DBMS Assignment 4 Report

Team Details :
Name : Abhishek Aditya BS
SRN : PES1UG19CS019

Name : Adithya M S
SRN : PES1UG19CS027

Name : Abhiram Puranik
SRN : PES1UG19CS018

Dependencies needed for frontend :

1. **Material UI (core + icons)** - MUI provides a robust, customizable, and accessible library of foundational and advanced components, enabling you to build your own design system and develop React applications faster. MUI components work in isolation. They are self-supporting, and will only inject the styles they need to display. They don't rely on any global style-sheets such as normalize.css.
2. **Axios** - Axios is a promise-based HTTP Client for node.js and the browser. It is isomorphic (it can run in the browser and nodejs with the same codebase). On the server-side it uses the native node.js http module, while on the client (browser) it uses XMLHttpRequests.
3. **React** - React is a JavaScript library for building user interfaces. React makes it painless to create interactive UIs. Design simple views for each state in your application, and React will efficiently update and render just the right components when your data changes. Build encapsulated components that manage their own state, then compose them to make complex UIs.

4. **React-Dom** - The react-dom package provides DOM-specific methods that can be used at the top level of your app and as an escape hatch to get outside of the React model if you need to.
5. **React-Router-Dom** - The react-router-dom package contains bindings for using React Router in web applications.
6. **React-google-charts** - React Google Charts offers a declarative API to make rendering charts fun and easy.

Dependencies needed for Backend :

1. **Concurrently** - Run multiple commands concurrently. Can be used to run both backend and frontend servers together without the use of multiple terminals.
2. **Cors** - CORS Anywhere is a NodeJS proxy which adds CORS headers to the proxied request. The url to proxy is literally taken from the path, validated and proxied. The protocol part of the proxied URI is optional, and defaults to "http". If port 443 is specified, the protocol defaults to "https". This package does not put any restrictions on the http methods or headers, except for cookies. Requesting user credentials is disallowed. The app can be configured to require a header for proxying a request, for example to avoid a direct visit from the browser.
3. **DotEnv** - Dotenv is a zero-dependency module that loads environment variables from a .env file into process.env. Storing configuration in the environment separate from code is based on The Twelve-Factor App methodology.
4. **Express** - Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. Express provides a thin layer of fundamental web application features, without obscuring Node.js features.

5. **Nodemon** - nodemon is a tool that helps develop node.js based applications by automatically restarting the node application when file changes in the directory are detected. Nodemon does not require any additional changes to your code or method of development. nodemon is a replacement wrapper for node. To use nodemon, replace the word node on the command line when executing your script.
6. **Pg** - Non-blocking PostgreSQL client for Node.js. Pure JavaScript and optional native libpq bindings which share the same API. Supported PostgreSQL features
    > Parameterized queries
    > Named statements with query plan caching
    >Async notifications with LISTEN/NOTIFY
    >Bulk import & export with COPY TO/COPY FROM

Migration to NO-SQL Database :

The four main types of NoSQL databases are :

> Document databases
> Key-value stores
> Column-oriented databases
> Graph databases

Document Databases

A document database stores data in JSON, BSON , or XML documents (not Word documents or Google docs, of course). In a document database, documents can be nested. Particular elements can be indexed for faster querying. Documents can be stored and retrieved in a form that is much closer to the data objects used in applications, which means less translation is required to use the data in an application. SQL data must often be assembled and disassembled when moving back and forth between applications and storage.

Document databases are popular with developers because they have the flexibility to rework their document structures as needed to suit their application, shaping their data structures as their application requirements change over time. This flexibility speeds development because in effect data becomes like code and is under the control of developers. In SQL databases, intervention by database administrators may be required to change the structure of a database. The most widely adopted document databases are usually implemented with a scale-out architecture, providing a clear path to scalability of both data volumes and traffic. The most widely adopted document databases are usually implemented with a scale-out architecture, providing a clear path to scalability of both data volumes and traffic.

Use cases include e-commerce platforms, trading platforms, and mobile app development across industries.

Key-Value Stores

The simplest type of NoSQL database is a key-value store . Every data element in the database is stored as a key value pair consisting of an attribute name (or "key") and a value. In a sense, a key-value store is like a relational database with only two columns: the key or attribute name (such as state) and the value (such as Alaska).

Use cases include shopping carts, user preferences, and user profiles.

Column-Oriented Databases

While a relational database stores data in rows and reads data row by row, a column store is organized as a set of columns. This means that when you want to run analytics on a small number of columns, you can read those columns directly without consuming memory with the unwanted data. Columns are often of the same type and benefit from more efficient compression, making reads even faster.

Columnar databases can quickly aggregate the value of a given column (adding up the total sales for the year, for example). Use cases include analytics. Unfortunately there is no free lunch, which means that while columnar databases are great for analytics, the way in which they write data makes it very difficult for them to be strongly consistent as writes of all the columns require multiple write events on disk. Relational databases don't suffer from this problem as row data is written contiguously to disk.

Graph Databases

A graph database focuses on the relationship between data elements. Each element is stored as a node (such as a person in a social media graph). The connections between elements are called links or relationships. In a graph database, connections are first-class elements of the database, stored directly. In relational databases, links are implied, using data to express the relationships. A graph database is optimized to capture and search the connections between data elements, overcoming the overhead associated with JOINing multiple tables in SQL. Very few real-world business systems can survive solely on graph queries. As a result graph databases are usually run alongside other more traditional databases. Use cases include fraud detection, social networks, and knowledge graphs.

Our Choice of NO-SQL Database :

Our choice of NO-SQL Database will be Document based Database, since all the details related to a user like his information, stock holdings, transactions made, associated broker, can be stored in a single JSON document and since each user is given a unique identifier it can be very easy to fetch all the details of the users from a single document instead of querying multiple tables in SQL and maintaining complex foreign key relationships between those tables.

MongoDB vs PostgreSQL: A Comparison in Brief

MongoDB is the leading document database. It is built on a distributed, scale-out architecture and has become a comprehensive cloud-based platform for managing and delivering data to applications. MongoDB handles transactional, operational, and analytical workloads at scale. If your concerns are time to market, developer productivity, supporting DevOps and agile methodologies, and building stuff that scales without operational gymnastics, MongoDB is the way to go.

PostgreSQL is a rock solid, open source, enterprise-grade SQL database that has been expanding its capabilities for 30 years. Everything you would ever want from a relational database is present in PostgreSQL, which relies on a scale-up architecture. If your concerns are compatibility, serving up thousands of queries from hundreds of tables, taking advantage of existing SQL skills, and pushing SQL to the limit, PostgreSQL will do an awesome job.

