Campus Management System

Full-Stack CRUD Application Project Document

Assignment:	Final Project - Full-Stack CRUD Application (Campus Management System)
Start Date:	November 18, 2025
Due Date:	December 12, 2025, 11:59 PM
Project Duration:	4 weeks
Team Size:	2 developers (David & Kevin)
Document Version:	1.0
Last Updated:	December 2, 2025

A PERN-based Campus Management System with CRUD functionality for campuses and students, client-side routing, and Redux state management.

Contents

1 Feature Requirements

1.1 Core Functional Views

1.1.1 Home Page View

- **Default Landing**: Users land on a visually pleasing home page by default.
- **Navigation**: Clear navigation options to access All Campuses and All Students views.
- **Branding**: Campus Management System title and brief description of the system.

1.1.2 All Campuses View

- **Campus List**: Display a list of all campuses, including at least campus name and image.
- **Empty State**: Show an informative message when no campuses exist.
- **Add Campus**: Provide a button/link to navigate to the Add Campus View.
- **Delete Campus**: Allow deletion of a campus via a button (either here or from Single Campus View).
- **Navigation**: Clicking a campus name navigates to the Single Campus View.

1.1.3 Single Campus View

- **Campus Details**: Show campus name, image, address, and description.
- **Enrolled Students**: Display a list of enrolled students' names, or a helpful message if none exist.
- **Student Navigation**: Clicking a student name navigates to that student's Single Student View.
- **Student Management**: Support adding new/existing students to the campus and removing students from the campus.
- **Campus Management**: Provide navigation to the Edit Campus View and an option to delete the campus.

1.1.4 Add Campus View

- **Form Inputs**: Fields for name, address, description, and image URL.
- **Validation**: Real-time validation with helpful error messages.
- **Persistence**: On valid submission, the new campus is persisted to the database and appears in the All Campuses View without a page refresh.

1.1.5 Edit Campus View

- **Pre-filled Form**: Existing campus information loaded into editable fields.
- **Form Inputs**: Fields for name, address, description, and image URL.
- **Validation**: Real-time validation similar to Add Campus View.
- **Update Behavior**: On valid submission, the campus is updated in the database and the UI reflects changes without a full page reload.

1.1.6 All Students View

- **Student List**: Display all students, including at least full name.
- **Empty State**: Show an informative message when no students exist.
- **Add Student**: Provide a button/link to navigate to the Add Student View.
- **Delete Student**: Allow deletion of a student via a button (either here or from Single Student View).
- **Navigation**: Clicking a student name navigates to the Single Student View.

1.1.7 Single Student View

- **Student Details**: Show full name, email, image, and GPA.
- **Campus Info**: Display the name of the student's campus, or a message if not enrolled.
- **Navigation**: Link to navigate to the student's campus Single Campus View.
- **Student Management**: Provide navigation to Edit Student View and an option to delete the student.

1.1.8 Add Student View

- **Form Inputs**: Fields for first name, last name, email, image URL, GPA, and optionally campus.
- **Validation**: Real-time validation for required fields, email format, and GPA range (0.0–4.0).
- **Persistence**: On valid submission, the new student is persisted and appears in All Students View without a page refresh.

1.1.9 Edit Student View

- **Pre-filled Form**: Existing student data loaded for editing.
- **Form Inputs**: Fields for first name, last name, email, image URL, GPA, and campus selection.
- **Validation**: Real-time validation consistent with Add Student View.
- **Update Behavior**: On submission, the student is updated in the database and the UI reflects the changes immediately.

1.2 Technical Requirements

1.2.1 UI (React)

- **All Campuses Component**: Displays list of campuses with name and image.
- **All Students Component**: Displays list of students with at least student name.
- **Single Campus Component**: Displays details for one campus and its students.
- **Single Student Component**: Displays details for one student and their campus.
- **Form Components**: Reusable form patterns for add/edit campus and add/edit student views with inline validation.

