Carter Williams

11/5/2022

Dr. Zhonghang Xia

CS 396

Phase Two Report

Phase two of the term project includes building a system that allows users to track their financial information through a website. This includes creating a UI that users use to interact with the system to retrieve, update, create, and delete information. A backend system then receives these requests and manipulates data in the database and/or sends a response back to the user.

**Description**

Overall, the system created for this project allows users to track financial information (trades, assets, agents) relating to their investments. Users can add buy and sell trades relating to a certain asset to a database. Four asset types can be tracked: stocks, properties, bonds, and miscellaneous assets. Users can then add these to their portfolio by adding a buy trade related to the asset. Each asset class holds relevant information related to that asset; for example, a bond has an issuer, face value, rating, etc. A stock does not have any of these values associated with it, but it has a ticker, 52-week high/low, etc.

The system also holds other information such as mortgage rates, agent information, and more personal information about the user. All the details related to what information the system stores, and how it stores it will be discussed further in the technical details section.

**Technical Details**

This section will go into further detail of the database’s design, queries, and general details about the association between Django and the database. Any database design begins by mapping out the information the database keeps a record of. Usually, this is done via an entity relationship (ER) diagram, and this can be seen in Fig. 1. ER diagrams help database designers visualize the requirements of the database prior to implementing a database schema.

**ER Diagram Details**

Diagram, schematic

Description automatically generated

Fig 1. ER Diagram for phase two database

Fig. 1 shows how the entities of the database relate to one another and what attributes the database must store of each entity. Firstly, there is a sub entity of User, FintechUser, that stores extra information about a user. This allows the database to hold more data about a user (their address, occupation, etc.) for the asset management system. Essentially, FintechUser inherits all the User’s attributes and adds some of its own. Next, there is an Asset entity, and this is a super entity for four, more specific, sub entities: Stock, Property, Bond, and MiscAsset. Asset has one attribute, a name, and each sub entity extend Asset by having more attributes specific to its class. For instance, Stock holds data pertinent to itself: the ticker and high/low price for the year. Property, Bond, and MiscAsset also hold data specific to their respective asset class. StockSnapshot is a weak entity that keeps track of stock data on a specific date. Agent is an entity that represents a person or organization that brokers a transaction. This could be a real-estate agent, stockbroker, car dealer, etc.

User, Agent, and Asset all relate to one another through a single relationship, Trade. Trade represents a buy or sell order for a single asset. A single Agent also relates to a trade, and multiple users can relate to a single trade. This means multiple users can “own” the same asset by adding them to the trade record. In the relationship, there must be at minimum one Asset, Agent, and User that relates to trade. Trade also carries attributes like date and price to hold information about a transaction.

**Database Schema**

The database for phase two was built onto the database for phase one, but they have no relationship among one another. Similarly to phase one, the database was created by using Django’s ORM by creating models and relying on Django to migrate the models to the database. The schema for the database is depicted in Fig. 2.

Graphical user interface, diagram, application

Description automatically generated



Fig 2. Schema for phase two’s database

Converting the ER diagram to the database schema required creating some extra tables to efficiently keep track of data. Each of these squares correspond to a table in the database, and the lines connecting them represent a foreign key (relationship). A key symbol next to a column name represents the primary key of the table.

**Normalizing to 3rd Normal Form**

After creating the schema, very little had to be done in order to put all tables in third normal form. In my opinion, this is because I followed the steps outlined in zyBooks’ book over databases. Before justifying each table’s normal form, I would like to give the definition of third normal form from zyBooks. Normal forms are rules for creating a database with less redundant data; redundant data is the repetition of repeated values. Redundancy is caused by functional dependence among two columns. If column A depends on column B, that means for each value of B there is at most one A value. Third normal form is when all non-key depend on the whole key and only the primary key. A non-key column is any column(s) that could not identify a row in a table; in other words, a non-unique column(s).

Before discussing each tables normal form individually, it should be noted that every table in the database is already in first and second normal form. This is because each primary key is unique and there are no composite primary keys. This means a non-key column cannot depend on only part of the primary key.

Each section will discuss each table individually and justify why it is in third normal form.

**Asset**

Primary Key: id

Candidate Key Columns: N/A

Non-key Columns: assetName

Since assetName is the only column other than the primary key, it must depend only on the primary key. Therefore, this table is in third normal form.

