Campus Event Management System - Design Document

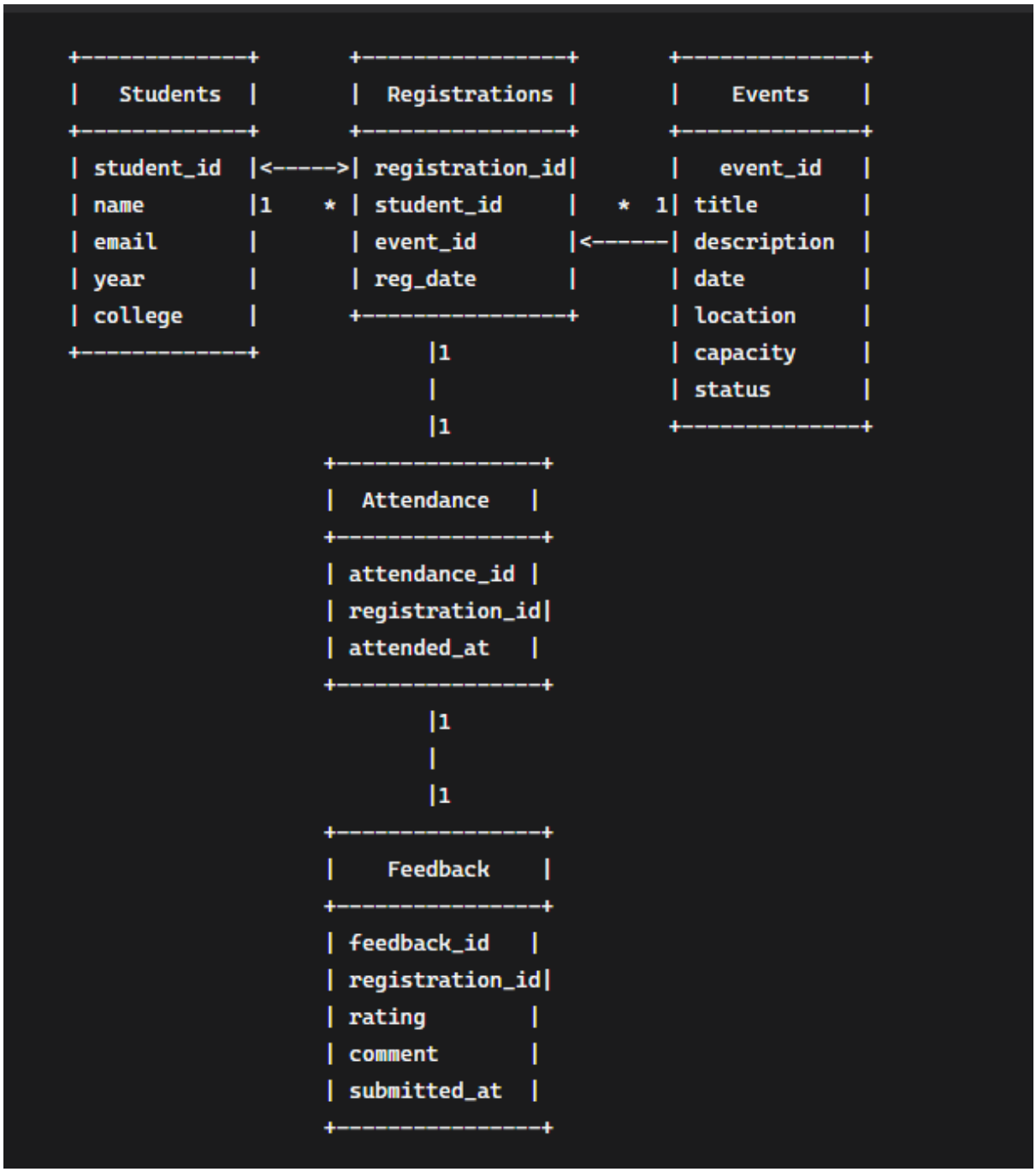
Data to Track

The system is designed to track the following core data entities and their lifecycle:

