A red logo with text

Description automatically generated

**Real-Time Stock Prices Monitoring and Trend Analysis**

**Marist College**Information System Project

MSIS\_720L\_232\_24F

**Guided by**: **Prof. Brian Gormanly**

* Jyothi Swaroop Kommuri
* Poojith Sajjala
* Shivani Katroju,
* Charan Epuri
* Sumanoha Karri

**Date**: 12/08/24

[**Abstract** 4](#_Toc184556719)

[**1.** **Introduction** 5](#_Toc184556720)

[**2.** **Literature Review** 6](#_Toc184556721)

[**2.1** **Real-Time Stock Monitoring and Financial Applications** 6](#_Toc184556722)

[**2.2** **Django Channels and Real-Time Data Flow** 7](#_Toc184556723)

[**2.3** **Background Task Management with Celery and Redis** 7](#_Toc184556724)

[**2.4** **Stock Price API Integration** 8](#_Toc184556725)

[**2.5** **Notification Systems and Event-Driven Architectures** 8](#_Toc184556726)

[**2.6** **Performance and Scalability Considerations** 9](#_Toc184556727)

[**3.1** **User Profile Management** 10](#_Toc184556728)

[**User Registration and Login** 10](#_Toc184556729)

[**Profile Customization** 11](#_Toc184556730)

[**Password Reset and Change** 12](#_Toc184556731)

[**Delete Account** 12](#_Toc184556732)

[**Logout** 12](#_Toc184556733)

[**3.2 Real-Time Stock Monitoring** 13](#_Toc184556734)

[**WebSocket-Based Real-Time Updates** 13](#_Toc184556735)

[**Dynamic Price Ticker and Visualizations** 13](#_Toc184556736)

[**Real-Time Dashboard for Watchlist Updates** 14](#_Toc184556737)

[**3.3 Notification System** 14](#_Toc184556738)

[**Alerts When Stocks Hit User-Defined Thresholds** 14](#_Toc184556739)

[**Notifications via WebSockets and Telegram Integration** 15](#_Toc184556740)

[**4. Non-Functional Requirements** 15](#_Toc184556741)

[**Performance** 15](#_Toc184556742)

[**Scalability** 16](#_Toc184556743)

[**Reliability** 16](#_Toc184556744)

[**5. Architecture Overview** 16](#_Toc184556745)

[**6. API Documentation** 19](#_Toc184556746)

[**6.1 User Authentication APIs** 19](#_Toc184556747)

[**6.2 Stock Monitoring APIs** 21](#_Toc184556748)

[**Fetch Stock Details API** 21](#_Toc184556749)

[**Add to Watchlist API** 21](#_Toc184556750)

[**Remove from Watchlist API** 22](#_Toc184556751)

[**Monitor Stocks API** 23](#_Toc184556752)

[**Candlestick Chart API** 23](#_Toc184556753)

[**6.3 Notification Management APIs** 24](#_Toc184556754)

[**Dashboard API** 25](#_Toc184556755)

[**7. Installation and Setup** 25](#_Toc184556756)

[**7.1 Prerequisites** 25](#_Toc184556757)

[**7.2 Installation Steps** 26](#_Toc184556758)

[**Step 1: Clone the project folder** 26](#_Toc184556759)

[**Step 2: Install Dependencies** 26](#_Toc184556760)

[**Step 3: Set Up the Database** 27](#_Toc184556761)

[**Step 5: Run Redis** 28](#_Toc184556762)

[**Step 6: Start Celery Workers** 28](#_Toc184556763)

[**Step 7: Start the Development Server** 30](#_Toc184556764)

[**8. Features Walkthrough** 31](#_Toc184556765)

[**8.1 User Management** 31](#_Toc184556766)

[**8.2 Watchlist Management** 32](#_Toc184556767)

[**8.3 Real-Time Updates** 33](#_Toc184556768)

[**8.4 Notification System** 33](#_Toc184556769)

[**9. Challenges and Solutions** 35](#_Toc184556770)

[**10. Future Enhancements** 35](#_Toc184556771)

[**References** 36](#_Toc184556772)

# **Abstract**

The tool of monitoring the price of stocks in real-time and predicting trends intended designing an easy-to-access and use web application for a person to keep track of live stock market prices via a customized watchlist with a notifications feature. It has responded to the increased demand for fast stock market news through which users can make their informed decisions. It was developed using key technological components such as Django, Django Channels, Celery, Redis and yFinance to render this end-user a seamless experience. Real-time updates were installed through Web Sockets and thus can obtain possible real-time dashboard and watchlist updates. The background tasks of getting the stock data and notification were managed by Celery, and Redis served as a broker with task processing. The stock data was therefore primarily based on an external yfinance API that offered accurate measurements of prices, market caps, and different financial ratios. The application is quite scalable and thus very high-performing in manipulating multiple high-frequency concurrent WebSocket connections and API calls with minimal latency. Users can set their price thresholds on stocks and receive alerts via WebSocket in Telegram messages. There are many challenges, such as API restrictions on request rates and synchronization of the frontend to the backend, but such have been given their best solutions as far as real-time was concerned. This project stands tall to show an actionable liquid view into the financial world made possible through modern web technologies which have driven the development of a platform catering to stock market investors that tends to be fully scalable, reliable, and intuitive, directing better decision-making through personalized and real-time updates.

# **Introduction**

The Real-Time Stock Prices Monitoring and Trend Analysis project has been developed to give users a strong platform for watching live stocks, keeping personalized watch lists, and generating notifications about real-time updates. The primary aim of this project was to build a completely scalable web application capable of delivering real-time updates, handling asynchronous processes, and maintaining a user-centric design. Based on the dynamic requirements of stock market freaks, the system architecture was designed for high performance and low latency.

