1

January 30, 2025

1 Personal Finance Tracker & Budgeting Assistant Project Report

Github
Website

`Important note: when opening the website for the first time, it may take ~50 seconds for it to load. This is because render might be in sleep mode. After the 50 seconds are up the website will function normally.`

This project fully implements a personal finance tracker as a web application using Python and the Flask framework, ensuring that users can access its functionality through a browser. A basic form is presented to capture user transactions, while reporting features are available on the dashboard to summarize total income, expenses, and other financial metrics. The application meets its data collection requirement through a simple process of submitting information such as the transaction amount, description, and category (income or expense) via a form, storing these records securely in the SQLite database. A data analyzer is then integrated into the dashboard to aggregate these transactions, calculate net balances, and offer an elementary predictive analysis feature that estimates future expenses based on historical data.

To guarantee that the application remains stable and well-structured, unit tests have been created to verify functions such as user registration, login, and basic database interactions. For integration testing, the Flask test client provides a way to simulate user workflows end-to-end, such as going from the login page to the dashboard and adding a transaction, thereby ensuring that all components operate correctly together. Data persistence is handled by an SQLite database, which was chosen for its simplicity and minimal configuration overhead, but the application's design allows for easily swapping out the data store if needed. RESTful collaboration is accommodated via endpoints that could be extended for internal or external API usage, ensuring that other services (or a future frontend) can interact with the transaction data directly in JSON format.

With regard to the product environment and deployment, the application can be containerized or set up in a virtual environment, and it is prepared to integrate into continuous integration pipelines such as GitHub Actions. In that pipeline, the application's tests are automatically executed, verifying that every new commit or pull request maintains the code's reliability. Continuous delivery is thus supported by building upon these automated checks: once the tests pass, the application can be deployed to a staging or production environment with minimal manual intervention. The project also includes a basis for production monitoring and instrumentation by using Python's logging capabilities and the readiness to incorporate monitoring tools like New Relic or OpenTelemetry. It likewise provides a mechanism for event collaboration messaging by sending alerts (in the current version, flashing messages in the UI) whenever the application detects scenarios such as net spending surpassing a threshold. Although these alerts are presently demonstrated in a simple manner,

the architecture allows for expansion toward more sophisticated messaging systems like Kafka or RabbitMQ should the application require real-time notifications or distributed event handling.

1.1 Code

1.1.1 app.py

App.py contains all the core functionality of the Flask application, including the configuration for the database, the definition of models for users and transactions, the routes responsible for handling user registration and login, the main dashboard view, as well as utility functions for user authentication. It also initializes the database tables if they do not exist yet, thus providing everything needed to run the application from a single entry point.

