Trading Bot Software

Participants

- Emre Topcu - 20220808011
- Onur Çetin - 20200808050
- Mazlum Arcanlı - 20180808055

1.1 : Project Subject and Purpose

Project Topic: This project aims to develop a crypto *trading bot* using Binance Testnet API and Python. The bot analyses market data based on different trading strategies and makes automatic trading decisions based on this data. The project focuses on learning the basics of algorithmic trading, understanding the working principles of trading bots, and improving the methods of data analysis in the market. In addition, backtesting is carried out to test the accuracy of the developed bot and evaluate how the strategies perform with historical data. Within the scope of the project, at least *three design patterns* will be implemented to increase the maintainability, flexibility, and reusability of the software. The use of these patterns aims to provide solutions to specific problems that arise in the project.

Project Objective:

1. **Creating Trading Bot Software**: Using the Binance Test-net API and Python, develop software that enables a trading bot to analyze data and make trading decisions. This bot will be designed to trade on cryptocurrencies and, after a specified time, will provide the results as output (e.g., Result balance) to the client.
2. **Analyzing Market Data and Developing Trading Strategies**: Analyzing market data using different technical analysis indicators (RSI, ADX, MACD, etc.) in the cryptocurrency market and creating trading strategies based on these analyses. Strategies will be tested based on historical data (back-test) and evaluate which indicators perform better in which market conditions.
3. **Implementation of Back-test Processes:** Back-testing will be conducted using historical data to evaluate the accuracy of the developed trading strategies. During this process, trades will be simulated across different time frames, with a starting balance of $10,000 allocated to the user. After a specified time interval (e.g., 1 hour ago UTC) and predefined candle intervals (e.g., 1 minute), the visualizer and trade logger will be notified, triggering an update of their values.
4. **Live Trading Scenarios (Demo):** Users will also be able to trade live. However, this will only be done on the Binance main-net for a hypothetical $10,000, without the use of real money. In a 1-second (Live Data) time frame, live trades will be executed, showing how trades are bought and sold and with what signals. After the intervals are met, the visualizer and trade logger will be notified and they will update their values.
5. **Design Pattern Implementation :** At the same time, software will be developed using at least 3 design patterns during this project. These design patterns will aim to find solutions to some problems in the software.
6. **Development of Trading Strategies**: In the project, more complex algorithms will be developed starting from simple strategies such as ‘buy low, sell high’. These strategies will help the bot to make decisions based on market movements and enable investors to execute their trades without manual intervention.

1.2 : Technologies We Use

- Python
- Python GUI
- WebSocket and Binance API

1.3 : Trade Strategies We Use

- **MACD Strategy**: Moving Average Convergence Divergence
- **RSI Strategy**: Relative Strength Index
- **ADX Strategy**: Average Directional Index
- **Default Strategy**: Default Strategy For Holding If No Strategies Above Are Met

