StockQuest

Where Your Stock Portfolio Meets Innovation

Github Repo Link : https://github.com/Shaneel-Reddy/Stock-Quest
Website Video Demo Link :
https://drive.google.com/file/d/1_hOu3Gn75tITtK8kYX848IP3WWJOF5SF/view
Website Deployed Link : http://stockquest.s3-website.ap-south-1.amazonaws.com/
(In case the link is inaccessible,kindly refer to the video demonstration or the attached screenshots for further details. Thank you for your understanding.)

Linkedin : https://www.linkedin.com/in/shaneel-reddy-0036b1263/
Resume :
https://drive.google.com/file/d/19vKjX_LgEtofg_tFKuyOZ2AJ9vBEc76f/view?usp=sharing
Name : L Shaneel Reddy
Email : shaneelreddy320@gmail.com

Overview

StockQuest is an advanced stock market analyzing platform designed to assist users in managing their investment portfolios, tracking asset performance, and staying updated with real-time stock data. The platform provides a seamless experience for both novice and experienced investors to monitor and manage their portfolios, analyze stocks, and make informed investment decisions. StockQuest uses a robust backend and frontend technology stack to ensure smooth functionality, secure user authentication, and real-time data fetching from third-party APIs.

Key Features

1. User Authentication & Authorization:
    - Secure login and signup functionality.
    - JWT-based authentication for secure access to protected routes.
    - Users can manage their profiles and view personalized data.
2. Portfolio Management:
    - Users can create and manage their portfolios with a real-time overview of their assets.

    - The platform tracks portfolio performance and asset values, updating them dynamically with the latest stock prices.
3. Real-Time Stock Data:
    - Integration with third-party APIs (Finnhub, TwelveData) to provide real-time stock prices and historical data.
    - Real-time stock charts and market trends.
4. Stock Asset Management:
    - Add, update, and delete stock assets in the portfolio.
    - Track each asset's value, purchase price, quantity, and gain/loss percentage.
5. Interactive Visualizations:
    - Visual representations of stock data and portfolio performance using interactive charts (e.g., StockChart component).
6. Educational Resources:
    - Provide a learning section with articles, tutorials, and videos to help users understand stock trading and investment concepts.
7. Chat Integration:
    - A chat feature that allows users to get stock-related queries answered in real time.

Tech Stack Used

1. Backend Framework:
    - Spring Boot (Java) - The core framework for developing the backend services, including RESTful APIs, security, and business logic.
2. Database:
    - MySQL - Relational database for storing user data, portfolios, and assets, with JPA for ORM (Object-Relational Mapping).
3. Security (Spring Security):
    - JWT (JSON Web Tokens) - For user authentication and authorization, ensuring secure access to protected resources.
4. API Communication:
    - RESTful APIs - For communication between the frontend and backend.
    - Finn Hub API and TwelveData API - For fetching real-time stock data and market information.
    - News Api - For fetching real time stock related news.
5. Frontend Framework:
    - ReactJS - For building the user interface and handling the frontend logic, including state management, routing, and real-time data fetching.
6. State Management:

    - React State - For managing application-level state, such as user authentication and real-time stock data.
7. Testing:
    - JUnit and Mockito - For unit testing backend services, ensuring the robustness and correctness of business logic.
8. Version Control:
    - Git - For version control, ensuring collaboration and effective code management.
9. Deployment & Hosting:
    - AWS - For hosting both the backend and frontend, ensuring scalability and performance.
10. Real-Time Data Integration:
    - Gemini API - For stock-related chat queries and real-time stock data.

Backend for StockQuest

Overview

StockQuest is a stock market analyzing platform that allows users to manage their investment portfolios, track the performance of assets, and access real-time stock data. The backend is built with **Spring Boot** and **Java**, with **JWT-based authentication** for security. It integrates with third-party APIs like **Finnhub,TwelveData Api** to fetch real-time stock prices.

Tech Stack

- **Backend Framework**: Spring Boot
- **Database**: MySQL (JPA repository for ORM)
- **Security**: JWT (JSON Web Token) authentication
- **API Communication**: RESTful APIs and integration with **Finnhub API** for stock data
- **Testing**: JUnit, Mockito for unit tests
- **Version Control**: Git

Core Components

1. Entity Models

Entities represent the data structure in the database.

