CS 157A Intro to Database Management Systems

Final Project Report
StockU: Stock Data Aggregation Web-App

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

Professor: Dr. Mike Wu
TA: Sriram Priyatham Siram

San Jose State University

December 5th , 2019

Project Requirement

Project Description

Our project will be a web-based application which 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 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 used**

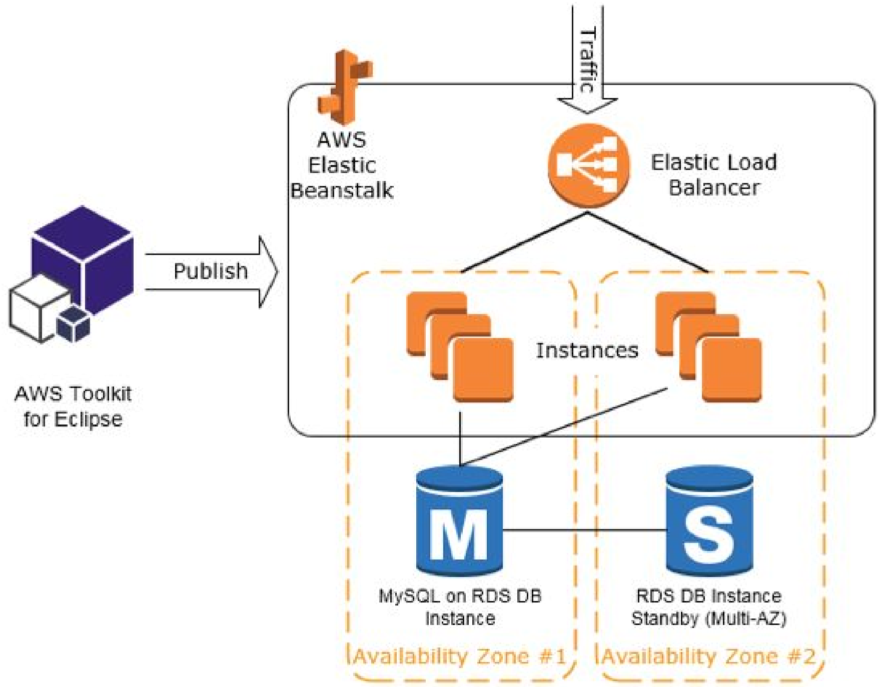


Figure 2. Deployment Diagram for application on AWS

- **RDBMS used**

MySQL Community Edition 8.0.17

- **Application Languages**

React, JSX, HTML, CSS, JSON, JSP, 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:

Search for Stock Information

- 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

View Past Search Results

- 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.

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

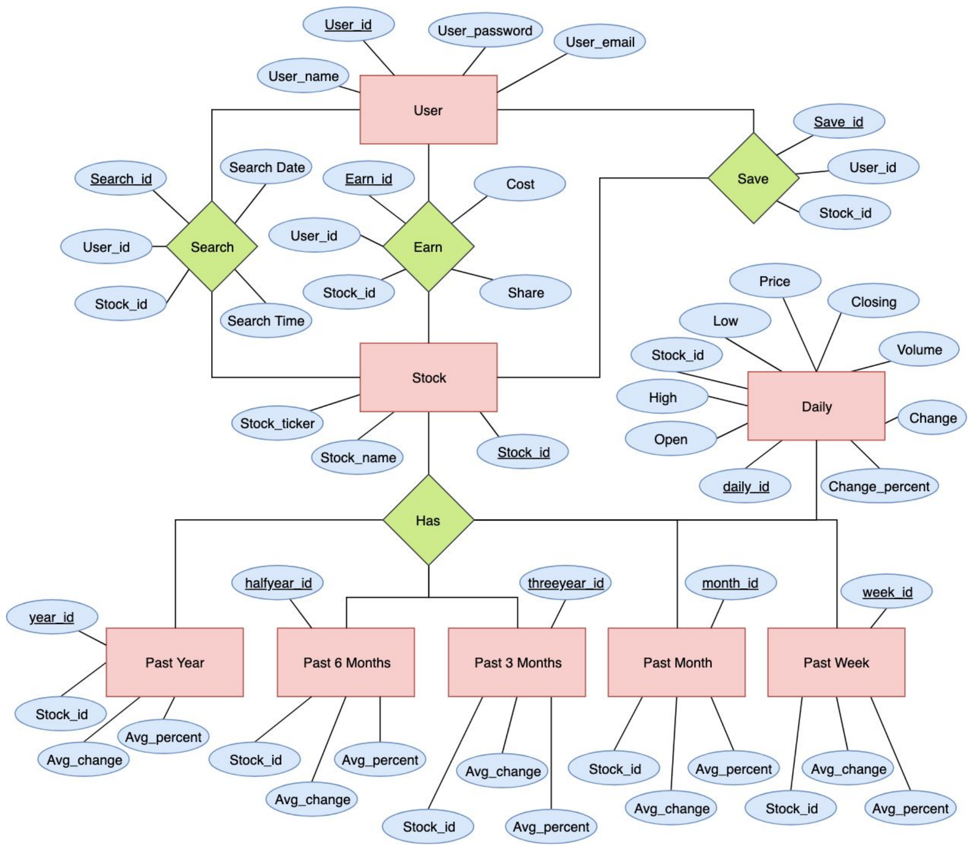


Figure 3. ER-Diagram

n Updated ERD Model for the application.

Ø Identify the entities, attributes, dependences, relationships, constraints, etc.
Ø Show multi-way relationship, inheritance, ISA, weak entity sets, etc.
Ø List all completely non-trivial FDs that apply to your design.
Ø Convert ER to schemas
Ø Explanation for each entity set and relationship, write a short description in plain English of what it represents or models. One or two sentences per entity set and relationship is enough. These descriptions are primarily to help us understand what you are modeling.

