

Личный финансовый учёт в веб-приложении на Flask и SQLite

Ран Максим Александрович

Приамурский государственный университет имени Шолом-Алейхема

студент

Аннотация

В данной работе рассматривается разработка веб-сервиса для личного финансового учёта, реализованного на языке программирования Python с использованием фреймворка Flask и встроенной СУБД SQLite. Приложение позволяет пользователю фиксировать доходы и расходы, группировать их по категориям, анализировать структуру трат и динамику изменения баланса во времени. Хранение данных организовано изолированно для каждого пользователя на основе механизма сессий, при этом идентификатор пользователя (`user_id`) присутствует во всех основных сущностях. На стороне сервера реализовано REST-подобное API для работы с пользователями, категориями и финансовыми операциями, а также специализированные маршруты для формирования агрегированной статистики. Клиентская часть включает единый интерфейс с формами авторизации, таблицей операций и графиками, построенными с помощью библиотеки Chart.js. Показано, что связка Flask + SQLite + простой фронтенд позволяет создать компактное и функциональное приложение, пригодное как для личного использования, так и для учебных целей.

Ключевые слова: личный финансовый учёт, Flask, Python, SQLite, веб-приложения, REST API, визуализация данных, Chart.js.

Personal finance tracking in a Flask and SQLite-based web application

Ran Maksim Aleksandrovich

Sholom-Aleichem Priamursky State University

Student

Abstract

This paper presents the development of a personal finance tracking web service implemented in Python using the Flask framework and the embedded SQLite database. The application enables users to record income and expenses, group operations by customizable categories, and analyze both spending structure and balance dynamics over time. Data storage is organized in an isolated manner for each user based on session management, with the user identifier (`user_id`) present in all core entities. The server side exposes a REST-like API for handling users, categories, and financial operations, as well as dedicated endpoints for aggregated statistics. The client side consists of a single-page interface combining authentication forms, a transaction table, and charts rendered using the Chart.js

library. It is shown that a minimal stack (Flask + SQLite + a simple frontend) is sufficient to build a compact yet functional application suitable for both personal use and educational purposes.

Keywords: personal finance tracking, web application, Flask, Python, SQLite, REST API, data visualization, Chart.js.

1 Введение

1.1 Актуальность

В условиях роста стоимости жизни и высокой нагрузки на домохозяйства умение планировать и контролировать личные финансы приобретает особую значимость. Отслеживание регулярных доходов и расходов, анализ структуры трат и динамики баланса позволяют своевременно выявлять «финансовые утечки», формировать резервные накопления и принимать более обоснованные решения в области потребления и кредитования.

Широкое распространение безналичных и электронных платежей облегчает проведение операций, но одновременно усложняет их восприятие: значительная часть транзакций распределена по различным сервисам и банковским приложениям. В результате без использования инструментов учёта пользователю становится сложно оценивать реальную картину своих финансовых потоков.

Существующие коммерческие и бесплатные решения для личного финансового учёта нередко являются закрытыми системами, не позволяющими гибко модифицировать логику работы и структуру хранимых данных. Для учебных задач и небольших проектов востребованы компактные, прозрачные в реализации приложения, которые можно локально развернуть, изучить и расширить под конкретные нужды.

1.2 Обзор исследований

Анализ литературы показывает, что фреймворк Flask активно используется как лёгкий и удобный инструмент для интеграции моделей машинного обучения в веб-приложения различного назначения.

В работе Н. К. К. Раджу, К. Кришнамурти, Б. П. Бхагавада и Н. Шанкара [1] Flask применяется для построения веб-интерфейса системы прогнозирования диабета. Авторы сравнивают несколько алгоритмов машинного обучения (SVM, KNN, логистическая регрессия, случайный лес) и интегрируют наиболее точную модель в веб-сервис на Flask. Тем самым Flask выступает связующим звеном между обученной моделью и пользователем: через веб-форму вводятся медицинские параметры, а серверная часть на Flask обрабатывает запрос, взаимодействует с моделью и возвращает прогноз риска диабета. В исследовании о А. Нат и Ш. Нема [2] Flask используется как платформа для визуализации результатов кластеризации и предсказания классов на эталонных наборах данных UCI (IRIS, Wine, Seed). В работе реализован веб-сервис, позволяющий запускать методы кластеризации (K-means, иерархическая кластеризация), уменьшения размерности (PCA, LDA),

