Stock-Tracing-Technical-Deepdive

Author : Shriniwas Kulkarni

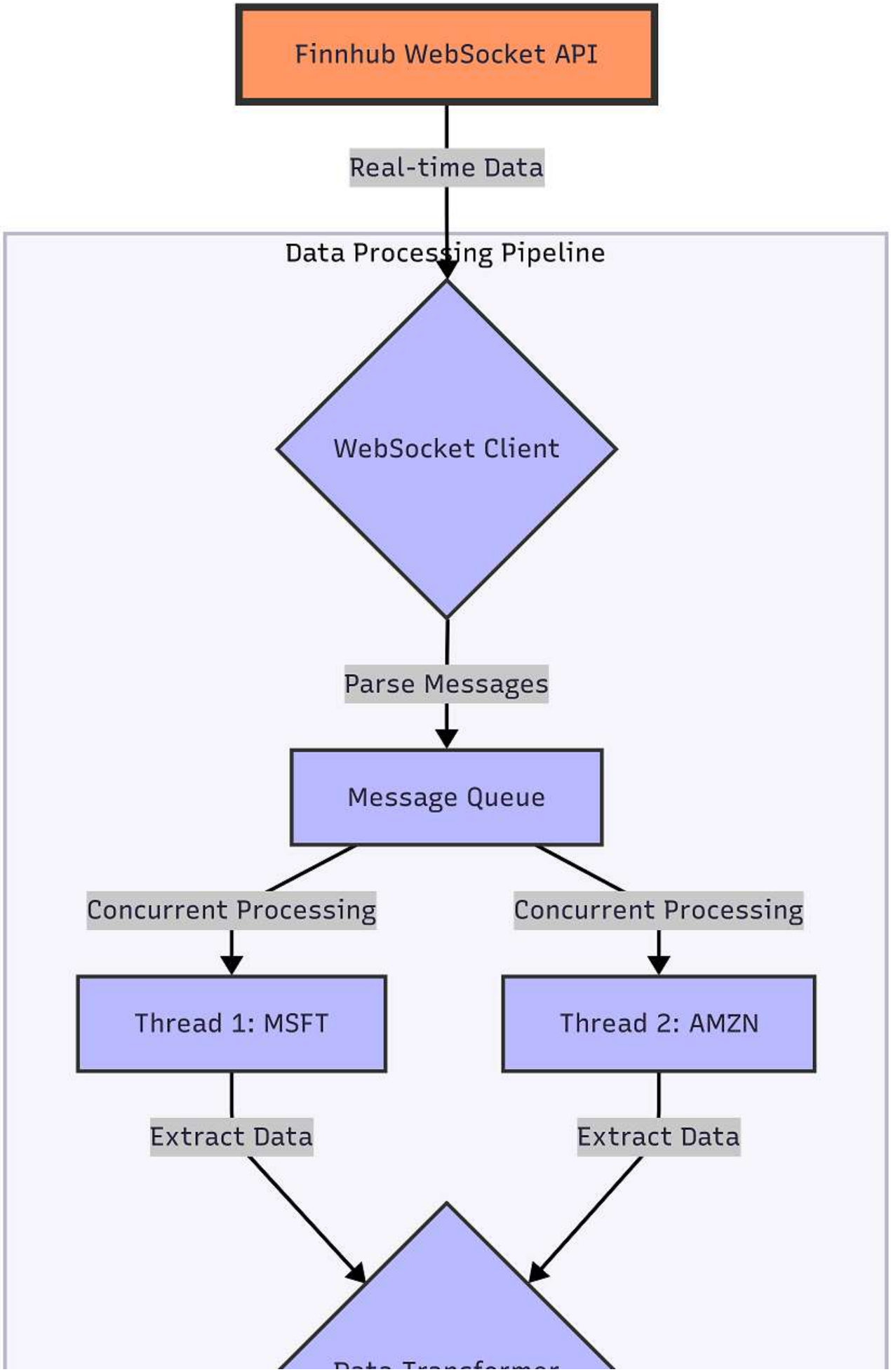
- **PCCOE 2026 BTech CSE(AIML)**
- **Email:** kshriniwas180205@gmail.com
- **Phone:** +91 [8999883480]
- **GitHub:** github.com/Shriniwas18K

