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>)

Perfect way to close the day ✅

Here’s a **concise update for your backend README.md** plus a **journal entry** you can drop into your progress tracker:

## 📖 Backend README.md Update

### 🔑 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>)

## 📝 Journal Entry

**Progress**

* Added **Authentication routes** (/auth/signup, /auth/login) with JWT integration.
* Added **Transactions routes** (GET /transactions, POST /transactions).
* Integrated Swagger docs with security: Bearer [] so JWT auth works in Swagger UI.

**Issues Faced**

* Subject must be a string → fixed by casting user ID to str in create\_access\_token.
* Swagger showing Missing Authorization Header → solved by ensuring security: Bearer [] was included in POST docs.
* Blueprint/indentation errors → corrected Python indentation and ensured unique blueprint names.

**Next Steps**

* Expand transaction model with categories, timestamps.
* Add update/delete transaction endpoints.
* Start connecting frontend to test auth + transactions flow.

Flutter UI

## 🏗 Main App & Navigation

### 📘 **Note:**

* This is the entry point of your app.
* MaterialApp holds all pages (called **routes**).
* initialRoute decides which page opens first.
* Navigator.pushNamed() moves forward to another page.
* Navigator.pop() goes back.

### 💻 **Code (main.dart):**

import 'package:flutter/material.dart';

import 'pages/login\_page.dart';

import 'pages/signup\_page.dart';

import 'pages/dashboard\_page.dart';

import 'pages/add\_transaction\_page.dart';

void main() {

runApp(const BudgetApp());

}

class BudgetApp extends StatelessWidget {

const BudgetApp({super.key});

@override

Widget build(BuildContext context) {

return MaterialApp(

title: "Budget App",

theme: ThemeData(primarySwatch: Colors.blue),

// App starts with the Login page

initialRoute: '/login',

// Routes map (page name → screen widget)

routes: {

'/login': (context) => const LoginPage(),

'/signup': (context) => const SignupPage(),

'/dashboard': (context) => const DashboardPage(),

'/add-transaction': (context) => const AddTransactionPage(),

},

);

}

}

## 🔐 Login Page

### 📘 **Note:**

* This page has **2 text fields** (email + password).
* A **Login button** → goes to Dashboard.
* A **Signup link** → goes to Signup page.

### 💻 **Code (login\_page.dart):**

import 'package:flutter/material.dart';

class LoginPage extends StatelessWidget {

const LoginPage({super.key});

@override

Widget build(BuildContext context) {

final emailController = TextEditingController();

final passwordController = TextEditingController();

return Scaffold(

appBar: AppBar(title: const Text("Login")),

body: Padding(

padding: const EdgeInsets.all(16.0),

child: Column(

mainAxisAlignment: MainAxisAlignment.center,

children: [

// Email field

TextField(

controller: emailController,

decoration: const InputDecoration(labelText: "Email"),

),

// Password field

TextField(

controller: passwordController,

decoration: const InputDecoration(labelText: "Password"),

obscureText: true, // hides password text

),

const SizedBox(height: 20),

// Login button → goes to Dashboard

ElevatedButton(

onPressed: () {

Navigator.pushNamed(context, '/dashboard');

},

child: const Text("Login"),

),

// Link → goes to Signup page

TextButton(

onPressed: () {

Navigator.pushNamed(context, '/signup');

},

child: const Text("Don't have an account? Sign up"),

)

],

),

),

);

}

}

## 📝 Signup Page

### 📘 **Note:**

* Same as login, but button is **Signup**.
* After signing up, you go **back to Login page**.

