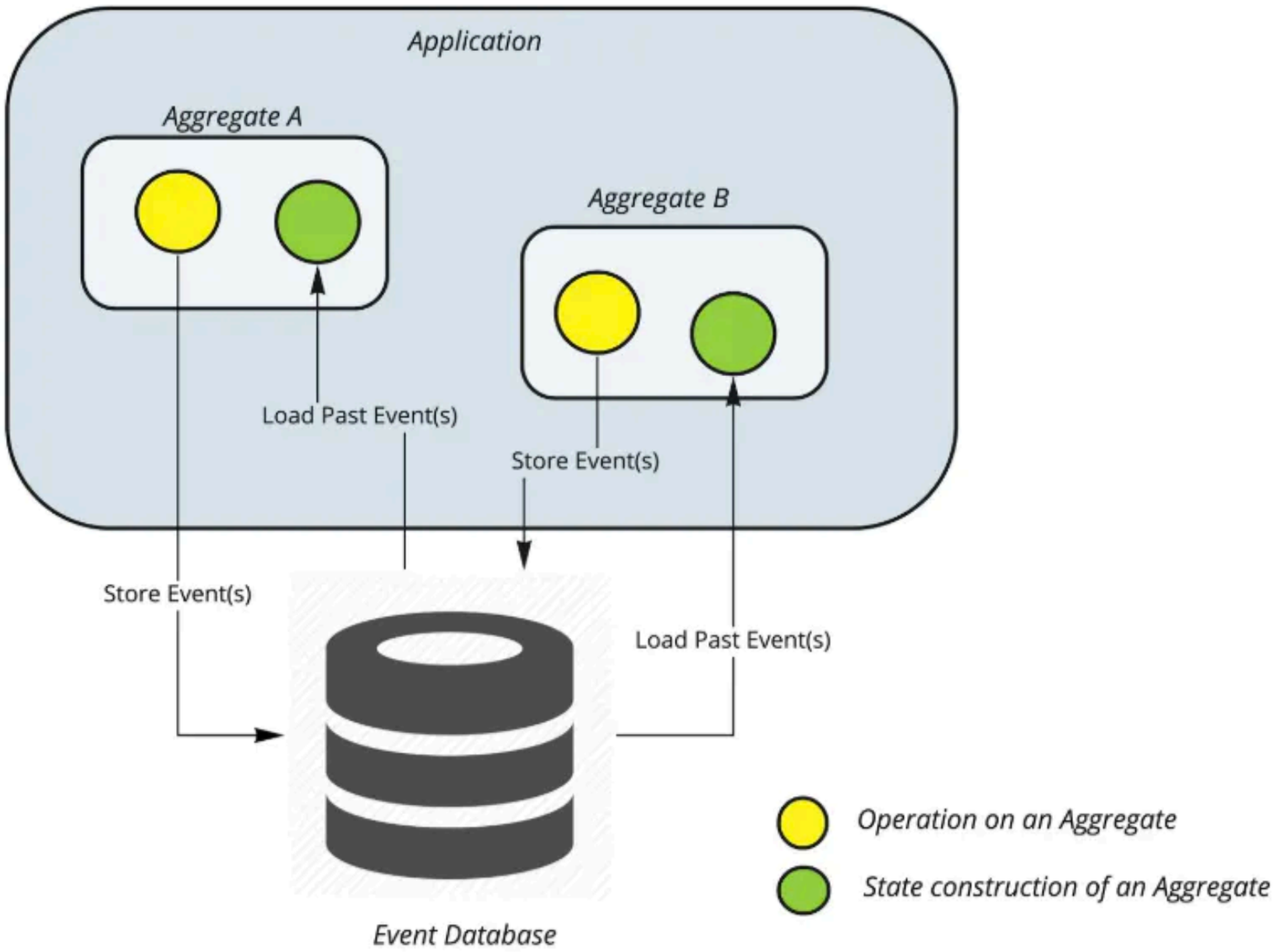
Event Sourcing Implementation

- Implementation of Event Sourcing involves implementing a set of concerns around
  - Domain Driven Design
  - Event Storage (Event Store)
  - CQRS
- Implementing these capabilities in an Event Sourced Architecture is done through the combination of a
  - **Physical Infrastructure**
    - An event store which as the database for the events
  - **Logical Infrastructure**
    - An Application Framework which provides an API to
      - Model Aggregates
      - Handle Command/Queries
      - Event Publishing
      - Event Handling
      - Perform event sourcing operation
      - Event Replay
      - Event Versioning
      - Exception Handling

