CQRS - The way to go if you want scalability, performance and to sound cool

Agenda

* Why CQRS & Event Sourcing?
* CQRS in a nutshell
* Event sourcing in a nutshell
* Proof of concept

**Why CQRS and Event Sourcing?**

Generally speaking, CQRS & Event Sourcing is a symbiosis of best practices and well known patterns. This architectural approach is deeply rooted into a bunch of DDD patterns which turned out to be efficient and valuable over time.

With the rise in popularity of microservice architecture, developers and architects started to consider CQRS/ES as it offers the luxury of fast and continuous feature delivery by keeping the components loosely coupled and a higher scalability by making operations as atomic as possible.

In the following I'll try to bring you the value of both CQRS and Event Sourcing, especially when combined, and will tell you more about why it might or might not be a great architectural decision for an enterprise project, even for the one that you're currently working on. I'll also highlight it's disadvantages and drawbacks in order to deliver the idea that it is not a panacea  and does not suit any type of project. However, there are some cases when this architectural decision is the magic bullet.

**CQRS in a nutshell**

CQRS is the acronym for Command Query Responsibility Segregation.  *Are you swill with me?*   Even if it sounds like it was taken from straight from Elon's Falcon 9 design sheets, in essence it's just the separation of the reads and writes. It is based upon the CQS principle- Command Query Separation, where each operation (usually referring to CRUD) is defined as either command or query:

Commands are those operations that change state. For instance, create, update and delete operations are considered to be commands because they change the state of the system. Ideally, a command should not return anything, but people usually break this rule and come up with an unbeatable argument: *Even Microsoft violates this rule with their Stack.Pop() method*.

On the other hand, queries *surprise surprise*do not alter the state and has a return value. Usually, read operations are considered queries. Note that in some cases, if the read goes along with persisting some data, then it could be considered a command.

​​​​​​​It is a query when it does not alter the state and has a return value. Usually, read operations take form of a query. Note that in some cases, if the read goes along with persisting some data, then it could be considered a command.

Note that commands can call other commands and queries, whilst queries can call only other queries. If a query calls a command - it's not a query anymore because it mutates the state.   
  
Now the question arrives: why to separate them?   
  
The code becomes simpler to understand and to work with when this separation is enforced. Also, as little bonus to that, it allows us to be confident that a query can be called millions of times and it won't break or modify anything.

The idea with CQRS is to separate the paths for operations that change the system from those that simply request data and to have them separated into different conceptual models for update and display, instead of having a single model that handles both of them and none of them properly.

Imagine how simple would it get to maintain a system composed of simple and separated queries and commands instead of having bunch of thousand-liners and god classes. If a query or command gets more complex or needs an update, the developer will have to change and test afterwards only that itsy-bitsy part of the project.

Example: We have a traditional incident tracker web application where employees can report technical problems. When an employee reports an issue (aka creates an incident record), the request goes through API Controller - IncidentService - IncidentRepository. The same path is triggered when the administrator issues a get request in order to view all incidents. Note that the queries and commands in this case are coupled.

Following CQRS principles, the easiest way to decouple the reads from writes would be to take that shinny axe and to cut the IncidentService into two services: IncidentCommandService (responsible for creating, updating and removing incidents) and IncidentQueryService, responsible only for getting the information about existing records.

A more elegant solution to this problem would be to have separate commands and queries, each of these having their own handler and using the mediator pattern to correlate the command with it's handler. Thus, instead of IncidentCommandService, we can end up with more granular commands and queries: CreateIncidentCommand - CreateIncidentCommandHandler, UpdateIncidentCommand - UpdateIncidentCommandHandler, GetIncidentsQuery - GetIncidentsQueryHandler and so on. The command or query itself serves for showing the intention (create/update/delete) and delivering the data, and the handler contains the logic itself for handling that specific operation. E.g: CreateIncidentCommand encapsulates the incident's reason, author and description, while the CreateIncidentCommandHandler builds the incident model from the data taken from the command and calls the repository in order to persist the model.

CQRS does not bring that many value to an incident-tracker application because normally, such applications are not that overloaded and it's not needed to make heavy weather of it, but if we replace it with the Google's Youtube application, where the most of the users are consumers, and only a few - creators. It this case, scaling the reads is crucial from the business perspective (btw, Google indeed uses CQRS and Event Sourcing).

CQRS scheme - placeholder

Starting with the idea that queries perform best if the data is stored in a rich schema format using a NoSQL database with great support for queries, having two separate databases for reads and writes starts to sound tempting, isn't it? Guess what, CQRS allows that. Moreover, this cunning is generally accepted and widely adopted.

If a system uses a relational database with complex dependencies and relational schemas, where each get request results in a tremendous SQL query with bunch of joins, then it might be worth to add one more non-relational database with the data in a digestible form, with the joins already applied, and to keep them in sync. Note that this approach adds a bit of complexity because of the database synchronization mechanism, which means that the pros and cons have to be balanced first before deciding whether to implement this separation or not.

**Event Sourcing in a nutshell**

If you are not aware of Event Sourcing pattern (ES) at the moment of reading this, then here is a disclaimer for you: At the beginning it will sound a bit strange, unusual, and you'll most probably be something like excuse mee? Why?? But bear with me, the things will get clear and you'll recognize the true value of ES as we progress throughout this section.

Event sourcing is nothing more than an alternative to the traditional stateful system.

In a stateful system, we always store the last state of something. E.g: If an incident record is created, it is inserted into the database. If it gets updated afterwards, we find the record and update it. If we update it the second time, it goes thorough the same process. If we delete it, it's wiped out from our data store. Note that we always have to deal with the last state of the incident. Once we update it, the original state is lost for good (unless we have some kind of an audit mechanism in place).

