Stock Market Management System

Ashkan Nikfarjam - ashkan.nikfarjam@sjsu.edu - 017214600
Maxim Dokukin - maxim.dokukin@sjsu.edu - 017025164
Ryan Fernald - ryan.fernald@sjsu.edu - 016884686
Seah Hsieh - sean-chen.hsieh@sjsu.edu - 014580566
Varun Ranjan - varun.ranjan@sjsu.edu - 017295642

December 2, 2024

Table of Contents

Goals and Description of the Application

The Stonks Market application is a virtual trading platform designed to address the complexities of stock investments, especially for beginner traders. Stock trading is risky and has a steep learning curve, so the user may be deterred. As a result, the main goal of this application is to create a safe environment for stock trading beginners to explore the world of stock. By leveraging real S&P 500 market data, the platform provides a risk-free environment where users can learn and practice stock trading. Users interact with the platform using virtual dollars to buy, sell, and analyze stocks while utilizing tools like watchlists, historical price trends, and market news.

Key Goals:

1. Enable beginners to build trading skills without financial risk.
2. Provide a simulation of real-market scenarios using real-time data.
3. Enhance user learning with analytics, portfolio tracking, and news updates.

Application/Functional Requirements and Architecture

Functional Requirements

1. User Management:
    a. Sign-up/login functionality.
    b. Virtual dollar wallet for deposits/withdrawals.
2. Stock Trading:
    a. Support for buying/selling stocks using S&P 500 tickers.
    b. Historical stock price tracking.
3. Analytics:
    a. Tools for portfolio performance analysis.
4. News Integration:
    a. Display and categorize stock-related news.
5. Watchlist:
    a. Customizable lists for monitoring favorite stocks.

Non-functional Requirements

1. a. Performance & Scalability
    a. i. Achieve a sub-150ms response time
    b. ii. Being able to handle a large amount of requests
2. b. Security
    a. i. Role-based access control
3. c. Data Integrity

    a. i. Achieve 98% referential integrity
    b. ii. Evaluate proper deletion methods(Cascade, Set Default, etc)
4. d. Maintainability
    a. i. Compose maintainable SQL scripts with standardized documentation
5. e. Recovery / Backup Security
    a. i. If crashes occur during mid-transaction / mid-action, the DBMS should roll back to its previous saved state within 5 minutes.

Architecture:

- Frontend: React for a dynamic and responsive user interface.
- Backend: Flask API, managing data requests and operations.
- Database: MySQL with a BCNF-normalized schema.
- Workflow: Users interact via a React-based UI, with backend operations through Flask APIs, fetching data from a MySQL database. A mock S&P 500 dataset is used for stock information.

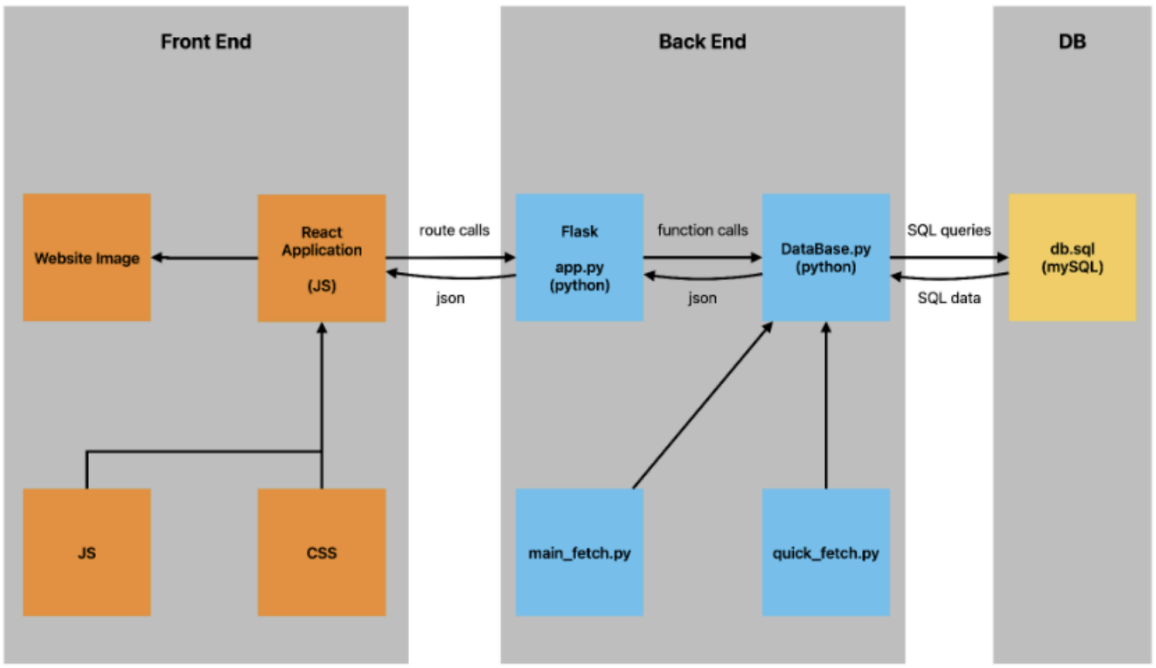


Figure 1: System Architecture Diagram.

