Design Documentation

1. Project overview

A lightweight Campus Event Management prototype that lets college admins create events and students browse & register, records attendance, collects feedback, and provides reports on event popularity and student participation.

2. Data to track

We track the core entities and interactions required to run events and produce useful analytics.

Entities & main data fields

- **College**
  - `college_id` (integer, internal id)
  - `name` (text)
  - `location` (text)
- **Student**
  - `student_id` (integer)
  - `college_id` (integer, FK)
  - `name` (text)
  - `email` (text, unique)
  - `year` (integer)
  - `department` (text)
- **Event**
  - `event_id` (integer)
  - `college_id` (integer, FK)
  - `title` (text)
  - `description` (text)
  - `event_type` (text: Workshop/Fest/Seminar/Hackathon)
  - `start_date` (datetime)
  - `end_date` (datetime)
- **Registration**
  - `registration_id` (integer)
  - `student_id` (integer, FK)
  - `event_id` (integer, FK)
  - `registration_date` (datetime)
  - constraint: unique(student_id, event_id)
- **Attendance**
  - `attendance_id` (integer)
  - `registration_id` (integer, FK)
  - `attended` (boolean)
  - `checkin_time` (datetime)

- **Feedback**
  - `feedback_id` (integer)
  - `registration_id` (integer, FK)
  - `rating` (integer 1–5)
  - `comments` (text)
  - `feedback_time` (datetime)

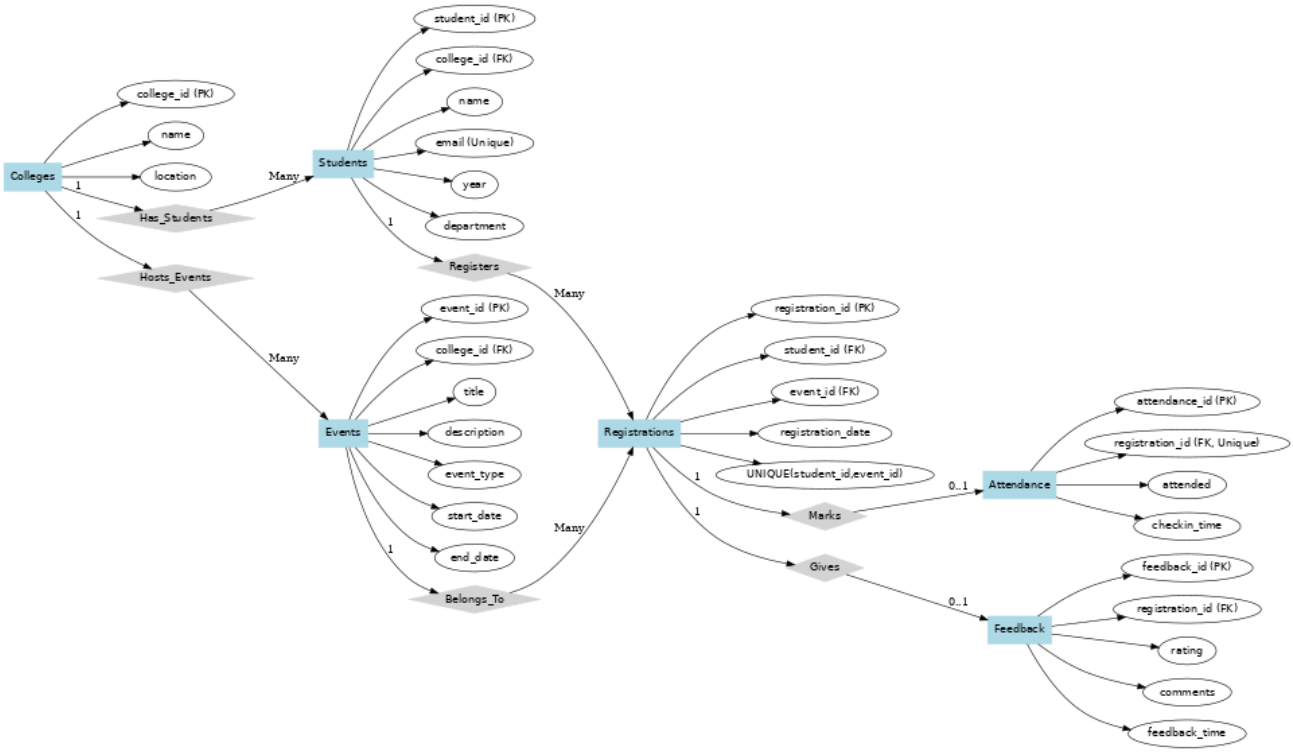
Why these fields?

- They capture the complete lifecycle: event creation → registration → attendance → feedback.
- `registration` links student and event (many-to-many). `attendance` and `feedback` are linked to a registration to ensure only registered students can be marked or give feedback.

