Distributed Event-Based System — AMQP, MQTT, and Kafka

Denis Conan

Septembre 2021
1. Motivations and objectives/requirements

2. Definition of Event-Based Systems

3. Which Topic-based filtering DEBS?

4. Topic-based filtering w/ OASIS AMQP v.0.9.1

5. Topic-based filtering w/ OASIS MQTT v.5.0

6. Topic-based filtering w/ Apache Kafka

7. Conclusion
Foreword

The content of these slides is extracted from the following references:

- https://kafka.apache.org/documentation/
1 Motivations and objectives/requirements

Foreword: We consider distributed architectures with application-layer messaging systems

1.1 E.g. IoT platforms
1.2 E.g. Web services with “Event sourcing”
1.3 E.g. Data-driven inference cycle
1.4 E.g. Autonomic computing—MAPE-K loop
1.5 Requirements
1.1 E.g. IoT platforms

- Communicate with lots of devices that are volatile
  \[\Rightarrow\] Scalability (\#clients, \#events)
  + Space-, time-, and synchronisation-decoupling

- E.g., Amazon IoT platform

https://aws.amazon.com/fr/iot-core/
1.1 E.g. IoT platforms II

- E.g. Microsoft Azure reference architecture

https://docs.microsoft.com/fr-fr/azure/architecture/reference-architectures/iot
1.2 E.g. Web services with “Event sourcing” I

- Routing, event-driven for high performance, scalability (number of events per second, GB per second)

1.2 E.g. Web services with “Event sourcing” II

- Kafka = Replayable log–based approach
  - React to events that are happening now
  - Push whole datasets to wherever they may be needed

1.3 E.g. Data-driven inference cycle

- Streams of events with data transfer and storage + functional programming

1.4 E.g. Autonomic computing—MAPE-K loop

- Model of the architecture at runtime for self-management: i.e. self-configuration, self-optimization, self-healing, and self-protection

1.5 Requirements

- Data production/consumption decoupling
  - **Space decoupling**: producers and consumers are distributed
  - **Synchronisation decoupling**: asynchronous and anonymous communication
  - **Time decoupling**: production and consumption at different times

- Scalability: in messages per second, in data per second, in clients (producers and consumers) at a given instant

- Data life-cycle management + filtering
  - Aggregation... is out of scope (See complex event processing and streaming)

- Adaptation to mobile, volatile, and heterogeneous environments

- One name for many “technologies”: distributed event-based systems, distributed publish-subscribe systems, distributed messaging service, message-oriented middleware, active databases, etc.
2 Definition of Event-Based Systems

2.1 Models of interaction and EBS
2.2 Constituents of an EBS
2.3 Notification filtering mechanisms
2.1 Models of interaction and EBS

2.1.1 “Request/Reply”
2.1.2 “Anonymous Request/Reply”
2.1.3 “Callback”
2.1.4 Studied in this lecture: “Event-Based”
2.1.5 Recap: Models of interaction and EBS
2.1.1 “Request/Reply”

- The consumer initiates the interaction
- The consumer knows the address of the producer for issuing the request
- The consumer waits for the reply: the call is synchronous
- The producer knows the address of the consumer
2.1.2 “Anonymous Request/Reply”

- The consumer initiates the interaction without knowing the address of the potential producers: there is an intermediate “entity” or “mechanism”
- The producers that can provide the requested data receive the request
- The producers reply to the consumer, i.e. they know the address of the consumer
- The consumer is willing to receive several replies
This is the Observable—Observer design pattern

- The consumer creates a data handler to manage registration and receptions
- The consumer knows the address of the producer and registers with it
- The producer sends the data updates to the consumer
- Consumer and DataHandler in the same process $\implies$ multi-threading
2.1.4 Studied in this lecture: “Event-Based”

- This is the Publish—Subscribe design pattern
- The consumer and the producers know the address of the PubSubSystem
- The consumer subscribes a filter to the PubSubSystem
- The producers publish data to the PubSubSystem
- The PubSubSystem applies subscription filters to route data and notify the relevant consumers
2.1.5 Recap: Models of interaction and EBS

