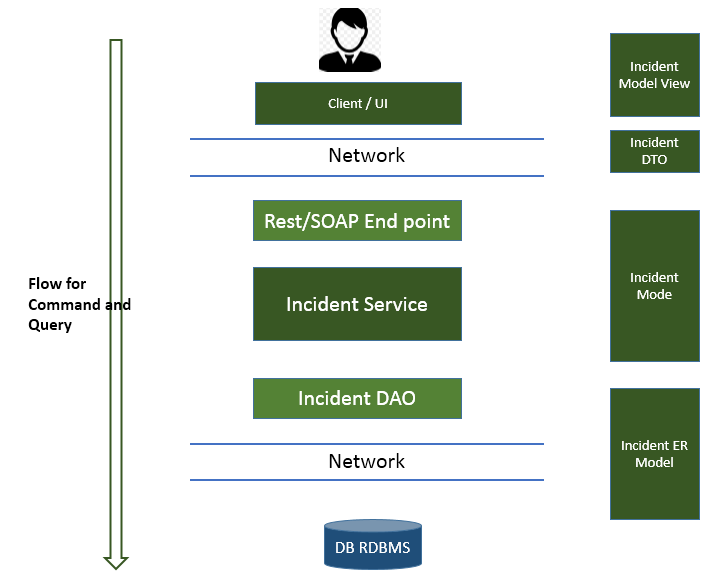
# Micro-service design

## Command Query Responsibility Separation (CQRS) pattern

Here we move from traditional approach of using the same architecture for writing (command) and reading (query) towards event sourcing and why we should separate the read and write and benefits of achieving it and also the challenges.

### Evolving towards event sourcing

Tradition approach of read and write in a typical monolithic application



### Observations:

Below could be some drawbacks to the above architecture

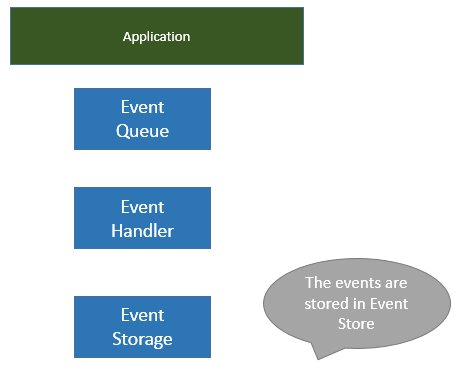
1. We read and write data from same layer/flow.
2. Same model, business service or DAO/DTOs are used for read and write (which might not be always fit to)
3. We use the same deployment model. I.e. reader and writer deployed at the same time.
4. We use the same data store to read and write data
5. We always get the final state of data (after a multiple updates). But we will never know how we reached there. Or what was the state of that particular record on some past business date. (Because we don’t have transaction log always).
6. We can’t scale read and write independently. If read needs more resource that write.
7. No data history or snapshot.
8. Because of using the same data base we need to do read and write in the same data object. Where to improve read performance we may put multiple indexes on database tables where the same index will kill the write performance.
9. This approach tends to create huge monolithic architecture.
10. When we write we write single entity but when we read we read list of entities.

One architecture which approaches these kind of challenges is **Event Sourcing.**

## Event Sourcing

Event sourcing is an architectural pattern where the state of and application is being determined by **series of events**. State is computed by series of events.

### Building blocks of event sourcing



The sequence of events in the event queue is called event stream.

Names of the event should be very expressive and should be in **past tense** as these happened in the past.

Ex:

* ShipmentDeliveredEvent
* CartCheckoutEvent
* CustomerVerifiedEvent

Event should be always **immutable.** We never **delete** event from **event store.**

Usually the event bus or event queue is implemented by a **message broker** architecture. (Kafka, Active MQ, Rabbit MQ).

Well known examples for event sourced systems are:

* Version control systems
* Database transaction logs

The event store has a very high business value, because it consists the complete state of our data from beginning. How it got changed. This could be used for analytics.

