Expense Tracker (Desktop App)

Name: Rhythm Paudel

Email: rhythmpaudel12@gmail.com

Github: github.com/rhythm-paudel

LinkedIn: linkedin.com/in/rhythmpaudel

Abstract

This documentation will provide a brief overview on a desktop application which can be used to track the financial change of the user. The user can perform various operation and view an overview of the cash flows, that were entered in the account. The application will deal mainly with inflows, outflows, and debts services.

Table of Contents

1. Introduction 1

2. Features and functionalities 2

2.1. Registration 2

2.2. Login 5

2.3. Transaction management 8

2.3.1. Adding inflows 8

2.3.2. Adding debts 10

2.3.3. Adding outflow 10

2.3.4. Clearing debts 11

2.4. Filtering transaction 12

2.5. Dashboard statistics 13

2.6. Pending debts 14

2.7. Logout 15

3. Proof of work 16

3.1. Designs 16

3.1.1. Data entity model 16

3.1.2. Class diagram 17

3.1.3. Use case diagram 18

3.2. Version control system 19

3.3. Testing 21

3.3.1. Register 21

3.3.2. Login 24

3.3.3. Adding Inflow 25

3.3.4. Adding debt 28

3.3.5. Adding outflow 30

3.3.6. Outflow Test 2 34

3.3.7. Searching and filtering 37

3.3.8. Dashboard testing 41

3.3.9. Insufficient balance outflow testing 43
3.3.10. Clearing debt testing 44
3.3.11. Insufficient balance to clear debt 46
4. Individual Reflections 48
4.1. Roles and responsibilities 48
4.2. Challenges faced 48
4.3. Learnings and growth 49
4.3.1. Technical skills 49
4.3.2. Problem-solving skill 49
4.3.3. Project management 49
4.3.4. Documentation 49
4.4. Impact on future work 50
5. Conclusion 51
5.1. Key findings 51
5.2. Limitations 51
5.3. Future work 51
6. Appendix 53
6.1. Code snippets 53

Table of figures

Figure 1: Login page 2
Figure 2: Registering a user 3
Figure 3: Userdetails.json 3
Figure 4: Existing user error message 4
Figure 5: Not registered user message 5
Figure 6: Incorrect login password 6
Figure 7: User session saving code snippet. 7
Figure 8: Dashboard (1) 7
Figure 9: Dashboard (2) 7
Figure 10: Adding transaction form. 8
Figure 11: Transaction saved message. 9
Figure 12: Debt form filled up. 10
Figure 13: Adding outflow 11
Figure 14: Clearing debt 12
Figure 15: Search feature 13
Figure 16: Dashboard summary. 14
Figure 17: Dashboard filtering. 14
Figure 18: Pending debts highlight 15
Figure 19: Entity relationship diagram 16
Figure 20: Class diagram 17
Figure 21: Use case diagram 18
Figure 22: Github repository (1) 19
Figure 23: Github repository(2) 20
Figure 24: Locating registration page 21
Figure 25: Registration of new user 22
Figure 26: User registered in Json file. 23
Figure 27: Logging in 24
Figure 28: Dashboard redirected after login 25
Figure 29: Error message due to incomple form details 26
Figure 30: Transaction detail for inflow 26

Figure 31: Inflow saved message 27
Figure 32: Update on Json file after inflow added 27
Figure 33: Updated dashboard after inflow added 28
Figure 34: Get debt transaction. 29
Figure 35: Get debt transaction saved message 29
Figure 36: Updated dashboard after adding debt. 30
Figure 37: Updated Json file after adding debt 30
Figure 38: Before cash outflow (Json file). 31
Figure 39: Filled form for outflow 32
Figure 40: Outflow transaction message 33
Figure 41: Updated Json after outflow 34
Figure 42: Debt addition (2) 35
Figure 43: Updated Json after adding debt 36
Figure 44: Outflow for test 36
Figure 45: Update in available to pay 37
Figure 46: Show all transaction result 38
Figure 47: Cleared debt search filter 39
Figure 48: Cash Inflow search filter 39
Figure 49: Date filtering 40
Figure 50: Dashboard before filtering 41
Figure 51: Dashboard with date filtering (1) 42
Figure 52: Dashboard with date filtering (2) 42
Figure 53: Large amount outflow 43
Figure 54: Low balance message. 44
Figure 55: Clearing debt 45
Figure 56: Json before clearing debt 45
Figure 57: Debts cleared message 46
Figure 58: Json after clearing debt 46
Figure 59: Error message while clearing debt 47
Figure 60: Logic for maintaining user log in session 53
Figure 61: Logic for filtering transactions 54

Figure 62: Inflow logic 54
Figure 63: Outflow logic..... 55
Figure 64: Clearing debt logic 56

1. Introduction

This application is being developed to track a financial status of a user. It can be seen that, people often lose track of their saving, expenditures, or other transactions like debts. With problems like this, sometimes people also forget about the deadlines of debts. This desktop application is being developed to solve such problems.

In this application a user can register with their preferred currency, making them easy to track their expenses. User will also be provided with a dashboard upon logging in, with brief summary of the transactions, their inflows, outflows and debts. Additionally, user will also be able to filter transactions by date from the dashboard. The application will facilitate the user with various operations like adding and clearing debt, adding inflows and adding outflows. User can also perform searching operations, filtering the transactions by date, and sorting by date.

This application will be developed using C# programming language, which will use .Net as the framework. In .Net, MAUI application will be prepared from which html tags can be used for designing and C# coding can also be performed in the same file with the help of .razor file. The details of any operations or user will be saved in Json in local machine for ease of using the file and dealing with Json.

2. Features and functionalities

The application will have several functionalities that will help the user to simplify the user. The application is developed with a simple UI avoiding overcomplexities for the user. The application will have several functionalities like keeping track of transactions, and filtering transactions. The list of functions in the application are listed below with further explanation.

2.1. Registration

The user is able to register with a unique username, their selected password and preferred currency. Once the preferred currency is set, the user can directly login with their entered credentials.

Upon first opening the application user will be interacted with a login form. If there are no user, a new user can be registered.

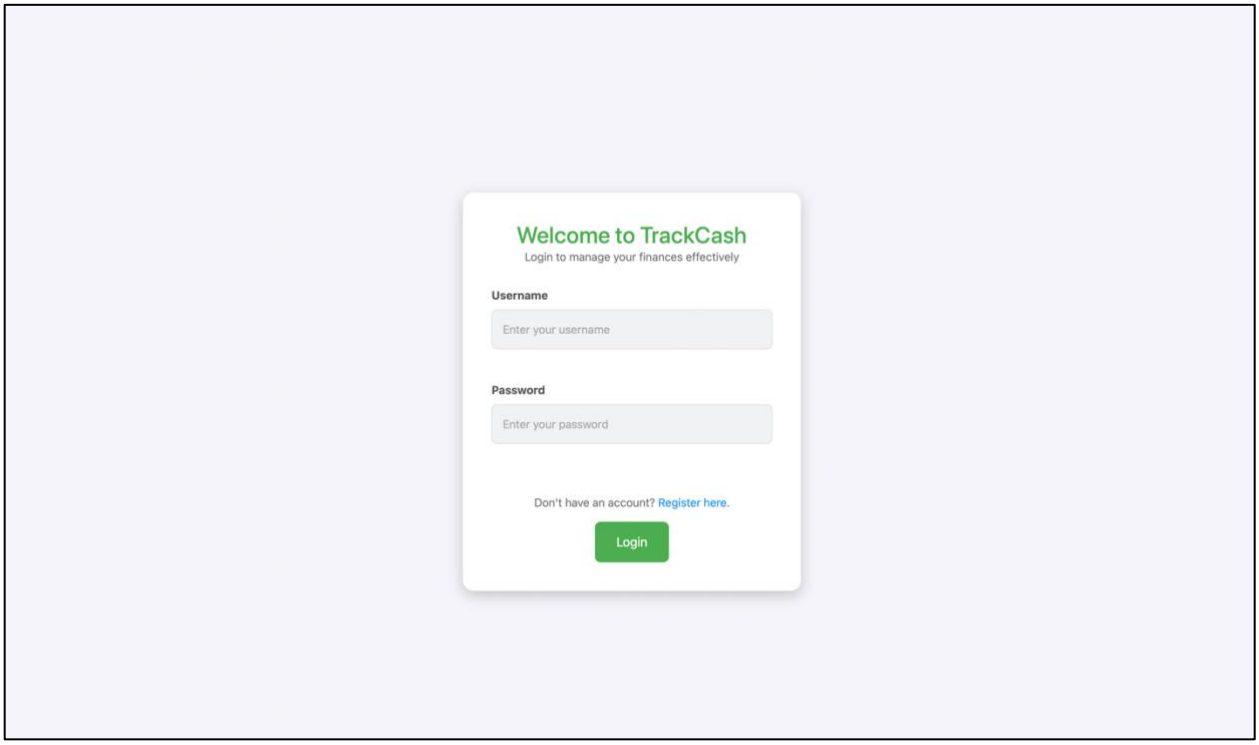

Figure 1: Login page

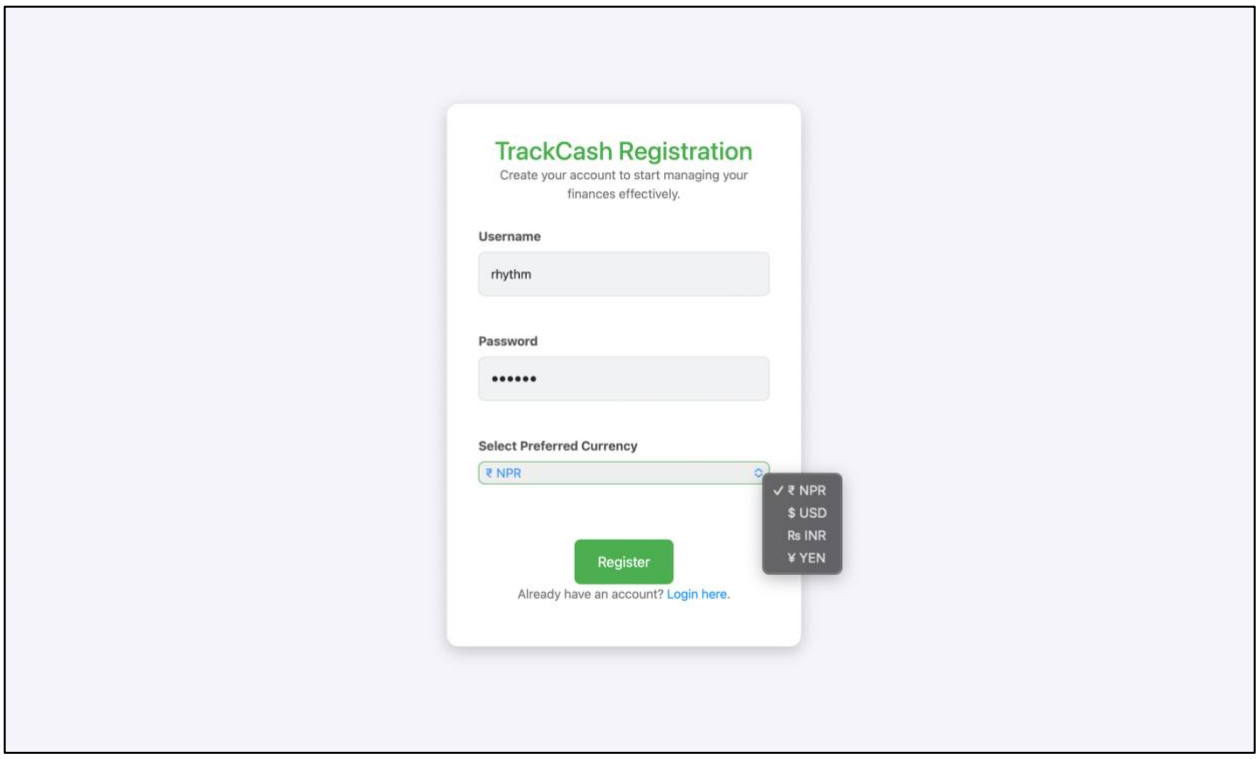

Figure 2: Registering a user

It can be seen that recently registered user has been stored in UserDetails.json. The preferred currency is set as code as the symbol “₹” is not supported in the text file. The password also has been hashed while saving the user.

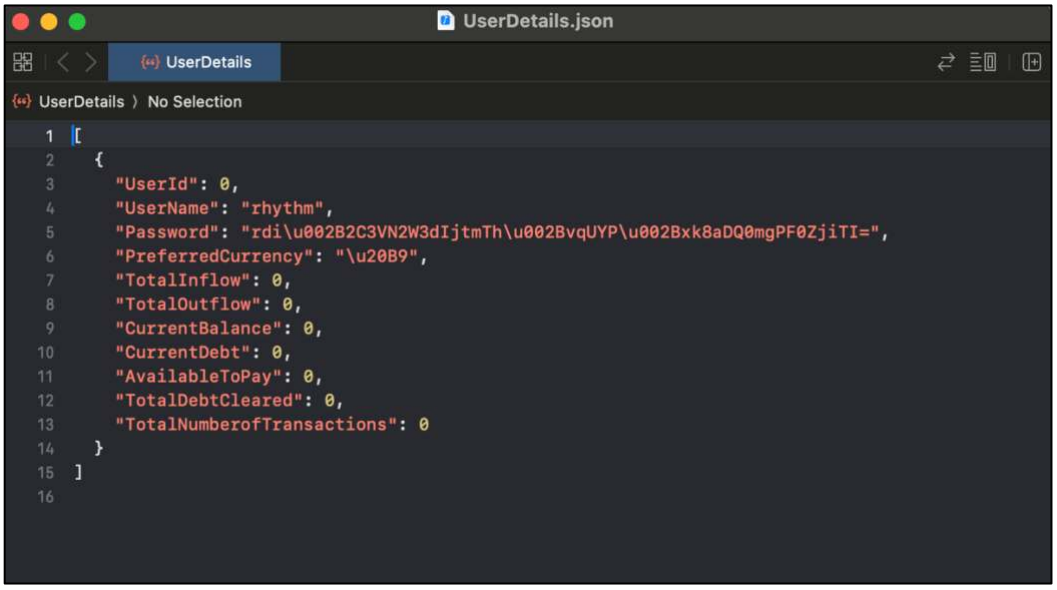

Figure 3: Userdetails.json

After the user is registered it this then redirected to the login page. However, the screenshot shown above only shows the status of user being saved. If user with same username is found a message is shown.

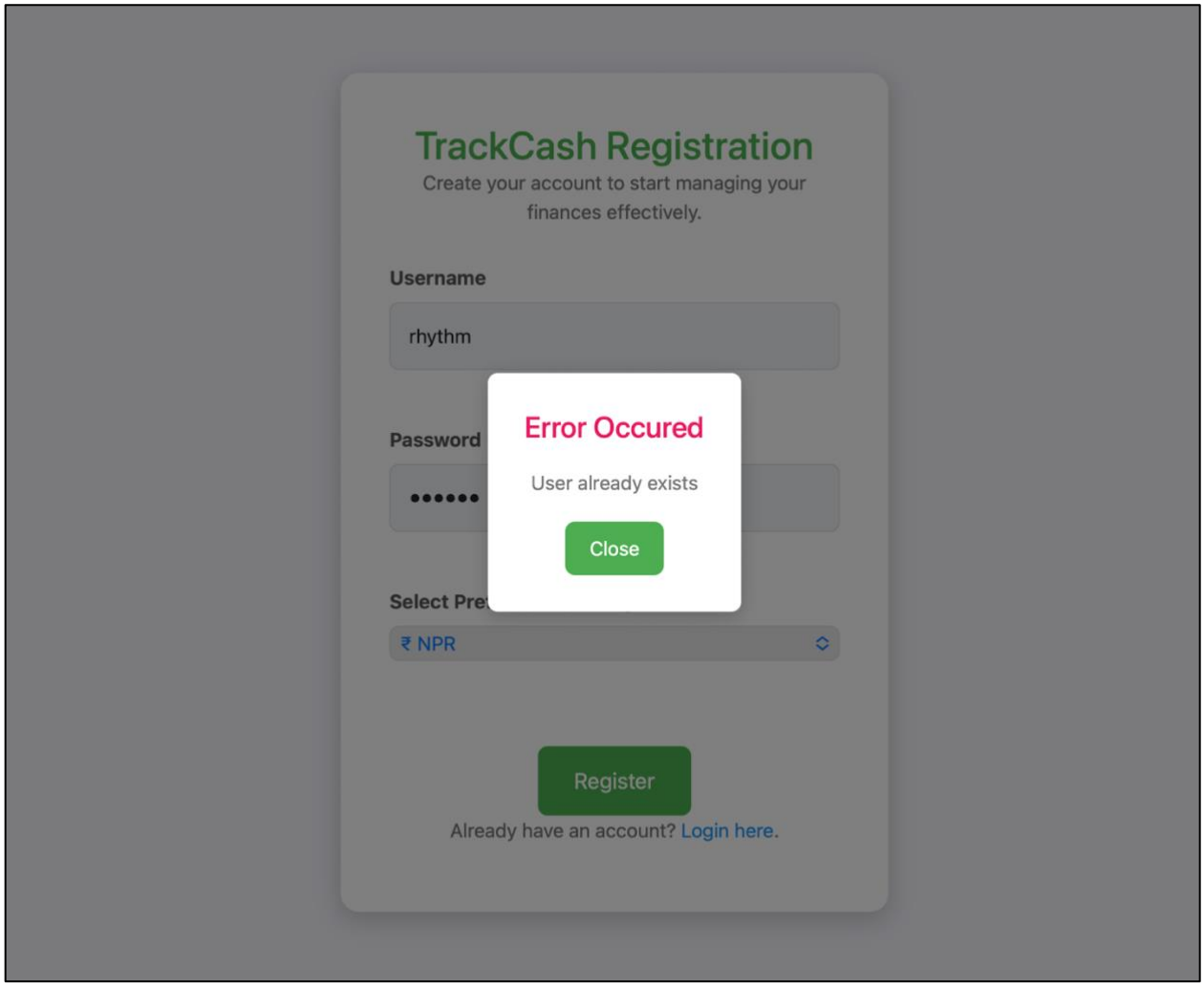

Figure 4: Existing user error message

2.2. Login

After the user is successfully registered, user can login with their username and password. If the password user entered do not match with the password saved in the Json file, user will get an error message. Similarly, if user tries to login with unavailable user will get a prompt accordingly.

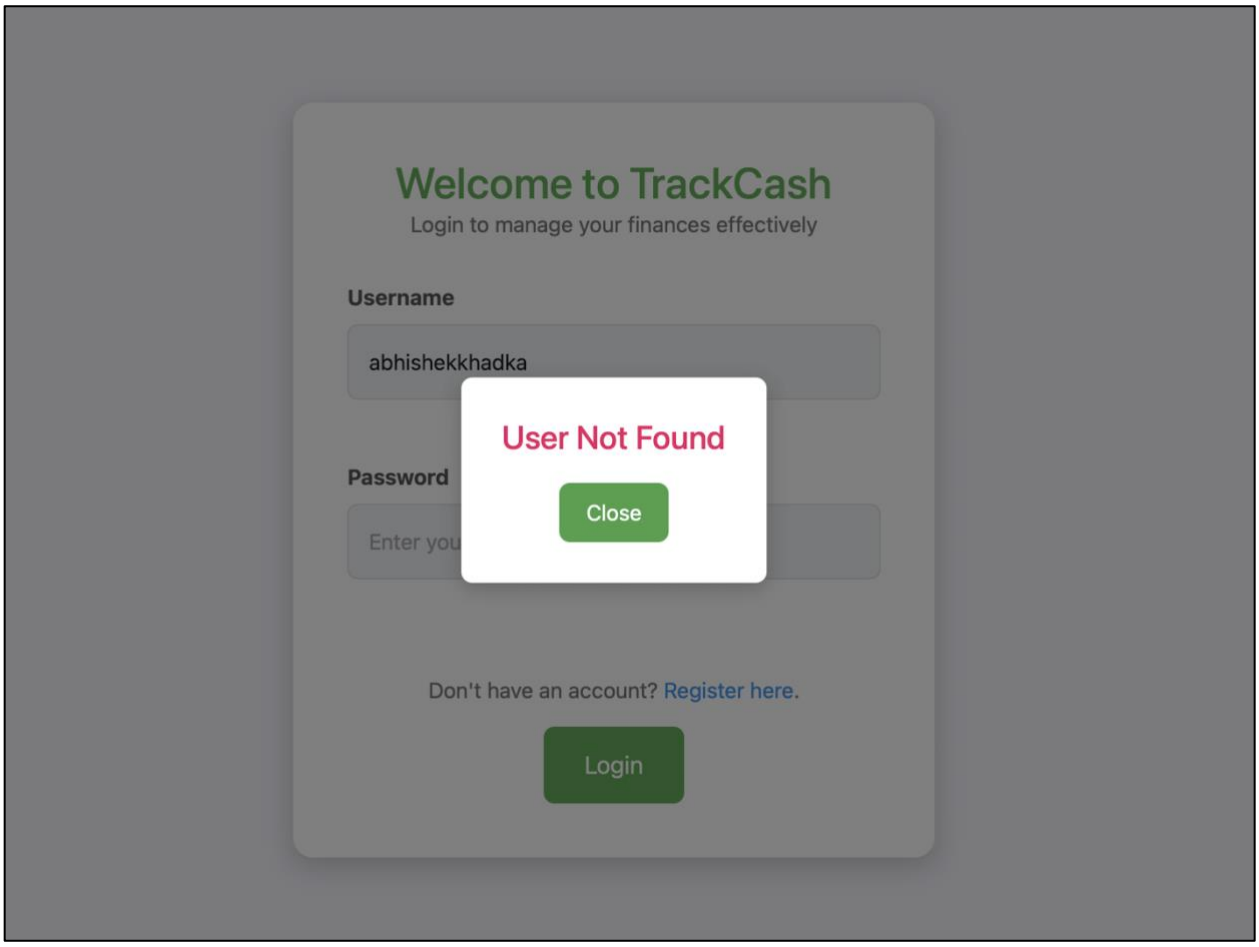

Figure 5: Not registered user message

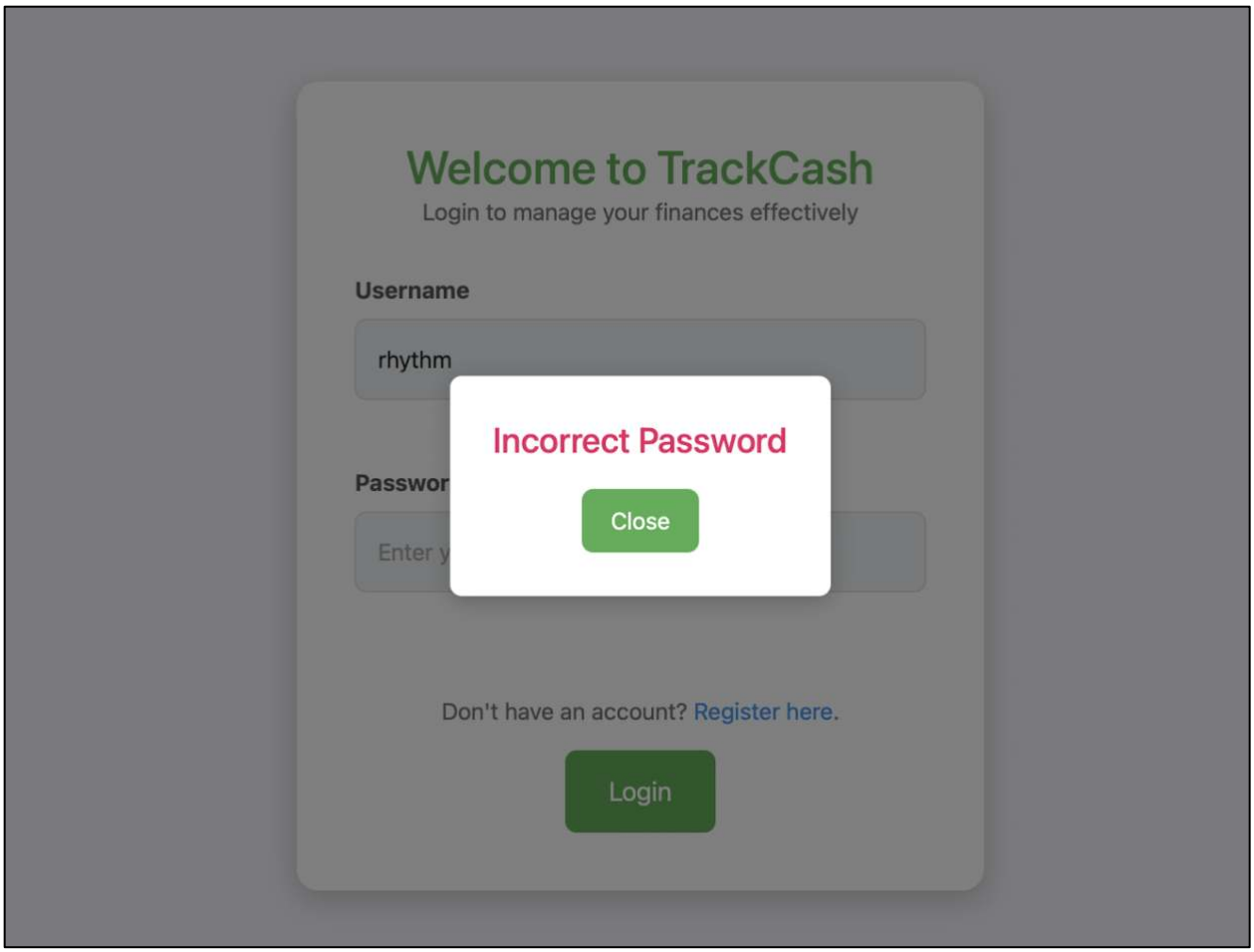

Figure 6: Incorrect login password

After successful login of user, user will be redirected to dashboard. If the user is found the user is also saved as a singleton scoped variable, which can be used to access the current user and details throughout the application.

