CS 157A Intro to Database Management Systems

Final Project Report
StockU: Stock Data Aggregation Web-App

TEAM 34
Yang Li
Sachin Shah
En-Ping Shih (Team Leader)

Professor: Dr. Mike Wu
TA: Sriram Priyatham Siram

San Jose State University

December 11th , 2019

Project Requirement

Project Description

Our project will be a web-based application that can handle user/client input and report back to the user all relevant information regarding a certain company's stock. The application will provide up-to-date, useful information on the security of the client's choice and also recommend to the client whether they should purchase the security in question-based on the earnings of their portfolio. The application will do this by first retrieving information from our database for financial information. Then the application will perform operations using the retrieved data to determine if the security should be purchased by the user. For example, retrieving the beta value, which is a measure of a securities volatility, could be compared to a benchmark value. Based on this comparison, the security could be assigned a Boolean value of "buy" or "sell." Another function of our application would be storing the results of client queries in a database which will be hosted on a remote server.

The primary stakeholders for this application are people who may not have extensive knowledge of the stock market and are looking for a way to get started with investing. Furthermore, these are people who are interested in investing in public securities and not within the private sector, as a lot of the information on private companies is difficult to come by online. Our stakeholder's value time and convenience when it comes to retrieving reliable information about publicly traded securities. Since this is a web-based application and it meeting the functional requirements requires it to be used by end-users, this stakeholder group outlined above is of the utmost importance. Our application is important because nowadays speculators and passive investors are looking for a way to get a buy or sell decision without having to do extensive research. Our site will retrieve information and not require the user to do anything other than entering the ticker name of security. The user saves time, energy, and resources while receiving a user-oriented service.

System Environment

To build this application successfully, we will need to set up the environment based on the three-tier architecture (Figure 1) which contains the client, the server, and the database. On the client part, we have to make sure our web-based application is working on internet browsers on any computer or laptop for our users and clients. (ex. Google Chrome and Safari), and JavaScript will be our frontend development language. We plan on using MySQL to structure our database and store user and application data.

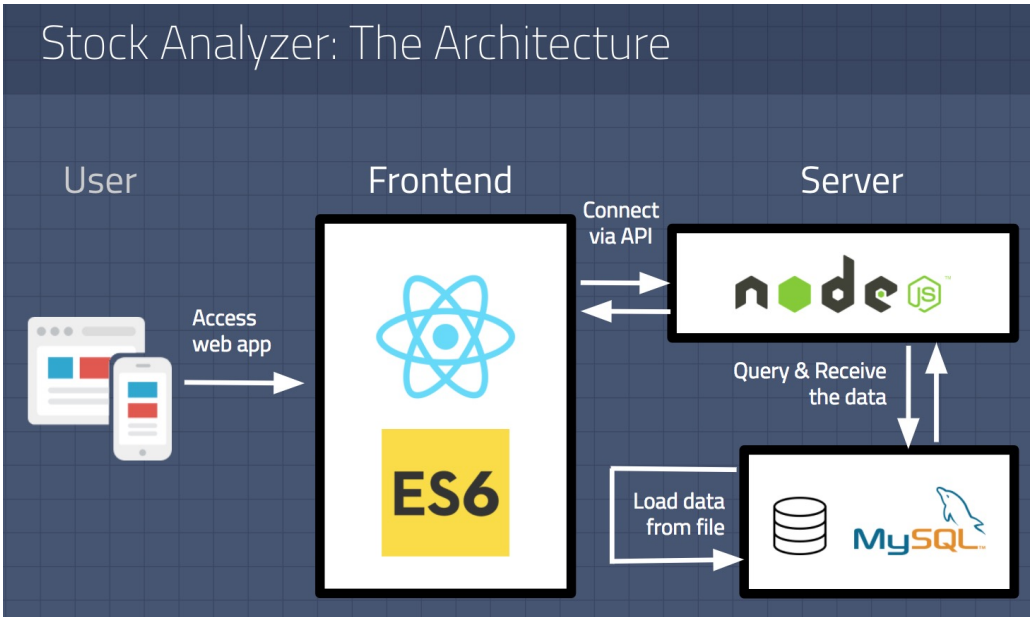


Figure 1. Three-Tier Architecture

HW/SW

- **RDBMS used**

MySQL Community Edition 8.0.17

- **Application Languages**

React, HTML, CSS, JSON, SQL, Javascript

- Server Layer
  - Node.JS server with Express,

Functional Requirements

<u>How Users will Access the System</u>

Our application is targeted towards consumers, and we will keep track of registered accounts using a table in the database. Users can access our application using a URL which they can access through their browser. Hosting our application on AWS will allow for this. Users will be able to register using their name and email. Users will have limited read and write capabilities. They will have the ability to query stock information from the GUI and based on their preferences, add the stock to their own watchlist. This watchlist is the only entity that users will have read, write, and delete capabilities for. The watchlist will be modeled using a table in the SQL database which will be associated with a user's unique PID, which will be generated upon account creation. Query results will also be stored, and users will have ONLY READ CAPABILITIES.

Functions:

<u>Search for Stock Information</u>

- User's should be able to enter input in the form of a stock ticker, and the application should display that stock's information on the presentation tier.
- Information about the query should be stored in the database layer so that the user can access the search results in the future

<u>View Past Search Results</u>

- Users should be able to see a view of their past search results after entering a time frame
- The system should be able to display past search results within the time frame that the user provides
- Users' READ ONLY capabilities of all search results

Add Stock to Watchlist

- After a user sees a given search result, they can choose to add the stock to their personal watchlist. This is the user's write capability to their own watchlist

View Entire Watchlist

- User's should be able to see a view of their entire watchlist.
- Input should be a button click, and the system should produce the output of a user's entire stock watchlist

Delete Stock from Watchlist

- User's should be able to remove a stock from their own watchlist by entering the ticker of the stock they want to delete
- If the stock is in the watchlist, the stock will be deleted from the watchlist table
- If the stock is not in the watchlist, the system will display a message saying "Stock not found in Watchlist"

Record Stock Earnings

- User's should be able to add the price they spent on a stock and the amount of share they have purchased.
- The system will calculate the returns by getting the latest price from the database.

Edit Stock Earnings

- User's should be able to update the price they spent on a stock and the amount of share they have purchased.
- The system will calculate the returns by getting the latest price from the database.

Non-Functional Requirements

- Front-end: React JS
- Server: Node.Js
- Database: using MySQL and React, to store stock information and personal stocks watchlists.

Access control and Security:

A second non-functional requirement is an issue regarding the security of user data. When our users create a new account, their information, such as user id, password, and email need to be stored in a secure location. Our users will only have read capabilities. In terms of access to the database storing users' stock watchlists, access control will be handled in the logic layer of our application. Once a user is logged in, they can edit and generate a view of the table associated with their unique user ID. They will only have access to their own watchlist.