MongoDB vs PostgreSQL: Detailed Comparison

MongoDB	PostgreSQL
Schema-free	SQL-based but supports various NoSQL features
Document database	Relational database
Uses BSON	Uses SQL
Distributed architecture	Monolithic architecture
Potential for ACID compliance	ACID-compliant

Uses collections	Uses tables
Uses documents to obtain data	Uses rows to obtain data
Does not support foreign key constraints	Supports foreign key constraints
Uses the aggregation pipeline for running queries	Uses GROUP_BY
Redundant replica sets	2-safe replication
Uses indexes	Uses joins
Not Required	`CREATE TABLE users ( user_id VARCHAR(20) NOT NULL, age INTEGER NOT NULL, status VARCHAR(10));`
`db.users.insert({ user_id: "bcd001", age: 45, status: "A" })`	`INSERT INTO users(user_id, age, status) VALUES ('bcd001', 45,"A");`
`db.users.find()`	`SELECT * FROM users;`

```
db.users.update(
    { age: { $gt: 25 } },
    { $set: { status: "C" } },
    { multi: true }
)
```

```
UPDATE users
SET status = 'C'
WHERE age &gt; 25;
```

```
session.startTransaction();
db.orders.insert ({
  order_id: '1a2b3c',
  product: 'T-shirt',
  quantity: 7
})
db.stock.update (
  { product: { $eq: 'T-shirt', } },
  { $inc: { quantity: -7 } }
})
session.commitTransaction();
```

```
START TRANSACTION;
INSERT INTO orders
(order_id, product, quantity)
VALUES ('1a2b3c', 'T-shirt', '7');
UPDATE stock
SET quantity=quantity-7
WHERE product='T-shirt';
COMMIT;
```

Screenshots for the statements executed to start the web app:

1. Execution of DDL.sql File

```
C:\Program Files\PostgreSQL\13\bin>psql -U postgres -f c:\Users\adith\Documents\Stock-Exchange-Simulator-DBMS_Project\Stock
ExchangeSimulator\config\stockExchange_DDL.sql
Password for user postgres:
DROP DATABASE
CREATE DATABASE
You are now connected to database "stockexchange" as user "postgres".
CREATE TABLE
CREATE EXTENSION
SET
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
ALTER TABLE
ALTER TABLE
ALTER TABLE
ALTER TABLE
ALTER TABLE
ALTER TABLE
```

2. Execution of DML.sql file

```
C:\Program Files\PostgreSQL\13\bin>psql -U postgres -f c:\Users\adith\Documents\Stock-Exchange-Simulator-DBMS_Project\Stock
ExchangeSimulator\config\stockExchange_DML.sql
Password for user postgres:
You are now connected to database "stockexchange" as user "postgres".
INSERT 0 8
INSERT 0 8
INSERT 0 8
INSERT 0 8
INSERT 0 8
INSERT 0 24
INSERT 0 64
INSERT 0 64
INSERT 0 56
INSERT 0 64
INSERT 0 16
INSERT 0 1
INSERT 0 1
INSERT 0 8
psql:c:/Users/adith/Documents/Stock-Exchange-Simulator-DBMS_Project/StockExchangeSimulator/config/stockExchange_DML.sql:115
2: NOTICE:  Value: true
 sell
------
    2
(1 row)


psql:c:/Users/adith/Documents/Stock-Exchange-Simulator-DBMS_Project/StockExchangeSimulator/config/stockExchange_DML.sql:115
3: NOTICE:  Value: true
 sell
------
    2
(1 row)
```

3. Execution of DCL.sql File

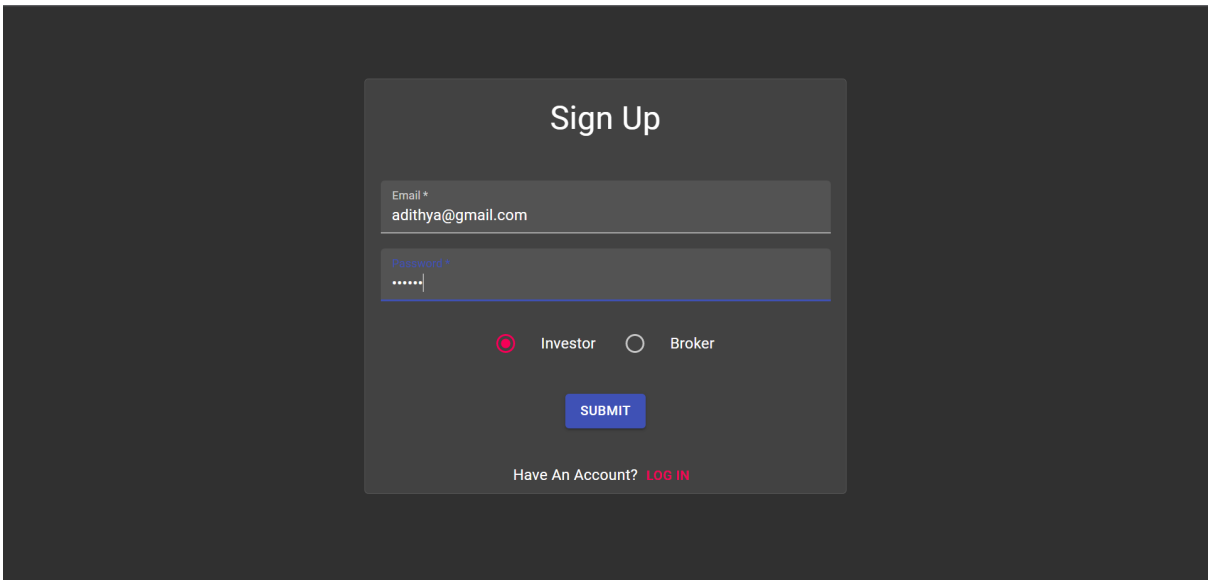
```
C:\Program Files\PostgreSQL\13\bin>psql -U postgres -f c:\Users\adith\Documents\Stock-Exchange-Simulator-DBMS_Project\Stock
ExchangeSimulator\config\stockExchange_DCL.sql
Password for user postgres:
You are now connected to database "stockexchange" as user "postgres".
DROP ROLE
DROP ROLE
DROP ROLE
DROP ROLE
CREATE ROLE
GRANT
GRANT
CREATE ROLE
GRANT
CREATE ROLE
GRANT
GRANT
GRANT
CREATE ROLE
GRANT
GRANT
```