а также классификаторы (логистическая регрессия, KNN, SVM, наивный Байес, деревья решений, случайный лес). Flask здесь обеспечивает интерактивную среду: пользователю доступны визуализации кластеров и результаты классификации непосредственно через браузер, без необходимости работать с кодом напрямую. В работе С. Кумара Бозе, Ш. Гош и С. Дас [3] предложен GUI-ориентированный подход к прогнозированию сердечных заболеваний. Графический интерфейс создаётся с использованием HTML и CSS, а Flask API отвечает за серверную логику: приём данных о пациенте, передачу их в модели машинного обучения и возврат результата. Таким образом, Flask выполняет роль «прослойки» между пользовательским интерфейсом и алгоритмами ML, превращая набор моделей в полноценное прикладное веб-приложение для раннего выявления сердечных патологий. Работа А. Ягантисваруду [4] расширяет этот подход до мультидисциплинарной системы: с помощью Flask API объединяются несколько моделей, предсказывающих вероятность различных заболеваний (диабет, диабетическая ретинопатия, сердечные заболевания, рак молочной железы). Модели реализованы с использованием TensorFlow и сохраняются/загружаются через механизм pickling. Flask в данном случае обеспечивает унифицированный API, через который клиентские приложения или медицинские системы могут передавать набор параметров и получать прогноз сразу по нескольким нозологиям. Это подчёркивает пригодность Flask для построения многомодульных медицинских экспертных систем. Наконец, в исследовании А. Лакшманарао, М. Раджа Бабу, М.М. Бала Кришна [5] Flask используется в иной предметной области – при обнаружении вредоносных URL-адресов. Авторы комбинируют методы обработки естественного языка (NLP) и алгоритмы машинного обучения для классификации ссылок как безопасных или вредоносных, а Flask обеспечивает развёртывание этой модели в виде веб-сервиса. Пользователь или другая система передают URL через интерфейс Flask-приложения, после чего на сервере выполняется предобработка, извлечение признаков и принятие решения моделью. Это демонстрирует, что Flask удобен не только в медицине, но и в задачах кибербезопасности.

1.3 Цель исследования

Целью данной работы является разработка и описание небольшого веб-приложения для личного финансового учёта на основе Flask и SQLite, обеспечивающего:

- регистрацию и аутентификацию пользователей;
- ведение справочника категорий доходов и расходов;
- добавление и просмотр финансовых операций с фильтрацией по датам и категориям;
- построение наглядных графиков структуры трат и динамики баланса.

2 Материалы и методы

Разрабатываемое веб-приложение использует следующий набор технологий:

- Python – основной язык программирования;
- Flask – лёгкий веб-фреймворк, обеспечивающий маршрутизацию, обработку HTTP-запросов и работу с шаблонами;
- SQLite – встраиваемая реляционная СУБД, представленная одним файлом `finance.db` в корневом каталоге проекта;
- HTML/CSS – разметка и оформление пользовательского интерфейса, в том числе тёмная тема, сетка карточек и таблиц;
- JavaScript – реализация клиентской логики взаимодействия с REST-подобным API и динамического обновления интерфейса;
- Chart.js – библиотека для построения графиков, используемая для визуализации структуры расходов по категориям и временных рядов баланса.

Выбор данного стека обусловлен его доступностью, простотой настройки и отсутствием необходимости в развёртывании отдельного сервера баз данных. Это делает решение удобным для локального использования и учебных демонстраций.

3 Результаты и обсуждения

Начнём исследование с создания файла `app.py`, в котором размещён основной код веб-приложения личного финансового учёта, а также логика взаимодействия с базой данных и клиентской частью.

Python-часть – обработка запросов и финансовых операций

Инициализация приложения и базы данных.

