Automated Stock Market Trading System Documentation

Table of Contents

1. Introduction

The Automated Stock Market Trading System is a web-based application designed to automate stock trading activities using Angel One APIs. It allows users to create and execute trading strategies, monitor market conditions in real-time, and manage their trading portfolio efficiently.

2. Objectives

- Provide seamless integration with Angel One APIs for executing trades and accessing real-time market data.
- Implement user authentication and authorization mechanisms to ensure secure access to the application.
- Enable users to create and execute custom trading strategies based on real-time market analysis.
- Provide comprehensive tools for monitoring trade performance, calculating profit and loss, and managing trading positions.

3. Features

- Seamless Integration with Angel One APIs
- User Authentication and Authorization
- Real-Time Market Data Integration
- Option Chain Analysis for Equities and Indexes
- Customizable Trading Strategies
- Automated Leg Execution based on Market Conditions
- Profit and Loss Calculation for Each Leg and Total Portfolio
- Square Off Functionality for Exiting Trading Positions

4. Architecture

- Frontend: React.js for building the user interface components and views. The frontend interacts with the backend through REST API endpoints.
- Backend: Django REST Framework for providing REST API endpoints to the frontend. Handles HTTP requests, business logic, and database interactions.
- Database: SQLite for storing user data, trading strategies, and transaction history.
- External APIs: Integration with Angel One Smart API for accessing real-time market data and executing trades.

5. Installation

1. Clone the project repository from GitHub.
2. Install dependencies using `pip install -r requirements.txt`. The following packages will be installed:
    - APScheduler
    - asgiref
    - attrs
    - certifi
    - charset-normalizer
    - click
    - colorama
    - decorator
    - Django

    - django-cors-headers
    - django-rest-framework
    - djangorestframework
    - future
    - geocoder
    - idna
    - logzero
    - numpy
    - pandas
    - pip
    - pyotp
    - python-dateutil
    - pytz
    - ratelim
    - requests
    - six
    - smartapi-python
    - sqlparse
    - tzdata
    - tzlocal
    - urllib3
    - websocket-client
    - Celery
3. Set up environment variables for Angel One API credentials (username, password, token).
4. Configure Django settings for database connection and external API integration.
5. Run the Django development server using `python manage.py runserver`.

6. Configuration

- Set up user authentication settings in Django **settings.py**: Configure the authentication system provided by Django in the settings.py file. Define settings related to user authentication such as specifying the authentication backend, setting the login/logout URLs, configuring session management, and any additional authentication-related settings required by your application.
- Configure API keys, tokens, and other parameters for seamless integration with Angel One APIs: Obtain API keys and tokens from Angel One and configure them in your Django application, typically through environment variables or directly in the settings.py file. Additionally, configure other parameters such as base URLs, endpoints, authentication methods, and any other settings required for seamless communication with the Angel One APIs.

- **Integration with Angel One APIs:**

    - Obtain API keys and tokens from Angel One.
    - Store API keys and tokens securely, preferably using environment variables.
    - Configure authentication methods required for communication with Angel One APIs.

    - Include any other settings necessary for seamless integration with Angel One APIs.
- **Variable Assignment in settings.py:**
    - Create variables for username, password, and token.
    - Assign values to these variables.
    - Ensure the variables are accessible from other parts of the application by importing **settings.py**.
- **Usage:**
    - Wherever authentication or interaction with Angel One APIs is required in the application, import the settings from **settings.py**.
    - Utilize the variables for username, password, and token as needed

7. User Authentication Documentation:

User authentication is a critical component of any web application, particularly one dealing with sensitive financial data like a stock market trading system. This documentation provides a comprehensive overview of the user authentication functionalities implemented in the Automated Stock Market Trading System.

- ## Sign-in and Login:

  Sign-in:
    - Sign-in Form: Implemented a sign-in form allowing users to input their credentials.
    - Validation: Validated the input data upon submission to ensure accuracy and completeness.
    - Authentication: Authenticated users by verifying provided credentials against stored data in the system's database.

  Login:
    - Session Creation: Created a session for the user to maintain login across different application pages.
    - Django Authentication Views: Utilized Django's built-in authentication views and forms to streamline the login process.

