Architecture Design of

"Expenditure distributor Management"

Author: Aakash Bisen

Documentation version: 1.8

Last version updated: 23/03/2023

Document Version Control

Date Issued	Version	Description	Author
08-02-2022	1.1	Created First Initial Draft of Architecture drawing	Aakash Bisen
25-03-2022	1.2	Added Workflow chart	Aakash Bisen
11-04-2022	1.3	Added Exception Scenarios Overall, Constraints	Aakash Bisen
08-06-2022	1.4	Added user I/O flowchart	Aakash Bisen
01-08-2022	1.5	Updated Workflow chart & I/O flowchart	Aakash Bisen
08-11-2022	1.6	Restructure and reformatting	Aakash Bisen
28-01-2023	1.7	Restructuring & Alignment	Aakash Bisen
23-03-2023	1.8	Final Architecture Design completed	Aakash Bisen

Contents

Abstract

Expense tracking is utmost important these days especially in creating your financial budget & in investment strategies. Expense tracking helps you take control of your finances and stay on your budget. Keeping a daily record of your expenses by tracking receipts, invoices, and other outgoing expenses improves your financial health. Tracking expenses can also help you stay on top of your cash flow and prepare you for tax season. This application distributes the entire expenditure among the participants and track who is investing where. Each user can see the statistics for their past expenses.

1.Introduction

1.1 Why this Architecture Design Document?

The goal of Architecture Design (AD) or a low-level design document is to give the internal design of the actual program code . Architecture Design describes the class diagrams with the methods and relation between classes and program specification. It describes the modules so that the programmer can directly code the program from the document.

1.2 Scope

Architecture Design(AD) is a component-level design process that follows a step-by-step refinement process. This process can be used for designing data structures, required software, architecture, source code, and ultimately, performance algorithms. Overall, the data organization may be defined during requirement analysis and then refined during data design work & the complete workflow.

1.3 Constraints

We will consider expenses spent by individual users to get it approved by other members of a group. This solution system is also user friendly, as automated as possible and users should not be required to know any of the workings.

1.4 Risks

Nevertheless no risk is involved but suppose if in future it gets deployed on real platform for public uses, the personal data like user's expenses on his/her expenditure will be recorded on user's consent (when they agree on the 'Terms and Conditions') to access their accounts & transactions.

