CourseLedger

CourseLedger

Live demo: https://student-registration-portal-plq6.onrender.com
(Free-tier hosting — the first load may take 30–60 seconds if the app has been idle.)

Repository: https://github.com/syal00/Student-Registration-portal

Course: CST8256 — Web Programming Languages I

Overview

CourseLedger is an academic management system built with ASP.NET Core MVC. It lets an administrator manage a small college's core records:

- **Courses** — catalog with code, title, description, weekly hours, and fee
- **Students** — student registry
- **Academic Records** — grades linking students to courses
- **Employees** — staff records with job role assignments
- **Roles** — job titles (Administrator, Instructor, Staff)

This is a demo/coursework application with **no login system** — all pages are publicly viewable by design, as a scope decision for the assignment rather than an oversight. See "Known Limitations" below.

Who This Is For & Where It Applies

CourseLedger was built as a coursework project, but the underlying pattern — a small, structured admin system for managing people, offerings, and the records that connect them — shows up constantly in real organizations. The same core model (catalog + participants + linking records + staff/roles) maps directly onto:

Field	How this pattern applies
Education	Course catalogs, student registries, gradebooks — the exact domain this project targets
Corporate training / L&D	Training modules ("courses"), employees ("students"), completion records ("grades"), and instructor/admin roles
Certification bodies	Exam/course offerings, candidates, pass/fail records, and staff who administer them
Membership organizations	Programs or workshops, members, attendance/completion tracking, staff roles
Small nonprofits / community centers	Class or program scheduling, participant rosters, outcome

	tracking
Internal ops tooling	Any “catalog of things + people who go through them + who administers it” workflow — one of the most common shapes of internal business software

The point of showing this in a portfolio isn’t just “I built a gradebook” — it’s demonstrating the ability to design a relational data model (one-to-many and many-to-many relationships, composite keys, referential integrity), build full CRUD around it, and ship it as a real, live, deployed application end to end.

Skills This Project Demonstrates

- **Relational database design** — normalized schema, composite primary keys, many-to-many relationships via join tables, foreign key constraints
- **Full-stack MVC development** — Razor views, server-side rendering, model binding, and validation using ASP.NET Core’s built-in patterns
- **Custom validation logic** — a hand-written validation attribute (two-word name format) beyond what built-in annotations provide
- **Custom sorting/comparison logic** — a purpose-built comparer for multi-field, nullable-aware sorting
- **Responsive, accessible UI design** — a from-scratch dark theme with a defined color system, not a default template
- **Real deployment experience** — containerizing the app with Docker, migrating the data layer from SQL Server to PostgreSQL for hosting compatibility, and deploying to a live cloud environment with a managed database
- **Automated testing** — unit tests around the trickiest logic (sorting, custom validation) rather than testing everything superficially

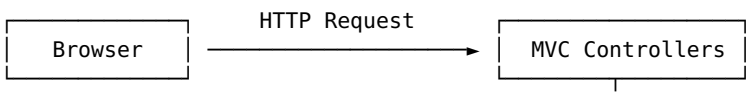
---

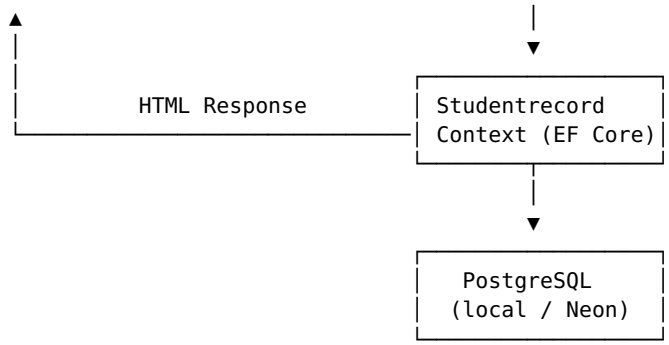
Tech Stack

Layer	Technology
Framework	ASP.NET Core 8 MVC (Razor views)
ORM	Entity Framework Core 8
Database	PostgreSQL (hosted on Neon)
UI	Bootstrap 5, custom dark theme
Fonts	Inter, JetBrains Mono
Testing	xUnit
Deployment	Docker container on Render

Architecture

Browser requests are handled by MVC controllers, which read/write data through EF Core (`StudentrecordContext`) to a PostgreSQL database. Razor views render server-side and are returned as HTML — there’s no separate frontend framework or API layer.