Event Sourcing Capability Map



Event Sourcing Concept

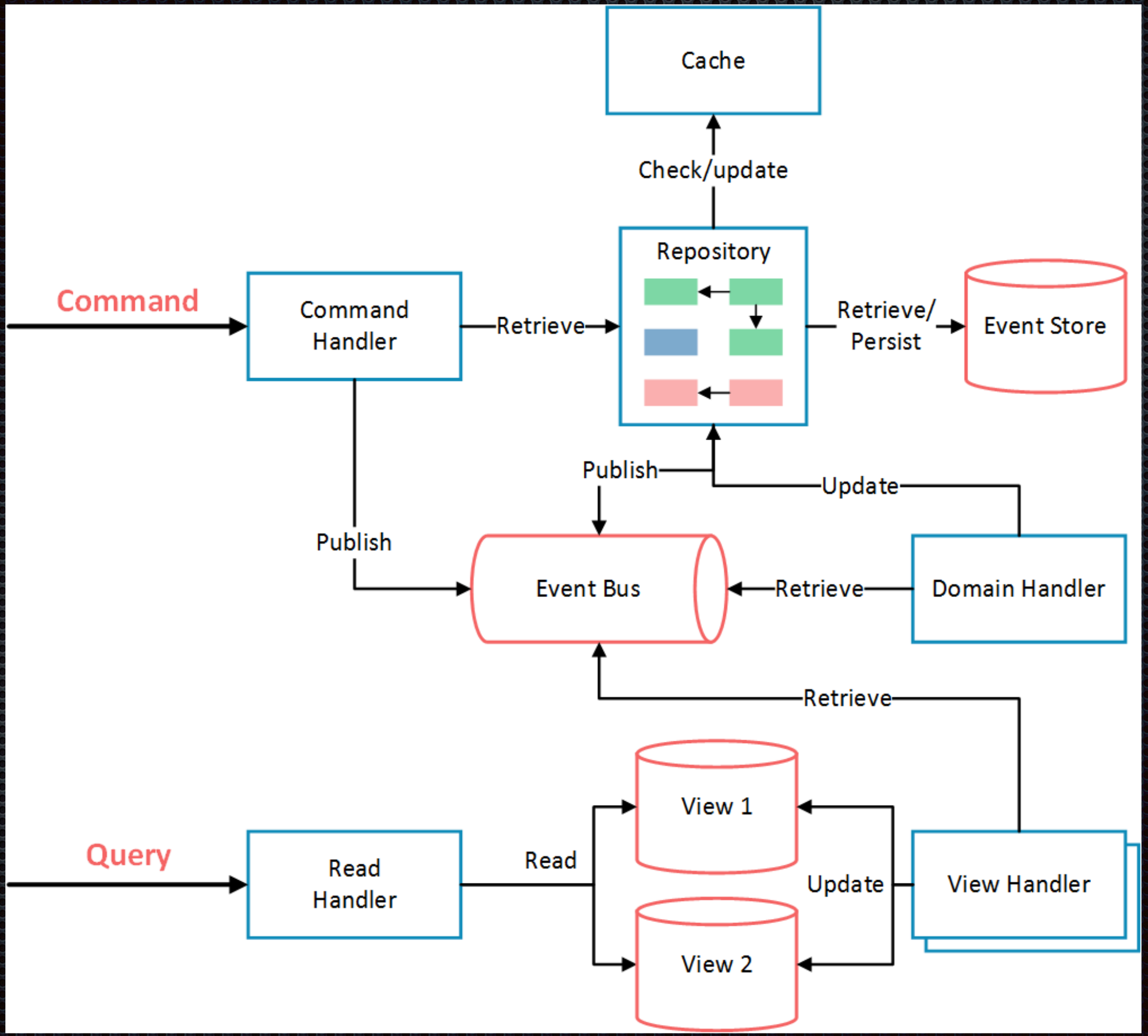


The Event Sourcing concept

CRUD Vs Event Sourcing

Online order	Traditional Storage	Event Sourcing
Select pizza	Order 123: Selected pizza	T1: Selected pizza
Select cola	Order 123: Selected pizza, cola	T1: Selected pizza T2: Selected cola
Select ice cream	Order 123: Selected pizza, cola, ice cream	T1: Selected pizza T2: Selected cola T3: Selected ice cream
Deselect ice cream	Order 123: Selected pizza, cola	T1: Selected pizza T2: Selected cola T3: Selected ice cream T4: Deselected ice cream
