SteelEye API Developer Assessment

SteelEye API Developer technical test

Link -> https://steeleye-1-c3055529.deta.app/docs

Github -> https://github.com/Saurabh932/SteelEye-FastAPI.git

Submitted by:

Saurabh C. Mandhalkar

Saurabhmandhalkar03@gmail.com

Approach:

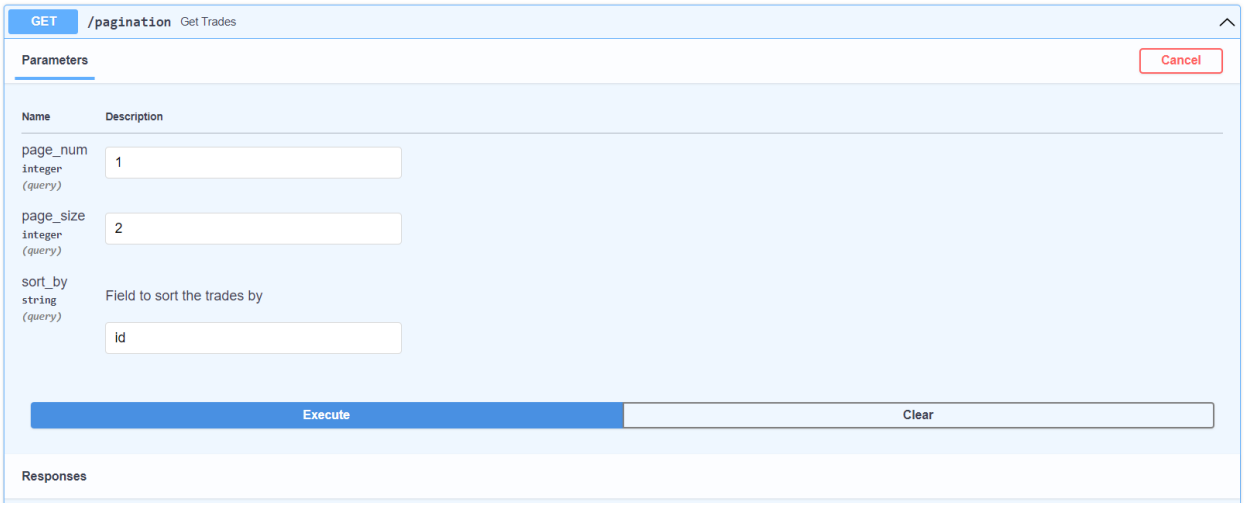
The solution for the assessment is RESTful API implemented using FastAPI . It includes various endpoints for retrieving, filtering, updating, and deleting trade records. It uses a dummy data list trades_db to store the trades instead of adatabase for simplicity. The code utilizes Pydantic models for defining the structure and validation of trade data.

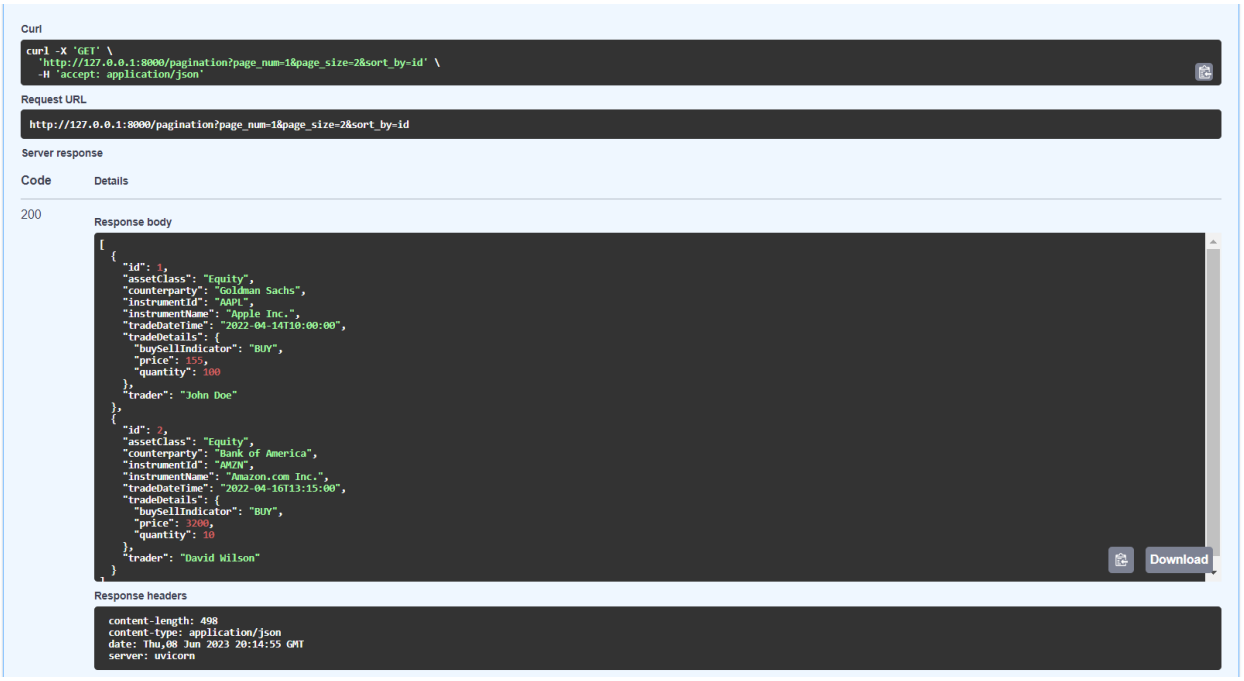
The code defines the following main components:

1. Models: It includes the **Trade** and **TradeDetails** Pydantic models, which represent the structure and validation rules for trade data.
2. Endpoints:
    - The **/pagination** endpoint allows paginated retrieval of trades, with options for sorting by a specific field.
    - The **/trades/{trade_id}** endpoint retrieves a specific trade record based on the provided trade ID.
    - The **/trades** endpoint enables filtering of trades based on various parameters such as keyword search, asset class, price range, trade date range, and buy/sell indicator.
    - The **/trades/{trade_id}** endpoint updates an existing trade record with the provided data.
    - The **/trades** endpoint allows creating a new trade record.
    - The **/trades/{trade_id}** endpoint deletes a trade record with the provided trade ID.
3. Dummy Database: The code includes a list (**trades_db**) containing dummy trade records to simulate a database. This is used for demonstration purposes in the absence of an actual database.

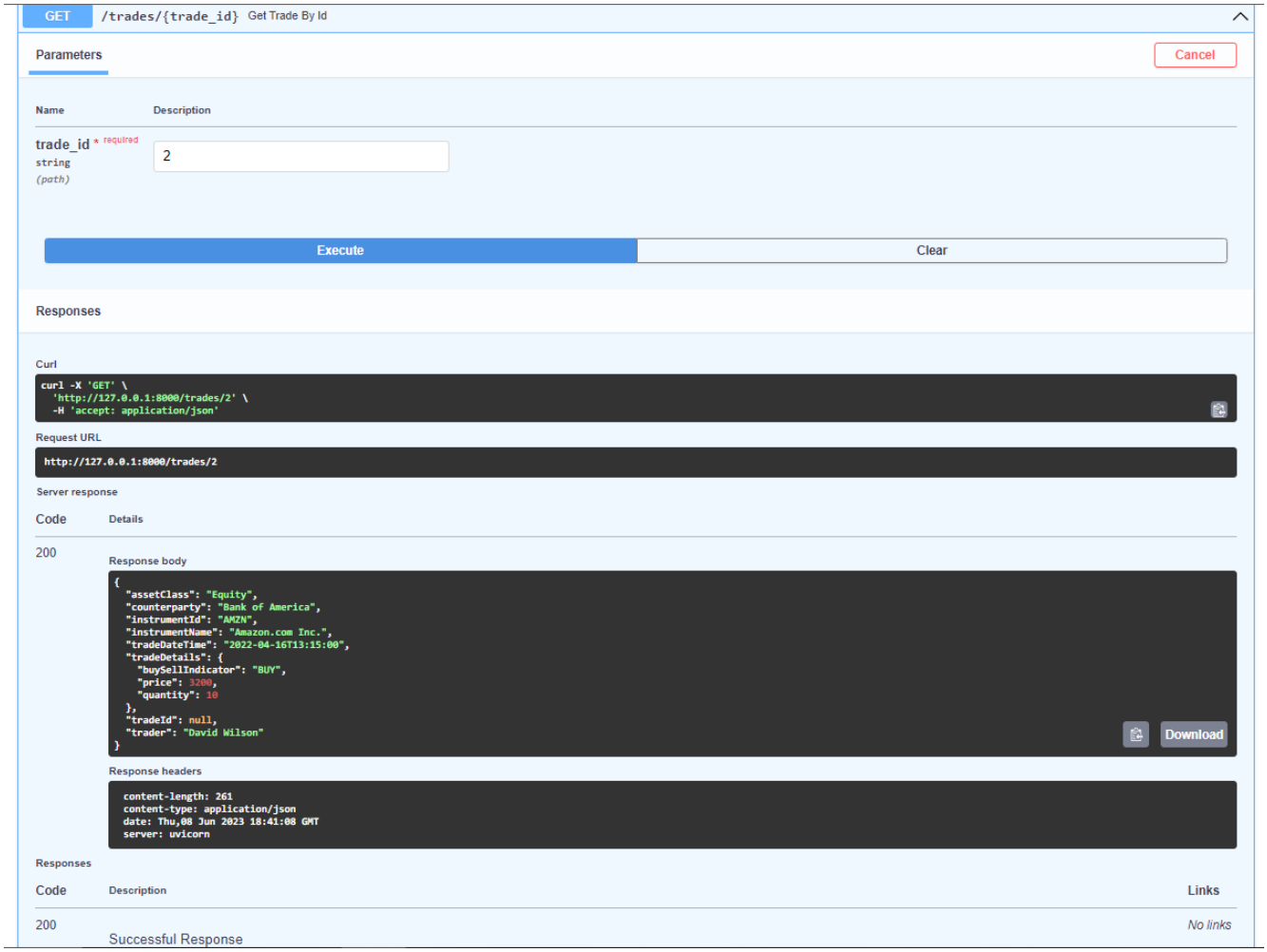
The code follows a RESTful API design and leverages the features of FastAPI to handle HTTP requests, perform data validation, and provide appropriate responses.

