Name: Somya Ranjan Sahu

Registration number: 12008793

Backend Assessment : API Developer Assessment

1. Installed fastapi, pydantic, uvicorn.

```
pip install fastapi, pydantic, uvicorn[standard]
```

Import statements:

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

This block of code imports necessary modules and packages for the FastAPI application, including datetime, typing, pydantic, and FastAPI.

2. Created FASTAPI instance.

```
app = FastAPI()
```

3. Defined pydantic model for the trade details.

```
# Defined a Pydantic model for the trade details
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.")
```

```python
# Defined a Pydantic model for the trade
class Trade(BaseModel):
    asset_class: 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")
    instrument_id: str = Field(
        alias="instrumentId", description="The ISIN/ID of the instrument traded.
E.g. TSLA, AAPL, AMZN...etc")
    instrument_name: str = Field(
        alias="instrumentName", description="The name of the instrument traded.")
    trade_date_time: dt.datetime = Field(
        alias="tradeDateTime", description="The date-time the Trade was
executed")
    trade_details: TradeDetails = Field(
        alias="tradeDetails", description="The details of the trade, i.e. price,
quantity")
    trade_id: str = Field(alias="tradeId", default=None,
                          description="The unique ID of the trade")
    trader: str = Field(description="The name of the Trader")
```

This code defines two Pydantic models: Trade and TradeDetails. These models specify the structure and types of data that will be sent and received by the API endpoints.

4. Defined a mock database class.

```python
class MockDB:
    def __init__(self):
        self.trades = []

    def add_trade(self, trade: Trade):
        self.trades.append(trade)
```

The MockDB class is a simple implementation of a database that stores trades in memory using a Python list.

The __init__ method initializes an empty list to store the trades. The add_trade method takes a Trade object as input and adds it to the list.

```
def get_trade_by_id(self, trade_id: str) -> Trade:
        for trade in self.trades:
            if trade.trade_id == trade_id:
                return trade
        return None
```

The get_trade_by_id method takes a trade_id as input and returns the Trade object with the corresponding ID, if it exists in the database. If the trade is not found, the method returns None.

```
def search_trades(self, search_str: str) -> List[Trade]:
        results = []
        for trade in self.trades:
            if search_str.lower() in str(trade).lower():
                results.append(trade)
        return results
```

The search_trades method takes a search_str as input and returns a list of Trade objects that match the search string. The method searches for the string in all the fields of the trade object (including nested fields like trade_details), and returns a list of trades that match.

```
def filter_trades(self, asset_class: str = None, start: dt.datetime = None, end:
dt.datetime = None,
                      trade_type: str = None, min_price: float = None, max_price:
float = None) -> List[Trade]:
        results = self.trades
        if asset_class:
            results = [
                trade for trade in results if trade.asset_class == asset_class]
        if start:
```

```
            results = [
                trade for trade in results if trade.trade_date_time >= start]
        if end:
            results = [
                trade for trade in results if trade.trade_date_time <= end]
        if trade_type:
            results = [
                trade for trade in results if
trade.trade_details.buySellIndicator == trade_type]
        if min_price:
            results = [
                trade for trade in results if trade.trade_details.price >=
min_price]
        if max_price:
            results = [
                trade for trade in results if trade.trade_details.price <=
max_price]
        return results
```

The filter_trades method takes several optional input parameters and returns a list of Trade objects that match the filter conditions. The method starts with all the trades in the database (self.trades) and applies the filters one by one.

```
def update_trade(self, trade_id: str, trade: Trade):
        for i in range(len(self.trades)):
            if self.trades[i].trade_id == trade_id:
                self.trades[i] = trade
                return True
        return False

    def delete_trade(self, trade_id: str):
        for i in range(len(self.trades)):
            if self.trades[i].trade_id == trade_id:
                del self.trades[i]
                return True
        return False
```

These two methods, update_trade and delete_trade, provide functionality to modify and delete Trade objects from the MockDB database.