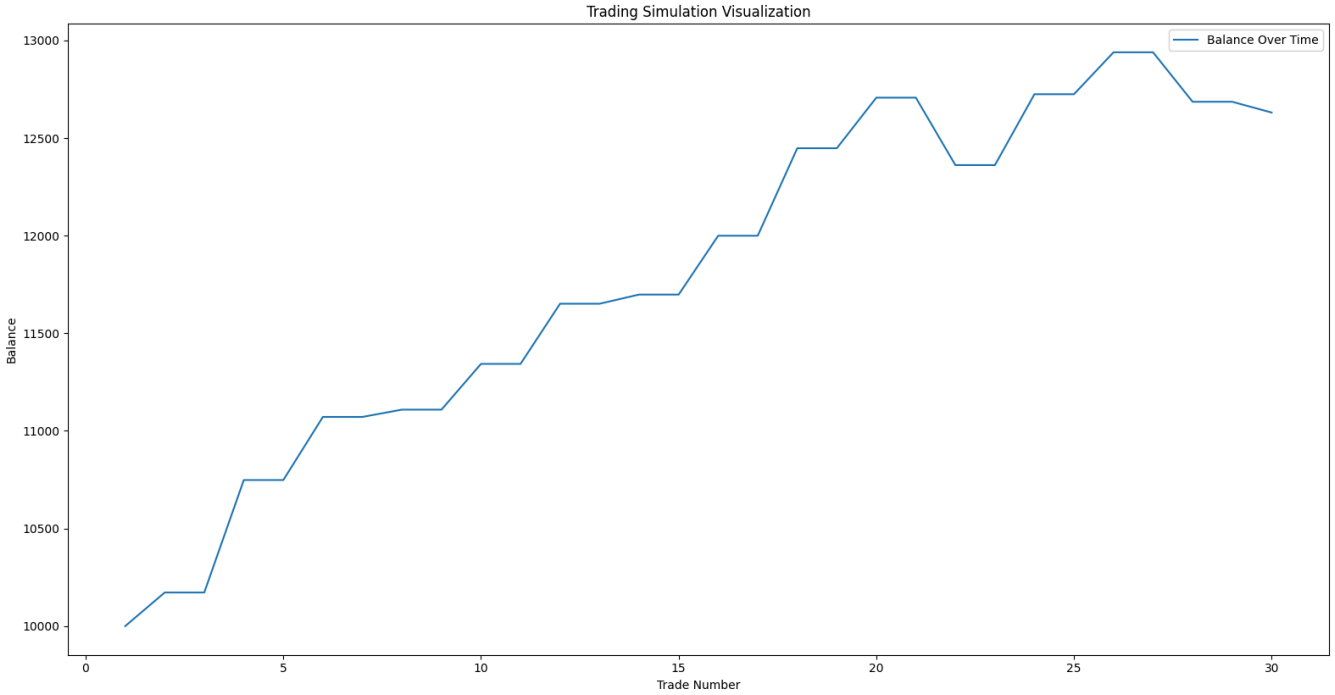
2.1 : Backtest and Results

```
Data we are working with:                          1411
Timestamp  2024-12-03 09:00:00
Open                    3655.13
High                    3662.42
Low                     3651.54
Close                   3656.87
Predicted action: Hold
Holding, Current Strategy:  RSIStrategy
Data we are working with:                          1412
Timestamp  2024-12-03 09:30:00
Open                    3656.87
High                     3670.0
Low                     3626.34
Close                   3634.14
Predicted action: Buy
Couldn't buy, Current Strategy:  DefaultStrategy
Data we are working with:                          1413
Timestamp  2024-12-03 10:00:00
Open                    3634.14
High                     3641.0
Low                     3613.34
Close                   3623.37
Predicted action: Hold
Holding, Current Strategy:  MACDStrategy
```

```
Holding, Current Strategy:  MACDStrategy
Data we are working with:                          1425
Timestamp  2024-12-03 16:00:00
Open                     3605.0
High                     3608.6
Low                     3585.44
Close                   3593.78
Predicted action: Sell
Sold ETHUSDT, Current balance is: 12413.311975265886, Current Strategy: MACDStrategy
```

```
Sold ETHUSDT, Current balance is: 12548.782175353237, Current Strategy: DefaultStrategy
Data we are working with:                          1403
Timestamp  2024-12-03 05:00:00
Open                    3619.01
High                    3638.62
Low                      3607.8
Close                    3633.0
Predicted action: Buy
Bought ETHUSDT, Current balance is: 12548.782175353237, Current Strategy: DefaultStrategy
Data we are working with:                          1404
Timestamp  2024-12-03 05:30:00
Open                     3633.0
High                    3658.16
Low                     3630.98
Close                   3644.38
Predicted action: Hold
Holding, Current Strategy:  RSIStrategy
```

→ As can be understood from the photos above, the **strategy pattern** is used in the **back-test** (It is also used in the Live Trade option) by **switching between strategies**.



→ We obtain the data generated as a result of trades with the **visualization part**.

