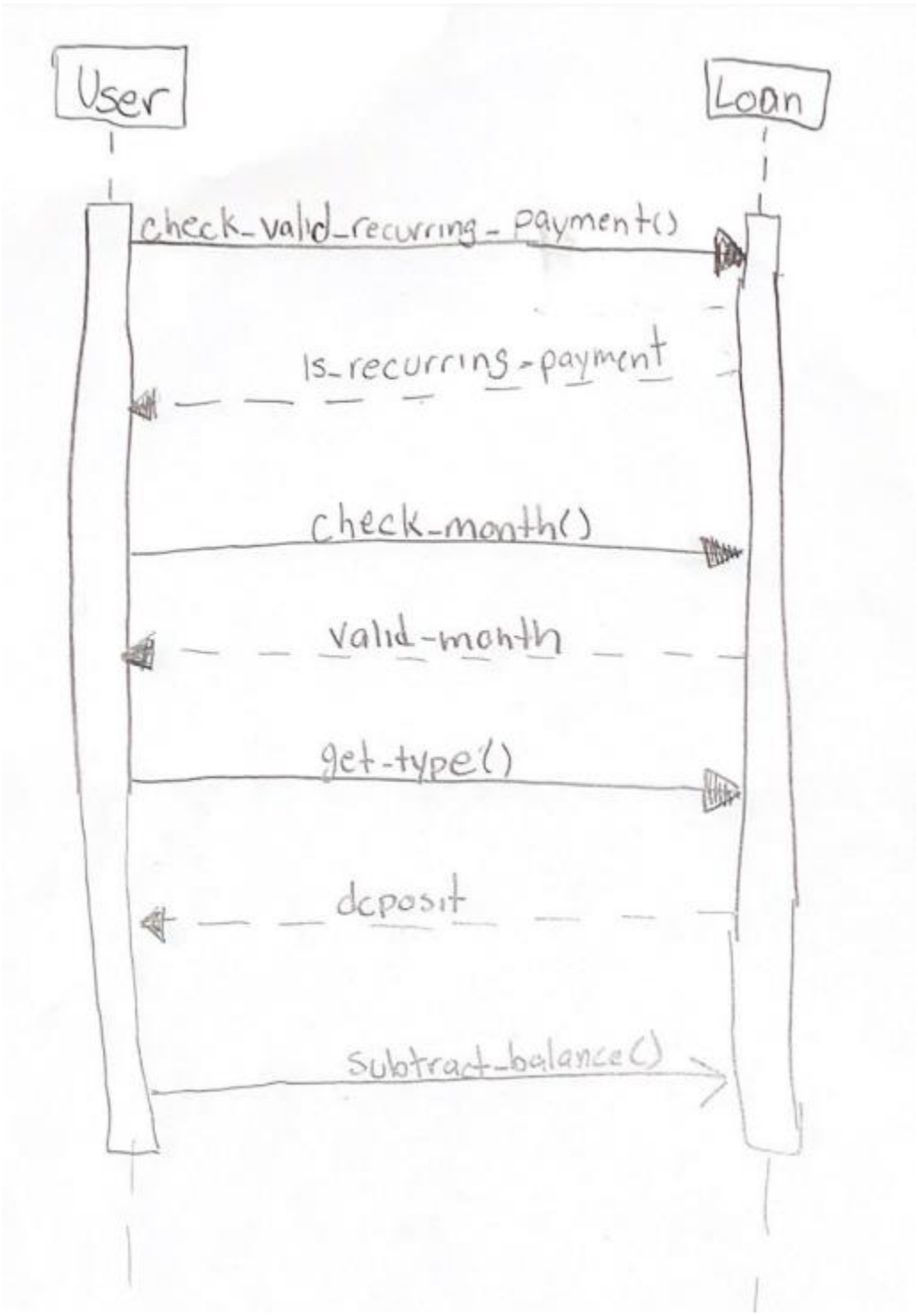
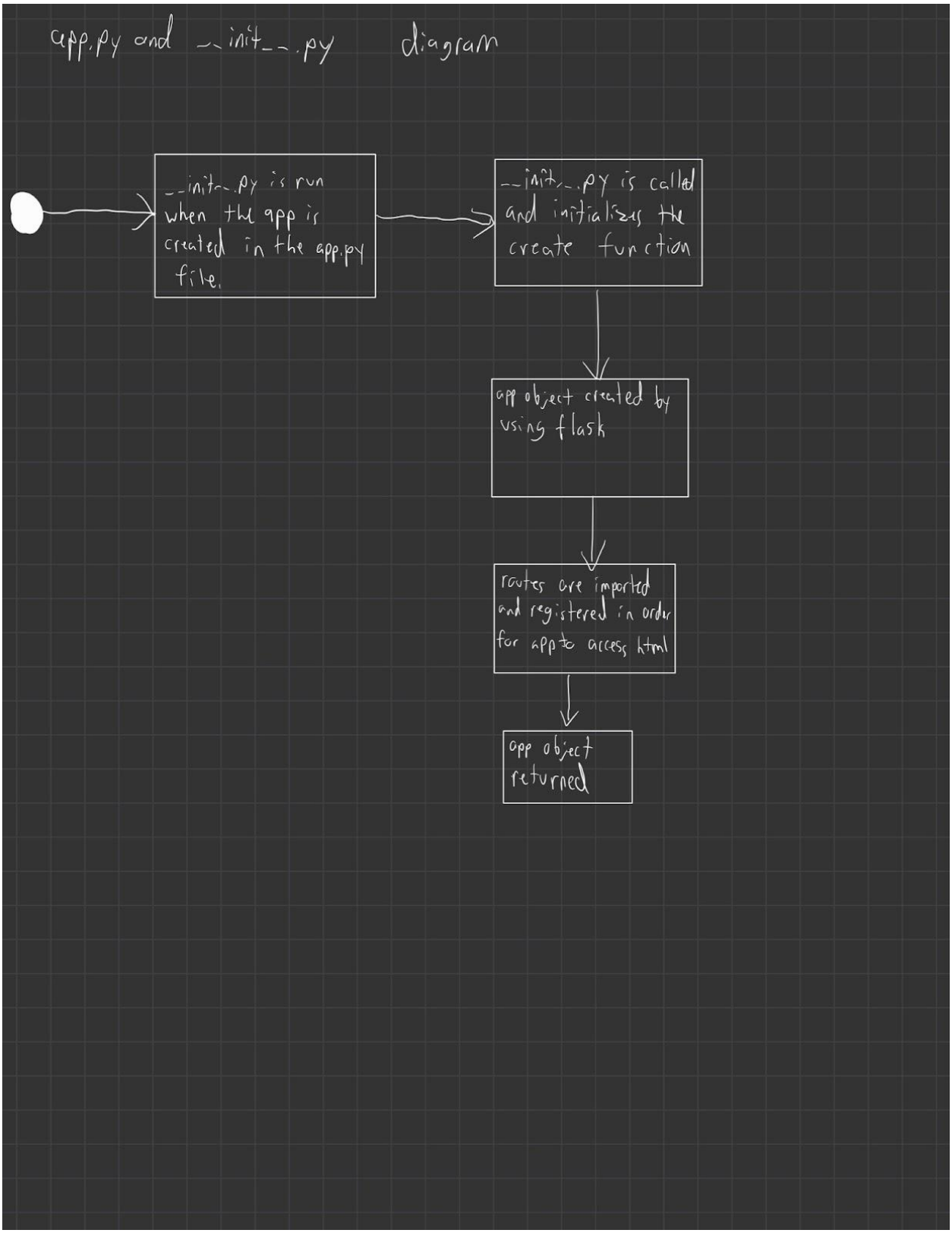
The goal of this project is to create a program that allows a user to login to their account and manage their finances; they can make deposits, withdrawals, have loans, and recurring payments. The following activity diagram shows how the user class interacts with the loan class. The implementation of 3.4.2.3 is demonstrated through the interactions of the loan and user classes. Through this interaction the user class has access to the loan data through methods in the loan class. The user class will be looping through the loan list to see if any are also a recurring payment and then if it is the valid date to perform the transaction. This will keep the user's data accurate and up to date.