1. **root() function**: This function serves as the handler for the root endpoint ("/"). It returns a JSON response with a welcome message.
2. **get_trades() function**: This function handles the "/pagination" endpoint. It accepts query parameters for page number, page size, and an optional sort field. It retrieves a subset of trades based on the pagination parameters, sorts them if a sort field is provided, and returns the paginated and sorted trades. Below is the image for pagination:

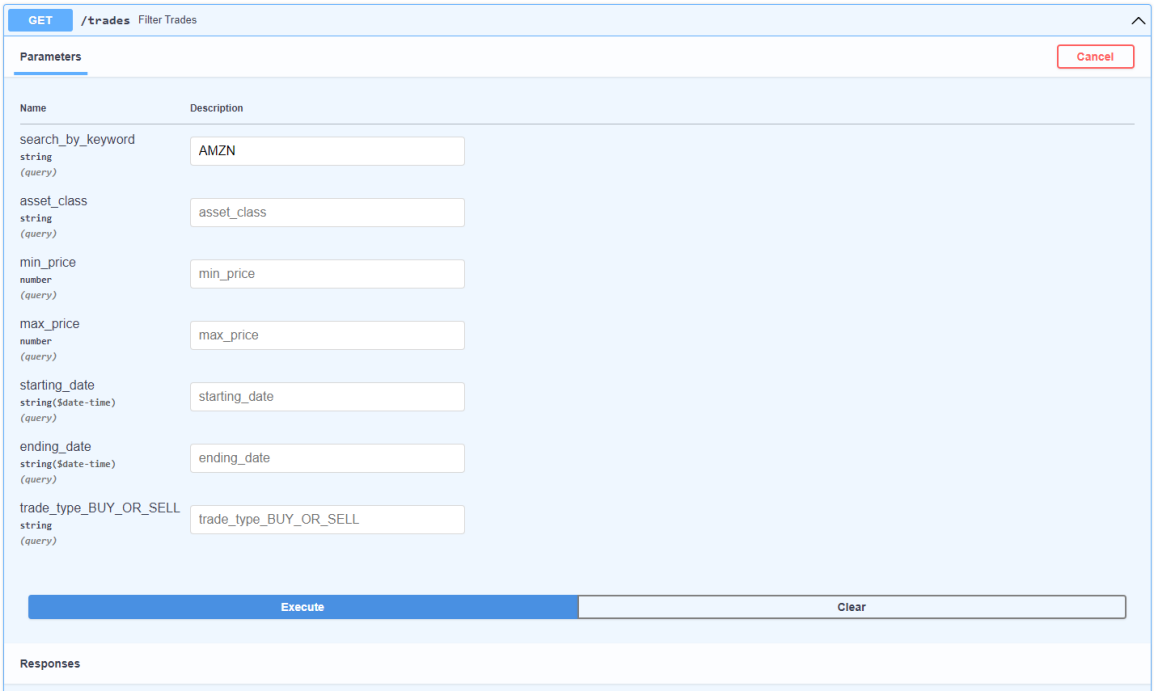


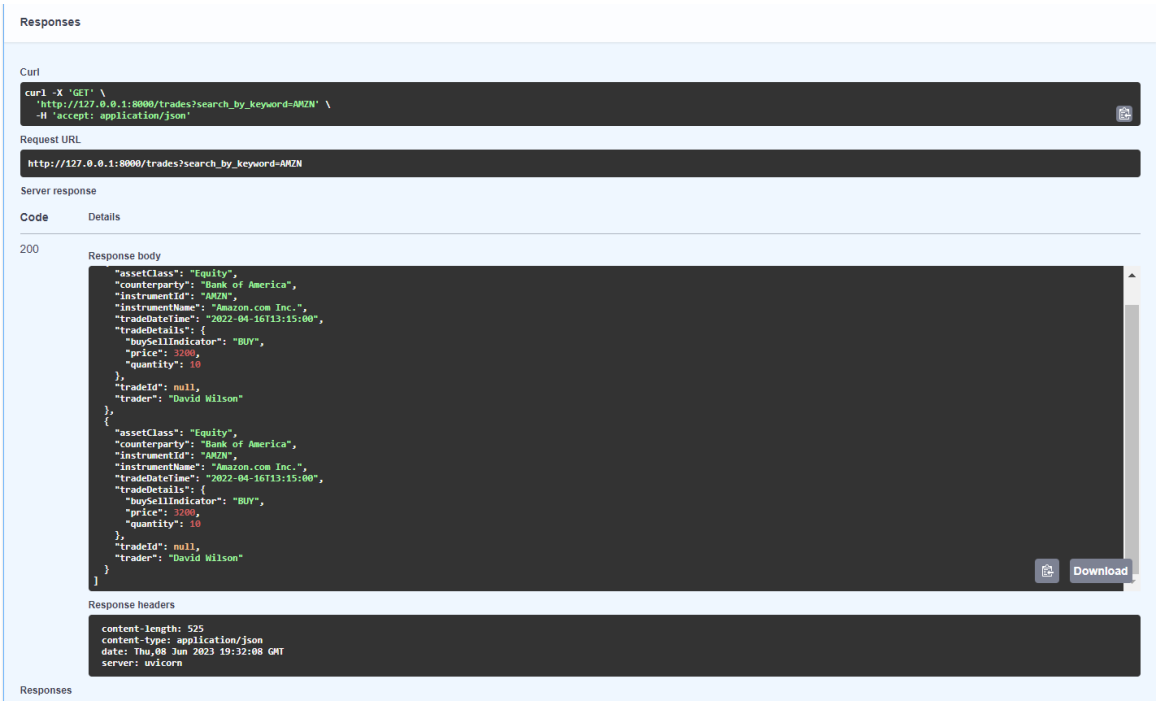


3. **get_trade_by_id() function**: This function is responsible for the "/trades/{trade_id}" endpoint. It takes a trade ID as a path parameter and retrieves the corresponding trade record from the **trades_db** database. If the trade ID is found, it returns the trade record; otherwise, it raises an HTTPException with a 404 status code.

