Ping Proposal Design Doc

mango db connection via VS code:
mongodb+srv://jclinjiachunn_db_user:YQo8mAnRaFvAfO4v@ping.qvbwgb5.mongodb.net/?appName=Ping

Use Google Map API for the base map

—-------------------------------------

Proposal

Ping is a real-time, location-based social platform designed for college students. It helps students find nearby peers to do everyday activities such as studying, eating, exercising, attending campus events, or organizing activities-based hangouts without requiring advance planning or existing social connections. The platform centers on a map-based interface that displays live activity posts around you, allowing users to browse, filter, and join events based on distance, category, and availability. To ensure safety and trust, users must verify their student identity through a school email or student ID to create or join events, while unverified users may browse only. By combining real-time data, geographic proximity, and student-only verification, Ping lowers the barrier to spontaneous, low-pressure social interaction and makes it easy for students to connect with others who are available right now.

App Pages

Page 1: Login/SignUp (Username/password/save user for later)

a. Can use any email for signing up/google log in
b. Authentication with school email/student ID verification - can only browse without being verified

Page 2: Add/Edit User Bio/Info, User Homepage; this information will be stored in SQL database so that the system can recommend/feed events/posts to the user

1. User name
2. User ID (generated by the system)
3. Age (hidden on default?)
4. School, Program, Major, Year
5. Preference for events/post feeding; event types include: sport, art, social, study – choose 1-4 types

6. Interest: type in/choose 3-5 specific interests (e.g. Golf, EDM, Hiking, Anime, Cooking, Museums, etc.)

Page 3: Main Page, map view by default, expand filters on the left, and expand list view on the right if necessary. Showing real-time posts from locations on/around campus; each data point represents the number of active posts that users can click into (zoom in - more details/individual posts, zoom out - show only the numbers of posts around this area/cluster or show only the more recent ones - like bookings listings)
Page 3.1 On the left - Filter your list based on distance (type in/move the bar 0-20 miles), availability (choose only show open ones - closed bc number reached), category (sport, art, social, study)
Page 3.2 Map view of the posts, show your current location and the posts around you
Page 3.3 List of events: show the top active posts - searching posts by keywords?
(Optional: Sort - most popular (most number of people going/fewer spots left), newest/most recent, closest, etc.)

Page 4: Event Page (enter through the icon in the map view or the listing). Showing the event title, time, location, number of participants (e.g. 1/3), organizer, event details. Users can comment (the only way to interact with the organizer for now), save (interested), or join (going) the event.

Page 5: Event Creation Page: enter through a plus sign on the map view?
Enter event title, time, location, total number of participants, organizer, event details

Page 6: Saved Events Page
List view of saved events

Architecture

- Frontend: React, Vite, Tailwind CSS
- Backend: Node.js, Express, PostgreSQL
- Authentication: Google OAuth

```
├── backend/
│   ├── src/
│   │   ├── routes/       # API route handlers
                - userRoutes
                - eventRoutes
                - discoveryRoutes
│   │   └── index.js      # Express server
│   └── config.json       # Database configuration
```

```
├── frontend/
│   ├── src/
│   │   ├── pages/        # React page components
│   │   ├── components/   # Reusable components
│   │   ├── services/     # API service layer
│   │   └── contexts/     # React contexts
│   └── .env              # For Google OAuth
└── query/                # Complex SQL queries
```

Datasets

1. Google Map API
2. User dataset (currently do not have access)

Basic Functionalities

Core Loop

1. User Login (Google OAuth) → can browse immediately
2. User verifies student status (school email / student ID)
3. User sets profile and preferences (1 - 4 tags) → improves matching/ranking
4. Map-first discovery: see LIVE activity on campus → Going / interested / comment
5. Click a pin → open event post → browse details → Going / interested / comment
6. User can activate/deactivate their own posts (default active duration is 2h)
7. User can save a list of event posts they are interested in (by clicking on the "interested" button)
8. Direct message to the organizer (pending)

Guest v.s. Verified

- Unverified: browse map + list + event details (organizer and exact location hidden)
- Verified: create post, join, like, comment, analytics

*This pdf contains the app's API spec documentation. Please refer to our GitHub repo for implementation.

