DDD, CQRS and Event Sourcing Explained

An explanation of the concepts at the basis of Event-Driven Microservices.

Allard Buijze - 2019

AxonIQ

DDD, CQRS and Event Sourcing Explained

WHITEPAPER

Microservices started to get popular around 2014 although the term was coined a few years earlier. As the interest has spread, the term has unfortunately become vaguer and today microservices is commonly seen as just a distributed system architecture.

During the recent years both event-driven architecture (EDA) and event sourcing has drawn an increasing interest, to some extent due to the interest in microservices which requires an effective way of communicating between services. These concepts have also brought some confusion about what they really mean and how they affect the architecture and design of applications.

In this white paper we describe the basic concepts, common misunderstandings we have seen and how the Axon platform helps by providing the foundation for building asynchronous message-driven systems based on the concepts of microservices, event-driven architecture and event sourcing. But using the same basic concepts and ideas Axon also supports developing applications that start as a monolith and later evolve into event-driven microservices without significant refactoring as the requirements change.

Domain-driven design (DDD)

DDD is an important approach when working with complex domains and even more important in the context of microservices. DDD has been around for more than 15 years but has received an increased interest due to microservices. Concepts like bounded context are important for finding the boundaries between modules or services from a business perspective, whether in monolithic or microservices based applications. Ubiquitous language is another important concept — a common language used by developers, businesspeople, and others involved in a specific bounded context to improve the understanding and minimizing the risk of misunderstanding.

Aggregates is a term at a more technical level and describes sets of entities that work together. An aggregate is composed of one or more entities that must always be consistent — the aggregate is a transactional consistency boundary. After one or more changes of the entities within an aggregate it must always end up being consistent when the transaction is committed. Especially in distributed, eventually consistent systems, the aggregate is an essential concept to ensure correctness on the longer term.

Monolith

A monolith in the software context is a single application deployed as one binary. Building a monolith has for a long time been the standard way of building applications or systems. Modularity as a way of simplifying has been around as long as the software industry itself and a well-designed, modularized monolith is for many applications a perfectly well-suited type of architecture. Unfortunately, they are in practise often not well structured which can lead to problems maintaining them, and in the worst case to a Big Ball of Mud.
A well-designed monolith is using a component-based design with each component or module responsible for a specific part of the business — a subdomain or a bounded context in DDD language. If for some reason the need to move out part of the monolith into separately deployable units (microservices) arises, this is now feasible. Note though that this should only be done because of non-functional requirements.
One problem when splitting out microservices from a monolith is communication. Within the monolith direct object calls can be used, but with microservices, communication is done over the network. Using Axon and its location transparency (explained further down) and always sending messages this problem is mitigated.

Microservices

The concepts and patterns that are the base of microservices have been known and described for years before the term microservices was established. Service boundaries, asynchronous message based communication, application databases, etc. are all well known concepts that have been in use for many years.

The main reason for adopting microservices should be communication between people. It's hard to work effectively on a piece of software with too many engineers, so we need teams to be able to work independently on different parts of a system. With well-designed microservices, teams can implement, deploy and run their services independently and with minimal impact on other teams. Sometimes present or foreseeable scalability issues are also a reason to work with microservices.

But microservices also comes with additional complexity, both from a technical and a domain or business perspective. Finding the correct bounded contexts and service boundaries, and communication between services are just two of a number of challenges that comes with microservices.

Finding the right set of bounded contexts from the beginning is very important when starting a microservices. Refactoring and moving functionality between services is much harder

than working within a monolith.
Microservices never work in isolation, they always need data from other services. Common ways of communication are request - response, maybe using HTTP, and publish - subscribe or some other messaging pattern. Data ownership and read-only copies of data are other issues that appear in a microservices architecture.
A common saying among architects and engineers with experience from distributed systems is that if you can't build a monolith, forget about building microservices.

Event-Driven Architecture

With the increased usage of microservices, there is also demand for communication between all services. Events are a great way to asynchronously distribute information about things happening at a business level in a service. A customer service can publish information about new customers or that a customer has moved to a new address. But events are not enough; commands and queries are two types of messages that are also essential to achieve a well-designed application built on an 'event-driven' architecture, so this type of architecture should really be called Message-Driven Architecture.