### 💻 **Code (signup\_page.dart):**

import 'package:flutter/material.dart';

class SignupPage extends StatelessWidget {

const SignupPage({super.key});

@override

Widget build(BuildContext context) {

final emailController = TextEditingController();

final passwordController = TextEditingController();

return Scaffold(

appBar: AppBar(title: const Text("Signup")),

body: Padding(

padding: const EdgeInsets.all(16.0),

child: Column(

mainAxisAlignment: MainAxisAlignment.center,

children: [

TextField(

controller: emailController,

decoration: const InputDecoration(labelText: "Email"),

),

TextField(

controller: passwordController,

decoration: const InputDecoration(labelText: "Password"),

obscureText: true,

),

const SizedBox(height: 20),

// Signup button → goes back to Login

ElevatedButton(

onPressed: () {

Navigator.pushReplacementNamed(context, '/login');

},

child: const Text("Signup"),

),

// Back to Login

TextButton(

onPressed: () {

Navigator.pop(context);

},

child: const Text("Already have an account? Login"),

)

],

),

),

);

}

}

## 📊 Dashboard Page

### 📘 **Note:**

* Shows **balance** and **list of transactions**.
* Uses a **mock list** (static data).
* Floating button ➕ → opens Add Transaction page.

### 💻 **Code (dashboard\_page.dart):**

import 'package:flutter/material.dart';

class DashboardPage extends StatelessWidget {

const DashboardPage({super.key});

// Mock static transactions

final List<Map<String, dynamic>> mockTransactions = const [

{"type": "Income", "amount": 5000, "category": "Salary", "date": "2025-09-01"},

{"type": "Expense", "amount": 1500, "category": "Groceries", "date": "2025-09-05"},

{"type": "Expense", "amount": 800, "category": "Transport", "date": "2025-09-07"},

];

@override

Widget build(BuildContext context) {

// Calculate balance

double balance = mockTransactions.fold(

0,

(sum, tx) => tx["type"] == "Income"

? sum + tx["amount"]

: sum - tx["amount"],

);

return Scaffold(

appBar: AppBar(title: const Text("Dashboard")),

body: Column(

children: [

// Balance summary

Container(

padding: const EdgeInsets.all(16),

color: Colors.blue.shade100,

child: Text("Balance: \$${balance.toStringAsFixed(2)}",

style: const TextStyle(fontSize: 20, fontWeight: FontWeight.bold)),

),

// List of transactions

Expanded(

child: ListView.builder(

itemCount: mockTransactions.length,

itemBuilder: (context, index) {

final tx = mockTransactions[index];

return ListTile(

title: Text("${tx['category']} - \$${tx['amount']}"),

subtitle: Text("${tx['date']} (${tx['type']})"),

);

},

),

)

],

),

// Floating Action Button → Add Transaction

floatingActionButton: FloatingActionButton(

onPressed: () {

Navigator.pushNamed(context, '/add-transaction');

},

child: const Icon(Icons.add),

),

);

}

}

## ➕ Add Transaction Page

### 📘 **Note:**

* Form with **Amount, Type, Category, Note, Date**.
* Save button → just **prints data to console** for now.
* Later, we’ll send this to backend.

### 💻 **Code (add\_transaction\_page.dart):**

import 'package:flutter/material.dart';

class AddTransactionPage extends StatefulWidget {

const AddTransactionPage({super.key});

@override

State<AddTransactionPage> createState() => \_AddTransactionPageState();

}

class \_AddTransactionPageState extends State<AddTransactionPage> {

final amountController = TextEditingController();

String type = "Income"; // Default selection

final categoryController = TextEditingController();

final noteController = TextEditingController();

DateTime selectedDate = DateTime.now();

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(title: const Text("Add Transaction")),

body: Padding(

padding: const EdgeInsets.all(16.0),

child: Column(

children: [

// Amount input

TextField(

controller: amountController,

decoration: const InputDecoration(labelText: "Amount"),

keyboardType: TextInputType.number,

),

// Dropdown for type

DropdownButton<String>(

value: type,

items: const [

DropdownMenuItem(value: "Income", child: Text("Income")),

DropdownMenuItem(value: "Expense", child: Text("Expense")),

],

onChanged: (value) {

setState(() {

type = value!;

});