1.2.2 Client-Side Routing (React Router)

- **Route Definitions**:
  - `/` – Home page.
  - `/campuses` – All Campuses View.
  - `/students` – All Students View.
  - `/campus/:campusId` – Single Campus View.
  - `/student/:studentId` – Single Student View.
  - `/campuses/new` – Add Campus View.
  - `/students/new` – Add Student View.
  - `/campuses/:campusId/edit` – Edit Campus View.
  - `/students/:studentId/edit` – Edit Student View.
- **Navigation**: Header navigation links to Home, All Campuses, and All Students.
- **Deep Linking**: Direct URL access to any supported route.

1.2.3 State Management (Redux)

- **Campuses Reducer**: Manages the collection and loading/error states for campuses.
- **Students Reducer**: Manages the collection and loading/error states for students.
- **Thunks**: Asynchronous action creators for fetching, creating, updating, and deleting campuses and students.
- **Global Store**: Single Redux store combining campus and student reducers, integrated with React via Provider.

1.2.4 Database (PostgreSQL via Sequelize)

- **Campus Model**:
  - `name`: non-null, non-empty string.
  - `address`: non-null, non-empty string.
  - `description`: text, nullable.
  - `imageUrl`: string with default image URL, nullable.
- **Student Model**:
  - `firstName`: non-null, non-empty string.
  - `lastName`: non-null, non-empty string.
  - `email`: non-null, non-empty string with validation.
  - `imageUrl`: string with default image URL, nullable.
  - `gpa`: decimal between 0.0 and 4.0, nullable.
- **Associations**:
  - One Campus has many Students.
  - One Student belongs to at most one Campus.

1.2.5 API / Server-Side Routing (Express + Sequelize)

- **All Resources**:
  - `GET /api/students` – all students.
  - `GET /api/campuses` – all campuses.
- **Single Resources**:
  - `GET /api/students/:id` – single student including campus.
  - `GET /api/campuses/:id` – single campus including students.
- **Create**:
  - `POST /api/students` – add a new student.
  - `POST /api/campuses` – add a new campus.
- **Update**:
  - `PUT /api/students/:id` – edit existing student.
  - `PUT /api/campuses/:id` – edit existing campus.
- **Delete**:
  - `DELETE /api/students/:id` – delete student.
  - `DELETE /api/campuses/:id` – delete campus.

1.3 Version Control and Code Quality Requirements

- **Git Workflow**: Use feature branches, pull requests, and small, descriptive commits.
- **Code Organization**: Keep code modular, well-formatted, and commented.
- **Documentation**: README files for both front-end and back-end list team members and GitHub usernames.

2 Application Architecture Description and Diagram

2.1 Architecture Overview

The Campus Management System follows a **PERN (PostgreSQL, Express, React, Node.js) full-stack architecture** based on the assignment specification in the project PDF. The back-end exposes a RESTful API for managing campuses and students, while the frontend is a **React Single-Page Application (SPA)** using React Router and Redux. The architecture emphasizes **separation of concerns**, **unidirectional data flow**, and **scalable CRUD operations**.

2.2 System Components

2.2.1 Presentation Layer (React Components)

- **HomePageView**: Default landing page with navigation to main views.
- **AllCampusesView**: Displays the list of campuses.
- **CampusView**: Displays a single campus and its enrolled students.

- **NewCampusView**: Form for adding a new campus.
- **EditCampusView**: Form for editing an existing campus.
- **AllStudentsView**: Displays the list of students.
- **StudentView**: Displays a single student's details.
- **NewStudentView**: Form for adding a new student.
- **EditStudentView**: Form for editing an existing student.

2.2.2 Routing Layer (React Router)

- **Router**: `BrowserRouter` wraps the entire React application.
- **Routes**: Map URL paths to campus and student views.
- **Navigation**: `Link` components used for navigating between views.
- **Error Handling**: Fallback route for invalid URLs (e.g., 404 view).

2.2.3 State Management Layer (Redux Store)

- **Campuses Slice**:
  - `campuses`: Array of campus objects.
  - `selectedCampus`: Detailed campus for Single Campus View.
  - `status`: Loading/error statuses.
- **Students Slice**:
  - `students`: Array of student objects.
  - `selectedStudent`: Detailed student for Single Student View.
  - `status`: Loading/error statuses.

