Advanced Task: Real-Time Data Streaming with WebSockets

Objective: To develop a Python script that connects to a broker's or exchange's WebSocket API to receive real-time market data and order updates. The task involves understanding WebSocket connections, handling real-time data streams, processing incoming data, and triggering trading actions based on specific events.

Task Description:

1. **WebSocket API Documentation:**
    - Choose a broker or exchange that provides WebSocket support for real-time data streaming (e.g., Binance, Alpaca, or Interactive Brokers).
    - Study the WebSocket API documentation to understand the required endpoints, message formats, and authentication processes.
2. **Environment Setup:**
    - Install necessary Python packages (websockets, asyncio, json, etc.).
    - Set up a secure environment to store API keys, either using environment variables or a .env file.
3. **Establishing WebSocket Connection:**
    - Implement a Python function to establish a WebSocket connection to the broker's API.
    - Example:
        - connect_to_websocket(): Establishes a connection to the WebSocket endpoint and handles the initial handshake.
    - Handle connection errors and implement automatic reconnection logic in case of disconnections.
4. **Authentication and Subscription:**
    - Implement the authentication process for the WebSocket connection, if required by the broker.
    - Example:
        - authenticate_websocket(): Sends authentication credentials over the WebSocket connection.
    - Subscribe to real-time data streams, such as ticker updates, order book changes, or trade executions.
    - Example:
        - subscribe_to_ticker(symbol): Subscribes to real-time ticker updates for a specific symbol.
        - subscribe_to_order_updates(): Subscribes to receive real-time updates on order status (e.g., filled, canceled).

5. **Handling Incoming Data:**
    - Implement an asynchronous function to continuously receive and process data from the WebSocket.
    - Example:
        - process_incoming_data(): Listens for incoming messages and processes the data accordingly (e.g., updating internal state, triggering alerts).
    - Parse and store incoming data in a structured format (e.g., Pandas DataFrame or custom data structure) for easy analysis.
6. **Real-Time Trading Actions:**
    - Implement logic to trigger trading actions based on real-time data received via WebSocket.
    - Example:
        - Place a market order when the price crosses a certain threshold.
        - Cancel an order if market conditions change (e.g., rapid price movement).
    - Ensure that trading actions are executed with minimal latency to take advantage of real-time data.
7. **Error Handling and Reconnection Strategy:**
    - Implement robust error handling to manage issues like connection drops, rate limits, and invalid messages.
    - Develop a reconnection strategy to automatically reconnect to the WebSocket after disconnection, including exponential backoff and retry logic.
    - Example:
        - handle_websocket_error(error): Handles errors by logging them and attempting to reconnect if necessary.
8. **Advanced Features (Optional but Recommended):**
    - Implement multiplexing to handle multiple WebSocket streams simultaneously (e.g., receiving data for multiple symbols at once).
    - Implement a message queue to buffer incoming data and process it sequentially to avoid missing any updates during high-traffic periods.
    - Develop a real-time dashboard using a Python library like Dash or Plotly to visualize live data updates.
9. **Testing and Simulation:**
    - Create a testing environment to simulate WebSocket messages and test the script's ability to handle real-time data.
    - Example:

        - Mock WebSocket server to simulate real-time market conditions.
        - Unit tests for message handling, data processing, and trading logic.
    - Ensure the system behaves correctly under various conditions, including network delays and sudden market changes.

10. **Deployment and Integration:**
    - Package your WebSocket client as a Python module for easy integration with other trading systems.
    - Provide documentation or a README file explaining how to use the WebSocket client, with examples of subscribing to data streams and triggering trades.
    - Optionally, integrate the WebSocket client with an existing trading bot or strategy to demonstrate real-time trading capabilities.

Deliverables:

- A well-documented Python script or module that includes all functions for connecting to a WebSocket, handling data, and executing trades.
- A set of unit tests and a mock WebSocket server for testing the script.
- A README file or notebook that explains how to use the WebSocket client, including example code snippets.
- Logs of WebSocket interactions showing successful connections, data handling, and trading actions.

This task sheet will help your students learn how to handle real-time data streaming using WebSockets, a critical component in high-frequency trading and real-time market analysis.