4. Starting front end and back end server.

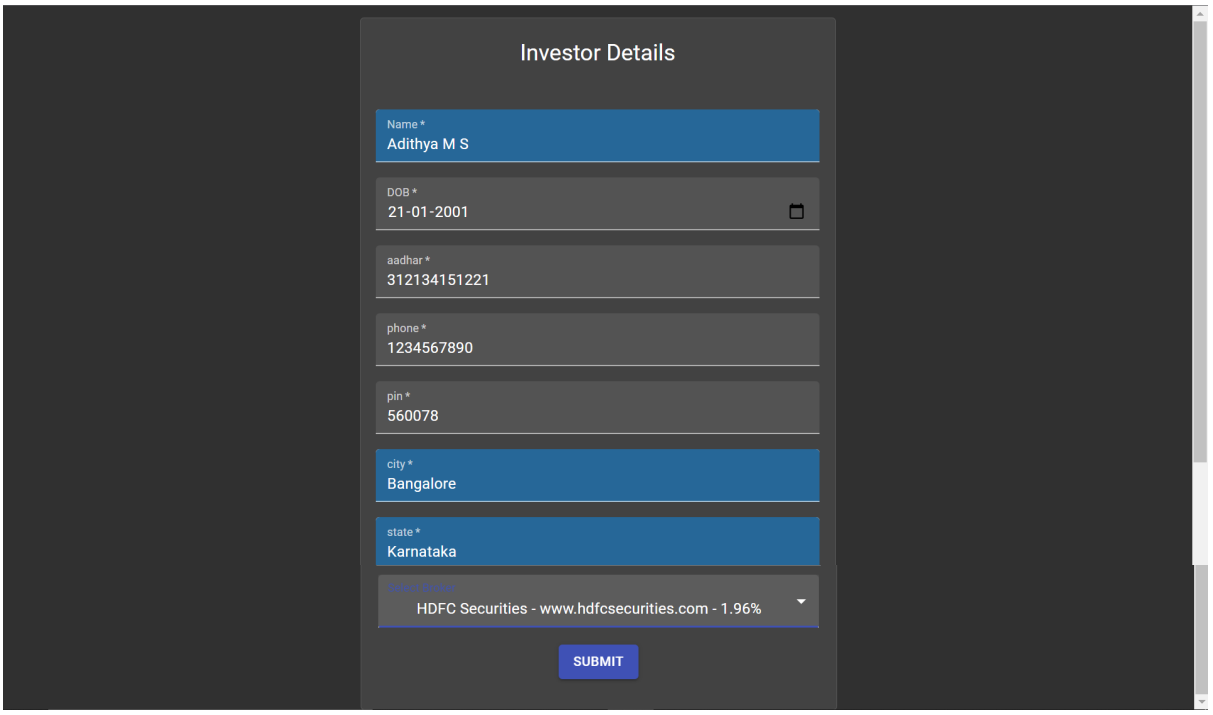
```
PS C:\Users\adith\Documents\Stock-Exchange-Simulator-DBMS_Project\StockExchangeSimulator> npm start

> StockExchangeSimulator@1.0.0 start C:\Users\adith\Documents\Stock-Exchange-Simulator-DBMS_Project\StockExchangeSimulator
> concurrently "npm run dev" "npm run client"

[1]
[1] > StockExchangeSimulator@1.0.0 client C:\Users\adith\Documents\Stock-Exchange-Simulator-DBMS_Project\StockExchangeSimul
ator
[1] > cd client && npm start
[1]
[0]
[0] > StockExchangeSimulator@1.0.0 dev C:\Users\adith\Documents\Stock-Exchange-Simulator-DBMS_Project\StockExchangeSimulato
r
[0] > nodemon server.js
[0]
[0] [nodemon] 2.0.15
[0] [nodemon] to restart at any time, enter `rs`
[0] [nodemon] watching path(s): *.*
[0] [nodemon] watching extensions: js,mjs,json
[0] [nodemon] starting `node server.js`
[0] Listening on port 8000
[1]
[1] > client@0.1.0 start C:\Users\adith\Documents\Stock-Exchange-Simulator-DBMS_Project\StockExchangeSimulator\client
[1] > react-scripts start
[1]
[0] Connected to database by Investor
[0] Connected to database through supervisor
[0] Connected to database by broker
[0] Connected to database by admin
```

Screenshots for the statements executed from the front end

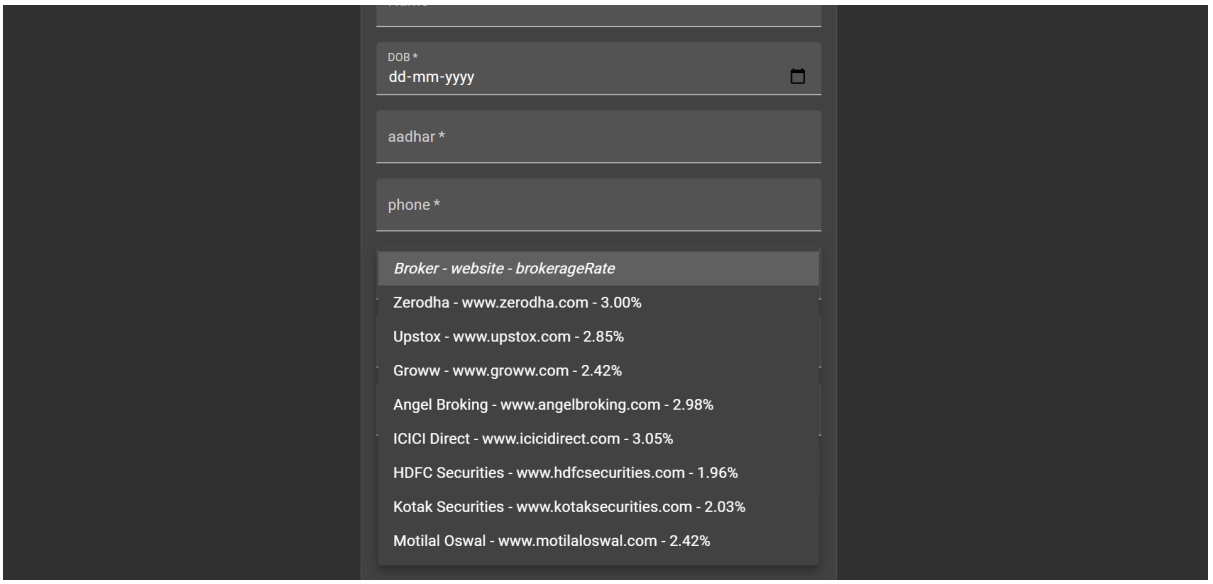
1. Sign Up page where the user selects to be an investor.



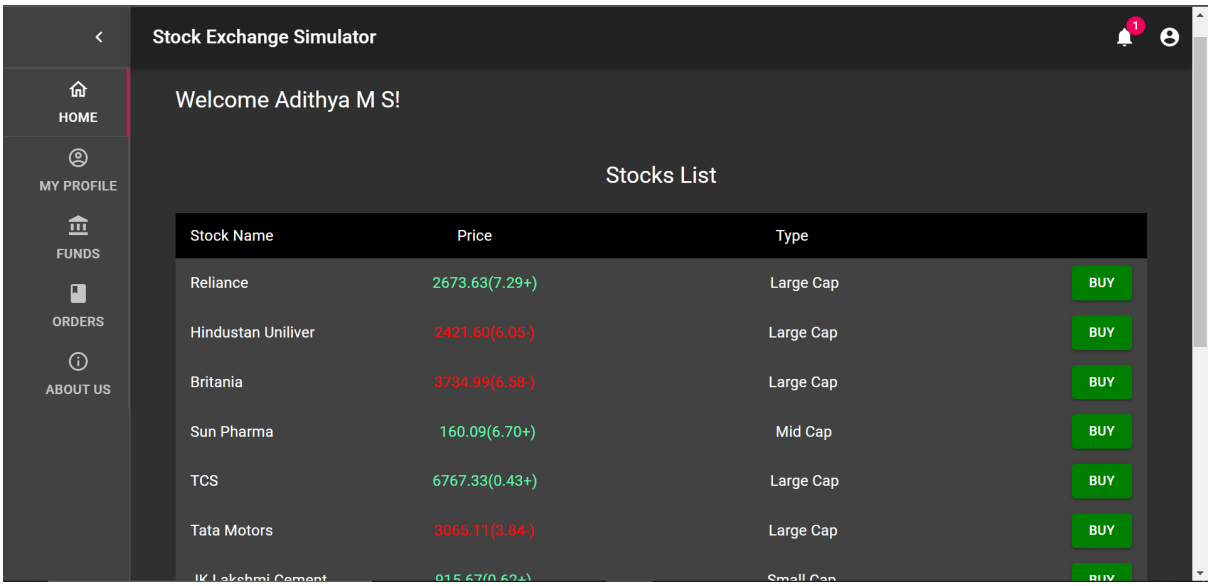
2. Filling up investor details.