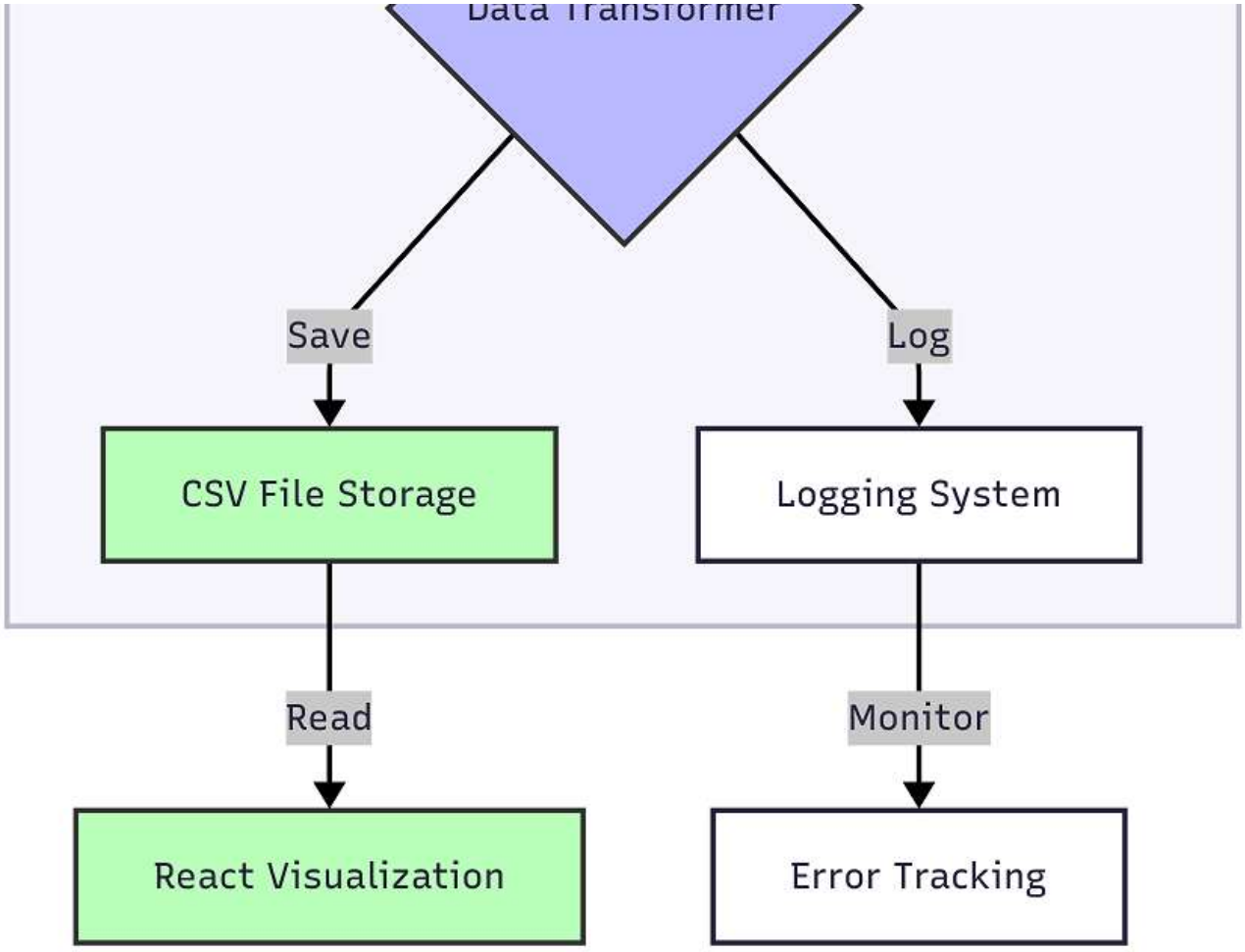
Project Overview

Technical Architecture

This project demonstrates a sophisticated real-time stock data streaming application utilizing websocket technology to capture live financial market data. The application leverages multiple advanced Python programming concepts and design patterns to create a robust, scalable data collection system.