Confirm order	Order 123: Ordered pizza, cola	T1: Selected pizza T2: Selected cola T3: Selected ice cream T4: Deselected ice cream T5: Confirmed order

How CQRS Work With Event Sourcing ?



Familiar concepts restructured

```
class Customer {

  public void reserveCredit(
      orderId : String,
      amount : Money) {

    // verify

    // update state
    this.xyz = …
}
```

```
public List<Event> process(
      ReserveCreditCommand cmd) {

   // verify
    …
   return singletonList(
      new CreditReservedEvent(…);
    )
 }
```

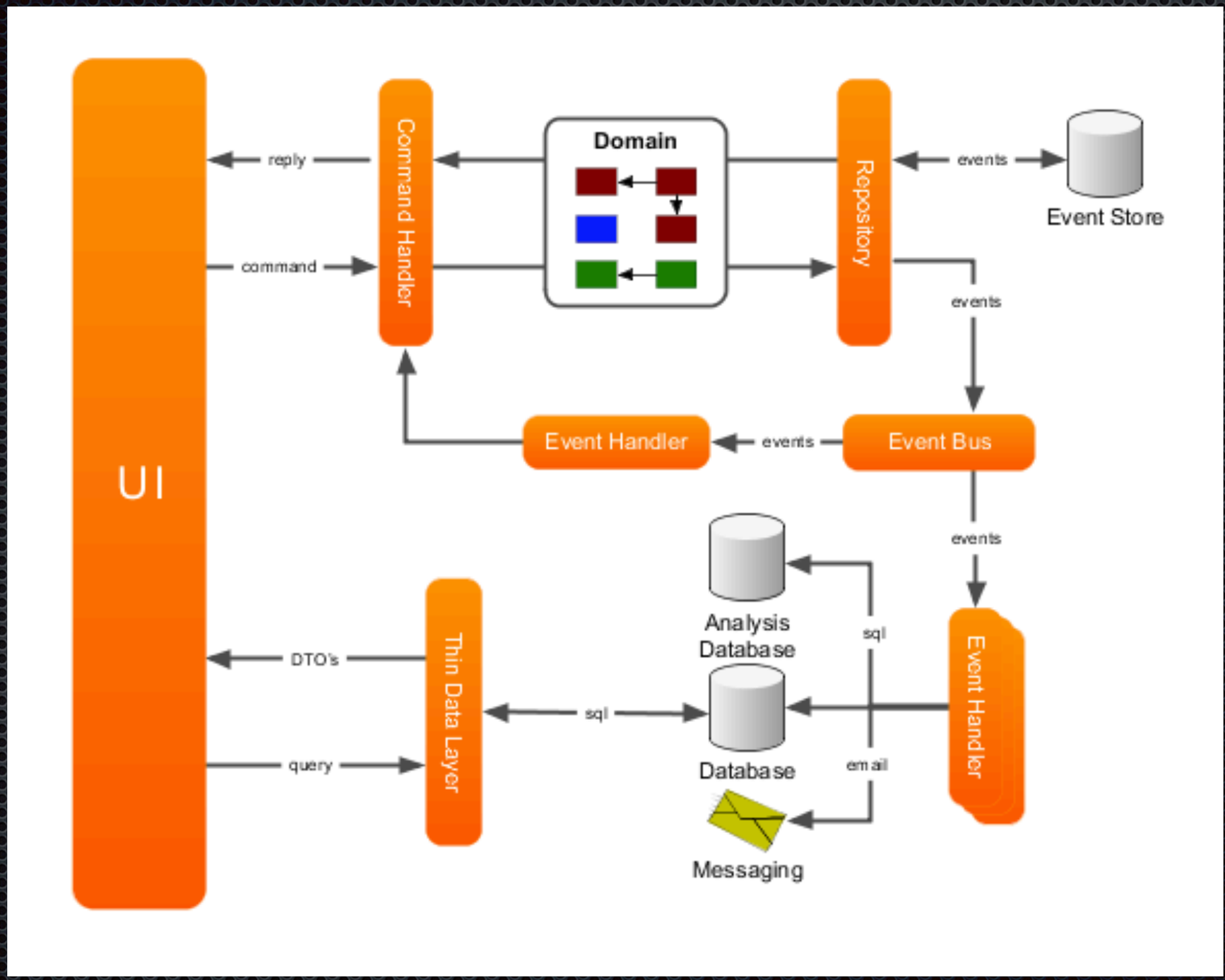
```
public void apply(
     CreditReservedEvent event) {

    // update state
    this.xyz = event.xyz
}
```