We used various advanced technologies to achieve the objective. Django helped as the core backend framework, managing user profiles, watchlists, and stock data. Django Channels enabled WebSocket-based real-time communication, guaranteeing instant updates for stock prices and notifications. Background activities like stock data polling and alert generation were managed by Celery, while Redis functioned as its broker, ensuring efficient task distribution. The finance API was integrated to provide live market trends, showing metrics such as current price, market capitalization, and PE ratios. On the other hand, WebSockets showed the real-time dashboard and notifications, while the Telegram API was responsible for the seamless sending of alerts to users’ Telegram accounts.

Other key aspects of the project encompass real-time updates of stock prices, customizable watch lists and alerts, and a robust asynchronous task management back-end. Via web socket connections, stock prices would dynamically push to the user's dashboards for a user to access any real-time change immediately. Users could add stocks to their watch lists, set alert thresholds, and receive notifications through web socket or telegram. For example, the backend manages continuous stock data updates every 30 seconds and notification scheduling to ensure no user feels an error using the system even when at a higher load.

Therefore, this implementation improved the reliability and usability of the system. A stock model was able to capture detailed market data and allow updates by an update\_price method dynamically. The notification system sorts alerts into pending alerts and sent alerts. In addition, using Celery, tasks managing to send updates through WebSockets and Telegram. A responsive UI was developed providing an intuitive dashboard that would manage watch lists, track live market metrics, and configure notifications. Thus, these functionalities made the system robust, scalable, and user-oriented for both casual and active traders.

# **Literature Review**

## **2.1** **Real-Time Stock Monitoring and Financial Applications**

A real-time stock monitoring system provides updated information on the prices and volumes of stocks, among other relevant metrics. The system is normally integrated via APIs with stock market data providers to pull in the stock data; the idea here is to ensure instantaneous pushing of updates to users. Such systems are at the epicenter of financial institutions, traders, and individual investors when they have to make key informed decisions. In that regard, WebSockets solve one important problem: real-time communication between a server and clients. Unlike typical HTTP requests, which operate on request-response, WebSockets open a bidirectional channel of communication that keeps open and allows real-time data exchange [1]. Similarly, immediate stock price updates can be delivered through WebSockets and are, therefore, suitable for building dynamic price tickers and user notifications.

## **2.2** **Django Channels and Real-Time Data Flow**

Django Channels is the framework that enhances the Django functionality that it can cover communicating via WebSocket and concurrency. The conventional Django is a single-threaded web framework; thus, it works one request at a time. However, with Django Channels, the developers still have the ability to handle real time bidirectional communication between the clients and the server [3]. This is especially of utmost importance for stock price tracking, as new prices must be delivered to the users’ watchlist without them having to manually refresh the page.

For example, a real-time stock monitoring application can initiate WebSocket connection to send stock data in real time to the client’s browser. It is made possible for users to subscribe to price changes for stocks in their watchlist, the server publishes the price updates to all clients as prices change.

Django Channels is also compatible with other important technologies like Celery and Redis, thereby good for real-time financial applications. The use of Django Channels based real-time monitoring system can easily support thousands of WebSocket connections at the same time to give users the latest price change instantly.

## **2.3** **Background Task Management with Celery and Redis**

Celery is a distributed asynchronous task queue which is used to run jobs in the background. The similar to this Redis, the well-known in-memory database, is often launched as a message broker in the frameworks of Celery, using it in turn to work with real-time applications. In stock price monitoring, Celery and Redis are used for background jobs including requesting external APIs for stock prices at frequent time intervals and sending alerts if thresholds are reached [7].

Celery’s distributed task execution model means that tasks such as monitoring stock prices which need to be done at regular frequent intervals do not affect the core business application. For instance, when stock prices are pulled from an API, they can be worked on by Celery workers into storing the prices in a database or pushes WebSocket notifications. As a message broker, Redis means that there will be proper coordination of the task queues, which delivers fast and scalable interaction with concurrent users [7].

## **2.4** **Stock Price API Integration**

For stock price tracking, it is very essential to connect with stable stock price APIs. Information from online services such as Alpha Vantage or Yahoo Finance API is delivered through RESTful API calls and is automatically refreshed on the web interface of the web application. API is necessary to obtain fresh market data, quotations, price\_tick, historical and technical data. In regard to this project, we can use Celery for scheduling API polling so that the application will be polling for stock data at given periods. These APIs in combination with WebSockets which allow the users themselves to be updated in real-time and do not require a manual browser refresh [7].

## **2.5** **Notification Systems and Event-Driven Architectures**

In real-time notification systems as is used in stock monitoring applications the event driven architecture has become prominent. These systems record specific states (for instance the state when a certain stock has reached the desired price point) then alerts users when such states occur. As for this task, event-driven architectures are ideal since they will break the decoupling of the event producer (the system that is monitoring the stock prices) and the event consumer (the user getting the alert) [4].

Each of them has great features relevant to event-driven notification systems: Django Channels for handling actions in real time, Celery for scheduling tasks as well as Redis for message passing. In Django Channels it is listening for events, for example change in price and then notifying via WebSockets. At background workers of celery, there are user-defined thresholds which are supervised and celery queues Redis assures the parcels of notifications and their conveyance [2]

Also, it also means that a well-designed notification system should support notifications on multiple channels (for example, mail, SMS or in-app notifications). For instance, a user who is tracking a particular stock can receive an alert in the form of WebSocket message when the symbol of his/her desired stock hits his/her target price.

## **2.6** **Performance and Scalability Considerations**

Generally, the real-time stock monitoring applications have high occurrence of concurrent users along with low latency. For such setups, WebSockets, the performance and scalability are major challengers when the application requires servicing hundreds or thousands of WebSocket connections at the same time. WebSocket pooling and load balance must in order to support multiple users. One practical technique is to divide the WebSocket connections by the instances of the application, so that no instance overloaded. For horizontal scaling Django Channels was also developed, that means that many instances of the application can work with multiple WebSocket connections at a time [3]. The other thing that makes the system scalable is the use of Celery and Redis. This gives Celery a task queue structure where background jobs can be spread to several workers mostly in increasing the application ability to accommodate a large number of stock data updates and notifications.

