**Final project report**

**Web application: finance management**

Andrea Giordano  
Student ID: 2256322

Tongji University, academic year 2023-2024

Communication networks, spring semester

**Abstract**

This project consists of a personal finance management web application, implemented using the Python Flask environment. The application provides users means to securely track and manage their expenses and income. The front end, built using HTML and CSS, aims to be as simple and intuitive as possible, granting the users a great in-app experience. This project showcases the effective use of web technologies to create a practical tool for personal finance management.

**Introduction**

The project consists of the development of a personal management web application. The back end is designed using Python, and, for what concerns the web part, the Flask environment was used. To construct databases, SQL language was used (in particular SQL Alchemy and the SQLite Python add-on). The front end is designed using primarily HTML; to make the web page more appealing to the user, basic functions of CCS have been used.

On the application, each user can add income and expenses. Each transaction will be saved with some additional information, so that it will be easy to access them. It is possible to visualize all the expenses, and filter among them (by date or by category). It is possible to add a monthly budget; if it has been set, then the user would see the remaining budget and how much he should spend on the budget to stay in it at the end of the month.

An additional feature is to split expenses. Here each user can add multiple people and then insert expenses to share between them. The system will automatically display how much each of them has spent and will do the calculations needed to reimburse who has spent the most with the fewest number of transactions possible.

**Page management**

Each page has its own URL. The web application is not running on the open web, only on one device, so the root of the URL is <http://localhost:5000/>, 5000 is the default port used by Python Flask. It is possible to change both the URL and the port number in the Python script. The page URL is straightforwardly defined for each different page: to the port number, it follows a name which explains the main function done in that particular page. For example, the register page is defined as <http://localhost:5000/register>, and the dashboard as <http://localhost:5000/dashboard>. Most of the pages require the user to be logged in in order to be accessed, again, that is ensured by a Flask function.

While using the app, the user is asked multiple times to enter data. All is done by using FlaskForm. Each FlaskForm corresponds to a class. Inside the class, the different inputs must be defined. Each different type of input corresponds to a different field: the main field is StringField (the user enters data from the keyboard), SelectField (the user chooses data from a given set of possibilities), DateField (the user enters a date) and SubmitField (button that sends the data). When defining a field of a form, it is also possible to state the properties the inputted data should have; first of all, it is needed to specify whether or not it is mandatory to fill the field (InputRequired()), it is possible to set lengths requirements (both minimum and maximum. The function *render\_kw* is used to specify which type of object we are dealing with for what is concerning the front end, which allows us to add some custom attributes to the corresponding HTML class.

**Flow of information**

It is possible to run the application from the command windows: the CMD should be open on the path containing the app.py file, to run it, simply write py app.py (notice that it may be required to write python instead of only py depending on how the shortcut has been set up). Once the script is running, on the command window it will be possible to see the URL at which the page is reachable, as well as if the debugger mode is on (if it is, the page will automatically be updated as the script is modified. When we access each page, on the command window it is possible to see the user request and system reply. On the command window, we will see the user IPv4 address (since on local devices it is by default 127.0.0.1), the date of the request message (day-month-year, time of request), the method of the request (GET or POST) HTTP version (so HTTP/1.1) and status code (the more common are 200 ok, 304 not modified (since the last visit time: it is saved in cache). On the command window, we will also see: “"GET /favicon.ico HTTP/1.1" 404 –”, 404 is the HTTP message for URL not found, that is because I did not create a favicon.ico file (this is the icon of the web application displayed on the left of the browser tab. The browser automatically searches for one icon to display and, since none is found, it will raise an error. Even though this error is displayed multiple times (one time each time a new tab is open) the web application will still work fine. The decision not to add an icon comes from the fact that the scope of the project is to learn how a web application works, not to make it appealing. When requesting a page, two messages will be displayed on the CMD, both are of GET type. The first one is to actually be redirected to the correct URL (for example: “"GET /dashboard HTTP/1.1" 200”) the second one, once the HTML is read, is to get the .css file associated with the current page (in the case of the example: “"GET /static/dashboard.css HTTP/1.1"”).

It is not possible to see the actual content of each HTML message directly from the CMD, it is necessary to use third-party software, for example, WireShark, with the Windows *npcap* packet installed, which allows capturing packets on the local host network. If we set up this environment, we will, of course, see the same message exchanges; this time, though, we can also see the contents inside the message. We learn in this manner that all the data is sent in clear text, password included, so the website is not secure. I tried to use HTTPS (HyperText Transfer Protocol Secure), but in order to use this protocol a certificate of security is required.

**Data storage**

To store the data required I have used two different structures. I have created a database, called user, that stores three different fields for each user: a user ID (string), username (string) and password (hashed string, explained in more detail below).