trade_log.txt

```
Timestamp, Trade Details
----------------------------
[2024-12-03 23:15:24] Bought BTCUSDT, Current balance is: 10000, Current Strategy: RSIStrategy

[2024-12-03 23:15:25] Sold BTCUSDT, Current balance is: 10172.15785425684, Current Strategy: RSIStrategy

[2024-12-03 23:15:26] Bought BTCUSDT, Current balance is: 10172.15785425684, Current Strategy: DefaultStrategy

[2024-12-03 23:15:26] Sold BTCUSDT, Current balance is: 10748.283518268072, Current Strategy: RSIStrategy

[2024-12-03 23:15:35] Bought BTCUSDT, Current balance is: 10748.283518268072, Current Strategy: ADXStrategy

[2024-12-03 23:15:40] Sold BTCUSDT, Current balance is: 11071.524026168787, Current Strategy: RSIStrategy

[2024-12-03 23:15:41] Bought BTCUSDT, Current balance is: 11071.524026168787, Current Strategy: ADXStrategy

[2024-12-03 23:15:42] Sold BTCUSDT, Current balance is: 11108.840354530328, Current Strategy: RSIStrategy

[2024-12-03 23:15:42] Bought BTCUSDT, Current balance is: 11108.840354530328, Current Strategy: ADXStrategy

[2024-12-03 23:15:43] Sold BTCUSDT, Current balance is: 11343.163937406847, Current Strategy: RSIStrategy

[2024-12-03 23:15:44] Bought BTCUSDT, Current balance is: 11343.163937406847, Current Strategy: ADXStrategy

[2024-12-03 23:15:45] Sold BTCUSDT, Current balance is: 11651.402464534543, Current Strategy: RSIStrategy

[2024-12-03 23:15:46] Bought BTCUSDT, Current balance is: 11651.402464534543, Current Strategy: ADXStrategy

[2024-12-03 23:15:46] Sold BTCUSDT, Current balance is: 11698.29398070788, Current Strategy: RSIStrategy

[2024-12-03 23:15:59] Bought BTCUSDT, Current balance is: 11698.29398070788, Current Strategy: DefaultStrategy

[2024-12-03 23:16:00] Sold BTCUSDT, Current balance is: 11999.702831069528, Current Strategy: RSIStrategy
```

→ At the same time, we can keep these trades in our **logger file**.

3.1 : Which Design Patterns Did We Use?

- Strategy Pattern
- Template Method
- Observer Pattern
- Singleton Pattern

3.2 : What Problems Can These Patterns Solve?

1. Strategy Pattern: We use strategies in our Trade Bot. These are **RSI, MACD, ADX, and Default.** One strategy makes more sense at a certain time, while another strategy makes more sense at another time. So we made an implementation where we can change our strategy in **runtime**. Thus, whichever strategy makes more sense, runtime will be able to switch between strategies.
2. Template Method: When we initially designed our strategies, we found out that we were using redundant coding in our workflow which is not a good practice. That's why we implemented the Template Method design pattern to create a flexible and reusable structure for different trading strategies. By defining a common algorithm for executing trading strategies in the TradingStrategy abstract class, we ensured that certain steps like **labeling data, training the model, and making trading decisions** are consistently followed across all strategies. However, we left specific logic, such as labeling data and defining feature columns, to be implemented by each concrete strategy (e.g., RSIStrategy, MACDStrategy, etc.). This approach allows us to easily add new strategies in the future without modifying the core flow, making the system more maintainable and extensible.
3. Observer Pattern: When we initially designed our logger, it was working great but when we wanted to implement a logic where there will be a need for a visualizer that uses the same data as our logger, we realized that **directly coupling them with the Trading Bot would make the system tightly coupled and hard to maintain** since in the future, we may need to add new components that gets data from our trading bot. To solve this, we implemented the Observer pattern. This pattern allows the Trading Bot to broadcast updates to its observers (Logger and Visualizer), **ensuring loose coupling.** Each observer reacts independently to the updates without relying on each other or the Trading Bot. This approach enhances modularity, extensibility, and scalability, making it easy to add new observers and maintain the system.
4. Singleton Pattern: When we initially designed our logger and visualizer, we found out that **we were creating more than one instance across multiple runs which is a bad practice** since we don't want to create instances of our objects each time we run the app to make it more efficient resource-wise. **By leveraging the Singleton design pattern in our system, only one instance of the Visualizer and Trade Logger classes is created.** This ensures that the same instance is consistently used throughout the application, avoiding redundant object creation across multiple runs. As a result, system resources are utilized more efficiently, and potential concurrency issues are mitigated.

3.3 : How Did We Apply These Patterns?

3.3.1: Strategy Pattern Implementation

→ In this code, the Strategy Pattern is used to enable the implementation of different trading strategies in a flexible and extensible way. The TradingStrategy abstract class provides a basic structure for all strategies and allows subclasses to define their custom trading logic (for example, the RSI strategy).