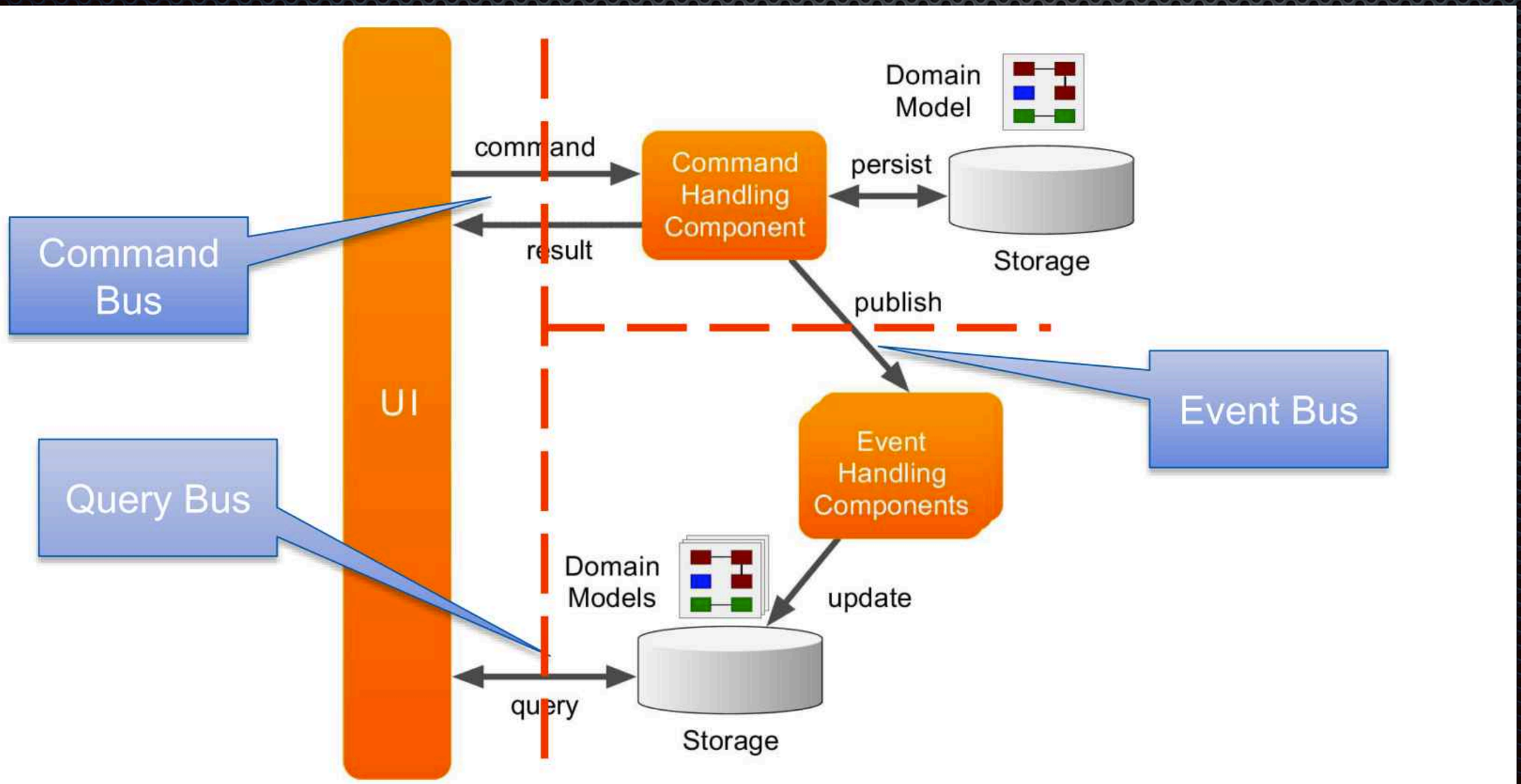
Request handling in an event sourced application

