PRIGYA GUPTA

Backend Assignment

STEELEYE LIMITED
TC.27648.2024.36882.

This is the file submission of the backend question, with my knowledge and understanding I have done the backend part as well as integrated it with the Front-end assignment

prigya gupta
University registration no:12003728

Brief description of the problem:

This project defines a FastAPI application that exposes two endpoints for retrieving trade data. The first endpoint, */trades,* returns a list of trades and supports filtering, sorting, and pagination. The second endpoint, */trades/{trade_id},* returns a single trade with the specified trade ID.

The application defines two data models using the pydantic library: *TradeDetails* and *Trade*. The TradeDetails model represents the details of a trade, including the buy/sell indicator, price, and quantity. The Trade model represents a trade and includes fields for the trade ID, instrument name and ID, trader name, counterparty name, and trade details.

The application also includes a list of sample trades that are used to demonstrate the functionality of the endpoints.

Approach used:

This code uses an object-oriented approach to define data models for trades and trade details using the `pydantic` library. The `TradeDetails` and `Trade` classes inherit from `pydantic.BaseModel` and define the fields and data types for each model.

The code also uses the FastAPI framework to define a web API with two endpoints for retrieving trade data. The `get_trades` function defines the `/trades` endpoint, which returns a list of trades and supports filtering, sorting, and pagination. The `get_trade` function defines the `/trades/{trade_id}` endpoint, which returns a single trade with the specified trade ID.

The code uses Python's built-in `List` class and list comprehensions to filter, sort, and paginate the list of trades in the `get_trades` function. The function accepts several query parameters that can be used to control the filtering, sorting, and pagination behavior .

Overall, this code demonstrates how to use FastAPI and pydantic to build a web API for retrieving trade data.

Explanation of the code:

```
from fastapi import FastAPI, HTTPException, Query
from pydantic import BaseModel
```

```
from typing import List, Optional
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from datetime import datetime
```

These lines import the necessary modules and classes for the application. `FastAPI`, `HTTPException`, and `Query` are imported from the `fastapi` module. `BaseModel` is imported from the `pydantic` module. `List` and `Optional` are imported from the `typing` module. `FastAPI` is imported again (this is redundant and can be removed). `CORSMiddleware` is imported from the `fastapi.middleware.cors` module. Finally, `datetime` is imported from the `datetime` module.

```
app = FastAPI()
```

This line creates a new instance of the `FastAPI` class and assigns it to the variable `app`. This instance represents the FastAPI application.

```
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

These lines add an instance of the `CORSMiddleware` class to the FastAPI application. This middleware is used to handle Cross-Origin Resource Sharing (CORS) requests. The middleware is configured to allow all origins, credentials, methods, and headers.

```
class TradeDetails(BaseModel):
    buySellIndicator: str
    price: float
    quantity: int
```

These lines define a new class named `TradeDetails` that inherits from the `BaseModel` class provided by the `pydantic` module. This class represents the details of a trade and has three fields: `buySellIndicator`, which is a string that indicates whether the trade is a buy or sell; `price`, which is a float that represents the price of the trade; and `quantity`, which is an integer that represents the quantity of the trade.

```
class Trade(BaseModel):
```

```
    trade_id: str
    instrument_name: str
    instrument_id: str
    trader: str
    counterparty: Optional[str]
    trade_details: TradeDetails
```

These lines define a new class named `Trade` that also inherits from the `BaseModel` class. This class represents a trade and has six fields: `trade_id`, which is a string that represents the unique identifier of the trade; `instrument_name`, which is a string that represents the name of the instrument being traded; `instrument_id`, which is a string that represents the unique identifier of the instrument being traded; `trader`, which is a string that represents the name of the trader who executed the trade; `counterparty`, which is an optional string that represents the name of the counterparty to the trade; and `trade_details`, which is an instance of the `TradeDetails` class that represents the details of the trade

```
trades = [
    Trade(
        trade_id="1",
        instrument_name="CFTC",
        instrument_id="CFTC",
        trader="Bob Smith",
        counterparty="Goldman Sachs",
        trade_details=TradeDetails(
            buySellIndicator="BUY",
            price=100.0,
            quantity=10
        )
    ),
    Trade(
        trade_id="2",
        instrument_name="FINRA",
        instrument_id="FINRA",
        trader="John Doe",
        counterparty=None,
        trade_details=TradeDetails(
            buySellIndicator="SELL",
            price=200.0,
            quantity=5
        )
    ),
    Trade(
        trade_id="3",
        instrument_name="IIROC",
```

