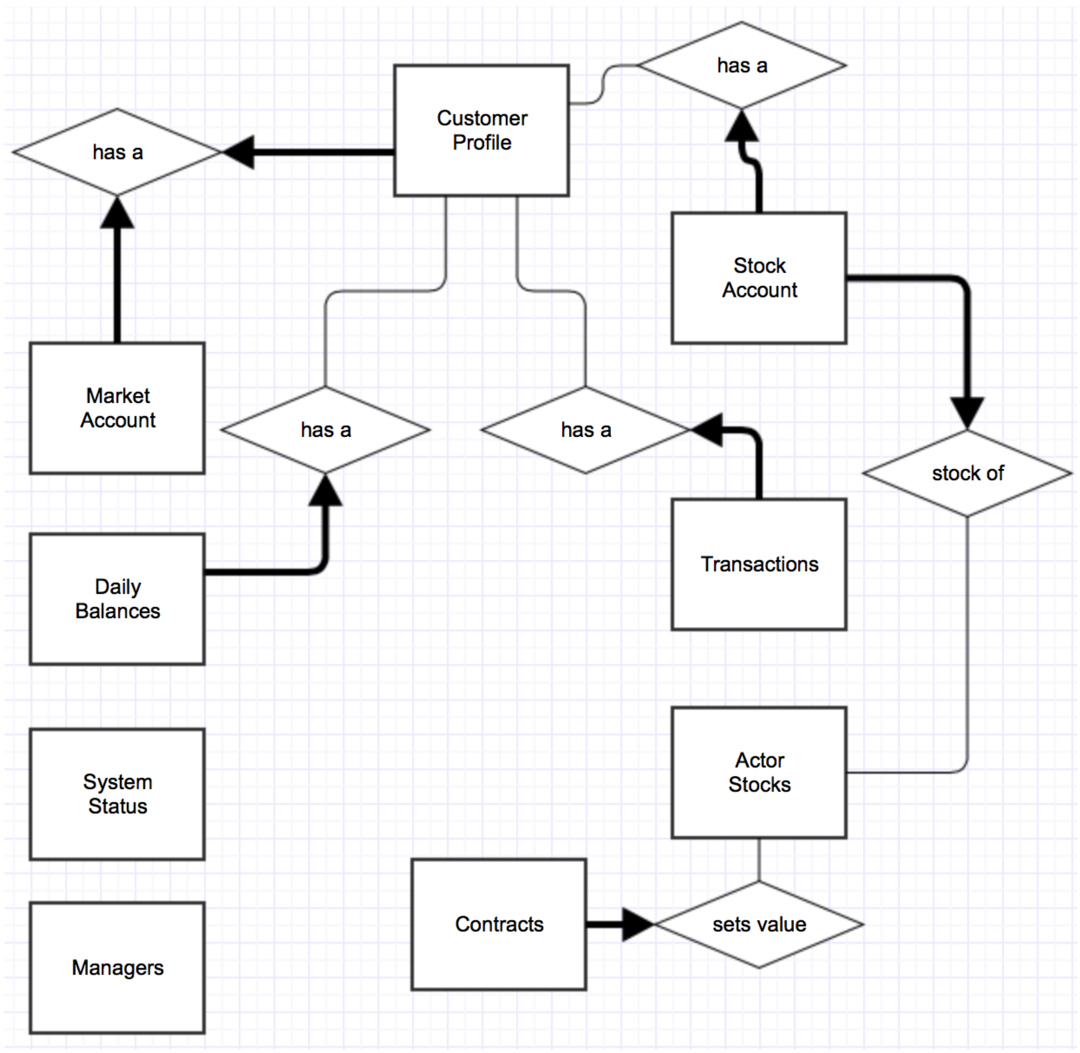
Project Report
Databases

Danish Vaid
Jake Can

ER Diagram



Note: Attributes not shown to make the diagram easier to see
For attributes see conceptual schema

Conceptual Schema

- actor_stocks(<u>stock_sym: string</u>, current_price: decimal, actor_name: string, dob: string)
- contracts(<u>c_id: integer</u>, stock_sym: string, movie_title: string, role: string, year: integer, value: decimal)
- customer_profiles(name: string, username: string, password: string, address: string, state: string, phone: string, email: string, <u>tax_id: integer</u>, ssn: integer)
- daily_balances(<u>b_id: integer</u>, tax_id: integer, balance: decimal, date: string, month: integer, day: integer)