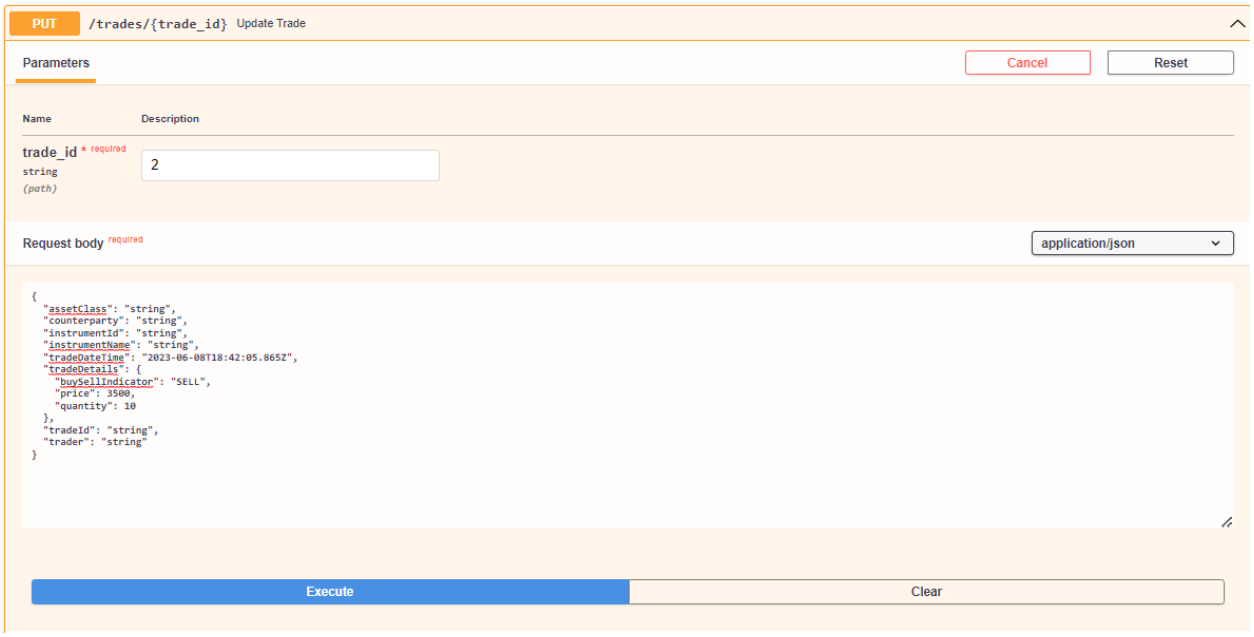


4. **filter_trades() function**: This function handles the "/trades" endpoint for filtering trades based on various query parameters. It allows filtering by keyword search, asset class, price range, trade date range, and buy/sell indicator. It applies the specified filters to the **trades_db** database and returns the filtered trade records.

