ITEC5010F:
Applied Prog I
Expense Tracker Project Report

Nadia Fathima Shabeer 101278538

Supervised By: Mehdi Niknam

Submitted: December 14, 2023

Contents

List of Figures

1 Introduction

The essential component of today's financial management is efficient expenditure tracking. This report describes implementing and analyzing a web application built with Flask that helps users manage their finances. The application's user-friendly interface enables users to track spending, create budgets, and provide thorough reports that help them better understand their spending habits. The idea for the project came from the need for people to have a simple and effective tool for managing their finances, freeing them from having to rely on pricey outside services or complex software. Its main goal is to provide consumers with a user-friendly, adaptable platform for tracking expenses and managing their finances.
The report looks more into the project analysis, covering the technologies used, the implementation details, and an overview of the program's features. It also explores the design and architecture of the system, providing a critical evaluation of the project's advantages and disadvantages. Finally, the paper ends with recommendations for prospective modifications and improvements in the future.

2 Objective

The purpose of this online application is to give consumers a simple and easy-to-use platform for tracking costs and budgeting. It places a high priority on data security by using an effective database management system and a strong authentication system. The application seeks to enable users to take charge of their money by enabling informed spending decisions through client-side validation and performance monitoring tools, as well as an intuitive interface that makes expense input and tracking simple.

3 System Overview

This web tool for tracking expenses is Flask-based and uses Python's micro web framework, which is well-known for its ease of use in web development. A variety of tools and extensions provided by Flask facilitate the building of simple applications and smooth database integration.
This application's main goal is to make it possible for users to keep track

of their spending by managing budgets, classifying costs according to payers and categories, and creating reports that visually represent spending. A PostgreSQL database allows users to safely register, log in, and store data. Budget management, thorough spending trend statistics, and expense addition and categorization are important elements. It also makes user account management easier with features like email confirmation and password reset. Important technologies used in the creation of this application are:

- Flask is a lightweight Python web framework for building online applications.
- SQLAlchemy is a Python-based ORM library and SQL toolkit for effective database administration.
- A Strong open-source relational database system for safe data storing is PostgreSQL.
- Popular template engine Jinja2 is used to create dynamic HTML templates.
- A popular framework for creating mobile-first, responsive websites is called Bootstrap.

The program provides an easy-to-use interface for creating reports and managing expenses. It makes the most of Flask's features and incorporates other technologies to guarantee a robust and safe online application.

4 Design

4.1 Design Process

1. **Requirement gathering and planning:** First, I listed the technologies that would be needed and described the functionality and scope of the application.
2. **Development Environment Setup:** After that, I installed Python, Flask, PostgreSQL, and the necessary packages to create the development environment.

3. **Database Schema Design:** I used PostgreSQL to organize the database, making tables for user information, spending plans, invoices, and categories.

4. **Implementation of Authentication and Authorization:** I used session cookies and Flask-Login to secure the application, granting access only to those who are approved with a login-required decorator.

5. **User Interface Development:** I created the interface using page templates and client-side validation to ensure consistent user inputs using Bootstrap, HTML, and CSS.

6. **Implementation of Main functions:** All major functions, including budget generation, spending management, registration/login, and classification, were carefully tested and put into use.

7. **Testing, Debugging, and Deployment:** Thorough testing and debugging, it is ensured that the program was stable, which led to its local deployment.

   The project proceeded in an organized manner, guaranteeing compliance with the requirements, extensive testing, and debugging before the deployment stage.

4.2 System Architecture (High-Level Overview)

The program has a typical three-tier architecture, with the data storage layer (Model), presentation layer (View), and business logic layer (Controller) making up the layers.
The Flask framework is utilized for server-side processing in conjunction with HTML, CSS, and JavaScript for the presentation layer. Jinja2 is incorporated to generate dynamic HTML.
The business logic layer is managed by Flask and Python, where Flask helps with request routing and points requests to the appropriate controller methods. These operations respond to queries, carry out business logic, communicate with the data storage layer, and return data for display.
The PostgreSQL database is the foundation of the data storage layer, which uses SQLAlchemy, an Object-Relational Mapping (ORM) tool, to abstract SQL syntax into a language more easily understood by Python.