ER Data Model

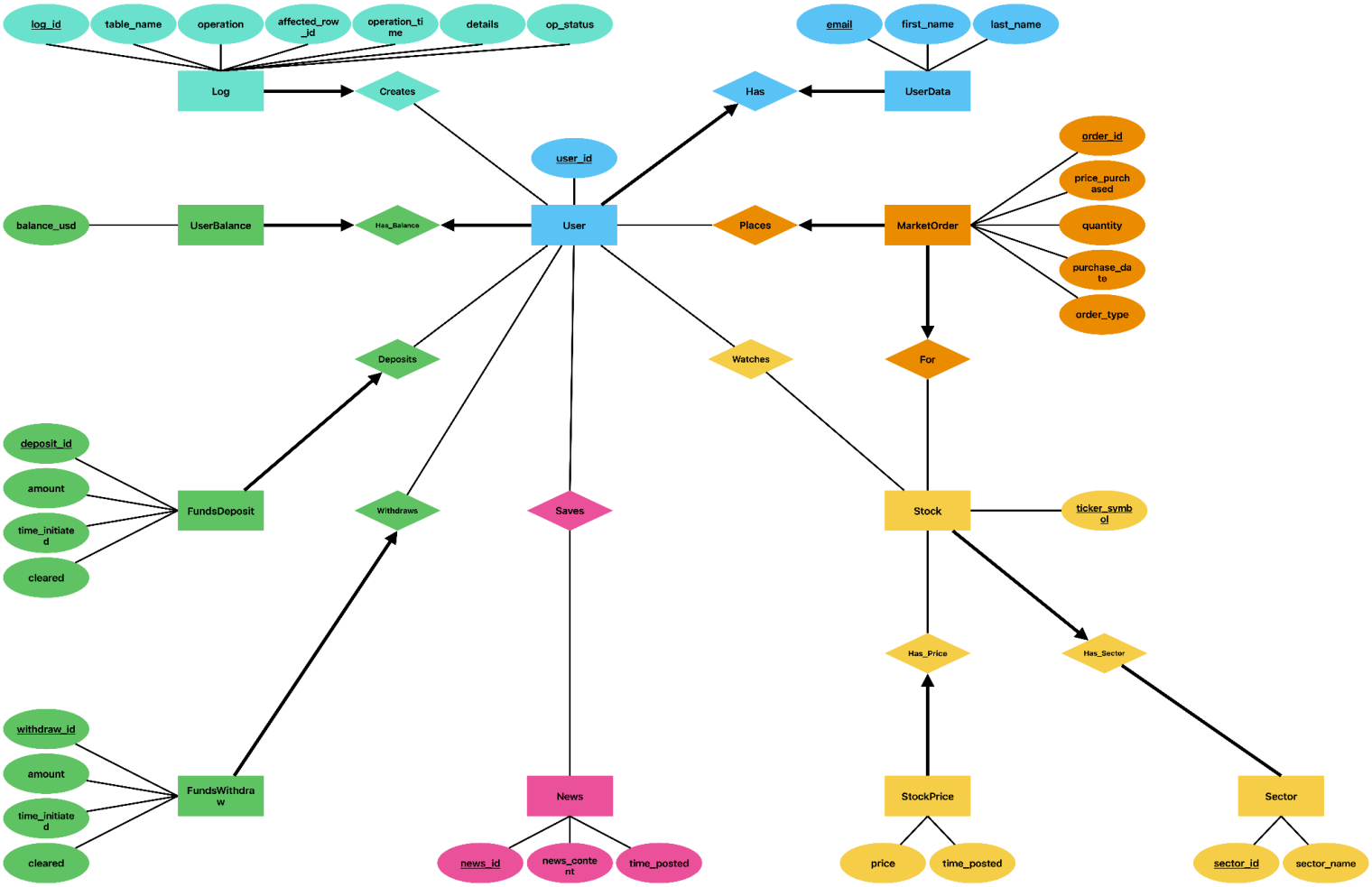


Figure 2: ER Diagram.

Entities:

1. **User**: Contains user-specific information such as `user_id` and `email`.
2. **UserData**: Stores additional user details like `first_name` and `last_name`.
3. **UserBalance**: Tracks the user's virtual wallet balance (`balance_usd`).
4. **FundsDeposit**: Records deposits into user accounts, including `amount` and `time_initiated`.
5. **FundsWithdraw**: Tracks withdrawals from user accounts, with attributes like `amount` and `cleared` status.
6. **Stock**: Represents tradable stocks with fields for `ticker_symbol` and associated `sector_id`.

7. **StockPrice**: Stores historical stock prices with timestamps (`price`, `time_posted`).
8. **Sector**: Categorizes stocks by market sector, such as `sector_name`.
9. **MarketOrder**: Records all user stock transactions, including `order_type`, `price_purchased`, and `quantity`.
10. **Watchlist**: Allows users to save and monitor specific stocks they are interested in.
11. **News**: Contains saved market news articles with fields for `news_id` and `news_content`.
12. **SavedNews**: Links users to news articles they have bookmarked.
13. **Log**: Captures all administrative database operations, such as `operation_type`, `affected_row_id`, and `details`.

Database Design

Tables and Relationships:

1. **User ↔ UserData**: One-to-one; `UserData` provides detailed information for each user.
2. **User ↔ UserBalance**: One-to-one; each user has a balance tracked in `UserBalance`.
3. **User ↔ MarketOrder**: One-to-many; users can initiate multiple stock transactions.
4. **MarketOrder ↔ Stock**: Many-to-one; each transaction involves a single stock.
5. **Stock ↔ StockPrice**: One-to-many; stocks have multiple historical price entries.
6. **Stock ↔ Sector**: Many-to-one; stocks are categorized by a specific sector.
7. **User ↔ FundsDeposit**: One-to-many; users can make multiple deposit transactions.
8. **User ↔ FundsWithdraw**: One-to-many; users can make multiple withdrawal requests.
9. **User ↔ Log**: One-to-many; users can create multiple log records.
10. **User ↔ Watchlist**: Many-to-many; users can add multiple stocks to their watchlist.
11. **User ↔ SavedNews**: Many-to-many; users can save multiple news articles.

Normalization to BCNF:

1. Ensured all tables have atomic attributes and are free of repeating groups (1NF).
2. Removed partial dependencies by creating distinct tables for dependent attributes (2NF).
3. Verified every non-trivial functional dependency had a superkey to satisfy BCNF.

Database Schema Highlights:

- Cascading rules (`ON DELETE SET NULL`, `ON UPDATE CASCADE`) ensure referential integrity.
- Constraints (e.g., `CHECK`, `NOT NULL`, `UNIQUE`) maintain data accuracy and validity.
- Multi-key primary keys are used in tables like `Watchlist` and `StockPrice` to uniquely identify relationships.

Major Design Decisions

1. Database Normalization: Implemented BCNF for minimal redundancy and data consistency.
2. Framework Choices:
    a. React for a seamless user experience.
    b. Flask for modular and scalable backend APIs.
3. Mock Real-Time Data: Stored and updated S&P 500 data in `StockPrice`.
4. Cascading Rules: Managed foreign key constraints for seamless deletions or updates.
5. Authentication: Integrated Firebase for secure and efficient user authentication.
6. FireBase Real-Time DB: To provide noSQL database framework for handling users IT support requests.

Implementation Details

- Frontend: React components to render navigation, dashboards, and data visualization.

Landing Page

When the user first launches our program they are greeted with the landing page. They can see an informative view of the most important headlines from the latest business news, and a view of Dow Jones industrial average for the last 6 months. The user has the option to adjust the chart to view different the NASDAQ and the S&P500, and Apple. They can also adjust the time frame of the chart to display data up to 1 year old. Below is an example of what the landing page looks like.

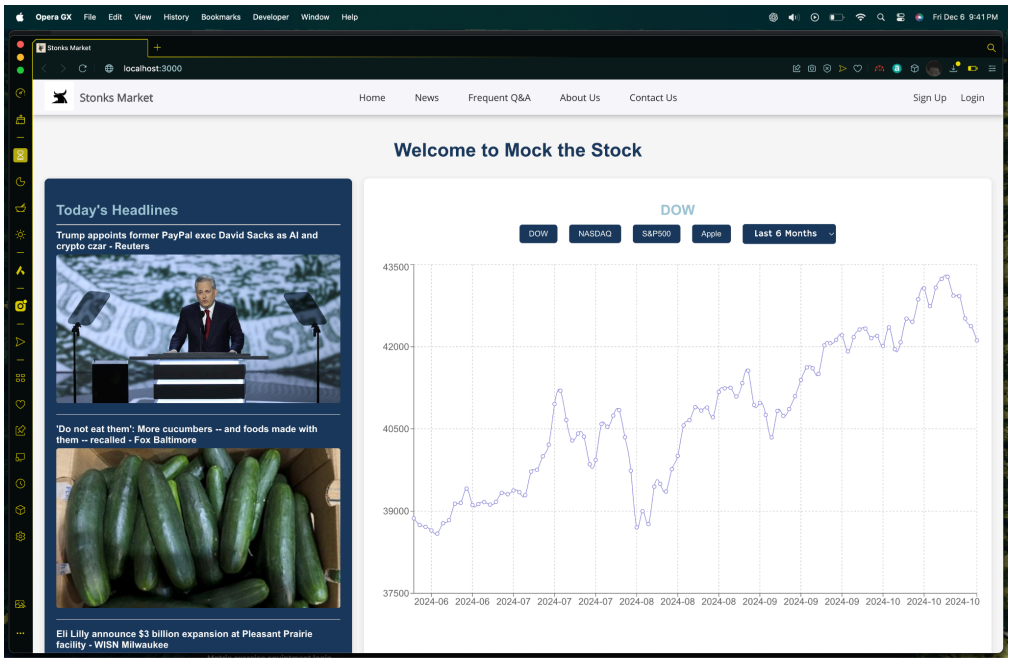


Figure 3: Landing Page.