- managers(name: string, username: string, password: string, address: string, state: string, phone: string, email: string, <u>tax_id: integer</u>, ssn: integer)
- market_accounts(<u>tax_id: integer</u>, balance: decimal)
- stock_accounts(<u>stock_acc_id: integer</u>, tax_id: integer, stock_sym: string, num_shares: decimal, date: string, type: string, price: decimal, earnings: decimal)
- system_status(market_open: boolean, date: string)
- transactions(<u>txn_id: integer</u>, tax_id: integer, date: string, month: integer, txn_type: string, txn_details: string)

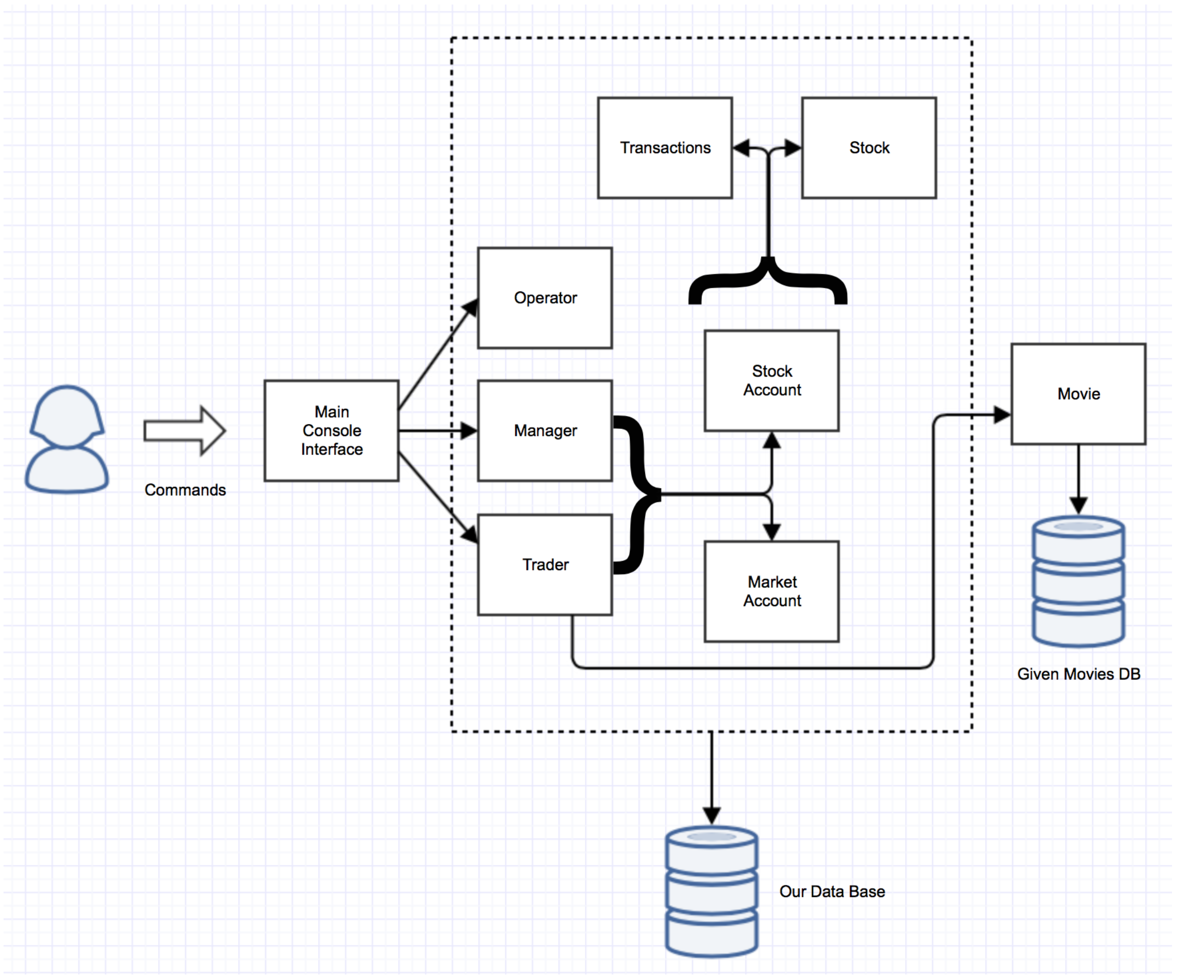
Integrity Constraints (IC)

- Customer must have a market accounts
- Market Account initializes with $1000 deposit
- Market Account balance must always be greater than 0
- Transactions must relate to a customer (tax_id)

IC Violation

The way we have dealt with IC violation is to simply tell the use the command they are trying to run is invalid, and tell them why the issue is there. I.e. if an user's "Market Account" balance is going to fall below 0, our program will them the transaction they are making is invalid as they have insufficient funds.