<table>
<thead>
<tr>
<th>Adressee</th>
<th>Initiator</th>
<th>Consumer</th>
<th>Provider</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>Request/Reply</td>
<td>Callback</td>
<td></td>
</tr>
<tr>
<td>Indirect</td>
<td>Anonymous Request/Reply</td>
<td>Event-Based</td>
<td></td>
</tr>
</tbody>
</table>

- **Initiator**: describes whether the consumer or the provider initiates the interaction
- **Addressing**: indicates whether the addressee of the interaction is known or unknown at the beginning of the interaction
- **The trade-off is between the simplicity of request/reply and the flexibility of event-based interaction**
2.2 Constituents of an EBS

Event

Producer

F, F' : Filters
N : Notification

2. publish (N)

[optional]
1.a. advertise (F)

1.b. subscribe (F')

Consumer

3. notify (N)

Notification Service

Publish/Subscribe interface

Communication Implementation

Distributed Event-Based System — AMQP, MQTT, and Kafka
2.2.1 Terminology

- **Event**: any happening of interest that can be observed from within a computer
  - Event example: physical event, timer event, etc.
- **Notification**: an object that contains data describing the event
- **Producer**: a component that publishes notifications
- **Consumer**: a component that reacts to notifications delivered to them by the notification service
- **Subscription**: describes a set of notifications a consumer is interested in
- **Advertisement**: is issued by a producer to declare the notifications it is willing to send
2.2.2 Publish/subscribe interface

- Specifies the functionalities for decoupling producers from consumers

- Proposes the following operations:
  - `publish(n)`: a producer pushes notification \( n \) to the notification service
  - `advertise(F)`: a producer advertises that it will send notifications that match the filter \( F \)
  - `subscribe(F)`: a consumer subscribes to receive notifications that match the filter \( F \)
  - `notify(n)`: the notification service delivers the notification \( n \) to those consumers that have a matching subscription
2.3 Notification filtering mechanisms

2.3.1 Channels-based filtering
2.3.2 Topic-based (a.k.a. subject-based) filtering
2.3.3 Content-based filtering
2.3.4 (Type-based filtering)
2.3.1 Channels-based filtering

- Producers select a channel into which a notification is published
- Consumers select a channel and will get all notifications published therein
- Channel identifier is only the visible message part to the event-based service
- **Framework examples:** CORBA Event Service, CORBA Notification Service, OASIS AMQP standard version 0.9.1 (exchange mode “fanout”)...
2.3.2 Topic-based (a.k.a. subject-based) filtering

- Uses string matching for notification selection
- Each notification and subscription is defined as a rooted path in a tree of topics

**Example:**
- A stock exchange application publishes new quotations of FooBar under the topic: `/Exchange/Europe/London/Technology/FooBar`
- Consumers subscribe for `/Exchange/Europe/London/Technology/*` to get all technologies quotations

- The subject or topic is in message header, the content is “opaque”

**Example of solution:** OASIS AMQP standard version 0.9.1 (exchange mode “topic”), OASIS MQTT standard version 3.1.1, TIBCO Rendezvous, JMS (Java Message Queue), WebSphere MQ Publish/Subscribe (WMQPS), Apache Kafka, Apache Qpid, Spring/Pivotal RabbitMQ, Amazon IoT Core, Microsoft Azure IoT Hub...
2.3.3 Content-based filtering

- Filters are evaluated on the whole content of notifications.
- Example solutions: template matching, extensible filter expressions on name value pairs, XPath expressions on XML schemas, etc.

Example:

\[ m1: \{ (company, "Telco"), (price, 120) \} \]
\[ m2: \{ (company, "Telco"), (price, 90) \} \]

Filter F: \{ (company, =, "Telco"), (price, <, 100) \}
2.3.4 (Type-based filtering)

- Uses subtype inclusion to select notifications

- If a consumer subscribes to the type StockQuote, it will receive Technology quotations and other notifications that are sub-types of StockQuote

```
LondonStockMarket
  |   |
  |   |
StockQuote  ...
  |   |
  |   |
Technology  ...
```

- All the producers and consumers must agree on the hierarchy of types
  \[\Rightarrow\] Not flexible at all, thus not used \[\Rightarrow\] the title in brackets
3 Which Topic-based filtering DEBS? I

- **Reminder:** We study application-layer distributed-event based systems

- Topic-based filtering = filtering currently used by IT industry
  - Channels-based filtering: previous middleware like CORBA
  - Type-based filtering: not usable
  - Content-based filtering: more for complex event processing and streaming