Project Design

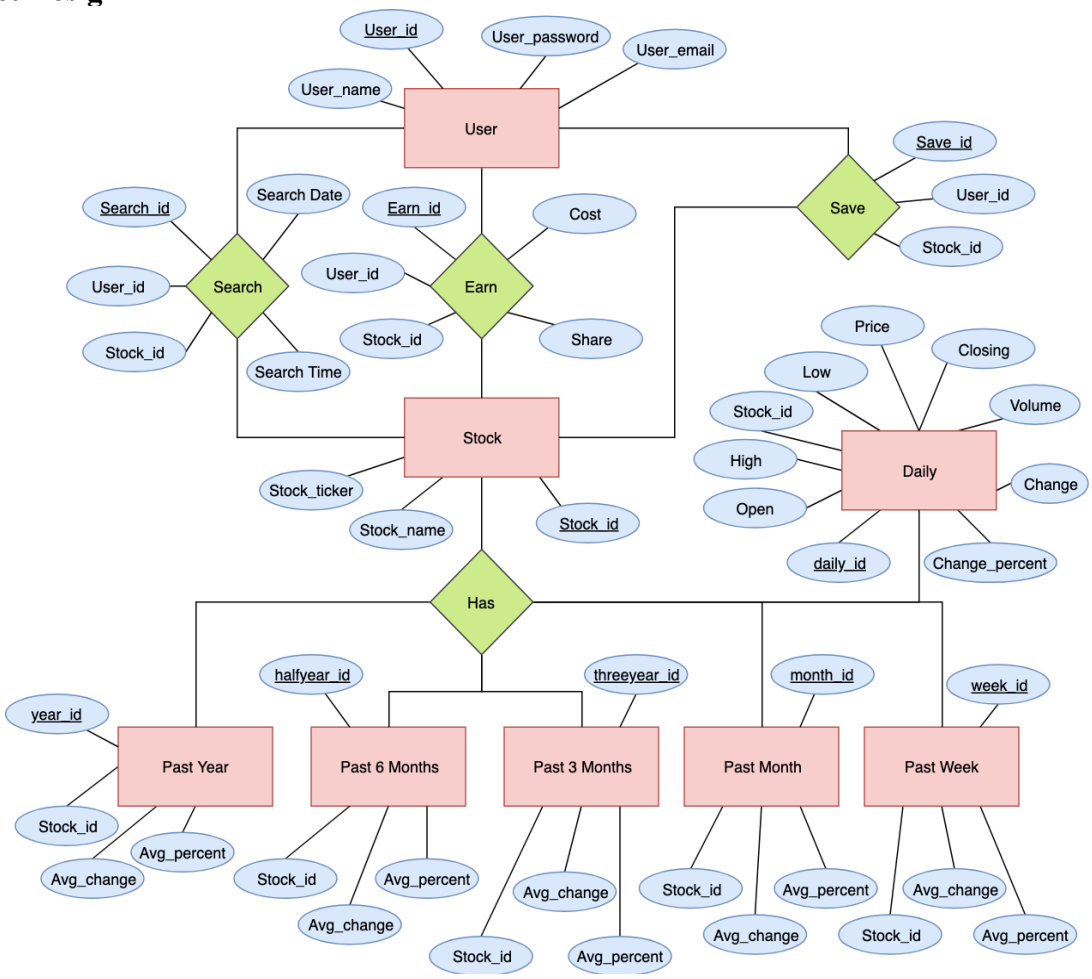


Figure 3. ER-Diagram

Entities and their Attributes:

- Entity: User
  - User(Attributes: User_id, User_name, User_password, User_email)
  - Constraint: User_id is primary key
  - User keeps track of all the unique users of the application
- Entity: Stock
  - Stock(Stock_id, Stock_ticker, Stock_name)
  - Constraint: Stock_id is primary key
  - Stock models a single company's stock
- Entity: Daily
  - Daily(Daily_id, Stock_id, Open, Closing, High, Low, Volume, Change, Change_percent)
  - Constraint: Daily_id is primary key
  - Daily shows the stocks data and metrics that indicate performance over the course of one trading day
- Entity: Week
  - Week(Week_id, Stock_id, Wk_change, Wk_percent)
  - Constraint: Week_id is primary key

    - Week shows the stocks data and metrics that indicate performance over the course of one trading Week
- Entity: Month
    - Month(Month_id, Stock_id, Mth_change, Mth_percent)
    - Constraint:Month_id is a primary key
    - Month shows the stocks data and metrics that indicate performance over the course of one trading Month
- Entity: Past 6 months
    - Half_year(Half_id,Stock_id,_Half_change,Half_percent)
    - Constraint: Half_id is primary key
    - Past 6 months shows the stocks data and metrics that indicate performance over the course of half a year
- Entity: Past Year
    - Year(Year_id,Stock_id, Year_percent,Year_change)
    - Constraint: Year_id is primary key
    - Past year shows the stocks data and metrics that indicate performance over the course of one trading year
- Entity: Past 3 months
    - Quarter(Quarter_id, Stock_id, Qt_change, Qt_percent)
    - Constraint: Quarter_id is primary key
    - Past 3 months shows the stocks data and metrics that indicate performance over the course of one trading quarter
- Relationship:Earn
    - Earn(Earning_id, Stock_id,User_id,Cost,Share)
    - Constraint:Earning_id is the primary key
    - Users can create an instance of an earning, which computes the potential gain users could have on an initial investment of a certain number of shares of a certain stock
- Relationship:Search
    - Search(Search_id,Stock_id, User_id, Search_date, search_time)
    - Constraint:Search_id is primary key
    - When users use the search bar in the GUI to search of a certain stock, the stock_id associated with that ticker is saved along with the user_id of the currently logged in user.
- Relationship:Save
    - Save(Save_id,User_id,Stock_id)
- Relationship:Has
    - Has(Stock, Past Week)
    - Has(Stock, Past Month)
    - Has(Stock, Daily)
    - Has(Stock, Past 6 months)
    - Has(Stock, Past Year)
    - Has(Stock, Past 3 months)
    - Each stock has an entity describing the financial information for the given time periods
- Dependencies:
    - Month, Daily, Past 3 months, Past 6 months, Past Year and Past week are all dependent on Stock
- Non-Trivial Functional Dependencies

- For User:
  - User_id -----> User_name,User_email,User_password
- For Stock
  - Stock_id -----> Stock_name, Stock_ticker
- For Earning
  - Earning_id -----> Cost, Share
    - If we know the earning_id, we know what the cost per share and the number of shares are
  - Earning_id, User_id-----> Stock_id
- For the multiple Has relationships
  - Daily_id------> Closing, Open, Low, High, Open, Change, Percent Change
  - Daily_id----> Stock_id
  - Week_id-----> Stock_id
  - Month_id----->Stock_id
  - halfyear_id-----> Stock_id
  - year_id------> Stock_id
- For Save:
  - Save_id------> Stock_id, User_id
    - If we know the save_id, we know the stock_id saved and what User saved the stock
- For Daily:
  - Daily_id, Stock_id------>Open, Closing, High, Low, Change, Change_percent
- For Week:
  - Week_id ------> Stock_id, Week_change, Week_percent
- For Month:
  - Month_id-------> Stock_id, Week_change, Week_percent
- For Past 3 months:
  - Quarter_id-----> Stock_id, Qt_change, Qt_percent
- For Past 6 months :
  - Half_id-----> Stock_id,Half_change,Half_percent
- For Past Year:
  - Year_id------Stock_id,Year_change, Year_percent

Relations:

- User
  - For User, User_id→ user_email, user_name, user_password so since the closure of User_id is all attributes, it is a superkey and User is in BCNF
  - No Trivial functional dependencies
- Earning
  - Earning_id is the key for this relation, and there are no trivial dependencies.
  - Earning_id→ User_id, Earning_id, User_id→ Cost, Share
  - Both functional dependencies are in BCNF, both are superkeys of the relation
- Stock
  - Stock_id is the primary key in this relation
  - Stock_id→ Stocker_ticker, stock_name
  - All functional dependencies are in BCNF

- Daily
  - Daily_id→ Stock_id
  - Daily_id, Stock_id→ Price, High, Low, Open, Change, Percent Change
  - Both left sides of the functional dependencies are superkeys
  - (SAME APPLIES FOR ALL TIME PERIOD TABLES)
  - Daily_id determines the stock whose information is stored, Daily_id becomes superkey because Stock_id will determine the rest of the metrics
- Week
  - Week_id→ Stock_id
  - Week_id, Stock_id→ Wk_change, Wk_percent
- Month
  - Month_id→ Stock_id
  - Month_id, Stock_id→ Mth_change, Mth_percent
- Quarter
  - Quarter_id→ Stock_id
  - Stock_id, Quarter_id→ Qtr_change, Qtr_percent
- Half_year
  - Half_id→ Stock_id
  - Half_id, Stock_id→ Half_change, Half_percent
- Year
  - Year_id→ Stock_id
  - Stock_id,Year_id--Yr_change, Yr_percent
- Save
  - Save_id→ User_id, Stock_id
  - Save_id is the key for this relation, each tuple is identified by the save ID

We can see that all the tables are in BCNF, and we see that Stock_id is the foreign key for most relations. There are no transitive dependencies in our design and it is in BCNF because for all functional dependencies, the left-hand side ends up being a superkey for that relation.