2.2.4 Business Logic Layer

- **CRUD Operations**: Thunks encapsulate create, read, update, and delete logic for both campuses and students.
- **Validation**: Shared validation utilities enforce constraints such as GPA range and required fields.
- **Association Management**: Logic for assigning and unassigning students to/from campuses.
- **Error Handling**: Graceful user feedback on API failures.

2.3 Architecture Diagram

2.4 Data Flow

1. **User Navigation**: User navigates to a view (e.g., All Campuses, Single Student) via header links or deep links.
2. **Route Matching**: React Router matches the URL path to a route and renders the correct view component.

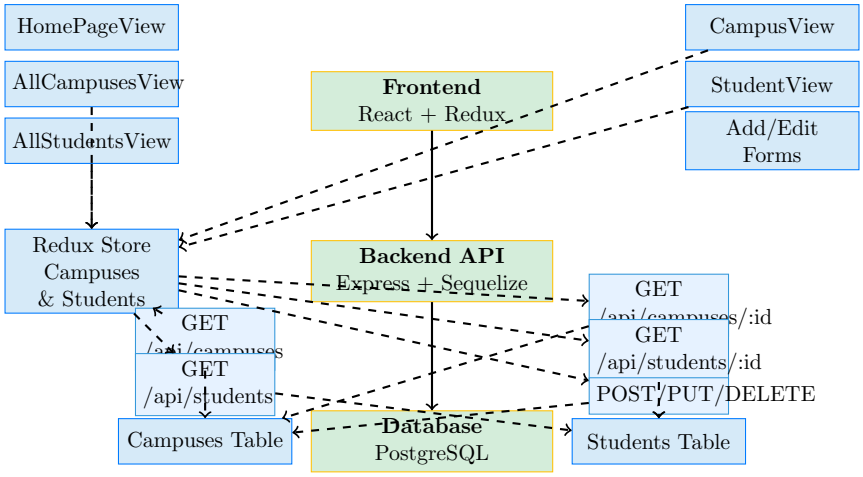

Figure 1: Campus Management System Application Architecture

3. **State Initialization**: Components dispatch Redux thunks to fetch initial data from the backend API.
4. **Backend Processing**: Express routes handle requests, interact with Sequelize models, and query/update PostgreSQL.
5. **Response Handling**: The backend responds with JSON data, which Redux reducers use to update the store.
6. **UI Update**: React components re-render with the latest state, reflecting changes without a page reload.
7. **Subsequent Actions**: User-initiated CRUD actions (add, edit, delete) follow the same flow through thunks and reducers.

2.5 Technology Stack

- **React**: Component-based UI library for building the SPA.
- **React Router**: Client-side routing for navigation between views.
- **Redux**: State management for campuses and students.
- **Node.js**: Runtime environment for the backend server.
- **Express**: Web framework for building RESTful API endpoints.
- **PostgreSQL**: Relational database for persistent data storage.
- **Sequelize**: ORM layer for interacting with PostgreSQL.
- **Git & GitHub**: Version control and repository hosting.

2.6 Key Design Patterns

- **Component Composition**: Break UI into reusable, focused components (views and containers).
- **Container/Presentational Split**: Containers handle data fetching and state, while views focus on layout and styling.

- **Unidirectional Data Flow**: Redux ensures predictable state updates with actions and reducers.
- **Separation of Concerns**: Backend API, database models, and frontend UI remain decoupled.
- **Validation and Error Handling**: Centralized validation logic to keep components simple and robust.

3 Epics, User Stories, and Acceptance Criteria

3.1 Epic 1: Home Page and Navigation

3.1.1 User Story 1.1: Home Page Navigation

As a user
I want to land on a home page with clear navigation
So that I can easily access campuses and students
Acceptance Criteria:

- Home page is rendered at route `/`.
- Page clearly presents links or buttons to All Campuses and All Students views.
- Navigation uses React Router `Link` components.
- Design is visually appealing and responsive.