Following code is invoked after a user is successfully found which can be accessed throughout the application.

```
//after the user logs in the values are set
1 usage  Rhythm Paudel
public void Login(UserModel user)
{
    UserID = user.UserId;
    PreferredCurrency = user.PreferredCurrency;
    UserName = user.UserName;
    totalInflow = user.TotalInflow;
    totalOutflow = user.TotalOutflow;
    currentBalance = user.CurrentBalance;
    currentDebt = user.CurrentDebt;
    User = user;

}
```

Figure 7: User session saving code snippet.

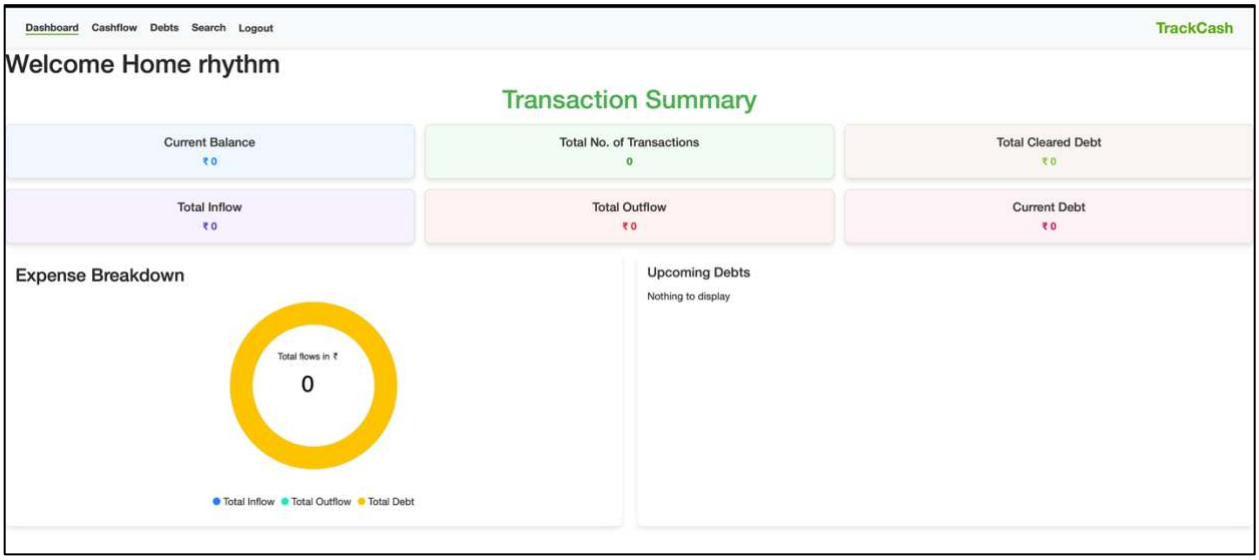

Figure 8: Dashboard (1)

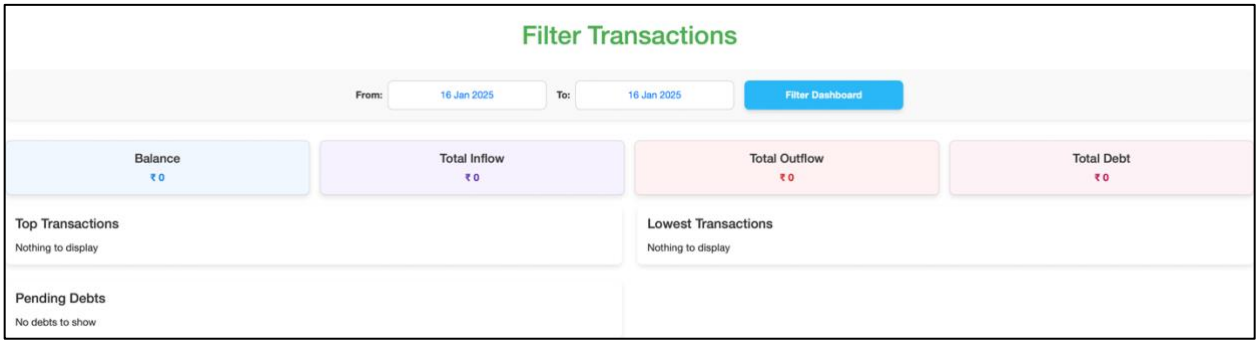

Figure 9: Dashboard (2)

2.3. Transaction management

The saved user can perform several transactions like, adding income, expense and tracking debt. Several conditions are also checked during the transaction process. For each transaction a form should be filled which has some mandatory and optional fields as well.

For each transaction, certain fields like description are optional. If the user wants a described version of transaction, user can add a note. However, for each transaction, one or many tags are compulsory for categorizing accordingly.

2.3.1. Adding inflows

This method is responsible for adding the balance of the user. For adding the balance several conditions are checked for form validation. However, unless the amount of transaction is not less than zero, without backend logic the amount is updated to current balance of the user.

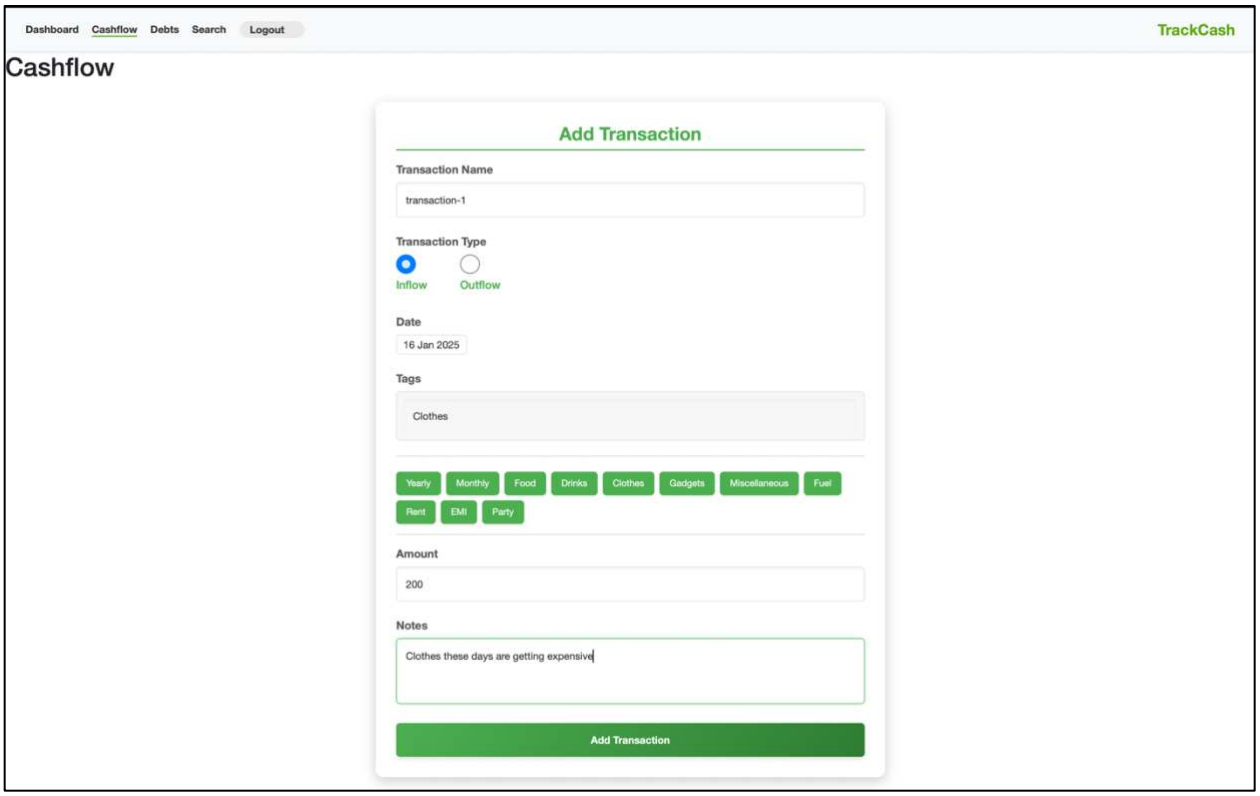

Figure 10: Adding transaction form.

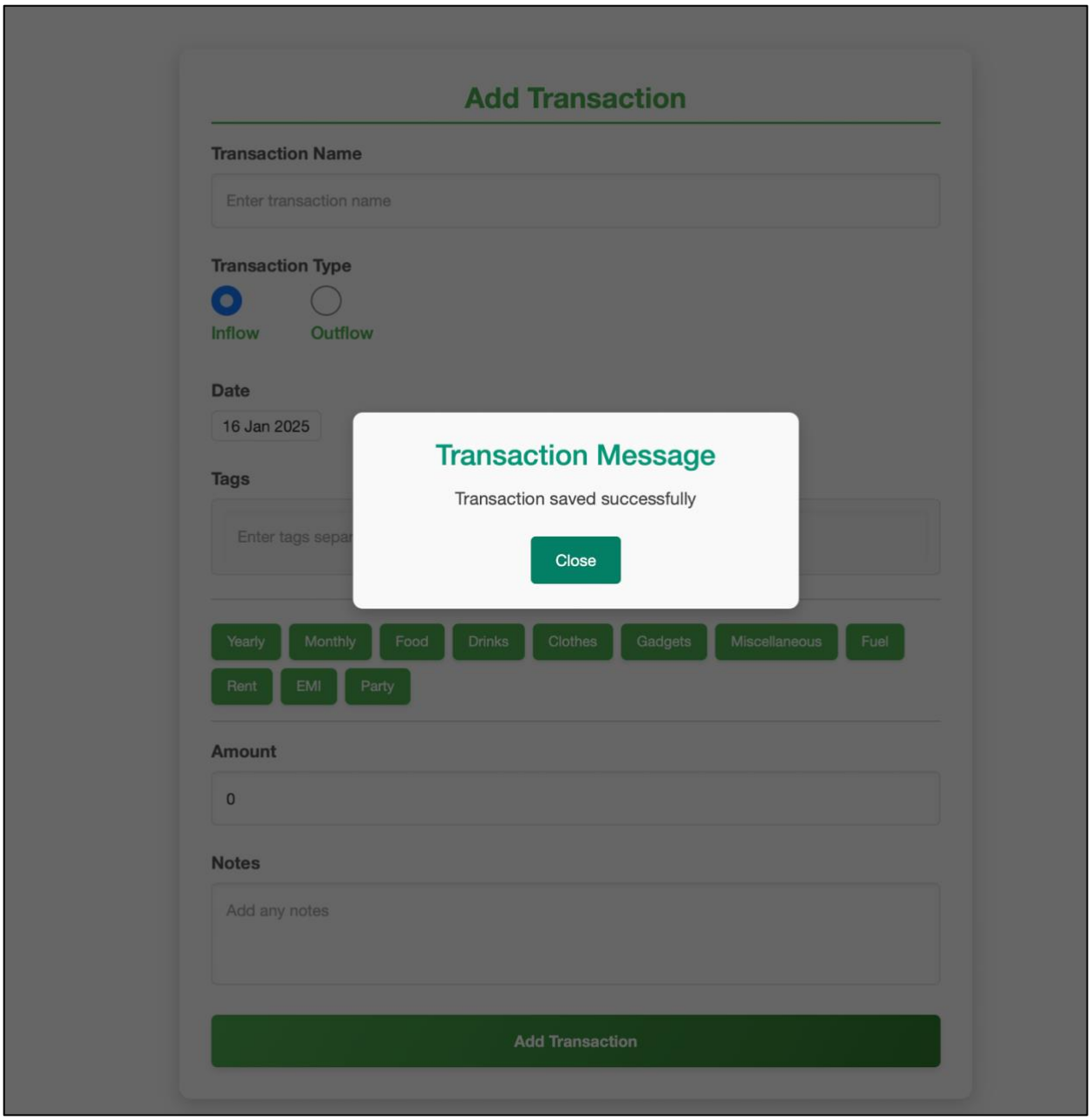

Figure 11: Transaction saved message.

After adding a transaction, user is prompted with a message. Similarly, JSON file for user and transaction are updated as well.

2.3.2. Adding debts

For adding debts, the current balance of the user is updated but the actual balance is not updated. In this application actual balance refers to the user's personal income, not debt. For adding a debt, a similar action is performed. After navigating to debt section from the navigation bar, the user should fill up necessary details. After the debts are filled the transaction and user files are updated. Additionally, a new debt is also added in debts Json file which will hold other debts details.

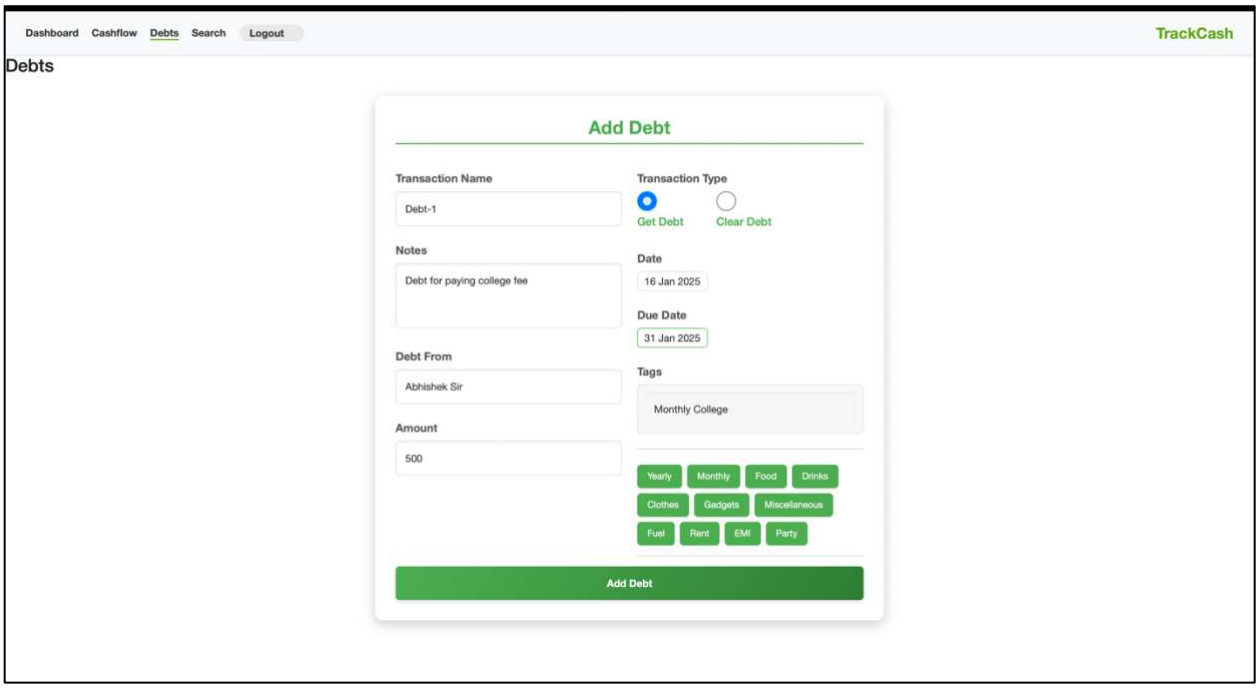

Figure 12: Debt form filled up.

2.3.3. Adding outflow

The application also features a functionality of adding outflow which is calculated through user available balance. The outflow only works if the user has more or an equal amount in current balance than the amount, the user wants to spend.

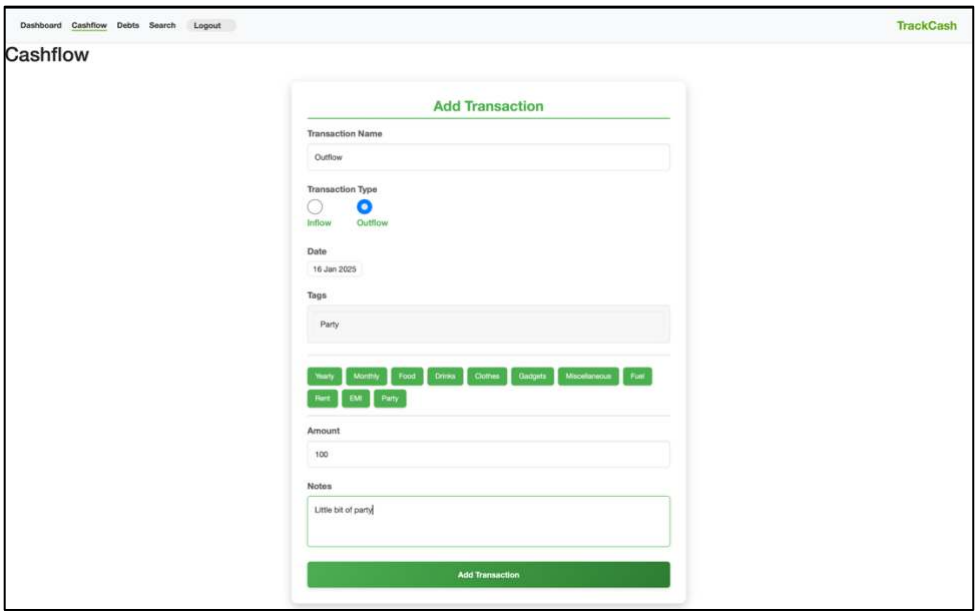

Figure 13: Adding outflow

2.3.4. Clearing debts

Along with adding a debt, the user is also able to clear their debts. However, if the debt to be paid is more than the available balance or the amount user want pay is more than the available balance the transaction is cancelled. For clearing debt in the debt section of the application clear debt button should be pressed, which will then list the available debts, and user can enter the amount they want to pay at once.

After selecting a pending debt, the relevant fields are filled automatically. Only data and amount field is accessible.

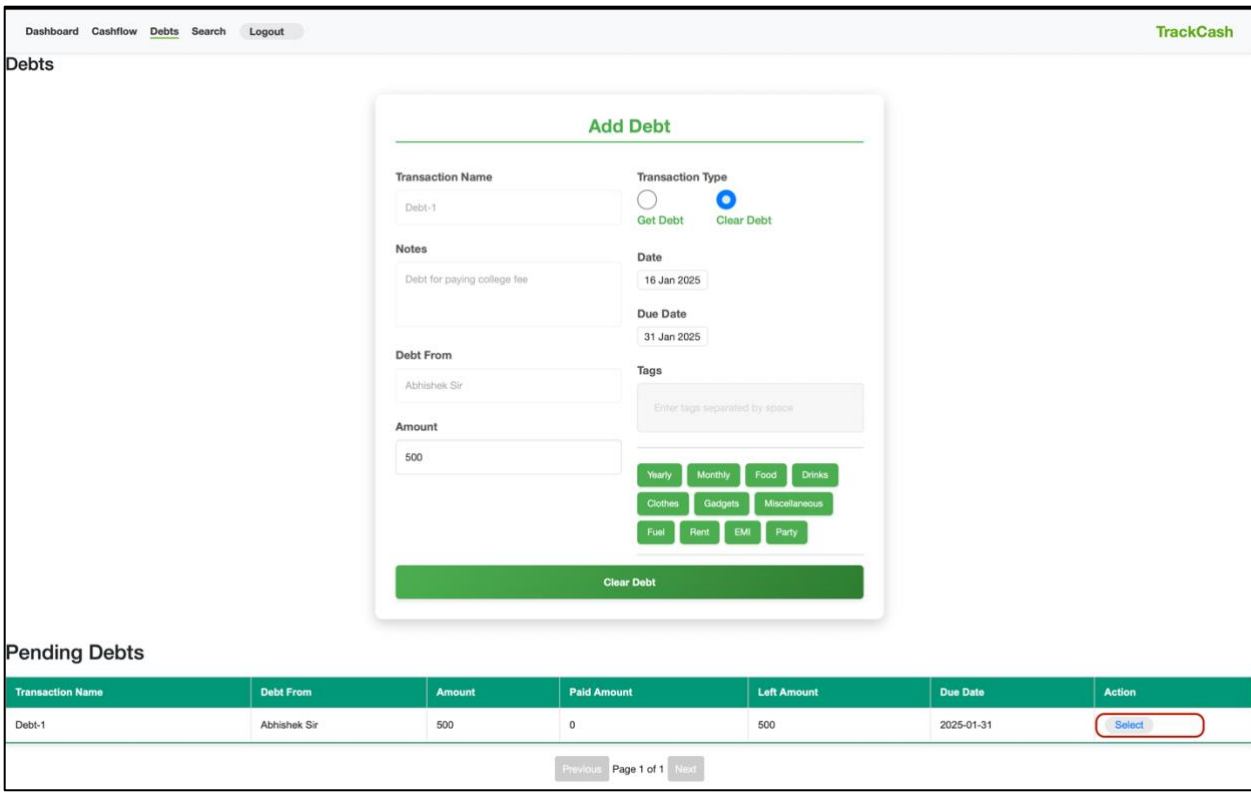

Figure 14: Clearing debt

2.4. Filtering transaction

Users are able to search their entire transaction history via search feature available in the application. Users are able to search by title, date, tags for each type of transactions or all at once. Furthermore, user also have the ability to sort the transaction by date, which will return the transaction by date in ascending order.

