HACK 4 FOOD

DDD com CQRS e Event Sourcing

Agenda

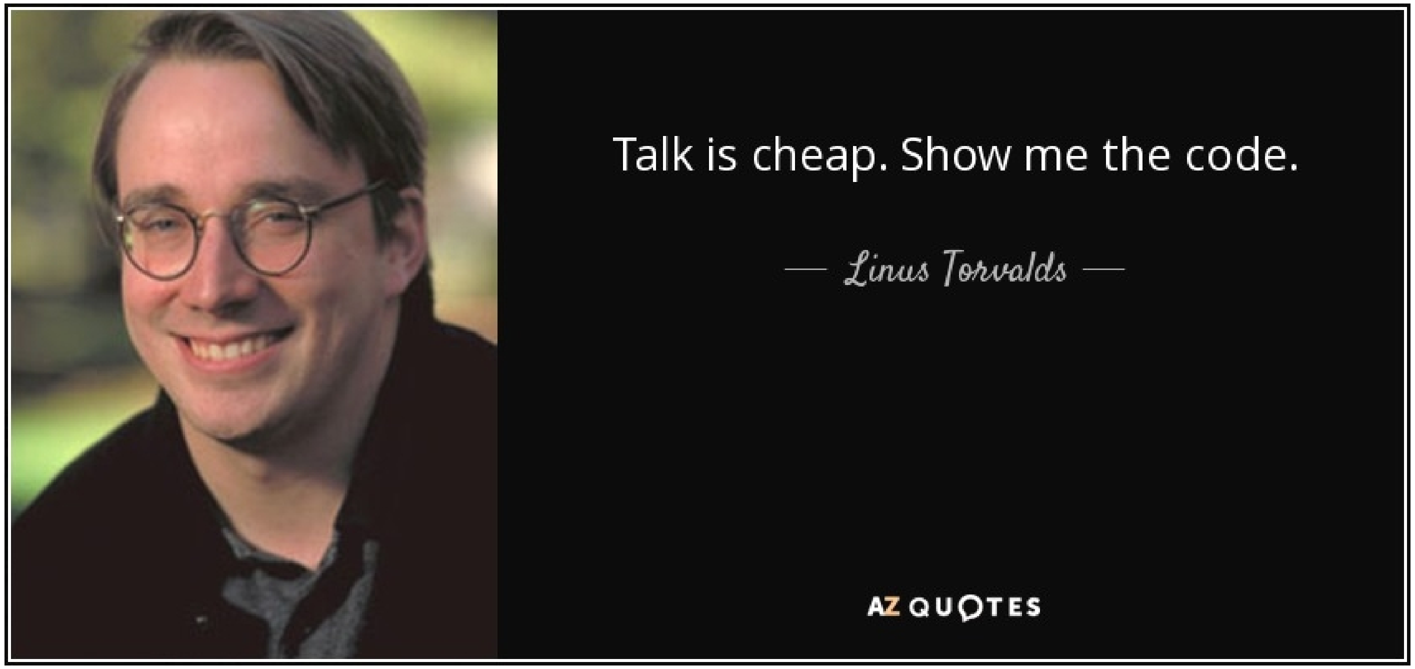
- DDD
- Event / Event Sourcing
- CQRS
- Framework Axon 3.0.4

Domain Driven Design

- Warning - Dogma Driven Design
  - *Dogma is an established belief or doctrine of a religion, ideology or any type of organization, considered a fundamental and indisputable point of a belief.*

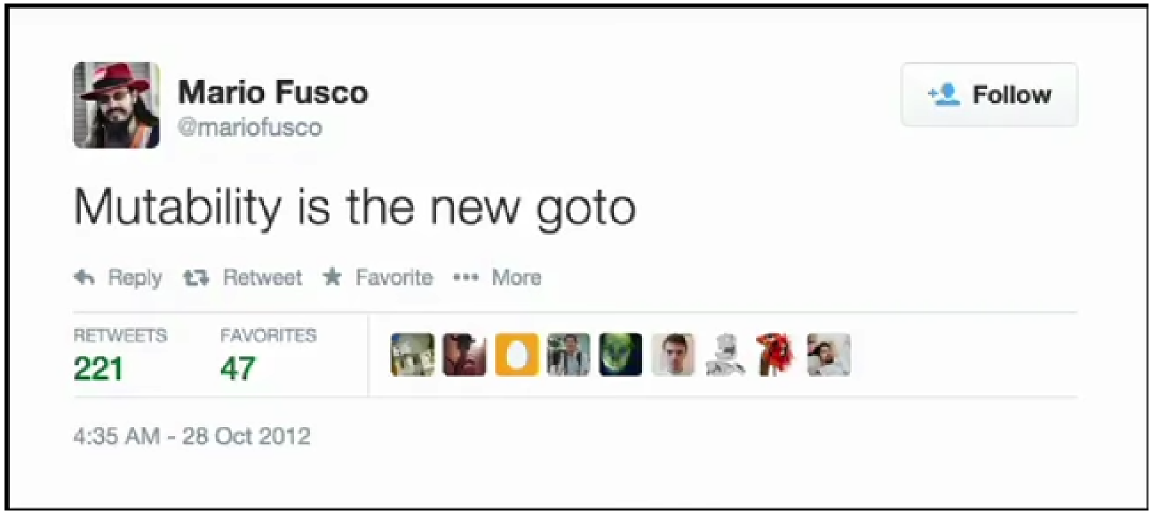
Domain Driven Design

- Why DDD?



