IT2002 The App Group Project Report (MoneyKakis) - Project Group 17

Kohn Henry Liam - A0269048U
Li Matthew - A0267853N
Muhammad Hud B Ayub - A0199727H
Rayden Teo Wei Xuan - A0268219W

1. Overview

The rise of digital transactions has led to a significant increase in shared spending, where one person pays and others pay them back later. However, this can become difficult in situations where there are many transactions, different currencies, and payment modes involved.

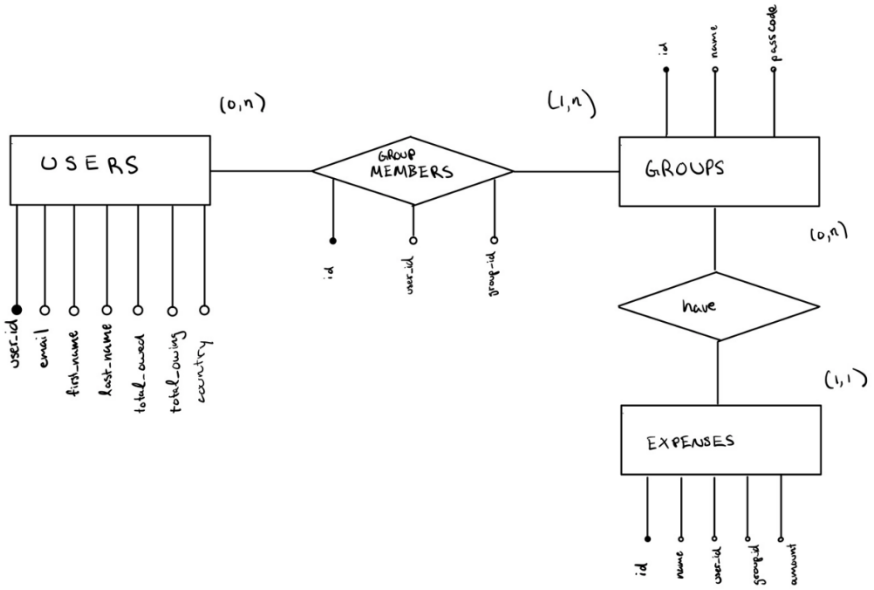
Recognizing this, our team drew inspiration from one of the leading spending-tracking apps for groups, *Splitwise*, and decided to emulate it for this course project. Our application, *MoneyKakis* enables users to join different payment groups and track how much is owed to them and vice versa, all through a user-friendly interface.

From a software engineering perspective, we took full advantage of the support for inheritance in HTML and the object-oriented nature of Python, which saved a lot of time in developing the frontend.

At a high level, the user is prompted to enter their login credentials. Depending on the credentials, they are taken to the admin page, dashboard page, or prompted to create a new account. Once the user has reached the dashboard page, they can navigate to the manage groups page, which contains all the necessary functionality.

2. Design

Figure 1: *Entity Relationship (ER) Diagram*



With reference to Figure 1, the app is comprised of 4 main tables: users, group_members, groups and expenses, three entities and two relations. As seen in the ER diagram, each table has a primary key that is a unique integer assigned to the record upon creation. This level of abstraction protects user identities while also facilitating easier implementation.

Users have the option to join a group, resulting in a (0,n) relationship, while a group must have users, resulting in a (1,n) relationship. Expenses require a group, resulting in a (1,1) relationship, but a group may not have any expenses, resulting in a (0,n) relationship.