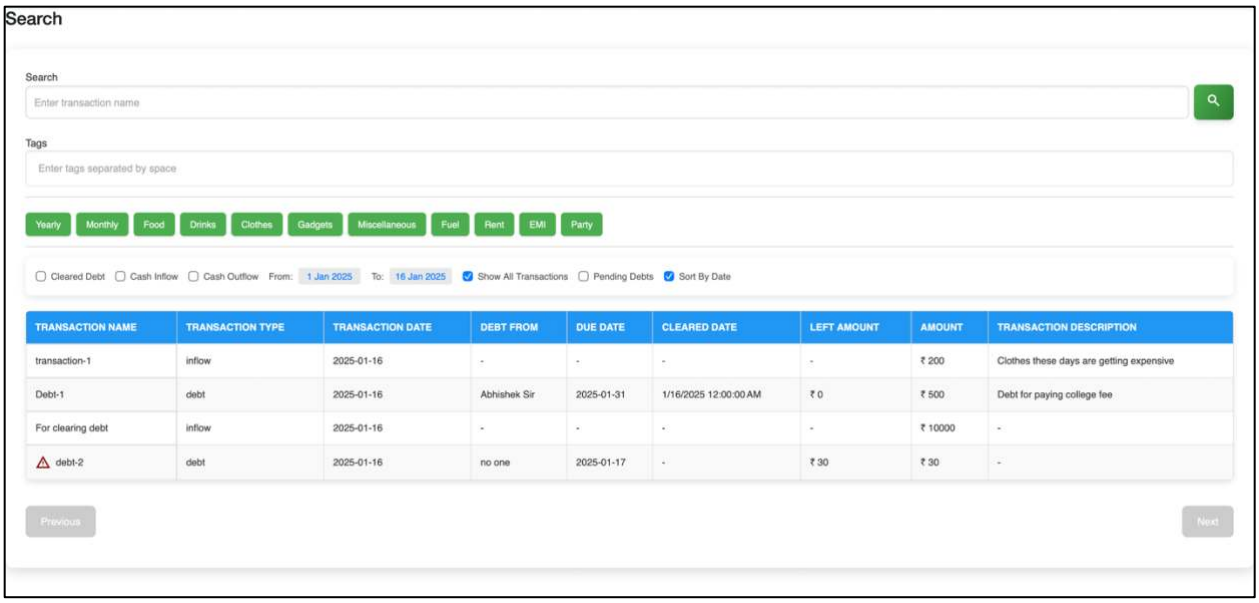

Figure 15: Search feature

In this example,only date, all transaction and sorting by date filters are enabled. If there are no title and tags it will list all transaction according to other filters applied. User can enter multiple tags, and any match found will be displayed. If one or more tags matches the result will appear, which is also same for the title.

2.5. Dashboard statistics

For this application the dashboard is divided into two different sections. The first section will contain a brief summary of user current transaction. The second section of the dashboard will have initial values of overall transactions. However, after a user filters the transaction by the date, the transactions within the date range will only be listed.

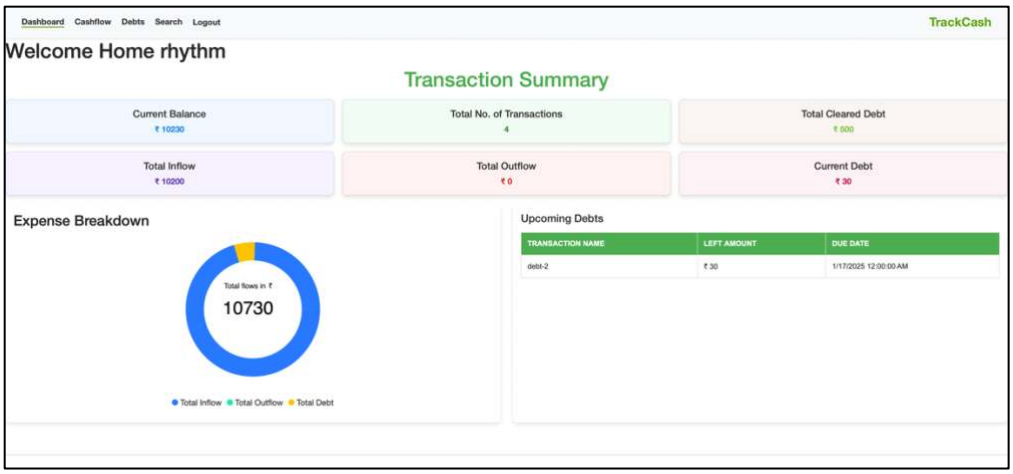

Figure 16: Dashboard summary.

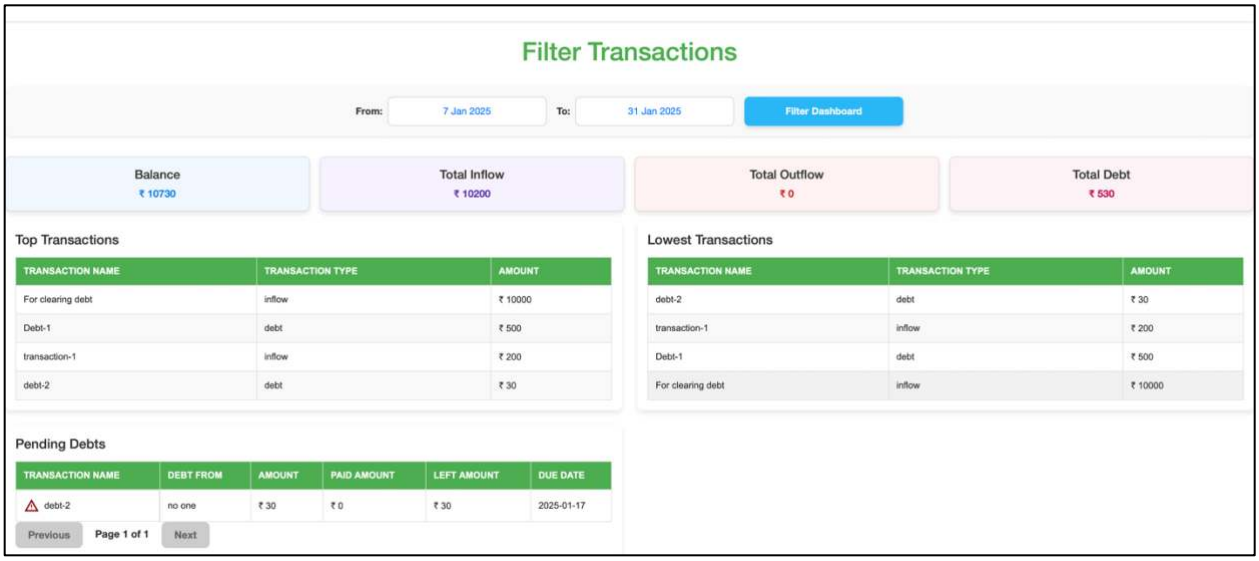

Figure 17: Dashboard filtering.

2.6. Pending debts

If the debts has not been cleared yet, then pending debts are highlighted, on dashboard and search section too.

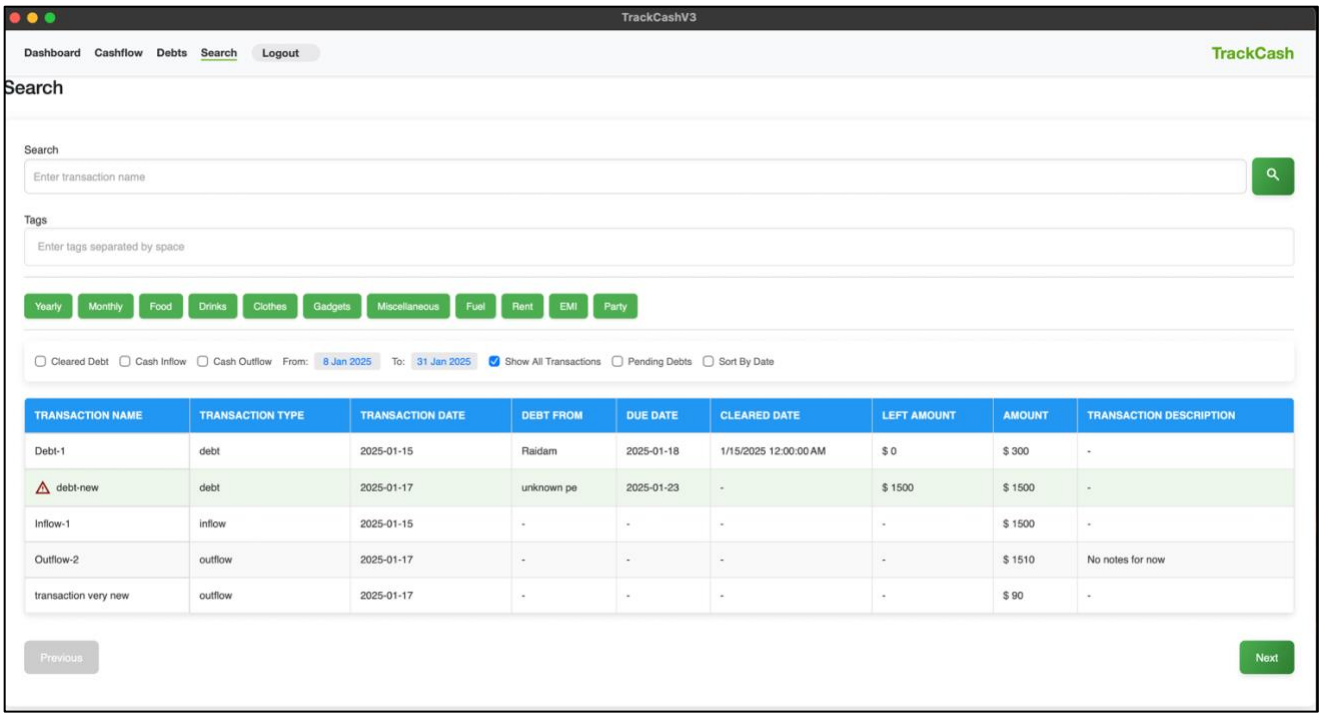

Figure 18: Pending debts highlight

2.7. Logout

If the user clicks on logout from the navigation bar, the user session is set to null and the user is redirecteed to user login page.

3. Proof of work

This section of the documentation will cover the proof of work, several UML diagrams and testing section. For testing, each function will be tested in every way possible.

3.1. Designs

3.1.1. Data entity model

Entity relationship diagram represent several entity that are related with each other. This entities define how records will be stored in database. In this application the entities are treated as a separate Json file.

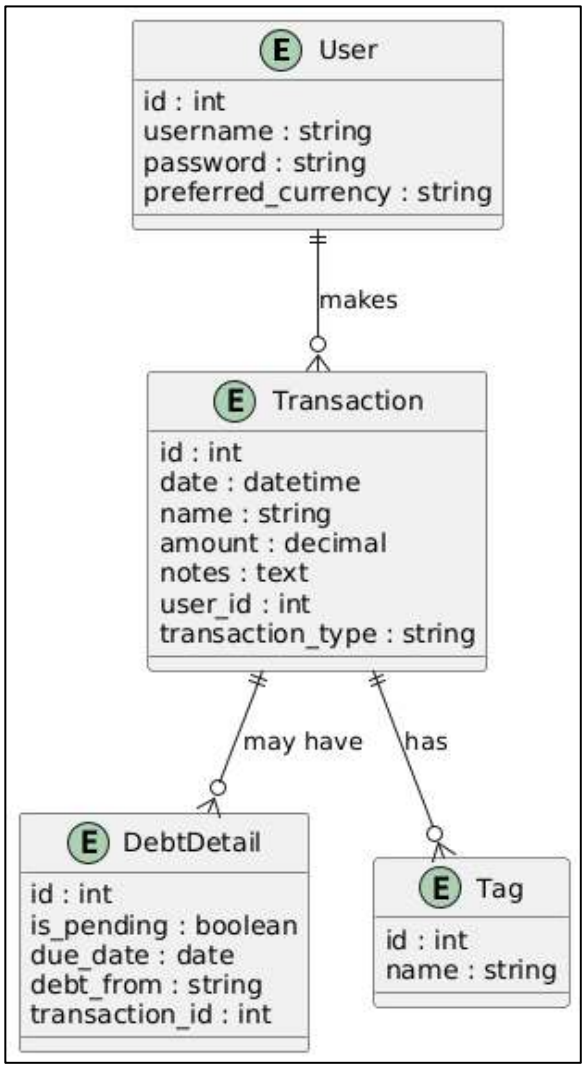

Figure 19: Entity relationship diagram

3.1.2. Class diagram

Class diagram represents how the classes are made and how the flow is working between classes. Since the application was made using MVC pattern, the whole development process was developed into model, which holds the data structure from the database, view which is responsible for UI and controller which holds all the logic and flow of the data throughout the system. Depending on the complexity of the application controller can be further divided into services layer. In the context of this application, controller and services are considered same.

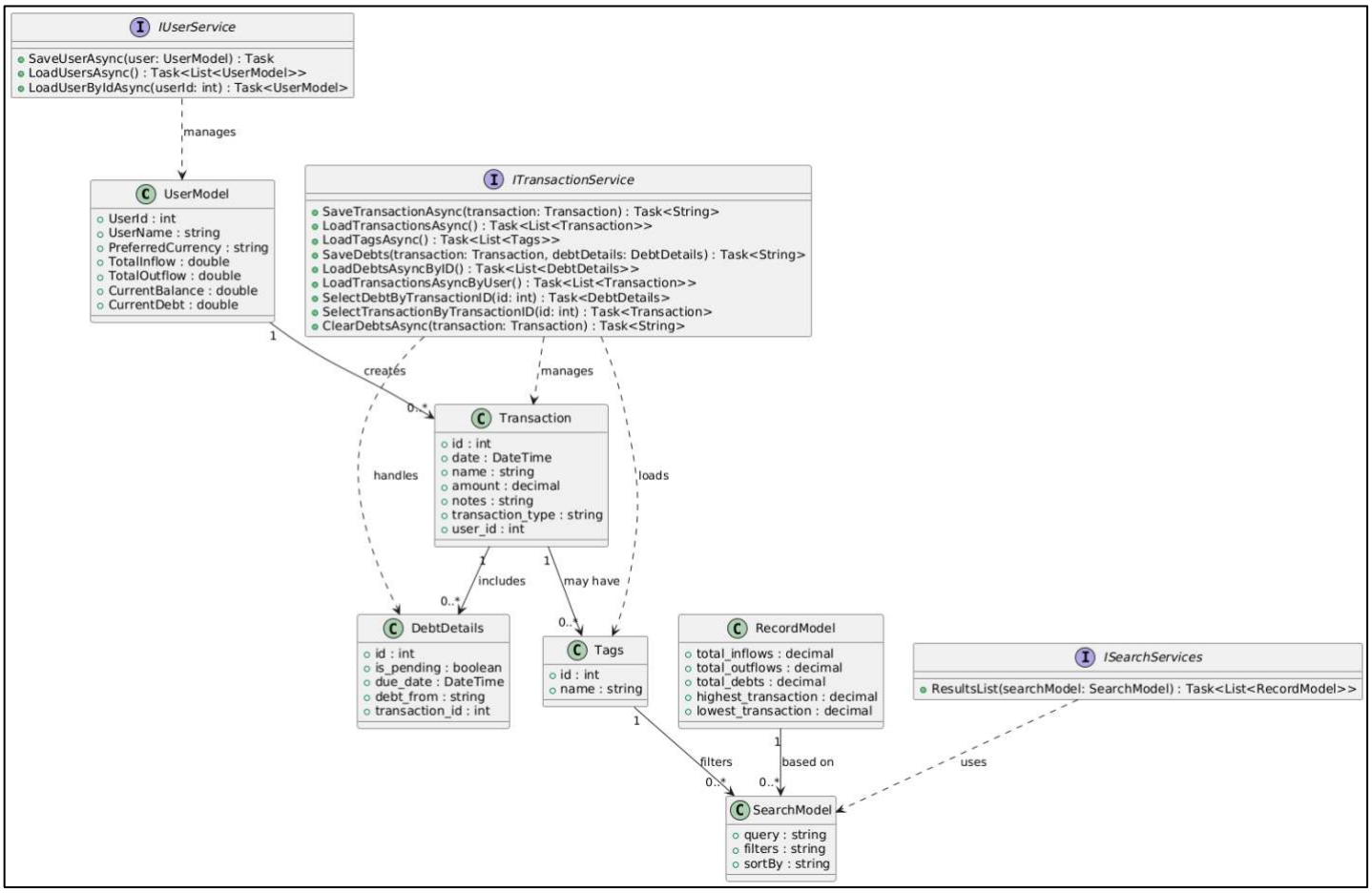

Figure 20: Class diagram

3.1.3. Use case diagram

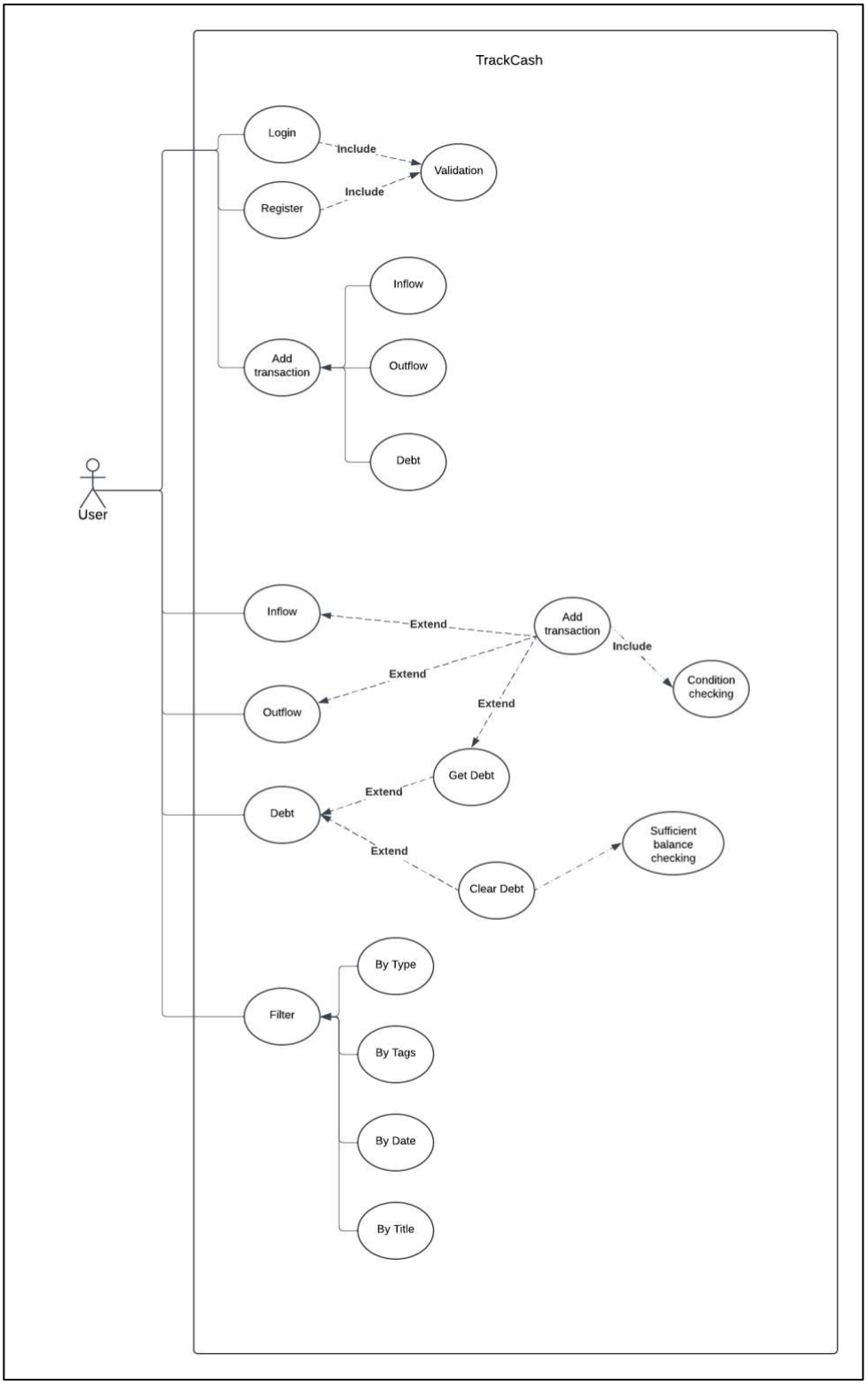

Figure 21: Use case diagram

3.2. Version control system

Version control system or VCS, refers to a system that helps in maintaining the history of development of application. It is widely used to keep track of changes, or sometimes is very helpful to rollback incase of system failure or unexpected error handled.

Similarly, to keep track of changes and to prevent in whole system failure, a git was initialized to for VCS. Furthermore, Github repository was used for pushing it online, which can be used for collaboration purpose as well.

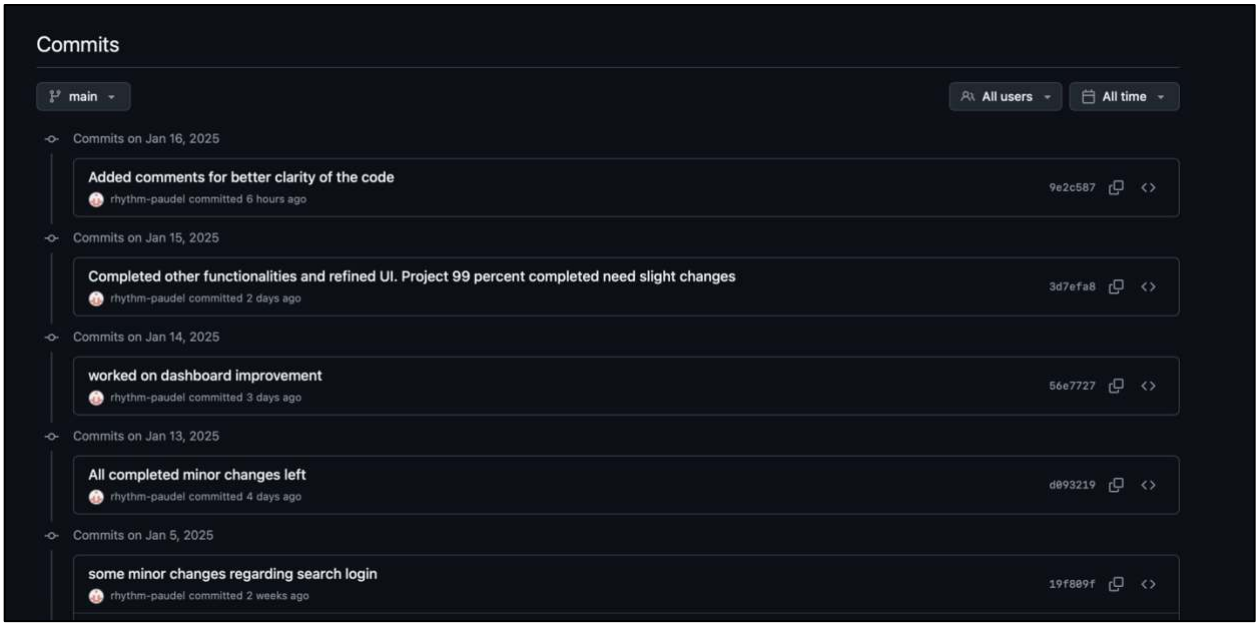

Figure 22: Github repository (1)

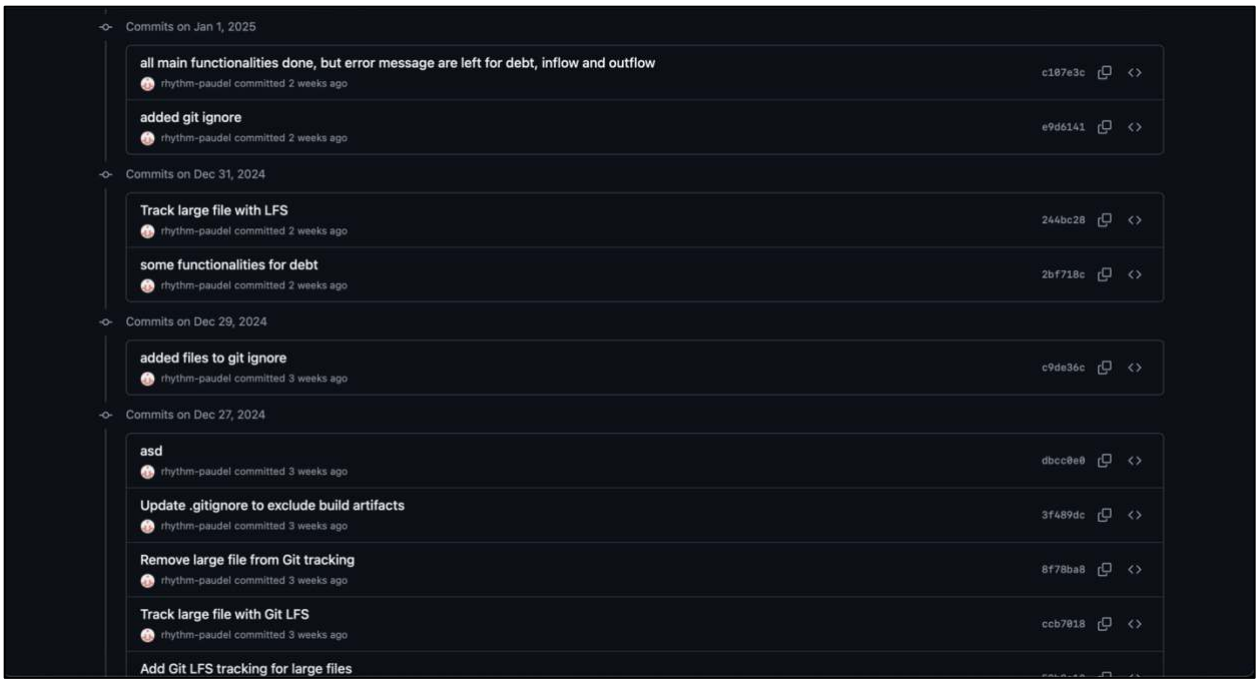

Figure 23: Github repository(2)

3.3. Testing

3.3.1. Register

Test case 01	
Objective	For user to be registered successfully.
Action	1) Click on “Register Here” 2) Enter username, password and preferred currency. 3) Click on “Register Here”
Expected Result	The user should be redirected to login page.
Actual Result	The user was redirected to login page after user registration.
Conclusion	Test was successful.