Order Service

Axon's Framework To Rescue



Axon's Framework Drill Down



DEMO

Event Storage Requirement

Event Storage Requirements

Read Events/Snapshots

- ☐ All for an aggregate
  - ☐ • Latests snapshot + later events
  - ☐ • All events
- ☐ All since point in time
  - ☐ • pushing new ones
- ☐ Read back in write order
- ☐ Ad-hoc queries
- ☐ Only read committed events
- ☐ Optimized for recent events

Append Events/Snapshots

- ☐ Validate aggregate sequence numbers
- ☐ Append multiple events at once
- ☐ Committed events protected against loss
- ☐ Append snapshots
- ☐ Constant performance as a function of storage size

AxonIQ

Why Kafka Is NOT Fit For Event Store

Kafka

Pros

- Messaging focussed
- Extremely scalable
  in #total events

Cons

- Not scalable in #aggregates

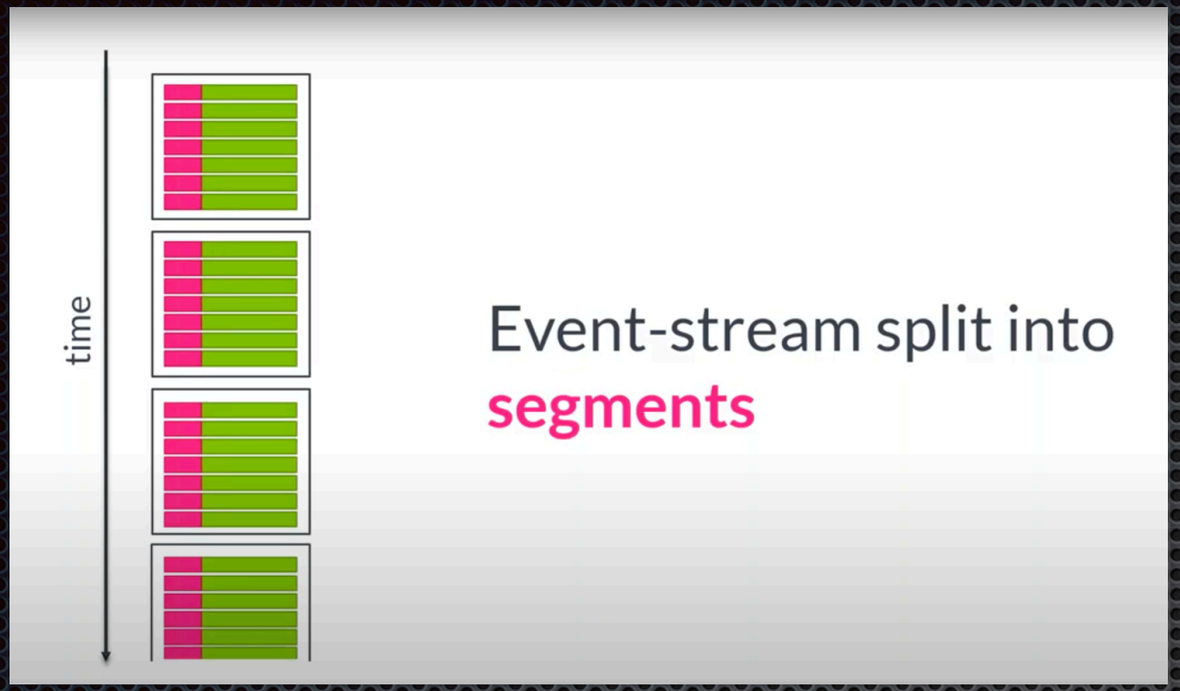
Read Events/Snapshots

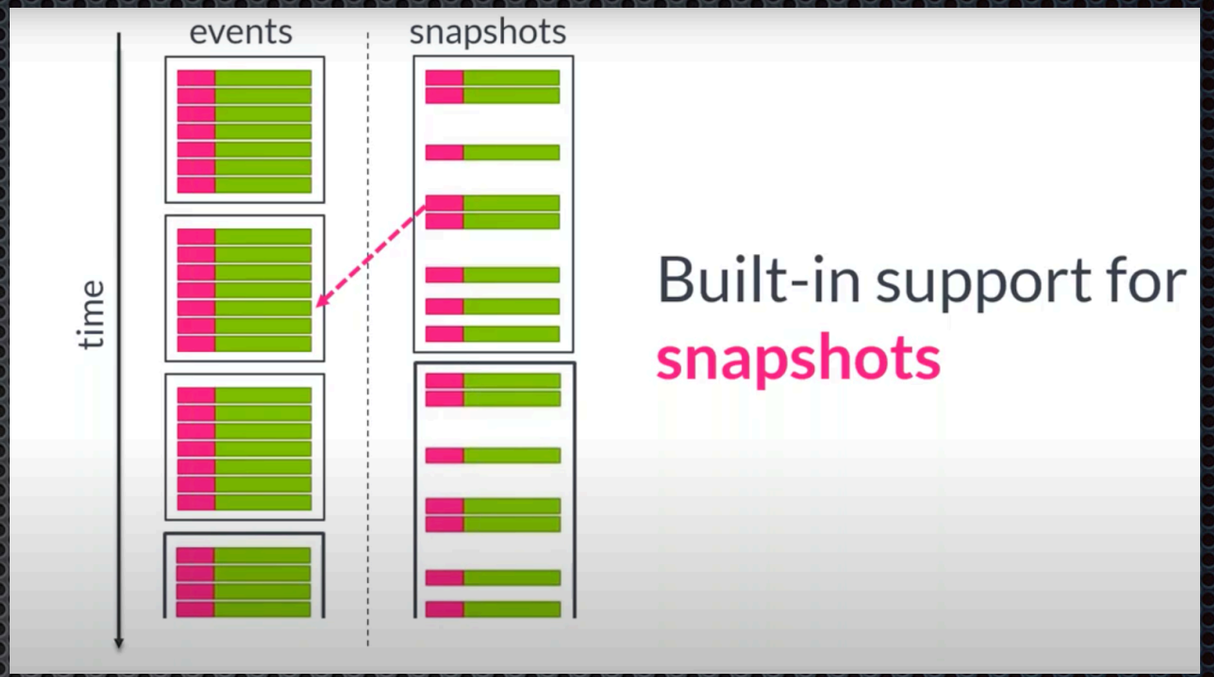
All for an aggregate
- [x] • Latests snapshot + later events
- [x] • All events
- [ ] All since point in time
- [ ] • Pushing new events
- [ ] Read back in write order
- [ ] Only read committed events
- [ ] Optimized for recent events