- ## Forget Password and Reset Password:

  Forget Password:
    - Form Submission: Provided a form for users to input their email address.
    - Token Generation: Generated a unique token upon submission and sent it to the user's registered email address.
    - Email Notification: Sent a password reset link to the user's email along with instructions.

  Reset Password:

- Password Reset Page: Created a page where users can enter a new password.
- Token Validation: Validated the token received in the email to ensure authenticity and expiration.
- Password Update: Updated the user's password in the database with the new password provided.

- ## Verify User Email Addresses:

  - Account Activation: Sent a verification email containing a unique token upon user registration.
  - Verification Link: Directed users to click on the verification link in the email to activate their account.
  - Security Enhancement: Verified email addresses during the sign-up process to ensure valid and active email addresses, enhancing system security.

- ## API Endpoints:

  SignInUserAPIView: URL - `/SignInUserAPIView/` - Sign-in endpoint for users.

  EmailverifyOTP: URL - `/EmailverifyOTP/` - Endpoint for verifying user-entered email by OTP.

  Send_userdetail: URL - `/Send_userdetail/` - Sample code for sending user details to the frontend**.**

  LoginAPIView: URL – `/LoginAPIView/` - Endpoint for user login.

  ForgotPasswordAPIView: URL - `/ForgotPasswordAPIView/` - Endpoint for the forget password functionality.

  ResetPasswordAPIView: URL - `/ResetPasswordAPIView/` - Endpoint for the reset password functionality.

By implementing these user authentication features in the **views.py** file, the Automated Stock Market Trading System ensures secure access to user accounts, enhances user experience, and maintains the integrity of sensitive financial data.

8.Real-Time Data Integration:

- ## Real-Time Data Integration:

The Real-Time Data Integration module of the Automated Stock Market Trading System facilitates the retrieval and processing of real-time market data from Angel One Smart API. This section focuses on three specific API endpoints designed to fetch and present option chain data:

- **API Endpoints:**

1. **Company Name and Expiry Date Endpoint**
    - **URL:** `/Companyname_expiry/`
    - **Description:** This endpoint retrieves all available company names along with their corresponding expiry dates. It provides users with a comprehensive list of companies and their respective expiry dates, allowing for better decision-making when formulating trading strategies

2. **Option Data CE/PE and Expiry Date Endpoint**
    - **URL:** `/Option_Data_CEPE_expiry/`
    - **Description:** This endpoint allows users to fetch option data based on a specific company name and expiry date. It provides Call (CE) and Put (PE) option data for the selected company and expiry date, enabling users to analyse available options and make informed trading decisions.
3. **Expiry CE/PE Only Endpoint**
    - **URL:** `/Expiry_CEPEonly/`
    - **Description:** This endpoint retrieves expiry dates for Call (CE) and Put (PE) options only. It provides users with a concise overview of expiry dates available for Call and Put options across all companies, simplifying the process of identifying upcoming expiration dates.

Note: The described functionalities are implemented in the **margin.py** file.

9.Option Chain Analysis:

The Option Chain Analysis module extracts essential information from the option chain data and analyses it to identify potential trading opportunities:

- Extracting Option Chain Data:
    - The module extracts token, trading symbol, expiry date, company name, lot size, and strike price from the option chain data provided by Angel One Smart API.
- Analysing Option Chain Data:
    - Once the relevant data is extracted, the system analyzes it to identify potential trading opportunities based on user-defined criteria. This analysis involves identifying patterns, evaluating liquidity, assessing volatility, and applying trading strategies specific to the NFO (National Futures and Options) exchange.

By leveraging the Real-Time Data Integration module to fetch option chain data and the Option Chain Analysis module to analyse it, users can make informed trading decisions and capitalize on opportunities in the stock market effectively.

10. Trading Strategy:

The Trading Strategy module empowers users to devise customized trading strategies aligned with their market analysis and preferences. This module facilitates the definition of entry, target, and stop-loss criteria for each leg of the strategy.

API Endpoints:

1.Retrieve Leg Fields Endpoint:

- **URL**: `/RetrieveLegFields/`
- **Description**: This endpoint retrieves the leg fields for each trading strategy based on the strategy name. It sends the leg details to the frontend, allowing users to view and customize the parameters of their trading strategies. The frontend can then present these fields to users for modification or selection.

