# Cryptocurrency Trading Simulation Documentation

## BalanceController Class

The BalanceController class is a **Spring Boot REST controller** that handles all operations related to the user's virtual account balance, cryptocurrency holdings, and transaction history. It provides endpoints for:

* Fetching the current balance.
* Resetting the balance to the initial value.
* Buying and selling cryptocurrencies.
* Fetching transaction history and holdings.

### **Class Fields**

1. **INITIAL\_BALANCE**:
   * A constant representing the starting balance of the user ($10,000).
2. **balance**:
   * Stores the current virtual balance of the user.
3. **holdings**:
   * A Map<String, Double> that stores the user's cryptocurrency holdings. The key is the cryptocurrency symbol (e.g., XBT/USD), and the value is the amount owned.
4. **transactionHistory**:
   * A List<String> that stores a log of all transactions (buy/sell) made by the user.
5. **purchases**:
   * A Map<String, Double> that stores the purchase price of each cryptocurrency. This is used to calculate profit/loss when selling.

### **Methods**

#### **1. getBalance()**

* **Purpose**: Fetches the current account balance.
* **Endpoint**: GET /api/balance
* **Returns**: The current balance as a double.

#### **2. resetBalance()**

* **Purpose**: Resets the account balance to the initial value ($10,000) and clears all holdings and transaction history.
* **Endpoint**: POST /api/balance/reset
* **Behavior**:
  + Resets the balance to INITIAL\_BALANCE.
  + Clears the holdings, transactionHistory, and purchases maps.

#### **3. buyCrypto()**

* **Purpose**: Allows the user to buy a specified amount of a cryptocurrency.
* **Endpoint**: POST /api/balance/buy
* **Parameters**:
  + symbol: The cryptocurrency symbol (e.g., XBT/USD).
  + amount: The amount of cryptocurrency to buy.
  + price: The current price of the cryptocurrency.
* **Behavior**:
  + Calculates the total cost (amount \* price).
  + Checks if the user has sufficient balance.
  + Deducts the cost from the balance.
  + Updates the holdings map with the new amount of the cryptocurrency.
  + Stores the purchase price in the purchases map.
  + Logs the transaction in transactionHistory.
* **Returns**: A transaction message.

#### **4. sellCrypto()**

* **Purpose**: Allows the user to sell a specified amount of a cryptocurrency.
* **Endpoint**: POST /api/balance/sell
* **Parameters**:
  + symbol: The cryptocurrency symbol (e.g., XBT/USD).
  + amount: The amount of cryptocurrency to sell.
  + price: The current price of the cryptocurrency.
* **Behavior**:
  + Checks if the user has sufficient holdings of the cryptocurrency.
  + Calculates the earnings (amount \* price).
  + Adds the earnings to the balance.
  + Updates the holdings map by reducing the amount of the cryptocurrency.
  + Removes the cryptocurrency from holdings and purchases if the amount reaches zero.
  + Calculates the profit/loss based on the purchase price and current price.
  + Logs the transaction in transactionHistory.
* **Returns**: A transaction message with profit/loss.

#### **5. getTransactionHistory()**

* **Purpose**: Fetches the transaction history.
* **Endpoint**: GET /api/balance/history
* **Returns**: A list of transaction messages.

#### **6. getHoldings()**

* **Purpose**: Fetches the user's cryptocurrency holdings.
* **Endpoint**: GET /api/balance/holdings
* **Returns**: A map of cryptocurrency symbols and amounts.

## CryptoWebSocketHandler Class

The CryptoWebSocketHandler class is a **Spring WebSocket handler** that manages WebSocket connections and broadcasts real-time cryptocurrency price updates to all connected clients. It is responsible for:

* Handling new WebSocket connections.
* Managing active WebSocket sessions.
* Broadcasting messages (e.g., price updates) to all connected clients.

### **Class Fields**

### **sessions**:

### A CopyOnWriteArrayList<WebSocketSession> that stores all active WebSocket sessions.

### CopyOnWriteArrayList is used for thread-safe operations, as WebSocket connections can be established or closed concurrently.