Database Table Screenshots

User Table

```
3 •  SELECT * FROM User ORDER BY User_name ASC;
4
```

100% 1:4 1 error found

Result Grid Filter Rows: Search Edit:

User_id	User_name	User_email	User_password
11	Alfi Kington	akingtona@bizjournals.com	aoWWyr
23	Anders Abad	aabadm@techcrunch.com	oTMS6w
29	Anette Carlesso	acarlessos@feedburner.com	8fx4HjHV
16	Anselm Patriche	apatrichef@umn.edu	DGZd65bVk
9837306...	Barry Allen	starlab@gmail.com	iamtheFlash
30	Bruce Wayne	batman@vinaora.com	iamBatman
21	Burtie Khalid	bkhalidk@ifeng.com	CjG5gO
22	Christyna O'Lehane	colehanel@wix.com	xiMfPT6kLclC
1	Corly Saipy	csaipy0@ebay.com	QdSIbjXV8W
26	Dennis Gibben	dgibbenp@dell.com	f2y2rY
8	Dov Belchambers	dbelchambers7@weather.com	jdRQIO2r
24	Durward MacQuist	dmacquistn@bing.com	1X4B8XnxtJTF
15	Fidelity Galero	fgaleroe@yellowpages.com	tmpvWsZat
13	Frederique McCague	fmccaguec@privacy.gov.au	wIDiMBvW9Q
5	Gianina Bentote	gbentote4@privacy.gov.au	2EwWQB
2	Giulietta Chasier	gchasier1@youtu.be	r8YOkvfD
17	Hilly Bummfrey	hbummfreyg@about.com	jXlYxjb09
18	Igor Stitcher	istitcherh@newyorker.com	CbSuKqcfuQ9E
12	Ingemar Pack	ipackb@sciencedirect.com	ssptea

Stock Table

```
3 •  SELECT * FROM Stock;
4
```

100% 1:2 1 error found

Result Grid Filter Rows: Search Edi

Stock_id	Stock_ticker	Stock_name
1	FB	Facebook, Inc.
10	OI	Owens-Illinois, Inc.
111	MSFT	Microsoft Inc
113	TSLA	Tesla
115	ROKU	Roku Inc
116	AMD	Advanced MicroDevices
117	UBER	Uber Technologies Inc
118	COKE	Coca-Cola Consolidated
119	CRM	Salesforce Inc
120	LYFT	Lyft Inc
121	IBM	IBM Incorporated
122	GOOGL	Alphabet Inc
123	NFLX	Netflix Inc
124	ACB	Aurora Cannabis Inc
125	BAC	Bank of America Inc
126	INTC	Intel Corp
128	CGSP	Costar Group
130	GWPH	GW Pharmaceuticals
133	FIT	Fitbit Inc

Earning Table

```
3 •  SELECT * FROM Earnings;
4
```

100% 1:1 1 error found

Result Grid Filter Rows: Search E

Earning_id	User_id	Stock_id	Cost	Share
13	10	134	9.21	3
15	27	111	152.01	1
16	25	123	314.12	4
18	13	126	58.02	2
19	28	125	33.52	3
2	13	122	1315.22	2
20	6	118	275.40	1
21	25	126	58.39	4
23	30	113	330.21	2
26	7	126	58.61	2
27	16	111	152.31	5
28	14	117	29.02	3
29	3	122	1316.56	5
3	24	120	49.50	2
31	4	128	618.90	4
32	7	133	9.08	3
33	98373069147471873	111	152.45	3

Save Table

```
3 •  SELECT * FROM Save;
4
```

100% 23:11 1 error found

Result Grid Filter Rows: Sear

Save_id	User_id	Stock_id
1	4	10
10	7	116
11	6	116
12	8	132
13	4	124
14	15	126
15	4	126
16	24	10
17	24	125
18	1	126
19	98373069147471873	117
2	25	10
20	98373069147471874	113
21	98373069147471879	120
22	9	128
23	22	122

Search Table

```
3 •  SELECT * FROM Search;
4
```

100% 1:1 1 error found

Result Grid Filter Rows: Search Edit: Export/Impo

Search_id	User_id	Stock_id	Search_time
11	30	11	Tue Nov 05 2019 15:34:11 GMT-0800 (Pacific Standard Time)
12	12	122	Wed Dec 04 2019 15:08:21 GMT-0800 (Pacific Standard Time)
13	13	113	Fri Dec 06 2019 10:05:07 GMT-0800 (Pacific Standard Time)
14	30	124	Tue Nov 05 2019 17:10:33 GMT-0800 (Pacific Standard Time)
15	15	115	Wed Dec 04 2019 13:07:37 GMT-0800 (Pacific Standard Time)
16	16	116	Tue Nov 05 2019 16:22:40 GMT-0800 (Pacific Standard Time)
17	17	117	Sat Nov 23 2019 21:34:16 GMT-0800 (Pacific Standard Time)
18	30	118	Sun Nov 24 2019 19:41:16 GMT-0800 (Pacific Standard Time)
19	19	119	Tue Nov 05 2019 17:48:25 GMT-0800 (Pacific Standard Time)
2	2	2	Fri Dec 06 2019 11:05:07 GMT-0800 (Pacific Standard Time)
20	20	120	Fri Dec 06 2019 12:05:07 GMT-0800 (Pacific Standard Time)
21	30	121	Tue Nov 05 2019 20:04:11 GMT-0800 (Pacific Standard Time)
22	12	122	Wed Nov 13 2019 15:25:27 GMT-0800 (Pacific Standard Time)
23	13	123	Tue Nov 05 2019 19:34:14 GMT-0800 (Pacific Standard Time)
24	30	124	Sat Nov 23 2019 17:34:16 GMT-0800 (Pacific Standard Time)
25	5	125	Fri Dec 06 2019 15:05:07 GMT-0800 (Pacific Standard Time)
26	6	126	Fri Dec 06 2019 16:05:07 GMT-0800 (Pacific Standard Time)
27	7	117	Fri Dec 06 2019 17:05:07 GMT-0800 (Pacific Standard Time)

Daily Table

```
3 •  SELECT * FROM Daily;
4
```

100% 1:1 1 error found

Result Grid Filter Rows: Search Edit: Export/Impor

Daily_id	Stock_id	Open	Price	Low	High	Volume	Change	Change_percent
1	1	200.60	200.90	200.07	201.57	12270000	1.69	0.85
10	10	10.27	10.36	10.26	10.46	1880000	0.21	2.07
111	111	152.32	152.33	151.52	152.50	15184000	0.29	0.19
113	113	331.12	331.29	328.57	333.93	5555600	2.37	0.72
115	115	167.05	168.85	159.61	161.64	19085100	-2.64	-1.61
116	116	39.46	39.76	39.07	39.41	33600100	0.42	1.08
117	117	29.42	28.88	29.49	29.49	22421400	-0.040	-0.14
118	118	273.90	275.45	272.05	274.96	18200	1.42	0.52
119	119	161.92	162.83	161.11	161.51	4012800	-1.03	-0.63
120	120	49.29	49.75	48.80	49.37	3044300	0.37	0.76
121	121	135.35	135.71	133.62	133.77	3280900	-1.32	-0.98
122	122	1313.42	1317.42	1309.47	1312.13	940000	-0.87	-0.66
123	123	313.93	316.92	312.75	315.93	4096900	3.44	1.10
124	124	2.47	2.54	2.43	2.52	36025000	0.13	5.44
125	125	33.49	33.60	33.31	33.42	32097700	0.0070	0.21
126	126	58.53	58.59	57.91	58.51	18184200	-0.39	-0.66
128	128	617.98	619.80	609.27	617.17	312600	-0.22	-0.036
130	130	99.16	101.24	98.01	100.43	597500	1.60	1.62
131	131	313.93	316.82	312.75	315.93	4096900	3.44	1.60