**Stock**

Primary Key: FK\_asset\_stock

Candidate Key columns: ticker

Non-key Columns: yearHigh, yearLow

The yearHigh and yearLow columns are the only non-key columns because ticker is a unique column that could also be used to identify a stock row. YearHigh and yearLow only depend on the primary key because for each value of the primary key, there is only one yearHigh and yearLow value that relate to it. This is due to the primary key not being repeated in the table. YearHigh and yearLow also have no dependence among each other because one does not determine the other.

**StockSnapshot**

Primary Key: id

Candidate Key columns: N/A

Non-key Columns: openPrice, closePrice, dayHigh, dayLow, volume, FK\_stock\_snapshot, snapshotDate

StockSnapshot contains many non-key columns, but no individual or composite column(s) determine any other column. Each non-key column depends only on the primary key because each column value has almost nothing to do with one another. This is due to the column values being determined by market data that is essentially “random.” One cannot determine, for example, the closing price of a stock based on the open price.

**Bond**

Primary Key: FK\_asset\_bond

Candidate Key columns: N/A

Non-key Columns: issuer, maturityDate, faceValue, interestRate, rating

In Bond all non-key columns depend only on the primary key because no non-key column can determine another. For example, there may be an issuer, Google, but Google sells multiple bonds with varying interest rates, maturity dates, etc. It would be impossible to functionally determine any of the other column’s values based on solely the issuer. This logic carries over when you apply it to any other non-key column.

**Property**

Primary Key: FK\_asset\_property

Candidate Key columns: FK\_address\_property

Non-key Columns: totalfee

In the property table, the only non-key column is totalfee; this is because FK\_address\_property is a unique column. Since totalfee is the only non-key column, it can only be determined from the primary key.

**MiscAsset**

Primary Key: FK\_asset\_misc

Candidate Key columns: N/A

Non-key Columns: description

This table is in third normal form for the same reason property is. There is only one non-key column, so it must rely only on the primary key.

**Trade**

Primary Key: id

Candidate Key columns: N/A

Non-key Columns: action, tradeDate, pricePerAsset, assetQuantity, FK\_trade, FK\_asset\_trade, FK\_agent\_trade

The Trade table is in third normal form because all non-key columns depend only on the primary key. Action, tradeDate, pricePerAsset, and assetQuantity are data inputted by the user, and can take on any value. This means they cannot functionally depend on another non-key column. All the foreign keys also do not depend on non-key columns because they do not depend on any of them to determine their value.

**CurrentHoldings**

Primary Key: FK\_trade

Candidate Key columns: N/A

Non-key Columns: N/A

There is only one column in this table, so it must be in third normal form.

**User**

Primary Key: id

Candidate Key columns: username

Non-key Columns: password, passwordSalt

Password cannot depend on passwordSalt because passwordSalt is randomly generated. This also means passwordSalt cannot depend on password. This means all non-key columns depend only on id, so the table is in third normal form.

**UserTrade**

Primary Key: id

Candidate Key columns: (FK\_trade, FK\_user)

Non-key Columns: N/A

There are no non-key columns in this table, so it must be in third normal form.

**FintechUser**

Primary Key: FK\_user\_assetUser

Candidate Key columns: N/A

Non-key Columns: FK\_address\_assetUser, occupation, sex, age

Each non-key column in this table is essentially “random” because they are inputted by the user. The address, for example, cannot be determined by the occupation, sex, or age. This works in any other direction; therefore, the table is in third normal form.

**Agent**

Primary Key: id

Candidate Key columns: FK\_address\_agent

Non-key Columns: name, firmName

Name does not depend on the address or firmName because neither of those two can functionally determine name. This logic works for firmName as well, so this table is in third normal form.

**AgentFee**

Primary Key: id

Candidate Key columns: N/A

Non-key Columns: feeRate, feeFlat, feeDescription, FK\_agent\_fee

Each non-key column depends only on the primary key because none of the non-key columns determine one or many of the others. This is because they are independent of one another and can take on any value of its data type. This means the table is in third normal form.

