Backend Assignment Steeleye

This repository is of Backend Assignment that was provided by Steeleye for One-year internship.

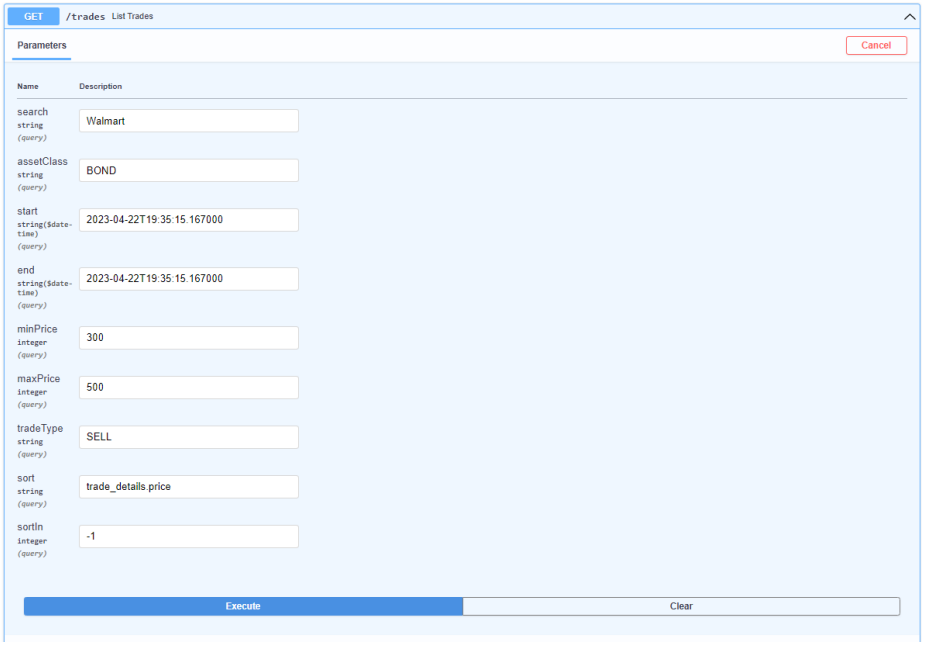
Name: Shubham Mishra
Registration No.: 12018763
BTech (CSE)

First run "uvicorn main:app --reload".
Then open the server in swaggerUI. easy testing use swaggerUI of fast API by "http://127.0.0.1:8000/docs"

API PARAMETERS

This is about input you can give to different parameters to every API.

1. **/trades** (List Trades)



1.1). Get all trades click execute leaving all parameters blank.
1.2). **search**: It will search for value in "counterparty", "instrumentId", "instrumentName" and “trader".
Input example: Walmart, BSA

Below are the Filtering options:
1.3). **assetClass**: filtering according to “assets. Input”.
Input example: BOND, FX, EQUITY.

1.4). **start and end**: date will be accepted in datetime format.
Input example: 2023-04-22T16:30:47.27

1.5). **minPrice and maxPrice**: Filter according to price <= and >= respectively. Will except integer.

1.6). **tradeType**: will take "BUY" and "SELL".

1.7). **sort**: according to which object member you want to sort the list of Trades.
For price: type "trade_details.price"
For quantity: type "trade_details.quantity"
For tradetype: type "tradeDetails.buySellIndicator"
For other members you can simply use its value like assetClass, instrument_id.

1.8). **sortIn**: to sort list of trades in ascending or descending order. Accept value in integer value 1 and -1.
Use "1" to sort in ascending order.
Use "-1" to sort in descending order.

About Project

I have used FastAPI, pydantic, MongoDB database and motor as a MongoDB driver.

The File that has been used and description.

- "main.py": It contains all API which connects with databases for different operations.
- "database.py": It has all the function required to interact with database in MongoDB.
- "model.py": Defines how our collections will look like.

1. Following API, I have implemented:
   1.1. "/trades": which give list of trades. Accepts nine optional parameters.
   1.2. "/trades/{trade_id}": to get a trade by id.
   1.3. "/trades/create/": to create a trade

These all API interacts with MongoDB functions which has been implemented in database.py and give response with desired output.

2. I used Pydantic model and MongoDB as a database for storing data because what the given trade model was like a **JSON Object** which makes a plus point.
   The MongoDB is a NoSQL database means; it does not store data in tables rather as a collection of complex data (as a JSON object) which makes it quite friendly for a developer who is not familiar with databases.
   Below I will explain the different function I used to interact with MongoDB database.