ERD Model for the Application Explained

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
    -

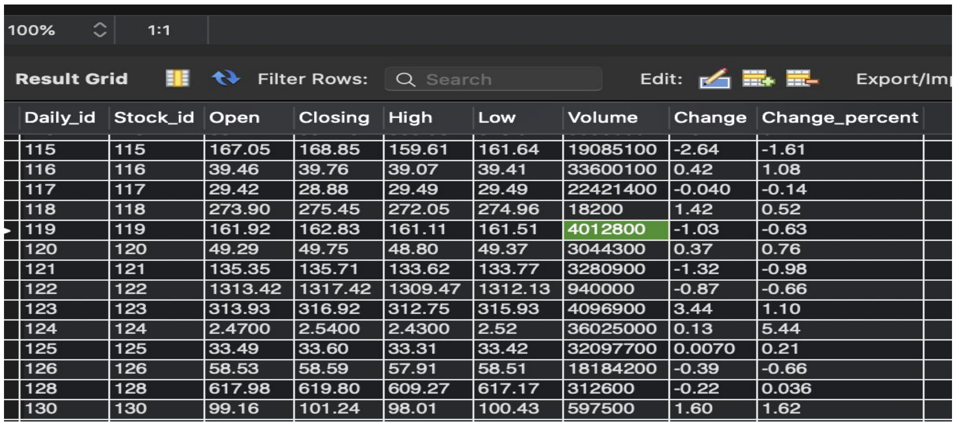
n Perform the normalization process, and perfect the relational database schemas to BCNF
n Create and show at least 10 tables according to schemas and model the data stored in the database (Each table must contains at least 15 tuple instances.)

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

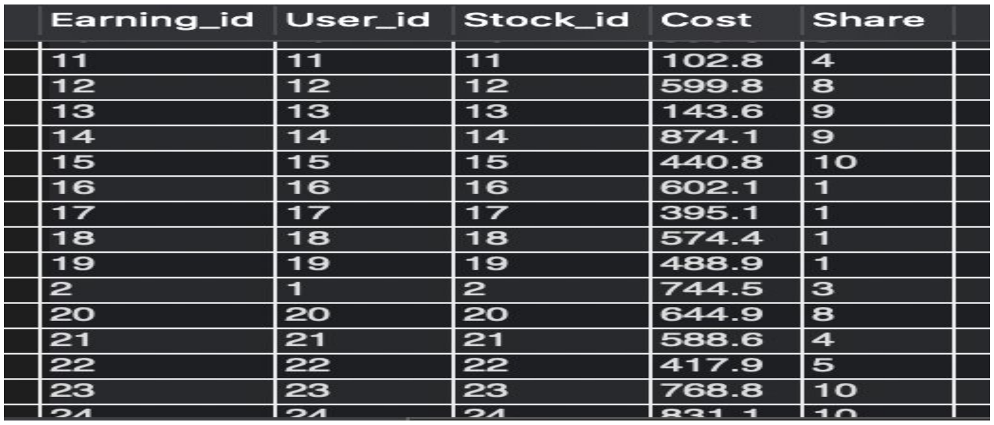
We can see that all the tables are in BCNF, and we see that Stock_id is the foreign key for most relations. There are no transitive dependencies in our design and it is in BCNF because for all functional dependencies, the left hand side ends up being a superkey for that relation.

Daily:



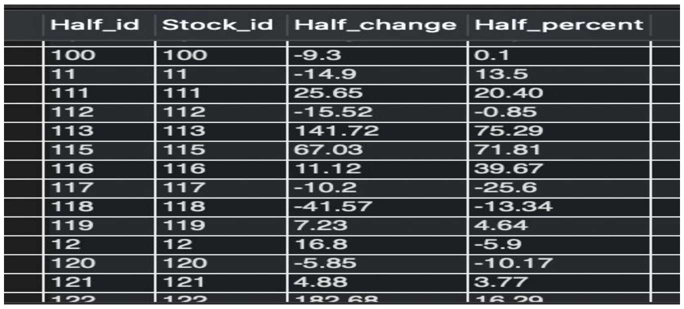
Daily_id	Stock_id	Open	Closing	High	Low	Volume	Change	Change_percent
115	115	167.05	168.85	159.61	161.64	19085100	-2.64	-1.61
116	116	39.46	39.76	39.07	39.41	33600100	0.42	1.08
117	117	29.42	28.88	29.49	29.49	22421400	-0.040	-0.14
118	118	273.90	275.45	272.05	274.96	18200	1.42	0.52
119	119	161.92	162.83	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	2.4700	2.5400	2.4300	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

Earnings:

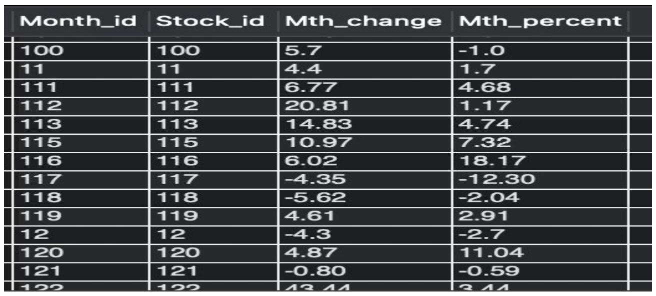


Earning_id	User_id	Stock_id	Cost	Share
11	11	11	102.8	4
12	12	12	599.8	8
13	13	13	143.6	9
14	14	14	874.1	9
15	15	15	440.8	10
16	16	16	602.1	1
17	17	17	395.1	1
18	18	18	574.4	1
19	19	19	488.9	1
2	1	2	744.5	3
20	20	20	644.9	8
21	21	21	588.6	4
22	22	22	417.9	5
23	23	23	768.8	10
24	24	24	831.1	10

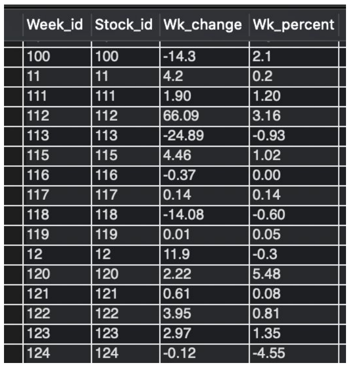
Week, Half_year, Month, Quarter, Year:



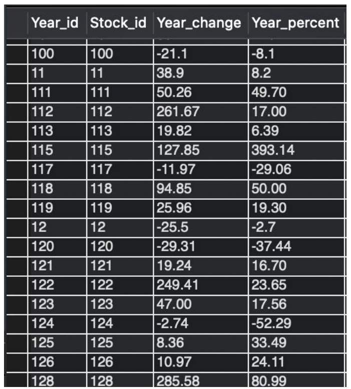
Half_id	Stock_id	Half_change	Half_percent
100	100	-9.3	0.1
11	11	-14.9	13.5
111	111	25.65	20.40
112	112	-15.52	-0.85
113	113	141.72	75.29
115	115	67.03	71.81
116	116	11.12	39.67
117	117	-10.2	-25.6
118	118	-41.57	-13.34
119	119	7.23	4.64
12	12	16.8	-5.9
120	120	-5.85	-10.17
121	121	4.88	3.77
122	122	182.68	16.29