- **Event Creation:**
    - **Data:** event_id, title, description, date, location, capacity, status (e.g., active, ca ncelled).
    - **Purpose:** To define and manage all campus events. The status field allows for soft deletion or cancellation without losing historical data.
- **Student Registration:**
    - **Data:** student_id, name, email, year, college.
    - **Purpose:** To maintain a unique record of all students who can interact with the system.
- **Registration (Event-Student Junction):**
    - **Data:** registration_id, student_id, event_id, registration_date.
    - **Purpose:** To create a many-to-many relationship between Students and Events, recording the intent to attend. This is the core entity for tracking popularity.
- **Attendance:**
    - **Data:** attendance_id, registration_id, attended_at (timestamp).
    - **Purpose:** To mark a specific registration as having resulted in actual attendance. This is a direct child of a Registration, ensuring a student can only be marked present if they registered first.
- **Feedback:**
    - **Data:** feedback_id, registration_id, rating (1-5), comment, submitted_at.
    - **Purpose:** To collect post-event feedback. Linking it to the registration_id ensures only students who actually attended (and were marked present) can submit feedback, maintaining data quality for reports.

Database Schema (ER Diagram)

The schema enforces data integrity through foreign key relationships and unique constraints.



API Design

The API is RESTful and returns appropriate HTTP status codes (200, 201, 400, 404, 500).

Core Resource Endpoints:

Endpoint	Method	Description	Request Body	Success Response
/api/events	GET	Get all events	-	200 OK, Array of event objects
/api/events	POST	Create a new event	{title, description, date, location, capacity}	201 Created, Created event
/api/events/:id	GET	Get a specific event	-	200 OK, Single event object
/api/students	GET	Get all students	-	200 OK, Array of student objects
/api/students	POST	Register a new student	{name, email, year, college}	201 Created, Created student

Business Operation Endpoints:

Endpoint	Method	Description	Request Body	Success Response
/api/registrations	POST	Register a student for an event	{student_id, event_id}	201 Created, Registration object
/api/attendance	POST	Mark a student's attendance	{student_id, event_id}	201 Created, Attendance object
/api/feedback	POST	Submit feedback for an event	{student_id, event_id, rating, comment}	201 Created, Feedback object