System Architecture



Class Breakdown

User

This class takes user input from the command line. It includes the signing in process and taking commands for the type of user (which includes manager, trader, and operator).

checkUserExists():

- `String query = String.format("SELECT * FROM %s WHERE username = '%s'", tableType` `username);`
  // tableType is based on user type: manager or trader. Query for checking that user does not exist for registration.

showUserInfo():

- `query = String.format("SELECT * FROM %s WHERE tax_id = '%d'", tableType, taxID);`
  // Verify Tax ID is unique by checking that none exist to avoid duplicate Tax ID's.
- `query = String.format("INSERT INTO %s VALUES ('%s', '%s', '%s', '%s', '%s', '%s', '%s', '%d', '%s');", tableType, name, username, password, address, state, phoneNumber, email, taxID, ssn);`
  // Stores user info that was obtained from the command line

getUserFromTable():

- `String query = String.format("SELECT * FROM %s WHERE username = '%s' AND password = '%s'", tableType, username, password);`
  //Check that a user exists for logging in.

<u>Trader</u>

The type of user that participates in the simulation. Has functions that follow suit with the expected commands that it can call. This includes buy, sell, deposit, and withdraw and some others. Most of these functions call on other functions in other classes that are relevant to the entity. This allows for code and queries to be isolated to sections and classes of where the relevance would be expected.

showActorProfile():

- `String query = String.format("SELECT * FROM actor_stocks LEFT JOIN contracts ON actor_stocks.stock_sym = contracts.stock_sym WHERE actor_stocks.stock_sym = '%s';", stockSymbol);`
  // Joins together information from actor_stocks table and contracts table on stock symbol

<u>Manager</u>

Includes all the commands that a manager can make. This class makes various SQL queries itself, depending on the command that is called.

listActiveCustomers():

- `String sub_query = String.format("(SELECT tax_id, SUM(ABS(num_shares)) AS traded FROM stock_accounts WHERE SUBSTRING_INDEX(date, '/', 1) = %s GROUP BY tax_id)", Integer.toString(CommandUI.currentDate.getMonth()));`
  `String query = String.format("SELECT name, username, C.tax_id FROM customer_profiles AS C, (SELECT tax_id FROM %s AS tab WHERE tab.traded >= 1000) AS Temp WHERE C.tax_id = Temp.tax_id ", sub_query);`
  // Nested query. Inside query grabs users and the sum of the shares they have traded in the current month. Middle query filters those users to those who traded more than 1000. Outside query lists the customer information for those who passed through the filter.

generateDTER():

- `String query = "SELECT tax_id, SUM(CAST(ints.interest as DECIMAL(10,2))) AS interest FROM (SELECT tax_id, SUBSTRING_INDEX(txn_details, ' ', -1) AS interest FROM transactions WHERE txn_type = 'interest') AS ints GROUP BY tax_id;";`
  // Gets sum of all interest earnings for each users for this DTER.
- `query = "SELECT tax_id, name, state, SUM(earnings) AS earnings FROM stock _accounts NATURAL JOIN customer_profiles GROUP BY tax_id;";`
  // Gets customer information and earnings from trading stock. Java code adds the two and prints the users who have earned more than $10,000 after the addition.

generateCustomerReport():

- `String query = String.format("SELECT tax_id, name, email, balance FROM customer _profiles NATURAL JOIN market_accounts WHERE tax_id = %d;", customerTaxID);`
  // Gets customer information and balance.
- `query = String.format("SELECT tax_id, stock_sym, SUM(num_shares) AS shares, SUM(earnings) AS earnings FROM stock_accounts WHERE tax_id = %d GROUP BY tax_id, stock_sym;", customerTaxID);`
  // Get all stock accounts for the given customer's Tax ID.

deleteTransactions():

- `String query = "DELETE FROM transactions";`
  // Deletes all rows in transactions table.z

Operator

A system admin type user. Has functions like opening and closing the market, to manage the functionality of the system. Can also set price and date to change the conditions of the simulation. Functions in this class have been stubbed out and explained since the code would be repetitive and long.

setSystem():