Week Table

```
3 •  SELECT * FROM Week;
4
```

100% 1:1

Result Grid Filter Rows: Search

Week_id	Stock_id	Wk_change	Wk_percent
1	1	-1.19	-0.59
10	10	0.39	3.86
111	111	1.90	1.20
112	112	66.09	3.16
113	113	-24.89	-0.93
115	115	4.46	1.02
116	116	-0.37	0.00
117	117	0.14	0.14
118	118	-14.08	-0.60
119	119	0.01	0.05
120	120	2.22	5.48
121	121	0.61	0.08
122	122	3.95	0.81
123	123	2.97	1.35
124	124	-0.12	-4.55
125	125	0.73	2.23
126	126	0.61	1.05
128	128	22.65	3.81
130	130	1.22	1.23

Month Table

```
3 •  SELECT * FROM Month;
4
```

100% 1:1

Result Grid Filter Rows: Search

Month_id	Stock_id	Mth_change	Mth_percent
1	1	6.87	3.54
10	10	1.08	1.6
111	111	6.77	4.68
112	112	20.81	1.17
113	113	14.83	4.74
115	115	10.97	7.32
116	116	6.02	18.17
117	117	-4.35	-12.30
118	118	-5.62	-2.04
119	119	4.61	2.91
120	120	4.87	11.04
121	121	-0.80	-0.59
122	122	43.44	3.44
123	123	23.21	7.96
124	124	-1.11	-30.75
125	125	1.70	5.38
126	126	1.45	2.56
128	128	62.82	11.42
130	130	-31.13	23.36

Quarter Table

```
3 •  SELECT * FROM Quarter;
4
```

00% 1:1

Result Grid Filter Rows: Search

Quarter_id	Stock_id	Qt_change	Qt_percent
1	1	13.17	7.02
10	10	0.16	1.57
111	111	13.26	9.87
112	112	56.68	2.21
113	113	107.79	48.52
115	115	14.48	9.93
116	116	9.48	31.35
117	117	-3.16	-9.85
118	118	-69.66	-20.50
119	119	11.06	7.28
120	120	-5.17	-9.55
121	121	-2.53	-1.92
122	122	134.77	11.53
123	123	11.96	6.39
124	124	-3.49	-58.26
125	125	6.29	23.27
126	126	10.82	22.44
128	128	10.08	1.09
130	130	-40.36	-27.27

Half_year Table

```
3 •  SELECT * FROM Half_year;
4
```

00% 1:1

Result Grid Filter Rows: Search

Half_id	Stock_id	Half_change	Half_percent
1	1	22.72	12.82
10	10	-0.97	-8.56
111	111	25.65	20.40
112	112	-15.52	-0.85
113	113	141.72	75.29
115	115	67.03	71.81
116	116	11.12	39.67
117	117	-10.2	-25.6
118	118	-41.57	-13.34
119	119	7.23	4.64
120	120	-5.85	-10.17
121	121	4.88	3.77
122	122	182.68	16.29
123	123	-28.62	-8.34
124	124	-5.50	-68.75
125	125	6.16	22.88
126	126	13.31	29.78
128	128	113.24	19.42
130	130	-74.42	-42.16

Year Table

```
3 •  SELECT * FROM Year;
4
```

00% 1:1

Result Grid Filter Rows: Search

Year_id	Stock_id	Year_change	Year_percent
1	1	61.65	50.13
10	10	-6.59	-38.88
111	111	50.26	49.70
112	112	261.67	17.00
113	113	19.82	6.39
115	115	127.85	393.14
117	117	-11.97	-29.06
118	118	94.85	50.00
119	119	25.96	19.30
120	120	-29.31	-37.44
121	121	19.24	16.70
122	122	249.41	23.65
123	123	47.00	17.56
124	124	-2.74	-52.29
125	125	8.36	33.49
126	126	10.97	24.11
128	128	285.58	80.99
130	130	1.10	1.09
133	133	1.78	29.70

Implementation

As mentioned before, we used React to implement our DB application. Since React cannot work with MySQL directly, we used express.js in order to embed SQL queries in our React. The backend of the application was handled pretty much within one file ‘app.js’. The

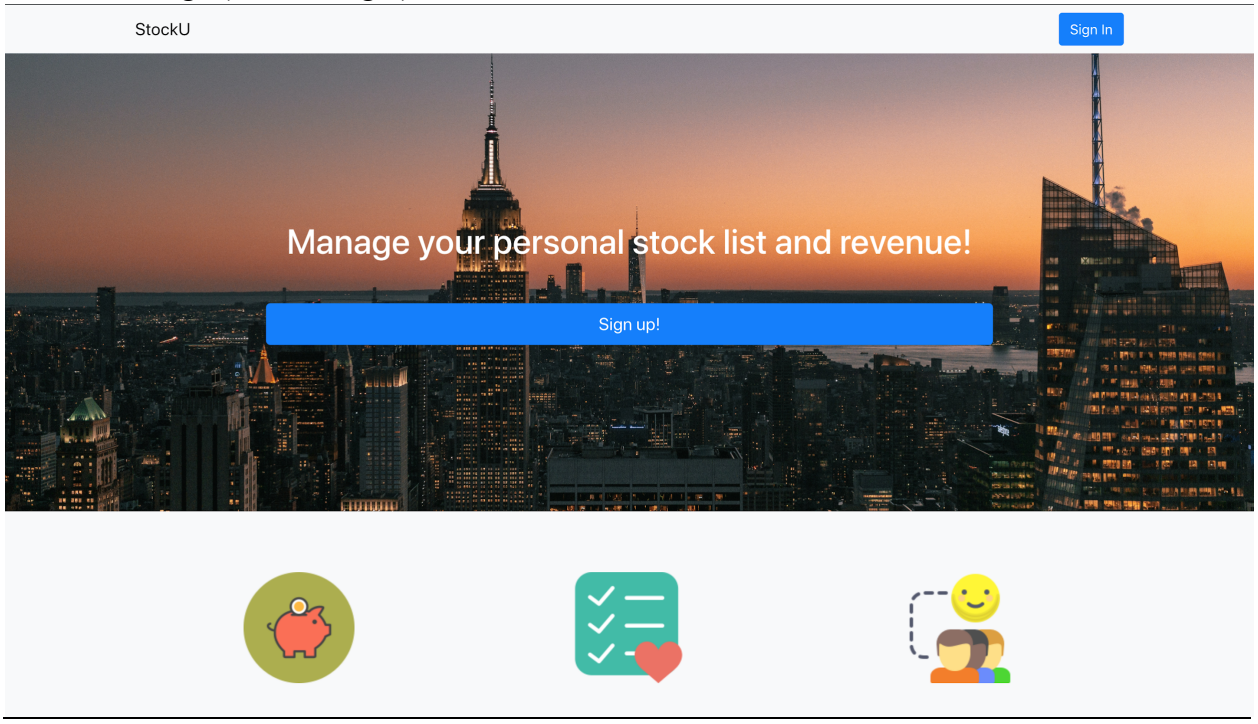
First off, we decided to implement the frontend of the application first. We used the React component approach where each component is represented by a folder in the source code. The NavBar class is for the navigation bar that spans the entire top part of every page. We decided to structure our application so that each static page was managed by one folder. The folders contained normally, a .js file which contained the structure for the page. A navbar.js, a header.js file, and a footer.js file. The header.js files in each of the components handled the dynamic functionality. Our code was successful in addressing all of our functional requirements.

The Non-functional requirements that we had to address were how User accounts were going to be created and how Users would be able to log in. Each user, per the ERD, would have to have a unique ID which would allow other tables such as Save and Earnings to store information unique to the currently logged in user. We implemented a User table in our schema which stored User profile information. We also had a profile page that would display the User name and email.