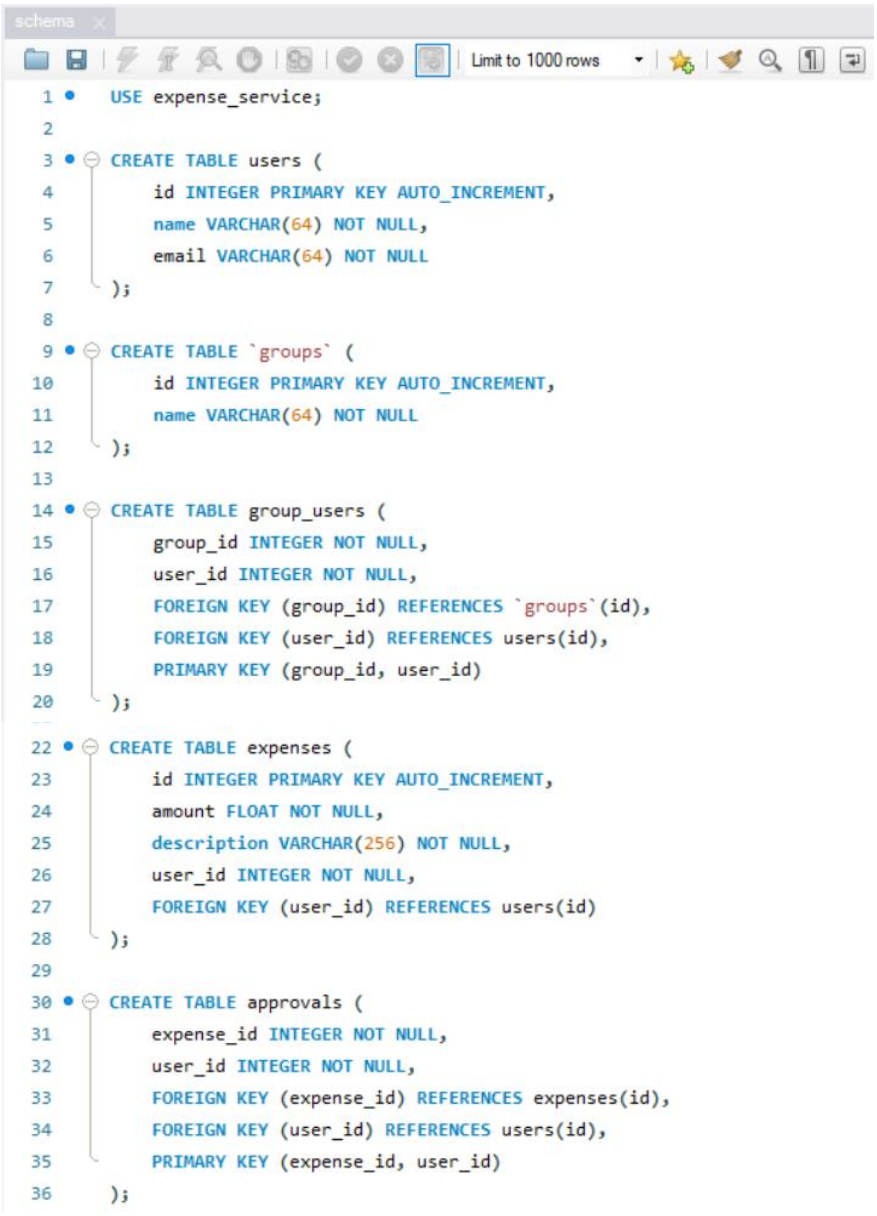
1.5 Out of Scope

This is a pure development project not prediction model. So users won't be able to forecast his/her/their future expenditure using past data of statistics of their previous expenses used as a metric to set realistic and viable budgeting goals.

2.Technical specifications

2.1 MySQL Schema

The following schema has been created for the requirements:



```
schema
Limit to 1000 rows
1  USE expense_service;
2
3  CREATE TABLE users (
4      id INTEGER PRIMARY KEY AUTO_INCREMENT,
5      name VARCHAR(64) NOT NULL,
6      email VARCHAR(64) NOT NULL
7  );
8
9  CREATE TABLE `groups` (
10     id INTEGER PRIMARY KEY AUTO_INCREMENT,
11     name VARCHAR(64) NOT NULL
12 );
13
14 CREATE TABLE group_users (
15     group_id INTEGER NOT NULL,
16     user_id INTEGER NOT NULL,
17     FOREIGN KEY (group_id) REFERENCES `groups`(id),
18     FOREIGN KEY (user_id) REFERENCES users(id),
19     PRIMARY KEY (group_id, user_id)
20 );
22 CREATE TABLE expenses (
23     id INTEGER PRIMARY KEY AUTO_INCREMENT,
24     amount FLOAT NOT NULL,
25     description VARCHAR(256) NOT NULL,
26     user_id INTEGER NOT NULL,
27     FOREIGN KEY (user_id) REFERENCES users(id)
28 );
29
30 CREATE TABLE approvals (
31     expense_id INTEGER NOT NULL,
32     user_id INTEGER NOT NULL,
33     FOREIGN KEY (expense_id) REFERENCES expenses(id),
34     FOREIGN KEY (user_id) REFERENCES users(id),
35     PRIMARY KEY (expense_id, user_id)
36 );
```

2.2 Logging

The application code has implemented log of every event so that the user will know what process is running internally.

- The System identifies at what step logging required
- The System logs each and every system flow.
- System doesn't hang/lag even after using so many loggings.
- Logging is important so that we can easily debug issues if encountered.

2.3 Database

System stores every request into the database(mentioned in 2.1 above) in such a way that it is easy to retrain the model.

- The system stores every expense/transaction spent by user in schema
- The User gives required information(if needed).
- The system stores each and every data given by the user or received on request to the database.

