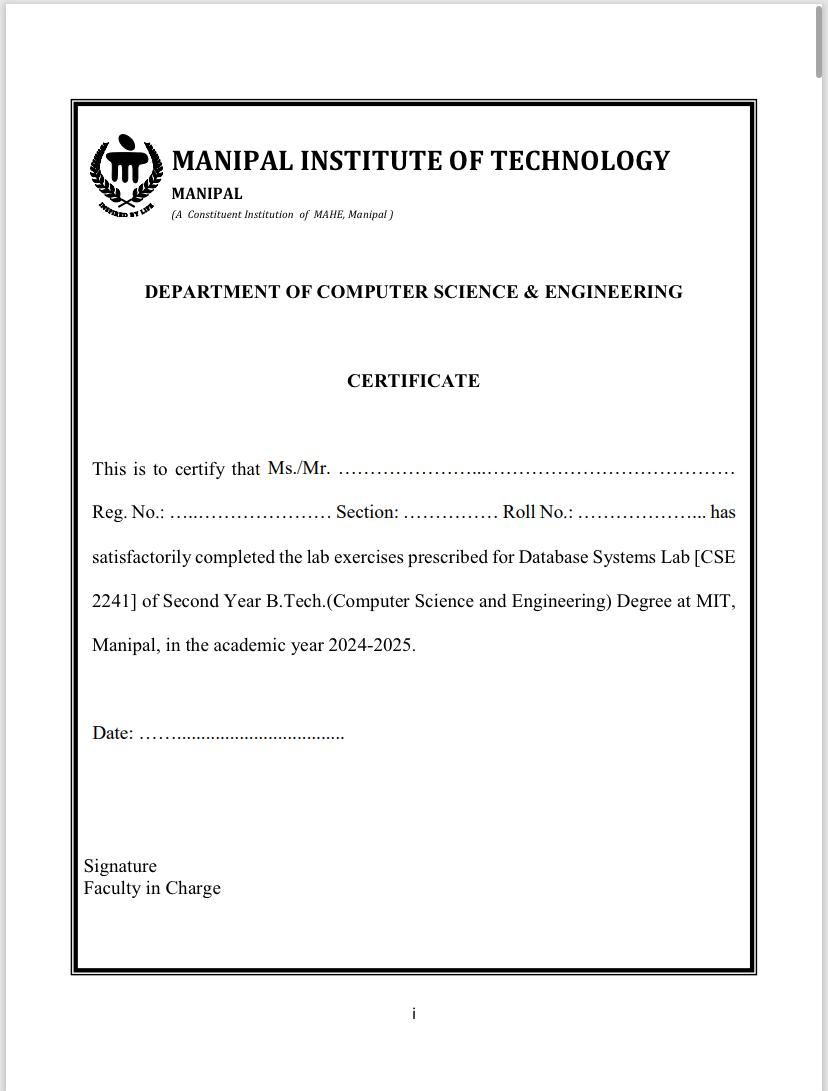
# DBS Lab Mini Project

# Expense Tracking System

Team Members:

Dhruv Kaushik Thacker - 230905566

Samarth R Patil - 230905342



# Index

1. Abstract
2. Problem Statement
3. Entity Relationship Diagram
4. Database Schema
5. Normalization
6. Relational Tables with Sample Data
7. DDL Scripts
8. Basic SQL Queries
9. Intermediate & Complex Queries
10. PL/SQL Procedures
11. Functions
12. Triggers
13. Python Setup Script
14. Flask Web Application Code
15. User Interface Design
16. Screenshots & File Directory
17. Team Members

# Abstract

The Expense Tracker is a versatile and user-friendly financial management application designed to empower individuals to take full control of their daily expenses. Leveraging the robust capabilities of Oracle SQL and advanced PL/SQL procedures and triggers, the backend ensures superior transactional integrity, data security, and high performance. The application’s server-side logic is built using Python with Flask, seamlessly connecting to the Oracle database. At the same time, the front end employs HTML, CSS, Bootstrap, and Chart.js to deliver an engaging, interactive, and responsive user experience.

This modular system allows users to track and categorise expenditures, set personalised budget limits, and receive real-time alerts when spending thresholds are exceeded. With built-in reporting and visualisation tools that illustrate trends and spending patterns over time, the Expense Tracker not only simplifies personal budgeting but also encourages smarter financial decisions. Designed with scalability and customization in mind, it is a comprehensive tool for individuals looking to develop better financial habits and maintain fiscal discipline.

# Problem Statement

The absence of systematic expense tracking leads to poor financial planning and overspending. Individuals often lack insight into where and how their money is spent, leading to budget overruns. The Expense Tracker addresses these issues by offering a structured database schema, user-defined limits, and automated alerting features that notify users when their expenditures approach or exceed pre-set thresholds.

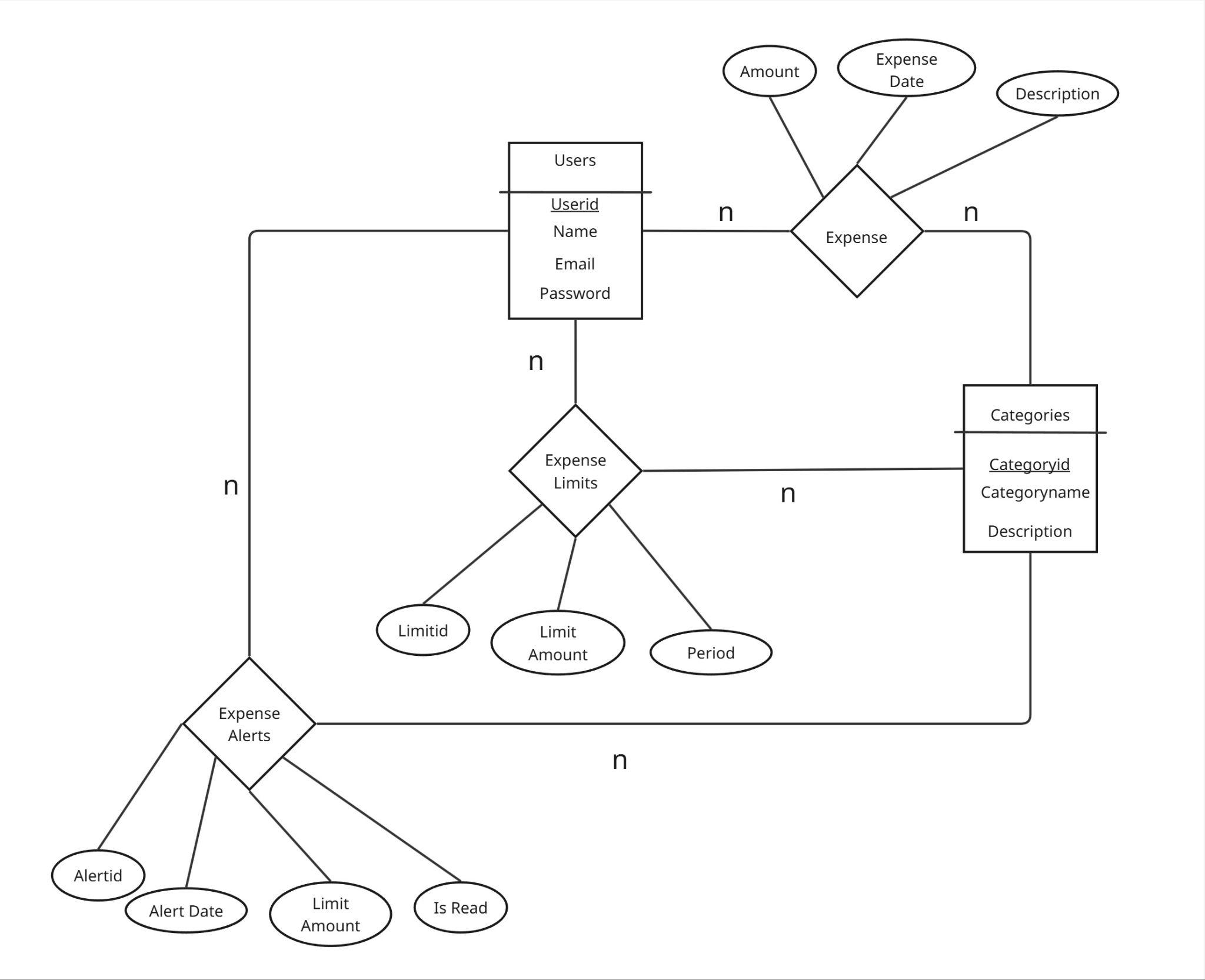
By categorizing expenses and maintaining historical records, users can monitor their financial behavior over time. This visibility encourages more thoughtful decision-making and better allocation of income. The system supports both category-specific and overall budget controls, making it flexible for diverse financial habits. Alerts serve as timely reminders that help curb unnecessary spending before it escalates. Graphical reports provide quick insights into trends, enabling users to identify high-spending areas. The application is especially useful for students, working professionals, and households aiming to manage finances more effectively. Real-time tracking prevents surprises at the end of the month. The tool promotes financial discipline and long-term savings. Overall, it transforms passive logging into active money management with actionable insights.