Register

- **Fields**:
    - id: Long (Primary key)
    - email: String (User email, unique)
    - password: String (User password)
- **Purpose**: Represents a registered user in the system. This entity is used for authentication and linking with portfolios and assets.

Portfolio

- **Fields**:
    - id: Long (Primary key)
    - totalValue: Double (Total portfolio value)
    - user: Register (User associated with the portfolio)
    - assets: List<Asset> (List of assets in the portfolio)
- **Purpose**: Represents an investment portfolio that contains a collection of assets. It tracks the total value of the portfolio based on asset values.

Asset

- **Fields**:
    - id: Long (Primary key)
    - name: String (Name of the asset, e.g., 'Apple Inc.')
    - symbol: String (Ticker symbol, e.g., 'AAPL')
    - quantity: Double (Number of units of the asset)
    - purchasePrice: Double (Price per unit at the time of purchase)
    - currentPrice: Double (Current market price fetched from Finnhub)
    - value: Double (Calculated value of the asset based on current price)
    - user: Register (User owning the asset)
    - portfolio: Portfolio (Portfolio containing the asset)
- **Purpose**: Represents an individual asset in a user's portfolio. Tracks the name, symbol, quantity, purchase price, current price, and value.

2. Repositories

Repositories are used to interact with the database and perform CRUD operations.

AssetRepository

- **Methods**:
    - save(Asset asset): Saves a new or updated asset to the database.

- saveAll(List<Asset> assets): Saves a list of assets.
- findById(Long id): Retrieves an asset by its ID.
- findAll(): Retrieves all assets in the database.
- deleteById(Long id): Deletes an asset by its ID.

PortfolioRepository

- **Methods**:
  - findByUser_Id(Long userId): Retrieves a portfolio by user ID.
  - save(Portfolio portfolio): Saves a portfolio to the database.

RegisterRepo

- **Methods**:
  - findByEmail(String email): Retrieves a user by email.

3. Services

Services contain the business logic of the application and interact with repositories to perform operations.

AssetService

- **Methods**:
  - createAsset(Register user, Asset asset, Portfolio portfolio): Creates a new asset and adds it to the portfolio. Fetches the current price from Finnhub API.
  - createDefaultAssetsForUser(Register user, Portfolio portfolio): Creates default assets (e.g., stocks) (i.e) 5 Stocks for a new user's portfolio.
  - updateAsset(Long id, Asset updatedAsset): Updates the details of an existing asset and recalculates its value.
  - deleteAsset(Long id): Deletes an asset from the database.
  - getAssetById(Long id): Retrieves an asset by its ID.
  - fetchCurrentPrice(String symbol): Fetches the current stock price from Finnhub API based on the asset symbol.
  - updateStockPrices(): Updates the prices of all assets in the system by fetching the current market price from Finnhub.

PortfolioService

- **Methods**:
  - getPortfolioByUserId(Long userId): Retrieves the portfolio associated with a given user. If no portfolio exists, a new one is created.

    - recalculatePortfolioValue(Portfolio portfolio): Recalculates the total value of the portfolio based on the current prices of the assets.
    - createPortfolio(Long userId): Creates a new portfolio for a user.

RegisterServiceImp

- **Methods**:
    - loadUserByUsername(String email): Loads user details by email for authentication. Throws an exception if the user does not exist.

4. Security

Security is implemented using **JWT (JSON Web Token)** to authenticate and authorize users.

JWT Authentication Flow:

1. **User Login**: The user provides their credentials (email and password) via the login endpoint.
2. **Token Generation**: If the credentials are valid, a JWT token is generated and returned to the client.
3. **Authorization**: For subsequent requests, the client includes the JWT token in the Authorization header. The backend validates the token for each protected endpoint.

- **JWT Constants**: JwtConstant contains keys for token generation, such as the secret key and expiration time.
- **JwtProvider**: This class is responsible for generating and validating JWT tokens.
- **JwtTokenValidator**: Ensures the JWT token is valid for accessing protected endpoints.

Endpoints

User Authentication & Authorization

- **POST /api/auth/login**
    - **Request**: Email and password.
    - **Response**: JWT token.
    - **Purpose**: Authenticates the user and returns a JWT token for further requests.

Portfolio Endpoints

- **GET /api/portfolio/{userId}**
    - **Request**: User ID.
    - **Response**: Portfolio details of the user.

