DBMS Project – Cover Page

Title: Personal Expense and Finance Tracker

Team Details

- **Sujoy Sen – *PES2UG23CS621***
- **Tahir Shafiq – *PES2UG23CS639***

Course: Database Management Systems – UE23CS351A

Guided By: Gamini Joshi

Institution: PES University

Short Abstract

This project presents a **database-driven Expense Tracker system** designed to manage users' financial activities such as incomes, expenses, budgets, and saving goals. The backend is built using **MySQL**, with well-structured tables, integrity constraints, foreign keys with cascading deletes, and business rules implemented through **triggers, stored procedures, and SQL functions**. The system includes validation triggers for transaction–category consistency, a goal-updating trigger that automatically applies income to active goals, and a stored procedure for securely adding new transactions. A balance-calculation function is also implemented to compute a user's net financial position.

A lightweight web interface is developed using **Flask (Python)** to interact with the database. The app allows users to add transactions via the stored procedure, view all tables, and demonstrate cascading deletes when a user is removed. The project integrates core DBMS concepts including ER-to-relational mapping, DDL and DML operations, constraints, joins, aggregates, triggers, functions, and procedures. Overall, the system illustrates how database logic and application logic work together to maintain data accuracy, enforce rules, and support real-world financial tracking in an efficient and consistent manner.

User Requirement Specification (URS)

1. Introduction

The Expense Tracker web application allows users to view database tables, add transactions through a stored procedure, and delete users with cascading effects. The system acts as a front-end interface for interacting with a MySQL database containing users, transactions, budgets, and goals.

---

2. Purpose

The purpose of this application is to provide a simple web interface that demonstrates:

- Calling MySQL stored procedures
- Executing triggers (indirectly through DB operations)
- Performing cascading deletes
- Viewing data from different tables

The system is a demo to showcase DBMS concepts using a minimal Flask front-end.

---

3. Scope

The application supports only the following operations

- Add a transaction using a stored procedure sp_add_transaction
- Display tables: users, transactions, goals, budgets
- Delete a user (which triggers ON DELETE CASCADE in MySQL)
- Simple UI using HTML forms to perform these actions

---

4. System Overview

The system consists of:

- A Flask web server (app.py)
- MySQL database connection using mysql.connector
- Inline HTML templates rendered using render_template_string

It has three primary routes:

- GET / → View tables
- POST /add_transaction → Add transaction (calls stored procedure)
- POST /delete_user/<id> → Delete user

---

5. Functional Requirements

5.1 View Data Requirements

Requirement ID	Description
FR-01	The system must allow users to view selected database tables: transactions, goals, budgets, users.
FR-02	On page load, the system must display *transactions* and *users* by default.
FR-03	Users must be able to select which tables to show using checkboxes.

5.2 Add Transaction Requirements

Requirement ID	Description
FR-04	The system must provide a form to add a transaction (user_id, category_id, amount, type, date).

Requirement ID	Description
FR-05	The system must call the stored procedure sp_add_transaction when a transaction is submitted.
FR-06	The system must commit the transaction to the database and display a flash message.
FR-07	The system must retrieve and display the last inserted transaction ID returned by the procedure.

5.3 Delete User Requirements

Requirement ID	Description
FR-08	The system must allow deleting users through a delete button.
FR-09	The system must call DELETE FROM users WHERE user_id = ?.
FR-10	The system must rely on MySQL's ON DELETE CASCADE to delete related data (transactions, goals, budgets).
FR-11	The system must show a success or error flash message after deletion.

5.4 Error Handling Requirements

Requirement ID	Description
FR-12	The system must catch MySQL errors and show them using flash messages.
FR-13	The system must close database connections after each operation (try-finally).

5.5 UI Requirements

Requirement ID	Description
FR-14	The system must use a simple HTML interface embedded inside Python as a string.
FR-15	The UI must display tables in HTML <table> format with headings.
FR-16	The UI must provide a form with text inputs and dropdown for adding transactions.
FR-17	The system must show flash messages at the bottom of the page.

---

6. Non-Functional Requirements

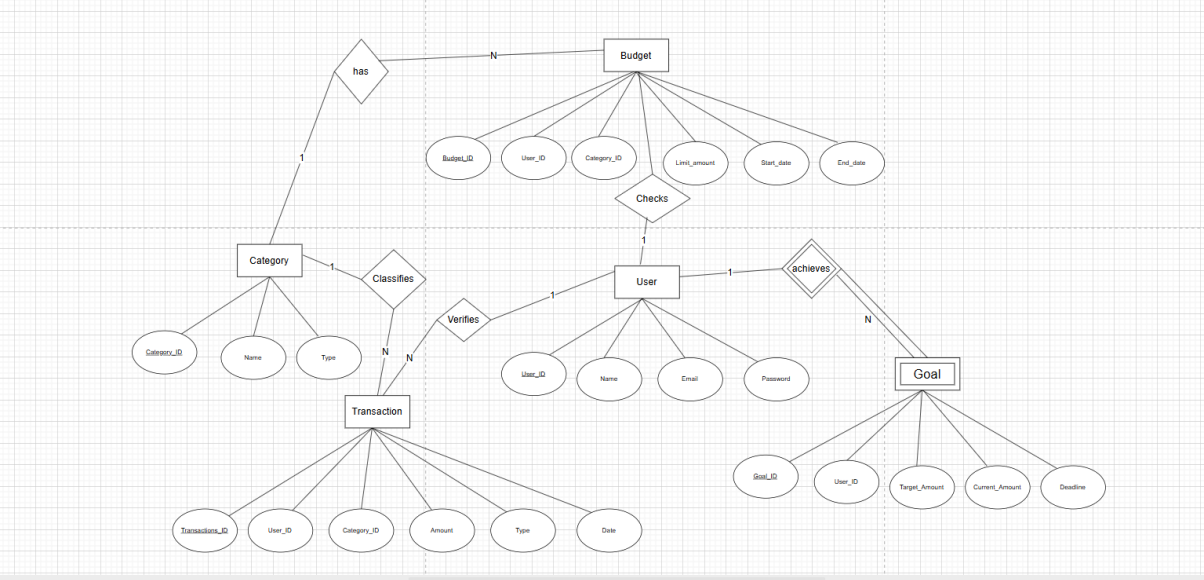
NFR ID	Requirement
NFR-01	The system must connect to MySQL using credentials provided in code.
NFR-02	The system must follow a synchronous request-response model.
NFR-03	The interface must load within a browser without additional CSS/JS frameworks.
NFR-04	The system must run locally using Flask's development server.

