**A Lightweight Financial Data Management and Trading Simulation System**

Wei Mu, Dahai Liu

Contents

[Project Summary 2](#_Toc8386414)

[Physical Model and Data Dictionary 3](#_Toc8386415)

[Entity Relationship Diagram 8](#_Toc8386416)

[Database System Infrastructure 8](#_Toc8386417)

[SQL Script for Creating Tables and Inserting Sample Data 9](#_Toc8386418)

[SQL Statements for Answering Major Data Questions. 11](#_Toc8386419)

[Interface Implementation 16](#_Toc8386420)

[Appendix 22](#_Toc8386421)

# Project Summary

Financial data are heavily used in the academic learning environment. Examples include financial statement data, historical stock prices and stock market indices, and derivatives information such as option prices. Whitman School of Management provides multiple sources of financial data, including Bloomberg Terminal and S&P Capital IQ, for professor and students. Although these tools are comprehensive and advanced, they are considered to be overkill for certain academic purposes. They require large amount of subscription fee or certain external device (Bloomberg Terminal), and they do not provide aid for some repetitive operations used in a class, which sometimes take hours of manual work. Yahoo! Finance suited the need in most cases. However, Yahoo! stopped providing Yahoo Stock API for historical price data since 2017.

The object of this project is to design and implement a lightweight financial data management and trading simulation system. The basic functions of the system are as follows:

1. Allow users to input their orders and simulate trading behavior using historical financial data. Users can also place orders and wait until new data is available in the future. In this scenario, it works like a “virtual stock exchange”.
2. automatically pull, scrape, and store real time and historical financial data from open-to-public sources provided on the internet. Sources include Yahoo! Finance and IEX Finance. Financial data includes real time stock prices, historical prices, basic statistics, financials, and options. The web-scraping part for Yahoo! Finance is based on an existing Excel tool written by one of the team members. The tool is currently in-use by the professor of Financial Modeling class.
3. Show the users’ positions at end and calculate returns using simulated order history.
4. provide interface for academic users to easily summarize data and generate Excel spreadsheets.

The project is implemented using SQL server Express as backend and Python Dash as the frontend which interacts with user as a web application. Some other Python packages and frameworks are also used in this project, such as “Plotly”, “SQLAlchemy”, “Flask”, “Flask-SQLAlchemy”, and “fix-yahoo-finance”.

The primary users are students and professors in Whitman School of Management. As all the data come from open-to-public sources, all users should at least have read-only access to all the financial data in the system, so that they may conveniently draw out information from the database in case the interface of the system does not provide certain functions for temporary academic purposes.

# Physical Model and Data Dictionary

1. Users.

The table representing this entity records the users of the system. This is a core table as it is referenced by Order, Position, and Portfolio. Because ‘user’ is a reserved keyword in SQL Server, the name of the table is changed to ‘Users’ in implementation.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entity Name: Users | Attribute Name | Field Type | Nullable | Foreign Key Constraints | Domain Constraints | Description |
| Primary Key | UserID | VARCHAR(20) | NOT NULL |  |  | Username, e.g. ‘wmu100’ |
|  | UserPassword | VARCHAR(100) | NOT NULL |  |  |  |
|  | UserFirstName | VARCHAR(30) |  |  |  |  |
|  | UserLastName | VARCHAR(30) |  |  |  |  |
|  | UserType | VARCHAR(20) | NOT NULL |  |  | Can be but not limited to: ‘student’, ‘admin’, ‘professor’ |

2. Ticker

This entity keeps track of stock tickers (e.g. “AAPL”, which is the stock symbol for “Apple, Inc.”) in the system. This is a core table as it is referenced by Orders, Position and HistoricalPrice. The name ‘Orders’ is used for the Order entity because ‘order’ is a reserved keyword in SQL Server. In implementation, this table is automatically filled using information from Yahoo.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entity Name: Ticker | Attribute Name | Field Type | Nullable | Foreign Key Constraints | Domain Constraints | Description |
| Primary Key | TickerID | VARCHAR(20) | NOT NULL |  |  | e.g. “AAPL”, which is the stock symbol for “Apple, Inc.” |
|  | TickerShortName | VARCHAR(80) | NOT NULL |  |  | e.g. "Apple, Inc." |
|  | TickerType | VARCHAR(20) | NOT NULL |  |  | e.g. “EQUITY” |

3. Portfolio

This entity records the current cash balance of the user before and after each transaction. This is a reference table because it contains foreign key to Users table. In implementation, the records in this table is automatically tracked and maintained by the frontend web trading system. In current version, the system assumes 100,000 as default cash balance for each user, and no margin trading. Features such as buy-on-margin will be implemented in future development.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entity Name: Portfolio | Attribute Name | Field Type | Nullable | Foreign Key Constraints | Domain Constraints | Description |
| Primary Key | UserID | VARCHAR(20) | NOT NULL | REFERENCES Users(UserID);  ON DELETE CASCADE |  | The user name, e.g. “wmu100” |
|  | Cash | FLOAT | DEFAULT 100000 |  | >=0 | Current cash balance for a user. |

4. Position

This table keeps track of the current holdings of users. It summarizes how many shares a user has in total. For example, if a user buy 200 shares of AAPL on 2010/01/31 and 300 shares AAPL on 2010/01/31, the number of shares in this entity becomes 500. The system assumes an initial holding of 0 shares for each ticker. This entity has a composite primary key and is a reference table because it contains foreign key to Users and Ticker table. The system assumes no partial share trading.

| Entity Name: Position | Attribute Name | Field Type | Nullable | Foreign Key Constraints | Domain Constraints | Description |
| --- | --- | --- | --- | --- | --- | --- |
| Primary Key | UserID | VARCHAR(20) | NOT NULL | REFERENCES Users(UserID) ON DELETE CASCADE |  | The user name, e.g. “wmu100” |
| Primary Key | TickerID | VARCHAR(20) | NOT NULL | REFERENCES Ticker(TickerID) |  | e.g. “AAPL”, which is the stock symbol for “Apple, Inc.” |
| Primary Key | Shares | INT | DEFAULT 0 |  | >= 0 | See description above |
|  | PositionType | VARCHAR(10) | NOT NULL |  | ‘Buy’ or ‘Short’ |  |

5. Orders

Because ‘Order’ is a reserved keyword in SQL server, this table is named as ‘Orders’. This table stores all order history of users. This system allows trading simulation using historical data, so that users can test certain trading strategies. This system uses historical closing price of stock tickers, so orders are simulated to be placed at the end of the date given by the user. In implementation, when a user places order on the web page, the frontend web server system set up a transaction, insert the record ,and modifies other tables accordingly, including Portfolio and Position. The underlying database controlling framework, SQLAlchemy ORM (Object Relational Mapper) has an internal mechanism which prevents Race Condition to some extent.

This table is a reference table because it contains foreign key to Users and Ticker table.

Explanation of Limit order and Stop order:

In real life, when a user Limit Buy 100 shares of AAPL at $200, the Buy order is sent to the broker but not executed until the price reaches $200 or above. When a user Stop Sell 100 shares of AAPL at $180, the Sell order is sent to the broker but not executed until the price reaches $180 or below. The system allows simulation of Limit order and Stop order. When a user places a Limit Buy 100 shares AAPL at $200 order, the ‘OrderType’ field is filled in with ‘Buy’, ‘Shares‘ is 100, ‘PriceType’ is ‘Limit’ and ‘Price’ is 200. Note that this ‘Price’ field is not a historical price, but is part of Limit or Stop order. In contrast, Market is executed immediately in trading day, and Price field is unnecessary in a Market order. Market order is the most common type of order.