3. Selecting broker.

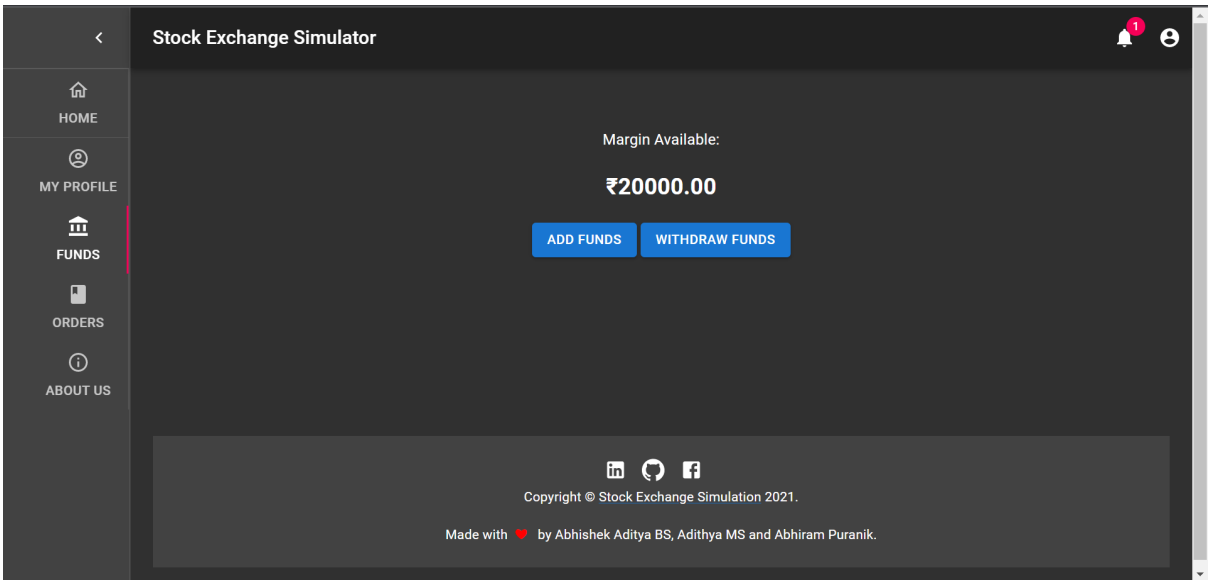


4. Investor home page displays all the stocks with their details and clicking the buy button makes a dialog pop up where the stock can be bought.

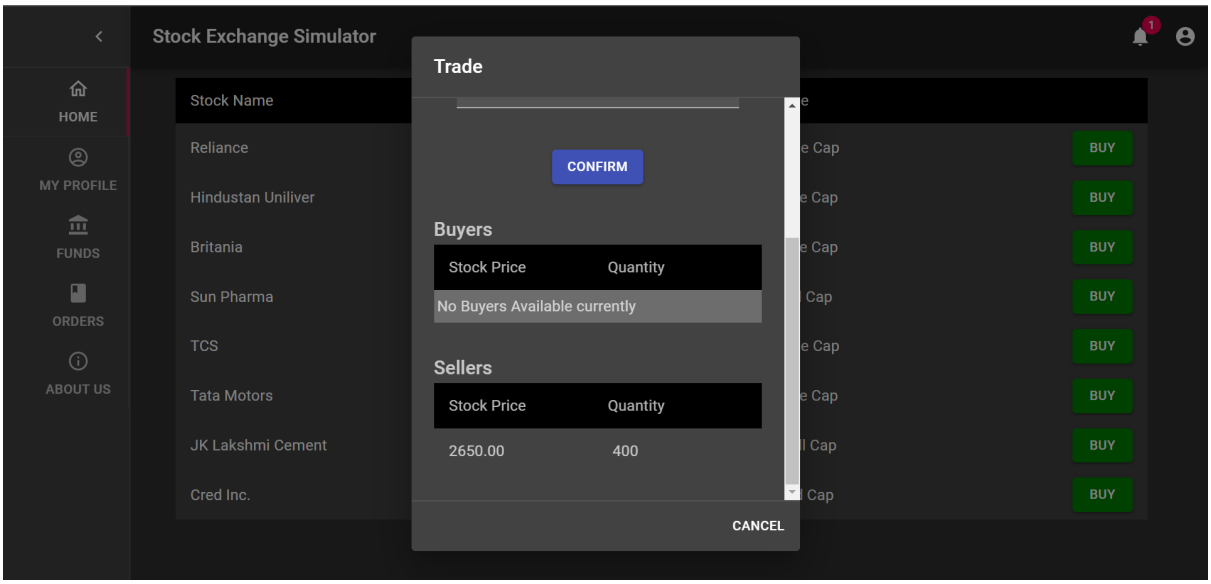


5. Profile page of investor showing Stock Market Trends. This page also shows the holdings of the user.

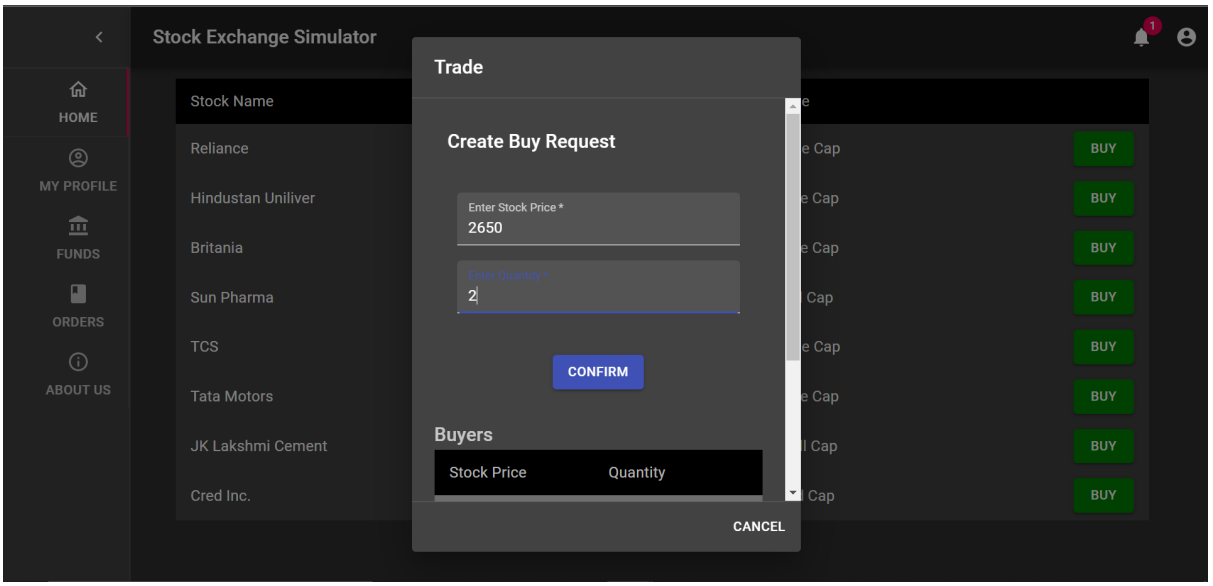
6. Funds page of investor showing margin available.