### **Methods**

#### **1. afterConnectionEstablished()**

### **Purpose: Handles new WebSocket connections.**

### **Parameters:**

### **session: The WebSocket session representing the new connection.**

### **Behavior:**

### **Adds the new session to the sessions list.**

### **Logs the connection ID for debugging purposes.**

### **2. afterConnectionClosed()**

### **Purpose: Handles WebSocket connection closures.**

### **Parameters:**

### **session: The WebSocket session that was closed.**

### **status: The reason for the connection closure.**

### **Behavior:**

### **Removes the closed session from the sessions list.**

### **Logs the connection ID for debugging purposes.**

### **3. sendUpdateToClients()**

### **Purpose: Broadcasts a message to all connected WebSocket clients.**

### **Parameters:**

### **message: The message to send (e.g., a JSON string containing price updates).**

### **Behavior:**

### **Iterates over all active sessions in the sessions list.**

### **Send the message to each session using session.sendMessage().**

### **Handles IOException if sending the message fails.**

### **Integration with KrakenService**

The CryptoWebSocketHandler is used by the KrakenService class to broadcast real-time price updates to the frontend. Here’s how they work together:

1. **KrakenService**:
   * Receives price updates from the Kraken WebSocket API.
   * Calls sendUpdateToClients() to broadcast the updates.
2. **CryptoWebSocketHandler**:
   * Manages WebSocket connections and broadcasts messages to all connected clients.

## KrakenService Class

The KrakenService class is a **Spring service** responsible for:

* Connecting to the Kraken WebSocket API to fetch real-time cryptocurrency prices.
* Fetching trading pairs and cryptocurrency names from Kraken and CoinGecko APIs.
* Processing WebSocket messages to extract price updates.
* Broadcasting price updates to connected clients via the CryptoWebSocketHandler.

### Class Fields

1. **WEBSOCKET\_URL**:
   * The WebSocket URL for Kraken's API (wss://ws.kraken.com).
2. **REST\_API\_URL**:
   * The REST API URL for fetching trading pairs from Kraken (https://api.kraken.com/0/public/AssetPairs).
3. **COINGECKO\_API\_URL**:
   * The REST API URL for fetching cryptocurrency names from CoinGecko (https://api.coingecko.com/api/v3/coins/list).
4. **cryptoData**:
   * A ConcurrentHashMap<String, Map<String, String>> that stores cryptocurrency data (symbol, name, and price).
5. **webSocketHandler**:
   * An instance of CryptoWebSocketHandler used to broadcast price updates to connected clients.
6. **restTemplate**:
   * A Spring RestTemplate used to make HTTP requests to external APIs.
7. **assetNames**:
   * A ConcurrentHashMap<String, String> that maps cryptocurrency symbols to their full names (fetched from CoinGecko).
8. **FALLBACK\_NAMES**:
   * A static map of fallback names for common cryptocurrencies (e.g., XBT to Bitcoin).
9. **client**:
   * A WebSocketClient instance used to connect to Kraken's WebSocket API.

### **Methods**

#### **1. fetchAssetNames()**

* **Purpose**: Fetches cryptocurrency names from the CoinGecko API and stores them in the assetNames map. Since Kraken Api does not contain a way to map the crypto currencies with their symbols CoinGecko is used to provide names, as well as a fallback map(FALLBACK\_NAMES)
* **Behavior**:
  + Sends a GET request to the CoinGecko API.
  + Parses the response to extract cryptocurrency symbols and names.
  + Stores the mappings in the assetNames map.
* **Error Handling**:
  + Logs errors if the API request fails.

#### **2. connectWebSocket()**

* **Purpose**: Connects to Kraken's WebSocket API and subscribes to ticker updates for trading pairs.
* **Behavior**:
  + Initializes a WebSocketClient and connects to the Kraken WebSocket URL.
  + Calls fetchTradingPairs() to fetch trading pairs dynamically.
  + Subscribes to ticker updates for the fetched trading pairs.
* **Error Handling**:
  + Logs errors if the WebSocket connection fails.
  + Schedules a reconnection attempt if the connection is lost.

#### **3. fetchTradingPairs()**

* **Purpose**: Fetches trading pairs from Kraken's REST API.
* **Behavior**:
  + Sends a GET request to the Kraken REST API.
  + Parses the response to extract trading pairs (e.g., XBT/USD).
  + Filters pairs that trade against USD.
* **Returns**: A list of trading pairs (e.g., ["XBT/USD", "ETH/USD"]).
* **Error Handling**:
  + Logs errors if the API request fails.

#### **4. subscribeToTickerUpdates()**

* **Purpose**: Subscribes to ticker updates for the specified trading pairs.
* **Parameters**:
  + tradingPairs: A list of trading pairs (e.g., ["XBT/USD", "ETH/USD"]).
* **Behavior**:
  + Constructs a JSON subscription message.
  + Send the message to Kraken's WebSocket API.
* **Error Handling**:
  + Logs errors if the subscription fails.