The user does not need to be logged in to the site to use the news tab, which can give the user a great insight into the most important headlines related to the stocks they may be interested in. They can also look at what's going on in pop culture, in the world of Science, Healthcare, Technology. The user can select the articles to preview what they may be interested in the article list on the left and if they want to read more, they can click the Read More button under the article. If the user wants to know about any company in particular they can use the search bar to look up all the headlines related to that company. Below is an example of what the News Page looks like.

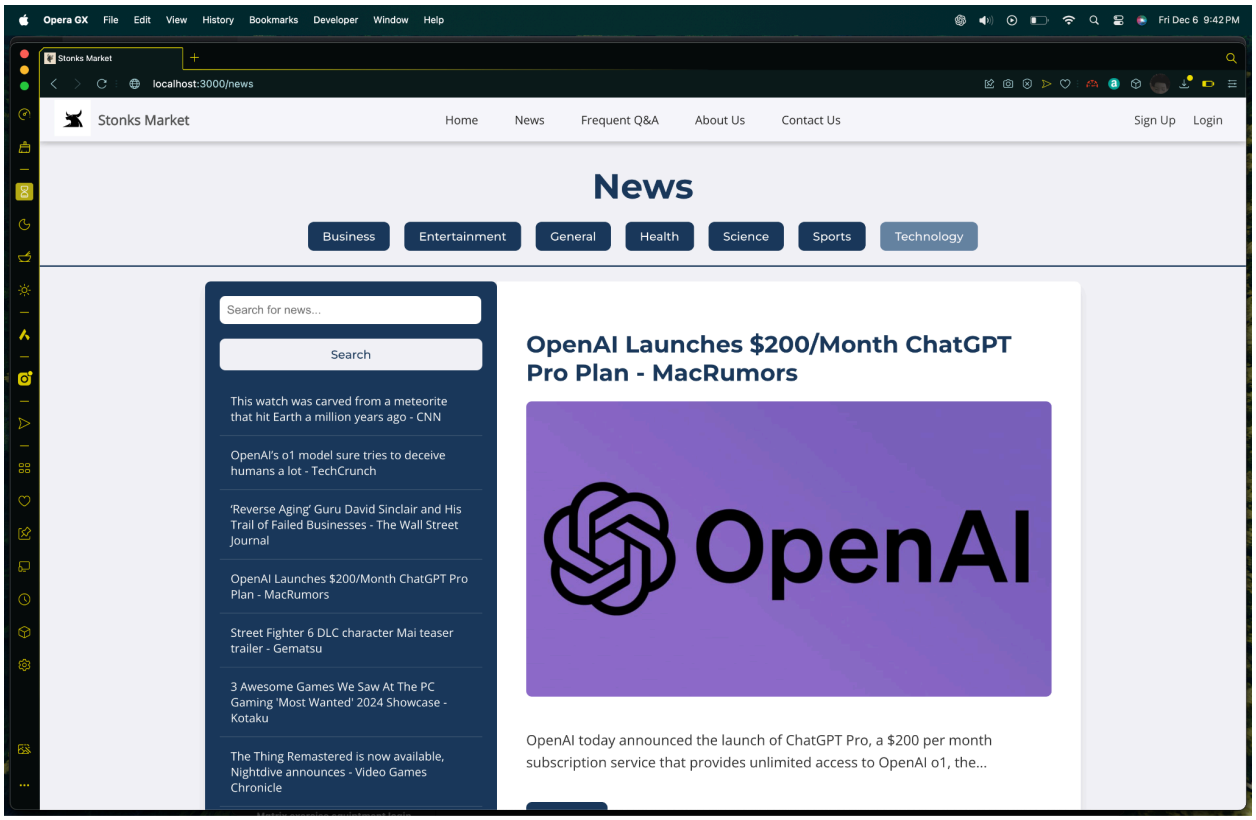


Figure 4: News Page.

Signup / Login

In order to most of the other features the user needs to register an account with us. They use can use the Sign Up button, or if they already have an account they can Login. All they need to Signup is an Email, Password and their name. We also have a connection with Google's Firebase API which allows the user to sign up / login with their Google Account. Signup and Login are displayed below:

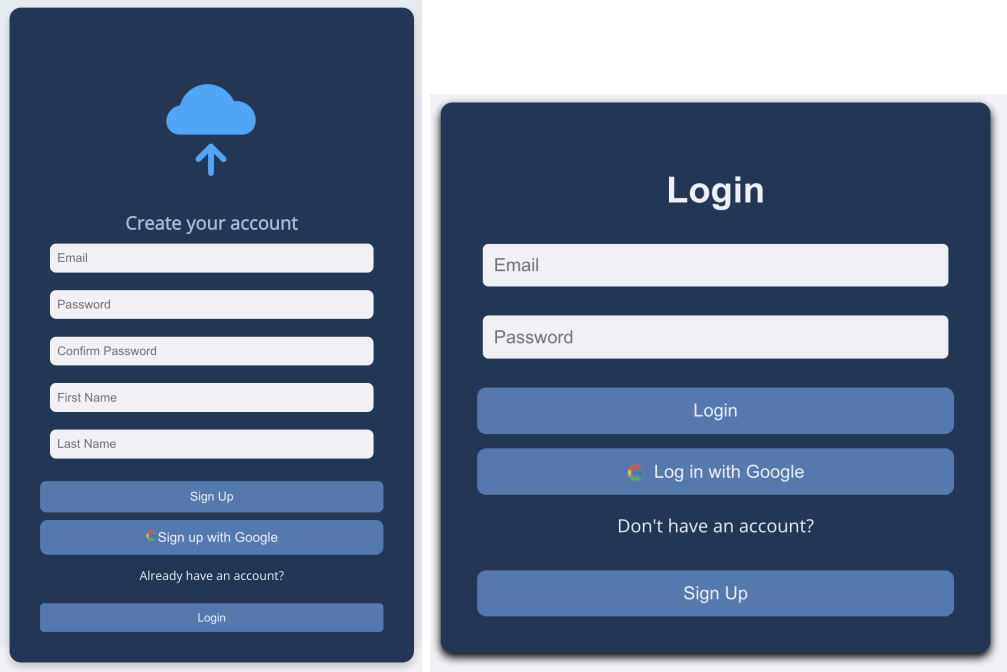


Figure 5: (Left) Sign Up, and (Right) Login Page.