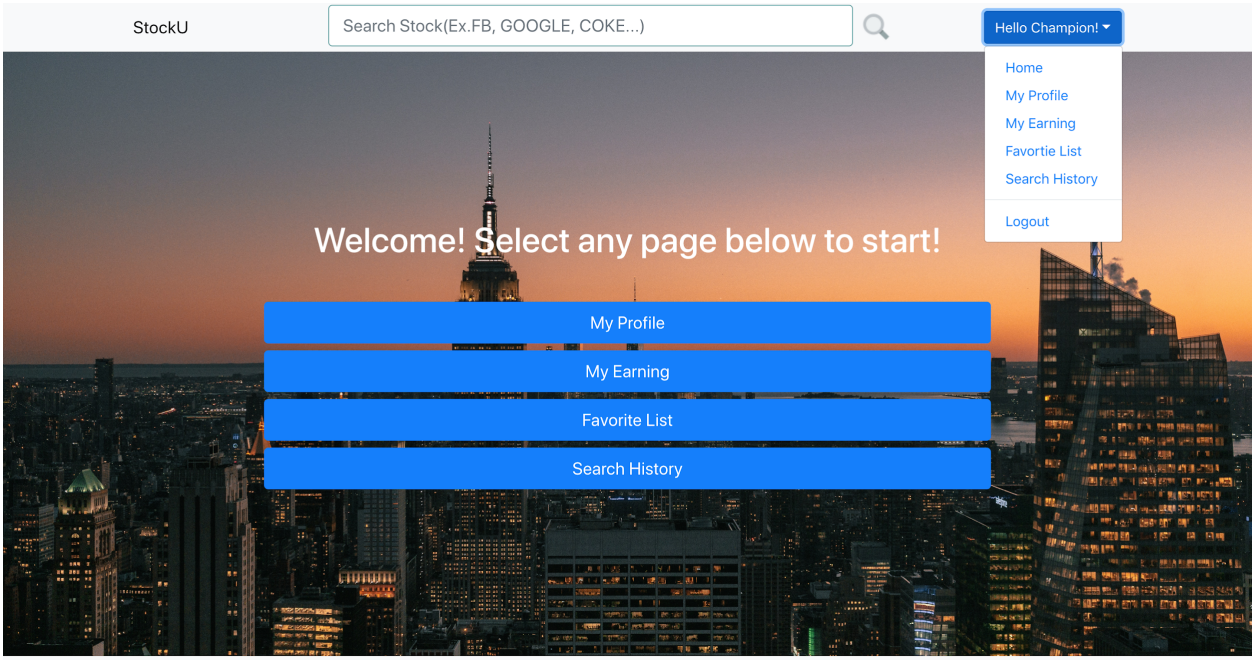
User sign up is handled by the User table, and inserting a new tuple into that table when a new user creates an account.

SQL Commands and Screenshots for GUI

Dashboard Page (Before Login)



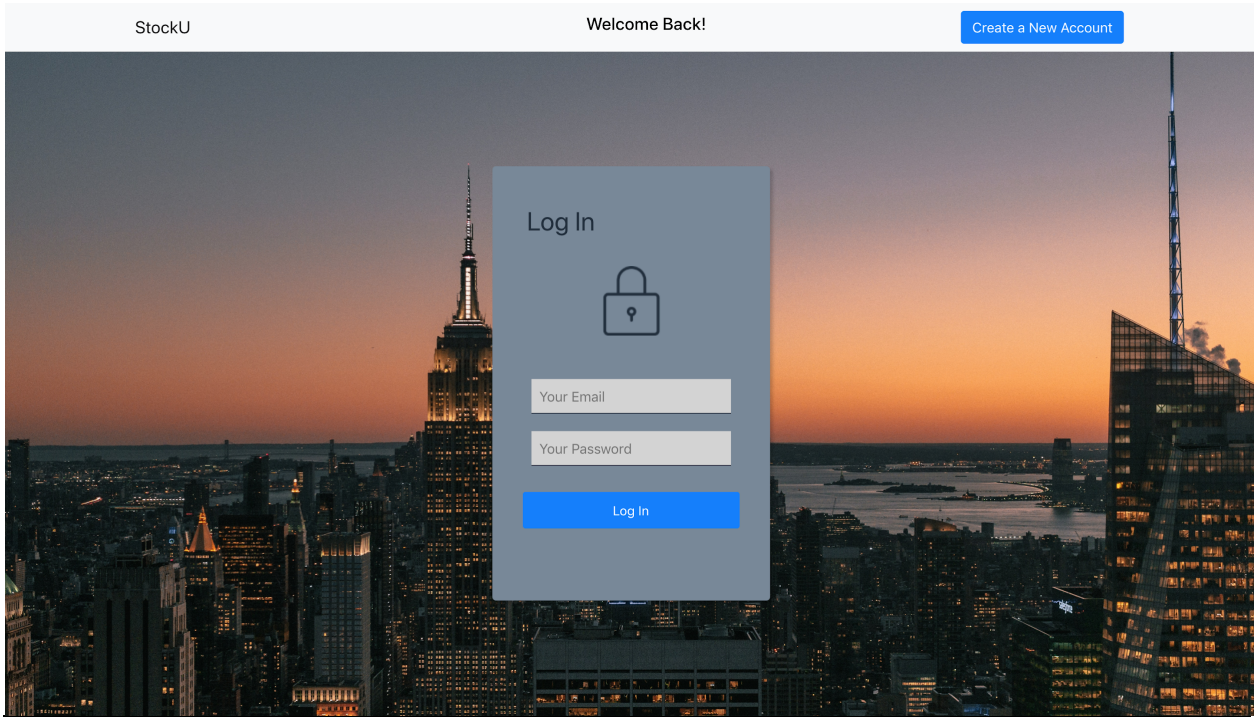
Home Page (After Login)



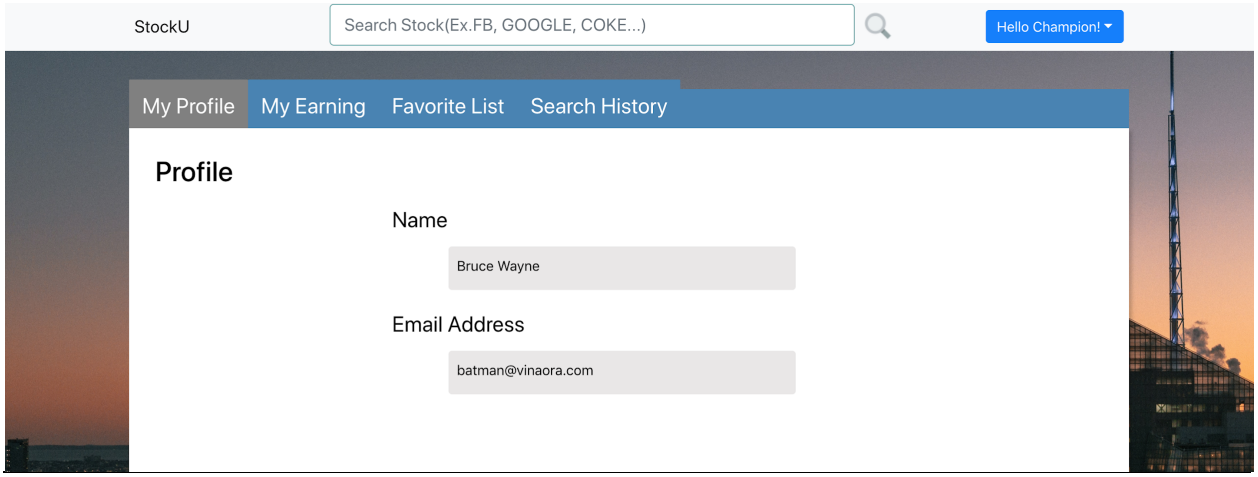
Login Page

When a User wants to Log In, they click on the sign-in button on the top navigation bar. And the following code handles the sign in:

```
const SELECT_USER = `SELECT * FROM User WHERE User_email= '${email}'`;
```