# ER Diagram

The ER diagram consists of five primary entities: Users, Categories, Expenses, Expense\_Limits, and Expense\_Alerts.

* Users table maintains account-level data.
* Categories define expense groups.
* Expenses table stores transactional data tied to user and category.
* Expense\_Limits manages category or global budget limits.
* Expense\_Alerts log budget breaches.

Each table is linked through well-defined foreign key constraints ensuring referential integrity.



# DDL Commands

The schema includes DDL for each table: Users, Categories, Expenses, Expense\_Limits, and Expense\_Alerts.

Sequences and triggers auto-increment primary keys to avoid manual errors. Triggers are defined on each insert operation for ID auto-generation.

Foreign key constraints are strictly enforced to maintain data consistency between Users, Categories, and Expenses.

CREATE TABLE Users (

user\_id NUMBER PRIMARY KEY,

name VARCHAR2(100) NOT NULL,

email VARCHAR2(100) NOT NULL UNIQUE,

password VARCHAR2(255) NOT NULL

);

CREATE TABLE Categories (

category\_id NUMBER PRIMARY KEY,

user\_id NUMBER NOT NULL,

category\_name VARCHAR2(100) NOT NULL,

description VARCHAR2(255),

CONSTRAINT fk\_user\_category FOREIGN KEY (user\_id) REFERENCES Users(user\_id)

);

CREATE TABLE Expenses (

expense\_id NUMBER PRIMARY KEY,

user\_id NUMBER NOT NULL,

category\_id NUMBER NOT NULL,

amount NUMBER(10,2) NOT NULL,

expense\_date DATE NOT NULL,

description VARCHAR2(4000),

CONSTRAINT fk\_user\_expense FOREIGN KEY (user\_id) REFERENCES Users(user\_id),

CONSTRAINT fk\_category\_expense FOREIGN KEY (category\_id) REFERENCES Categories(category\_id)

);

CREATE TABLE Expense\_Limits (

limit\_id NUMBER PRIMARY KEY,

user\_id NUMBER NOT NULL,

category\_id NUMBER,

limit\_amount NUMBER(10,2) NOT NULL,

period VARCHAR2(20) DEFAULT 'monthly',

CONSTRAINT fk\_user\_limit FOREIGN KEY (user\_id) REFERENCES Users(user\_id),

CONSTRAINT fk\_category\_limit FOREIGN KEY (category\_id) REFERENCES Categories(category\_id)

);

CREATE TABLE Expense\_Alerts (

alert\_id NUMBER PRIMARY KEY,

user\_id NUMBER NOT NULL,

expense\_id NUMBER NOT NULL,

alert\_date DATE DEFAULT SYSDATE,

limit\_amount NUMBER(10,2) NOT NULL,

is\_read NUMBER(1) DEFAULT 0,

CONSTRAINT fk\_user\_alert FOREIGN KEY (user\_id) REFERENCES Users(user\_id),

CONSTRAINT fk\_expense\_alert FOREIGN KEY (expense\_id) REFERENCES Expenses(expense\_id)

);

# Normalization

The database schema follows **3NF normalization**:

* Each table has atomic attributes.
* There are no partial dependencies as non-key attributes depend on the whole primary key.
* There are no transitive dependencies; derived data such as expense totals is calculated via queries, not stored redundantly.

This ensures the database is optimized for storage, consistency, and scalability.  
 By eliminating redundancy, the schema reduces the risk of anomalies during insert, update, or delete operations.  
 It also simplifies query logic and improves performance when handling large volumes of user expense data.  
 The modular structure makes the system easier to extend in the future, such as adding shared budgets or multi-user reports.

# Relational Tables with Sample Data

Example:

Users Table:

(1, 'Alice', ['alice@example.com',](mailto:%27alice@example.com) 'hashed\_pwd') Categories Table:

(101, 1, 'Groceries', 'Food-related shopping') Expenses Table:

(5001, 1, 101, 1200.00, TO\_DATE('2024-04-10', 'YYYY-MM-DD'), 'Monthly groceries at supermarket')

These entries illustrate how user-specific categories and associated expenses are linked through foreign keys.

-- Insert record into Users table:

INSERT INTO Users (user\_id, name, email, password)

VALUES (1, 'Alice', 'alice@example.com', 'hashed\_pwd');

-- Insert record into Categories table:

INSERT INTO Categories (category\_id, user\_id, category\_name, description)

VALUES (101, 1, 'Groceries', 'Food-related shopping');

-- Insert record into Expenses table:

INSERT INTO Expenses (expense\_id, user\_id, category\_id, amount, expense\_date, description)

VALUES (5001, 1, 101, 1200.00, TO\_DATE('2024-04-10', 'YYYY-MM-DD'), 'Monthly groceries at supermarket');

# SQL Queries

### 1. User Authentication (Login)

SELECT user\_id, name

FROM Users

WHERE email = :email

AND password = :password;

*Explanation*: This query retrieves the user ID and name for a user whose email and password match the provided values. It is used to verify credentials during login.

### 2. Dashboard Queries

#### a. Retrieve Recent Expenses:

SELECT e.expense\_id, e.expense\_date, c.category\_name, e.amount, e.description

FROM Expenses e

JOIN Categories c ON e.category\_id = c.category\_id

WHERE e.user\_id = :user\_id

ORDER BY e.expense\_date DESC;

*Explanation*: Fetches recent expense entries for the logged-in user, including the associated category names, sorted by the date in descending order.

#### b. Calculate Current Month’s Total Expense:

SELECT NVL(SUM(amount), 0)

FROM Expenses

WHERE user\_id = :user\_id

AND expense\_date BETWEEN :first\_day AND :last\_day;

*Explanation*: Computes the total expenses made by the user within the current month. The NVL function ensures that if no expenses exist, it returns 0.