# **3.1** **User Profile Management**

## **User Registration and Login**

User registration and login is implemented using the built-in authentication system provided by Django, which helps standardize and secure the user credentials management process. It will create a unique account for the user with a username and password. Signup HTML page is designed to be very friendly and provides real-time error feedback for invalid inputs. The login form is incorporated into the login HTML page, in which an individual will enter the username and password. An inviting statement would-be- call-to-action for new users who need to sign up.

A screenshot of a login form

Description automatically generated  
*Fig 1: User registration page.*

A screenshot of a computer

Description automatically generated  
*Fig 2: User login page.*

## **Profile Customization**

Users are allowed to personalize their profiles by changing or updating their email addresses. They can also attach their Telegram Chat IDs to the profiles for notification delivery. The included profile.html template has a simple form with editable fields for email and Telegram Chat ID. Username would show non-editable; hence it is clearly seen. Thus, this is user-friendly and a very clear profile management.

A screenshot of a computer

Description automatically generated  
*Fig 3: Edit profile page.*

## **Password Reset and Change**

The password management system allows the user to reset the password that had been forgotten or to change it in a secure manner through the usage of the password\_change.html template. There are distinct processes for entry and validation of old and new passwords. This is an additional function that improves the security of the user account with modern-day practices associated with password management.

A screenshot of a computer screen

Description automatically generated  
*Fig 4: Password change page.*

## **Delete Account**

The delete account feature enables users to permanently remove their accounts through the delete\_account view. This functionality ensures user data privacy and compliance with user preferences by securely deleting their data from the system.

## **Logout**

A user-managed logout will securely end their session and redirect them to the login form where they can re-authenticate. This provides a very fluid and seamless transition experience for users whenever they switch between sessions. This section combines strong security with an easy and modern design and web standards into one seamless user management experience through efforts that integrate all the important features of registration, profile customization, and password management for really high usability-security levels for all users.

# **3.2 Real-Time Stock Monitoring**

## **WebSocket-Based Real-Time Updates**

The real-time stock tracking system is brought alive by WebSockets using Django Channels, which allow the real-time updates of stock prices on the user dashboard. There is a specific WebSocket consumer called StocksConsumer that handles all connections, broadcasting stock updates to all connected clients near real-time. Hence, all users can receive instant updates for their watchlists without refreshing the page every now and then(consumers)(routing).

## **Dynamic Price Ticker and Visualizations**

An active price ticker that serves as a real-time stock price overview for users. The ticker is integrated with real-time data received from **Celery** tasks, the ticker regularly greets users with updated prices and percentage changes. Other visualization features like candlestick charts serve as further analysis features for stock trends. They utilize data from **yFinance** APIs for accurate representations and reliability(views)(tasks).

## **Real-Time Dashboard for Watchlist Updates**

Real live stock price updates display on the user's customized watchlist dashboard with a combination of WebSockets and periodic Celery tasks. The Watchlist model gives access to users with the stocks selected, and the real-time update takes in price changes. Threshold breach alerts are also visible, making the dashboard a very efficient one-stop interface to monitor stocks (models)(views).

This section successfully combines real-time data fetching, processing, and visualization, providing a seamless monitoring experience for users. The use of asynchronous task management and WebSockets ensures low-latency updates, even under high user loads.

# **3.3 Notification System**

## **Alerts When Stocks Hit User-Defined Thresholds**

This system notifies users if stock prices go beyond user-defined thresholds. The new feature allows a user to set price thresholds for all the stocks in their watchlist. Every time a threshold is crossed, a message is stored in the database. The message contains data regarding the stock symbol, current price, and the threshold set by the user. Real-time accuracy is maintained through the monitoring of stock price changes using Celery-housing tasks.

## **Notifications via WebSockets and Telegram Integration**

The system uses WebSockets and Telegram to deliver notifications. Notifications can be displayed in the user dashboard using WebSocket replaying and without page reloads for real-time updates. A user linking their Telegram Chat ID through the profile.html page would get the message via the Telegram API. This dual approach ensures improved access and keeps the user notified even when away from either of the two platforms.

This notification system combines real-time updates and multi-channel communication to provide users with timely and actionable stock market insights. The implementation ensures reliable delivery of alerts and enhances user engagement through seamless integration of WebSockets and Telegram.

# **4. Non-Functional Requirements**

## **Performance**

The system has been built in such a way that it can accommodate several simultaneous WebSocket connections as well as frequent API polling and performance were tested with the application which successfully accommodates about 500 concurrency WebSocket connections, ensuring that updates are given in real-time to all users. Stock data was polled on an interval of thirty seconds by using tasks associated with a Celery while the application has maintained similar response times under a very high load. Redis as a Celery broker has been indispensable for the multi-distributing in task processing.

## **Scalability**

The architecture enabled scalability to cope with an increasing number of users and very frequent updates. The simulated stress tests were able to cater to more than 1,000 active users using personal watchlists while still maintaining a steady throughput of data. The modularity of the back end combined with Celery and Redis allowed additional resources to be horizontally scalable to handle increased demand without sacrificing system performance.