},

),

// Category

TextField(

controller: categoryController,

decoration: const InputDecoration(labelText: "Category"),

),

// Note

TextField(

controller: noteController,

decoration: const InputDecoration(labelText: "Note"),

),

const SizedBox(height: 10),

// Date Picker

Row(

children: [

Text("Date: ${selectedDate.toLocal()}".split(' ')[0]),

const Spacer(),

ElevatedButton(

onPressed: () async {

final picked = await showDatePicker(

context: context,

initialDate: selectedDate,

firstDate: DateTime(2020),

lastDate: DateTime(2030),

);

if (picked != null && picked != selectedDate) {

setState(() {

selectedDate = picked;

});

}

},

child: const Text("Select Date"),

),

],

),

const SizedBox(height: 20),

// Save button → prints data

ElevatedButton(

onPressed: () {

print("Transaction Added: "

"Amount: ${amountController.text}, "

"Type: $type, "

"Category: ${categoryController.text}, "

"Note: ${noteController.text}, "

"Date: $selectedDate");

Navigator.pop(context);

},

child: const Text("Save"),

)

],

),

),

);

}

}

Great! Let’s expand the notes with **extra beginner explanations** + **simple diagrams** so your future self can read and immediately understand the app flow and testing concepts.

# 📘 Beginner Notes for Testing in Flutter (Expanded with Diagrams)

## 🧪 1. What is Testing in Flutter?

Testing = making sure your app works as expected **before** you release it.

* **Why test?**
  + Prevents bugs before users see them.
  + Helps when changing code → you can quickly confirm nothing else broke.
  + Builds confidence in your project.

## 🧩 2. Types of Flutter Tests

### ✅ Unit Test

* Smallest test: checks a single function.
* **Example:**
* int add(int a, int b) => a + b;
* void main() {
* test('add function adds two numbers', () {
* expect(add(2, 3), 5);
* });
* }
* Tests **logic only**, no UI.

📊 **Diagram (Unit Test)**

[Function] ---> [Expected Result?] ---> Pass/Fail

### ✅ Widget Test

* Tests widgets (buttons, text, input fields).
* Runs in a lightweight environment.

**Example:**

testWidgets('Counter increments test', (WidgetTester tester) async {

await tester.pumpWidget(const BudgetApp());

expect(find.text('0'), findsOneWidget);

await tester.tap(find.byIcon(Icons.add));

await tester.pump();

expect(find.text('1'), findsOneWidget);

});

📊 **Diagram (Widget Test)**

[Widget] ---> [Simulate Tap/Text Input] ---> [Check UI Updates]

### ✅ Integration Test

* Tests the **whole app flow** (like a real user).
* Example scenario:
  + Open app → Login → See Dashboard → Add Transaction.

📊 **Diagram (Integration Test)**

[User Action Flow]

Login → Dashboard → Add Transaction → Confirm Balance Updated

## 📦 3. Testing Packages

* flutter\_test → already included, for unit + widget tests.
* integration\_test → for app-wide tests.
* mockito → simulate APIs or databases.

## 📝 4. Running Tests

* Run all tests:
* flutter test
* Run a specific file:
* flutter test test/widget\_test.dart
* With detailed logs:
* flutter test -v

# ⚠️ Errors Faced & Fixes

## 1. ****The name 'MyApp' isn’t a class****

* **Cause**: The test file used MyApp, but your root widget is BudgetApp.
* **Fix**: Change:
* await tester.pumpWidget(const MyApp());

→ to:

await tester.pumpWidget(const BudgetApp());

## 2. ****pubspec.yaml Errors****

* **Duplicate dependencies: key**
  + ✅ Keep only **one dependencies block**.
* **Bad indentation**
  + ✅ Fix:
  + dependencies:
  + flutter:
  + sdk: flutter
  + flutter\_riverpod: ^2.5.1
  + cupertino\_icons: ^1.0.8

📊 **Diagram**

dependencies:

flutter: <-- correct spacing

sdk: flutter

## 3. ****Unused Import Warning****

* Cause: Imported flutter\_riverpod before using it.
* Fix: Either use it or remove the import temporarily.

## 4. ****Transaction Model Errors****

* Tried using title, but Transaction has no title.
* Fix: Use existing fields: type, amount, category, date, note.