    - **Purpose**: Fetches the portfolio of the user, creates a new portfolio if it doesn't exist.
- **POST /api/portfolio**
    - **Request**: User ID.
    - **Response**: Newly created portfolio.
    - **Purpose**: Creates a new portfolio for a user if they do not already have one.

Asset Endpoints

- **POST /api/assets**
    - **Request**: Asset details.
    - **Response**: Created asset details.
    - **Purpose**: Adds a new asset to the portfolio, including fetching its current price from Finnhub.
- **GET /api/assets/{assetId}**
    - **Request**: Asset ID.
    - **Response**: Details of the asset.
    - **Purpose**: Fetches asset details by ID.
- **PUT /api/assets/{assetId}**
    - **Request**: Updated asset details.
    - **Response**: Updated asset details.
    - **Purpose**: Updates an existing asset's details.
- **DELETE /api/assets/{assetId}**
    - **Request**: Asset ID.
    - **Response**: Status message.
    - **Purpose**: Deletes an asset from the user's portfolio.

Exception Handling

Custom Exception Classes:

- **UnauthorizedException**: Thrown for unauthorized actions (e.g., invalid JWT).
- **ResourceNotFoundException**: Thrown when a resource (asset, portfolio) is not found.
- **ConflictException**: Used for conflicts like duplicate resources.

Global Exception Handler: The GlobalExceptionHandler class manages exceptions globally using **@ControllerAdvice**. It handles:

- **UnauthorizedException**: Returns FORBIDDEN (403).
- **ResourceNotFoundException**: Returns NOT_FOUND (404).
- **General Exception**: Returns INTERNAL_SERVER_ERROR (500).
- **ConflictException**: Returns CONFLICT (409).

This approach ensures consistent error handling and appropriate HTTP status codes for a better user experience.

Testing

The backend uses **JUnit** for unit testing. Below are some of the key test cases:

AssetService Tests

- **testCreateAsset()**: Verifies the creation of a new asset and ensures the current price is fetched correctly from Finnhub.
- **testUpdateAsset()**: Tests updating an asset and verifies that the current price is fetched and applied.
- **testDeleteAsset()**: Verifies the deletion of an asset.
- **testUpdateStockPrices()**: Tests updating stock prices for all assets and recalculating their values.

PortfolioService Tests

- **testGetPortfolioByUserId()**: Tests fetching a portfolio for an existing user.
- **testRecalculatePortfolioValue()**: Verifies the recalculation of the portfolio's total value based on asset values.

RegisterService Tests

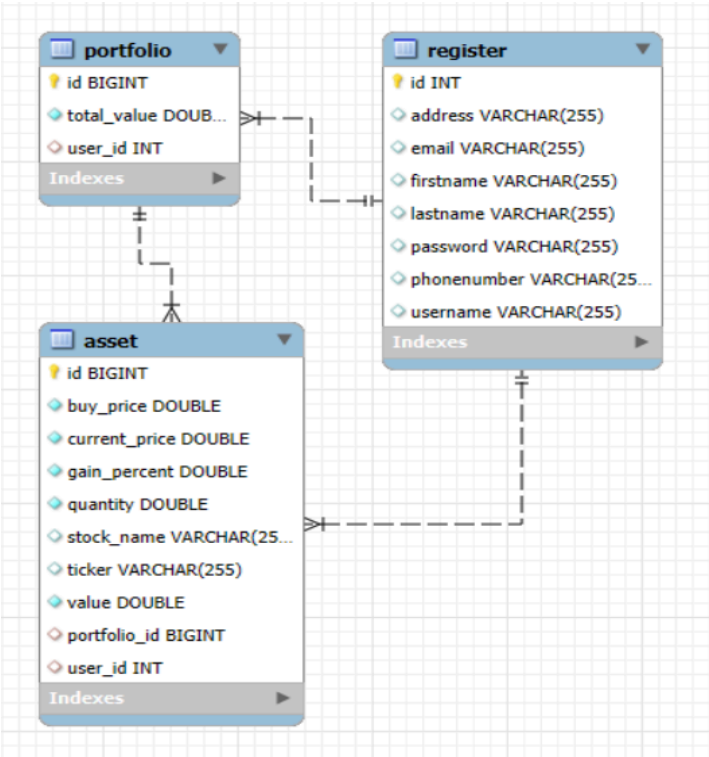
- **testLoadUserByUsername()**: Verifies user loading by email for authentication.