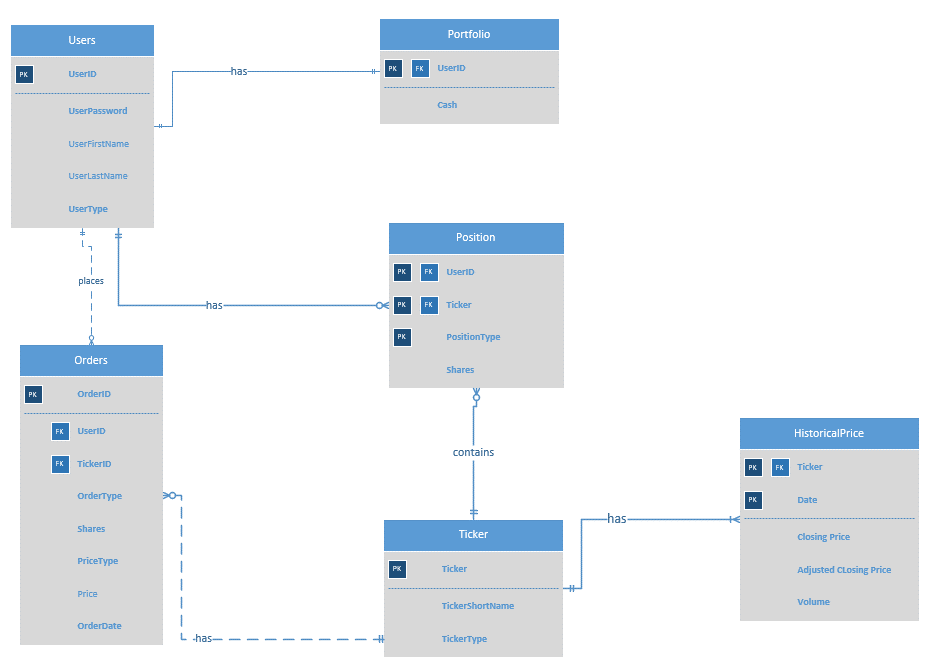
| Entity Name: Orders | Attribute Name | Field Type | Nullable | Foreign Key Constraints | Domain Constraints | Description |
| --- | --- | --- | --- | --- | --- | --- |
| Primary Key | OrderID | INT IDENTITY | IDENTITY |  |  |  |
|  | UserID | VARCHAR(20) | NOT NULL | REFERENCES Users(UserID) ON DELETE CASCADE |  |  |
|  | TickerID | VARCHAR(20) | NOT NULL | REFERENCES Ticker(TickerID) |  |  |
|  | OrderType | VARCHAR(20) | NOT NULL |  | ‘Buy’, ‘Sell’, ‘Short’, or ‘Buy to cover’ | ‘Buy to cover’ means buy to cover short positions. |
|  | Shares | INT | NOT NULL |  | >=0 |  |
|  | PriceType | VARCHAR(10) | NOT NULL |  | ‘Market’, ‘Limit’, or ‘Stop’ | See description above |
|  | Price | FLOAT |  |  |  | See description above |
|  | OrderDate | DATETIME | NOT NULL |  |  |  |

6. HistoricalPrice

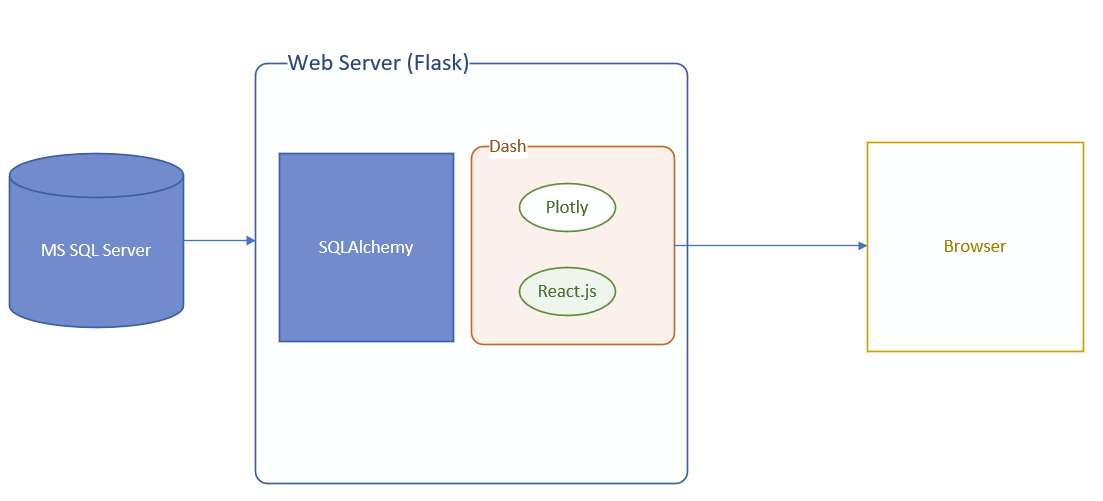
This table is used to store the historical price info from Yahoo! Finance. Particularly, investors care more about closing price and adjusted closing price. Adjusted closing price is the price adjusted to stock-splits and dividends, so it is more helpful when a stock-split or a reverse stock split happened in the given range of price history. In implementation, this table stores the automatic scraping result after the admin user initiates a refresh data request. Before storing the result, a transaction is created, and tickers are appended to the Ticker table as needed, so that it does not violate referential integrity.

| Entity Name: Position | Attribute Name | Field Type | Nullable | Foreign Key Constraints | Domain Constraints | Description |
| --- | --- | --- | --- | --- | --- | --- |
| Primary Key | TickerID | VARCHAR(20) | NOT NULL | REFERENCES Ticker(TickerID) |  |  |
| Primary Key | \_Date | DATETIME | NOT NULL |  |  | Date is a reserved keyword so it is renamed as \_Date. |
|  | ClosingPrice | FLOAT | NOT NULL |  |  | See description above |
|  | Adj\_Close | FLOAT | NOT NULL |  |  | See description above |
|  | Volume | BIGINT | NOT NULL |  |  | Trading volume in a given trading day |

# Entity Relationship Diagram



# Database System Infrastructure



This graph is an overview of the System Infrastructure. The database is provided by Microsoft SQL Server. The middleware interacting with both database system and the Web server (Flask) is SQLAlchemy. The user interface is implemented in Python Dash framework. Python Dash runs Flask as the backend web server, but Dash also allows multi-process and scaling, so it can be made into a production server. Dash is built on top of React.js, and supports its Plotly packages for visualization. The advantage of Dash is that it allows programmers to build data science web application in pure Python, without the knowledge of JavaScript and CSS. (However, in our system we wrote some CSS to fix some rendering issues.) In addition, it does allow integration of React components.

Compared to MS Access, the Python Dash + SQLAlchemy framework is harder to develop, but it allows programmer or data scientist to build web application for production and visualization. Dash is also suitable for professionals with deadlines, because coding difficulty is lower in Python, and many useful packages were written in Python. The web application can also be packed up with Docker ,or pushed to cloud platform.

The automatic web scraping from Yahoo is implemented using Python package ‘fix-yahoo-finance’. No need to manually insert real data.

# SQL Script for Creating Tables and Inserting Sample Data

-- Create tables

-- Can be created automatically by SQLAlchemy but this course requires SQL.

CREATE TABLE Users

(

UserID VARCHAR(20) PRIMARY KEY,

UserPassword VARCHAR(100) NOT NULL,

UserFirstName VARCHAR(30),

UserLastName VARCHAR(30),

UserType VARCHAR(20) NOT NULL,

);

CREATE TABLE Ticker

(

TickerID VARCHAR(20) PRIMARY KEY,

TickerShortName VARCHAR(80) NOT NULL,

TickerType VARCHAR(20) NOT NULL

);

CREATE TABLE Portfolio

(

UserID VARCHAR(20) PRIMARY KEY,

Cash float DEFAULT 1000000,

CONSTRAINT FK\_User FOREIGN KEY (UserID) REFERENCES Users(UserID) ON DELETE CASCADE,

CONSTRAINT chk\_cash CHECK (Cash>=0),

);

CREATE TABLE Position

(

UserID VARCHAR(20) NOT NULL ,

TickerID VARCHAR(20) NOT NULL ,

Shares INT DEFAULT 0,

PositionType VARCHAR(10) NOT NULL,

CONSTRAINT FK\_Position\_User FOREIGN KEY (UserID) REFERENCES Users(UserID) ON DELETE CASCADE,

CONSTRAINT FK\_Position\_Ticker FOREIGN KEY (TickerID) REFERENCES Ticker(TickerID),

CONSTRAINT chk\_shares CHECK (Shares>=0),

CONSTRAINT pk\_position PRIMARY KEY (UserId, TickerID, PositionType),

CONSTRAINT chk\_position CHECK (PositionType='Buy' OR PositionType='Short')

);

CREATE TABLE Orders

(

OrderID INT IDENTITY PRIMARY KEY ,

UserID VARCHAR(20) NOT NULL FOREIGN KEY REFERENCES Users(UserID) ON DELETE CASCADE,

TickerID VARCHAR(20) NOT NULL FOREIGN KEY REFERENCES Ticker(TickerID),

OrderType VARCHAR(20) NOT NULL,

Shares INT NOT NULL ,

PriceType VARCHAR(10) NOT NULL,

Price float,

OrderDate DATETIME NOT NULL,

CONSTRAINT chk\_ordershares CHECK (Shares>0),

CONSTRAINT chk\_pricetype CHECK (PriceType='Market' OR PriceType='Limit' OR PriceType = 'Stop'),

CONSTRAINT chk\_ordertype CHECK (OrderType IN ('Buy', 'Sell', 'Short', 'Buy to cover'))

);

CREATE TABLE HistoricalPrice

(

TickerID VARCHAR(20) NOT NULL,

\_Date DATETIME NOT NULL,

ClosingPrice float NOT NULL,

Adj\_Close float NOT NULL,

Volume bigint NOT NULL

CONSTRAINT pk\_history PRIMARY KEY (TickerID, \_Date),

CONSTRAINT FK\_History\_Ticker FOREIGN KEY (TickerID) REFERENCES Ticker(TickerID)

);

-- Insert admin, professor, and student users.