В начале файла создаётся экземпляр Flask-приложения, задаётся секретный ключ `secret_key`, а также путь к файлу базы данных `finance.db` (рис.1).

```
import os
import sqlite3
from datetime import datetime, date, timedelta
from pathlib import Path
from typing import Any, Dict, List, Optional

from flask import Flask, g, jsonify, render_template, request, session
from werkzeug.security import check_password_hash, generate_password_hash

DB_PATH = Path(__file__).with_name("finance.db")

app = Flask(__name__)
app.secret_key = os.getenv("SECRET_KEY", "dev-secret-change-me")
```

Рисунок 1. Код

Функция `get_db()` отвечает за установку соединения с SQLite, включает поддержку внешних ключей и сохраняет соединение в объекте `g` (рис.2).

```
def get_db() -> sqlite3.Connection:
    if "db" not in g:
        conn = sqlite3.connect(DB_PATH)
        conn.row_factory = sqlite3.Row
        conn.execute("PRAGMA foreign_keys = ON;")
        g.db = conn
    return g.db
```

Рисунок 2. Код

В функции `init_db()` при первом запуске формируются таблицы `users`, `categories` и `operations`, а также создаётся уникальный индекс на пару (`user_id`, `name`) в таблице категорий, что предотвращает дублирование названий категорий у одного и того же пользователя (рис.3).

```
def init_db() -> None:
    db = get_db()
    db.executescript(
        """
        CREATE TABLE IF NOT EXISTS users (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            email TEXT NOT NULL UNIQUE,
            password_hash TEXT NOT NULL,
            created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
        );

        CREATE TABLE IF NOT EXISTS categories (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            user_id INTEGER NOT NULL,
            name TEXT NOT NULL,
            kind TEXT NOT NULL CHECK (kind IN ('income', 'expense')),
            FOREIGN KEY (user_id) REFERENCES users (id) ON DELETE CASCADE
        );

        CREATE TABLE IF NOT EXISTS operations (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            user_id INTEGER NOT NULL,
            category_id INTEGER NOT NULL,
            amount REAL NOT NULL,
            description TEXT,
            occurred_at DATE NOT NULL,
            created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
            FOREIGN KEY (category_id) REFERENCES categories (id) ON DELETE CASCADE,
            FOREIGN KEY (user_id) REFERENCES users (id) ON DELETE CASCADE
        );

        ensure_column(db, "categories", "user_id", "INTEGER NOT NULL DEFAULT 1")
        ensure_column(db, "operations", "user_id", "INTEGER NOT NULL DEFAULT 1")
        db.executescript(
            """
            CREATE UNIQUE INDEX IF NOT EXISTS idx_categories_user_name
            ON categories(user_id, name);
            """
        )
    db.commit()
```

Рисунок 3. Код

Управление соединением и сессией пользователя.

Обработчик `@app.teardown_appcontext` закрывает соединение с базой данных по завершении обработки запроса (рис.4).

```
@app.teardown_appcontext
def close_db(error: Optional[BaseException]) -> None:
    db = g.pop("db", None)
    if db is not None:
        db.close()
```

Рисунок 4. Код

В функции `ensure_db()`, вызываемой через `@app.before_request`, выполняется инициализация схемы БД и попытка загрузить текущего пользователя по идентификатору `user_id`, сохранённому в сессии. При успешном поиске данные о пользователе помещаются в `g.user`, в противном случае идентификатор удаляется из сессии, что предотвращает работу с «несуществующим» аккаунтом (рис.5).

```
@app.before_request
def ensure_db():
    init_db()
    g.user = None
    user_id = session.get("user_id")
    if user_id:
        row = (
            get_db()
            .execute("SELECT id, email, created_at FROM users WHERE id = ?", (user_id,))
            .fetchone()
        )
        if row:
            g.user = row
        else:
            session.pop("user_id", None)
```

Рисунок 5. Код

Маршруты аутентификации пользователей.

Группа маршрутов `/api/auth/*` реализует регистрацию, вход и выход из системы.