The following code snippets are taken by the TradingBot class where the class has a composite strategy variable (Used for changing strategies at runtime).

PYTHON

```python
def evaluate_strategies(self, row, stop_loss, historical_data):
        """
        Dynamically evaluate and select the best strategy based on market conditions and
historical data.
        """
        # Calculate market volatility
        volatility = historical_data['Close'].std()

        # Assign weights dynamically based on market conditions
        rsi_weight = 0.4 * (1 + volatility)
        macd_weight = 0.3 * (1 - volatility)
        adx_weight = 0.3 * volatility

        # Compute scores for each strategy
        rsi_score = row['rsi'] - historical_data['rsi'].mean()
        macd_score = row['macd'] - historical_data['macd'].mean()
        adx_score = row['adx'] - historical_data['adx'].mean()

        # Calculate weighted scores
        weighted_scores = [rsi_score * rsi_weight, macd_score * macd_weight, adx_score *
adx_weight]

        # Choose the strategy with the highest score
        if max(weighted_scores) > 0:
            if weighted_scores.index(max(weighted_scores)) == 0:
                return RSIStrategy(stop_loss)
            elif weighted_scores.index(max(weighted_scores)) == 1:
                return MACDStrategy(stop_loss)
            else:
                return ADXStrategy(stop_loss)
        else:
            return DefaultStrategy(stop_loss)
```

→ This code dynamically selects the best trading strategy based on market conditions and historical data. It first calculates market volatility, and then assigns weights to the RSI, MACD, and ADX indicators based on volatility. By comparing the current values of these indicators with their historical averages, it generates scores and selects the strategy with the highest score. If no strategy is suitable, a default strategy is used.

PYTHON

```python
def change_strategy(self, row, stop_loss, df):
        """
        Evaluate and change the trading strategy dynamically.
        """
        best_strategy = self.evaluate_strategies(row, stop_loss, df)
        self.set_strategy(best_strategy)
```

→ This code evaluates the current market conditions selects the best trading strategy (with evaluate_strategies) and then sets the selected strategy as the active strategy.

3.3.2: Template Method Implementation

PYTHON

```python
@final
def execute_strategy(self, row, df):
        """
        Template method to execute the strategy workflow.
        1. Label data
        2. Train the model
        3. Extract features
        4. Decide action
        """
        # Step 1: Label data
        df = self.label_data(df, self.label_logic)

        # Step 2: Train the model
        self.train_model(df, self.feature_columns(), self.label_logic)

        # Step 3: Extract features for prediction
        features = self._get_features(row, self.feature_columns())

        # Step 4: Decide action
        action = self.decide_action(features)
        return action
```

→ execute_strategy defines the overall workflow of a strategy:

- Labeling of data,
- Training the model,
- Extraction of features,
- Identification of action.

This method ensures that the strategies share common steps and that each strategy only needs to provide its specific rationale.

3.3.3: Observer Pattern Implementation

PYTHON

```python
from abc import ABC, abstractmethod
class Subject(ABC):
    @abstractmethod
    def register_observer(self, observer):
        pass

    @abstractmethod
    def remove_observer(self, observer):
    pass

    @abstractmethod
    def notify_observers(self):
    pass
```

→ Subject provides an interface in which observers are registered and notified.

- The Subject abstract class defines observer register, remove, and notify operations.
- This structure allows other classes to implement methods necessary for observer management.

```python
from abc import ABC, abstractmethod

class Observer(ABC):
    @abstractmethod
    def update(self, message):
        pass
```

→ Observer provides an interface for receiving messages from the observed object.

- The Observer class requires observers to implement an update method.
- This method is triggered when the observed object (Subject) reports a state change.

