Event-Driven cloud architectures, CQRS and Event Sourcing

Lorenzo Busin

August, 2019

CONTENTS

1 EVENT-DRIVEN ARCHITECTURES

1.1 What is an Event-Driven architecture?

Event-Driven architecture is a popular distributed asynchronous architecture pattern used to produce highly scalable and adaptable applications. It is made of highly decoupled and single-purpose event processing components that receive and process events.

1.2 What is an event?

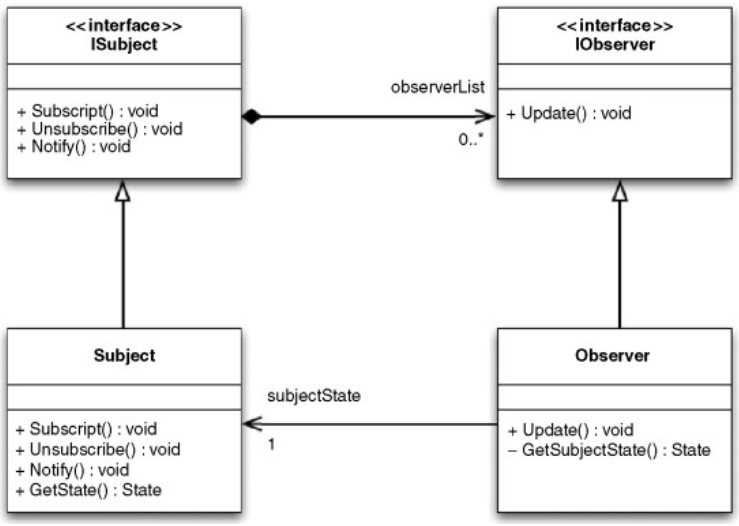
An event is an action recognized and handled by the software, often asynchronously. Events can be generated or triggered by the system, the user or in other ways. The software can also trigger its own set of events to forward information to other services or communicate with them. An application that changes its behavior in response to events is called "Event-Driven".
The main difference between commands and events is that the firsts say to the system *"*This is what you have to do" while the seconds just say *"This is happened"* and then is up to the handler to decide what to do.

1.3 What is Event-Driven programming?

Event-Driven programming is a programming paradigm where the flow of the program is determined by events such as user actions, sensor outputs or messages from other programs. This paradigm is mostly used in graphical user interfaces and other applications such as JavaScript web apps that are centered on performing actions in response to inputs.
In an Event-Driven application there is a main loop that listens for events and then triggers a callback function when one of those is detected. The events' listener is a common thing among applications, for this reason many programming frameworks take care of their implementation and expect the user to provide only the code for the event handlers.

1.4 Events communication

Events just communicate that something happened, so thanks to them you let a system know that something has happened. When an event occurs it will activate its handler which implements the execution logic. The result of the event's execution can turn into changes in the model-view or can trigger other handlers by forwarding it.
The *observer* pattern is a flawless example of how events communicate with each other.



A subject maintains a list of its observers and notifies them automatically of any state changes, usually by calling one of their methods. This design pattern is mainly used to implement distributed event handling systems and is also a key part in MVC pattern.

According to *Martin Fowler*, there are two ways of communicating between events: **Event notification** and **Event-Carried State Transfer**.

1.4.1 Event notification

To understand how Event notification works we can think of when our smartphone notifies us that a new message has arrived without adding any information. This happens when a system sends event messages to notify other systems of a change in its domain. The receiver knows something has changed but then issues a request back to the sender to decide what to do next, sending a request message and receiving a response for every event that happened.
The key element of Event notification is that the source system doesn't really care much about the response. So if the event's body does not need, Event notification is an excellent solution as it guarantees decoupling and high performance.

