Course Registration System

System Architecture, Development & Testing Documentation

Project Manager: S. Nathiskar(2021/E/190)

Backend Developer: A. Jenarththan(2021/E/006)

Frontend Developer: K. Kanistan(2021/E/064)

Table of Contents

4. Testing

4.1 Functional Testing

4.1.1 Unit Testing (xUnit)

- ACC01 – SignInUser_ShouldCreateCorrectClaims
- AUTH01 – Login_WithValidStudentCredentials_ReturnsSuccessResponse
- AUTH02 – Login_WithInvalidCredentials_ReturnsUnauthorized
- DB01 – AdminDashboard_ShouldReturnViewResult_WhenUserIsAuthorized

Registration Management

- AC01 – ExportSession_ShouldReturnPDFFile
- AC02 – CreateRegistrationSession_ShouldInsertSessionAndOfferings
- AC03 – GetPendingRegistrations_ShouldReturnData
- ST01 – SubmitCourseRegistration_ShouldInsertPendingRegistration

Advisor and Coordinator Features

- ADV01 – SubmitApprovals_ShouldUpdatePendingRegistration
- ADV02 – GetStudentResults_ShouldReturnCorrectData
- COORD01 – GetStudentResults_ShouldReturnCorrectData

Supporting Services

- CS01 – CloseExpiredSessions_ShouldCloseAll
- CT01 – GetRegisteredCredits_ShouldReturnCorrectCredit
- SS01 – GetCurrentSemester_ShouldReturnCorrectSemester_WhenWithinRange

4.1.2 API Testing (Postman)

Authentication

- AUTH01 – Valid Login
- AUTH02 – Invalid Login

Registration

- AC03 – Fetch Pending Registrations

Advisor Access

- ADV02 – Get Student Results

4.1.3 UI Testing (Selenium)

Authentication Flow

- AUTH00 – Login Page Loads Properly
- AUTH01 – Valid Student Login
- AUTH02 – Invalid Login Attempt
- AUTH03 – Logout Redirection

Navigation Flow

- NAV01 – Student Dashboard Navigation
- NAV02 – Admin Sidebar Links Navigation

Registration Form Validation

- REG01 – Empty Field Submission
- REG02 – Valid Course Selection and Submit

Dashboard Data Verification

- DASH01 – Student Information Displayed
- DASH02 – Administrator Statistics Displayed

4.2 Non-Functional Testing

4.2.1 Performance Testing

- Load Testing
  - Objective: Simulate 50 concurrent users on the login page
  - Tool Used: Apache JMeter 5.6.3
  - Status: Passed

4.2.2 Security Testing

- Authentication and Authorization Validation
  - Result: Passed
- Role Escalation Attempts
  - Result: Passed
- SQL Injection
  - Result: Passed
- Password Encryption and Storage
  - Observation: Plain text storage (demo purpose); to be improved
  - Status: Known Limitation

4.2.3 Compatibility Testing

- Browser Compatibility
  - Browsers Tested: Chrome, Firefox, Edge
  - Status: Passed
- Device Compatibility
  - Tool: Chrome DevTools
  - Result:
    - Desktop – Fully responsive
    - Mobile/Tablet – Partial responsiveness
  - Status: Partially Passed

5. Installation Details

5.1 System Requirements
5.2 Prerequisites Setup
5.2.1 Database Server

6. Reference Details

Architecture

1.1 Purpose and Scope

This section describes the software architecture of the Faculty Course Registration System. It provides an **overview of the system's design and structure**, addressing different architectural *views* (per IEEE 1016 and the "Views & Beyond" approach) to communicate how the system is organized (Views and Beyond Collection). The architecture documentation is intended for developers, software architects, and technical stakeholders who need to understand or evaluate the system's design. It covers all major components and interactions of the Course Registration System backend (ASP.NET Core MVC application and SQL Server database), and how various user roles (Student, Advisor, Course Coordinator, AR Officer) interact with the system. The scope of this section is limited to the internal architecture of the Course Registration System – external interfaces or unrelated systems are out of scope.

1.2 Architectural Goals & Constraints

Architectural Goals: The system was designed with several key goals in mind:

- **Digital Automation & Usability:** Provide a user-friendly web interface for students and staff to replace manual course registration processes. Multiple user roles must be supported with role-specific functionalities.
- **Separation of Concerns & Modularity:** Adopt a multi-tier architecture and MVC pattern to clearly separate the presentation, business logic, and data access layers (Understanding the architecture of a 3-tier application). This improves maintainability and scalability (e.g., easier updates or future extension to a mobile app) .
- **Data Integrity & Consistency:** Ensure that course registrations, approvals, and prerequisites are consistently enforced through a well-designed relational data model. The system must handle complex rules (e.g., multiple registration attempts, prerequisite checking, advisor approvals) reliably.
- **Security:** Enforce role-based access control so that each user role can only perform authorized actions. Protect sensitive data (like passwords) and use secure communication (HTTPS) for client-server interactions.
- **Reliability & Auditability:** Provide a robust workflow for course registrations with proper status tracking (pending, approved, rejected) and notifications, so that no requests are lost and each action (advisor/AR approvals) is recorded.

Constraints: Several constraints influenced the design:

- **Technology Stack Constraint:** The system is built with ASP.NET Core MVC 8.0 (C#) and SQL Server as required by project guidelines (e.g. institutional preference for Microsoft technologies). No third-party integrations or external APIs are used, and the solution needed to be implemented with available frameworks (ASP.NET Core, Entity Framework Core) within a limited time.
- **Deployment Environment:** Initially, the system is hosted locally (development machine). It runs on a single server (or localhost) with the database on a local SQL Server instance. This

implies a *monolithic deployment*. Future deployment may involve an IIS or cloud VM for the web app and a separate SQL Server instance. The design assumes a controlled network environment (e.g., campus intranet) with no internet-facing services at present.

- **Authorization Approach:** The project did not use the full ASP.NET Core Identity Membership system. Instead, a simplified **cookie-based authentication** with custom role handling was used (storing users in *Student* and *Staff* tables). This imposed a constraint that role management and password security had to be handled manually in code (e.g., hashing passwords, using the [Authorize] attribute with role names).
- **Project Scope & Time:** Given this is a student project, certain "enterprise" features (e.g., comprehensive logging, automated testing beyond basic cases, advanced optimization) were limited. The architecture favors clarity and separation of duties over highly optimized performance, which is acceptable for the anticipated moderate usage in a faculty setting.

1.3 System Context – Overview and External Interfaces

At the highest level, the Course Registration System is a **web-based information system** that serves four types of users within the faculty. **Figure 1** below shows the system in its context, with the primary user roles as external actors:

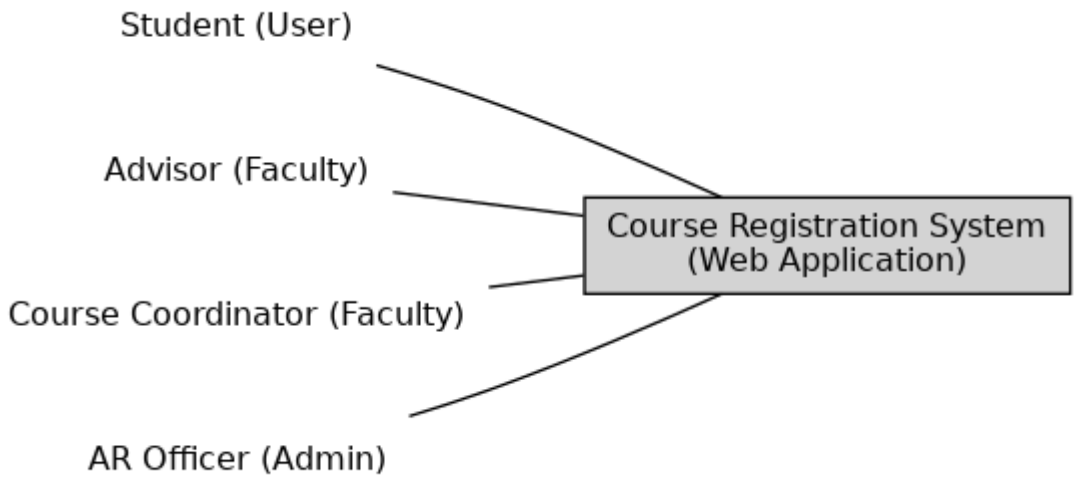


Figure 1: System Context Diagram – The Course Registration System and its user actors.

Each actor interacts with the *Course Registration System (Web Application)* to perform tasks:

- **Student:** Uses the system to view available courses for the semester, register for courses, view their current registrations and past course history, and check grades/GPA. Students are the initiators of the course registration requests.
- **Student Advisor:** Faculty members who serve as advisors log in to review their advisees' pending course registration requests. An Advisor can approve or reject a student's course selections. Advisors can also view student info (profile, academic results, GPA) to inform their decisions.

- **Course Coordinator:** In this context, a Course Coordinator (usually the instructor or department staff responsible for a course) uses the system to view the list of students enrolled in courses they coordinate and to access in-course assessment results. They can monitor how many students have registered for their course and view performance statistics (e.g., GPA of enrolled students).
- **AR Officer (Admin):** The Academic Registry Officer is the administrator of the system. The AR Officer manages the overall registration process – e.g., setting up which courses are available for registration each semester, monitoring all pending requests, and giving final approval or rejection for course registrations (typically after advisors have approved). The AR Officer can also manage master data (like adding courses to the system for a new semester) and oversee whether Advisors have completed their approvals.

In the system context, these human actors are the only external entities. **No external IT systems** or third-party services interact with the Course Registration System (for example, there is no external payment gateway or identity provider in scope). The system's environment is an intranet web application accessible via a web browser. The *Course Registration System* itself encompasses both the application server and the database (see Deployment View). The context diagram illustrates that all user types access the system through the web application's interface. Interactions between users and the system occur through standard web requests over a network (HTTP/HTTPS), but for context we treat the entire web app as a single system box. All persistent data (student records, course info, registrations) is stored internally in the system's database; there are no external data feeds.

1.4 High-Level Architecture Overview

At a high level, the Course Registration System follows a **standard 3-tier architecture** separating the application into Presentation, Application Logic, and Data layers. This design was chosen to enforce *separation of concerns*, improving modularity and maintainability (Understanding the architecture of a 3-tier application). **Figure 2** shows the tiered architecture:

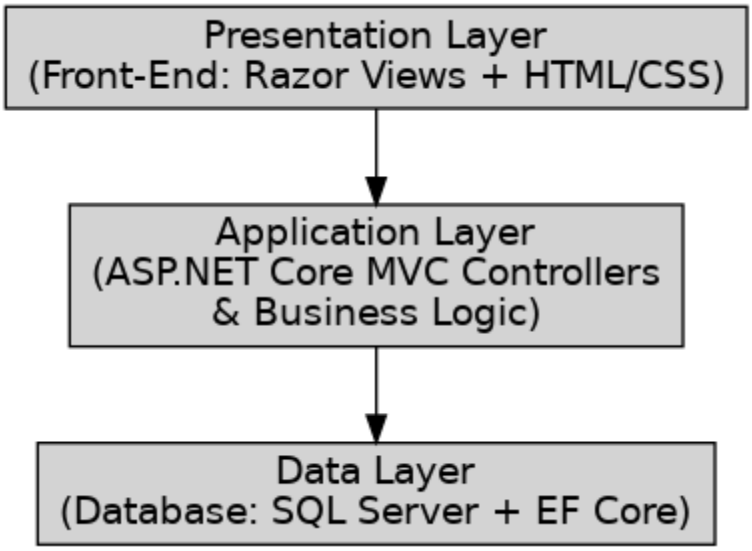


Figure 2: Three-Tier Architecture – Presentation Layer (UI), Application Layer (MVC/Web API logic), and Data Layer (Database).