3. Database schema (table sketch / ER description)

ER overview (textual)

- One `College` has many `Students` and many `Events`.
- A `Student` can register for many `Events` → modeled by `Registrations`.
- Each `Registration` can have one `Attendance` record and zero-or-one `Feedback` record.
- Referential integrity enforced by foreign keys.



4. API design (REST endpoints, inputs, outputs, and status behavior)

All endpoints are JSON-based. Common status codes used:

- `200 OK` – success with resource(s)
- `201 Created` – resource created
- `400 Bad Request` – client error (missing field, duplicate registration)
- `404 Not Found` – resource does not exist
- `500 Internal Server Error` – unexpected failure

Events

- `GET /events`
  - Returns: array of events
  - Example response: `[{ event_id, college_id, title, description, event_type, start_date, end_date }, ...]`
- `POST /events`
  - Body: `{ college_id, title, description, event_type, start_date, end_date }`
  - Success: `201 { event_id }`
  - Error: `400` if required fields missing
- `GET /events/:id`
  - Returns single event object or `404` if missing.

Students

- `POST /students`
  - Body: `{ college_id, name, email, year, department }`
  - Success: `201 { student_id }`
  - Error: `400` if email duplicate → return friendly message
- `GET /students` (optional) – list of students (admin)

Registrations

- `POST /register`
  - Body: `{ student_id, event_id }`
  - Success: `201 { registration_id }`
  - Error: `400` if duplicate (student already registered) or missing fields
- `GET /registrations/:event_id`
  - Returns: list of students registered for an event (join Students)
- `GET /students/:student_id/registrations`
  - Returns: list of events student has registered for (used by student dashboard)

Attendance

- `POST /attendance`
  - Body: `{ registration_id, attended, checkin_time }`
  - Success: `201 { attendance_id }` (or `200` on update)
  - Error: `400` if registration_id invalid
  - Admin route alternative: `POST /admin/attendance` that allows marking/overwriting
- `GET /attendance/:event_id`
  - Returns attendance rows for an event (join registrations & students)

Feedback

- `POST /feedback`
  - Body: `{ registration_id, rating (1–5), comments }`
  - Success: `201 { feedback_id }`
  - Error: `400` if registration_id invalid or rating out of range
- `GET /feedback/:event_id`
  - Returns list of feedback for an event (student name, rating, comments)

Reports

- `GET /reports/popular-events`
  - Returns events ordered by `COUNT(registrations)` descending.
  - Output: `{ title, total_registrations }`
- `GET /reports/student-participation`
  - Returns `{ name, events_attended }` for each student (count of attended marked records)
- `GET /reports/top-students`
  - Returns top N active students: `{ name, events_attended }` sorted desc; default `N=3`.

5. Workflows (sequence-style descriptions)

Below are step-by-step sequences written like human-readable sequence diagrams. Each step lists the actor + action.

A. Registration workflow (student browsing → registering)

1. **Student** opens Student Portal (frontend requests `GET /events`).
2. **Frontend** displays event list.
3. **Student** selects an event and clicks Register.
4. **Frontend** calls `POST /register` with `{ student_id, event_id }`.
5. **Server** validates: student exists, event exists, and checks unique(student_id, event_id).
   - If duplicate → return `400` with message “already registered”.

6. On success, **Server** inserts a `Registrations` row and returns `registration_id`.
7. **Frontend** confirms success to student and updates UI (optionally fetch `/students/:id/registrations`).

B. Attendance workflow (admin marks on event day)

1. **Admin** opens Event registrations (frontend calls `GET /admin/events/:event_id/registrations`).
2. **Server** returns list of registered students and any existing attendance rows.
3. For each check-in or when marking present:
    - **Admin** triggers `POST /admin/attendance` with `{ registration_id, attended }`.
    - **Server** inserts or updates the `Attendance` row for that `registration_id`.
4. After marking, server responds success; UI refreshes the registration list to reflect attendance.

C. Feedback & reporting workflow

1. After event ends, **Student** navigates to "My Registrations" and submits rating under a registration.
2. **Frontend** calls `POST /feedback` with `{ registration_id, rating, comments }`.
3. **Server** verifies `registration_id` and rating range, inserts Feedback row.
4. **Admin/Reports** call `GET /reports/...` endpoints to aggregate:
    - Popular events: aggregate registrations per event
    - Attendance %: join Attendance vs Registrations
    - Average feedback: `AVG(rating)` per event

6. Assumptions & edge cases with handling strategies

Assumptions

- Each student uses a unique email and has an internal `student_id`.
- Event IDs are unique across the whole database (we assume global uniqueness rather than per-college).
- System is initially single-tenant data store (not sharded) but designed for moderate scale (50 colleges × 500 students each).
- Admins are trusted users (no fine-grained RBAC in prototype).

Edge cases & how we handle them

1. **Duplicate registration**
    - Problem: Student clicks register twice or double-submit.
    - Handling: DB-level `UNIQUE(student_id, event_id)` constraint, and API checks returning a friendly `400` message like "You are already registered".