2.Strategy View Endpoint:

- **URL:** `/StrategyView/`
- **Description:** Upon receiving user inputs for strategy and leg details, this endpoint stores the provided information in the database. Users can define their trading strategies, including entry, target, and stop-loss criteria, through the frontend interface. The endpoint ensures the persistence of these strategies for future reference and execution.

3.Retrieve Strategy by User ID Endpoint

- **URL:** `/RetrieveStrategyByUserId/`
- **Description:** This endpoint retrieves all trading strategies associated with a specific user ID. By querying the database, it gathers strategy and leg data linked to the user's account and sends it to the frontend. Users can then access and manage their trading strategies conveniently through the frontend interface, including viewing, editing, and deleting them.

These API endpoints facilitate seamless interaction between users and the trading platform, empowering users to optimize their trading performance and capitalize on market opportunities effectively. Through these functionalities, users can refine and execute their trading strategies with precision and flexibility.

Note: The described functionalities are implemented in the **views.py** file.

11. Leg Execution:

The Leg Execution module is a critical component of a trading system responsible for executing various trading actions based on predefined conditions and market dynamics. Let's break down the API endpoints provided:

1.Order on LTP Endpoint:

- **URL:** `/Order_on_ltp/`
- **Description:**
- The Order_on_ltp endpoint facilitates the placement of orders based on the Last Traded Price (LTP) of a financial instrument. It accepts HTTP POST requests and is

responsible for monitoring the market price and executing orders when the LTP matches predefined conditions.

- **Code Explanation**:
- The Order_on_ltp class defines a view for handling HTTP POST requests to the /Order_on_ltp/ endpoint.
- It attempts to generate a session and obtain a JWT token for authentication with the Angel Broking API.
- API request headers containing authentication details are set up.
- For each trading leg with a status of "matching…", a connection is established with the Angel Broking API.
- LTP data is fetched for the trading symbol associated with the leg.
- The LTP is compared with the predefined price of the leg.
- If the LTP matches the predefined price, an order is placed using the provided parameters.
- The order ID and unique order ID returned from the API response are saved to the leg object.
- The leg's status is updated to 'placed'.
- This process continues for each leg that meets the criteria.
- Upon completion, a response indicating the successful completion of LTP checking and updating is returned, along with the count of orders placed.
- **Additional Information:**
- The endpoint is integrated into trading systems or applications to automate order placement based on real-time market conditions.
- By continuously monitoring the market price and executing orders when predefined conditions are met, traders can efficiently manage their trading strategies.
- The URL for accessing this endpoint is /Order_on_ltp/. It is typically defined in the URL configuration of the Django application to map to the corresponding view.

Note: The described functionalities are implemented in the **orderonltp.py** file.

2.Trigger on LTP Endpoint:

- **URL**: `/Trigger_on_ltp/`
- **Description:**
- The Trigger_on_ltp endpoint enables the triggering of orders based on specific conditions related to the Last Traded Price (LTP) of financial instruments. It accepts HTTP POST requests and is responsible for monitoring the market price and executing orders when the LTP matches predefined trigger prices.
- **Code Explanation:**
- The Trigger_on_ltp class defines a view for handling HTTP POST requests to the /Trigger_on_ltp/ endpoint.
- It attempts to generate a session and obtain a JWT token for authentication with the Angel Broking API.
- API request headers containing authentication details are set up.
- For each trading leg with a status of "matching...", a connection is established with the Angel Broking API.
- LTP data is fetched for the trading symbol associated with the leg.
- The LTP is compared with the predefined trigger price of the leg.
- If the LTP is less than or equal to the trigger price, the order is modified accordingly.
- The order modification involves updating the trigger price to the current LTP.

- The leg's status is updated to 'triggered'.
- Upon completion, a response indicating the successful triggering of orders based on LTP conditions is returned.
- **Additional Information:**
- The endpoint is integrated into trading systems or applications to automate order triggering based on real-time market conditions.
- By continuously monitoring the market price and executing orders when predefined trigger conditions are met, traders can implement sophisticated trading strategies.
- The URL for accessing this endpoint is /Trigger_on_ltp/. It is typically defined in the URL configuration of the Django application to map to the corresponding view.