PYTHON

```python
import matplotlib.pyplot as plt
from .observer import Observer

class VisualizationObserver(Observer):
    _instance = None  # Class-level attribute to hold the singleton instance

    def __new__(cls, *args, **kwargs):
        # Ensure only one instance of the class is created
        if not cls._instance:
            cls._instance = super().__new__(cls, *args, **kwargs)
        return cls._instance

    def __init__(self):
        # Check if instance is already initialized
        if not hasattr(self, "initialized"):
            self.trade_counter = 0  # Initialize the trade counter
            self.balances = []  # Track balance cumulatively
            self.actions = []  # Stores actions like "Buy", "Sell", "Hold"
            self.initialized = True

    def update(self, message):
        if "Bought" in message or "Sold" in message:
            # During each trade, we only store the trade count and balance
            parts = message.split(", ")
            balance_part = parts[1].split(": ")[1]  # Extract the balance part "Current balance
is: {balance}"

            # Extract the balance as float
            balance = float(balance_part)

            # Increment the trade counter
            self.trade_counter += 1

            # Append the balance to the balances list
            self.balances.append(balance)

        elif "Simulation complete" in message:
            # Plot data when simulation ends
            self._plot_balances()

    def _plot_balances(self):
        """Plot the balance changes over trades at the end of the simulation."""
        # Create a plot showing the balance change over time (across trades)
        plt.plot(range(1, self.trade_counter + 1), self.balances, label="Balance Over Time")

        # Set axis labels and title
        plt.title("Trading Simulation Visualization")
        plt.xlabel("Trade Number")
        plt.ylabel("Balance")
        plt.legend()

        # Display the plot
        plt.show()
```

→ This class processes messages from the trading simulation and visualizes the results.

- The VisualizationObserver class monitors balances based on incoming messages and creates a graph when the simulation is complete.
- This class implements the Observer interface and processes messages with the update method.

PYTHON

```python
import datetime
import os
from .observer import Observer

class LoggingObserver(Observer):
    _instance = None

    def __new__(cls, *args, **kwargs):
        """Ensure only one instance of LoggingObserver is created."""
        if cls._instance is None:
            cls._instance = super().__new__(cls, *args, **kwargs)
        return cls._instance

    def __init__(self):
        """Initialize the log file, but only the first time the instance is created."""
        if not hasattr(self, 'log_file_initialized'):  # Check if already initialized
            self.log_file = "trade_log.txt"
            # Initialize the log file only if it doesn't exist
            if not os.path.exists(self.log_file):
                with open(self.log_file, "w") as f:
                    f.write("Timestamp, Trade Details\n")
                    f.write("----------------------------\n")
            self.log_file_initialized = True  # Mark as initialized

    def update(self, message):
        """Update the log file with a new message."""
        timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        with open(self.log_file, "a") as f:
            if message == "Simulation complete":
                f.write(f"[{timestamp}] Simulation complete\n")
                f.write("\n")
            else:
                f.write(f"[{timestamp}] {message}\n")
                f.write("\n")
```

→ This class writes trade transactions to a log file.

- This class logs messages from the observed object to a file.
- Each message is written to the log file with a timestamp.

PYTHON

```python
class TradingBot(Subject):
    """
    A trading bot implementation that uses strategies to make buy/sell/hold decisions.
    It also acts as a Subject in the Observer design pattern to notify observers about its state.
    """

    def __init__(self, coin_symbol):
        """
        Initialize the trading bot with the coin symbol and other necessary attributes.
        """
        self.strategy = None  # Strategy to be used for trading
        self.observers = []  # List of registered observers
        self.coin_symbol = coin_symbol  # The cryptocurrency symbol to trade
        self.df = None  # Dataframe to hold historical data
        self.api_key = os.getenv("BINANCE_API_KEY")  # Binance API Key from environment
        self.api_secret = os.getenv("BINANCE_API_SECRET")  # Binance API Secret from environment
        self.indicator_calculator = IndicatorCalculator()  # Indicator calculator object
        self.data_loader = DataLoader()  # Data loader objectt
```

→ This class acts as a **Subject** in the Observer design pattern. The Subject manages the observers connected to it and sends notifications to these observers when there is a change in its state.

PYTHON

```python
def register_observer(self, observer):
        """
        Add an observer to the list.
        """
        self.observers.append(observer)

    def remove_observer(self, observer):
        """
        Remove an observer from the list.
        """
        self.observers.remove(observer)

    def notify_observers(self, message):
        """
        Notify all registered observers with a message.
        """
        for observer in self.observers:
            observer.update(message)
```

→ This code provides dynamic communication between objects using the Observer design pattern. Observers can be added and removed from the list, and all registered **observers are notified when a state change occurs.** The register_observer method adds an observer, remove_observer removes an observer, and the notify_observers method calls the update method of all registered observers to notify them. This structure provides loose coupling between objects, allowing changes to be easily managed.