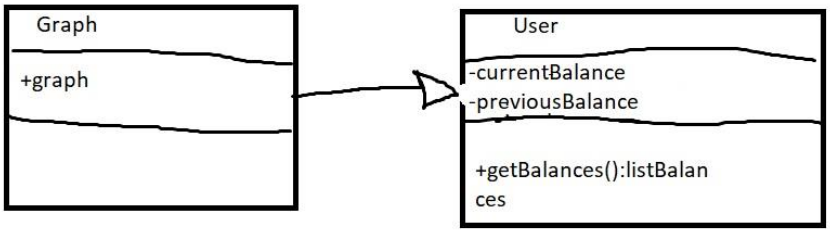

User
Loan
check_valid_recurring_payment()
is_recurring_payment
check_month()
valid_month
get_type()
deposit
subtract_balance()

The following is an activity diagram showing how the app is created and the background for html is set up. First app.py is run which in turn runs __init__.py which defines the function needed to start the app. Firstly the app is created using Flask which allows the program to not just run in the code editor but instead on a website. Then the routes are established, allowing the website to have different tabs such as /home, /login, and many more. Then finally, the app object is returned to the app.py function so the

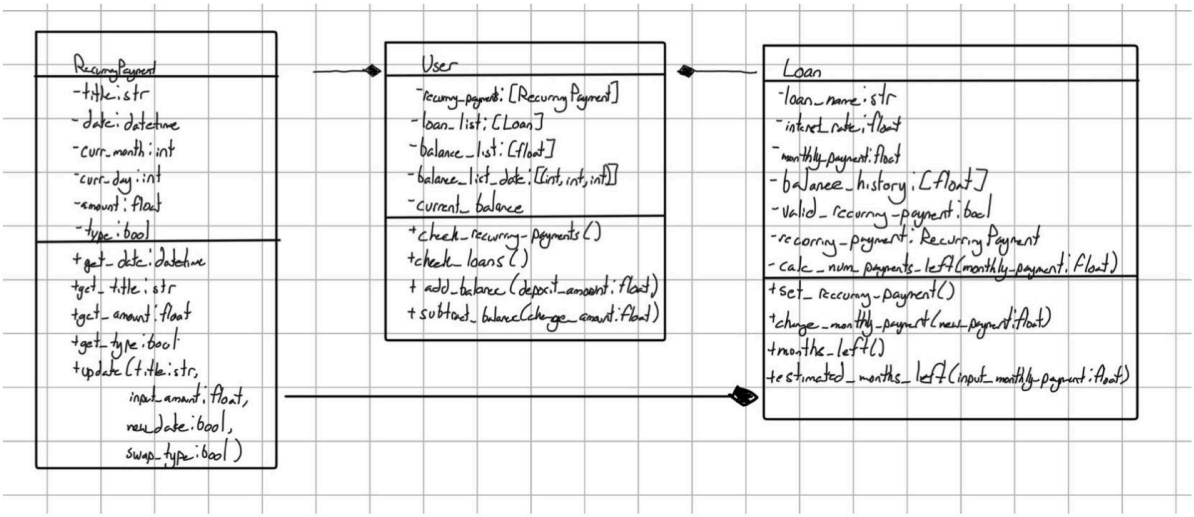
website can run.