Once the user Logs in they are greeted with our Dashboard. We have multiple Dashboards for our general Users and Administrators.

Dashboards:

1. User Dashboard: Grants users the ability to deposit, withdraw, and buy and sell stocks. Additionally, give the user an overview of their current account information.

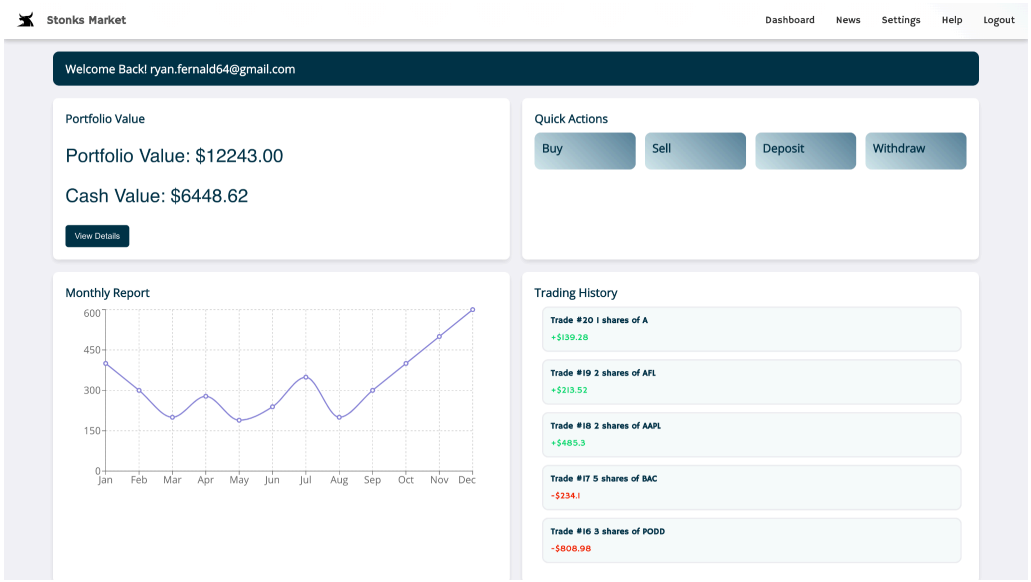


Figure 6: User Dashboard.

Admin Dashboard:

The administrator dashboard provides a robust component for maintaining and monitoring our app.

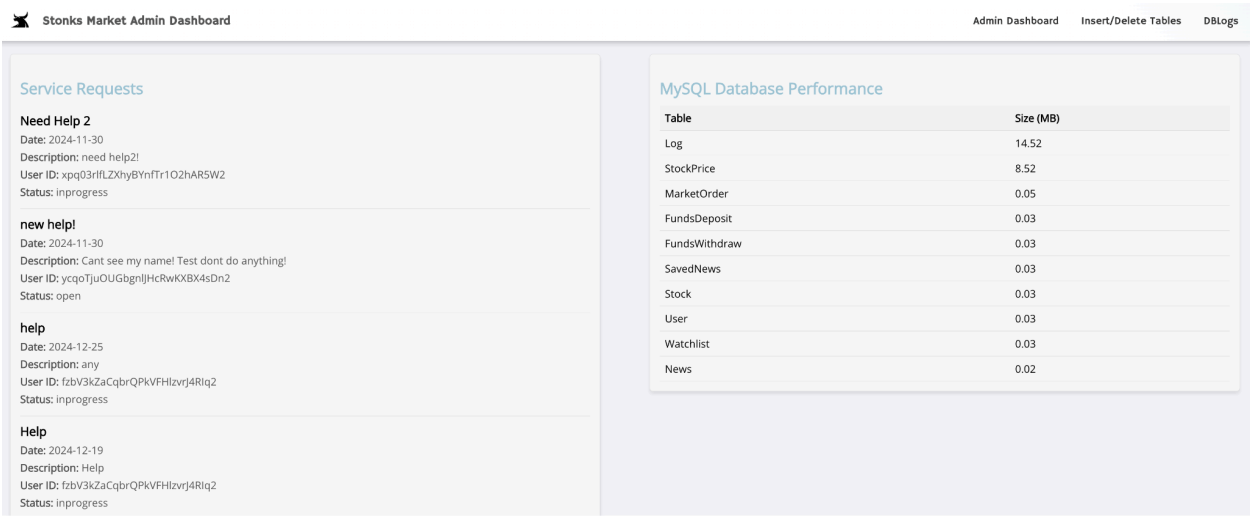


Figure 7: Admin Dashboard Page.

When a general user logs in they have access to quick actions to Deposit and Withdraw as seen in figure 8 below, to manage funds in their account. When the user clicks the deposit or withdrawal button the amount which they have deposited / withdrawn is instantly sent to the database as our routers manage the connection to the database which sends and receives the data to the database. The user can see an update in their Portfolio Value and Cash Value reflective of that transaction they made to their balance.