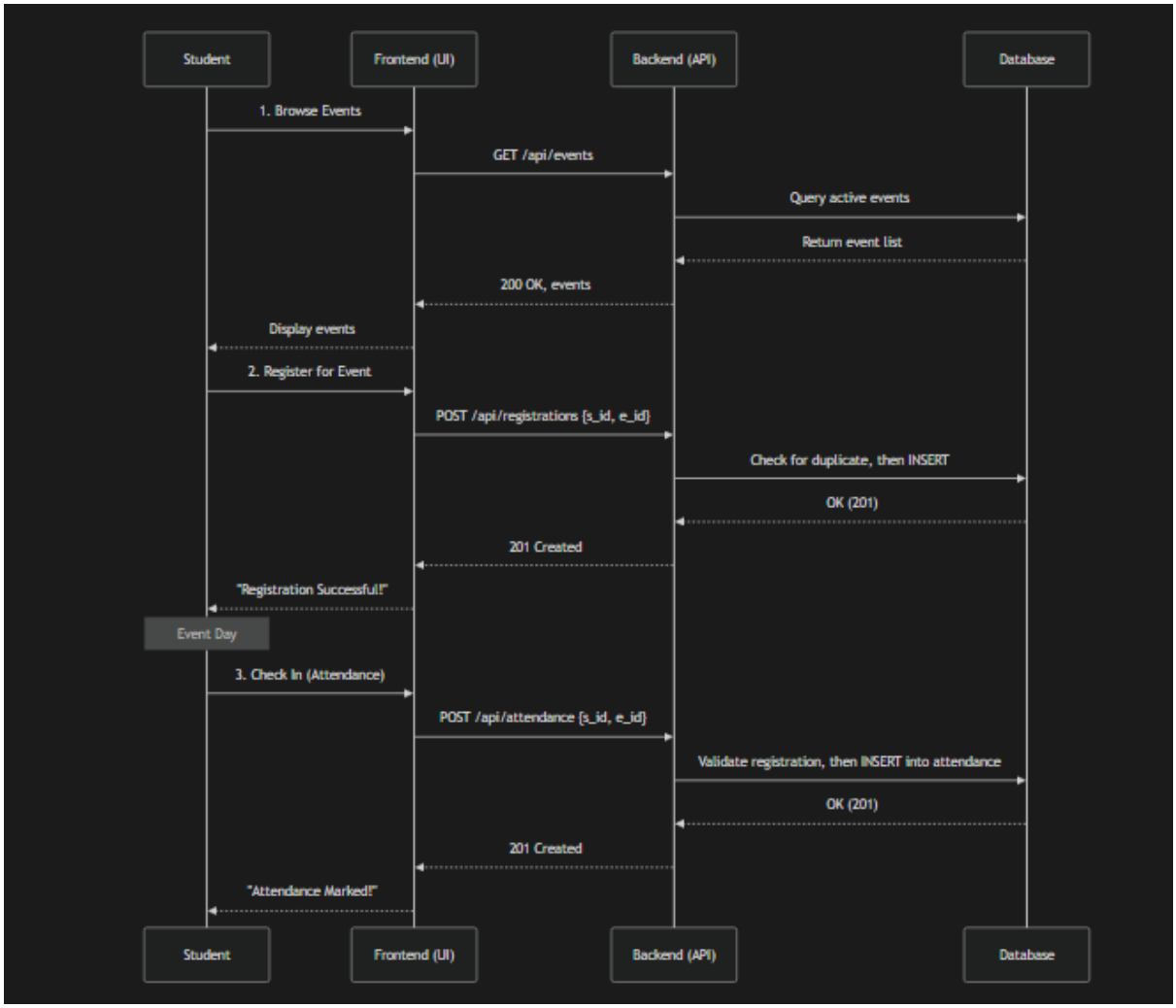
Reporting Endpoints:

Endpoint	Method	Description	Parameters	Response
/api/reports/event-popularity	GET	Events ranked by registration count	-	200 OK, [{event_name, total_registrations}, ...]
/api/reports/attendance	GET	Attendance % per event	-	200 OK, [{event_name, total_registered, attended, percentage}, ...]
/api/reports/feedback	GET	Avg. feedback score per event	-	200 OK, [{event_name, avg_rating}, ...]
/api/reports/top-students	GET	Top 3 most active students	-	200 OK, [{student_name, events_attended}, ...]
/api/reports/student-participation/:id	GET	Events attended by a student	student_id (in path)	200 OK, Array of event objects