Database Design

users: contains the basic information and bio of each user
uid, username, age, school (upenn), program, major, is_verified(bool), interest_tags(text[]), interested

events: single table containing all events (emphasize real-time)

event_id: PK
creator_uid, uid FK
title, description
start_time, end_time: start timestamp and ends after 2h by default
status: active, ended, canceled

join_like_comments
event_joins(event_id, uid, joined_at)
event_likes(event_id, uid, liked_at)
event_comments(comment_id, event_id, uid, body, created_at)

API Specification

Use Google Map API

User Routes

Route 1

Route: /auth/google

Method: GET

Description: Start Google OAuth login flow.

Route Parameter(s): None

Route Handler: authGoogle(req, res)

Return Type: Redirect (302)

Return Parameters: N/A

Expected (Output) Behavior: Redirects user to Google consent screen

Route 2

Route: /auth/google/callback

Method: GET

Description: Handle Google OAuth callback, create/find user, establish session.

Route Parameter(s): None (Google provides code in query)

Route Handler: authGoogleCallback(req, res)

Return Type: Redirect (302)

Return Parameters: N/A

Expected (Output) Behavior: If OAuth succeeds, set a session cookie and redirect the user to the app (e.g., `/map`). Otherwise return an error.

Route 3

Route: /auth/me

Method: GET

Description: Return the currently logged-in user's basic account + verification state.

Route Parameter(s): None

Route Handler: authMe(req, res)

Return Type: JSON Object

Return Parameters: { uid (uuid), username (text), email (text), is_verified (bool), interest_tags (text[]) }

Expected (Output) Behavior: If user is logged in, return their user object; otherwise return 401.

Route 4

Route: /auth/logout

Method: POST

Description: Log the user out by clearing the session.

Route Parameter(s): None

Route Handler: authLogout(req, res)

Return Type: JSON Object

Return Parameters: { logged_out (bool) }

Expected (Output) Behavior: Clears session cookie and returns logged_out=true.

Route 5

Route: /users/me

Method: GET

Description: Get the logged-in user’s full profile.

Route Parameter(s): None

Route Handler: getMyProfile(req, res)

Return Type: JSON Object

Return Parameters: { uid (uuid), username (text), age (int), school (text), program (text), major (text), is_verified (bool), interest_tags (text[]) }

Expected (Output) Behavior: If logged in, return user profile; otherwise return 401.

Route 6

Route: /users/me

Method: PUT

Description: Create or update the logged-in user’s profile + preference tags (1–4).

Route Parameter(s): None

Route Handler: updateMyProfile(req, res)

Return Type: JSON Object

Return Parameters: { uid (uuid), username (text), age (int), school (text), program (text), major (text), interest_tags (text[]) }

Expected (Output) Behavior: If request is valid, update user record and return updated profile; if tags not in 1–4, return 400.

Route 7

Route: /users/me/interested
Method: GET
Description: Return the user’s saved “Interested” event list.
Route Parameter(s): None
Route Handler: getMyInterested(req, res)
Return Type: JSON Object
Return Parameters: { items (list of JSON) } where each item is { event_id (uuid), title (text), status (text), start_time (timestamptz), end_time (timestamptz), place_name (text), lat (float), lng (float) }
Expected (Output) Behavior: If verified, return saved list; otherwise return 403.

Route 8

Route: /users/me/interested/:event_id
Method: POST
Description: Add an event to the user’s saved “Interested” list.
Route Parameter(s): event_id (uuid)
Route Handler: addInterested(req, res)
Return Type: JSON Object
Return Parameters: { event_id (uuid), interested (bool) }
Expected (Output) Behavior: If verified and event exists, add to list and return interested=true; otherwise return error.