→ You might be wondering where we used this register_observer logic. As we asked to our T.A for this documentation, he told us that we don't need to add the main class into the UML diagram but I would like to add the code snippet where we are registering them into the Trading Bot so you would understand that it is a different implementation but the main purpose is the same.

PYTHON

```python
# This code snippet can be found at main.py

try:
    trading_bot = TradingBot(coin_symbol=symbol)
    visualizer = VisualizationObserver()
    logger = LoggingObserver()

    trading_bot.register_observer(visualizer)
    trading_bot.register_observer(logger)

    if trade_mode == "live":

        result = trading_bot.simulate_trading(interval=interval,   stop_loss=stop_loss,
initial_balance=self.current_balance)

    else:

        result = trading_bot.backtest_trading(interval=interval, check_date=lookback_days,
stop_loss=stop_loss, initial_balance=self.current_balance)

    print(result)

    self.current_balance = result

    self.balance_label.config(text=f"${self.current_balance}")

    messagebox.showinfo("Trading Bot", "Trading Bot has finished running.")
```

3.3.4: Singleton Pattern Implementation

PYTHON

```python
import matplotlib.pyplot as plt
from .observer import Observer

class VisualizationObserver(Observer):
    _instance = None  # Class-level attribute to hold the singleton instance

    def __new__(cls, *args, **kwargs):
        # Ensure only one instance of the class is created
        if not cls._instance:
            cls._instance = super().__new__(cls, *args, **kwargs)
        return cls._instance

    def __init__(self):
        # Check if instance is already initialized
        if not hasattr(self, "initialized"):
            self.trade_counter = 0  # Initialize the trade counter
            self.balances = []  # Track balance cumulatively
            self.actions = []  # Stores actions like "Buy", "Sell", "Hold"
            self.initialized = True

## Other Methods
```

- **new**: Called when creating an instance of the class. If the _instance variable is None, a new object is created. Otherwise, the existing object is returned.
- **init**: The hasattr(self, 'initialized') check ensures that the **init** method only works for the first object created.

PYTHON

```python
import datetime
import os
from .observer import Observer

class LoggingObserver(Observer):
    _instance = None

    def __new__(cls, *args, **kwargs):
        """Ensure only one instance of LoggingObserver is created."""
        if cls._instance is None:
            cls._instance = super().__new__(cls, *args, **kwargs)
        return cls._instance

    def __init__(self):
        """Initialize the log file, but only the first time the instance is created."""
        if not hasattr(self, 'log_file_initialized'):  # Check if already initialized
            self.log_file = "trade_log.txt"
            # Initialize the log file only if it doesn't exist
            if not os.path.exists(self.log_file):
                with open(self.log_file, "w") as f:
                    f.write("Timestamp, Trade Details\n")
                    f.write("----------------------------\n")
            self.log_file_initialized = True  # Mark as initialized

## Other Methods
```

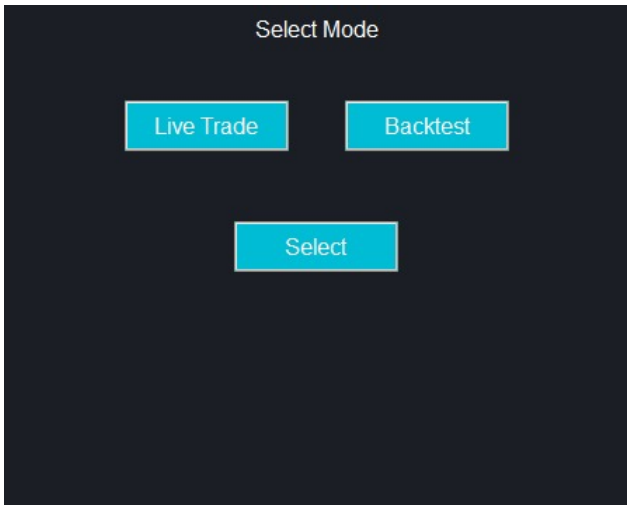
- A similar method is followed in this class. The **new** method ensures that a single instance is created.
- The log_file_initialised control prevents **init** from running more than once.