#### c. Get Category-wise Expense Breakdown for Current Month:

SELECT c.category\_name, NVL(SUM(e.amount), 0) as total

FROM Categories c

LEFT JOIN Expenses e ON c.category\_id = e.category\_id

AND e.expense\_date BETWEEN :first\_day AND :last\_day

WHERE c.user\_id = :user\_id

GROUP BY c.category\_name

ORDER BY total DESC;

*Explanation*: Joins Categories with Expenses to provide a total expense sum per category for a specified period, useful for visualizing which categories are consuming the most budget.

#### d. Count of Unread Alerts:

SELECT COUNT(\*)

FROM Expense\_Alerts

WHERE user\_id = :user\_id

AND is\_read = 0;

*Explanation*: Counts the number of alerts for the logged-in user that haven't been marked as read.

### 3. Expense Management

#### a. Adding an Expense (via PL/SQL Procedure)

-- This is called using a PL/SQL procedure:

CALL add\_expense\_with\_limit\_check(:user\_id, :category\_id, :amount, :expense\_date, :description, :limit\_exceeded, :limit\_amount, :expense\_id);

*Explanation*: Adds a new expense record while simultaneously checking if it exceeds a user-defined limit. The procedure encapsulates the insert logic and limit checking.

#### b. Creating a Limit Alert (if required)

CALL create\_limit\_alert(:user\_id, :expense\_id, :limit\_amount);

*Explanation*: Invoked if an expense surpasses the set limit, this procedure logs an alert into the Expense\_Alerts table.

### 4. Budget Management

#### a. Fetch Available Categories for a User:

SELECT category\_id, category\_name

FROM Categories

WHERE user\_id = :user\_id;

*Explanation*: Retrieves all categories associated with a user to facilitate setting up or managing budgets.

#### b. Retrieve Existing Budget Limits:

SELECT l.limit\_id, l.category\_id, c.category\_name, l.limit\_amount, l.period

FROM Expense\_Limits l

LEFT JOIN Categories c ON l.category\_id = c.category\_id

WHERE l.user\_id = :user\_id;

*Explanation*: Provides details of the budget limits set by the user, including the category, amount, and time period.

#### c. Insert a New Budget Limit:

INSERT INTO Expense\_Limits (limit\_id, user\_id, category\_id, limit\_amount, period)

VALUES (:limit\_id, :user\_id, :category\_id, :limit\_amount, :period);

*Explanation*: Adds a new budget limit record for the user. The limit can be associated with a specific category or be overall if category\_id is null.

#### d. Delete an Existing Budget Limit:

DELETE FROM Expense\_Limits

WHERE limit\_id = :limit\_id

AND user\_id = :user\_id;

*Explanation*: Removes a budget limit entry for the user.

### 5. Monthly Reporting

SELECT

c.category\_name,

NVL(SUM(e.amount), 0) as total\_amount,

COUNT(e.expense\_id) as transaction\_count,

MIN(e.amount) as min\_expense,

MAX(e.amount) as max\_expense,

NVL(AVG(e.amount), 0) as avg\_expense

FROM

Categories c

LEFT JOIN

Expenses e ON c.category\_id = e.category\_id

AND e.user\_id = :user\_id

AND e.expense\_date BETWEEN :first\_day AND :last\_day

WHERE

c.user\_id = :user\_id

GROUP BY

c.category\_name

ORDER BY

total\_amount DESC;

*Explanation*: Generates a summary for the current month, breaking down expenses by category along with aggregate data like total, count, and average amounts.

### 6. Expense Trends Over Time

SELECT

TO\_CHAR(TRUNC(e.expense\_date, 'MM'), 'YYYY-MM') as month,

c.category\_name,

SUM(e.amount) as total\_amount

FROM

Expenses e

JOIN

Categories c ON e.category\_id = c.category\_id

WHERE

e.user\_id = :user\_id

AND e.expense\_date BETWEEN :start\_date AND :end\_date

GROUP BY

TO\_CHAR(TRUNC(e.expense\_date, 'MM'), 'YYYY-MM'),

c.category\_name

ORDER BY

month, c.category\_name;

*Explanation*: Provides a month-by-month breakdown of expenses by category for the user, essential for charting and trend analysis.

### 7. User Registration

#### a. Check if Email Already Exists:

SELECT COUNT(\*)

FROM Users

WHERE email = :email;

*Explanation*: Determines whether an email is already registered to prevent duplicate accounts.

#### b. Insert a New User:

INSERT INTO Users (name, email, password)

VALUES (:name, :email, :password);

*Explanation*: Registers a new user into the system.

#### c. Retrieve the Newly Created User ID:

SELECT user\_id

FROM Users

WHERE email = :email;

*Explanation*: Fetches the newly registered user's ID, which is used for setting up default categories and other user-specific records.

#### d. Insert Default Categories for a New User:

INSERT INTO Categories (user\_id, category\_name)

VALUES (:user\_id, :category\_name);

*Explanation*: Automatically assigns a set of default categories (like Food, Housing, etc.) to the new user.

### 8. Dashboard Chart Data (API Endpoint)

SELECT TO\_CHAR(TRUNC(expense\_date, 'MM'), 'YYYY-MM') as month, SUM(amount) as total

FROM Expenses

WHERE user\_id = :user\_id

AND expense\_date >= :start\_date

GROUP BY TO\_CHAR(TRUNC(expense\_date, 'MM'), 'YYYY-MM')

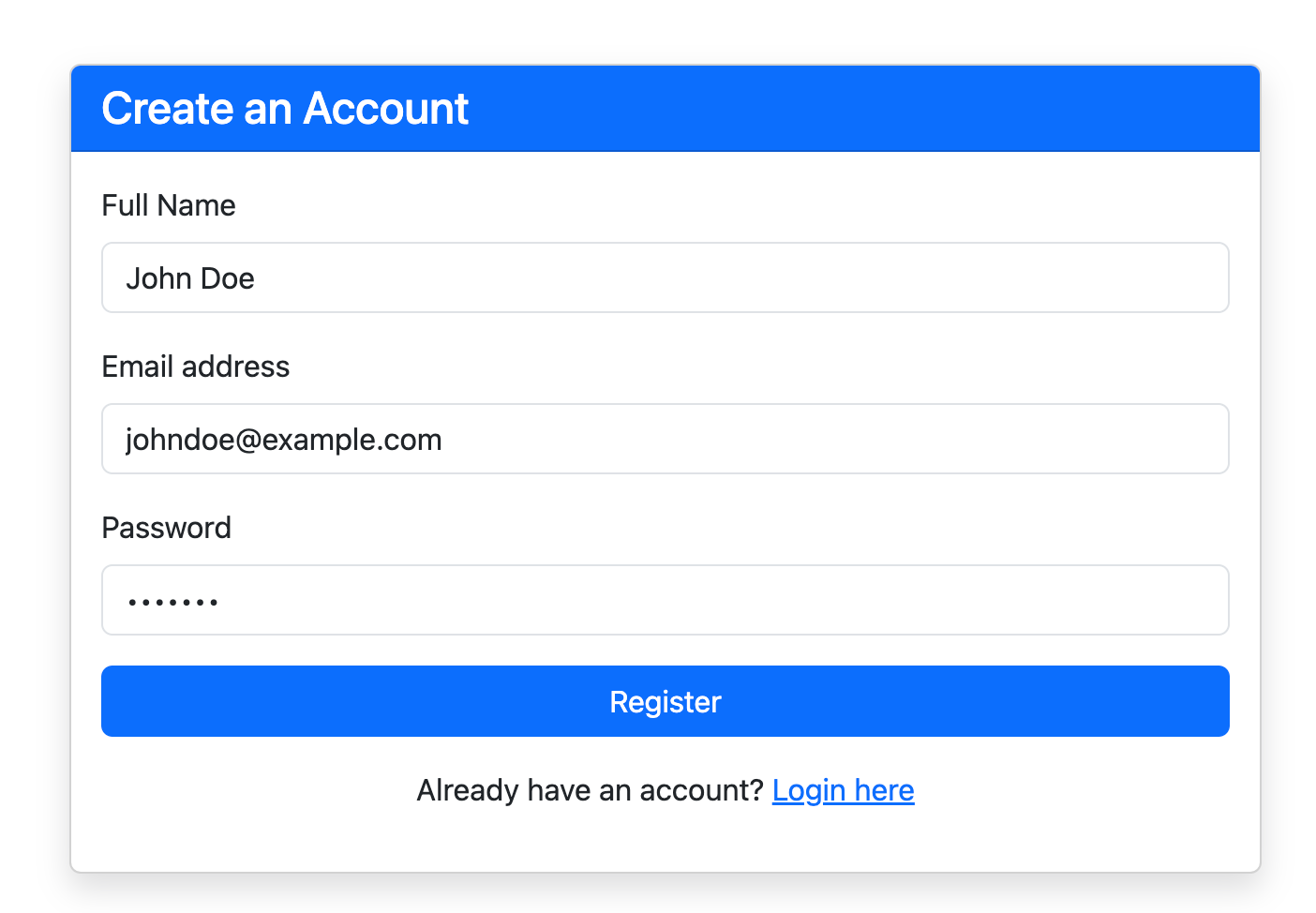
ORDER BY month;