Route 9

Route: /users/me/interested/:event_id
Method: DELETE
Description: Remove an event from the user’s saved “Interested” list.
Route Parameter(s): event_id (uuid)
Route Handler: removeInterested(req, res)
Return Type: JSON Object
Return Parameters: { event_id (uuid), interested (bool) }
Expected (Output) Behavior: If verified and event is in list, remove and return interested=false; otherwise return error.

Event Routes

Route 10

Route: /events
Method: POST
Description: Create a new real-time event/buddy post. Defaults to active for 2 hours.
Route Parameter(s): None
Route Handler: createEvent(req, res)
Return Type: JSON Object
Return Parameters: { event_id (uuid), creator_uid (uuid), title (text), description (text), start_time (timestamptz), end_time (timestamptz), status (text), place_name (text), lat (float), lng (float) }
Expected (Output) Behavior: If verified, create event with start_time=now() and end_time=now()+2h, status=active; otherwise return 403.

Route 11

Route: `/events/:event_id`
Method: GET

Description: Get full details for a single event (viewable by guests).
Route Parameter(s): `event_id (uuid)`
Route Handler: `getEvent(req, res)`
Return Type: JSON Object
Return Parameters:
`{ event_id (uuid), creator_uid (uuid), title (text), description (text), mood (text), intention (text), start_time (timestamptz), end_time (timestamptz), status (text), place_name (text), lat (float), lng (float), counts (json), viewer_state (json) }`
Where `counts = { going (int), interested (int), comments (int) }` and `viewer_state = { going (bool), interested (bool) }`
Expected (Output) Behavior: If event exists, return event details; otherwise return 404.

Route 12

Route: `/events/:event_id/deactivate`
Method: POST
Description: Manually deactivate (end early) the creator's own event.
Route Parameter(s): `event_id (uuid)`
Route Handler: `deactivateEvent(req, res)`
Return Type: JSON Object
Return Parameters: `{ event_id (uuid), status (text), end_time (timestamptz) }`
Expected (Output) Behavior: If verified and user is creator, set `status=ended` (and optionally `end_time=now()`) and return updated state; otherwise return 403.

Route 13

Route: `/events/:event_id/activate`
Method: POST
Description: Reactivate a previously ended event (sets a fresh 2-hour window).
Route Parameter(s): `event_id (uuid)`
Route Handler: `activateEvent(req, res)`
Return Type: JSON Object
Return Parameters: `{ event_id (uuid), status (text), start_time (timestamptz), end_time (timestamptz) }`
Expected (Output) Behavior: If verified and user is creator, set `status=active`, `start_time=now()`, `end_time=now()+2h`; otherwise return 403.

Discovery Routes

Route 14

Route: `/discover`
Method: GET
Description: Return a ranked list of active events near the user. If too many exist, filter by user preferences/custom filters and rank by latest posted (start_time DESC).
Route Parameter(s): None
Route Handler: `discover(req, res)`
Return Type: JSON Object
Return Parameters: `{ items (list of JSON), next_cursor (text|null), applied (json) }`
Each item: `{ event_id (uuid), title (text), mood (text), intention (text), start_time (timestamptz), end_time (timestamptz), status (text), place_name (text), lat (float), lng (float), distance_m (float), ttl_minutes (int), counts (json) }`
Expected (Output) Behavior: Return only active events (`status=active` and `now()<end_time`), apply filters, then sort by most recent `start_time`. Default returns top 10.