Domain Driven Design

- Mutability EVERYWHERE!!!



Mario Fusco
@mariofusco

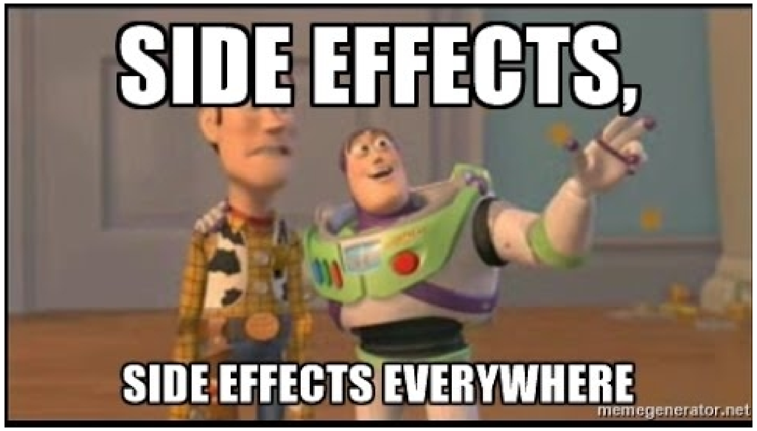
Mutability is the new goto

RETWEETS 221
FAVORITES 47

4:35 AM - 28 Oct 2012

Domain Driven Design

- No Type Save – Primite Obsession
  - *Strings, Integers, Floats and more Strings*
  - *Invalids Objects / Wrong meaning*



Domain Driven Design

- Anemic Domain Model

...is the use of a software domain model where the domain objects contain little or no business logic (validations, calculations, business rules etc.).

getters / setters / builders

Domain Driven Design

- Anemic Domain Model leads to ...

where Services and Commands Layers are OVER ABUSED!!!

Domain Driven Design

- *Where‘s the OO?*

Domain Driven Design

- How did we get here?
  - Bad Frameworks and Technology
  - J2EE, EJB
  - Frameworks Javabeans
  - IDEs

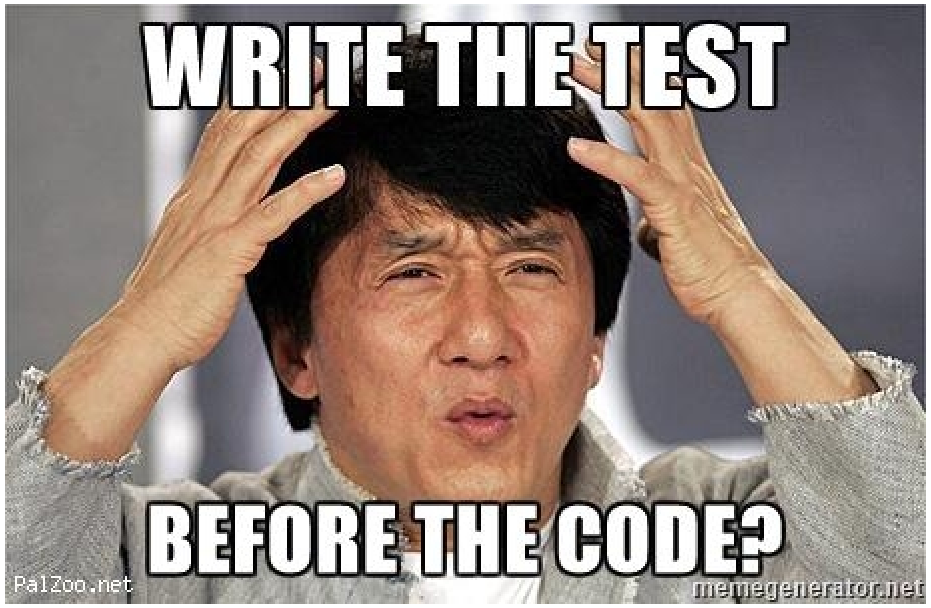
  Your best Seniors are focused on fix Technology problems and limitations.