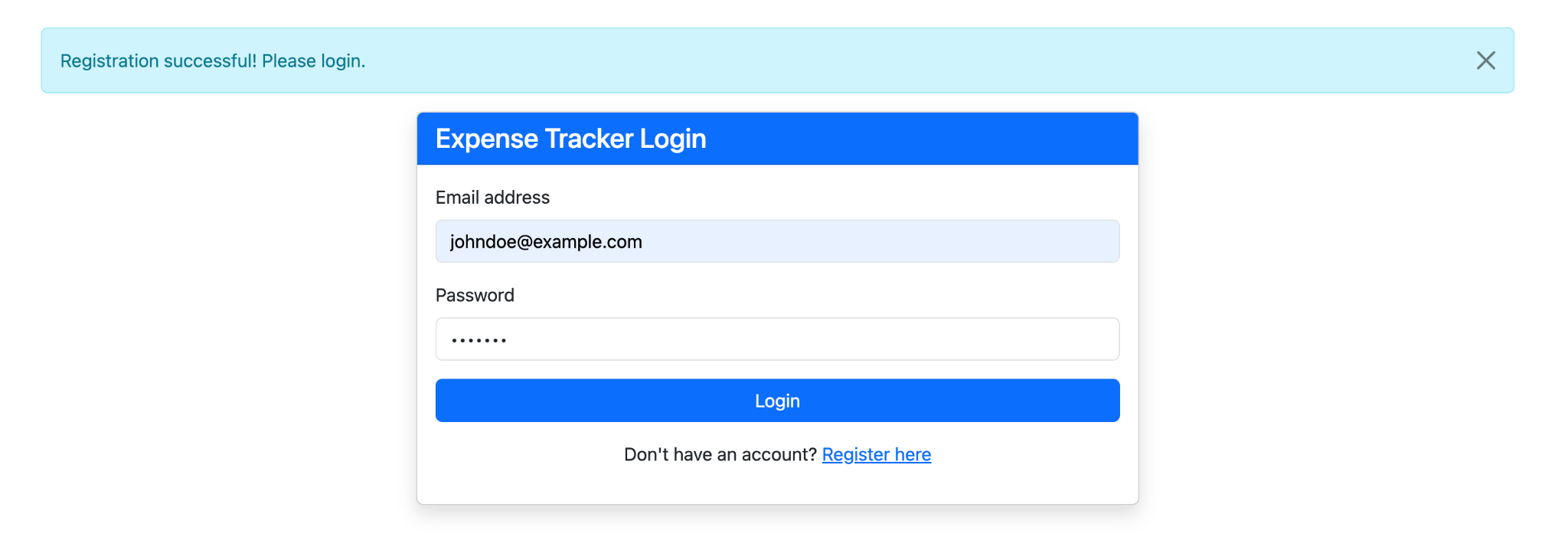
*Explanation*: Gathers expense totals per month over the last six months to be used for generating visual charts on the dashboard.

# User Interface Design

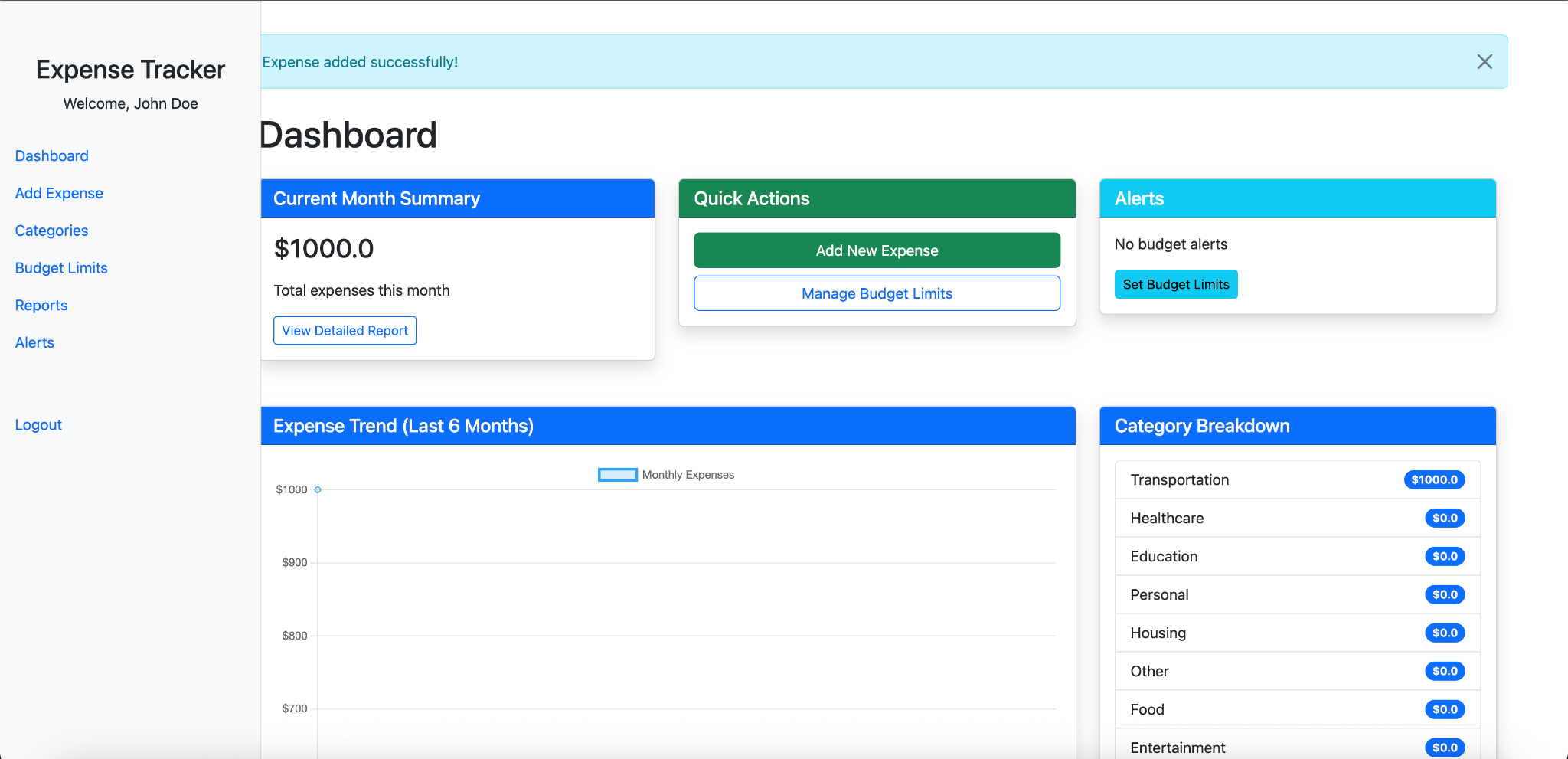
The application interface is built using Flask + HTML + Bootstrap.

