BudgetApp

App Requirements

The various Minimum Viable Product features includes the following:

**User Management Includes:**

* *Signup***:**
  + Users can create an account with email + password.
  + Passwords stored securely (hashed, not plain text).
* *Login***:**
  + Users log in with email + password.
  + System returns a secure token (JWT) to keep them logged in.

**Transactions**

* *Add income/expense:*
  + Users enter: amount, type (income/expense), category, date, and optional note.
* *View transactions:*
  + Users see a list of their past income/expenses.
  + Transactions should be linked to the user account (each user sees only their own data).

**Categories**

* Foods & Drinks :

1. Fast Food
2. Restaurant, fast-food
3. Groceries

* Shopping:

The following are the categories involved :-

* Drug-store, chemist
* Free time
* Stationery, tools
* Gifts, joy
* Electronics, accessories
* Pets, animals
* Home, garden

1. Toilertries
2. Kitchen

* Kids
* Health and beauty
* Jewels, accessories
* Men's

1. Fragrances
2. Footwear
3. Clothes

* Housing

1. Energy nd Utilities

* Transport
* Vehicle
* Life & Entertainment

1. TV, Streaming
2. Activeness sport nd fitness
3. Holiday nd trips

* Communication and PC

1. Internet
2. Airtime
3. Bundles

* Financial Expenses

1. Charges & fees
2. Loans & intrests

* Investments

1. Trade
2. MMF
3. Savings

* Income
* Others

***Assign category:*** Every transaction must belong to one category.

**Monthly Summary**

* **Auto-summary:**
  + Show total income, total expenses, and balance for the month.
  + Breakdown by category (e.g., Food = 25% of expenses).

**Export Data**

* **CSV export:** Users can download their transaction history as a spreadsheet.
* **PDF export:** Optionally generate a simple PDF report of the monthly summary.

Tools

* **Python** (for backend with Flask or FastAPI)

Download: [python.org](https://www.python.org/downloads/)

* **Flutter SDK** (for frontend mobile app)

Download: flutter.dev

* **VS Code** (recommended editor for both Python + Flutter)

Extensions: Flutter, Python

* Android Studio
* **Git & GitHub account** (for version control)

Install Git: [git-scm.com](https://git-scm.com/downloads)

* **Postman** (for testing APIs easily).

Project Structure

**budget\_app\_backend/**

│── venv/ # Python virtual environment

│── app.py # Main Flask entry point

│── models.py # Database models (User, Transaction, Category)

│── routes.py # API routes (signup, login, transactions)

│── database.py # DB setup (SQLAlchemy)

│── requirements.txt # Python dependencies

│── README.md # Documentation for backend

**budget\_app\_frontend/**

│── lib/

│ ├── main.dart # Entry point

│ ├── pages/

│ │ ├── login\_page.dart # Login UI

│ │ ├── signup\_page.dart # Signup UI

│ │ ├── dashboard\_page.dart # Transaction list

│ │ └── add\_txn\_page.dart # Add income/expense

│ ├── services/

│ │ └── api\_service.dart # Connects Flutter to Flask API

│ ├── models/

│ │ └── transaction.dart # Transaction data model

│ └── state/

│ └── app\_state.dart # State management (Riverpod/Provider)

│── pubspec.yaml # Flutter dependencies

│── README.md # Documentation for frontend

🏗 Backend Development Tasks – Beginner Notes

## 1️⃣ Create Models with SQLAlchemy (models.py)

**Models** are like **blueprints for your database tables**. Each model defines the structure of a table.

* **User model**

class User(db.Model):

id = db.Column(db.Integer, primary\_key=True)

email = db.Column(db.String(120), unique=True, nullable=False)

password\_hash = db.Column(db.String(128), nullable=False)

* + id → unique identifier for each user
  + email → user’s email, must be unique
  + password\_hash → encrypted password
* **Category model**

class Category(db.Model):

id = db.Column(db.Integer, primary\_key=True)

name = db.Column(db.String(50), nullable=False)

* + Categories like “Food”, “Salary”, “Entertainment”
* **Transaction model**

class Transaction(db.Model):

id = db.Column(db.Integer, primary\_key=True)

user\_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)

amount = db.Column(db.Float, nullable=False)

type = db.Column(db.String(10), nullable=False) # income or expense

category\_id = db.Column(db.Integer, db.ForeignKey('category.id'))

date = db.Column(db.Date, nullable=False)

note = db.Column(db.String(200))

* + Links **transactions** to users and categories
  + Stores amount, type (income/expense), date, and optional note

✅ **Tip:** Think of models as the **skeleton of your database**. You define what each table should store here.