Domain Driven Design

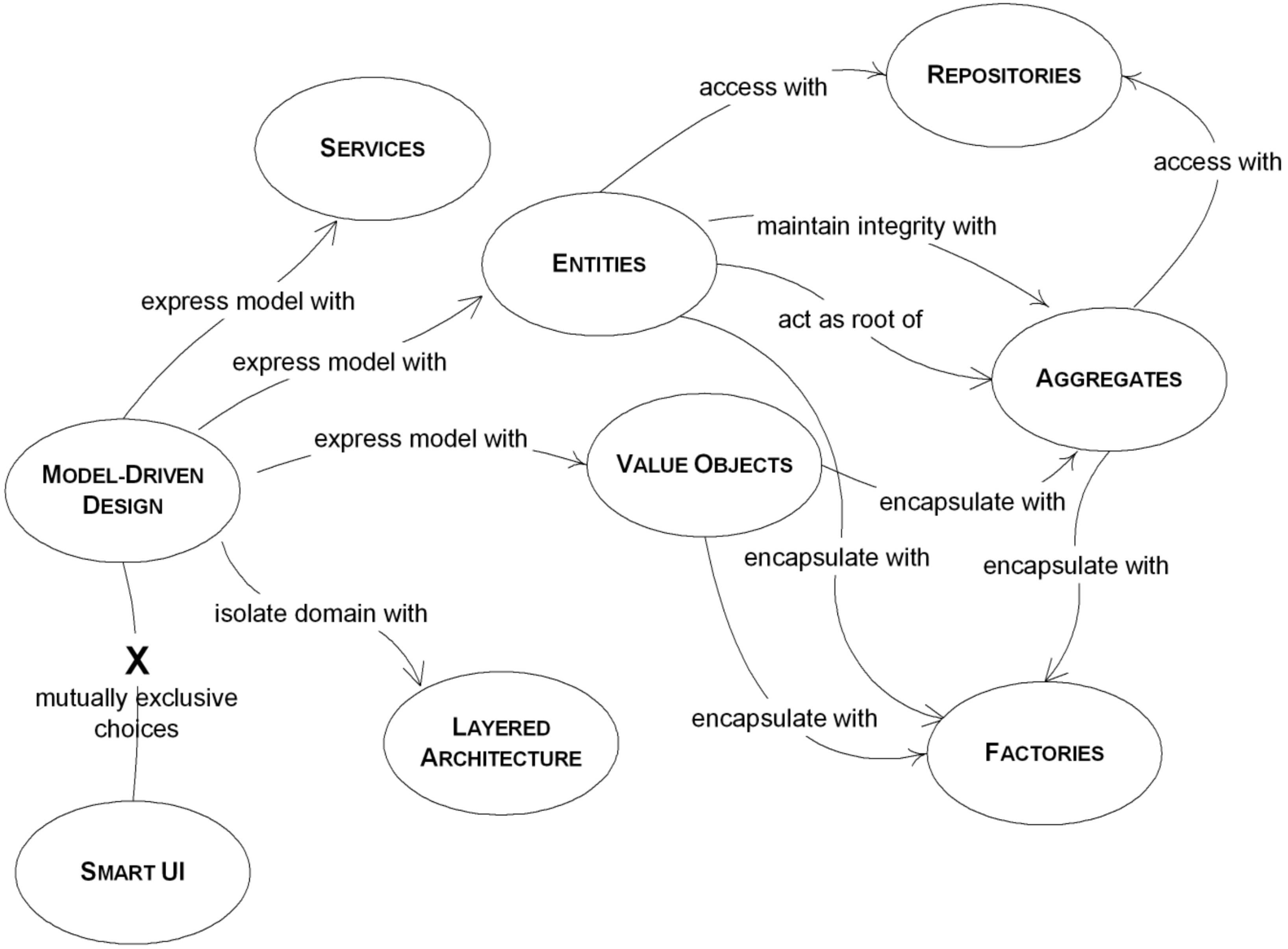
- DDD Concepts:
  - No technology envolved
  - Ubiquitous Language
    - Express your Domain business actions in your code
    - avoid: save(…), update(…), execute(…)

Domain Driven Design

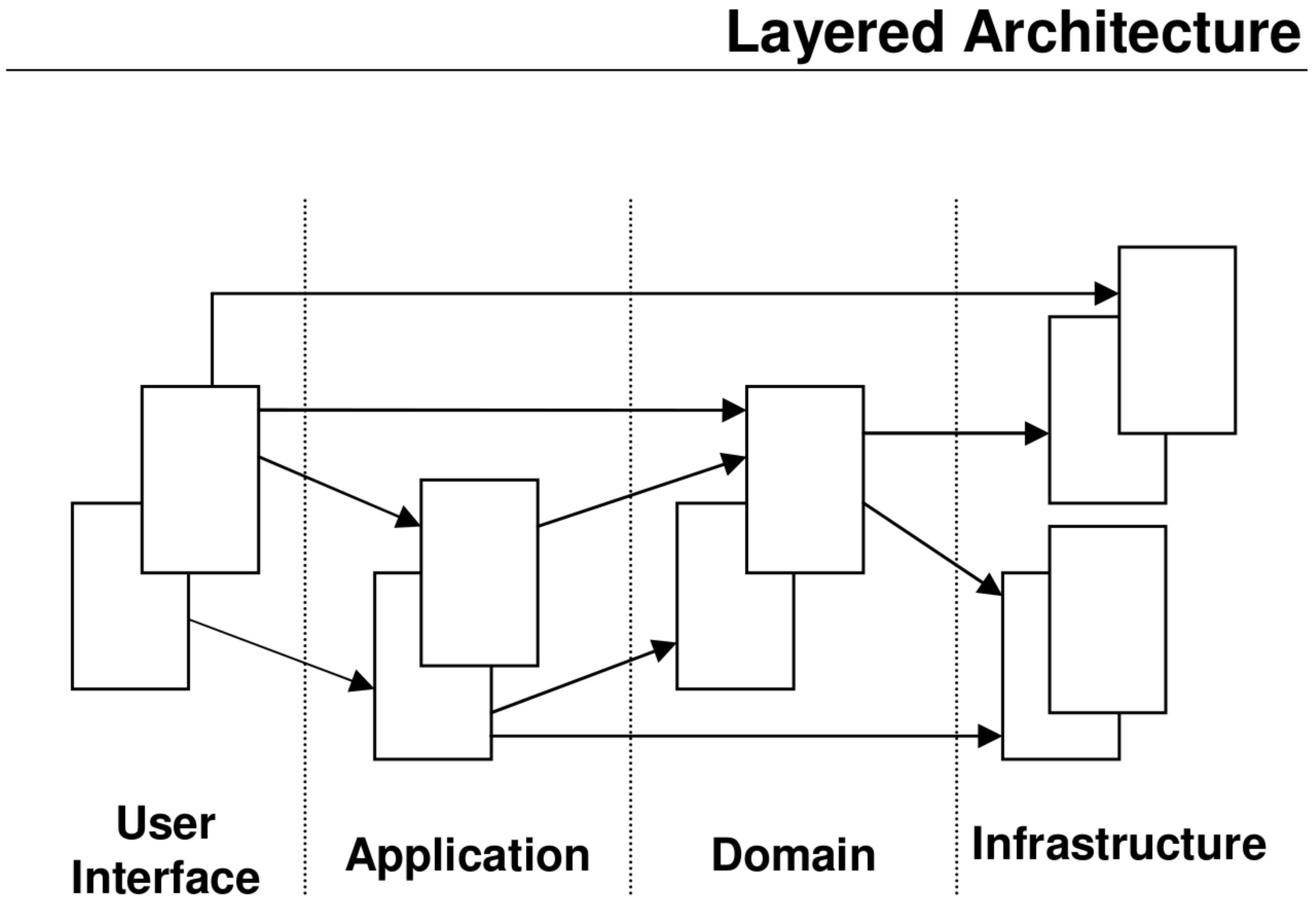
- DDD Concepts:
  - Focus on DOMAIN, not on Services
    - Come back to OO
  - Tests