Note: The described functionalities are implemented in the **orderonltp.py** file.

3.Order on Price Endpoint:

- **URL:** `/orderonprice/`
- **Description:**
- This endpoint allows users to set a target price or other criteria, and when the market price matches the specified value, the endpoint initiates the placement of an order for executing a trading leg associated with the provided leg_id.
- **Code Explanation:**
- The post method handles HTTP POST requests to this endpoint.
- It retrieves the leg_id from the request data.
- Authentication details and necessary headers are set up for making requests to an external API.
- Data related to the trading leg corresponding to the provided leg_id is fetched from the database.
- Order parameters are constructed based on the trading leg's details, including variety, symbol details, transaction type, etc.
- The code then enters a loop to continuously check market conditions until the market price matches the specified price set by the user.
- Once the market price matches the specified price:
- It places an order for executing the trading leg.
- Updates the leg's status to reflect the execution.
- Retrieves and saves additional order details, such as order status and unique order ID.
- Returns a response indicating the successful execution of the order.
- Exception handling is implemented to catch and handle any errors that occur during the process. If an error occurs, it updates the leg's status accordingly and returns an appropriate error response.

This endpoint enables users to automate the execution of trading legs based on specific price criteria, streamlining the trading process and allowing for timely execution of orders.

Note: The described functionalities are implemented in the **exit.py** file.

4.Exit on Trigger Endpoint:

- **URL**: `/exitontrigger/`
- **Description:**

- This endpoint allows users to define trigger conditions, such as specific price movements or indicators. When these conditions are met in the market, the trading leg associated with the provided orderid is automatically executed.
- **Code Explanation:**
- The post method handles HTTP POST requests to this endpoint.
- It retrieves the orderid and userid from the request data.
- Authentication details and necessary headers are set up for making requests to an external API.
- Data related to the trading leg corresponding to the provided orderid is fetched from the database.
- Exit trigger parameters are constructed based on the trading leg's details.
- The code then enters a loop to continuously check market conditions until the trigger condition is met.
- Depending on the type of option (Call or Put), it checks if the spot price matches the trigger price.
- If the trigger condition is satisfied, it places an order for executing the trading leg, updates the leg's status, and saves profit/loss information for the user.
- The put method handles HTTP PUT requests to start or stop the trigger based on the provided orderid.
- It sets a flag in the database to indicate whether the trigger should be stopped.

This endpoint integrates with an external trading API to automate the execution of trading legs based on user-defined trigger conditions, enhancing efficiency and enabling timely decision-making in trading operations.

Note: The described functionalities are implemented in the **exit.py** file.

5.Exit on Stop Loss Endpoint:

- **URL:** `/exitonstoploss/`
- **Description:**
- This endpoint is designed to trigger the execution of a stop-loss order for a given trading leg when the market price reaches a user-defined stop-loss price. It aims to help traders mitigate potential losses by automatically closing the position when the market price hits or exceeds the specified stop-loss threshold.
- **Code Explanation:**
- The post method of the exit_on_stoploss class handles HTTP POST requests to this endpoint.
- It retrieves the orderid from the request data.
- The code attempts to generate a session and obtain a JWT token for authentication with the Angel Broking API. If the token is invalid, an error is logged.
- API request headers containing authentication details are set up.
- A connection is established with the Angel Broking API.
- For each Leg object corresponding to the provided orderid, the code:

- Constructs parameters for placing a stop-loss order, including variety, trading symbol, transaction type, exchange, order type, product type, duration, price, square-off value, stop-loss value, and quantity.
- Makes an API request to retrieve the latest trading price (LTP) data for the trading symbol.
- Compares the retrieved LTP with the stored trigger price for the stop-loss order.
- If the LTP matches the trigger price, the stop-loss order is placed using the placeOrder method of the smartApi.
- The status of the stop-loss order is updated to 'stoploss executed' in the database.
- After processing all relevant trading legs, a response indicating successful completion of LTP checking and triggering is returned.
- **Additional Information:**
- This endpoint is responsible for automatically executing stop-loss orders based on predefined trigger conditions set by traders.
- By monitoring market prices and triggering stop-loss orders when necessary, it helps traders manage risk and protect their investment portfolios.
- The URL for accessing this endpoint is /exitonstoploss/. It is typically integrated into trading applications or systems to automate stop-loss order execution.