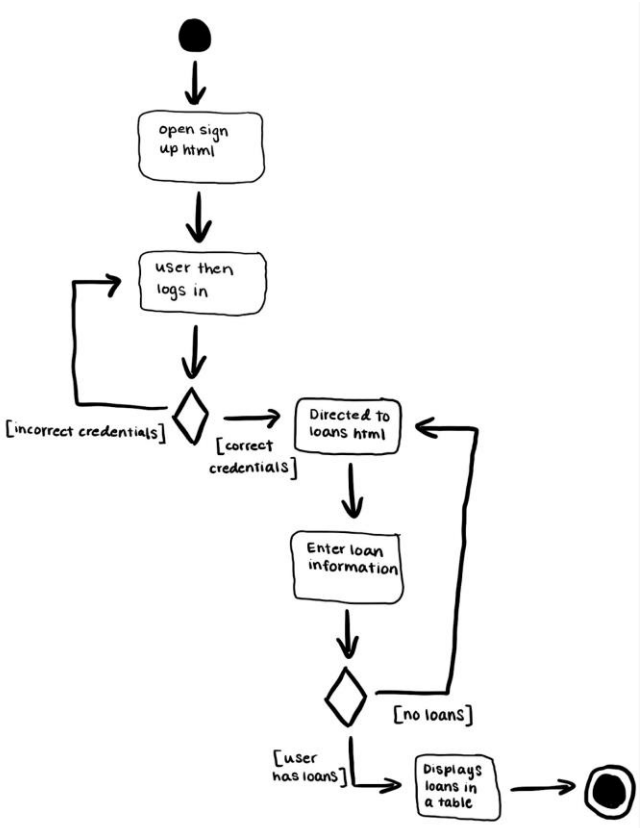
The diagram is for the graph file of the Money Management App project. It gives a visualization of the user's balances throughout time. The graph class relies on the user class to get the list of balances of the users throughout time.



Nathyn Tran Team H
Money Manager

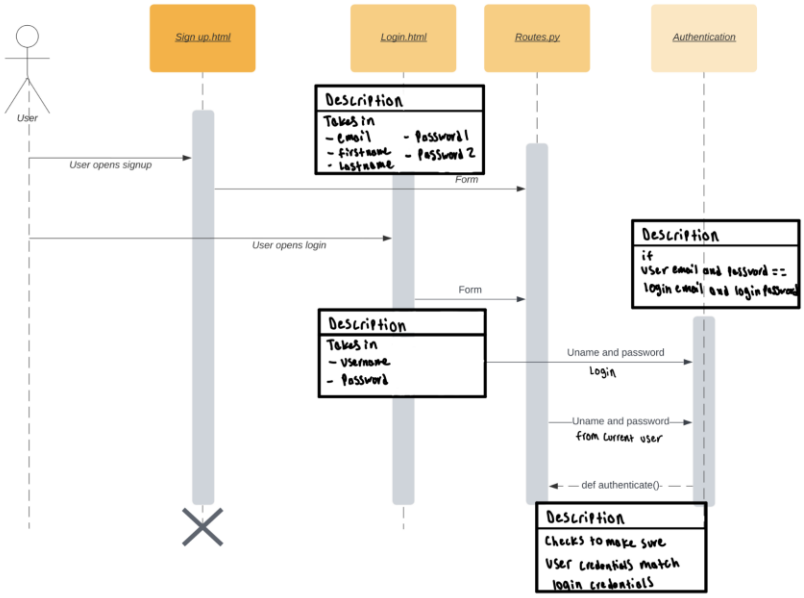


In order to explain the implementation of how the Recurring Payment and Loan classes work with the User class (which refers to SRS 3.4.3.4 and 3.4.3.5), I created a class diagram to show the design of the class relationships. The User class can contain an array of loans and recurring payments, so the user can have multiple types of loans and different types of recurring payments. The recurring payments could either be monthly deposits or charges, such as monthly paychecks and online subscriptions. Both the recurring payments and the loans have a composition relationship with the User, because the User contains both. The Loans can possess a recurring payment, which shows the composition relationship between Recurring payments and Loans. The user can check both the recurring payments and the loans on whether the required date has passed for the monthly charge/deposit using the functions “check_recurring_payments()” and “check_loans()”. These will iterate through both arrays of the loan_list and recurring_payment list and determine whether the current month is one greater than the previous, and check if the day is the same. If the date is correct, it will then proceed to subtract the amount of the recurring payment from the balance of the user. If the loan has a valid_recurring_payment, it will change the balance and the loan by subtracting the monthly payment from the balance.