The application also has modules and utility functions for tasks like reporting, data validation, and user authentication.
Because of the architectural design's guarantee of *modularity* and *scalability*, adding new features is easy and maintenance is simplified.

4.3 Design Components

4.3.1 Project Model

The database schema and its interactions are handled by the application's model section. It uses SQLAlchemy Object-Relational Mapping (ORM) to create and manage the database structure. This component contains the following files, each of which has a specific purpose:

- **models.py:** Uses SQLAlchemy ORM to define the database table structures and relationships.
- **app_expenses.py:** Contains functions for interacting with the expenses table in the database.
- **app_categories.py:** Houses routines that interact with the database's categories and budgetcategories tables.
- **app_budgets.py:** Contains functions for interacting with the database's budgets and budgetcategories tables.

4.3.2 User Interface Design

The application's view component governs the user interface and how data is presented to users. It includes HTML templates and CSS stylesheets, which determine the application's visual appearance and layout. This component is made up of the following parts:

- **Templates:** This directory contains HTML templates that specify the structure and content of various pages inside the application.
- **Static:** This directory includes CSS stylesheets used to format and style HTML templates, improving visual presentation and user experience.

4.3.3 Application Logic (Controller)

The controller component serves as the application's nerve center, regulating the business logic and workflow. It includes Python scripts that are in charge of managing HTTP requests and responses, as well as coordinating interactions between the model (data) and view (presentation) components to provide appropriate responses for users. This essential component of the application consists of the following files:

- **app.py:** The basic Flask application, containing the routing code that directs and processes incoming HTTP requests.
- **auth.py:** This module is in charge of the authentication and authorization functions used by the Flask application to ensure secure user access and permissions.
- **app_reports.py:** This file contains routines for generating reports for usage within the Flask application, which aid in the extraction and presentation of specific data for users.
- **helpers.py:** This file contains utility functions that are required for various tasks within the Flask application, streamlining and improving its functionality.

4.3.4 Database Component

The database component acts as the application's backbone for data storage and retrieval. Its principal function is to administer the PostgreSQL database, which stores user-related information such as spending, budgets, categories, and payers. This section contains numerous important files:

- **schema.sql:** The SQL schema used to construct and create tables in the PostgreSQL database, detailing their structure and relationships, is contained in this file.
- **config.py:** This file stores configuration settings pertinent to the PostgreSQL database, ensuring seamless integration and functionality.
- **db.py:** This file contains the code that handles establishing connections to the PostgreSQL database using SQLAlchemy, enabling communication between the application and the database.

Each component of the application works independently, catering to distinct functions. This separation helps with code maintenance, testing, and modification. Furthermore, these components have loose coupling, which allows one component to be changed or replaced without affecting others, allowing flexibility and ease of modification without widespread consequences.

4.3.5 MVC Architecture in Flask Application

Flask is built on the Model-View-Controller (MVC) architecture, which splits the program into three distinct layers: Model, View, and Controller.

- **Model Layer:** In charge of data storage and retrieval. The SQLAlchemy library is used in this application as an object-relational mapper (ORM) to facilitate interaction with relational databases. This mapping enables the handling of database data in a more object-oriented manner.
- **View Layer:** It is in charge of displaying data to consumers in an understandable fashion. HTML, CSS, and JavaScript are used to create the View layer. Flask includes Jinja2 as its templating engine, allowing the building of dynamic HTML templates that smoothly render server data.
- **Controller Layer:** It is in charge of managing user requests and directing them to the relevant Model or View components. This layer, implemented using Flask, provides a lightweight Python framework for web app building. Flask creates URL pattern mappings to Python functions, allowing interactions with the Model layer and HTML template rendering via the View layer.

By isolating concerns and decreasing interdependencies across multiple application components, the MVC design pattern acts as a structural framework, improving code organization, maintainability, and scalability. This separation allows for easy updates and additions to certain layers without requiring changes throughout the program, enabling flexibility and efficient development.