Note: The described functionalities are implemented in the **exit.py** file.

6.Exit Where It Is Endpoint:

- **URL:** `/exitwhereitis/`
- **Description:**
- This endpoint enables users to exit a trading leg at the prevailing market price. It provides flexibility to close the position at the current market rate without relying on predefined target prices or stop-loss thresholds.
- **Code Explanation:**
- The post method handles HTTP POST requests to this endpoint.
- It retrieves the orderid from the request data.
- Authentication details and necessary headers are set up for making requests to an external API.
- Data related to the trading leg corresponding to the provided orderid is fetched from the database.
- Order parameters are defined, including details such as variety, trading symbol, transaction type, quantity, etc.
- A loop is initiated to continuously monitor the market conditions and execute the exit order when conditions are met.
- Within the loop:
- An order to exit the trading leg at the current market price is placed.
- The status of the exit order and the trading leg is updated accordingly.
- Additional position data is retrieved and saved for further analysis.
- Exception handling is implemented to catch and handle any errors that occur during the process. If an error occurs, an appropriate error response is returned.

- Overall, these API endpoints enable automated trading actions, including entry, exit, and risk management, based on user-defined criteria and real-time market data, enhancing efficiency and effectiveness in trading operations.

Note: The described functionalities are implemented in the **exit.py** file.

12.Profit and Loss Calculation API Endpoints:

1.Cancel/Delete Endpoint:

- **URL:** `/cancel_delete/`
- **Description:** This endpoint handles the cancellation or deletion of specific orders or trading strategies. It accepts HTTP POST requests to cancel or delete orders or strategies.
- **View Class:** CancelAPI
- **Functionality:** Allows users to cancel or delete orders or entire trading strategies based on provided parameters.
- **Usage:**
- Users need to send a POST request to this endpoint.
- Parameters:
- legid: The ID of the order or strategy to be canceled or deleted.
- c_d: Specifies whether to cancel or delete. Accepted values are "cancel" or "delete".
- If the provided legid or c_d parameter is missing, the endpoint returns a 400 Bad Request response.
- If the specified legid is not found, the endpoint returns a 404 Not Found response.
- If c_d is set to "cancel", the corresponding order or strategy will be marked as canceled, and its status will be updated accordingly.
- If c_d is set to "delete", the corresponding order or strategy will be deleted from the system.
- **Example Usage:**
- To cancel an order with legid 123, send a POST request with legid=123 and c_d=cancel.
- To delete a strategy with legid 456, send a POST request with legid=456 and c_d=delete.
- **Response:**
- Upon successful cancellation or deletion, the endpoint returns a success message indicating that the operation was completed.
- If the provided c_d parameter is invalid, the endpoint returns a 400 Bad Request response with an error message.

Note: The described functionalities are implemented in the **profitloss.py** file.

2.Delete Strategy Endpoint:

This endpoint is designed to delete entire trading strategies. It accepts HTTP PUT requests to delete trading strategies.

- **Endpoint Details:**
- **URL:** `/delete_strategy/`
- **HTTP Method:** PUT
- **View Class:** DeleteStrategy
- **Parameters:**
- **strategy_id:** The ID of the trading strategy to be deleted.
- **Functionality:**
- Users need to send a PUT request to this endpoint with the strategy_id parameter.
- Upon receiving the request, the endpoint will attempt to delete the specified strategy.
- If the strategy is found and successfully deleted, the endpoint returns a success message indicating that the strategy has been deleted.
- If the provided strategy_id parameter is missing or invalid, the endpoint returns a 400 Bad Request response with an appropriate error message.
- If the strategy with the provided ID is not found, the endpoint returns a 404 Not Found response.
- Upon successful deletion, the endpoint returns a JSON response with a success message indicating that the specified strategy has been deleted.
- If the provided strategy_id parameter is missing or invalid, the endpoint returns a JSON response with a 400 Bad Request status and an appropriate error message.
- If the strategy with the provided ID is not found, the endpoint returns a JSON response with a 404 Not Found status and an error message indicating that the strategy was not found.