Domain Driven Design *– by Eric Evans 2005*



Domain Driven Design *– by Eric Evans 2005*



Domain Driven Design

- Entity
  - *If objects have different attribute values but same identity value they are equals*
  - *Mutable, Persistent ¨somehow¨*
  - *EqualsHashCode: Identifier*

Domain Driven Design

- Value Objects
  - *Value Objects should represent concepts in your Ubiquitous Language, and a domain expert should be able to recognize it*
  - *Immutable*
  - *EqualsHashCode: ALL Attributes*
  - *Not just getters/setters. They can and probably will have business logic*

Domain Driven Design

- Aggregates and Aggregate Root
  - *Aggregates draw a boundary around one or more Entities*
  - *is a cluster of domain objects that can be treated as a single unit*
  - *Any references from outside the aggregate should only go to the Aggregate Root*
  - *More than one Aggregate for the same Model*

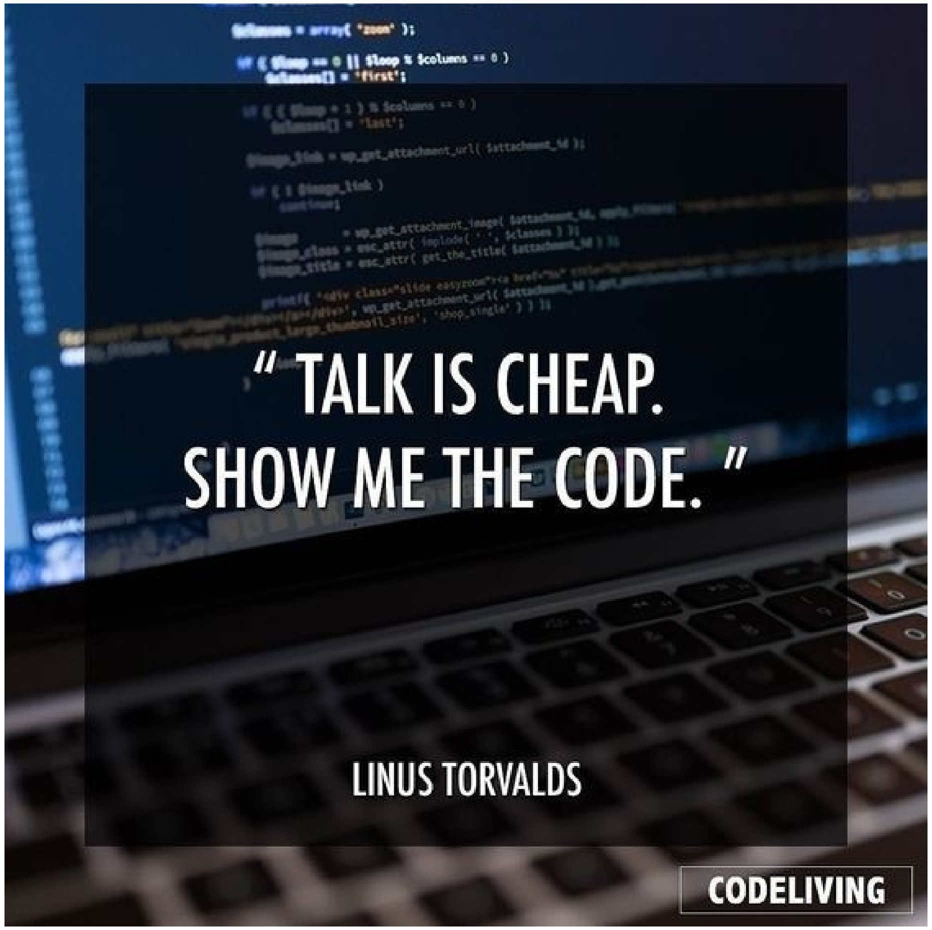
Domain Driven Design

- *Services*
  - *A Service should not replace the operation which normally belongs on domain objects*
  - *The operation performed refers to a domain concept which does not naturally belong to an Entity or Value Object*
  - *The operation performed refers to other objects in the domain*
  - *The operation is stateless*
  - *Application Service and Domain Service*

