**Student Course Enrollment System**

**Developers Guide v1.0**

***Prepared by Uday Shankar (2868122) & Yogeshwar Suresh (2862061)***

**Project Overview**

The **Student Course Enrollment System** is designed to streamline the process of course enrollment and administration in educational institutions. It includes user roles for students and admins, enabling:

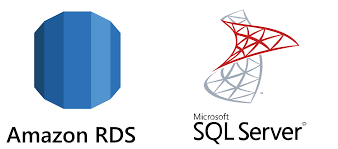
* **Students**: View available classes, enroll in courses, and track progress.
* **Admins**: Manage class details, monitor enrollments and monitor student details.

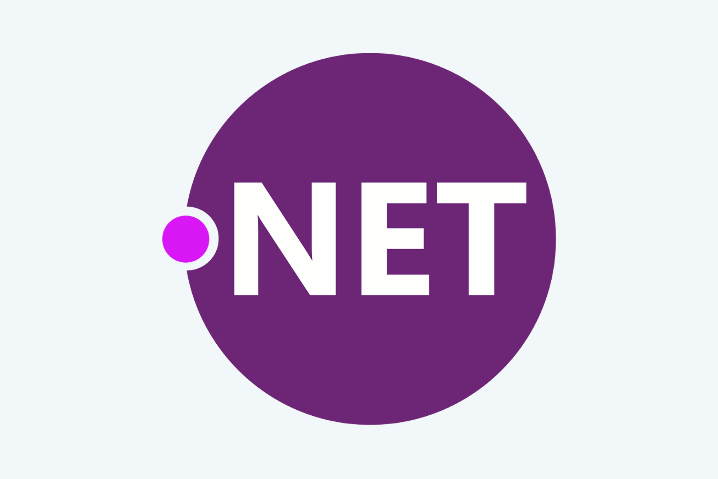
**Target Audience**

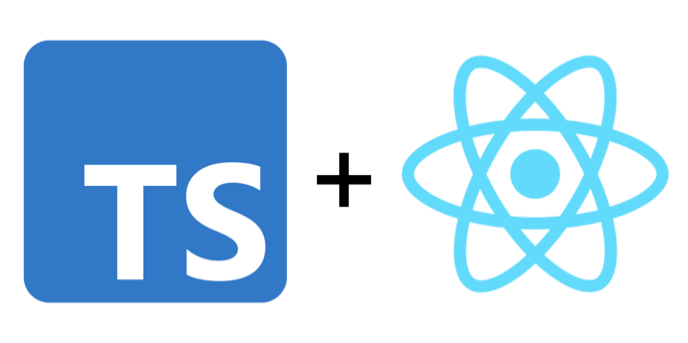
The guide is intended for developers tasked with maintaining or extending the application.

**Technologies Used**

* **Frontend**: React.Ts (Chakra UI for styling, Axios for API calls)
* **Backend**: .NET 8 Web API with Entity Framework Core
* **Database**: MS SQL Server hosted on Amazon Web Services as RDS DB.
* **Testing**: Manual Testing (outlined in test specification)
* **Deployment**: Local host Currently