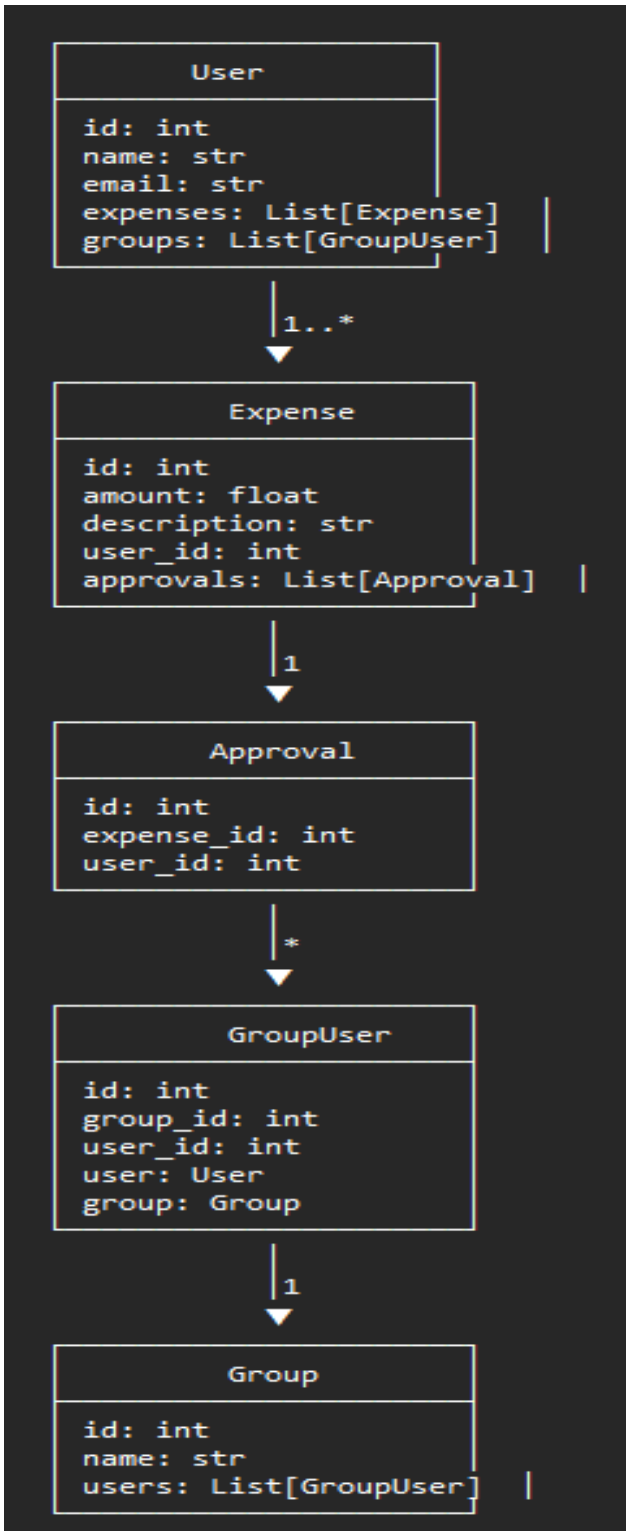
3. Deployment

4.Technology stack

- PyCharm is used as IDE.
- Python Flask is used for backend development.
- Flask-Testing to provide unit testing utilities for Flask.
- unittest2 module to provide a rich set of tools for constructing and running tests.
- AWS is used for deployment of the model.
- SQLAlchemy is used to retrieve, insert, delete, and update the database.
- GitHub is used as version control system.
- The complete solution is exposed as REST API.

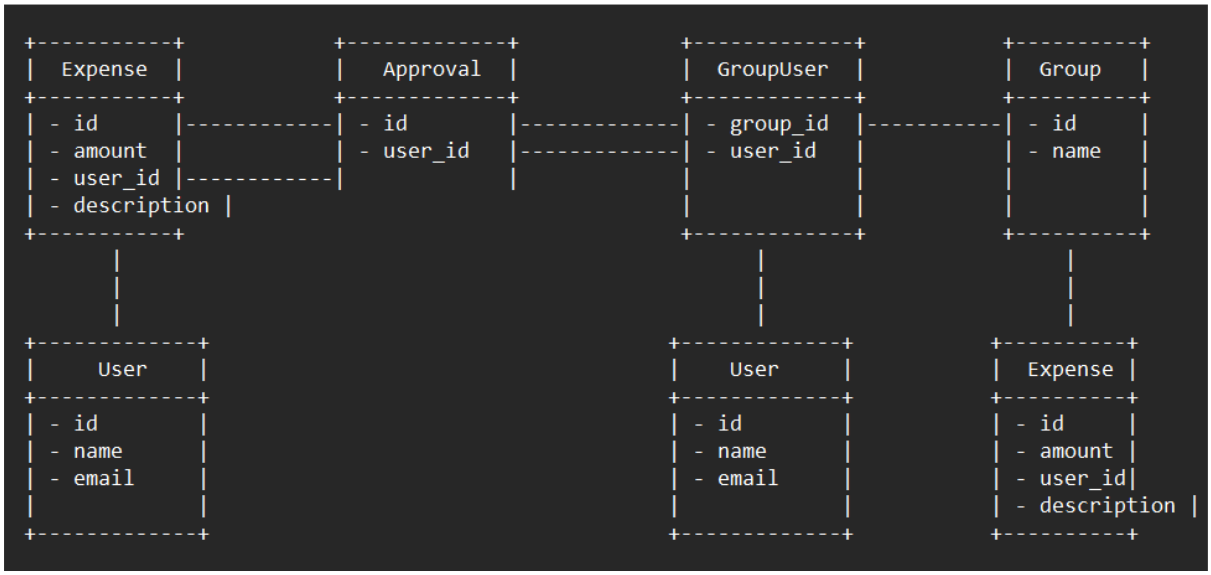
5.Proposed Solution

Low Level Design:



In this diagram, arrows represent the relationships between entities. For example, a User object has a one-to-many relationship with Expense objects, as represented by the arrow pointing from User to Expense with a "1..*" label. Each Expense object has exactly one User object associated with it, as represented by the arrow pointing from Expense to User with a "1" label. Similarly, a Group object has a one-to-many relationship with GroupUser objects, and each GroupUser object has exactly one User and one Group associated with it.

