**Event Sourcing**

On a high-level, Event Sourcing deals with storing the state of the application in the form of the domain events.

# CQRS

CQRS stands for Command Query Responsibility Segregation.

In a nutshell, CQRS works on the principle that the model to update information can be different to the model used to read that information.

But why do you need it?

The need for this pattern arises from the Database-per-service pattern. The Database-per-service pattern says that each microservice should be responsible for its own data. While it sounds neat, this introduces a situation where it can become problematic to implement queries on data scattered across microservices.

Consider an example to understand this. Suppose we are running a food delivery application. When an order is placed by a customer, below services could be triggered:

* *Order Service* to create an order in the system.
* *Restaurant Service* to contact the restaurant and place the order in the restaurants queue
* *Delivery Service* to assign a delivery boy to the order and provide him/her the location to the restaurant and delivery location
* *Payment Service* to handle the payment for the order.

All these services deal with a particular functionality. Each of these services can have their own database. They can have their own set of business rules.

However, if we want to provide a view to the customer about the entire Order lifecycle. This view should show everything that has happened in the life of an order.

Arguably, such a view will need data from each of the above services.

1. One way is to aggregate the data from each of those microservices by calling interfaces exposed by the services. However, this leads to unwanted in-memory joins. It also leads to tight coupling between the aggregator application and the individual microservices.
2. The other approach is to maintain a separate query store. This query store maintains the query specific view. Basically, this query store is built by listening to domain events from individual microservices. In the case of Order lifecycle example, such a query store could maintain an up-to date view of the order’s lifecycle ready to be served whenever the user requests.

CQRS makes sense when you see a need to split your conceptual or domain model into separate model for updates and reads.

# Event Sourcing and CQRS

# Event Sourcing and CQRS are used in conjunction with each other.

# In fact, it is often a critical requirement that an event sourced system also uses CQRS. It is harder to query an event sourced system. And hence, an efficient query store might be an indispensable need.

# Some of the advantages of using Event Sourcing and CQRS are:

* You can scale up the command (or update) side separately than your query (or read) side. This could be a great advantage for a system where *reads* outnumber *writes* by a huge margin.
* You can chose different strategies for *event store* and *query store*. For example, event store can be a typical RDBMS. You can handle queries using NoSQL database (like MongoDB).
* Using Event Sourcing and CQRS together, you can basically get rid of data aggregation pattern.

# Event Sourcing and CQRS — A Classic Implementation Approach

# Event Sourcing and CQRS are basically two separate patterns serving a common use-case. Ideally, we should implement them as two separate applications. C:\Spring Boot-Microservices-Introduction\Day2\Event Sourcing-CQRS.png

Let’s understand what’s happening here..

* We have a *command handler.* Basically, all command requests are received here.
* The *command processing* part takes care of handling all the commands and generating appropriate events. The events are persisted in the event store. Of course, validations and enforcement of business rules care performed before the events are persisted. Also, after the events are persisted, they are published on a message queue.
* The messaging queue could be a broker like RabbitMQ or Kafka.
* The Query Processing application listens to the events. Basically, this application takes the event payload and persists the data in the query store based on the required read models.
* The *query handler* part handles the incoming read requests. It retrieves the data from the query store and outputs it.

# Implement Event Sourcing and CQRS

# Axon Framework and Axon Server (Enterprise) for event-driven microservices.

# For Event Sourcing and CQRS use here Axon Framework. Since we are using the same application for both Event Sourcing and CQRS, we will be using RDBMS (in this case an in-memory H2 database) as both a event store and query store. To connect to the database, we will leverage Spring Data JPA.

# Download AXON server as AxonQuickStart.zip

# <https://axoniq.io/download>

# To strat the Axon Server

# Put the server JAR file axonserver-4.3..jar in

# the directory where you want it to start, and start it using from command line

# java -jar axonserver-4.3.jar

# You will see that it creates a subdirectory data where it will store its

# information. Open a browser to with URL http://localhost:8024 to view the dashboard.

# Axon Server provides two servers; one serving HTTP requests, the other gRPC.

# By default these use ports 8024 and 8124 respectively.

# The HTTP server has in its root context a management Web GUI, a health

# indicator is available at "/actuator/health", and the REST API at "/v1'. The

# API's Swagger endpoint finally, is available at "/swagger-ui.html", and gives

# the documentation on the REST API.

**CQRS Components**

Defining the Query Entity

### To show an example of CQRS, we will create a separate entity for handling the query view. We will call it AccountQueryEntity.

It is basically just a JPA entity containing the fields we want in the query view.

### Query Entity Manager

### Now, we want a way to populate the query entity. The idea behind this is that whenever there is a domain event on a particular aggregate, we should store the event data in the event store. On the other hand, the latest data for that particular instance of the aggregate should be stored in the query entity. All the query requests will then be served from the query entity.

Basically, we need a handler that can hook into the aggregate events and populate the query view and defined as **AccountQueryEntityManager**

Request Data Flow🡪

 Inject an instance of *AccountRepository*into the*AccountQueryEntityManager.*

 Inject an instance of *EventSourcingRepository* for **AccountAggregate**. The *AccountAggregate* is the aggregate class for the entity managed by Axon.

 A method is declared to handle the events occurring on the aggregate. This particular method is annotated with *@EventSourcingHandler*. All our events extend the BaseEvent class. Therefore, we just use the BaseEvent instance as the method input.

 Next, we simply use the instance of *EventSourcingRepository* to extract the correct *AccountAggregate* object from the event store.

