Whitepaper

The CQRS/ES Playbook

Allard Buijze & Vijay Nair

AxonIQ

Executive Summary

When building software in complex domains, developers are likely to use modern application concepts such as event-driven architecture, functional programming, and microservices to add flexibility and deal with complexity. Occasionally, a traditional n-tier architecture based on CRUD (Create, Read, Update, and Delete data storage APIs) might be suitable for a modern app. However, most of the applications in development now are better served by isolating the read and write models and handling them separately. As a result, CQRS/ES (Command Query Responsibility Separation and Event Sourcing) have gained traction as flexible alternatives to traditional, more rigid architectures.

Inspired by domain-driven design (DDD), CQRS/ES addresses the needs of modern apps with an architectural style that works and makes sense. CQRS is based on the concept that every method should either be a command that performs an action or a query that retrieves a result. Functionally, that means that query methods should be referentially transparent so developers can use them in any part of a system without any contextual knowledge. Because developers can split a CQRS system into separate services that communicate with event messaging, it opens the door to event sourcing. Event sourcing stores each event that happened in a domain. For that reason, it is often associated with AI and machine learning because this method stores more data.

Because CQRS/ES can provide the architecture needed for modern applications and even modular monoliths, it's an essential addition to your development arsenal. Like any new technology, however, for your organization to benefit from CQRS/ES, you need to implement it with care and preparation.

This guide covers the important decisions required across all levels of your organization for a successful implementation of CQRS/ES. Your IT teams will gain a holistic overview of CQRS/ES patterns and a guided pathway for implementation. Whether your organization wants as much information as possible before it moves away from traditional architecture or is ready to go, this paper shows how to optimize your adoption of CQRS/ES and even get a troubled project back on track.

Introduction

"CQRS/ES" (Command Query Responsibility Separation and Event Sourcing) have become frequent topics at conferences and in blog posts, articles, customer case studies, and more. They are likely to appear at internal organizational summits whenever the subject of event-driven architecture modernization comes up.

Of the organizations taking a hard look at the adoption of CQRS/ES, there are two camps. One camp is the "fence-sitters." These organizations are sufficiently intrigued by the promise of this new technology, have read a ton of literature, and have done some proofs-of-concept, but they are a bit hesitant to take the next step. In the other camp are the "fast-movers." These are organizations that have a culture of bringing a new technology, applying it in smaller measures, and then gradually expanding its adoption by implementing a cycle of optimization, such as best practices, lessons learned developer experience, and automation.

Of course, not all technology adoption results in a happily-ever-after story. The adoption of these patterns does sometimes go awry, and the trough of disillusionment (courtesy Gartner) is real. But are the patterns to blame? Probably not! There are likely other factors, such as misinterpretation of these patterns, poor organizational readiness, choice of a wrong tool to help do the job, and so on.

A lot of factors go into ensuring the successful adoption of any technology. This is particularly true when you choose to implement something as fundamental as CQRS/ES, which has an impact on all three main architectural characteristics - structural, operational, and cross-cutting. So, you need to be extra prepared to get it right.

This paper lays out the proper foundational decisions that need to be taken at all levels of the organization to help implement CQRS/ES successfully. Starting off with a holistic overview of these patterns, it offers guidance for organizations that fall in both camps. The fence-sitters will understand what needs to be done for taking that leap of faith. The fast movers will understand what needs to be done to optimize the adoption and even how to get a derailed project back on track to successfully recoup the investment.

Let's begin.

What Exactly is CQRS/ES?

Before we dive into CQRS/ES, here's a bit of history. The genesis of these patterns can be traced back to Domain-Driven Design (DDD). In a nutshell, DDD aims to provide answers to two fundamental questions that have plagued the software development field for a long time.

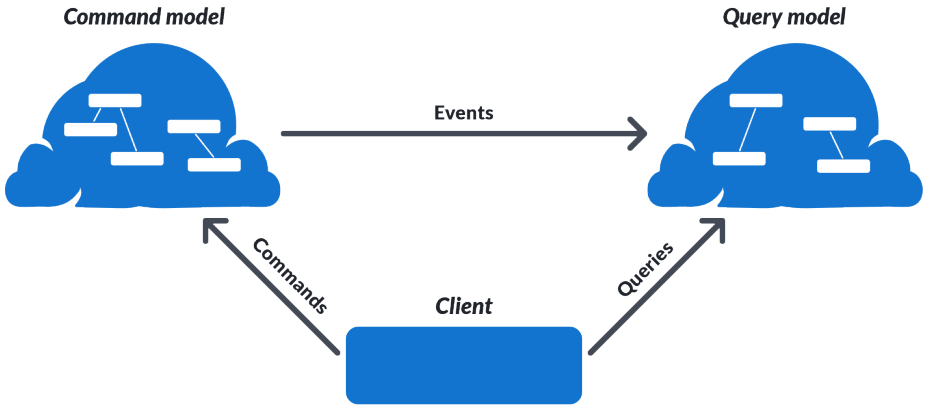
- How do we describe business domains, such as banking, retail, and hospitality in a language that is understandable across the enterprise – business and technology?
- How do we decompose software systems into cohesive modules (or components and microservices) that are loosely coupled?

