Command Query Responsibility Segregation (CQRS) and Event Sourcing design patterns

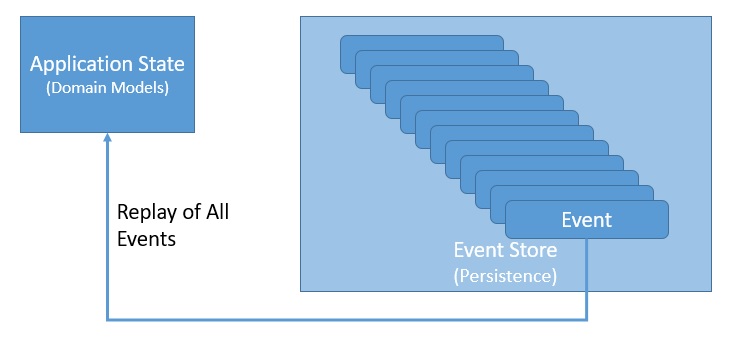
[CQRS and Event Sourcing in Java | Baeldung](https://www.baeldung.com/cqrs-event-sourcing-java)

This pattern is used together in Enterprise application

Event Sourcing :

It gives the new way of persisting the application state as an ordered sequence of events

Selectively query these events and reconstruct the state of application at any point of time .



Replay events selectively , replay some events in reverse with event log as our primary source of truth.

CQRS – segregating the command and query side of application architecture.

CQRS is based on Command query separation (CQS)

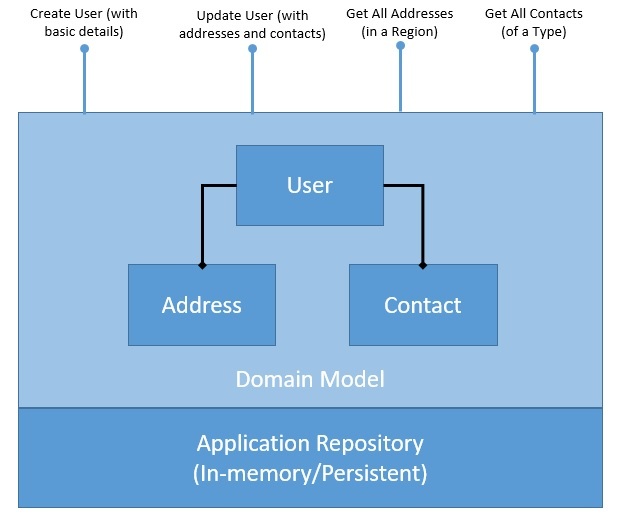
CQS suggests that divide the operations on domain objects into 2 categories – Queries and commands

Queries return a result and do not change the observable state of the system

Commands change the state of the system but do not necessarily return a value

A Simple Application :

DDD addresses the analysis and design of software that relies on complex domain – specific knowledge .

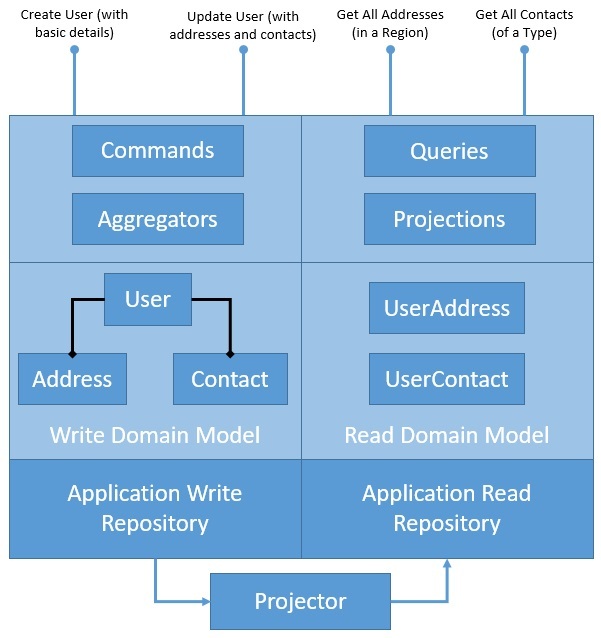


Here the persisitance can be in-memory database or the user database.

Problem in the normal CRUD application is the Domain Model and Persistence

**Introducing CQRS**

We'll begin addressing the first problem we discussed in the last section by introducing the CQRS pattern in our application. As part of this, **we'll separate the domain model and its persistence to handle write and read operations**. Let's see how CQRS pattern restructures our application:

[](https://www.baeldung.com/wp-content/uploads/2020/05/CQRS-in-Application-3.jpg)

The diagram here explains how we intend to cleanly separate our application architecture to write and read sides. However, we have introduced quite a few new components here that we must understand better. Please note that these are not strictly related to CQRS, but CQRS greatly benefits from them:

* *Aggregate/Aggregator*:

Aggregate is **a pattern described in Domain-Driven Design (DDD) that logically groups different entities by binding entities to an aggregate root**. The aggregate pattern provides transactional consistency between the entities.

CQRS naturally benefits from the aggregate pattern, which groups the write domain model, providing transactional guarantees. Aggregates normally hold a cached state for better performance but can work perfectly without it.

* *Projection/Projector*:

Projection is another important pattern which greatly benefits CQRS. Projection **essentially means representing domain objects in different shapes and structures**.

These projections of original data are read-only and highly optimized to provide an enhanced read experience. We may again decide to cache projections for better performance, but that's not a necessity.