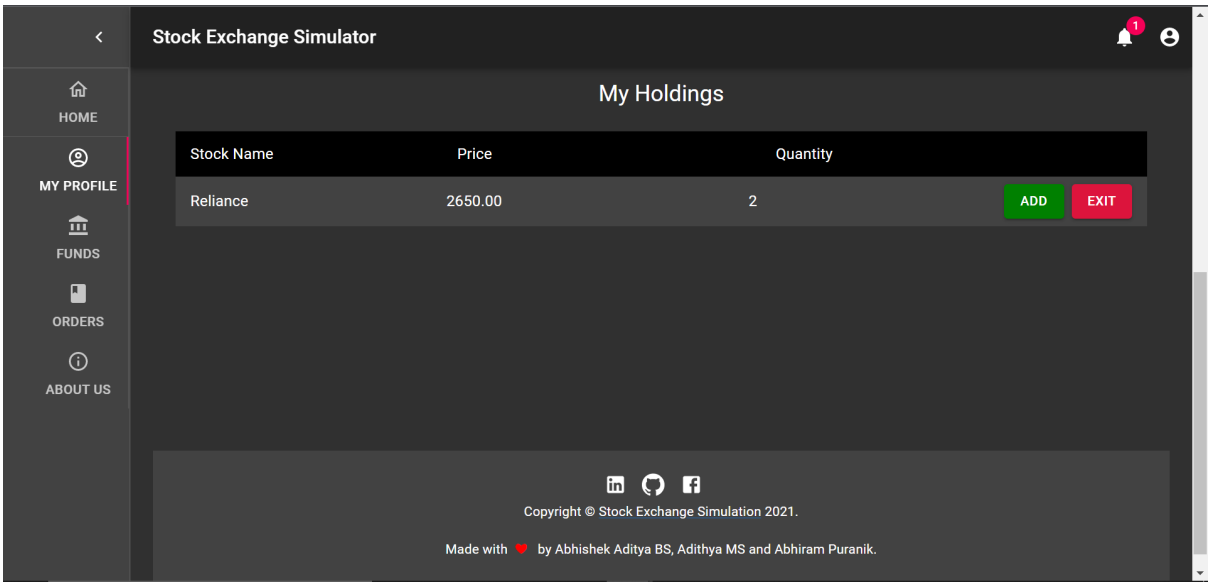
7. Clicking on the 'BUY' button in home displays the list of buyers and sellers for that particular stock.



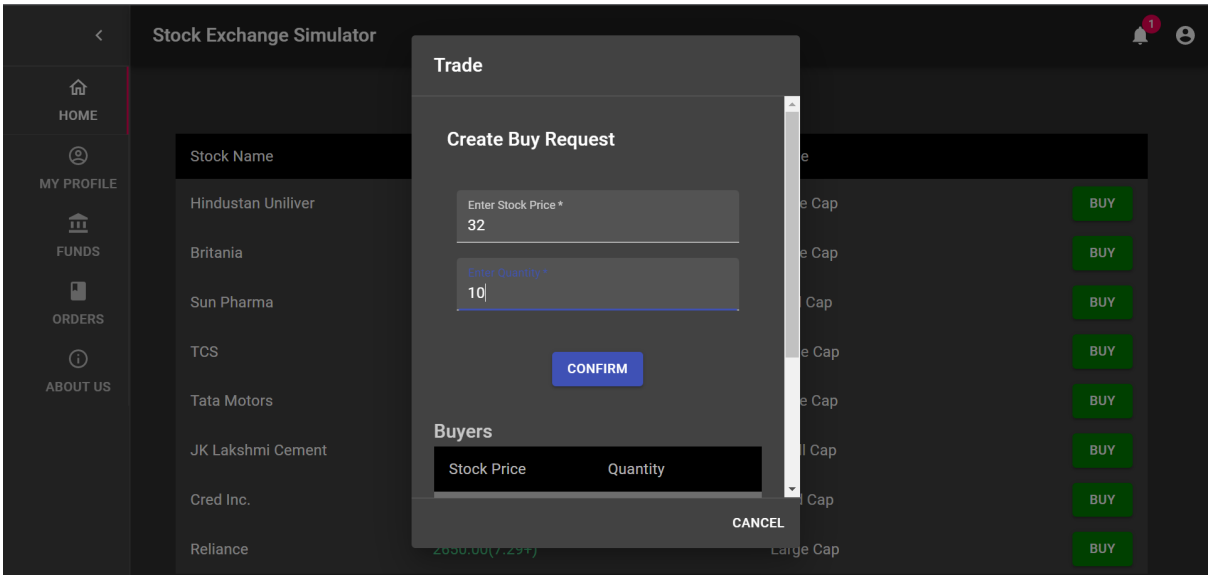
8. To buy stocks, an investor has to enter details regarding the price and quantity that he wishes to buy.



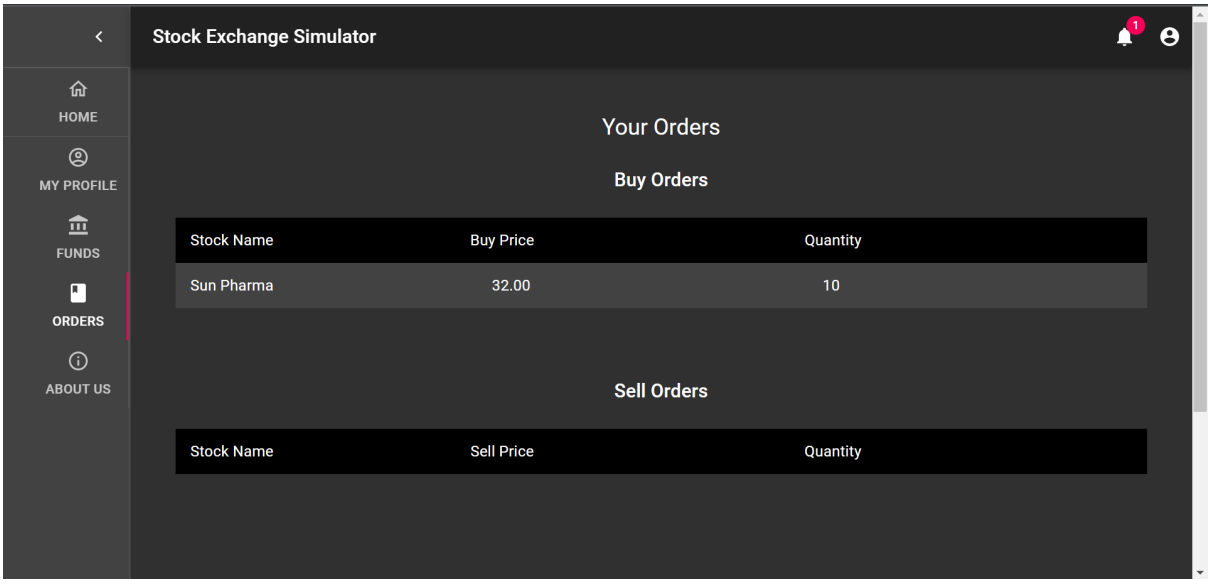
9. Investor's holdings will be displayed after he has bought it.