List of Software / Tools / Programming Languages Used

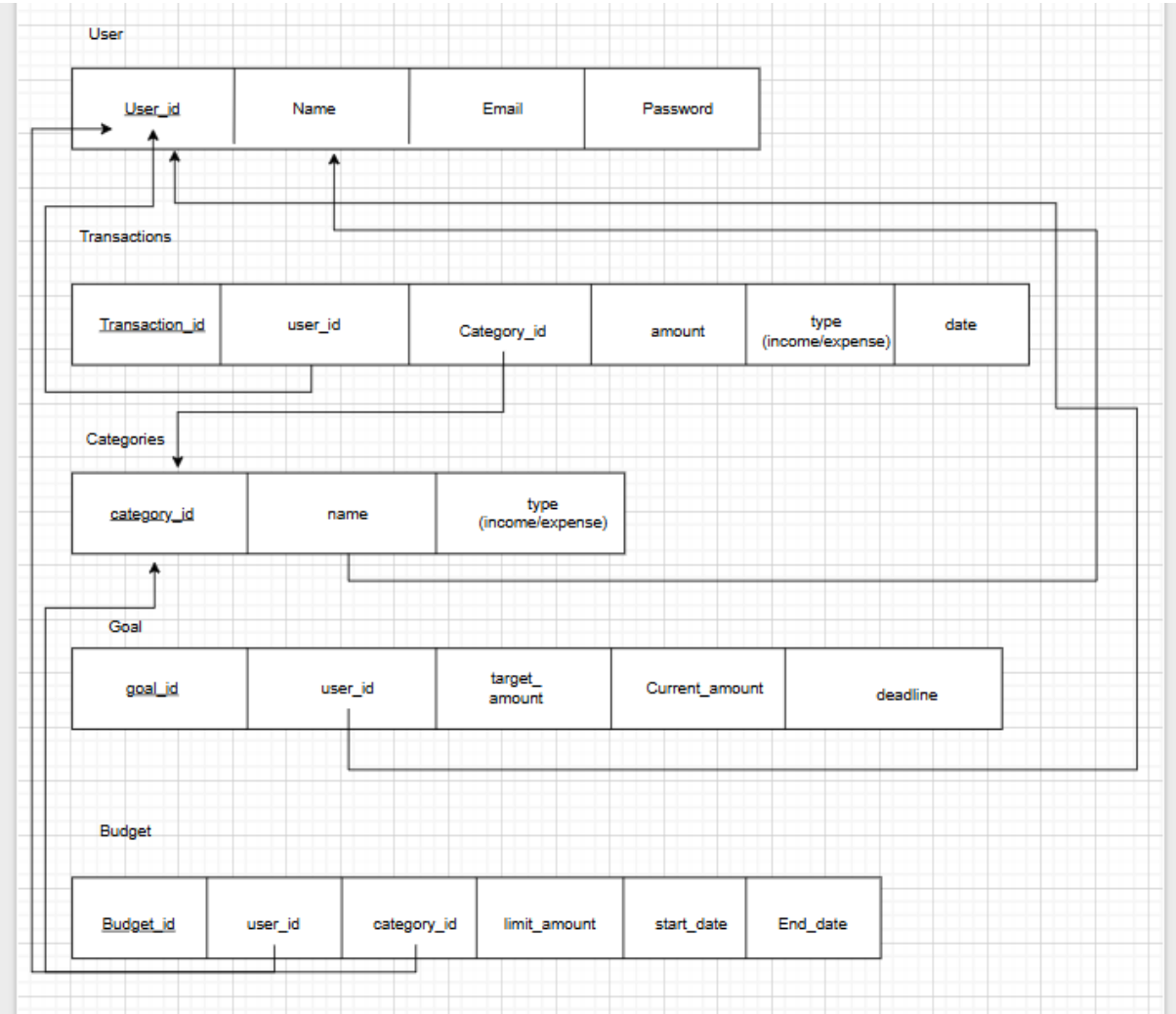
- **Python 3.x** – For backend logic and web server
- **Flask Framework** – To create the web application and routes
- **MySQL Database** – To store all user, transaction, goal, and budget data
- **mysql-connector-python** – Python library for connecting Flask with MySQL

- **HTML & CSS (inline)** – For the simple user interface rendered using Flask
- **MySQL Workbench / Command Line** – For executing SQL queries, procedures, and triggers
- **Browser (Chrome/Firefox)** – To run and view the web interface

ER Diagram:



Relational Schema:



DDL Commands

```
CREATE DATABASE expense_tracker;
USE expense_tracker;

CREATE TABLE users (
  user_id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  email VARCHAR(255) NOT NULL UNIQUE,
  password VARCHAR(255) NOT NULL
);


CREATE TABLE categories (
  category_id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  type ENUM('income','expense') NOT NULL
);


CREATE TABLE transactions (
  transaction_id INT AUTO_INCREMENT PRIMARY KEY,
  user_id INT NOT NULL,
  category_id INT NOT NULL,
  amount DECIMAL(12,2) NOT NULL CHECK (amount > 0),
  type ENUM('income','expense') NOT NULL,
  date DATE NOT NULL,
  FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE,
  FOREIGN KEY (category_id) REFERENCES categories(category_id)
);
```

```
DELIMITER $$

CREATE TRIGGER check_transaction_type
BEFORE INSERT ON transactions
FOR EACH ROW
BEGIN
  DECLARE cat_type ENUM('income','expense');
  SELECT type INTO cat_type FROM categories WHERE category_id = NEW.category_id;
  IF cat_type <> NEW.type THEN
    SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = 'Transaction type does not match category type';
  END IF;
END$$

DELIMITER ;



CREATE TABLE goals (
  goal_id INT AUTO_INCREMENT PRIMARY KEY,
  user_id INT NOT NULL,
  target_amount DECIMAL(12,2) NOT NULL CHECK (target_amount > 0),
  current_amount DECIMAL(12,2) NOT NULL CHECK (current_amount >= 0),
  deadline DATE,
  FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE,
  CHECK (current_amount <= target_amount)
);
```