For all the other data that I need to save (expenses, income, category…, I explain each in a further detailed way each of this type of data later on) I used JSON (JavaScript Object Notation) file. I chose to use JSON files instead of creating one database for each set of data for various reasons, mainly for the easiness of writing and modifying the data stored. Data in JSON file is stored like the file is one big dictionary: the key is the user ID, and the value, depending on what file is being read, can be either a list or another dictionary. The .json files are saved in the root directory, and their path is declared as a global variable. Two functions, namely write\_to\_file and read\_from\_file are used to respectively write and read the files. Both functions take as argument the file path (so the same function may be reused to write in all the different files), the write function takes as argument also the data to rewrite to the file. Notice that each time the file is rewritten, all the data is erased and then rewritten, that to ensure that the new changes are correctly saved. Since the same data may be used in different parts of the code, I believe it is more straightforward to declare the data read from each file as a global parameter, rather than each time having to read the data again.

**Login and registration** (<http://localhost:5000/login>; <http://localhost:5000/register>)

To register to the web application, it is required to pick a username and a password. There are requirements on the length of both the username and password: username must have at least 4 characters, at most 20, for the password the minimum requirement is 8 characters, and the maximum stay unchanged. The system will automatically generate a unique user ID, that will be used to link the data inputted by the user to the user itself.

The system will check if the username does not already exist (majuscule and minuscule letters count as different characters). The password will be hashed before being saved in the User database. To hash the passwords, I have utilized the *bcrypt* functionality of *flask\_bcrypt*.

Since the application protocol the application uses is HTTP/1 the password is transmitted as clear text, hence visible to someone who is sniffing the network. The only way to solve this problem is to use HTTPS, but to do so one must have a license configured, which I had not. Once the user has correctly registered, he will be redirected to the login page, here, after completing the login, he will access the dashboard page.

**Dashboard** (<http://localhost:5000/dashboard>)

This is the page where the user is redirected once the login is successful. As for all the following pages I am going to describe further on, it is only reachable after successfully loggin-in, if one tries to access this page without having logged in the browser will raise a page not found error. The dashboard is the page of my web application that manages all the other pages, like a menu. It consists of two distinct parts: on the upper part of the page, other than the name of the current user, it is possible to see a short recap of the main functionality of the web application. That includes the income of the current month, the total expense of the current month, the monthly budget, the remaining budget, and the maximum average daily expense the user should stay under budget.

The other part of the screen is occupied by the links that redirect the user to the different pages of the web application. On each of the other pages, there will always be (at the far bottom of the page) a link to go back to the dashboard. The last link that the user sees is the link to log out from the web application.

**Add expense and category** (<http://localhost:5000/add_expense>)

The main functionality of a finance management tool is to have the possibility to enter all the transactions one does. Saving, other than the amount and the currency of the transaction also other data may be useful when looking at all the transactions done. That is why the user should also input when the transaction has been made, the category of belonging and a short description of the transaction.

The users can add a new category and a new expense on the same page. This is particularly convenient since the category of the new expense must be selected from the category the user has set. To define a new category is sufficient to choose its name, which shall have at most 20 characters (those can be whichever the user wants, there are no controls on what the user enters, even if the user enters twice the same name, no error will be raised). All the categories are saved in the category.json file; in this case, the value of each dictionary is just a list, that lists all the categories of the user.

On the top of the page, all the categories that the user had set are visualized, as a list. At the same level of each category name, there is a delete button. By clicking on it, the category corresponding to the clicked button will be removed from the possible choices.

Let’s analyse in further detail the field that the user must input before the “AddExpense” form is completed. Notice that all the following fields are mandatory. The fields the user should complete are as follows:

* **Amount**: specify how much the transaction consists of.
* **Currency**: it is selected from a list (defined as a global variable in the script) that contains some of the principal currency in the word: ¥, €, £, $.
* **Description**: up to a maximum length of 100 characters, is used to have more details on the transactions.
* **Date**: in the European format (dd-mm-yyyy), is used for budgeting functionality.
* **Category**: chosen from the category the user had defined.

All the expenses are saved in the expenses.json file, here, the structure is a little more complicated than the structure of the category.json file. As a matter of fact, here the value is another dictionary, with several keys, to each key is associated with a single value (string). The keys of this dictionary, with only one exception, are the field of the form explained above. There is one more key, this is a transaction ID that the system assigns automatically to each expense, and it is never seen by the user. It is important (we will see why later on) that the transaction ID is unique so that each transaction is distinguishable from the others.

When an expense has been correctly added, the user will be redirected to the dashboard.

**View expenses, filtering options and delete expense** (<http://localhost:5000/view_expenses>)

On this page, all the expenses the user has registered are shown. These will be automatically divided into different paragraphs, each paragraph corresponding to one month. Each expense occupies one line, at the end of which, a button to delete the expense is found. The functionality of the button, as well as the code used for the delete category bottom. Here it is possible to get why inside the expenses.json file I have saved also a transaction ID. In this manner, it is possible to recognize which transaction the user would like to remove from the list. When a transaction is removed, then we must rewrite all the expenses back in the JSON file. Being that each expense is defined by the same parameters, when the expenses are shown on the screen, the user always sees them in the same format, that is as follows: date (in Chinese format yyyy-mm-dd), amount, transaction currency, description and finally category (this is shown inside brackets).

