

A Unified Web-Based Smart Campus ERP Platform: Building Fourteen Integrated Modules with Next.js, Express.js, and MySQL

Author : Niveditha G

MCA 4th sem GM University,Davangere

Guide:Krupa Rani S

Department:Master of Computer Applications, GM Univesity, Davangere,Karnataka, India

Abstract—Colleges and universities coordinate a diverse mix of back-office and student-life activities—ranging from admissions and course delivery to fee handling, hostel life, library circulation, transport, and health—that are commonly served by loosely coupled or paper-driven tools. The work reported here introduces a Smart Campus ERP built as one cohesive web application that brings fourteen such workflows under a shared, role-aware interface. Its runtime is a three-tier stack: a React front end delivered by Next.js, a stateless collection of REST endpoints on Express.js, and a MySQL store organized around a normalized relational schema. Sign-in is protected by a lightweight bearer-token scheme that distinguishes four user classes—administrators, teaching staff, students, and parents—and dispatches views accordingly. We describe how the layers cooperate, how the schema is laid out, how each module is decomposed, and how the API is shaped, and we close with a candid discussion of the design compromises together with a roadmap covering horizontal scaling, durable session storage, and analytics driven by machine learning.

Keywords—smart campus, integrated ERP, JSON over HTTP, Next.js, Express.js, MySQL, RBAC, layered web architecture

I. INTRODUCTION

It is common for tertiary institutions to operate a patchwork of independent tools for admissions intake, timetabling and grading, treasury operations, hostel or library duties, and campus-support functions such as buses, dining, athletics, and clinical care. When these silos multiply, integration cost climbs, the same data is captured more than once, and the experience presented to administrators, instructors, students, and guardians drifts out of alignment. Consolidating institutional processes inside a single, centrally governed platform—with views tailored to the caller's role but grounded in one shared model—offers a direct remedy that we refer to as a Smart Campus ERP.

The present paper documents one such platform, delivered end-to-end as a browser-based application. Its interface is realized on top of Next.js and React using the App Router pattern; its service tier is a REST layer written for Express.js; and its long-lived state is kept in MySQL, reached through a

pooled driver. Four categories of user—administrator, instructor, learner, and parent—interact with fourteen feature groups covering academic operations, treasury and admissions, library and hostel logistics, dining, athletics, personal grooming services, buses, medical visits, alerts, bulletins, an audit-style activity stream, and rolled-up analytics.

The remainder of the paper is arranged as follows: Section II surveys prior campus-management research; Section III sketches the overall runtime architecture; Section IV walks through the module breakdown; Section V presents the relational design; Section VI reports on implementation particulars, including sign-in flow and the REST surface; Section VII reflects on outcomes and trade-offs; Section VIII catalogs current shortcomings and planned enhancements; and Section IX offers concluding remarks.

II. Problem Statment

Institution-management software occupies a well-explored niche within enterprise web engineering. The first generation of college-focused portals concentrated on moving pupil registers and mark sheets on-line, most often as monolithic, server-rendered code bases with only rudimentary separation of privileges [1]. Follow-on studies broadened this scope to university-scale administration by folding admissions and financial workflows into the same platform, and repeatedly pointed to synchronization and permission control as the persistent pain points [2].

From an architectural angle, the REST style has, over the last two decades, become the default way to keep clients and servers loosely coupled so that either side may evolve on its own cadence [3]. It is now increasingly common for both tiers to share a JavaScript runtime—typically an Express.js service [5] paired with a React-driven framework such as Next.js [4]—which lowers cognitive switching cost and permits shared tooling across the team. For persistent institutional records, relational engines such as MySQL [6] still dominate because they are mature, ACID-compliant, and align naturally with the tabular data typical of administrative operations.

Restricting features by user category through role-based access control (RBAC) is a long-standing pattern in business-oriented information systems [8], and earlier work on campus tooling has consistently argued for separating administrative

and student-facing surfaces. Newer contributions look further, describing multi-tenant, cloud-hosted smart-campus stacks that reach beyond core records into ancillary services such as transportation and health [9]. Our platform draws on these accumulated conventions—REST decoupling, a full-JavaScript toolchain, a relational back-end, and RBAC—but expands the functional envelope to fourteen tightly integrated modules living inside a single repository, and it grounds the underlying schema in classical entity-relationship modeling [10].

III. SYSTEM ARCHITECTURE

The runtime obeys the familiar three-tier convention, isolating presentation, service, and storage concerns, as depicted in Fig. 1. Because the layers are decoupled and only exchange JSON over HTTP, the client and the API can be released, versioned, and scaled on independent tracks.