3.2 Epic 2: All Campuses Management

3.2.1 User Story 2.1: View All Campuses

As a user
I want to see a list of all campuses
So that I can browse the available campuses
Acceptance Criteria:

- Route `/campuses` displays All Campuses View.
- Each campus card shows at least campus name and image.
- If no campuses exist, an informative message is displayed.
- Clicking a campus name navigates to its Single Campus View.

3.2.2 User Story 2.2: Add Campus from All Campuses View

As a user
I want to add a new campus from the All Campuses View
So that I can grow the list of campuses
Acceptance Criteria:

- A button or link navigates to the Add Campus View.
- After successfully creating a campus, the All Campuses list updates without a full page refresh.

3.2.3 User Story 2.3: Delete Campus

As a user
I want to delete a campus
So that I can remove campuses that are no longer active
Acceptance Criteria:

- Each campus has an accessible Delete button (in All Campuses or Single Campus View).
- Clicking Delete sends a request to remove the campus from the database.
- The campus disappears from the list without requiring a page reload.

3.3 Epic 3: Single Campus Management

3.3.1 User Story 3.1: View Single Campus Details

As a user
I want to view details about a specific campus
So that I can see its information and enrolled students
Acceptance Criteria:

- Route `/campus/:campusId` displays Single Campus View.
- View shows campus name, image, address, and description.
- View lists names of all enrolled students, or a message if there are none.

3.3.2 User Story 3.2: Navigate to Student from Campus

As a user
I want to click a student from a campus
So that I can see that student's details
Acceptance Criteria:

- Student names in Single Campus View are clickable.
- Clicking a student navigates to `/student/:studentId`.

3.3.3 User Story 3.3: Manage Students in a Campus

As a user
I want to add or remove students from a campus
So that I can manage enrollment
Acceptance Criteria:

- UI controls allow assigning existing students to the campus.
- UI controls allow removing students from the campus.
- Changes are persisted via backend routes and reflected in the UI without a page reload.

3.4 Epic 4: All Students Management

3.4.1 User Story 4.1: View All Students

As a user
I want to see a list of all students
So that I can browse student information
Acceptance Criteria:

- Route `/students` displays All Students View.
- Each student shows at least their full name.
- If no students exist, a helpful message is shown.
- Clicking a student name navigates to Single Student View.

3.4.2 User Story 4.2: Add Student from All Students View

As a user
I want to add a new student from the All Students View
So that I can register new students
Acceptance Criteria:

- A button or link navigates to Add Student View.
- After adding a student, the All Students list updates without a full page reload.

3.4.3 User Story 4.3: Delete Student

As a user
I want to delete a student
So that I can remove students who are no longer part of the system
Acceptance Criteria:

- Each student has a Delete button (in All Students or Single Student View).
- Clicking Delete removes the student from the database.
- The student disappears from the list without a page reload.

3.5 Epic 5: Single Student Management

3.5.1 User Story 5.1: View Single Student Details

As a user
I want to view details of an individual student
So that I can see their information and enrollment
Acceptance Criteria:

- Route `/student/:studentId` displays Single Student View.
- View shows full name, email, GPA, and image.
- If enrolled, the student's campus name is displayed; otherwise, a helpful message appears.

3.5.2 User Story 5.2: Navigate to Campus from Student

As a user
I want to navigate from a student to their campus
So that I can quickly see campus information
Acceptance Criteria:

- Campus name in Single Student View is clickable.
- Clicking the campus name navigates to `/campus/:campusId`.

3.6 Epic 6: Add & Edit Campus

3.6.1 User Story 6.1: Add Campus

As a user
I want to add a new campus via a form
So that I can register campuses in the system
Acceptance Criteria:

- Add Campus View includes fields for name, address, description, and image URL.
- Required fields cannot be submitted empty.
- On success, a new campus is persisted and visible in All Campuses View.

3.6.2 User Story 6.2: Edit Campus

As a user
I want to edit an existing campus
So that I can keep campus information up to date
Acceptance Criteria:

- Edit Campus View is accessible from Single Campus View.
- Form is pre-populated with current campus data.
- On success, updated data is persisted and Single Campus View reflects changes.