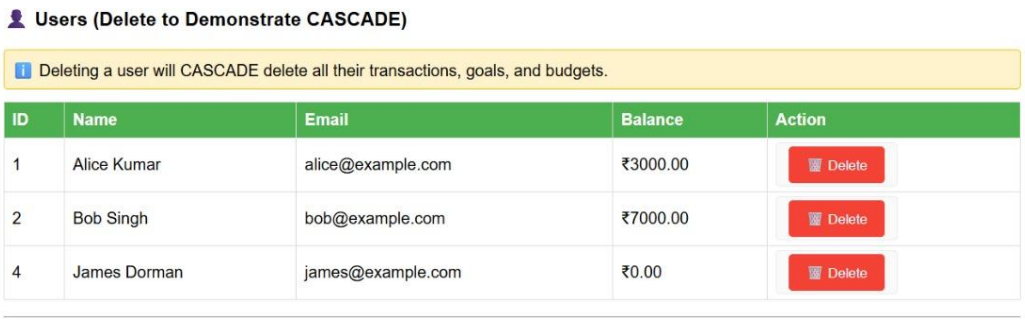
```
CREATE TABLE budgets (
  budget_id INT AUTO_INCREMENT PRIMARY KEY,
  user_id INT NOT NULL,
  category_id INT NOT NULL,
  limit_amount DECIMAL(12,2) NOT NULL CHECK (limit_amount > 0),
  start_date DATE NOT NULL,
  end_date DATE NOT NULL,
  FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE,
  FOREIGN KEY (category_id) REFERENCES categories(category_id),
  CHECK (start_date <= end_date)
);



INSERT INTO users (name, email, password) VALUES
('Alice Kumar', 'alice@example.com', '1234'),
('Bob Singh', 'bob@example.com', 'abcd');

INSERT INTO categories (name, type) VALUES
('Salary', 'income'),
('Gift', 'income'),
('Groceries', 'expense'),
('Rent', 'expense'),
('Entertainment', 'expense');
INSERT INTO transactions (user_id, category_id, amount, type, date) VALUES
(1, 1, 50000.00, 'income', '2025-08-01'),
(1, 3, 1500.00, 'expense', '2025-08-03'),
(2, 4, 8000.00, 'expense', '2025-08-05'),(2, 2, 2000.00, 'income', '2025-08-10'),
(1, 5, 500.00, 'expense', '2025-08-15');
```

```
INSERT INTO goals (user_id, target_amount, current_amount, deadline) VALUES
(1, 20000.00, 5000.00, '2026-06-30'),
(2, 50000.00, 12000.00, '2026-12-31');
INSERT INTO budgets (user_id, category_id, limit_amount, start_date, end_date) VALUES
(1, 3, 6000.00, '2025-08-01', '2025-08-31'),
(2, 5, 2000.00, '2025-08-01', '2025-08-31');
SELECT * FROM users;
SELECT * FROM categories;
SELECT * FROM transactions;
SELECT * FROM goals;
SELECT * FROM budgets;
```

CRUD Operation Screenshots

User Added:

Showing that user is added:



Updated the bank balance of James Dorman(new user):

Users (Delete to Demonstrate CASCADE)

Deleting a user will CASCADE delete all their transactions, goals, and budgets.

ID	Name	Email	Balance	Action
1	Alice Kumar	alice@example.com	₹3000.00	Delete
2	Bob Singh	bob@example.com	₹7000.00	Delete
3	James Dorman	james@example.com	₹1500.00	Delete

Demo Application | Stored Procedures ✓ | Triggers ✓ | CASCADE Delete ✓

Reduced 1200 from 1500 so now its 300

Add Transaction (Stored Procedure)

User: James Dorman (ID: 3) | Category: Rent (expense) | Amount: 1200 | Type: Expense | Date: 18-11-2025

Add Transaction

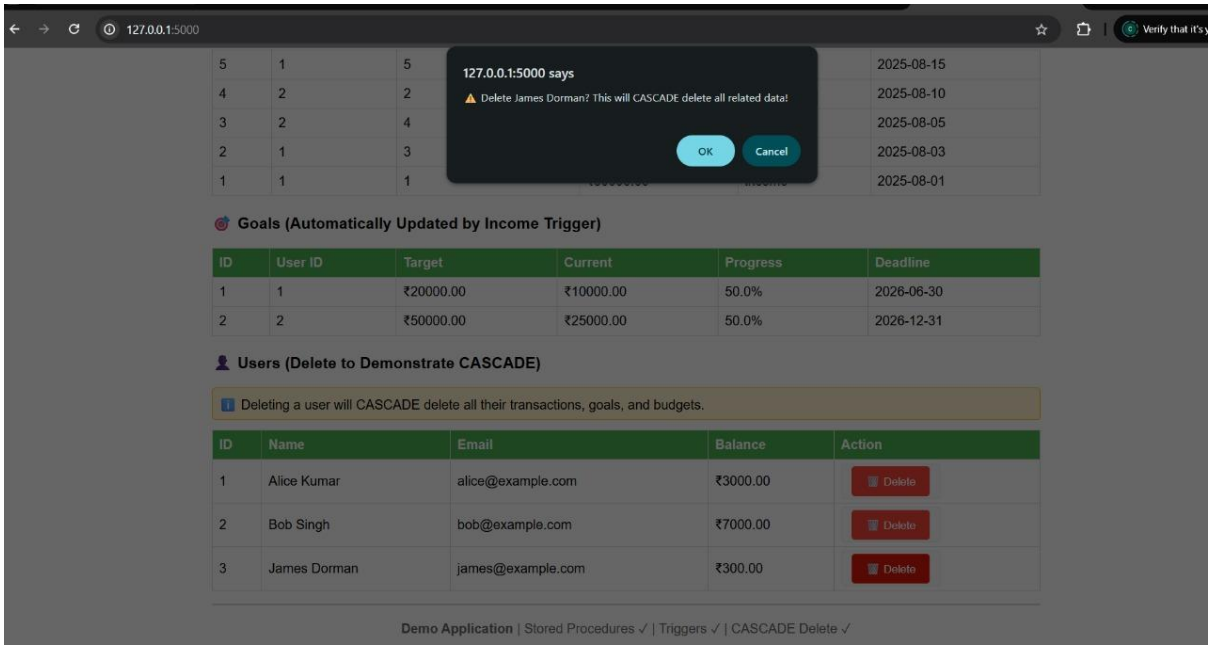
Users (Delete to Demonstrate CASCADE)