[ ]:
```
from flask import Flask, render_template, request, redirect, url_for, session,␣
 ↪flash
from flask_sqlalchemy import SQLAlchemy
import datetime
from flask_bcrypt import Bcrypt

# --------------------------------------------------------------------------
# 1. CONFIG
# --------------------------------------------------------------------------
app = Flask(__name__)
bcrypt = Bcrypt(app)
app.config['SECRET_KEY'] = '123'  # Replace with a more secure key
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///finance.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

db = SQLAlchemy(app)

# --------------------------------------------------------------------------
# 2. DATABASE MODELS
# --------------------------------------------------------------------------
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(200), nullable=False)

    def set_password(self, password):
        self.password_hash = bcrypt.generate_password_hash(password).
 ↪decode('utf-8')

    def check_password(self, password):
        return bcrypt.check_password_hash(self.password_hash.encode('utf-8'),␣
 ↪password.encode('utf-8'))


class Transaction(db.Model):
```

```
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)
    date = db.Column(db.Date, default=datetime.date.today)
    description = db.Column(db.String(200), nullable=True)
    amount = db.Column(db.Float, nullable=False)
    category = db.Column(db.String(50), nullable=True)  # e.g. "income",␣
 ↪"expense"

    user = db.relationship('User', backref=db.backref('transactions',␣
 ↪lazy=True))

# -----------------------------------------------------------------------------
# 3. UTILITY FUNCTIONS
# -----------------------------------------------------------------------------
def current_user():
    """
    Returns the current logged-in user object
    based on session['user_id'].
    """
    if 'user_id' in session:
        return db.session.get(User, session['user_id'])
    return None


def login_required(func):
    """
    Simple decorator to ensure a user is logged in.
    """
    def wrapper(*args, **kwargs):
        if not current_user():
            flash("Please log in to access this page.", "warning")
            return redirect(url_for('login'))
        return func(*args, **kwargs)
    wrapper.__name__ = func.__name__  # to avoid issues with Flask's route
    return wrapper

# -----------------------------------------------------------------------------
# 4. ROUTES
# -----------------------------------------------------------------------------

@app.route('/')
def home():
    return render_template('layout.html')  # A simple homepage or landing

@app.route('/register', methods=['GET', 'POST'])
def register():
    if request.method == 'POST':
```

```
        email = request.form.get('email')
        password = request.form.get('password')

        # Check if user exists
        if User.query.filter_by(email=email).first():
            flash("Email already registered. Please log in.", "danger")
            return redirect(url_for('login'))

        new_user = User(email=email)
        new_user.set_password(password)
        db.session.add(new_user)
        db.session.commit()

        flash("Registration successful! You can now log in.", "success")
        return redirect(url_for('login'))

    return render_template('register.html')


@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        email = request.form.get('email')
        password = request.form.get('password')
        user = User.query.filter_by(email=email).first()

        if user and user.check_password(password):
            session['user_id'] = user.id
            flash("Logged in successfully!", "success")
            return redirect(url_for('dashboard'))
        else:
            flash("Invalid credentials.", "danger")
            return redirect(url_for('login'))

    return render_template('login.html')


@app.route('/logout')
def logout():
    session.pop('user_id', None)
    flash("Logged out.", "info")
    return redirect(url_for('login'))


@app.route('/dashboard')
@login_required
def dashboard():
```

```
    user = current_user()

    # Fetch all transactions for the logged-in user
    transactions = Transaction.query.filter_by(user_id=user.id).all()

    # Simple analytics
    total_income = sum(t.amount for t in transactions if t.category == 'income')
    total_expense = sum(t.amount for t in transactions if t.category ==␣
↪'expense')
    net = total_income - total_expense

    # Example: naive predictive analytics
    # If the user has X average monthly expense, let's guess at next month's
    # We'll keep it ultra-simple: average monthly = total_expenses / # months␣
↪(hard-coded to 1 if no data)
    # Real logic would group by months/dates.
    months_count = 1
    if len(transactions) > 0:
        first_date = min(t.date for t in transactions)
        now = datetime.date.today()
        months_count = max((now.year - first_date.year)*12 + (now.month -␣
↪first_date.month), 1)
    avg_monthly_expense = total_expense / months_count

    predicted_next_month_expense = round(avg_monthly_expense, 2)

    # Event-driven notifications example
    # If net < 0, you might want to notify user
    if net < 0:
        # For demonstration, just flash a message
        flash("Warning: You have a negative net balance this period!",␣
↪"warning")

    return render_template('dashboard.html',
                           transactions=transactions,
                           total_income=total_income,
                           total_expense=total_expense,
                           net=net,
                          ␣
↪predicted_next_month_expense=predicted_next_month_expense)


@app.route('/add_transaction', methods=['POST'])
@login_required
def add_transaction():
    user = current_user()
    amount = float(request.form.get('amount', 0))
```

```
    category = request.form.get('category')  # "income" or "expense"
    description = request.form.get('description')

    new_txn = Transaction(
        user_id=user.id,
        date=datetime.date.today(),
        description=description,
        amount=amount,
        category=category
    )
    db.session.add(new_txn)
    db.session.commit()

    flash("Transaction added successfully!", "success")
    return redirect(url_for('dashboard'))

# ------------------------------------------------------------------------------
# 5. DB INIT
# ------------------------------------------------------------------------------
def create_tables():
    """
    Create the database tables if they don't exist yet.
    """
    with app.app_context():
        db.create_all()

# ------------------------------------------------------------------------------
# 6. RUN (for development)
# ------------------------------------------------------------------------------
if __name__ == '__main__':
    # In production, use a WSGI server (gunicorn, etc.) and not debug mode.
    create_tables()
    app.run(debug=True)
```

1.1.2 layout.html

The layout.html template serves as the base layout for the application's HTML pages, defining a consistent navigation bar, the overall structure of the page, and a space for displaying any flashed messages. All other templates extend this file, ensuring a cohesive and uniform style across the website.

[ ]:
```
<!DOCTYPE html>
<html>
<head>
    <title>My Finance Tracker</title>
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.
 ↪5.2/css/bootstrap.min.css">
```

```
</head>
<body>

<nav class="navbar navbar-expand-lg navbar-light bg-light">
  <a class="navbar-brand" href="{{ url_for('home') }}">Finance Tracker</a>
  <div class="collapse navbar-collapse">
    <ul class="navbar-nav ml-auto">
      {% if session['user_id'] %}
        <li class="nav-item">
          <a class="nav-link" href="{{ url_for('dashboard') }}">Dashboard</a>
        </li>
        <li class="nav-item">
          <a class="nav-link" href="{{ url_for('logout') }}">Logout</a>
        </li>
      {% else %}
        <li class="nav-item">
          <a class="nav-link" href="{{ url_for('login') }}">Login</a>
        </li>
        <li class="nav-item">
          <a class="nav-link" href="{{ url_for('register') }}">Register</a>
        </li>
      {% endif %}
    </ul>
  </div>
</nav>

<div class="container mt-4">
    {% with messages = get_flashed_messages(with_categories=true) %}
        {% if messages %}
            {% for category, message in messages %}
                <div class="alert alert-{{ category }}" role="alert">
                    {{ message }}
                </div>
            {% endfor %}
        {% endif %}
    {% endwith %}

    {% block content %}{% endblock %}
</div>

</body>
</html>
```

1.1.3 register.html

The register.html file contains a simple form to allow users to sign up for the service by providing their email address and a password. Upon submitting this form, the user's credentials are validated and added to the database before redirecting to the login page.