3.7 Epic 7: Add & Edit Student

3.7.1 User Story 7.1: Add Student

As a user
I want to add a new student via a form
So that I can register students in the system
Acceptance Criteria:

- Add Student View includes fields for first name, last name, email, image URL, GPA, and optional campus selection.
- GPA must be validated to fall within 0.0–4.0.
- Email must be in a valid format.
- On success, student is added to the database and appears in All Students View.

3.7.2 User Story 7.2: Edit Student

As a user
I want to edit an existing student's information
So that I can correct or update their details
Acceptance Criteria:

- Edit Student View is accessible from Single Student View.
- Form is pre-filled with the current student data.
- Changes to campus assignment are persisted and reflected in both Student and Campus views.

3.8 Epic 8: Code Quality, Git Workflow, and Submission

3.8.1 User Story 8.1: Maintain Clean Code and Git History

As a developer
I want to follow good version control and code organization practices
So that the project remains maintainable and easy to review
Acceptance Criteria:

- Feature branches are used for major features (e.g., `feat/campuses-crud`).
- Commits are small, frequent, and descriptive.
- Code is well-organized into logical directories (models, routes, components, containers, etc.).
- README files list team members and setup instructions, as required in the PDF.

4 Project Schedule Chart (Gantt Chart)

4.1 Project Timeline Overview

Project Duration: 4 weeks
Start Date: November 18, 2025
Due Date: December 12, 2025, 11:59 PM
Team Size: 2 developers (David & Kevin)