- **Presentation Layer (Frontend):** This consists of the **web-based UI** built with ASP.NET Core Razor Views, HTML, CSS, and Bootstrap. The UI layer is responsible for presenting information to the users and capturing user input. It includes Razor view pages for login, course lists, forms for course selection, approval dashboards, etc. The views are rendered on the server and delivered to the user's browser. This layer also includes client-side scripts or validations if any (though primarily it's server-rendered HTML forms). The Presentation layer communicates with the application layer by issuing HTTP requests (when a user submits a form or clicks a link, it triggers a controller action).
- **Application Layer (Backend Logic):** This is the core of the ASP.NET Core MVC application – it contains the **Controllers, business logic, and domain services**. Controllers are C# classes that handle incoming HTTP requests, process input (often interacting with the data layer or invoking domain logic), and produce HTTP responses (typically rendering a Razor view or returning data). Each major role or feature has its own controller(s) (e.g., StudentController, AdvisorController, CoordinatorController, AdminController for AR Officer). These controllers enforce role-based rules (using [Authorize] attributes or internal checks) so that, for example, only an *Advisor* can access advisor actions. Within the application layer, there are also **service classes** that encapsulate certain business rules or periodic tasks – for instance, a CreditTrackingService to compute student credit totals/GPA, or a CloseExpiredSessionsService to automatically close registration sessions after their end date. By isolating complex logic in services, the design improves reusability (the logic could be reused in future mobile apps or APIs) and keeps controllers simpler. The application layer is the **mediator** between UI and Data layers: controllers retrieve or update data via the data layer and then decide what UI to show.
- **Data Layer:** This layer consists of the **database and data access logic**. The system uses Microsoft SQL Server for the relational database, and Entity Framework Core (EF Core) as an Object-Relational Mapper. The data layer includes the EF Core *DbContext* (called ApplicationDbContext) and the entity models that map to database tables (such as Student, Course, Registration, etc.). Controllers (or services) use the DbContext to query or persist entities. All actual data storage happens here – the database enforces constraints like primary keys, foreign keys, etc. The Data layer is abstracted behind the DbContext and repository-like patterns; therefore, higher layers do not directly execute SQL but use C# objects. This separation ensures that business logic is database-agnostic to some extent and that data integrity is centrally managed. The data layer is also responsible for **authorization data** – e.g., the Staff and Student tables contain user credentials and roles, which the application layer uses for authentication.

The **three-tier architecture** ensures that each layer can evolve or be scaled independently (Understanding the architecture of a 3-tier application). For example, one could replace or distribute the database (Data layer) without affecting UI or business logic, or update the UI technology without changing how data is stored. In our case, the Presentation and Application layers are both part of the ASP.NET Core web application (running on a single server), but logically they are separate concerns. This approach also facilitates **security** – the database can be isolated, and only the application layer interacts with it, so we can centralize validation and checks there.

Architectural Style: The system architecture can also be described as a **Layered MVC architecture** – within the application layer we follow MVC (Model-View-Controller) pattern, and across the whole system we have a layered structure (UI -> MVC Controllers -> Data). This combination of MVC (for internal structure) and 3-tier (for deployment separation) is a common pattern for web applications, providing clear boundaries between user interface, domain logic, and data persistence.

1.5 Logical View (Module Decomposition)

In this section, we decompose the system into its *logical modules and components* and describe their responsibilities and relationships. The logical view corresponds to the **structural organization of the software** – essentially, how the codebase is modularized into components such as controllers, models, and services ().

Key Components and Modules:

- **ASP.NET Core MVC Framework:** At the base, the application uses the ASP.NET Core MVC framework. This provides the infrastructure for routing (mapping URLs to controller actions), model binding, view rendering, etc. The framework itself can be seen as part of the platform rather than a module we wrote, but it dictates the structure of our modules (e.g., we have controllers and views).
- **Controllers:** Each user role or major functionality is implemented by a dedicated *controller* (or set of controllers). For example:
    - AccountController – handles authentication (login/logout) for both Students and Staff. It validates credentials against the database and issues authentication cookies with the user's role claims.
    - StudentController – handles student-specific actions (view available courses, submit course registration, view registration history, etc.).
    - AdvisorController – handles advisor-specific actions (view pending registrations of advisees, approve/reject requests, view advisee details).
    - CoordinatorController – handles course coordinator actions (view enrollment lists, view student grades in their course).
    - AdminController – handles AR Officer (administrative staff) actions (set up registration sessions, manage course offerings, approve registrations after advisor approval, monitor overall status).
    - DashboardController exist to redirect users to the appropriate dashboard based on role after login.

Each controller is essentially a **module** corresponding to a subset of use cases. Controllers make use of models and services to fulfill requests. They also enforce **access control** by being decorated with [Authorize(Roles="…")] attributes (e.g., the AdvisorController class might have [Authorize(Roles = "Advisor")] to ensure only Advisor accounts invoke its actions). In the logical structure, controllers depend on the underlying data/model layer but are independent of each other (except for shared concerns like the auth framework).

- **Models (Data Entities):** The system's core business entities are represented as C# classes in the *Models* folder. These correspond one-to-one with database tables (since we used Entity Framework code-first). Key model classes include Student, Staff, Course, CourseOffering, RegistrationSession, RegistrationSessionCourse, PendingRegistration, Registration, Result, Department, HasPrerequisite, etc. These classes primarily hold data fields and relationships (with DataAnnotations attributes to define keys and foreign keys). They serve as **Data Transfer Objects** within the app – e.g., a controller querying pending registrations of a student will retrieve a list of PendingRegistration objects. In the logical architecture, these model classes are used by both the data access layer (EF) and the business logic.
- **Domain Services & Utilities:** To keep controllers lightweight, some complex logic is implemented in service classes:
    - SemesterService – might encapsulate logic to determine current academic semester or to manage academic schedules.
    - CreditTrackingService – contains methods to calculate a student's earned credits and GPA by aggregating results. A student's "View GPA" feature likely calls this service.
    - CloseExpiredSessionsService – a background service (perhaps triggered on a schedule or at login) that scans RegistrationSessions and closes any that have passed their end date (ensuring students can no longer register after deadlines).

These services are **helper components** that the controllers or other parts of the application layer can call. They often interact with the DbContext as well, but encapsulate specific **business rules or maintenance tasks**. For instance, when a student attempts to register for a course, the PendingRegistration creation might involve checking prerequisites; the logic to check if a student has already passed the prerequisite course could be in a service method (or done via a DB query).

- **Data Access Layer (EF Core DbContext):** The ApplicationDbContext class (in the *Data* folder) is the central gateway to the database. It is configured with DbSet<TEntity> properties for each entity (e.g., DbSet<Student> Students, DbSet<Course> Courses, etc.). This context is registered with the ASP.NET dependency injection and made available to controllers and services. Any database interaction (queries or saves) goes through this context. In some architectures, one might introduce repository classes, but here the DbContext itself acts as the repository for entities. The logical dependency is such that **controllers and services depend on the DbContext** (or on repository interfaces, if they were abstracted) for persistence.
- **Views and ViewModels:** While much of the logic is in controllers, the *Views* (Razor .cshtml files) constitute the presentation of data. In some cases, there are also *ViewModel* classes (in **Models/ViewModels** folder) used to shape data specifically for views (for example, a combined model for a student's dashboard that includes counts of pending approvals, etc.). These view-specific models help pass only the necessary data to the UI. The logical relationship is: controllers populate ViewModels (or use entity models directly) and pass them to the Views for rendering. View files themselves contain minimal logic (mainly presentation), pulling data from the model passed in.

Interactions: The following diagram illustrates the major logical components and their interactions/dependencies:

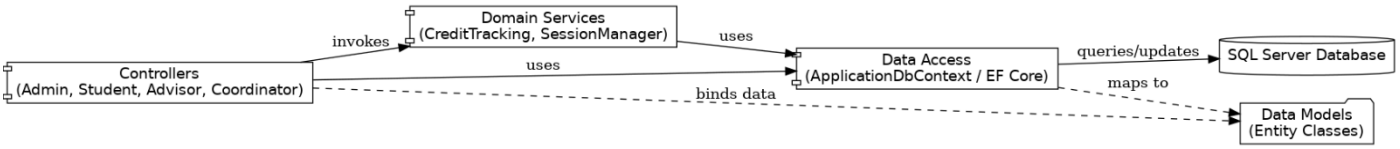


Figure 3: Logical Component Diagram – illustrating controllers (by role), domain services, data context, data models, and their relationships.

In **Figure 3**, the controllers (grouped on the left) invoke domain services or use the data access layer as needed. For example, when a Student submits a course registration request, the StudentController calls the ApplicationDbContext to add a new PendingRegistration record. When an Advisor approves a request, the AdvisorController may call a service (to perform any complex validation or notifications) and then update the PendingRegistration via the DbContext. The data models (on the right, shown as entity classes) are used across these calls – e.g., the DbContext maps models to database tables (), and controllers bind HTTP form inputs to model instances (model binding).

Summary of Module Dependencies:

- The **Presentation layer** (Views) depends on Controllers (for data context in which they render) and on ViewModel/Data models (for the data to display).
- **Controllers** depend on: (a) the *Models* (entities or viewmodels) for data structures, (b) the *DbContext* for persistence, and (c) optionally on *Services* for additional business logic. Controllers also rely on the ASP.NET Core framework (routing, model binding) which is not shown in the diagram but is an underlying dependency.
- **Services** depend on *Models* and *DbContext* (and possibly on other services) to perform their tasks. For instance, CreditTrackingService uses the DbContext to load a student's grades (Results) and Student model to compute GPA.
- **DbContext** depends on *Models* (it needs the model classes to know the schema). The DbContext is used to mediate between the object-oriented model and the relational database.
- The **SQL Database** is accessed only via the DbContext; no other component talks to the database directly. This one-point data access ensures that we can enforce constraints and catch errors at a single layer.
- The **authorization subsystem** (part of ASP.NET Core) ties into this logical view as well: upon login, the AccountController establishes the user's identity and roles (creating a ClaimsPrincipal). Subsequent controller actions have [Authorize] attributes that the framework checks against this principal (Role-based authorization in ASP.NET Core | Microsoft Learn). In our project, roles like "Student", "Advisor", "Coordinator", "AR" correspond to the Role field in the Staff or Student record, and those roles are included in the

user's claims on sign-in. This mechanism ensures each controller only runs if the user has the correct role.

In summary, the logical view shows a **clean separation of responsibilities**: UI rendering in views, input handling and coordination in controllers, core logic in services, and persistence in the data layer. This modular structure aligns with standard MVC architecture (each controller roughly corresponds to a module serving a particular actor or functionality).

1.6 Process View (Runtime Behavior)

The process view describes **dynamic behavior –** how the system's components interact at runtime to fulfill key scenarios (). We detail the major runtime workflows (use cases), focusing on how control flows between the user interface, controllers, services, and database.

Major Process: Course Registration and Approval Workflow

One of the most important runtime scenarios is a student registering for a course and the request flowing through advisor and AR officer approvals. Below we break this process into steps and illustrate it with a sequence diagram.

- **Scenario:** *Student submits a course registration, which must be approved by their Advisor and then finalized by the AR Officer.*

1. **Student Submission:** A Student logs in and navigates to the list of available courses for the active semester. This page (rendered by StudentController) shows which courses are open for registration (pulled from the RegistrationSessionCourses and CourseOfferings in the database). The student selects one or more courses and clicks "Register". This triggers an HTTP POST to the StudentController (e.g., RegisterCourses action).

2. **StudentController – Create Pending Record:** The StudentController action receives the student's course selection. It first checks business rules (e.g., has the student exceeded the credit limit? Is the registration period still open? If a prerequisite is required, has the student passed it?). These checks might be done directly in the controller or via a call to a SemesterService or similar. Assuming validation passes, the controller creates a new **PendingRegistration** entry via the Data layer. This involves calling ApplicationDbContext.PendingRegistrations.Add(...) with a new PendingRegistration model (which includes StudentID, CourseCode, etc.) and SaveChanges(). The PendingRegistration is saved with status "Pending" and a flag IsApprovedByAdvisor=false.

3. **Confirmation to Student:** After saving, the system returns a response to the student – typically redirecting them to a confirmation page or their current registrations list. The UI will show the course as "Pending Advisor Approval". (In terms of sequence, the Student's browser receives a success page – this completes the initial request/response cycle for the student submission.)

4. **Notify Advisor:** In the background, the system will ensure the Advisor is made aware of the new request. There are a couple of ways this happens: the *Advisor's dashboard* will query all pending registrations for their advisees from the database (so when the Advisor next loads their page, they'll see it), and/or the system creates a **Notification** record. In the design, there is a Notification table which can store messages. The StudentController could create a

Notification entry like "Student X has submitted a registration for Course Y" addressed to that student's Advisor (with fields ReceiverID = Advisor's StaffID, ReceiverType = Staff, etc.). This depends on implementation – but logically, the **Advisor needs to find out about the pending request**. (The sequence diagram indicates a "Notify Advisor" step after student submission.)