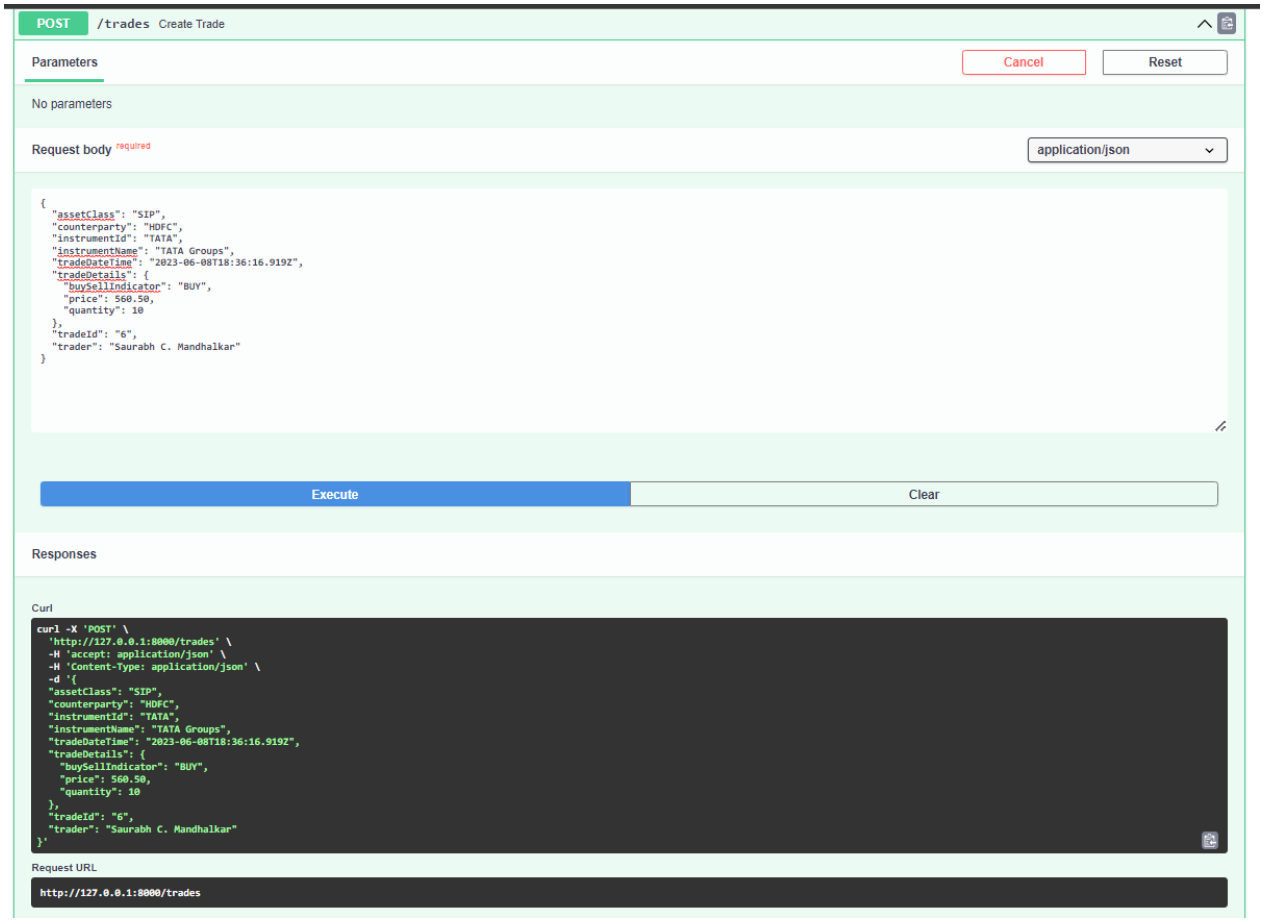




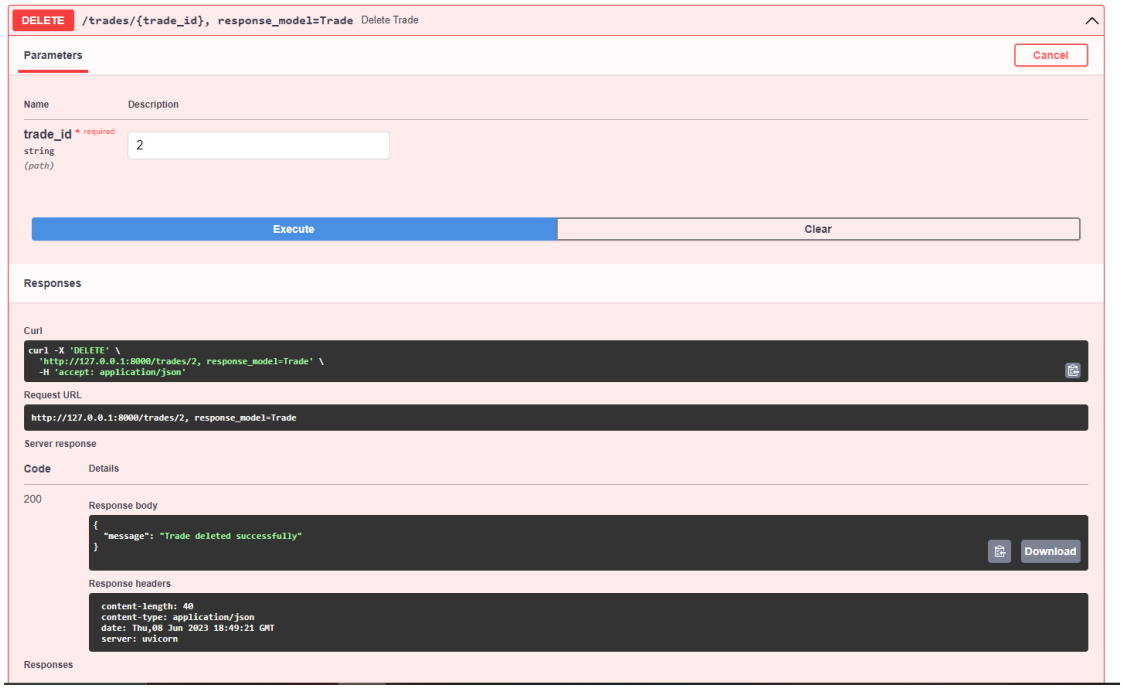
5. **update_trade() function**: This function is responsible for the "/trades/{trade_id}" endpoint with the HTTP PUT method. It takes a trade ID as a path parameter and the updated trade data as the request body. It searches for the trade with the provided ID in the **trades_db** database, replaces it with the updated trade data, and returns the updated trade record. If the trade ID is not found, it raises an HTTPException with a 404 status code.