Table 1: Registration testing

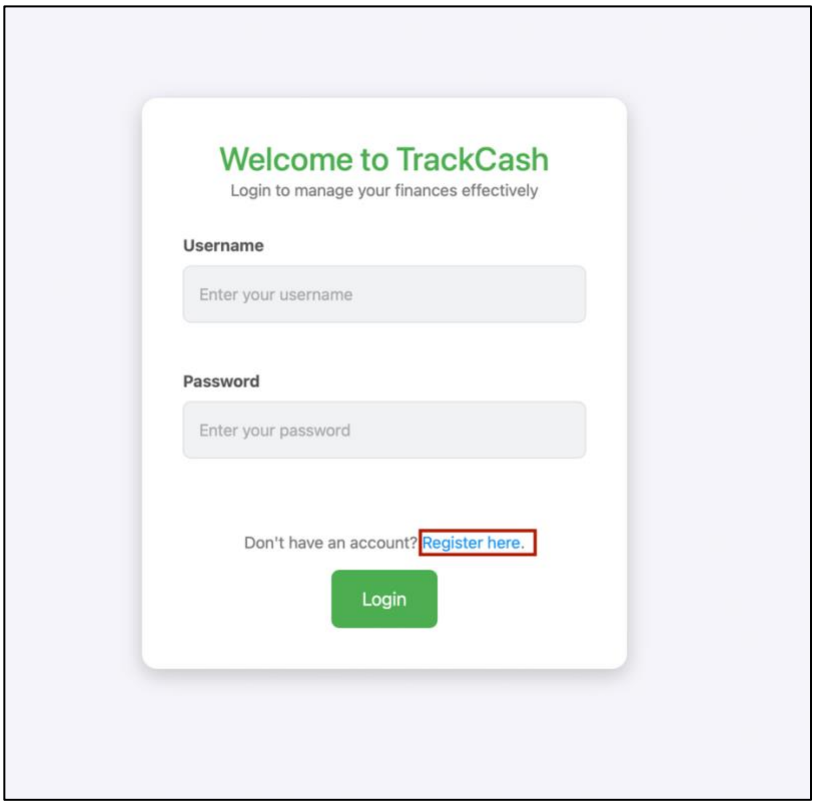

Figure 24: Locating registration page

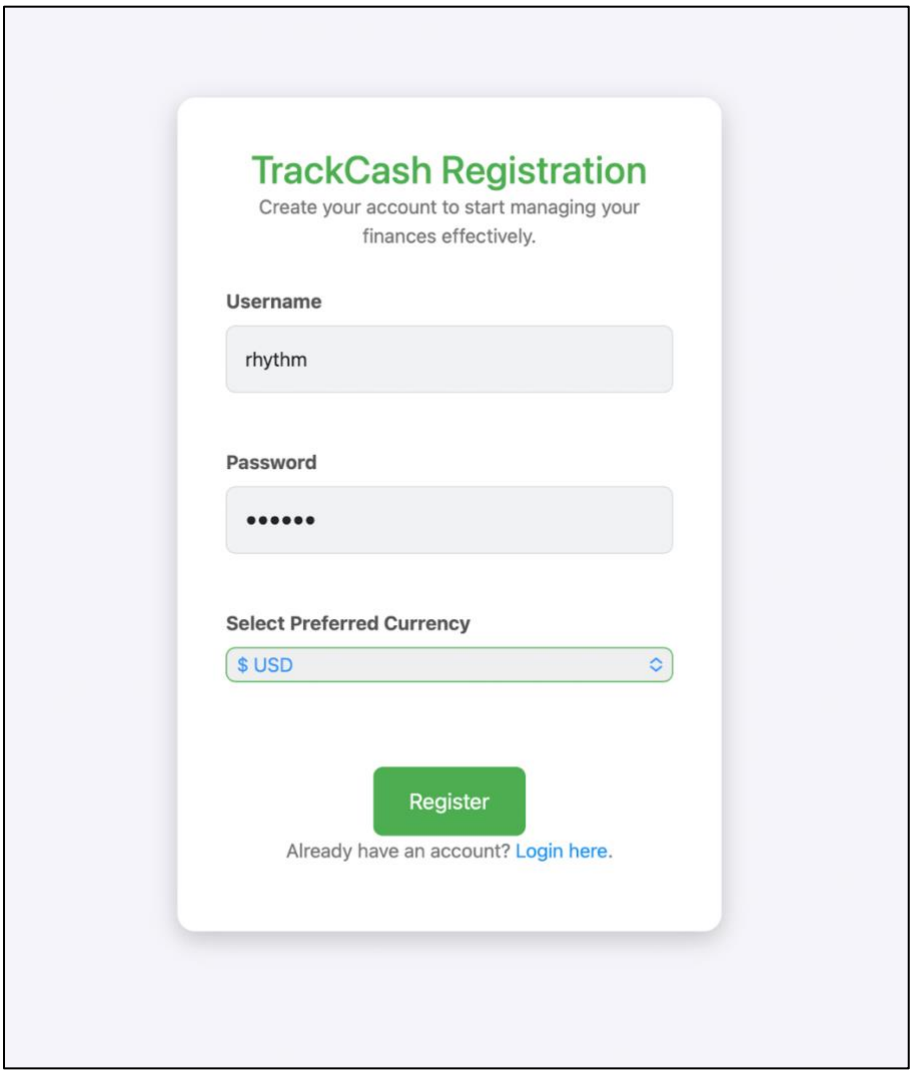

Figure 25: Registration of new user

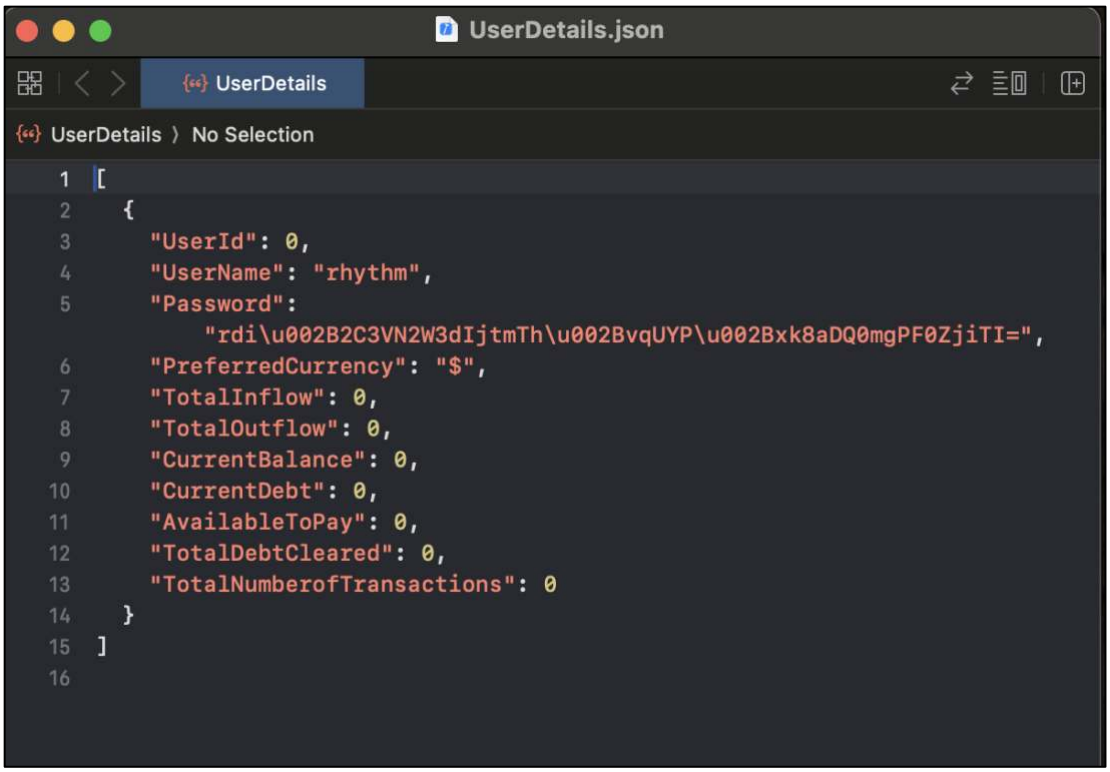

Figure 26: User registered in Json file.

3.3.2. Login

Test case 02	
Objective	For user to be logged in successfully.
Action	1) Enter username, password. 2) Click on “Login”
Expected Result	The user should be redirected to dashboard
Actual Result	The user was redirected to dashboard page after user registration.
Conclusion	Test was successful.

Table 2: Login test

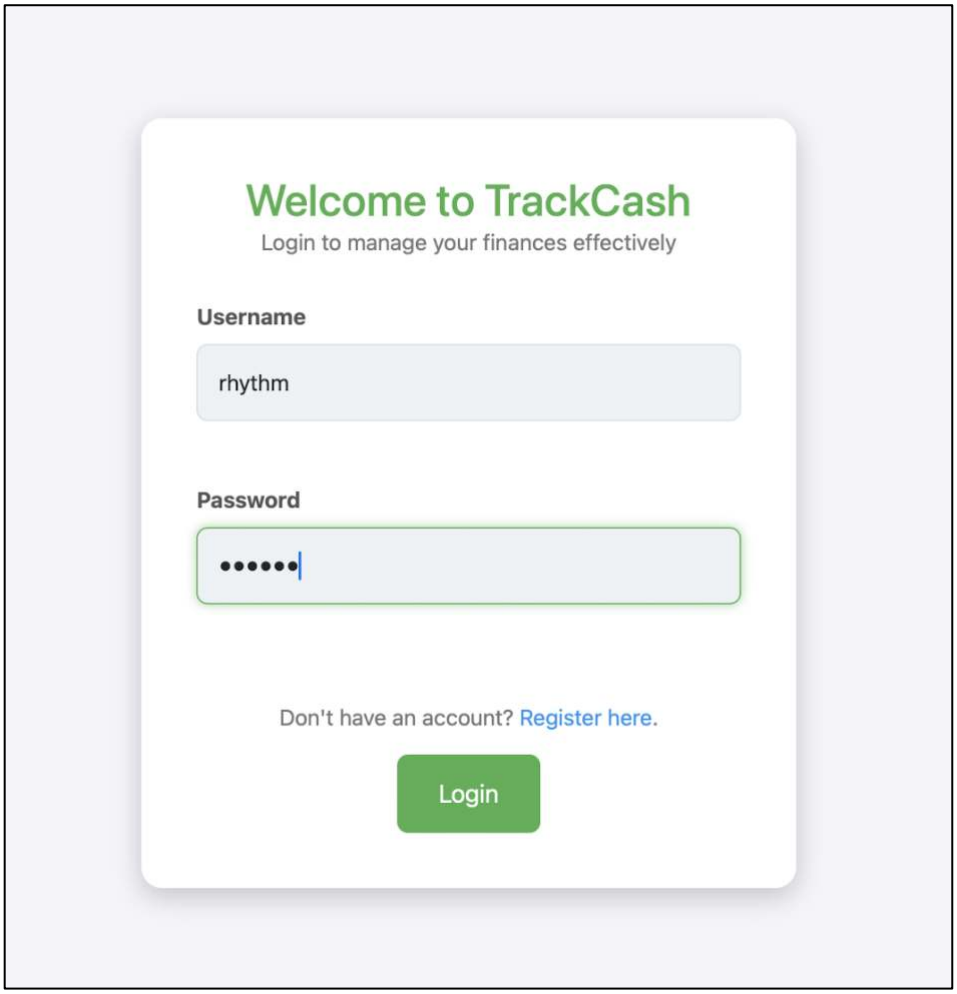

Figure 27: Logging in

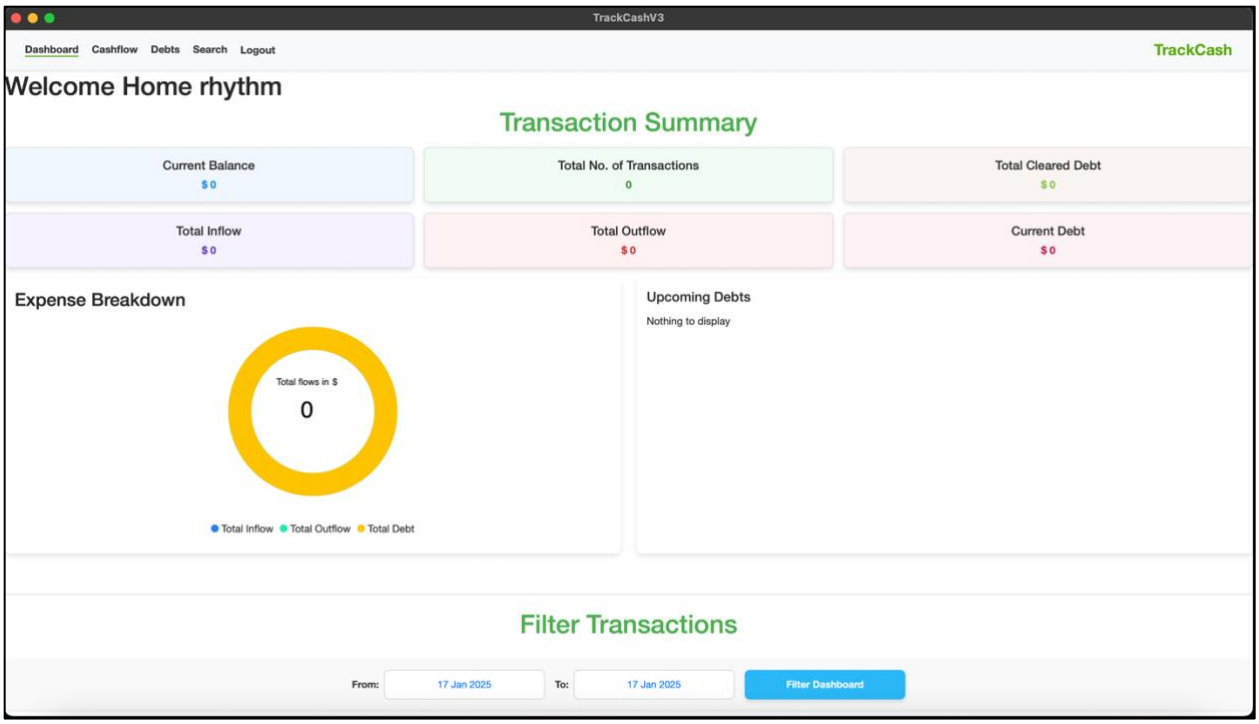

Figure 28: Dashboard redirected after login

3.3.3. Adding Inflow

Test case 03	
Objective	To add inflow and update current balance of user
Action	1) Navigate to cashflow section 2) Click on add transaction 3) Fill up required details 4) Select transaction type as inflow 5) Click on add transaction
Expected Result	On first try error message should popup and second time the transaction should be added.
Actual Result	An error prompt was received and on the second try the transaction was added succesfully.
Conclusion	Test was successful.

Table 3: Adding inflow test

The Json file of users and transaction was updated after the transaction was added.