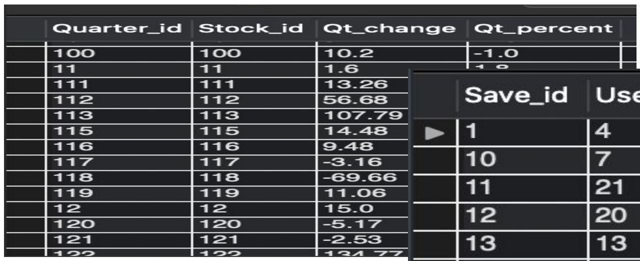
Month_id	Stock_id	Mth_change	Mth_percent
100	100	5.7	-1.0
11	11	4.4	1.7
111	111	6.77	4.68
112	112	20.81	1.17
113	113	14.83	4.74
115	115	10.97	7.32
116	116	6.02	18.17
117	117	-4.35	-12.30
118	118	-5.62	-2.04
119	119	4.61	2.91
12	12	-4.3	-2.7
120	120	4.87	11.04
121	121	-0.80	-0.59
122	122	43.44	3.44



Week_id	Stock_id	Wk_change	Wk_percent
100	100	-14.3	2.1
11	11	4.2	0.2
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
12	12	11.9	-0.3
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

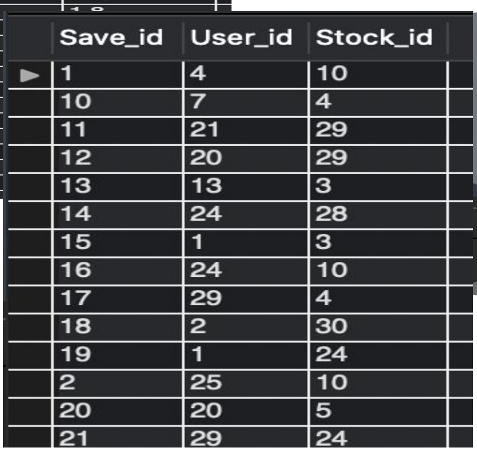


Year_id	Stock_id	Year_change	Year_percent
100	100	-21.1	-8.1
11	11	38.9	8.2
111	111	50.26	49.70
112	112	261.67	17.00
113	113	19.82	6.39
115	115	127.85	393.14
117	117	-11.97	-29.06
118	118	94.85	50.00
119	119	25.96	19.30
12	12	-25.5	-2.7
120	120	-29.31	-37.44
121	121	19.24	16.70
122	122	249.41	23.65
123	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

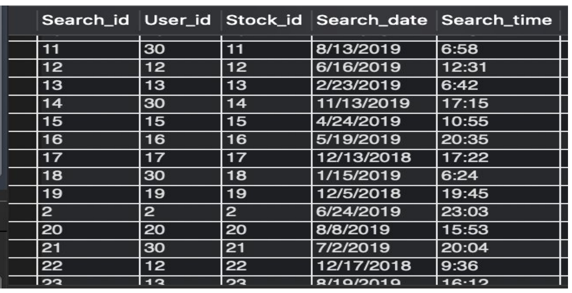


Quarter_id	Stock_id	Qt_change	Qt_percent
100	100	10.2	-1.0
11	11	1.6	
111	111	13.26	
112	112	56.68	
113	113	107.79	
115	115	14.48	
116	116	9.48	
117	117	-3.16	
118	118	-69.66	
119	119	11.06	
12	12	15.0	
120	120	-5.17	
121	121	-2.53	

<span style="color:red">**Save, Search:**</span>

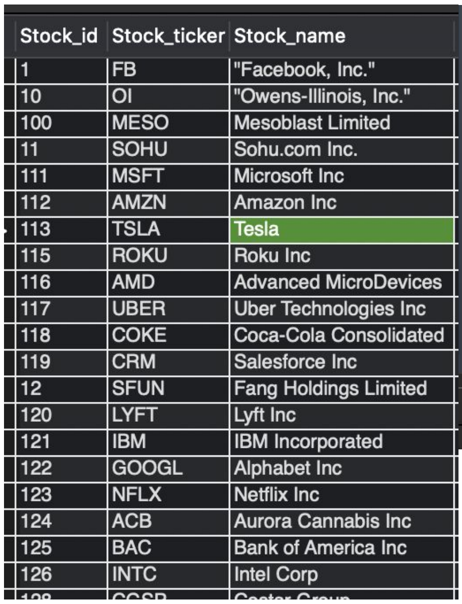


Save_id	User_id	Stock_id
1	4	10
10	7	4
11	21	29
12	20	29
13	13	3
14	24	28
15	1	3
16	24	10
17	29	4
18	2	30
19	1	24
2	25	10
20	20	5
21	29	24

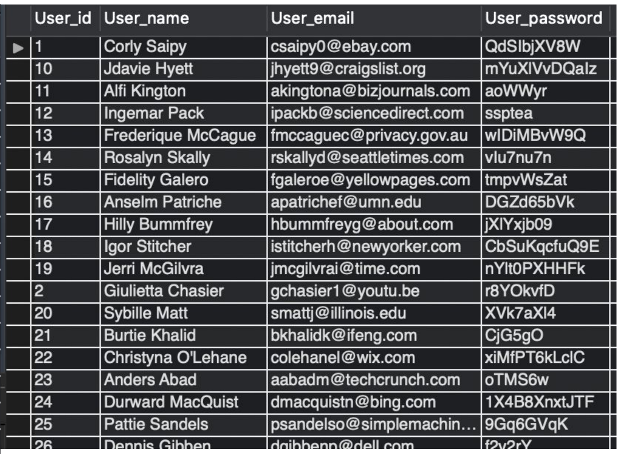


Search_id	User_id	Stock_id	Search_date	Search_time
11	30	11	8/13/2019	6:58
12	12	12	6/16/2019	12:31
13	13	13	2/23/2019	6:42
14	30	14	11/13/2019	17:15
15	15	15	4/24/2019	10:55
16	16	16	5/19/2019	20:35
17	17	17	12/13/2018	17:22
18	30	18	1/15/2019	6:24
19	19	19	12/5/2018	19:45
2	2	2	6/24/2019	23:03
20	20	20	8/8/2019	15:53
21	30	21	7/2/2019	20:04
22	12	22	12/17/2018	9:36