6. User I/O workflow



In this diagram, the boxes represent the classes/entities, and the arrows represent the relationships between them. Here's a brief description of each entity:

• **User:** represents a user of the expense service. Each user has a unique id, a name, and an email.

• **Expense:** represents an expense that a user has incurred. Each expense has a unique id, an amount, a description, and a user_id field that references the User that incurred the expense.

• **Approval:** represents an approval for an expense by a user. Each approval has a unique id, an expense_id field that references the Expense being approved, and a user_id field that references the User giving the approval.

• **GroupUser:** represents the membership of a user in a group. Each GroupUser has a group_id field that references the Group that the user is a member of, and a user_id field that references the User that is a member of the group.

• **Group:** represents a group of users who share expenses. Each group has a unique id, and a name.

(**Note:** The relationships between the entities are represented by the arrows in the diagram. For example, each Expense is associated with a User, and this is represented by the arrow pointing from Expense to User. Similarly, each Approval is associated with an Expense and a User, and this is represented by the arrows pointing from Approval to Expense and User, respectively.)

7. Error Handling

Errors are encountered with exceptions and explanation is displayed as to what went wrong. An error will be defined as anything that falls outside the normal and intended usage.

8.Test cases

Getting started:
* Install the required packages by running `pip install -r requirements.txt`.
* Set the Flask app environment variable by running `export FLASK_APP=app.py` (on Linux/Mac) or `set FLASK_APP=app.py` (on Windows).
* Start the Flask app by running the `app.py` file.
API Endpoints

Create User
`POST /users`
Body:
- `name` (string): The name of the user.
- `email` (string): The email of the user.
Response:
- `200`: The user was successfully created. Returns the ID of the newly created user.
- `400`: The request was invalid.

Example:
POST /users Content-Type: application/json
{ “name”: “John Doe”, “email”: “john.doe@example.com” }
200 OK Content-Type: application/json
{ “id”: 1 }

Create Group
`POST /groups`
Body:
- `name` (string): The name of the group.
- `user_ids` (array of integers): The IDs of the users to add to the group.
Response:
- `200`: The group was successfully created. Returns the ID of the newly created group.
- `400`: The request was invalid.

Example:
POST /groups Content-Type: application/json
{ “name”: “My Group”, “user_ids”: [1, 2, 3] }
200 OK Content-Type: application/json
{ “id”: 1 }

Add Expense
`POST /expenses`
Body:
- `amount` (float): The amount of the expense.
- `description` (string): The description of the expense.
- `user_id` (integer): The ID of the user who added the expense.
Response:
- `200`: The expense was successfully added. Returns the ID of the newly added expense.
- `400`: The request was invalid.

Example:
POST /expenses Content-Type: application/json
{ “amount”: 10.5, “description”: “Lunch”, “user_id”: 1 }
200 OK Content-Type: application/json
{ “id”: 1 }

Approve Expense
`POST /approvals`
Body:
- `expense_id` (integer): The ID of the expense to approve.
- `user_id` (integer): The ID of the user who approved the expense.
Response:
- `200`: The expense was successfully approved.
- `400`: The request was invalid.

Example:
POST /approvals Content-Type: application/json
{ “expense_id”: 1, “user_id”: 2 }
200 OK Content-Type: application/json

Distribute Expenses
`POST /groups/{group_id}/distribute`
Path Parameters:
- `group_id` (integer): The ID of the group to distribute expenses for.
Response:
- `200`: The expenses were successfully distributed.
- `400`: The request was invalid.

Example:
POST /groups/1/distribute Content-Type: application/json
200 OK Content-Type: application/json

View Statistics
`GET /users/{user_id}/statistics`
Path Parameters:
- `user_id` (integer): The ID of the user to view statistics for.
Response:
- `200`: The statistics were successfully retrieved. Returns the total amount, average amount, and number of expenses for the user.
- `400`: The request was invalid.

Example:
GET /users/1/statistics Content-Type: application/json
200 OK Content-Type: application/json
{ “total_amount”: 10.5, “average_amount”: 5.25, “num_expenses”: 2 }

9. Conclusion

The application of distribution management has been successfully created. It'll distribute expenditure among participants and also allows users to create a group, track their expenses, get them approved by other group members, distribute expenses fairly, make payments to each other, and view past expenses statistics.