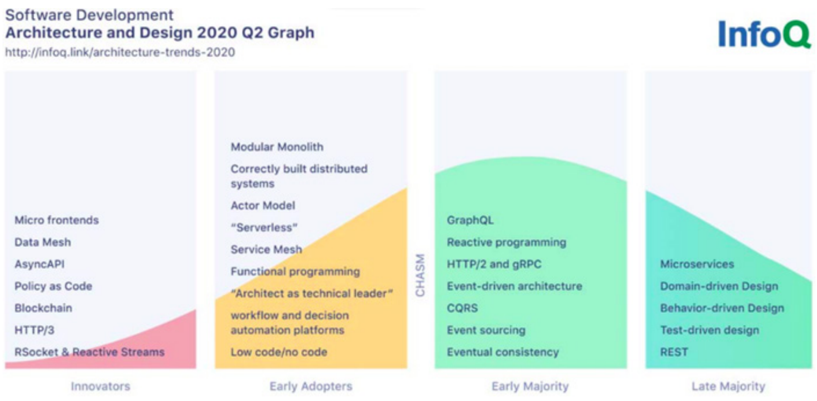
Why is CQRS-ES a good option for instant payments?

IPF Technology Series

Why is CQRS-ES a good option for instant payments?

CQRS (Command Query Responsibility Segregation) and Event Sourcing (ES) are not new patterns. ES has appeared on ThoughtWorks Technology Radar from time to time since 2011 and CQRS has gained more popularity over the years. According to the InfoQ Architecture and Design Trend Report (April 2020) it has now even moved from early adopters to early majority by the InfoQ editors indicating that more and more organisations are adopting these patterns in their software.



The popularity of CQRS is in line with the increasing demand for high throughput, low-latency solutions, especially for the distributed systems when data consistency is not the biggest concern.

CQRS and Event Sourcing explained

CQRS is an architectural pattern that suggests a separation between processing commands and answering queries. The part that processes commands and manages updates to the application state, also known as the write part, is modelled and built independently from the part that is responsible for responding to the queries, also known as the read part. This separation allows us to address distinct concerns for each part and apply specific optimisation strategies without impacting the other.

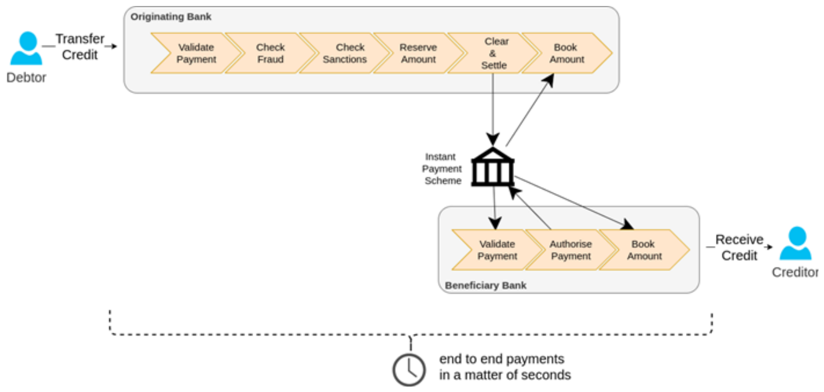
ES is a way of representing the application state by constructing it from the history of events that have happened in the past. In event sourced applications, it is not just the current state that is important but also the steps of how it got there. Therefore, the risk of inconsistencies such as publishing an event but failing to update the current state is eliminated and the audit trail comes for free as it becomes a critical part of the domain. ES is often seen as a natural fit for CQRS, when modelling the command processing side. It is used to model the write side of CQRS by recording every critical state change of a business entity as distinctive domain events and using those events to construct the current state when needed. Domain events are the key part of the application and it is critical to model them to reflect the business processes. An increasingly used event modelling technique is “Event Storming” which facilitates higher collaboration between domain experts and technologists and helps understand business processes and translate them into domain events that can be event sourced.

The query side on the other hand, is built independently, based on various projection requirements and allows future extension and variety of view models without impacting the command model.

How does it relate to instant payments?

Many software solutions offer some sort of payments functionality to their customers and payments is often described as one big black box, usually represented as communicating to a payment processor or a gateway. Behind the scenes however, there is a is very rich and complex domain. There are numerous types of payments with different SLAs, regulatory checks, validations, execution steps, exception handling and clearing and settlement mechanisms.

Instant payments, being a sub-domain of payments, stands out with distinctive requirements such as fast to real-time processing times, various checks and controls on payment data, instant decisioning and making funds available immediately along with real-time clearing and settlement mechanisms and many others.



Above is a simplified view of an instant payment process, it is important to note that the process differs depending on the country, region, financial institution and payment schemes.