Stock, User :



Stock_id	Stock_ticker	Stock_name
1	FB	"Facebook, Inc."
10	OI	"Owens-Illinois, Inc."
100	MESO	Mesoblast Limited
11	SOHU	Sohu.com Inc.
111	MSFT	Microsoft Inc
112	AMZN	Amazon 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
12	SFUN	Fang Holdings Limited
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



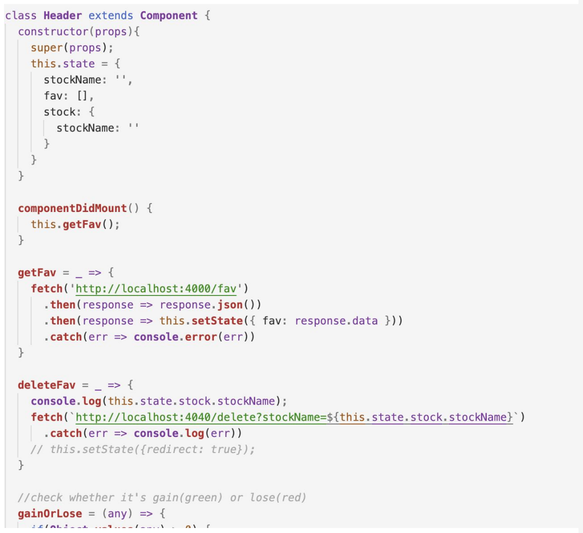
User_id	User_name	User_email	User_password
1	Corly Saipy	csaipy0@ebay.com	QdSIbjXV8W
10	Jdavie Hyett	jhyett9@craigslist.org	mYuXIVvDQaIz
11	Alfi Kington	akingtona@bizjournals.com	aoWWyr
12	Ingemar Pack	ipackb@sciencedirect.com	ssptea
13	Frederique McCague	fmccaguec@privacy.gov.au	wIDiMBvW9Q
14	Rosalyn Skally	rskallyd@seattletimes.com	vlu7nu7n
15	Fidelity Galero	fgaleroe@yellowpages.com	tmpvWsZat
16	Anselm Patriche	apatrichef@umn.edu	DGZd65bVk
17	Hilly Bummfrey	hbummfreyg@about.com	jXIYxjb09
18	Igor Stitcher	istitcherh@newyorker.com	CbSuKqcfuQ9E
19	Jerri McGilvra	jmcgilvrai@time.com	nYlt0PXHHFk
2	Giulietta Chasier	gchasier1@youtu.be	r8YOkvfD
20	Sybille Matt	smattj@illinois.edu	XVk7aXI4
21	Burtie Khalid	bkhalidk@ifeng.com	CjG5gO
22	Christyna O'Lehane	colehanel@wix.com	xiMfPT6kLclC
23	Anders Abad	aabadm@techcrunch.com	oTMS6w
24	Durward MacQuist	dmacquistn@bing.com	1X4B8XnxtJTF
25	Pattie Sandels	psandelso@simplemachin…	9Gq6GVqK

Implementation

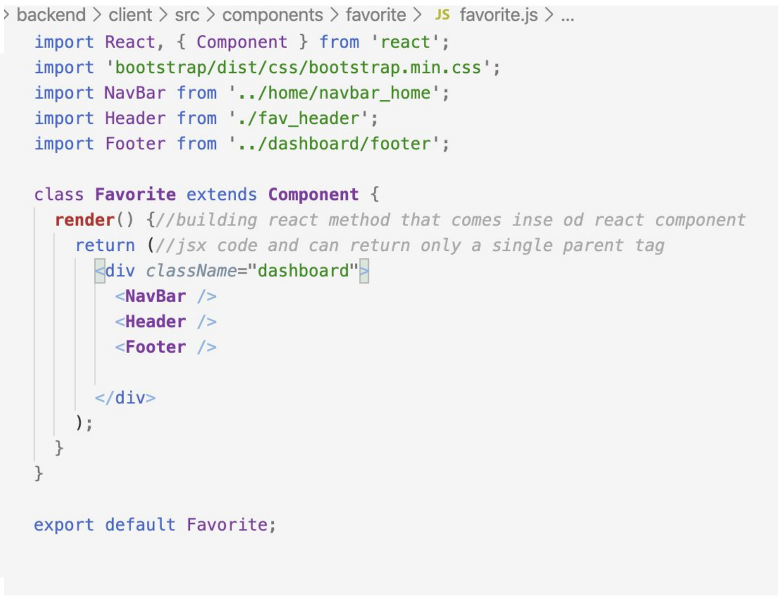
n Detail explanations of how your DB application system was implemented.
n Keep tracks of implementations from design.
n Identify the entities, attributes, dependences, relationships, constraints, etc. (show screenshots of corresponding tables, GUI, execution results, and so on.)
n Show functions/features associated with query, insertion, updating, and deletion operations. (Screenshots)
n Procedures (step by step) of how to set up and run your system

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 header.js file, and a footer.js file. The header.js files in each of the components handled the dynamic functionality.



```
class Header extends Component {
  constructor(props){
    super(props);
    this.state = {
      stockName: '',
      fav: [],
      stock: {
        stockName: ''
      }
    }
  }

  componentDidMount() {
    this.getFav();
  }

  getFav = _ => {
    fetch('http://localhost:4000/fav')
      .then(response => response.json())
      .then(response => this.setState({ fav: response.data }))
      .catch(err => console.error(err))
  }

  deleteFav = _ => {
    console.log(this.state.stock.stockName);
    fetch(`http://localhost:4040/delete?stockName=${this.state.stock.stockName}`)
      .catch(err => console.log(err))
    // this.setState({redirect: true});
  }

  //check whether it's gain(green) or lose(red)
  gainOrLose = (any) => {
```

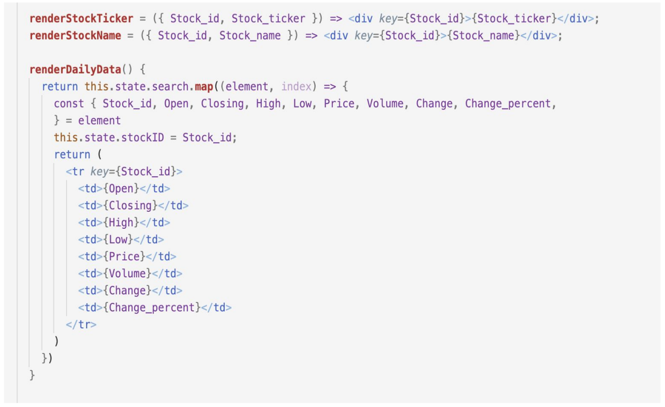


backend > client > src > components > favorite > favorite.js > ...