Default routing follows the standard `{controller=Home}/{action=Index}/{id?}` pattern, e.g. `/Courses/Edit/CST8256`.

Project Structure

```
/                                    ← Repository root
├── Dockerfile                           ← Docker build for Render
├── Procfile                             ← Heroku-style start
command
├── render.yaml                          ← Render blueprint (Docker
runtime)
├── railway.json                         ← Alternate Railway deploy
config
├── README.md
├── CourseLedger/                        ← Main web application
│   ├── CourseLedger.csproj
│   ├── Program.cs                       ← App entry point,
seeding, middleware
│   ├── appsettings.json
│   ├── appsettings.Development.json
│   ├── Properties/launchSettings.json
│   ├── Controllers/                     ← 6 controllers
│   ├── DataAccess/                      ← 5 entities + DbContext
│   ├── Models/                          ← Validators, comparers,
error VM
│   ├── Migrations/                      ← EF migration history
│   ├── Views/                           ← Razor views per
controller
│   └── wwwroot/                         ← Static files (CSS, JS,
libs)
└── CourseLedger.Tests/                  ← xUnit test project
    ├── AcademicRecordComparerTests.cs
    └── NameFormatAttributeTests.cs
```

Data Models

The schema has five core entities plus one join table, designed around a central many-to-many relationship (Students ↔ Courses, via grades) and a secondary many-to-many relationship (Employees ↔ Roles).

```
Course ||--o{ AcademicRecord : "has grades for"
Student ||--o{ AcademicRecord : "receives grades in"
Employee }o--o{ Role : "assigned via Employee_Role"

Course {
    string  Code        PK
    string  Title
    string  Description
```

```
    int     HoursPerWeek
    decimal FeeBase
}
Student {
    string  Id          PK
    string  Name
}
AcademicRecord {
    string  StudentId   PK, FK
    string  CourseCode  PK, FK
    int     Grade
}
Employee {
    int     Id          PK
    string  Name
    string  UserName    UK
    string  Password
}
Role {
    int     Id          PK
    string  Role
}
Employee_Role {
    int     Employee_Id PK, FK
    int     Role_Id     PK, FK
}
```

Design notes: - `AcademicRecord` is the classic “association entity with extra data” — it doesn’t just link a Student and a Course, it also carries the `Grade` itself, so it can’t be a plain join table. - `Employee` ↔ `Role` is a pure many-to-many relationship (an employee can hold multiple roles, a role can belong to multiple employees), resolved through the `Employee_Role` join table with a composite key. - `UserName` on `Employee` carries a **unique constraint** at the database level, not just application-level validation.

Database Design

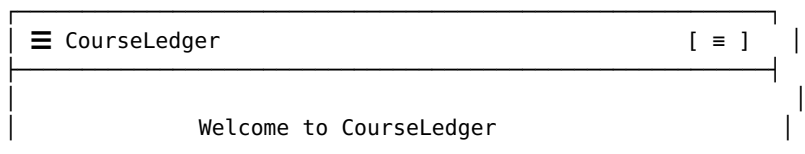
Six tables: `Course`, `Student`, `AcademicRecord`, `Employee`, `Role`, and a many-to-many join table (`Employee_Role`).