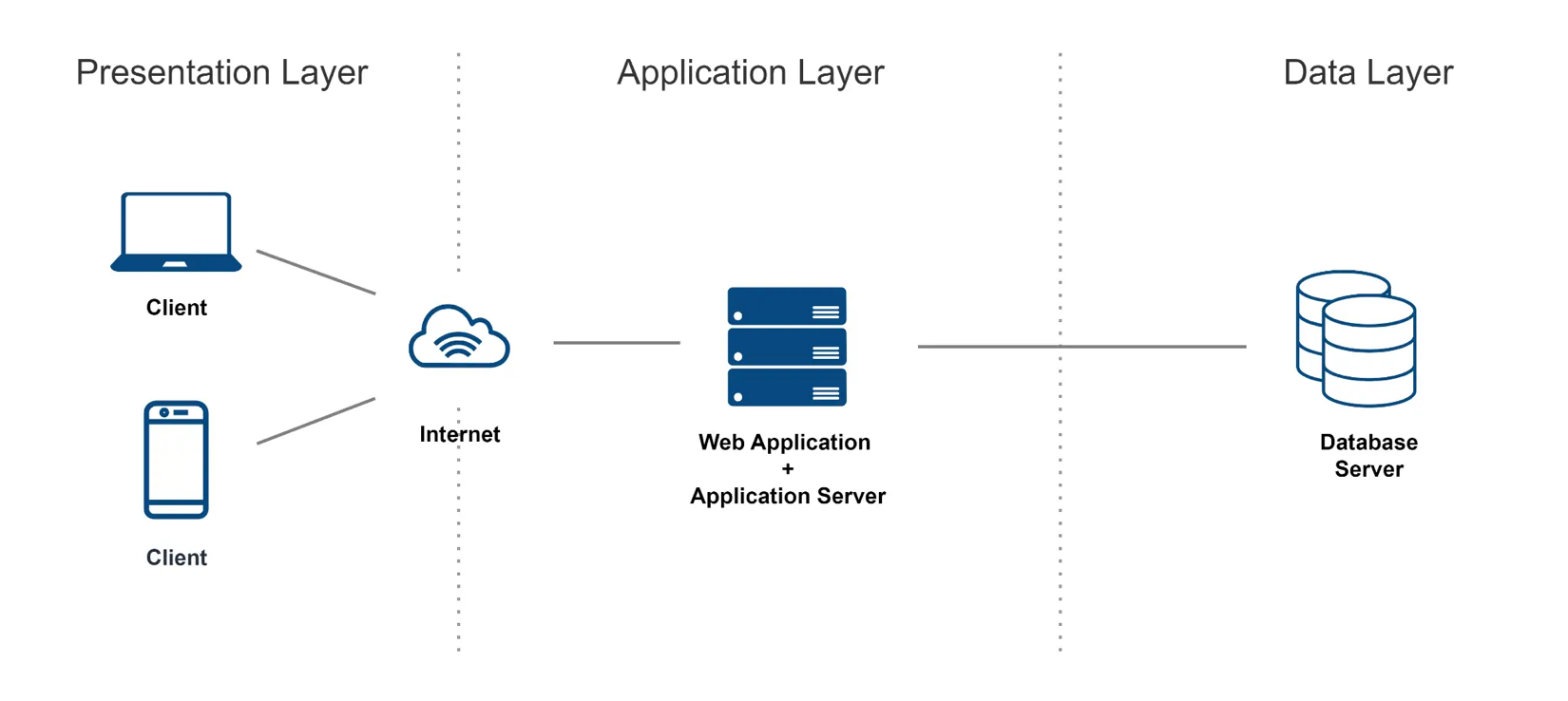




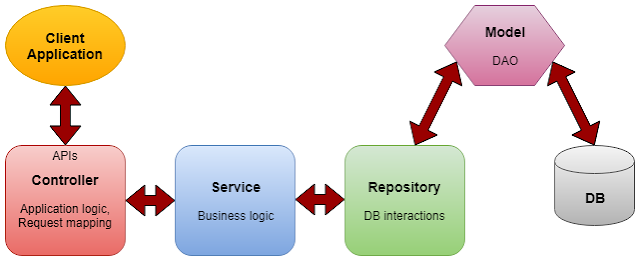
**Project Architecture**

The project employs a **Layered Architecture** with a separation of concerns between:

1. **Controllers**: Handle HTTP requests and responses.
2. **Services**: Business logic.
3. **Data Models**: Represent database entities.
4. **Data Access**: Entity Framework Core with a database-first approach.



**The Application layer Expanded (How the API works)**



**Prerequisites**

1. **Software Requirements**:
   * Visual Studio Code
   * SQL Server Management Studio to access the AWS DB on local
   * Postman (API testing)
   * Node.js (for running the frontend)
2. **Tools and Framework Versions**:
   * .NET 8
   * Entity Framework Core 7
   * React.js 18
   * Chakra UI (latest)
   * MS SQL Server 2022

**Repository Setup**

1. **Clone the repository:**

git clone <https://github.com/Uday77799/EEC521_Software_Project.git>

1. **Follow the branching strategy:**
   * **main**: Stable production code.
   * **Backend**: backend code, mainly the WEB API and DB codes.
   * **Feature** branches: frontend code, mainly for the REACT.JS codes.