Configuration

- **Finnhub API URL**: Configured in application.properties under finnhub.api.url.
- **Finnhub API Key**: Configured in application.properties under finnhub.api.key.

Database Schema Design for StockQuest

ER Diagram



1. User Table (Register)

This table stores user information, including personal details and their relationship with portfolios.

Table Name: register

Column	Data Type	Constraints	Description
id	BIGINT	PRIMARY KEY, AUTO_INCREMENT	Unique identifier for each user.
username	VARCHAR(255)	NOT NULL, UNIQUE	The username of the user.
password	VARCHAR(255)	NOT NULL	The password of the user (encrypted).
email	VARCHAR(255)	NOT NULL, UNIQUE	User's email address.
firstname	VARCHAR(255)	NOT NULL	User's first name.

lastname	VARCHAR(255)	NOT NULL	User's last name.
address	VARCHAR(255)	NOT NULL	User's address.
phonenumber	VARCHAR(20)	NOT NULL	User's phone number.

Relationships:

- One-to-One relationship with portfolio (linked via the user_id in portfolio).

---

2. Portfolio Table

This table stores the user's portfolio details, including the stocks they own and the total value of the portfolio.

Table Name: portfolio

Column	Data Type	Constraints	Description
id	BIGINT	PRIMARY KEY, AUTO_INCREMENT	Unique identifier for each portfolio.
total_value	DOUBLE	NOT NULL	The total value of the portfolio.
user_id	BIGINT	NOT NULL, UNIQUE	Foreign key referencing the id in register.

Relationships:

- One-to-One relationship with register (linked via the user_id column).
- One-to-Many relationship with asset (linked via the portfolio_id in asset).

---

3. Asset Table

This table stores individual stock asset details for each user, including stock name, ticker, quantity, purchase price, current price, gain percentage, and total value.

Table Name: asset

Column	Data Type	Constraints	Description
id	BIGINT	PRIMARY KEY, AUTO_INCREMENT	Unique identifier for each asset.
stock_name	VARCHAR(255)	NOT NULL	Name of the stock.
ticker	VARCHAR(255)	NOT NULL	Ticker symbol of the stock.
quantity	DOUBLE	NOT NULL	Quantity of the stock owned by the user.
buy_price	DOUBLE	NOT NULL	The purchase price of the stock.
current_price	DOUBLE	NOT NULL	The current price of the stock.
gain_percent	DOUBLE	NOT NULL	Percentage change in the stock price.
value	DOUBLE	NOT NULL	Total value of the asset (quantity * current price).
portfolio_id	BIGINT	FOREIGN KEY (portfolio_id)	Foreign key referencing the id in portfolio.
user_id	BIGINT	FOREIGN KEY (user_id)	Foreign key referencing the id in register.

Relationships:

- Many-to-One relationship with portfolio (linked via the portfolio_id).
- Many-to-One relationship with register (linked via the user_id).

Database Relationship Diagram

The relationships between entities in the database are as follows:

- **Register** has a One-to-One relationship with **Portfolio** (A user has one portfolio).
- **Portfolio** has a One-to-Many relationship with **Asset** (A portfolio contains multiple assets).

- **Asset** has Many-to-One relationships with both **Portfolio** and **Register** (Each asset belongs to one portfolio and one user).

Foreign Key Constraints

- portfolio.user_id references register.id.
- asset.portfolio_id references portfolio.id.
- asset.user_id references register.id.

These foreign key constraints ensure referential integrity across related entities.

Frontend For StockQuest

Overview

The frontend of the StockQuest project is built using ReactJS and integrates with the backend to handle user authentication, portfolio management, real-time stock data, and interactive visualizations. The frontend ensures secure and seamless access to the application through JWT authentication.

Key Features

- **User Authentication**: Allows users to log in, sign up, and manage their session using JWT tokens.
- **Portfolio Management**: Enables users to manage their stock portfolios and track portfolio performance with real-time stock data.
- **Real-Time Stock Data**: Displays stock quotes and trends, integrated with the backend.
- **Private Routes**: Ensures that only authenticated users can access protected pages like the dashboard, portfolio, and learn pages.