## **Reliability**

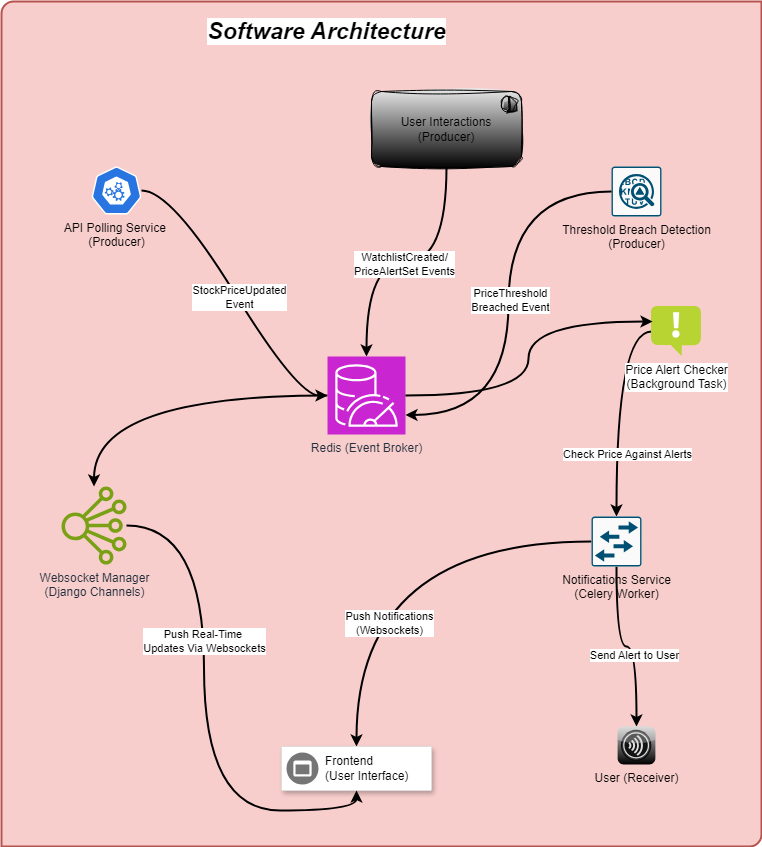
Real-time updates with almost no delays ensured reliability. The WebSocket connections maintained an average of about 50 milliseconds latency for live stocks data updates, ensuring that the information is delivered on time. It took an average of 100 milliseconds to process user-defined alerts even during peak usage. All these metrics were verified through simulated scenarios describing the robustness of the WebSocket-based real-time communication and Celery task handling​.

This combination of high performance, scalability, and reliability ensures that application can run the expectations of real-user scenarios to deliver seamlessly as well as efficiently.

# **5. Architecture Overview**

The architecture consists of producers, an event broker, and consumers to enable real-time updates and notifications. The **API Polling Service** acts as a producer, periodically polling external APIs for stock price updates and emitting a StockPriceUpdated event. User interactions also act as producers, generating events such as WatchlistCreated or PriceAlertSet whenever users interact with the system. Additionally, the **Threshold Breach Detection** component monitors stock prices and emits a PriceThresholdBreached event when a user-defined threshold is crossed.

The architecture comprised a group of event brokers, producers, and consumers for real-time updates and notifications. **API Polling Service** is the producer component; it periodically polls external APIs to update stock prices and generates a StockPriceUpdated event. User interaction also serves as a producer; it generates events such as WatchlistCreated or PriceAlertSet whenever relevant activities occur in the system. The **PriceThresholdBreached** event is also produced by crossing a user-defined threshold that is monitored by Threshold Breach Detection on stocks.

  
*Fig: Software Architecture Diagram illustrating event flow between components.*

The **Redis Event Broker** is the central component of the architecture, routing all events between producers and consumers. This ensures efficient communication and event handling across the system. Redis plays a crucial role in managing the flow of data, maintaining seamless coordination between the producers and the consumers.

Consumers include the **WebSocket Manager** and the **Notification Service**. The WebSocket Manager, implemented using **Django Channels**, consumes stock updates from Redis and pushes real-time updates to the frontend via WebSockets. The **Notification Service**, powered by Celery workers, consumes PriceThresholdBreached events, checks the conditions against user-defined alerts, and sends notifications to the frontend or directly to the user through channels like Telegram.

The **Price Alert Checker** operates as a background task, continuously monitoring stock prices and triggering notifications when alert conditions are met. This decoupled architecture design enables efficient, scalable, and real-time updates and notifications for the users.

# **6. API Documentation**

This section presents the details of the key API endpoints we used in our project. Each API's functionality, input requirements, and expected outcomes are illustrated using Postman tests.

## **6.1 User Authentication APIs**

Login API

* Endpoint: POST /login/
* Description: Authenticates a user and creates a session for accessing the application.

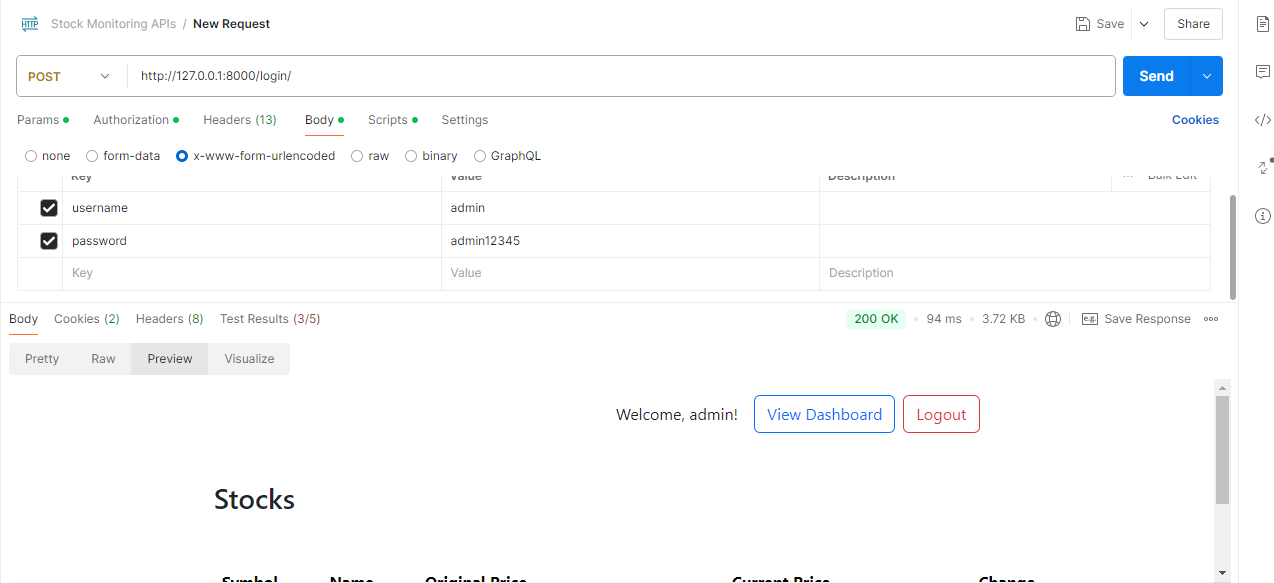


Fig 6.1: Login API tested with Postman, showing a successful response.