5. **Advisor Approval:** When the Advisor logs in and checks their pending requests (handled by AdvisorController, e.g., an action ViewAdviseeRequests), the system retrieves all PendingRegistrations for students under that advisor (likely a query on PendingRegistrations joined with Student where Student.StaffID (advisor) matches the logged-in advisor's StaffID). The advisor reviews the request and clicks "Approve" (or "Reject"). This triggers the AdvisorController.ApprovePending(int pendingId) action (for example). That controller action will first ensure the logged-in user is indeed the advisor for that request (security check), then update the PendingRegistration record: set IsApprovedByAdvisor=true, and perhaps set a Status field to "AdvisorApproved" and record the date/time.
6. **AdvisorController – Update Pending:** The controller saves these changes via the DbContext. Now the PendingRegistration reflects that the advisor has approved. Typically, the student might be notified at this point that their advisor approved (again via a Notification record or simply by seeing the status change in their view).
7. **Notify AR Officer:** Once advisor approval is done, the AR Officer (Admin) needs to finalize. The system could either immediately notify the AR or simply the AR's dashboard will show all pending registrations that are advisor-approved. In practice, the PendingRegistration might remain in the table until AR acts on it. The AR Officer's view (handled by AdminController) can filter pending requests by status. The system *could* insert a Notification for the AR as well (ReceiverType=Staff (AR), RelatedPendingRegistrationID, etc.) indicating a request is ready for final approval. In our sequence we show a "Notify AR" step.
8. **AR Final Approval:** The AR Officer logs in and opens the administration interface for course registrations (via AdminController, e.g., ViewPendingRequests). The system queries all PendingRegistrations (or specifically those advisor-approved). The AR selects the particular student's request and clicks "Approve" (or "Reject"). This triggers AdminController.FinalizeRegistration(pendingId, approve=true/false). The controller will then carry out finalization:
    - If approved, it will **insert a new Registration record** into the Registrations table (this table represents officially registered courses for a student). It takes data from the PendingRegistration (Course, Student, etc.) and creates a Registration entry. It may also mark the OfferingID (linking to the specific course offering section) and copy relevant info (attempt number, etc.). The PendingRegistration entry can then be either deleted or marked as completed (to keep history).
    - If rejected, it might just update the PendingRegistration status to "Rejected" (and potentially not create any Registration entry).
    - In either case, the AR action is saved (and an ApprovalDate for AR might be set).

9. **Post-Finalization Actions:** On final approval, the system could update related data: e.g., increment the enrollment count for that CourseOffering, generate a Notification to the student ("Your registration for Course Y has been approved by AR"), and remove the pending entry from outstanding lists.
10. **Student Notification:** Finally, the student will find out the outcome. Typically, when the student next checks their registration status page, the course will either appear in "Registered Courses" (if approved) or in a "Rejected requests" list with reason. Additionally, using the Notification mechanism, the system can send the student a message for record. In our sequence, after AR approves, we show "Notify Student (approved)" as the last step.

The following sequence diagram summarizes this flow with the main interactions between the *Student, Advisor, AR Officer,* the *Web Application* (controllers), and the *Database*:

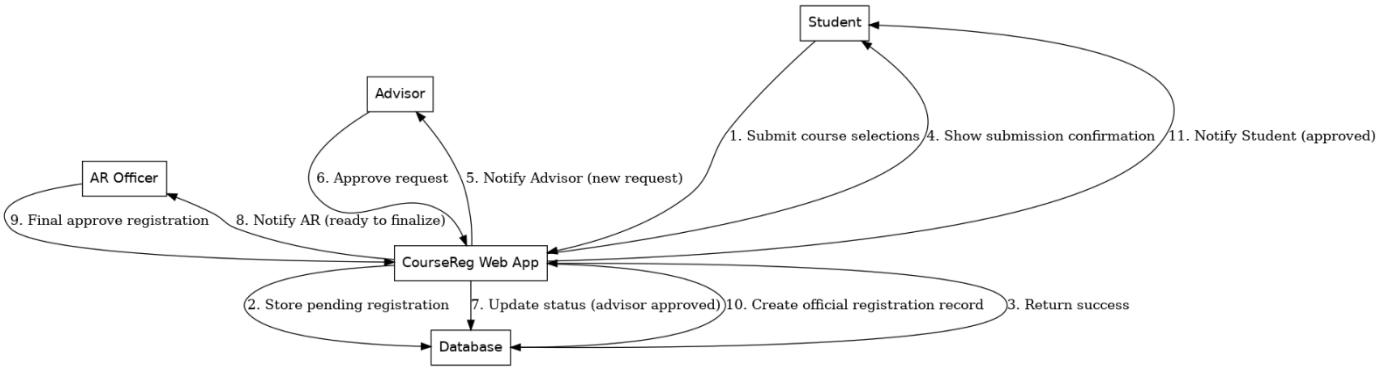


Figure 4: Sequence Diagram – Course Registration end-to-end process (Student request, Advisor approval, AR final approval).

In **Figure 4**, time progresses from top to bottom. The *CourseReg Web App* lifeline represents the server-side application (controllers/services) and the *Database* lifeline represents data storage (via EF Core). We see the student's initial request (1) going to the web app, which in step (2) writes a pending record to the database. The confirmation flows back to the student (steps 3-4). Then the app notifies the advisor (5) – this might be an actual push notification or simply data ready for when the advisor next polls. The advisor's action (6) comes in later, updating the database (7). The AR Officer is then notified (8), performs final approval (9), which causes the app to store the final registration (10). Finally, the student is notified of the approval (11).

This scenario demonstrates the *asynchronous* nature of the multi-actor workflow: the Student's request and the approvals happen in different HTTP requests at different times, but the system maintains state (via the PendingRegistrations in the DB) between these steps. The use of a central database and status fields allows the process to be **resilient** – e.g., if the student logs out after step 4, the request still resides in the system for the advisor to act on later. The Notifications subsystem (if used) further ensures no manual step is missed by alerting the relevant parties.

Other Process Scenarios:

- **Course Offering Setup (AR Officer):** Before students can register, the AR Officer must set up the semester's offerings. This involves creating a new RegistrationSession (for the

semester and program) and associating courses. In practice, the AR Officer (via AdminController) might create a session (specifying AcademicYear, Semester, start/end dates). Then they either select courses from the master list to include, or if courses are fixed, the system automatically includes all active courses for that semester. For each course included, a CourseOffering record is created, possibly assigning a staff member as the coordinator/instructor and linking to the session. The AR Officer also toggles the session "IsOpen" flag when registration should be open to students. The sequence for this is straightforward: AR fills a form -> AdminController -> DbContext creates records (RegistrationSession, CourseOffering etc.). After this, students logging in will see those courses as available.

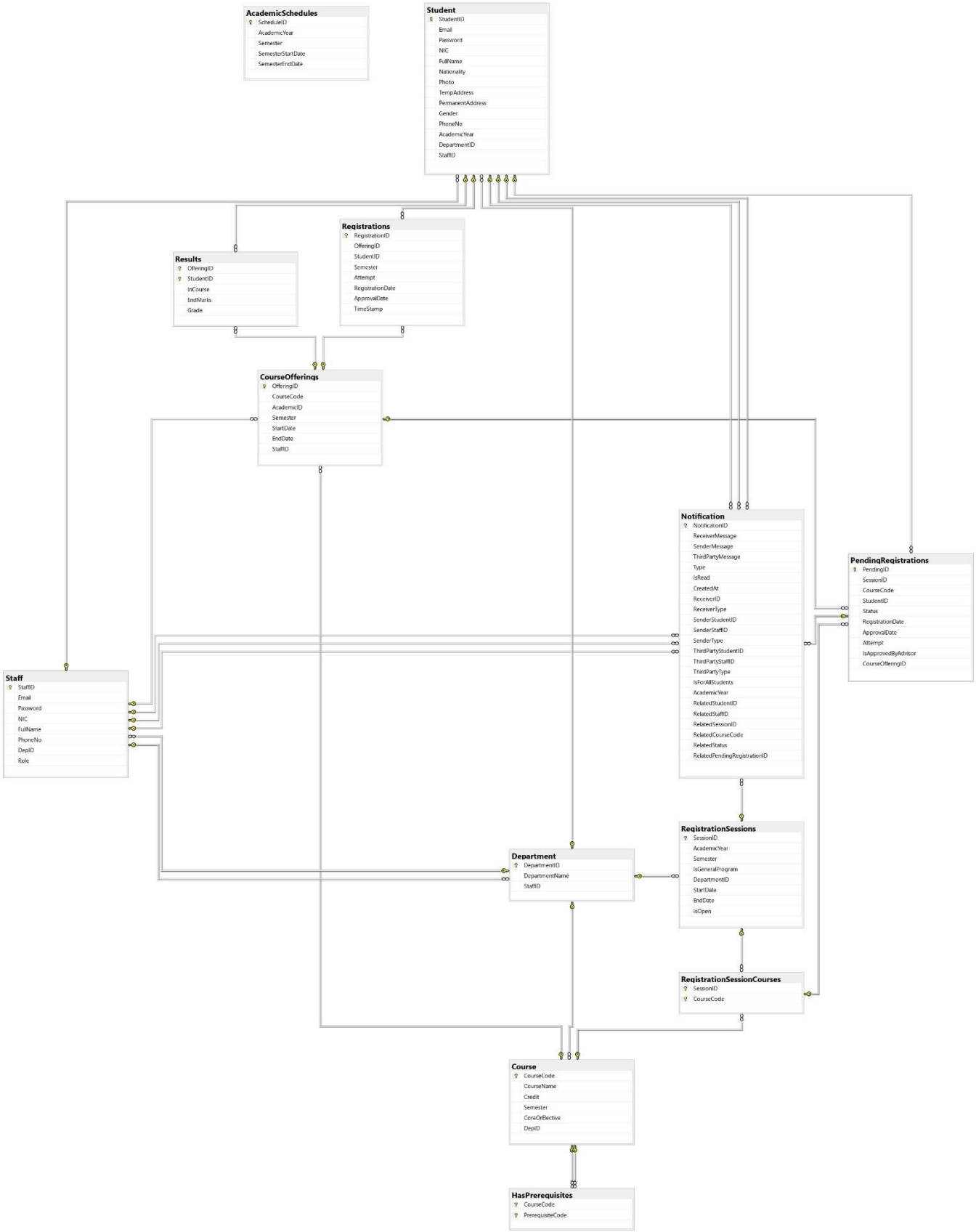
- **Advisor Viewing Student Details:** When an advisor views an advisee's profile or transcript, the AdvisorController will query the Student table (for personal info) and perhaps the Results table for that student's grades, and the Registration table for completed courses. It then displays this in a view. This is a simple request/response with some joins or multiple queries. (Any role-specific data access like this is protected so an advisor can only retrieve their own advisees – likely ensured by checking Student.StaffID of those records.)
- **Coordinator Viewing Enrollments:** A Course Coordinator will select one of their courses (where CourseOffering.StaffID is the coordinator's StaffID). The CoordinatorController queries the Registration table for all students with that OfferingID and displays the list (and perhaps uses the Result table to show in-course marks or final grades for those students). This again is a straightforward flow: Coordinator request -> Controller -> DB -> View.
- **Periodic Session Closure:** If implemented, the CloseExpiredSessionsService might run periodically (maybe triggered by a scheduled background task or invoked when admins log in). It checks RegistrationSessions where EndDate < Now && IsOpen=true, and for each, sets IsOpen=false (closing further registration). This ensures time-bound enforcement. This service operates without user intervention (possibly as a hosted service in ASP.NET or a manual trigger by AR Officer).

Overall, the **Process View** confirms that the system's layered architecture is working as designed at runtime: *Controllers* receive inputs and coordinate, *Services* handle computations or policies, and the *Database* persists all changes. The **interaction viewpoint** helps validate that responsibilities are allocated properly – for instance, all data writes happen in the data layer (no direct file or memory hacks), and all role checks happen in controllers or via the framework (to ensure security at runtime).

1.7 Data Model View

The Data Model view describes the **persistent data structures** of the system – essentially the database schema and how the data entities relate () (). The Course Registration System uses a Microsoft SQL Server database, with tables corresponding to the main entities in the domain. The schema is designed to support multi-role workflows, course offerings per term, and tracking of registrations and approvals.