Deleting a user will CASCADE delete all their transactions, goals, and budgets.

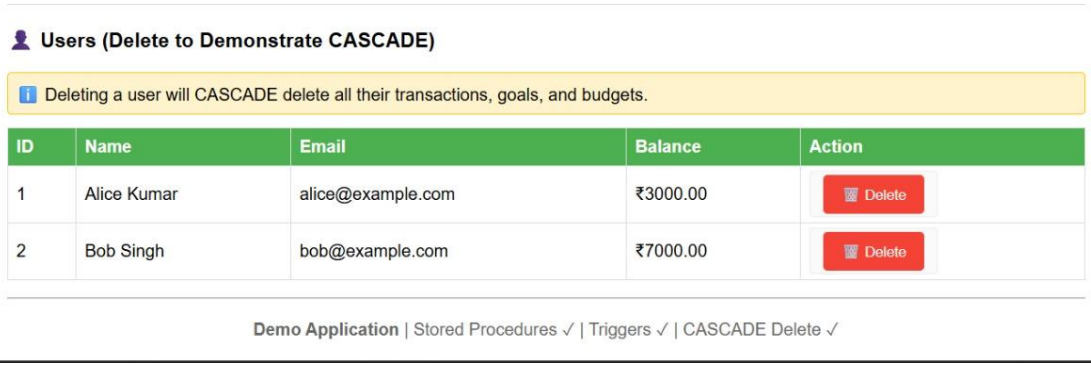
ID	Name	Email	Balance	Action
1	Alice Kumar	alice@example.com	₹3000.00	Delete
2	Bob Singh	bob@example.com	₹7000.00	Delete
3	James Dorman	james@example.com	₹300.00	Delete

Demo Application | Stored Procedures ✓ | Triggers ✓ | CASCADE Delete ✓

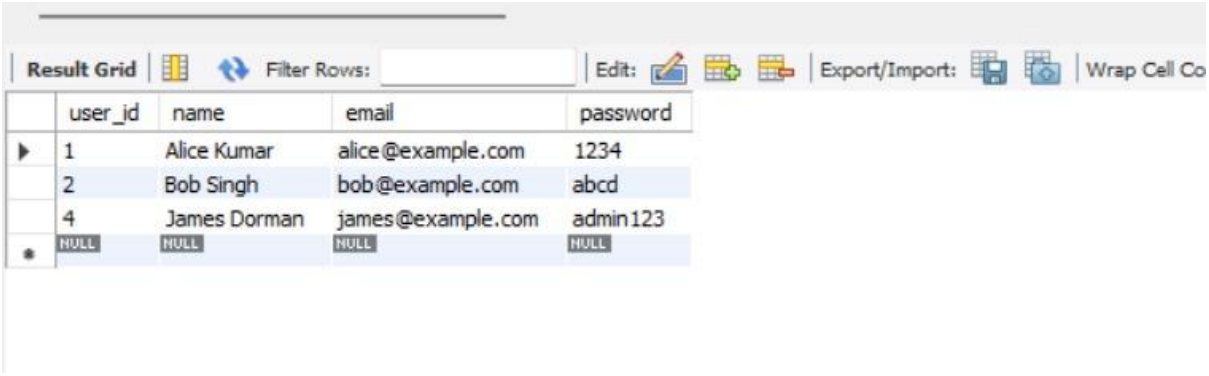
Delete the user james dorman:



User is deleted:



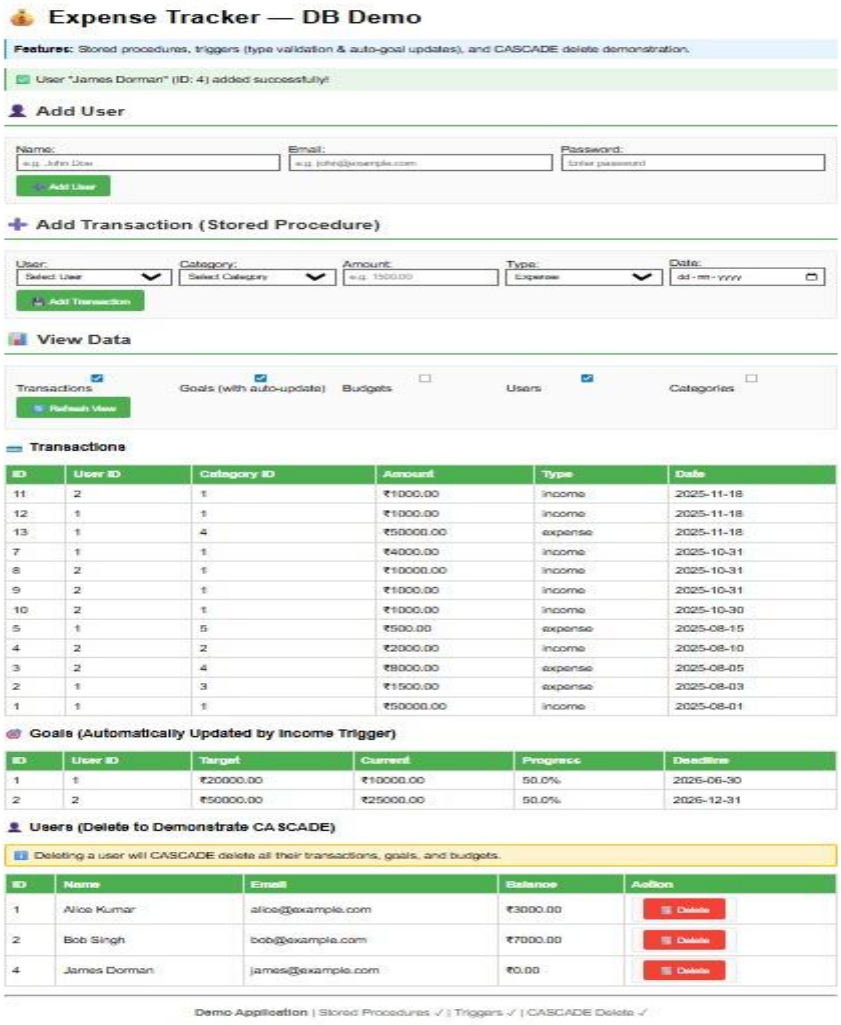
SQL Database showing data



List of functionalities/features of the application

1. Home page — Default view (Transactions + Users)

- **Feature: Display transactions and users by default.**
- **Action: Open http://localhost:5000 (no query parameters).**

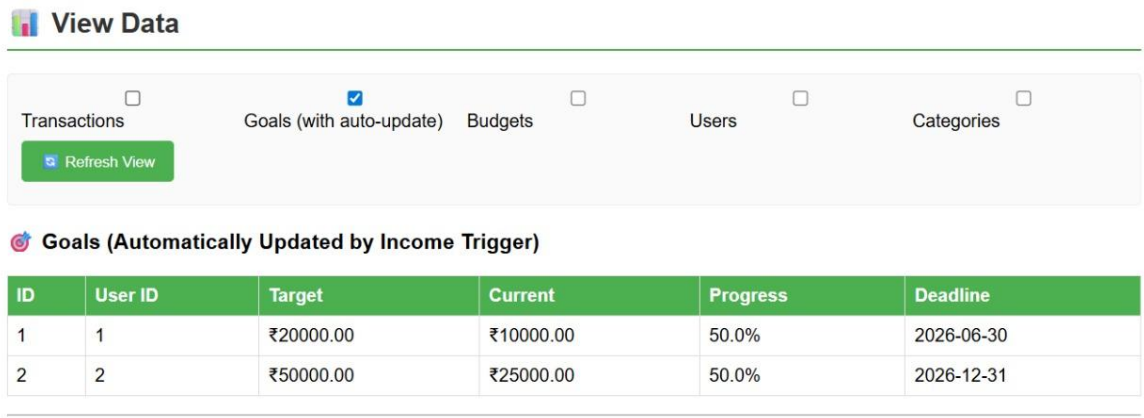


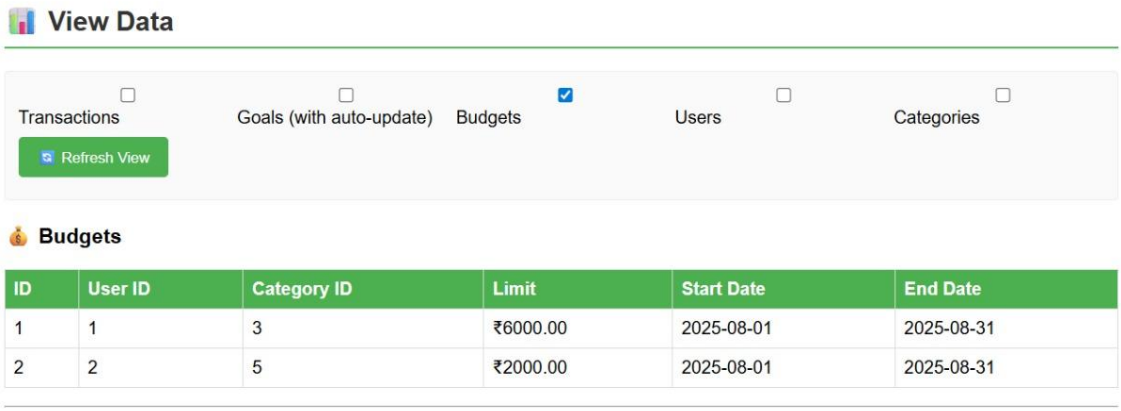
- **Why: Proves basic UI rendering and default tables loaded from DB.**

---

2. Toggle displayed tables (show Goals & Budgets)

- **Feature: Choose which tables to display via checkboxes.**
- **Action: On the "List Data" form, check Goals and Budgets, uncheck Transactions if desired, click Refresh.**





- **Why: Demonstrates that the UI can filter which tables are fetched/displayed.**

---

3. Add Transaction — Form filled (before submit)

- **Feature: Add a transaction using the stored procedure (sp_add_transaction).**
- **Action: In "Add Transaction" form fill:**
  - **user_id: (e.g., 1)**
  - **category_id: (e.g., choose 1 for Salary or an expense category as appropriate)**
  - **amount: 1500**
  - **type: income or expense (choose to match category)**
  - **date: YYYY-MM-DD (e.g., 2025-11-18)**
  - **Click Add Transaction.**
- **Screenshot to take (form filled):**

- **Why: Shows the input that will call the stored procedure and the UI fields available.**

