Project Description for Macedonian Stock Exchange Data Retrieval

This project automates the retrieval, processing, and storage of historical stock data from the Macedonian Stock Exchange (MSE) using the **Pipe and Filter architecture**. The purpose is to collect, transform, and maintain a clean dataset for all MSE-listed issuers, covering at least the last ten years of daily trading data. This project uses asynchronous programming in Python and aiohttp to make post requests to the servers of MSE.mk, for data retrieval

Libraries Used

1. **`os`**: Manages directory creation and file system paths, helping to organize the storage of processed data.
2. **`re`**: Enables regular expression operations, particularly in parsing and formatting price values.
3. **`time`**: Used to calculate the runtime of data gathering and processing, enabling performance benchmarking.
4. **`aiohttp` and `asyncio`**: Facilitate asynchronous data retrieval. By using `aiohttp` for HTTP requests in conjunction with `asyncio` for task management, the project optimizes network calls, allowing concurrent data retrieval without the need for threads. This minimizes context-switching time, making the process faster and more resource-efficient.
5. **`requests`**: Retrieves the initial stock symbols list from the MSE site. This library handles synchronous HTTP requests.
6. **`BeautifulSoup`**: Parses HTML responses, extracting stock symbols directly from the MSE webpage. This ensures that all valid issuers are captured without manual intervention.
7. **numpy (np)**: Though not heavily used in this project, it is included to handle data manipulation tasks that may require efficient numerical operations.
8. **pandas (pd)**: Used extensively for data handling. It loads, transforms, and saves stock data tables in CSV format, facilitating easy manipulation and ensuring data consistency.
9. **`StringIO`**: Converts HTML responses to a format compatible with `pandas.read_html`, allowing seamless parsing of tabular data.

10. **`datetime` and `timedelta`**: Handle date manipulations, which are crucial for tracking and updating the last available date for each stock.

Asynchronous Programming vs. Threading

In our program we opted for Asynchronous functions over threads to optimize performance during data retrieval. Threading, although commonly used for concurrency, introduces overhead due to context switching, which occurs when the operating system shifts focus between threads. Each switch involves saving and restoring thread states, slowing down operations in I/O-intensive tasks like network requests. Asynchronous programming avoids these context switches by using event loops to manage tasks, allowing data retrieval functions to operate concurrently without the additional processing cost of managing thread states. This choice greatly improves speed and scalability, especially in handling multiple network requests simultaneously.

Pipe and Filter Structure

The **Pipe and Filter architecture** structures data processing into a sequence of independent filters, each responsible for a specific task. This modular design makes the pipeline flexible, maintainable, and easy to extend. Here's how each filter is implemented:

1. **Filter 1: Data Retrieval of Issuers**
    a. The ***get_stock_names*** function scrapes the MSE website for issuer symbols, filtering out irrelevant entries (like bonds or symbols containing numbers). This automatic gathering of stock symbols serves as the initial step in the pipeline, providing a complete and up-to-date list of issuers to be processed.
2. **Filter 2: Check Last Available Date**
    a. Using ***check_last_date_available,*** the project reads each issuer's CSV file (if it exists) to determine the last recorded date of available data. If no prior data exists, the function initializes a start date covering the last ten years. This filter ensures the pipeline only requests data not yet recorded, reducing redundant processing and avoiding unnecessary HTTP requests.
3. **Filter 3: Data Retrieval and Storage**
    a. The ***get_all_data_for_one_stock*** function retrieves historical data year by year for each stock up to the last recorded date. Data is fetched using ***get_one_year_stock_data***, which loads each year's data and appends it to an internal data array. This collected data is saved locally in a structured

directory with separate CSV files for each stock, merging it with existing data if the file already exists.

4. **Filter 4: Data Transformation and Formatting**
    a. Finally, ***process_data*** applies necessary transformations, such as date and price formatting, using helper functions like ***format_date_to_european*** and ***format_price***. This filter ensures that each data entry is clean and consistent in format, preparing it for storage or analysis.

Workflow and Function Flow

The program follows a clear workflow with each function contributing to one or more stages in the pipeline:

1. **Entry Point (`main` function)**
    a. The `main` function serves as the entry point, initiating stock symbol retrieval and triggering asynchronous data retrieval tasks. It also measures the program's runtime for performance assessment.
2. **Stock Symbol Retrieval (`get_stock_names`)**
    a. `get_stock_names` fetches issuer codes from the MSE website and calls `check_last_date_available` to determine the last available date for each stock, preparing them for data retrieval.
3. **Data Retrieval (`get_all_stocks_data`)**
    a. This function initiates data collection for each stock, creating a task for each one using `asyncio.gather`. It retrieves data up to the most recent date for each issuer, ensuring the dataset is current.
4. **Data Transformation (`process_data`)**
    a. After data retrieval, `process_data` applies formatting transformations to ensure dates and prices follow a consistent structure. Finally, the data is stored in CSV files organized by stock code.

Conclusion

This project efficiently automates MSE stock data processing while retaining consistency and modularity using the Pipe and Filter architectures. The use of asynchronous functions over threads improves efficiency in I/O-bound operations, and the architecture ensures that each processing stage remains separated and reusable. Additionally, using the aiohttp package, which is designed for asynchronous requests, makes the data retrieval process faster, and is integrated seamlessly to our project solution.