Documentation for the “Ea$y Finance” web application

Table of Contents

Foreword

The following is a summary of my final project for CS50’s Introduction to Computer Science. I have audited the course on edx.org. Disclaimer: I did not build this from scratch. Much of the preliminary “groundwork” was done by the CS50 staff. Please keep this in mind.

The origins of this program

The students taking the CS50 course need to build a web application called “C$50 Finance” to complete Problem Set 7. This stock-trading application has the following elements:

- A registration form on the “/register” route that enters users into the database upon successful completion and sets their cash to $10,000.00.
- A login form on the “/login” route that logs users in upon successful completion and redirects them to the main page. This and the logout function were completed by the CS50 staff prior to the assignment.
- A table on the main page that contains information about the user’s stock portfolio. It needs to show the number of shares owned by the user, the current price per share, and the current value of their holdings, for each stock. It also needs to show users how much cash do they have and how the total value of their portfolio.
- A form that allows users to obtain the price of a selected stock on the “/quote” route. This element uses the Yahoo Finance API to get the information the user wants. Users have to enter the stock’s symbol for this function to work.
- A form on the “/buy” route that allows users to buy shares of the stock of their choice, as long as they have sufficient funds.

- A form on the "/sell" route that allows users to sell shares of stocks they own, as long as they have enough shares for the transaction.
- A table on the "/history" route that contains all stock transactions made by the user. It has to list the type (buy or sell) of the transaction, the stock bought or sold, the number of shares involved, the price of the stock at the time of the transaction, and a timestamp.
- A form on the "/change-pw" route that allows users to change their passwords.

(Pictures of each element are included in the "CS50_Finance_pics" folder.) Python (Flask), SQL, Jinja2, HTML and CSS were used in the making of this program. As you can see, no JavaScript of any sort is present. The program communicates with the user via redirects.

I have built my final project based on this program. I have decided to add a "financial tracking" component to it and combine it with the stock-trading "simulator". My aim was to create a simple and easy-to-use, yet wholesome application. I am somewhat invested in the topic of personal finance, so the idea was formed quickly.

Summary of the features

With all this said, it is time to summarize the functions and elements of the "Ea$y Finance" application:

- A registration form on the "/register" route. In addition to entering their usernames and passwords, users are encouraged to enter the starting balance for their accounts. Filling this element of the form out is optional, though. If leaves it empty, the starting balance will be set to $10,000.00 by default.
- A login form on the "/login" route and a logout function. These two are virtually unchanged.
- A table on the main page that shows a summary of the user's income, expenses, cash-flow (income minus expenses) and their cash. In hindsight, showing the total value of the user's portfolio may be desirable.
- Another table on the main page that lists recurring income/expense elements the user has. It shows the following details: the ID, the type (income or expense), the title, the amount and the frequency (daily, weekly or monthly) of the entries. It also shows when these entries were added.
- A form on the main page that allows the user to add recurring entries. The users can select from three frequency options, two type options. They need to enter a title and an amount which is higher than zero.
- Edit recurring entries button on the main page: when users click on this button a form appears which allows users to edit an already existing recurring entry. Users only need to enter the ID of the entry they wish to modify to change one or more attribute(s) of it.
- Delete recurring entries button on the main page: when the user clicks on this button a form and another button appear. The form allows users to delete a recurring entry of their choosing from the list. They only need to enter the chosen entry's ID. The button that appears allows users to delete all recurring entries from the list. They need to check a checkbox to actually delete all entries, just to be sure.
- A table on the "/stocks" route that contains information about the user's portfolio. This table was also left unchanged.