I chose an activity diagram to demonstrate how the user gets to the loan web page. This shows the steps of how the user signs up and then logs in and what information pops up on the loan web page. The login and sign-up html pages have forms to collect the user's information. This information goes into the routes file to determine if it is accurate or not. From the routes file the sign-up data uses the setter function to take the users credentials from the signup form and put it into the user object. The login information works similarly but instead of setting it to a user object it is stored in the routes file locally. The authentication will compare the username and password the user inputted with the username and password. If the login is successful, it will take the user to the loan's html page. If the login is unsuccessful, it will redirect the user to input their information again. The loan html page will be where the user can enter their loan information including loan name, loan amount, loan term and interest rate. When the user submits their loan, it will display the information in the form of a table. To make the table we use Jinga to loop through the data inserted into a loan object in order to display it.

Signup/Login Sequence Diagram



I chose a sequence diagram to represent the Sign up and login process on our web app. The reason I chose this is because the sequence diagram did the best job of showing how the login, signup and authentication work on our website. The sign up and login html pages are both html pages that have forms on them that allow the user to input data into the the routes file. The signup and login pages both use post HTTP requests which the routes file is able to interpret. From the signup html page, the users credentials entered from the signup form using HTTP post methods and copies it into the currentUser object using the setters of the User class. It is then stored to the global currentUser object. The login information works similarly but instead of setting it to a user object it is stored in the routes file locally. Then the authentication compares the user username and password with the username and password taken in from the login. If the user login is successful the user will get a message that says “successful login” and the user will be directed to the home page but if not then it will say “invalid login credentials” and stay on the login page.

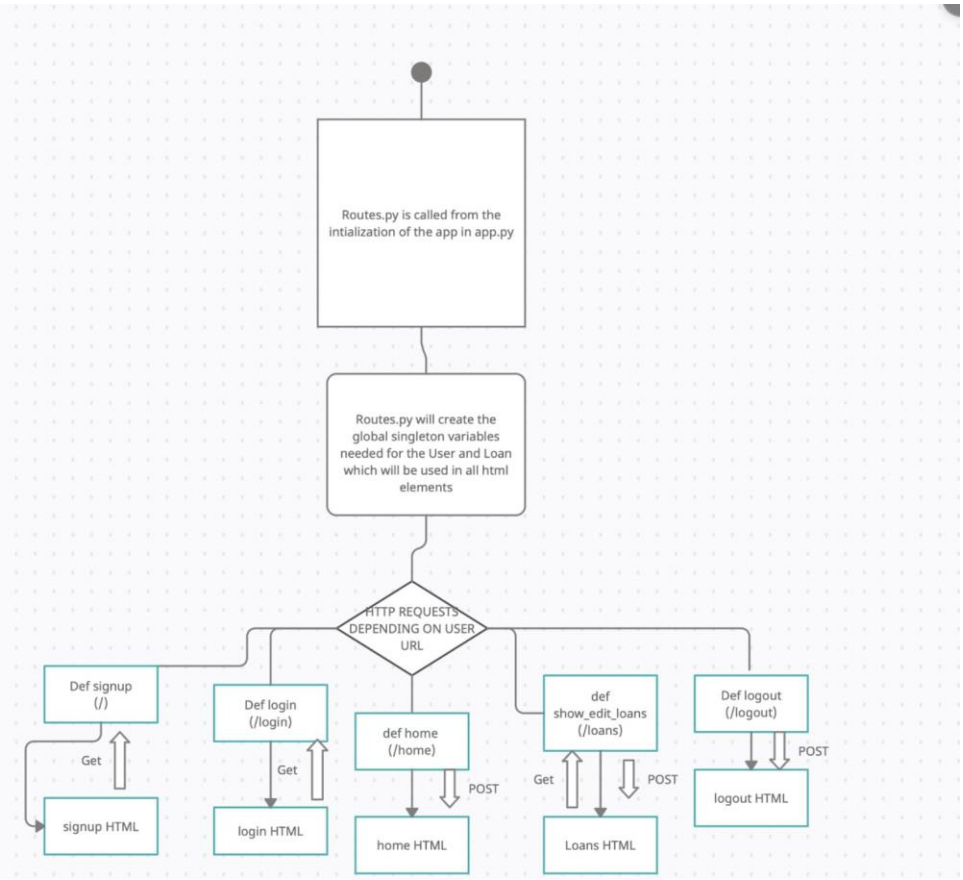
else:

    flash("Successful login", category="success'")

if not valid_login:

flash("Invalid login credentials", category='error')

Routes.py is the so-called “Brain” of our application. Since we are utilizing the Singleton design methodology, Routes.py is where all of our needed single, global, instances of our objects are created. The main purpose of Routes.py is to define what happens when a user enters a specific URL. For instance, if their URL ends with /login, they are directed to the login method, in which some logic will be conducted (which is referenced in other documentation within our design documentation) as well as a redirect to its corresponding HTML. Each page will also indicate the type of HTTP requests the method and page will be conducting.

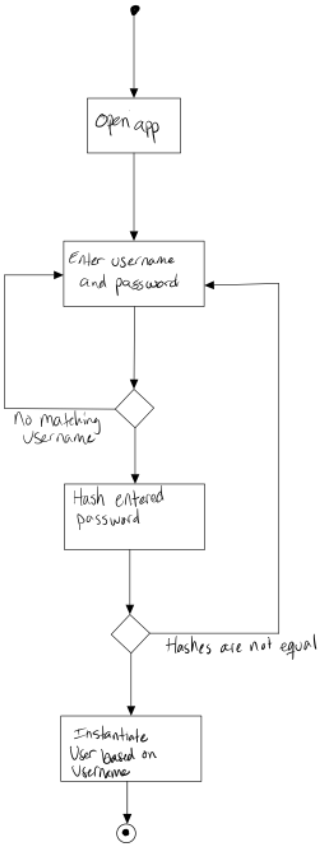


In the corresponding method in routes.py, we can see examples of how POST logic is conducted when the user activates the method by using its corresponding route of ‘/login’. We can also see how the method will return its corresponding HTML file using the render_template method provided by Flask, which looks for the corresponding file in the templates directory, and then renders that desired HTML. We can also pass in our Singleton variables, such as currentUser, in order for the object to be integrated with HTML/Python “Jinja” code in the template. This way the object can be affected by the corresponding post and get requests in the code logic. It is also important to note that whenever a post or get action is conducted on the client, the page is “refreshed” in order to re-call the method by utilizing the URL path again. So say if a user hits submit on the page corresponding to moneymanager.com/login, the page will refresh under the login, so the login method will be called again, but this time will have new

information to pass to the method with its new POST request.

```python
@routes.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        email = request.form.get('email')
        password = request.form.get('password')
        print(currentUser.get_email())
        valid_login = authenticate(email, password)
        if not valid_login:
            flash("Invalid login credentials", category='error')
        else:
            flash("Successful login", category="success'")
            return redirect('/home')

    return render_template('login.html', currentUser=currentUser)
```



I chose a sequence diagram to model how authentication works in our app. It starts with the user opening the app. After the user enters their username and plaintext password, the app checks if there is an account with the username that the user entered; if there is no matching username, the user is prompted to try again. If the username matches a known username then the app will hash the user's entered plaintext password and compares it to the known password hash; if the two hashes do not match then the user is prompted to try again. If the hashes are equal, then a new user object is created with the data related to the user that just logged in.