5 Result

5.1 Application Structure and Components Analysis

5.1.1 Database Schema Overview: Project Tables Description

This section describes the project's numerous tables, each of which serves a specific purpose within the spending tracker application.

1. **Users Table:** Stores information on registered app users, such as usernames, password hashes, income, registration, and last login dates. It maintains a one-to-many link with budgets and expenses, allowing each user to be associated with one or more budgets and expenses.

2. **Budgets Table:** This table contains information on individual user budgets such as budget names, assigned amounts, and owner user IDs. It has a many-to-one link with the users' table, allowing them to have multiple budgets.

3. **Table of categories:** Stores many expense categories such as food, housing, and dining out. It has a many-to-many link with the tables userCategories and budgetCategories.

4. **User Categories Table:** Creates a many-to-many relationship between users and expense categories, allowing users to choose specific expense categories for tracking. This table contains foreign keys that refer to the table's users and categories.

5. **Budget Categories Table:** This table establishes a many-to-many relationship between budgets and expense categories, allowing users to allocate budget amounts to specified categories. It has foreign keys that refer to budget and category tables.

6. **Payers Table:** This table stores information about individuals who paid for expenses, such as their names and user IDs. It has a foreign key that refers to the users' table.

7. **Expenditures Table:** This table contains information about each expense recorded in the program, such as the description, type, amount, payer, and owner user ID. It is linked to the users' table via a foreign key.

The database schema is rigorously developed to suit the application's spending tracking features efficiently. It facilitates a structured and organized approach to expenditure management by allowing for fast querying and updating of user, budget, category, and expense data.
In Figure 1, you can see the table for *expenses* on pgadmin4 which is an open-source graphical user interface (GUI) administration tool for PostgreSQL databases. Using the script below we select all the variables to display the entire table in the figure.

```
1 SELECT * FROM public.expenses;
```

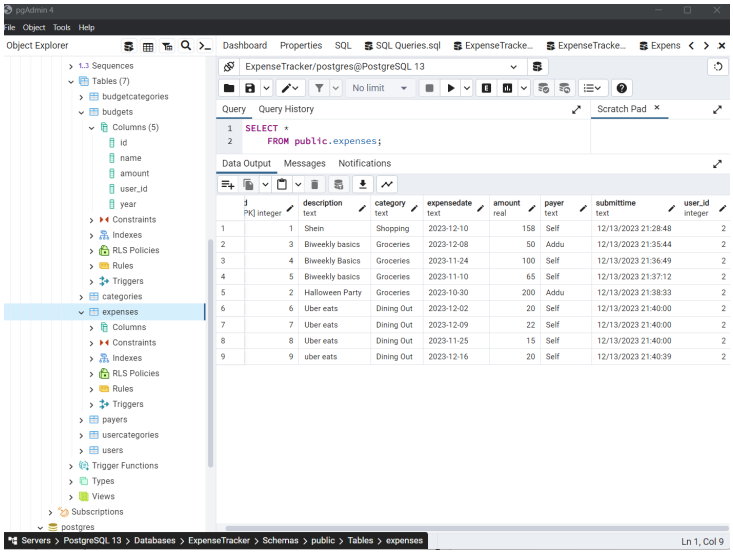

Figure 1: PostgreSQL databases on pgAdmin 4 user interface..

The SQL code for creating each of the tables is provided in the Appendix A

5.1.2 Application Features

1. **Registration and Login:**
   To access the application, users must first register for an account and then log in. While logging in if the username or password is wrong, a fun image will appear stating so. This can be seen in Figure 2

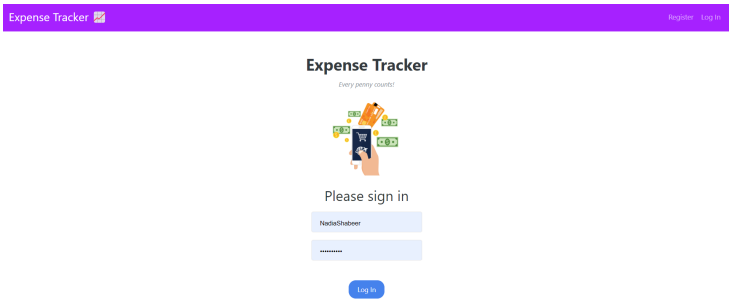

Figure 2: Login Page.