1. **OASIS AMQP:** introduce the concept of “broker”

2. **OASIS MQTT:** better adapt to IoT constraints

3. **Apache Kafka:** introduce replayable log-based approach with storage, thus design pattern “Event Sourcing”
3 Which Topic-based filtering DEBS? II

In this slide, let’s take the application domain of the IoT


Also, RabbitMQ is one of the engines of: Amazon MQ, the Google Cloud Platform through bluemedora, IBM Cloud in the context of Web and mobile applications.
4.1 Overview of topic-based filtering of AMQP
4.2 Exchange, binding, and queue
4.3 Message delivery properties

The content of this section is extracted from

http://www.amqp.org/specification/0-9-1/amqp-org-download
and from

4.1 Overview of topic-based filtering of AMQP

- Lots of implementations: RabbitMQ, Apache Qpid, Microsoft Azure IoT Hub, etc.

- We propose to follow a tutorial on RabbitMQ
4.2 Exchange, binding, and queue

- **Queue** = name for a “post box” that lives inside the AMQP server
  - Messages are only stored inside a queue, never in exchanges
  - A queue is essentially a large message buffer
  - Many producers can send messages that go to one queue
  - Many consumers can try to receive data from one queue

- **An exchange** = A matching and routing engine
  - It inspects notifications (headers), and using its binding tables, decides how to forward these notifications to message queues or other exchanges

- **A binding key** = A criteria for notification routing
- **A binding** = A relationship (queue, exchange) with a binding key
4.2.1 Exchange of type “fan-out”

- The “fan-out” exchange type implements channel-based filtering
  - A message queue binds to the exchange with no arguments
    - Nothing on the arrow/binding from the exchange to the queue
  - A publisher sends notifications to the exchange
  - The notification is passed to the message queue unconditionally
4.2.2 Exchange of type “direct”

- The “direct” exchange type implements a simplistic form of topic-based filtering
  - A message queue binds to the exchange using a routing key K (a string)
  - A publisher sends to the exchange a notification with the routing key R
  - The notification is passed to the message queue if $K = R$
4.2.3 Exchange of type “topic”

- The “topic” exchange type works as follows:
  - A queue binds to the exchange using a binding key \( B \) as the routing pattern
  - A publisher sends to the exchange a notification with the routing key \( R \)
  - The notification is passed to the message queue if \( R \) matches \( B \)

- Routing key used for a topic exchange = 0 or more words delimited by dots

- Each word may contain [A--Z], [a--z], and [0--9]

- The binding key follows the same rules as the routing key with:
  - “*” that matches a single word and “#” that matches 0 or more words
4.2.4 Emulation of RPC-like call

Using message properties

- The AMQP 0-9-1 protocol defines a set of 14 message properties
- "deliveryMode": Marks a message as persistent or transient
- "contentType": Used to describe the mime-type of the encoding (e.g. application/json)
- "replyTo": Commonly used to name a callback queue
- "correlationId": Useful to correlate RPC responses with requests
4.3 Message delivery properties

- **Message acknowledgement**
  - What happens if a consumer fails while treating a message?
  - Consumer can choose to autoAck or not
    1. autoAck=true: Once delivered, the server immediately marks the message for deletion
       \[\implies \text{May be lost if the consumer fails}\]
    2. autoAck=false: The server waits for an explicit acknowledgement
       \[\implies \text{Memory leakage if the consumer forgot to send the acknowledgement}\]

- **Message durability**
  - When the server quits/crashes it forgets queues and messages unless told to do so
  - Two properties to make nearly sure that messages aren’t lost:
    1. Mark both the queue and messages as “durable”
    2. Mark messages of queue as “persistent”
4.3.1 More about message reliability

- A server forgets the queues and messages unless it is told not to

- Message reliability capabilities in a continuum:

  1. Mark queues and messages as durable = eventually stored in database
     - But, e.g., RabbitMQ doesn’t do `fsync(2)` for every message
       - Messages may be just saved to cache and not really written to the disk
  2. Clustering = Replicate broker for highly available queues (active replication)
     - Not in the AMQP specification, but provided in RabbitMQ for instance
  3. Publisher confirms =
     - Consumers acknowledge the treatment of a message
     - The broker sends a confirm message to the publisher when all the clients have acknowledged
5 Topic-based filtering w/ OASIS MQTT v.5.0