Frontend Components

1. **Login/Signup Pages**
    - Users log in using email or phone number, and sign up with personal details.
    - The login page sends a POST request to /auth/signin, and the signup page sends a POST request to /auth/signup.
2. **Dashboard**
    - Displays an overview of the user's stock portfolio, including a list of stocks and portfolio performance.

    - Utilizes the StockChart and Card components to visualize stock-related data.
3. **Stock Market Page**
    - Displays real-time stock data for top trending stocks like AAPL, MSFT, etc and also displays live stock news fetched from NewsApi.
    - Uses the /stocks/{symbol} API to fetch stock details, including high, low, and previous close prices.
4. **Portfolio Page**
    - Allows users to view and manage their portfolio.
    - Displays total portfolio value,stock information, such as quantity,current price,buy price,gain percent,value and offers options to add, update, or remove stocks.
5. **Learn Page**
    - Provides educational resources related to stock trading and investment.
    - Displays various learning materials, articles, or videos, to guide users on stock market concepts.
6. **PrivateRoute Component**
    - A wrapper for the routes that require authentication.
    - Redirects users to the login page if they try to access a protected route (e.g., Dashboard, Portfolio, Market,Learn page) without being authenticated.
    - Ensures that only logged-in users can access these pages.
7. **StockChart Component**
    - Displays real-time stock data in the form of interactive charts (e.g., line charts).
    - Fetches stock data from the backend and visualizes it based on user selection.
8. **Chat Component**
    - Helps users resolve stock-related queries through real-time messaging.
    - Integrated with Gemini API to provide accurate stock-related information.

State Management

- **React State** is used to manage the application's state, including:
    - User authentication status.
    - Stock data.
- **JWT Tokens** are stored in localStorage and included in the Authorization header for API requests to access protected resources.

Error Handling

- The frontend includes robust error handling for API calls, such as:
    - Incorrect credentials during login.
    - Failed API requests due to network errors.
    - Validation errors when adding or updating stocks.
    - User-friendly error messages are displayed to guide users.

Frontend Authentication Flow

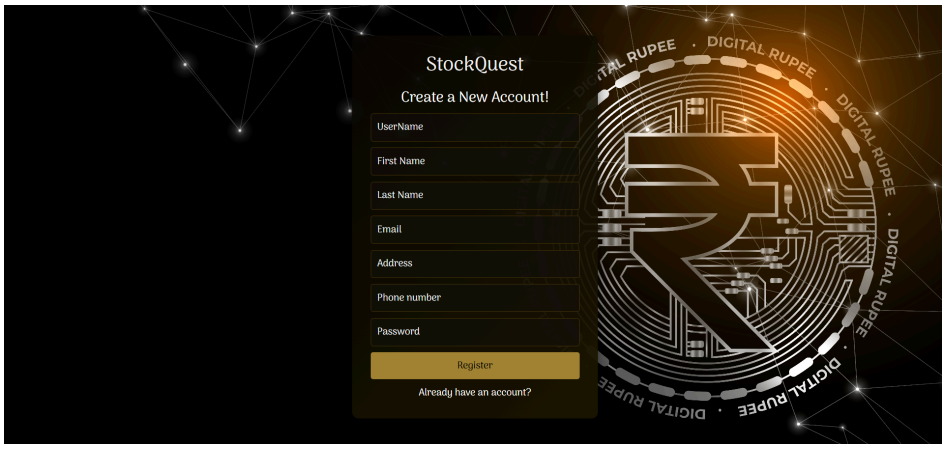
1. **Login/Signup**: The user can log in or sign up.
2. Upon successful login, the JWT token is stored in localStorage.
3. The token is sent with the Authorization header in all subsequent API requests.
4. Protected pages, such as the **Dashboard**, **Portfolio**,**Market,Chat** and **Learn** pages, are accessible only to authenticated users.
5. If the JWT token is invalid or expired, the user is redirected to the login page.

