

Design of a Policy-Aware Event Attendance and Academic Incentive System

Sam Utzinger
swu3@nau.edu

Northern Arizona University
Flagstaff, AZ, USA

Abstract

Northern Arizona university (NAU) holds its students to a regressor career developed plan that helps students achieve the mission of excellence at NAU. The School of Informatics, Computing, and Cyber Systems (SICCS) holds many in person event through the year that students can attend and revive extra credit in a class of their choice. A full stack application was developed to help admin create events and students to sign in and track their attendance. A SQL data base with python endpoints runs the back end administrative checks while a VUE

Keywords

attendance tracking, extra credit, higher education, policy enforcement, CRUD application, FastAPI, Vue, PostgreSQL

1 Introduction

Northern Arizona University (NAU) encourages students to participate in career-development activities outside of class to build skills that are useful beyond the classroom. Within the School of Informatics, Computing, and Cyber Systems (SICCS), students can attend in-person events (e.g., industry nights, technical talks, workshops) and earn extra credit that can be applied to a course of their choice. While the program is valuable, the traditional workflow for tracking attendance is often manual and inconsistent. Paper sign-in sheets are error-prone, course eligibility rules are difficult to enforce reliably, and generating end-of-semester reports frequently becomes a time-consuming spreadsheet task.

To address these issues, I developed a web application that supports extra credit event check-in for students and administrative tooling for event creation. The larger goal of this writing sample is to serve as a reproducible guide and a reference that shows the skills developed. This paper explains the major decisions, the database design, and the implementation workflow.

This project follows a common *Create, Read, Update, Delete* (CRUD) application pattern used across modern web systems. However, building a real university-facing application adds constraints that do not appear in toy CRUD examples. NAU operates within organizational security standards and must comply with the U.S. Family Educational Rights and Privacy Act (FERPA), which restricts access to student enrollment information. The system and this paper enforce role-based access and carefully manage how enrollment data is used.

2 Method

2.1 System Overview

This system follows a three-tier web architecture consisting of a client, an API service, and a relational database. The client **front end** provides the user interface for student check-in and administrator management. The API **back end** (FastAPI) validates requests, applies policy logic, and mediates access to protected data. The **PostgreSQL** database stores users, courses, enrollments, events, and attendance records as the auditable source of truth, using integrity constraints to prevent invalid or duplicate records.

In deployment, the front end can be hosted (e.g., on AWS) so that students and faculty can access the site from anywhere. The back end runs as a service (FastAPI) that responds to front-end actions such as “send verification code,” “create event,” or “submit attendance.” The database serves as the source of truth and provides auditability.

2.2 Design Goals and Challenges

The development process has four core challenges that drive the design and need to be solved best:

- (1) **Secure check-in (non-shareable access):** Attendance should be difficult to be claimable if not that the event through sharing a static password. The system needs a stronger mechanism to reduce casual abuse (e.g., time-limited codes or verification steps).
- (2) **Advising and reporting:** Administrators and advisors need a career development report for a new implementation of a campus wide policy.
- (3) **FERPA-aware authentication:** The sign-in process must protect student records using role-based access (admin vs. student) and a verification-based login flow (e.g., email verification / 2FA-style codes, DUO NAU).
- (4) **Policy enforcement:** SICCS policies determine which courses can receive credit, when check-in is allowed, and how often certain event types can be applied (e.g., one-per-course-per-semester rules). These policies must be enforced before an attendance record is accepted.

2.3 Technology Stack

This project relies on several tools and languages to support development and reproducibility. Dependency versions are recorded in requirements.txt (back end) and package.json (front end). Table 1 summarizes the primary components.

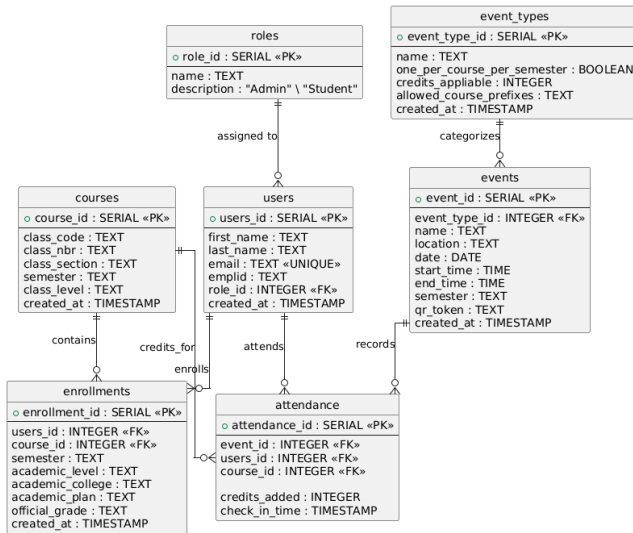
Table 1: Primary implementation components.