- The “quote”, “buy” and “sell” forms were moved to the “/stocks” route. They are arranged in this order, below the table I mentioned earlier. The quote function now shows the price of the selected stock via a JavaScript alert. Other than this, the forms are the same as they were.
- A table on the “/history” route that lists all entries in the user’s financial history. It shows the ID, the type (balance, income, expense), the title and the amount for each entry, as well as the time when they were entered. The first entry is the starting balance of the user.
- Change starting balance button on the “/history” route: when the user clicks this button (situated above the financial history table), a form with a single element appears. This form allows the user to set a new starting balance for their account.
- A form on the “/history” route that allows users to add entries to their financial history. This form is almost identical to the “recurring entries” form. The only difference is the lack of the “frequency” element.
- Edit entry button (below the enter entries form): when the user clicks on this button, a form appears. This form allows the user to modify an existing entry in the financial history table by changing one or more of its attributes. The user only needs to add the ID of the selected entry. All other form elements are optional.
- Delete entry button (next to the edit entry button): when the user clicks on this button a form and another button appear. The form allows the user to delete an entry from the financial history table. They only need to enter the selected entry’s ID. Clicking on the delete all entries button deletes all entries from the financial history table, the transaction history table and from the portfolio table on the “/stocks” route. It also sets the user’s starting balance to $10,000.00. I decided to do this “hard reset” because the user’s “cash situation” would become hard to track otherwise. The users need to check a checkbox before they can do this.
- A table on the “/history” route summarizing the user’s transaction history. This table is the same as it was in the C$50 Finance app.
- Hide transaction history button (above the transaction history table): when the user clicks on this button the transaction history is hidden and the button is replaced with the show transaction history button.
- Show transaction history button: when the user clicks on this button the transaction history is shown to them and the button is replaced with the hide transaction history button. This button is not visible by default.
- A form on the “/change-pw” route that allows users to change their passwords. The users need to enter their old password and their new password twice. They are redirected to the main page upon success.

(Pictures of each element are included in the “Easy_Finance_pics” folder.) Python (Flask), SQL, JavaScript, Jinja2, HTML and CSS were used in the making of this program. Getting rid of the redirect-based communication between the user and the program was one of the first things I did. I replaced it with JavaScript-based “real-time reactions”. This renewed program can be divided into three major components: the logic and the backend operations (Python and SQL), the real-time communication elements (mostly JavaScript event listeners) and the appearance of the pages (HTML, CSS and Jinja2).

Details and functional nuances

I am going into the minute details of the program in this section. As I mentioned above, the application has 3 major components. The combination I used here is somewhat peculiar, I would say. Generally speaking, the program behaves in the following manner:

- App routes which assume that the user is logged in use the "login_required" function (located in helpers.py) as a decorator. This function was written by the CS50 staff.
- If a route is reached via the GET method and there is an HTML file with the same name, then that file is rendered for the user to see.
- If a route is reached via the POST method, or there is no corresponding HTML file, the program proceeds to handle the user's input.
- The user's input is sent to the Python script, via a custom AJAX-request in JavaScript.
- Correctness is checked on the server's side, within the Python script.
- If there is an issue with the input, the server sends a message in JSON format to the JavaScript element. The JavaScript element shows an appropriate message to the user and highlights the area where the error has occurred.
- If all checks for correctness are checked, the Python script executes the back-end operations. This usually involves some sort of database operation and thus, the usage of SQL. A custom module called "SQL" was provided by the CS50 staff for these tasks.
- The program either refreshes the page or redirects the user to the appropriate page to showcase the changes made by their actions.
- The changes are usually shown in one or more of the many HTML tables. This usually involves some Jinja2 code, mostly for-loops calling elements of the tables in the database.

I have read on a forum (Stack Overflow or Quora, or maybe somewhere else) that server-side validation of the user's input is "safer". I do understand that this method uses more resources in return. I have decided that the trade-off is worth it.

Register page

Register:

Files involved:

- register.html
- application.py (lines 561 – 603)
- helpers.py (usd function, made by the CS50 staff)
- scripts.js (lines 228 – 301)
- final_project.db (tables: users, statement)

This element is responsible for registering new users. The user's input is handled as I have described above. If all checks (5 in total) are cleared the user's name and their hashed password are entered in the "users" table, along with their starting balance. If the user has entered a starting balance while filling the form out, that amount is used. If not, the default of $10,000.00 is entered into the database. The user's starting balance is also entered into the "statement" table, which is used for the "financial history" table. The user is then redirected to the main page. (This was a specification set up for Problem Set 7 by the CS50 staff. In a "real" app, it is probably more acceptable to redirect users to the login page and have them verify their account before they could log in.)