4.2 Detailed Gantt Chart

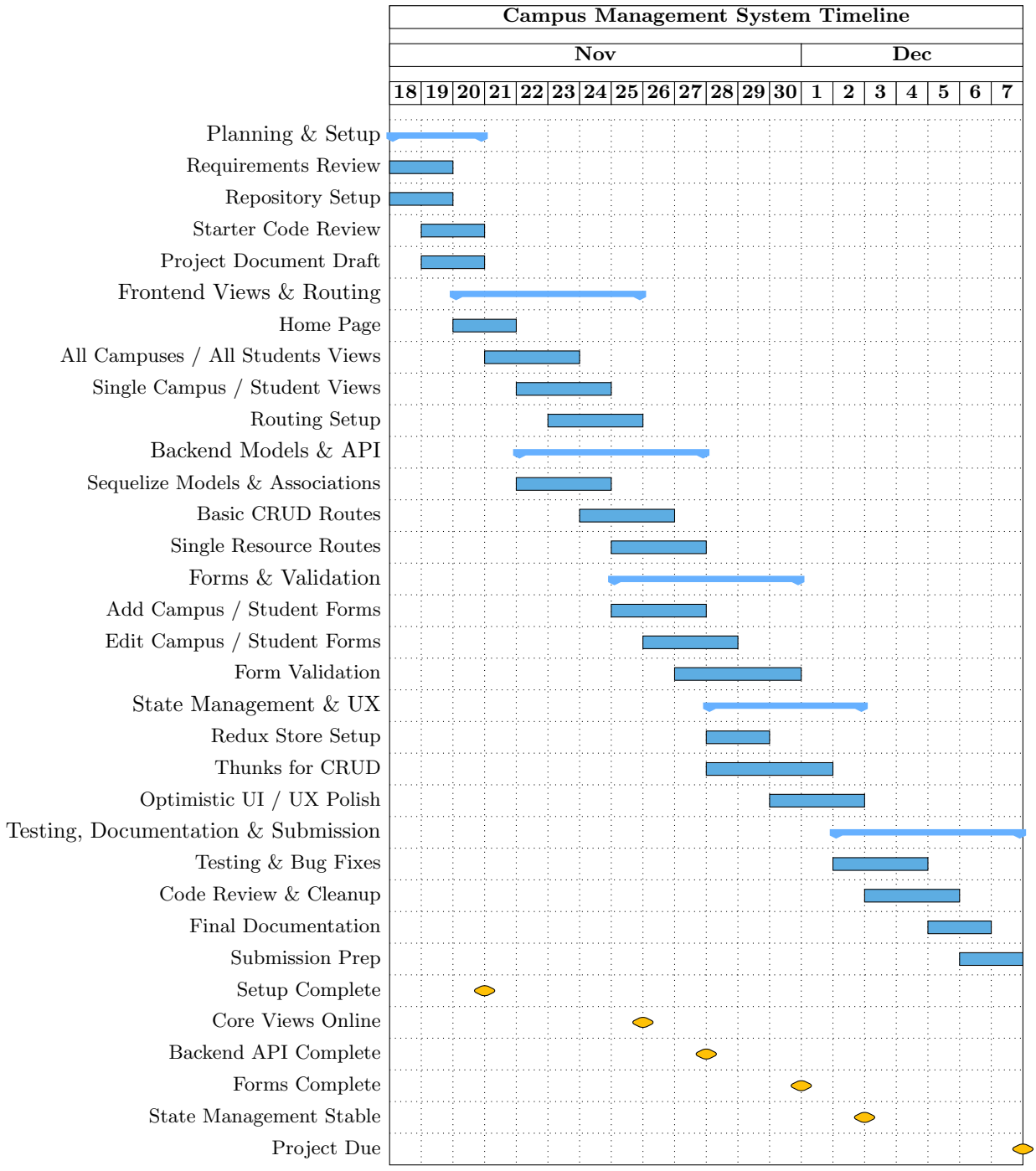

Figure 2: Project Timeline - Gantt Chart (Nov 18 - Dec 12, 2025)

4.3 Phase Breakdown

4.3.1 Phase 1: Planning & Setup (Days 1-3, Nov 18-20)

Tasks:

- Review assignment requirements in the PDF.
- Analyze provided starter code structure for backend and frontend.

- Set up GitHub repositories (client and server).
- Draft project document structure.

Deliverables:

- Project document outline.
- GitHub repositories configured.
- Starter code imported and reviewed.

4.3.2 Phase 2: Home Page & Routing (Days 3-6, Nov 20-23)

Tasks:

- Implement Home page component.
- Set up React Router in the frontend.
- Configure routes for core views (campuses, students, single campus, single student).
- Test navigation between pages.

Deliverables:

- Working home page.
- Router configuration.
- Basic navigation across core views.

4.3.3 Phase 3: Backend Models & API (Days 5-10, Nov 22-27)

Tasks:

- Define Sequelize models for Campus and Student.
- Implement associations between campuses and students.
- Build RESTful routes for all campuses and all students.
- Build routes for single campus and single student (with associations).
- Test CRUD endpoints using Postman or similar tools.

Deliverables:

- Synchronized database schema with seed data.
- Working API endpoints for campuses and students.
- Error handling and basic validation at API level.

4.3.4 Phase 4: Forms & Validation (Days 8-13, Nov 25-30)

Tasks:

- Implement Add Campus and Add Student forms.
- Implement Edit Campus and Edit Student forms.
- Add client-side validation and inline error messages.
- Connect forms to Redux thunks and backend API.

Deliverables:

- Fully functional add/edit flows for campuses and students.
- Consistent validation and UX patterns across all forms.

4.3.5 Phase 5: State Management & UX Polish (Days 11-15, Nov 28-Dec 2)

Tasks:

- Finalize Redux store slices for campuses and students.
- Optimize data fetching and minimize duplicate network calls.
- Improve loading states, error states, and empty states.
- Polish UI with responsive design and consistent styling.

Deliverables:

- Stable Redux-based state management.
- Smooth user experience across core flows.

4.3.6 Phase 6: Testing, Documentation & Submission (Days 15-20, Dec 2-7)

Tasks:

- Comprehensive testing of all features.
- Code cleanup and refactoring.
- Add code comments in complex areas.
- Complete project document and export as PDF.
- Update README files with team info and instructions to run Docker / scripts.

Deliverables:

- Fully tested full-stack Campus Management System.
- Clean, commented code in both repositories.
- Complete project document (PDF).
- Updated README files for frontend and backend with group info and run instructions.
- Submission ready for Brightspace (PDF + GitHub links).