2. **Add and Edit Expenses:**
   Figure 3 shows how users can add expenses to their accounts by providing a description, the amount spent, the expense category, and the payer's name. After a cost has been added, users can amend or delete it.

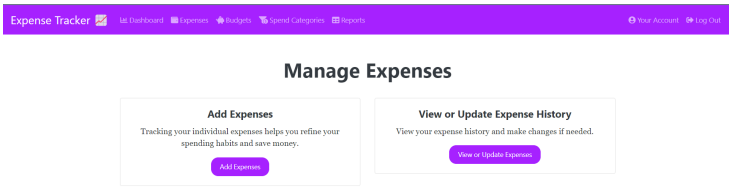

Figure 3: Manage Expenses Page.

3. **Create and Manage Budgets:**
   Figure 4 shows how users can construct budgets by giving the budget a name and dollar value. Users can also add budget categories and apply dollar amounts to each one. After creating a budget, users can change or delete it.

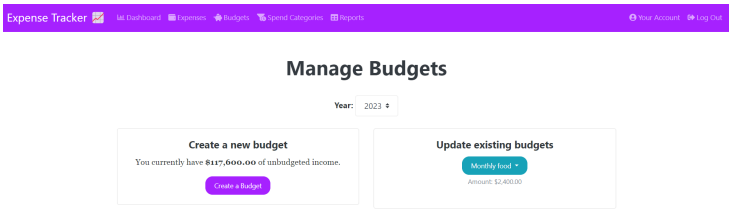

Figure 4: Budget Management Page.

4. **Categorize Expenses:**
   Figure 5 shows how users can organize their spending into predefined categories or build their own. Categorizing expenses allows customers to track their spending better and understand where their money goes.

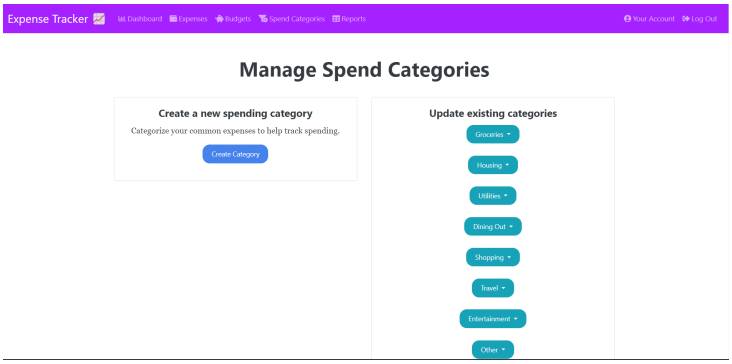

Figure 5: Manage Expense Category Page.

5. **View and Filter Expenses:**
   Users can examine and filter their spending by date, category, and payer. Users can utilize the filter tool to focus on specific expenses and better understand their spending trends.

6. **Add and Manage Payers:**
   Figure 6 shows how users can examine and filter their spending by date, category, and payer. Users can utilize the filter tool to focus on specific expenses and better understand their spending trends.

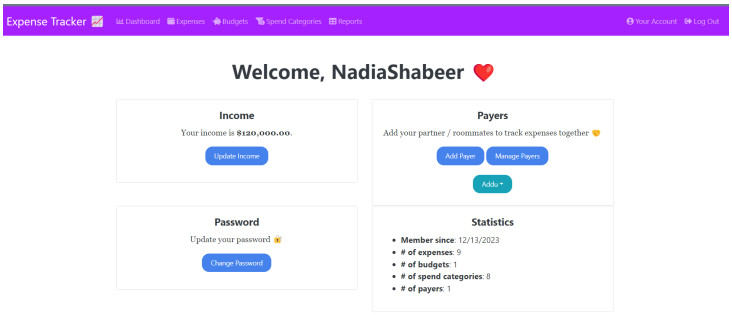

Figure 6: Your Account Page: Manage Payers.