 If found, we use the values from the *AccountAggregate* object to populate the *AccountQueryEntity* object. If not found, we create a new instance of the *AccountQueryEntity*.

 Finally, we use the *AccountRepository* to save the *AccountQueryEntity* instance to the query store.

to make this work, the **EventSourcingRepository** should be made available in the Spring context. **AxonConfig** is defined for this task.

## The Service Layer

All the important stuff to handle **Event Sourcing and CQRS** is done. The next step is to expose the queries using the **AccountQueryEntity**. For that, we will create a service class as **AccountQueryService**

**AccountQueryController** is where the REST end-point that should be used by the consumers to fetch the Bank Account information from our application.

The first method annotated with @GetMapping. This method uses an instance of the service class we created in the last step to fetch the account information. The same is then returned to the caller. The other method annotated with @GetMapping simply gets a list of events for a particular account.

## Test the Application

1. Start the AXON Server.
2. Chcek the health of application on http://localhost:8090/actuator/health
3. Open the page <http://localhost:8090/swagger-ui.html> and http://localhost:8090/v2/api-docs
4. Make the following actions with swagger ui/curl/RESTED/browser
   1. Create new account with initial balance of 14560 Rs.
   2. Credit 800 Rs to the account.
   3. Withdraw 500 Rs from the account.
5. Open the H2 database console on <http://localhost:8090/h2-console>.
6. Connect to datasource url jdbc:h2:mem:appdata
7. Check the entries in appdata database, on the  **ACCOUNT\_QUERY\_ENTITY** table.

Either you test with curl or rested plug-ins

**Create a new account**

curl -X POST "http://localhost:8090/bank-accounts" -H "accept: \*/\*" -H "Content-Type: application/json" -d "{ \"currency\": \"RS\", \"startingBalance\":100}"

OR with POST method

<http://localhost:8090/bank-accounts> with request body as

{ startingBalance: 1700, currency: Rs.}

The output is account number value as **bb788e0b-10a8-4c5e-9525-91ff41ded4bd**

**Credit the amount to Accont number created above**

curl -X PUT "http://localhost:8090/bank-accounts/credits/bb788e0b-10a8-4c5e-9525-91ff41ded4bd" -H "accept: \*/\*" -H "Content-Type: application/json" -d "{ \"creditAmount\": 1200, \"currency\": \"Rs.\"}"

OR

Use PUT method with

http://localhost:8090/bank-accounts/credits/bb788e0b-10a8-4c5e-9525-91ff41ded4bd with RequestBody as

{ creditAmount: 1200, currency: Rs.}

**Debit the amount on account with the same account Number**

curl -X PUT "http://localhost:8090/bank-accounts/debits/bb788e0b-10a8-4c5e-9525-91ff41ded4bd" -H "accept: \*/\*" -H "Content-Type: application/json" -d "{ \"currency\": \"Rs.\", \"debitAmount\": 160}"

OR

<http://localhost:8090/bank-accounts/debits/bb788e0b-10a8-4c5e-9525-91ff41ded4bd>

with PUT method and request body as

{ debitAmount: 700, currency: Rs.}

**Query using the account Query API**

Get the account details for account number as bb788e0b-10a8-4c5e-9525-91ff41ded4bd

curl -X GET "http://localhost:8090/bank-accounts/bb788e0b-10a8-4c5e-9525-91ff41ded4bd" -H "accept: \*/\*"

OR

http://localhost:8090/bank-accounts/bb788e0b-10a8-4c5e-9525-91ff41ded4bd

##### Response body

{

"id":"bb788e0b-10a8-4c5e-9525-91ff41ded4bd",

"accountBalance": **2340**,

"currency":"RS",

"status":"ACTIVATED"

}

**Query for the Listing the Events on account**

curl -X GET "http://localhost:8090/bank-accounts/bb788e0b-10a8-4c5e-9525-91ff41ded4bd/events" -H "accept: \*/\*"

OR

<http://localhost:8090/bank-accounts/bb788e0b-10a8-4c5e-9525-91ff41ded4bd/events>

in the browser

**Response**

[

{

"id":"bb788e0b-10a8-4c5e-9525-91ff41ded4bd",

"accountBalance": **100**,

"currency":"RS"

},

{

"id":"bb788e0b-10a8-4c5e-9525-91ff41ded4bd",

"status":"ACTIVATED"

},

{

"id":"bb788e0b-10a8-4c5e-9525-91ff41ded4bd",

"creditAmount": **1200**,

"currency":"Rs."

},

{

"id":"bb788e0b-10a8-4c5e-9525-91ff41ded4bd",

"creditAmount": **1200**,

"currency":"Rs."

},

{

"id":"bb788e0b-10a8-4c5e-9525-91ff41ded4bd",

"debitAmount": **160**,

"currency":"Rs."

}

]

**Query the axon server on** [**http://localhost:8024/#queries**](http://localhost:8024/#queries) **and**

[**http://localhost:8024/#commands**](http://localhost:8024/#commands)

**Check and Verify with H2 Database console on** [**http://localhost:8090/h2-console**](http://localhost:8090/h2-console)

**Connect to** jdbc:h2:mem:appdata in H2 console.

**DOMAIN\_EVENT\_ENTRY** table. Is the event store. You have events recorded here.

The query store is denoted by the table **ACCOUNT\_QUERY\_ENTITY** table.

Run the query

**SELECT \* FROM ACCOUNT\_QUERY\_ENTITY;**

**Select \* from DOMAIN\_EVENT\_ENTRY**;

The event store shows all the events on a particular Account. And the query store only shows the current state of the Account.

-----------