```
        instrument_id="IIROC",
        trader="Olivia Brown",
        counterparty=None,
        trade_details=TradeDetails(
            buySellIndicator="SELL",
            price=560.0,
            quantity=4
        )
    ),
    Trade(
        trade_id="4",
        instrument_name="BAR",
        instrument_id="BAR",
        trader="Ethan Davis",
        counterparty=None,
        trade_details=TradeDetails(
            buySellIndicator="SELL",
            price=480.0,
            quantity=2
        )
    ),
    Trade(
        trade_id="5",
        instrument_name="MIFID II",
        instrument_id="MIFID II",
        trader="Ava Garcia",
        counterparty=None,
        trade_details=TradeDetails(
            buySellIndicator="SELL",
            price=700.0,
            quantity=7
        )
    )
]
```

These lines define a list named `trades` that contains several instances of the `Trade` class. Each instance represents a different trade with its own unique values for each field.As suggested in the project I can use any type of database Here, I am locally creating my database.

```
@app.get("/trades", response_model=List[Trade])
async def get_trades(
    assetClass: Optional[str] = None,
    end: Optional[datetime] = None,
    maxPrice: Optional[float] = Query(None, alias="max-price"),
    minPrice: Optional[float] = Query(None, alias="min-price"),
    start: Optional[datetime] = None,
    tradeType: Optional[str] = Query(None, alias="trade-type"),
    skip: int = 0,
```

```
    limit: int = 100,
    sort_by: Optional[str] = None
):
```

These lines define a new route for the FastAPI application using the `@app.get` decorator. The route's path is `/trades` and it accepts GET requests. The route has several parameters that can be used to filter, sort, and paginate

```
 result = trades
    if assetClass:
        result = [trade for trade in result if trade.instrument_name ==
assetClass]
    if end:
        result = [trade for trade in result if trade.tradeDateTime <= end]
    if maxPrice:
        result = [trade for trade in result if trade.trade_details.price <=
maxPrice]
    if minPrice:
        result = [trade for trade in result if trade.trade_details.price >=
minPrice]
    if start:
        result = [trade for trade in result if trade.tradeDateTime >= start]
    if tradeType:
        result = [trade for trade in result if
trade.trade_details.buySellIndicator == tradeType]

    # Sort the results
    if sort_by:
        reverse = False
        if sort_by.startswith("-"):
            reverse = True
            sort_by = sort_by[1:]

        result.sort(key=lambda x: getattr(x, sort_by), reverse=reverse)

    # Paginate the results
    start_index = skip
    end_index = skip + limit
    result = result[start_index:end_index]

    return result

@app.get("/trades/{trade_id}", response_model=Trade)
async def get_trade(trade_id: str):
    for trade in trades:
        if trade.trade_id == trade_id:
            return trade
```