A. Presentation Layer

On the browser side, Next.js 16 with React 19 provides the rendering framework, laid out under the App Router convention. One shell component (`app/page.js`) is responsible for switching between navigation trees and views suitable for administrators, instructors, students, and parents, while a separate route (`app/login/page.js`) hosts the sign-in flow. Every network call originating from the UI passes through a single helper module (`api-client.js`) that wraps `fetch`, appends the current bearer token, and surfaces one function per resource. Keeping the transport layer in a single place lets the visible components stay declarative and makes it straightforward to swap the fetching strategy later.

B. Application Layer

A lone Express.js process (`server.js`) constitutes the service tier and publishes the REST surface on a dedicated port. CORS handling and JSON body parsing are declared as global middleware. Route files are grouped by concern—sign-in, students, faculty, library, treasury, alerts, bulletins, admissions, hostel, dining, athletics, salon, buses, health, and analytics—and each handler validates its input, dispatches a parameterized query, and returns JSON. A small helper appends significant events to a rolling window of the last eight entries, which feeds the administrator activity view.

C. Authentication and Session Management

Sign-in is credential-driven: the login handler checks the submitted address and secret against the users table, and, on success, mints a 32-byte hexadecimal bearer token from a cryptographically secure random source. Token metadata—user identity, role, and role-specific foreign keys—is retained in an in-memory map. Guarded routes flow through a middleware layer that verifies the Authorization header and attaches the resolved session object to the request. The scheme keeps sign-in cheap while sparing the deployment the operational cost of running a separate identity provider.

D. Data Layer

Persistence is handled by a MySQL instance reached through a connection pool built on `mysql2/promise`. A

migration utility creates the schema and seeds it from a canonical JSON dataset, ordering the CREATE statements to respect foreign keys. Attributes whose cardinality or shape varies—a learner's per-subject scores, or the list of stops along a bus route—are held as native JSON columns and marshalled at the boundary of the API, blending the flexibility of a document representation for nested fields with relational integrity for first-class entities.

IV. FUNCTIONAL MODULE DECOMPOSITION

Functionality is partitioned across fourteen domain modules, listed in Table I. The academic and back-office core (Academics, Admissions, Finance) is surrounded by service-oriented modules for hostel life, library, dining, athletics, salon appointments, buses, and health visits, together with horizontal facilities covering alerts, bulletins, a global activity stream, and rolled-up analytics.

TABLE I



Module	Description
Academics	Manage student profiles, attendance, grades, and academic records.
Admissions	Handle forms, admit cards, and enrollment processes.
Finance & Fees	Recording of fees, payment schedules, and scholarship management.
Library	Manage book inventory, loans, and digital resources.
Hostel	Room allocation, mess plans, and hostel fee management.
Cafeteria	Ordering, payment, and inventory management.
Sports	Manage sports events, registrations, and facilities.
Salon	Booking of appointments, staff management, and payment options.
Transport	Route and stops, vehicle scheduling, and fare management.
Health Centre	Manage patient records, appointments, and health services.
Notifications	Push, email, and in-app notifications for various events.
Announcements	Manage and publish important campus-wide messages.
Activity Feed	A central stream of events and updates from various modules.
AI Insights / Analytics	Dashboard showing key metrics like student performance, financial trends, and library usage.

A uniform implementation recipe is applied to every module on the server: GET handlers fetch rows and, where appropriate, deserialize JSON-typed columns; POST handlers first mint a readable, prefix-based identifier (STU-001 for a student, TXN-00000 for a transaction, and so on) and then insert; and status-transition handlers advance the record's state and append an entry to the activity log. This regularity keeps the cost of introducing new modules low.

V. DATABASE DESIGN

The schema is composed of twenty tables. Sign-in credentials live in a central users table, which relates by foreign key to a students table and, downstream, to domain tables such as transactions, library loans, hostel allocations, and facility reservations. Normalization is applied conventionally, but attributes whose cardinality changes between rows—grade trends, per-subject marks, and the stops along a bus route—are stored inside JSON columns rather than being broken out into narrow join tables. This keeps queries for the corresponding

views concise while sparing the schema an inflation of auxiliary tables holding data that is always read and written together.

Cross-table dependencies are enforced from application code: for example, approving an admission simultaneously creates a matching student record and issues a fresh user account with generated credentials, extending both the academic and authentication domains through a single administrative click. The migration script honors these dependencies, dropping and recreating child tables such as hostel visitors and library fines ahead of their parents whenever the database is reset.

VI. IMPLEMENTATION

The service layer runs on Express.js 4 and publishes more than sixty REST endpoints across the modules already listed; representative groupings are shown in Table II. Every database interaction uses parameterized statements to blunt SQL injection, and mutating handlers are enclosed in try/catch blocks that emit a JSON error body together with an HTTP status code when something goes wrong. A dedicated analytics endpoint rolls up counts and sums drawn from several tables—active enrolments, teaching-staff headcount, fees collected versus fees invoiced, books currently out on loan, admissions awaiting review, and unread notifications—so that the summary dashboard needs only a single round trip.