Figure 2: *SQL Data Definition Language Code*

```
CREATE TABLE IF NOT EXISTS users (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    email TEXT NOT NULL UNIQUE,
    password text NOT NULL,
    country TEXT NOT NULL
);
CREATE TABLE IF NOT EXISTS groups (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    passcode TEXT NOT NULL
);
CREATE TABLE IF NOT EXISTS group_members (
    id SERIAL PRIMARY KEY,
    user_id INTEGER NOT NULL,
    group_id INTEGER NOT NULL,
    FOREIGN KEY (user_id) REFERENCES users(id),
    FOREIGN KEY (group_id) REFERENCES groups(id)
);
CREATE TABLE IF NOT EXISTS expenses (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    user_id INTEGER NOT NULL,
    group_id INTEGER,
    amount FLOAT NOT NULL CHECK (amount > 0),
    FOREIGN KEY (user_id) REFERENCES users(id),
    FOREIGN KEY (group_id) REFERENCES groups(id)
);
CREATE TABLE IF NOT EXISTS transactions (
    id SERIAL PRIMARY KEY,
    amount REAL NOT NULL,
    expense_id INTEGER NOT NULL,
    payer_id INTEGER NOT NULL,
    payee_id INTEGER NOT NULL,
    FOREIGN KEY (expense_id) REFERENCES expenses(id),
    FOREIGN KEY (payer_id) REFERENCES users(id),
    FOREIGN KEY (payee_id) REFERENCES users(id)
);
```

The SQL CREATE TABLE statements with constraints are given in Figure 2. It is important to note the serial primary key statements included next to each unique identifier in every table.

3. Dashboard/Homepage

After a successful login, the user will be directed to a home page where a brief user guide on how to use the website and application will be displayed. Next, the user can access the dashboard tab on the navigation bar to view the groups they belong to. The group names, along with their respective group ID, will be displayed with a hashtag to avoid confusion in cases where multiple groups share the same name, as the group ID is the primary key for the groups table. By clicking the "View Expenses" button, the user can access a list of all expenses within each group. To display all groups that the user belongs to, a simple SQL query is used as shown in Figure 3. The retrieved data (group_data) is then iterated through in the HTML code to list down the groups.

Figure 3: *Display All Groups*

```
@app.route("/home", methods=["GET", "POST"])
def home():
    if 'email' in session:
        statement = sqlalchemy.text("SELECT g.name, g.id \
                                    FROM group_members gm, groups g \
                                    WHERE gm.group_id = g.id \
                                    AND gm.user_id = :userid")
        params = {"userid": session["id"]}
        group_data = db.execute(statement, params).fetchall()
        return render_template("home.html", name=session["name"], group_data=group_data)
    return redirect(url_for("login"))
```

When the user clicks on "View Expenses" for a particular group, they will be directed to a page where all the outstanding expenses related to that group will be displayed in a tabular format, as shown in Figure 4. Any expenses that the user owes to the payer will be displayed in red, while any expenses that other members of the group owe to the user will be displayed in green. Please note that for this website, we have assumed an equal sharing of expenses.

