Some thoughts on using CQRS without Event Sourcing

Marco Bürckel Follow
Aug 22, 2018 · 7 min read

In the last couple of years, I had the opportunity to design one or the other CQRS-based system together with my clients. In these projects, I regularly came across one question: How to build a CQRS-based architecture without adapting the Event Sourcing pattern? Especially when a developer team is new to both CQRS and Event Sourcing, trying to adapt these patterns can be quite challenging.

Both CQRS and Event Sourcing are powerful building blocks in architectural design, but they also add complexity and might not fit every situation. Thus, if you want to build a CQRS-based architecture, it is beneficial to know the alternatives to a persistence based on Event Sourcing.

Some blog articles and posts on Stackoverflow refer to CQRS and Event Sourcing as "orthogonal concepts", which can be applied independently of each other. However, most articles and examples present both concepts baked into one event-based architecture, without discussing any approach to decouple them. Decoupling them is exactly what this article is about.

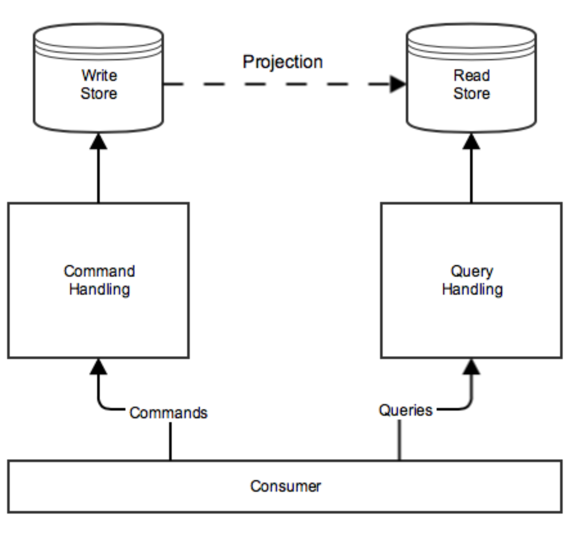
Do you want to use CQRS, but hesitate about the implications of Event Sourcing? Do you wonder how omitting Event Sourcing might affect a CQRS-based architecture? If the answer to this question is "yes", this article might be interesting for you.

The Essence of CQRS

Looking at the core concept of the pattern, CQRS states that an application architecture can be divided into two parts:

- *The write-side*: A part that modifies the application state by executing commands.
- *The read-side*: A part that reads the application state by executing queries.

Both concerns are implemented independently of each other. How far the segregation is taken, depends on the concrete architectural design. One may apply it to the logical layers of a system and let both sides share the same database. But often, the segregation is extended to the database level. In that case, write operations and read operations work on different databases.



Write Operations and Read Operations affecting different Databases

Just like segregating the application in a write-side and a read-side, the persistence layer is then segregated in a *write-store* and a *read-store*:

- The *write-store* is the database that acts as the source of truth and is affected by executing commands. It either contains the current application state or contains all information needed for rebuilding it. Thus, the write store is optimized towards data integrity and maintaining a correct application state. Using the typical RDBMS way of doing things, this could be achieved by data normalization, integrity checks and transactions. Another way of keeping the application state clean is to make modifying it dead simple. This is the approach the Event Sourcing pattern takes: The state is represented as an append-only sequence of events. Existing data is never modified or removed. This makes it hard to mess things up.
- The *read-store* is the database used by queries and is tailored towards the needs of the data consumer. This could include denormalized, aggregated or hierarchical data. The same data could also be stored in a redundant way to serve different representation needs.

Of course, the data stores need to synchronize. Each modification of the write-store should be reflected within the read-store. This step is called the **projection** of data. The projection can be implemented in various ways and greatly affects the application architecture. Especially, the choice over projection is closely related to the decision on using Event Sourcing or not. Thus, the next sections will take a closer look at different implementation strategies.

Event-based Projection

Most CQRS implementations utilize event streams as a trigger for data projection. This is an elegant approach, as events naturally describe application state changes. And most (distributed) systems somehow already process them. Processing an application state change usually involves the following steps:

- A command is instantiated, which represents the action that leads to a certain application state change.
- The command is dispatched to a command handler which validates the command and triggers some domain logic to run.
- The domain logic modifies the application state and emits one or more immutable event objects describing the state change that just happened.
- The events are dispatched to appropriate event handlers which update the read database for each event. Events are usually processed asynchronously, in an eventual consistent manner.