- В запросе `POST /api/auth/register` сервер принимает e-mail и пароль, проверяет их заполненность, создаёт запись в таблице `users` с хешированным паролем и сохраняет `user_id` в сессии (рис.6).

```
@app.route("/api/auth/register", methods=["POST"])
def register():
    payload = request.get_json(force=True, silent=True) or {}
    email = (payload.get("email") or "").strip().lower()
    password = payload.get("password") or ""
    if not email or not password:
        return jsonify({"error": "email and password are required"}), 400
    db = get_db()
    try:
        cur = db.execute(
            "INSERT INTO users (email, password_hash) VALUES (?, ?)",
            (email, generate_password_hash(password)),
        )
        db.commit()
    except sqlite3.IntegrityError:
        return jsonify({"error": "user already exists"}), 409
    user_id = cur.lastrowid
    session["user_id"] = user_id
    seed_user_categories(user_id)
    return jsonify({"user": {"id": user_id, "email": email}})
```

Рисунок 6. Код

- После создания аккаунта вызывается функция `seed_user_categories()`, добавляющая базовые категории доходов и расходов (зарплата, фриланс, продукты, транспорт и др.) (рис.7).

```
def seed_user_categories(user_id: int) -> None:
    db = get_db()
    seed_categories = [
        ("Зарплата", "income"),
        ("Фриланс", "income"),
        ("Продукты", "expense"),
        ("Транспорт", "expense"),
        ("Жилье", "expense"),
        ("Развлечения", "expense"),
    ]
    for name, kind in seed_categories:
        try:
            db.execute(
                "INSERT INTO categories (user_id, name, kind) VALUES (?, ?, ?)",
                (user_id, name, kind),
            )
        except sqlite3.IntegrityError:
            continue
    db.commit()
```

Рисунок 7. Код

- В запросе POST /api/auth/login производится поиск пользователя по e-mail и проверка введенного пароля с использованием check_password_hash (рис.8). В случае успеха в сессию записывается идентификатор пользователя; при ошибке возвращается сообщение о неверных учетных данных.

```
@app.route("/api/auth/login", methods=["POST"])
def login():
    payload = request.get_json(force=True, silent=True) or {}
    email = (payload.get("email") or "").strip().lower()
    password = payload.get("password") or ""
    db = get_db()
    user = db.execute(
        "SELECT id, email, password_hash FROM users WHERE email = ?", (email,)
    ).fetchone()
    if not user or not check_password_hash(user["password_hash"], password):
        return jsonify({"error": "invalid credentials"}), 401
    session["user_id"] = user["id"]
    return jsonify({"user": {"id": user["id"], "email": user["email"]}})
```

Рисунок 8. Код

- Маршрут POST /api/auth/logout очищает сессию, а GET /api/auth/me позволяет фронтенду получить информацию о текущем авторизованном пользователе или ошибку, если сеанс не установлен (рис.9).

```
@app.route("/api/auth/logout", methods=["POST"])
def logout():
    session.pop("user_id", None)
    return jsonify({"ok": True})

@app.route("/api/auth/me")
def me():
    if not g.user:
        return jsonify({"error": "not authenticated"}), 401
    return jsonify({"user": {"id": g.user["id"], "email": g.user["email"]}})
```

Рисунок 9. Код

Маршрут категорий – работа со справочником доходов и расходов.

Маршрут /api/categories предназначен для управления пользовательскими категориями (рис.10):

```
@app.route("/api/categories", methods=["GET", "POST"])
def categories():
    if (resp := require_auth()) is not None:
        return resp
    db = get_db()
    if request.method == "POST":
        payload = request.get_json(force=True, silent=True) or {}
        name = (payload.get("name") or "").strip()
        kind = payload.get("kind")
        if not name or kind not in {"income", "expense"}:
            return jsonify({"error": "name and valid kind are required"}), 400
        try:
            cur = db.execute(
                "INSERT INTO categories (user_id, name, kind) VALUES (?, ?, ?)",
                (g.user["id"], name, kind),
            )
            db.commit()
            return jsonify({"id": cur.lastrowid, "name": name, "kind": kind}), 201
        except sqlite3.IntegrityError:
            return jsonify({"error": "category with this name already exists"}), 409

    cur = db.execute(
        "SELECT id, name, kind FROM categories WHERE user_id = ? ORDER BY kind, name",
        (g.user["id"],),
    )
    return jsonify({"categories": [dict(row) for row in cur.fetchall()]})
```

