Event Facilitation Module — Design & Deliverables

> Companion doc for your Educational Management System (EMS). This contains the data model, workflows, user roles, architecture, example views/URLs, migration notes, README and `requirements.txt` content (Option A).

---

1. Overview

This module handles event planning, approvals, registration, participation management and feedback. It is purpose-built to work with your current lightweight `UserProfile` model and minimal session-based auth.

Goals: - Admins create & approve events, allocate venues/resources - Teachers propose events and manage participants - Students register/unregister and give feedback - Parents receive alerts and optionally submit feedback

---

2. Data Model (final recommended)

I recommend using a canonical set of models (single source of truth) — `UserProfile`, `Event`, `EventRegistration`, `Feedback`.

Models (fields & notes)

UserProfile

```
first_name, last_name, email (unique)
department, designation, role
password (plain for test only)
roll_number (optional)
phone (optional)
created_at = auto_now_add
```

Event

```
title (str)
description (text, optional)
event_type (choices)
department (str)
date (DateField)
start_time, end_time (TimeField)
```

```
venue (str)
staff_coordinators (str)  # comma-separated names for now
equipment_required (text, optional)
status (choices: pending/approved/rejected)
created_by (FK to UserProfile, null=True, SET_NULL)  # safer than storing id
capacity (PositiveIntegerField, optional)
created_at = auto_now_add
```

EventRegistration

```
student (FK -> UserProfile)
event (FK -> Event)
registered_at (DateTime auto_now_add)
attended (BooleanField default False)
attendance_marked_at (DateTime null True)

Meta: unique_together = ('student','event')
```

Feedback

```
event (FK -> Event)
student (FK -> UserProfile)
rating (Integer 1..5)
feedback (text optional)
created_at = auto_now_add
```

> Optional `Participant` model is no longer necessary because `EventRegistration` is the canonical participant record. If you prefer a separate flat Participant log (for exports) you can keep one but ensure you populate it from `EventRegistration` in the same transaction.

---

3. ER Relationships (textual)

- `UserProfile (1)` — creates —> `Event (M)` via `created_by`
- `Event (1)` — has —> `EventRegistration (M)` (students who registered)
- `UserProfile (1)` — registers for —> `Event (M)` through `EventRegistration`
- `Event (1)` — has —> `Feedback (M)` from students/parents

---

4. Workflows

Each workflow includes minimal checks you should implement in views.

A. Event creation & approval (Admin+Teacher)

1. Teacher fills proposal form — status = `pending`, `created_by` = teacher.
2. Admin views `pending` events, inspects details (conflicts, venue, resources).
3. Admin clicks Approve → set `status='approved'` and optionally send email notifications.
4. Rejected events: `status='rejected'` with optional `rejection_reason` field.

Notes: check for venue/time overlaps when approving. Add `rejection_reason` field if you want auditability.

B. Resource and venue allocation

- Admin UI shows calendar and resource usage.
- On approval, mark resources reserved (you can model a `Resource` table later).

C. Student registration

- Student sees only events with `status='approved'`.
- When Register (POST): create `EventRegistration(student, event)` in a transaction. If capacity is set, check `seats_available()` before creating.
- On success: return updated template (or redirect) and show "Registered" instead of Register button.
- Also send notification (email or session message).

D. Feedback collection

- After event.date (or after attended), student or parent can submit feedback linked to event.
- Save `Feedback(event, student, rating, feedback)`.
- Optionally prevent multiple feedback by same student for same event (unique_together). Decide business rule.

E. Automated notifications

- Hook points: on event approval, event reminder (1 day before), registration confirmation.
- Use Django `send_mail` or a task runner (Celery / cron) for production. For test/machine-test, print to logs or use console backend.

---

5. Views / URL endpoints (recommended)

These follow simple RESTy patterns (session-auth). Add `login_required` wrappers if/when migrating to Django auth.

```
GET  /admin_dashboard/                -> admin_dashboard
POST /create_event_admin/             -> create_event_admin
GET  /approve_event/<int:event_id>/   -> approve_event
GET  /reject_event/<int:event_id>/    -> reject_event
GET  /teacher_dashboard/              -> teacher_dashboard
```

```
GET  /teacher/propose/               -> teacher_event_proposal (form)
POST /teacher/propose/               -> teacher_event_proposal (save)
GET  /student_dashboard/             -> student_dashboard
POST /register_event/<int:event_id>/ -> register_for_event
POST /unregister_event/<int:event_id>/ -> unregister_event
GET  /manage_participants/<int:event_id>/ -> manage_participants
POST /mark_attendance/<int:reg_id>/  -> mark_attendance
POST /unmark_attendance/<int:reg_id>/ -> unmark_attendance
GET/POST /parent_dashboard/          -> parent_dashboard (feedback submit)
```

Implementation notes: - `register_for_event` should `get_or_create` EventRegistration using `student` object (not IDs mixed with different user models). - When returning to student_dashboard, template should check `if event.registrations.filter(student=me).exists()` to show Registered state client-side.