Note: The described functionalities are implemented in the **profitloss.py** file.

3.Check Profit and Loss Endpoint:

This endpoint calculates the profit and loss for each leg based on executed trades and current market prices. Additionally, it provides a summary of the total profit and loss for the entire trading portfolio.

- **Endpoint Details:**
- **URL:** `/checkpl/`
- **HTTP Method:** GET
- **View Class**: checkpl
- **Functionality:**
- Users can trigger this endpoint to retrieve detailed profit and loss information for their trading activities.
- It performs profit and loss calculations for each leg of executed trades and summarizes the total profit and loss for the entire trading portfolio.
- The endpoint fetches position data and calculates profit and loss based on the provided data.
- It also associates profit and loss data with the corresponding user and trading symbols.
- If profit and loss data is successfully calculated and saved, the endpoint returns a success message along with the details.
- If no matching trading symbols are found or if there is any error during the process, appropriate error responses are returned.
- **Usage:**

- Users can make a GET request to this endpoint to retrieve profit and loss information.
- The endpoint does not require any additional parameters in the request.
- Response:
- Upon successful calculation and saving of profit and loss data, the endpoint returns a JSON response with a success message and the details of the saved data.
- If no matching trading symbols are found, the endpoint returns a JSON response with a 404 Not Found status and a message indicating that no matching symbols were found.
- If there is an error during the process, such as an invalid token or serialization error, the endpoint returns a JSON response with a 500 Internal Server Error status and an error message.
- **Additional Information:**
- This endpoint is a crucial component of the trading system, providing traders with insights into their trading performance.
- By integrating profit and loss calculations, traders can analyze the profitability of their trades and optimize their trading strategies accordingly.

Note: The described functionalities are implemented in the **profitloss.py** file.

13.Theme:

1.Themes

- **Endpoint Details:**
- **URL:** `/ThemeView/`
- **HTTP Method:** POST
- **View Class:** ThemeView
- **Functionality:**
- The ThemeView endpoint allows users to customize the theme of their trading interface.
- Users can specify the colors for the header, navbar, body, and legs of the trading interface.
- Upon receiving a POST request, the endpoint extracts the user ID and theme color preferences from the request data.
- It retrieves the user associated with the provided user ID.
- The endpoint checks if a theme already exists for the user. If not, it creates a new theme instance.
- If a theme already exists, the endpoint updates the theme fields with the new color preferences.
- Finally, it saves the changes to the theme instance.
- **Usage:**
- Users can trigger this endpoint by sending a POST request.
- The request must include the following parameters:
- user_id: The unique identifier of the user.
- header: The color preference for the header of the trading interface.
- navbar: The color preference for the navigation bar of the trading interface.
- body: The color preference for the body (main content area) of the trading interface.

- legs: The color preference for the legs (individual trading components) of the trading interface.
- **Response:**
- Upon successful execution, the endpoint returns a message indicating whether the theme was updated or created.
- If the theme was updated, it returns a status code of 200 (OK). If a new theme was created, it returns a status code of 201 (Created).
- **Additional Information:**
- This functionality enhances user experience by allowing them to personalize the appearance of their trading interface according to their preferences.
- Users can tailor the colors of different elements of the trading interface to suit their visual preferences or to distinguish between different types of information.

Note: The described functionalities are implemented in the **theme.py** file.

2.Theme Retrieval

- **Endpoint Details:**
- **URL:** `/ThemeRetrive/`
- **HTTP Method:** POST
- **View Class:** ThemeRetrive
- **Functionality:**
- The ThemeRetrive endpoint retrieves the theme settings for a specific user.
- Users can request their theme settings by providing their unique user ID.
- Upon receiving a POST request, the endpoint extracts the user ID from the request data.
- It retrieves the user object associated with the provided user ID.
- The endpoint then fetches the theme instance corresponding to the user.
- Once the theme instance is retrieved, it serializes the theme data using the ThemesSerializer.
- Finally, it returns the serialized theme data as a response.
- **Usage:**
- Users can trigger this endpoint by sending a POST request.
- The request must include the following parameter:
- user_id: The unique identifier of the user for whom the theme settings are being retrieved.
- **Response:**
- Upon successful execution, the endpoint returns the serialized theme data for the user.
- The serialized data includes the color preferences for the header, navbar, body, and legs of the trading interface.
- The response status code is set to 200 (OK).
- **Additional Information:**
- This functionality enables users to retrieve their previously saved theme settings.
- Users can utilize this endpoint to restore their preferred theme settings after logging in or when accessing the trading interface from different devices.