An important part of emitting events is that the publisher doesn't care if anyone is listening, if no one is listening to the emitted events the publishing service should still be able to perform its tasks.

There is a common misunderstanding today that a service that wants something to be done in another service publishes an event. This makes the responsibility for the outcome of the business task at hand unclear, and often creates a need for close monitoring of all individual events in an attempt to find problems in a business flow. This uncertainty is very close to the pinball machine architecture style sometimes mentioned in connection with serverless architecture where it can be hard to understand where data is and what functions are invoked.

If the publisher expects something to happen it should instead asynchronously send a request to another service as a command, and then asynchronously wait for the outcome. This makes it clear that the service sending the command is responsible for fulfilment of the business task at hand.

One example is an Order fulfilment service that requires a payment for an order before it continues with the order. The service sends an asynchronous command to a Payment service. When the payment has been completed the Payment service returns success, or failure if the payment has failed. The return message is picked up by the Order fulfilment service which now can continue with the order and request shipment if payment was successful. In this scenario the Order fulfilment service is aware of a Payment

service, which is correct, but the Payment service is not aware of what the payment is for.
If the business now decides to ship to a trusted customer before payment has been received, this can easily be handled within the Order fulfilment service without touching any other services. In a pure event-driven approach this change would require changes to at least a couple of services, potentially to many services.
In this scenario, the Payment service has full responsibility of handling the payment, which may include asking the customer for a new credit card or other information to be able to complete the payment. This process that may take hours or even days if the customer never responds to a failing payment but that is not a problem. If the payment is never completed, eventually the Payment service will cancel the payment and return a failure to the requesting service.
Events should have a focus on behaviour and correspond to events at a business level. A service should not just emit events that data have changed, but describe changes that have a meaning to the business. Instead of a CustomerChanged event emitted when a customer has moved to a new address, a better event might be CustomerMoved. Modelling events in this way forces a behavioural focus instead of on structure, which is beneficial from a DDD perspective and makes it easier to understand what an application is doing. Events mimicking real world events in the domain often also result in less changes because domains don't change that often.

Events and messages

As already described, there are three major types of messages in an EDA:

- An Event represents something that has happened. They are an immutable fact and can therefore not be changed or deleted.
- A Command represents an action that the sender wants the recipient to perform. The result of the execution is then returned to the sender. Commands always have exactly one destination.
- A Query represents a request of information the sender wants from one or more recipients. The requested data is then fetched and returned to the sender.

An important aspect of events is that they are immutable, they represent a fact of something that has happened and must never be updated or deleted. One option when the need to change or remove events arises, a new output event stream can be created and used to create new events from the original immutable stream of events.