Figure 4: *Expenses Page*

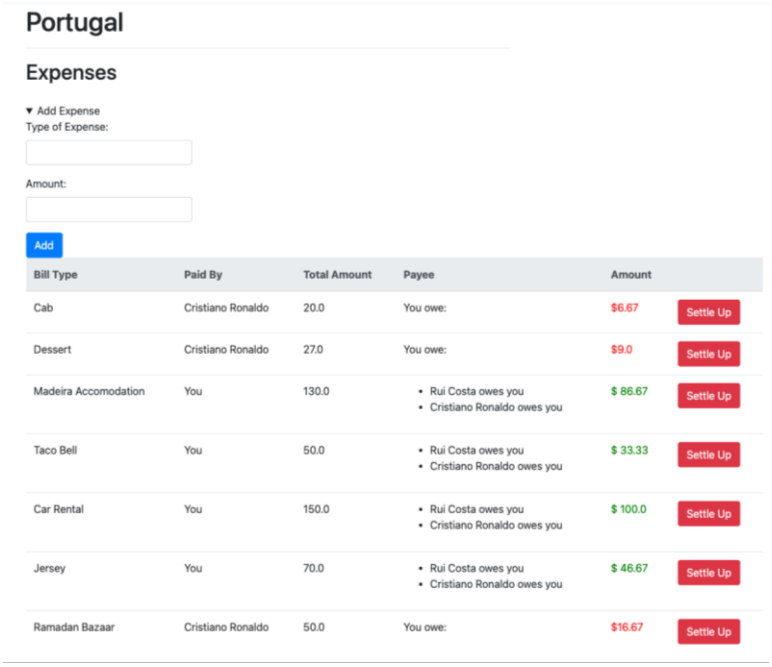
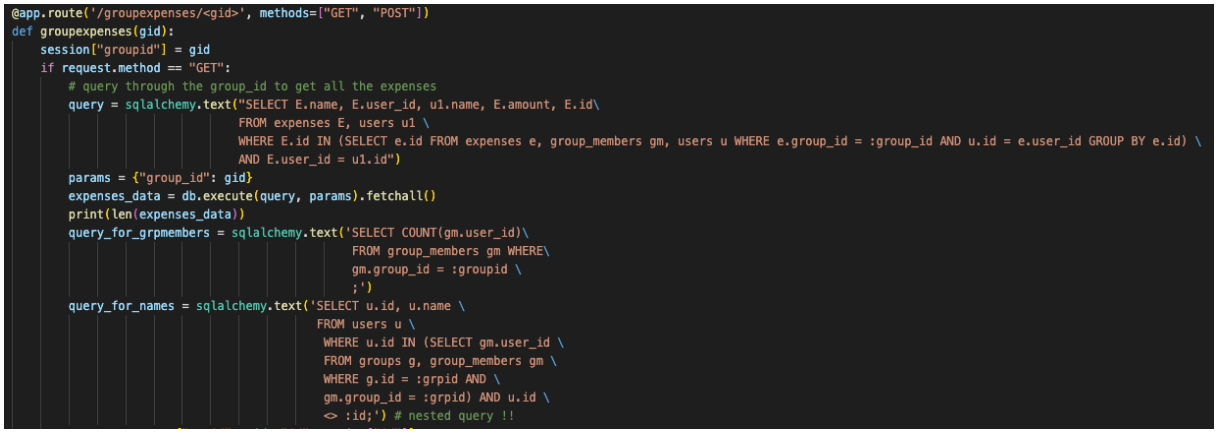


Figure 5: *Group Expenses Function*



```
@app.route('/groupexpenses/<gid>', methods=["GET", "POST"])
def groupexpenses(gid):
    session["groupid"] = gid
    if request.method == "GET":
        # query through the group_id to get all the expenses
        query = sqlalchemy.text("SELECT E.name, E.user_id, u1.name, E.amount, E.id\
                                FROM expenses E, users u1 \
                                WHERE E.id IN (SELECT e.id FROM expenses e, group_members gm, users u WHERE e.group_id = :group_id AND u.id = e.user_id GROUP BY e.id) \
                                AND E.user_id = u1.id")
        params = {"group_id": gid}
        expenses_data = db.execute(query, params).fetchall()
        print(len(expenses_data))
        query_for_grpmembers = sqlalchemy.text('SELECT COUNT(gm.user_id)\
                                               FROM group_members gm WHERE\
                                               gm.group_id = :groupid \
                                               ;')
        query_for_names = sqlalchemy.text('SELECT u.id, u.name \
                                          FROM users u \
                                           WHERE u.id IN (SELECT gm.user_id \
                                           FROM groups g, group_members gm \
                                           WHERE g.id = :grpid AND \
                                           gm.group_id = :grpid) AND u.id \
                                           <> :id;') # nested query !!
```

With reference to Figure 5, to display all outstanding expenses involving a group, we use two nested queries and one simple query. The first complex query fetches the expense type, payer, payer's name, and total amount of the expense. The simple query counts the total number of group members using the aggregate function COUNT to split the cost equally. The last nested query finds the names of the payees who are not the user and displays them in the "Payee" column for each expense.

Figure 6: *Settle Up Function*

```
@app.route('/settleup', methods=["POST"])
def settleup():
    expense_id = request.form.get('expenseid')
    print(expense_id)
    query_delete = sqlalchemy.text('DELETE FROM expenses e WHERE e.id = :eid;')
    params = {"eid":expense_id}
    db.execute(query_delete, params)
    db.commit()
    return redirect(url_for('groupexpenses', gid=session["groupid"]))
```

As mentioned earlier, the table also includes a "Settle Up" button to delete the expense entry once payment is made, like how *Splitwise* works. Additionally, the user can add a group expense using a dropdown that requires entering the expense label and total amount. We assume that the user will be the payer as they would typically have the receipt. We use a simple INSERT operation in a try/except structure to add the expense to the database, with a rollback to handle any exceptions or errors related to the entered amount. The code logic can be seen in Figure 6.

