Bipin Kumar

Shael.dhn88@gmail.com

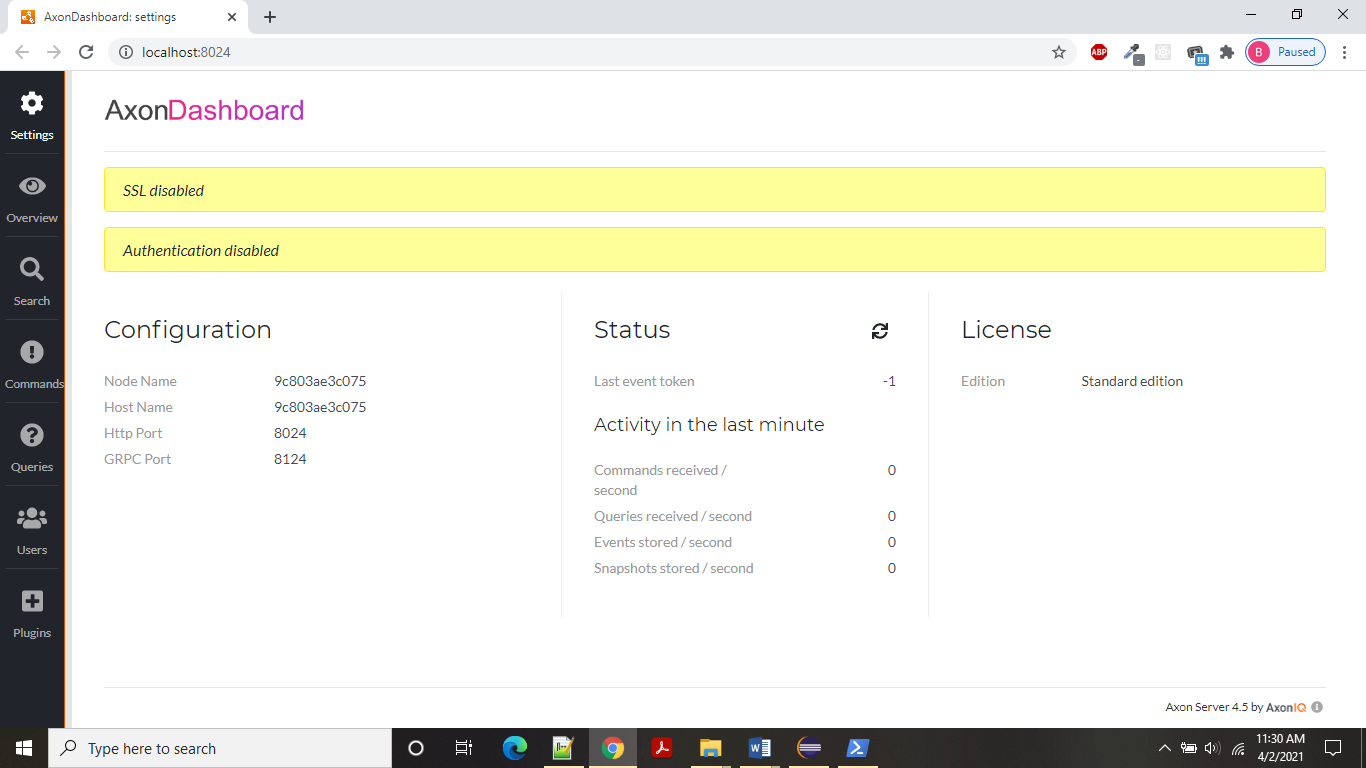
Event Sourcing And CQRS

ES And CQRS Doc

**Docker image for Axon server**:

docker run -d --name axonserver -p 8024:8024 -p 8124:8124 axoniq/axonserver

It will start axon server on <http://localhost:8024>



Event Sourcing:

Whenever we are using Domain driven design pattern every bound context has Aggregate. Let us assume we are writing an application of account management system. We have AccountAggregate class. So, whenever a command comes which change state of aggregate an event should be raised so that other bounded context will be notified and so that they can do necessary changes.

Suppose we have C1 bounded context which raised an event and send message to MQ then other bounded context let’s assume C2 will consume that event and after that MQ will remove that event and it will not be persisted in any DB or somewhere else. So, Event sourcing is saying whenever is event raised event should persisted or saved somewhere let say in DB which is called Event source.