## 5. ****Const Widget Hot Reload Error****

* Error: Const class cannot remove fields.
* Fix: Use **Hot Restart** instead of Hot Reload when changing widget fields.

📊 **Diagram**

Hot Reload ✅ → UI tweaks

Hot Restart ✅ → Adding/removing fields

## 6. ****Git Errors****

* **fatal: 'origin' not a repository**
  + Fix: Add remote
  + git remote add origin <repo-url>
* **Rejected push (fetch first)**
  + Fix:
  + git pull origin main --rebase
  + git push origin main

# 🔄 Budget App UI Flow Diagram

Here’s a simple **visual flow of your app**:

[Login Page]

| (Sign Up button)

v

[Signup Page] <-- back

|

v

[Dashboard Page]

|

+--> Shows: Balance + Transactions

|

+--> (➕ Button)

v

[Add Transaction Page]

# ✅ Key Beginner Takeaways

1. Match your widget name in tests (BudgetApp, not MyApp).
2. Always check **pubspec.yaml indentation**.
3. Update model + UI consistently.
4. Use **Hot Restart** when modifying fields.
5. Git workflow:
   * git pull origin main --rebase
   * Fix conflicts
   * git push origin main

Got it ✅ — let’s make this **beginner-friendly notes style**, same way we did for testing.

# Adding **Dio** (API Calls Setup)

## What is Dio? 🤔

* **Dio** is a powerful **HTTP client** for Flutter/Dart.
* It helps your app talk to your backend (Flask in our case).
* Think of it as the **postman inside your app**: it sends requests (GET, POST, PUT, DELETE) and receives responses.

## Step 1: Add Dio to your project 📦

1. Open your pubspec.yaml file.
2. Under **dependencies**, add this line:

dependencies:

dio: ^5.7.0 # latest version as of 2025

1. Save the file and run:

flutter pub get

👉 This installs Dio so you can start using it.

## Step 2: Create a reusable API Client 🛠️

* Instead of writing Dio everywhere, we’ll create **one central class** for handling API calls.
* Create a new file:

lib/services/api\_client.dart

Paste this:

import 'package:dio/dio.dart';

class ApiClient {

final Dio \_dio = Dio(

BaseOptions(

baseUrl: "http://127.0.0.1:5000", // Flask backend URL

connectTimeout: const Duration(seconds: 10),

receiveTimeout: const Duration(seconds: 10),

headers: {

"Content-Type": "application/json",

},

),

);

Dio get client => \_dio;

}

## Step 3: Test your setup 🧪

Later when you have endpoints in Flask, you can test like this:

import 'services/api\_client.dart';

void fetchExample() async {

final api = ApiClient().client;

final response = await api.get("/transactions"); // Example endpoint

print(response.data);

}

### 🔹 Step 1: Confirm Flask Endpoints

Let’s say in your Flask backend you have something like:

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

def get\_transactions():

return jsonify([

{"type": "Income", "amount": 5000, "category": "Salary", "date": "2025-09-01"},

{"type": "Expense", "amount": 1500, "category": "Groceries", "date": "2025-09-05"}

])

That means you can access this by visiting:  
👉 http://127.0.0.1:5000/transactions

### 🔹 Step 2: Call this endpoint in Flutter using Dio

In Flutter, inside your services/api\_client.dart, you already have ApiClient.

Now you just need a **service function** to fetch the transactions:

import 'package:dio/dio.dart';

import '../models/transaction.dart';

import 'api\_client.dart';

class TransactionService {

final Dio \_dio = ApiClient().client;

Future<List<Transaction>> getTransactions() async {

try {

final response = await \_dio.get("/transactions");

// Map response JSON into Transaction objects

List<Transaction> transactions = (response.data as List)

.map((json) => Transaction.fromJson(json))

.toList();

return transactions;

} catch (e) {

throw Exception("Failed to load transactions: $e");