```
    raise HTTPException(status_code=404, detail="Trade not found")
```

This code defines two functions: `get_trades` and `get_trade`. The `get_trades` function filters, sorts, and paginates a list of trades based on the provided query parameters. The `get_trade` function returns a single trade with the specified trade ID.

In the `get_trades` function, the `result` variable is initialized to the full list of trades. The function then applies several filters to the list of trades based on the provided query parameters. For example, if the `assetClass` parameter is provided, the function filters the list of trades to only include trades with an `instrument_name` that matches the provided value.

The function then sorts the filtered list of trades based on the provided `sort_by` parameter. If the `sort_by` parameter starts with a `-`, the list is sorted in reverse order.

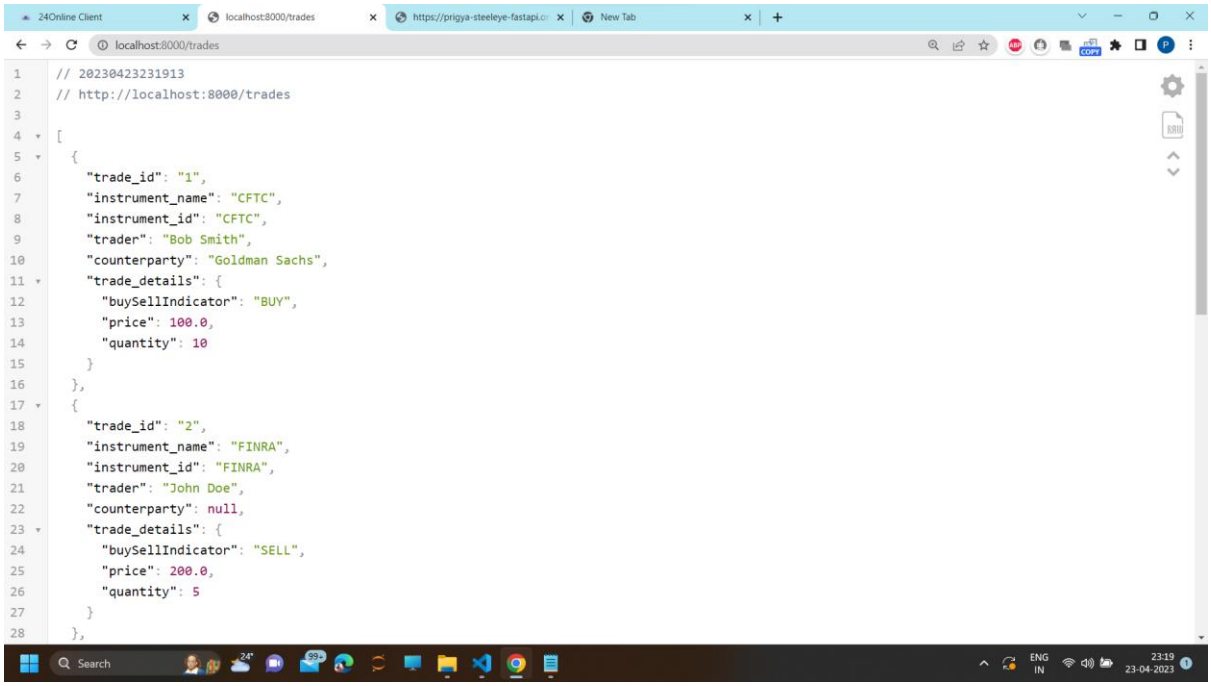
Finally, the function paginates the sorted list of trades based on the provided `skip` and `limit` parameters. The function returns a slice of the sorted list of trades starting at index `skip` and ending at index `skip + limit`.

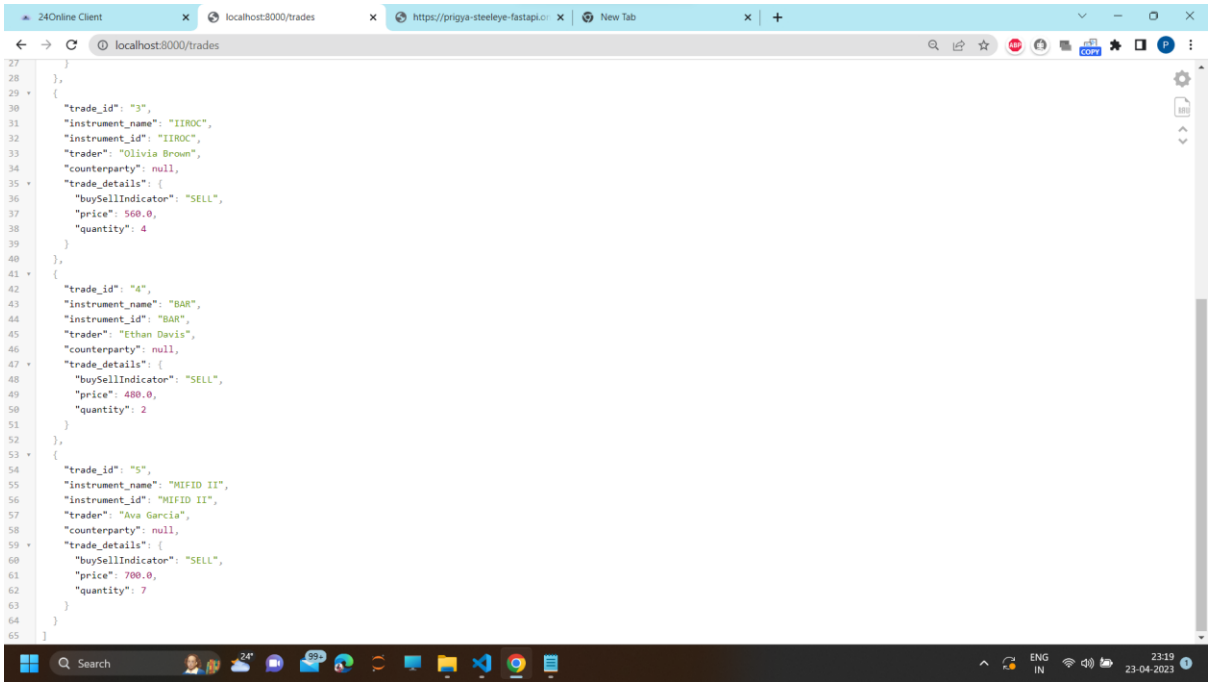