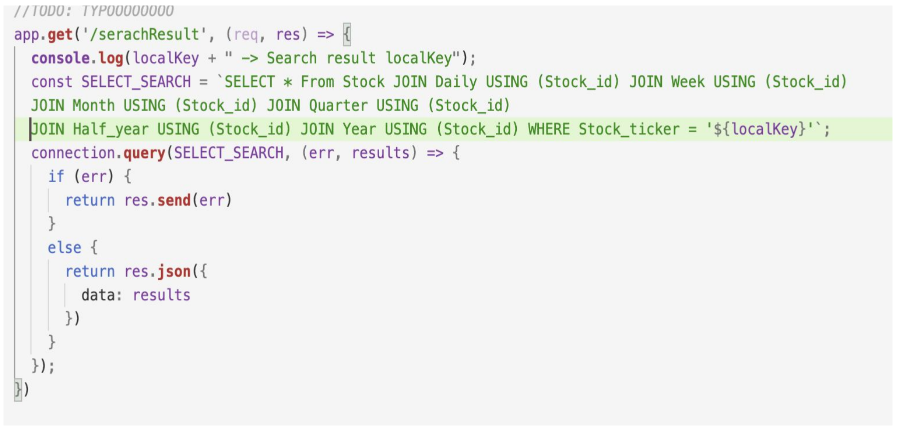
```
import React, { Component } from 'react';
import 'bootstrap/dist/css/bootstrap.min.css';
import NavBar from '../home/navbar_home';
import Header from './fav_header';
import Footer from '../dashboard/footer';

class Favorite extends Component {
  render() {//building react method that comes inse od react component
    return (//jsx code and can return only a single parent tag
      <div className="dashboard">
        <NavBar />
        <Header />
        <Footer />

      </div>
    );
  }
}

export default Favorite;
```

Our code was successful in
addressing all of our functional requirements. The first one being that a 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 AAPL. This functionality is handled in the backend using the following SQL query embedded within React.

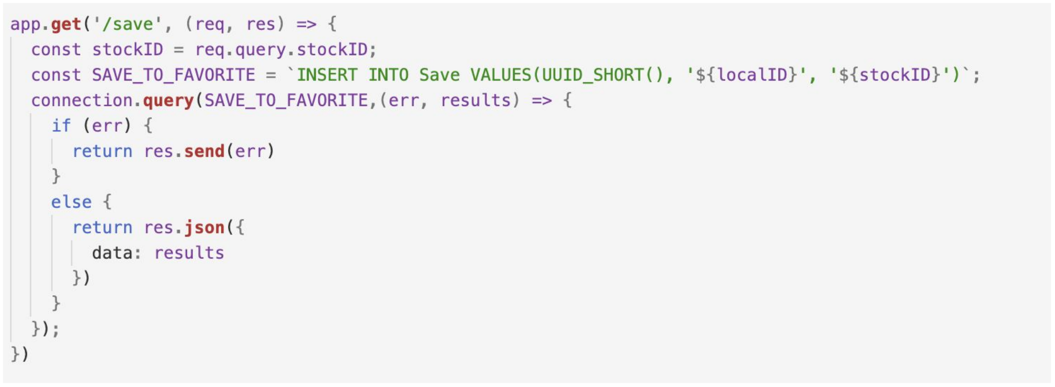


```
renderStockTicker = ({ Stock_id, Stock_ticker }) => <div key={Stock_id}>{Stock_ticker}</div>;
renderStockName = ({ Stock_id, Stock_name }) => <div key={Stock_id}>{Stock_name}</div>;

renderDailyData() {
  return this.state.search.map((element, index) => {
    const { Stock_id, Open, Closing, High, Low, Price, Volume, Change, Change_percent,
    } = element
    this.state.stockID = Stock_id;
    return (
      <tr key={Stock_id}>
        <td>{Open}</td>
        <td>{Closing}</td>
        <td>{High}</td>
        <td>{Low}</td>
        <td>{Price}</td>
        <td>{Volume}</td>
        <td>{Change}</td>
        <td>{Change_percent}</td>
      </tr>
    )
  })
}
```

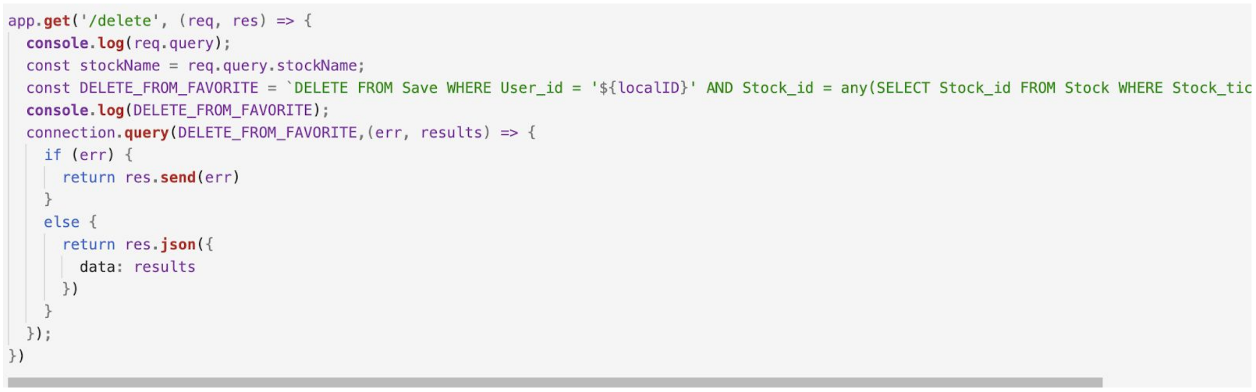


```
//TODO: TYPO0000000
app.get('/serachResult', (req, res) => {
  console.log(localKey + " -> Search result localKey");
  const SELECT_SEARCH = `SELECT * From Stock JOIN Daily USING (Stock_id) JOIN Week USING (Stock_id)
  JOIN Month USING (Stock_id) JOIN Quarter USING (Stock_id)
  JOIN Half_year USING (Stock_id) JOIN Year USING (Stock_id) WHERE Stock_ticker = '${localKey}'`;
  connection.query(SELECT_SEARCH, (err, results) => {
    if (err) {
      return res.send(err)
    }
    else {
      return res.json({
        data: results
      })
    }
  });
})
```