2. **Missing feedback**
    - Problem: Student never submits feedback.
    - Handling: Feedback is optional; reports compute averages only from existing feedback. If no feedback exists, show “No feedback yet”.
3. **Cancelled events**
    - Problem: Event canceled after some registrations.
    - Handling: Add a flag to Events (e.g., `status` = `active/cancelled`) — when cancelled, prevent new registrations and inform registered students (email in production). For prototype, include `event_type` or `status` and filter UI.
4. **Invalid IDs**
    - Problem: client sends non-existent `student_id`, `event_id`, or `registration_id`.
    - Handling: API validates existence and returns `400` or `404` with a clear message like “Invalid registration_id”.
5. **Attendance recorded multiple times**
    - Problem: multiple attendance rows for same registration.
    - Handling: model either allows one attendance row per registration (by enforcing unique `registration_id` in Attendance) or treat Attendance as append-only logs and use latest row. For simplicity, enforce uniqueness: upsert-style insert/update.
6. **Concurrent writes**
    - Problem: race when many students register at once.
    - Handling: defer to DB transactional guarantees; unique constraint prevents duplicates. For high scale, implement locking or queuing; not necessary for prototype.
7. **Time-based rules**
    - Problem: registering after event start or feedback before event ends.
    - Handling: API can check `now` vs `start_date` / `end_date` and reject/allow based on application rules (e.g., allow registration until event start).
8. **Data privacy**
    - Problem: exposing student emails publicly.
    - Handling: Only return required student fields in public endpoints; sensitive data visible only to admin endpoints.

7. Additional design notes (practicalities)

- **Prototype DB**: Start with SQLite (single file). If scaling needed, move to Postgres and add migrations.
- **Backend**: Node + Express chosen for same-language stack with React. Keep API layer thin and well-documented.
- **Frontend**: React (component based). Use Axios or fetch for API calls. Use routing for Admin vs Student views.
- **Logging & errors**: APIs should return friendly JSON errors; server should log full stack trace.
- **Testing**: Manual tests with Postman, plus unit tests for key API behaviors (duplicate registration, invalid ids).

- **Deployment notes**: For a simple demo, host backend on a service that supports Node + static SQLite (or switch DB to Postgres). Frontend can be deployed on Vercel/Netlify; configure CORS to allow frontend origin.

8. Minimal sequence summary

A student visits the portal and loads events (frontend calls `/events`). When they register, the client posts to `/register`, the server validates and writes the registration record. On the event day, the admin views registrations and marks attendance via `/admin/attendance`. After the event the student can post feedback using `/feedback`. The admin can then query `/reports/*` endpoints to compute popularity, attendance percentages, and top participants.

9. Appendices (copy-ready SQL schema)

You can include this short SQL snippet in an appendix of your document if asked to show table creation code:

```
CREATE TABLE Colleges (
  college_id INTEGER PRIMARY KEY AUTOINCREMENT,
  name TEXT NOT NULL,
  location TEXT
);

CREATE TABLE Students (
  student_id INTEGER PRIMARY KEY AUTOINCREMENT,
  college_id INTEGER,
  name TEXT NOT NULL,
  email TEXT UNIQUE NOT NULL,
  year INTEGER,
  department TEXT,
  FOREIGN KEY (college_id) REFERENCES Colleges(college_id)
);

CREATE TABLE Events (
  event_id INTEGER PRIMARY KEY AUTOINCREMENT,
  college_id INTEGER,
  title TEXT NOT NULL,
  description TEXT,
  event_type TEXT,
  start_date DATETIME,
  end_date DATETIME,
  FOREIGN KEY (college_id) REFERENCES Colleges(college_id)
);

CREATE TABLE Registrations (
  registration_id INTEGER PRIMARY KEY AUTOINCREMENT,
  student_id INTEGER,
  event_id INTEGER,
  registration_date DATETIME DEFAULT CURRENT_TIMESTAMP,
  UNIQUE(student_id, event_id),
  FOREIGN KEY (student_id) REFERENCES Students(student_id),
  FOREIGN KEY (event_id) REFERENCES Events(event_id)
```

```
);

CREATE TABLE Attendance (
  attendance_id INTEGER PRIMARY KEY AUTOINCREMENT,
  registration_id INTEGER UNIQUE,
  attended BOOLEAN DEFAULT 0,
  checkin_time DATETIME,
  FOREIGN KEY (registration_id) REFERENCES Registrations(registration_id)
);

CREATE TABLE Feedback (
  feedback_id INTEGER PRIMARY KEY AUTOINCREMENT,
  registration_id INTEGER,
  rating INTEGER CHECK(rating BETWEEN 1 AND 5),
  comments TEXT,
  feedback_time DATETIME DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (registration_id) REFERENCES Registrations(registration_id)
);
```

(Note: I set `registration_id` UNIQUE in Attendance above to simplify upsert behavior.)