DDD provides a systematic approach to application design with a set of strategic and tactical patterns that break down the complexity associated with business domains. DDD advocates breaking a business domain into a set of problems that you would like to solve for the domain. For example, in retail banking, you have to solve the problem of opening checking accounts. DDD helps teams then come up with something known as a "domain model" that can be applied to solve the problem. The domain model is an artifact that teams can use to communicate a common language as well as be a unit of deployment, such as a microservice, for that specific problem.

As the exploration of the domain started expanding, a challenging problem arose - that of model complexity. The model started to become big and unwieldy, making it becoming difficult to maintain, "Model fatigue" became a thing. Finally, different parts of the model started to have different non-functional requirements (e.g. scalability).

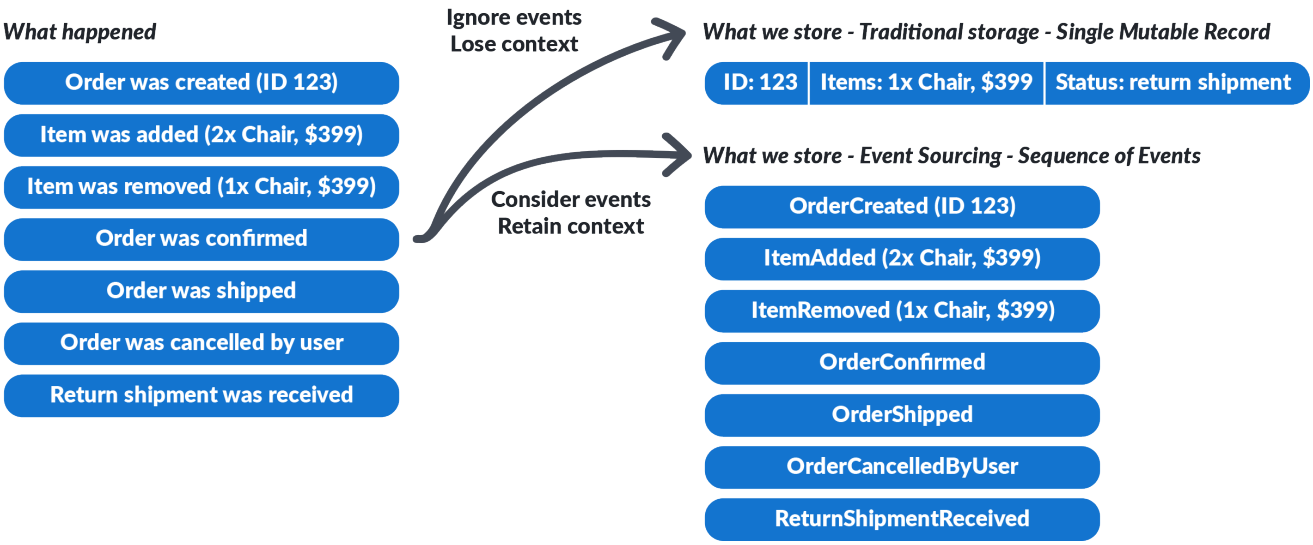
Enter CQRS, aka Command Query Responsibility Separation. CQRS was proposed primarily to tackle model complexity by dividing and conquering the domain model. Simply put, it advocates the vertical split of your domain model into two distinct partitions, a command model and a query model. The command model is focused on executing tasks, is primarily expressed in operations (e.g., open an account), and only contains the data necessary for task execution and decision making. The query model (aka projections) is focused on delivering information with data stored the way it is used (e.g., relational or NoSQL). Suddenly the complexity associated with bloated domain models disappears due to the optimization that CQRS provides. The split also helps you focus on different types of NFRs for each side of the domain model. For example, certain problems might have more reads than writes while certain problems might be the exact opposite. CQRS offers a great way to help address these kinds of problems.

However, the adoption of CQRS does present one problem that needs to be taken care of. How do we keep the two models in sync. In other words, how should changes in the command model eventually be reflected in the query model? An obvious answer would be "events," which could act as the glue between the command model and the query model and since events are the result of a task like "Open an Account" becoming an "Account Opened" event, they are tied to the command model.



This brings us to the final piece of the puzzle. Event sourcing deals with the concept of immutable data. Essentially, ES proposes to exclusively store the state of your model as a series of immutable events in a data store. The more traditional way is to store it as a mutable set of records. Event sourcing has a natural alignment to CQRS. Commands typically result in the events that need to be stored, and the same events are also consumed to help in the construction of various projections of the state that can then be served by queries. In short, Event sourcing without CQRS is quite difficult.