5.1 MQTT features
5.2 Topic filters w. wildcards and topic names
5.3 QoS—Message reliability
5.4 Disconnections

The content of this section is extracted from
https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html.
5.1 MQTT features

- Initially, a proposition made by IBM

- MQTT v.3.1: an OASIS standard in Oct. 2014
  Then, MQTT v.3.1: an ISO/IEC standard (20922:2016) in June 2016
  Today, MQTT v.5.0: OASIS Standard, March 2019

- It runs over TCP/IP, or over other network protocols that provide ordered, lossless, bidirectional connections
  - MQTT-SN was proposed for sensor networks in which these network conditions cannot be assumed, for instance using UDP

- Topic-based filtering with 3 levels of Quality of Service / message reliability

- Concept of sessions, in addition to connections

- Popular implementations: Eclipse Mosquitto and Paho, Amazon IoT Core, BevyWise, HiveMQ, Microsoft Azure IoT Hub, VerneMQ, etc.
5.2 Topic filters w. wildcards and topic names

- **Topic filter** = an expression contained in a subscription
  - ≈ AMQP binding key
  - “/,#,+” can be used in topic filters

- **Topic name** = the label attached to a message which is matched against the subscriptions
  - ≈ AMQP routing key
  - A server can change the topic name of a publish packet

- The forward slash (“/”) is used to separate each level within a topic tree and provide a hierarchical structure to the topic names
5.2 Topic filters w. wildcards and topic names II

- The number sign ("#") matches any number of levels within a topic
  - The multi-level wildcard represents the parent and any number of child levels
  - "#" must be specified either on its own or following a topic level separator
  - "#" must be the last character specified in the topic filter
  - E.g.
    - "sport/tennis/player1/#" matches "sport/tennis/player1", "sport/tennis/player1/ranking", and "sport/tennis/player1/score/wimbledon"
    - "sport/#" matches "sport", since "#" includes the parent level
    - "sport/tennis#" is not valid
    - "sport/tennis/#/ranking" is not valid
5.2 Topic filters w. wildcards and topic names III

- The plus sign ("+") matches only one topic level
  - The single-level wildcard can be used at any level in the Topic Filter, including first and last levels
  - Where it is used it must occupy an entire level of the filter
  - E.g.
    - "sport/tennis/+" matches "sport/tennis/player1" and "sport/tennis/player2", but not "sport/tennis/player1/ranking"
    - "sport/+" does not match "sport" but it does match "sport/"
    - "+" and "+/tennis/#" are valid
    - "sport+" is not valid
    - "/finance" matches "+/+" and "/+", but not "+".
5.2 Topic filters w. wildcards and topic names IV

- Special character “$”
  - Server implementations may use topic names that start with a leading “$” character for other purposes
    - E.g. “$SYS/” has been widely adopted as a prefix to topics that contain server-specific information or control APIs
  - The server must not match topic filters starting with a wildcard character (“#” or “+”) with topic names beginning with “$”
  - The server should prevent clients from using such topic names to exchange messages with other Clients
5.3 QoS—Message reliability

- Published messages have associated quality of service (QoS)
  - QoS0/“At most once”: best efforts of the operating environment
    - Message loss can occur
    - This level could be used, for example, with ambient sensor data where it does not matter if an individual reading is lost as the next one will be published soon after
  - QoS1/“At least once”: assured to arrive but duplicates can occur
  - QoS2/“Exactly once”: assured to arrive exactly once
- Client and broker store session state in order to provide QoS levels 1 and 2

- A subscription contains a topic filter and a maximum QoS
  - The server might grant a lower maximum QoS than the subscriber requested
  - QoS properties are associated with a single session (consecutive connections)
  - When filters overlap, the delivery respects the maximum QoS of all the matching subscriptions
5.3.1 Delivery of QoS0/“At most once” messages

- The delivery protocol is concerned solely with the delivery of an application message from a single sender to a single receiver.
- No response is sent by the receiver and no retry is performed by the sender.
- The sender sends a publish packet with QoS=0, DUP=0\(^1\)
- The receiver accepts ownership of the message when it receives the publish packet.