7. **Track Income:**
   The app allows clients to monitor their revenue by entering it manually or by giving a default amount. Tracking income allows customers to efficiently build budgets and organize their expenses.

8. **Dashboard:**
   The software has a dashboard that shows an overview of the user's expenses, income, and budgets. The dashboard displays a summary of the user's finances and assists users in keeping track of their expenses.

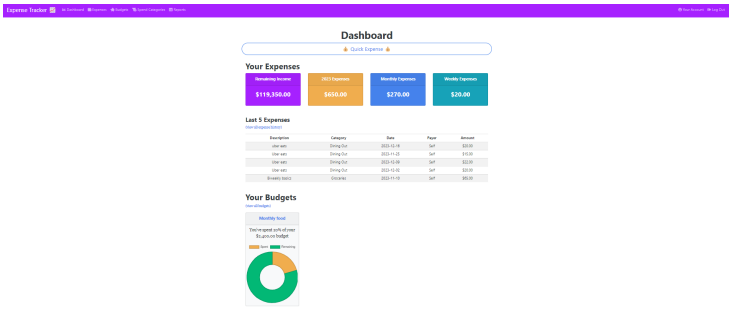

Figure 7: Dashboard Main Page (a).

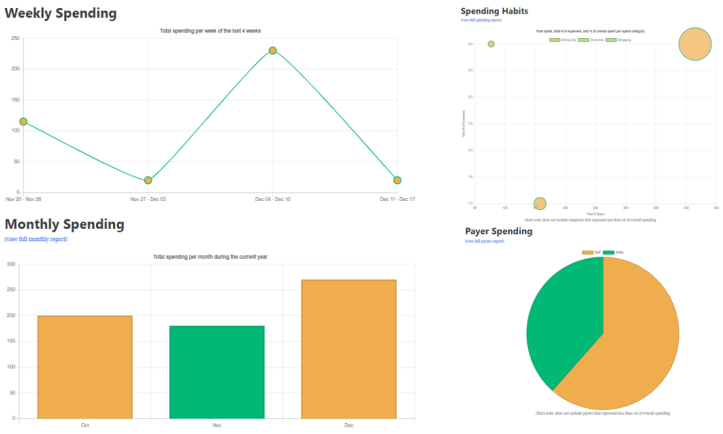

Figure 8: Dashboard Main Page (b).

9. **Profile:** Figure 9 shows that users can access and change their profile information, including their login, password, and salary, using the app. Users can make changes to their information at any time.

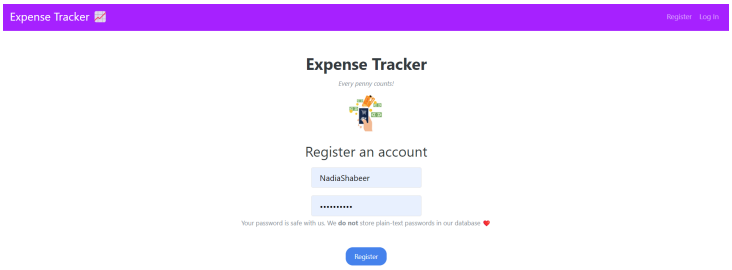

Figure 9: Register Page.

10. **Log Out:** Once the users are done tracking their expenditures, they can safely log out of the application.

5.1.3 Application Functionality

1. **Registration and Login:**
   The user registration and login systems of the app make use of Flask's session management and password hashing features. Users' passwords are hashed and securely stored in the database's *users* table when they

register. Passwords are hashed during login and compared to the stored hash for authentication. The user's ID is saved in the session for future requests after successful authentication. This functionality is implemented in the files *app.py* and *helpers.py*.