A side-by-side comparison of event-sourcing-based storage versus traditional storage is shown below. In the example, you see the lifecycle of an order being placed, confirmed, shipped, and finally canceled by the user. While in the traditional storage mechanism, we store just one record which reflects the current state of the order, in an event-sourcing-based storage mechanism, we store it as a series of immutable events. Suddenly you have complete context of why your system is in a particular state because you have the complete sequence of events that led to it - an extremely powerful feature and the fundamental selling point of event sourcing.



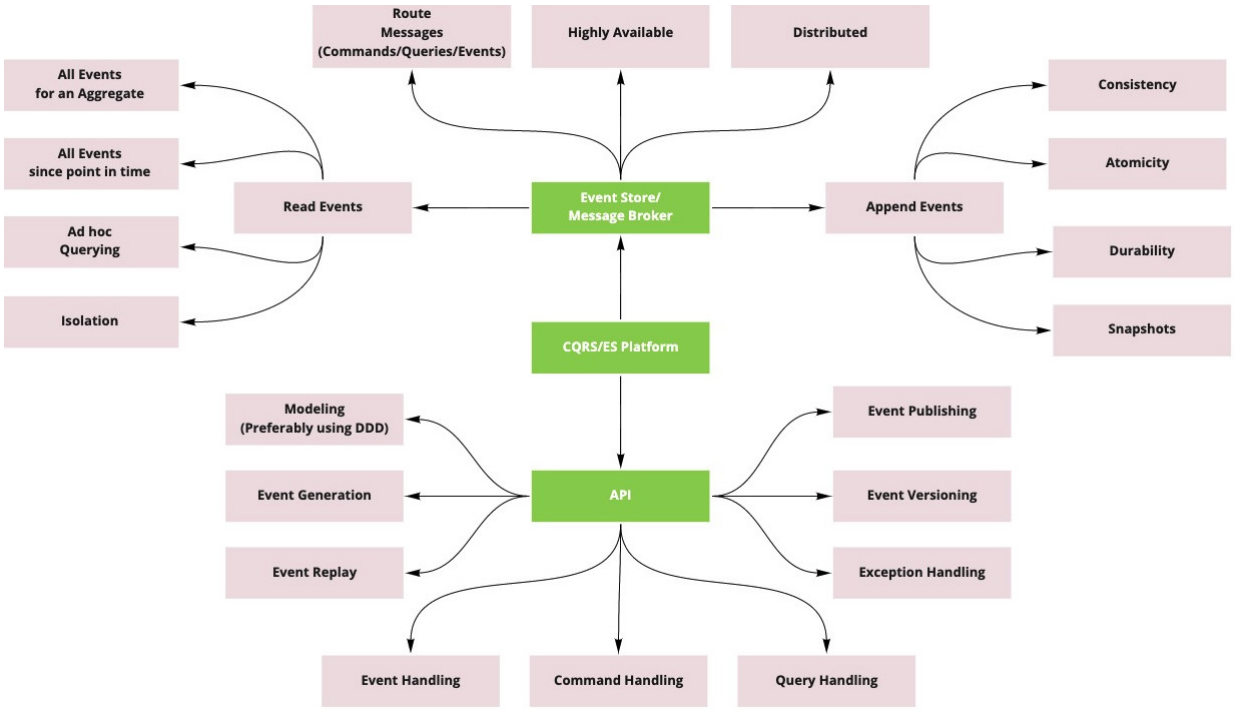
In summary, Event Sourcing will offer you:

- Immutability
- A reliable source of truth
- Naturalized audit trail
- Data mining/analytics
- Design flexibility
- Temporal reporting
- Easier to debug
- Enhanced user experience
- Event-driven distributed applications, such as microservices
- Reactive and scalable
- Predictable development model

To summarize, the twin patterns of CQRS/ES help build a completely new class of applications that are of a higher fidelity when compared to those that went before. Essentially they become the foundational patterns for your organization's architectural modernization effort.

Rolling out a CQRS/ES platform in your organization

While the concepts are easy to understand and the benefits of adoption are quite obvious, the big question is: how do you roll this out in your organization? You can start with the AxonIQ CQRS/ES Capability Map.



The Capability Map details the essential features that are required for rolling out an enterprise-grade CQRS/ES infrastructure across your application portfolio.

The Capability Map consists of two main areas

- A logical infrastructure, that is, an API to support CQRS/ES operations, preferably using DDD concepts.
- A physical infrastructure to support event persistence/retrieval and message routing (for commands/queries and events).