Instant payments scheme rules and SLAs impose high demands on payments applications such as real-time processing, reliable and available service with near-zero downtimes. An instant payment solution must support high throughput low latency payments processing, handle failures elegantly, cope with load in a cost-effective manner with no contention or bottlenecks, stay responsive under load ensuring consistent response times and stay responsive during failures providing interactive feedback. In short, it should be able to cope with distributed systems problems efficiently and naturally support reactive systems characteristics.

Applying CQRS with ES to such a solution is not the only way to solve some of those problems, but certainly a good one with many advantages.

- **Efficient and non-blocking writes**

Every critical state change is recorded in the write side as domain events, in an append only fashion. As there is no need to persist and maintain a separate current state, there is no need for update-in-place and no need to implement optimistic locking type solutions, eliminating the risk of contention.

- **Efficient read-side queries with complex view models, without impacting the command side**

Complex queries for searching and analytics purposes are executed on the separate read side. Accepting the reality that there is always going to be an element of delay when reading/fetching data and eventual consistency allows decoupling of read and write models. A variety of read models and complex projections are created by fetching the data from the write side as part of projection/synchronisation, without impacting the overall payment processing.

- **Independent scaling of teams and use of technology**

The separation of the read and write side also allows wider decoupling of teams and technology, providing the opportunities of scaling different teams and applying specialised technologies to improve efficiency of the teams and the read and write solutions. It is possible to apply technology options and optimisations which can differ between read and write side.

- **No need to maintain a separate execution history**

Domain events are immutable facts representing the change in state. By empowering domain events to be the critical part of the solution and constructing the application state based on them also gives the advantage of maintaining business critical execution history as part of the payment processing, not as an afterthought.

- **Consistent state management**

Domain events become the single source of truth, eliminating the risk of publishing a domain event to the outside but failing to update the state consistently.

- **Improved self-healing and troubleshooting**

In case of temporary failures, recorded domain events can be replayed again to recover the application state to allow self-healing, debugging or troubleshooting.

- **Archiving is much easier on immutable data**

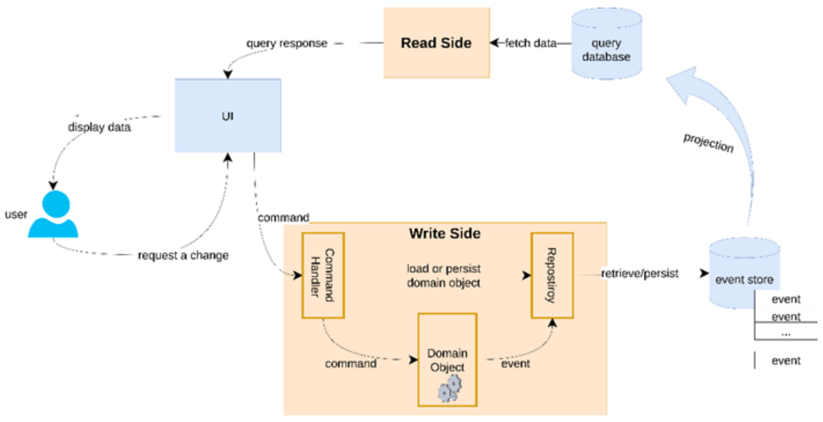
It certainly is much safer to archive immutable domain events to a slower storage.

- **Object model does not have to be same as the data model**

CQRS naturally guides you to model your domain with the right concerns from the very beginning. Commands are modelled to contain concise information that is enough to make decision on how to handle the command. This allows the model to be more focused and not populated with concerns which do not serve the intent.

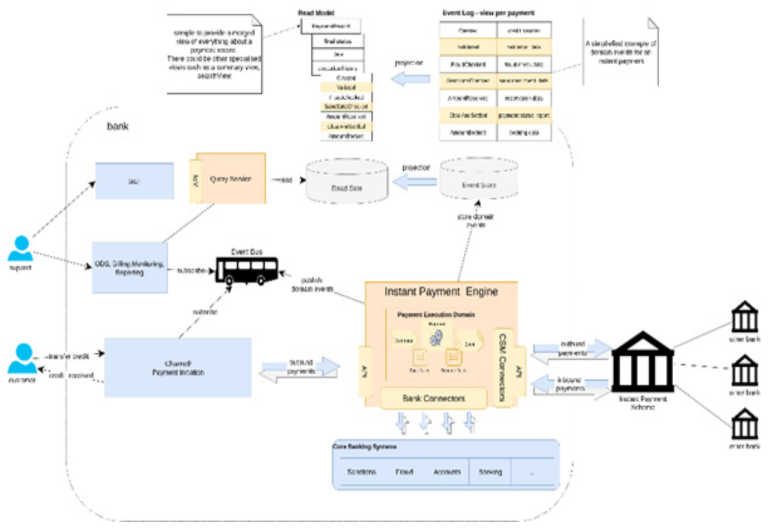
What does it look like?