So, whenever we want current state of aggregate we can get from event source. Now let assume we want to know the current state of aggregate having account id 100, What we can do we can write a utility which will query the event source and get all the event and it will apply those changes to aggregate. So, if you see the code as per command we have written command handler which raise event and also, we have written event handler for those command.

* The concept behind event sourcing is that every change to the state of the application should be captured.
* In other words, every action performed on an application entity should be persisted.
* Then we can query those events. We can also use the list of events to reconstruct the current state of the object.
* Let’s assume the following transaction occur on a particular account.
* Creation of account with initial balance of 1000 USD.
* Account set to ACTIVE status.
* Deposit of 550 USD in the account.
* Withdrawal of 200 USD.
* Withdrawal of 500 USD.
* Withdrawal of 1000 USD.
* Account set to HOLD status because of negative balance.
* Deposit 1000 USD to the account.
* Account set to ACTIVE status.
* In a typical data store, the account details, after all the above transactions have occurred, will be as follows:

**Account# Amount Currency Staus**

ABCD45464 850.00 USD ACTIVE

So, if we use event sourcing system we can persist all the events and in case of DB corruption or due to any prod issue we can query those events and recreate the data.

* The traditional way is to persist the current state of the aggregate. However, in event sourcing the aggregate is stored as a sequence of events. Basically, for every state change a new event is created and added to the event store. Event store maintains a complete history of the aggregate. This makes it extremely easy to reconstruct the historical state of the object. In case of any production issues, we can reconstruct the state of the object at the time of issue to investigate.

**CQRS (command query responsivity segregation):**

In DDD, we are working on bounded context and it has aggregate. Any request which change the state of aggregate is called command. There may be some request which use for retrieving the state of aggregate that is called as query.

In CRUD, create, update and delete are commands where as read is query. In traditional way we are saving the state of aggregate is database which is in normalize form.

Let’s take example we have two table customer and accounts in normalize form. Can we delete customer without deleting accounts, No DB will give some error. Suppose if we are able to delete customer table in that case data in DB will be inconsistent. So that’s why we required normalization. Now let’s assume we have query to read customer data, then we have query with join to fetch data of customer and account data. And if there are multiple table then in that case query will be more complex and will take more time.

Now suppose if one bounded context divided into two application i.e. one for command and one for query request. Suppose if command comes to change the state of application it will fire a event which will save the data into it’s DB which has normalize form. Similarly, our query application will have its own database and event handler which will save data in its own database. But query application needs to return data to user in some specific format. That’s why here data is saved in some format (not in normalize form) which we can easily send to user. So, we can say query application will have its own DB which is design as per UI. Similarly, in case of reporting application it will very difficult to generate report by joining all the bounded context data. But using CQRS it will be very easier.

Implementing CQRS using Axon:

We can imagine Axon server as MQ which behind the scene work as MQ, which have command bus and event bus listening on some port. In our microservice we have axon client connector which will send all the request from microservice to axon server using axon APIs. So based on data come to which bus it will invoke that handler means if data come to command bus command handler will be executed by axon client and if data come to event bus event handler will be executed.

1. Add below dependency in pom.xml of command app.

<dependency>

<groupId>org.axonframework</groupId>

<artifactId>axon-spring-boot-starter</artifactId>

<version>4.2</version>

<!-- <exclusions>

<exclusion>

<groupId>org.axonframework</groupId>

<artifactId>axon-server-connector</artifactId>

</exclusion>

</exclusions> -->

</dependency>

1. Now we have AccountAggregate class which is annotated with @Aggregate.Its lifecycle will be managed by axon, similarly we have id field which is annotated with @AggregateIdentifier
2. Similarly, once changes of state change then for this we have some method which will annotated with @CommandHandler which will take commands as parameter and based on that event will be fired and method which is annotated with @EventSourcingHandler will be called based on event fired.
3. We have basecommand class which is having a field called id and annotated with @TargetAggregateIdentifier annotation. And this class is extended by all the command class.
4. For all command we required event handler so we have base event class which extended by all the events classes.
5. In controller we are having service class which will inject commandgateway bean which will publish or send data to axon command bus. From there it will tell the client to execute the method which is annotated with commandhandler annotation.
6. Axon will by default store data to its data base which by default use h2 data source.
7. we have config class which return eventsourcingrepository bean in config package. It is by default use for persiting event to eventsorcing DB.