The Map guides you in figuring out the various components that would be needed to roll out the required infrastructure. A very important point that needs to be considered is the incorporation and implementation of a strong modeling and governance process as a complimentary aspect to the CQRS/ES component set. Event storming, event modeling, and storyboard mapping are examples of modeling processes that work very well with the CQRS/ES design paradigms.

To summarize, rolling out a CQRS/ES platform requires the implementation of a set of logical/physical infrastructural components complemented by a robust modeling/governance process.

The Market for Purpose-Built CQRS/ES Platforms

Why a Purpose-Built platform?

Implementing the infrastructure CQRS/ES requires is complex. While the first tendency often is to build out these patterns utilizing custom-built frameworks and tools, it quickly spirals out of control. The chance of failure is very high as the various aspects of these patterns start to become clearer. This in turn translates into a messy architecture, poor developer experience, complex and operational aspects. The cost is high but without much to show for the effort spent.

A wiser decision is to use a purpose-built CQRS/ES platform. This helps accelerate the adoption of these patterns quickly and effectively. It provides a complete ecosystem of tools, a single infrastructure, and techniques.

This enables technology teams to adopt these patterns rapidly:

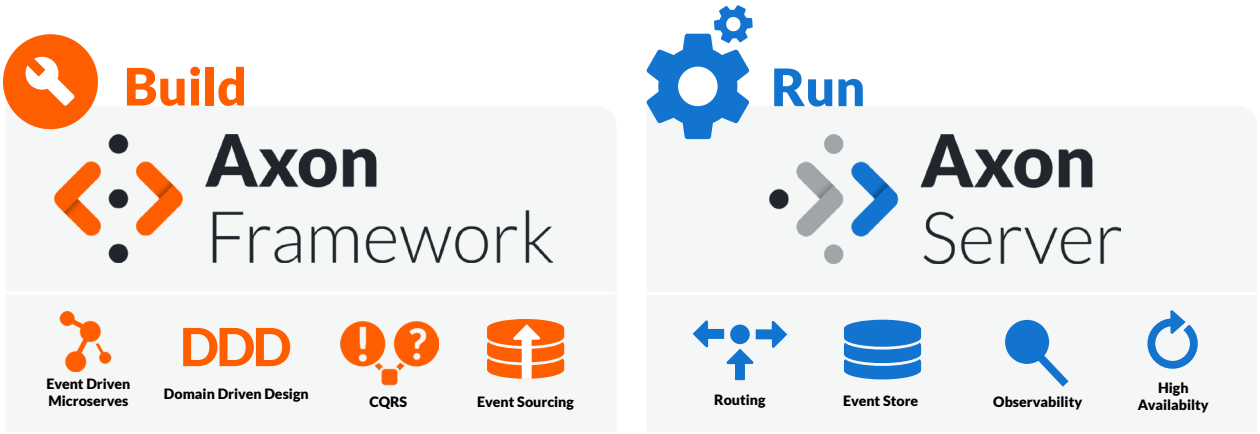
- Architects can implement a streamlined and enforceable architecture.
- Developers can focus on what they do best, which is writing business code.
- Operators have a single physical infrastructure to monitor, optimize, and automate.
- CxOs can realize significant savings while at the same time showcase real architecture modernization progress.

The Axon Platform

Axon is the leading purpose-built CQRS/ES platform that helps rollout enterprise-grade CQRS/ES infrastructures. Axon elevates the concept of traditional CQRS/event sourcing architectures by treating every operation in an application (commands/queries and events) as “messages.” Axon processes these messages using the location transparency pattern, enabling teams to focus on getting the correct boundaries for their applications, allowing them to be split into microservices as the requirements evolve.

The Axon Platform provides two main components:

- **Axon Server (Physical Infrastructure)**
  A highly scalable, distributed, and purpose-built Event Store and zero-configuration Message Router. It is available in two editions Standard and Enterprise (for more advanced capabilities like Clustering).
- **Axon Framework (Logical Infrastructure)**
  Implements the full range of API capabilities required for event sourcing/message routing operations. It provides the building blocks required to deal with all the non-functional requirements, allowing developers to focus on the functional aspects of their application instead.



The Axon Platform has multiple customers across a wide range of industries who use it to roll out an enterprise-grade CQRS/ES infrastructure. An important aspect here is that the Axon Platform is an application platform, not a deployment platform. Applications built with the Axon Platform can be deployed by any existing cloud infrastructure (e.g. Kubernetes, OpenShift, and Tanzu) that organizations currently have invested in.

Conclusion

To conclude, the CQRS/ES patterns can serve as the foundation for your architecture modernization efforts within your organization. The adoption of these patterns requires implementing a set of capabilities that these patterns prescribe. Utilizing a purpose-built CQRS/ES platform like Axon helps organizations reap the significant benefits that these patterns offer in a faster/cost-optimized/streamlined and efficient way.

Contact us

Document number WP -PB-2022-04