The user is free to set any password. There are no requirements of minimum length, having a capital letter and having non-alphabetic characters for the password. This needs to be changed before the app could be used in a real-life scenario.

Login page

Login:

Files involved:

- login.html (written by the CS50 staff)
- application.py (lines 496 – 529, written by the CS50 staff)
- scripts.js(lines 332 – 376)
- final_project.db (users table)

This element is responsible for logging in users. First, the program forgets all users by clearing Flask.session. Then, if the user's input passes all checks (a total of 3), their ID is stored as Flask.session's user ID. This allows the program to remember which user has logged in. Finally, the user is redirected to the main page.

Logout:

Files involved:

- application.py (lines 532 – 540)

Flask.session is cleared, thus the program forgets all users. Users are redirected to the login page.

Main page

Financial summary table

Files involved:

- index.html (lines 10 – 25)
- helpers.py (usd function)
- final_project.db (tables: statement, users)
- application.py (lines 48 – 51, 60 – 71)

This table shows a summary of the user's overall financial situation. The following values are shown: the sum of expenses and income, their difference and the user's cash. I have used SUM statements in the "statement" table for the income and expense values. Adding the total value of the user's holdings would arguably be a good idea.

Recurring entries table

Files involved:

- index.html (lines 26 – 48)
- application.py (lines 52-59, 70-71)
- helpers.py (usd function)
- final_project.db(auto_values table)

This table shows a list of the user' recurring income and expense titles, using elements from the "auto_values" table in the database. The following values are shown: the ID, the type (income, expense), the title, the amount and the frequency (daily, weekly, monthly) of the entry. The time when the entry was added to the list is also presented.

Add recurring entry form (INCOMPLETE)

Files involved:

- index.html (lines 49 – 79)
- scripts.js (lines 377 – 449)
- application.py (lines 73 – 99)
- final_project.db (auto_values table)

This form allows the user to add elements to the list of recurring income and expense titles. The user needs to select the frequency and the type of the entry, then enter the title and the amount. If all checks are cleared (a total of 4), the entry is added to the "auto_values" table and thus it is shown in the recurring entries table.

The Python function of this form is supposed to serve a different purpose as well, however. I originally intended to have the program add rows to the "statement" table using the values in the "auto_values" table. You can see the "remains" of these attempts in application.py (lines 100 – 134) and helpers.py (daily, weekly and monthly functions). In its complete state, the application should entries with "daily" frequency every day, "weekly" ones every week and "monthly" ones every month.

I could not quite find the right solution for this. I have experimented with the Schedule and Celery libraries. With Schedule, I could not work out how to run the code asynchronously. I could not use Celery because the CS50 IDE (which I have used to write this application) uses a message broker for its own purposes and it blocked my attempts at using one myself. I am certain that there is a solution to this. I'm hoping to find answers in CS50's Web Development course, which is the next on my list.

The table is not exactly worthless as is, though. It can be used to remind users about their recurring expenses. Still this functionality is rather limited compared to what I originally intended to do.

Edit recurring element button

Files involved:

- index.html (lines 80 – 83)
- scripts.js (lines 450 – 455)

Clicking on this button displays the edit recurring entry form, which is hidden by default. At the same time, the delete recurring entry form and the delete all recurring entries button are hidden if they were displayed.

Delete recurring element button

Files involved:

- index.html (lines 80 – 83)
- scripts.js (lines 456 – 461)

Clicking on this button displays the delete recurring entry form and the delete all recurring entries button, which are hidden by default. At the same time, the edit recurring entry form is hidden if it was displayed.