---

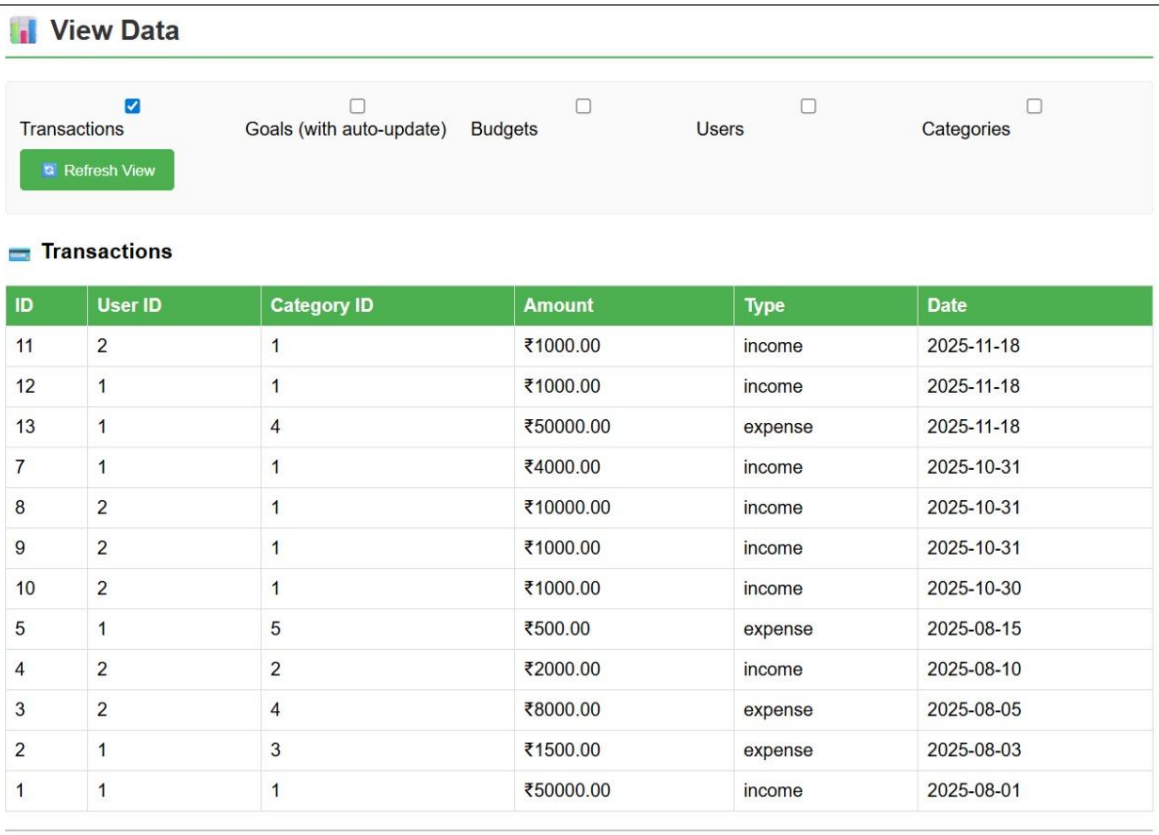
4. Add Transaction — Success flash message (after submit)

- **Feature: Stored procedure is called; the app commits and flashes a message with returned id (if procedure SELECTs LAST_INSERT_ID()).**
- **Action: Submit the form from step 3.**
- **Screenshot to take:**
  - **Filename: 04_add_transaction_success.png**
  - **Caption: *Flash message confirming transaction added and showing returned transaction id.***
- **Why: Proves stored procedure invocation and that application reads stored_results().**

---

5. Transactions table — showing newly inserted row

- **Feature: Transactions table updated with newly added transaction.**
- **Action: Make sure the Transactions checkbox is selected and Refresh (or revisit /). Locate inserted row.**
- **Screenshot to take:**



ID	User ID	Category ID	Amount	Type	Date
11	2	1	₹1000.00	income	2025-11-18
12	1	1	₹1000.00	income	2025-11-18
13	1	4	₹50000.00	expense	2025-11-18
7	1	1	₹4000.00	income	2025-10-31
8	2	1	₹10000.00	income	2025-10-31
9	2	1	₹1000.00	income	2025-10-31
10	2	1	₹1000.00	income	2025-10-30
5	1	5	₹500.00	expense	2025-08-15
4	2	2	₹2000.00	income	2025-08-10
3	2	4	₹8000.00	expense	2025-08-05
2	1	3	₹1500.00	expense	2025-08-03
1	1	1	₹50000.00	income	2025-08-01

- **Why: Proof of successful insertion + data visible via the UI.**

---

6. Goals table — AFTER adding income (trigger effect)

- **Feature: Trigger apply_income_to_goal_after_insert updates goals.current_amount.**
- **Action: Refresh the Goals table in the UI (select Goals and Refresh).**
- **Screenshot to take:**
  - **Filename: 07_goals_after_income.png**
  - **Caption: *Goals table after adding the income transaction — current_amount increased (trigger applied).***
- **Why: Demonstrates the database trigger automatically updated goal progress — a key DBMS feature.**

---

7. Users table — BEFORE Deleting a User

- **Feature: Show users list to choose which user to delete.**
- **Action: Ensure Users is selected and Refresh. Capture the Users table before deletion.**

Users (Delete to Demonstrate CASCADE)

Deleting a user will CASCADE delete all their transactions, goals, and budgets.