Domain Driven Design

- Repository
  - *All data access must be performed though a Repository*
  - *Interfaces - aggregates only knows the Interface*
- Factories
  - *helps encapsulate the process of complex object creation*
  - *GOF – Design Patterns – Abstract Factory e Factory Method*

Domain Driven Design



Aggregates and Events

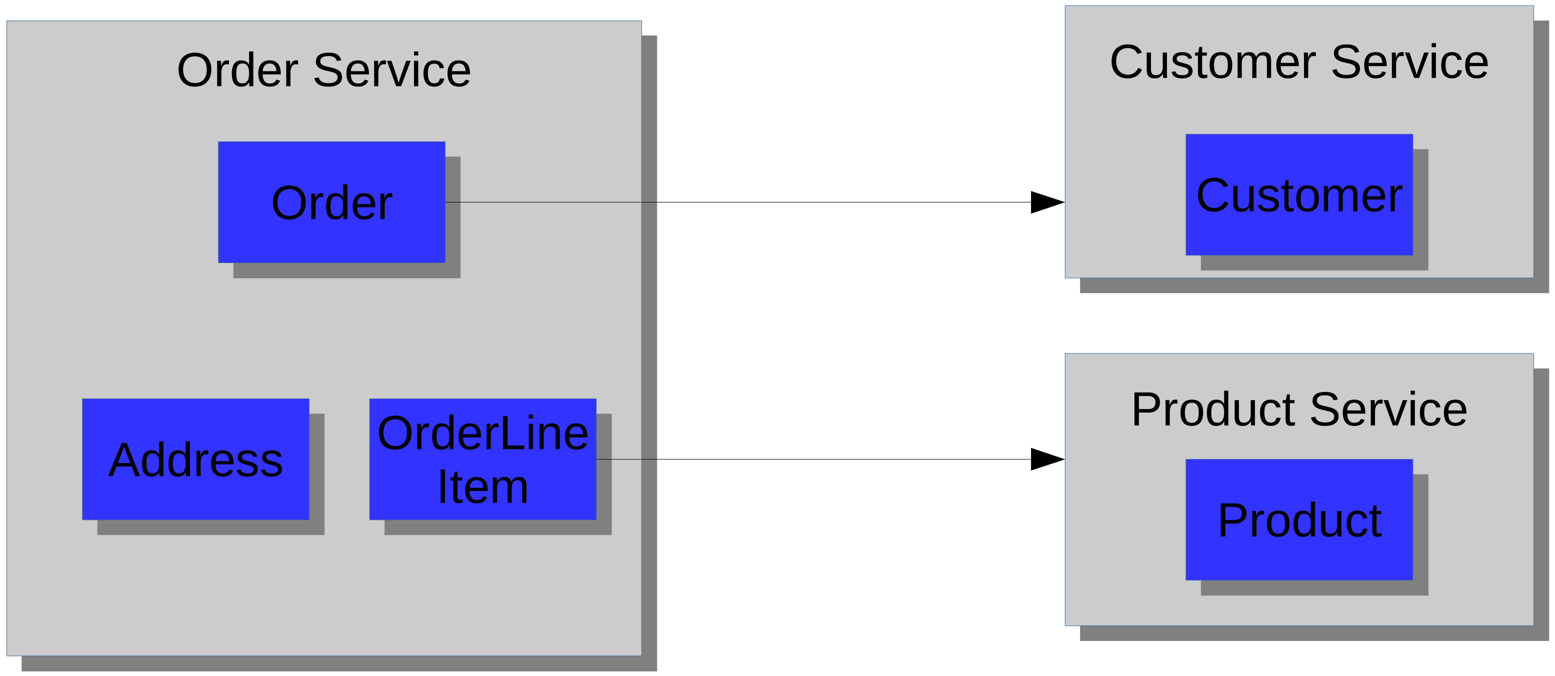
- Monolithic Applications
  - Begin Transaction

    ```
    aggregate1.doSomething1()
    aggregate2.doSomething2()
    aggregate3.doSomething3()
    ```

  - Commit/Rollback Transaction

Aggregates and Events

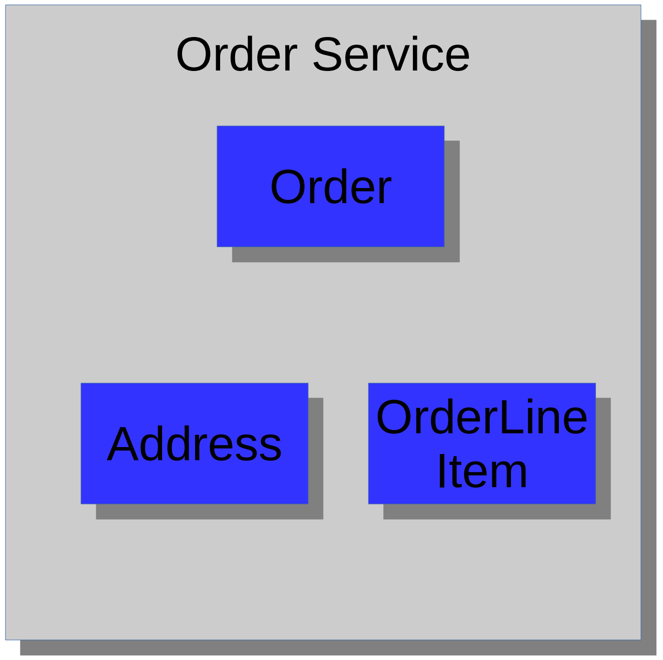
- Aggregates concept creates clear boundaries
  - Loosely connected – Identifier only, not Object Reference
  - Modularity / Microservices