---

1. DUP is set to 1 when the sender knows it is a duplicate.
5.3.2 Subscription and publication with QoS0/“At most once”

- In this scenario, let us consider that the server grants a maximum QoS0.
- A QoS1/“At least once” message might either get lost or duplicated.
- A QoS2/“Exactly once” message might get lost but the server should never send a duplicate.
5.3.3 Delivery of QoS1/“At least once” messages

- A QoS1 publish packet has an Id and is acknowledged

- The Sender:
  1) assigns an Id and sends a publish packet containing Id, QoS=1, DUP=0

- The Receiver:
  1) acknowledges, having accepted ownership of the message
  2) treats any incoming publish packet with same Id as being a new publication

*(1) The receiver is not required to complete the delivery before sending the PUBACK
*(2) The sender knows that ownership of the message is transferred to the receiver
5.3.4 Subscription and publication with QoS1/“At least once”

- The server grants a maximum QoS1

- A QoS0 message matching the filter is delivered at QoS0/“At most once”

- A QoS2 message published to the same topic is downgraded by the server to QoS1

- Client might receive duplicate copies of the message
5.3.5 Delivery of QoS2/“Exactly once” messages

- The receiver acknowledges receipt with a two-step acknowledgement process.

- The Sender:
  1) assigns an Id and sends a publish packet containing Id, QoS=2, DUP=0
  3) treats the publish packet as “unack” until it has received the PUBREC
  4) sends a PUBREL (release) packet when it receives a PUBREC packet
  5) treats the PUBREL packet as “unack” until it has received the PUBCOMP (complete)
  6) do not re-send the publish packet once it has sent the PUBREL

- The Receiver:
  1) responds with a PUBREC, having accepted ownership of the message
  2) until it has received the corresponding PUBREL packet, acknowledges any subsequent publish packet with the same PUBREC
  3) responds to a PUBREL packet by sending a PUBCOMP
5.3.5 Delivery of QoS2/“Exactly once” messages

Sender

- Store message with <Id>
- Send PUBLISH with QoS=2, DUP=0, <Id>
- Send PUBREC <Id>
- Send PUBREL <Id>
- Send PUBCOMP <Id>
- Discard <Id>

Receiver

- Store <Id> and then initiate onward delivery
- Receiver acknowledges content i.e. asks for "stop sending content"
- Discard <Id>, no more duplicate
- Receiver sends "<Id> complete", i.e. "<Id> can be removed"

...sender may resend content

Discard message (*1) and store PUBREC received <Id>
Sender stops resending content
Sender sends "<Id> can be released"

Discard <Id>

(*1) The sender knows that ownership of the message is transferred to the receiver
5.3.6 Subscription and publication with QoS2/“Exactly once”

- The server grants a maximum QoS2
- A topic filter at QoS 2 = delivery of a message at the QoS with which it were published
5.4 Disconnections

5.4.1 Sessions
5.4.2 RETAIN flag in a publish packet
5.4.3 Message ordering
5.4.1 Sessions

- Session = A stateful interaction between a client and a broker

- Some sessions last only as long as the network connection, others can span multiple consecutive network connections

- When a client connects with CleanSession set to 0, it is requesting that the server maintain its state after disconnection

- When a client has determined that it has no further use for the session, it should connect with CleanSession set to 1 and then disconnect

- A server is permitted to disconnect a client that it determines to be inactive or non-responsive at any time
5.4.2 RETAIN flag in a publish packet

- If the RETAIN flag is set to 1, the server must store the message and its QoS so that it can be delivered to future subscribers.

- When a new subscription is established, the last retained msg (if any) is sent.

- If the server receives a QoS0/“At most once” message with the RETAIN flag, it discards any message previously retained for that topic.
  - It should store the new message as the new retained message for that topic.
  - But it may choose to discard it at any time.
    - If this happens there will be no retained message for that topic.
5.4.3 Message ordering

- When a client reconnects with CleanSession set to 0 when connecting, both the client and server must re-send any unacknowledged publish packets (where QoS > 0) and PUBREL packets using their Ids.