[ ]:
```
{% extends "layout.html" %}
{% block content %}
<h2>Register</h2>
<form method="POST">
    <div class="form-group">
        <label>Email:</label>
        <input type="email" name="email" class="form-control" required/>
    </div>
    <div class="form-group">
        <label>Password:</label>
        <input type="password" name="password" class="form-control" required/>
    </div>
    <button type="submit" class="btn btn-primary">Sign Up</button>
</form>
{% endblock %}
```

1.1.4 login.html

Similarly to register.html, login.html includes a user-friendly form for existing users to enter their credentials. Upon successful authentication, they are granted access to their personal dashboard.

[ ]:
```
{% extends "layout.html" %}
{% block content %}
<h2>Login</h2>
<form method="POST">
    <div class="form-group">
        <label>Email:</label>
        <input type="email" name="email" class="form-control" required/>
    </div>
    <div class="form-group">
        <label>Password:</label>
        <input type="password" name="password" class="form-control" required/>
    </div>
    <button type="submit" class="btn btn-primary">Log In</button>
</form>
{% endblock %}
```

1.1.5 dashboard.html

The dashboard.html file is responsible for displaying a user's transaction data, such as income, expenses, and net balance, and provides a form for adding new transactions. It also features a section that shows a rudimentary predictive calculation of future expenses. This file stands as the primary interface through which users can track their financial activity and view immediate feedback in the form of basic analytical insights.

[ ]:
```
{% extends "layout.html" %}
{% block content %}
<h2>Dashboard</h2>
```

```
<p>Total Income: {{ total_income }}</p>
<p>Total Expenses: {{ total_expense }}</p>
<p>Net: {{ net }}</p>
<p>Predicted Next Month Expense: {{ predicted_next_month_expense }}</p>

<h3>Add Transaction</h3>
<form method="POST" action="{{ url_for('add_transaction') }}">
    <div class="form-group">
        <label>Amount:</label>
        <input type="number" step="0.01" name="amount" class="form-control"␣
 ↪required/>
    </div>
    <div class="form-group">
        <label>Category:</label>
        <select name="category" class="form-control">
            <option value="income">Income</option>
            <option value="expense">Expense</option>
        </select>
    </div>
    <div class="form-group">
        <label>Description:</label>
        <input type="text" name="description" class="form-control"/>
    </div>
    <button type="submit" class="btn btn-success">Add Transaction</button>
</form>

<hr/>

<h3>Transaction History</h3>
<table class="table">
    <thead>
        <tr>
            <th>Date</th>
            <th>Description</th>
            <th>Category</th>
            <th>Amount</th>
        </tr>
    </thead>
    <tbody>
    {% for txn in transactions %}
        <tr>
            <td>{{ txn.date }}</td>
            <td>{{ txn.description }}</td>
            <td>{{ txn.category }}</td>
            <td>{{ txn.amount }}</td>
        </tr>
    {% endfor %}
```

```
    </tbody>
</table>
{% endblock %}
```

1.1.6 tests/test_app.py

The tests/test_app.py file houses a set of tests that verify the application's correctness and reliability. These include unit tests for functions like user registration and login, along with integration-style checks that ensure each step—from creating an account to reaching the dashboard—functions properly as a coherent workflow. This test suite serves as the foundation for maintaining code quality and protecting against regressions as new features are developed.

[ ]:
```python
import unittest
from app import app, db, User

class TestFinanceApp(unittest.TestCase):

    def setUp(self):
        # Configure the app for testing
        app.config['TESTING'] = True
        app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///test_finance.db'
        self.client = app.test_client()

        with app.app_context():
            db.create_all()

    def tearDown(self):
        with app.app_context():
            db.drop_all()

    def test_register_and_login(self):
        # Register user
        response = self.client.post('/register', data={
            'email': 'test@example.com',
            'password': 'password123'
        }, follow_redirects=True)

        self.assertIn(b'Registration successful', response.data)

        # Login user
        response = self.client.post('/login', data={
            'email': 'test@example.com',
            'password': 'password123'
        }, follow_redirects=True)

        self.assertIn(b'Logged in successfully!', response.data)
```

```
    def test_dashboard_requires_login(self):
        response = self.client.get('/dashboard', follow_redirects=True)
        self.assertIn(b'Please log in to access this page.', response.data)

if __name__ == '__main__':
    unittest.main()
```