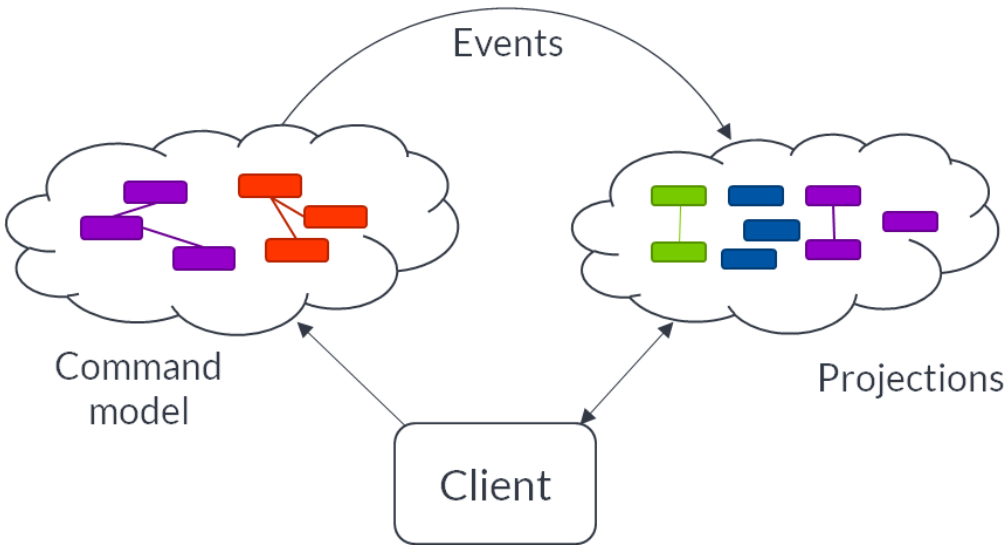
Eventual Consistency

The world is eventually consistent. Transfer of money between two accounts in different banks is a chain of transfers that eventually will be correctly reflected in both accounts. Unfortunately, strong consistency has for a long time been the norm in the software community. Everything must be in sync, sometimes even between services, which has meant the use of different types of complex transaction protocols which has added a lot of unnecessary complexity to systems. With microservices the software community is adopting eventual consistency; changes in one service are transferred to other services using events, messages or other forms of transports that eventually will be consistent with the emitting service . But, it is also important to understand that each individual aggregate, wherein decisions are made, must always be consistent. It's the result of these decisions that are eventually "visible" to other components.

While eventual consistency is an inevitable concept in large-scale distributed systems, not everything in such system is eventually consistent. Certain decisions within a business domain should never be made on inconsistent data. Within an eventually consistent system, there are several individual components that are internally strictly consistent. The aggregate described earlier is such a component. These strictly consistent components allow for decisions to be made based on a reliable and consistent source of information, while the results of those decisions eventually get updated accordingly in other components.

CQRS

Command Query Responsibility Segregation (CQRS) is quite a simple concept. It states that execution of a command should be segregated from queries returning state. For a software system this means that the part that changes the state is separated from the part that queries the state.



This has several advantages. A command should affect one aggregate only, whereas a query often retrieve a larger amount of data or lists of data, which means we can minimize the effect of a write and optimize the amount of data retrieved. It's also possible to use separate types of storage for writes and reads, which enables the use of event sourcing. Often reads have to be more performant than writes and with a separate read model this model can be optimized for reads.

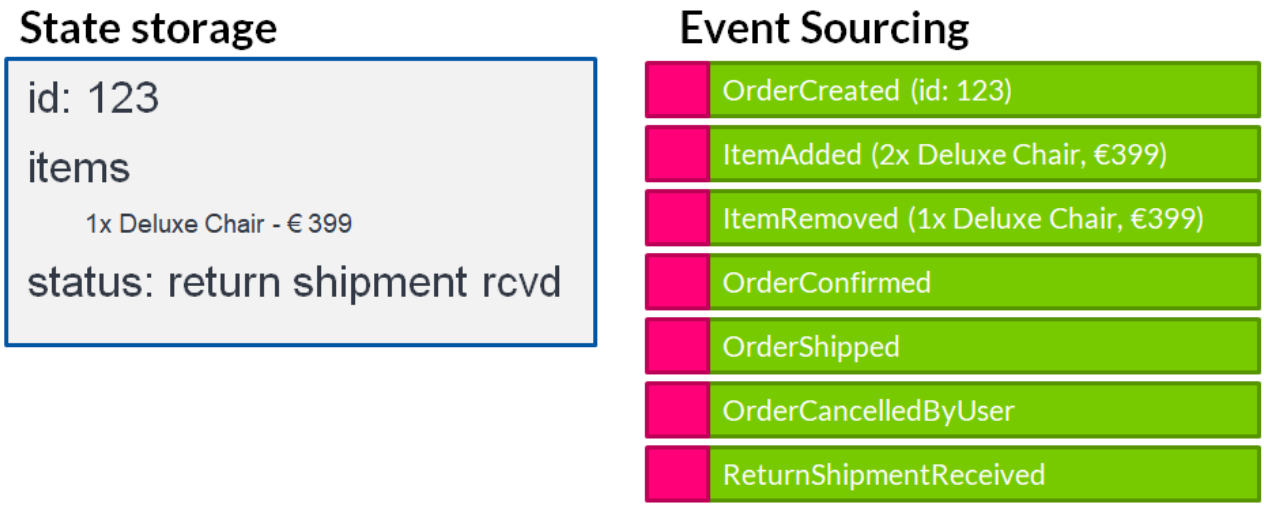
But this separation also means the read model will be updated by things happening on the write side. Doing this asynchronously means a command may not be reflected in an immediately following read — the application itself is now eventually consistent which may have to be considered in e.g. a GUI.

Event sourcing

The ideas of event sourcing are not new, systems were sometimes built this way in mainframes a long time ago, and databases often work with event sourcing internally.