- A server must by default treat each topic as an “Ordered Topic”.
  - It may provide an administrative or other mechanism to allow one or more topics to be treated as an “Unordered Topic”.
6 Topic-based filtering w/ Apache Kafka

6.1 Cluster-based architecture
6.2 Topics as structured commit logs
6.3 Consumer groups
6.4 Fault tolerance
6.5 From Event Collaboration to CQRS
6.1 Cluster-based architecture

- Kafka is run as a cluster of servers that can span multiple datacenters
- The Kafka cluster stores streams of records in categories called topics
- Each record consists of a key, a value, and a timestamp
- Producers publish a stream of records to one or more Kafka topics
- Consumers consume an input stream from one or more topics
6.2 Topics as structured commit logs

- A topic = stream of records = partitioned log = structured commit log
- Each partition is an ordered, immutable sequence of records that is continually appended to
- A partition must fit on the server that hosts it
- A topic may have many partitions \(\Rightarrow\) an arbitrary amount of data
- Partitions act as the unit of parallelism
- Records are assigned a sequential id. number called the offset
- Consumers can consume records in any order they like
6.3 Consumer groups

- Consumers join groups, which are labelled with a consumer group name.
- Each record published to a topic is delivered to one consumer within each subscribing consumer group.
  - If all the consumers are in the same group, then records are load balanced.
  - If all the consumers are in different groups, then records are replicated.
- There cannot be more consumers in a group than partitions.
6.4 Fault tolerance

- Each partition is replicated across a configurable number of hosts.
- One host acts as the “leader” and the others act as “followers”
  - Usually, each host acts as a leader for some of its partitions and as a follower for others.
- The process of maintaining membership in the group is handled by Kafka dynamically
  - If an instance joins a group, it takes over partitions from existing instances.
  - If an instance dies, its partitions are distributed to the remaining instances.
- Total order over records within a partition, not between different partitions in a topic.
6.5 From Event Collaboration to CQRS

6.5.1 Design pattern “Event Collaboration”
6.5.2 Design pattern “Event Sourcing”
6.5.3 Design pattern “Command Query Responsibility Segregation”
6.5.1 Design pattern “Event Collaboration”

- Each (micro-)service listens events and creates new events
- No service knows the other services nor owns the entire workflow
  - This is called a **choreography**
    - ≠ An orchestration, in which a process controls the whole workflow

Image from B. Stopford, “Designing Event-Driven Systems: Concepts and Patterns for Streaming Services with Apache Kafka”, O’Reilly, 2018

6.5.2 Design pattern “Event Sourcing”

- [https://martinfowler.com/eaaDev/EventSourcing.html](https://martinfowler.com/eaaDev/EventSourcing.html)
- Make the events “the source of truth”: include commands into Kafka log
- Use Kafka as a data store of the events in the order of their creation
- Fault-tolerance using passive replication by rollback recovery
  - Consider (micro-)services that have a pseudo-deterministic execution
    - Any state of the execution can be computed from an initial state and the sequence of events that leads to this state
  - Periodic creation of snapshots + replay of events in order
6.5.3 Design pattern “Command Query Responsibility Segregation”

- Separate the write path from the read path and links them with an asynchronous channel
- Provide adequate view(s) of the (micro-)service and query the view(s)

Image from B. Stopford, “Designing Event-Driven Systems: Concepts and Patterns for Streaming Services with Apache Kafka”, O’Reilly, 2018

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

- Distributed Event-Based Systems for acquiring data

- Other names of this architectural style: Distributed Publish Subscribe System, Distributed Messaging Service

- Interaction mode = event-based
  - Producers initiate the exchanges of data (push mode)
  - Producers do not know the potential consumers when pushing

- Properties of this architectural style =
  - Space decoupling: Producers and consumers do not know each others
  - Time decoupling: Producers and consumers do not need to be active at the same time
  - Synchronisation decoupling: asynchronous communication (producers and consumers are not blocked while producing or being notified, respectively)
AMQP and MQTT = server-based architecture using mostly topic-based filtering

MQTT for small devices in the context of the IoT

- Different QoS:
  - 0/“at most once” = best effort
  - 1/“at least once” = assured to arrive but duplicates can occur
  - 2/“exactly once” = assured to arrive exactly once

- Concept of session, which spans several connections

Kafka looks more like a distributed commit logging system

- A topic is a set of partitions, which are append-only files
- From event collaboration to event sourcing to CQRS