The `get_trade` function iterates over the list of trades and returns the trade with a matching `trade_id`. If no matching trade is found, the function raises an HTTP 404 error.

DEMONSTRATION :

This is the api I have created

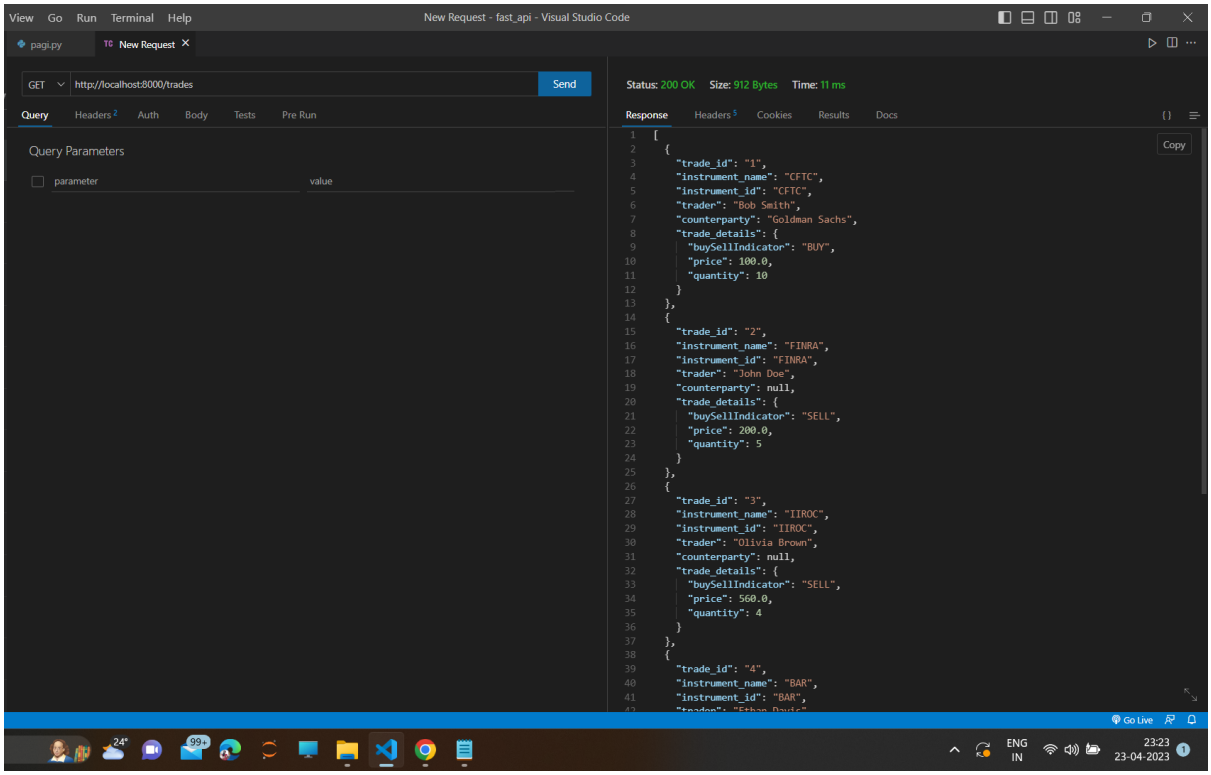
Path of this api is : http://localhost:8000/trades





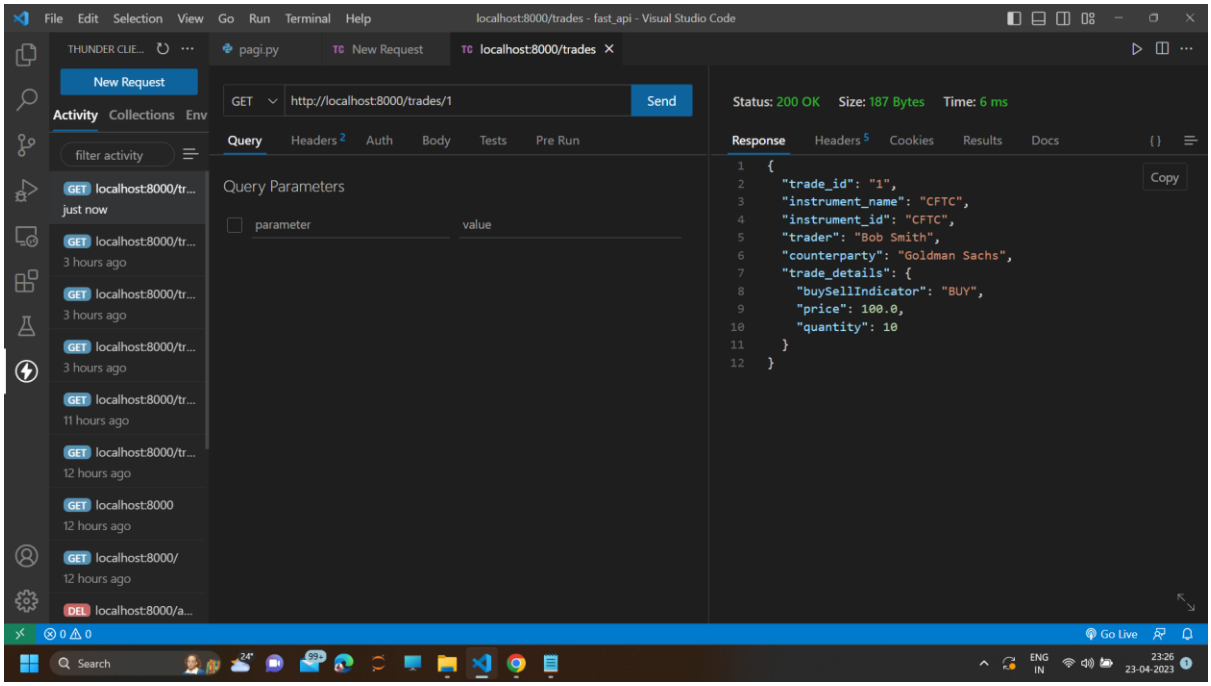
Listing trades

provided an endpoint to fetch a list of trades.



Single trade

Users would like to be able to retrieve a single trade from the API. Provided an endpoint to fetch a trade by Id .Here I am sending a get request by searcing id==1 using the command