Figure 7: *Add Expenses Function*

```python
@app.route('/addexpenses', methods=["POST"])
def addexpense():
    type = request.form.get('name')
    g_id = session["groupid"]
    amount = request.form.get('amount')
    u_id = session["id"]
    try:
        insertion_query = sqlalchemy.text('INSERT INTO expenses(name, user_id, group_id, amount) VALUES (:type, :userid, :groupid, :amount);')
        params = {"type": type, "userid":u_id, "groupid":g_id, "amount":amount}
        db.execute(insertion_query,params)
        db.commit()
        flash('Expense added successfully', category='success')
        return redirect(url_for('groupexpenses', gid=g_id))
    except Exception:
        db.rollback()
        flash('Amount entered is invalid: make sure amount is greater than $0 and key in 2 decimal places (i.e 30.00)', category='error')
        return redirect(url_for('groupexpenses', gid=g_id))
```

Figure 8: *Trigger for Error Handling of Expense Amount*

```sql
CREATE OR REPLACE FUNCTION expenses_amount_check()
RETURNS TRIGGER
AS $$
BEGIN
    IF NEW.amount <= 0 THEN
        RAISE EXCEPTION 'Amount must be greater than zero';
    END IF;
    RETURN NEW;
END;
$$
LANGUAGE plpgsql;

CREATE TRIGGER expenses_amount_trigger
BEFORE INSERT ON expenses
FOR EACH ROW
EXECUTE FUNCTION expenses_amount_check();
```

Figure 7 displays the function for adding expenses, while Figure 8 shows the creation of a trigger with a function to perform a check on every amount inserted by the user.

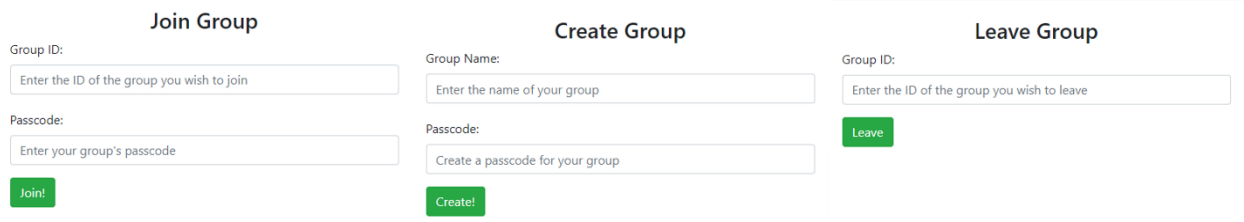
4. Manage Groups Page

The manage groups page can be accessed by users when they log in, through the navigation bar at the top of the screen. The page includes three forms that enable users to create a new group, join an existing group, or leave a group they are currently a member of.

To create a new group, the user needs to fill in the group name and a passcode of their choice. The passcode is required for other users to join the group. Once the user submits the form, the application automatically generates a unique group ID using the SERIAL data type in SQL, which becomes the group's identifier. The application inserts a new row into the group_members table to add the user as the creator of the group. The unique ID is needed for other users to join the group or leave it. Duplicate group names and passcodes are allowed, but the unique ID serves as the new group's identifier.

After successful group creation, the application displays a message containing the group ID and confirms that the creation was successful. The new group will appear on the user's dashboard.

Figure 9: *Join, Create, and Leave Group User Interface*



To join a new group, the user needs to provide the unique group ID and its passcode set by the creator of the group. The application first verifies if the group with the entered ID exists in the groups table, using the query SELECT * FROM groups g WHERE g.id = <group ID entered in the form>. If the group exists, the application checks if the entered passcode is correct. Then, the application verifies if the user has already joined the group by executing the query SELECT * FROM group_members gm WHERE gm.user_id = <ID of the currently logged in user> AND gm.group_id = <ID of the group to be joined>. Once all three checks pass, the user is added to the group using an INSERT INTO group_members statement. Although the checks could be combined into a single query, they are separated to provide specific error messages to the user. For instance, if the first two checks succeed but the third fails, the application notifies the user that he/she has already joined the group. As with group creation, the joined group becomes visible on the user's dashboard immediately after a successful joining.