}

}

}

### 🔹 Step 3: Connect with Riverpod (later)

Once you’re ready, you’ll create a **provider** that uses the TransactionService:

import 'package:flutter\_riverpod/flutter\_riverpod.dart';

import '../services/transaction\_service.dart';

import '../models/transaction.dart';

final transactionServiceProvider = Provider((ref) => TransactionService());

final transactionsProvider = FutureProvider<List<Transaction>>((ref) async {

final service = ref.read(transactionServiceProvider);

return await service.getTransactions();

});

Now in your UI:

Consumer(

builder: (context, ref, child) {

final transactions = ref.watch(transactionsProvider);

return transactions.when(

data: (data) => ListView.builder(

itemCount: data.length,

itemBuilder: (context, index) {

final t = data[index];

return ListTile(

title: Text(t.category),

subtitle: Text("${t.type} - ${t.amount}"),

);

},

),

loading: () => CircularProgressIndicator(),

error: (e, \_) => Text("Error: $e"),

);

},

)

### 🔹 Diagram 🖼️ (Future State)

Flutter UI

↓

Riverpod (transactionsProvider)

↓

TransactionService (uses Dio)

↓

ApiClient (baseUrl + config)

↓

Flask Backend (/transactions endpoint)

↓

Database

✅ **Summary**:

1. Setup Dio now (we did).
2. Later → write a **service class** per feature (e.g., TransactionService).
3. Connect service to **Riverpod providers**.
4. UI listens to providers and displays data.

Would you like me to **already prepare a skeleton TransactionService + provider setup** (commented out) so you’ll only need to “uncomment and tweak” once Flask is running?

## Diagram 🖼️ (Simple Flow)

Flutter App ----> Dio (ApiClient) ----> Flask Backend ----> Database

* Your **app** calls Dio
* Dio uses the **base URL** (http://127.0.0.1:5000)
* Dio sends request to **Flask backend**
* Flask talks to **Database** and sends response back

## Key Notes 📝

* ✅ **Setup only** → no real API calls yet, just structure.
* ✅ Base URL is set → change later if backend is on a server.
* ✅ One central class (ApiClient) → easier to maintain.
* ✅ No need to use Riverpod yet (we’ll do that when connecting frontend to backend).

Great step 👌 — adding **flutter\_secure\_storage** now will prepare your app to securely store JWT tokens later when you implement login/signup with your Flask backend.

I’ll give you beginner-friendly notes + code setup.

🔐 Adding flutter\_secure\_storage

* 🔑 **What it does**: Think of it as a **safe/locker inside the phone** where you can store sensitive information (like JWT tokens, API keys, or passwords).
* 🛡️ Why not SharedPreferences? → Because SharedPreferences stores data in plain text (not secure). flutter\_secure\_storage encrypts data before saving.
* 📱 Works on Android & iOS, and even supports web with some limitations.

Later, when a user logs in and the backend sends back a **JWT token**, you’ll **store it securely**. Every time you need to call a protected API, you’ll fetch it from this secure storage and attach it to your API headers.

## ⚙️ Step 1: Install the package

Run in terminal:

flutter pub add flutter\_secure\_storage

## ⚙️ Step 2: Create a Token Storage Helper

📂 Create a new file: lib/services/token\_storage.dart

import 'package:flutter\_secure\_storage/flutter\_secure\_storage.dart';

class TokenStorage {

// Create a secure storage instance

final \_storage = const FlutterSecureStorage();

// Save JWT token

Future<void> saveToken(String token) async {

await \_storage.write(key: "jwt\_token", value: token);

}

// Read JWT token

Future<String?> getToken() async {

return await \_storage.read(key: "jwt\_token");

}

// Delete JWT token (logout)

Future<void> deleteToken() async {

await \_storage.delete(key: "jwt\_token");

}

}

## ⚙️ Step 3: Usage Later (Example)

When user **logs in** and gets a token from Flask:

final tokenStorage = TokenStorage();

// Save the token

await tokenStorage.saveToken("eyJhbGciOiJIUzI1...");

// Get token later

String? token = await tokenStorage.getToken();

print("Saved JWT token: $token");

// Delete on logout

await tokenStorage.deleteToken();

### 1. Inside your **Login Page** (after API call succeeds)

When you call Flask login API (using Dio), you’ll get back a JWT token. At that point:

// Assume Dio login response gives you token in response.data['token']

final tokenStorage = TokenStorage();

await tokenStorage.saveToken(response.data['token']);

👉 This saves the token securely on the device.