Рисунок 10. Код

- Метод GET возвращает список категорий текущего пользователя, отсортированных по типу (income/expense) и названию.
- Метод POST позволяет добавить новую категорию. Перед вставкой данные валидируются: проверяется непустое название и корректность типа (kind ∈ {income, expense}). В случае конфликта по уникальному индексу (одинаковое имя категории у одного пользователя) сервер возвращает сообщение об ошибке, что предотвращает появление дубликатов.

Маршрут операций – регистрация и просмотр финансовых записей.

Маршрут /api/operations обеспечивает основной функционал учёта операций (рис.11-12):

```

@app.route("/api/operations", methods=["GET", "POST"])
def operations():
    if (resp := require_auth()) is not None:
        return resp
    db = get_db()
    if request.method == "POST":
        payload = request.get_json(force=True, silent=True) or {}
        category_id = payload.get("category_id")
        amount = payload.get("amount")
        description = (payload.get("description") or "").strip()
        occurred_at_raw = payload.get("occurred_at")

        try:
            category_id = int(category_id)
            amount = float(amount)
        except (TypeError, ValueError):
            return jsonify({"error": "category_id and amount are required"}), 400

        occurred_at = parse_date(
            occurred_at_raw, default=date.today()
        )
        if occurred_at is None:
            return jsonify({"error": "invalid occurred_at format"}), 400

        cat = db.execute(
            "SELECT kind FROM categories WHERE id = ? AND user_id = ?",
            (category_id, g.user["id"]),
        ).fetchone()
        if not cat:
            return jsonify({"error": "category not found"}), 404

        signed_amount = abs(amount)
        if cat["kind"] == "expense":
            signed_amount = -signed_amount

        cur = db.execute(
            """
            INSERT INTO operations (user_id, category_id, amount, description, occurred_at)
            VALUES (?, ?, ?, ?, ?)
            """,
            (g.user["id"], category_id, signed_amount, description, occurred_at.isoformat()),
        )
        db.commit()

```

Рисунок 11. Код

```

op = db.execute(
    """
    SELECT o.*, c.name AS category, c.kind
    FROM operations o
    JOIN categories c ON c.id = o.category_id
    WHERE o.id = ? AND o.user_id = ?
    """,
    (cur.lastrowid, g.user["id"]),
).fetchone()
return jsonify({"operation": serialize_operation(op)}), 201

start = parse_date(request.args.get("start"), None)
end = parse_date(request.args.get("end"), None)
category_id = request.args.get("category_id")
filters: List[str] = []
params: List[Any] = []
if start:
    filters.append("o.occurred_at >= ?")
    params.append(start.isoformat())
if end:
    filters.append("o.occurred_at <= ?")
    params.append(end.isoformat())
if category_id:
    filters.append("o.category_id = ?")
    params.append(category_id)

where_clause = f"WHERE {' AND '.join(filters)}" if filters else ""
cur = db.execute(
    f"""
    SELECT o.*, c.name AS category, c.kind
    FROM operations o
    JOIN categories c ON c.id = o.category_id
    WHERE o.user_id = ?
    {('AND ' + ' AND '.join(filters)) if filters else ''}
    ORDER BY o.occurred_at DESC, o.created_at DESC
    """,
    [g.user["id"]] + params,
)
return jsonify({"operations": [serialize_operation(row) for row in cur.fetchall()]})

```

Рисунок 12. Код

- При POST запросе сервер принимает идентификатор категории, сумму, описание и дату совершения. Сумма переводится в абсолютное значение, после чего для расходов записывается с отрицательным знаком, а для доходов – с положительным. Такой подход упрощает последующие расчёты баланса. Дата операции парсится функцией `parse_date()` с поддержкой значения по умолчанию (текущая дата). Перед вставкой в таблицу `operations` выполняется проверка, что указанная категория принадлежит текущему пользователю.
- При GET запросе реализован выбор операций с фильтрацией по интервалу дат (`start`, `end`) и по конкретной категории (`category_id`). На основе переданных параметров формируется динамическое условие `WHERE` в SQL-запросе. Результаты сортируются по дате совершения и времени создания записи, что позволяет отображать сначала наиболее свежие операции.