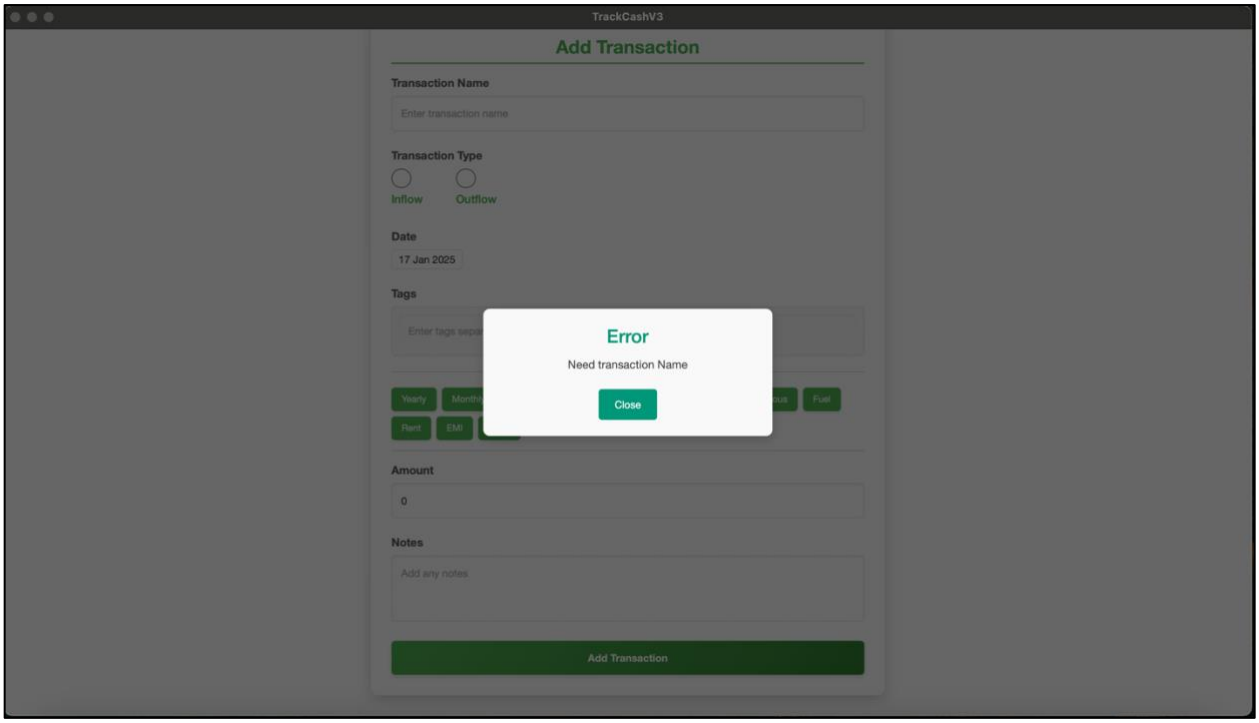

Figure 29: Error message due to incomple form details

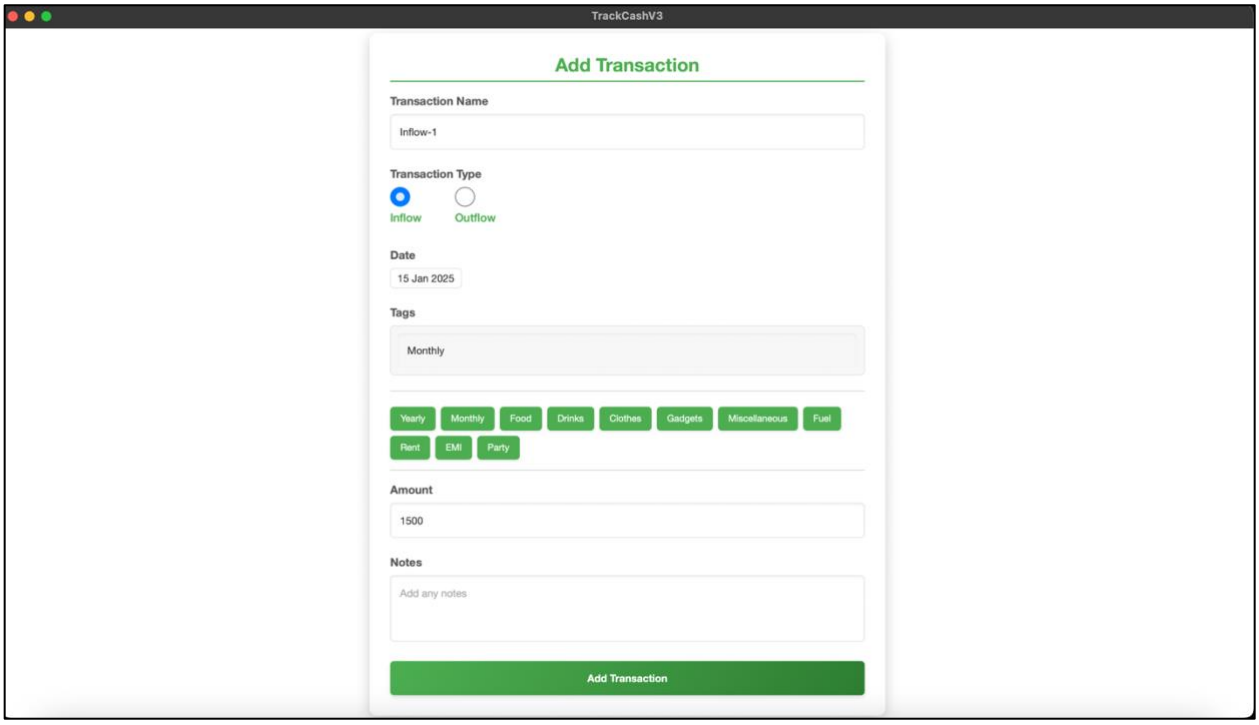

Figure 30: Transaction detail for inflow

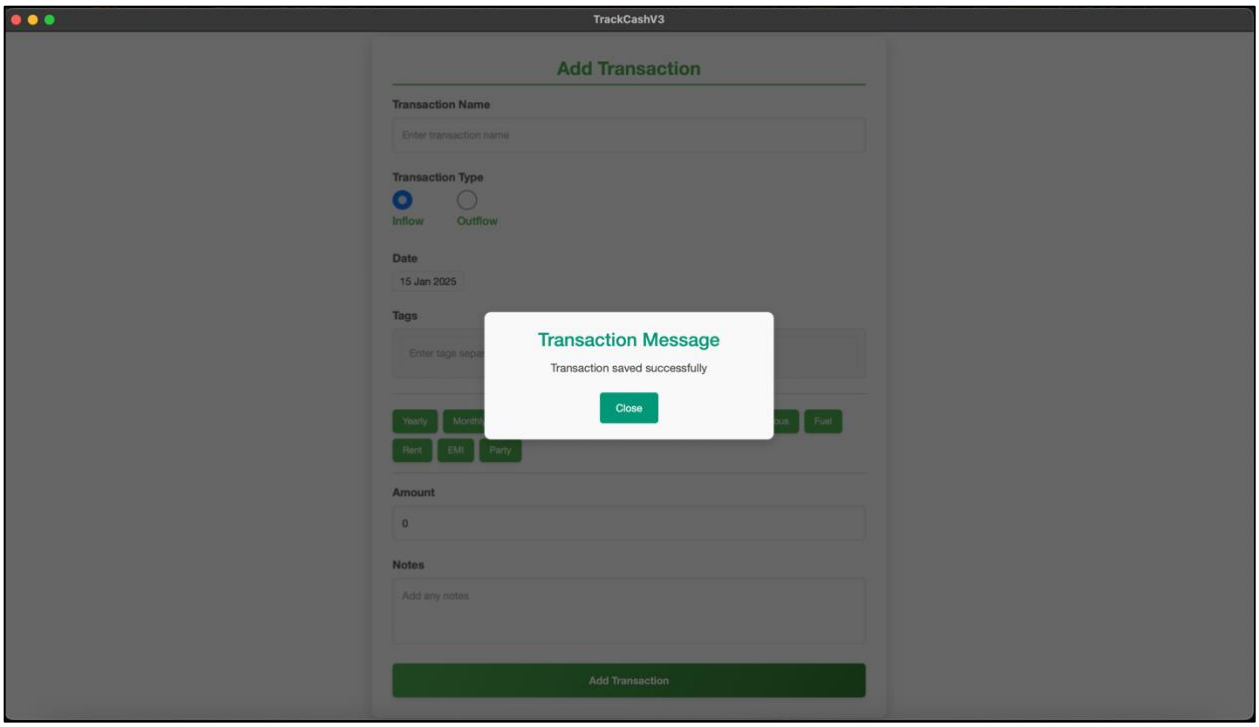

Figure 31: Inflow saved message

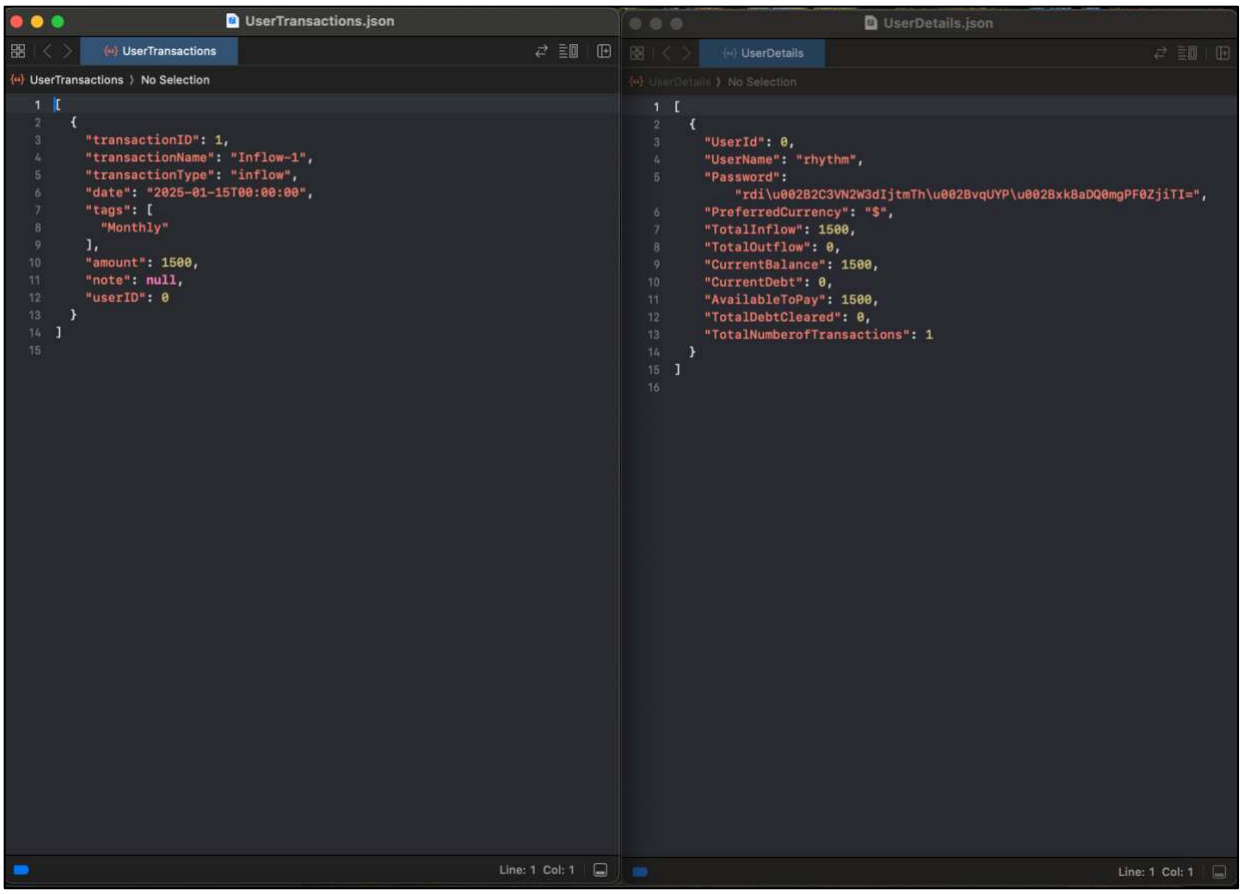

Figure 32: Update on Json file after inflow added

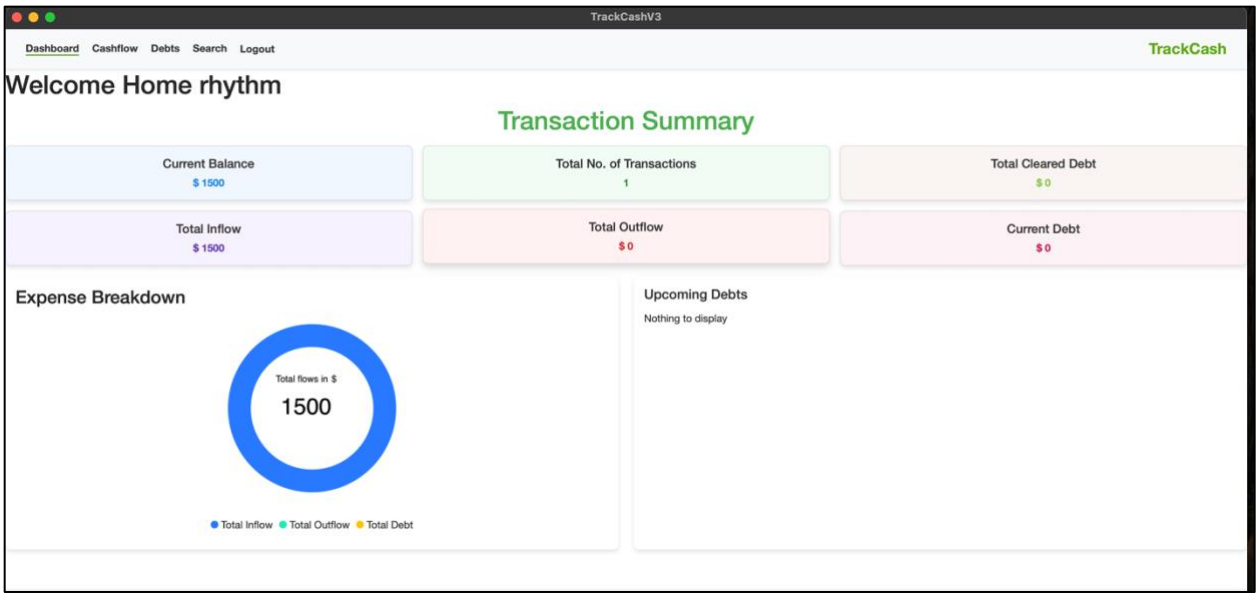

Figure 33: Updated dashboard after inflow added

3.3.4. Adding debt

Test case 04	
Objective	To add debt and update current balance of user
Action	1) Navigate to debt section 2) Click on get debt 3) Fill up required details 4) Add debt
Expected Result	Debt should be added, transaction should be added, and user details should be updated
Actual Result	Debt was added, transaction was added and user details was updated.
Conclusion	Test was successful.

Table 4: Adding debt test

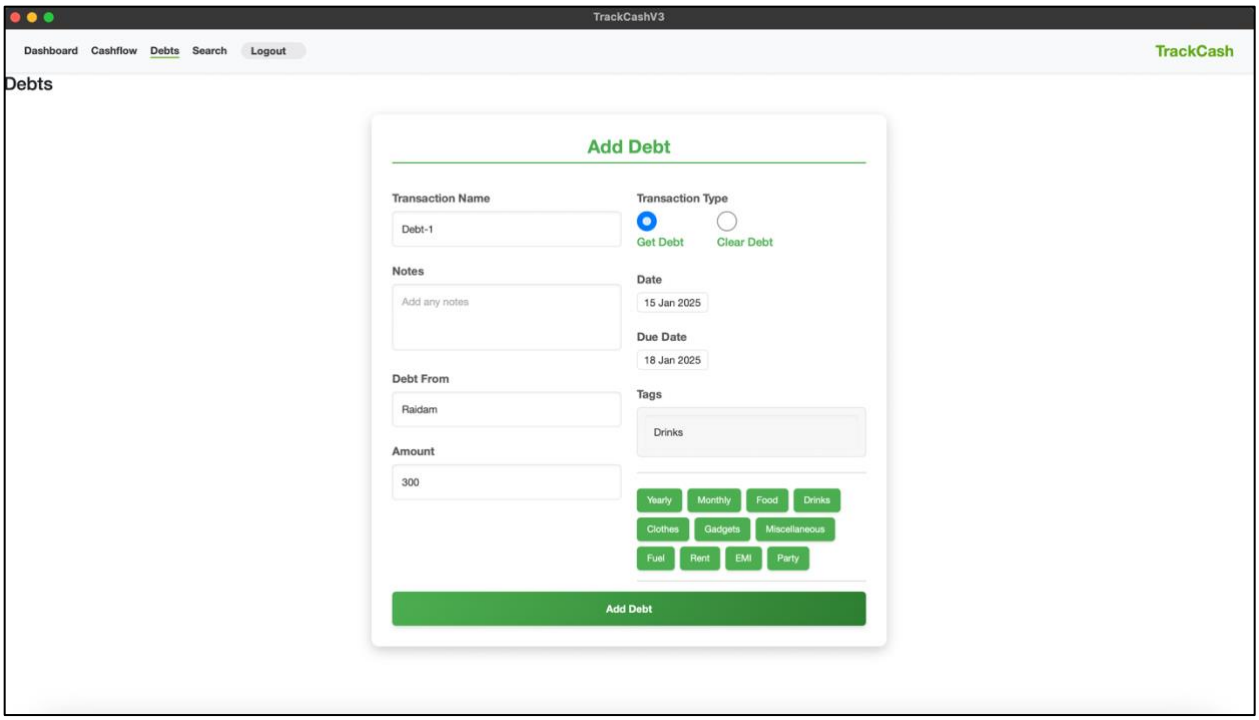

Figure 34: Get debt transaction.

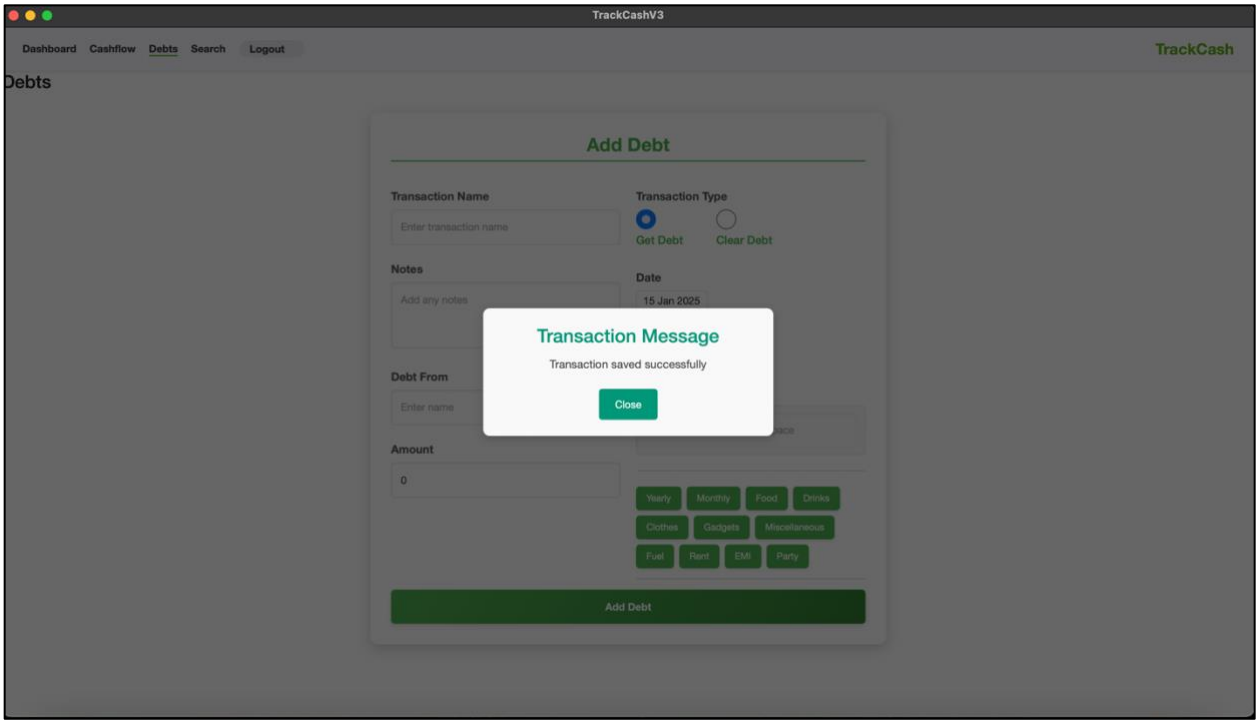

Figure 35: Get debt transaction saved message

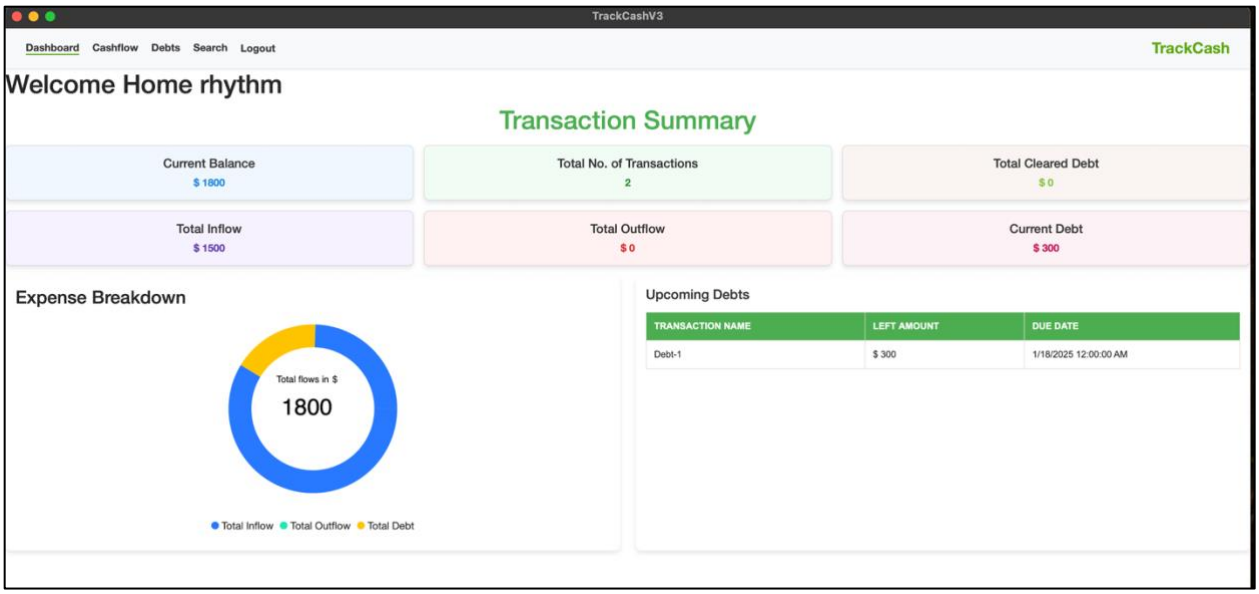

Figure 36: Updated dashboard after adding debt.

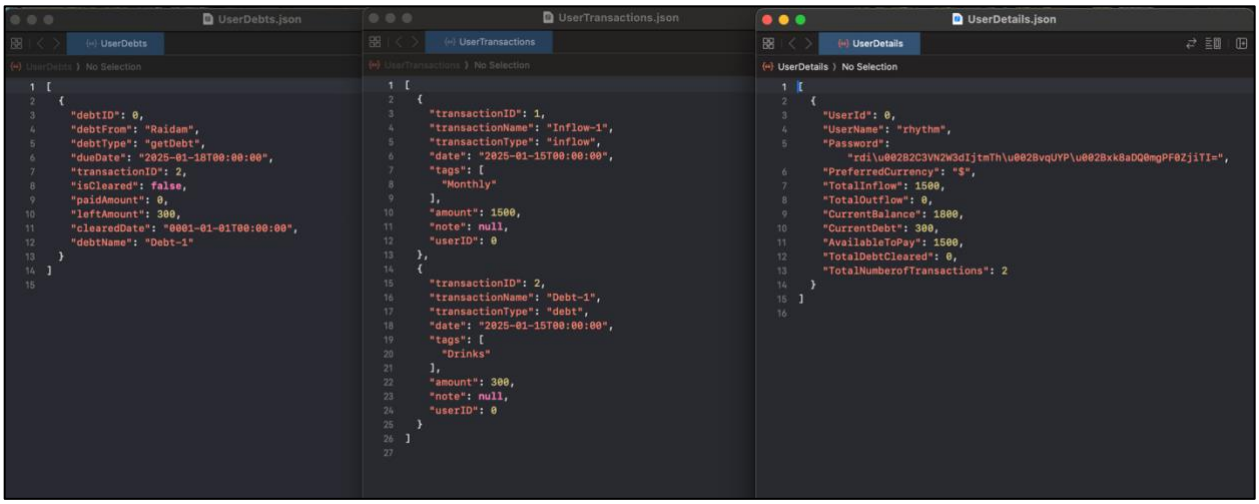

Figure 37: Updated Json file after adding debt

3.3.5. Adding outflow

Test case 05	
Objective	To add outflow and update current balance of user
Action	1) Navigate to cashflow section 2) Click on outflow 3) Fill up required details 4) Click on add transaction

Expected Result	If user has taken debt first the outflow should be deducted from debt balance not from inflow amount. If debt balance is finished then amount is deducted from inflow balance.
Actual Result	The entered amount was deducted first from debt balance and then available to pay balance
Conclusion	Test was successful.

Table 5: Adding outflow test

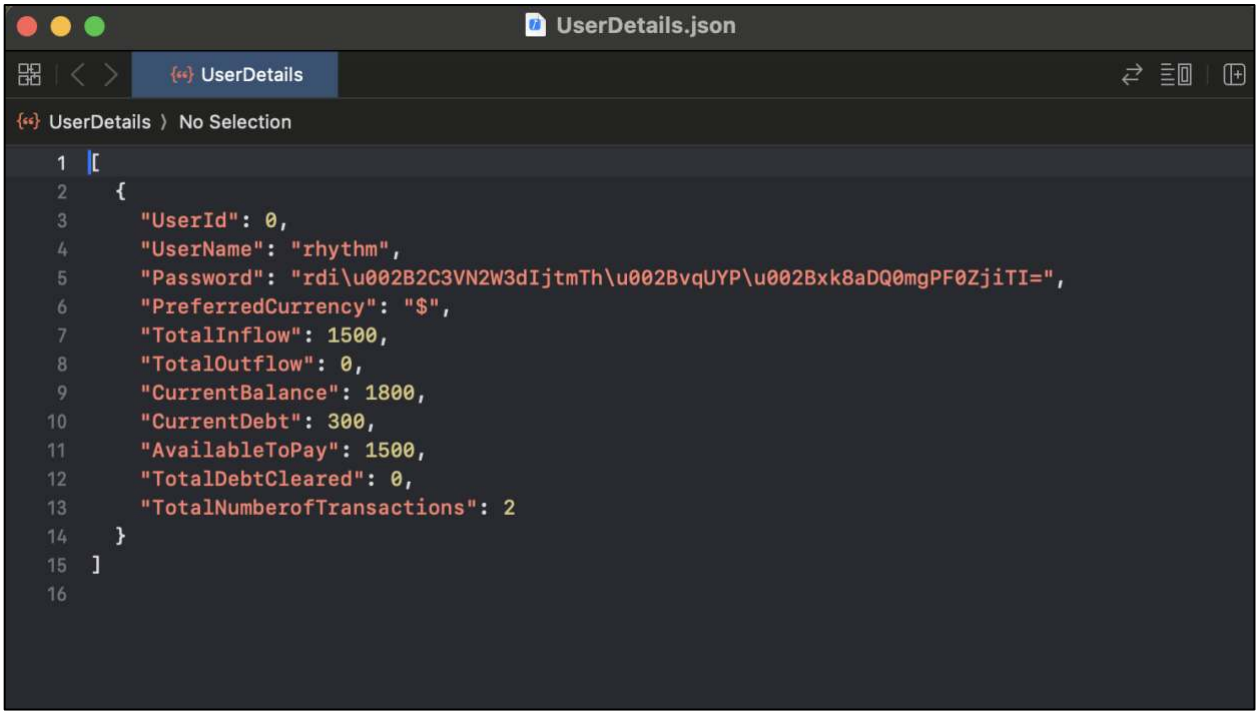

Figure 38: Before cash outflow (Json file).

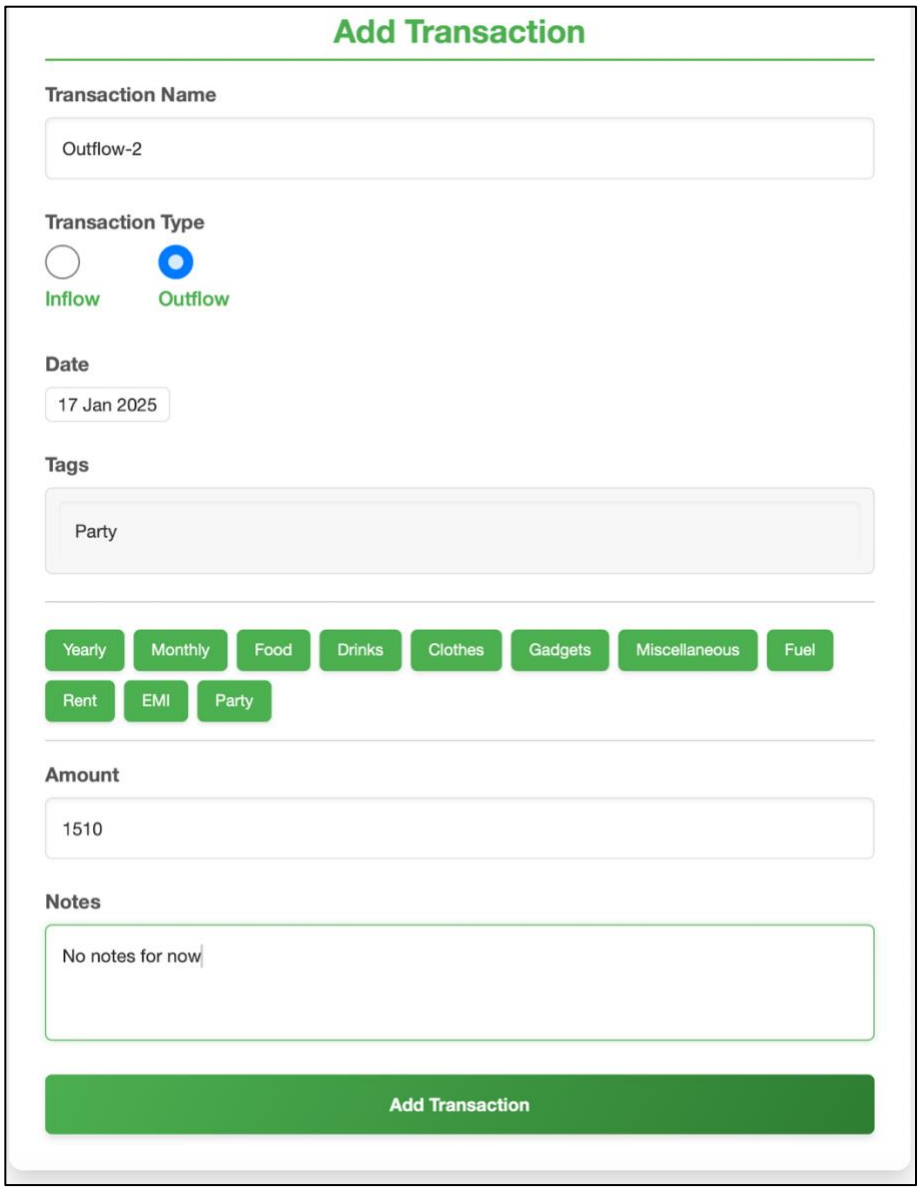

Figure 39: Filled form for outflow

Here 1510 is entered and the user debt was 300. So, if 1510 is entered the outflow should firstly be deducted by 300 making the user available to pay as it is. Once the 300 which is debt income is exceeded, only then the inflow current balance should be deducted. In this case user current balance and available to pay balance after the transaction should be 290. But if the user entered 290 instead of 1510, the user's current balance would be 1800-290, i.e. 1510 but there would not be any change in available to pay balance, because the debt balance was not fully spent.