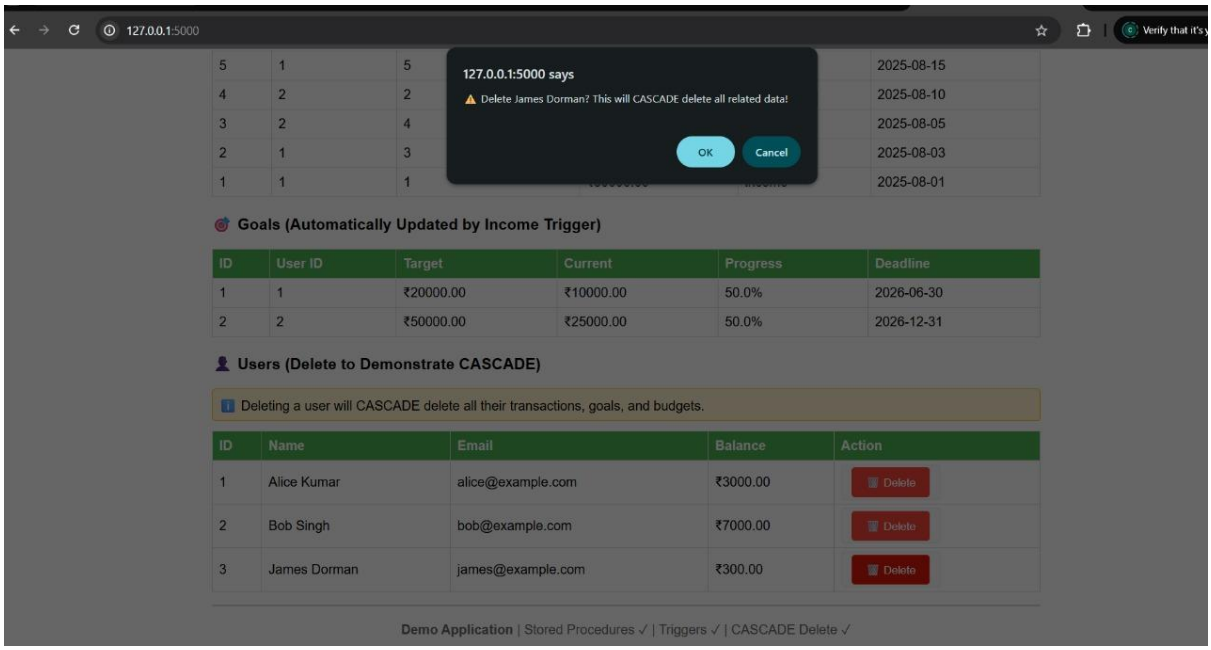
ID	Name	Email	Balance	Action
1	Alice Kumar	alice@example.com	₹3000.00	Delete
2	Bob Singh	bob@example.com	₹7000.00	Delete
4	James Dorman	james@example.com	₹0.00	Delete

- **Why: Baseline before delete for cascade demonstration.**

---

8. Delete User — Confirmation (use the delete button)

- **Feature: Delete a user via the delete button in the Users table.**
- **Action: Click Delete on a user row; browser confirm box will appear (click OK).**
- **Screenshot to take:**



- **Why: Shows the UI action that triggers the cascade delete.**

Triggers, Procedures/Functions, Nested query, Join, Aggregate queries

1. Triggers (SQL + explanation)

1.1 check_transaction_type_before_insert — validate category type before inserting

```
DELIMITER $$
CREATE TRIGGER check_transaction_type_before_insert
BEFORE INSERT ON transactions
FOR EACH ROW
BEGIN
  DECLARE cat_type ENUM('income','expense');
  SELECT type INTO cat_type FROM categories WHERE category_id = NEW.category_id;
  IF cat_type <> NEW.type THEN
    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Transaction type does not match category type';
  END IF;
END$$
DELIMITER ;
```

What it does: prevents inserting a transaction whose type (income/expense) does not match the type of the chosen category.
Why useful: enforces data integrity at the DB level so the application cannot create inconsistent rows.

1.2 check_transaction_type_before_update — validate on update

```
DELIMITER $$
CREATE TRIGGER check_transaction_type_before_update
BEFORE UPDATE ON transactions
FOR EACH ROW
BEGIN
  DECLARE cat_type ENUM('income','expense');
  IF NEW.category_id IS NOT NULL THEN
```

```
    SELECT type INTO cat_type FROM categories WHERE category_id = NEW.category_id;
    IF cat_type <> NEW.type THEN
      SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Updated transaction type does not match updated category type';
    END IF;
  END IF;
END$$
DELIMITER ;
```

What it does: same validation but for updates to transactions.
Why useful: avoids inconsistent state if transaction fields change later.

1.3 apply_income_to_goal_after_insert — update goals when income is added

```
DELIMITER $$
CREATE TRIGGER apply_income_to_goal_after_insert
AFTER INSERT ON transactions
FOR EACH ROW
BEGIN
  DECLARE v_goal_id INT;
  IF NEW.type = 'income' THEN
    SELECT goal_id INTO v_goal_id
    FROM goals
    WHERE user_id = NEW.user_id
      AND current_amount < target_amount
      AND (deadline IS NULL OR deadline >= NEW.date)
    ORDER BY deadline IS NULL, deadline ASC
    LIMIT 1;

    IF v_goal_id IS NOT NULL THEN
      UPDATE goals
      SET current_amount = LEAST(target_amount, current_amount + NEW.amount)
```

```
    WHERE goal_id = v_goal_id;
  END IF;
 END IF;
END$$
DELIMITER ;
```

What it does: after an income transaction is inserted, picks the earliest open goal for that user and adds the income amount to current_amount (but caps it at target_amount).
Why useful: automatic goal progress update demonstrates triggers automating business rules without app code changes.

---

2. Stored Procedure and Function (SQL + explanation)

2.1 sp_add_transaction — add transaction with validation & return id

```
DELIMITER $$
CREATE PROCEDURE sp_add_transaction(
 IN p_user_id INT,
 IN p_category_id INT,
 IN p_amount DECIMAL(12,2),
 IN p_type VARCHAR(10),
 IN p_date DATE
)
BEGIN
 IF p_amount <= 0 THEN
  SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Amount must be > 0';
 END IF;

 INSERT INTO transactions (user_id, category_id, amount, type, date)
 VALUES (p_user_id, p_category_id, p_amount, p_type, p_date);

 SELECT LAST_INSERT_ID() AS transaction_id;
END$$
```

```
DELIMITER ;
```

What it does: centralizes transaction insertion, enforces amount > 0, inserts row, and returns the new transaction_id.
Why useful: app calls this single entry point (app.py uses cur.callproc('sp_add_transaction', ...)), ensuring validations and side-effects (triggers) occur consistently.

2.2 fn_get_user_balance — compute net balance (income - expense)

```
DELIMITER $$

CREATE FUNCTION fn_get_user_balance(p_user_id INT)

RETURNS DECIMAL(14,2)

DETERMINISTIC

BEGIN

  DECLARE v_income DECIMAL(14,2) DEFAULT 0.00;

  DECLARE v_expense DECIMAL(14,2) DEFAULT 0.00;


  SELECT IFNULL(SUM(amount),0.00) INTO v_income

  FROM transactions

  WHERE user_id = p_user_id AND type = 'income';


  SELECT IFNULL(SUM(amount),0.00) INTO v_expense

  FROM transactions

  WHERE user_id = p_user_id AND type = 'expense';


  RETURN v_income - v_expense;

END$$

DELIMITER ;
```

What it does: returns the user's current balance based on stored transactions.
Why useful: quick, reusable calculation inside SQL or from the Flask app.