Edit recurring entry form

Files involved:

- index.html (lines 84 – 118)
- scripts.js (lines 474 – 513)
- application.py (lines 169 – 207)
- final_project.db (auto_values table)

This form allows users to update a recurring income or expense entry. I have decided to make all form elements other than the ID optional for the sake of user experience. This means the user's input only needs to pass two checks. Once that happens, the row with the selected ID in the "auto_values" is updated. I had to use CASE statements to carry out the partial updates. It was a really educational challenge.

Delete recurring entry form

Files involved:

- index.html (lines 119 – 127)
- scripts.js (lines 564 – 596)
- application.py (lines 278 – 304)
- final_project.db (auto_values table)

This form allows users to delete a recurring income or expense entry. If all checks are cleared (a total of two) the row with the selected ID in the "auto_values" table is deleted.

Delete all recurring entries button

Files involved:

- index.html (lines 128 – 131)
- scripts.js (727 – 331)

Clicking on this button displays the delete all recurring entries form, which is hidden by default.

Delete all recurring entries form

Files involved:

- index.html (lines 132 – 140)
- scripts.js (lines 553 – 563, 570)
- application.py (lines 286 – 291)
- final_project.db (auto_values table)

This form allows the user to delete all recurring income and expense entries. The user needs to check a checkbox to do this. Technically, this form is part of the delete recurring entry form. I had to use a

JavaScript event listener to change the checkbox's value in HTML as a workaround. I have done the same for the delete all entries checkbox on the "/history" route.

Stocks page

Portfolio table

Files involved:

- stocks.html (lines 8 – 39)
- helpers.py (usd function)
- application.py (lines 377 – 402)
- final_project.db (tables: portfolio, users)

This table shows a list of stocks currently owned by the user, using elements from the "portfolio" table in the database. The user's cash (from the "users" table) and the total value of the portfolio (cash + value of stocks) is also shown. The following values are presented: the name, the number of shares owned the current price for each stock and the value of each portfolio element (current price * number of shares).

There is definitely room for improvement here. A way to show the user how did the value of the stocks change since they were purchased would be nice. They can the buy price up in the transaction table, but it can be a hassle for them to jump back and forth between the pages and compare values. Maybe adding a column with the average buy price (weighted with the number of shares purchased) could do the trick.

Quote form

Files involved:

- stocks.html (lines 40 – 48)
- scripts.js (lines 302 – 331)
- helpers.py (lookup function, written by the CS50 staff)
- application.py (lines 543 – 558)

This form allows the users to get the price of a stock of their choosing. They only need to enter the stock's symbol. If the input passes all checks (a total of two) the price of the selected stock shows up in a JavaScript alert. The API of Yahoo Finance is used (via the "lookup" function) for getting the prices.

Buying shares form

Files involved:

- stocks.html (lines 49 – 62)
- scripts.js (lines 3 – 68)
- application.py (lines 406 – 466)
- final_project.db (tables: portfolio, history, users, statement)

This form allows users to buy shares of the stocks they choose. They need to enter the stock's symbol and the number of shares they wish to buy. If the user's input passes all checks (a total of 5) a number of database operations happen. If the user has bought shares of a stock he or she already owned, the "portfolio" table is updated, if not, a new row is added to it. The transaction is then added to the "statement" (as an expense entry) and "history" tables. Finally, the cash value in the "users" table is updated.

Selling shares form

Files involved:

- stocks.html (lines 63 – 82)
- scripts.js (lines 69 – 143)
- application.py (lines 606 – 664)
- final_project.db (tables: portfolio, history, users, statement)

This form allows users to sell shares from their portfolio. They need to choose a stock from a down list and enter the number of shares they wish to sell. If the input passes all checks (a total of 6), the database operations take place. The transaction is added to the "statement" table (as an expense entry) and the "history" table. The "portfolio" table and the cash value in the "users" table are updated.