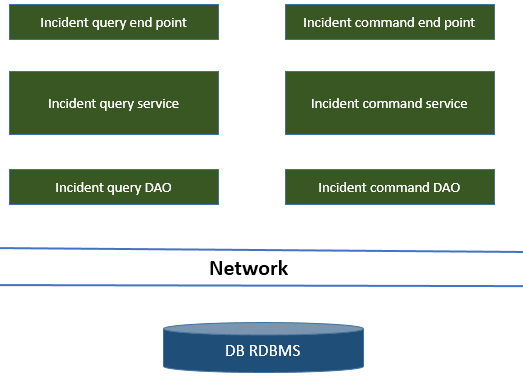
### Performance problems

If we do temporary queries on event store to determine the state of the application we may run into performance problems.

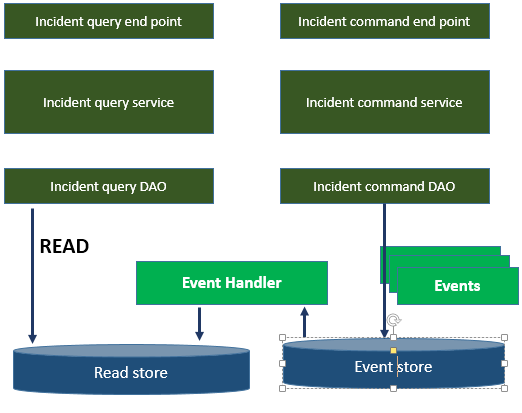
We can pre-compute some application state. A way to create pre-processed application state is **CQRS pattern.**

## First refactoring step towards CQRS event sourcing

In our previous application design we can separate the read and write steps as below



Here the issue we still have is we are running against same data storage, so in above we don’t get too much advantage and insufficient. So we shouldn’t step there. We should design as below,



**Read Store** could be highly optimized, aggregated from performance optimization. Could be a non-relational data base.

### Benefits with above model

We can individually scale read and write based on need. Or auto scale at a particular time of a business day when write is more or read is more.

We have high degree of technology choice freedom to design read and write. But don’t make it a **technology** **soup** unnecessarily. Else management will be very difficult down the line in long run.

This kind of architecture very fit for **Bounded Context** in **Domain Driven design**.Always let the business domain drive your software architecture not the technology.

### Challenges in event sourcing architecture

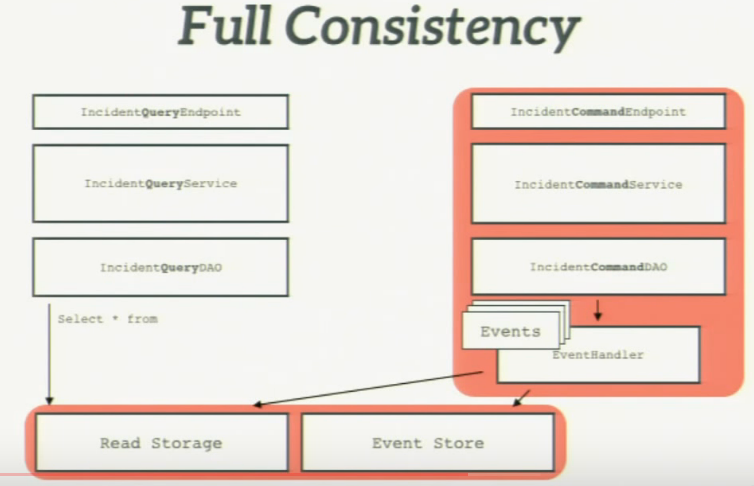
* Consistency
* Validation (on aggregation from multiple domain)
* Parallel updates, parallel processing of events

### Consistency

Based on **CAP** theorem, system either could be highly consistent or highly available at a particular point of time but not both.

Below design could make the system highly consistent. If we make read and event/write store in a relational database in same node and can update to event store and read store under a single transactional boundary.

But this kind of system will not **scale.**



**Increased but eventual consistency.**

Here we can make the inserts into write store extremely transactional, But read store will be updated asynchronously. Which may contain stale data to read for certain amount of time