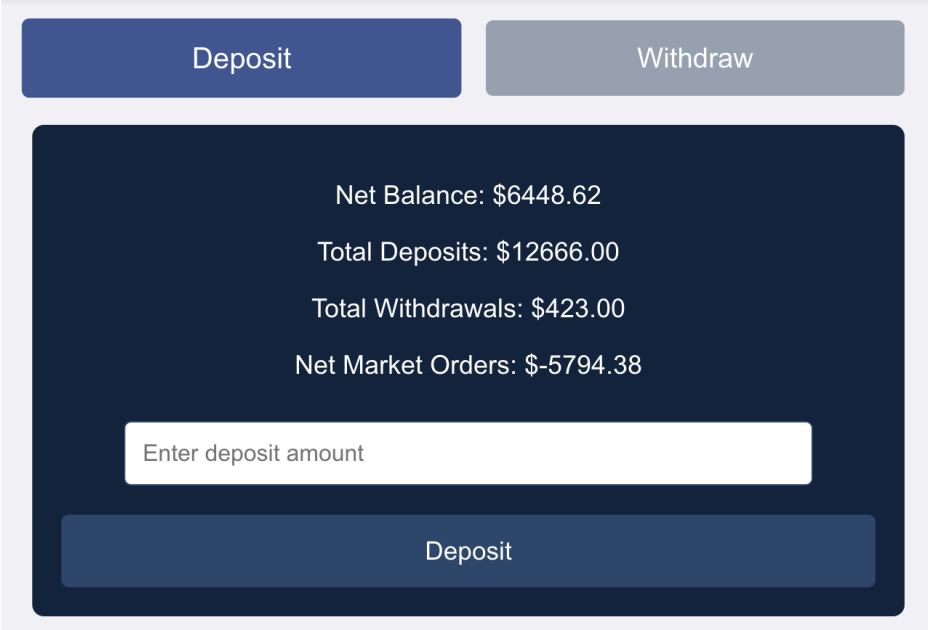
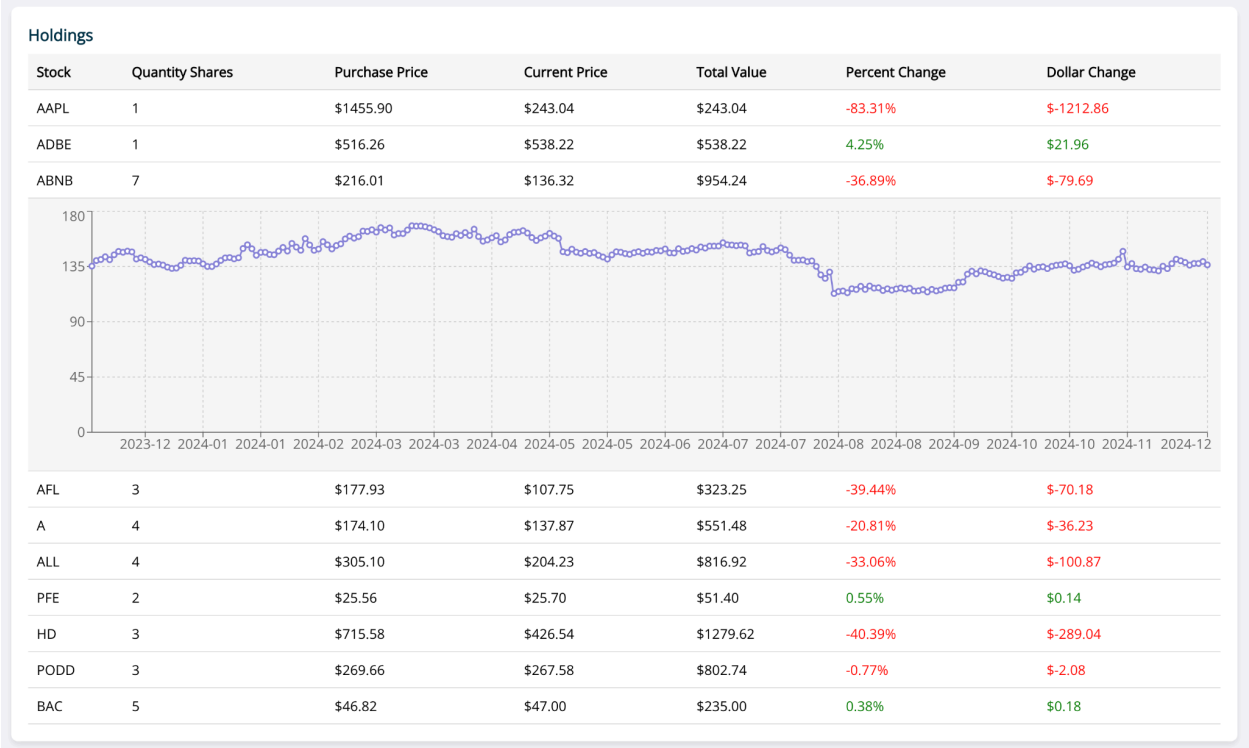


Figure 8: Deposit and Withdrawal page.

Now that our user has some funds in their account they have the opportunity to buy and sell stocks from their account with the buy and sell pages. As currently implemented the user can only buy full shares of each equity, so there are no partial or fractional shares. We have functions which will be discussed in more detail later, quick_fetch, which gets the latest stock pricing for each stock in the S&P500. For now, mostly because of storage limitations in our database, we decided to only hold the stocks listed in the S&P500. The When the user accesses the Buy Stock page they can Search for the Stock by their Ticker Symbol, Company Name. If they want to see all the stocks listed in a specific secretary of the economy they can search the sector in the search bar. Displayed below is the User Buy page.

Figure 9: UserBuy Page.

Once the user has clicked the Purchase Stock button the stock information will instantly be connected to our database with the stock manipulation router which sends the information as a Market Order. The Cost of the transaction is reduced from the User's Cash balance. Going back to the User's Dashboard we can see the Holdings Table has been updated with the stock purchased. The holdings table shows all market orders. Each row in the table is clickable and will display a beautiful chart of the stock's 1 year price history. One of our big goals is to feed the user with as much information as possible and we believe this 1 year price history is an excellent visualization to help the user decide which stocks are worth holding and for how long. Shown below is our Holdings Table.



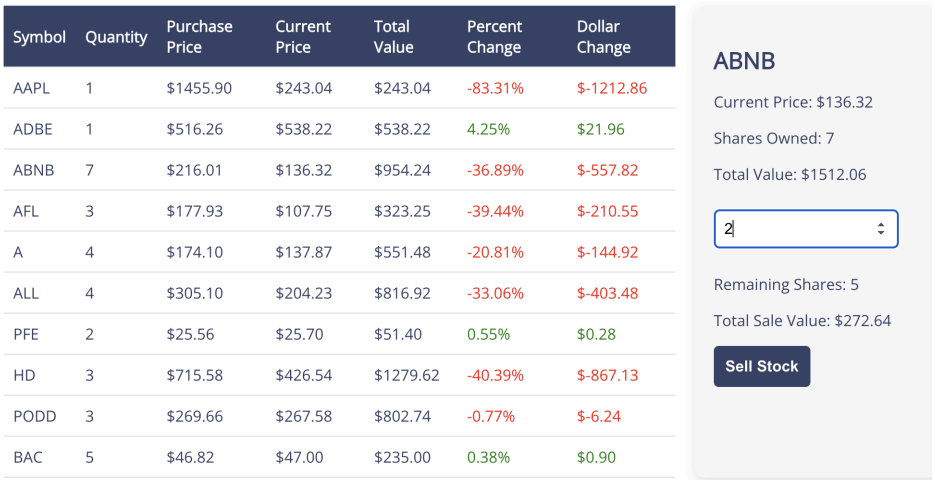
Holdings

Stock	Quantity Shares	Purchase Price	Current Price	Total Value	Percent Change	Dollar Change
AAPL	1	$1455.90	$243.04	$243.04	-83.31%	$-1212.86
ADBE	1	$516.26	$538.22	$538.22	4.25%	$21.96
ABNB	7	$216.01	$136.32	$954.24	-36.89%	$-79.69

AFL	3	$177.93	$107.75	$323.25	-39.44%	$-70.18
A	4	$174.10	$137.87	$551.48	-20.81%	$-36.23
ALL	4	$305.10	$204.23	$816.92	-33.06%	$-100.87
PFE	2	$25.56	$25.70	$51.40	0.55%	$0.14
HD	3	$715.58	$426.54	$1279.62	-40.39%	$-289.04
PODD	3	$269.66	$267.58	$802.74	-0.77%	$-2.08
BAC	5	$46.82	$47.00	$235.00	0.38%	$0.18

Figure 10: Holdings Table

If the user has purchased the same stock multiple times, the Purchased Price displayed in the holdings table is the average of all the Purchase Price for each share weighted by the number of shares they own from each purchase. Displayed in the holdings table are the metrics for Percent Change and Dollar Change so the user can easily see how much profit they've made, or decide to sell it if they're losing money. The User Can Sell a Stock from their holdings, in exchange for cash. Shown below is our User Sell page where a user can choose how many shares of Equity they own.