History page

Change starting balance button

Files involved:

- history.html (line 9)
- scripts.js (lines 695 – 699)

Clicking on this button displays the change starting balance form, which is hidden by default

Change starting balance form

Files involved:

- history.html (lines 10 – 18)
- scripts.js (lines 700 – 726)
- application.py (lines 352 – 375)
- final_project.db (tables: users, statement)

This form allows users to change the starting balance of their account. The user has to enter the new amount for their starting balance. If the single check is cleared, the "statement" table's starting balance value and the "users" table's cash value is updated (the old balance is subtracted from it and the new balance is added to it).

I wanted to allow users to "mess around" using this function. Having it lets users test out different financial scenarios if they wish to do so.

Financial history table

Files involved:

- history.html (lines 19 – 39)
- helpers.py (usd function)
- application.py (484 – 493)
- final_project.db (statement table)

This table shows all income and expense entries (plus the starting balance) that was created in the user's account, using elements of the "statement" table in the database. The following elements are shown: the ID, the type (income, expense), the title, and the amount of the entry, as well as the time and date when it was added.

Enter income or expense element form

Files involved:

- history.html (lines 40 – 61)
- scripts.js (lines 641 – 694)
- application.py (lines 135 – 167)
- final_project.db (tables: statement, users)

This form allows users to add income or expense titles. The users need to choose the type (income or expense), then they need to add the title and the amount of the entry. If the user's input passes all checks (a total of 3), the amount they entered is either added to (income) or subtracted from (expense) the cash value in the "users" table.

Edit entry button

Files involved:

- history.html (line 63)
- scripts.js (lines 462 – 467)

Clicking on this button displays the edit entry form, which is hidden by default. It also hides the delete entry form and the delete all entries button if they are displayed.

Delete entry button

Files involved:

- history.html (line 64)
- scripts.js (lines 468 – 473)

Clicking on this button displays the delete entry form and the delete all entries button, which are hidden by default. It also hides the edit entry form if it is displayed.

Edit entry form

Files involved:

- history.html (lines 66 – 91)
- scripts.js (lines 514 – 552)
- application.py (lines 209 – 276)
- final_project.db (tables: statement, users)

This form allows users to edit income and expense titles. They can choose a new type (income or expense), add a new title and amount. All form fields other than the ID are optional. If the user's input passes all checks (a total of 2) the program updates the "statement" table and the cash value in the "users" table. The latter can happen in 4 different ways:

- if the original type of the entry and the new type are both “income”, the old amount is subtracted from the cash value and the new amount is added to it
- if the original type is “income” and the new type is “expense”, both the old and the new amount are subtracted from the cash value
- if the original type and the new type are both “expense” the old amount is added to the cash value and the new amount is subtracted from it
- if the original type is “expense” and the new type is “income” both the old and the new amount are added to the cash value

In hindsight, this can be a really inefficient way to solve this problem. If the amount and type are unchanged the program uses a lot of resources needlessly. I will probably have to find a way to deal with this when I take another look at this project.

Delete entry form

Files involved:

- history.html (lines 92 – 100)
- scripts.js (lines 608 – 640)
- application.py (lines 306 – 350)
- final_project.db (tables: statement, users)

This form allows users to delete income and expense entries. They only need to enter the ID of the selected entry. If all checks (a total of 2) are cleared, the selected entry is deleted from the “statement” table and the cash value in the “users table is updated. If the deleted entry was an income, its amount gets subtracted from the cash value. If it was an expense, its amount gets added to the cash value.

Delete all entries button

Files involved:

- history.html (lines 101 – 107)
- scripts.js (lines 732 – 736)

Clicking on this button displays the delete all entries form, which is hidden by default.

Delete all entries form

Files involved:

- history.html (lines 108 – 115)
- scripts.js (lines 597 – 607, 614)
- application.py (lines 313 – 324)
- final_project.db (tables: history, portfolio, statement, users)

This form allows users to delete all entries from the “history”, “portfolio” and statement tables, and reset their starting balance to $10,000.00. They only need to check a checkbox before they can go through with this. Technically, this form is a part of the delete entry form.

I’m starting to think this “form in a form” concept is a bad idea. It creates a lot of needless complication. I may get rid of this later on.