* Login and registration page

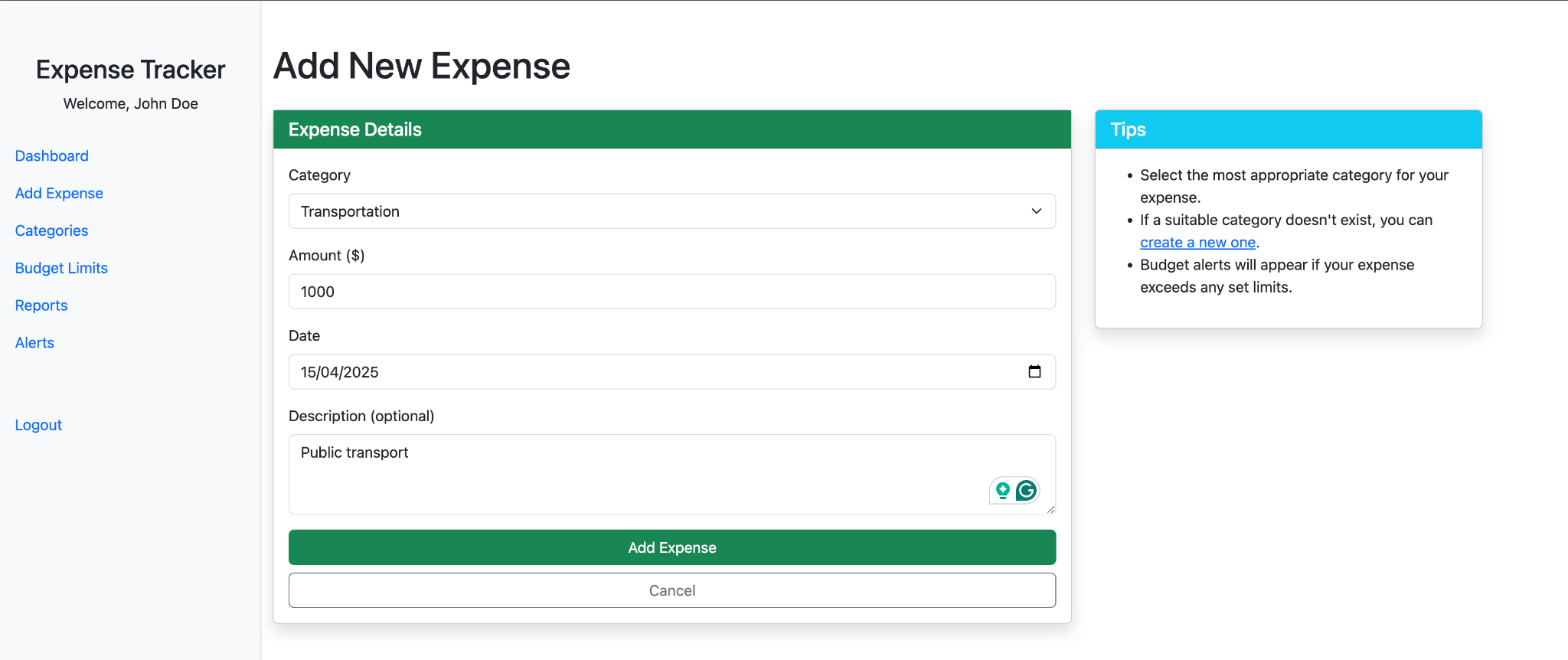




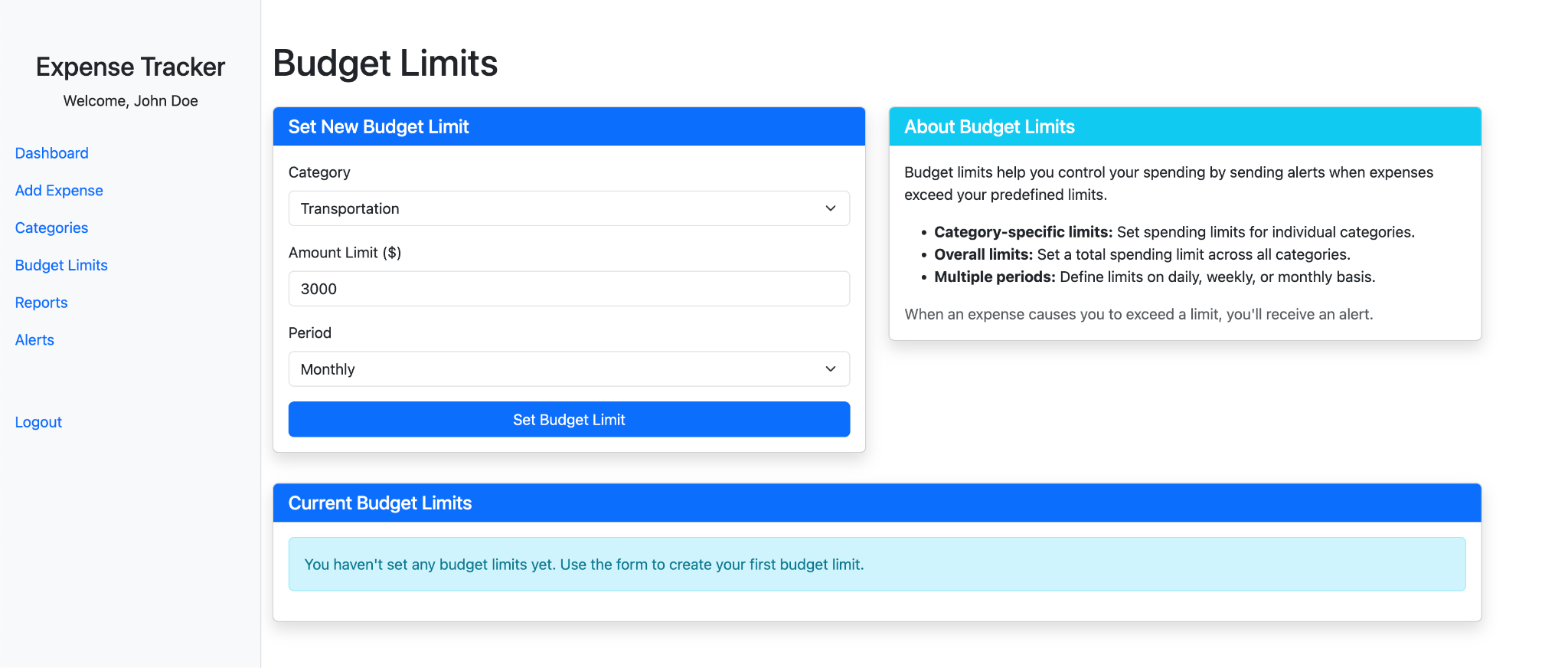
* Dashboard with charts and expense breakdowns