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

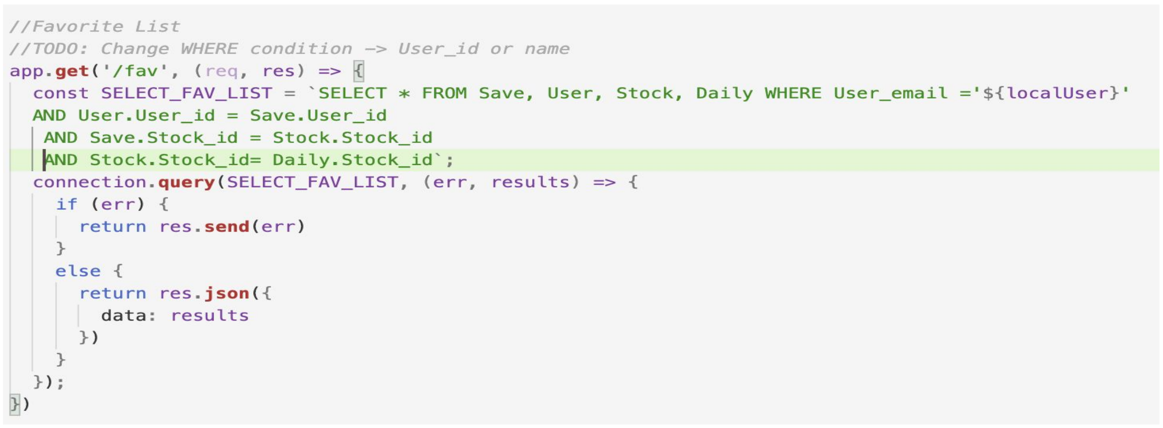


```
app.get('/save', (req, res) => {
  const stockID = req.query.stockID;
  const SAVE_TO_FAVORITE = `INSERT INTO Save VALUES(UUID_SHORT(), '${localID}', '${stockID}')`;
  connection.query(SAVE_TO_FAVORITE,(err, results) => {
    if (err) {
      return res.send(err)
    }
    else {
      return res.json({
        data: results
      })
    }
  });
})
```



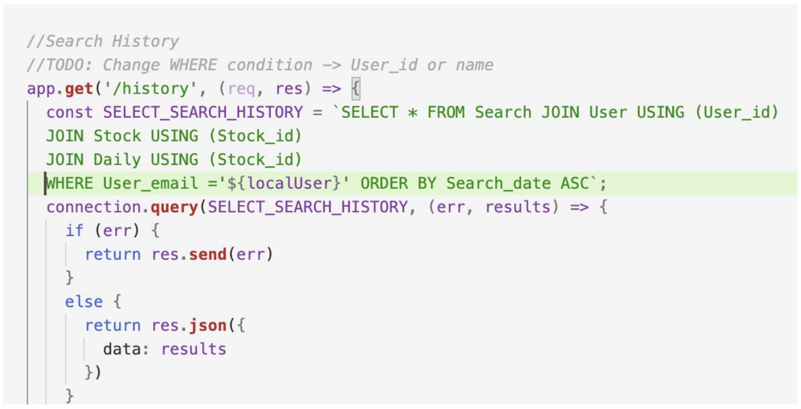
```
app.get('/delete', (req, res) => {
  console.log(req.query);
  const stockName = req.query.stockName;
  const DELETE_FROM_FAVORITE = `DELETE FROM Save WHERE User_id = '${localID}' AND Stock_id = any(SELECT Stock_id FROM Stock WHERE Stock_tic
  console.log(DELETE_FROM_FAVORITE);
  connection.query(DELETE_FROM_FAVORITE,(err, results) => {
    if (err) {
      return res.send(err)
    }
    else {
      return res.json({
        data: results
      })
    }
  });
})
```

User’s 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



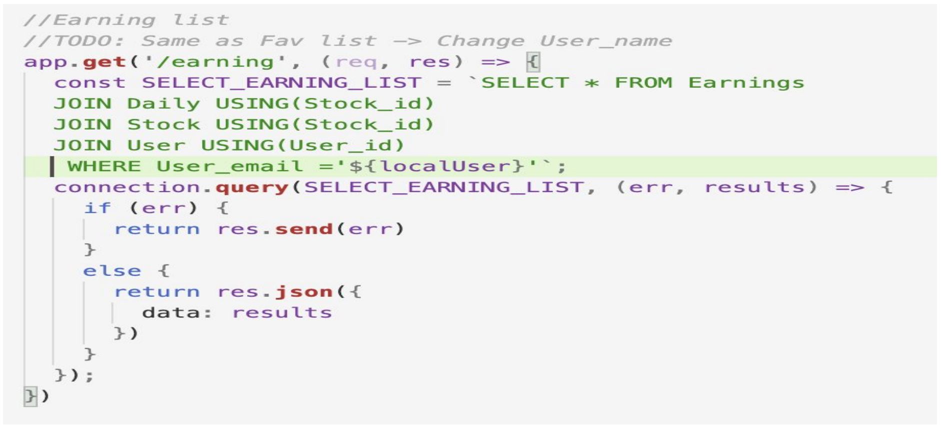
```
//Favorite List
//TODO: Change WHERE condition -> User_id or name
app.get('/fav', (req, res) => {
  const SELECT_FAV_LIST = `SELECT * FROM Save, User, Stock, Daily WHERE User_email ='${localUser}'
  AND User.User_id = Save.User_id
   AND Save.Stock_id = Stock.Stock_id
   AND Stock.Stock_id= Daily.Stock_id`;
  connection.query(SELECT_FAV_LIST, (err, results) => {
    if (err) {
      return res.send(err)
    }
    else {
      return res.json({
        data: results
      })
    }
  });
})
```

Another functional requirement was that User’s 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.



