# Event Management System Design Document

This document provides a detailed design overview of the event management system, based on the Django and Django REST Framework code you provided. It covers the data, database structure, API, workflows, and key assumptions of the system.

### 1. Data to Track

The system is designed to manage four primary types of data, with clear roles and responsibilities for each:

* **Event Creation**: The core data for any event, including its **title**, **description**, **event\_type** (e.g., Hackathon, Workshop), **start\_time** and **end\_time**, **location**, **capacity**, and **status** (active, cancelled, or completed). Events are created and managed by an AdminProfile.
* **Student Registration**: A record that links a **Student** to an **Event**, indicating their intent to attend. It includes a registered\_at timestamp and tracks the registration **status** (registered, attended, or cancelled).
* **Attendance**: A log of when a student physically checks in for an event. This updates their registration status and records a check\_in\_time. Attendance is marked by an AdminProfile or authorized event staff.
* **Feedback**: A post-event record from a student, containing a numerical **rating** from 1 to 5 and an optional **comment**. Only students who have attended an event can submit feedback.

### 2. Database Schema

The database schema is built around a series of interconnected tables that define the relationships between users, colleges, and events.

Here is a sketch of the core tables with their relationships:

* **User**: (from Django's auth module)
  + id (PK), username, first\_name, last\_name, email
* **College**
  + id (PK), name, college\_code (unique), address, contact\_email
* **Student**
  + id (PK), user\_id (OneToOne to User), college\_id (FK to College), roll\_number, year\_of\_study
* **AdminProfile**
  + id (PK), user\_id (OneToOne to User), college\_id (FK to College)
* **Event**
  + id (PK), college\_id (FK to College), created\_by\_id (FK to User), title, description, event\_type, start\_time, end\_time, location, capacity, status
* **Registration**
  + id (PK), student\_id (FK to Student), event\_id (FK to Event), registered\_at, status, check\_in\_time
  + **Unique constraint**: (student\_id, event\_id)
* **AttendanceLog**
  + id (PK), student\_id (FK to Student), event\_id (FK to Event), check\_in\_time
* **Feedback**
  + id (PK), student\_id (FK to Student), event\_id (FK to Event), rating, comment, submitted\_at
  + **Unique constraint**: (student\_id, event\_id)

### 3. API Design

The system's functionality is exposed through a set of RESTful API endpoints.

| **Endpoint** | **Method** | **Description** |
| --- | --- | --- |
| /events/ | GET, POST | GET: List all events. POST: Create a new event. |
| /registrations/ | GET, POST | GET: List all registrations. POST: Create a new registration. |
| /attendance/ | POST | **Mark Attendance**: Updates a registration and creates an attendance log entry. Body: { "student": student\_id, "event": event\_id }. |
| /feedback-create/ | POST | **Submit Feedback**: Creates a feedback record for an event. Body: { "student": student\_id, "event": event\_id, "rating": int, "comment": str }. |
| /events/popularity/ | GET | **Report**: Returns events sorted by the number of registrations. |
| /reports/event-popularity/ | GET | **Report**: Returns a detailed popularity report for events. |
| /reports/attendance-percentage/ | GET | **Report**: Calculates attendance percentage for one or all events. |
| /reports/average-feedback/ | GET | **Report**: Calculates the average feedback rating for one or all events. |
| /registrations/top\_active/ | GET | **Report**: Returns a list of the top active students based on attended events. |

### 4. Workflows

#### Registration → Attendance → Reporting

This sequence illustrates the core workflow of a student registering, having their attendance marked, and how the data is used for a report.

1. A **Student** sends a POST request to /api/registrations/ with their ID and the event ID.
2. The **Server** creates a Registration record in the **Database** with status = 'registered'. It returns a successful response to the student.
3. An **Admin** at the event sends a POST request to /api/attendance/ with the student and event IDs.
4. The **Server** looks for the Registration record and updates its status to 'attended' and populates the check\_in\_time. It also creates a new entry in the AttendanceLog table.
5. A **User** (e.g., an admin or a reporter) sends a GET request to /api/reports/attendance-percentage/.
6. The **Server** queries the **Database** to count all registrations for the event and counts the number of registrations with status = 'attended'.
7. The **Server** calculates the percentage and returns the report data to the user.

### 5. Assumptions & Edge Cases

* **Duplicate Registrations**: This is handled by a **unique\_together** constraint on the Registration model, which ensures that a student can only have one registration record per event.
* **Missing Feedback**: The code handles this gracefully. The average\_feedback property on the Event model returns a rounded average of ratings if feedback exists. If no feedback has been submitted for an event, it returns None.
* **Cancelled Events**: The Event model includes a status field. While the API handles this, the frontend application would be responsible for filtering out cancelled events from public listings while keeping the data for historical records.
* **Invalid Attendance**: To mark attendance, the system first verifies that a student is already registered for the event. If a matching registration is not found, the attendance request is rejected with a 400 Bad Request error.
* **Event Capacity**: The capacity field is a positive integer, but the code does not enforce a strict check at the time of registration. This could allow for registrations beyond the set capacity under high concurrency.
* **User Deletion**: If a User who created an event is deleted, the event's created\_by field is set to NULL rather than deleting the event itself, preventing data loss.