Aggregates and Events

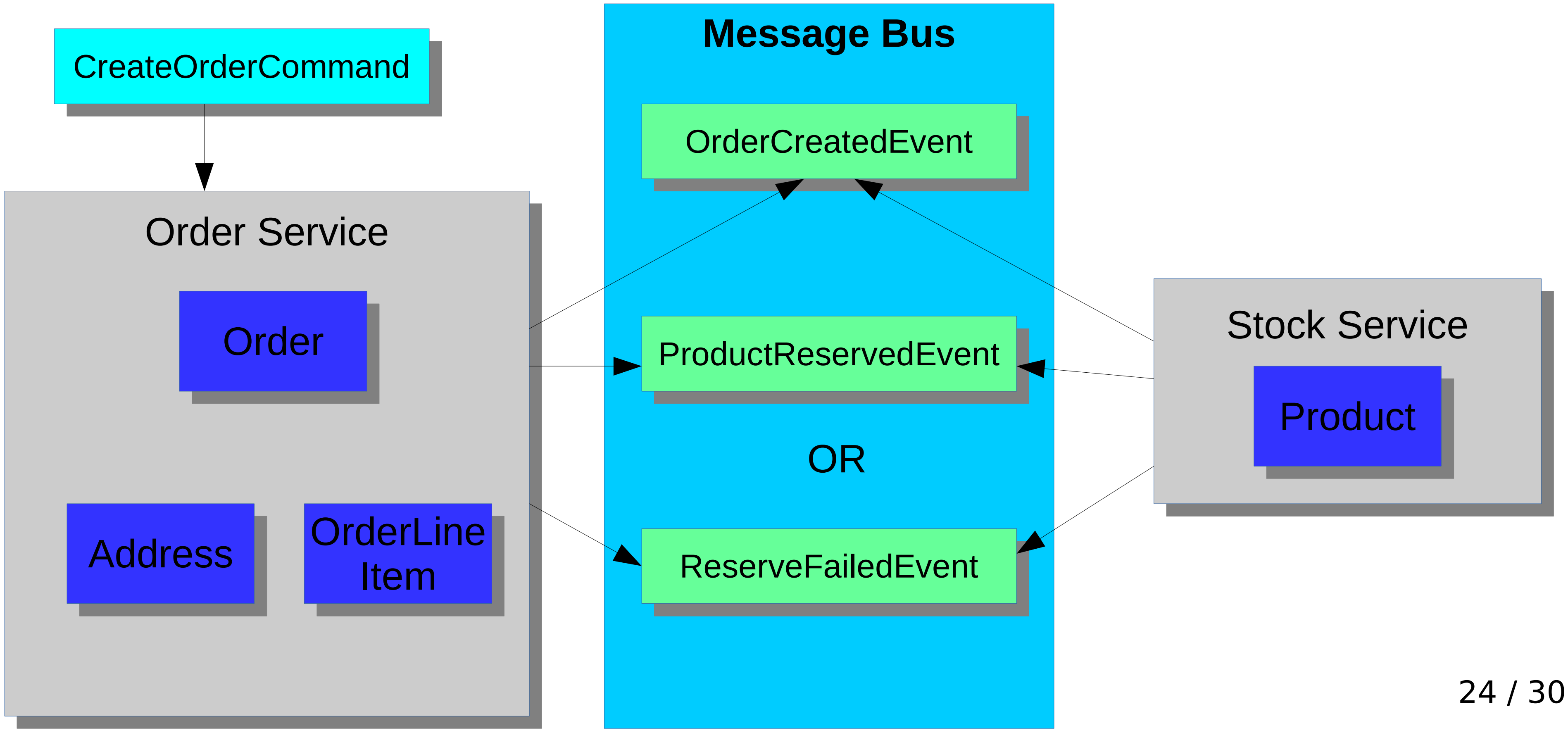
- One Command by Aggregate – keep the boundaries