In event sourcing the state of a business entity is persisted as a time-ordered sequence of events.

When the state of the entity changes, a new event is appended to the list of events. Current state of the entity is created by replaying all the events. Periodically saving a snapshot of current state is a way to optimize loading when an entity has a large number of events.



When using event sourcing within a component or a service you get a complete and reliable audit trail of that service. Past state can also be reconstructed by replaying all events up to a certain point in time which can be useful when evaluating the result of bugs in a system. It's also easy to migrate from event sourcing to state-storage-with-events-as-side-effect, but not vice versa.

Commonly, a whole system should not be event sourced. Instead the use of event sourcing should be a decision made per bounded context, or possibly per aggregate. One important argument

for when to use event sourcing is if state transitions are an important part of the problem space and should be modelled within the domain.

As already mentioned, modelling using events, a lot of problems in a domain will go away. In industries like finance, banking or insurance the event sourcing concept is often used. One reason is the need to keep everything that happens.

Event sourcing events should not be published outside of the bounded context. Within one context all services speak the same language and can understand and may share all events, publishing to the outside will create unnecessary dependencies. Instead, create domain events for the published interface of a context. One exception to this is services used for analytics or reporting.

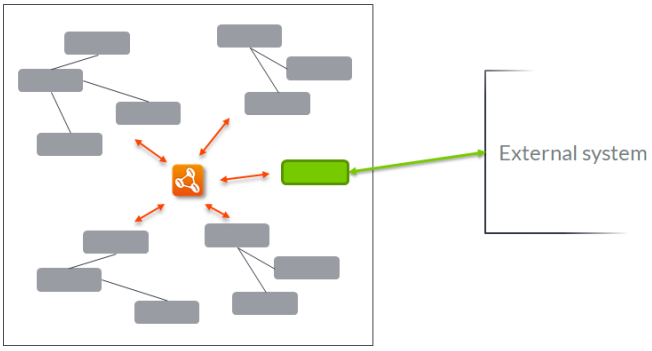
One argument often used for event sourcing is the possibility for replaying events in case an error is detected. This can look harmless for the service doing the replay — it just replays all of or part of an event stream from another service and reconstitutes its state. But if this service also emits events, the replay probably has an effect on these already emitted events which has already been consumed by services further down the chain. This causes a ripple effect with unforeseeable impact on the whole system. If an event in practice is handled as a command – an order has been placed — it may result in the order fulfilled twice. A replay must therefore be done with care, deciding if the result of a read event should be carried out or not.

Location transparency

One important concept to enable a move from a monolith to microservices is location transparency — a component should neither be aware of, nor make any assumptions about the location of a component it interacts with. This allows for a system to migrate from a structured monolith, where all components are deployed as part of the same unit, to a microservices system, with each component deployed individually. All done without any changes to code. Commonly location transparency is done by using messaging in some form.