4.4 Milestones

- **November 20**: Project setup and planning complete.
- **November 23**: Home page and routing complete.
- **November 27**: Backend models and core API complete.
- **November 30**: All forms and validation implemented.
- **December 2**: Redux state management and UX polish complete.
- **December 7**: Project tested, documented, and ready for submission.

4.5 Risk Management

4.5.1 Identified Risks

- **State Management Complexity**: Coordinating campuses and students with associations.
- **API Integration Issues**: Handling errors, loading states, and out-of-sync UI.
- **Validation Bugs**: Inconsistent client-side and server-side validation.
- **Merge Conflicts**: Multiple feature branches touching shared files.
- **Time Constraints**: Balancing full-stack requirements within the deadline.

4.5.2 Mitigation Strategies

- **Incremental Development**: Build and test one major feature at a time.
- **Shared Validation**: Reuse validation logic where possible.
- **Error Handling**: Implement consistent error patterns in thunks and components.
- **Small Commits**: Make frequent, small commits to minimize merge conflicts.
- **Code Reviews**: Review pull requests carefully before merging.

4.6 Resource Allocation

4.6.1 Team Structure (2-person team)

- **David**: Frontend views (Home, All/Single Campuses, All/Single Students), routing, and styling.
- **Kevin**: Backend models and routes, Redux state management, and thunks.
- **Shared**: Testing, documentation, Docker/scripts, and final polish.

4.6.2 Recommended Task Division

- **Developer 1**: Home page, routing setup, All/Single Campuses views, Add/Edit Campus forms.
- **Developer 2**: All/Single Students views, Add/Edit Student forms, association management.
- **Both**: API testing, error handling, documentation, and submission preparation.

4.6.3 Required Tools

- **Text Editor/IDE**: VS Code or similar.
- **Web Browser**: Chrome, Firefox, or Edge with DevTools.
- **Node.js & npm**: For React and Node development.
- **PostgreSQL**: For database.
- **Git**: For version control.
- **GitHub Account**: For repository hosting.
- **LaTeX Editor**: For project document.

5 Implementation Notes

5.1 React Router Implementation

5.1.1 Setting Up Router

- Import `BrowserRouter as Router` from `react-router-dom`.
- Wrap the React app in `<Router>` in `index.js`.
- Define routes for all required views using `<Route>` components.
- Use `exact` where appropriate for precise path matching.

5.1.2 Navigation

- Use `<Link to="/path">` for navigation between views.
- Ensure navigation bar is present on core pages for easy access.
- Support deep linking by handling route parameters for campus and student IDs.

5.2 Redux State Management

5.2.1 Store Structure

```
state = {
  campuses: {
    list: [],
    selected: null,
    status: 'idle',
    error: null
  },
  students: {
    list: [],
    selected: null,
    status: 'idle',
    error: null
  }
}
```

5.2.2 Thunks

- Fetch all campuses/students on initial load of list views.
- Fetch single campus/student when visiting detail views.
- Post, put, and delete requests for campuses and students.
- Update Redux state based on success or failure of each request.

5.3 Backend Implementation

5.3.1 Models and Associations

- Implement `Campus` and `Student` Sequelize models with validation.
- Define `Campus.hasMany(Student)` and `Student.belongsTo(Campus)` associations.
- Seed database with example campuses and students for development.

5.3.2 Routes

- Organize routes under `/api/campuses` and `/api/students`.
- Include associated models using Sequelize `include` options for single-resource routes.
- Handle errors with appropriate HTTP status codes and messages.

5.4 Best Practices

- **Component Structure**: Keep containers and views separated.
- **State Management**: Use Redux only for shared/global state.
- **Form Handling**: Use controlled components and reusable form inputs where possible.
- **Validation**: Validate inputs both client-side and server-side for robustness.
- **Error Handling**: Provide clear user feedback on errors.
- **Styling**: Maintain a consistent design system (colors, typography, spacing).
- **Git Workflow**: Use feature branches and code reviews to maintain quality.