Here available to pay means, the amount that can be used to pay debt, as debt cannot be cleared from debt.

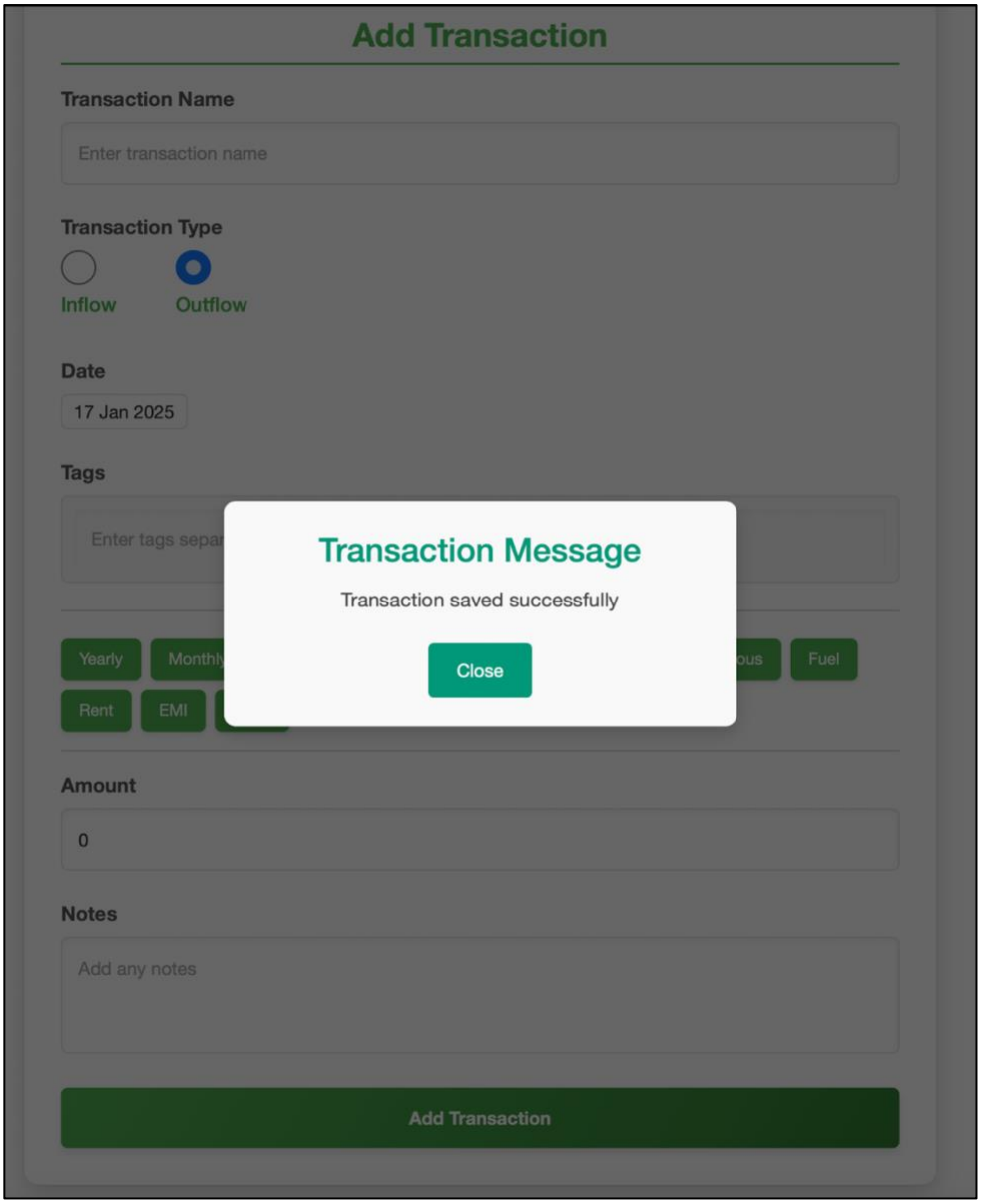

Figure 40: Outflow transaction message

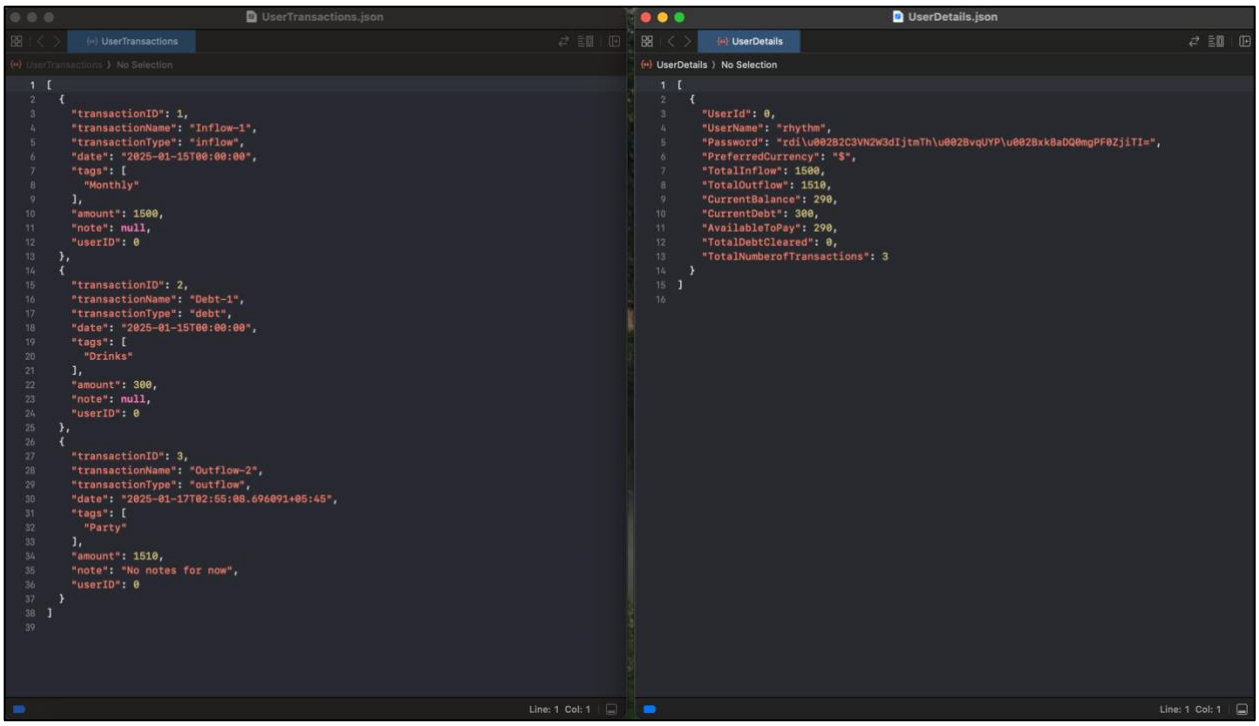

Figure 41: Updated Json after outflow

3.3.6. Outflow Test 2

Since the obtained debt can only use for outflow, during outflow first the balance obtained from debt is taken. This is also useful as available to pay section is used for paying debts, which is only used after balance from taken debt is finished.

Test case 06	
Objective	To add outflow and update current balance of user
Action	1) Navigate to debt section 2) Add new debt of 1500. 3) Click on outflow 4) Fill up required details 5) Click on add transaction
Expected Result	Available to pay section should not be deducted, but current balance should be deducted.

Actual Result	The entered amount was deducted from current balance and not available to pay.
Conclusion	Test was successful.

Table 6: Outflow test 2

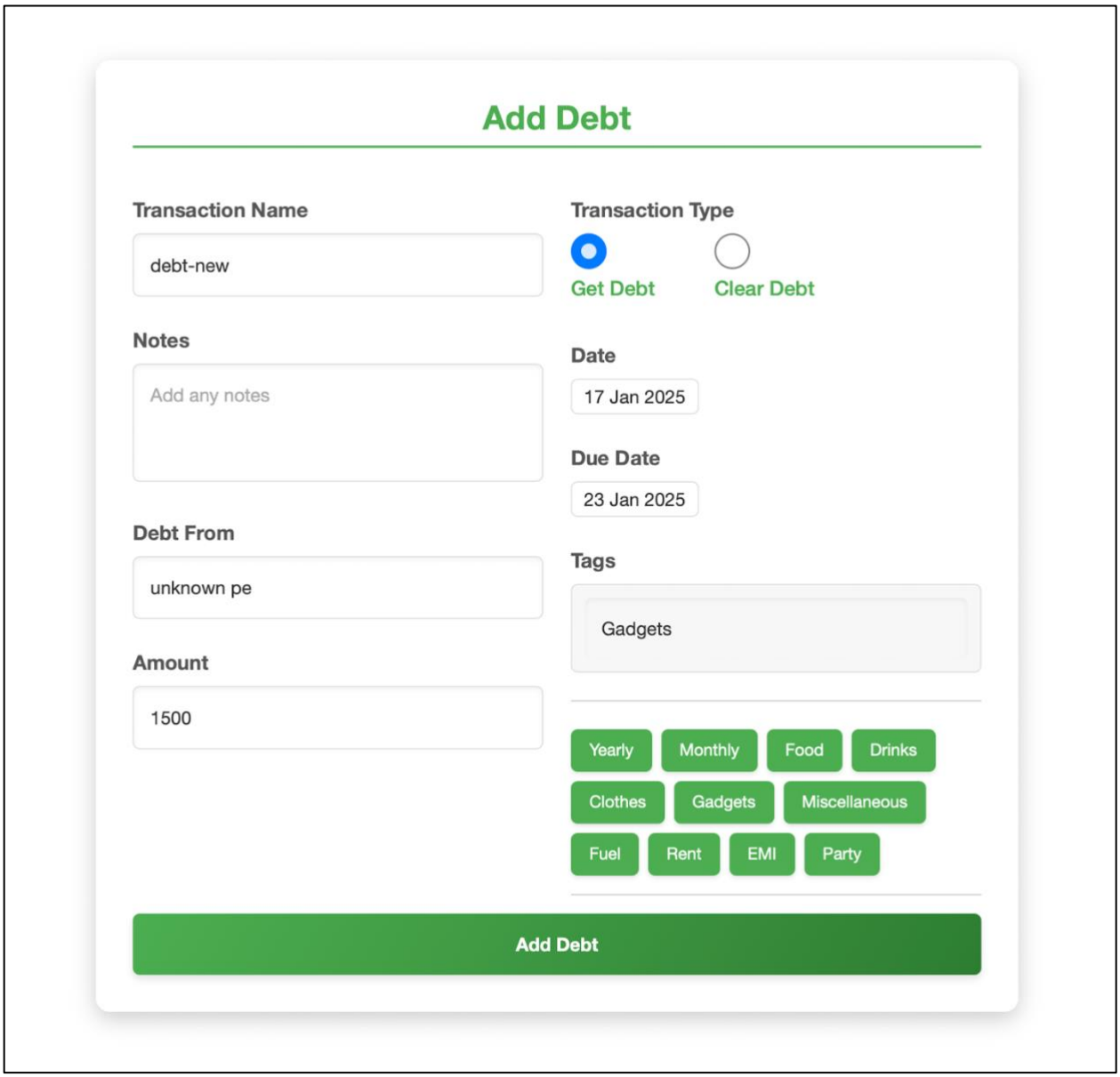

Figure 42: Debt addition (2)

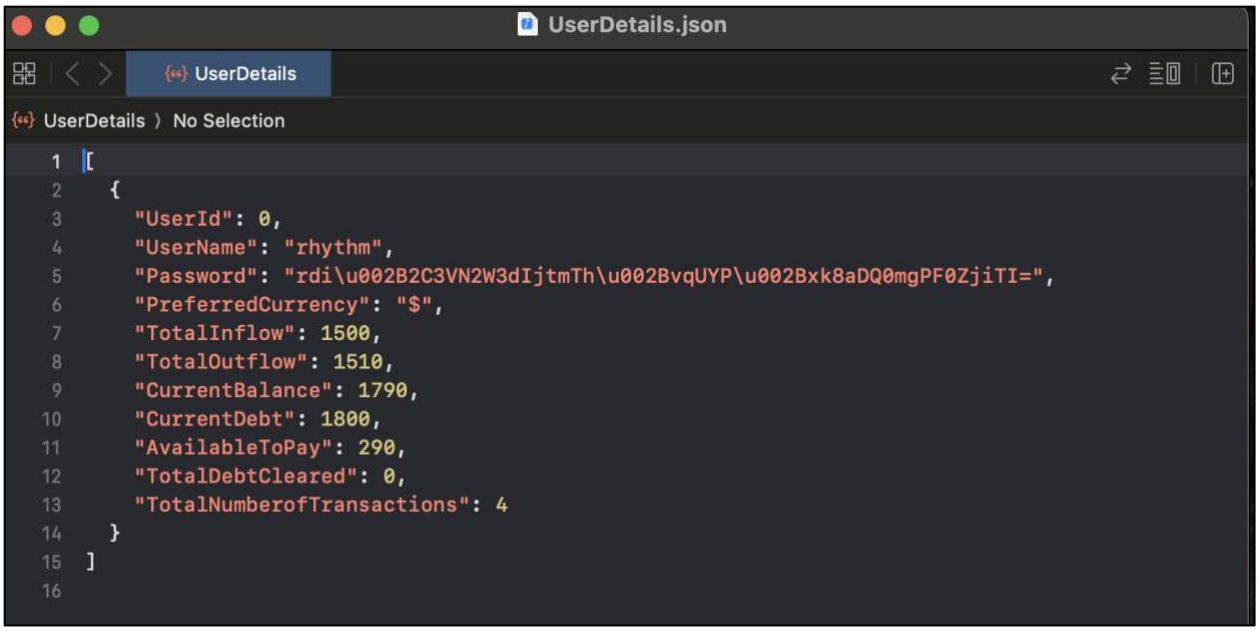

Figure 43: Updated Json after adding debt

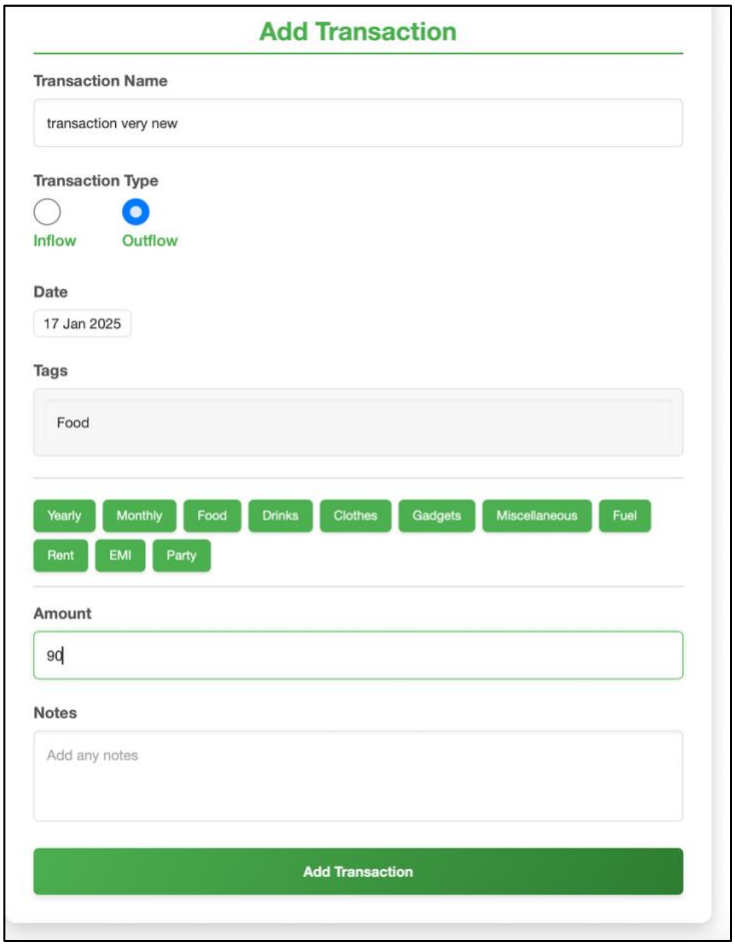

Figure 44: Outflow for test

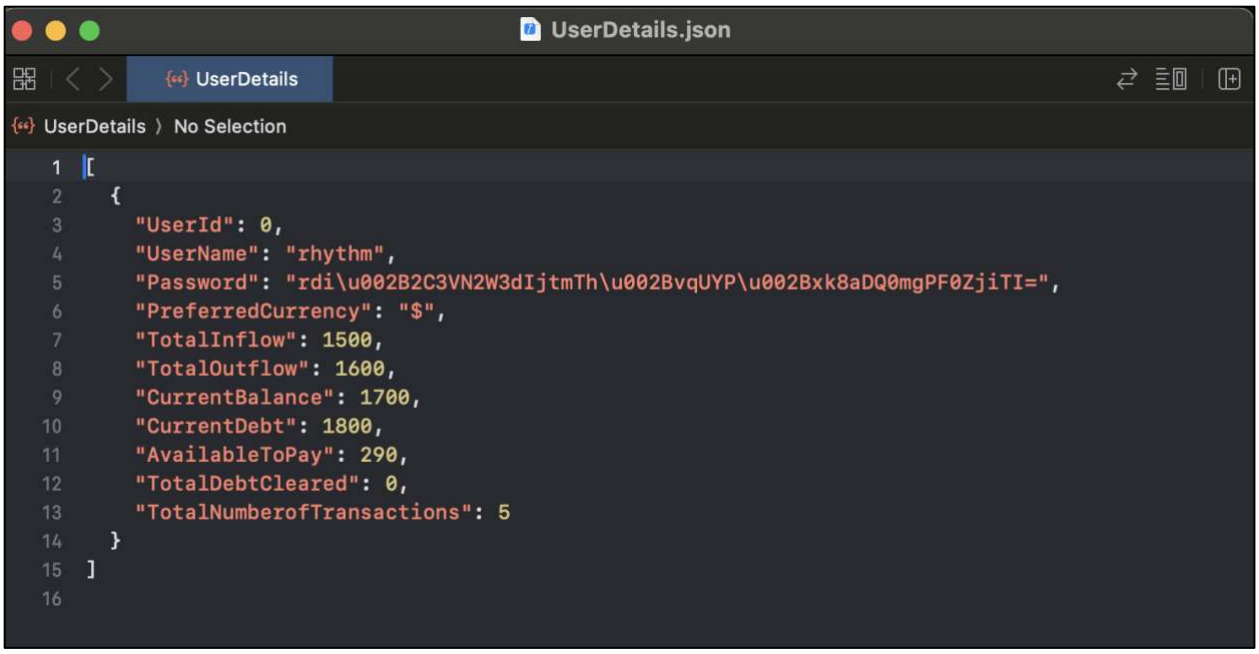

Figure 45: Update in available to pay

3.3.7. Searching and filtering

Test case 07	
Objective	To obtain the result as per the filtering.
Action	1) Navigate to search section 2) Select “Show all transaction”. 3) Click on search button. 4) Select the transaction date of a cleared date. 5) Select “cleared debt”. 6) Select “Drinks” tag 7) Click on search button. 8) Select “Cash Inflow” 9) Type “flow” on seach bar 10) Click on search buttton. 11) Select future date 12) Select “Show all transaction 13) Click on search button

Expected Result	Output data should be filtered by the input value.
Actual Result	Output data was filtered by the input value
Conclusion	Test was successful.

Table 7: Search test

Note: Although a single tag is selected for testing purpose, multiple tags work in the same way.