1.4.2 Event-Carried State Transfer

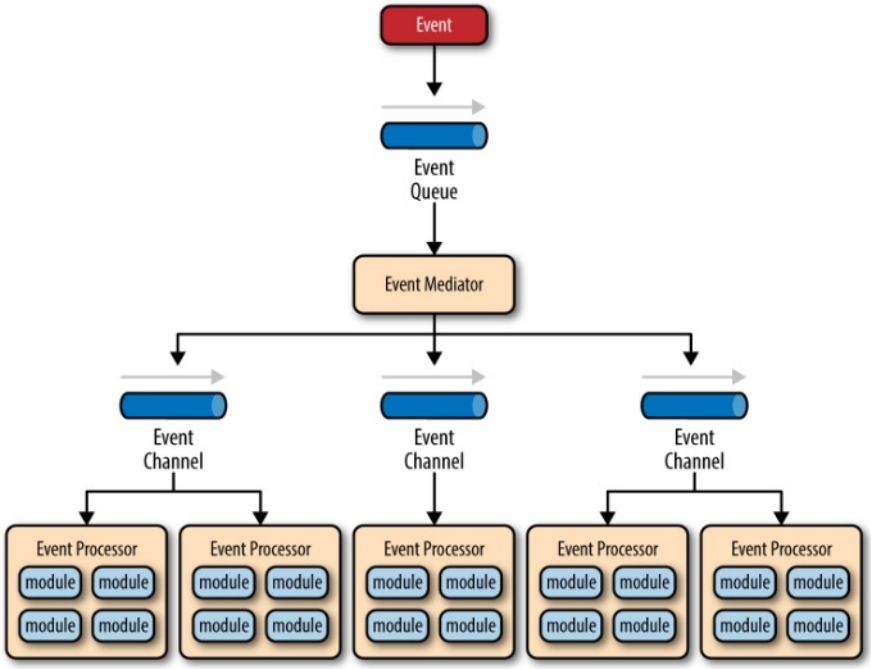
To understand how Event-Carried State Transfer works we can think of when our smartphone notifies us that a new message has arrived but in this case it shows other information like the sender's name and the message content. When something changes, the generated event contains the details of what has been changed. In this way the receiver does not need to ask the sender what has been changed.
A down-side of this model is that the messages that are sent contain more data but this fact reduces latency, because it is not necessary to query the source for further details. It does involve more complexity on the receiver since it has to sort out maintaining all the state, when it's usually easier just to call the sender to obtain more information.
So if the receiver needs to know extra data with the event object, Event-Carried State Transfer is an excellent solution as it guarantees better performance and lower demands on the supplier but more expensive communications.

1.5 Event-Driven architecture's topologies

Event-Driven architecture consists of two main topologies: **mediator** and **broker**. The mediator one is commonly used when you need to orchestrate multiple steps within an event through a central mediator, while the broker topology is used when you want to chain events together without the use of a central mediator.

1.5.1 Mediator topology

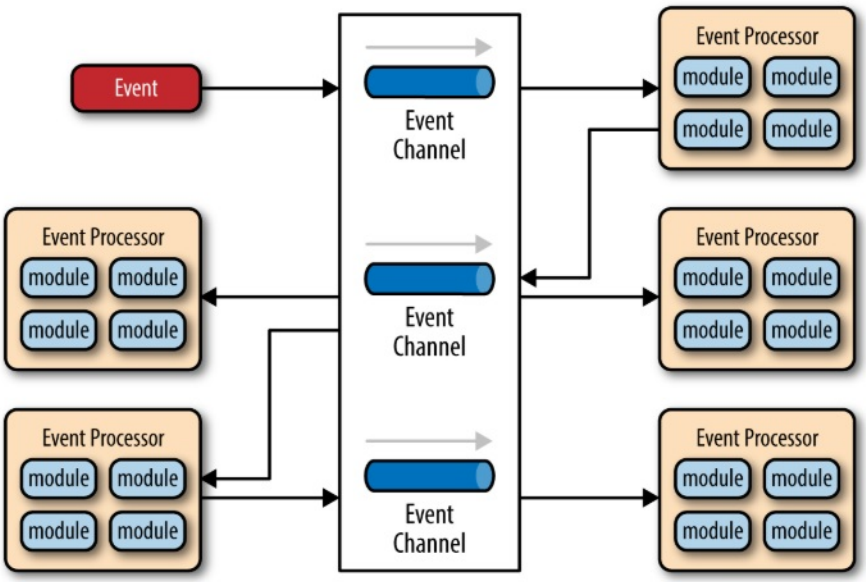
The mediator topology is useful for events that have multiple steps and require some level of orchestration to process the event. For example, a single event may consist of multiple steps that would require a certain level of orchestration to determine their order of execution.
There are four main types of architecture components within the mediator topology: event *queues*, an event *mediator*, event *channels* and event *processors*.
The event flow starts with a client sending an event to an event queue, which is used to transport the event to the mediator. It receives the initial event and orchestrates that event by sending additional events to event channels. Then, the event processors listen on the channels, receive the event and execute specific business logic.