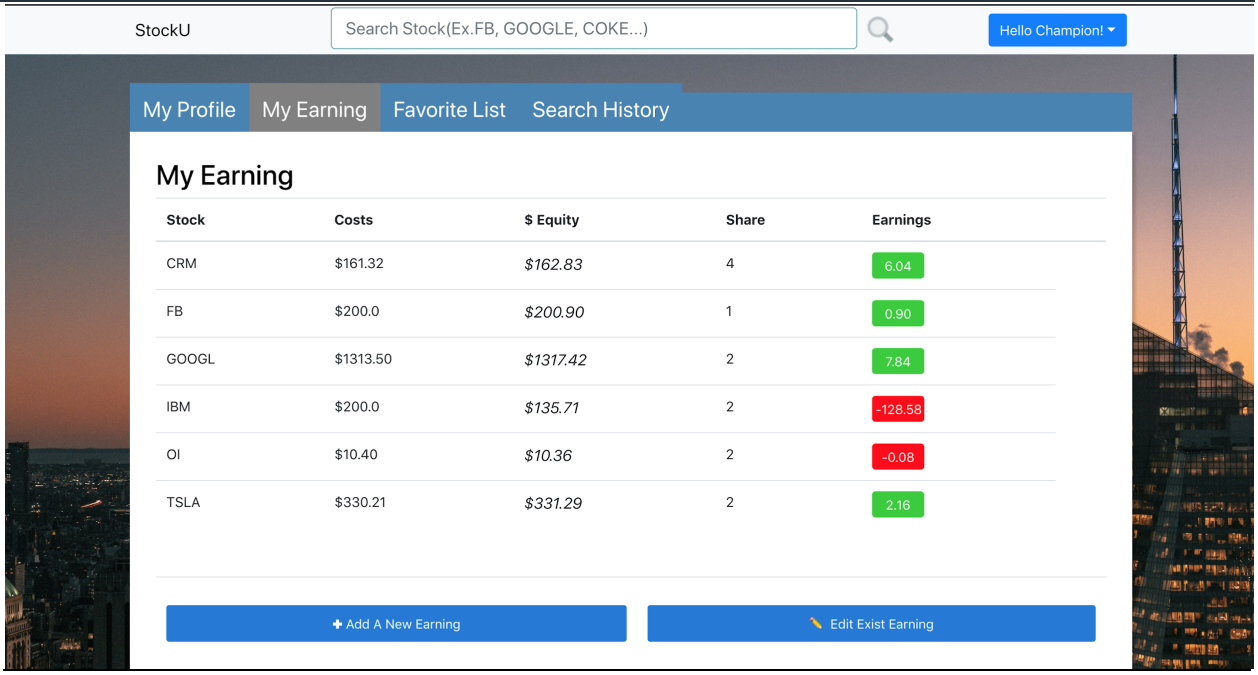
User Profile Page



My Earning Page

Another functional requirement to implement was the record earnings function, which allowed the user to input the number of shares they bought and the price per share the transaction was made and have the earnings from that purchase returned to them. In our current iteration of the application, the way that the Earnings table gets its data is from when the user goes to the manage earning page, where they can input their shares and costs and that information is stored in the backend in the Earnings table.

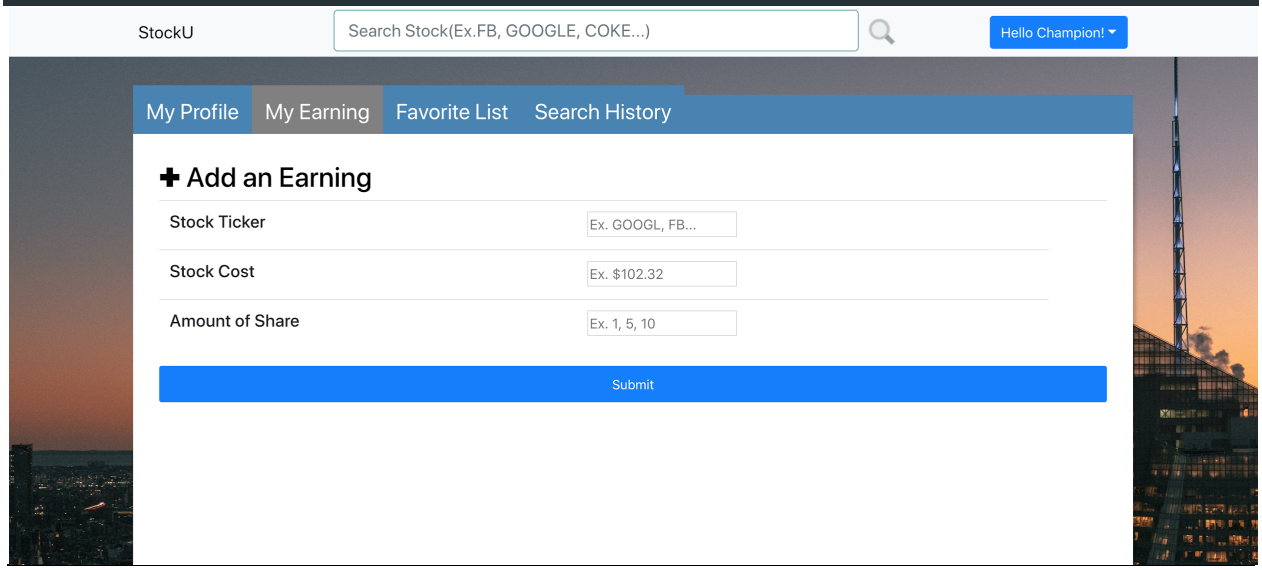
```
const SELECT_EARNING_LIST = `SELECT * FROM Earnings JOIN Daily USING(Stock_id)
JOIN Stock USING(Stock_id) JOIN User USING(User_id) WHERE User_email ='${localUser}'
ORDER BY Stock_ticker ASC`;
```



Add New Earning

The ability for users to add to their earnings was made possible by the Earnings table, and the entities involved are User and Stock entities.

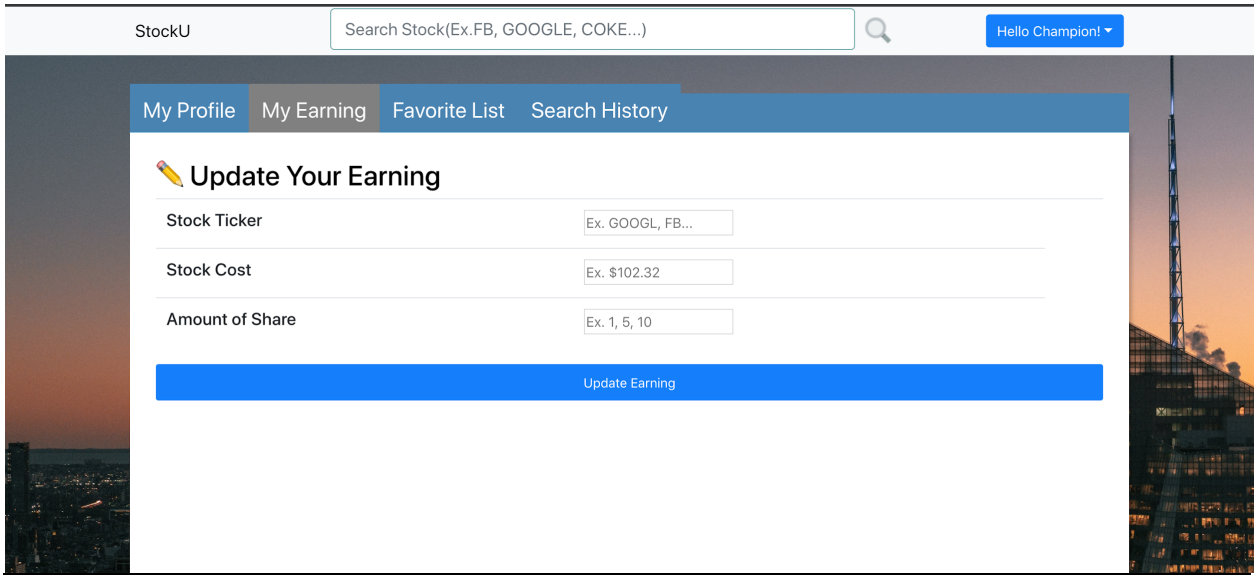
```
const INSERT_EARNING = `INSERT INTO Earnings VALUES(UUID_SHORT(),'${localID}',
(SELECT Stock_id FROM Stock WHERE Stock_ticker
='${stockName}'),'${cost}','${share}')`;
```



Update Earning

The ability for users to update to specific earnings in their Earnings Table, and the entities involved are User and Stock entities.

```
const UPDATE_EARNING = `UPDATE Earnings JOIN User USING(User_id) JOIN Stock
USING(Stock_id) SET Cost='${cost}', Share='${share}' WHERE Stock_ticker='${stockName}'
AND User_email ='${localUser}'`;
```



Search Result

Another functional requirement that was addressed was that users could add a stock to their favorites list, and delete from that list as well. The add to list functionality was handled by a button on the search result page and by INSERT SQL statements. If a user wanted to add the stock into their favorite list, the stock information, most importantly the stock_id, would be inserted into the Save table in our backend schema.