Symbol	Quantity	Purchase Price	Current Price	Total Value	Percent Change	Dollar Change
AAPL	1	$1455.90	$243.04	$243.04	-83.31%	$-1212.86
ADBE	1	$516.26	$538.22	$538.22	4.25%	$21.96
ABNB	7	$216.01	$136.32	$954.24	-36.89%	$-557.82
AFL	3	$177.93	$107.75	$323.25	-39.44%	$-210.55
A	4	$174.10	$137.87	$551.48	-20.81%	$-144.92
ALL	4	$305.10	$204.23	$816.92	-33.06%	$-403.48
PFE	2	$25.56	$25.70	$51.40	0.55%	$0.28
HD	3	$715.58	$426.54	$1279.62	-40.39%	$-867.13
PODD	3	$269.66	$267.58	$802.74	-0.77%	$-6.24
BAC	5	$46.82	$47.00	$235.00	0.38%	$0.90

Figure 11: UserSell page.

When the user Sells a Stock from their holdings table their cash Balance is updated as well as the holdings table with the appropriate number of shares remaining from their Sale. Each time the user buys or sells a stock the Trading History is updated with information about each stock purchase or sale with the number of shares, the ticker symbol and the cost / sale value of each transaction. Shown below is our Trading History:

Figure 12: Trading History.

Monitor the storage consumed but our DB tables:

This component displays the performance of the database by fetching the space consumed by each entity in SM's database from the app's backend and sort and display the received information. Monitoring space consumption allows future capacity planning in large scale databases, allows developers to improve and tune the performance of the database as data grows. Enabling efficient storage management(Surajit Chaudhuri et al, *Continuous Monitoring of RDBSM*).

Service Request Ticket handling:

The other component of the admin dashboard is the Service Request handling component. Where amind user can view the help requests submitted by the front end users and leave notes an update the status of the ticket.

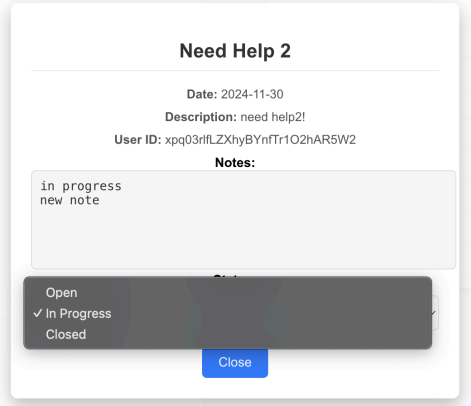


Figure 13: Support Ticket Handling.

Admin Dashboard Table Insertion and Deletion:

This component utilizes DataBase/App Admin to insert and delete data from the SQL server. This helps to improve maintenance of the database. In instances such as the user's buy/sell transaction is missing, then DB admin is able to insert that transaction into the relevant DB table manually. As well as monitoring the storage space consumed by our entities. As seen in Figure 14 Below.

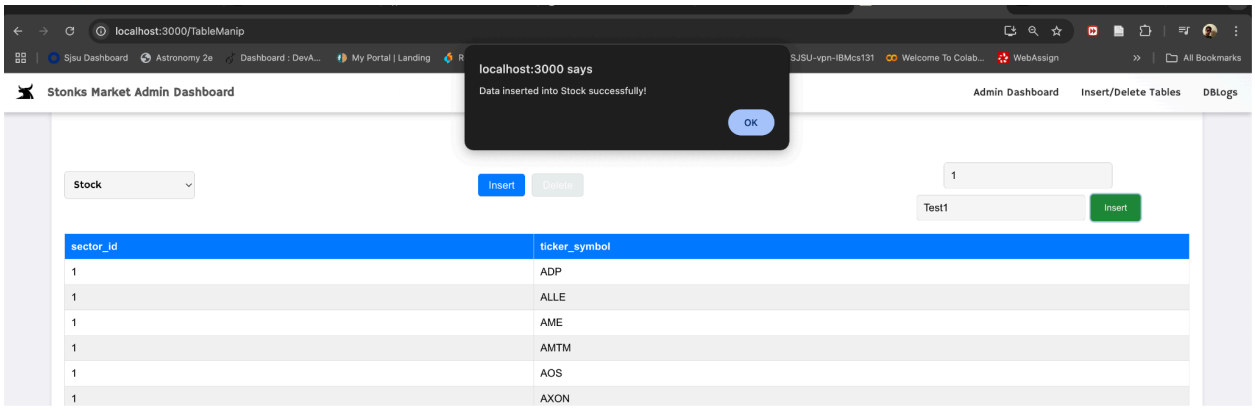


Figure 14: Insertion and Deletion Management System.

At this layer the structure of our table is mapped and after selecting the desired table the number and description of the entry boxes changes accordingly. During insertion insertion the name of the table and the inserted data gets packaged as jason and sent to the backend for insertion. If the row was inserted successfully then the front end receives a response that indicates successful insertion or otherwise. Manually insertion for the values of the table as shown in Figure 15 below.

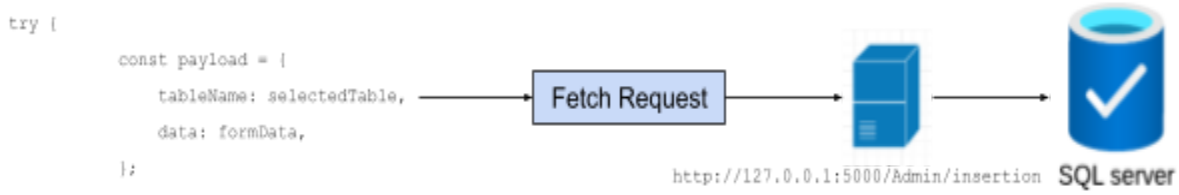


Figure 15: Methodology for Manual Insertion to SQL Tables.

Similarly for deletion the table_name and data related to selected row and its values gets selected and sends a request to the backend with post method and after deletion happens successfully, it receives a signal from backend displaying a message indicating deletion was successful. Additionally the table contents similarly gets fetched and updated from the back end layer.