There are two types of events within this pattern: an initial event and a processing event. The initial event is the original event received by the mediator, whereas the processing events are the ones that are generated by the mediator and received by the event-processing components. The event-mediator component is responsible for orchestrating the steps contained within the initial event. For each step, the mediator sends out a specific processing event to an event channel, which is received and executed by the event processor. It is important to note that the event-mediator doesn't perform any business logic.
The event channels can be either message queues or message topics; the seconds are widely used with the mediator topology because the events can be processed by multiple event processors.
The event-processors contain the business logic necessary to process events. They are self-contained, independent and highly decoupled that perform a specific task in the application or system. It is important to keep in mind that each event-processor component should perform a single business task and not rely on others to complete it.

1.5.2 Broker topology

The broker topology is different from the mediator topology because there is no central event mediator. The message flow is distributed across the event processor components in a chain through a lightweight message broker. This topology is useful when you have a simple event processing flow.



As you can see from the diagram, there is no central event-mediator controlling and orchestrating the initial event. Each event-processor component is responsible for processing events and publishing new events related to the action just performed. When an event processor receives an initial event, it performs its business logic and then publishes a new event to the broker, which would then be picked up by a different event processor.

1.5.3 Hybrid topology

The use of one or more mediators may cause a tightly coupled system with strong dependencies. Can be hard instead to implement and to handle a broker topology because of the complex event flow.
In complex and intricate systems the best choice is tho adopt an hybrid topology mixing the pros of both to efficiently handle the application workflow.

1.6 Pattern analysis

Characteristic	Rating	Description
Overall agility	⇧	Changes are generally isolated and can be made quickly with small impacts
Ease of deployment	⇧	Ease to deploy due to the decoupled nature of event-processor components. Broker topology is easier to deploy
Testability	⇩	It requires some specialized testing client to generate events
Performance	⇧	In general, the pattern achieves high performance through its asynchronous capabilities
Scalability	⇧	Scaling separately event-processors, allowing for fine-grained scalability
Ease of development	⇩	Asynchronous programming, requires hard contracts, advanced error handling conditions

Overall agility(High): is the ability to respond quickly to a constantly changing environment. Since event-processor components are single-purpose and completely decoupled from the others, changes are generally isolated to one or a few event processors and can be made quickly without impacting other components;

Ease of deployment(High): this pattern is easy to deploy due to the decoupled nature of the event-processor components. The broker topology tends to be easier to deploy than the mediator topology, primarily because the event mediator component is tightly coupled to the event processors: a change in an event processor component might also require a change in the mediator;

Testability(Low): while unit testing is not overly difficult and requires a testing client and a testing tool to generate events, others kind of test, like integration tests, requires more complicated logic to be implemented because in Event-Driven applications the events flow goes through different functions, tasks or services, so you need a specialized tool to develop them;

Performance(High): the pattern achieves high performance, especially through its asynchronous capabilities;

Scalability(High): is achieved in this pattern through highly independent and decoupled event handlers;

Ease of development(Low): development can be complicated due to the asynchronous nature of the pattern from the contract creation to the need for more advanced error handling, but also due to the complex events flow.