2. **Adding and Managing Budgets:**
   Users have budget control via the Budgets tab, which allows users to create, update, and delete budgets. These budget records are stored in the database's *budgets* table and are linked to users via the *user_id* field. These actions' functionality is coded in the *app.py* and *helpers.py* scripts.

3. **Adding and Managing Categories:**
   The Categories tab gives users authority over categories, allowing them to create, edit, and delete category items. These categories are saved in the database's *categories* table. The functionality for controlling these activities is coded in the files *app.py* and *helpers.py*.

4. **Adding and Managing Expenses:**
   Users have the option to include and manage expenses on the Expenses page. They can create new expense records, amend the information of existing ones, and delete expenses from their records. These expenses are saved in the database's *expenses* table, with the *user_id* column showing the user associated with each expense. Additionally, consumers can receive a simplified overview of their spending organized by expense type or payer. The implementation of these features is split across the *app.py* and *helpers.py* scripts.

5. **Overview of User Dashboard:**
   The dashboard provides customers with a rapid summary of their current month's budget, expenses, and balance. This data is retrieved from the database and displayed.

5.1.4 Program Files

The main program file, app.py, is the heart of the Flask application, containing the application structure and routes. It includes necessary package imports required for various functionalities:
In app.py, type:

- **Flask:** The primary package for developing and managing Flask applications, which handles HTTP requests and responses.
- **Flask_session:** A Flask extension that allows for server-side session support within the application.
- **Werkzeug.security:** A password hashing and verification tool.
- **Helpers:** A custom module containing critical helper functions that are used throughout the application.

Models.py describes the database schema and creates database tables using the SQLAlchemy module. The following packages are imported by this file:

- **datetime:** A set of functions for managing dates and times.
- **sqlalchemy:** Object-Relational Mapping (ORM) support for Python database interactions.

Furthermore, the helpers.py file contains several critical utility functions used throughout the application. The following packages are imported by this file:

- **OS:** Allows communication with the operating system.
- **requests:** Helps with HTTP request handling. urllib.parse: Helps with URL parsing and manipulation. Support for decimal floating-point arithmetic.
- **functools:** Provides higher-order functions for function operation.

These imported packages work together to create and execute the Flask application, manage sessions, handle database interactions, and provide a variety of assistance methods across multiple functionalities.

5.1.5 Implementations

For its functionalities, the Flask web application relies on a variety of technologies and frameworks, combining flexibility and efficiency:

- **Flask Web Framework:** Flask is a versatile and lightweight Python framework that provides a variety of tools and capabilities suited for web app development.

- **PostgreSQL Database:** Using PostgreSQL, a strong and scalable database management system noted for its reliability in managing massive datasets, to store critical user, budget, and expense-related data.
- **HTML and CSS:** The user interface of the program is designed with HTML for structure and content and CSS for aesthetic styling, resulting in an engaging visual presentation.
- **Bootstrap Framework:** Using Bootstrap, a popular front-end framework, helps improve the appearance of the application. Bootstrap provides pre-designed components and classes for creating responsive and visually appealing web pages.
- **Jinja2 Template Engine:** Using Jinja2, a dynamic and powerful template engine, to develop dynamic websites. Jinja2 allows developers to create HTML templates that include placeholders for dynamic content insertion.
- **SQLAlchemy ORM:** Using SQLAlchemy as an Object-Relational Mapping (ORM) tool to streamline database interactions. SQLAlchemy, a well-known Python module, provides a high-level interface for interacting with relational databases.
- **bcrypt for Password Hashing:** Using bcrypt, a highly secure password hashing method, to efficiently encrypt passwords. Bcrypt provides strong security against password-related security breaches.

5.1.6 User Interface

The project's user interface is built with HTML, CSS, and Jinja2 templates and includes the following components:

- **Navigation bar:** It developed with the Bootstrap framework, sits atop each page, and includes links to the home, budget, expense, categories, and logout pages.
- **Forms:** Forms such as login, registration, add a budget, add expense, and add category allows users to enter information. HTML and Bootstrap are used to structure these forms.