Entity-Relationship Diagram: The figure below shows an ER diagram of the database, including tables, primary keys, foreign keys, and relationships:


AcademicSchedules
ScheduleID
AcademicYear
Semester
SemesterStartDate
SemesterEndDate
Student
StudentID
Email
Password
NIC
FullName
Nationality
Photo
TempAddress
PermanentAddress
Gender
PhoneNo
AcademicYear
DepartmentID
StaffID
Registrations
RegistrationID
OfferingID
StudentID
Semester
Attempt
RegistrationDate
ApprovalDate
TimeStamp
Results
OfferingID
StudentID
InCourse
EndMarks
Grade
CourseOfferings
OfferingID
CourseCode
AcademicID
Semester
StartDate
EndDate
StaffID
Notification
NotificationID
ReceiverMessage
SenderMessage
ThirdPartyMessage
Type
IsRead
CreatedAt
ReceiverID
ReceiverType
SenderStudentID
SenderStaffID
SenderType
ThirdPartyStudentID
ThirdPartyStaffID
ThirdPartyType
IsForAllStudents
AcademicYear
RelatedStudentID
RelatedStaffID
RelatedSessionID
RelatedCourseCode
RelatedStatus
RelatedPendingRegistrationID
PendingRegistrations
PendingID
SessionID
CourseCode
StudentID
Status
RegistrationDate
ApprovalDate
Attempt
IsApprovedByAdvisor
CourseOfferingID
Staff
StaffID
Email
Password
NIC
FullName
PhoneNo
DeptID
Role
Department
DepartmentID
DepartmentName
StaffID
RegistrationSessions
SessionID
AcademicYear
Semester
IsGeneralProgram
DepartmentID
StartDate
EndDate
IsOpen
RegistrationSessionCourses
SessionID
CourseCode
Course
CourseCode
CourseName
Credit
Semester
CoreOrElective
DeptID
HasPrerequisites
CourseCode
PrerequisiteCode

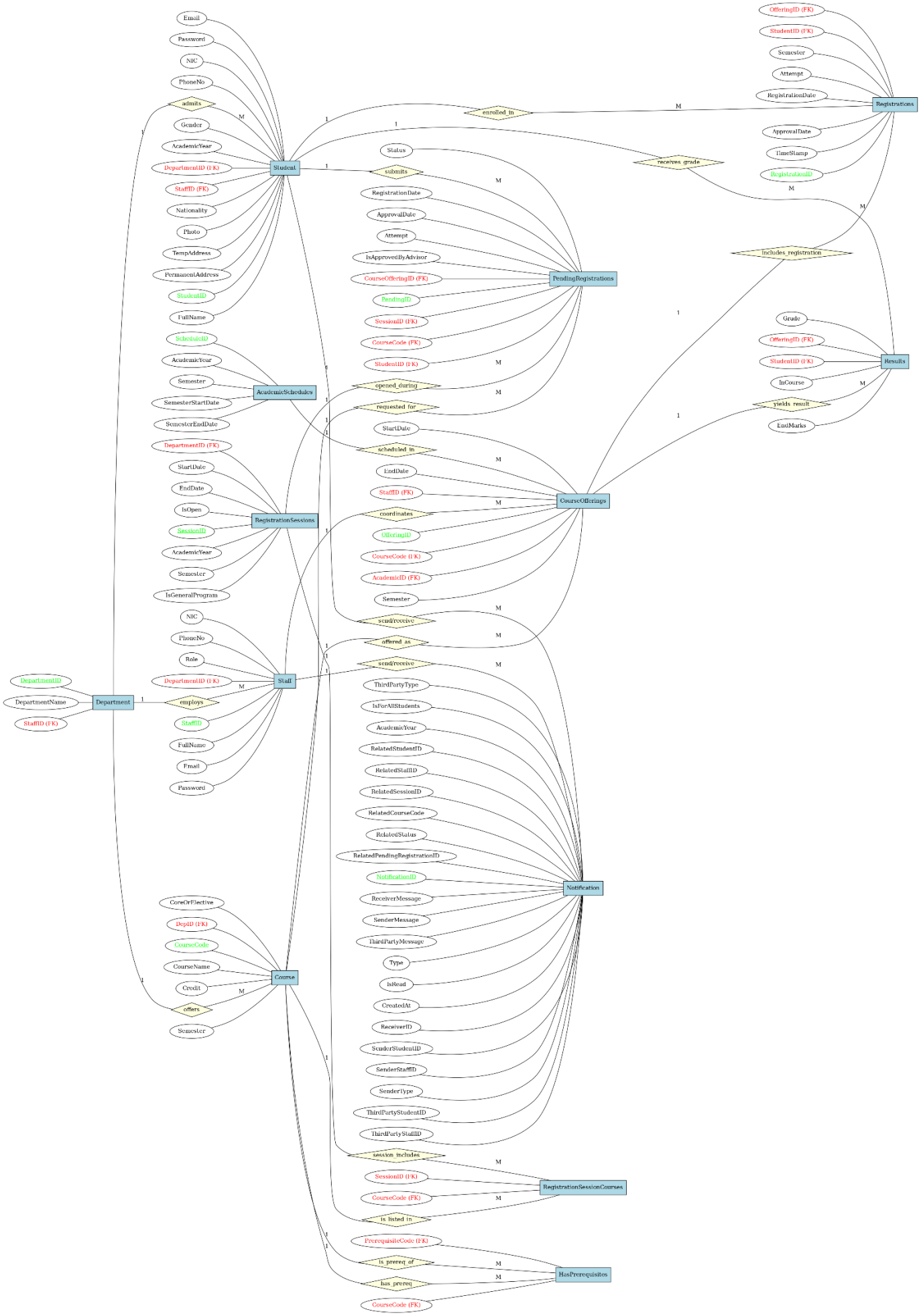


Figure 5: Entity-Relationship Diagram of the Course Registration System database.

From the ER diagram, we identify the main entities and relationships:

- **Student –** Stores student information. StudentID is the primary key (likely an integer). Other fields include Email, Password, personal info (NIC, FullName, etc.), contact info, AcademicYear, DepartmentID, and StaffID.
    - The StaffID in Student is a **foreign key referencing the Staff table**, indicating the advisor assigned to that student (each student has one advisor).
    - DepartmentID links to Department to note the student's department.
    - A Student has a *Role* field as well – in this system, presumably all entries in Student are actual students (so Role might be "Student" or could be unused here and only used in Staff).
- **Staff** – Stores staff (faculty/administrative users) info. StaffID (PK), Email, Password, FullName, etc., and Role.
    - The Role field in Staff differentiates AR Officers, Advisors, and Course Coordinators (and possibly other staff roles if needed). For example, an Advisor entry in Staff would have Role = "Advisor". (It appears AR Officer is treated as Admin in code, possibly Role = "AR" or "Admin").
    - Staff also has DepartmentID (which department they belong to, if applicable).
    - There's a self-referencing or cross reference: Department has a StaffID (likely the head of department), but Staff linking to Department is straightforward many-to-one.
- **Department –** DepartmentID (PK), DepartmentName, and possibly a StaffID (FK) indicating the head of department or coordinator. In context, Department is used to group students and courses.
    - The diagram shows StaffID in Department as well, likely referencing a staff member who is head or coordinator of that department.
- **Course –** Represents a course (a subject) offered by the faculty. CourseCode is the primary key (as a code like "CS101"). Other fields: CourseName, Credit (credit value), Semester (perhaps which *typical* semester it's offered or number of semesters long), CoreOrElective (flag), DepID (foreign key to Department indicating which department offers the course).
    - The **HasPrerequisites** table is an associative entity for Course prerequisites. It likely has a composite PK of (CourseCode, PrerequisiteCode) both referring to Course.CourseCode. This indicates that one course may require another course as prerequisite. (In the diagram, *HasPrerequisites* table connects to Course twice – one link labeled "Requires" or similar, indicating a course has many prerequisites, and each prerequisite relates to many courses – a many-to-many self-relationship resolved by this table).
- **AcademicSchedule –** Possibly stores academic year and semester info. The diagram shows AcademicSchedules with ScheduleID (PK), AcademicYear, Semester, StartDate, EndDate. It might define the timeline for each semester or term.

- The CourseOfferings table has an AcademicID which likely links to AcademicSchedule.ScheduleID (i.e., which academic schedule a course offering is part of).

- **CourseOfferings –** This table links a Course to a specific semester and instructor. Fields: OfferingID (PK), CourseCode (FK to Course), AcademicID (FK to AcademicSchedule), Semester, StartDate, EndDate, StaffID (FK to Staff, indicating the instructor or coordinator for that offering), Department (possibly FK to Department as well, though DepID might be redundant since course already has dept).
  - Essentially, **CourseOffering** = an instance of Course taught in a particular academic term by a particular staff. Students register for course offerings via the registration process.
  - CourseOffering is related to **Results** (see below) and to **Registration** (each Registration likely points to the Offering the student is enrolled in).
- **RegistrationSessions –** Represents a *registration session/window* during which students can register for courses. Fields: SessionID (PK), AcademicYear, Semester, DepartmentID, StartDate, EndDate, IsGeneralProgram, IsOpen.
  - This allows possibly multiple overlapping sessions (e.g., one for general courses, one for department-specific electives). If IsGeneralProgram is true, it might mean the session is for all students; otherwise DepartmentID might indicate it's a session only for that department's students.
  - "IsOpen" indicates if the session is currently accepting registrations.
  - There is a linking table **RegistrationSessionCourses** (with SessionID + CourseCode as composite PK or both FKs) which lists which Courses are available in that session. This is how the AR Officer assigns courses each semester: by populating RegistrationSessionCourses linking a session to each offered Course (the presence of a Course in a session also implies a CourseOffering will exist for it with that academic year/semester).
- **PendingRegistrations –** This table holds registration requests that are *awaiting or undergoing approval*. Fields: PendingID (PK), SessionID (FK to RegistrationSessions to know which cycle), CourseCode (FK to Course), StudentID (FK to Student), Status, RegistrationDate, ApprovalDate, Attempt, IsApprovedByAdvisor (boolean), CourseOfferingID.
  - **Status** might be an enum or string such as "Pending", "AdvisorApproved", "Rejected", etc. The presence of both Status and IsApprovedByAdvisor suggests Status may capture final outcome ("Approved" or "Rejected"), while IsApprovedByAdvisor specifically tracks the advisor's action.
  - **Attempt** could track if this is the student's first attempt or a reattempt (repeat) of the course.

- **CourseOfferingID** likely links to the specific offering that the student will be enrolled in if approved. Possibly the system determines the CourseOffering (e.g., finds the Offering for the selected CourseCode and current semester).
- This table is central to the approval workflow – it queues up student requests for processing. The use of a separate PendingRegistrations table (apart from final Registrations) allows the system to maintain a clean record of *official* enrollments vs. *requests in progress*.

- **Registrations** – This table holds finalized (approved) course registrations (enrollments). Fields: RegistrationID (PK), OfferingID (FK to CourseOfferings), StudentID (FK), Semester, Attempt, RegistrationDate, ApprovalDate, TimeStamp, etc.
  - Once a PendingRegistration is approved by the AR Officer, it likely results in a new Registration record. The *OfferingID* ties it to a specific course in a specific term.
  - Having Semester and Attempt in this table might be partially redundant (OfferingID implies term, but they might store Semester for quick access; Attempt denotes if it's a repeat attempt).
  - The separation of Pending vs. Registration means the system can also keep a history: a rejected request might remain in PendingRegistrations with a status, while only approved moves to Registrations.
- **Results** – Stores grade/marks for students in course offerings. Fields: OfferingID (PK part), StudentID (PK part) – together likely the primary key (each student per offering), plus fields like InCourse (internal assessment score), EndMarks (final exam score), Grade.
  - This table is populated at the end of a course's term to record outcomes. It links to Registration because effectively a Result should exist only if the student was registered in that offering. Possibly the primary key of Results is also a foreign key to Registrations (though the diagram shows separate, it could be implicitly linking by OfferingID+StudentID).
  - Course Coordinators access this to see the performance of students in their course, and advisors can view their advisees' results.
- **Notification** – Table for system notifications or messages. This table is a bit complex, with many fields: NotificationID (PK), ReceiverMessage, SenderMessage, ThirdPartyMessage, Type, IsRead, CreatedAt, ReceiverID, ReceiverType, SenderStudentID, SenderStaffID, SenderType, ThirdPartyStudentID, ThirdPartyStaffID, ThirdPartyType, IsForAllStudents, AcademicYear, RelatedStudentID, RelatedStaffID, RelatedSessionID, RelatedCourseCode, RelatedStatus, RelatedPendingRegistrationID.
  - This design supports a very flexible notification system. It can capture who sent the notification and who is the receiver (which could be a student or staff, hence IDs and Types to distinguish).
  - It can also optionally involve a "third party" – for example, if an advisor (sender) notifies a student (receiver) about another staff (third party) involvement.