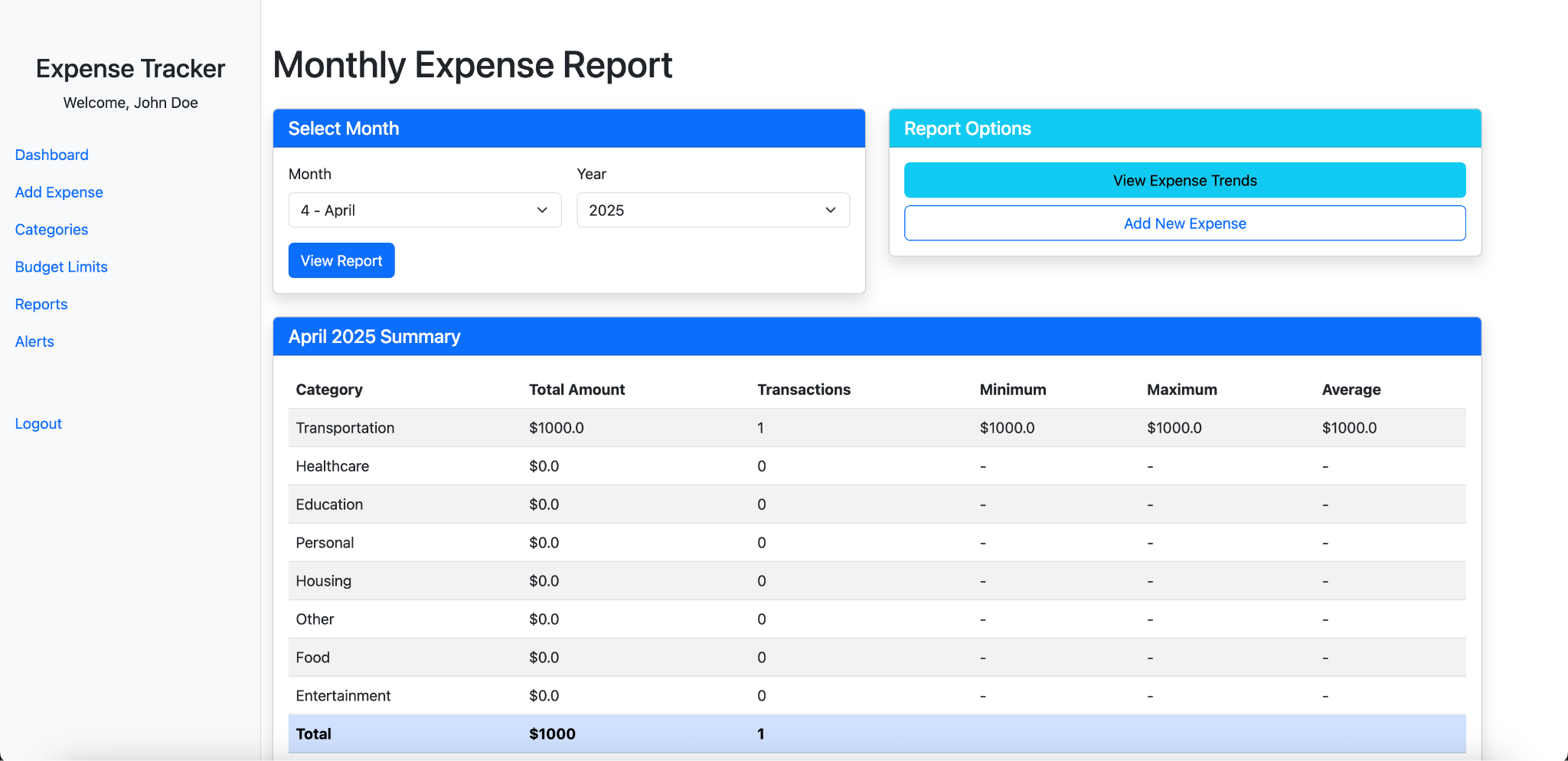
* Add expense with real-time validation



* Set budget limits and receive pop-up alerts



* Export reports monthly



# PL/SQL Queries

### 1. Procedure: check\_expense\_limit

This procedure checks whether a new expense, when added, exceeds the user- or category-specific budget limit. It calculates the total expenses within a specified period (monthly, weekly, or daily) and sets output parameters indicating whether the limit is exceeded and what the limit amount is.

CREATE OR REPLACE PROCEDURE check\_expense\_limit(

p\_user\_id IN NUMBER,

p\_category\_id IN NUMBER,

p\_amount IN NUMBER,

p\_expense\_date IN DATE,

p\_limit\_exceeded OUT NUMBER,

p\_limit\_amount OUT NUMBER

) AS

v\_month\_expenses NUMBER := 0;

v\_limit\_amount NUMBER := 0;

v\_period VARCHAR2(20) := 'monthly';

v\_start\_date DATE;

v\_end\_date DATE;

v\_limit\_exists NUMBER := 0;

BEGIN

-- Initialize output parameters

p\_limit\_exceeded := 0;

p\_limit\_amount := 0;

-- First check if category-specific limit exists

SELECT COUNT(\*) INTO v\_limit\_exists

FROM Expense\_Limits

WHERE user\_id = p\_user\_id

AND category\_id = p\_category\_id;

-- Only proceed if a limit exists for this category

IF v\_limit\_exists > 0 THEN

SELECT NVL(limit\_amount, 0), NVL(period, 'monthly')

INTO v\_limit\_amount, v\_period

FROM Expense\_Limits

WHERE user\_id = p\_user\_id

AND category\_id = p\_category\_id

AND ROWNUM = 1;

IF v\_limit\_amount > 0 THEN

-- Determine period dates based on expense date

IF v\_period = 'monthly' THEN

v\_start\_date := TRUNC(p\_expense\_date, 'MM');

v\_end\_date := LAST\_DAY(p\_expense\_date);

ELSIF v\_period = 'weekly' THEN

v\_start\_date := TRUNC(p\_expense\_date, 'IW');

v\_end\_date := v\_start\_date + 6;

ELSIF v\_period = 'daily' THEN

v\_start\_date := TRUNC(p\_expense\_date);

v\_end\_date := v\_start\_date;

END IF;

-- Get total expenses for this category in the period

SELECT NVL(SUM(amount), 0) INTO v\_month\_expenses

FROM Expenses

WHERE user\_id = p\_user\_id

AND category\_id = p\_category\_id

AND expense\_date BETWEEN v\_start\_date AND v\_end\_date;

-- Check if this new expense will exceed the limit

IF (v\_month\_expenses + p\_amount) > v\_limit\_amount THEN

p\_limit\_exceeded := 1;

p\_limit\_amount := v\_limit\_amount;

RETURN;

END IF;

END IF;

END IF;

-- Check if overall limit exists (for user without category specificity)

SELECT COUNT(\*) INTO v\_limit\_exists

FROM Expense\_Limits

WHERE user\_id = p\_user\_id

AND category\_id IS NULL;

-- Then check overall limit if no category-specific limit was exceeded

IF v\_limit\_exists > 0 THEN

SELECT NVL(limit\_amount, 0), NVL(period, 'monthly')