- `String query = "SELECT * FROM system_status;";`
  // Gets market status and date from system_status table

generateTables():

-
  ```
  String[] tables = new String[9];
  tables[0] = "CREATE TABLE customer_profiles ( "
  + "name CHAR(30) NOT NULL, "
  + "username CHAR(30) NOT NULL, "
  + "password CHAR(30) NOT NULL, "
  + "address CHAR(40) NOT NULL, "
  ```

```
+ "state CHAR(2) NOT NULL, "
+ "phone CHAR(15) NOT NULL, "
+ "email CHAR(30) NOT NULL, "
+ "tax_id INT UNSIGNED NOT NULL, "
+ "ssn CHAR(11) NOT NULL, "
+ "PRIMARY KEY(tax_id) "
+ ");";
```

// Similar structure to create the rest of our SQL Tables

deleteDB():

```
- for(String tableName :  tableNames)
  JDBC.statement.executeUpdate("DROP TABLE " + tableName + ";");
```

// Goes through a list of all table names (in relational/key order) and deletes all table

populateDataFromFile():
- // Parses the given data and populates our database with it.

recordOpenMarket():
- // Set DB market status as open.

recordCloseMarket():
- // Set DB market status as close.

recordCurrentDate():
- // Records the current date in the DB.

<u>MarketAccount</u>

Each function in this class relates to the possible transactions that are relevant to the market account like buying and selling, which will affect the customer's monetary balance.

createAccount():

```
- String query = String.format("INSERT INTO market_accounts (tax_id, balance)
  VALUES (%d, %d);", tax_id, 1000);
```

// Deposits the initial 1000 and creates row for new customer/Tax ID.

deposit():

```
- String query = String.format("UPDATE market_accounts SET balance = balance
  + %.2f WHERE tax_id = %d;", amount, User.currentTaxID);
```

// Updates balance to include the deposited amount.

withdraw():

- `String query = String.format("UPDATE market_accounts SET balance = %.2f WHERE tax_id = %d;", newBalance, User.currentTaxID);`
  // Updates balance to include the withdrawn amount.

accureInterestOnAllAccounts():

- `ResultSet sql_tax_ids = JDBC.statement.executeQuery("SELECT tax_id FROM market_accounts");`
  // Gets all customers' Tax ID's.
- `query = String.format("SELECT AVG(balance) FROM daily_balances WHERE tax_id = %d AND month = %d;", tax_id, CommandUI.currentDate.getMonth());`
  // Averages daily balances for each customer.
- `query = String.format("UPDATE market_accounts SET balance = balance + %.2f WHERE tax_id = %d;", interest, tax_id);`
  // Updates balance to include the interest amount.

getBalance():

- `String query = String.format("SELECT balance FROM market_accounts WHERE tax_id = %d;", taxID);`
  // Gets balance for specified user.

recordAllDailyBalances():

- `String query = "SELECT * FROM market_accounts";`
  // Gets Tax ID and balance from market_accounts table.
- `String baseQuery = "INSERT INTO daily_balances (tax_id, balance, date, month, day) ";`
  // Prepare for multiple similar insert queries.
- `valueQuery = String.format("VALUES (%d, %.2f, '%s', %d, %d); \n", result .getInt("tax_id"), result.getDouble("balance"), currentDate.toString(), currentDate.getMonth(), currentDate.getDay());`
  // Gives values of each required attribute.

<u>StockAccount</u>

This class has transactional functions that are related to the StockAccount, like buying or selling stock.

buy():

-
  ```
  String insert_part = "INSERT INTO stock_accounts (tax_id, stock_sym, num_shares,
  date, type, price, earnings)";
  String values_part = String.format("VALUES (%d, '%s', %.3f, '%s', 'buy',
  %.2f, 0.0)", User.currentTaxID, stockSymbol, numShares, CommandUI.currentDate
  ```

```
.toString(), Stock.getStockPrice(stockSymbol));
```
// Insert a row for a buy transaction with the necessary information.

sell():

- `String query = String.format("SELECT SUM(num_shares) FROM stock_accounts WHERE tax_id = %d AND stock_sym = '%s' AND price = %.2f;", User.currentTaxID, stockSymbol, buyPrice);`
  // Get sum of remaining shares of a certain type at a certain buy price.
- `String insert_part = "INSERT INTO stock_accounts (tax_id, stock_sym, num _shares, date, type, price, earnings)";`
  `String values_part = String.format("VALUES (%d, '%s', %.3f, '%s', 'sell', %.2f, %.2f)", User.currentTaxID, stockSymbol, (-1 * numShares), CommandUI. currentDate, buyPrice, earnings);`
  // Insert a row for a sell transaction with the necessary information.