API Security

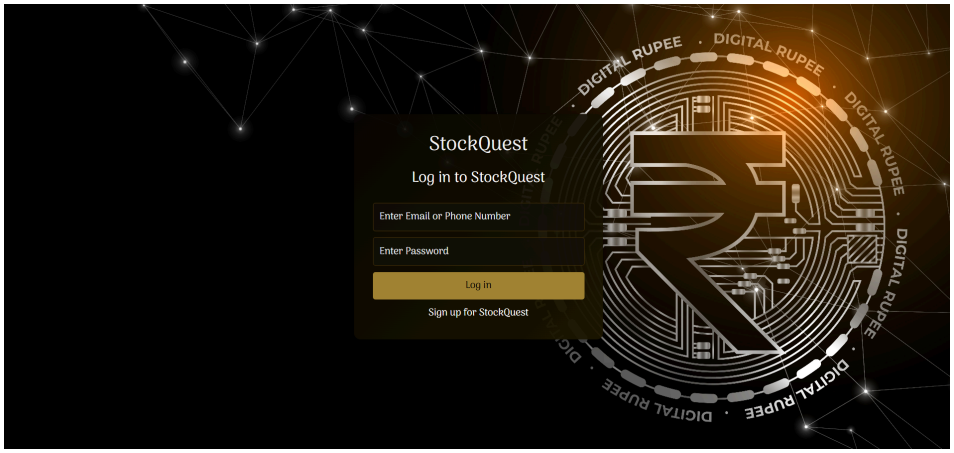
- The frontend ensures that sensitive data, such as passwords and tokens, are not exposed.
- All API requests use **HTTPS** for secure communication with the backend.
- The JWT token is used to protect access to routes and data.