---

6. Templates — UX checklist

- Student list: loop `events` and for each event compute `is_registered = event.registrations.filter(student_id=session_user_id).exists()` and render button accordingly.
- After POST register: redirect back to dashboard so the page re-renders from DB (session flash message shows success).
- Manage participants page should iterate `registrations = EventRegistration.objects.filter(event=event).select_related('student')` to get `student.first_name` etc. (Prefer this over a separate Participant table.)
- Mark attendance should toggle the `attended` boolean on `EventRegistration`.

---

7. Admin / Teacher participant management (less complexity approach)

Option A (recommended — less complexity): - Use `EventRegistration` as canonical participants table. Add `attended` and `notes` fields there. - `Participant` model is redundant; if you need a flat export table, create it but populate it from `EventRegistration` as part of the same transaction.

Why: one source of truth reduces FK mismatches and migration headaches.

---

8. Migration & dev safety notes (critical, given deadline)

1. **Do not change models in-place on main branch** — create a new branch for big changes.
2. **Backup DB** file (sqlite `.db`) before running migrations: copy `db.sqlite3` to `db.sqlite3.bak`.

3. If you have existing migration files with broken defaults, remove only the problematic migration files and recreate clean migrations — but only after backup.
4. When adding DateTimeField with a default, prefer `auto_now_add=True` or `default=timezone.now` rather than `default=datetime.date.today` (that's a date, not datetime). Use `django.utils.timezone.now`.
5. When you see `TypeError: fromisoformat: argument must be str` during migrate — it usually means a migration has a non-serializable default value (e.g., a `datetime.date` object). Fix migration to use `timezone.now` or `auto_now_add`.

Quick safe migration example for adding `registered_at`:

```
# models.py
registered_at = models.DateTimeField(auto_now_add=True)
```

Then `makemigrations` & `migrate`.

---

9. Code snippets / best-practice view functions

Register view (robust)

```
from django.db import transaction

@require_POST
def register_for_event(request, event_id):
    if request.session.get('user_role') != 'student':
        return redirect('login')

    student_id = request.session.get('user_id')
    student = get_object_or_404(UserProfile, id=student_id)
    event = get_object_or_404(Event, id=event_id, status='approved')

    with transaction.atomic():
        if event.capacity and event.seats_taken() >= event.capacity:
            messages.error(request, 'Event full')
        else:
            registration, created =
EventRegistration.objects.get_or_create(student=student, event=event)
            if created:
                messages.success(request, 'Registered')
            else:
                messages.info(request, 'Already registered')

    return redirect('student_dashboard')
```

Manage participants view (teacher)

```
def manage_participants(request, event_id):
    event = get_object_or_404(Event, id=event_id)
    # security: verify teacher owns event or is admin
    registrations =
EventRegistration.objects.filter(event=event).select_related('student')
    return render(request, 'manage_participants.html', {'event': event,
'registrations': registrations})
```

In template use `for r in registrations: r.student.first_name` etc.

---

10. Edge cases & checks

- Ensure session `user_id` always points to `UserProfile` row; handle case where session user was deleted.
- Avoid mixing `get_user_model()` (Django auth User) with `UserProfile` unless you wire them together.
- For production, switch to Django's `AbstractUser` or `AbstractBaseUser`.

---

11. README (for repository — minimal)

```
# Event Facilitation Module – EMS

## Overview
This module manages event creation, approval, student registration, participant
management and feedback for the Educational Management System.

## Setup (local / machine test)
1. Create virtualenv: `python -m venv .venv` and activate it.
2. Install: `pip install -r requirements.txt`.
3. Configure settings (if necessary) in `event_management_system/settings.py`.
4. Run migrations: `python manage.py migrate`.
5. Create test data (or run `python manage.py loaddata demo.json` if included).
6. Run server: `python manage.py runserver`.

## Notes
- This project currently uses a lightweight `UserProfile` for the machine test.
For real apps migrate to Django auth.
- Backup your DB before running destructive migrations.
```

---

12. requirements.txt (suggested)

```
Django==6.0
pytz
sqlparse
python-dotenv
```

Add extra libs if you integrate email or Celery later.

---

13. Deliverables I produced for Option A

- This document (data model, workflows, architecture + advice)
- README content + `requirements.txt` content above

If you want, I can now: 1. Produce the **full, ready** `models.py` file (safe, final version) — I already wrote a recommended version earlier; I can paste a final copy ready for you to drop in. 2. Produce the **views.py** skeleton for all endpoints listed above. 3. Produce **manage_participants.html** and **student_dashboard.html** template snippets that show how to check registration state.

Tell me which of (1),(2),(3) you want next — I'll generate the full code you can copy-paste into your project.

Good luck — you're close to submission. I recommend creating a branch and testing migrations against a copy of your DB before pushing to main.