http://localhost:8000/trades/1 and it is returning me the only instance which has id equal to 1.



Searching trades

Users would now like to be able to search across the trades using the API. Your endpoint for fetching a list of trades will need to support searching for trades through the following fields:

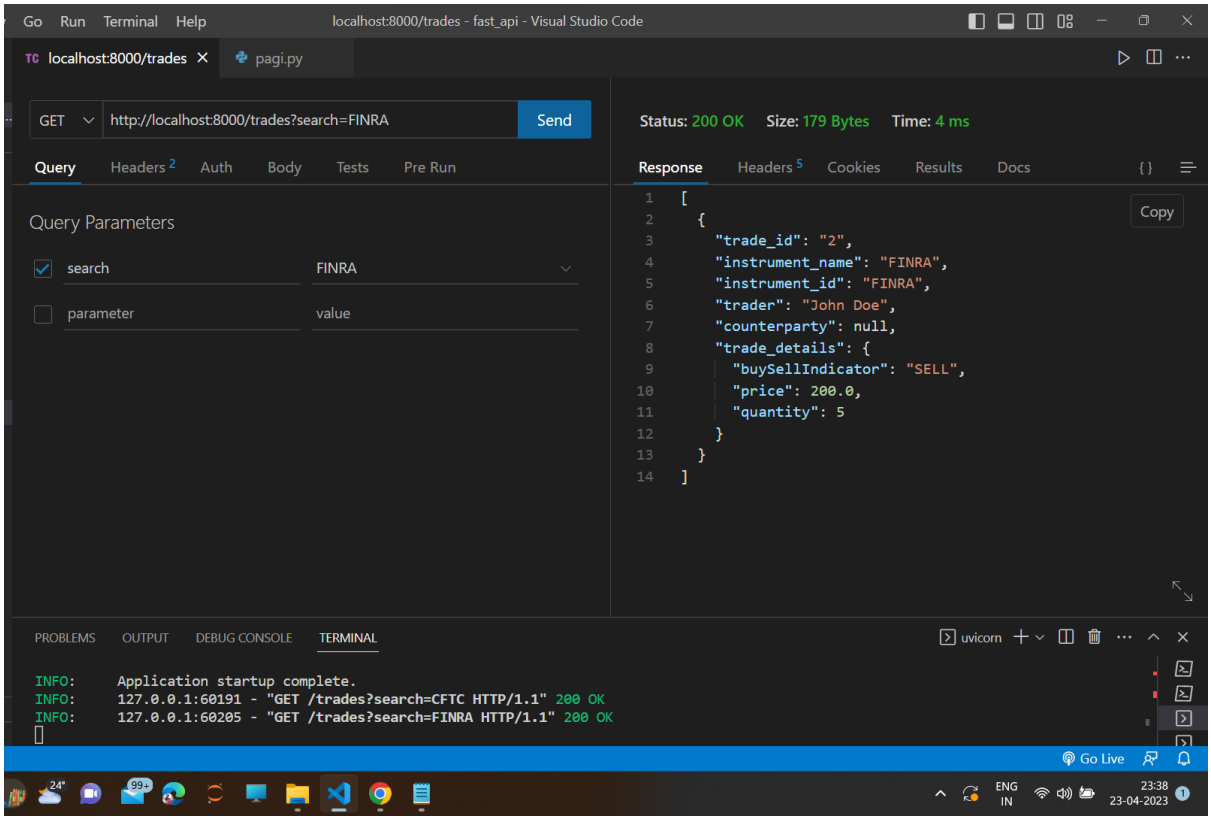
- `counterparty`
- `instrumentId`
- `instrumentName`
- `trader`

If a user was to call your endpoint and provide a `?search=bob%20smith` query parameter, your endpoint will return any trades where the text `bob smith` exists in any of the fields listed above.