Маршруты статистики – агрегированные данные по расходам и балансу.

Для формирования графиков и аналитики реализованы два специальных маршрута:

- GET `/api/stats/spending` агрегирует расходы по категориям за выбранный период (рис.13). В выборку включаются только операции с типом категории `expense`, суммы суммируются и приводятся к положительным значениям. Результирующий набор данных используется на фронтенде для построения диаграммы структуры расходов.

```
@app.route("/api/stats/spending")
def spending():
    if (resp := require_auth()) is not None:
        return resp
    today = date.today()
    start = parse_date(request.args.get("start"), today.replace(day=1))
    end = parse_date(request.args.get("end"), today)
    data = aggregate_spending(start, end, g.user["id"])
    return jsonify({"spending": data})
```

Рисунок 13. Код

- GET `/api/stats/timeseries` отвечает за формирование временного ряда чистого денежного потока (рис.14). Интервал агрегации задаётся параметром `interval` и может быть дневным, месячным или годовым. Для каждого «шага» времени рассчитывается суммарная величина операций (с учётом знака), что позволяет отобразить динамику изменения баланса за заданный период.

```
@app.route("/api/stats/timeseries")
def timeseries():
    if (resp := require_auth()) is not None:
        return resp
    today = date.today()
    start = parse_date(
        request.args.get("start"),
        today - timedelta(days=30),
    )
    end = parse_date(request.args.get("end"), today)
    interval = request.args.get("interval", "daily")
    if interval not in {"daily", "monthly", "yearly"}:
        interval = "daily"
    data = aggregate_timeseries(start, end, interval, g.user["id"])
    return jsonify({"timeseries": data})
```

Рисунок 14. Код

Внешний вид веб-приложения.

- Страница авторизации (рис.15).

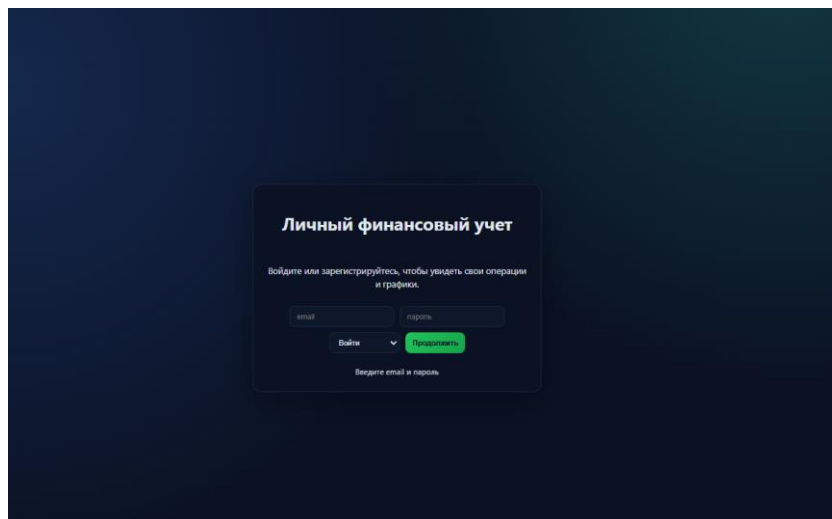


Рисунок 15. Страница авторизации

- Главная страница веб-приложения (рис.16).