Note: The described functionalities are implemented in the **theme.py** file.

14. Square Off Functionality

Square Off Functionality

This functionality empowers users to manually square off trading positions for legs that haven't reached their target or stop-loss prices. Users also enjoy the convenience of exiting all active trading positions with a single click.

- **Endpoint Details:**
- **URL:** `/squareofff/`
- **HTTP Method:** GET
- **View Class:** squareofff
- **Functionality:**
- The squareofff endpoint retrieves the current trading positions using the provided token and establishes a session with the trading platform.
- It assesses the net quantity of each trading position to determine if it needs to be squared off (closed).
- For each position with a non-zero net quantity, the endpoint initiates an order to square off the position.
- These orders are executed as market orders, ensuring immediate execution.
- After all positions are squared off, the endpoint provides the updated net positions.
- **Usage:**
- Users can activate this endpoint by sending a GET request.
- No additional parameters are necessary in the request.
- **Response:**
- Upon successful execution, the endpoint furnishes the updated net positions, signaling that all active trading positions have been squared off.
- If there are no active trading positions to square off, the endpoint communicates a message indicating that there are no positions to square off.
- **Additional Information:**
- This functionality offers users the flexibility to manually manage their trading positions, enhancing their control over their investments.
- By enabling users to exit all active trading positions with a single click, the functionality simplifies the process of closing positions and mitigating risk.

Note: The described functionalities are implemented in the **orderonltp.py** file.

15. Conclusion

Conclusion

The Automated Stock Market Trading System represents a significant advancement in the field of stock trading, offering a comprehensive platform that combines automation, real-time data analysis, and portfolio management features. Throughout this project, we have developed a robust system that streamlines the stock trading process and empowers users to make informed investment decisions. Below are the key points highlighting the system's capabilities and benefits:

1. **Trade Automation:** The system automates trade execution, allowing users to set up predefined trading strategies and execute trades automatically based on specified conditions. This automation minimizes manual intervention and ensures timely execution of trades, optimizing trading efficiency.
2. **Real-time Market Data Analysis:** By leveraging real-time market data, the system provides users with valuable insights into market trends, price movements, and trading opportunities. Advanced analysis tools enable users to make data-driven decisions and adapt their trading strategies dynamically to changing market conditions.
3. **Customizable Trading Strategies:** Users have the flexibility to customize trading strategies according to their preferences, risk tolerance, and investment objectives. They can define parameters such as entry and exit conditions, stop-loss levels, and profit targets, tailoring the strategies to suit their individual trading styles.
4. **Portfolio Management:** The system offers comprehensive portfolio management features, allowing users to track their investments, monitor portfolio performance, and analyze trading results. Users can review their trading history, assess the profitability of their trades, and identify areas for improvement.
5. **User-friendly Interface**: With its intuitive user interface, the system ensures ease of use and accessibility for traders of all experience levels. The interface is designed to provide a seamless trading experience, enabling users to navigate the platform efficiently and execute trades with confidence.
6. **Empowering Users:** By providing users with the tools and resources they need to succeed in the stock market, the system empowers them to take control of their financial future. Whether they are seasoned traders or newcomers to the world of investing, users can leverage the system's capabilities to enhance their trading performance and achieve their investment goals.

In conclusion, the Automated Stock Market Trading System represents a comprehensive solution for traders seeking to optimize their trading process and maximize their investment returns. By combining automation, real-time data analysis, and portfolio management features, the system equips users with the tools they need to thrive in today's dynamic and competitive market environment. With its user-friendly interface, customizable trading strategies, and robust performance, the system is poised to revolutionize the way traders engage with the stock market and achieve success in their trading endeavours.