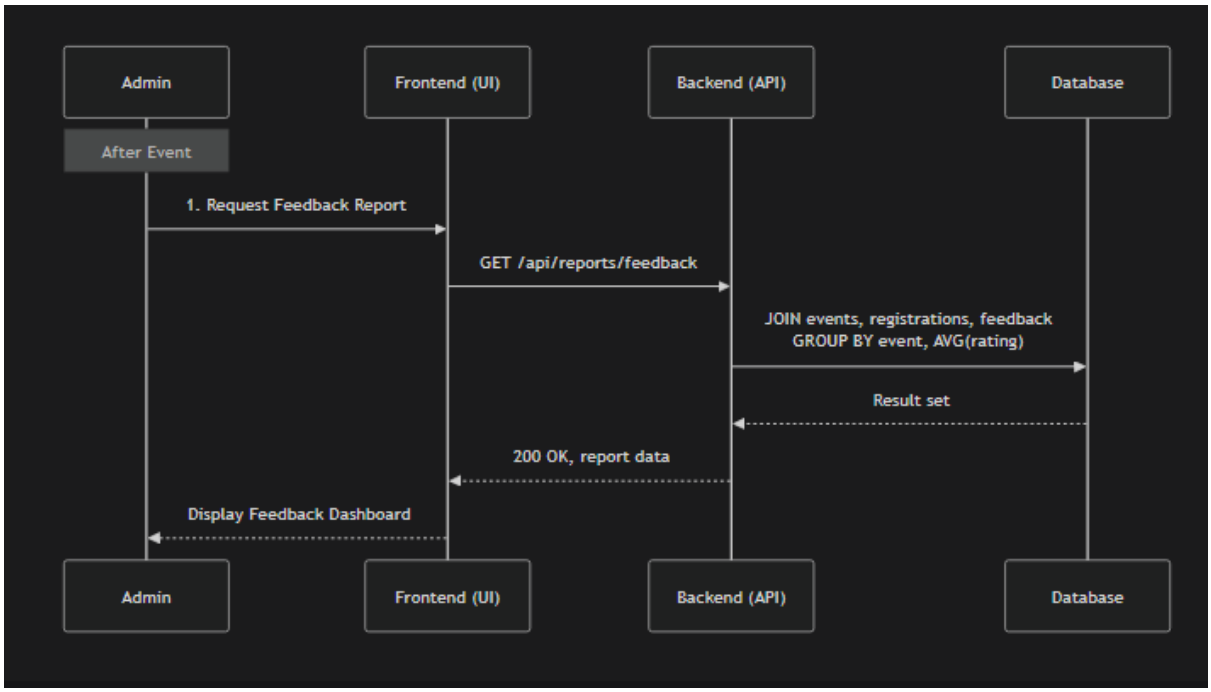
Workflows (Sequence Diagrams)

Workflow 1: Student Registration & Attendance

This sequence diagram shows the process from event discovery to attendance marking.



Workflow 2: Feedback Submission & Reporting
This sequence diagram shows the process after an event, from feedback submission to report generation.



Assumptions & Edge Cases

- **Duplicate Registrations:**
    - **Assumption:** A student can register for an event only once.
    - **Handling:** Enforced by a UNIQUE (student_id, event_id) constraint in the database. The API will return a 400 Bad Request error if violated.
- **Missing Feedback:**
    - **Assumption:** Feedback is optional and not all attendees will provide it.
    - **Handling:** The feedback table allows NULL comments. Reporting queries use AVG() and COUNT() which ignore NULL values, providing accurate averages based only on submitted ratings.
- **Cancelled Events:**
    - **Assumption:** Events should not be deleted to preserve historical data for reporting.
    - **Handling:** An event has a status field (active/cancelled). The UI and API listing endpoints (GET /api/events) can filter to only show active events. Registrations for a cancelled event are kept but can be excluded from reports based on the event's status.
- **Marking Attendance without Registration:**
    - **Assumption:** Attendance is only valid if a prior registration exists.
    - **Handling:** The attendance table has a foreign key to registrations. The database will reject any attempt to create an attendance record without a corresponding registration_id. The API logic must first find the correct registration ID for a given (student_id, event_id) pair.
- **Event Capacity:**
    - **Assumption:** Events have a finite capacity.
    - **Handling:** The registration API endpoint should include a check: SELECT COUNT(*) FROM registrations WHERE event_id = ?. If the count is >= the event's capacity, the API should return a 400 Bad Request with a message "Event is at capacity."
- **Data Integrity on Deletion:**
    - **Assumption:** If a student is deleted, their associated data should be cleaned up.
    - **Handling:** Using ON DELETE CASCADE on foreign keys ensures that deleting a student automatically removes their registrations, attendance records, and feedback, preventing orphaned records.