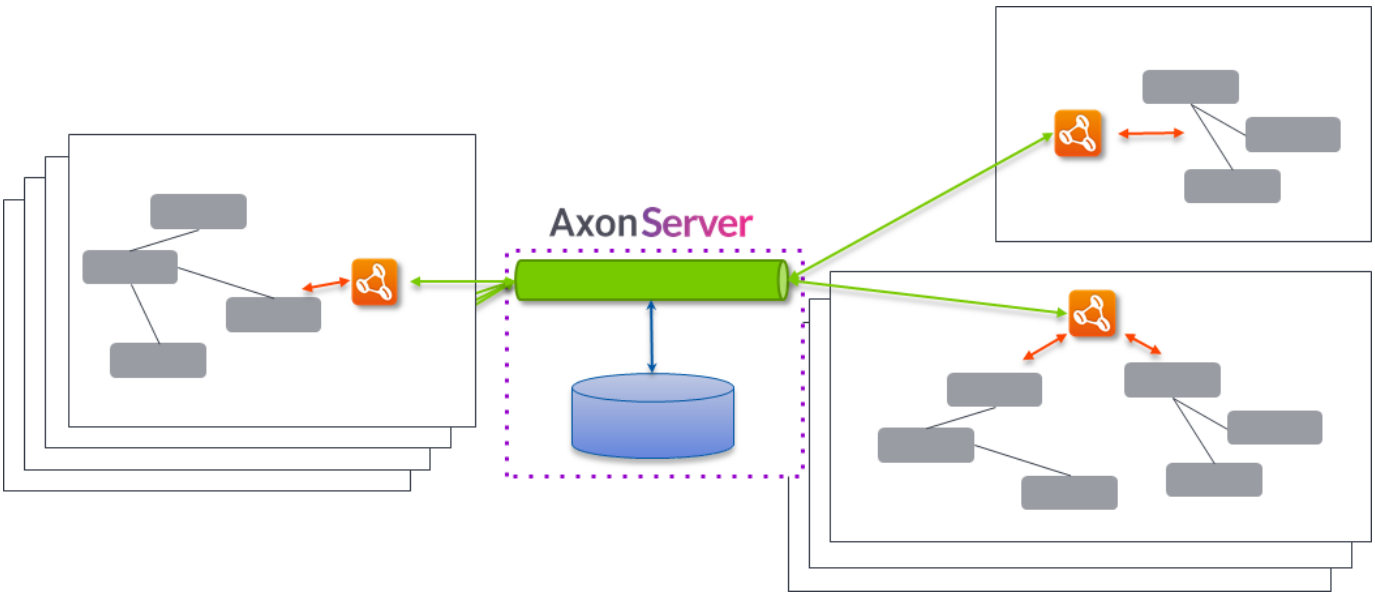
Events and messages

Architecture of a new application or system should be as simple as possible but still allow for a growth if the need arises. A great start for many systems is a message-driven modular and component based monolith, event sourced in parts where it's needed. It probably uses the hexagonal architecture style internally and leans on the Axon platform to provide most of the infrastructure and enable for a migration to microservices if the application usage is a success. This style of building a monolith is beneficial even if it's never split up in microservices. It simplifies maintenance, updates of business logic, etc.



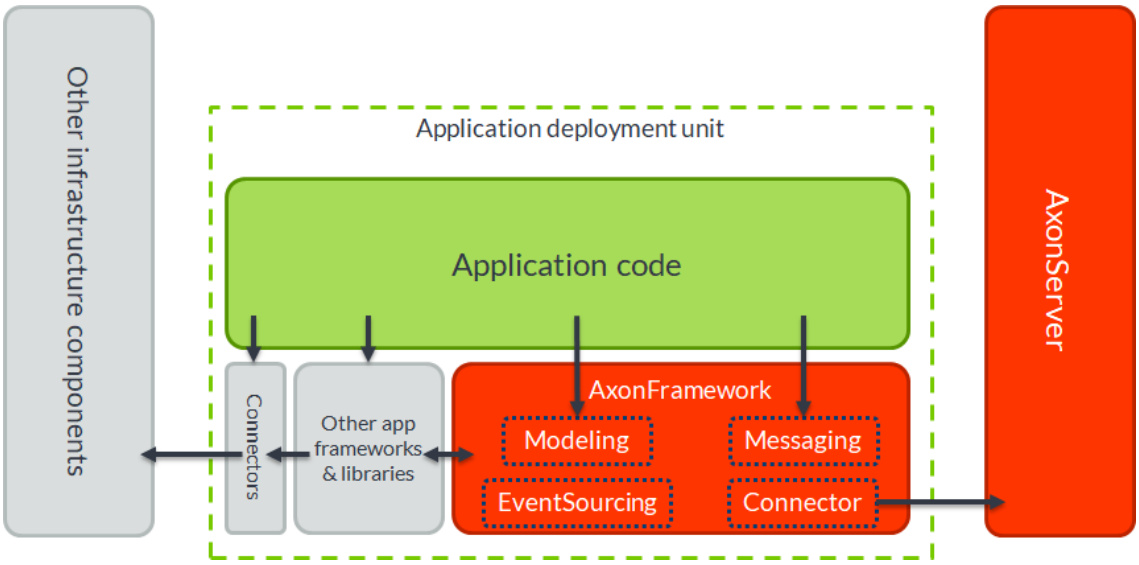
Splitting up the architecture

Gradually evolving from a modular monolith to microservices is greatly simplified if the monolith is built for this from the beginning and is using Axon. Each microservice becomes independently scalable, allowing it to address the non-functional aspects specific to each microservice instance.