Hide transaction history button

Files involved:

- history.html (lines 116 – 118)
- scripts.js (lines 737 – 742)

Clicking on this button hides itself and the transaction history table, which are displayed by default. It also displays the show transaction history button, which is hidden by default.

Show transaction history button

Files involved:

- history.html (lines 119 – 121)
- scripts.js (lines 743 – 749)

Clicking on this button hides itself and shows the transaction history table and the hide transaction history button.

Transaction history table

Files involved:

- history.html (lines 122 – 143)
- helpers.py (usd function)
- application.py (lines 472 – 483, 491 – 493)
- final_project.db (history table)

This table shows all stock-related transactions the user has made so far. The following elements are shown: the type (buy, sell) of the transaction, the stock in question, the number of shares involved, the price at which the transaction happened, plus the date and time of the transaction. The "history" table in the database is used for this.

Change-pw page

Change password form

Files involved:

- change-pw.html
- scripts.js (lines 144 – 227)
- application.py (lines 666 – 714)
- final_project.db (users table)

This form allows users to change their passwords. They need to enter their old password and their new password twice. If all checks are cleared (a total of 6), the hashed password value in the "users" table is updated.

This function suffers from the same issue as the register form. Users can enter any password they want, without any restrictions.

A breakdown of the files

The following is a brief summary of the files that make up this web application:

- apology.html (redundant): this file was used for handling invalid user input in the "C$50 Finance" application
- application.py: this file contains the server-side operations of the program (SQL, Flask).
- buy.html (redundant): this file contains the form that is used for buying shares. Originally, the form had its own page ("/buy").
- celerybeat-schedule (not working): this file is supposed to contain a schedule, which was to manage the recurring income and expense entries
- change-pw.html: this file contains the HTML component for the change-pw page (the form that is used for changing passwords).
- final_project.db: the database file that contains all the tables used in this program:
  - auto_values: the table containing the recurring income and expense entries. Columns: id, type, title, amount, user_id, time, frequency
  - history: the table containing the user's transaction history. Columns: id, user_id, transaction_type, stock, price, shares, time
  - portfolio: the table containing the user's current portfolio. Columns: user_id, stock, current_price (not used), shares
  - statement: the table containing the user's financial history. Columns: id, type, title, amount, user_id, time
  - users: the table containing each user's username, their hashed passwords and the amount of cash they own. Columns: id, username, hash, cash
- flask_celery.py (not working): this file configures Celery (a library for asynchronous programming) to work in Flask's environment
- helpers.py: this file contains functions that were used for multiple elements of the program: apology (redundant), login_required, lookup, usd, daily (not working), weekly (not working) and monthly (not working)
- history.html: this file contains the HTML component of the history page
- index.html: this file contains the HTML component of the main page
- layout.html: this file serves as the template for all other HTML files. It contains the style elements (such as the navigation bar, for example) that are shared by all pages in the web application
- login.html: this file contains the HTML component of the login page (the login form)
- quote.html, quoted.html (redundant): these files contain the original quote function. quote.html contains the quote form, while quoted.html returns the information the user seeks.
- register.html: this file contains the HTML component of the register page (the register form)
- scripts.js: this file contains the JavaScript code used in the application
- sell.html (redundant): this file contains the form that is used for selling shares. Originally, the form had its own page ("/sell").
- stocks.html: this file contains the HTML component of the stocks page
- styles.css: this file contains the CSS elements used in the program

The list of sources used

While I cannot hope to name all of the sites and sources I have relied on while I was writing this program, I will try to name all I can from the top of my head:

- Tutorialspoint
- W3Schools
- Pretty Printed (YouTube)
- SQLAlchemy
- Python docs
- Flask docs
- Schedule docs
- Celery docs
- Stack Overflow
- Stack Exchange
- Reddit
- Quora
- CS50 docs
- GitHub
- Microsoft Docs (SQL)
- JQuery docs

I'm grateful for having all these amazing sources at hand. I feel that I have learned a lot while I was working on this project, largely thanks to the tons of quality information and tutorials available.