Route 15

Route: `/map/points`
Method: GET
Description: Return map points (pins/clusters) for active events in current viewport; overload-safe by limiting to top recent results and applying preferences/filters.
Route Parameter(s): None
Route Handler: `mapPoints(req, res)`
Return Type: JSON Object
Return Parameters: `{ points (list of JSON), overload_control (json), applied (json) }`
Each point: either

- Cluster: `{ id (text), type ("cluster"), lat (float), lng (float), count (int), newest_start_time (timestamptz) }`

- Event pin: `{ id (uuid), type ("event"), lat (float), lng (float), title (text), mood (text), start_time (timestamptz), ttl_minutes (int) }`
  Expected (Output) Behavior: Return only active events; if too many candidates, keep top `max_points` by latest `start_time` after filters/preferences

Engagement (Going/Interested/Comment)

Route 16

Route: `/events/:event_id/going`
Method: POST
Description: Mark the user as “Going” to an event (join).
Route Parameter(s): `event_id (uuid)`
Route Handler: `setGoing(req, res)`
Return Type: JSON Object
Return Parameters: `{ event_id (uuid), going (bool), going_count (int) }`
Expected (Output) Behavior: If verified and event is active, insert into `event_joins`; return updated going count.

Route 17

Route: `/events/:event_id/going`
Method: DELETE
Description: Remove “Going” status (leave event).
Route Parameter(s): `event_id (uuid)`
Route Handler: `unsetGoing(req, res)`
Return Type: JSON Object
Return Parameters: `{ event_id (uuid), going (bool), going_count (int) }`
Expected (Output) Behavior: If verified, delete from `event_joins`; return updated going count.

Route 18

Route: `/events/:event_id/interested`
Method: POST
Description: Mark the user as “Interested” (like). Also used to build the user’s saved interested list.
Route Parameter(s): `event_id (uuid)`
Route Handler: `setInterested(req, res)`
Return Type: JSON Object
Return Parameters: `{ event_id (uuid), interested (bool), interested_count (int) }`
Expected (Output) Behavior: If verified, insert into `event_likes`; return updated interested count.

Route 19

Route: `/events/:event_id/interested`
Method: DELETE

Description: Unmark "Interested" (unlike) and remove from saved list.
Route Parameter(s): `event_id (uuid)`
Route Handler: `unsetInterested(req, res)`
Return Type: JSON Object
Return Parameters: `{ event_id (uuid), interested (bool), interested_count (int) }`
Expected (Output) Behavior: If verified, delete from `event_likes`; return updated interested count.

Route 20

Route: `/events/:event_id/comments`
Method: GET
Description: Fetch comments for an event.
Route Parameter(s): `event_id (uuid)`
Route Handler: `getComments(req, res)`
Return Type: JSON Object
Return Parameters: `{ items (list of JSON), next_cursor (text|null) }`
Each item: `{ comment_id (uuid), event_id (uuid), uid (uuid), body (text), created_at (timestamptz) }`
Expected (Output) Behavior: If event exists, return recent comments; otherwise return 404. Guests may read.

Route 21

Route: `/events/:event_id/comments`
Method: POST
Description: Post a new comment on an event.
Route Parameter(s): `event_id (uuid)`
Route Handler: `createComment(req, res)`
Return Type: JSON Object
Return Parameters: `{ comment_id (uuid), event_id (uuid), uid (uuid), body (text), created_at (timestamptz) }`
Expected (Output) Behavior: If verified and event exists, insert into `event_comments` and return the comment; otherwise return error.

Route 22

Route: `/events/:event_id/comments/:comment_id/replies`
Method: GET
Description: Fetch replies for a specific comment within an event thread.

Route Parameter(s): `event_id (uuid)`, `comment_id (uuid)`
Route Handler: `getReplies(req, res)`
Return Type: JSON Object
Return Parameters:
`{ items (list of JSON), next_cursor (text|null) }`
Each item: `{ comment_id (uuid), event_id (uuid), parent_comment_id (uuid), uid (uuid), body (text), created_at (timestamptz) }`
Expected (Output) Behavior: If `event_id` and `comment_id` are valid, return replies ordered by `created_at ASC` (or DESC). If not found, return 404.

Live demo script

1. login via Google authentication
2. Profile
    a. enter/edit/save your profile, the system will recommend events for you according to the tags you selected (to be implemented)
    b. check out and manage the events you saved
3. home page
    a. map view of all current events that are going on around you, you can click on the pins from the map to show the event details
        i. join, add to the participants list
        ii.