## 2️⃣ Set up Database Connection (database.py)

We need a **connection** so our app can talk to the database.

from flask\_sqlalchemy import SQLAlchemy

db = SQLAlchemy()

Then in your main app:

from flask import Flask

from database import db

app = Flask(\_\_name\_\_)

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///budget.db'

db.init\_app(app)

* SQLALCHEMY\_DATABASE\_URI → tells SQLAlchemy where the database is
* db.init\_app(app) → connects SQLAlchemy to your Flask app

✅ **Tip:** For beginners, **SQLite** is easiest because it’s just a single file.

## 3⃣ Apply Migrations or Auto-Create SQLite Database

* **Migrations** → track changes to database tables over time
* **For beginners:** You can auto-create tables quickly:

with app.app\_context():

db.create\_all()

* This reads your models and creates tables in budget.db automatically.

✅ **Tip:** Later, when your app grows, learn **Flask-Migrate** for proper migrations.

## ****Visual Map****

┌───────────────────┐

│ app.py │

│ │

│ - Creates Flask │

│ app instance │

│ - Configures DB │

│ - Registers routes│

└─────────┬─────────┘

│

│ imports db & models

▼

┌───────────────────┐

│ database.py │

│ │

│ - Creates SQLAlchemy │

│ instance (db) │

│ - Handles DB connection │

└─────────┬─────────┘

│

│ used by models

▼

┌───────────────────┐

│ models.py │

│ │

│ - Defines tables │

│ (User, Category,│

│ Transaction) │

│ - Uses db from │

│ database.py │

└───────────────────┘

### **Step-by-Step Explanation**

1. **app.py**
   * Main file that runs your Flask app
   * Imports db from database.py and models from models.py
   * Configures the database URI and initializes SQLAlchemy
2. **database.py**
   * Only job: create the SQLAlchemy **database instance (db)**
   * This instance is shared across the app and models
3. **models.py**
   * Defines **all database tables** using SQLAlchemy
   * Uses the db object from database.py to define columns, relationships, etc.

### ✅ **Analogy for beginners:**

Think of it like building a house:

* database.py → provides the **bricks and cement** (the database connection)
* models.py → draws the **blueprints** for each room (tables)
* app.py → is the **construction manager** that puts everything together and runs the house (app)

📌 Concept 1: **Model**

* A **Model** is a Python class that represents a table in the database.
* Example:

class Category(db.Model):

id = db.Column(db.Integer, primary\_key=True)

name = db.Column(db.String(100), nullable=False)

parent\_id = db.Column(db.Integer, db.ForeignKey('category.id'))

* Here, Category is the **model**, and it defines how the **categories table** looks.

📌 Concept 2: **Parent–Child Relationship (Self-Referencing Foreign Key)**

* A **Foreign Key** connects one row in a table to another row in the same or different table.
* In this case, a category can be a **parent** or a **child (subcategory)**.
* Example: "Foods & Drinks" (parent) → "Fast Food" (child).
* This makes your categories **hierarchical**.

📌 Concept 3: **Seeding (Database Seeding)**

* **Seeding** means adding initial data into the database automatically.
* We wrote seed\_categories.py to insert all categories and subcategories.
* This is common in new apps (to preload data like roles, admin user, categories, etc.).

📌 Concept 4: **App Context (Flask App Context)**

* Flask needs to know which app is running before using the database.
* We use:

with app.app\_context():

# database queries go here

* Without **App Context**, you’ll get an error: "Working outside of application context".

📌 Concept 5: **Querying the Database (SQLAlchemy Query)**

* To fetch data:

categories = Category.query.all()

.all() → gets all records

.first() → gets the first record

.filter\_by(name="Foods & Drinks").first() → gets a specific record

📌 Concept 6: **Primary Key (id)**

* Every row in the database needs a **unique identifier**.
* In our model, id = db.Column(db.Integer, primary\_key=True) means id is the **primary key**.

📌 Concept 7: **Foreign Key (parent\_id)**

* Connects one record to another.
* parent\_id links a subcategory to its parent.
* Example: "Fast Food" has parent\_id = 1, meaning its parent is "Foods & Drinks".

## How to verify

That your categories actually exist in the database. Let’s do it

📌 Step 1: Open Flask Shell

From your backend folder (where app.py is), run in terminal:

python

This opens the Python shell.  
(If you prefer, you can also use **Flask shell** if configured, but for beginners plain Python is fine.)