**Environment Setup**

1. **Backend Configuration**:
   * Install the dot net dependencies

Dotnet restore

* + Update appsettings.json with your database connection string

"ConnectionStrings":

{

"DefaultConnection": "Server=localhost;Database=StudentEnrollment;Trusted\_Connection=True;"

}

1. **Frontend Configuration**:
   * Install dependencies:

npm install

* + Start the frontend:

npm start

**Project Structure**

**Backend**

1. **Controllers**:
   * StudentsController.cs
   * CourseController.cs
   * EnrollmentsController.cs
2. **Models**:
   * Student.cs.
   * Course.cs
   * Enrollment.cs
3. **Data**:
   * StudentManagementDbContext.cs
4. **DTOs**:
   * StudentDTO
   * CourseDTO
   * EnrollmentDTO

**Frontend**

1. **Pages**:
   * Login.tsx
   * Dashboard.tsx
2. **Services**:
   * Axios API calls for backend integration.

**Dependency Injection**

We are adding the SQL injection to add the DB with our application.

builder.Services.AddDbContext<StudentManagementDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

**Database**

**Schemas**

| **Table** | **Attributes** |
| --- | --- |
| Users | UserID, FirstName, Email, UserType |
| Courses | CourseID, CourseName, MaxSeats |
| Enrollments | EnrollmentID, StudentID, CompletionStatus |

For every table, to maintain consistently and to ensure that every day the data stays consistent throughout the entire development phase, there are proper triggers in place to ensure to record the whenever any update or delete is processed, there are checks and old data is stored in backup table.

**Triggers**

1. BeforeUpdate\_Users, BeforeUpdate\_Courses
2. BeforeDelete\_Users, BeforeDelete\_Courses
3. BeforeDelete\_Enrollments

**Scripts**

**Create Users Table (Admins and Students)**

CREATE TABLE Users (

UserID INT PRIMARY KEY IDENTITY(1,1),

FirstName VARCHAR(20) NOT NULL,

LastName VARCHAR(20) NOT NULL,

Email VARCHAR(30) NOT NULL,

Password VARCHAR(20) NOT NULL,

UserType INT NOT NULL, -- 1 for Admin, 2 for Student

EnrollmentDate DATE NULL -- Only applicable for students

);

**Create Courses Table**

CREATE TABLE Courses (

CourseID INT PRIMARY KEY IDENTITY(1,1),

CourseName VARCHAR(20) NOT NULL,

Description VARCHAR(500) NOT NULL,

MaxSeats INT NOT NULL,

CurrentSeats INT NOT NULL,

StartDate DATE NOT NULL,

EndDate DATE NOT NULL,

AdminID INT, -- Foreign Key from Users table where UserType

= 1 (Admin)

CONSTRAINT FK\_AdminID FOREIGN KEY (AdminID) REFERENCES

Users(UserID) );

**Create Enrollments Table**

*CREATE TABLE Enrollments (*

*EnrollmentID INT PRIMARY KEY IDENTITY(1,1),*

*StudentID INT, -- Foreign Key from Users table where*

*UserType = 2 (Student)*

*CourseID INT, -- Foreign Key from Courses table*

*EnrollmentDate DATE NOT NULL,*

*CompletionStatus VARCHAR(20) NOT NULL, — 1 is active, 2 is*

*complete*

*CONSTRAINT FK\_StudentID FOREIGN KEY (StudentID) REFERENCES*

*Users(UserID),*

*CONSTRAINT FK\_CourseID FOREIGN KEY (CourseID) REFERENCES*

*Courses(CourseID)*

*);*

**Trigger : BeforeUpdate\_Users**

CREATE TRIGGER BeforeUpdate\_Users

BEFORE UPDATE ON Users

FOR EACH ROW

BEGIN

-- Prevent updating UserType to an invalid value

IF NEW.UserType NOT IN (1, 2) THEN

SIGNAL SQLSTATE '45000'

SET MESSAGE\_TEXT = 'Invalid UserType. Only 1 (Admin) and 2 (Student) are allowed.';

END IF;