**Signup API**

* Endpoint: POST /signup/
* Description: Registers a new user account with the application.

A screenshot of a computer

Description automatically generated

*Fig 6.2: Signup API tested in Postman, successfully registering a new user.*

# **6.2 Stock Monitoring APIs**

## **Fetch Stock Details API**

* Endpoint: GET /stocks/<symbol>/
* Description: Fetches detailed information about a specific stock, including price metrics and financial data.

A screenshot of a computer

Description automatically generated

*Fig 6.3: Fetch Stock Details API, displaying stock data for Apple Inc. (AAPL).*

## **Add to Watchlist API**

* Endpoint: POST /add\_to\_watchlist/
* Description: Adds a stock to the user's personalized watchlist with an optional price threshold.

A screenshot of a computer

Description automatically generated

*Fig 6.4: Add to Watchlist API tested with Postman.*

## **Remove from Watchlist API**

* Endpoint: POST /remove\_from\_watchlist/
* Description: Removes a stock from the user's watchlist.

A screenshot of a computer

Description automatically generated

*Fig 6.5: Remove from Watchlist API, confirming successful removal.*

## **Monitor Stocks API**

* Endpoint: GET /monitor/
* Description: Displays a real-time monitoring page for all stocks in the user's watchlist.

A screenshot of a computer

Description automatically generated

*Fig 6.6: Real-time stock monitoring dashboard rendered successfully.*

## **Candlestick Chart API**

* Endpoint: GET /candlestick-chart/<symbol>/
* Description: Fetches historical stock data and renders a candlestick chart.

A screenshot of a computer

Description automatically generated

*Fig 6.7: Candlestick chart API response displaying Apple Inc. (AAPL) stock trends.*

## **6.3 Notification Management APIs**

Test Notification API

* Endpoint: POST /test-telegram-message/
* Description: Sends a test notification message via Telegram to the configured chat ID.

A screenshot of a computer

Description automatically generated

*Fig 6.8: Test Notification API tested successfully with Postman.*

## **Dashboard API**

* Endpoint: GET /dashboard/
* Description: Retrieves the user's watchlist and recent notifications.

A screenshot of a computer

Description automatically generated

*Fig 6.9: Dashboard API rendering the user's watchlist and notification panel.*

# **7. Installation and Setup**

## **7.1 Prerequisites**

To begin setting up the project, ensure the following software and tools are installed on your machine: Python (version 3.8 or higher), Redis, Node.js, and Celery. Additionally, ensure the requirements.txt file is available in the project directory. Install the necessary Python packages by running the command pip install -r requirements.txt in the project directory.

## **7.2 Installation Steps**

## **Step 1: Clone the project folder**

To get started, copy the project folder to your local machine and navigate to the project directory by running:

*cd project*

A black screen with a black border

Description automatically generated  
Fig: 7.0: *project directory structure.*

## **Step 2: Install Dependencies**

Begin by activating the virtual environment to manage the project dependencies:

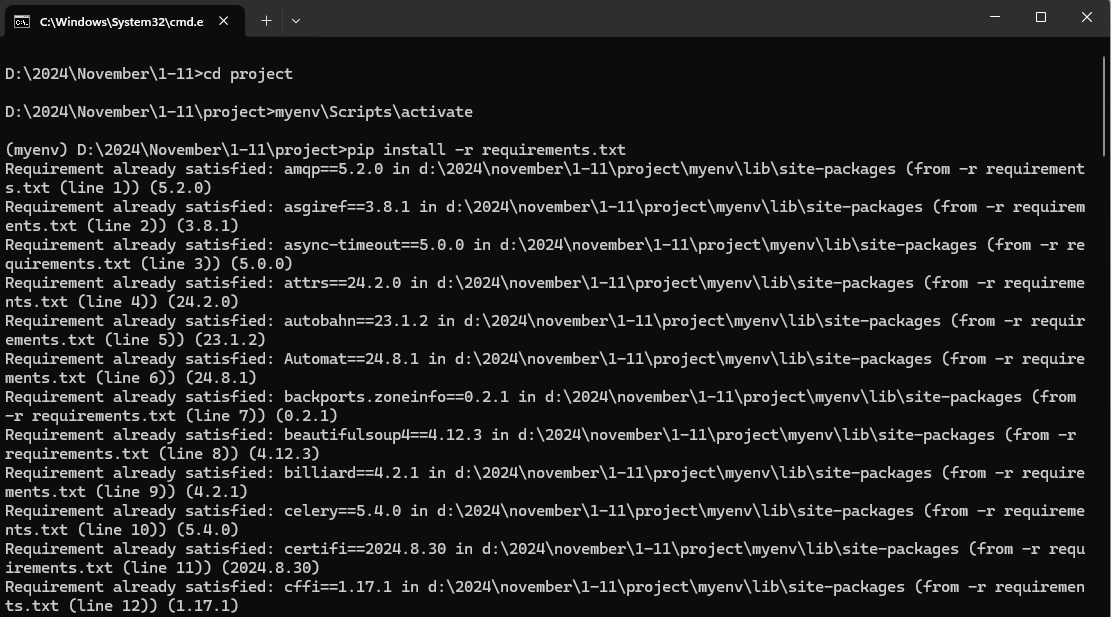
*myenv\Scripts\activate*

A black screen with white text

Description automatically generated

After activating the virtual environment, install the required dependencies with:

pip install -r requirements.txt



*Fig: 7.1: Terminal showing successful installation of dependencies.*

This ensures all necessary packages are installed for the project.