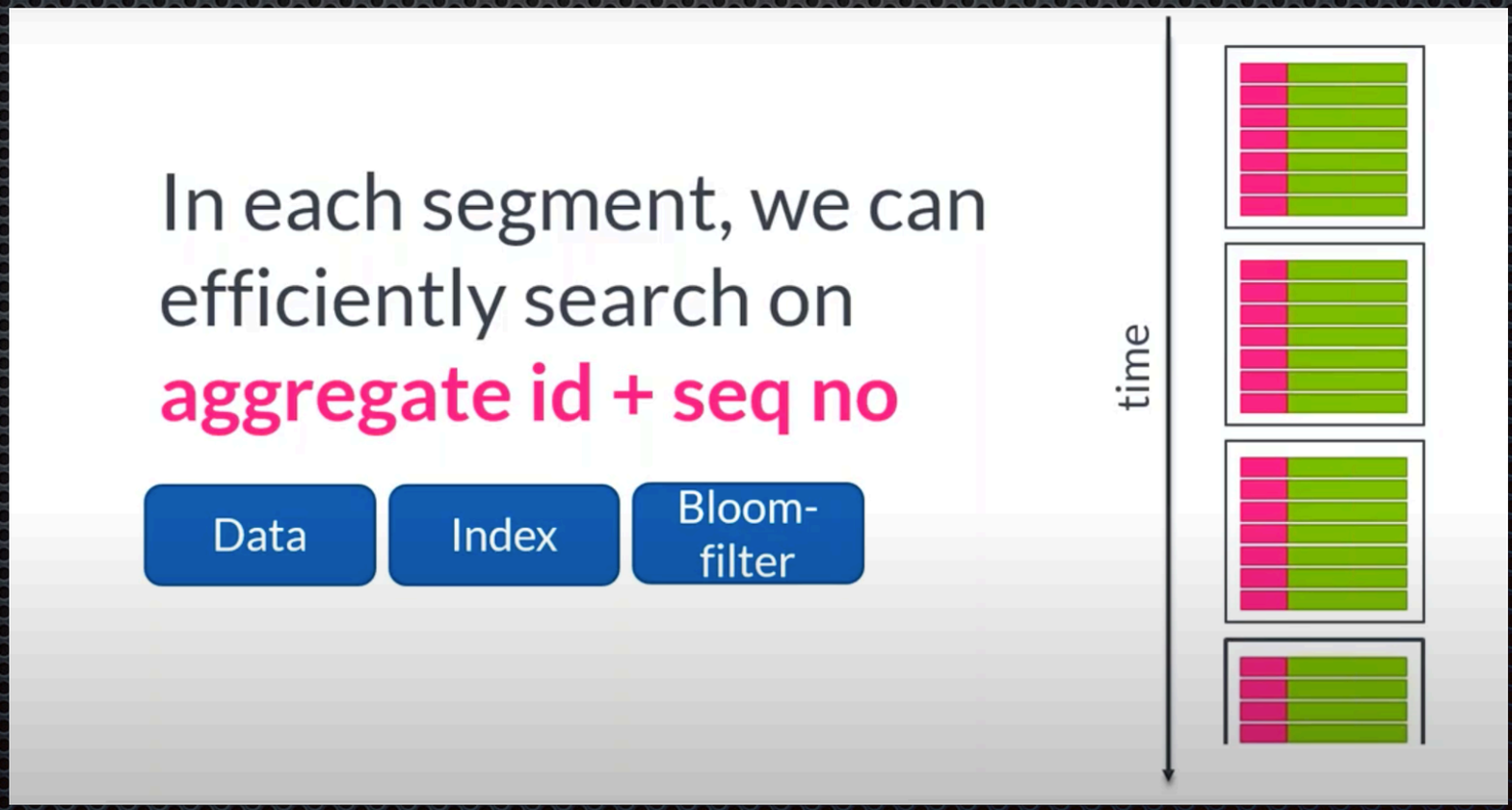
Append Events/Snapshot

- [x] Validate aggregate sequence numbers
- [ ] Append multiple events at once
- [ ] Committed events protected against loss
- [ ] Append snapshots
- [ ] Constant performance as a function of storage size

Event Storage Insight







CQRS And ES

Pros :