-- Prevent updating the UserID (Primary Key)

IF OLD.UserID != NEW.UserID THEN

SIGNAL SQLSTATE '45000'

SET MESSAGE\_TEXT = 'Cannot update UserID (Primary Key).';

END IF;

END;

**Trigger : BeforeDelete\_Users**

CREATE TRIGGER BeforeDelete\_Users

BEFORE DELETE ON Users

FOR EACH ROW

BEGIN

-- Check if the user is referenced in Courses

IF EXISTS (SELECT 1 FROM Courses WHERE AdminID = OLD.UserID) THEN

SIGNAL SQLSTATE '45000'

SET MESSAGE\_TEXT = 'Cannot delete User as they are referenced in Courses.';

END IF;

-- Check if the user is referenced in Enrollments

IF EXISTS (SELECT 1 FROM Enrollments WHERE StudentID = OLD.UserID) THEN

SIGNAL SQLSTATE '45000'

SET MESSAGE\_TEXT = 'Cannot delete User as they are referenced in Enrollments.';

END IF;

END;

**Trigger : BeforeUpdate\_Courses**

CREATE TRIGGER BeforeUpdate\_Courses

BEFORE UPDATE ON Courses

FOR EACH ROW

BEGIN

-- Ensure MaxSeats is not reduced below CurrentSeats

IF NEW.MaxSeats < OLD.CurrentSeats THEN

SIGNAL SQLSTATE '45000'

SET MESSAGE\_TEXT = 'MaxSeats cannot be less than CurrentSeats.';

END IF;

END;

**Trigger : BeforeUpdate\_Courses**

CREATE TRIGGER BeforeDelete\_Courses

BEFORE DELETE ON Courses

FOR EACH ROW

BEGIN

-- Check if the course is referenced in Enrollments

IF EXISTS (SELECT 1 FROM Enrollments WHERE CourseID = OLD.CourseID) THEN

SIGNAL SQLSTATE '45000'

SET MESSAGE\_TEXT = 'Cannot delete Course as it is referenced in Enrollments.';

END IF;

END;

**Trigger : BeforeDelete\_Enrollments**

CREATE TRIGGER BeforeDelete\_Enrollments

BEFORE DELETE ON Enrollments

FOR EACH ROW

BEGIN

-- Check if the Enrollment is marked as 'Completed'

IF OLD.CompletionStatus != 'Completed' THEN

SIGNAL SQLSTATE '45000'

SET MESSAGE\_TEXT = 'Cannot delete active or incomplete Enrollment records.';

END IF;

END;

**Rollback Scripts**

**Drop Enrollments Table**

DROP TABLE IF EXISTS Enrollments;

**Drop Courses Table**

DROP TABLE IF EXISTS Courses;

**Drop Users Table**

DROP TABLE IF EXISTS Users;

**Drop the trigger for the Users table**

DROP TRIGGER IF EXISTS BeforeUpdate\_Users;

DROP TRIGGER IF EXISTS BeforeDelete\_Users;

**Drop the trigger for the Courses table**

DROP TRIGGER IF EXISTS BeforeUpdate\_Courses;

DROP TRIGGER IF EXISTS BeforeDelete\_Courses;

**Drop the trigger for the Enrollments table**

DROP TRIGGER IF EXISTS BeforeDelete\_Enrollments;

**Entity Relationship Model**

**A diagram of a course

Description automatically generated**

**Brief Explanation of the API endpoints for Future Developers**

**Overview of API Endpoints**

The API provides functionality for managing courses, enrollments, and students in an educational management system. Below is a concise summary of the available endpoints:

**Course Management**

Endpoints to manage courses:

* GET /api/Course/ShowAllCourses: Fetch a list of all available courses.
* GET /api/Course/ShowCourseByID/{id}: Retrieve detailed information about a specific course by its ID.
* POST /api/Course/CreateCourse: Add a new course to the system.

**Enrollment Management**

Endpoints to handle student enrollments:

* GET /api/Enrollments/GetAllEnrollments: Retrieve all enrollments along with student and course details.
* POST /api/Enrollments/JoinCourse: Enroll a student in a specific course while validating seat availability and enrollment limits.