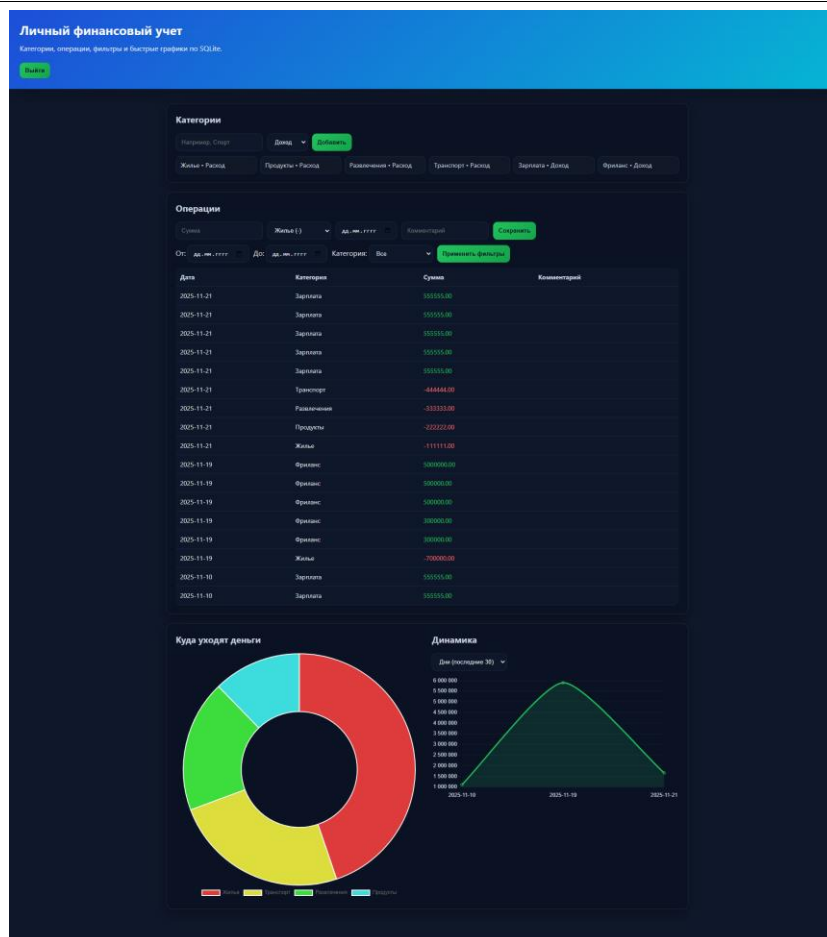


Рисунок 16. Главная страница

4 Выводы

Реализованный механизм личного финансового учёта демонстрирует, каким образом на базе связки Flask + SQLite + простой фронтенд можно построить полноценный веб-сервис для регистрации доходов и расходов, управления категориями и анализа финансовых потоков. Чёткое разделение логики аутентификации, операций и статистики, использование сессий для привязки данных к конкретному пользователю, а также визуальное представление информации в виде таблиц и графиков делают приложение удобным и наглядным как для конечного пользователя, так и для целей обучения основам веб-разработки.

Библиографический список

1. Raju N.K.K. et al. Diabetes Prediction Using Machine Learning and Flask // Biomedical & Pharmacology Journal. 2024. Т. 17. № 2. С. 1307–1316.
2. Nath A., Nema S. Clustering Visualization and Class Prediction using Flask of Benchmark Dataset for Unsupervised Techniques in Machine learning // International Journal of Innovative Technology and Exploring Engineering. 2020. Т. 9. № 7. С. 1297–1302.
3. Bose S. K., Ghosh S., Das S. A GUI-Based Approach to Predict Heart Disease Using Machine Learning Algorithms and Flask API // Computational

- Intelligence in Communications and Business Analytics. 2023. C. 301–309. DOI: 10.1007/978-3-031-48876-4_23.
4. Yaganteeswarudu A. Multi Disease Prediction Model by using Machine Learning and Flask API // 2020 5th International Conference on Communication and Electronics Systems (ICCES). 2020. C. 1–6. DOI: 10.1109/ICCES48766.2020.9137896.
 5. Lakshmanarao A., Raja Babu M., Bala Krishna M.M. Malicious URL Detection using NLP, Machine Learning and FLASK // 2021 International Conference on Innovative Computing, Intelligent Communication and Smart Electrical Systems (ICSES). 2021. C. 1–6. DOI: 10.1109/ICSES52305.2021.9633889.