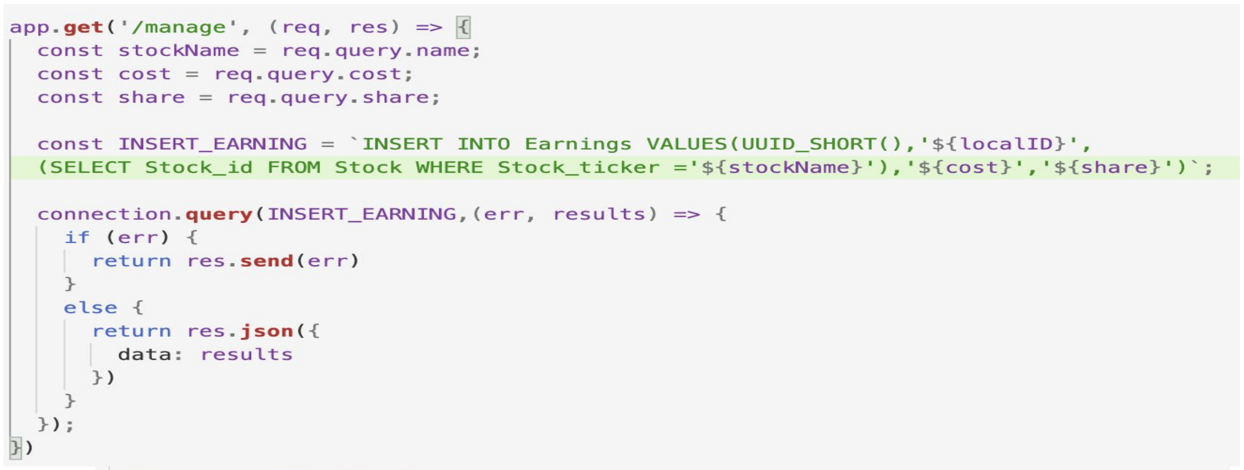
```
//Search History
//TODO: Change WHERE condition -> User_id or name
app.get('/history', (req, res) => {
  const SELECT_SEARCH_HISTORY = `SELECT * FROM Search JOIN User USING (User_id)
  JOIN Stock USING (Stock_id)
  JOIN Daily USING (Stock_id)
  WHERE User_email ='${localUser}' ORDER BY Search_date ASC`;
  connection.query(SELECT_SEARCH_HISTORY, (err, results) => {
    if (err) {
      return res.send(err)
    }
    else {
      return res.json({
        data: results
      })
    }
```

The most difficult functional requirement to implement was the record earnings function, which allowed the user to input the amount 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

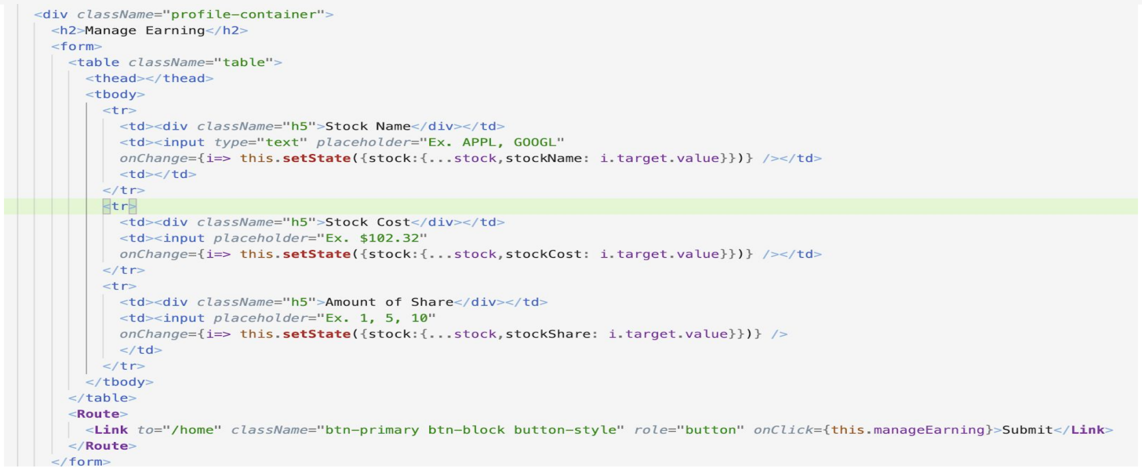


```
//Earning list
//TODO: Same as Fav list -> Change User_name
app.get('/earning', (req, res) => {
  const SELECT_EARNING_LIST = `SELECT * FROM Earnings
  JOIN Daily USING(Stock_id)
  JOIN Stock USING(Stock_id)
  JOIN User USING(User_id)
  WHERE User_email ='${localUser}'`;
  connection.query(SELECT_EARNING_LIST, (err, results) => {
    if (err) {
      return res.send(err)
    }
    else {
      return res.json({
        data: results
      })
    }
  });
})
```

input their shares and costs and that information is stored in the backend in the Earnings table.