For many use cases, a stateful system is more than enough. For others, it simply can not fulfill all the requirements by itself, and I'm referring here to those cases where the history matters or when an entity is represented better via a sequence of event rather than its final state. This problem can be resolved either via a stateful system + an auditing mechanism, or with the Event Sourcing technique with is a symbiosis of those two. Kind of.

The fundamental idea of ES is ensuring that every change to the state of an application is captured in an event object, therefore offering history tracking out of the box and allowing traversing to the particular state of the system to any point of its lifetime. The unique source of truth for an entity's state or shape is its own chronology of events. It is similar to how git version control system is designed, where each commit is an event, and the project's final state is defined by the chronology of its commits.

image placeholder - create update update

*So you're trying to say that if I want to get the current state of an entity, I have to go thorough all of its events and to recreate it from scratch every time?*

Well, yes and no.

Yes because indeed, this way you'll 100% get the precise entity's state, as it is the most reliable source of truth,

No because an additional data store can be added which will hold the final entities' state, often referred to as the Read Store *(do you feel the smell of CQRS in the air?)*. That data store should be used exclusively for reading purposes, and the system is responsible for keeping the data consistent and to update it every time an event occurs. An extremely important principle of ES is that events can only be created, but never updated or deleted.

Also, you can create snapshots (capture object's state) from time to time if this is what business requires.

Advantage

* Much easier to migrate data
* Data does not get lost
* Flexibility or designing read models
* Observability
* Asynchronous first

Disadvantages

* Might be slower (can be solved with snapshots)
* Multiple points of failure (ex. Sync)
* Eventual consistency (create then get => get might be null because data is still processing)
* Changing event model data structure is a headache (or is it?)

**What is an event?**

In the context of Event Sourcing, an event is something that happened in the past or, in other word, a small and granular transition from state A to state B, and there are some rules that apply to them:

* Events should be named in the past tense (IncidentCreated, IncidentRemoved)
* Events are immutable
* Events are broadcasted, thus they do not have a specified listener or handler.

Due to the fact that events are broadcasted, we can hook several subscribers as the project evolves and to treat the event itself a notification to perform follow up actions. E.g: Notify the incident author when IncidentResolved event occurs. It enables to divide the business logic into smaller chunks and tiny classes, which is nice.

Usually, an event has the following structure:

* Id (events have to be uniquely identified)
* Type or Name (identified the event's intent, like IncidentCreated)
* Timestamp
* Stream Id (will get to this later)
* Stream position (Also referred to as version)
* Data (the data carried by the event, usually in JSON or XML format)
* Metadata (optional, but might come handy in more complex systems. It stored information that is not directly related to the event, such as correlation ID or other data used for logging, tracking etc)

image placeholder - event

In order to project an object's state, we have to go through it's events one by one chronologically and to apply their changes, and that's why there is a StreamId attribute. It points to a specific stream or sequence of events referring to a particular domain or domain object which contains the full history of changes.

Keeping in mind that the order is extremely important, the event has to encapsulate the position within the stream it is a part of. The stream position attribute helps to both define the order of events and to detect any concurrency issues and to hook some conflict-resolving mechanism if needed (If an incident is updated at the same time by 2 people, we might end up with 2 event with the same stream position. In this case, based on the position, the system can do one of the following:

* Accept both events (the position count of one of the events has to be incremented)
* Accept the first, reject the second
* Resolve conflicts (usually if both events are of the same type)

Image placeholder - event stream, stream id

**Proof of concept**

Logistify

Business requirements:

Logistify allows both companies and individuals to create shipping inquiries for their goods that have to be moved from point A to point B which, in consequence, can be picked up by other companies or individuals that are available to offer this service.

Any user can either place a or deliver a shipping order.

This platform is going to be used world-wide, therefore the system should be always up and available to display the nearest shipping orders.

Initially, Logistify will operate only in Moldova, but the system has to be design in a way to facilitate further expansion to other countries.

A shipping order's lifetime is composed out of several possible events:

Placing the order - When an user creates the shipping inquiry itself, along with other relevant details about the order such as description, weight, volume etc.

* Updating order details - When the details about the an existing shipping order get updated by the author.
* Canceling the order - When an user decide to cancel his order for any reason. This event occurs when the order was placed but wasn't picked up yet.
* Picking up an order - When a company or individual picks up the previously placed order and engages into delivering it to the destination.
* Abandoning an order - When the order has to be abandoned for some reason.
* Marking the order as delivered - When the company or individual finishes the delivery and the order has arrived to the final destination.
* Returning an order - If the order, which has been marked as delivered, is returned by the recipient to the sender.
* Redirecting an order - If the order was marked as delivered and the recipient sets a new destination.
* Order lost - When an order gets lost along the way
* For the sake of simplicity and to not deviate from the original scope of this proof of concept, the billing of the shipping service is excluded.

Technical details:

Starting from business requirements, there will be way more reads than writes, therefore making CQRS a suitable solution for this. Assuming that the application is intended to go world-wide, the solution should be highly scalable.

Also, due to a shipping order being represented more via its history records rather than it's final state, it is worth to implement the Event Sourcing pattern, but this should be scoped only to order management as user management does not require any history tracking or complex auditing and it can adopt the traditional state-based CRUD model.

The application will be composed out of a single K8S cluster, with an API gateway talking to instances of order query service, order command service and user service, making it possible to scale the reading part via increasing the number of instances if needed.