Axon

Axon is a framework and a server, both open source, that together builds a platform for event-driven microservices based systems that provides the foundation and infrastructure needed for successful design and implementation of systems overcoming the challenges described.
In Axon all communication between components is done using message objects. This gives these components the location transparency needed to be able to scale and distribute these components when necessary, without any changes to business logic. It also means that an application only has to be split across deployable units if the non-functional requirements, such as team size, release cycle, availability requirements, performance, etc. require so.



Axon Framework

Axon Framework provides the building blocks for applications based on principles like DDD, CQRS and event sourcing. Axon Framework has been designed to separate the business logic from infrastructural concerns and it supports an evolutionary approach by supporting a monolith to evolve into microservices.

Axon heavily stimulates the separation of logic into smaller components, which communicate with each other through messages. This significantly reduces the mental burden on developers working with specific components, letting them focus on the logic correlated to a specific message, instead of all the infrastructure needed for handling the messages themselves.
Location transparency is a key element in the framework ensuring that a component communicating with another component do not need to know where that other component is located.

As already mentioned, replaying events can be extremely complicated. Axon Framework provides granular control to event handlers in terms which events get replayed and which don't. Also, it is possible to monitor the progress of the replay procedure.

Axon Server

Axon Server is a message router and an event store used in a distributed environment.

Axon Server is responsible for routing of messages between all services. It has knowledge about the different types of messages that are being used and know how to deal with each type; events are sent from one service to one or many other services, commands are sent to one service to do something, potentially waiting for and returning result, and queries are sent to one or more services to retrieve information, always returning a result.

Axon Server also includes a purpose-built event store, used for storing the events created by event sourced aggregates. When storing events, it also pushes the event to listeners and event processors that are running, thus removing the need for regular polling and the latency that it brings. One important feature is the constant performance irrespective of storage size. The number of events can be extremely high in an event sourced system and a storage that becomes slower as it fills up will lead to significant performance degradation for the whole system. Other important features include the possibility to append multiple events in one transaction, snapshots for storing current state of an aggregate, thus avoiding the need to read potentially a large number of events to create state. It is also optimized for recent events. Especially when using snapshots, only the most recent events are read from the store.

As already described, replaying events should be done with care. Events emitted publicly, or event handlers contacting some external systems can cause huge and irrecoverable problems. In Axon, gateways to other systems can be disabled when event are replayed. Axon also provides granular control over event handlers deciding which components need to get their events replayed.

AxonServer Enterprises comes with the ability to run in a clustered environment. This helps ensure that any failure of a single node will not impact the availability of the cluster as a whole, giving it availability guarantees that you may expect from any production-grade system. Axon Server's multi-context support allows for separate teams or departments to manage their own virtual environment on a centrally deployed cluster, simpifying operations in enterprise environments.

Are you interested in using Axon or do you have any questions?
Then don't hesitate to contact us via info@axoniq.io.

References

A great and thorough definition of Microservices, written by James Lewis and Martin Fowler back in 2014 when the term was new.
https://martinfowler.com/articles/microservices.html

A blog post by Martin Fowler where he writes that he have noticed a pattern where almost all the successful microservice stories has started with a monolith.
https://www.martinfowler.com/bliki/MonolithFirst.html

Stefan Tilkov wrote in a blog post shortly after Martin Fowler's blog post that he is firmly convinced that when the goal is a microservices architecture, starting with a monolith is usually the wrong thing to do.
https://martinfowler.com/articles/dont-start-monolith.html

Eric Evans described in his keynote at DDD Europe 2019 different kinds of bounded contexts, some that may be especially useful in an event based system.
https://www.infoq.com/news/2019/06/bounded-context-eric-evans/

Martin Fowler describes event-driven applications and the potential problem with event notification in a business flow.
https://martinfowler.com/articles/201701-event-driven.html

A series of blog posts from 2008 that describes many of the design ideas and patterns that led to microservices. Note that the blog post are from a time when SOA was popular and before the experience of using microservices, so some ideas may in some parts have changed.
http://bill-poole.blogspot.com/

A three part story where Vaughn Vernon in detail describes how to implement aggregates using DDD.
https://kalele.io/effective-aggregate-design/

The AxonIQ website has lots of information about the core principles and architectural concepts behind Axon. There is also a reference guide and a quick start guide.
https://axoniq.io/