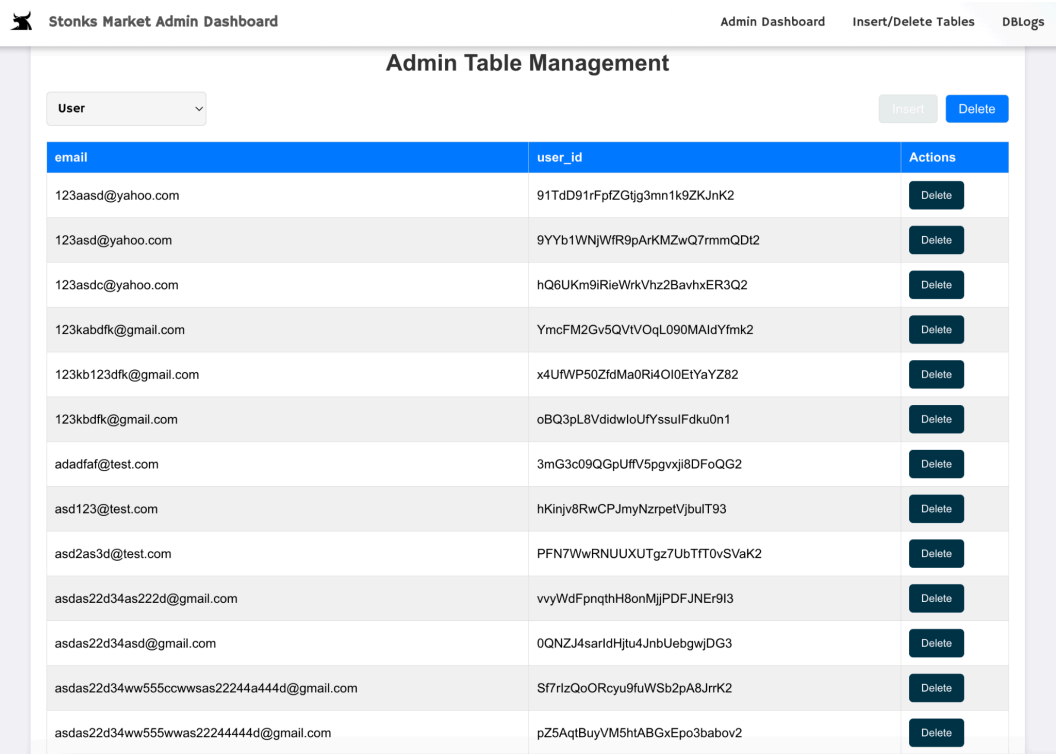


Figure 16: Admin Table Management; Deletion Tab.

Admin Database Log monitoring page:

This component displays all insertion, deletion and update logs into the sql tables. These logs are fetched from the back end quarrying them from the SQL server. Since the size of the logs was so long, it was decided to display only the most 100 recent logs.

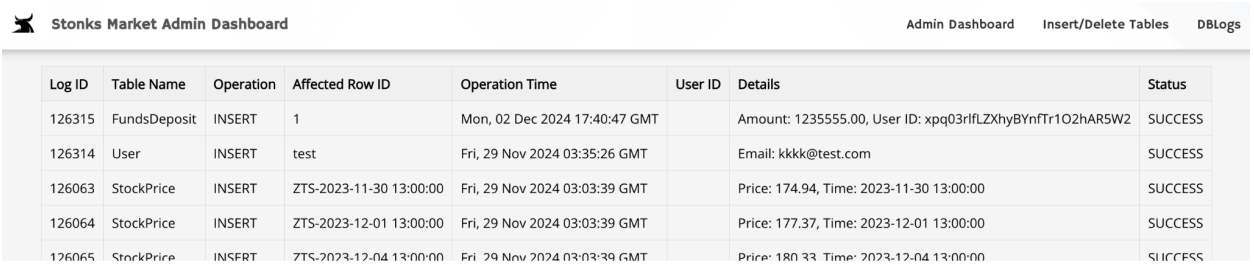


Figure 17: SQL Tables Insertion / Deletion / Update Logs.

Backend: Flask to define routers that handle user requests like login, stock trading, and news fetching. The `database.py` functions serve as the foundation for all database interactions. It provides streamlined functions to insert or update stock and price data, check for existing entries, and retrieve information. Beyond database updates, it enables exporting historical price data for individual stocks or detailed JSON files containing stock symbols, names (sourced from Wikipedia), sectors, and their latest prices. These files are saved in structured directories for easy

access and integration with external systems. Inside the routers, database functions are used to execute query requests. Since both mySQL and Postgres were used during the test stage of development, the DataBase object is optimized to communicate with both vendors' software.

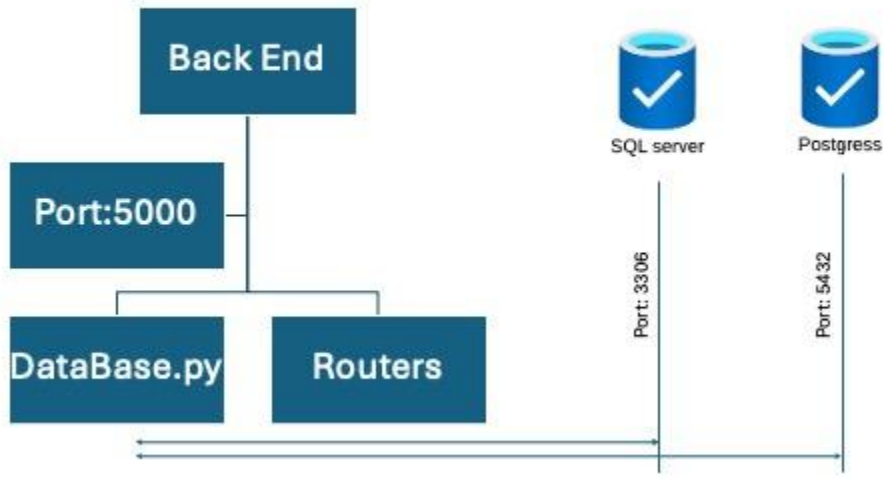


Figure 18: Backend SQL Deployment Structure.

Routes

1. **User components related routers:**
    - **Portfolio Router:** Opens endpoint '/<user_id>' for frontend portfolio component. It receives a [‘Get’] request containing users id, and it uses DB’s get_user_portfolio() function to fetch the content of the portfolio table where the user id matches the desired ID.

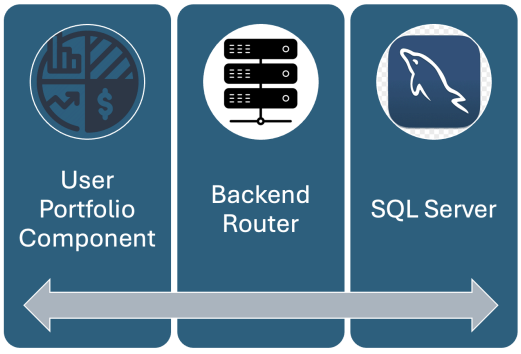


Figure 19: Portfolio Router.

- **Stock Manipulation Router:** Opens end point for stock purchase order.
- **Stock Price Router:** End point for main fetch and quick fetch.
- **Transaction History:**
- **User-Balance**

2. **Admin components related routers:**
   - **Admin Performance Router:** Opens an endpoint '/api/performance' to receive requests from SQL performance monitor and uses database function .Admin_performance() to fetch related information from SQL server.

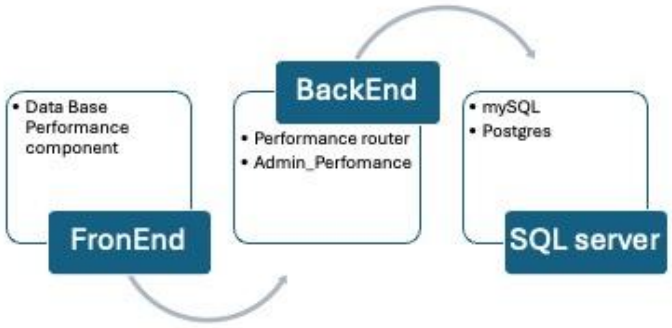


Figure 20: Router Workflow for Admin performance component.