- **Tables:** Data is represented using tables such as budget, expense, and category tables, which are styled with HTML and Bootstrap.
- **Modals:** They are pop-ups that present users with additional information or confirmations, such as delete budget, remove expense, and delete category prompts.
- **Flash Messages:** For a smoother interface experience, informative messages (e.g., login/logout confirmation, registration, adding, changing, or deleting budgets, spending, or categories) are displayed to users via Jinja2 templates.

5.2 Challenges

Authentication and Authorization: One of the main challenges was putting in place a secure framework for these processes. Flask-login and session cookies were used to handle user authorization and authentication securely to remedy this. It made sure that user sessions were protected from possible efforts by malevolent attackers to take control of them. Further strengthening security was the addition of a 'login-required' decorator, which limited access to authenticated users on particular sites.

UI/UX Design: It was difficult to design a user-friendly and straightforward interface. Using HTML, CSS, and Bootstrap to create a UI that is both aesthetically pleasing and responsive was the solution. It was essential to make sure the user interface (UI) was easy to use, accessible, and responsive on different devices. Furthermore, to ensure authenticity and consistency, client-side validation procedures were implemented to ensure that user inputs satisfied predetermined requirements.

Database Design and Management: Creating an effective database structure that can store a variety of data types with the least amount of redundancy was the challenge at hand. The database management system that will be used to hold user-related data, budget specifics, spending statistics, and category names is PostgreSQL. To maximize storage and retrieval effectiveness, strong linkages between tables and data normalization were prioritized. Flask-Migrate was used to manage database schema changes efficiently by providing a streamlined method of handling updates.

5.3 Future Improvements

- **Integration with Financial Services:** To automate processes like budget adjustments based on real-time financial data, account synchronization, or transaction classification, investigate integrating APIs or services from financial institutions.

- Extra Security Measures: To protect user data from possible security breaches, implement extra security layers like two-factor authentication (2FA) or improved encryption methods.

- Localization and Customization: To accommodate a varied user base and improve user experience, allow users to customize parameters such as language choices, date formats, or currency representations.

- Expense Prediction and Recommendations: Make advantage of machine learning algorithms to forecast future costs by analyzing historical data trends, and provide consumers with tailored budgeting advice.

- Support and Feedback: Include a support system that offers quick assistance or query resolution, as well as a way for users to submit recommendations or feedback directly within the program.

- Multi-level User Roles: To give distinct users or user groups varying degrees of access and permissions, implement a role-based access control system. This can entail designating roles with distinct permissions within the program, such as managers, regular users, and administrators.

6 Conclusion

The goal of the Expense Tracker project is to empower users in their endeavors to manage their expenses by providing a comprehensive web-based solution. With its extensive feature set, the project helps users to manage many financial issues efficiently. Features include budget development and management, careful spending addition and classification, and the production of thorough expense reports. Flask, a popular Python web framework, serves as the project's backbone, while a PostgreSQL database is used to store data. The program makes use of a variety of technologies, including

JavaScript, HTML, and CSS, to guarantee a user-friendly and responsive interface.

By successfully aligning its functionality with user needs, the application fulfills its primary objective of offering a straightforward and effective expense-tracking mechanism. Users can effortlessly craft budgets, assigning specific allocations to distinct categories, thereby fostering efficient expense management. Moreover, the application streamlines the process of adding and categorizing expenses, simplifying the task of monitoring expenditures without complications. Through its robust capacity to generate in-depth expense reports, the application furnishes users with invaluable insights into their spending behaviors. This functionality becomes instrumental in enabling users to make informed and astute financial decisions, leveraging the knowledge derived from detailed expense analyses.

References

[1] Pallets Projects, "Flask Documentation," https://flask.palletsprojects.com/en/2.0.x/.

[2] SQLAlchemy, "SQLAlchemy Documentation," https://docs.sqlalchemy.org/en/14/.

[3] Bootstrap, "Bootstrap Documentation," https://getbootstrap.com/docs/5.1/getting-started/introduction/.