-- require at least one user.

-- Can be created automatically by SQLAlchemy but I did not implement it.

INSERT INTO Users

VALUES

('wmu100', 'mw2010a@gmail.com', 'Wei', 'Mu', 'admin'),

('ProfHoyos', 'hoyos@abc.com', 'H', 'Hoyos', 'professor'),

('ytong', 'ytong@def.com', 'Yuntong', 'Liu', 'student'),

('admin', 'root123', 'admin', 'user', 'admin');

-- Insert some random tickers

-- This is not required because the web app program automatically fixes it.

INSERT INTO Ticker

VALUES

('AAPL', 'Apple, Inc.', 'Stock'),

('IBM', 'International Business Machines', 'Stock'),

('MCD', 'McDonald"s Corporation', 'Stock');

-- Insert initial portfolio for one of the users

-- This is not required because the web app program automatically fixes it.

INSERT INTO Portfolio

VALUES

('wmu100', 1000000);

Note: Some of the insert statements above are not required because it is maintained by our main Dash program. There are tables with no insert statements above because they are automatically maintained by our Dash program**.**

**The full Python code of our project is available on Github:**

<https://github.com/mason1900/DashAppProject>

**Currently the project has also been deployed to ~~<deleted from Github>~~ and is accessible via a web browser. Please log in as ~~<deleted from Github>~~ with password ~~<deleted from Github>~~**

# SQL Statements for Answering Major Data Questions.

Business questions:

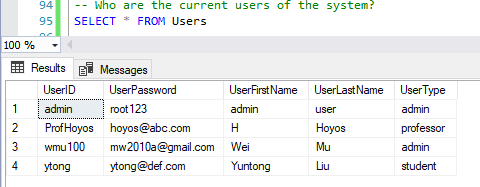
1. Who are the current users of the system?
2. How many students have been registered in this system?
3. Show the price trend, 42-day and 252-day moving average of a stock (e.g. “AAPL”) in the last 10 years.
4. Stock simulation: what is the return if User A bought 500 (as an example) shares of a stock (e.g. AAPL) at the closing price at a given date (e.g. 01/15/2010), 300 the same stock at another day (e.g. 01/15/2013), 400 shares of another stock (e.g. IBM) at another date(e.g. 01/15/2015) and sell all of the above at given date (e.g. 05/08/2019)?
5. Assume that users (students) have starting cash of $10000000. After a certain strategy has been applied, what is the remaining cash available at certain trading day?
6. What is the overall gain in cash if the user accepted the trading strategy in business question 4 (i.e. she placed order exactly as in question 4)?

These questions are important because it is the basic functionality of the system. The system is mainly designed to fetch real financial data and let users apply their own strategies on the stocks to perform simulations.

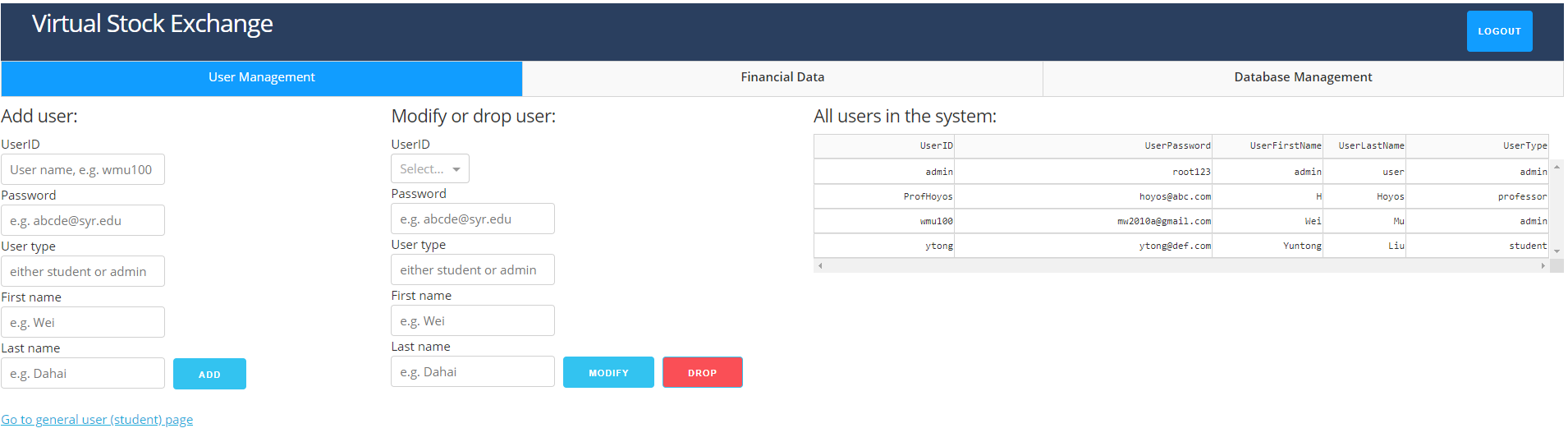
1. Who are the current users of the system?

SELECT \* FROM Users

Screenshot:



Screenshot of the web app showes the same result: (please zoom in)

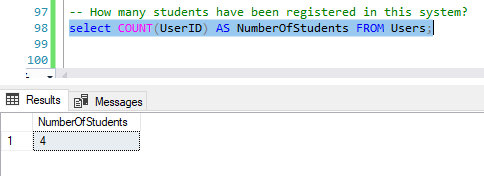


1. How many students have been registered in this system?

When there are many students in the system, this information will be helpful.

select COUNT(UserID) AS NumberOfStudents FROM Users;

Screenshot:



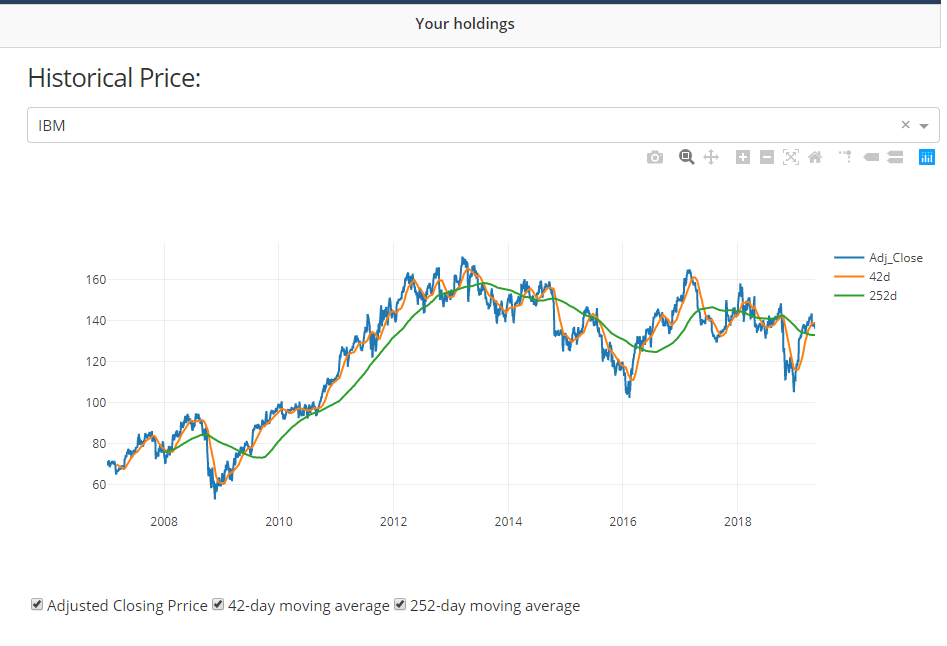
1. Show the price trend, 42-day and 252-day moving average of a stock (e.g. “IBM”) in the last 10 years.

This is done through the web application. SQLAlchemy queries the SQL Server when the following line of Python code is executed. Then it returns the results to a callback. The query code is SQLAlchemy ORM which wraps around the raw SQL statement.