### 2. Inside your **API Service Layer**

When making future requests with Dio (e.g., /transactions), you’ll need the token. That’s where you’ll do:

final tokenStorage = TokenStorage();

String? token = await tokenStorage.getToken();

final response = await dio.get(

"/transactions",

options: Options(

headers: {"Authorization": "Bearer $token"},

),

);

### 3. Inside your **Logout Button**

When the user logs out, you’ll just clear it:

final tokenStorage = TokenStorage();

await tokenStorage.deleteToken();

📌 **For Now (Setup Only)**:

* You only needed to **create the token\_storage.dart file**.
* You don’t need to call saveToken() or getToken() yet.
* When your **login/signup feature is working**, that’s where you’ll add this code.

## 📊 Diagram

Login API (Flask) → returns JWT token

↓

Flutter app (Dio)

↓

TokenStorage (flutter\_secure\_storage)

↓

Secure Encrypted Storage on device

✅ **Summary for Setup Only (Now)**

* Installed flutter\_secure\_storage ✅
* Created TokenStorage helper ✅
* Not using it yet → will plug it in once **login/signup API** is working.

Perfect! Let’s write a **journal entry** for Day 3 in a **beginner-friendly, reflective style**. You can add this to a Journal.md file in your project or in your documentation folder.

# 📔 Budget App – Day 3 Journal Entry

**Date:** 2025-09-19  
**Goal:** Build the basic Flutter UI with mock data (no backend connection yet).

## 1️⃣ Pages Created Today

|  |  |
| --- | --- |
| **Page** | **Description** |
| **Login Page** | Simple login form with **Email** and **Password** fields, **Login button**, and **navigation to Signup page**. |
| **Signup Page** | Form with **Email** and **Password**, **Signup button**, and **navigation back to Login page**. |
| **Dashboard Page** | Displays **mock transactions** in a list, **balance summary** at the top, and a **Floating Action Button** (➕) to add a transaction. |
| **Add Transaction Page** | Form to enter **amount, type (income/expense), category, date, note**. Save button prints input to console. |

## 2️⃣ Design Challenges Faced

1. **Navigation between pages**
   * Learned how to use **initialRoute and routes** in MaterialApp.
   * Challenge: remembering the correct route names (/login, /signup, /dashboard, /add-transaction).
2. **Mock Data in Dashboard**
   * Needed to calculate **balance** by adding incomes and subtracting expenses.
   * Challenge: making sure the list updates when new transactions are added (for now, just printing to console).
3. **State Management Preparation**
   * Added **Riverpod** for future state management, but didn’t use it fully yet.
   * Learned that you need to structure **providers first**, then integrate them later with real data.
4. **Flutter Widget Testing**
   * Initial widget test failed because the root widget in main.dart is BudgetApp (not MyApp).
   * Learned that **widget test class names must match actual app widget names**.
5. **Dependencies & Pubspec.yaml**
   * Several errors with **duplicate dependencies** and **bad indentation**.
   * Learned that YAML is very sensitive to **spaces and indentation**.
6. **Hot Reload vs Hot Restart**
   * Hot reload **doesn’t work** when changing widget fields in const constructors.
   * Learned to use **Hot Restart** for structural changes.
7. **Git Workflow**
   * Encountered **push rejected** errors because remote had changes.
   * Learned to always **pull first, resolve conflicts, then push**.

## 3️⃣ Key Takeaways

* **Beginner-friendly approach**: Build UI first, mock data, then integrate backend.
* **Documentation** is essential: writing down errors + fixes helps avoid repeating mistakes.
* **State management setup** early (Riverpod) prepares for API integration later.
* **Secure storage & API client** setup now, so integrating login/auth later is easier.