   2.1. For search and match I have used "aggregate" of MongoDB which is a more efficient version of find so that I can apply multiple filtering as well as search operation.
   - For search query I have use "$match" and "$or" to find whether the search string is present in any one of the members. "$match" in aggregation returns collections which fulfil those queries

```
if(search!=None):
    Matchquery.append({"$match":{"$or":[
     {"counterparty":search},
     {"instrument_id":search},
     {"instrument_name":search},
     {"trader":search}]}})
```

   - For filter I run "$match" based on the given parameters.

```
if(assetClass!=None):
    Matchquery.append({"$match":{"asset_class": assetClass}})
```

   - "$sort" is used to sort collection according to a specific member

   2.2. Aggregation in MongoDB is a step-by-step filtering, grouping of collections. It accepts each step whether multiple match or group in the form of array and perform those steps sequentially.
   Example: db.collections.aggregate([{expression1}, {expression2}, {expression3}])

```
Matchquery=[]
if(search!=None):
    Matchquery.append({"$match":{"$or":[
     {"counterparty":search},
     {"instrument_id":search},
     {"instrument_name":search},
     {"trader":search}]}})

if(assetClass!=None):
    Matchquery.append({"$match":{"asset_class": assetClass}})

if(start!=None):
    Matchquery.append({"$match":{"trade_date_time":{"$gte": start}}})
if(end!=None):
    Matchquery.append({"$match":{"trade_date_time":{"$lte":end}}})
if(minPrice!=None):
     Matchquery.append({"$match":{"trade_details.price":{"$gte": minPrice}}})
if(maxPrice!=None):
     Matchquery.append({"$match":{"trade_details.price":{"$lte": maxPrice}}})
if(tradeType!=None):
     Matchquery.append({"$match":{"trade_details.buySellIndicator":tradeType}})
if(sort!=None):
     if(sortIn!=None):
        if(sortIn==1 or sortIn==-1):
            Matchquery.append({"$sort":{sort:sortIn}})
        else:
            return ("invalid sortIn value")
     else:
        Matchquery.append({"$sort":{sort:1}})
```

I have declared an array "Matchquery" which will have search and filter parameters that are provided by API "/trades" and parameters with None will not be performed.
If you observe in image are there many if else condition. This is because there are multiple parameters in "/trades", now what if some parameters are given and others are "None".
So, we will not put those parameters with "None" in our aggregation array to prevent from wrong retrieval of collections. The “db.collection.aggregate(Matchquery)” return a cursor of desired collections, this means we need to traverse these collections. Thus, I have used an array “tradeL” in which I append those collections.
Here is the example of aggregation command that will look for “db.collection.(Matchquery)”:

db.collections.aggregate[{'$match': {'$or': [{'counterparty': 'Walmart'}, {'instrument_id': 'Walmart'}, {'instrument_name': 'Walmart'}, {'trader': 'Walmart'}]}},
{'$match': {'asset_class': 'BOND'}},
{'$match': {'trade_date_time': {'$gte': datetime.datetime(2023, 4, 22, 19, 35, 15, 167000)}}},
{'$match': {'trade_date_time': {'$lte': datetime.datetime(2023, 4, 22, 19, 35, 15, 167000)}}},
{'$match': {'trade_details.price': {'$gte': 300}}},
{'$match': {'trade_details.price': {'$lte': 500}}}, {'$match': {'trade_details.buySellIndicator': 'SELL'}},
{'$sort': {'trade_details.price': -1}}]

2.3. The function "fetchTrade" uses "find_one" function to retrieve collection which has the desired Trade Id.
2.4. The function "createTrade" to insert one collection to the database

3. To make retrieval and response of data synchronized I have used “async” and “await” in many function. So that first operation is done then only give response output,
4. Here is the predefined collections in database that can used as reference: https://github.com/ShubhamMishra6862/Backend_Assginment_Steeleye/blob/main/tradesList.json


```
  "_id": {
    "$oid": "64440c5d41e8a38ca2d1c1cb"
  },
  "asset_class": "Equity",
  "counterparty": "L&T",
  "instrument_id": "U45C",
  "instrument_name": "ATAGS",
  "trade_date_time": {
    "$date": "2023-04-22T16:30:47.273Z"
  },
  "trade_details": {
    "buySellIndicator": "BUY",
    "price": 200,
    "quantity": 2
  },
  "trade_id": "36L",
  "trader": "DRDO"
}
```