Component	Implementation
Front end	Vue.js (version in package.json)
Back end	Python + FastAPI (version in requirements.txt)
Database	PostgreSQL
Auth	Role-based access (admin vs. student) + email
Deployment	AWS w Uvicorn; DB hosted via PostgreSQL

3 Database

3.1 Database Design

The database stores and enforces the core relationships needed for a policy-aware attendance system. PostgreSQL is used as the relational database management system (RDBMS).


Figure 1: Data base design visualization using PlantUML
Table 2: Core database entities and their purpose.

Entity	Purpose
Users	Students and administrators with roles that control permissions.
Courses	The set of courses eligible to receive extra credit.
Enrollments	Links students to courses for a specific semester.
Event Types	Defines categories (e.g., industry night, workshop) and related policy parameters.
Events	Scheduled instances with dates and valid check-in windows.
Attendance	Audit record linking a student, an event, and the course receiving credit.

Because students can enroll in many courses and courses contain many students, the design requires a many-to-many relationship

between users and courses, represented by an enrollments table. Similarly, students can attend many events, and events include many students, represented by an attendance table.

To enforce correctness, the schema relies on foreign keys to prevent invalid references and uniqueness constraints to prevent duplicate claims (e.g., a student cannot claim the same event twice).

3.2 Database Initialization Workflow

The database is created and populated through three artifacts: a **schema (DDL) file** that defines tables and constraints, a **seed file** that inserts baseline data (e.g., event types and an initial administrator), and an **initialization script** that resets the database and ingests enrollment data to create real users, courses, and enrollments.

4 Front End

The front end is a lightweight web client built with **Vue 3 + Vite + TypeScript**. Its primary responsibility is to provide a clear, low-friction interface for two distinct roles—**administrators** and **students**—while leaving enforcement of sensitive rules (FERPA-aware access, eligibility, and attendance policies) to the back end and database. In practice, the client collects user input through forms and selections, communicates with the API using `fetch()`, renders responses as pages and dashboards, and surfaces validation errors in a way that is easy to understand during live check-in.

Because the same site serves both roles, navigation is role-gated. After authentication, the client resolves the current user context (including role) and conditionally exposes routes and actions. This avoids accidental disclosure of administrative tools while keeping the student experience fast and minimal.

Table 3: Front-end responsibilities by role.

Role	Primary responsibilities
Administrator	Semester setup (enrollment ingestion and validation), event lifecycle management (create/update/cancel), attendance review and auditing, and report export for advising and policy tracking. The interface also supports event check-in setup by generating QR codes or event codes designed for quick in-person use.
Student	Simple discovery and check-in: viewing active and upcoming events, submitting attendance during a valid window (via QR scan or code entry), selecting an eligible course to apply credit, and reviewing personal attendance history for transparency.

4.1 Administrator Workflow

Administrator interactions are organized around two time scales: start-of-semester setup and day-of-event operations. At the beginning of a term, the interface supports uploading an enrollment spreadsheet and provides feedback if required fields are missing or

if rows fail validation. Once enrollment data is ingested, administrators use the same UI to create and maintain events, including setting the check-in window that the back end will enforce.

During events, the administrative view emphasizes speed and auditability. Administrators can open an event, monitor attendance totals, and generate the check-in artifact used on-site (e.g., a QR code or short code). After the event, the interface supports reviewing who checked in, exporting summaries, and producing advising-oriented reports that reflect policy constraints enforced by the database and API.

4.2 Student Workflow

The student experience is intentionally compact to reduce friction during live check-in. Students authenticate, land on a dashboard showing relevant events, and complete check-in by scanning a QR code or entering an event code. When submitting attendance, the interface prompts the student to select an eligible course so that the attendance record is tied to the correct incentive target.

After check-in, students can view their attendance history and course context. This provides transparency (what was recorded and when) without exposing administrative controls or other students' data.