Movie

Specific functions for getting movie information. Allows for getting top movies and movie reviews, as desired.

getMovieInfo():

- `String query = String.format("SELECT * FROM Movies WHERE title = '%s'", movieName);`
  // Get movie information for a given movie title.

getTopMovie():

- `String query = String.format("SELECT * FROM Movies WHERE production_year >= %d AND production_year <= %d AND rating >= 5", beginYear, endYear);`
  // Get a list of all movies within the time period with a rating of 5.

getMovieReviews():

- `String query = String.format("SELECT author, review FROM Movies JOIN Reviews ON Movies.id = Reviews.movie_id WHERE Movies.title = '%s';", movieName);`
  // Grab the reviews for desired movie

Stock

Contains functions that are relevant to stocks, like getting and setting stock prices.

getStockPrice():

- `String query = String.format("SELECT current_price FROM actor_stocks WHERE stock_sym = '%s'", stockSymbol);`
  // Get current price of a given stock

setStockPrice():

- `String query = String.format("UPDATE actor_stocks SET current_price = %.2f WHERE stock_sym = '%s';", newPrice, stockSymbol);`
  // Set stock with a given stock symbol and the desired price.

Transactions

Manages all functions that include the transaction table, including the manager command that requests information on customer transactions.

generateMonthlyStatement():

- `String query = String.format("SELECT name, email FROM customer_profiles WHERE tax_id = %d;", taxID);`
  // Get name and email of specified customer's Tax ID.

generateMonthlyStatement():

- `String query = String.format("SELECT name, email FROM customer_profiles WHERE tax_id = %d;", taxID);`
  // Get name and email of specified customer's Tax ID.
- `sub_query = String.format("SELECT MIN(day) FROM daily_balances WHERE month = %d AND tax_id = %d", CommandUI.currentDate.getMonth(), taxID);`
  `query = String.format("SELECT balance FROM daily_balances WHERE month = %d AND day = (%s) AND tax_id = %d", CommandUI.currentDate.getMonth(), sub_query, taxID);`
  // Nested query. Inside query gets the first recorded day of the month of daily balances. Outside query gets the respective balance for the specified day of the month as the initial balance.
- `sub_query = String.format("SELECT MAX(day) FROM daily_balances WHERE month = %d AND tax_id = %d", CommandUI.currentDate.getMonth(), taxID);`
  `query = String.format("SELECT balance FROM daily_balances WHERE month = %d AND day = (%s) AND tax_id = %d", CommandUI.currentDate.getMonth(), sub_query, taxID);`
  // Nested query. Similar as above, but gets the last day of the month of daily balances for the final balance.
- `query = String.format("SELECT COUNT(*) FROM transactions WHERE tax_id = %d AND month = %d AND(txn_type = 'buy' OR txn_type = 'sell');", taxID, CommandUI.`
  // Gets number of transactions to calculate commissions paid.

showTransactionHistory():

- `String query = String.format("SELECT * FROM transactions WHERE tax_id = %d", tax_id);`
  // Get all transactions for a specified customer Tax ID.

addInterestRecord():

- `String query = String.format("INSERT INTO transactions(tax_id, date, month, txn_type, txn_details) VALUES (%d, '%s', %d, 'interest', '%s');", tax_id, CommandUI.currentDate.toString(), CommandUI.currentDate.getMonth(), notes);`
  // Add a transaction record for adding interest.

getInterestGiven():
- `String query = String.format("SELECT txn_details FROM transactions WHERE tax_id = %d AND month = %d", tax_id, CommandUI.currentDate.getMonth());`
  // Get the interest given to a customer in the current month.

insertRecord():
- `String query = String.format("INSERT INTO transactions (tax_id, date, month, txn_type, txn_details) VALUES (%d, '%s', %d, '%s', '%s');", User.currentTaxID, CommandUI.currentDate.toString(), CommandUI.currentDate.getMonth(), type, note);`
  // Query to insert a transaction into the table, given the type of the transactions, and details of the transactions.

<u>CommandUI</u>

This class takes user input from the command line. It includes the signing in process and taking commands for the type of user (which includes manager, trader, and operator).

<u>Date</u>

Custom class for dealing with the date and relative time of transactions between other transactions of other or same users. Includes functions like adding days to the Date object, finding the difference between two dates, and parsing a date formatted string.

<u>JDBC</u>

Class to create a connection to the DB. Allows for the java program to make SQL queries. Uses a JSON parser to look for an external JSON file to read in the DB credentials to avoid hard-coding credentials into our source code.