#### **5. processMessage()**

* **Purpose**: Processes WebSocket messages from Kraken's API.
* **Parameters**:
  + message: The raw WebSocket message (JSON format).
* **Behavior**:
  + Parses the message to extract cryptocurrency prices.
  + Updates the cryptoData map with the latest prices.
  + Calls sendUpdateToClients() to broadcast the updates.
* **Error Handling**:
  + Logs errors if the message cannot be parsed.

#### **6. scheduleReconnect()**

* **Purpose**: Schedules a reconnection attempt if the WebSocket connection is lost.
* **Behavior**:
  + Waits for 5 seconds before attempting to reconnect.
  + Calls connectWebSocket() to re-establish the connection.
* **Error Handling**:
  + Logs errors if the reconnection fails.

#### **7. reconnect()**

* **Purpose**: Attempts to reconnect to Kraken's WebSocket API.
* **Behavior**:
  + Waits for 5 seconds before attempting to reconnect.
  + Calls connectWebSocket() to re-establish the connection.
* **Error Handling**:
  + Logs errors if the reconnection fails.

#### **8. getCryptoName()**

* **Purpose**: Retrieves the full name of a cryptocurrency based on its symbol.
* **Parameters**:
  + symbol: The cryptocurrency symbol (e.g., XBT/USD).
* **Behavior**:
  + First check the FALLBACK\_NAMES map for common symbols.
  + If not found, check the assetNames map (fetched from CoinGecko).
  + If still not found, returns the symbol itself.
* **Returns**: The full name of the cryptocurrency (e.g., Bitcoin).

#### **9. sendUpdateToClients()**

* **Purpose**: Sends the latest cryptocurrency data to all connected WebSocket clients.
* **Behavior**:
  + Sorts the top 20 cryptocurrencies by price.
  + Converts the data to a JSON string.
  + Calls webSocketHandler.sendUpdateToClients() to broadcast the data.

## WebSocketConfig Class

The WebSocketConfig class is a **Spring configuration class** that enables and configures WebSocket support in the application. It registers the CryptoWebSocketHandler to handle WebSocket connections at a specific endpoint (/api/crypto/live).

### **Class Fields**

1. **cryptoWebSocketHandler**:
   * An instance of CryptoWebSocketHandler that handles WebSocket connections and messages.

### **Methods**

#### **1. registerWebSocketHandlers()**

* **Purpose**: Registers the CryptoWebSocketHandler to handle WebSocket connections at the /api/crypto/live endpoint.
* **Parameters**:
  + registry: A WebSocketHandlerRegistry used to register WebSocket handlers.
* **Behavior**:
  + Registers cryptoWebSocketHandler to handle WebSocket connections at /api/crypto/live.
  + Allows connections from any origin (setAllowedOrigins("\*")).

## AppConfig Class

The AppConfig class is a **Spring configuration class** that defines beans for the application. In this case, it provides a RestTemplate bean, which is used to make HTTP requests to external APIs (e.g., Kraken and CoinGecko).

## Frontend JavaScript

The JavaScript file (app.js) is responsible for handling the frontend logic of the Cryptocurrency Trading Simulator.

### **Key Functions**

1. **openModal(symbol, price, isBuyAction)**:
   * Opens the Buy/Sell modal and sets the cryptocurrency symbol, price, and transaction type (buy/sell).
2. **closeModal()**:
   * Closes the Buy/Sell modal and resets its state.
3. **fetchBalance()**:
   * Fetches the current account balance from the backend and updates the UI.
4. **fetchHoldings()**:
   * Fetches the user's cryptocurrency holdings from the backend and updates the UI.
5. **fetchTransactionHistory()**:
   * Fetches the transaction history from the backend and updates the UI.
6. **buyCrypto(symbol, amount, price)**:
   * Sends a request to the backend to buy the specified amount of cryptocurrency.
   * Updates the balance, holdings, and transaction history.
7. **sellCrypto(symbol, amount, price)**:
   * Sends a request to the backend to sell the specified amount of cryptocurrency.
   * Updates the balance, holdings, and transaction history.
   * Calculates and displays profit/loss.
8. **connectWebSocket()**:
   * Establishes a WebSocket connection to the backend for real-time price updates.
9. **resetPortfolio()**:
   * Resets the account balance and clears holdings and transaction history.
10. **openTransactionHistoryModal()**:
    * Fetches and displays the transaction history in a modal.
11. **closeTransactionHistoryModal()**:
    * Closes the transaction history modal.