- The flags like IsForAllStudents allow broadcasting a notification to all students.
- The Related* fields link the notification to context: e.g., RelatedPendingRegistrationID can tie a notification to a specific pending registration record (like "Your request ID 123 was approved").
- In practice, when the system says "Notify Advisor" or "Notify Student", it likely creates a Notification row populating these fields appropriately. For instance, after a student submits a request, the system might create a Notification with ReceiverID = Advisor's StaffID, ReceiverType = Staff, SenderStudentID = that student, Type = "RegistrationSubmitted", RelatedPendingRegistrationID = the new request ID, etc. The advisor's UI can then show this message.

As seen, the **data model is normalized** to avoid duplication: e.g., we don't store advisor name on each registration, we store advisor in Staff and link via Student->Staff relationships. This ensures consistency (if an advisor changes, the Student's StaffID can be updated, pending requests automatically reflect the new advisor).

Relationships and Cardinalities:

- One **Staff (Advisor)** can advise *many Students* (one-to-many Staff–Student via Student.StaffID).
- One **Staff (Coordinator)** can coordinate *many CourseOfferings* (one-to-many Staff–CourseOffering via CourseOffering.StaffID).
- One **Department** has many Students, many Staff, and many Courses.
- One **Course** can have many CourseOfferings over time (usually one per term, but possibly multiple offerings if multiple sections).
- One **RegistrationSession** can cover many Courses (via RegistrationSessionCourses linking table). Conversely, a Course can appear in many sessions (each semester it's offered).
- One **Student** can have many PendingRegistrations (they may request multiple courses) and many Registrations (enrolled courses). Each PendingRegistration and Registration is for exactly one Student.
- One **CourseOffering** can have many PendingRegistrations and Registrations (many students registering for the same offering). Each Registration is linked to one CourseOffering, and each PendingRegistration ideally to one CourseOffering as well (if the student selected a specific course, the system knows which offering that corresponds to).
- **PendingRegistration** and **Registration** have a relationship: typically, an approved PendingRegistration leads to a Registration. In the schema they are separate; the PendingRegistrationID might be referenced in Notification or might simply be removed upon approval. There isn't a direct foreign key between them (they serve different stages).
- **HasPrerequisites** relates Course to Course (many-to-many). From the diagram, we interpret cardinality: a Course can have multiple prerequisite courses; a Course can be a prerequisite for multiple other courses.

- **Results**: One CourseOffering + Student yields at most one Results entry (composite PK ensures uniqueness). Over time, a student could retake a course in a new offering, resulting in another Results entry for that different Offering.

The **data model supports the workflows** as follows: Students register by creating PendingRegistrations tied to a session and course. Approvals update those entries. Final enrollment is recorded in Registrations. The separation ensures that data such as *Results* (grades) are only associated with confirmed registrations (since only those appear in offerings).

Consistency & Integrity: The use of foreign keys (enforced by the database and represented in the EF Core model) ensures, for example, that a PendingRegistration refers to an existing Student, Course, and Session. If a student is deleted (which in practice would be rare or handled carefully), cascade rules or checks would prevent orphaned pending records. The database would also enforce uniqueness where appropriate (e.g., maybe a unique constraint on a Student cannot have two PendingRegistrations for the same Course in the same Session).

By designing according to the domain, the data model can answer key queries efficiently:

- *What courses is Student X registered for this semester?* → Query Registrations joined with CourseOfferings for the current AcademicID.
- *Has Student X passed Course Y?* → Query Results for Student X and Course Y (join through Offering and Course).
- *How many students enrolled in Course Z (offering W)?* → Query count of Registrations for that OfferingID.
- *Which pending requests are awaiting AR approval?* → Query PendingRegistrations where IsApprovedByAdvisor=true but not yet finalized (Status still pending).
- *Who is the advisor of Student X?* → Follow Student.StaffID to Staff.FullName.
- *List prerequisites of Course C?* → Query HasPrerequisites for CourseCode = C to get PrerequisiteCodes (then join Course to get names).

The **Entity-Relationship diagram** provides a visual summary of the database design, which underpins all application logic. It closely matches the code-first C# model classes used in the application (each class corresponds to a table and navigation properties correspond to these relationships). This alignment between the conceptual data model and the implementation ensures that the code and database remain in sync through migrations.

1.8 Deployment View

The deployment view describes the **physical environment and how the software is deployed** on hardware or hosting infrastructure. Currently, the system is not deployed to a public server, but we describe the target deployment architecture for a production or test environment.

Deployment Environment: The Course Registration System is a web-based application that will be deployed on a web server and accessed by users via web browsers over a network. There are two main runtime components to deploy:

- The ASP.NET Core MVC **Web Application** (which includes all controllers, services, and UI views).
- The **SQL Server Database** which stores the persistent data.

In a possible deployment scenario, these can be hosted either on the same machine or separate machines depending on scale and security needs. For the faculty environment, a typical setup might be:

- An **IIS or Kestrel Web Server** hosting the ASP.NET Core application on a Windows Server (or Linux server with .NET Core). This is the application server that clients connect to.
- A **SQL Server Database instance** running on a database server (could be Microsoft SQL Server on Windows, or Azure SQL if cloud, etc.), accessible only by the application server.

Figure 6 illustrates a basic deployment diagram for production:

Figure 6: Deployment Diagram – Client-Server deployment with web server and database server.

In Figure 6, we see:

- **User Browser (Client):** End-users (students, advisors, etc.) use standard web browsers from their PCs or devices. The client side just runs the browser; no special software installation is needed. They connect to the web application using HTTPS (to ensure encrypted communication, given that sensitive information like passwords and grades are transmitted).
- **Web Server (ASP.NET Core MVC Application):** The Course Registration System is published to a web server. This could be an on-premises server in the faculty's network or a cloud VM. The application runs as a process (possibly under IIS or as a standalone Kestrel server behind IIS/Nginx). This server hosts all the application logic. It listens for HTTP/HTTPS requests on a certain URL (domain or IP). When requests come in (e.g., student logs in or posts a registration), the server executes the appropriate controller code and responds with HTML pages or redirects. The web server needs network connectivity to the database server but does not expose the database directly to clients.
- **SQL Server Database:** The database can reside on a dedicated database server (for performance and security isolation). The application connects to it using a connection string (likely via TCP, default port 1433, using credentials). In a production environment, the connection would be secured (encrypted and using authenticated logins). The Web Server communicates with SQL Server using the TDS protocol (Tabular Data Stream) which we label as "SQL Connection" in the diagram. The database server stores all data on disk and handles concurrent access, backups, etc.

Current Hosting (Development): In the current development phase, the web app and database are often on a single developer machine (with the database as a local SQL Express instance). The deployment view above is forward-looking to a multi-server setup. The architecture does not

change whether it's one machine or two – thanks to the layered design, the web app simply needs the connection string to the DB. The team can easily move the DB to another host by updating the connection string configuration, without code changes.

Scalability and Future Deployment Considerations:

- The 3-tier architecture allows scaling horizontally if needed. For example, if many users start using the system, we could run multiple instances of the web application behind a load balancer, all pointing to a central database (or a cluster). Session state in this app is minimal (likely using cookies for auth, and short-lived context for each request), so scaling out the stateless web tier is feasible.
- The database could be scaled up (more CPU/RAM, or a managed Azure SQL for reliability).
- Since there are no external integrations, deployment is relatively straightforward: deploy the web app to an IIS site or container, ensure the configuration points to the correct DB server, and that's it. No additional services are required on the host aside from the .NET runtime and database engine.

Security in Deployment: The deployment should consider network security:

- Use HTTPS for client->server to protect credentials and data in transit.
- The database server should be firewalled such that only the web server can talk to it (e.g., via an internal network or secure VPN, not open to the public Internet).
- The web server should enforce authentication (already handled by the app) and perhaps integrate with faculty's single sign-on if needed in future (currently it has its own login system).
- Regular backups of the database server should be scheduled, as the data (registrations, grades) is critical.

To summarize, the Course Registration System will be deployed as a **web-based client-server application**: user browsers connect to the **ASP.NET Core web server**, which in turn persists data on a **SQL Server database**. This separation ensures that the heavy data operations are handled by the database, and the client is thin (only rendering UI). The deployment architecture aligns with the logical 3-tier model we designed, providing a clear path to host and scale the system in a real environment.

1.9 Security & Access Control

Security in the Course Registration System is primarily about **authentication and authorization** – ensuring that only legitimate users access the system, and that each user can only perform actions permitted for their role. Additionally, basic data security (password hashing, using HTTPS, etc.) is considered.

User Authentication: The application uses a login system for two user types:

- Students (credentials stored in the Student table).

- Staff (which includes Advisors, Course Coordinators, AR Officers – credentials in the Staff table).

Upon login, the AccountController authenticates the user by verifying the provided email and password against the database. Passwords in the database are stored in (presumably) hashed form (the intention is to hash passwords for security – e.g., using an algorithm like BCrypt or SHA256 with salt). On successful login, the system creates an authentication cookie storing the user's identity (user ID) and role. We use ASP.NET Core's cookie authentication middleware to manage this. The user's session is then identified by that cookie on subsequent requests, meaning the user doesn't have to re-login on each page.

Role-Based Authorization: The system defines four roles (as mentioned: Student, Advisor, Coordinator, AR Officer (Admin)). **Role-based access control (RBAC)** is enforced throughout the application:

- Each account in the database has a role attribute (Student.Role might be always "Student"; Staff.Role could be "Advisor", "Coordinator", or "AR"). This is mapped to a role claim in the authentication cookie.
- Controllers and actions are decorated with the [Authorize] attribute specifying allowed roles. For example, the AdvisorController might have [Authorize(Roles = "Advisor")] at the class level, meaning all its actions require an Advisor login. Similarly, AdminController has [Authorize(Roles = "AR")] (or "Admin"). This is consistent with ASP.NET Core's standard role-check mechanism (Role-based authorization in ASP.NET Core | Microsoft Learn) – before an action executes, the framework checks the user's roles and will return 403 Forbidden if not authorized.
- Some controllers might allow multiple roles or different roles per action. For instance, the HomeController or DashboardController might be accessible by any authenticated user but then redirect based on role. In those cases, we might use [Authorize] without specifying role (meaning any logged-in user).
- The [Authorize] attribute approach is a declarative way to protect routes; it is effective because it's built into the ASP.NET Core pipeline. We also have additional programmatic checks inside actions for finer control. For example, when an Advisor tries to approve a student's request, the AdvisorController will not only require the Advisor role but also verify that the student's record is indeed one of their advisees (matching Student.StaffID). This prevents an advisor from approving someone else's students by tampering with IDs.
- AR Officer (Admin) rights are the highest – the AdminController likely allows role "AR" only. AR Officers can view and modify all pending registrations, courses, etc. There may also be an *Admin Dashboard* view that shows summary statistics; this too is locked to AR role.

Data Access Security: Because the application controls all data queries through the controllers and Entity Framework, we can ensure users only fetch data they're allowed to see:

- Students: In StudentController actions, the code always uses the current logged-in student's ID (from the auth context) to query their own data. There's no functionality for a student to arbitrarily query others' data. Even if a student attempted to manipulate a query parameter

(like guessing another registration ID), the controller would cross-check that the registration belongs to that student. For example, ViewRegistrationDetails(int regId) would internally do something like: find Registration by ID, then ensure Registration.StudentID == currentUserId. If not, it returns a NotFound or Forbidden.

- Advisors: An advisor can only retrieve info about their advisees. This is enforced by queries that filter by Student.StaffID = advisor.Id. The advisor has no direct access to other staff or students outside their purview. Any attempt to load data not tied to them would either yield nothing or be explicitly blocked by the controller.
- Coordinators: A course coordinator's abilities are limited to courses they teach. The CoordinatorController when loading course enrollments will filter CourseOfferings by StaffID = coordinator.Id. They cannot just view any course; the UI only offers their courses, and the backend double-checks the association.
- AR Officer: By design, the AR role is all-powerful in this system – AR can see all students, all registrations, etc. This is intended (as they manage the whole process). Even so, AR actions are segregated in the AdminController, which normal users cannot access at all.