4.3 Implementation Notes

Client-side routing is handled with vue-router, and pages are implemented as views under `src/views/`. Session state is represented by a stored `session_id`; on load, the client queries `/auth/me` to determine identity and role, and it uses that result to restrict navigation to administrator-only routes. Across pages, API interactions follow a consistent pattern: submit a request with `fetch()`, update local state from the response, render the result using consistent UI components, and display server-provided errors in context. Shared components are used to standardize layout and messaging and to maintain a consistent look aligned with NAU branding.

5 Back End

The back end is implemented as a FastAPI service that exposes a JSON API to the Vue client and enforces all sensitive rules before any record is written to the database. Its design prioritizes (1) FERPA-aware access control, (2) non-shareable event check-in, and (3) policy-consistent attendance records that can be audited and reported at the end of a semester.

5.1 Service Responsibilities and Architecture

The API acts as the system's policy boundary. The front end may guide the user through workflows, but the back end is the trusted layer that validates identity, checks eligibility, enforces time windows, and produces normalized database writes. Implementation follows a common pattern: request schemas and response schemas define the public contract, and each route performs input validation, authorization checks, policy checks, and then a transactional database operation.

5.2 Secure Check-In via Rotating QR Codes

A primary design goal is to prevent "casual abuse" where a static password can be shared with non-attendees. There are many ways

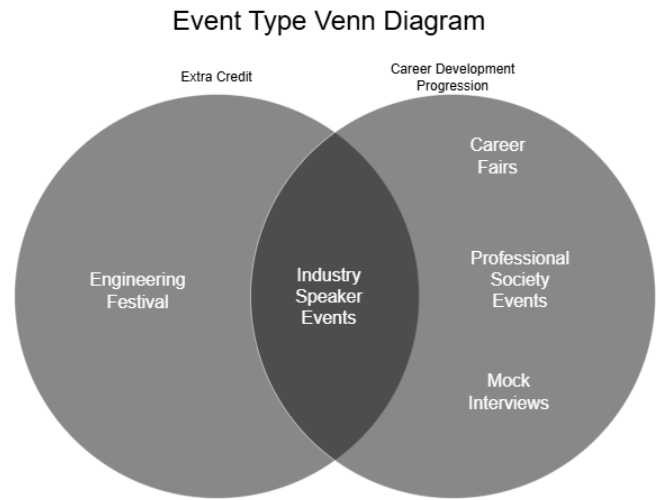


Figure 2: Data base design visualization using PlantUML

to achieve a simple working method but it is complex to make a unbreakable password. We selected to use rotating QR codes.

During an event, an administrator displays a QR code that changes on a fixed interval. When scanned, the client submits the token to the back end. The server validates that the token is (i) unexpired, (ii) associated with the correct event, and (iii) presented during the event's allowed check-in window. The goal is to expire quickly so they bind check-in to real-time presence at the event.

5.3 FERPA-Aware Authentication and Role Authorization

Because enrollment data and attendance records are FERPA-sensitive, the back end enforces role-based authorization for every protected operation. All users authenticate using a emailed verification-code "2FA-style". After a code is verified, the server issues a session identifier that the client presents on subsequent requests. We check for every admin process (e.g., creating event types, uploading enrollments, exporting reports) that they are restricted to administrator accounts.

5.4 Advising and Reporting for the Career Development Policy

A second core requirement is administrative report for advisors to check what events a student has attended. The back end supports report generation by aggregating attendance records per student and semester, computing both (a) extra-credit points where applicable and (b) milestone progress for advising.

The Professional Events Milestone is modeled as a semester-aware requirement: students must complete at least 8 professional events before graduation, including at least one event in each category (career fairs, mock interviews, professional society events, and industry speaker events). Engineering Fest and Industry Nights additionally award points (3% and 2%, respectively).

6 Current Results and Discussion

At the time of writing, the system has not yet been deployed at a live SICCS event; therefore, evaluation focuses on rule correctness and end-to-end workflow validation. Using seeded enrollment and event data, we executed scenario-based tests aligned with SICCS policies (e.g., preventing duplicate attendance, enforcing event time windows, restricting eligible course prefixes, and applying one-per-course-per-semester rules for selected event types). Across tested scenarios, the system produced normalized attendance records and prevented invalid submissions via back-end validation and database-level integrity constraints. These results indicate the system is

production-ready for a controlled pilot deployment, after which usability and adoption metrics can be collected.

Future work This application will be tested at SICCS's first spring 2020 event an insduty talk at the du bois center. we will also provide a physical sign in sheet for redundancys.

1

References

¹Replication package (data-processing scripts and evaluation pipeline): <https://github.com/Coupur/developersInactivityPrediction>.