Code: (Please turn off MS Word Spellchecking to have a better look)

result = db.session.query(HistoricalPrice.\_Date, HistoricalPrice.Adj\_Close).filter\_by(TickerID=ticker).all()  
if result is not None and len(result) > 0:  
 df = pd.DataFrame(result, columns=['Date', 'Adj\_Close'])  
 df['42d'] = df['Adj\_Close'].rolling(center=False, window=42).mean()  
 df['252d'] = df['Adj\_Close'].rolling(center=False, window=252).mean()  
 data = []  
 for temp in values:  
 data = data + [go.Scatter(x=df.Date, y=df[temp], name =temp)]  
 # data = [go.Scatter(  
 # x=df.Date,  
 # y=df['Adj\_Close'])]  
 return {"data": data,  
 "layout": go.Layout()}  
else:  
 return {}

Screenshot:



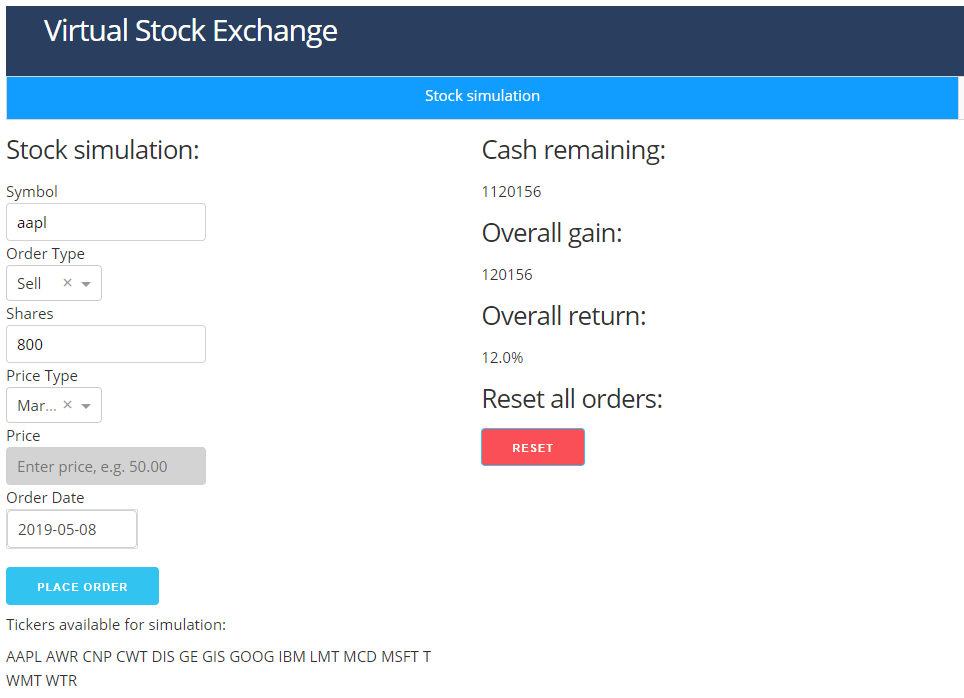
1. Stock simulation: what is the return if User A bought 500 (as an example) shares of a stock (e.g. AAPL) at the closing price at a given date (e.g. 01/15/2010), 300 the same stock at another day (e.g. 01/15/2013), 400 shares of another stock (e.g. IBM) at another date(e.g. 01/15/2015) and sell all of the above at given date (e.g. 05/08/2019)?

This question is hard to answer by pure raw SQL, but it is implemented in the code of our web application. Our Dash program manages the position and portfolio when user places an order, and it generates result to the web page.

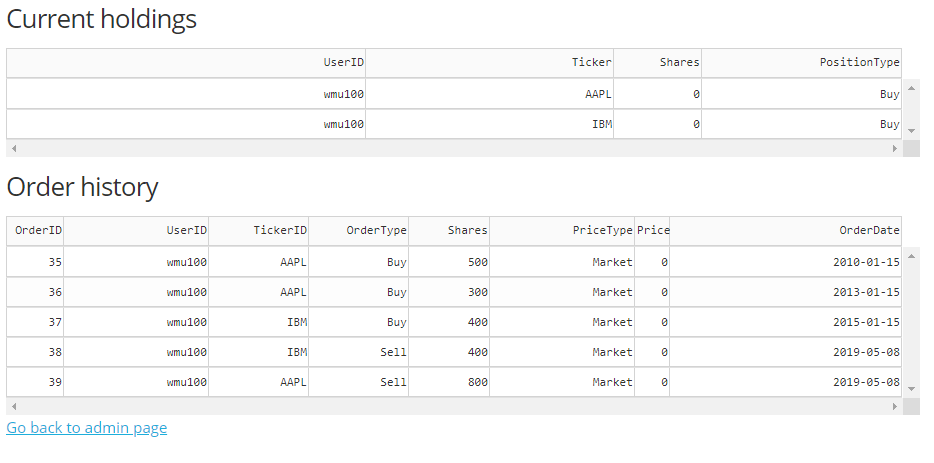
The code which is executed when user places a Buy or Sell order: : (Please turn off MS Word Spellchecking to have a better look)

def add\_order(\_order):# insert order  
 db.session.add(\_order)  
 # get corresponding price  
 price\_instance = db.session.query(HistoricalPrice).filter\_by(TickerID=\_order.TickerID, \_Date=\_order.OrderDate).first()  
 if price\_instance is None:  
 db.session.rollback()  
 msg = 'no corresponding price in db'  
 print(msg)  
 return False, msg  
 price = price\_instance.ClosingPrice  
  
 # Sell: check Position Shares >= Order shares  
 if \_order.OrderType == 'Sell':  
 p\_instance = db.session.query(Position.Shares).filter\_by(  
 UserID=\_order.UserID, TickerID=\_order.TickerID, PositionType='Buy').first()  
 if p\_instance is None or (p\_instance.Shares < int(\_order.Shares)):  
 msg = 'Selling more shares than available'  
 print(msg)  
 return False, msg  
  
 # get or set portfolio  
 portfolio = get\_or\_create(db.session, Portfolio, UserID=\_order.UserID)  
 if \_order.OrderType == 'Buy':  
 portfolio.Cash = portfolio.Cash - float(\_order.Shares) \* price  
 elif \_order.OrderType == 'Sell':  
 portfolio.Cash = portfolio.Cash + float(\_order.Shares) \* price  
 # get or set position  
 position = get\_or\_create(db.session, Position,  
 UserID=\_order.UserID, TickerID=\_order.TickerID, PositionType='Buy')  
 if \_order.OrderType == 'Buy':  
 position.Shares = position.Shares + int(\_order.Shares)  
 elif \_order.OrderType == 'Sell':  
 position.Shares = position.Shares - int(\_order.Shares)  
 # clean up  
 db.session.commit()  
 del portfolio  
 del position  
 return True, ''

Screenshot of result:



Another screenshot:



1. Assume that users (students) have starting cash of $10000000. After a certain strategy has been applied, what is the remaining cash available at certain trading day?
2. What is the overall gain in cash if the user accepted the trading strategy in business question 4 (i.e. she placed order exactly as in question 4)?

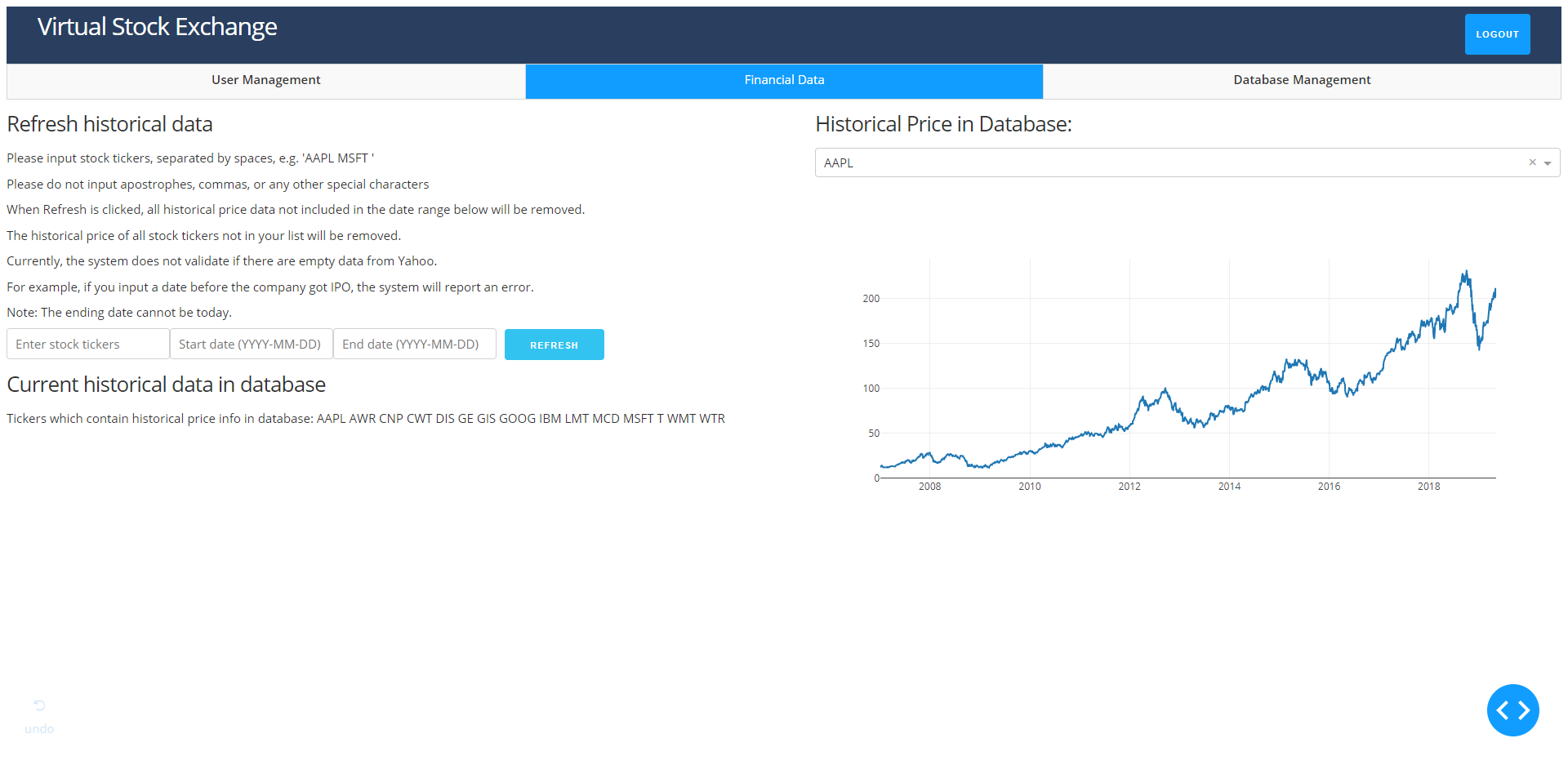
Question 5 and 6 is the same as Question 4. The code and results are all in Question 4.