6. **create_trade() function**: This function handles the "/trades" endpoint with the HTTP POST method. It takes the trade data as the request body, generates a unique trade ID using the **uuid** module, appends the trade record to the **trades_db** database, and returns the created trade record.



7. **delete_trade() function**: This function is responsible for the "/trades/{trade_id}" endpoint with the HTTP DELETE method. It takes a trade ID as a path parameter and searches for the trade with the provided ID in the **trades_db** database. If found, it removes the trade record from the database and returns a JSON response indicating a successful deletion. If the trade ID is not found, it raises an HTTPException with a 404 status code.



In summary, These functions together define the API endpoints and their corresponding functionalities for retrieving, filtering, updating, and deleting trade records. The code leverages the FastAPI framework to handle HTTP requests, perform data validation using Pydantic models, and provide appropriate responses.

Source Code:

```
import uuid
from fastapi import FastAPI, HTTPException, Query
from typing import Optional, List
from pydantic import BaseModel, Field
import datetime as dt

app = FastAPI()


# Pydantic model representing a single Trade
class TradeDetails(BaseModel):
    buySellIndicator: str = Field(description="A value of BUY for buys, SELL
for sells.")
    price: float = Field(description="The price of the Trade.")
    quantity: int = Field(description="The amount of units traded.")


class Trade(BaseModel):
    assetClass: Optional[str] = Field(alias="assetClass", default=None,
                                      description="The asset class of the
instrument traded. E.g. Bond, Equity, FX...etc")
    counterparty: Optional[str] = Field(default=None,
                                        description="The counterparty the
trade was executed with. May not always be available")
    instrumentId: str = Field(alias="instrumentId",
                              description="The ISIN/ID of the instrument
traded. E.g. TSLA, AAPL, AMZN...etc")
    instrumentName: str = Field(alias="instrumentName", description="The name
of the instrument traded.")
    tradeDateTime: dt.datetime = Field(alias="tradeDateTime",
description="The date-time the Trade was executed")
    tradeDetails: TradeDetails = Field(alias="tradeDetails",
                                       description="The details of the trade,
i.e. price, quantity")
    tradeId: Optional[str] = Field(alias="tradeId", default=None,
description="The unique ID of the trade")
    trader: str = Field(description="The name of the Trader")


# Dummy data to be used in place of a database
trades_db = [
    {
        "id": 1,
        "assetClass": "Equity",
        "counterparty": "Goldman Sachs",
        "instrumentId": "AAPL",
        "instrumentName": "Apple Inc.",
        "tradeDateTime": "2022-04-14T10:00:00",
        "tradeDetails": {
            "buySellIndicator": "BUY",
            "price": 155.0,
            "quantity": 100
        },
        "trader": "John Doe"
```

```
    },
    {
        "id": 2,
        "assetClass": "Equity",
        "counterparty": "Bank of America",
        "instrumentId": "AMZN",
        "instrumentName": "Amazon.com Inc.",
        "tradeDateTime": "2022-04-16T13:15:00",
        "tradeDetails": {
            "buySellIndicator": "BUY",
            "price": 3200.0,
            "quantity": 10
        },
        "trader": "David Wilson"
    },

    {
        "id": 3,
        "assetClass": "Equity",
        "counterparty": "Morgan Stanley",
        "instrumentId": "MSFT",
        "instrumentName": "Microsoft Corporation",
        "tradeDateTime": "2022-04-15T09:30:00",
        "tradeDetails": {
            "buySellIndicator": "BUY",
            "price": 250.0,
            "quantity": 200
        },
        "trader": "Alice Smith"
    },
    {
        "id": 4,
        "assetClass": "FX",
        "counterparty": "Citigroup",
        "instrumentId": "EURUSD",
        "instrumentName": "Euro/US Dollar",
        "tradeDateTime": "2022-04-15T14:45:00",