- **Admin Insertion Router:** This router opens insertion endpoint '/insertion' for frontend's Admin Table-Insertion component. It receives the name of the desired table and the data to be inserted. Then it uses admin_insertion() db object's function to make the insertion.
- **Admin Fetch/Deletion Routers:** This router opens two endpoints for the front end admin table manipulation component. The '/fetch_table' endpoint fetches the data needed to be displayed on the desired table and '/delete_row' is designed to delete the designated table row from the database.

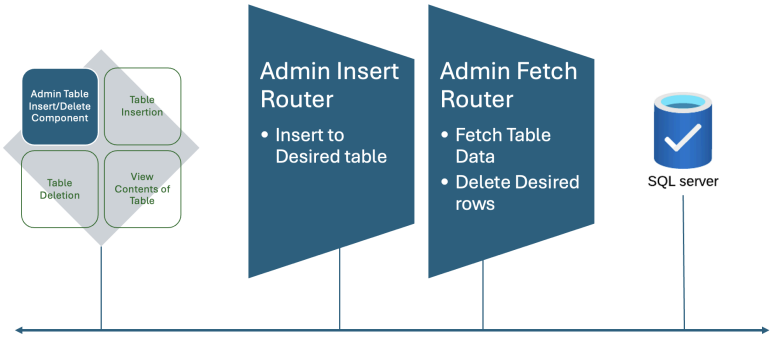


Figure 21: Administrative Routing.

- **Admin User Insertion Router:** Opens a user only insertion endpoint '/insertUser' for both sign up components, and also for syncing all the users information with the FB authentication data. Since the authentication is done through Firebase there is also a script that syncs the user ID and other info with the sql user and user-data tables. This router uses DataBase function .add_use() to communicate with the sql server and make the insertion.

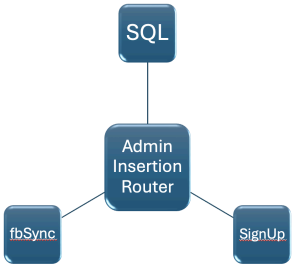


Figure 22: Firebase and Sign Up User Insertion Router

Daily Stock Data Fetching, Insertion, and Retrieval:

This system is designed to efficiently manage and retrieve stock market data, combining two scripts—`main_fetch` and `quick_fetch`—with functions in `database.py` for seamless database interactions. Together, these components ensure historical and current stock data for S&P 500 companies is stored, updated, and retrievable for further analysis.

The DataBase object's main_fetch script is the backbone for historical data collection. It retrieves the list of S&P 500 companies and their sectors from Wikipedia, inserting this information into the `Stock` and `Sector` tables. For each company, it fetches historical closing prices starting from a configurable start date (defaulting to 2010) up to the current day. It primarily uses Yahoo Finance as the data source but incorporates a fallback mechanism to Finnhub for resilience. This script also localizes all timestamps to Pacific Time and stores the data in the `StockPrice` table. By systematically iterating through all companies and ensuring data completeness, `main_fetch` provides a comprehensive database of historical stock prices.

The `quick_fetch` script is optimized for speed and focuses solely on fetching the latest available closing prices. It checks the database to see if the latest market day's data for a given stock is already recorded. If the data exists, it skips fetching; otherwise, it retrieves the price from Yahoo Finance, with Finnhub as a fallback. This script updates or inserts the price into the database, ensuring only new data is added. Designed for daily execution, `quick_fetch` efficiently keeps the database current without redundant operations.

Overall, this system is designed to maintain an accurate and up-to-date database of S&P 500 stock information. The `main_fetch` script ensures historical data completeness,

`quick_fetch` focuses on timely updates, and `database.py` supports seamless data management and export capabilities. Together, they form a cohesive pipeline for stock market data collection and retrieval.

- Database: MySQL stores normalized data, maintaining relationships among entities.
- Firebase: It is a cloud-base platform offering various services for building, deploying and scaling small and mobile apps. Used for authentication and real-time support ticket management. FB's authentication is an easy to use SDK, and ready to use UI libraries for user authentication. The Real-Time DB is a cloud-hosted NoSQL database that allows users to store and sync data in real time. The dbSDK json file and fbConfiguration.js are the files where all the information about its endpoints and its authentication keys are stored.

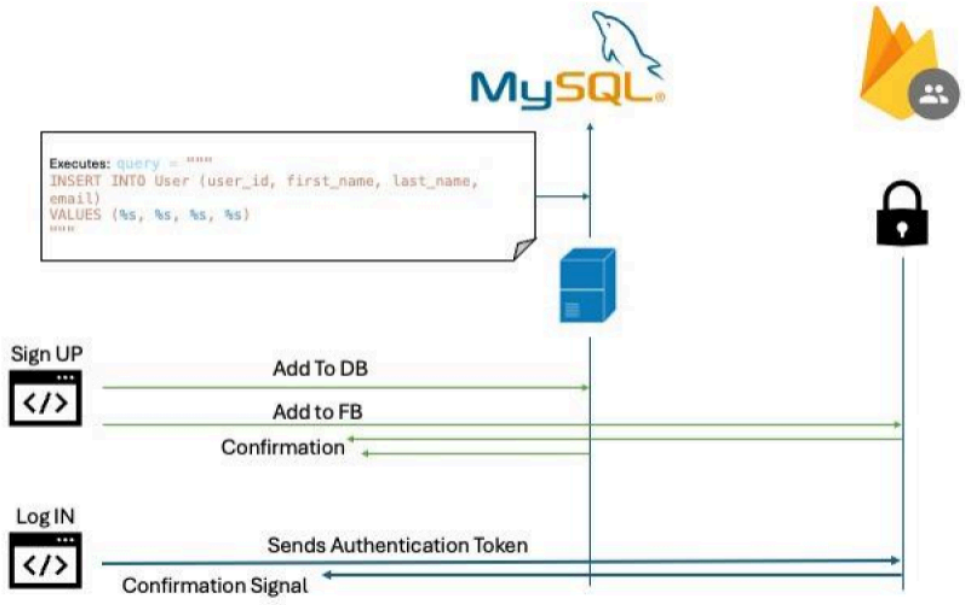


Figure 23: Implementation of Firebase Authentication.

Demonstration of Example System Run

https://youtu.be/xWGt5QM9U0g?si=T0GV_vL6rvoQz1AX

Conclusions, Lessons Learned, and Possible Improvements

Conclusions:

The platform successfully bridges the gap between theory and practice, providing a risk-free environment to learn and simulate trading. It is equipped with robust features, a responsive UI, and a reliable backend.

Lessons Learned:

1. Database normalization ensures system efficiency but increases design complexity.
2. Mock APIs simplify development but lack real-time dynamism.
3. Effective teamwork is critical to address challenges like API integration and data management.

Future Improvements:

1. Live Data Integration: Connect to real-time S&P 500 APIs for up-to-date trading experiences.
2. Advanced Analytics: Introduce performance metrics and risk analysis tools.
3. Gamification: Add leaderboards and rewards to boost user engagement.
4. Enhanced Security: Implement two-factor authentication for better account safety.

This concludes the Stonks Market application report, providing a comprehensive overview of its development, features, and future potential.