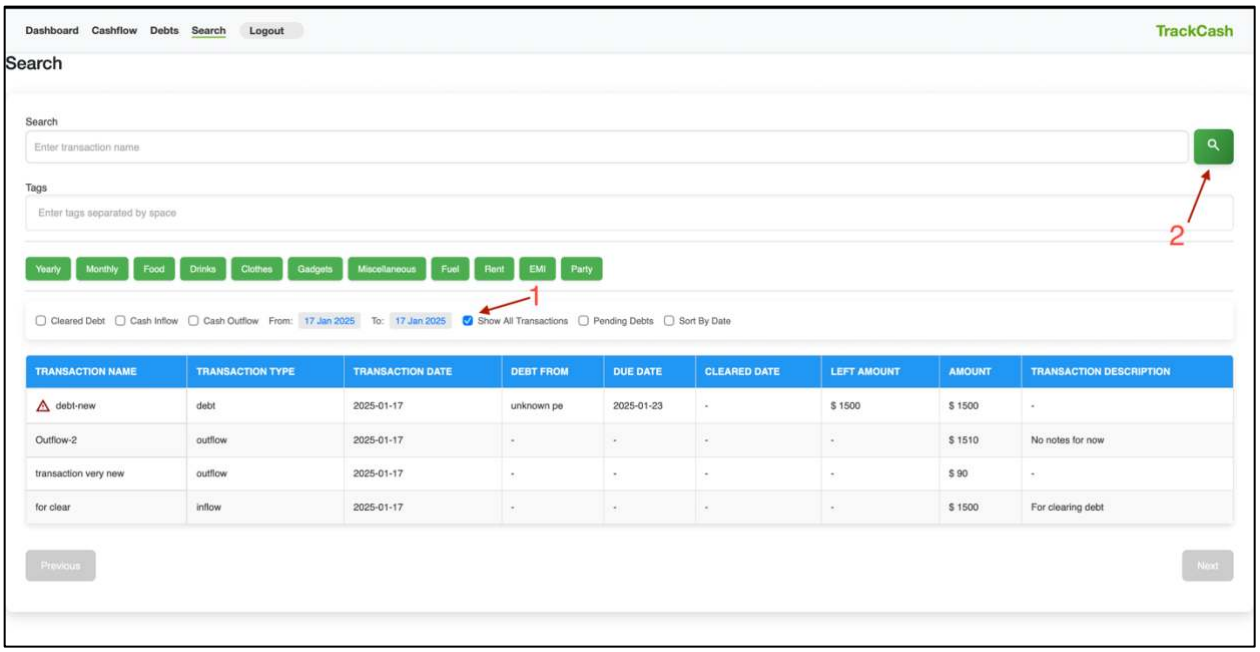

Figure 46: Show all transaction result

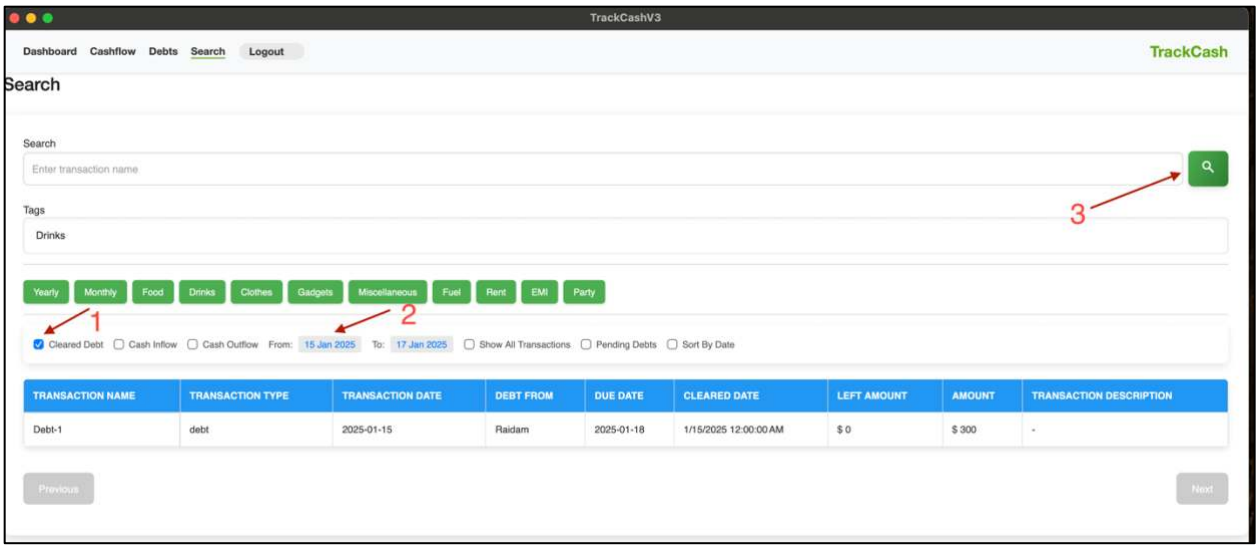

Figure 47: Cleared debt search filter

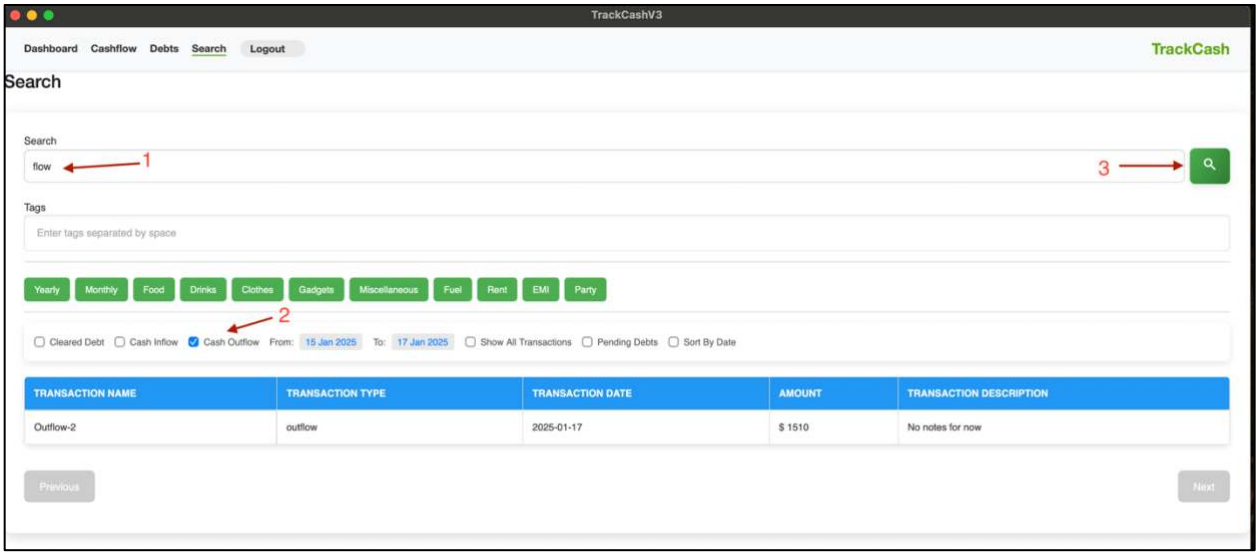

Figure 48: Cash Inflow search filter

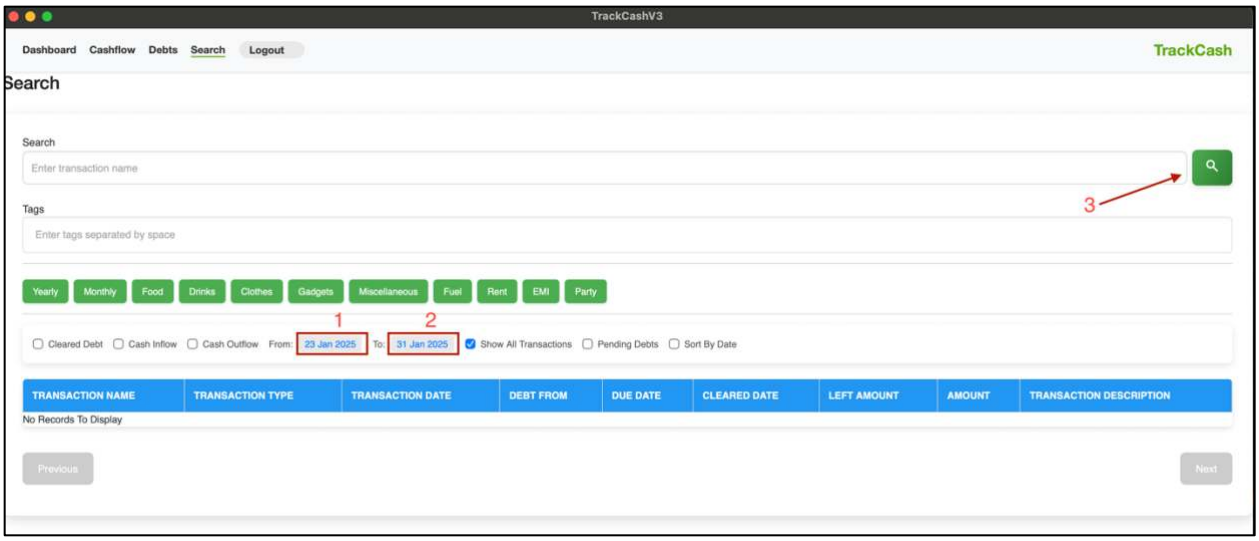

Figure 49: Date filtering

3.3.8. Dashboard testing

Test case 08	
Objective	To obtain the result as per the filtering.
Action	1) Navigate to homepage 2) Select transaction from date and to date that has transactions. 3) Click on filter button. 4) Select transaction from date and to date that has no transactions but a debt to be paid between that date. 5) Click on filter button.
Expected Result	Output data should be filtered by the input value.
Actual Result	Output data was filtered by the input value
Conclusion	Test was successful.

Table 8: Dashboard test

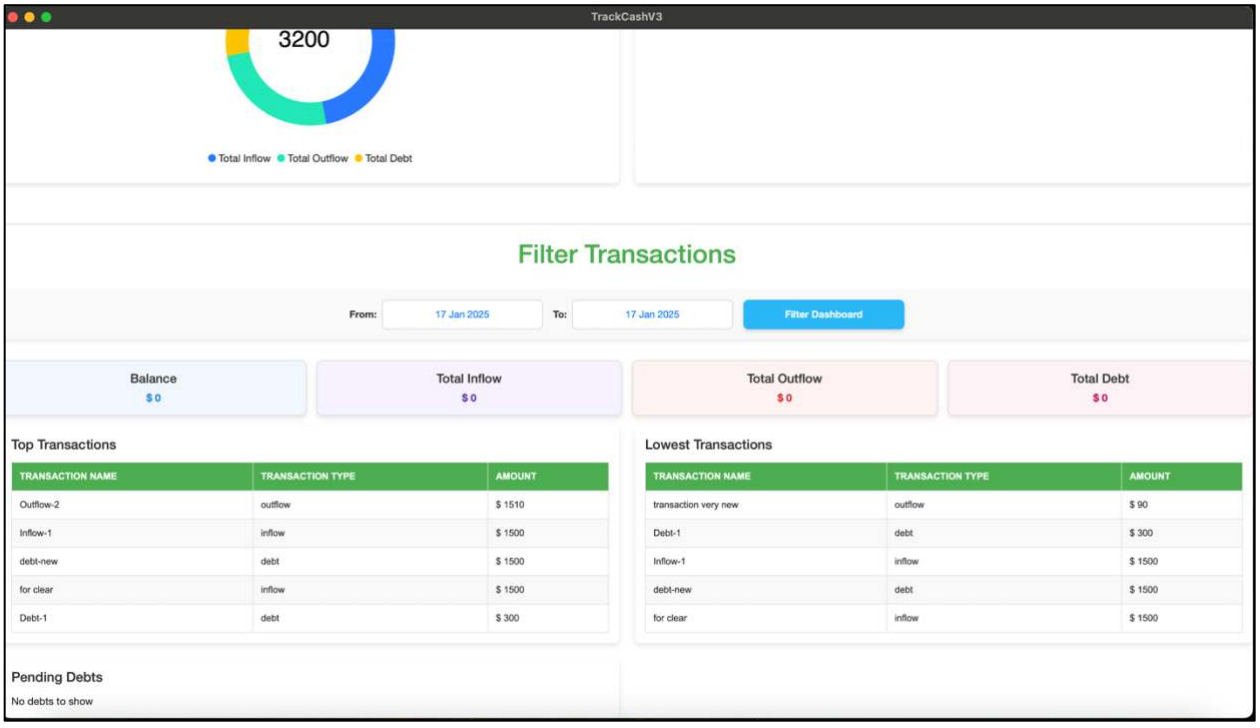

Figure 50: Dashboard before filtering

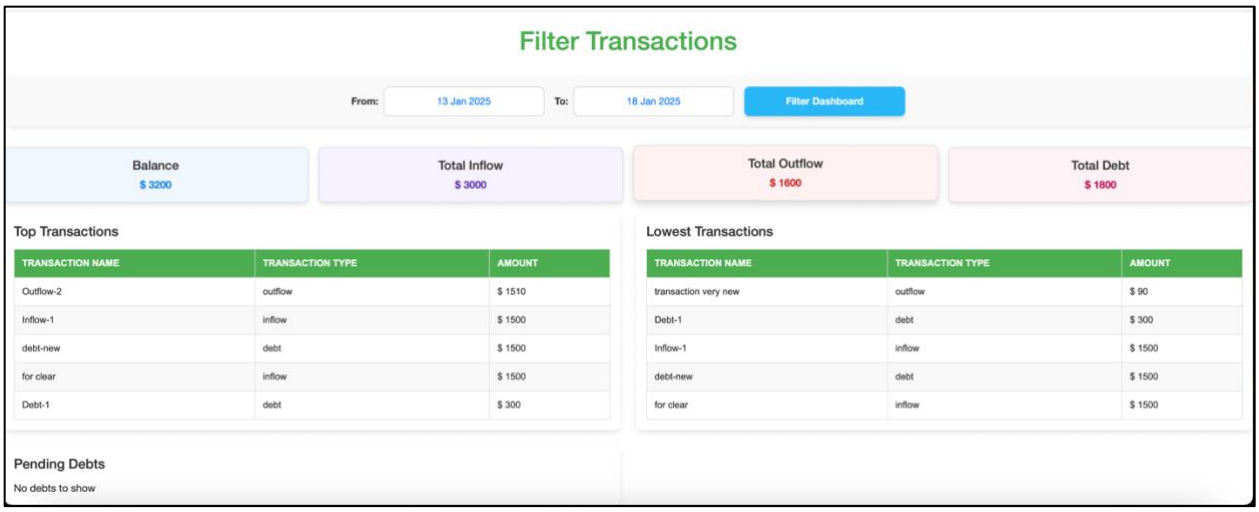

Figure 51: Dashboard with date filtering (1)

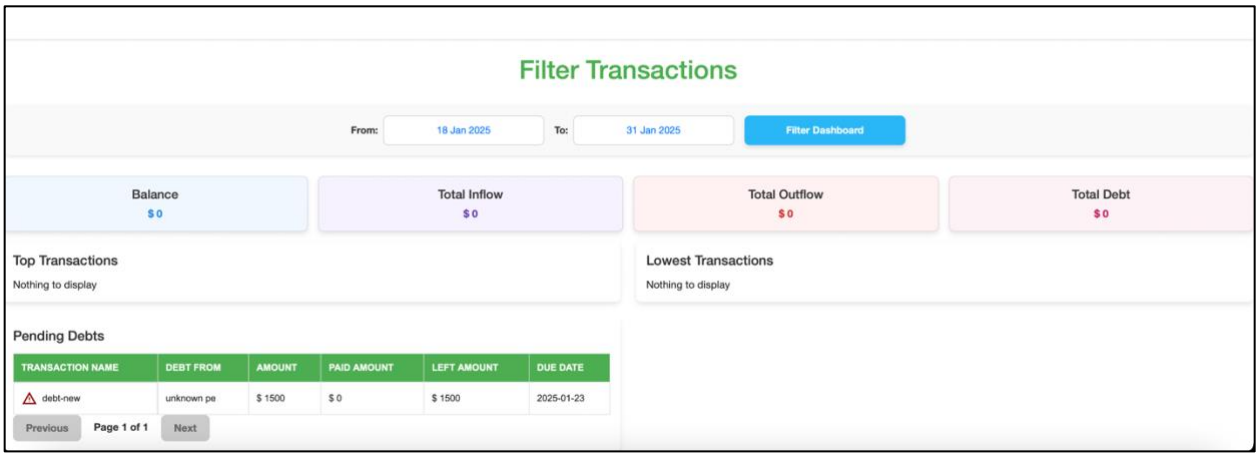

Figure 52: Dashboard with date filtering (2)

3.3.9. Insufficient balance outflow testing

Test case 09	
Objective	To make sure outflow transaction is stopped if balance is not sufficient.
Action	1) Navigate to cashflow page 2) Enter required fields. 3) Enter amount more than balance 4) Click on add transaction button.
Expected Result	A proper error message should be shown.
Actual Result	A proper error message was shown.
Conclusion	Test was successful.

Table 9: Insufficient balance outflow testing

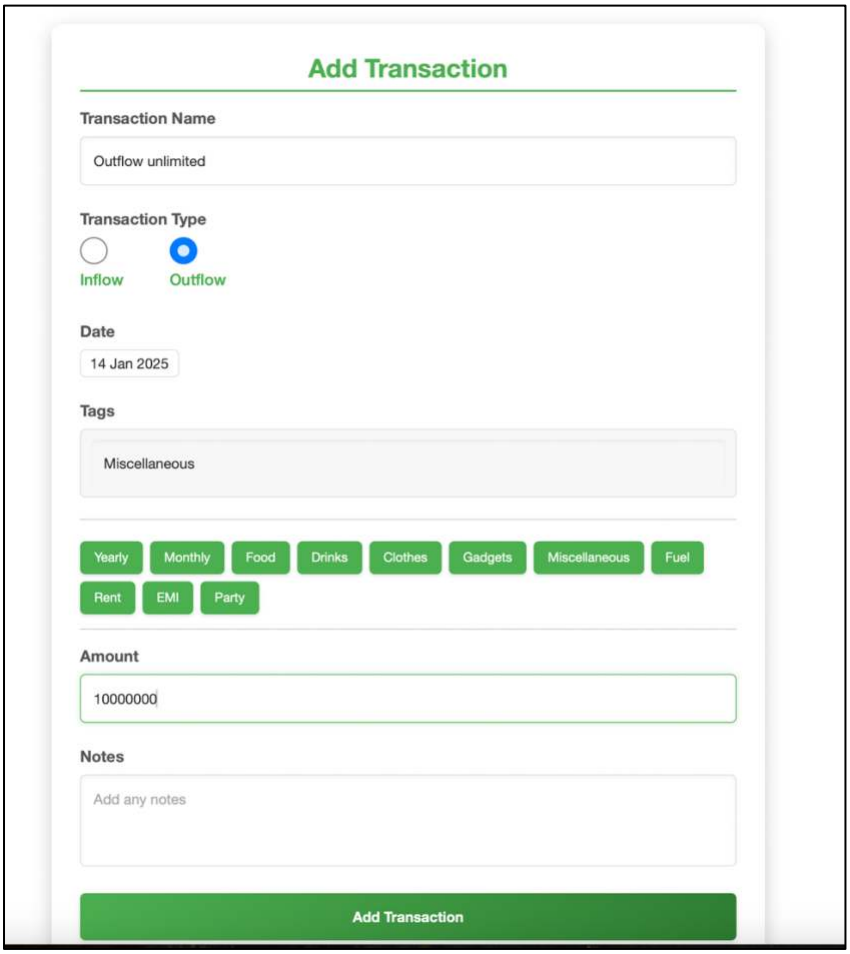

Figure 53: Large amount outflow

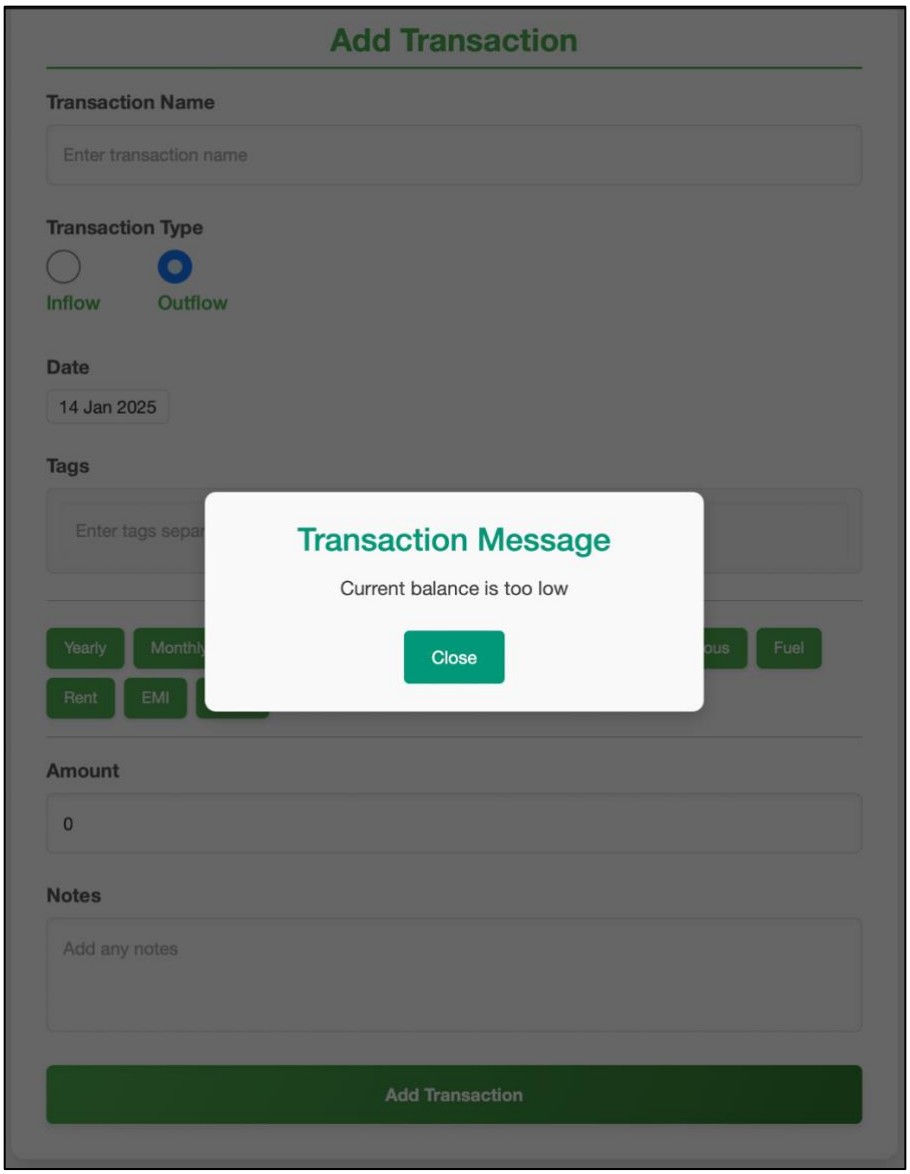

Figure 54: Low balance message.

3.3.10. Clearing debt testing

Test case 10	
Objective	To make sure debts are being cleared if sufficient balance
Action	1) Navigate to debt page 2) Select clear debt. 3) Select a pending debt from the list 4) Enter amount

	5) Click “Clear Debt”
Expected Result	Debt should be cleared
Actual Result	Debt was cleared
Conclusion	Test was successful.