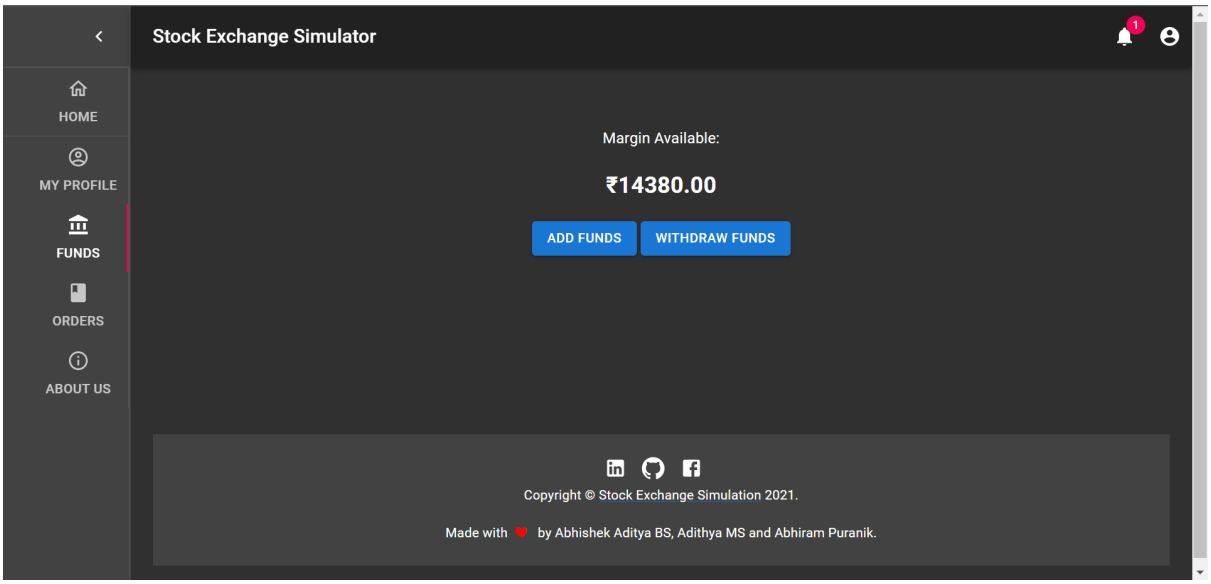
10. If the investor wishes to buy stock at a different price, then a buy request is created.



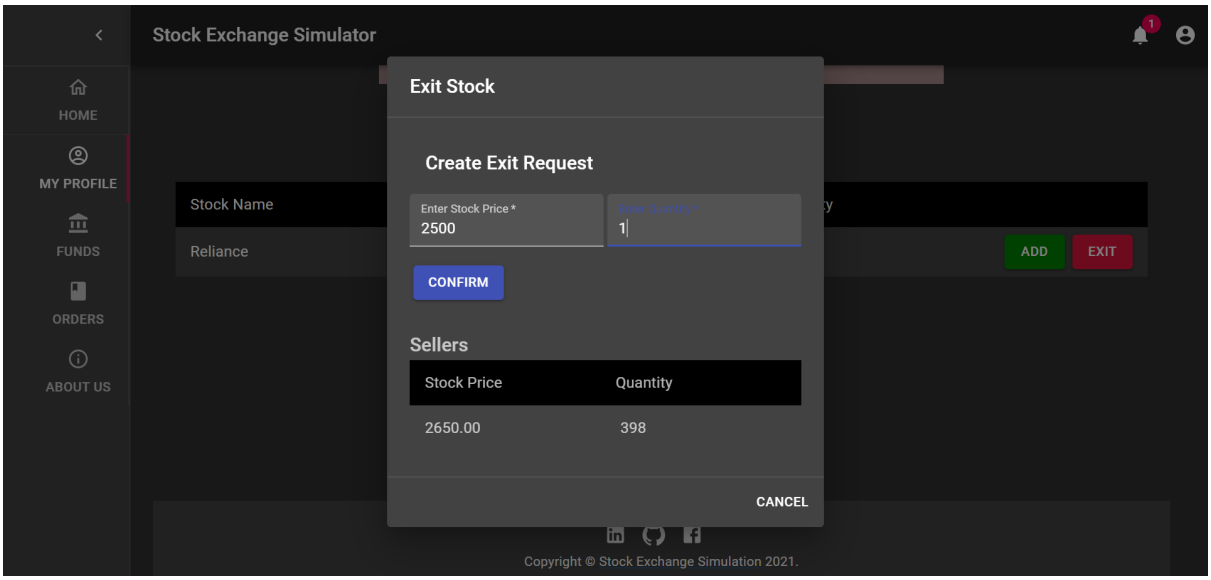
11. In the orders tab, the list of buy and sell requests are shown.



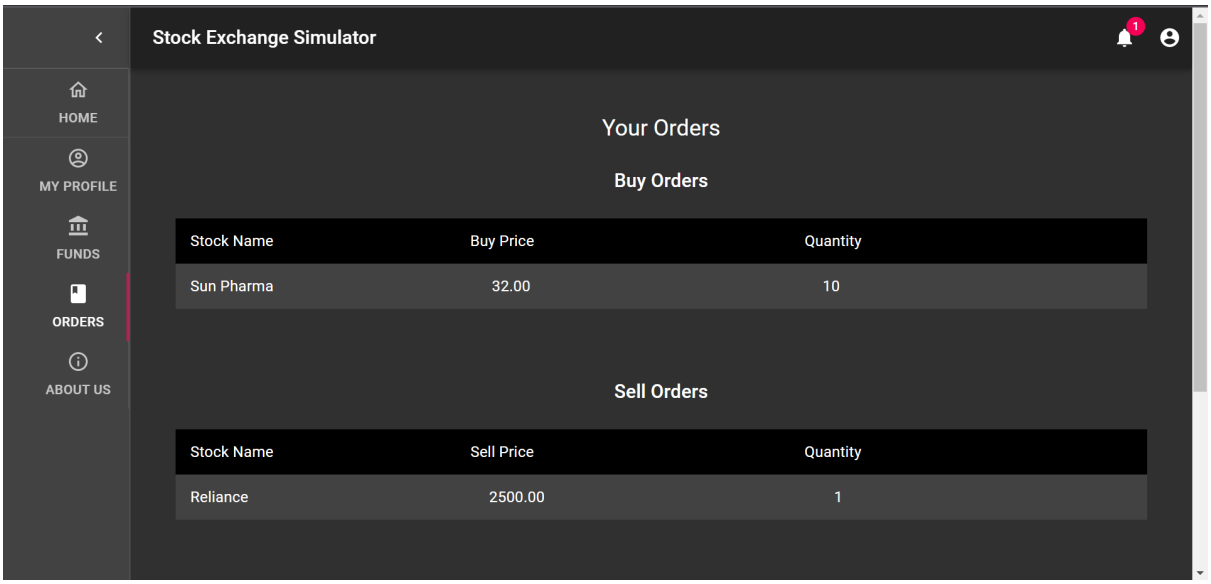
12. Margin Available is updated accordingly.