**Student Management**

Endpoints to manage student details and authentication:

* GET /api/Students/ShowAllStudents: Retrieve a list of all students in the system.
* GET /api/Students/ShowStudentByID/{id}: Fetch details of a specific student by ID.
* POST /api/Students/Login: Authenticate a student using their email and password.
* POST /api/Students/Logout: Log out the currently authenticated user.

Purpose of the Endpoints

1. Administrative Operations:
   * Manage courses and enrollment data efficiently.
2. Student Experience:
   * Enable students to browse courses, enroll, and access their details.
3. System Integrity:
   * Maintain limits on enrollments and validate data during operations.

**Brief Explanation of the Frontend for Future Developers**

The frontend for this project is developed using React and contains three primary pages that are essential for the system's functionality: Login, Dashboard, and the routing logic in App.tsx. Below is an explanation of each page and its purpose.

**Login Page (Login.tsx)**

* Purpose:
  + Allows users to authenticate by entering their email and password.
  + Verifies user credentials by calling the backend API (POST /api/Students/Login).
  + Directs the user to the dashboard upon successful login.
* Key Features:
  + Input validation to ensure all fields are correctly filled.
  + Displays error messages for invalid login attempts.
* Flow:
  + The user enters their credentials and submits the form.
  + A request is sent to the backend to validate the login.
  + On success, the user is redirected to the dashboard.

**Dashboard Page (Dashboard.tsx)**

* Purpose:
  + Acts as the central hub for users after login.
  + Displays personalized information depending on the user type (student or admin).
* Key Features:
  + Students:
    - View available courses.
    - Enroll in a course using a simple interface.
    - View current enrollments and course statuses.
  + Admins:
    - Manage courses and view enrollments.
    - Access detailed statistics and manage course limits.
* Flow:
  + The page fetches user-specific data on load (e.g., courses, enrollments).
  + Actions such as enrolling in a course or creating a course trigger API calls to update data.
  + UI updates dynamically based on API responses.

**Routing and Application Setup (App.tsx and index.tsx)**

* Purpose:
  + Manages routing and navigation between pages.
  + Ensures a smooth transition from login to dashboard.
* Key Features:
  + Sets up React Router for navigation.
  + Defines routes for the login page (/) and dashboard (/dashboard).
  + Handles unknown routes by redirecting users to a default page or error page.

**Coding practices**

* + Keep components small and focused on a single task (e.g., LoginForm, CourseList).
  + Use reusable components for common UI elements like buttons or forms.
  + Utilize useState or useReducer for local component states.
  + Consider integrating a global state management tool like Redux if the application scales.

**Development Practices**

**Coding Standards**

1. **Naming Conventions**: CamelCase for methods, PascalCase for classes.
2. **Folder Structure**: Keep controllers, models, and services separate.

**Testing**

1. Use Postman to validate API endpoints.
2. Test cases for login:
   1. Valid login
   2. Invalid credentials

**Version Control in Git**

**Repository Structure**

* **Repository Name**: EEC521\_Software\_Project
* **Branches**:
  + **backend**: Contains the up-to-date backend API code in the backend folder.
  + **frontend**: Contains the up-to-date frontend code in the frontend folder.

**Branching Strategy**

1. **Main Branches**:
   * **backend**: Used exclusively for backend development.
   * **frontend**: Used exclusively for frontend development.
2. **Create a feature branch:**

Example: git checkout -b feature/add-login-endpoint

1. **Merging Strategy:**

Once development or fixes are complete in a feature branch, create a pull request to merge the changes into the backend or frontend branch as appropriate.

**Deployment**

For this project, we will be going to deploy this project locally.

Now navigate to the API/API folder under the backend folder.

Then **BUILD** and **RUN** the backend.

1. dotnet build
2. dotnet run

Your backend is now live on localhost:5000

Now navigate to the frontend folder.

1. npm install
2. npm start

Your frontend is now live on localhost:3000

**Maintenance**

**Common Issues**

* **Database Connection Failure**:
  + Verify appsettings.json everytime if you move the database from local to cloud and vice versa
* **Missing Dependencies**:
  + Run npm install for frontend
  + dotnet restore for backend.