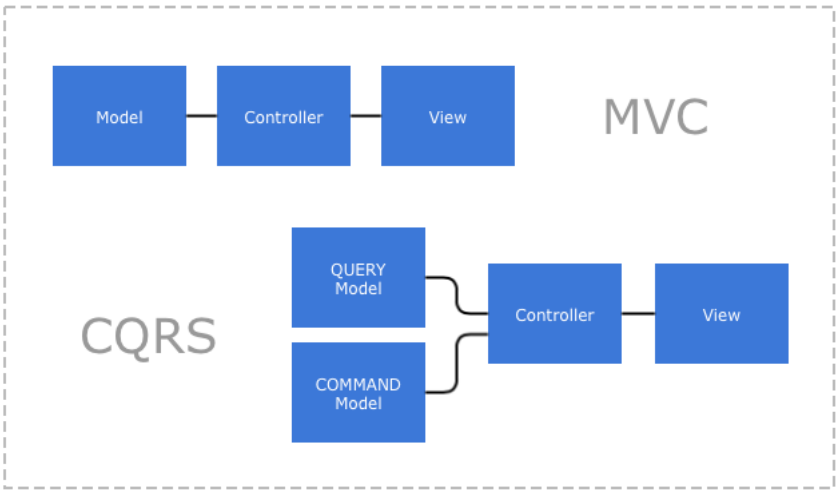
2 CQRS

2.1 What is CQRS?

Command Query Responsibility Segregation is an architectural pattern which separates the responsibility for modifying data(*Command*) from reading them(*Query*). The change that CQRS introduces is to split the conceptual model into separate models for updating and displaying data. The use of distinct models for writing and reading allows the infrastructure to easily scale to best fit the needs and allows to design and optimize each model for its responsibilities; this also allows the selection of the most appropriate technologies.

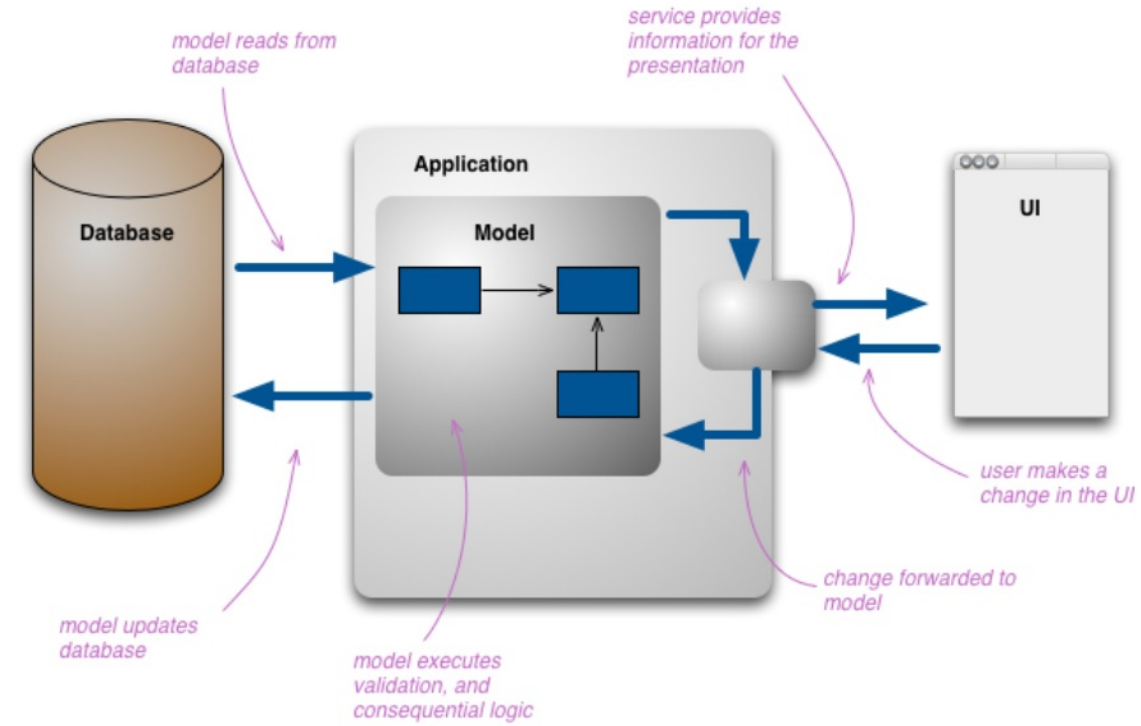
2.2 When using CQRS?

