Solution Description:

Firstly imported the required model. Created a FastAPI app and to host this app using uvicorn for testing api using in-build documentation.

Trade and TradeDetails Models:

The code defines two Pydantic models: Trade and TradeDetails. These models represent the structure and validation rules for the trade data. They are used to parse the JSON data received in the POST requests and ensure that the data is valid.

Loading Initial Trade Data:

The code reads the trade data from a JSON file called "SteelEye\_fastapi.json" using the json.load() function. This initial trade data is stored in the trade variable and is used by the API endpoints to perform operations.

POST /addtrade Endpoint:

This endpoint is used to add a new trade to the existing trade data. It accepts a JSON payload containing trade information and validates it against the Trade model. The code generates a unique tradeId for the new trade by finding the maximum tradeId from existing trades and incrementing it by 1. It then creates a new trade object and appends it to the trade data. Finally, it updates the JSON file with the new trade and returns the added trade.

GET /listtrade/{tradeId} Endpoint:

This endpoint retrieves a single trade based on the provided tradeId. It searches for a trade in the trade data with a matching tradeId and returns it. If no trade is found, an empty dictionary is returned.

GET /trades Endpoint:

This endpoint allows searching for trades based on a provided search string. It searches for trades in the trade data where the search string matches the counterparty, instrumentId, instrumentName, or trader fields. The matching trades are returned as a list.

POST /filtertrades Endpoint:

This endpoint provides advanced filtering of trades based on multiple criteria. It accepts optional query parameters such as assetClass, end, maxPrice, minPrice, start, and tradeType. It filters the trades based on the provided criteria and returns the filtered trades.

GET /listtrade Endpoint:

This endpoint retrieves a paginated list of trades based on the provided filter criteria. It accepts optional query parameters for filtering (assetClass, end, maxPrice, minPrice, start, and tradeType), as well as parameters for pagination (page and limit), sorting (sort and order), and ordering. It utilizes the advanced\_filter function to filter the trades and then applies pagination, sorting, and ordering on the filtered trades before returning the result.

Reasoning behind the Approach:

Pydantic Models:

Using Pydantic models provides several benefits. They allow defining the structure and validation rules for the trade data, ensuring that the received data is valid and well-formed. Pydantic models also provide automatic data parsing and serialization, making it easier to work with the trade data in the application.

JSON Data Storage:

The initial trade data is stored in a JSON file. This approach allows persistence of trade data beyond the lifetime of the application. The code reads the initial data from the file and writes the updated data back to the file after any modifications. This ensures that the trade data is preserved across application restarts.

Flexibility and Extensibility:

The code provides flexibility by allowing filtering, sorting, and pagination of trades. It accepts various query parameters to customize the result based on specific criteria. This flexibility makes the API versatile and usable in different scenarios. Additionally, the code is extensible, allowing for the addition of new features or endpoints to manage trades based on evolving requirements.

Overall, the solution utilizes the FastAPI framework and Pydantic models to create a robust and scalable API for managing trades.