📌 Step 2: Import App and Database

Inside the shell:

from app import app

from database import db

from models import Category

📌 Step 3: Use App Context

Flask needs an **app context** to talk to the database. Enter:

with app.app\_context():

categories = Category.query.all()

for c in categories:

print(c.id, c.name, "-> Parent:", c.parent\_id)

📌 Step 4: Expected Output

You’ll see something like:

1 Foods & Drinks -> Parent: None

2 Fast Food -> Parent: 1

3 Restaurant, fast-food -> Parent: 1

4 Groceries -> Parent: 1

5 Shopping -> Parent: None

6 Drug-store, chemist -> Parent: 5

7 Free time -> Parent: 5

...

* Parent: None = main category
* Parent: [id] = subcategory belonging to that parent

📌 Step 5: Exit Shell

When you’re done, type:

exit()

**🔑 Authentication system**.

🔑 Authentication – Beginner Notes

## 1. Concept

* **Flask-JWT-Extended** → library for handling JSON Web Tokens (JWT).
* **Signup** → user registers with email + password.
* **Login** → user logs in, gets a JWT token.
* **Protected Routes** → JWT token must be sent in request headers.

## 2. Install Flask-JWT-Extended

pip install flask-jwt-extended

## 3. Update app.py

from flask\_jwt\_extended import JWTManager

# After app = Flask(\_\_name\_\_)

app.config['JWT\_SECRET\_KEY'] = 'your\_secret\_key\_here' # change to env variable later

jwt = JWTManager(app)

## 4. Create routes/auth.py

from flask import Blueprint, request, jsonify

from werkzeug.security import generate\_password\_hash, check\_password\_hash

from flask\_jwt\_extended import create\_access\_token

from database import db

from models import User

auth\_bp = Blueprint('auth', \_\_name\_\_)

# Signup endpoint

@auth\_bp.route('/auth/signup', methods=['POST'])

def signup():

data = request.get\_json()

email = data.get('email')

password = data.get('password')

if User.query.filter\_by(email=email).first():

return jsonify({"error": "Email already exists"}), 400

new\_user = User(

email=email,

password\_hash=generate\_password\_hash(password)

)

db.session.add(new\_user)

db.session.commit()

return jsonify({"message": "User created successfully"}), 201

# Login endpoint

@auth\_bp.route('/auth/login', methods=['POST'])

def login():

data = request.get\_json()

email = data.get('email')

password = data.get('password')

user = User.query.filter\_by(email=email).first()

if not user or not check\_password\_hash(user.password\_hash, password):

return jsonify({"error": "Invalid credentials"}), 401

# Create JWT token

access\_token = create\_access\_token(identity=user.id)

return jsonify(access\_token=access\_token), 200

## 5. Register Blueprint in app.py

from routes.auth import auth\_bp

app.register\_blueprint(auth\_bp)

## 6. Test with Postman

1. **Signup** → POST /auth/signup

{

"email": "test@example.com",

"password": "mypassword123"

}

1. **Login** → POST /auth/login
   * Get back {"access\_token": "..."}
2. Use token in headers for protected routes:
3. Authorization: Bearer <your\_token\_here>

### **Testing the Signup endpoint with Postman**,

#### ✅ Prerequisites

* Your Flask app is running (example: python app.py) and reachable at

http://127.0.0.1:5000 (or whichever host/port you use).

Flask-JWT-Extended, Flask, and Flask-SQLAlchemy are installed.

* auth blueprint (route POST /auth/signup) is registered in app.py.

#### 1. Start your server

Run from your backend folder:

python app.py

Confirm you can open http://127.0.0.1:5000/ and see your test message.

#### 2. Create a new request in Postman

1. Open Postman → **New** → **Request**.
2. Name it Signup and save to a collection (optional).

#### 3. Configure the request

* **Method:** POST
* **URL:** http://127.0.0.1:5000/auth/signup
* **Headers:**
  + Content-Type: application/json
* **Body:** choose **raw** → **JSON** and paste:

{

"email": "test@example.com",

"password": "mypassword123"

}

#### 4. Send the request

Click **Send**.

**Expected success response:**

* **Status code:** 201 Created
* **Response body:** e.g.

{

"message": "User created successfully"

}

If you see this, signup worked.

#### 5. Verify the user exists (two simple ways)

##### A — Using Python / Flask shell

Open Python in your project directory:

python

Then:

from app import app

from models import User

with app.app\_context():

user = User.query.filter\_by(email="test@example.com").first()

print(user, user.id, user.email)

If you get a User object, the record is in the DB.

##### B — Using DB Browser for SQLite

* Open budget.db in DB Browser for SQLite and browse the user table to confirm the row exists.