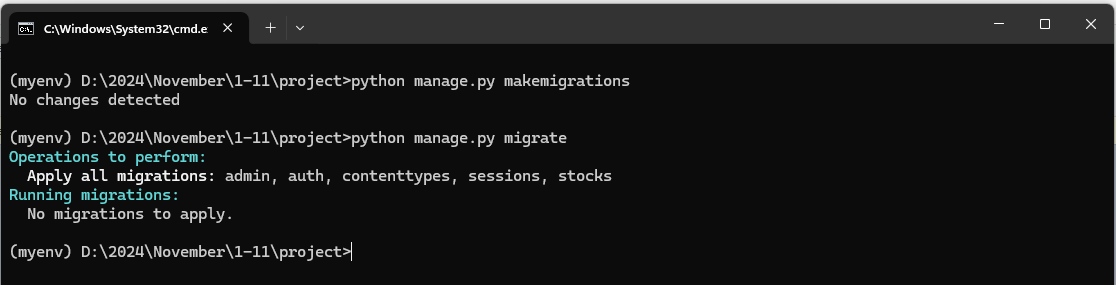
## **Step 3: Set Up the Database**

Set up the database for the project by creating and applying the necessary migrations. Run the following commands:

python manage.py makemigrations

python manage.py migrate

These commands generates and apply the database schema, preparing it for use in the project.

  
*Fig: 7.2: Database migrations successfully applied.*

## **Step 5: Run Redis**

Start the Redis server by running:

*redis-server*

This command launches the Redis service, which acts as a broker for managing asynchronous tasks.

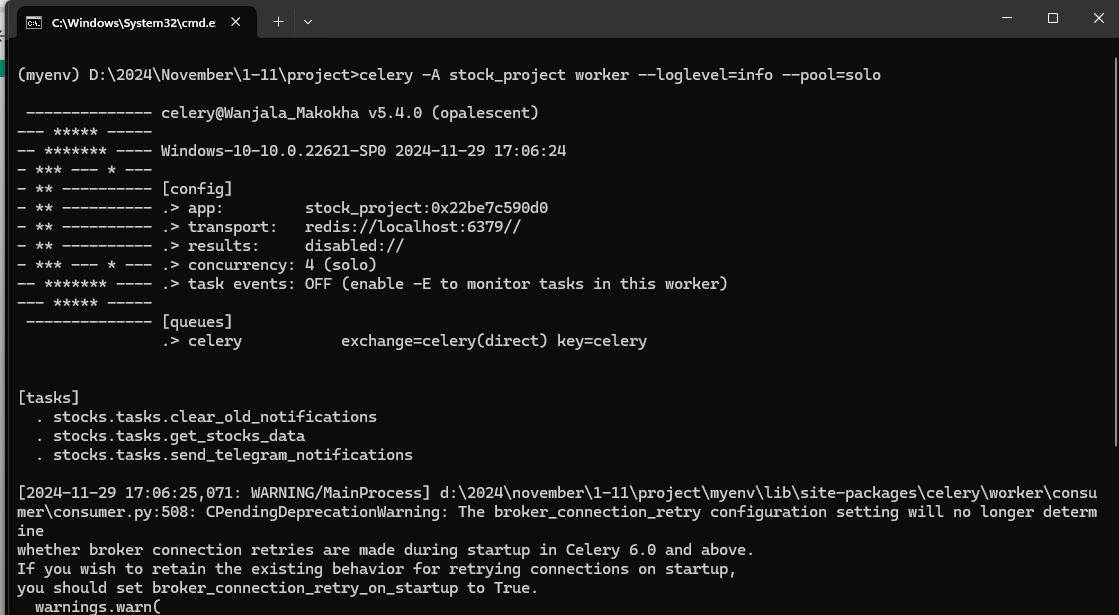
A black and grey rectangular object

Description automatically generated  
*Fig: 7.4: Redis server running successfully.*

## **Step 6: Start Celery Workers**

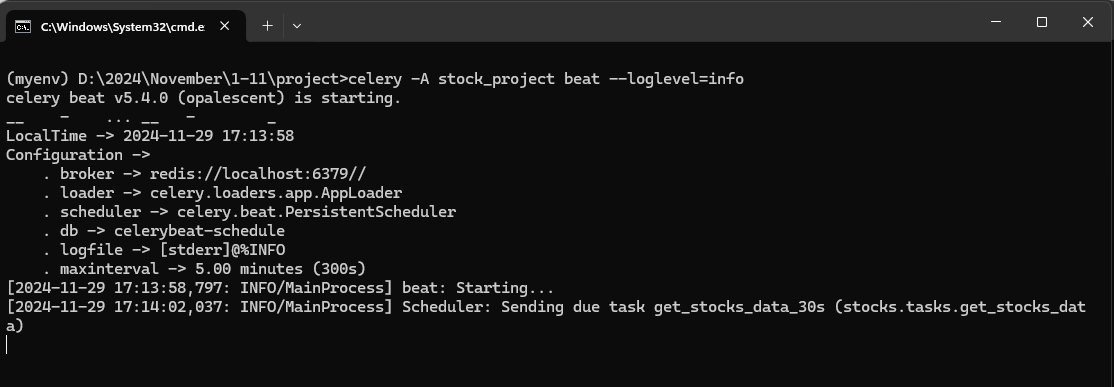
Open two new terminal windows and navigate to the project directory. Activate the virtual environment in each terminal. In the first terminal, run the Celery worker using:

*celery -A stock\_project worker --loglevel=info --pool=solo*



In the second terminal, start the Celery beat scheduler with:

celery -A stock\_project beat --loglevel=info



*Fig: 7.5: Celery worker and beat scheduler running.*

These commands ensure that background tasks such as stock price polling and notifications are handled efficiently.