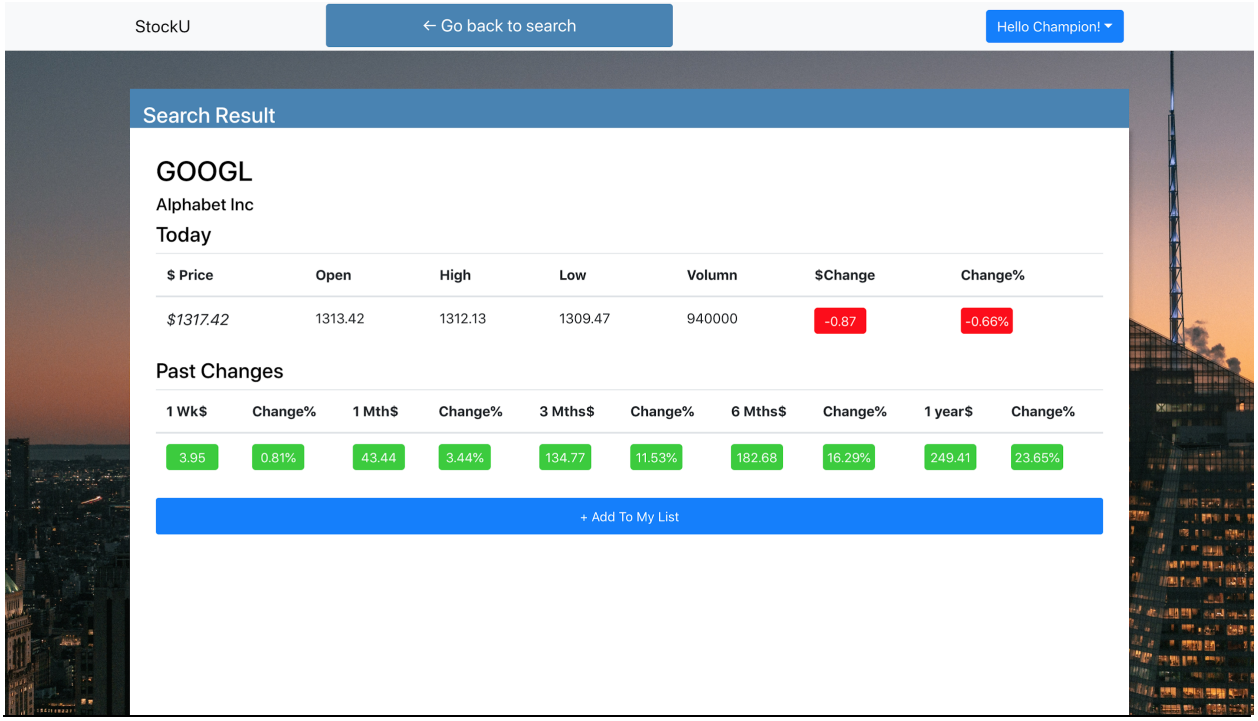
```
const SAVE_TO_FAVORITE = `INSERT INTO Save VALUES(UUID_SHORT(), '${localID}',
'${stockID}')`;
```

The User can search for a given stock. Once a user is logged in, they can search for a given stock using that stock's ticker. For example, if you wanted data returned for Apple, you would search for GOOGL. This functionality uses NATURAL JOIN to get all the stock information we needed in all six tables.

```
const SELECT_SEARCH_RESULT = `SELECT * From Stock
JOIN Daily USING (Stock_id) JOIN Week USING (Stock_id) JOIN Month USING (Stock_id)
JOIN Quarter USING (Stock_id) JOIN Half_year USING (Stock_id) JOIN Year USING
(Stock_id) WHERE Stock_ticker = '${search_key}'`;
```

When the backend is fetching the specific stock results, the search will be added to the user's search history. This is also a challenging part since we will need to use a subquery to INSERT the value from another table.

```
const INSERT_SEARCH_HISTORY =`INSERT INTO Search
VALUES(UUID_SHORT(),'${localID}',(SELECT Stock_id FROM Stock WHERE
Stock_ticker='${hisotry_key}'),'${searchTime}')`;
```



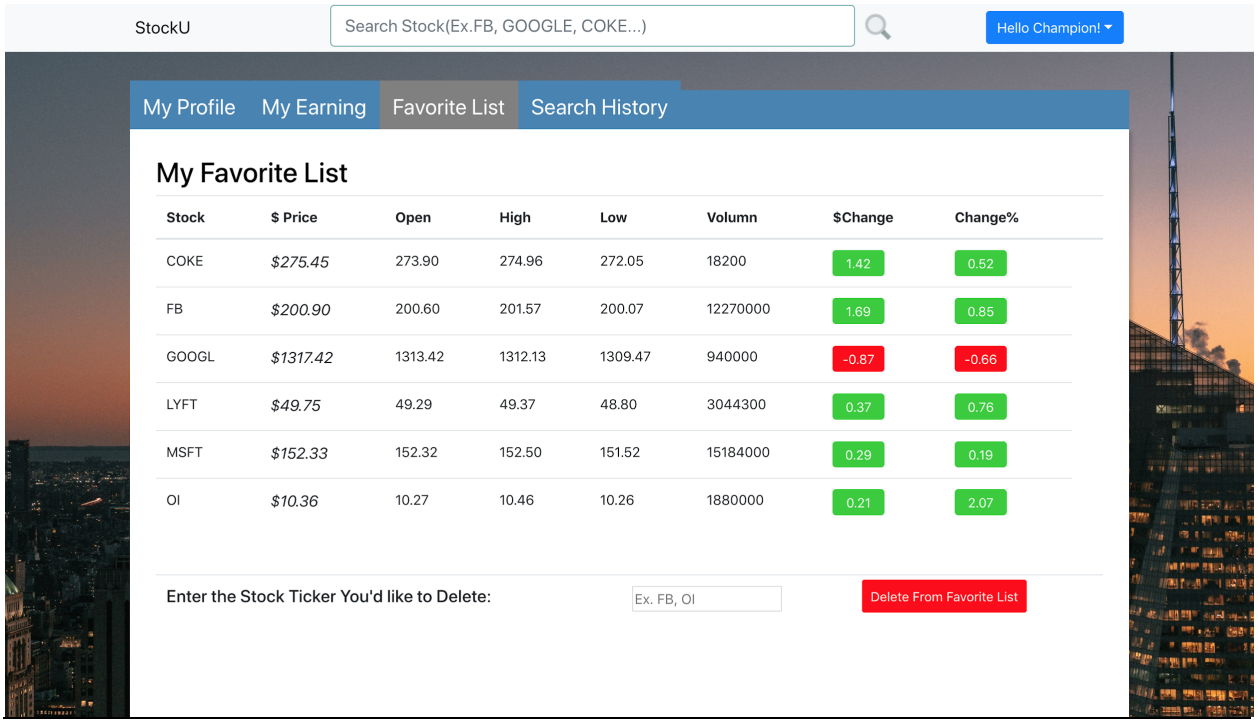
My Favorite List

Users can also view their entire favorite list, in the GUI it is under its own page titled "Favorite List." A SELECT statement is used to implement this functionality.

```
const SELECT_FAV_LIST = `SELECT * FROM Save, User, Stock, Daily
WHERE User_email ='${localUser}' AND User.User_id = Save.User_id AND Save.Stock_id =
Stock.Stock_id AND Stock.Stock_id= Daily.Stock_id ORDER BY Stock_ticker ASC`;
```

The delete function under the favorite list is one of the challenges, in the DELETE needs 'any(SELECT...)' to delete specific rows.