Event-based projection does not necessarily imply an event sourced persistence approach, but there are quite some facts which make it a rather obvious choice:

- **Event-based projection requires persisted events.**
  Projections will not always go well. There might be some database connectivity issue or a bug inside an event handler implementation. Eventually, some projections will fail. Thus, the system needs a way to replay event projection. This requires a persistent event store.
- **Having a write-database other than the event store requires a (distributed) transaction.**
  What if we persist events inside an event store, but additionally store the current application state within a database? For example, could we put the current state of all entities in a relational database while storing the history of events within an additional event store? We could easily load entity states without touching any events and would still be able to replay projections whenever we need.
  The problem is, that both write operations have to be handled as an atomic operation: To ensure data integrity, writing to the database and the event store must either both succeed or both fail. Unless both write stores use the same database technology and reside in the same database, a distributed transaction is needed to guarantee data integrity. While there are solutions to this problem, handling write operations significantly gets more complex. In contrast, when using Event Sourcing, this problem does not exist at all.
- **Multiple write-stores are harder to manage.**
  Using an additional write store next to an event store also increases database management complexity. As the source of truth is formed by both databases, they always have to be synchronized. In this case, this means that both databases must reflect the state where the exact same sequence of commands has been written to each respective database. This gets interesting when defining backup/restore strategies or when it comes to manual data correction.

As you can see, when using event-based projection, it makes a lot of sense to make the persisted sequence of events the only source of truth. This is exactly what Event Sourcing does. It is the choice of event-based projection that make CQRS and Event Sourcing such a great match. And in such a system, CQRS and Event Sourcing can hardly be called "orthogonal concepts", because without Event Sourcing, implementing a reliable event-based projection will become significantly harder.

Top highlight

So in case you want to use CQRS without Event Sourcing, what are the alternatives?
From a more abstract point of view, event-based projection is a projection strategy which is based on small deltas. As any projection needs to be replayable, any such strategy requires persisting these deltas as a source of truth. Thus, an alternative projection strategy would not be based on information about deltas at all, but rather on the current application state.

State-based Projection

I came across the term of "state-based projection" in a blog article written by Vladik Khononov [Tackling Complexity in CQRS] and I think it perfectly describes the alternative type of projection strategies that exist besides a delta-based approach:

Instead of projecting small deltas, a state-based projection works on entities as a whole. There are different implementation strategies:

Database Views

When working with a relational database, the read-side of the CQRS-based system can operate on database views. The projection is then entirely handled by the database itself. Given SQL as a query language, this approach offers a good deal of flexibility regarding data aggregation and transformation. Using a Micro-ORM framework like Dapper, it is easy to implement a thin and lightweight read-side while taking full advantage of the CQRS pattern.
However, using database views, the denormalization is limited when it comes to hierarchical data structures.

Entity-based Projection Handlers

In event-based projection, the projection logic is implemented using a set of event handlers. In a similar way, a state-based projection logic can be implemented using entity-based projection handlers:

```
interface IEntityProjection<TEntity>
{
    void Project(TEntity entity);
}
```

Whenever an entity of type T is changed on the write-side, the projection handlers for type T need to be triggered. In larger systems, this can be done asynchronously in an eventual consistent manner. As Vladik suggests in his aforementioned article, the projection could be triggered by setting either a row-based dirty flag or a row version number that is observed by the projection engine.

Implementing state-based projection does not require events or event handlers. Thus, this approach works well if you want to build a CQRS-based system which does not utilize Event Sourcing. Of course, that does not necessarily mean that there are no events or event handlers in your system design. For example, you still might want to utilize them in scenarios related to communication between system components.

Final Words

In many CQRS-related articles, an event-based projection strategy is assumed without really naming implications or alternatives. However, in my opinion, choosing the right projection strategy is the most important question when it comes to CQRS-based system design. I hope this article helped you to gain some more generic idea about the pattern.

If you have any questions or feedback, just leave me a comment. 😊

Software Architecture | Programming | Cqrs | Event Sourcing | Microservices

WRITTEN BY
Marco Bürckel Follow
Together with my team at axio concept, I help companies build great software products. Our focus is Software Architecture, Cloud Solutions, AI & IoT-Systems.