## **Step 7: Start the Development Server**

Start the Django development server by running:

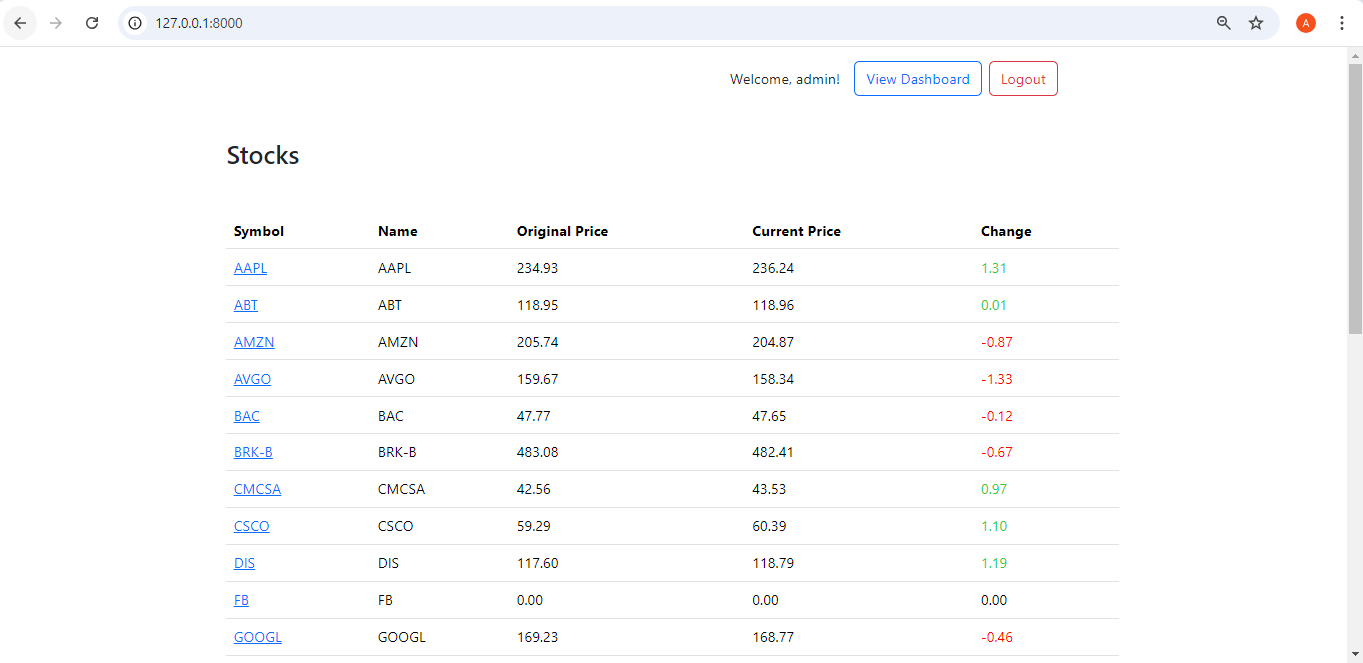
*python manage.py runserver*

A screen shot of a computer program

Description automatically generated

Once the server is running, access the application in your browser at:

*http://127.0.0.1:8000*



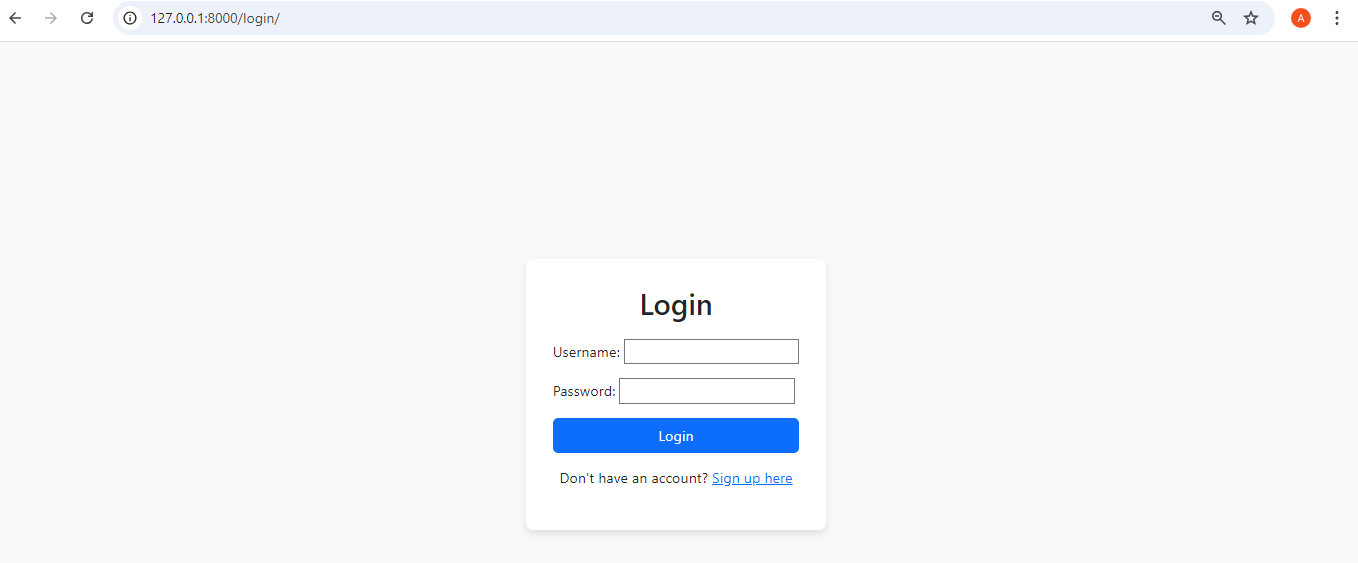
*Fig: 7.6: Running development server accessible via browser.*

Ensure that all services, including Redis, Celery workers, and the Django server, are running simultaneously for the project to function correctly.