A CQRS-ES type application brings the best parts of both patterns together. A simplified view could look as below:



The overall application design is typically backed by other software patterns and styles as well, depending on the requirements. For instance, to increase the asynchronous messaging and decoupling of the components and layers, an event bus could be introduced. A distributed cache will also help managing domain objects in memory. Depending on the nature of the application you may value consistency over availability and apply application level cluster sharding solutions. Domain driven design approaches and hexagonal architecture principles are also known as good fits to applications with CQRS-ES core.

In an instant payments type solution, the overall picture wouldn't look very different. At a very high-level, an example of an instant payment engine utilising CQRS and ES could look as illustrated below. It is a simplified view omitting payment scheme connectivity and other infrastructure services for the sake of highlighting the separation between the write and read sides. A very simple list of domain events is also given to illustrate the possible usage of event sourcing.



A simple scenario could be where a customer of the originating bank wants to transfer credit to another customer who is banking with another bank. Both banks support instant payments and have connectivity with an instant payment scheme.

- An initial command to execute a credit transfer request is received in the payment engine.
- The execute payment command triggers the execution of a payment flow which is essentially a combination of predefined execution steps.
- Each execution step is handled as a command to the payment and based on the business logic they may cause a payment state change and trigger a payment domain event.
- The execution steps may be internal decisions based on the business logic and rules or delegation of the decisions to external systems.
- Payment domain events contain payment data and relevant external system responses. They are broadcasted after they are successfully persisted.

The object model of the payment processing side contains payment state, which is represented by the aggregated view of persistent domain events, and relevant external system responses and payment business logic and rules.

The query side is populated by going through the persisted domain events and modelling required views. A sample view could be aggregating the payment data and restructuring the domains events to drive audit trail information. As the writes are made to an append-only log, this process does not impact the inflight payment processing.

What are the ins and outs?

It only makes sense to use a pattern or architecture style if it is going to solve your problems. CQRS and ES come with some really good ideas, but it is also important to understand the consequences and possible impact of introducing such solutions to your application.

CQRS is generally more beneficial when;

- the application logic does more than just CRUD type operations
- there is a large difference between the number of reads and writes
- performance is a critical requirement
- read queries are far more complex

As nothing comes for free, implementing CQRS will have consequences.

- **Embrace eventual consistency**

With CQRS, there will be a natural lag between the writes and reads, the data updates may not be immediately available on the read side. Understand the acceptable limits and apply optimisations to reduce the latency. It is best to start with challenging the reasons behind the requirements for a strong consistency. It may become clear that all that effort and cost for a strong consistency may not give you the desired business benefit at all; it may even degrade the application proving to be even more costly.

- **Steep Learning curve**

As anything else, there is a learning curve that requires a shift from traditional design thinking. However, the pattern is not new and there are many different implementations with examples and articles and a large community of developers and technologists.

Event sourcing is a natural fit for CQRS but it would be good to understand its cost before embracing the benefits:

- **Dependency to a 3rd party framework**

It may be challenging to roll out your own event sourcing solution as you need to consider from persistence to rehydration, snapshotting and versioning. However, there are good examples of such frameworks available as more are also being developed. At Icon we use the Akka framework which provides a good API and approach for handling commands and managing events complemented with Akka Persistence and Akka Persistence Query, together with asynchronous messaging and many other features providing a good base for CQRS-ES type applications.

- **Backward compatibility**

In event sourcing, it is easier to introduce a new state as it will not really impact the overall model. However, in time, you may need to apply more breaking changes, as in mandatory changes, in an event structure. A good approach is to be flexible in what you are accepting from outside and be strict when exposing data to outside. This may allow the introduction of adapters when necessary. Snapshotting before introducing breaking changes and broadcasting a major event are also helpful strategies.

- **Increased volume of data**

Naturally there will be more data than the application will be persisting. This is primarily because the audit trail of 'how we got there' information is now the critical part of the domain. Archiving is not as problematic anymore as it is an append-only log. Removing the events after broadcasting and snapshotting could also help with data volume concerns.

- **Rebuilding state can be costly**

As the number of events to represent the state changes in a domain object grows, it may become inefficient to rehydrate all the events for the object to build its current state. Snapshotting could be useful and can be applied to the domain events after a major change. This would reduce the number of events to be rehydrated and improve the overall performance. Another option could be to keep the domain object in memory, perhaps to introduce a distributed cache. This is a risky approach that you need to make sure cache is updated consistently as the domain events are persisted.