CQRS fits well with CRUD operations and with event-based programming models. The adoption of this design pattern is strongly recommended in complex domains, where a single model to handle both reads and writes gets too hard but it can be made easier by separating the two models. Also, often happens that the number of writes in a system is lower than the number of readings so this pattern guarantees high performance avoiding traffic slowdown.

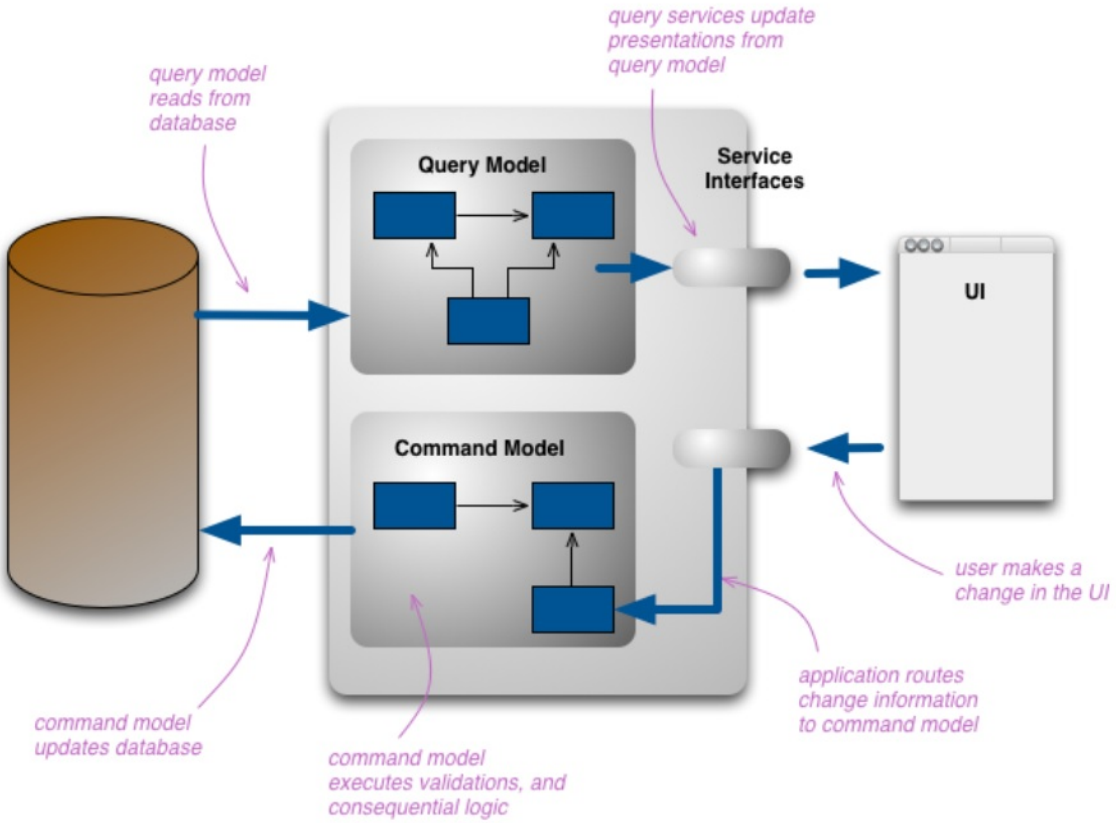


2.3 How to implement CQRS?

In complex applications usually happens that different services are integrated and the event's workflow from its birth until its complete execution often goes through different phases(calling different functions, different tasks etc.) while a reading operation just have to query a database to retrieves the needed data.
In these cases the ideal solution is to separate the system into two independent models: one for writing operations, which perform changes to the database, and one for reading operations, which do not change anything.