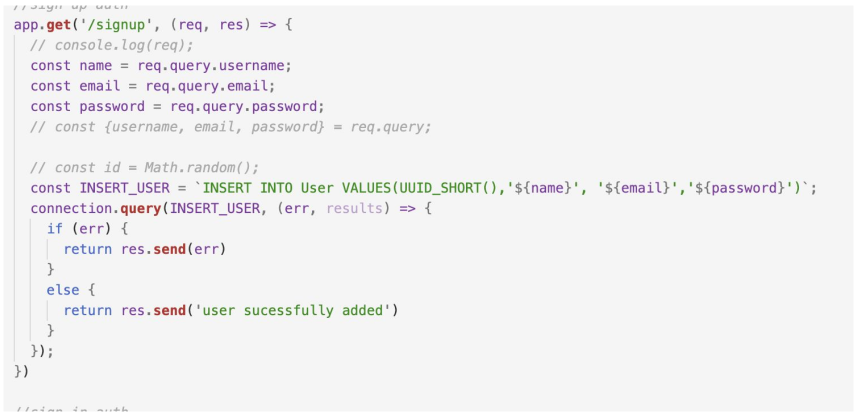
```
app.get('/manage', (req, res) => {
  const stockName = req.query.name;
  const cost = req.query.cost;
  const share = req.query.share;

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

  connection.query(INSERT_EARNING,(err, results) => {
    if (err) {
      return res.send(err)
    }
    else {
      return res.json({
        data: results
      })
    }
  });
})
```



```
<div className="profile-container">
  <h2>Manage Earning</h2>
  <form>
    <table className="table">
      <thead></thead>
      <tbody>
        <tr>
          <td><div className="h5">Stock Name</div></td>
          <td><input type="text" placeholder="Ex. APPL, GOOGL"
          onChange={i=> this.setState({stock:{...stock,stockName: i.target.value}})} /></td>
          <td></td>
        </tr>
        <tr>
          <td><div className="h5">Stock Cost</div></td>
          <td><input placeholder="Ex. $102.32"
          onChange={i=> this.setState({stock:{...stock,stockCost: i.target.value}})} /></td>
        </tr>
        <tr>
          <td><div className="h5">Amount of Share</div></td>
          <td><input placeholder="Ex. 1, 5, 10"
          onChange={i=> this.setState({stock:{...stock,stockShare: i.target.value}})} />
          </td>
        </tr>
      </tbody>
    </table>
    <Route>
      <Link to="/home" className="btn-primary btn-block button-style" role="button" onClick={this.manageEarning}>Submit</Link>
    </Route>
  </form>
```

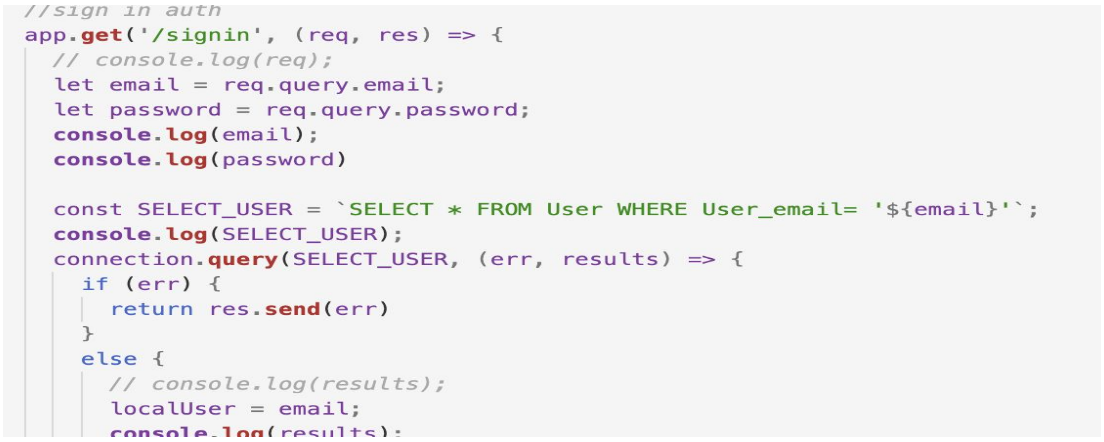
The ability for users to add to their earnings was made possible by the Earnings table, and the entities involved are User and Stock entities.

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 login. 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 which would display the User name and email.



```
app.get('/signup', (req, res) => {
  // console.log(req);
  const name = req.query.username;
  const email = req.query.email;
  const password = req.query.password;
  // const {username, email, password} = req.query;

  // const id = Math.random();
  const INSERT_USER = `INSERT INTO User VALUES(UUID_SHORT(),'${name}', '${email}','${password}')`;
  connection.query(INSERT_USER, (err, results) => {
    if (err) {
      return res.send(err)
    }
    else {
      return res.send('user sucessfully added')
    }
  });
})
```

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



```
//sign in auth
app.get('/signin', (req, res) => {
  // console.log(req);
  let email = req.query.email;
  let password = req.query.password;
  console.log(email);
  console.log(password)

  const SELECT_USER = `SELECT * FROM User WHERE User_email= '${email}'`;
  console.log(SELECT_USER);
  connection.query(SELECT_USER, (err, results) => {
    if (err) {
      return res.send(err)
    }
    else {
      // console.log(results);
      localUser = email;
      console.log(results);
```

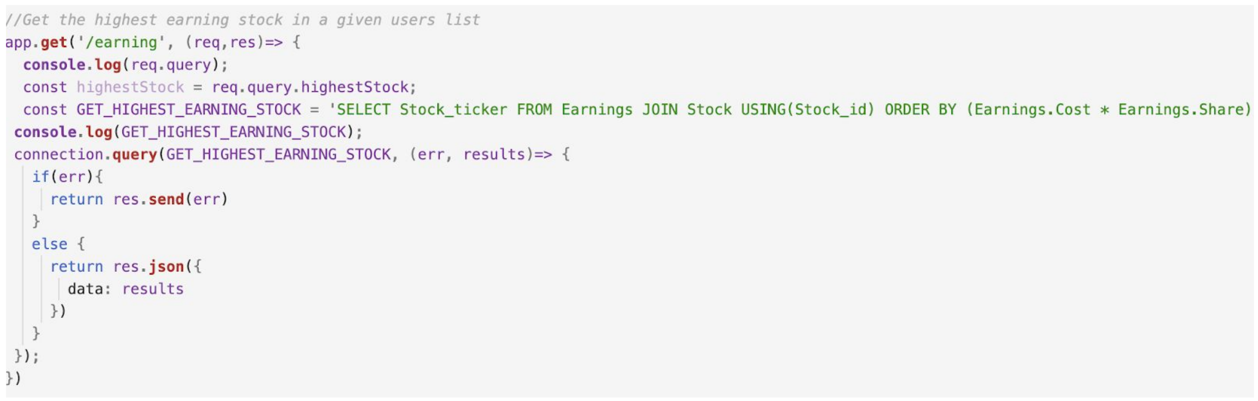
Project Conclusion

n Statements from each team member about Lesson Learned from this DB project.

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 alot 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
     })
   }
 });
})
```

Yang Li

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.

En-Ping Shih (Team Leader)

n Future improvement of your DB application.