**Address**

Primary Key: id

Candidate Key columns: (street, city, zipCode, state, city, unitNum)

Non-key Columns: N/A

The columns in address, other than the primary key, can be used to uniquely identify a row. This is because the whole address is required to be unique in order to reduce the amount of mostly duplicate rows. Since there are no non-key columns, the table is in third normal form.

**MortgageRate**

Primary Key: id

Candidate Key columns: N/A

Non-key Columns: bankName, interestRate

BankName and interestRate are the only non-key columns, and neither of them functionally depend on each other. This is because a bank’s name does not functionally determine the interest rate it offers and vice versa. Due to this, the table is in third normal form.

**Description of Queries**

All database queries are run through views in the assetman/views.py file. This subsection will describe three of the most crucial views, and the queries it uses to operate.

1. home(request)

This view is what renders the home page for the /assets/ page. It grabs all the currently held stocks, bonds, properties, miscellaneous assets, and their associated trades to display to the user. It does this by retrieving trades from the CurrentHoldings table and searching the UserTrades table to see if the user is associated with any of those trades. It then categorizes those trades based on the asset class and passes that information to the home.html to be rendered and sent to the user. This is a simple view that uses select and join operations on tables to view the currently held assets by a user.

1. addAsset(request)

This view adds a buy trade for an asset. First, it grabs the asset’s class from the form data, and calls the appropriate function to grab the rest of the form data. That data is then used to insert new rows into the associated Asset, Trade, UserTrade, and CurrentHoldings tables. This uses multiple insert queries to add a buy trade to the system.

1. editAsset(request, trade\_id)

This view edits the tables associated with the previous view. It can update the associated Asset and Trade tables. It does not allow you to add users to the trade. This acts as an update query by grabbing form data and changing data in the associated Asset and Trade tables.

**Sample Queries**

1. /assets/delete/<int:trade\_id>/

This URL causes the system to execute a select query with the trade\_id specified in the WHERE clause. If it is found, the trade row is deleted using a DELETE statement; if not, the system responds with a 404 error.

1. /assets/findtrades/

This URL uses a form to search for trades on a specified date made by the user. It does this by using a SELECT statement that specifies the date in the WHERE clause. After it grabs all the trades, it then joins it to the UserTrade table on the user’s id and trade id. The trades are then rendered on the webpage.

**Implementation Details**

Django provides almost all functionality related to interacting with the database. For this project, no SQL had to be written because Django has libraries that handle that for you. Models had to be defined that became associated with tables in the SQLite database. This project uses the Model-view-controller architecture to implement the overall system. Models were already described, but they are what holds information. In this case, they are represented as tables in a database. Views display the models in some fashion. In this project, views are rendered HTML webpages. Controllers facilitate communication between the model and views by performing CRUD operations on the models and updated the views. Controllers are implemented in the asssetman/views.py file, and they take HTTP requests to interact with the models.

**Summary of Achievement**

Phase two of this project is a system that allows users to track various financial data. Overall, I believe this project is a well-built system that could be used for keeping track of my financial assets. It is a pretty basic system compared to something like Yahoo Finance or Seeking Alpha, but with some extra work it could be great for tracking your net worth.

More generally, this project has taught me a great deal about building software systems, and, more specifically, about building server-client systems. I made many mistakes and could have designed a better, more efficient system, but I learned from those mistakes made in this project.

The system allows users to keep track of asset transactions, view information related to stocks, properties, bonds, and miscellaneous assets, keep track of an agent’s assets, view mortgage rates, and show a gain or loss over a period. This project meets the requirements for phase two, and is set up to begin phase three.

**Discussion**

One problem I encountered while progressing through this project was switching the DBMS to MySQL from SQLite. Django kept throwing errors related to certain migrations, so I restored my previous system that used SQLite through my git repository. I do not believe switching from SQLite to MySQL would improve performance, however. This is because my system does not have many concurrent transactions occurring, and situation like that are where MySQL excels compared to SQLite. Overall, I believe SQLite is probably a better choice for my project because it is more simplistic than MySQL which means it handles queries, generally, in a faster manner.

Code contained in assetman/views.py could likely be refactored for better performance. No operation takes longer than O(N2) where N is the number of rows in the database. This should allow the system to scale decently well, but many of the O(N2) functions could likely be taken down to O(N) or O(logN) by altering the physical design of the database.