Table 10: Clear debt testing

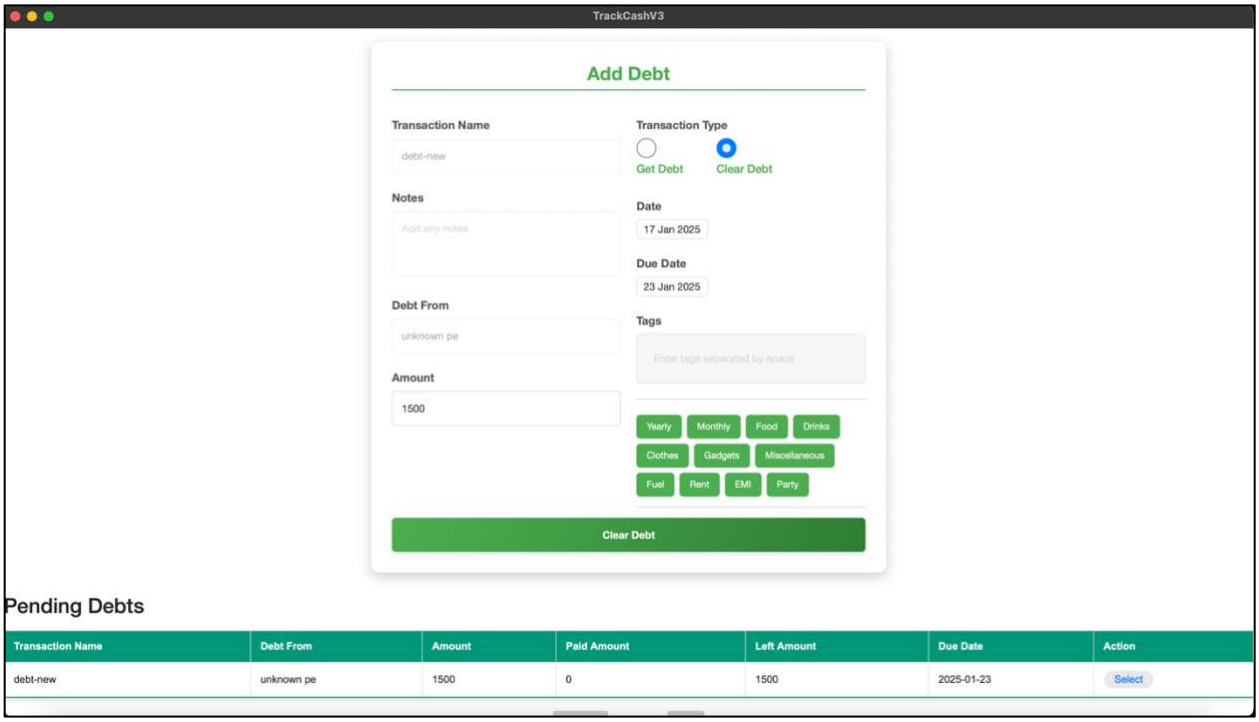

Figure 55: Clearing debt

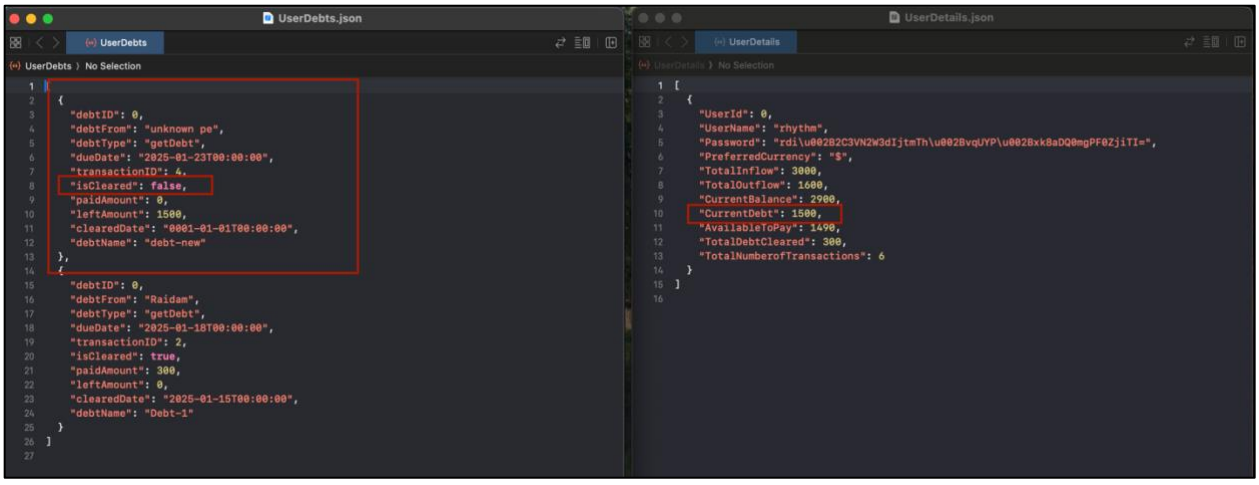

Figure 56: Json before clearing debt

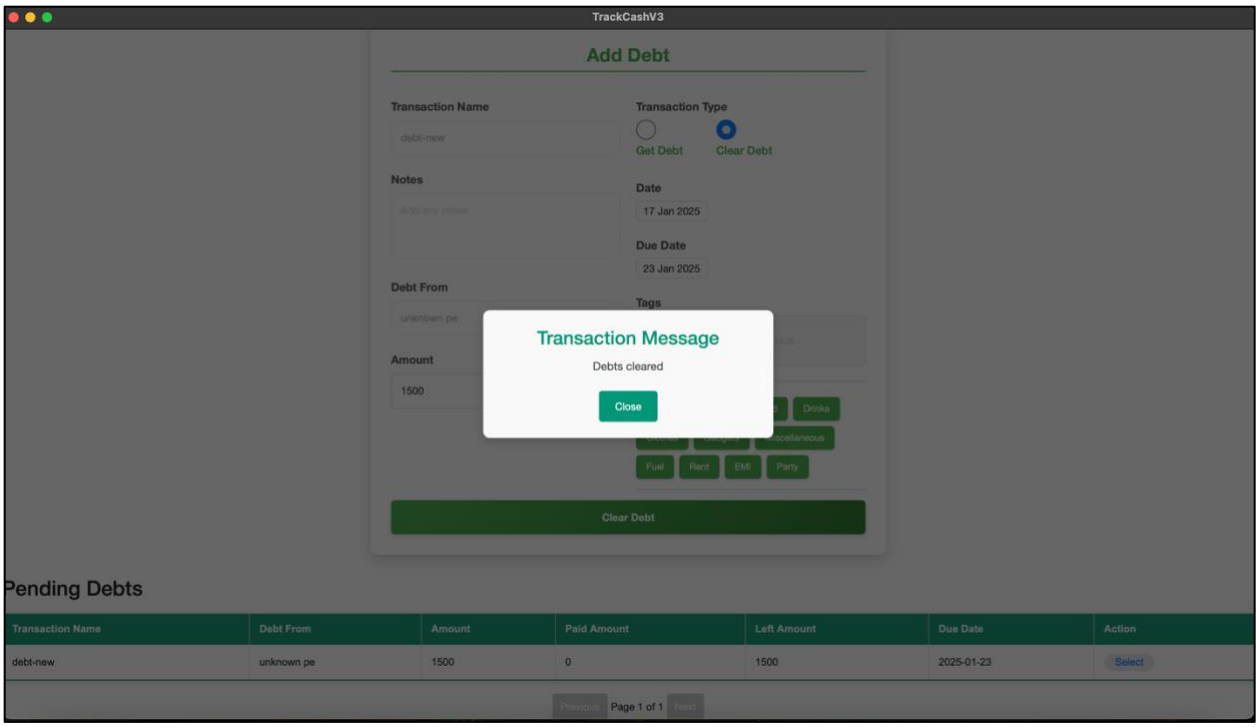

Figure 57: Debts cleared message

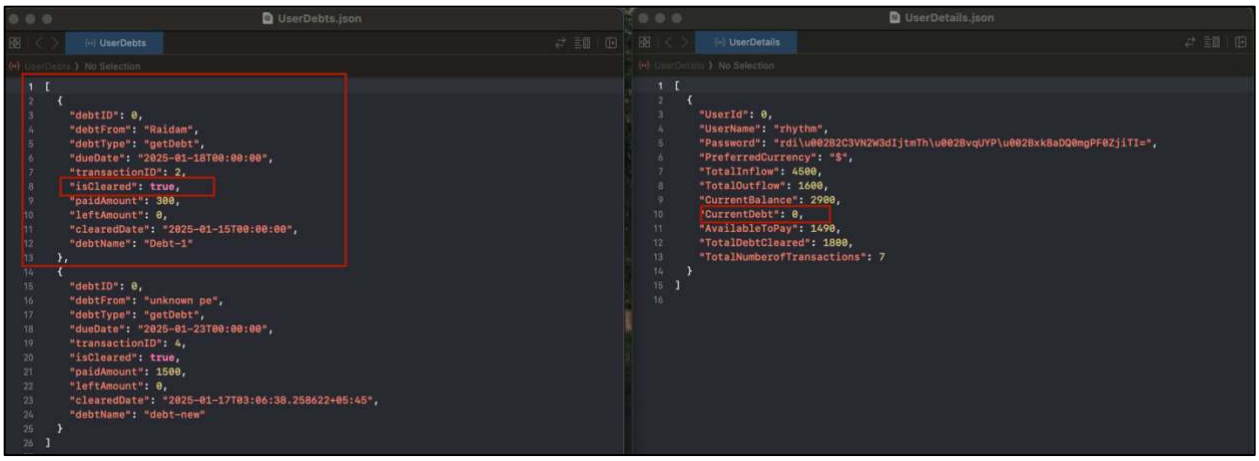

Figure 58: Json after clearing debt

NOTE: There might be slight changes in Json before clearing debt and after clearing, because a inflow was 1500 was added between the process.

3.3.11. Insufficient balance to clear debt

Test case 11	
Objective	To make sure a proper message is displayed if debts cannot be cleared
Action	1) Navigate to debt page

	2) Select clear debt. 3) Select a pending debt from the list that has more amount than available to pay balance 4) Enter amount 5) Click “Clear Debt”
Expected Result	Debt should not be cleared
Actual Result	Debt was not cleared with proper information.
Conclusion	Test was successful.

Table 11: Insufficient balance to clear debt testing

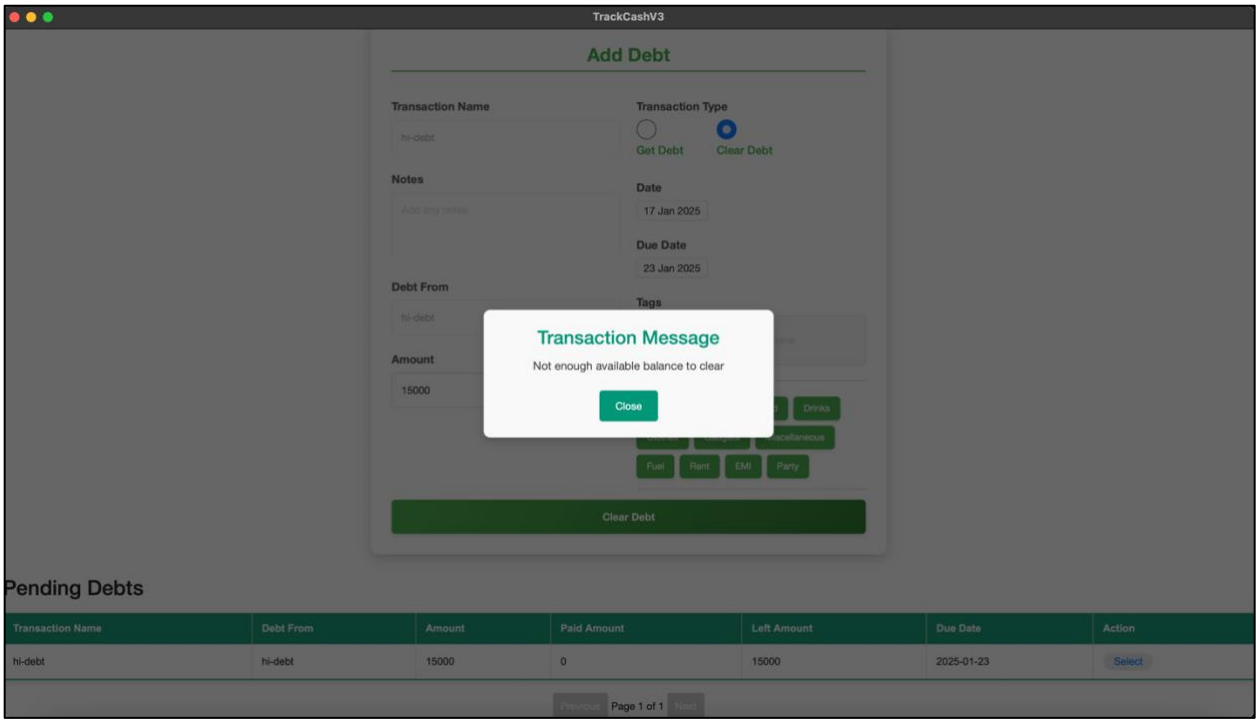

Figure 59: Error message while clearing debt

4. Individual Reflections

4.1. Roles and responsibilities

As an individual developer working on the development, there were some primary responsibilities that needed a high attention. Some major responsibilities were requirement analysis, where the system requirement was understood properly to make sure the application did not go out of scope.

Similarly, design and development were a huge task that played a vital role in development as seventy eighty percent of the progress falls under this stage. All the logic and functionalities of the application was developed in this phase.

Testing and debugging was performed after the completion of the application. For this application, blackbox testing was performed. For each and every functionality of the application, testing was performed to ensure the application was working properly and as expected.

4.2. Challenges faced

Integrating the search feature, was a challenge as multiple filtering was needed. The transaction loading part was easy, but obtaining result with all searches, tags and other filters enabled was challenging.

Since this was a completely new programming language, framework, adapting to the working environment was time consuming. The configuration of the project was a challenge as many problems occurred during the initial setup.

Keeping the UI simple yet integrating complex logic in the backend and making the application with required features was a challenging task. While designing the dashboard, the logic behind was time consuming and logically confusing, yet making it meaningful from the user perspective was one of the hardest challenges faced during this development.

Using third party libraries like MudBlazor for graphical representation was initially confusing to set up. Since Json was used a local storage, there was some performance issues as the application is not properly optimized when it needs to operate with whole Json file instead of needed records only.

4.3. Learnings and growth

4.3.1. Technical skills

A completely new and demanding programming language was learned properly during the development of this project. Getting familiar with popular frameworks like .Net was a challenge but a rewarding task. Concept of working with files asynchronously was further clarified. Development of MAUI hybrid app was also learned that uses razor files. Many technical skills were developed during the development of this application.

4.3.2. Problem-solving skill

The project helped in enhancing the debugging skill and solving problems that were faced constantly developing as many unexpected errors had occurred and some of errors had no messages. Furthermore, designing modular for problems was improved as it was highly required for the reusability of the code.

4.3.3. Project management

During the development, the management skill was developed drastically. The project could not be completed all at once, which required a well-planned division of tasks, priority wise. The project was divided into manageable tasks which helped in completion of development on time.

4.3.4. Documentation

This documentation is being written for both technical and non-technical readers, which should be considered. Hence, some part may be totally technical, it was kept as simple as possible. Describing a technical documentation to a non-technical reader helped in writing the documentation in more organized way.

4.4. Impact on future work

This project has significantly contributed to boost knowledge in various aspect including technical and non-technical skills. Working with one of the most demanding programming languages along with handling real-world based problems has boosted the thinking way. The application development has also opened path to solving other complex problem with ease and proper management. Additionally, importance of performance and prioritizing task lesson was learned which can be implemented while creating other applications.

5. Conclusion

The primary goal of this project was to create a well-developed and robust application that could be used to keep track of personal finances. The system is able to keep track of inflows, outflows, and debt. The system can filter the transactions according to need. The application fulfills the basic requirement for a finance tracking with some additional useful features.

Using .Net framework, a blazor hybrid application was developed that required a high level logical thinking and problem-solving skill with existing programming knowledge. The application was developed fulfilling the requirements that were required in the application yet maintaining a simple UI yet including all the necessary features.

5.1. Key findings

- The use of modular programming ensured the code is kept cleaned and developed for maximus reusability.
- Features like sorting date, filtering date by title, tags, date, and by transaction type provided a summary of the financial data to the user.
- Challenges occurred while working with asynchronous file and other bugs were successfully resolved by debugging and testing of the application.

5.2. Limitations

- Real-time synchronization with other cloud platform is yet to be developed.
- Users are not able to delete certain transactions.
- Due to Json file loading and uploading performance issues could be found when dealing with larger data.
- A report generation is yet to be implemented which could hold the details of the transactions of the user.

5.3. Future work

- The application could be developed for cross-platforms devices ensuring the easiness of the accessibility.
- Certain common features like, report download, and transaction deletion could be added.

- AI could be used for predicting the spending behavior of the user, and helping the user to control the expenses.
- The service could be made cloud based integrated with database, ensuring the application is well optimized.

6. Appendix

6.1. Code snippets

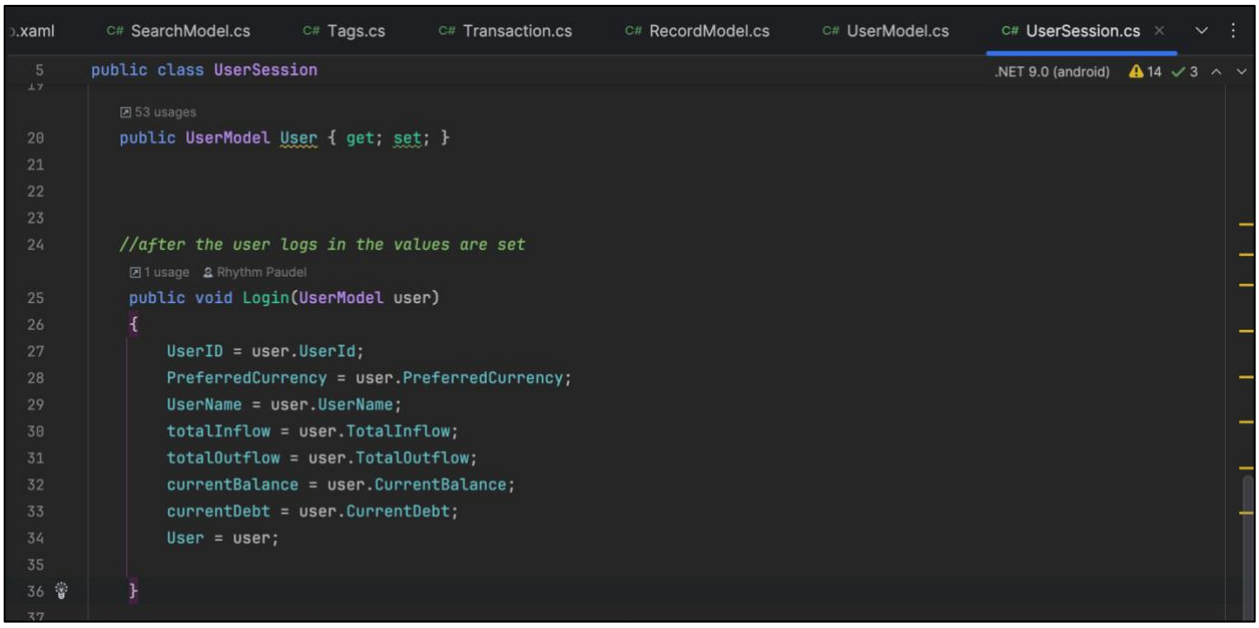

Figure 60: Logic for maintaining user log in session

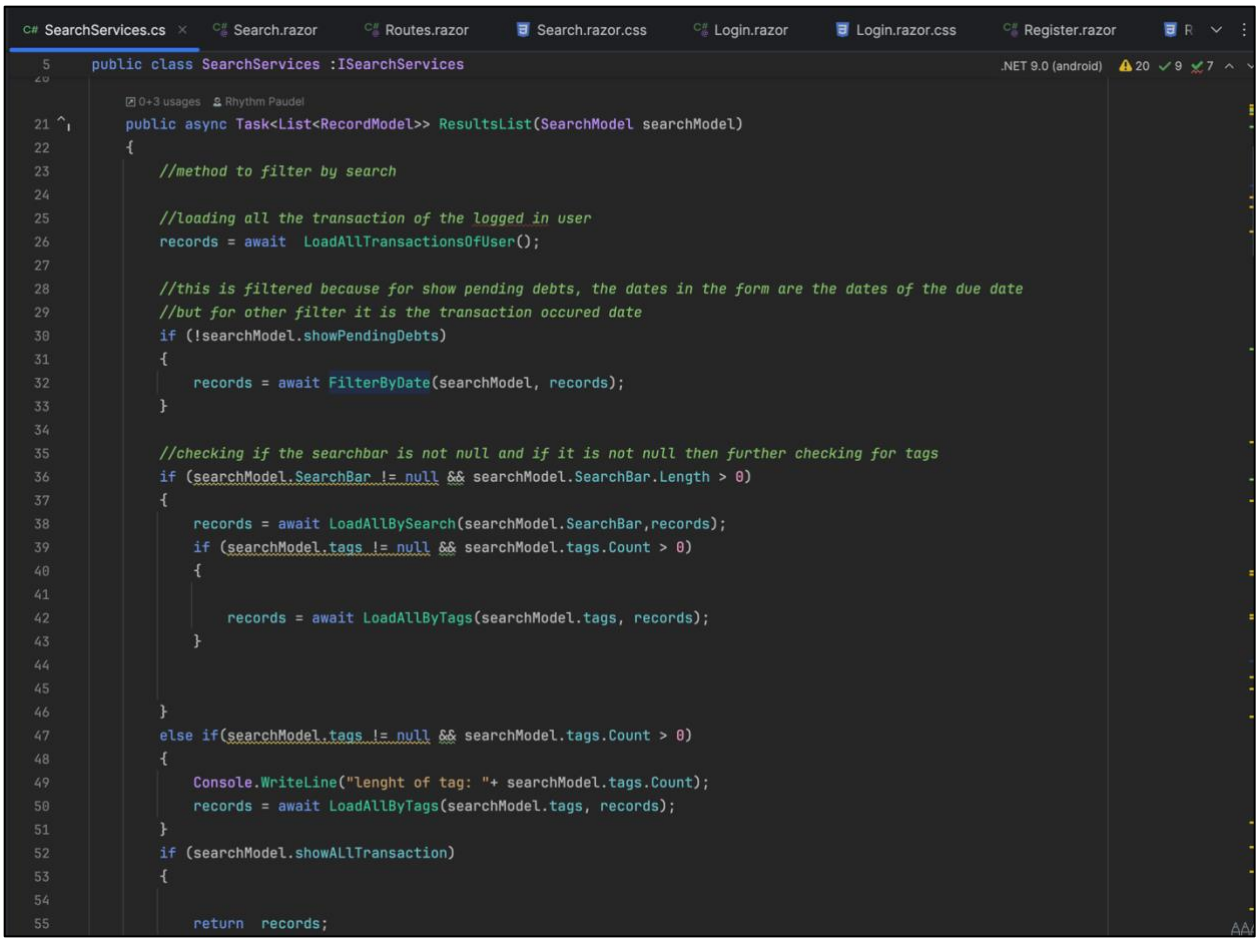

Figure 61: Logic for filtering transactions

```
if (transaction.transactionType == "inflow")
{
    try
    {

        transaction.transactionID = transactions.Count + 1;
        transactions.Add(transaction);
        await SaveTransactionsAsync(transactions);
        _userSession.User.TotalInflow += transaction.amount;
        _userSession.User.CurrentBalance += transaction.amount;
        _userSession.User.AvailableToPay += transaction.amount;
        _userSession.User.TotalNumberofTransactions += 1;
        await UpdateUserInfo(_userSession);
        return "Transaction saved successfully";
    }
    catch (Exception e)
    {
        return "Sorry An error occured while saving the transaction";
    }

}
```

Figure 62: Inflow logic

```
//outflow transaction logic
if (transaction.transactionType == "outflow")
{
    if (_userSession.User.CurrentBalance >= transaction.amount)
    {
        try
        {
            //adding transaction
            transaction.transactionID = transactions.Count + 1;
            transactions.Add(transaction);
            await SaveTransactionsAsync(transactions);
            _userSession.User.TotalOutflow += transaction.amount;

            //finding the difference
            double difference = _userSession.User.CurrentBalance - _userSession.User.AvailableToPay;
            _userSession.User.CurrentBalance -= transaction.amount;

            //this is checked because if the user has debt and inflow balance the outflow firstly is dedecuted f
            if (difference < transaction.amount)
            {
                _userSession.User.AvailableToPay -= transaction.amount-difference;
            }
            _userSession.User.TotalNumberofTransactions += 1;
            await UpdateUserInfo(_userSession);
            return "Transaction saved successfully";
        }
        catch (Exception e)
        {
            return "Sorry An error occured while saving the transaction";
        }
    }
    return "Current balance is too low";
}
```

Figure 63: Outflow logic

```
public class TransactionService: ITransactionService                                    .NET 9.0 (android)  23  5  9
    public async Task<String> ClearDebtsAsync(Transaction transaction)
    {
        try
        {
            DebtDetails debtDetails = await SelectDebtByTransactionID(transaction.transactionID);


            //since the user can also pay only certain amount of the debt the condition for paying the debt is checked
            if ((_userSession.User.AvailableToPay >= debtDetails.leftAmount &&
                 transaction.amount<=debtDetails.leftAmount) || (transaction.amount <=_userSession.User.AvailableToPay
                                                                 && transaction.amount <= debtDetails.leftAmount))
            {
                debtDetails.paidAmount += transaction.amount;
                debtDetails.leftAmount -= transaction.amount;

                //if only certain amount is paid is cleared field should be updated accordinly
                debtDetails.isCleared = debtDetails.leftAmount > 0 ? false : true;
                if (debtDetails.isCleared)
                {
                    debtDetails.clearedDate = transaction.date;
                }

                //updating current logged in user information and updating debts in json file
                await UpdateDebtsAsync(debtDetails);
                _userSession.User.CurrentDebt -= transaction.amount;
                _userSession.User.CurrentBalance -= transaction.amount;
                _userSession.User.AvailableToPay -= transaction.amount;
                _userSession.User.TotalDebtCleared += transaction.amount;
                await UpdateUserInfo(_userSession);

                return "Debts cleared";

            }

            return "Not enough available balance to clear";
        }
        catch (Exception ex)
```

Figure 64: Clearing debt logic