On the top of the page, the user can see a form, this is to add a filter to the data visualization. It consists of two fields, none of which is mandatory to complete. The fields are as follows:

* **Category**: the user will see only the expenses belonging to the chosen category. One category may be selected at a time, once again the possible choices for the category come from the categories that the user has set previously.
* **Start date**: the user will visualize only the expenses done after the selected day.
* **End date**: the user will visualize only the expenses done before the selected day.

**Set monthly budget** (<http://localhost:5000/budget>)

In this page, the user can set a monthly budget. If one is already set, but the user nevertheless compiles the form, the former budget will be substituted by the last user entry. Budgets are saved once again in a JSON file (budget.json). This time the value stored is simply a string containing the budget (notice that it is a string and not a float number just because in the FlaskForm the field is a StringField, so the data that contains is a string.

Other than the form to set a monthly budget, on the budget page, the user sees the current monthly budget and a list with the total he has spent in all the past month. It will be specified if the user has spent less or more than the monthly budget, in one case the user will see: “remaining budget”, followed by the amount of the unspent budget, in the latter, “overspent”, followed by how much the budget was exceeded. The logic to change the screen output is in the HTML file. This file, namely budget.html contains both a for loop (that goes through all the month) and an if that checks whether the remaining budget is positive, so the user has saved money, or it is negative, so the user has overspent. I also have defined a function in the main Python script, namely absolute, to use in the HTML files. As the name suggests it is used to calculate the absolute value of the input arguments. This is useful so that the amount of the overspent is shown as a positive value and not as a negative number, how it is saved in the files. The user will also see for all the month, the average daily expense. For the current month it will also be visible how much he can spend each day to remain exactly within the budget (that is with a final balance of 0 saved).

**Add income** (<http://localhost:5000/income>)

There exists a dedicated page to add income sources. Once again it consists of a FlaskForm. Again, the user will be asked to fill in some required information, quite similar to the one that is asked when adding a new expense:

* **Amount**: specify the amount the user has received.
* **Currency**: it is selected from a list (defined as a global variable in the script) that contains some of the principal currency in the word: ¥, €, £, $.
* **Description**: up to a maximum length of 100 characters, specify the nature of the income source.
* **Date**: in the European format (dd-mm-yyyy), is used to have a sum up of all the income sources for each month.

Other than the form, below it, there is a list containing all the different income transactions the user has saved, it is shown before the date of the transaction, in the format yyyy-mm-dd, amount and currency and finally the additional description.

**Split expenses** (<http://localhost:5000/shared>)

A functionality that often is not included in any financial management tool, but nevertheless is quite useful, is to evenly split expenses between groups. That may be particularly useful when travelling within big groups for several days. I include such functionality in my web application.

When the user opens the split expenses page, he will see two different forms: one is to add people to the group he is splitting the expenses with, and the other is to add the expenses.

The user should include himself in the group of people (the username he chooses may be different from the username with which he logs in to the web application (if he does not include himself, the system will not recognize him as one of the people participating in sharing the expenses, since this functionality does not retrieve any information about the current user).

The user can see all the people with whom he currently has shared expenses, just after the form. There is another form on this same page, this is the same form to add an expense. It is just like the expense form (without the description), of which I have already discussed above (see add expense) with just one more field. This field is denoted as **paid by**, as the name suggests in this field the user should specify who has paid the expenses he is adding. The user can only choose among the added people to fill this field. The added expenses are shown after the form. Like in the view expense page, it is first displayed the date, then the amount and the currency, the category of belongings and finally who has paid the expense (this is written in bold, so is the first thing one sees. Other than showing all the expenses, scrolling down the page the user can see:

* The total expense for each friend, this is how much each friend has paid in total.
* The balance, which states the money that each person in the group should receive (if have paid more than the total expense over the number of people) or pay (if he has paid less)
* The settlement, where the system specifies all the transactions that need to be done to balance all the expenses the group has done.

At the very bottom of the page, there is a “clear everything” button, by clicking on it the logged-in user can delete all the information on the page (that is both the list of friends and the list of transactions). This is needed to have the possibility to create another shared expense with another group. Notice that at the current state of things, each user can create only one group at a time and that the data, once deleted, cannot be restored in any manner.

**Conclusion**

This project demonstrates the development of a web application, trying to build a personal finance management system. The project has been carried out using Python Flask (back end), HTML and CSS (front end), SQL and JSON (data storage).

During the development process, several challenges were encountered. Using hashed passwords was crucial to protect user data, although the absence of HTTPS posed a security limitation since the data is sent as clear text. The decision to use both databases and JSON files for the storage of different data, while facilitating data manipulation, also highlights the trade-off between different data storage solutions.

This project has been very useful in improving both knowledge in different programming languages and of the internet structure. Future improvements can focus on implementing HTTPS, optimizing the storage and perhaps adding more functionality.