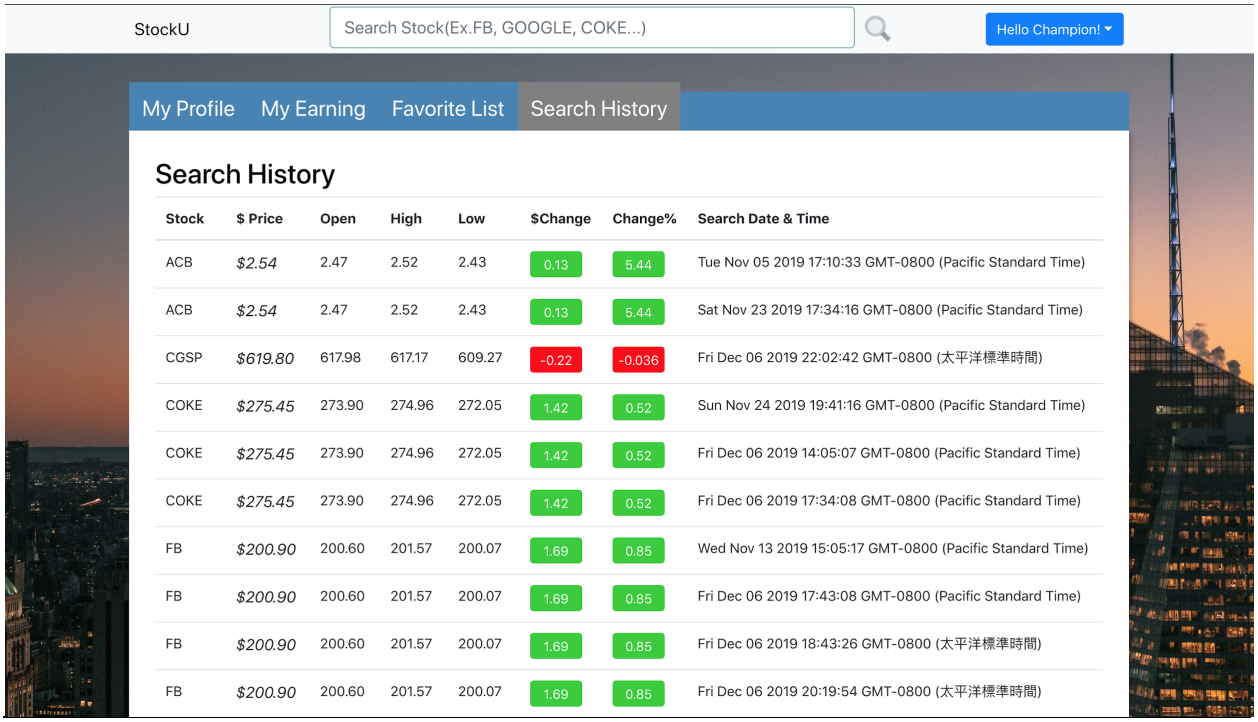
```
const DELETE_FROM_FAVORITE = `DELETE FROM Save WHERE User_id = '${localID}' AND
Stock_id = any(SELECT Stock_id FROM Stock WHERE Stock_ticker ='${stockName}')`;
```



Search History

Another functional requirement was that Users should be able to see their past search history. This was accomplished through the use of the search table, and a JOIN operation. Through the side navigation bar, the user can click on view history and the tuples in the Search table that correspond to the User's email will be returned.

```
const SELECT_SEARCH_RESULT = `SELECT * From Stock
JOIN Daily USING (Stock_id) JOIN Week USING (Stock_id) JOIN Month USING (Stock_id)
JOIN Quarter USING (Stock_id) JOIN Half_year USING (Stock_id) JOIN Year USING
(Stock_id) WHERE Stock_ticker = '${search_key}'`;
```



Procedures (step by step) of how to set up and run your system
The detailed setup will be on the README.md on our Github Organization:
https://github.com/CS157A-34/CS157A-Team34/blob/master/README.md

Project Conclusion
Yang Li (Liam)

Throughout this database project, I have learned the three-tier architecture, the database design, ER-diagram implementation and Web-based Application using React, Node.js, and MySQL. By manipulating the data using MySQL query language transferred to JavaScript accessible in Node.js helped me understand how the architecture works. Before this project, I only have very little knowledge and experience about React, Node.js and MySQL including database. But with my teammates, Sharon and Sachin, we are successfully implemented a web-based application with three-tier architecture.

Sachin Shah

Throughout this database project, I gained a greater understanding of how to design the schema of databases in a practical sense, and how the backend schema affects how you create your frontend and how well you can fulfill your functional requirements. I also was exposed to React and javascript, which are two technologies I had not worked with much in the past. Also working with mySQL to setup the schema on the backend, which was my role, was eye-opening to me. I also have a lot more to learn in terms of react, as the frontend aspect of it was slightly confusing and very difficult to pickup. In the future, we can continue to implement more features such as being able to display the highest earning stock based on the User_id of the logged in user.

For example:

```
//Get the highest earning stock in a given users list
app.get('/earning', (req,res)=> {
  console.log(req.query);
  const highestStock = req.query.highestStock;
  const GET_HIGHEST_EARNING_STOCK = 'SELECT Stock_ticker FROM Earnings JOIN Stock USING(Stock_id) ORDER BY (Earnings.Cost * Earnings.Share)
 console.log(GET_HIGHEST_EARNING_STOCK);
 connection.query(GET_HIGHEST_EARNING_STOCK, (err, results)=> {
   if(err){
     return res.send(err)
   }
   else {
     return res.json({
       data: results
     })
   }
 });
})
```

En-Ping Shih(Sharon) - Team Leader

Look back on the entire project development process, I do learn a lot from the project. As a team leader, I have led all the parts, including the UI design, Frontend, Backend, Database, and Documentation. This project gives us a chance to practice what we learned in class into actual application. For example, the 3-tier architecture, ER diagram for entity relationship, create MySQL database table and schema, and SQL command implementation. I think I learned the most is to create SQL command to fetch and modify data. I created the database tables, and use SELECT, INSERT, DELETE, UPDATE, JOIN, and subquery to apply the functionality. Even though the functionality is pretty throught, but there are a few functions that we can add on if we have more help.
Besides technical skills, I have also experienced how hard people management could be. Everyone has different schedules, classes, and sometimes stuff happens. During this project, I believe my time management skills has improved for both the team and my schedule. I not only have gained a lot of technical skills and knowledge during the problem-solving and developing process, but also had an excellent opportunity to train my leadership skills.

Future Improvement of the Application

There are several fixes that we could have made to our application, and also there are ways that we could have improved upon it. For example, in the favorite list page as well as the earnings page, there were duplicate entries for the same stock. In the future, we could have fixed this error by first sending a request to the backend that checks for duplicate tickers in a user's favorites list. Another thing that could improve our application was that we could utilize a real time stock data API that allows for the tables in Daily, Week, Month, Half_year, and Year to be consistently updated. For the current iteration of the application, we used DDL to insert data into the tables. Other improvements include using Machine Learning to predict earnings of certain stocks, and allowing users to compute a variety of statistics for stocks. One thing that could also improve the accessibility of the application would be if the application was hosted using AWS, because then we could improve the connection pool and allow for concurrent connections to the same database by creating different instances for each user. This would allow our application to scale over time and incorporate more tuples of data.