        "tradeDetails": {
            "buySellIndicator": "SELL",
            "price": 1.22,
            "quantity": 5000
        },
        "trader": "Bob Johnson"
    },
    {
        "id": 5,
        "assetClass": "Equity",
        "counterparty": "Bank of America",
        "instrumentId": "AMZN",
        "instrumentName": "Amazon.com Inc.",
        "tradeDateTime": "2022-04-16T13:15:00",
        "tradeDetails": {
            "buySellIndicator": "BUY",
            "price": 3200.0,
            "quantity": 10
        },
        "trader": "David Wilson"
```

```
    }
]


@app.get("/")
async def root():
    return {"message": "Welcome to the Trade API"}


@app.get("/pagination")
async def get_trades(
        page_num: int = Query(1, gt=0), # gt --> it ensures that the provided
value must be greater than 0.
        page_size: int = Query(2, gt=0),
        sort_by: Optional[str] = Query(None, description="Field to sort the
trades by")
):
    # Calculate the start and end indices based on the pagination parameters
    start = (page_num - 1) * page_size
    end = start + page_size

    # Create a copy of the trades database
    sorted_trades = trades_db.copy()

    # Sort the trades if a sort field is provided
    if sort_by:
        try:
            sorted_trades.sort(key=lambda trade: trade[sort_by])
        except KeyError:
            # Raise an HTTPException if the sort field is invalid
            raise HTTPException(status_code=400, detail="Invalid sort field")

    # Return the sorted and paginated trades
    return sorted_trades[start:end]


@app.get("/trades/{trade_id}", response_model=Trade)
async def get_trade_by_id(trade_id: str):
    for trade in trades_db:
        if trade["id"] == int(trade_id):
            return trade
    raise HTTPException(status_code=404, detail="Trade not found")


@app.get("/trades", response_model=List[Trade])
async def filter_trades(
        search_by_keyword: Optional[str] = None,
        asset_class: Optional[str] = None,
        min_price: Optional[float] = None,
        max_price: Optional[float] = None,
        starting_date: Optional[dt.datetime] = None,
        ending_date: Optional[dt.datetime] = None,
        trade_type_BUY_OR_SELL: Optional[str] = None
) -> List[Trade]:
    filtered_trades = trades_db.copy()

    if search_by_keyword:
```

```
        filtered_trades = [trade for trade in filtered_trades if
search_by_keyword.lower() in str(trade).lower()]

    if asset_class:
        filtered_trades = [trade for trade in filtered_trades if
trade.assetClass == asset_class]

    if starting_date:
        filtered_trades = [trade for trade in filtered_trades if
trade.tradeDateTime >= starting_date]
    if ending_date:
        filtered_trades = [trade for trade in filtered_trades if
trade.tradeDateTime <= ending_date]

    if min_price:
        filtered_trades = [trade for trade in filtered_trades if
trade.tradeDetails.price >= min_price]
    if max_price:
        filtered_trades = [trade for trade in filtered_trades if
trade.tradeDetails.price <= max_price]

    if trade_type_BUY_OR_SELL:
        filtered_trades = [trade for trade in filtered_trades if
                           trade.tradeDetails.buySellIndicator ==
trade_type_BUY_OR_SELL]

    return filtered_trades


@app.put("/trades/{trade_id}", response_model=Trade)
async def update_trade(trade_id: str, trade: Trade):
    for t in trades_db:
        if t["id"] == int(trade_id):
            trades_db.remove(t)
            trades_db.append(trade.dict())
            return trade
    raise HTTPException(status_code=404, detail="Trade not found")


@app.post("/trades", response_model=Trade)
async def create_trade(trade: Trade):
    trade_dict = trade.dict()
    trade_dict["tradeId"] = str(uuid.uuid4())
    trades_db.append(trade_dict)
    return trade_dict


@app.delete("/trades/{trade_id}, response_model=Trade")
async def delete_trade(trade_id: str):
    for t in trades_db:
        if t["id"] == int(trade_id):
            trades_db.remove(t)
            return {"message": "Trade deleted successfully"}
    raise HTTPException(status_code=404, detail="Trade not found")
```