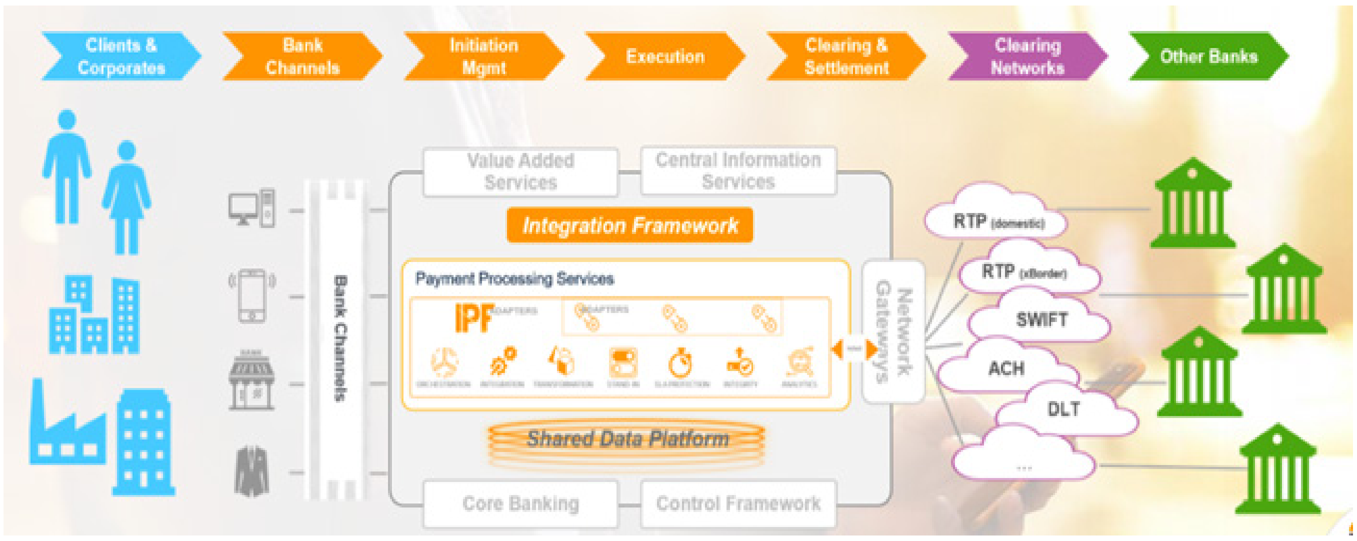
- **Challenges in Duplicate checking**

Duplicate checking is often used as a criticism to event sourcing. As there is no update-in-place type data management, it may be more challenging to detect duplicate entities. Again, there are various strategies to overcome that. One strategy is to allow duplicates but to implement a rigorous after process which handles duplicate data. However, this will not work for instant payments as the money movement is very critical and allowing duplicate payments would be too risky. Another strategy could be to use the unique identifier for the payment as for the primary key for events, not allowing duplicate events for the same payment id. However, as payment id is often a combination of mandatory and optional fields, it could be challenging to implement such a unique identifier. A better strategy is to keep a temporary cache of payments objects close to the application and new payment execution commands could be checked against the cached payments and rejected if duplication is detected.

Conclusion

While CQRS and ES are not the answer to everything, they are certainly good patterns for instant payments solutions enabling high throughput and low latency payments processing with the increased abilities of self-healing, troubleshooting and extensive audit logging.

At Icon Solutions we provide payment solutions and enterprise services to the global financial sector based on a decade of developing leading IT capabilities. We have combined our years of payments domain expertise with software craftsmanship and created IPF, high-performance low-cost collaborative payment platform that allows building such solutions in a consistent way.



IPF combines CQRS and ES along with domain driven design and hexagonal architecture principles, providing a highly decoupled, scalable and a performant application. Its highly extensible and modular framework allows us to embrace unique bank requirements without disrupting the core product components and principles. Through IPF we have the ability to build client specific payment applications which integrate to proprietary systems but benefit from a consistent core domain that is the result of years of payments domain expertise and the implementation of highly efficient design patterns like CQRS, ES and many more.

Inspired by:

https://martinfowler.com/bliki/CQRS.html

http://codebetter.com/gregyoung/2010/02/13/cqrs-and-event-sourcing/

http://codebetter.com/gregyoung/2010/02/16/cqrs-task-based-uis-event-sourcing-agh/

https://martinfowler.com/eaaDev/EventSourcing.html

https://microservices.io/patterns/data/event-sourcing.html

Reactive manifesto : https://www.reactivemanifesto.org/

How to use CQRS in Akka 2.6 video : https://akka.io/blog/news/2020/02/05/akka-cqrs-video?_ga=2.134613489.102623210.1593520300-198436246.1591741041

IF YOU WOULD LIKE TO FIND OUT MORE ABOUT IPF, PLEASE CALL US ON +44 (0)20 147 9955 OR EMAIL INFO@ICONSOLUTIONS.COM