STEELEYE DATA ENGINEERING INTERNSHIP ASSIGNMENT

By Ranjan P

Instructions to run:

1. Clone the repository - *https://github.com/RanjanKiran707/ranjan_assignment*
2. Install the requirements using - *pip install -r requirements.txt*
3. To start the app use - *python -m uvicorn main:app*

FastAPI and Pydantic –

I hadn't used these libraries for a Rest API project but I had used Flask before which is very similar to FastAPI

Database Layer:

For this Layer, I implemented a simple generator function with a for loop to generate 100 records of the Trade data model. These records were stored in a list. Each record's fields were randomized using the random library. For string fields, I used a predefined set of values, which were then randomly assigned using the random library. You can find this implementation in the generate_db() function.

Pydantic Models:

TradeDetails: This contains a Data model with 3 parameters, this is not used alone, its used inside the Trade Data model as a field. It contains buySellIndicator, price and quantity fields, which are some basic trade details

Trade: This model contains fields like asset class, instrument id, trade details and etc. This gives all the details of a single trade. The Trade model is utilized in the response for all three endpoints.

Endpoints:

There are 3 endpoints :

1. /trades
2. /trades/id/{trade_id}
3. /trades/search

/trade: This endpoint returns a paginated response of all trades.

You can specify query parameters to filter the trades based on various fields of the Trade data model, allowing you to retrieve only the trades that match your query.

The query parameters are as follows: asset_class, end, max_price, min_price, start, trade_type, page,limit, sort.

To filter the trades, I first store the entire list of trades in a variable. Then, using list comprehension, I filter the trades based on each specified field in the query parameters. Finally, I perform pagination operations by calculating the start and end indexes for the

specified page using the limit parameter. I slice the filtered trades list accordingly and return it in a dictionary format, which includes the page number and the length of the filtered trades list.

Example Link ->
http://localhost:8000/trades?asset_class=Bond&max_price=100&min_price=20

Example Link 2 ->

http://localhost:8000/trades?asset_class=Equity

/trades/id/{trade_id}: This returns a single trade which matches the trade_id specified by the user. If trade is found I return the trade, else I return a dict containing error field with the error message. The return type is a Union[Trade,dict] which means either the return type will be a Trade object or a dict object.

Example Link -> http://localhost:8000/trades/id/2

/trades/search: This returns a paginated response of all trades where the text specified in the search query parameter matches with the text of any of the following fields: counterparty,instrumentId, instrumentName, trader.

I iterate through all the Trade objects in the database. And if the query text is present in any of these fields I append that object to the searched_trades list. Then I perform my pagination operations, slice the searched_trades list with the calculated start and end index, and return a dict similar to /trades endpoint, with total_count, the page number and the trades in that page.

Example Link -> http://localhost:8000/trades/search?q=googl