Screenshots

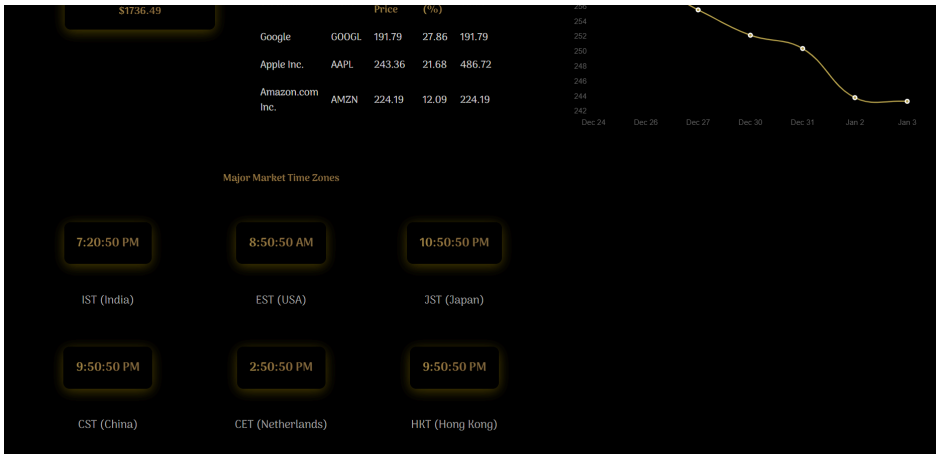
Signup Page



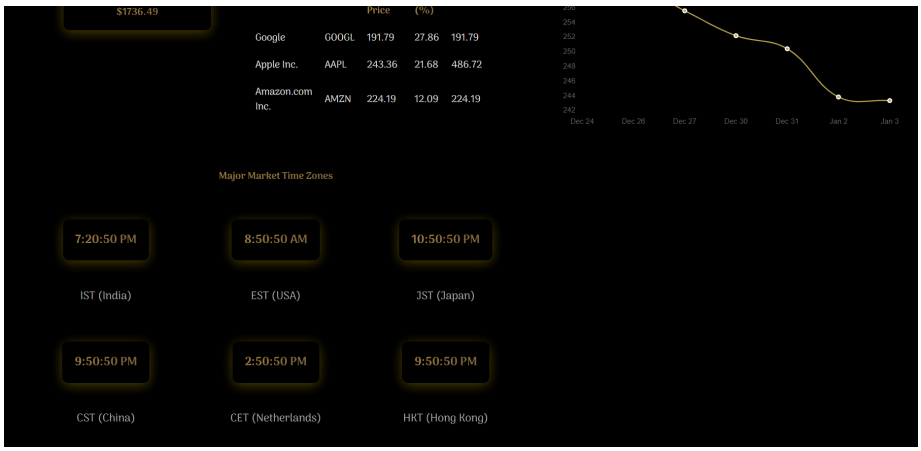
Login Page



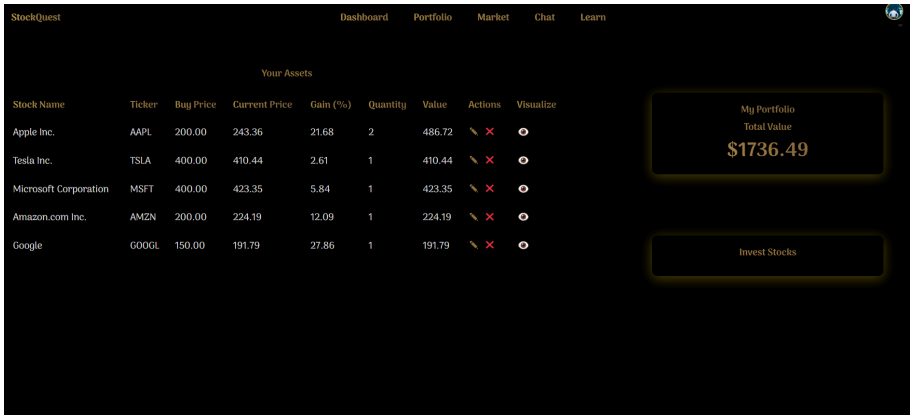
Dashboard Page (1)



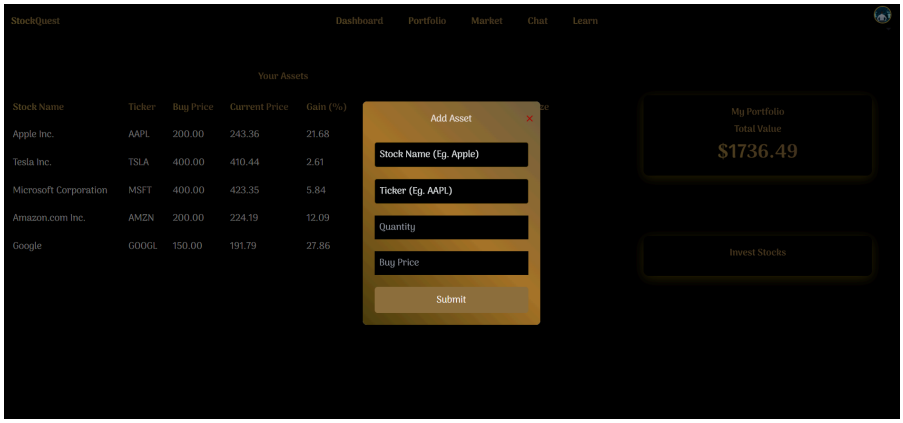
Dashboard Page (2)



Portfolio Page

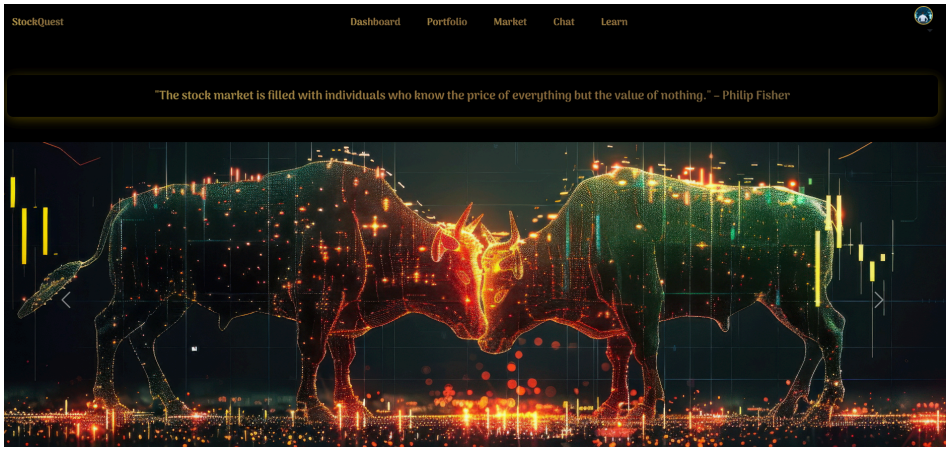


Portfolio Page [Add Asset]