Input Validation & Protection: The system likely uses model binding and data annotations to enforce validation rules (like [Required], [StringLength] on fields). This prevents malformed data from being accepted. Also, ASP.NET Core MVC by default includes anti-forgery tokens for form posts, which helps prevent CSRF (Cross-Site Request Forgery) attacks – e.g., the login form or course registration form will include an anti-forgery token such that a malicious site cannot post on behalf of a logged-in user.

Password Security: As mentioned, passwords are stored hashed (the code comments indicate "Password – stored in hashed format"). During authentication, the plaintext password from user input is hashed with the same algorithm and compared to the stored hash. This ensures that even if the database were compromised, the actual passwords are not exposed. In the current implementation, the team may have used a simple hash or the Identity library's PasswordHasher. This could be an area to double-check (ensuring a strong algorithm and salting are used).

Session Management: The system uses ASP.NET's cookie authentication, which issues a secure cookie. We ensure this cookie is marked HttpOnly (not accessible via JavaScript) and Secure (only sent over HTTPS) to mitigate XSS and network sniffing risks. Session timeout policies can be configured – for instance, the cookie might expire after 20 minutes of inactivity (forcing re-login). This prevents unauthorized use if someone leaves a browser unattended.

Audit Logging: While not fully specified, the database design inherently logs certain actions – e.g., PendingRegistration has timestamps and who approved (advisor/AR). We know who performed actions by looking at related IDs. If needed, the system could output logs on the server for key events (login attempts, data changes), but at minimum the database can be used for audit trails:

- PendingRegistration.Status and dates show when approvals happened.
- The Notifications table also serves as an audit of communications (who was notified of what and when).

Access Control Summary by Role:

- **Student:** Can only view/modify *their own* data. Allowed to:
    - Register for courses (creates PendingRegistration for self).
    - View their registrations history and grades.
    - Edit their profile (if implemented).
    - Cannot see other students' info, cannot approve anything, cannot modify courses.
- **Advisor:** Can:
    - View advisee students' profiles and results (only those where Student.StaffID = AdvisorID).
    - View pending course requests of advisees and approve/reject them.
    - Possibly send messages to advisees (via Notifications).
    - Cannot see or affect students of other advisors, cannot manage courses or sessions.
- **Course Coordinator:** Can:
    - View courses (offerings) they are coordinator for, including enrolled students and their grades.
    - Enter or update student results for their course (if result entry is part of system, though not explicitly mentioned, often coordinators/instructors would input grades – the presence of a Results table suggests input of grades is done, presumably by coordinators).
    - They cannot view info for courses they don't coordinate, and have no authority on approvals or registrations beyond viewing their course's roster.
- **AR Officer (Admin):** Can:
    - Access everything relevant to registration management: create sessions, manage course lists, view all pending registrations, override or finalize any registration, see all students and staff data.
    - Essentially full read-write access across the system's data (within the application's provided UI). This role should be granted to very few users.
    - The AR Officer cannot impersonate others per se, but they have their own screens to see the data. If any action requires acting "on behalf" (like marking advisor approval), the system typically wouldn't allow that; AR can directly approve final registration but not pretend to be an advisor – advisors have to do their part or AR could reject if advisor hasn't approved (depending on policy).

Security of External Communication: Since the system is planned to be accessed over a network, using HTTPS is recommended (and implied in the deployment diagram). This ensures that credentials (email/password) and other sensitive info (grades, personal data) are encrypted in transit. Within the faculty network, if it's all intranet, HTTPS is still good practice. Also, since no third-party APIs are used, we don't have to handle external API keys or secrets.

Development Security Practices: The team leveraged the framework's security features (e.g., the [Authorize] attribute) rather than writing custom checks from scratch – this is beneficial because it's less error-prone and aligns with tested patterns (Role-based authorization in ASP.NET Core | Microsoft Learn). By using role attributes, the system is essentially self-protecting at the controller level – one cannot simply craft a URL to an admin action as a student; the framework will block it.

In summary, the system employs a **defense-in-depth** approach for access control:

- Authentication ensures only registered users enter.
- Role-based authorization ensures users only reach the functions they should.
- Additional logic in controllers ensures data-level protection (each query is scoped to the user's context).
- The database design, by linking records to user IDs, inherently supports these checks.
- Sensitive data (passwords) is protected by hashing, and communications are secured via HTTPS.
- There is no anonymous access except perhaps the login page; everything meaningful requires sign-in.

This security model is adequate for the current scope. In the future, improvements could include integrating with the university's single sign-on or directory (so credentials aren't stored in our DB), implementing two-factor authentication for AR officers, etc. But as it stands, the architecture addresses the critical need: **each user role has access to a defined subset of the system, and the enforcement of those boundaries is built into the design**.

1.10 Design Decisions and Rationale

Throughout the development of the Course Registration System, the team made several important design and technology decisions. Here we explain those choices and the rationale behind them:

- **Choice of ASP.NET Core MVC:** The team selected ASP.NET Core MVC (C#) for the backend. This decision was driven by familiarity with the technology and its suitability for building a robust web application within the given timeframe. ASP.NET Core provides an integrated framework for implementing an MVC architecture with minimal boilerplate, built-in security features, and easy connection to SQL Server (via Entity Framework). Using ASP.NET Core aligned with the requirement of a web-based solution and allowed the use of the MVC pattern to separate concerns. It also ensured easy deployment on Windows (IIS) which is commonly available in university IT infrastructure. The decision to use server-side rendering (Razor views) over a single-page application was due to simplicity and the primarily form-driven interface. This means less complexity on the client side and leveraging server-side HTML generation, which is appropriate for the target environment (campus network, mostly desktop users).
- **3-Tier Layered Architecture:** The architecture was deliberately structured into a **3-tier layered model** – presentation, application, data. This follows industry best practices for web apps and is recommended by architecture documentation standards (Understanding the architecture of a 3-tier application). The rationale was to achieve **separation of concerns**

(Presentation.pptx): UI designers can work on HTML/CSS without touching business logic, and database changes can be handled in the data layer without affecting UI code. This layered approach improves maintainability and testability. For example, if a decision was made later to open an API for mobile apps, the team could reuse the application layer (controllers/services) or add Web API controllers that call the same services, thus reusing business logic – fulfilling the *reusability* goal (Presentation.pptx). Another benefit is scalability: each layer could potentially be scaled or replaced (the database could be changed to another RDBMS or moved to a cloud service with minimal changes to code).

- **Use of Entity Framework Core (Code-First):** Instead of raw SQL or a different ORM, the team used EF Core in code-first mode. This means C# model classes define the schema, and EF Core migrations are used to create the SQL Server schema. The rationale includes:
    - Productivity: EF Core eliminates a lot of boilerplate SQL. The team can work with high-level C# objects and collections, making development faster and less error-prone.
    - Maintainability: The model classes serve as a single source of truth for the data structure. Changes in the model (e.g., adding a field) can be propagated via migrations, ensuring the code and DB schema remain in sync.
    - EF Core automatically handles relationships and can enforce referential integrity (via cascade rules configured in the model). It also provides LINQ querying, which integrated nicely with business logic.
    - Given that no complex stored procedures or specialized DB tuning was needed (project of this scope can rely on EF's generated SQL), using EF was an appropriate trade-off for convenience.
- **Custom Authentication vs. Identity Framework:** The project did not fully implement ASP.NET Core Identity (which is a comprehensive membership system) and instead did a simpler custom auth (verifying against Student/Staff tables). The rationale here could be:
    - Simplicity: Identity would introduce many tables (AspNetUsers, AspNetRoles, etc.) and complexity not strictly needed since roles and users are already naturally part of the domain (students and staff).
    - Control: By using their own tables, the team could integrate authentication with the existing data (e.g., using the same Student table to store login info avoids duplicating student records in an identity user table).
    - However, the decision meant implementing some things manually (password hashing, role checking). The team leveraged [Authorize(Roles="...")] which can work with custom principal setup, which they achieved using cookie auth and claims (Role-based authorization in ASP.NET Core | Microsoft Learn).
    - This decision was likely acceptable given the context, but in a production scenario, using Identity or an SSO system might be revisited for stronger security and maintainability.

- **Role-Based Access Control:** Deciding to hard-code role distinctions (Student, Advisor, etc.) into separate controllers and use role attributes was a clear design choice. The rationale is clarity and security. By separating logic by role (different controllers/views), the UI and code could be tailored to each user type, reducing the chance of unauthorized functionality appearing where it shouldn't. It also simplifies development since each controller assumes a single role context (e.g., AdvisorController doesn't need to constantly check "if current user is advisor" – that's guaranteed). This follows the principle of *least privilege*, giving each part of the code only the permissions it needs.
- **Use of Notifications Table:** The design includes a flexible Notifications system, which might be beyond minimal requirements. The rationale for implementing it (or planning it) is to have an audit and messaging mechanism in-app. Rather than sending emails (which could be external integration and was avoided by constraints), they created a table to track notifications. This allows the system to show, for example, a bell icon with pending approvals for advisors or a list of announcements for students. It also decouples messaging from immediate actions – e.g., a notification can be created in the DB and read later by the user. This decision enhances user experience within the app without relying on external email or SMS services.
- **UML and Documentation Approach:** Though not part of the running system, a decision was made to document the architecture using **multiple views** (context, logical, process, etc.), following IEEE 1016 and "Views and Beyond" guidance. This is a meta-decision to ensure maintainability and clarity of the design. By preparing proper documentation, the team acknowledged that communicating the architecture is as important as building it (Views and Beyond Collection) (Views and Beyond Collection). This will help future maintainers and also ensured the team thought through the design from different angles (which likely improved the design quality).
- **No Third-Party Integrations:** By constraint, the system avoids third-party APIs. The design decision here was to keep the system *self-contained*. For instance, rather than integrating an email API for notifications or a payment API for any fees (if that was in scope), they omitted those. The rationale is simplicity, focusing on core functionality first. It also avoids complications of network calls, API keys, and error handling for external systems. In the given context (internal faculty system), this is acceptable because users (students/staff) will log in regularly to check status, and critical notifications can be communicated in-person or by internal means if needed.
- **Using Standard Web Technologies (HTML/CSS/Bootstrap):** For the front-end, they chose server-side Razor views with Bootstrap for styling. The rationale is to achieve a responsive, clean UI without needing a dedicated front-end framework (like React or Angular). Bootstrap provides ready-to-use components and a grid system, speeding up UI development. Razor views allow embedding C# for dynamic content, which is straightforward for developers already working on the C# backend. This decision kept the technology stack unified (all in .NET/C# except the HTML styling) and reduced the learning curve. It's sufficient given that the UI flows are form-based and not highly interactive beyond standard navigation.
- **Design for Extensibility:** The architecture shows some foresight for extension:

- The separation of CourseOffering, RegistrationSession, etc., indicates they planned for scenarios like multiple sessions, perhaps short semesters or special programs. This means if the university introduces a summer semester or department-specific enrollment windows, the design can accommodate it (just add sessions or toggle IsGeneralProgram).
- The use of a service for closing sessions means tomorrow they could implement an automated scheduler (e.g., Windows Task or a hosted service) to run that without admin intervention.
- The presence of fields like Attempt in registration allows tracking multiple attempts at a course (for GPA calculations). This aligns with actual academic policies where students can repeat courses. Including it from the start was a design decision to meet that requirement.
- The team mentioned "Extendable for mobile apps later" (Presentation.pptx) as a rationale for their architecture. While they didn't implement a separate API now, their layered design would support adding a Web API controller layer on top of the service/data layer to serve a mobile app. Because the logic isn't tightly coupled to the web UI (it's in controllers and services), they could expose the same actions via JSON endpoints relatively easily.

- **Trade-offs and Alternatives Considered:** It's worth noting any alternatives and why they were not chosen:
  - *Alternative 1:* Using a single combined table for users instead of separate Student and Staff tables. They chose separate, likely because students and staff have different attributes. This simplifies queries (no need for a generic User table with columns for both). The trade-off is duplicate handling for auth, but since roles are distinct, this made sense.
  - *Alternative 2:* Using stored procedures for complex operations (like approving registration could be a stored proc updating multiple tables). They instead did it in the application logic (C#) for clarity and because the logic is not too performance-heavy. This keeps logic in one place (the C# code) rather than spread into the DB layer. It aids maintainability as well (developers might be more comfortable debugging C# than SQL).
  - *Alternative 3:* Not using a PendingRegistrations table (i.e., just mark registrations with an "approved" flag). They introduced a separate table, which adds complexity but improves clarity of workflow and data integrity. It prevents mixing pending and final records, reducing confusion in queries (e.g., you don't have to always filter out unapproved ones when looking at official enrollments).