INTO v\_limit\_amount, v\_period

FROM Expense\_Limits

WHERE user\_id = p\_user\_id

AND category\_id IS NULL

AND ROWNUM = 1;

IF v\_limit\_amount > 0 THEN

-- Determine period dates

IF v\_period = 'monthly' THEN

v\_start\_date := TRUNC(p\_expense\_date, 'MM');

v\_end\_date := LAST\_DAY(p\_expense\_date);

ELSIF v\_period = 'weekly' THEN

v\_start\_date := TRUNC(p\_expense\_date, 'IW');

v\_end\_date := v\_start\_date + 6;

ELSIF v\_period = 'daily' THEN

v\_start\_date := TRUNC(p\_expense\_date);

v\_end\_date := v\_start\_date;

END IF;

-- Get total expenses for all categories in the period

SELECT NVL(SUM(amount), 0) INTO v\_month\_expenses

FROM Expenses

WHERE user\_id = p\_user\_id

AND expense\_date BETWEEN v\_start\_date AND v\_end\_date;

-- Check if this new expense will exceed the overall limit

IF (v\_month\_expenses + p\_amount) > v\_limit\_amount THEN

p\_limit\_exceeded := 1;

p\_limit\_amount := v\_limit\_amount;

END IF;

END IF;

END IF;

END;

/

### 2. Procedure: add\_expense\_with\_limit\_check

This procedure is responsible for adding a new expense entry. It first calls the check\_expense\_limit procedure to verify if the expense exceeds the set limits. If so, the calling routine can create an alert accordingly.

CREATE OR REPLACE PROCEDURE add\_expense\_with\_limit\_check(

p\_user\_id IN NUMBER,

p\_category\_id IN NUMBER,

p\_amount IN NUMBER,

p\_expense\_date IN DATE,

p\_description IN VARCHAR2,

p\_limit\_exceeded OUT NUMBER,

p\_limit\_amount OUT NUMBER,

p\_expense\_id OUT NUMBER

) AS

BEGIN

-- First check if expense would exceed any limits

check\_expense\_limit(

p\_user\_id,

p\_category\_id,

p\_amount,

p\_expense\_date,

p\_limit\_exceeded,

p\_limit\_amount

);

-- Generate a new expense\_id

SELECT seq\_expenses.NEXTVAL INTO p\_expense\_id FROM dual;

-- Insert the expense

INSERT INTO Expenses (

expense\_id,

user\_id,

category\_id,

amount,

expense\_date,

description

) VALUES (

p\_expense\_id,

p\_user\_id,

p\_category\_id,

p\_amount,

p\_expense\_date,

p\_description

);

END;

/

### 3. Procedure: create\_limit\_alert

This procedure creates an alert entry in the Expense\_Alerts table when an expense exceeds the predefined limit.

CREATE OR REPLACE PROCEDURE create\_limit\_alert(

p\_user\_id IN NUMBER,

p\_expense\_id IN NUMBER,

p\_limit\_amount IN NUMBER

) AS

BEGIN

INSERT INTO Expense\_Alerts (

user\_id,

expense\_id,

limit\_amount

) VALUES (

p\_user\_id,

p\_expense\_id,

p\_limit\_amount

);

END;

/

### 4. Function: get\_monthly\_summary

This function returns a cursor (i.e., a reference to a result set) containing a summary of expenses for each category for a specific month. It aggregates totals, counts, minimum, maximum, and average expense values.

CREATE OR REPLACE FUNCTION get\_monthly\_summary(

p\_user\_id IN NUMBER,

p\_month IN NUMBER,

p\_year IN NUMBER

) RETURN SYS\_REFCURSOR AS

v\_cursor SYS\_REFCURSOR;

v\_start\_date DATE;

v\_end\_date DATE;

BEGIN

-- Calculate start and end dates of the month

v\_start\_date := TO\_DATE(p\_year || '-' || p\_month || '-01', 'YYYY-MM-DD');

v\_end\_date := LAST\_DAY(v\_start\_date);

OPEN v\_cursor FOR

SELECT

c.category\_name,

NVL(SUM(e.amount), 0) as total\_amount,

COUNT(e.expense\_id) as transaction\_count,

MIN(e.amount) as min\_expense,

MAX(e.amount) as max\_expense,

NVL(AVG(e.amount), 0) as avg\_expense

FROM

Categories c

LEFT JOIN

Expenses e ON c.category\_id = e.category\_id

AND e.user\_id = p\_user\_id

AND e.expense\_date BETWEEN v\_start\_date AND v\_end\_date

WHERE

c.user\_id = p\_user\_id

GROUP BY

c.category\_name

ORDER BY

total\_amount DESC;

RETURN v\_cursor;

END;

/

### 5. Function: get\_expense\_trends

This function returns a cursor containing expense trend data over a series of months. It groups expense amounts by month and category, making it suitable for generating charts and trend analysis.

CREATE OR REPLACE FUNCTION get\_expense\_trends(

p\_user\_id IN NUMBER,

p\_months IN NUMBER

) RETURN SYS\_REFCURSOR AS

v\_cursor SYS\_REFCURSOR;

v\_end\_date DATE := LAST\_DAY(TRUNC(SYSDATE));

v\_start\_date DATE := ADD\_MONTHS(TRUNC(v\_end\_date, 'MM'), -p\_months+1);

BEGIN

OPEN v\_cursor FOR

SELECT

TO\_CHAR(TRUNC(e.expense\_date, 'MM'), 'YYYY-MM') as month,

c.category\_name,

SUM(e.amount) as total\_amount

FROM

Expenses e

JOIN

Categories c ON e.category\_id = c.category\_id

WHERE

e.user\_id = p\_user\_id

AND e.expense\_date BETWEEN v\_start\_date AND v\_end\_date

GROUP BY

TO\_CHAR(TRUNC(e.expense\_date, 'MM'), 'YYYY-MM'),

c.category\_name

ORDER BY

month, c.category\_name;

RETURN v\_cursor;

END;

# Python Code

### 1. Database Connection Function

**Purpose:** Creates and returns an Oracle database connection. This is used throughout your application to execute SQL commands.

def get\_db\_connection():

"""Create and return a database connection"""

conn = oracledb.connect(user=DB\_USER, password=DB\_PASSWORD, dsn=DB\_DSN)

return conn

### 2. User Login Route

**Purpose:** Handles user authentication by verifying the submitted email and password against the Users table. On success, it initializes session variables.

@app.route('/', methods=['GET', 'POST'])

def login():

if request.method == 'POST':

email = request.form['email']

password = request.form['password']

conn = get\_db\_connection()

cursor = conn.cursor()

cursor.execute('SELECT user\_id, name FROM Users WHERE email = :email AND password = :password',

{'email': email, 'password': password})

user = cursor.fetchone()

cursor.close()

conn.close()

if user:

session['user\_id'] = user[0]

session['name'] = user[1]

return redirect(url\_for('dashboard'))

else:

flash('Invalid credentials')

return render\_template('login.html')

### 3. Dashboard Route

**Purpose:** Queries recent expenses, calculates the total for the current month, breaks down expenses by category, and checks for unread alerts. It passes these statistics to the dashboard template.