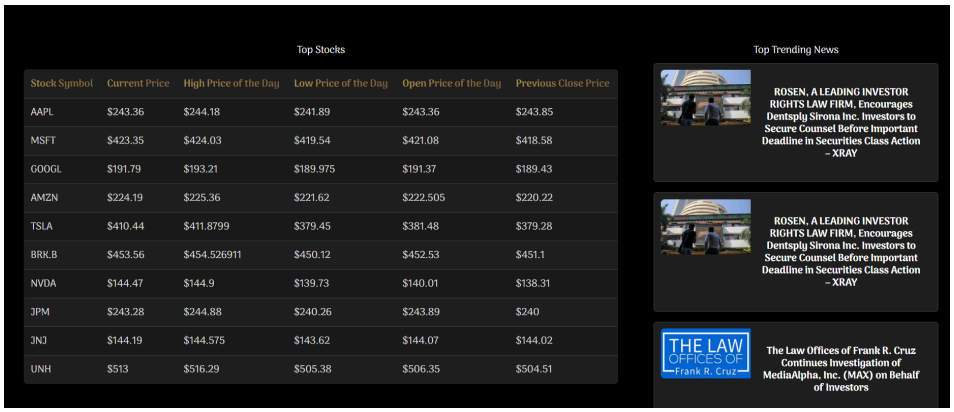


Portfolio Page [Visualise Asset]

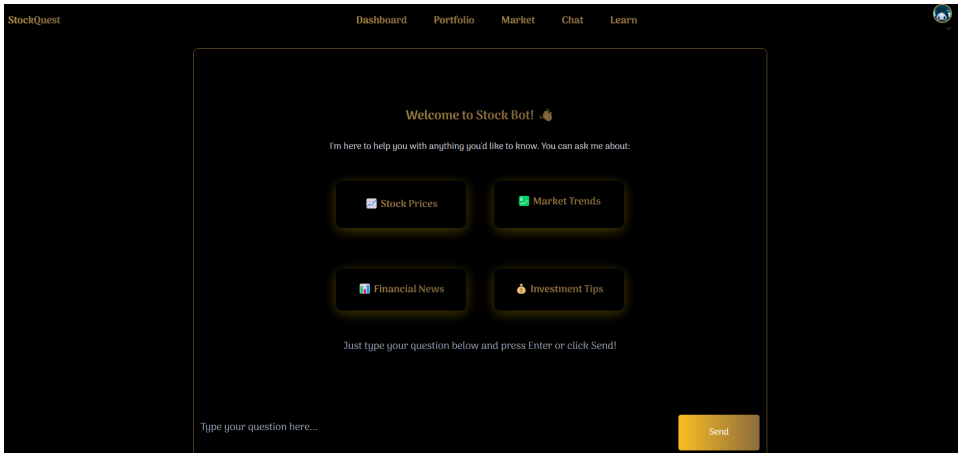
Market Page (1)



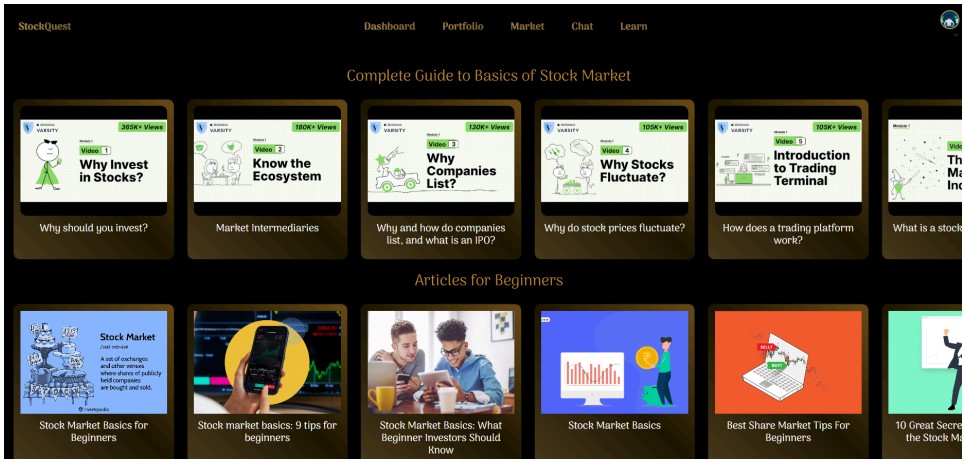
Market Page (2)



Chat Page



Learn Page



Future Scope

1. **AI/ML-based Predictions:** Integrate machine learning for stock predictions and personalized recommendations.
2. **Mobile App:** Develop iOS/Android apps for on-the-go portfolio management.
3. **Real-Time Alerts:** Implement notifications for stock price changes and portfolio performance.