Each of these decisions was made to **balance functionality, complexity, and future needs**. The result is an architecture that is **organized, modular, and aligned with the system requirements** and quality attributes. According to IEEE 1016, a good design should address stakeholders' concerns () () – here the stakeholders (university AR office, students, faculty) needed a system that is reliable, secure, and easy to use/maintain. The chosen architecture addresses those concerns by using

proven frameworks and patterns, and by clearly delineating the responsibilities of different parts of the system.

In conclusion, the architecture of the Course Registration System reflects deliberate design choices aimed at building a secure, scalable, and maintainable application. By leveraging a layered MVC design, role-based security, and a well-structured data model, the team has created a foundation that meets current requirements and can be extended or scaled to meet future demands.

2.1 Development Environment & Tools

The Course Registration System is built with **ASP.NET Core MVC (C#)** on the **.NET 8.0** framework, using **Visual Studio** as the primary IDE. The team used GitHub for version control and collaborated via tools like Zoom/Teams. The backend leverages **Entity Framework Core (EF Core)** for the database (SQL Server) and uses standard ASP.NET Core project structure. Frontend development is done with **Razor Views** (server-rendered HTML), styled by **Bootstrap CSS** and custom CSS. Client-side interactivity uses **jQuery** (for AJAX calls) and unobtrusive validation. The system follows a 3-tier architecture: a clear separation between the presentation layer (Razor views & controllers), the application logic (services/controllers), and the data layer (EF Core models and context) .This separation of concerns makes the project organized, scalable, and easier to maintain or extend in the future.

2.2 Code Organization and Structure

The project is organized into logical folders following ASP.NET MVC conventions:

- **Models:** Contains EF Core entity classes (e.g. Student, Staff, Course, etc.), plus a ViewModels subfolder for view-specific DTOs (e.g. LoginViewModel). These classes define the data schema and are annotated for EF Core.
- **Data:** Contains the ApplicationDbContext (EF Core DbContext) which configures the database sets and relationships. It also includes migration files for schema changes.
- **Controllers:** Each user role has a dedicated controller (e.g. StudentController, AdvisorController, AdminController, CoordinatorController) to handle requests for that module. Additionally, AccountController manages authentication, and a DashboardController maps each role to its dashboard view. Controllers use **[Authorize]** attributes to restrict access by role. For example, the Student controller actions are [Authorize(Roles = "Student")], and similarly for Advisor, AR (admin), and Coordinator.
- **Views:** Razor views are organized under subfolders by role or feature. For instance, Views/Dashboard/Student/ contains student-facing pages, Views/Dashboard/Admin/ contains admin (AR officer) pages, etc. There are also shared partial views (e.g. for validation scripts). Each Razor view is an HTML page with embedded C# for dynamic content. *(Note: The project does not use a shared _Layout.cshtml; each view defines its full HTML structure, which is a simple approach for this application.)*
- **Services:** Contains business logic classes that are not directly tied to UI. Examples include CreditTrackingService (for enforcing credit limits and attempt counts), SemesterService (for date and semester utility logic), and CloseExpiredSessionsService (a hosted background

service to close registration sessions after their end date). These services are registered in DI and used where appropriate (e.g. the background service runs automatically on startup).

- **wwwroot:** Static assets like CSS, JS, and libraries. The project includes Bootstrap and jQuery libraries under *wwwroot/lib*. A default *site.js* is present (though mostly unused) and custom scripts are embedded directly in views as needed.

Overall, the structure reflects a classic ASP.NET Core MVC solution with clear separation of concerns. The **controllers orchestrate data from the Data layer to the Views**, often calling services or EF Core queries for business logic, and then passing models or ViewData to the Razor views for rendering.

2.3 Presentation Layer (Frontend UI)

2.3.1 MVC Controllers and Razor Views

The presentation layer is implemented with ASP.NET Core MVC controllers returning Razor views. Each page or UI function has a corresponding controller action and view: for example, the Advisor's "Advisee Students" list is generated by AdvisorController.AdviseeStudents() and rendered in *Views/Dashboard/Advisor/AdviseeStudents.cshtml*. Controllers prepare any needed data (often via EF Core queries or by calling services) and pass it to views using model objects, ViewData, or ViewBag. The Razor views use this data to render dynamic content, mixing HTML with C# syntax.

For instance, when a **Student** logs in, the AccountController identifies the user's role and redirects to the appropriate dashboard (e.g. students go to StudentController.Dashboard which returns the *StudentDashboard.cshtml* view). That view in turn presents links to student functions like course registration, viewing registered courses, results, etc. Similarly, an **Advisor** who logs in is taken to the advisor dashboard view (*AdviserDashboard.cshtml*), and so forth for **Admin (AR Officer)** and **Coordinator**. Each of these main dashboard views provides navigation into the module's sub-pages.

Because no site-wide layout is used, each Razor view includes the necessary HTML <head> and navigation elements for that page. Bootstrap is referenced for a consistent look and feel (ensuring responsive design and basic styling). Form inputs and tables in the views utilize Bootstrap classes. Client-side validation is enabled via data annotations on models and the inclusion of jQuery Validation scripts (the standard *_ValidationScriptsPartial.cshtml* is included in forms). This means forms will show immediate feedback if required fields are missing, etc., without needing a full server round-trip.

2.3.2 UI Navigation Flow

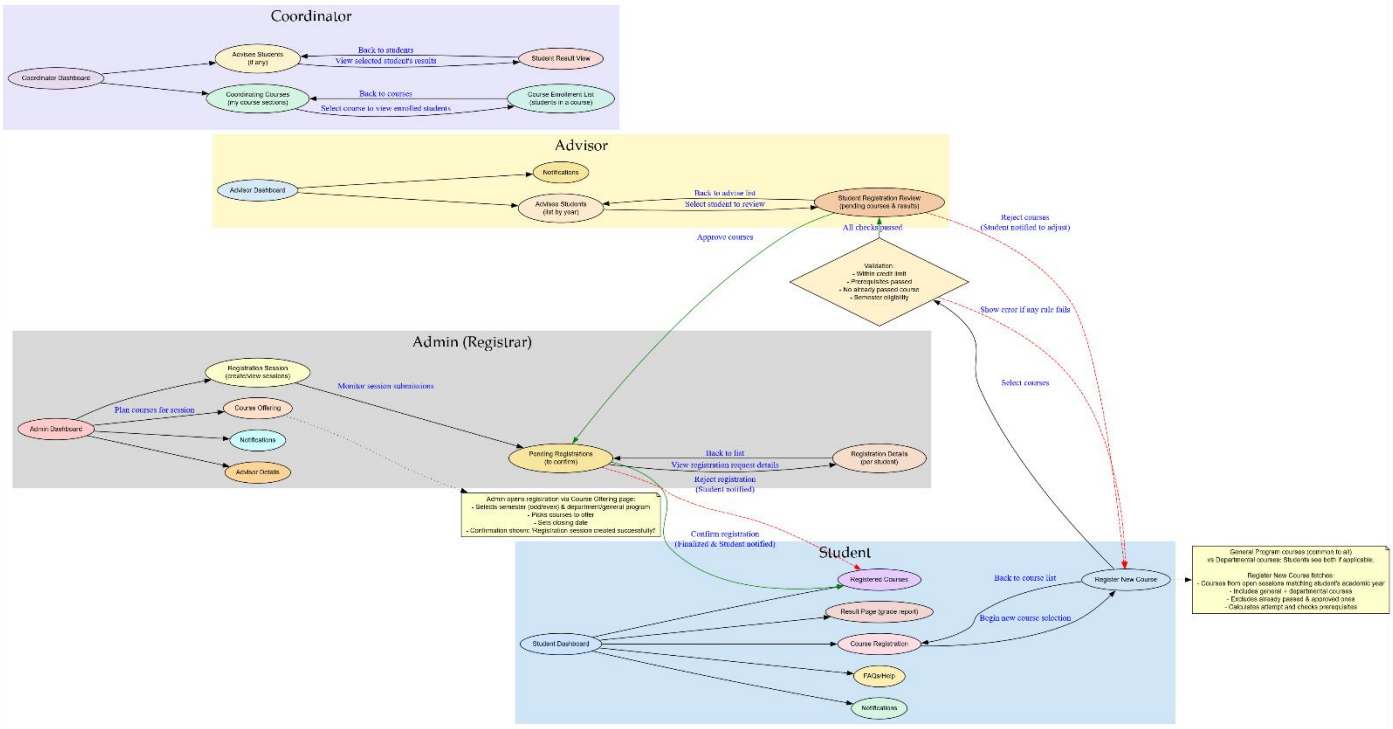


Figure 2.3.2: High-level UI navigation flow for each user role (Student, Advisor, Admin/Registrar, Coordinator).

Each user role has a distinct UI flow, aligned with their permissions and tasks. Upon login, users see a role-specific **dashboard** with menus for their available actions. The diagram above illustrates the main navigation and interactions for each role:

- **Student:** The student dashboard provides access to view *Registered Courses*, the *Course Registration* page, *Results* (grade report), *Notifications*, and help/FAQ. From "Course Registration", a student can begin a new registration request. The UI guides the student to **select courses** from those available in the current open registration session. When the student submits a new course selection, the system performs validation checks (e.g. credit limit, prerequisites, no duplicate courses) – if any check fails, an error is shown immediately (the student must adjust selections). If all checks pass, the request is recorded as pending and the student is notified that it's awaiting approval. The student can then view their pending or registered courses in the Registered Courses page, and later see final status updates (approved or rejected) in Notifications.
- **Advisor:** An academic advisor's dashboard shows options to view their *advisee students* and any pending course registrations requiring approval. The advisor can select a student from their advisee list to review that student's registration request and academic info. The *Student Registration Review* page (for a selected advisee) displays the student's requested courses alongside their current results/GPA for context. The advisor can then either **approve all courses** (if they meet academic requirements) or **reject** some/all courses. Approving will mark those requests as advisor-approved so they move on to the AR Office for final confirmation. Rejecting will immediately update the request status to rejected and trigger a notification to the student (so the student can adjust and resubmit) – e.g. "Reject courses

(Student notified to adjust)" in the flow. After taking action, the advisor can navigate back to their advisee list or dashboard.

- **Admin (AR Officer/Registrar):** The AR admin dashboard includes management functions: *Plan courses for session*, *Course Offering*, *Pending Registrations*, *Advisor Details*, and *Notifications*. **Planning courses for a session** involves creating or opening a new registration session for a given academic year/semester and department (or general program) in the *Registration Session* page, then assigning available courses to that session in the *Course Offering* page. The admin selects which courses (from the master course catalog) will be offered for the semester and can assign a coordinator (instructor) to each course offering. Once a session is open, student registrations flow in. The *Pending Registrations* page allows the AR to monitor all course registration requests that are awaiting final confirmation (i.e. those approved by advisors but not yet finalized). The admin can click into *Registration Details* for a specific student's request to review the courses, check statuses (advisor approval, etc.), and then **confirm or reject** the registration. Confirming will finalize the student's enrollment in those courses (creating official registration records) and notify the student (and possibly their advisor) that the registration is approved. If the admin rejects a request (for example, if an issue was overlooked or capacity is full), the student is notified so they can attempt a different course. The admin can also use *Advisor Details* to view which staff are assigned as advisors to which students, and manage any advisor-role information.
- **Coordinator (Course Coordinator/Instructor):** The coordinator dashboard shows two main sections: *Advisee Students* (if the coordinator also serves as an academic advisor to some students) and *Coordinating Courses* (the courses that this coordinator is teaching or overseeing in the current semester). The coordinating courses page lists all course offerings for which the staff member is the assigned coordinator. The coordinator can select a course to view the *course enrollment list* – i.e. the roster of students enrolled in that course and possibly their in-class assessments. From there, the coordinator can also view a *Student Result View* for any student in their class, which shows that student's overall performance (grade/GPA) in the course or overall. This helps the coordinator monitor class performance and identify if any student is struggling. Coordinators can also navigate back to their dashboard or other courses easily. (Note: Coordinators do not approve registrations, but they can see how many students have enrolled in their courses and the students' academic profiles, fulfilling the requirement to view enrollment numbers and student GPAs.