@app.route('/dashboard')

def dashboard():

if 'user\_id' not in session:

return redirect(url\_for('login'))

conn = get\_db\_connection()

cursor = conn.cursor()

# Get recent expenses

cursor.execute('''

SELECT e.expense\_id, e.expense\_date, c.category\_name, e.amount, e.description

FROM Expenses e

JOIN Categories c ON e.category\_id = c.category\_id

WHERE e.user\_id = :user\_id

ORDER BY e.expense\_date DESC

''', {'user\_id': session['user\_id']})

expenses = cursor.fetchall()

# Get current month total

today = date.today()

first\_day = date(today.year, today.month, 1)

last\_day = date(today.year, today.month, calendar.monthrange(today.year, today.month)[1])

cursor.execute('''

SELECT NVL(SUM(amount), 0)

FROM Expenses

WHERE user\_id = :user\_id

AND expense\_date BETWEEN :first\_day AND :last\_day

''', {'user\_id': session['user\_id'], 'first\_day': first\_day, 'last\_day': last\_day})

monthly\_total = cursor.fetchone()[0]

# Get category breakdown for current month

cursor.execute('''

SELECT c.category\_name, NVL(SUM(e.amount), 0) as total

FROM Categories c

LEFT JOIN Expenses e ON c.category\_id = e.category\_id

AND e.expense\_date BETWEEN :first\_day AND :last\_day

WHERE c.user\_id = :user\_id

GROUP BY c.category\_name

ORDER BY total DESC

''', {'user\_id': session['user\_id'], 'first\_day': first\_day, 'last\_day': last\_day})

category\_totals = cursor.fetchall()

# Check for unread alerts

alert\_count = 0

try:

cursor.execute('''

SELECT COUNT(\*)

FROM Expense\_Alerts

WHERE user\_id = :user\_id AND is\_read = 0

''', {'user\_id': session['user\_id']})

alert\_count = cursor.fetchone()[0]

except oracledb.DatabaseError as e:

if 'ORA-00942' in str(e):

pass

else:

raise

cursor.close()

conn.close()

return render\_template('dashboard.html',

expenses=expenses,

monthly\_total=monthly\_total,

category\_totals=category\_totals,

alert\_count=alert\_count)

### 4. Adding a New Expense (with Limit Check)

**Purpose:** This route uses a PL/SQL procedure to insert a new expense while checking whether it exceeds a set limit. If it does, an alert is created.

@app.route('/add-expense', methods=['GET', 'POST'])

def add\_expense():

if 'user\_id' not in session:

return redirect(url\_for('login'))

conn = get\_db\_connection()

cursor = conn.cursor()

# Fetch categories for this user

cursor.execute('SELECT category\_id, category\_name FROM Categories WHERE user\_id = :user\_id',

{'user\_id': session['user\_id']})

categories = cursor.fetchall()

if request.method == 'POST':

category\_id = request.form['category\_id']

amount = float(request.form['amount'])

expense\_date = request.form['expense\_date']

description = request.form.get('description', '')

# Use PL/SQL procedure to add expense and check limits

limit\_exceeded = cursor.var(oracledb.NUMBER)

limit\_amount = cursor.var(oracledb.NUMBER)

expense\_id = cursor.var(oracledb.NUMBER)

cursor.callproc('add\_expense\_with\_limit\_check',

[session['user\_id'], category\_id, amount,

datetime.strptime(expense\_date, '%Y-%m-%d').date(),

description, limit\_exceeded, limit\_amount, expense\_id])

# If limit exceeded, create alert

if limit\_exceeded.getvalue() == 1:

cursor.callproc('create\_limit\_alert',

[session['user\_id'], expense\_id.getvalue(), limit\_amount.getvalue()])

flash(f'Expense added, but it exceeds your budget limit of ${limit\_amount.getvalue():.2f}!')

else:

flash('Expense added successfully!')

conn.commit()

cursor.close()

conn.close()

return redirect(url\_for('dashboard'))

cursor.close()

conn.close()

return render\_template('add\_expense.html', categories=categories, now=date.today().strftime('%Y-%m-%d'))

### 5. User Registration

**Purpose:** Checks for duplicate email registrations, inserts a new user, and automatically assigns default categories for new users.

@app.route('/register', methods=['GET', 'POST'])

def register():

if request.method == 'POST':

name = request.form['name']

email = request.form['email']

password = request.form['password']

conn = get\_db\_connection()

cursor = conn.cursor()

# Check if email already exists

cursor.execute('SELECT COUNT(\*) FROM Users WHERE email = :email', {'email': email})

if cursor.fetchone()[0] > 0:

flash('Email already registered')

cursor.close()

conn.close()

return render\_template('register.html')

# Register the user

cursor.execute('''

INSERT INTO Users (name, email, password)

VALUES (:name, :email, :password)

''', {'name': name, 'email': email, 'password': password})

# Get the new user\_id

cursor.execute('SELECT user\_id FROM Users WHERE email = :email', {'email': email})

user\_id = cursor.fetchone()[0]

# Create default categories for new user

default\_categories = ['Food', 'Housing', 'Transportation', 'Entertainment', 'Healthcare', 'Personal', 'Education', 'Other']

for category in default\_categories:

cursor.execute('''

INSERT INTO Categories (user\_id, category\_name)

VALUES (:user\_id, :category\_name)

''', {'user\_id': user\_id, 'category\_name': category})

conn.commit()

cursor.close()

conn.close()

flash('Registration successful! Please login.')

return redirect(url\_for('login'))

return render\_template('register.html')

### 6. API Endpoint for Dashboard Chart Data

**Purpose:** This endpoint returns JSON data for a chart showing monthly expense totals over the last six months.

@app.route('/api/dashboard/chart-data')

def dashboard\_chart\_data():

if 'user\_id' not in session:

return jsonify({'error': 'Not logged in'}), 401

conn = get\_db\_connection()

cursor = conn.cursor()

# Get data for the last 6 months

today = date.today()

start\_date = date(today.year - 1 if today.month <= 6 else today.year,

(today.month - 6) % 12 + 1, 1)

cursor.execute('''

SELECT TO\_CHAR(TRUNC(expense\_date, 'MM'), 'YYYY-MM') as month, SUM(amount) as total

FROM Expenses

WHERE user\_id = :user\_id AND expense\_date >= :start\_date

GROUP BY TO\_CHAR(TRUNC(expense\_date, 'MM'), 'YYYY-MM')

ORDER BY month

''', {'user\_id': session['user\_id'], 'start\_date': start\_date})

months = []

totals = []

for row in cursor:

months.append(row[0])

totals.append(float(row[1]))

cursor.close()

conn.close()

return jsonify({

'labels': months,

'datasets': [{

'label': 'Monthly Expenses',

'data': totals

}]

})

# References

**Database System Concepts**

Abraham Silberschatz, Henry F. Korth, S. Sudarshan. *Database System Concepts (Seventh Edition)*. McGraw-Hill Education.

**Oracle Documentation**

Oracle. *Oracle Database SQL and PL/SQL Language Reference*. Available at:<https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/index.html>.

**Flask Documentation**

Pallets Projects. *Flask Web Framework Official Documentation*. Available at: <https://flask.palletsprojects.com/>.

**Chart.js Documentation**

Chart.js Team. *Chart.js Documentation*. Available at: <https://www.chartjs.org/docs/latest/>.

**Bootstrap Official Documentation**

Bootstrap. *Bootstrap 5 Documentation*. Available at:<https://getbootstrap.com/>.

**W3Schools**

W3Schools. *HTML, CSS, and JavaScript Tutorials*. Available at:<https://www.w3schools.com/>.

**StackOverflow Community**

Various Authors. *StackOverflow Discussions on Oracle, Python, Flask, and Web Development*. Available at:<https://stackoverflow.com/>.

**Manipal Institute of Technology Laboratory Manual**

CSE 2241 Database Systems Laboratory Manual, January 2024 Edition.