To leave a group, a user can enter the ID of the group they want to leave and submit the "leave group" form. The application performs a series of checks before removing the user from the group. First, the application checks whether the user trying to leave the specified group is indeed a member of it by executing a SELECT * FROM group_members gm WHERE gm.group_id = <group ID entered> AND gm.user_id = <ID of user currently logged in> query. If this query returns a result, the application then checks to ensure that the group has no outstanding expenses by executing SELECT * FROM expenses e WHERE e.group_id = <group ID entered> query, which should return no results. A user can only leave a group once all expenses have been settled up since the amount owed by each user is a function of the number of people in the group. A user who leaves a group that still has outstanding expenses may be avoiding paying for those expenses. If this check is successful, the application removes the user from the group by executing DELETE FROM group_members gm WHERE gm.user_id = <ID of user currently logged in>. The two checks above could be combined into a single SQL query, but they have been separated for the application to provide specific error messages. If the leave group request is successful, the user receives a message indicating that they have successfully left the group. They can confirm this by navigating back to their dashboard, where the group they have left will no longer be visible. Users can rejoin groups at any time by using the join group function.

A try…except structure is used for all insert and delete queries in this section.

5. Admin Page

For administrative purposes, we have developed an admin page that can only be accessed through specific admin credentials. The admin section of the Flask application has five different routes related to administration tasks.

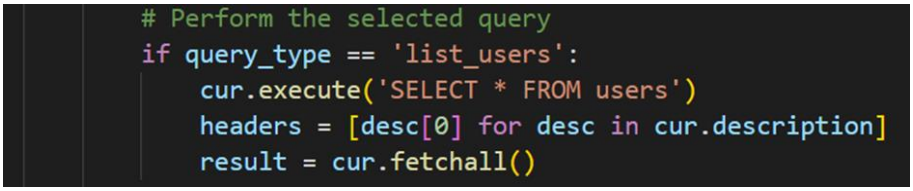
1. admin_home - Homepage for the administrative interface.
Through this page, administrators can perform two types of tasks: data analytics queries or adding/deleting groups and users.

When the function receives a POST request, it checks the value of the action parameter in the submitted form to determine whether the user has initiated a data analytics query or requested a user/group modification.

Figure 10: *Admin Data Analytics Functionality*

With reference to Figure 10, if the action parameter is Query, the function establishes a connection to a PostgreSQL database using the psycopg2 library and retrieves the selected data based on the specified query type. The available query types are list_users, list_groups, list_group_members, show_empty_groups, and show_big_spenders. The headers and results of the query are stored in the headers and result variables, respectively, and are passed to the adminHome.html template for rendering.

Figure 11: *Simple Query to List Users*



A simple query call can be Figure 11. For example, if the admin chooses to list all available users, the code calls 'SELECT * FROM users'.

Figure 12: *Nested Query to Show Empty Groups*

```
elif query_type == 'show_empty_groups':
    cur.execute('SELECT * \
                 FROM groups g \
                 WHERE NOT EXISTS ( \
                     SELECT * \
                     FROM group_members gm \
                     WHERE g.id = gm.group_id)')
    headers = [desc[0] for desc in cur.description]
    result = cur.fetchall()
```

On the other hand, some other data analytics calls require much more complicated SQL code. For instance, showing the empty groups requires using a nested query to select groups that do not have any users, as shown in Figure 12.

Figure 13: *Nested Aggregate Query to Show Big Spenders*

```
elif query_type == 'show_big_spenders':
    cur.execute("SELECT u.email, temp.average_spend \
                 FROM (SELECT e.user_id, AVG(e.amount) AS average_spend \
                     FROM expenses e \
                     GROUP BY e.user_id \
                     HAVING AVG(e.amount) >= 100) temp, users u \
                 WHERE temp.user_id = u.id")
    headers = [desc[0] for desc in cur.description]
    result = cur.fetchall()
```

We also experimented with some data analytics queries. For instance, the above query displays the email addresses and average amount spent of all users with an average spend greater than or equal to $100. This information might be relevant to a luxury brand that is looking to advertise their products, as users that have a high average spend might be more likely to buy luxury products.