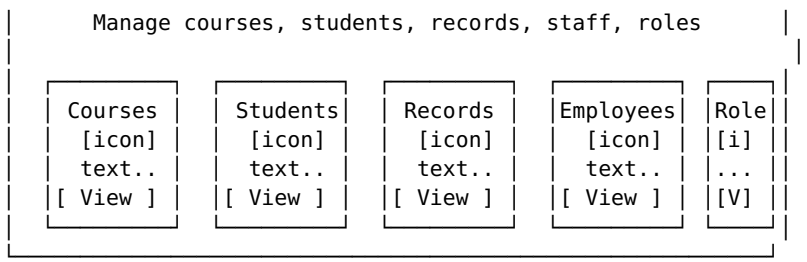
- `AcademicRecord` uses a **composite primary key** (`StudentId`, `CourseCode`) — one grade per student per course.
- `Employee.UserName` has a **database-level unique constraint**.
- Deletes are handled at the application level (related child records are removed in code before a parent record is deleted), rather than relying on database cascade rules.

Wireframes

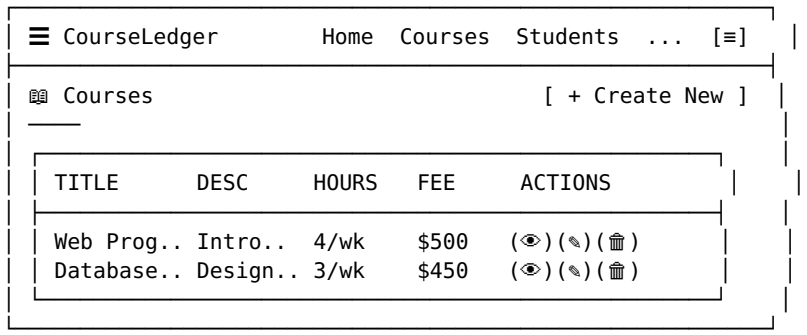
Rough layout of the two most representative screens — the homepage and a data table view. Every list screen (Courses, Students, Academic Records, Employees, Roles) follows the same table/mobile-card pattern shown below.

Homepage layout

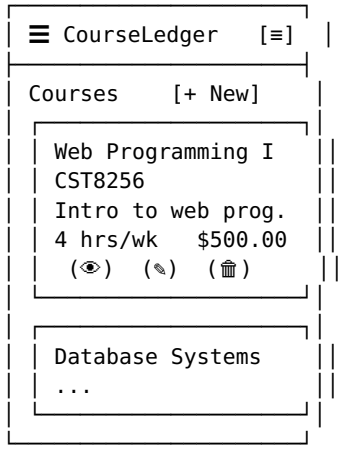




List/table screen layout (desktop ≥ 992px)



Same screen, mobile layout (< 992px) — rows collapse into stacked cards instead of a horizontal table, and the navbar collapses into a hamburger menu:



UI Elements & Design System

Color system (CSS custom properties, applied consistently app-wide):

Variable	Color	Used for
`--bg-base`	`#0B0F19`	Page background
`--bg-elevated`	`#131826`	Cards, navbar, table rows
`--brand-primary`	`#7C3AED`	Primary buttons, headings, links
`--accent-info`	`#22D3EE`	View actions, informational badges
`--accent-success`	`#2DD4BF`	Passing grades, positive states
`--accent-danger`	`#F87171`	Delete actions, failing grades
`--accent-warning`	`#FBBF24`	Borderline grades, warning states
`--text-primary`	`#F1F5F9`	Main body text
`--text-muted`	`#94A3B8`	Secondary/labels text

Reusable components:

Component	Description
Feature cards	Icon + heading + description + action button, used on the homepage
Circular action buttons	Outline-to-fill icon buttons for view/edit/delete, color-coded by action type
Grade badges	Colored pill labels showing pass/warning/fail/ungraded at a glance
Delete confirmation modal	Shared dark-themed modal that dynamically fills in the item name and delete URL
Empty state	“Nothing here yet” placeholder with an optional call-to-action button
Mobile card list	Auto-swaps in for tables below the 992px breakpoint

---

Typography

Use	Font	Notes
Body text, headings, labels	**Inter** (Google Fonts)	Clean, highly legible sans-serif used throughout the UI
IDs and codes (student IDs, course codes)	**JetBrains Mono**	Monospace font applied specifically to identifiers, so codes like `CST8256` or `S001` are visually distinct from regular prose

Font weights lean toward bold for headings and section titles, and regular weight for body copy and table content, keeping a clear visual hierarchy without introducing additional typefaces.