TABLE II

Module	Primary Responsibility
Academics	Manage academic policies, attendance, course administration, and grade management.
Admissions	Manage admissions, apply forms, and admission status.
Finance & Fee	Managing of fees, payment, receipt, and fee collection.
Library	Managing library, book borrowing, and book return.
Hostel	Hostel management and hostel allocation.
Campus	Managing campus, hostel, and other facilities.
Transport	Managing transport, vehicle, and other facilities.
Open	Managing open, hostel, and other facilities.
Staff	Managing staff, hostel, and other facilities.
Health Center	Managing health center, hostel, and other facilities.
Notification	Managing notification, hostel, and other facilities.
Attendance	Managing attendance, hostel, and other facilities.
Library/Fine	Managing library, hostel, and other facilities.
Analytics	Managing analytics, hostel, and other facilities.

On the client, the API helper module offers exactly one function per resource operation—for instance `fetchStudents`, `createTransaction`, or `approveAdmission`—returning the parsed body or raising an exception on any non-2xx response. The

view layer therefore sees a promise-oriented, one-to-one mirror of the REST surface, and the mapping between UI intent and backend effect stays direct.

VII. RESULTS AND DISCUSSION

The finished platform shows that an extensive slate of administrative and student-life responsibilities—covering course records, treasury, and seven distinct facility services—can live inside a single, well-organized full-stack repository without demanding a break-up into microservices. Reusing the same per-module recipe (list, create, update, transition) keeps the API predictable in spite of its breadth, and holding shape-varying attributes in JSON columns avoids fragmenting the schema while preserving relational integrity for the core entities.

The chosen approach carries the trade-offs typical of an early-stage build. Sign-in state is retained inside an in-memory map bound to a single process, which means that a restart wipes active sessions and multiple back-end replicas cannot share them; a production posture would move sessions into a shared store such as Redis or move to stateless signed tokens (for example, JWT). In the same vein, the current login handler performs a plain equality check on the stored password rather than a salted-hash comparison, and this would need to be replaced before the platform could be considered production-grade from a security standpoint.

VIII. LIMITATIONS AND FUTURE WORK

Several enhancements would meaningfully harden the platform for real-world use. First, credential storage should be moved to a salted, adaptive hashing scheme (bcrypt or Argon2 are natural picks) in place of plaintext matching. Second, session storage should migrate to a durable, shareable back-end so that the service tier can be scaled horizontally and can survive restarts without evicting users. Third, the analytics module could grow from descriptive rollups toward predictive signals—forecasting shortfalls in fee collection, or flagging students at academic risk from combined attendance and grade-trend evidence—which is the direction implied by the “AI Insights” entry point already present in the navigation. Finally, systematic automated tests, explicit API versioning, and dedicated input-validation middleware would keep the code base resilient as the module count keeps rising.

IX. CONCLUSION

We have presented the design and construction of a modular Smart Campus ERP platform that consolidates fourteen administrative and student-life functions behind a single role-aware web surface. Built on a three-tier stack—a Next.js/React client, an Express.js REST tier, and a MySQL store—the system demonstrates that a broad institutional feature set can be delivered through a consistent, extensible module pattern. While the current build deliberately favors functional breadth and architectural clarity, the security and scalability items highlighted above sketch a concrete path toward a production-ready deployment.

REFERENCES

- [1] S. Bhardwaj, R. Kumar, and A. Sharma, "A review on college management system using web technologies," *Int. J. Comput. Appl.*, vol. 178, no. 32, pp. 1–5, 2019.
- [2] M. A. Rahman, S. Islam, and T. Ahmed, "Design and implementation of a university management system," in *Proc. Int. Conf. Comput. Commun. Chem. Mater. Electron. Eng. (IC4ME2)*, 2018, pp. 1–4.
- [3] R. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, Univ. of California, Irvine, CA, USA, 2000.
- [4] Vercel Inc., "Next.js documentation," 2024. [Online]. Available: <https://nextjs.org/docs>
- [5] OpenJS Foundation, "Express.js—Node.js web application framework," 2024. [Online]. Available: <https://expressjs.com>
- [6] Oracle Corp., "MySQL 8.0 reference manual," 2024. [Online]. Available: <https://dev.mysql.com/doc>
- [7] M. Fowler, *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley, 2002.
- [8] A. Gupta and N. Verma, "Role-based access control for campus information systems," *Int. J. Eng. Res. Technol.*, vol. 9, no. 6, pp. 112–116, 2020.
- [9] J. Kabir and T. Rahman, "Cloud-based smart campus architecture for higher education institutions," in *Proc. IEEE Int. Conf. Innov. Sci. Eng. Technol. (ICISSET)*, 2021, pp. 1–6.
- [10] P. Chen, "The entity-relationship model—toward a unified view of data," *ACM Trans. Database Syst.*, vol. 1, no. 1, pp. 9–36, 1976.