5. Created instance of the mock database.

```
mock_db = MockDB()
```

6. Created “Create_trade” endpoint.

```
@app.post("/trades")
def create_trade(trade: Trade):
    mock_db.add_trade(trade)
    return {"message": "Trade created successfully"}
```

This endpoint is used to create a new trade. It expects a Trade object as input in the request body. The create_trade function in the code adds the new trade to the MockDB database using the add_trade method, and returns a JSON response with a success message.

7. Created “Filter_trades” endpoint.

```
@app.get("/trades")
def filter_trades(asset_class: str = None, start: dt.datetime = None, end:
dt.datetime = None,
                  trade_type: str = None, min_price: float = None, max_price:
float = None,
                  limit: int = 100, offset: int = 0, sort_by: str = None):
    results = mock_db.filter_trades(asset_class=asset_class, start=start,
end=end, trade_type=trade_type,
                                    min_price=min_price, max_price=max_price)
    total_results = len(results)
    if sort_by:
        results = sorted(results, key=lambda x: getattr(x, sort_by))
    results = results[offset:offset+limit]
    return {"total_results": total_results, "trades": results}
```

This endpoint is used to retrieve a list of trades from the database, filtered by various input parameters. The input parameters include asset_class, start, end, trade_type, min_price, max_price, limit, offset, and sort_by, and can be passed as query parameters in the URL. The filter_trades function in the code applies the filters to the trades in the database using the filter_trades method and returns a JSON response with the total number of results and a list of trades that match the filters.

8. Created "Get_trade_by_id" endpoint.

```python
@app.get("/trades/{trade_id}")
def get_trade_by_id(trade_id: str):
    trade = mock_db.get_trade_by_id(trade_id)
    if trade:
        return trade
    else:
        return {"error": "Trade not found"}
```

This endpoint is used to retrieve a single trade from the database by its ID. The trade_id is passed as a path parameter in the URL. The get_trade_by_id function in the code retrieves the trade from the MockDB database using the get_trade_by_id method and returns a JSON response with the trade object. If the trade is not found, the function returns an error message.

9. Created "Search_trades" endpoint.

```python
@app.get("/trades/search")
async def search_trades(string: Optional[str] = None,
                        counter_party: Optional[str] = None,
                        instrument_id: Optional[str] = None,
                        instrument_name: Optional[str] = None,
                        trader: Optional[str] = None):

    if string:
```

```
        results = mock_db.search_trades(string)
        return results
    elif counter_party:
        results = mock_db.search_trades(counter_party)
        return results
    elif instrument_id:
        results = mock_db.search_trades(instrument_id)
        return results
    elif instrument_name:
        results = mock_db.search_trades(instrument_name)
        return results
    elif trader:
        results = mock_db.search_trades(trader)
        return results
    else:
        return {"Trade not found"}
```

This endpoint is used to search for trades in the database based on a search string or other input parameters. The input parameters include string, counter_party, instrument_id, instrument_name, and trader, and can be passed as query parameters in the URL. The search_trades function in the code searches for the input string in all the fields of the trades in the database using the search_trades method and returns a JSON response with a list of trades that match the search string or input parameters.

10.Created "Update_trade" endpoint.

```
@app.put("/trades/{trade_id}")
async def update_trade(trade_id: str, trade: Trade):
    if mock_db.update_trade(trade_id, trade):
        return {"status": "success", "msg": "Trade updated successfully"}
    else:
        return {"status": "failure", "msg": "Trade not found"}
```

This endpoint is used to update an existing trade in the database by its ID. The trade_id is passed as a path parameter in the URL, and the updated Trade object is passed in the request body. The update_trade function in the code updates the trade in the MockDB database using the update_trade method and returns a JSON response with a success or failure message.

11.Created “Delete_trade” endpoint.

```python
@app.delete("/trades/{trade_id}")
async def delete_trade(trade_id: str):
    if mock_db.delete_trade(trade_id):
        return {"status": "success", "msg": "Trade deleted successfully"}
    else:
        return {"status": "failure", "msg": "Trade not found"}
```

This endpoint is used to delete an existing trade from the database by its ID. The trade_id is passed as a path parameter in the URL. The delete_trade function in the code deletes the trade from the MockDB database using the delete_trade method and returns a JSON response with a success or failure message.

Deployed API Link :

https://fast-api-gcci.onrender.com/docs