Certainly! Here's a document describing the solution and the reasoning behind the approach:

Trade Data API - Solution Description

• *Introduction*

This document describes the solution for building a Trade Data API using the FastAPI framework. The API provides endpoints to retrieve a list of trades, fetch a single trade by ID, search for trades based on specific fields, and filter trades based on search criteria. The solution incorporates a mock database layer to simulate the interaction with Elasticsearch.

• *Solution Overview*

The Trade Data API is built using the FastAPI framework, a modern and high-performance web framework for building APIs with Python. The API exposes several endpoints to fulfil different functionalities required by the users.

• *Endpoints*

1. \`/trades\` (GET): Retrieves a list of all trades. It supports an optional \`search\` query parameter to filter trades based on specified search criteria.

2. \`/trades/{trade_id}\` (GET): Retrieves a single trade by its ID.

3. \`/docs\` (GET): Provides API documentation and interactive testing interface using Swagger UI.

4. \`/redoc\` (GET): Provides API documentation using ReDoc.

• *Mocked Database Layer*

In order to simulate the interaction with Elasticsearch, a mock database layer is implemented using a Python dictionary. The trade data is stored in this dictionary, where trade IDs serve as the keys. Each trade object in the dictionary represents a trade and contains fields such as ID, symbol, quantity, price, counterparty, instrument ID, instrument name, and trader. For the purpose of this solution, the data in the mock database layer is pre-defined and hardcoded.

• *Search and Filtering*

The \`/trades\` endpoint supports the \`search\` query parameter, allowing users to search for trades based on specific fields. The searchable fields include:
- Counterparty
- Instrument ID
- Instrument Name
- Trader

When the `search` parameter is provided, the endpoint performs a case-insensitive search and filters the trades that contain the search query in any of the specified fields. The resulting trades are returned as a list.

- ***Reasoning Behind the Approach***

The chosen approach aims to provide a lightweight and scalable solution for building the Trade Data API. Here are the reasons behind the decisions made:

1. ****FastAPI Framework**:** FastAPI is chosen as the framework for building the API due to its simplicity, high performance, and extensive support for Python type hints. It allows for efficient request handling, automatic request/response validation, and interactive API documentation generation.

2. ****Mocked Database Layer****: Since setting up an actual instance of Elasticsearch is outside the scope of the submission, a mock database layer is implemented using a Python dictionary. This approach enables us to simulate the interaction with Elasticsearch while providing a simple and efficient means to store and retrieve trade data.

3. ****Search and Filtering**:** The `/trades` endpoint is designed to support searching and filtering of trades based on specific fields. By utilising the `search` query parameter, users can search for trades that contain a specific text in any of the searchable fields. The case-insensitive search allows for flexible and intuitive querying of the trade data.

- ***Conclusion***

The Trade Data API solution presented here provides the necessary endpoints to retrieve a list of trades, fetch a single trade by ID, search for trades based on specific fields, and filter trades based on search criteria.
The FastAPI framework, along with the mocked database layer, offers a performance and scalable solution for building the API. Although the database layer is mocked, the API design can be easily extended to integrate with Elasticsearch or any other storage technology in a real-world scenario.

I hope this document provides a clear overview of the solution and the reasoning behind the approach.