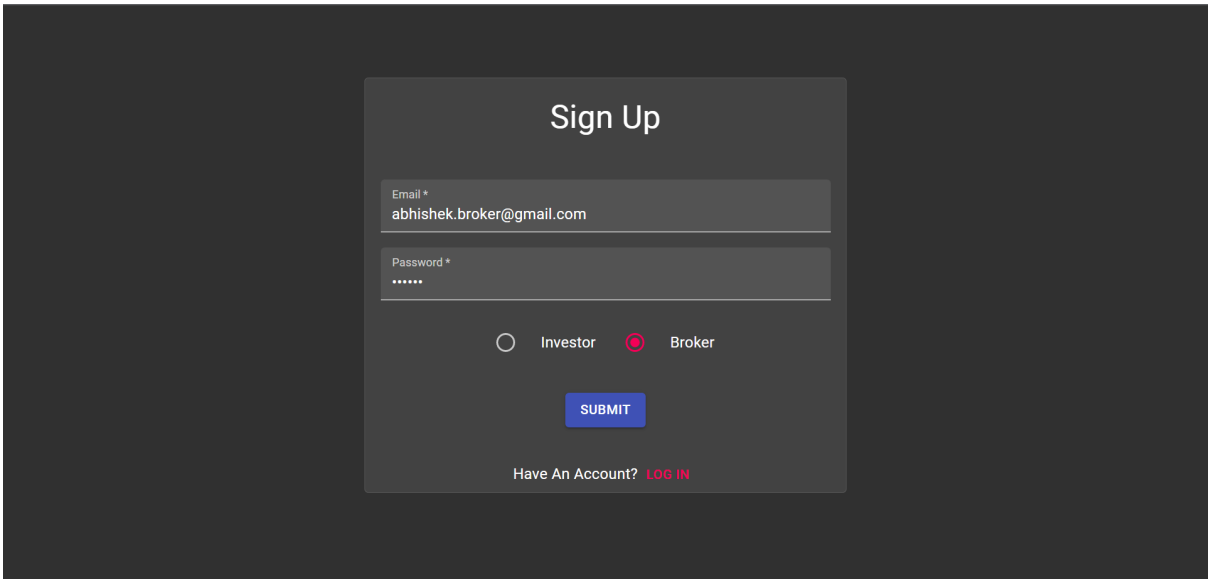
13. Investors can exit stock using the exit button and filling up the required details.



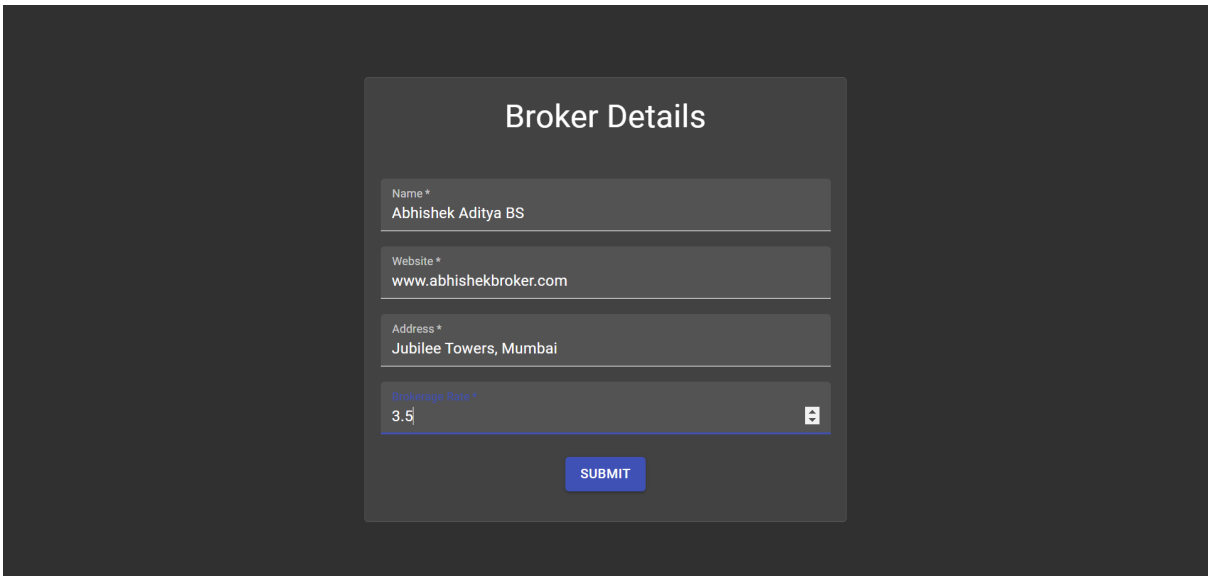
14. Sell request is created for the same.



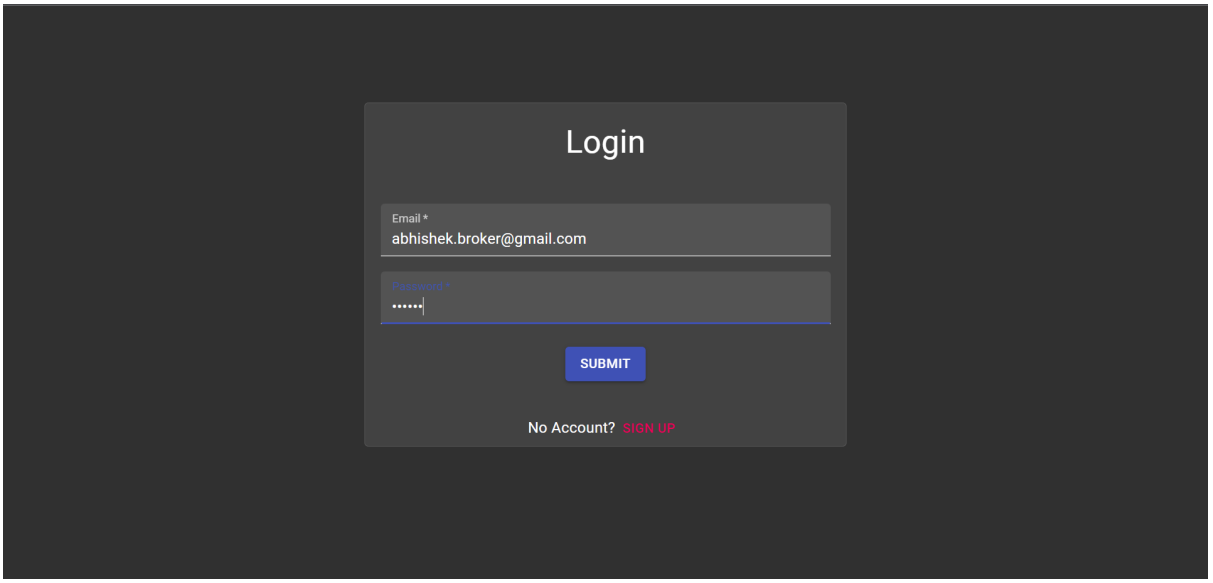
15. Now, the user signs up as a broker.