  - CreateOrderCommand
  - AddItemCommand
  - CancelOrderCommand



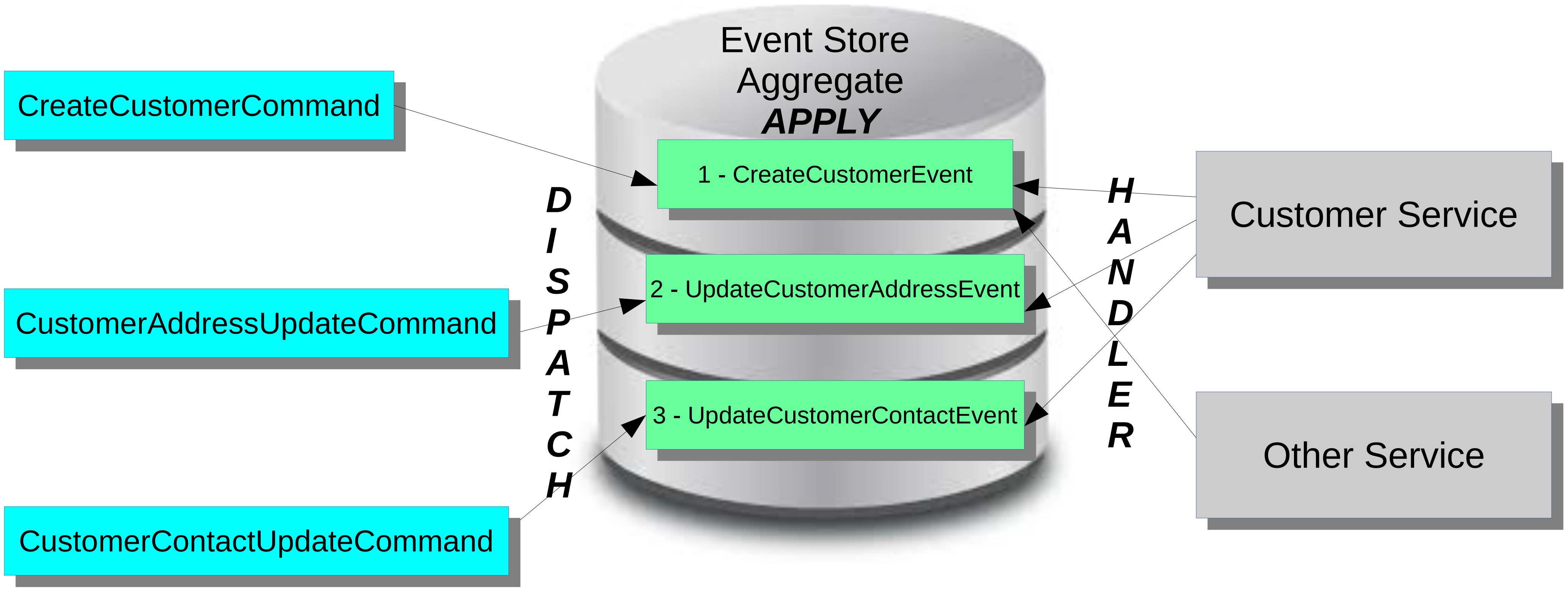
Aggregates and Events

- Event-driven architecture
- Events are <u>Immutable</u>
- Events stream is just a LeftFold concept in Functional Programming



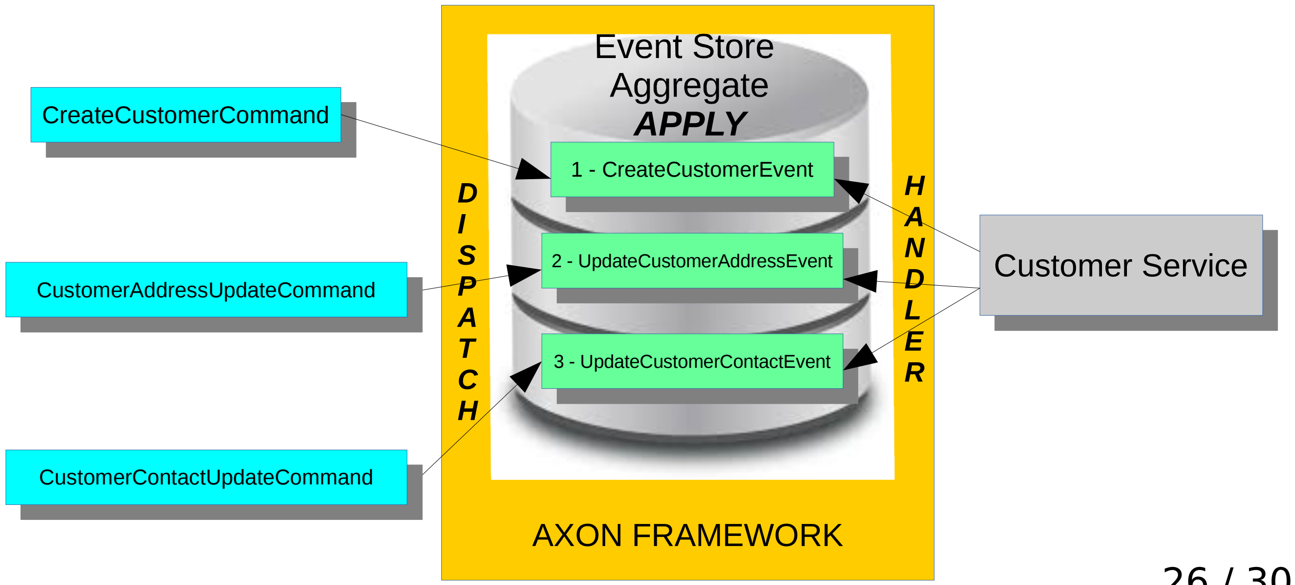
Event Sourcing

- Event Sourcing is an architectural pattern in which the state of the application is being determined by a sequence of events


