Designing a Scalable Local Activity Platform

Team Members: Jin Yang Chen, Abraham Chang, Salamun Nuhin, Omer Yurekli, Omar Tall, Arona Gaye

Real-World Problem

Problem Statement:

- Boston residents lack a centralized, accessible platform to explore and participate in local activity groups.

Key Challenges:

- Residents have difficulty finding events that match their interests, age group, or availability
- Activity organizers struggle to manage memberships, schedule sessions, and collect participant feedback
- Existing solutions are either too generic or lack the local/community focus

Our Solution:

We developed a lightweight, scalable Flask web application that connects residents with local activity groups, streamlines event registration, and provides organizers with intuitive group management tools.

Target Audience & Goals

Target Audience:

- Boston residents seeking to discover, join, and engage with local activity
- Community organizers looking for tools to manage events, track members, and gather feedback

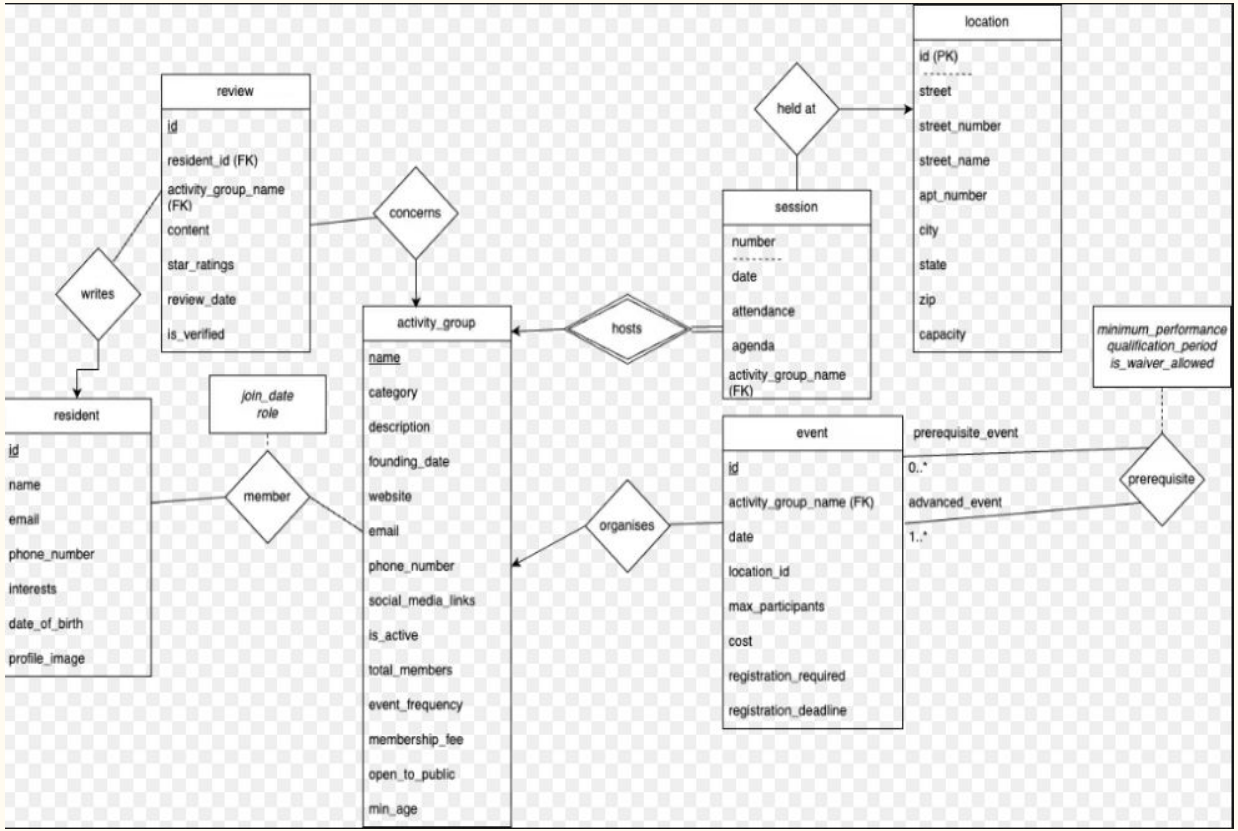
Project Goals:

- Enable residents to discover and filter activity groups by interest, cost, age group, and frequency
- Allow users to register for events and track their participation
- Provide organizers with tools to manage group details, schedule events, and monitor engagement
- Ensure data integrity, secure access, and minimal redundancy across the system

E-R Diagram

Our E-R diagram models how residents join activity groups, attend events, and leave reviews.