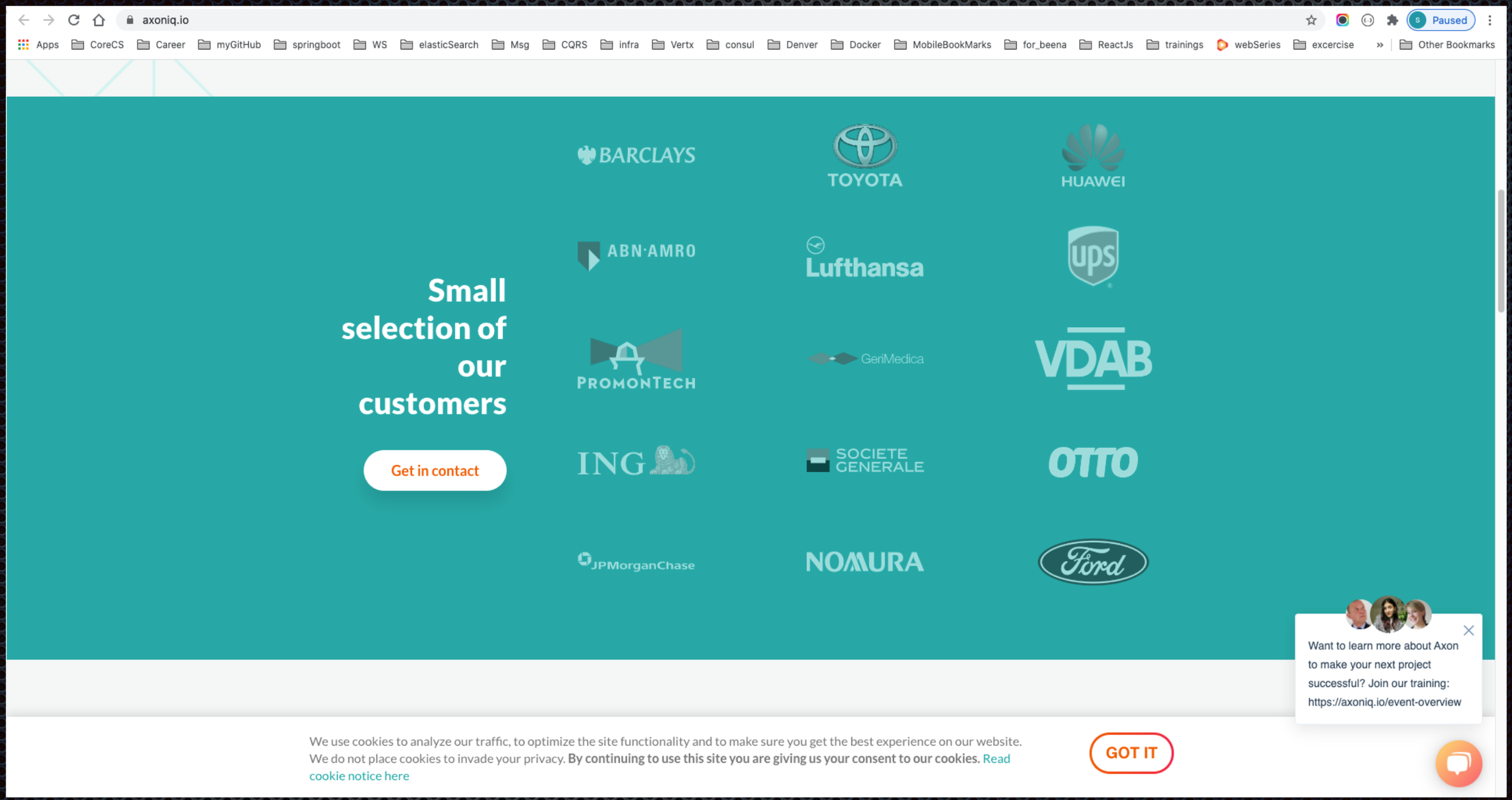
- **Solves data consistency issue in Micro-services**
- **Better Performance and Scalability of your system**
- **No Loss of Data**
- **Regulatory Compliance**
- **Audit trail for free**
- **Data Mining / Analytics**
- **Design Flexibility**
- **Temporal Reporting**
- **Flexibility – many different domain models can be constructed from the same stream of events.**
- **Ability To Add New Features Easily**
- **No deadlocks.We can get object states at any point in time.**
- **Easy to test and debug.**
- **Data model is decoupled from the domain model.No impedance mismatch (object model vs. data model).**
- **We can use this model with reversing events, retroactive events.**
- **No more ORMs – thanks to the fact that our object is built from events, we do not have to reflect it in a relational database.**

CQRS And ES

Cons :

- **Eventual consistency (But Remember CAP Theorem)**
- **Weird and unfamiliar style of programing**
- **Events = a historical record of bad design decisions**
- **Must detect and ignore duplicate events**
  - **idempotent event handlers**
  - **Track most recent event and ignore older ones**
  - **...**
- **Querying the event store can be challenging**
- **Maintenance and administration costs if you choose two different engines for the read and write sides**
- **Event schema changes is much harder than in case of relational model (lack of standard schema migration tools)**

AxonIQ In Production



Summary

- Aggregates are the building blocks of microservices
- Use events to maintain consistency between aggregates
- Event sourcing is a good way to implement a event-driven architecture

Does this fit to our ecosystem ?

Questions ?

References

- https://microservices.io/
- https://axoniq.io/
- https://axoniq.io/blog-overview/axon-and-kafka-how-does-axon-compare-to-apache-kafka

Thank You !