# Interface Implementation

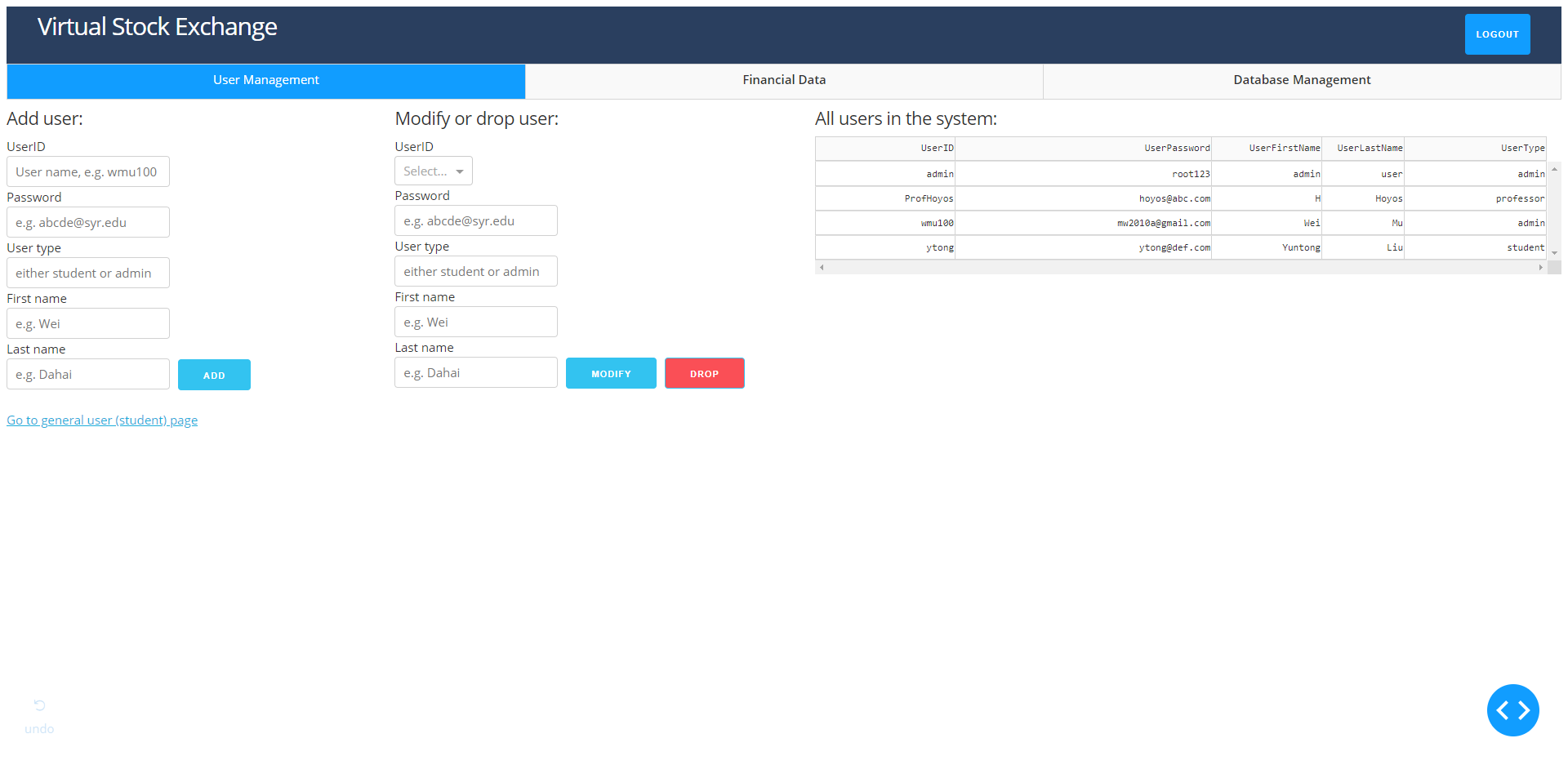
The general design of the system is as follows:

First, the administrator or professor login and input a list of tickers as the pool. By clicking Refresh, the system automatically fetch the financial data and store them in the database (which is HistoricalPrice table):

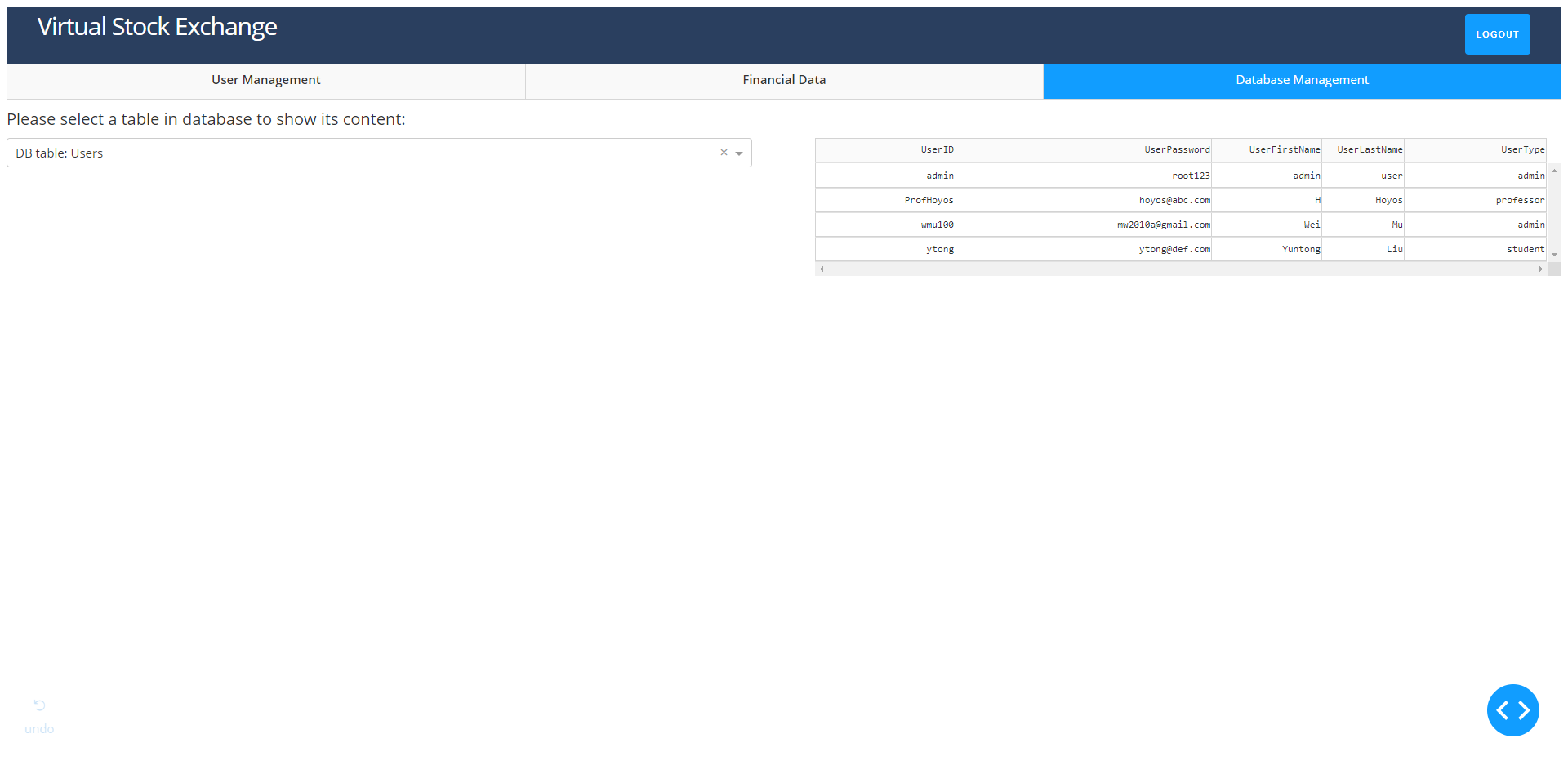
(Please zoom in to see the details)