Thanks to the Singleton Pattern, only one instance of the `VisualizationObserver` and `LogObserver` classes is created during the application's lifecycle. This ensures:

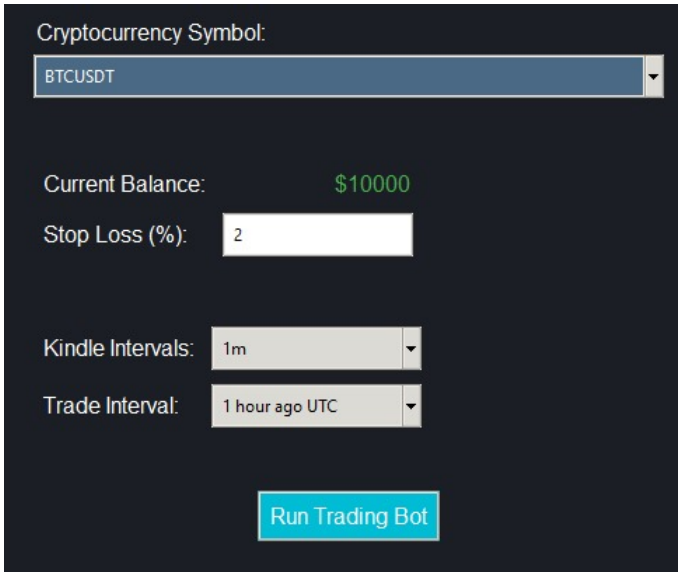
- Efficient resource usage by avoiding repeated object creation.
- Consistent access to the same instance across different parts of the application.
- Prevention of concurrency issues as all threads access the same instance.
  For example, in the `VisualizationObserver`, the `__new__` method ensures only one instance is created, and the `__init__` method ensures initialization is done only once.

4.1 : UI Parts

→ When the application first runs, the page you see on the screen comes up and asks you whether you want to do a backtest or live trade.

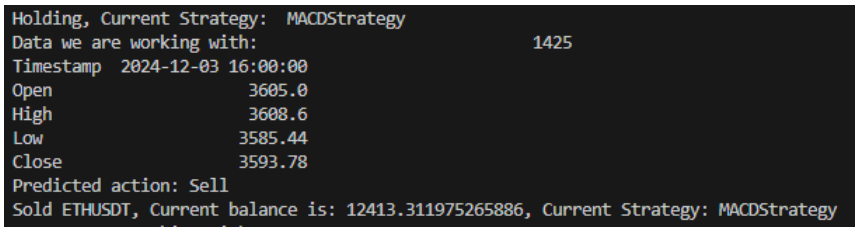


4.1.1 : Backtest Part



→ As seen above, the user can start back-testing after entering the coin they want to trade, the stop loss amount, and the interval value.

4.1. : Live Trade Part



→ If the user presses the live trade button, the above page opens and asks for some inputs from the user. After entering these inputs, he can start live trading. It is a difficult and potentially harmful process to find a signal and trade while live trading in short-term transactions. Therefore, it is not possible to trade live trade in a short time, but it can trade in long-term transactions.

```
Calculated Indicators
First trade, Buying...
Bought ETHUSDT, Current balance is: 10000, Current Strategy: RSIStrategy
Acquired Data
Calculated Indicators
Predicted action: Hold
Holding, Current Strategy:  RSIStrategy
Acquired Data
Calculated Indicators
Predicted action: Hold
Holding, Current Strategy:  RSIStrategy
Acquired Data
Calculated Indicators
Predicted action: Hold
Holding, Current Strategy:  RSIStrategy
Acquired Data
Calculated Indicators
Predicted action: Hold
Holding, Current Strategy:  RSIStrategy
```