#### 6. Common errors & fixes

* **400 Bad Request / Email already exists**
  + Cause: trying to sign up with an email already in the DB.
  + Fix: use a different email or delete the existing record.
* **500 Internal Server Error**
  + Cause: server crash or missing expected fields.
  + Fix: check Flask console logs; ensure your route reads email and password from JSON and db is configured.
* **'sqlite3' is not recognized** (Windows)
  + Cause: sqlite3 CLI not in PATH — only relevant if you try to use sqlite CLI.
  + Fix: install SQLite and add sqlite3.exe to PATH (we covered this earlier).
* **No response / connection refused**
  + Cause: server not running or wrong port.
  + Fix: ensure python app.py is running and URL/port match.

#### 7. (Optional) Add a quick Postman Test to assert success

In Postman, open the **Tests** tab and paste:

pm.test("Status is 201", function () {

pm.response.to.have.status(201);

});

pm.test("Message created", function () {

var json = pm.response.json();

pm.expect(json.message).to.eql("User created successfully");

});

This will show pass/fail in Postman test results.

#### 8. Next step — Login

After signup, test login to get a JWT:

* Endpoint: POST /auth/login
* Body:

{ "email": "test@example.com", "password": "mypassword123" }

Expected: 200 OK and { "access\_token": "..." }

You’ll use that token for protected routes:

* Add header: Authorization: Bearer <access\_token>

#### 9. Useful concepts to search later (keywords)

* **Flask-JWT-Extended**
* **Werkzeug generate\_password\_hash / check\_password\_hash**
* **POST request JSON Content-Type**
* **Flask blueprints**
* **Postman collections & tests**
* **SQLAlchemy querying** (Model.query.filter\_by(...))
* **HTTP status codes: 201, 400, 401, 500**

👉 By finishing this, you’ll have:  
✔ User signup & login system  
✔ Secure password hashing  
✔ JWT authentication for protected routes

🔄 Transactions API – Step by Step

## 1. Concept

* Each **Transaction** belongs to a user (via user\_id).
* Endpoints must be **JWT-protected** → only logged-in users can create/view their own transactions.
* We’ll use a **blueprint** for better organization (routes/transactions.py).

## 2. Create routes/transactions.py

from flask import Blueprint, request, jsonify

from flask\_jwt\_extended import jwt\_required, get\_jwt\_identity

from database import db

from models import Transaction

# Create blueprint

transactions\_bp = Blueprint('transactions', \_\_name\_\_)

# GET /transactions → fetch all transactions for current user

@transactions\_bp.route('/transactions', methods=['GET'])

@jwt\_required()

def get\_transactions():

user\_id = get\_jwt\_identity() # get logged-in user ID from JWT

transactions = Transaction.query.filter\_by(user\_id=user\_id).all()

# Convert results to list of dictionaries

result = []

for t in transactions:

result.append({

"id": t.id,

"amount": t.amount,

"type": t.type,

"category\_id": t.category\_id,

"date": t.date.isoformat() if t.date else None,

"note": t.note

})

return jsonify(result), 200

# POST /transactions → create a new transaction

@transactions\_bp.route('/transactions', methods=['POST'])

@jwt\_required()

def create\_transaction():

user\_id = get\_jwt\_identity()

data = request.get\_json()

new\_transaction = Transaction(

user\_id=user\_id,

amount=data['amount'],

type=data['type'], # "income" or "expense"

category\_id=data['category\_id'],

date=data['date'], # string e.g. "2025-09-10"

note=data.get('note', "")

)

db.session.add(new\_transaction)

db.session.commit()

return jsonify({"message": "Transaction added successfully!"}), 201

## 3. Register blueprint in app.py

In app.py, after registering auth\_bp, add:

from routes.transactions import transactions\_bp

app.register\_blueprint(transactions\_bp)

## 4. Test with Postman

#### (A) Login first

* POST /auth/login with JSON body:

{

"email": "test@example.com",

"password": "mypassword123"

}

* Copy the access\_token from response.

#### (B) Add a transaction

* POST /transactions
* Headers:
  + Content-Type: application/json
  + Authorization: Bearer <your\_token\_here>
* Body:

{

"amount": 1500,

"type": "expense",

"category\_id": 2,

"date": "2025-09-10",

"note": "Bought groceries"

}

**Expected Response:**

{

"message": "Transaction added successfully!"

}

#### (C) Fetch transactions

* GET /transactions
* Headers:
  + Authorization: Bearer <your\_token\_here>

**Expected Response:**

[

{

"id": 1,

"amount": 1500,

"type": "expense",

"category\_id": 2,

"date": "2025-09-10",

"note": "Bought groceries"

}

]

👉 By finishing this, you now have:  
✔ Add transaction  
✔ View transactions (per-user)  
✔ JWT-protected routes

Ah thanks for catching that 🙌 — let’s make this step **learnable notes style**, so when you look back you’ll understand what you did and why.