CreateCustomerCommand
CustomerAddressUpdateCommand
CustomerContactUpdateCommand
DISPATCH
Event Store
Aggregate
APPLY
1 - CreateCustomerEvent
2 - UpdateCustomerAddressEvent
3 - UpdateCustomerContactEvent
HANDLER
Customer Service
Other Service

Event Sourcing

- *Event Store*
- *Audit Log / Immutable*
- *Replays*
- *New queries from the begin*
- *Sagas*
- *Event Snapshots*
- *<u>Storage cost?</u>*


Event Store
Aggregate
APPLY
CreateCustomerCommand
CustomerAddressUpdateCommand
CustomerContactUpdateCommand
DISPATCH
1 - CreateCustomerEvent
2 - UpdateCustomerAddressEvent
3 - UpdateCustomerContactEvent
HANDLER
Customer Service
AXON FRAMEWORK

CQRS - Command Query Responsibility Segregation

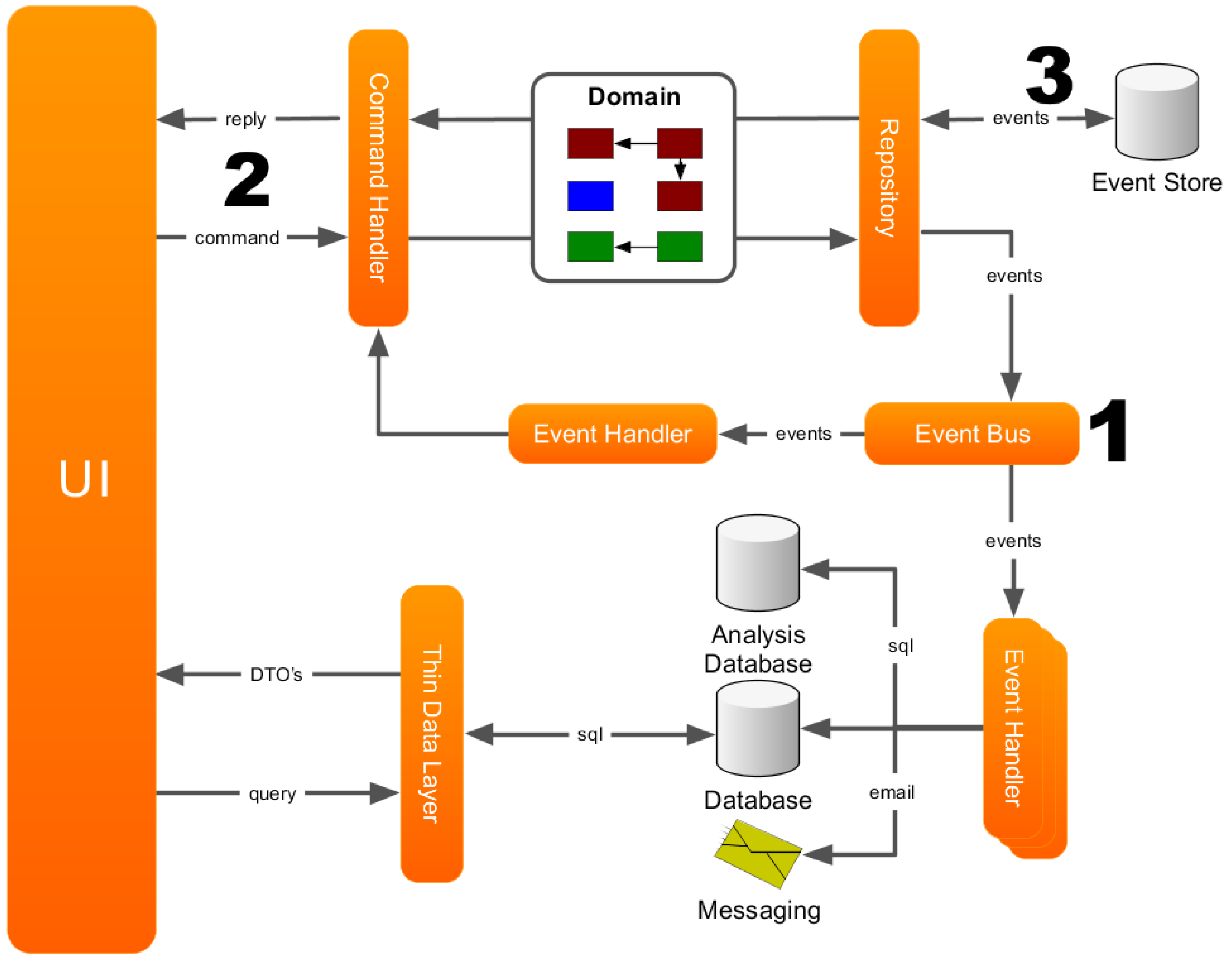
It's a pattern that I first heard described by Greg Young. At its heart is the notion that you can use a different model to update information than the model you use to read information. For some situations, this separation can be valuable, but beware that for most systems CQRS adds risky complexity.

Martin Fowler – 14 July 2011

CQRS - Command Query Responsibility Segregation

- *Can exist without Event Sourcing*
- *CQRS is a Pattern and not a entire architecture*
- *A method should either change state of an object, or return a result, but not both.*
  - *Commands: Change the state of a system but do not return a value*
  - *Queries: Return a result and do not change the observable state of the system (are free of side effects)*

CQRS - Command Query Responsibility Segregation



CQRS - Axon, Springboot demo