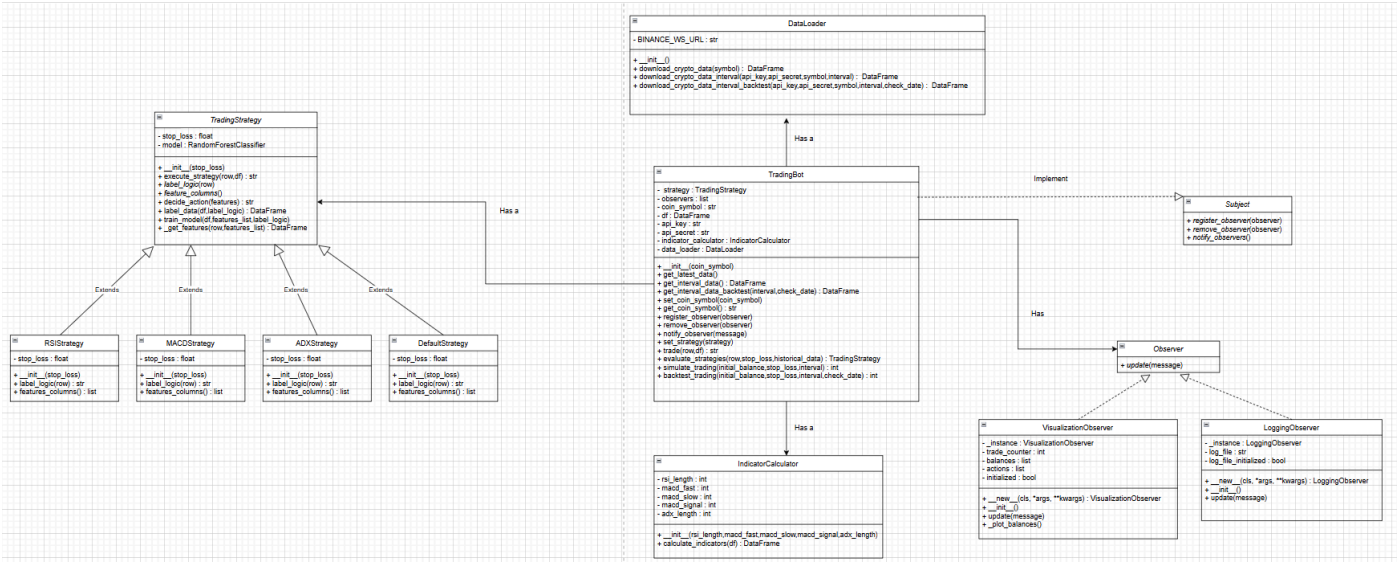
→ This is an output generated during live trading.

5.1: WebSocket Connections

```python
class DataLoader:  3 usages

    # WebSocket URL for Binance Kline (Candle) data stream

    def __init__(self):
        self.BINANCE_WS_URL = "wss://stream.binance.com:9443/ws/{symbol}@kline_1s"

    # Function to download crypto data via WebSocket
    # This function connects to the Binance WebSocket to fetch real-time kline data for the specified symbol.
    async def download_crypto_data(self, symbol):  1 usage
        url = self.BINANCE_WS_URL.format(symbol=symbol.lower())  # Format symbol for WebSocket URL
        df = []  # Initialize an empty list to hold the data rows

        try:
            # Connect to the WebSocket
            async with websockets.connect(url) as websocket:
                # Receive only one message and process it
                msg = await websocket.recv()  # Receive data from the WebSocket
                data = json.loads(msg)  # Parse the JSON data

                # Extract kline (candlestick) data from the message
```

WebSocket is a technology that enables real-time, bidirectional communication between web browsers and servers. Unlike the request-response model of traditional HTTP, WebSocket establishes a continuous connection so that data flow continues uninterruptedly. Thanks to this feature, it is used in many applications where real-time updates are required, such as instant messaging applications, online games, live score tracking. WebSocket ensures that new data generated by the server is instantly transmitted to the client, thus greatly improving the user experience.

6.1 : UML Diagram



We also understand the fact that it will be hard to read from here so we will **attach this image** into the .zip file.

7.1 : How Can You Run ?

1-You must first create a virtual environment.

```
cd trading-bot-project
```

```
python -m venv venv
```

2-You must then activate this .venv file.

- Windows : `venv\Scripts\activate`
- Linux/MacOS : `source venv/bin/activate`

3- Required project requirements to download

`pip install -r requirements.txt`

4- Set your Binance API key and secret as environment variables:

- On Windows: `setx BINANCE_API_KEY "your_api_key_here"` `setx BINANCE_API_SECRET "your_api_secret_here"`
- On macOS/Linux: `export BINANCE_API_KEY="your_api_key_here"` `export BINANCE_API_SECRET="your_api_secret_here"`

5- To Run Project
`python main.py`