[4] PostgreSQL Global Development Group, "Postgresql Documentation," https://www.postgresql.org/docs/.

[5] Python Software Foundation, "Python Documentation," https://docs.python.org/3/.

[6] Real Python, "Real Python Flask Tutorials," https://realpython.com/tutorials/flask/.

[7] Stack Overflow, "Stack Overflow," https://stackoverflow.com.

A SQL Code

```
CREATE TABLE IF NOT EXISTS users (
        id      serial PRIMARY KEY,
        username        TEXT NOT NULL,
        hash    TEXT NOT NULL,
        income REAL NOT NULL DEFAULT 60000.00,
        registerdate    TEXT NOT NULL,
        lastlogin       TEXT NOT NULL
);

CREATE TABLE IF NOT EXISTS budgets (
        id      serial PRIMARY KEY,
        name    TEXT NOT NULL,
        amount REAL,
        user_id INTEGER NOT NULL,
        year    INTEGER
        CONSTRAINT budgets_user_id_fkey FOREIGN KEY (user_id)
                REFERENCES users (id) MATCH SIMPLE
                ON UPDATE NO ACTION ON DELETE NO ACTION
);

CREATE TABLE IF NOT EXISTS categories (
        id      serial PRIMARY KEY,
        name    TEXT NOT NULL
);

CREATE TABLE IF NOT EXISTS userCategories (
        category_id     INTEGER NOT NULL,
        user_id INTEGER NOT NULL,
        CONSTRAINT userCategories_category_id_fkey FOREIGN KEY
            (category_id)
                REFERENCES categories (id) MATCH SIMPLE
                ON UPDATE NO ACTION ON DELETE NO ACTION,
        CONSTRAINT userCategories_user_id_fkey FOREIGN KEY (user_id)
                REFERENCES users (id) MATCH SIMPLE
                ON UPDATE NO ACTION ON DELETE NO ACTION
);

```

```
CREATE TABLE IF NOT EXISTS budgetCategories (
        budgets_id     INTEGER NOT NULL,
        category_id    INTEGER NOT NULL,
        amount REAL NOT NULL DEFAULT 0,
        CONSTRAINT budgetCategories_budgets_id_fkey FOREIGN KEY
            (budgets_id)
                REFERENCES budgets(id) MATCH SIMPLE
                ON UPDATE NO ACTION ON DELETE NO ACTION,
        CONSTRAINT budgetCategories_category_id_fkey FOREIGN KEY
            (category_id)
                REFERENCES categories (id) MATCH SIMPLE
                ON UPDATE NO ACTION ON DELETE NO ACTION
);

CREATE TABLE IF NOT EXISTS payers (
        user_id INTEGER NOT NULL,
        name   TEXT NOT NULL,
        CONSTRAINT payers_user_id_fkey FOREIGN KEY (user_id)
                REFERENCES users (id) MATCH SIMPLE
                ON UPDATE NO ACTION ON DELETE NO ACTION
);

CREATE TABLE IF NOT EXISTS expenses (
        id     serial PRIMARY KEY,
        description    TEXT NOT NULL,
        category       TEXT NOT NULL,
        expensedate    TEXT NOT NULL,
        amount REAL NOT NULL,
        payer  TEXT NOT NULL,
        submittime     TEXT NOT NULL,
        user_id INTEGER,
        CONSTRAINT budgets_user_id_fkey FOREIGN KEY (user_id)
                REFERENCES users (id) MATCH SIMPLE
                ON UPDATE NO ACTION ON DELETE NO ACTION
);

INSERT INTO categories(name) VALUES ('Groceries');
INSERT INTO categories(name) VALUES ('Housing');
INSERT INTO categories(name) VALUES ('Utilities');
INSERT INTO categories(name) VALUES ('Dining Out');
```

```
75 INSERT INTO categories(name) VALUES ('Shopping');
76 INSERT INTO categories(name) VALUES ('Travel');
77 INSERT INTO categories(name) VALUES ('Entertainment');
78 INSERT INTO categories(name) VALUES ('Other');
```