```
@app.get("/trades", response_model=List[Trade])
async def get_trades(search: Optional[str] = None):
    if search:
        search = search.lower()
        result = []
        for trade in trades:
            if (search in trade.instrument_name.lower() or
                search in trade.instrument_id.lower() or
                search in trade.trader.lower() or
                (trade.counterparty and search in
trade.counterparty.lower())):
                result.append(trade)
```

```
        return result
    else:
        return trades
```

Here I have search for http://localhost:8000/trades?search=FINRA



Advance filtering:

The users would now like the ability to filter trades. Your endpoint for fetching a list of trades will need to support filtering using the following optional query parameters:

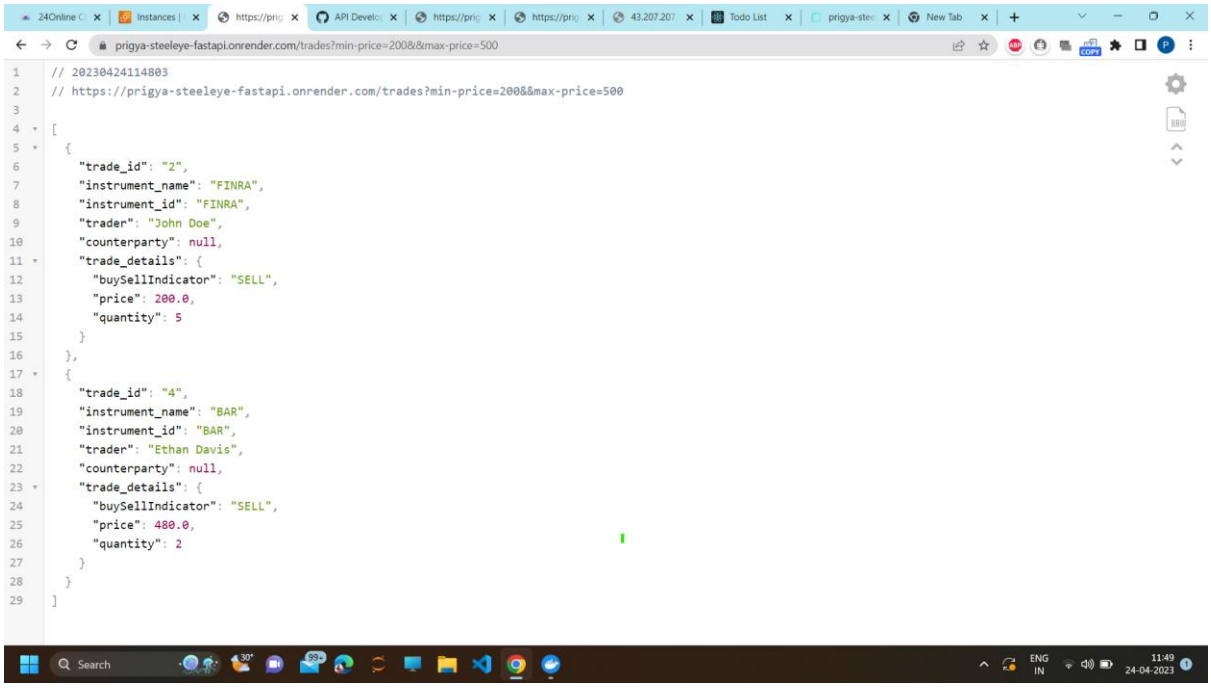
Parameter	Description
`assetClass`	Asset class of the trade.
`end`	The maximum date for the `tradeDateTime` field.
`maxPrice`	The maximum value for the `tradeDetails.price` field.
`minPrice`	The minimum value for the `tradeDetails.price` field.

