## Axon Framework

**Axon Framework** is a Java [microservices framework](http://progressivecoder.com/microservices-frameworks-the-only-guide-you-need/) that helps you build microservices architecture in-line with **Domain Driven Design (DDD)** principles.

Apart from DDD, Axon Framework also allows you to implement microservices patterns such as **Command-Query-Responsibility-Segregation (CQRS)** and **Event-Driven Architecture.**

### **Aggregate**

As with DDD, Axon Framework also uses **Aggregate** as the core idea of an application. Aggregate is basically your **Domain Object**. In other words, it is the reason for the existence of the application.

### **Domain-Events**

Axon Framework also embraces the concept of **Domain-Events**. In other words, any command targeting an Aggregate creates one or more events. These events are also known as Domain-Events.

## Message Driven Aspect of Axon Framework

Axon is predominantly message-driven. In other words, various components connected in an Axon-based application communicate through messages.

Messages contain the message payload, meta-data and some sort of identifier. Also, the messages in Axon Framework are immutable. Therefore, they are safe to use in a multi-threaded distributed environment.

There are different types of messages:

### **Commands**

**Commands** are an intent to change the application state. They are mainly read-only POJOs.

The sender sends a **Command** to an application. The sender might not care about the command handling. However, it might want to know the outcome of the command. Therefore, you can return a result for a command that can be useful to the sender.

### **Events**

**Events** are basically objects that describe something that has happened in an application. From a **Domain-Driven-Design** perspective, events are applied on the **Aggregate** or the **Domain Entity**.

## Data Management Patterns for Microservices

## Database Per Service

## Saga Pattern

Saga pattern is the solution to implementing business transactions spanning multiple microservices.

A **Saga** is basically a sequence of local transactions. For every transaction performed within a Saga, the service performing the transaction publishes an event. The subsequent transaction is triggered based on the output of the previous transaction. And if one of the transactions in this chain fails, the Saga executes a series of compensating transactions to undo the impact of all the previous transactions.

To understand this better, let’s take a simple example. Assume that there is a food-delivery app. When a customer tries to order food, below steps can occur:

* Food Order service creates an order. At this point, the order is in PENDING state. A Saga manages the chain of events.
* The Saga contacts the restaurant via the Restaurant service.
* The Restaurant service attempts to place the order with the chosen restaurant. After getting a confirmation, it sends back a reply.
* The Saga receives the reply. And depending on the reply, it can Approve the order or Reject the order.
* The Food Order service then changes the state of the order. If the order was Approved, it would inform the customer with the next details. If Rejected, it will also inform the customer with an apology message.

As you can see, this is drastically different from the usual point-to-point call approach. This approach adds complexity. However, in my view, Sagas are a very powerful tool to solve some tricky challenges. But they should be used sparingly.

## API Composition

This pattern is a direct solution to the problem of implementing complex queries in a microservices architecture.

In this pattern, an API Composer invokes other microservices in the required order. And after fetching the results it performs an in-memory join of the data before providing it to the consumer.

As evident, the downside to this pattern is the use of inefficient in-memory joins on potentially large datasets.

## CQRS

CQRS or Command Query Responsibility Segregation is an attempt to get around the issues with API Composition pattern.

An application listens to domain events from other microservices and updates the view or query database. You can serve complex aggregation queries from this database. You could optimize the performance and scale up the query microservices accordingly.

The downside to this is increase in complexity. All of a sudden, your microservice should be handling events. This can cause latency issues where the view database is eventually consistent rather than always consistent. It can also increase code duplication.

## Event Sourcing

In event sourcing, you store the state of the entity or the aggregate as a sequence of state changing events. A new event is created whenever there is an update or an insert. The event store is used to store the events.

You can use this pattern in conjunction with CQRS. By doing so, a lot of challenges around event handling and maintaining query data can be solved.

However, as a drawback, this pattern imposes an unfamiliar programming style. Also, the data is eventually consistent which may not be suitable for some use cases.