Key Features

- Full CRUD for Courses, Students, Academic Records, Employees, and Roles
- Custom validation, including a two-word name format rule for employee names
- Sortable, color-coded grade list (pass/warning/fail/ungraded) with a bulk grade editor
- Responsive design — tables collapse into stacked cards on mobile
- Custom dark theme with a consistent color system across the whole app
- Delete confirmation modals instead of native browser confirm dialogs

---

How to Use It

The app is organized around five modules, all reachable from the homepage or the top navigation bar.

Courses — the catalog of offerings. Create a course by entering a code, title, description, weekly hours, and base fee. Every course row supports view, edit, and delete actions, with a confirmation step before anything is removed. Deleting a course also clears out any academic records tied to it, so grade data never ends up pointing at a course that no longer exists.

Students — the registry of people enrolled. Students are added with a manually assigned ID and a name. Editing a student only allows changing the name — the ID stays fixed once created, the same way a real student number wouldn't change over time. Deleting a student first clears their academic records, then removes the student record itself.

Academic Records — where courses and students connect. This is the heart of the data model: each record pairs one student with one course and (optionally) a grade. The list view is sortable by course title or student name in either direction, and grades are color-coded — teal for passing, amber for borderline, red for failing, and gray for not yet graded — so patterns are visible at a glance. There's also a bulk editor for updating many grades in a single pass, rather than opening each record individually.

Employees — the staff who administer the system. Each employee has a name (validated to require a first and last name), a network-style username (checked for uniqueness), a password field (present in the schema but not used for login, since this build has no authentication layer), and one or more assigned job roles via checkboxes.

Roles — the job titles employees can hold. A lightweight lookup table (Administrator, Instructor, Staff, etc.) that feeds the role checkboxes on the Employees screens. Deleting a role clears it from any employees who held it before removing the role itself.

Across every module, deletions go through a confirmation modal rather than a plain browser pop-up, and empty tables show a clear "nothing here yet" message instead of a blank page.

Testing

The project includes an xUnit test suite covering the custom grade-sorting comparer and the employee name-format validator (10 tests total). Controller and integration-level tests are not yet included — see Known Limitations.

Running It Locally

```
git clone https://github.com/syal00/Student-Registration-portal.git
cd Student-Registration-portal
dotnet restore
dotnet run --project CourseLedger/CourseLedger.csproj
```

You'll need a local or cloud PostgreSQL instance and a connection string set in `appsettings.Development.json` or via an environment variable. On first run, EF Core migrations apply automatically and the database is seeded with sample data.

Deployment

Deployed as a Docker container on **Render**, connected to a **Neon** PostgreSQL database. The Dockerfile builds and publishes the app with the .NET 8 SDK, then runs it on the lightweight ASP.NET runtime image. Database credentials are supplied via environment variables at deploy time — never committed to the repository.

Known Limitations

This was scoped as a coursework assignment, and these are deliberate trade-offs rather than oversights:

- **No authentication** — all pages are publicly accessible; employee password fields exist in the schema but aren't used to gate access
- **No REST API** — server-rendered pages only, no JSON endpoints
- **No export/reporting** — no PDF, CSV, or print output
- **Manual ID entry** — student IDs and role IDs are entered by hand rather than auto-generated
- **Application-level cascades** — related records are cleaned up in code, not via database triggers
- **Free-tier hosting** — the live demo may take up to a minute to respond after a period of inactivity

---

Built as part of CST8256 — Web Programming Languages I.