Parameter	Description
`start`	The minimum date for the `tradeDateTime` field.
`tradeType`	The `tradeDetails.buySellIndicator` is a `BUY` or `SELL`

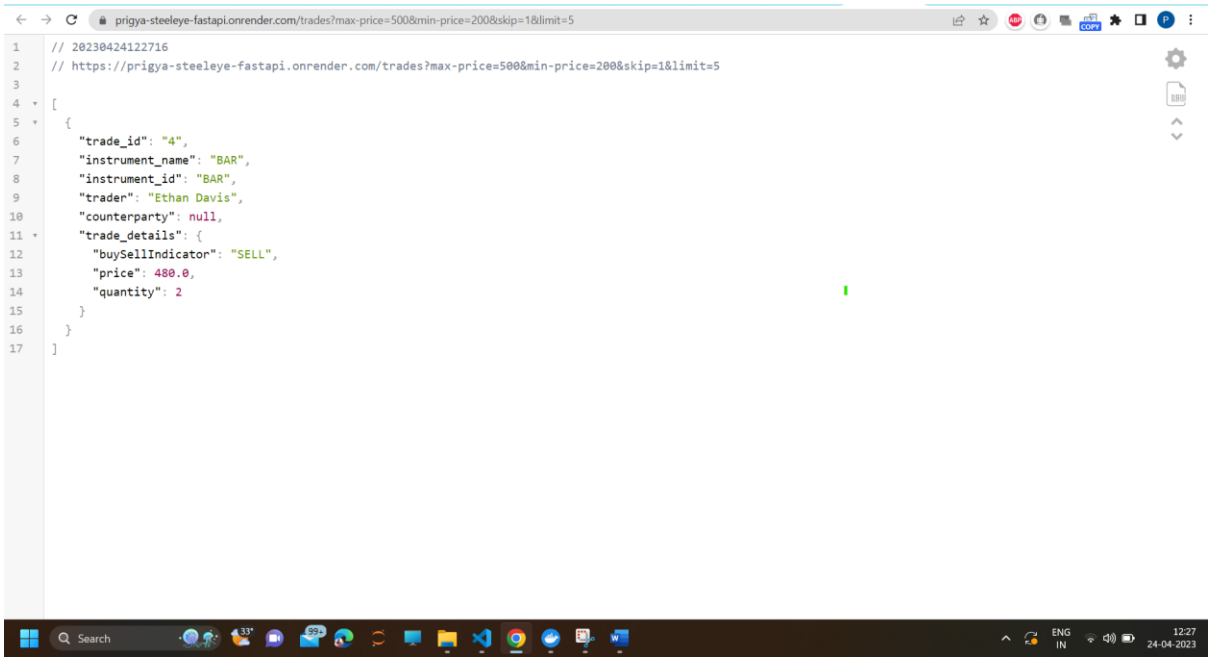
All maximum and minimum fields are inclusive (e.g. `minPrice=2&maxPrice=10` will return `2 <= tradeDetails.price <= 10`).

Here I have search for query =https://prigya-steeleye-fastapi.onrender.com/trades?min-price=200&&max-price=500

So the trades with minimum price of 200 and maximum price has 500 will be returned

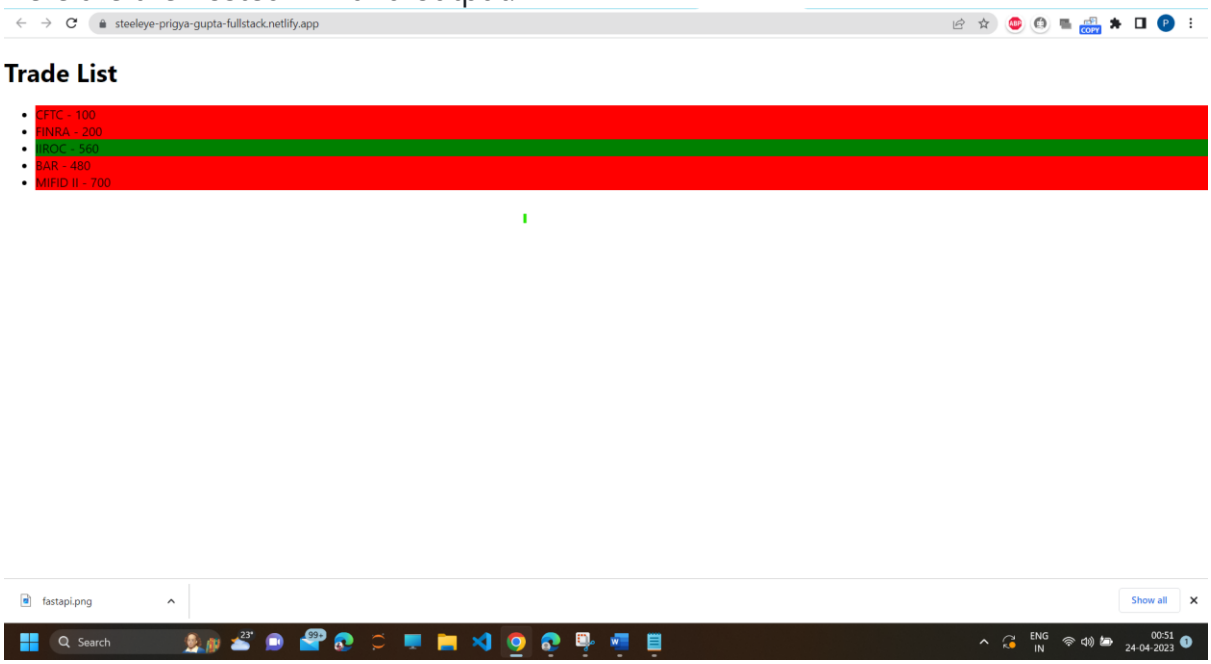


Pagination and sorting on the list of trades.



I have also hosted my api on render and fetched the data from this api and rendered a list using frontend which will show the instrument name and price and if it is selected it will be in green otherwise red, with my knowledge I have joined both the frontend and backend part using axios and CORSMiddleWare.

Here are the hosted link and output:



Netlify: https://steeleye-prigya-gupta-fullstack.netlify.app/

Render: https://prigya-steeleye-fastapi.onrender.com/trades