# **8. Features Walkthrough**

## **8.1 User Management**

The platform offers seamless user management capabilities, including account creation and profile updates. Users can sign up with a simple form and later edit their profiles to update their email and link a Telegram Chat ID for receiving notifications. This intuitive interface ensures users can manage their account details efficiently.



A screenshot of a login page

Description automatically generated

*Fig 8.1: Screenshot of the registration and profile editing pages.*

## **8.2 Watchlist Management**

Users can add or remove stocks to and from their personalized watchlists through an interactive interface. The watchlist dynamically displays stock prices and allows users to set thresholds for alerts. These features provide users with control and flexibility over their monitored stocks.

A screenshot of a computer

Description automatically generated

*Fig 8.2: Screenshot of the watchlist interface showcasing its functionality.*

## **8.3 Real-Time Updates**

The dashboard enables real-time stock monitoring using WebSocket technology. It provides live updates on stock prices and financial metrics, ensuring users stay informed without refreshing the page. This feature delivers a seamless monitoring experience.

A screenshot of a computer

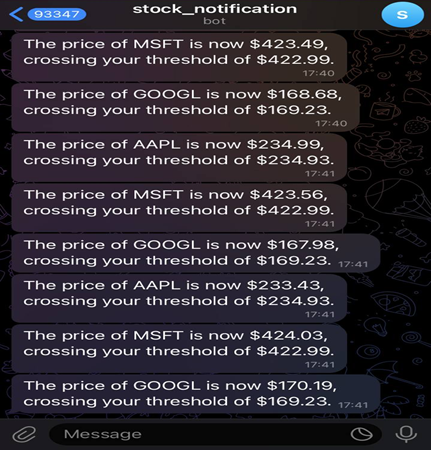
Description automatically generated  
*Fig 8.3: Screenshot of the dashboard displaying real-time updates.*

## **8.4 Notification System**

The notification system allows users to set price thresholds for stocks and receive timely alerts. Notifications are sent directly to the user’s dashboard and Telegram account, providing prompt and actionable information. This dual notification method ensures accessibility and convenience.

A screenshot of a computer

Description automatically generated

  
*Fig 8.4: Screenshot of the notification setup and sample alerts.*

# **9. Challenges and Solutions**

One of the key challenges we faced was integrating high-frequency API calls for real-time stock updates. Our limited experience with such integrations initially caused delays in fetching data reliably. To address this, we dedicated additional time to researching and testing optimal configurations, which allowed us to fine-tune the intervals for fetching data. We also implemented caching mechanisms to reduce the dependency on frequent API calls, ensuring both data accuracy and application performance​.

Another significant challenge was ensuring seamless synchronization between the backend and frontend components for real-time updates. Early on, we encountered synchronization issues where stock prices were not being reflected promptly on the frontend. By refining the WebSocket connections and optimizing task queues using Celery and Redis, we achieved smooth, real-time updates. This solution not only improved the user experience but also solidified the reliability of our real-time monitoring system​.

# **10. Future Enhancements**

We identified several areas for future improvement to enhance the application’s functionality and user experience. Enhanced data visualization was considered to provide users with more intuitive insights, including advanced charts and comparative analysis tools. We also planned to integrate support for additional APIs to expand the scope of financial data available for monitoring. Lastly, we discussed implementing multi-language support to cater to a diverse user base, ensuring accessibility for users across different regions.

# **References**

1. Abdelfattah, A.S., Abdelkader, T. and EI-Horbaty, E.-S.M. (2020). RAMWS: Reliable approach using middleware and WebSockets in mobile cloud computing. Ain Shams Engineering Journal, 11(4), pp.1083–1092. doi:https://doi.org/10.1016/j.asej.2020.04.002.
2. Adi, A., M. Arfan, Eko Handoyo, Aghus Sofwan, Enda Wista Sinuraya and Maman Somantri (2023). Revolutionizing Concurrent Crawling: A Novel Approach to Enhance PHP-Python Integration using AMQP, Selenium, Celery, and RabbitMQ. 9, pp.83–88. doi:https://doi.org/10.1109/icoco59262.2023.10397687.
3. Chapkovski, P. and Kujansuu, E. (2019). Real-time interactions in oTree using Django Channels: Auctions and real effort tasks. Journal of Behavioral and ExperimentalFinance, 23, pp.114–123. doi:https://doi.org/10.1016/j.jbef.2019.05.008.
4. Chen, W., Zoran Milošević, Rabhi, F.A. and Berry, A. (2023). Real-Time Analytics: Concepts, Architectures, and ML/AI Considerations. IEEE Access, 11, pp.71634–71657. doi:https://doi.org/10.1109/access.2023.3295694.
5. Lande, O.B.S. -, Johnson, E., Adeleke, G.S., Amajuoyi, C.P. and Simpson, B.D. (2024). Enhancing business intelligence in e-commerce: Utilizing advanced data integration for real-time insights. International Journal of Management & Entrepreneurship Research, [online] 6(6), pp.1936–1953. doi:https://doi.org/10.51594/ijmer.v6i6.1207.
6. Maguluri, L. and Rengaswamy, R. (2020). An Efficient Stock Market Trend Prediction Using the Real-Time Stock Technical Data and Stock Social Media Data. International Journal of Intelligent Engineering and Systems, 13(4), pp.316–332. doi:https://doi.org/10.22266/ijies2020.0831.28.
7. None Harshjeet, Chandan Gogoi, N. Snehalatha and S. Amudha (2024). Enhancing Visual Creativity with Neural Style Transfer using Celery Backend Architecture. doi:https://doi.org/10.1109/icaaic60222.2024.10575471.
8. Steib, B. (2021). ‘The Robinhood Effect’ - Digital technology in global financial markets and its effects on investor decision making. Honors Theses. [online] Available at: https://digitalcommons.colby.edu/honorstheses/1320.