📘 Notes: API Documentation with Flasgger (Swagger UI)

## ✨ Concept: API Documentation

* **API Documentation** explains how your backend endpoints work.
* Instead of writing plain text, you use **Swagger/OpenAPI** so you (and others) can interactively test your API in the browser.

### ⚙️ Tool: Flasgger

* **Flasgger** = a Flask extension that integrates Swagger UI.
* Swagger UI shows your endpoints in a web page at /apidocs/.
* This is great for debugging, learning, and sharing with teammates.

#### 🛠 How to Install

Run in your backend environment:

pip install flasgger

#### 🛠 How to Configure in app.py

Add this near the top:

from flasgger import Swagger

Then after initializing Flask:

app = Flask(\_\_name\_\_)

swagger = Swagger(app) # enable Swagger UI

Now you can open:

http://127.0.0.1:5000/apidocs/

to see documentation.

### 📝 Concept: YAML Docstrings

* You “document” each API route by writing a **docstring** (triple quotes """) above it.
* The docstring uses **YAML format** (structured text) to describe:
  + **tags** (group endpoints together)
  + **parameters** (what input the endpoint expects)
  + **responses** (possible outcomes, e.g. success or error)

### 📍 Example: Signup Route

@auth\_bp.route('/auth/signup', methods=['POST'])

def signup():

"""

User Signup

---

tags:

- Authentication

parameters:

- in: body

name: body

required: true

schema:

properties:

email:

type: string

password:

type: string

responses:

200:

description: User successfully created

400:

description: User already exists

"""

# logic here...

### 📍 Example: Transaction Route

@transactions\_bp.route('/transactions', methods=['POST'])

@jwt\_required()

def create\_transaction():

"""

Create a new transaction

---

tags:

- Transactions

security:

- Bearer: [] # requires JWT token

parameters:

- in: body

name: body

required: true

schema:

properties:

amount:

type: number

type:

type: string

enum: [income, expense]

category\_id:

type: integer

date:

type: string

example: "2025-09-10"

note:

type: string

responses:

200:

description: Transaction created successfully

"""

# logic here...

### Auto-inject the token into Swagger UI

Great 👍 Let’s make Swagger UI automatically include the **JWT token** after login so you don’t have to paste it every time.

#### 1. Add securityDefinitions to your Swagger config

In your app.py (where you set up Swagger), update the Swagger template like this:

from flasgger import Swagger

swagger\_template = {

"swagger": "2.0",

"info": {

"title": "Budget App API",

"description": "API documentation for the Budget App",

"version": "1.0.0"

},

"securityDefinitions": {

"Bearer": {

"type": "apiKey",

"name": "Authorization",

"in": "header",

"description": "JWT Authorization header using the Bearer scheme. Example: 'Bearer {token}'"

}

},

"security": [

{"Bearer": []}

]

}

swagger = Swagger(app, template=swagger\_template)

#### 2. Update your protected endpoints to include security

In your transactions.py (and any other protected route docs), add:

@transactions\_bp.route('/transactions', methods=['GET'])

@jwt\_required()

def get\_transactions():

"""

Get Transactions

---

tags:

- Transactions

security:

- Bearer: []

responses:

200:

description: List of all transactions for the current user

401:

description: Unauthorized (JWT missing/invalid)

"""

# your logic...

#### 3. Workflow in Swagger UI

1. First, **login** via /auth/login.
   * Copy the access\_token value.
2. At the top right of Swagger UI, click the **Authorize 🔒 button**.
3. Paste the token as:
4. Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
5. Click **Authorize**, then close.

Now every subsequent request (like GET /transactions) will **auto-attach the token** in the Authorization header.

### 🔑 Rule of Thumb

* Keep your **existing function code** exactly as it is.
* Just **add the docstring (triple quotes + YAML block)** right below the def function\_name(): line.
* Flasgger reads these docstrings automatically when you visit /apidocs/.

### ✅ Summary

* **Flasgger** lets you add Swagger docs to Flask.
* Add it to app.py → swagger = Swagger(app).
* Write **YAML docstrings** above routes to describe them.
* Open /apidocs/ to test interactively.

#### 🔑 Authentication Endpoints

* **POST /auth/signup** → Register a new user
* **POST /auth/login** → Authenticate user & return JWT

#### 💸 Transaction Endpoints

* **GET /transactions** → Get all transactions for the authenticated user (requires Authorization: Bearer <token>)
* **POST /transactions** → Create a new transaction (requires Authorization: Bearer <token>)