Example of application model that **not** adopts CQRS



Example of application model that adopts CQRS

3 EVENT SOURCING

"Event Sourcing ensures that all changes to application are stored as a sequence of events."
Martin Fowler

3.1 What is Event Sourcing?

The core idea of **Event Sourcing** is that whenever we make a change to the state of a system, we record that state change as an event and we can confidently rebuild the system state by reprocessing the events at any time in the future. It means you do not store the state of an object but all the events impacting its state. Then, to retrieve an object state you have to read the different events related to this object and applied them one by one.
Before storing the event you must validate it because you don't have to store events that generate errors or exceptions.
When working with an event log can be useful to build snapshots of the working copy so you don't have to process all the events when rebuild the system or every time you need to query the database. For this reason when a new event is stored triggers another function to update the views.

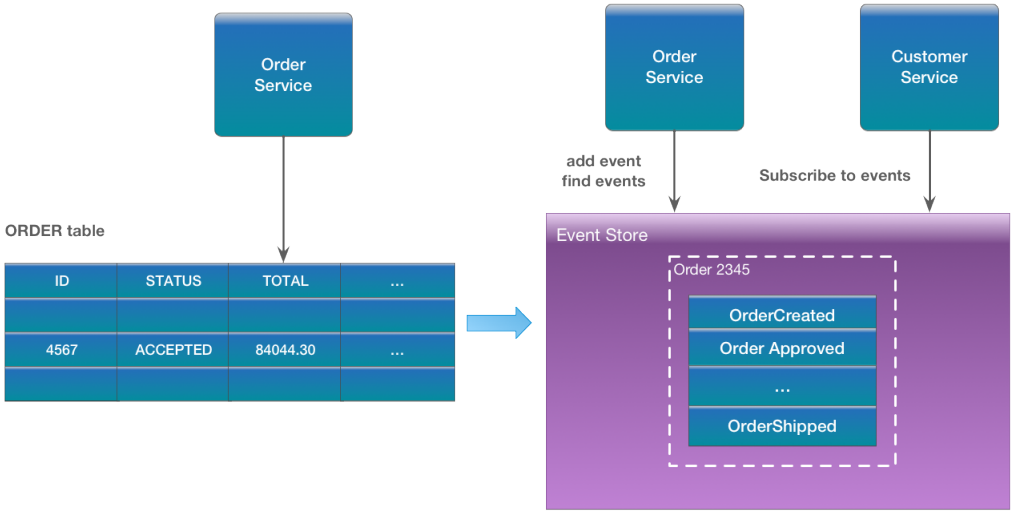
3.2 When using Event Sourcing?

Event sourcing can be used to:

- **Complete Rebuild**: you can completely discard the application state and rebuild it by re-running the events from the event store on an empty application.
- **Temporal Query**: you can determine the application state at any point in time. This can be used considering multiple time-lines (like branching in a VCS).
- **Event Replay**: if you find that a past event was incorrect, you can replaying from then with the correct event. The same technique can handle events received in the wrong sequence with systems that communicate with asynchronous messaging.

3.3 Event Store

Event Store is a database's table that contains all the events occurred from the begin. With event sourcing the event store becomes the principal source of truth and the system state is completely derived from it.
Is essential that the table can't be modified and the events that are replayed don't have to be stored again. This because the event store can reach huge dimensions and we must avoid to overload it.



In a complex system with billions of users the best choice is to implement an event store table for each aggregate object in order to keep track of the events that occurred to a single entity. In this way there are more event stores but this tables have less entries and this increase the speed of a query operation.

3.4 CQRS and Event Sourcing

Both patterns are frequently grouped together. Applying Event Sourcing on CQRS means persisting each event on the write part of our application. Then the read part is derived from the sequence of events.
Event Sourcing isn't required when you implement CQRS, but in most of the use cases, CQRS is required when you implement Event Sourcing. This because you have to retrieve a state in $O(1)$ without computing n different events.

4 Comparison between cloud providers