Key Technical Highlights


Finnhub WebSocket API
Real-time Data
Data Processing Pipeline
WebSocket Client
Parse Messages
Message Queue
Concurrent Processing
Concurrent Processing
Thread 1: MSFT
Thread 2: AMZN
Extract Data
Extract Data


Data Transformer
Save
Log
CSV File Storage
Logging System
Read
Monitor
React Visualization
Error Tracking

1. Websocket Communication

- **Technology**: Websocket-client library for real-time data streaming
- **Endpoint**: Finnhub.io WebSocket API
- **Features**:
  - Dynamic ticker subscription
  - Continuous real-time data capture
  - Automatic reconnection handling

2. Concurrent Processing Architecture

- **Multithreading**: Implements background processing for message handling
- **Queue Management**:
  - Utilizes Python's `queue` module for thread-safe message buffering
  - Separate message queues for each stock ticker
  - Prevents data loss during high-frequency data streams

3. Robust Error Handling

- Comprehensive logging mechanism
- Exception handling for:
  - WebSocket connection errors
  - Message parsing
  - Data saving operations
- Automatic reconnection strategy

4. Data Persistence

- CSV file-based data storage
- Append-only logging
- Automatic file handling with built-in error management

Technical Design Patterns

1. **Observer Pattern**
   - WebSocket callbacks (`on_open` , `on_message` , `on_error`)
   - Allows reactive programming model
2. **Producer-Consumer Pattern**

    - Message queues act as buffers
    - Separate threads for data processing and storage

3. **Singleton-like Resource Management**
    - Global file and queue management
    - Centralized logging configuration

Performance Considerations

- Low-overhead threading
- Non-blocking I/O operations
- Minimal memory footprint
- Scalable design supporting multiple tickers

Code Structure Analysis

Main Components

- **Websocket Connection**: Establishes real-time connection to Finnhub
- **Message Processing**:
    - Parse incoming stock data
    - Extract ticker, price, volume
- **Concurrent Processing**: Background threads for each ticker
- **Logging**: Comprehensive event tracking

Security Considerations

- Configurable API key management
- Potential for environment variable integration
- File-based logging for audit trail

Potential Improvements

1. Add data validation mechanisms
2. Implement more sophisticated error recovery
3. Create real-time visualization dashboard
4. Add support for dynamic ticker management

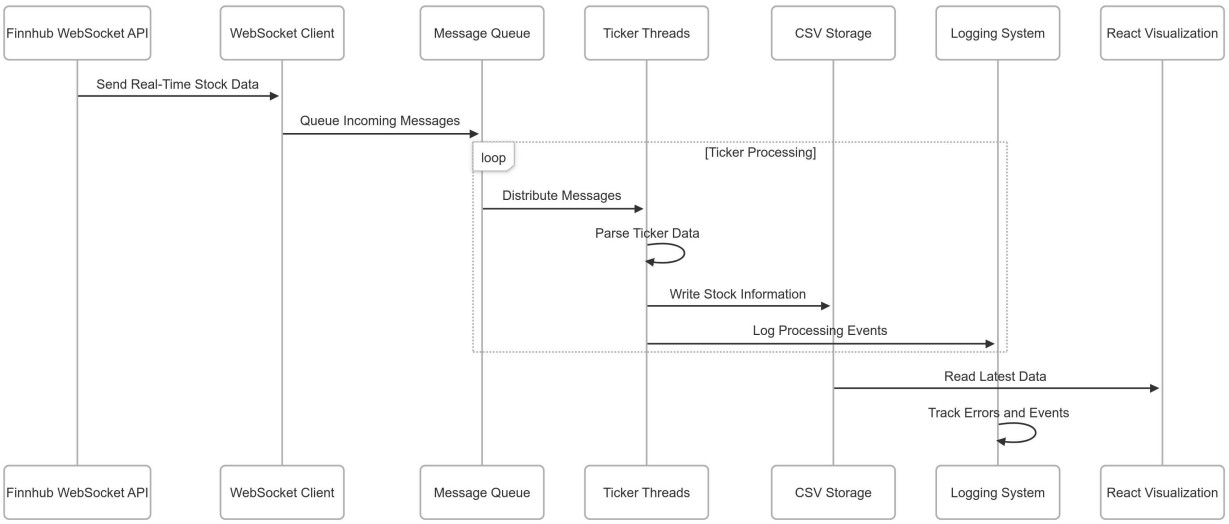
5. Implement advanced data compression techniques

Technology Stack

- **Language**: Python 3.8+
- **Libraries**:
  - `websocket-client`
  - `json`
  - `threading`
  - `logging`
- **API**: Finnhub.io WebSocket
- **Storage**: CSV

Data Flow

1. Connect to Finnhub WebSocket
2. Subscribe to specified tickers
3. Receive real-time market data
4. Parse and extract relevant information
5. Save to CSV
6. Log events and potential errors



Conclusion

This project showcases advanced Python programming techniques in real-time data streaming, demonstrating expertise in:

- Concurrent programming
- Event-driven architectures
- API integration
- Robust error handling