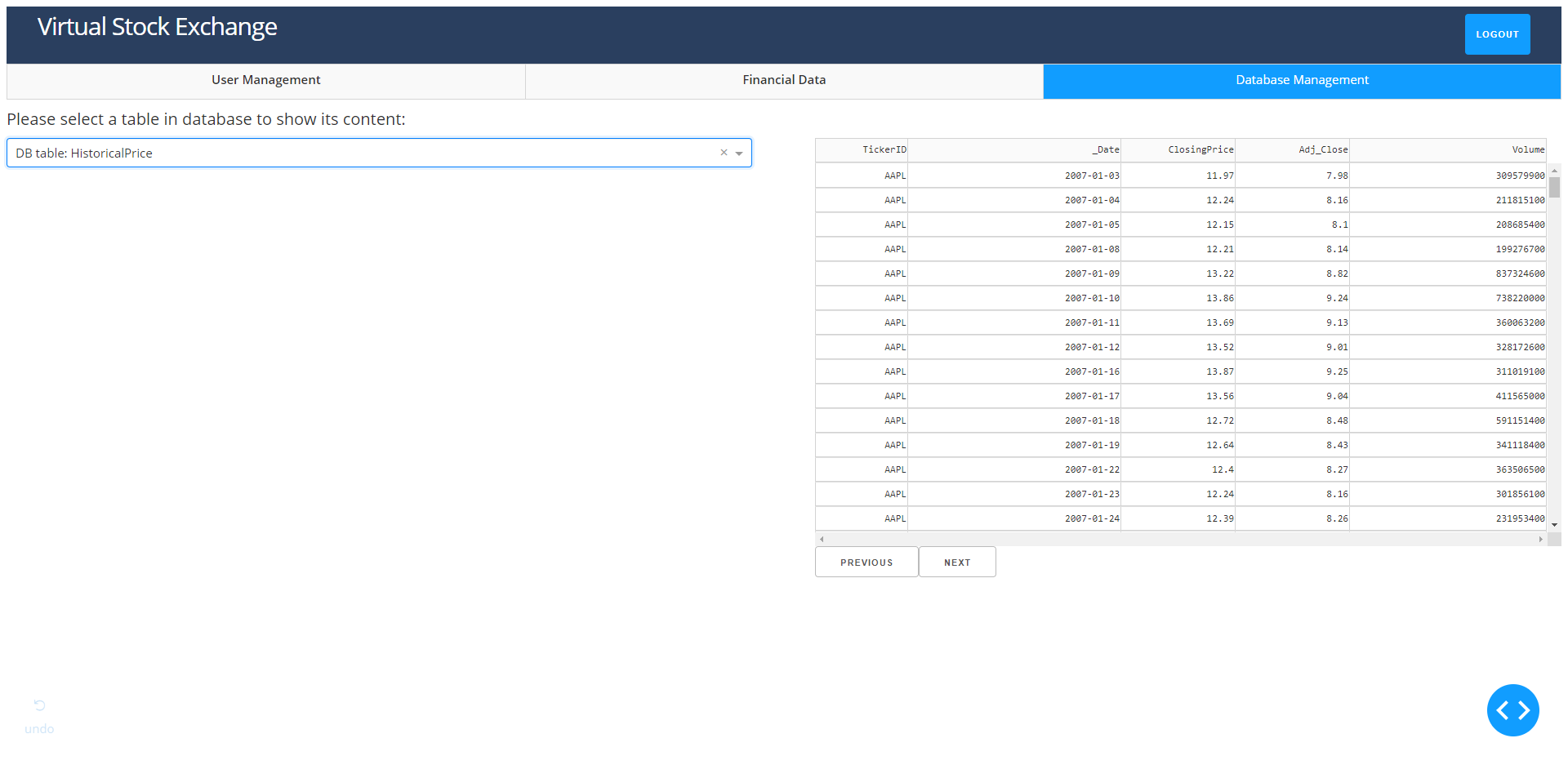
The system also provides an interface to add, modify or drop users:



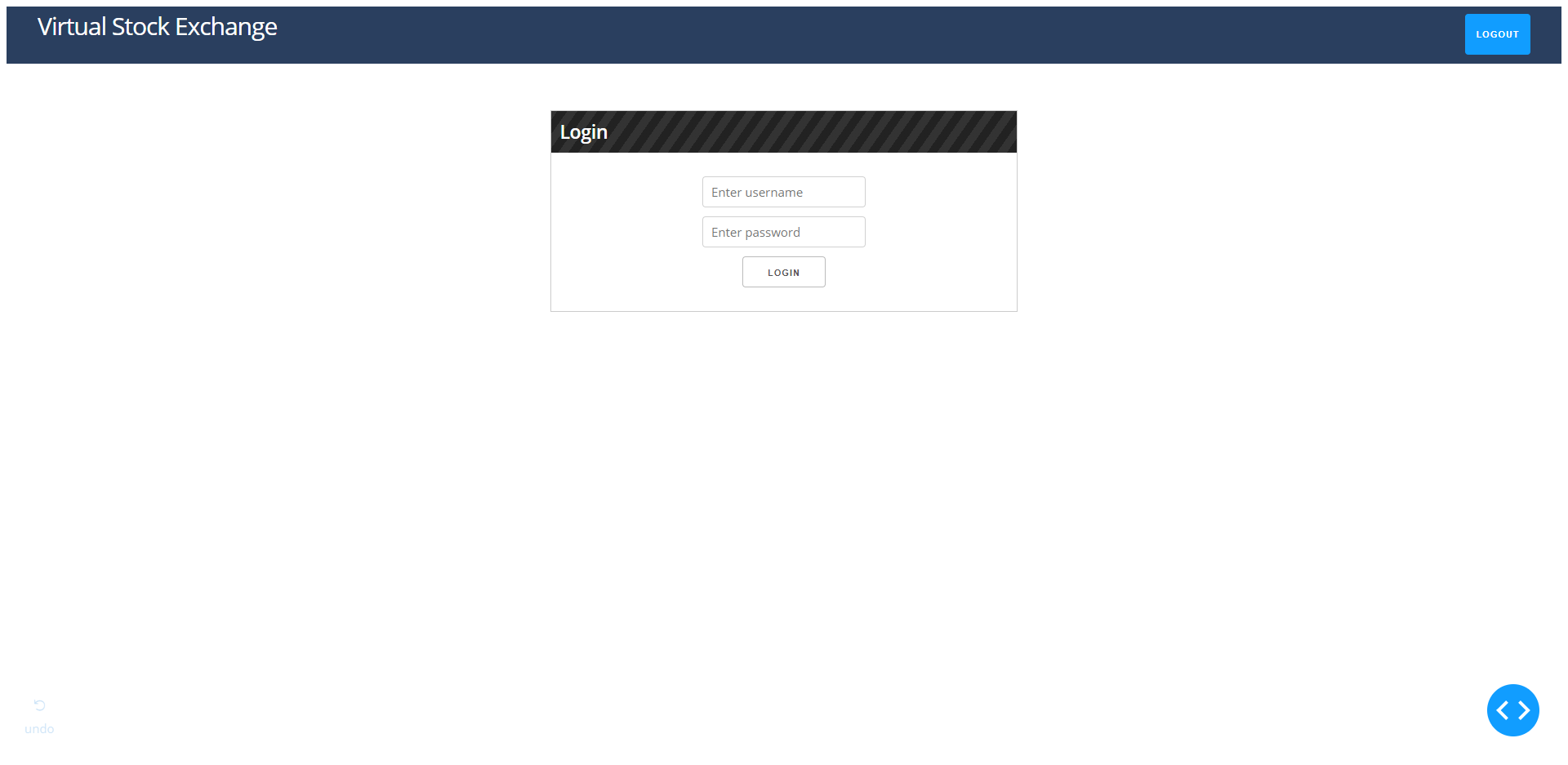
It also provides a database management tab, which can view all the data without requiring a SQL Server Management tool:

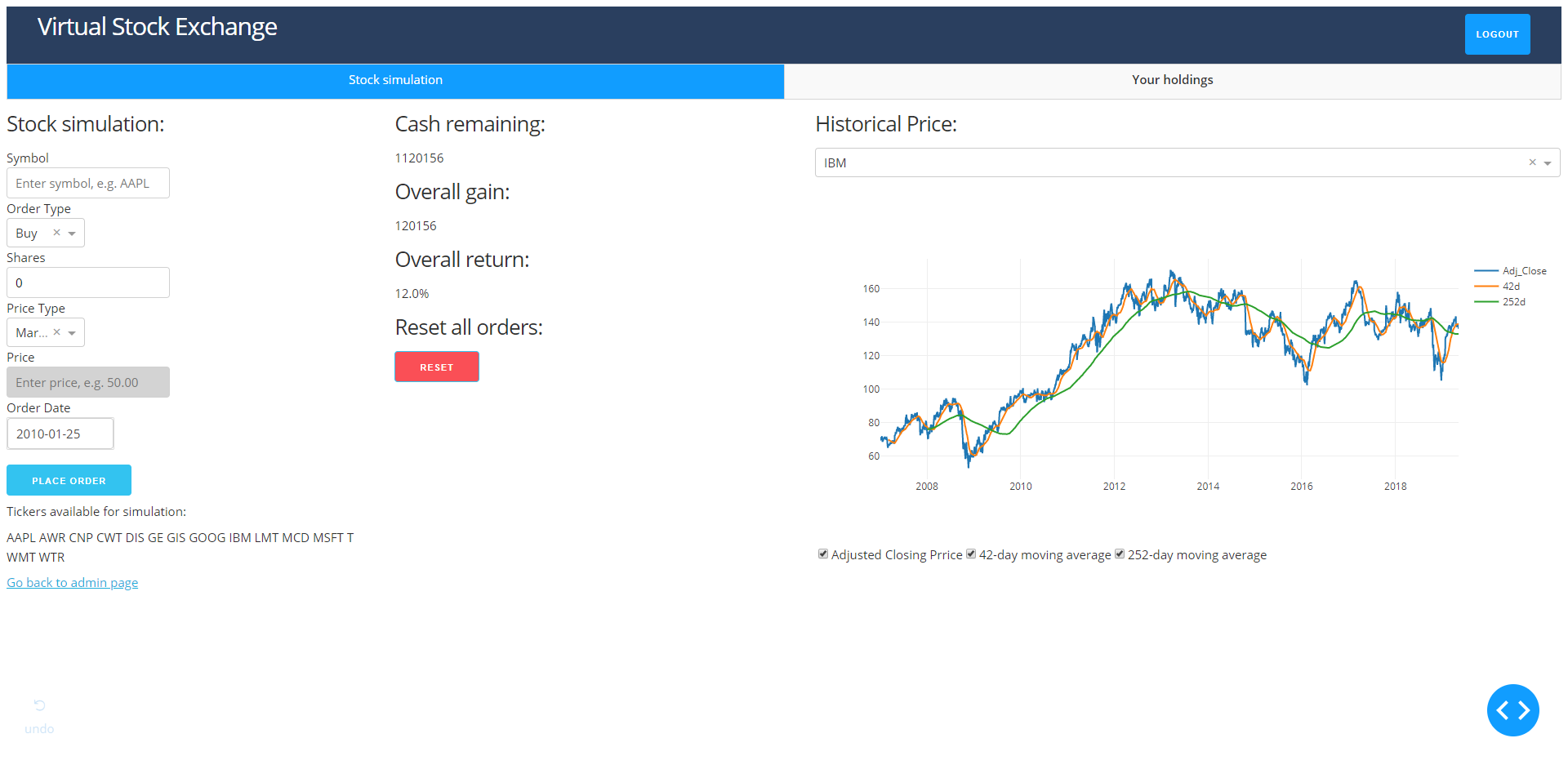


It can separate into multiple pages when there is too much data:

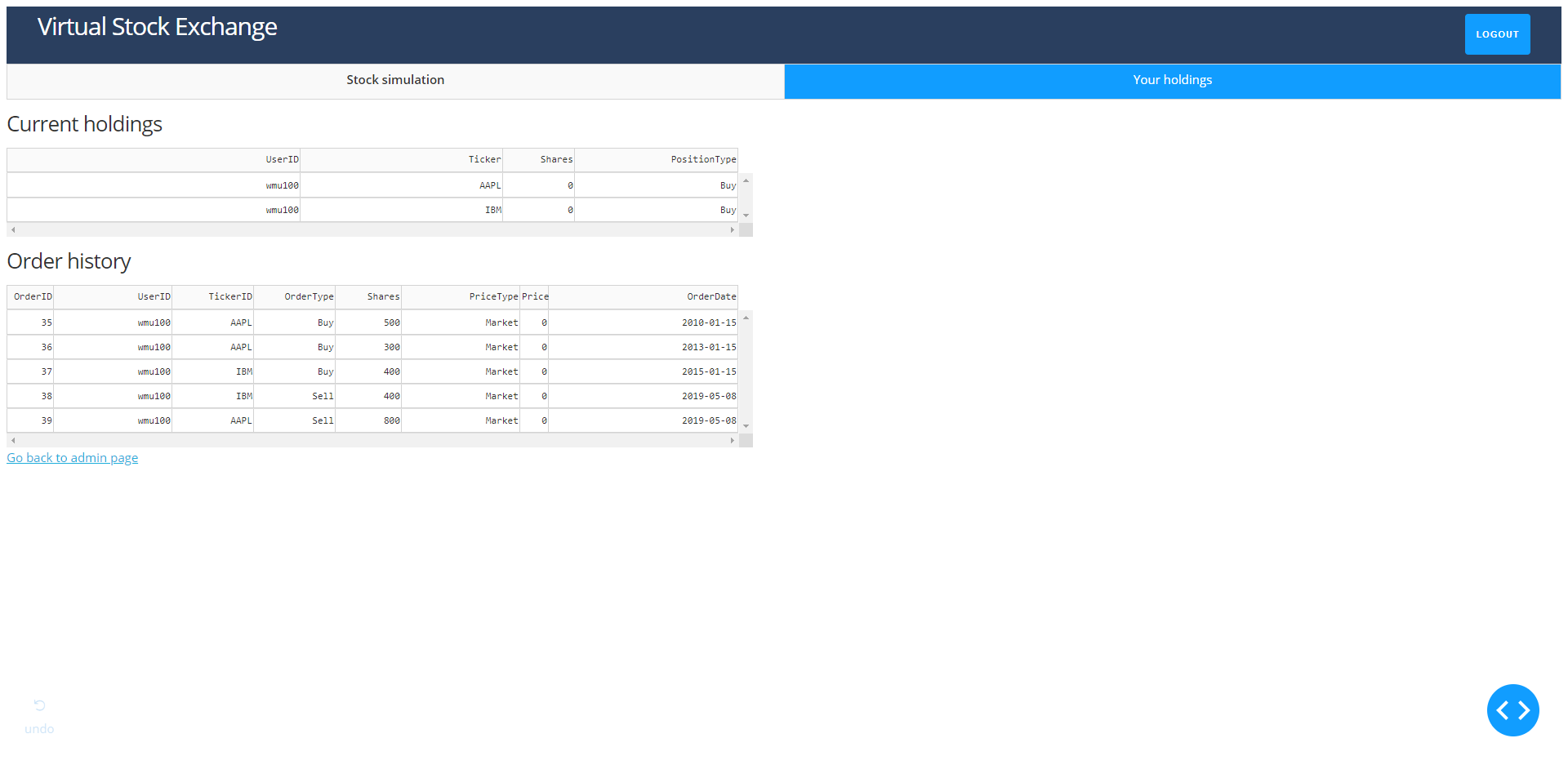


After admin or professor finishes configuration, students can log in using their own account:



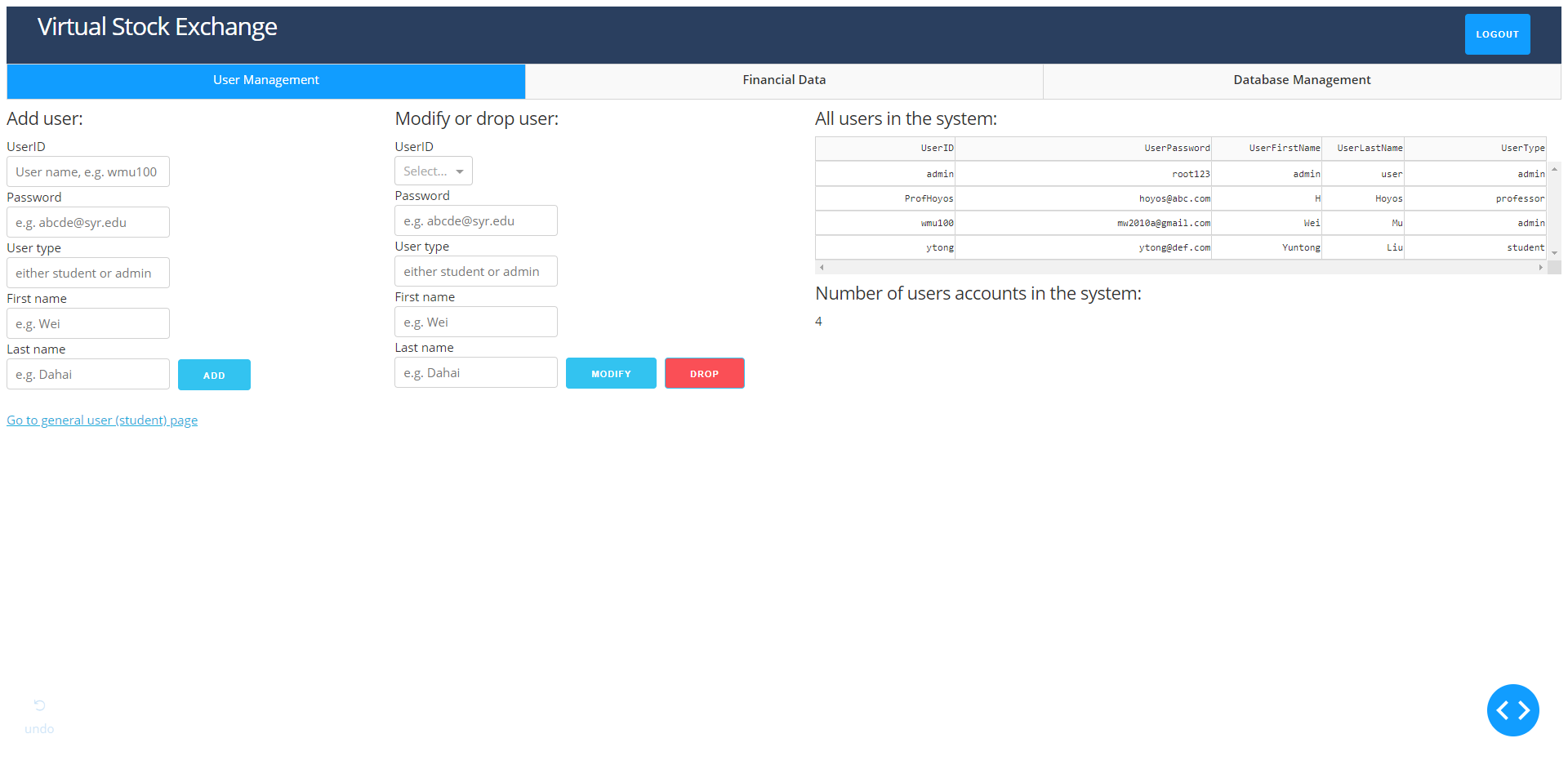


Holding page:



**Report:**

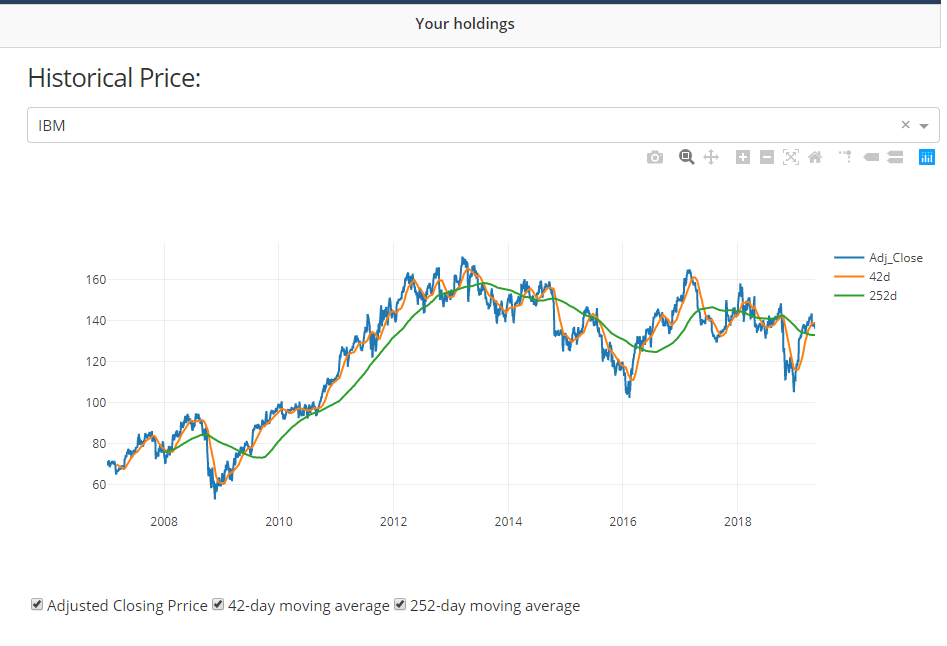
1. Who are the current users of the system?



2. How many students have been registered in this system?

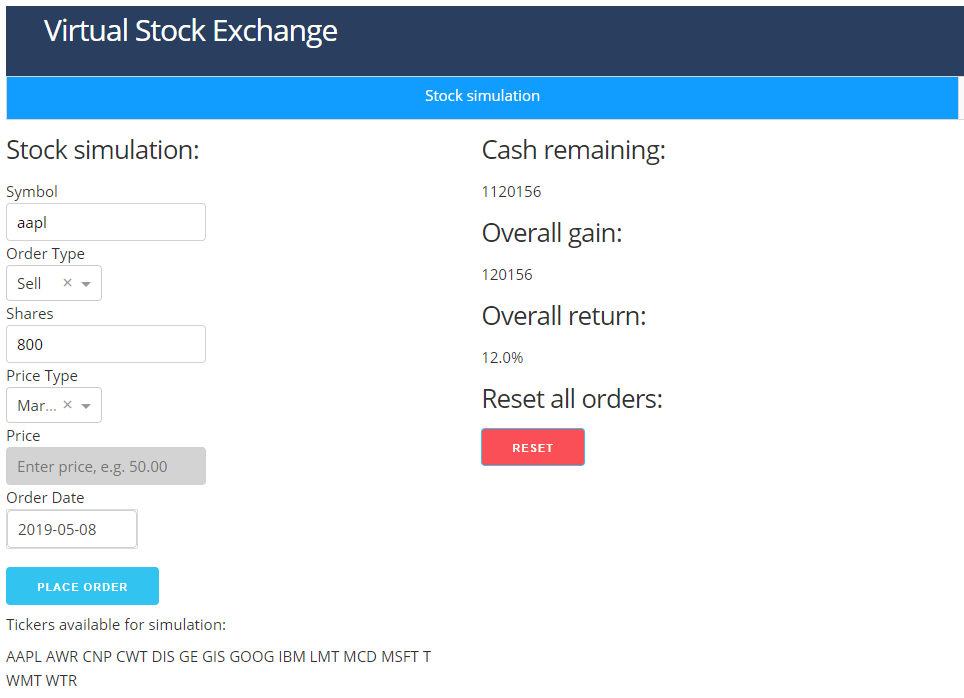


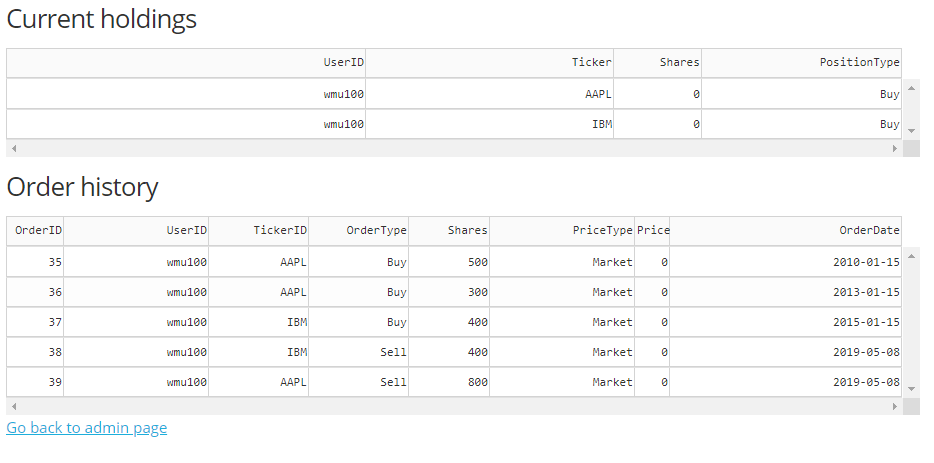
3. Show the price trend, 42-day and 252-day moving average of a stock (e.g. “IBM”) in the last 10 years.



4. Stock simulation: what is the return if User A bought 500 (as an example) shares of a stock (e.g. AAPL) at the closing price at a given date (e.g. 01/15/2010), 300 the same stock at another day (e.g. 01/15/2013), 400 shares of another stock (e.g. IBM) at another date(e.g. 01/15/2015) and sell all of the above at given date (e.g. 05/08/2019)?

1. Assume that users (students) have starting cash of $10000000. After a certain strategy has been applied, what is the remaining cash available at certain trading day?
2. What is the overall gain in cash if the user accepted the trading strategy in business question 4 (i.e. she placed order exactly as in question 4)?





# Appendix

The full Python code of our project is available on Github:

<https://github.com/mason1900/DashAppProject>

Currently the project has also been deployed to ~~<deleted from Github>~~ and is accessible via a web browser. To test the user interface, please log in as ~~<deleted from Github>~~ with password ~~<deleted from Github>~~