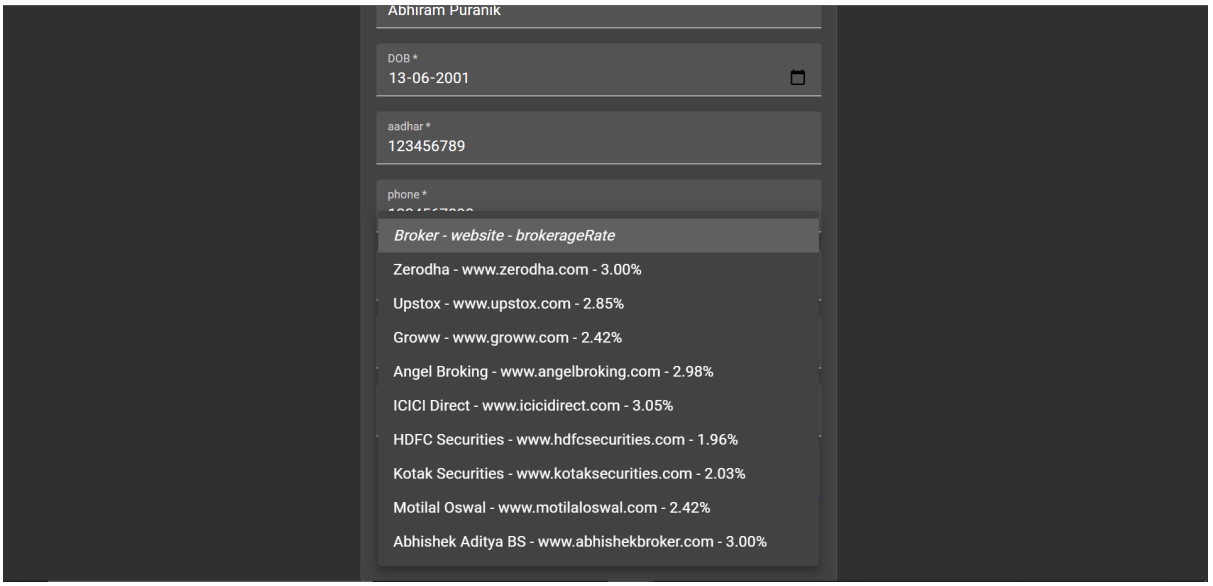
16. Broker details form.



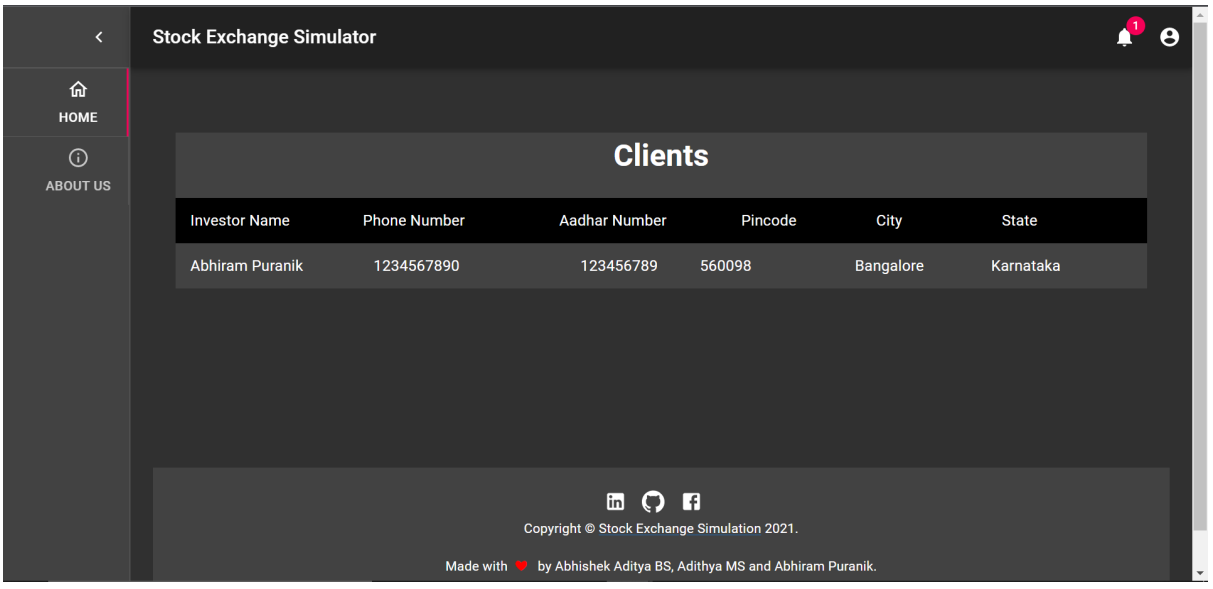
17. User logs in with broker credentials that he created.



18. Now, an investor can see this new broker that he created while filling the investor form.



19. Broker's home page displays the list of clients.



20. Log out button in the top right button logs out the user.

Screenshots for the schema change statements

1. Created ‘users’ table, which has serial id and stores username, encrypted password and isBroker bool value which indicates if the user is investor or broker.

```
CREATE TABLE users(
    id SERIAL PRIMARY KEY,
    username varchar UNIQUE,
    password varchar,
    isBroker BOOLEAN
);
CREATE EXTENSION pgcrypto;
```

2. New ‘marketValue’ table stores the totalValue of the stock market at different instances of time. Basically acts as a storage for history.

```
set timezone to 'Asia/Kolkata';
CREATE TABLE marketvalue(
    id SERIAL PRIMARY KEY,
    t TIME DEFAULT CURRENT_TIME,
    dt DATE DEFAULT CURRENT_DATE,
    totalValue DECIMAL(10,2)
);
```

3. Transactions table is updated, where id is made serial.

```
CREATE TABLE transactions(
    transaction_id SERIAL,
    amount DECIMAL(10,2),
    DATEOfTransaction VARCHAR,
    modeOfTransaction VARCHAR(30),
    typeOfTransaction VARCHAR(30),
    i_id INT,
    primary key(transaction_id)
);
```

Constraint changes:

1. **Granted usage and select on sequences to 'admin' role.**

```
grant USAGE,SELECT ON ALL SEQUENCES IN SCHEMA public to admin;
```

2. **Granted usage and select on transaction_id sequence to 'investor' role.**

```
grant USAGE,SELECT ON SEQUENCE transactions_transaction_id_seq TO
investor;
```

3. **Granted insert, update, delete on marketValue to 'investor' role.**

```
grant insert,update,delete on
investorsAndTraders,transactions,holdsCommodities,holdsBonds,holdsStock
s,holdsMF,marketValue TO investor;
```

Additional Constraints implemented in Backend :

>> All the investor related queries are done by opening the database connection by the user ' investor ', which has constraints on the tables related to the investors.
>> All the broker related queries are done by opening the database connection by the user ' broker ', which has constraints on the tables related to the brokers.
>> All the other queries like managing the stocks, updating prices of the stocks, authenticating users are done by the user ' admin ' who has privileges to all the tables in the database.

DBMS migration (from SQL based to No-SQL)

- Create a document called 'stockexchange'.
- Create collections for all the corresponding tables.
- All the rows will be converted to corresponding rows.

Contribution and Time Spent:

Abhiram Puranik - Worked on ‘sell’ and ‘buy’ functionalities and handled multiple edge cases and the website UI for the same, backend postgres queries and report write up. (15 hrs)
Abhishek Aditya BS - Worked on user authentication, connecting backend to frontend , backend routes and postgres queries and report write up. (15 hrs)
Adithya M S - Worked on the layout of UI and the home screen of investor, flow of the web app, graph in my profile page and backend postgres queries and report write up. (15 hrs)