However, the implementation of this query proved to be quite difficult. We initially tried to join the users table with the expenses table and execute an aggregate query with a HAVING clause; however, this did not allow us to display users' emails as this attribute did not appear in the GROUP BY or HAVING clauses. Instead, we needed to execute a nested aggregate query with a HAVING clause on the expenses table, return user ID and average spend, and then join this intermediate table with the users table in order to match user IDs to email addresses. The final query is shown in Figure 13.

Figure 14: *Add or Delete User/Group*

```
else:
    group_or_user = request.form.get('group_or_user')
    add_or_delete = request.form.get('add_or_delete')

    if (group_or_user=="group" and add_or_delete=="add"):
        return redirect(url_for('admin_group_add'))
    elif (group_or_user=="group" and add_or_delete=="delete"):
        return redirect(url_for('admin_group_delete'))
    elif (group_or_user=="user" and add_or_delete=="add"):
        return redirect(url_for('admin_user_add'))
    elif (group_or_user=="user" and add_or_delete=="delete"):
        return redirect(url_for('admin_user_delete'))
```

If the action parameter is not Query, the function determines whether the user requested to add or delete a user/group and redirects the user to the corresponding page, as shown in Figure 14.

2. admin_group_add - Handles the addition of a new group to the database.

Figure 15: *Admin Create New Group*

```
cur.execute("INSERT INTO groups (name, passcode) VALUES (%s, %s)", (name, passcode))
conn.commit()
```

If a POST request is received, the function retrieves the name and passcode values from the submitted form and uses them to create a new group in the groups table in the database.

3. admin_group_delete - Handles the deletion of an existing group from the database.

Figure 16: *Admin Delete Group*

```
cur.execute("DELETE FROM group_members WHERE group_id = %s", (id,))
cur.execute("DELETE FROM groups WHERE id = %s", (id,))
conn.commit()
```

If a POST request is received, the function retrieves the id value from the submitted form and deletes the corresponding records from the group_members and groups tables. Since the group being deleted is referenced in group_members as a FOREIGN KEY, the corresponding record from group_members must be deleted first, if any.

4. admin_user_add - Handles the addition of a user to a group in the database.

Figure 17: *Admin Add User to Group*

```
cur.execute("INSERT INTO group_members (user_id, group_id) VALUES (%s, %s)", (userID, groupID))
conn.commit()
```

If a POST request is received, the function retrieves the userID and groupID values from the submitted form and inserts them into the group_members table in the database, allowing the user with the given userID to join the group with the groupID.

5. admin_user_delete - Handles the removal of a user to a group in the database.

Figure 18: *Admin Remove User from Group*

```
cur.execute("DELETE FROM group_members WHERE user_id = %s AND group_id = %s", (userID,
groupID))
conn.commit()
```

If a POST request is received, the function retrieves the userID and groupID values from the submitted form and removes the user with the input userID from the group with the groupID in the group_members table in the database.

In summary, the administrative interface is a straightforward tool for managing user and group data in a PostgreSQL database, while also enabling the execution of some queries for data analytics purposes.

6. Conclusion

In retrospect, although we were able to achieve more than we initially expected, there are still areas that we would like to improve.

Initially, we spent a lot of time trying to get the virtual machine up and running and trying to understand the backend and frontend code that was provided to us, as none of us had a background in web development. However, through enough trial and error, as well as online readings, we were able to get the web app running.

Moving forward, we hope to implement functionality that allows users to collate payments that they are owed from the same person into their desired home currency. Currently, *MoneyKakis* shows transactions between two users as multiple unique transactions on the dashboard. We would like to sum these expenses for users so that they do not have to manually go through every transaction between the two to determine how much should be transferred. Regarding differing currencies, we could use an API to get the exchange rate at the time of the transaction and update all currencies to the user's desired currency.

While there is still much work to be done, we believe that there is great potential for this app.