---

3. Nested Queries (Subqueries) — examples & explanations

3.1 Users whose total expense > 10000 (subquery)

```
SELECT user_id
FROM (
  SELECT user_id, SUM(amount) AS total_expense
  FROM transactions
  WHERE type = 'expense'
  GROUP BY user_id
) t
WHERE total_expense > 10000;
```

What it does: inner query computes total expense per user; outer query filters users exceeding threshold.
Why useful: find heavy spenders for reporting.

3.2 Users with their latest transaction date (correlated subquery)

```
SELECT u.user_id, u.name,
  (SELECT MAX(t.date) FROM transactions t WHERE t.user_id = u.user_id) AS last_txn_date
FROM users u;
```

What it does: for each user, picks the most recent transaction date via correlated subquery.
Why useful: shows user activity recency.

3.3 Goals close to deadline with remaining amount > X (nested + filter)

```
SELECT g.goal_id, g.user_id, g.target_amount, g.current_amount, g.deadline,
      (g.target_amount - g.current_amount) AS remaining
FROM goals g
WHERE g.deadline IS NOT NULL
  AND g.deadline <= DATE_ADD(CURDATE(), INTERVAL 30 DAY)
  AND (g.target_amount - g.current_amount) > 0;
```

What it does: finds goals expiring in 30 days with remaining amount > 0.
Why useful: actionable reminders/alerts.

---

4. Joins — examples & explanations

4.1 Transactions with category name and user name

```
SELECT t.transaction_id, u.user_id, u.name AS user_name,
    c.category_id, c.name AS category_name,
    t.amount, t.type, t.date
FROM transactions t
JOIN users u ON t.user_id = u.user_id
JOIN categories c ON t.category_id = c.category_id
ORDER BY t.date DESC
LIMIT 200;
```

What it does: joins three tables to show human-friendly transaction rows.
Why useful: essential for UI listing (this is basically what app.py shows for transactions).

4.2 Budgets with category and user info

```
SELECT b.budget_id, u.name AS user_name, c.name AS category_name,
    b.limit_amount, b.start_date, b.end_date
FROM budgets b
JOIN users u ON b.user_id = u.user_id
JOIN categories c ON b.category_id = c.category_id
ORDER BY b.start_date DESC;
```

What it does: creates readable budget rows with user and category names.
Why useful: frontend display and reports.

---

5. Aggregate Queries — examples & explanations

5.1 Monthly total per category for a user

```
SELECT c.name AS category_name, c.type, SUM(t.amount) AS total_amount
FROM transactions t
JOIN categories c ON t.category_id = c.category_id
WHERE t.user_id = 1 AND DATE_FORMAT(t.date, '%Y-%m') = '2025-08'
GROUP BY c.category_id, c.name, c.type
ORDER BY total_amount DESC;
```

What it does: sums amounts per category for a single user in a month.
Why useful: generate monthly expense/income breakdown.

5.2 Total income and total expense per user (one row per user)

```
SELECT u.user_id, u.name,
    COALESCE(SUM(CASE WHEN t.type='income' THEN t.amount END),0) AS total_income,
    COALESCE(SUM(CASE WHEN t.type='expense' THEN t.amount END),0) AS total_expense
FROM users u
LEFT JOIN transactions t ON u.user_id = t.user_id
GROUP BY u.user_id, u.name;
```

What it does: aggregates income/expense per user in one query using conditional aggregation.
Why useful: quick summary of users' cashflow.

5.3 Top 3 expense categories across all users

```
SELECT c.name, SUM(t.amount) AS total_spent
FROM transactions t
JOIN categories c ON t.category_id = c.category_id
WHERE t.type='expense'
GROUP BY c.category_id, c.name
ORDER BY total_spent DESC
LIMIT 3;
```

What it does: finds categories where users spend most.
Why useful: analytics and insights.

---

6. How to call the stored procedure from Flask (your app does this already)

Your app.py uses this exact pattern:

```
conn = get_conn()
cur = conn.cursor()
cur.callproc('sp_add_transaction', (int(user_id), int(category_id), float(amount), typ, date))
conn.commit()
```

```
# To read any result set the stored proc SELECTed:
for res in cur.stored_results():
    rows = res.fetchall()
    if rows:
        last_id = rows[0][0] if isinstance(rows[0], tuple) else rows[0]
```

Code snippets for invoking the Procedures/Functions/Trigger

1. Invoking Stored Procedure – sp_add_transaction

```
conn = get_conn()
cur = conn.cursor()

cur.callproc('sp_add_transaction', (
    int(user_id),
    int(category_id),
    float(amount),
    typ,
    date
))

conn.commit()

# Fetch the returned transaction_id
for result in cur.stored_results():
    last_id = result.fetchone()[0]
```

What this does:
Calls the stored procedure to insert a new transaction and retrieves the inserted ID.

---

2. Invoking SQL Function – fn_get_user_balance

```
conn = get_conn()
cur = conn.cursor()

cur.execute("SELECT fn_get_user_balance(%s)", (user_id,))
balance = cur.fetchone()[0]
```

What this does:
Executes the SQL function and returns the user's balance directly from the database.

---

3. Trigger Invocation (Indirect)

Triggers run automatically; the app doesn't call them directly.
They fire when the app inserts transactions.

Example: Trigger runs after INSERT

```
conn = get_conn()

cur = conn.cursor()


# This INSERT triggers:

# - BEFORE triggers (validate category-type)

# - AFTER trigger (auto-update goals)

cur.callproc('sp_add_transaction', (

    user_id, category_id, amount, typ, date

))


conn.commit()
```

What this does:
The trigger such as apply_income_to_goal_after_insert executes automatically **after the stored procedure inserts a new transaction**.

GitHub Link: https://github.com/Kingjoy7/Personal-Expense-and-Finance-Tracker