- Residents can join multiple activity groups through memberships.
- Each group organizes events, which may have sessions held at specific locations.
- Residents can write reviews for groups.
- Some events require a prerequisite event to be completed.



Database Schema

```sql
CREATE TABLE resident (
    resident_id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    email TEXT UNIQUE NOT NULL,
    phone_number TEXT UNIQUE,
    interests TEXT,
    date_of_birth TEXT,
    profile_image TEXT,
    username TEXT UNIQUE,
    hashed_password TEXT,
    is_deleted INTEGER DEFAULT 0 -- Add this column
);

-- Table: activity_group
CREATE TABLE activity_group (
    name TEXT PRIMARY KEY,
    category TEXT,
    description TEXT,
    founding_date TEXT,
    website TEXT,
    email TEXT,
    phone_number TEXT,
    social_media_links TEXT, -- stored as JSON string
    is_active INTEGER DEFAULT 1,
    total_members INTEGER DEFAULT 0,
    event_frequency TEXT,
    membership_fee INTEGER,
    open_to_public INTEGER,
    min_age INTEGER
);
```

Entities and Attributes

- Users
  - id (PK), username, email, hashed_password
- Groups
  - id (PK), name, description, category, cost
- Events
  - id (PK), group_id (FK), title, date, location
- Registrations
  - id (PK), user_id (FK), event_id (FK), status

Application Architecture

```
├── app/
│   ├── models/ # Data models used across the application
│   ├── routes/ # Routes handle the HTTP requests and render the appropriate templates (no business logic)
│   ├── services/ # Services handle the business logic of the application
│   ├── static/ # Static files like CSS
│   ├── templates/ # Jinja templates
│   ├── utils/ # Utility functions that are used across the entire application
│   ├── activity.db # Built database file
│   └── run.py # Main executuable file
└── tests/ # Tests for each file are labelled with the file name
    ├── routes/ # Tests for route handlers
    ├── services/ # Tests for service layer
    ├── integration_test.py # End-to-end tests
    └── conftest.py # Test fixtures and configuration
```

- Followed MVC separation
- Reusable services & clear route logic
- Organized tests mirroring structure

Key Features

- Secure user authentication via Flask-Login and Flask-Bcrypt
- Role-based access control (if admin/user distinction
- Product catalog management
- Order history
- Session management using cookies
- Form validation and flash messages for UX

Technologies Used

- Backend: Flask, Flask-Login, Flask-Bcrypt, SQLite
- Frontend: HTML, CSS, Jinja2 templates
- Environment: .env for secrets, makefile for automation
- Testing & Linting: pytest, black, pylint, mypy, isort, autoflake

Demo Setup

Translating common user flows into SQL queries

- Find which are the highly-rated activity groups by checking the average rating
  - SELECT AVG(star_rating) as avg_rating

    FROM review

    WHERE activity_group_name = ?
- Know the exact the details of a particular event
  - SELECT * FROM event WHERE event_id = ?

Database access

- READ
  - **Available to the public
- CREATE / UPDATE / DELETE
  - Authenticated users, can only update + delete the users own themselves
  - Interface: Form on the website
- Note: SQLite doesn't have built in Row-Level security (RLS) policies in the same way PostgreSQL does by design, so we have to manage access on the application level and not the database level

Things Not Fully Implemented

- Sessions
- Waitlist and waitlist notifications
- Abilities to register for an event directly from the site

Challenges & Lessons Learned

- Challenge: Managing auth securely in a minimal app
- Challenge: Getting services and routes to stay cleanly separated
- Lesson: Effective use of tooling (linting, env configs) helped maintain quality
- Lesson: ER design decisions early on simplify logic later

Q&A