Overall, the UI navigation is role-driven: menus and pages are enabled or disabled based on the user's role. The use of separate controllers and views for each role, combined with ASP.NET's authorization, ensures that each type of user only sees the relevant navigation items. The interface is designed to be straightforward, with each action (registering courses, approving requests, etc.) guided by confirmation prompts or status messages so users know the outcome.

2.3.3 Client-Side Behavior

Client-side scripting is used sparingly, primarily to enhance user experience for form submissions and dynamic data without full page reloads. The project includes **jQuery** and **Bootstrap JS**, which are used for interactive components and AJAX calls. Key client-side behaviors include:

- **AJAX Course Selection (Student Registration):** When a student opens the "Register New Course" page, a script triggers an AJAX request (using jQuery's $.ajax) to fetch the list of available courses for that student's academic year and department. The endpoint /Student/GetAvailableCourses returns JSON data of courses (the controller action is marked with [HttpGet] and returns Ok with a JSON result). The client script then populates the course selection UI (e.g. a multi-select or list of checkboxes) with these courses. This happens asynchronously so that any changes (like session closing or course availability) are reflected in real-time. The code logs messages to console (e.g. "Calling GetAvailableCourses API…") and then populates a JavaScript array of courses for filtering in the UI.
- **Submitting Registrations via AJAX:** The student's course registration submission is also handled via AJAX. Rather than doing a traditional form POST and page reload, the "Submit" button gathers the selected courses and sends them as a JSON payload to the /Student/SubmitCourseRegistration API (an [HttpPost] action). This call is done with fetch or jQuery AJAX, and on success, the client either refreshes the list of registered courses or displays a confirmation message (e.g. "Registration request submitted"). This improves responsiveness – the student can remain on the same page and see immediate feedback. A similar approach is used by advisors: when approving/rejecting a student's pending courses, the advisor's page sends a JSON with the decisions for each course to /Advisor/SubmitApprovals. The use of JSON APIs for these operations decouples the UI from the view rendering, effectively treating them as micro-APIs for the frontend.
- **Dynamic Content and Validation:** Bootstrap's components (like modals, tooltips) are utilized via their JS, though the UI is fairly straightforward. The project also references **Chart.js** on the admin dashboard (included via a CDN script) – this suggests the intention to display summary charts (perhaps for number of registrations or credit distribution). For example, the Admin Pending Registrations page includes Chart.js; if implemented, it could show a chart of pending vs confirmed registrations. (In the current code, the chart may be a placeholder since no explicit chart initialization code is present.)
- **Form Validation:** All forms use ASP.NET's built-in validation with unobtrusive JavaScript. The presence of jquery.validate and jquery.validate.unobtrusive scripts means any <input> with data annotation validation will automatically enforce rules on the client side. For instance, the login form and course selection forms will show errors if required fields are empty or inputs are invalid, without needing a server round-trip. This ensures quicker feedback to users.
- **UI Feedback:** The client scripts also handle minor UX feedback like showing loading indicators or enabling/disabling buttons based on state. Console logs (e.g., console.log("API called…")) in the code suggest the developers used them during development to trace behavior. In a production scenario, these could be removed or replaced with user-visible messages.

In summary, the frontend relies on **server-side rendering** for the main UI and uses **JavaScript/jQuery for incremental updates** (fetching data and posting decisions). This hybrid approach keeps the core logic on the server (maintaining consistency with the MVC pattern) while still providing a smooth user experience for critical interactions like course selection and approvals.

2.4 Application Layer (Backend Business Logic)

2.4.1 Business Logic Services

The application layer encompasses the C# logic that enforces rules and processes user actions. While some logic is implemented directly in controllers, the project also defines dedicated service classes for reusable or background tasks:

- **CreditTrackingService:** This service is responsible for tracking and validating credit-based rules for course registration. It checks how many credits a student has registered for and how many times they have attempted a course. For example, when a student submits a set of courses, the service could calculate the total credit load and ensure it does not exceed the maximum allowed per semester. It also can determine the "attempt" number for each course by checking past **Results** records (i.e., if a student is reattempting a failed course, attempt count increases). In the code, before saving a PendingRegistration, there is logic grouping past results by course to get attempt counts. This ties into assigning Attempt in the pending registration (e.g., a course the student is taking for the second time will have Attempt = 2). The service centralizes these calculations so that both the Student controller and potentially the Advisor/Admin logic can use consistent rules (e.g., preventing registration if a course was already passed or if it's a repeat beyond allowed attempts).
- **SemesterService:** This service likely provides utilities related to academic terms and sessions. For instance, it could compute current academic year/semester based on date, or format semester names, or help determine if a registration window is open. In planning courses for a session, the admin chooses an academic year and semester; SemesterService might validate those values or provide the list of possible semesters. It might also be used in conjunction with CreditTrackingService (for example, to fetch current semester credit limits).
- **CloseExpiredSessionsService:** Implemented as a *Hosted Service* (registered with AddHostedService), this background service periodically checks for any open registration sessions that have passed their end date. Its job is to automatically "close" those sessions by setting IsOpen to false. This ensures that once the registration deadline is reached (EndDate), students can no longer submit new registrations for that session. The service likely runs on a timer (e.g., checking every few minutes or hours). This automation relieves the admin from manually closing sessions and prevents late submissions. Any session closure might also trigger finalization steps (though in this project, primary finalization is manual by the AR).
- **Notification Service (planned):** Although not explicitly a service class, the logic for handling notifications can be considered part of the application layer. The system was designed with a **Notification** model (table) to store messages for users (students and staff). When certain events occur (e.g. an advisor approves or rejects a request, or an admin confirms a registration), the business logic will create notification entries to inform the relevant parties. For example, if an advisor rejects a student's course, the application layer would create a notification record where the *sender* is the advisor, the *receiver* is the student, and the message indicates the rejection reason. Similarly, when the admin approves a registration, a notification can be recorded for the student (and possibly the advisor) indicating the request was approved. The notification model supports various fields (ReceiverMessage,

SenderMessage, etc.) and links to relevant entities (related student, staff, course, session) (DB_Schema_1.pdf) (DB_Schema_1.pdf), enabling rich context. While the current code includes controller actions to view notifications, the creation of notifications is likely implemented within the workflow logic (e.g., in the advisor's SubmitApprovals method, after marking as rejected, add a notification record for the student). The presence of the Notification table and its fields shows the intent to implement a robust messaging system, even if some wiring might be incomplete in code.

- **Direct Controller Logic:** In addition to services, much of the business logic is directly in controller actions (which is typical for a project of this scope). For instance, the StudentController.SubmitCourseRegistration method contains logic to validate selected courses: it filters out courses the student already has pending or registered, computes the attempt number for each, and only then saves new PendingRegistration records for valid courses. Similarly, AdvisorController.SubmitApprovals contains logic to update each PendingRegistration: setting IsApprovedByAdvisor = "Yes" for approved courses or Status = "Rejected" for rejections, then saving changes. By handling this in the controller, the application ensures business rules are applied at the point of request handling. In a larger system, these might be refactored into service methods (e.g., a RegistrationService to handle the entire workflow), but here it is straightforward and tied to the MVC actions.

All these components together enforce the **business rules** of course registration: credit limits, prerequisite checks, multi-step approvals, session availability, etc. They also handle cross-cutting concerns like notification and data consistency. The use of EF Core means that these business operations are typically transactional (each controller action that modifies data calls SaveChanges() once, which wraps the changes in a transaction).

2.4.2 Workflow Implementation

Student Course Registration Workflow: The multi-step course registration process is the core workflow of the system. It unfolds as follows:

1. **Opening a Registration Session (Admin):** The process starts with the AR Officer (Admin) creating a new registration session for the upcoming semester. In the Admin interface, the **RegistrationSession** page allows specifying the academic year/semester, whether it's a general session (for common courses) or department-specific, the registration period start/end dates, and marking it open. When saved, this creates a RegistrationSession record (and the system will allow student registrations only while IsOpen is true). The admin then goes to the **Course Offering** page to populate this session with available courses. Behind the scenes, for each course the admin selects to offer, a RegistrationSessionCourse entry is created linking that session and course.If needed, the admin can also assign an instructor (coordinator) which might create/update a CourseOffering record for that course and semester. At this point, the session is live: students of the appropriate year/department can see these courses as available for registration.
2. **Student Submission (Pending Registration):** A student uses the *Course Registration* page to choose courses from the open session. When the student submits their selection, the system (via StudentController.SubmitCourseRegistration) performs several validations:

- **Credit Load Check:** The total credits of selected courses is computed (using each Course's Credit value) to ensure the student does not exceed the maximum credit limit in that semester.
- **Prerequisite Check:** For each course, the system checks if the student has met prerequisites. This is done by looking up the HasPrerequisite table and verifying the student has passed those prerequisite courses. If any prerequisite is not satisfied, that course is flagged (and the front-end can display an error like "Prerequisite not met for Course X").
- **Already Passed or Registered Check:** The system ensures the student is not registering for a course they have already passed or currently registered. It queries the student's existing Results (grades) and Registrations to filter out courses with a passing grade or active registration. It also checks PendingRegistrations to avoid duplicate requests. In code, the query filters out any selected course that the student already has a pending record for.
- **Attempt Assignment:** For each remaining course, the attempt number is determined. The code groups past results by course to count how many times the student took each course. If a student is registering for a course they failed before, Attempt will be previous attempts + 1; if it's their first time, Attempt = 1. This is stored in the PendingRegistration (and later carries to the Registration record) to track how many times the course has been attempted.
- **Status Initialization:** The newly created PendingRegistration entries are initialized with Status = "Pending" and IsApprovedByAdvisor = "No" (as they await advisor action). The current date is recorded as RegistrationDate. If a CourseOffering (specific class section) exists for that course, its ID is linked; otherwise, that field is left null until final assignment. The system adds all valid PendingRegistration records via _context.PendingRegistrations.AddRange(...) and saves changes. At this point, the student's request is in the **PendingRegistrations** table with status "Pending" (DB_Schema_1.pdf) (DB_Schema_1.pdf). The student will see their request as "pending approval" in their Registered Courses view. Also, ideally, a notification would be generated to the student's advisor informing them of a new request (the system knows the student's advisor via the Student's StaffID field).

3. **Advisor Approval (Pending -> Advisor Approved):** The student's academic advisor is notified (through the app's UI) of the new pending request. On the advisor's dashboard, a counter or list shows how many pending registrations are awaiting their review. The advisor opens the advisee's request (via *AdviseeStudentsResult.cshtml* page). The controller (AdvisorController) fetches the PendingRegistration records for that student (still with Status "Pending" and IsApprovedByAdvisor "No") and likely also the student's transcript (Result records) to display current grades/GPA. The advisor reviews the courses and decides to approve or reject each. The advisor can then submit their decision. The AdvisorController.SubmitApprovals action processes the decisions: for each course in the request, if the advisor marked it "Approved", the system updates that PendingRegistration's IsApprovedByAdvisor field to "Yes"; if "Rejected", it updates the Status to "Rejected". All

changes are saved in one transaction. Any courses the advisor rejects effectively end the workflow for those courses – the student will be notified of the rejection (via a notification entry and seeing the status change). For courses approved by the advisor, the request remains pending but now carries an advisor approval. In the database, those records still have Status = "Pending" but IsApprovedByAdvisor = "Yes". This indicates they are **pending final admin approval**. The advisor's action date could be recorded (the design has an ApprovalDate in PendingRegistration which might be used for either advisor or admin – in practice, advisor setting "Yes" could trigger filling an AdvisorApprovalDate, but the current schema has one ApprovalDate which is likely used at final approval) . At this point, the advisor's part is done. *(If all courses were rejected, the process stops here for those entries; the student would have to submit a new request if needed.)* Approved ones move forward.

4. **Admin Confirmation (Pending -> Final Registration):** The AR Officer (admin) can now see advisor-approved requests in the Pending list. The *PendingRegistrations* admin page filters to show only those with Status = "Pending" (still pending final action) – possibly highlighting those where **IsApprovedByAdvisor = "Yes"** so the admin knows advisor has cleared them. The admin selects a pending request (by student) to view details. In *RegistrationDetails.cshtml*, the admin sees the list of courses, the advisor's decision on each, and possibly the student's current credit total or other info. The admin then chooses to **confirm** (approve) or reject the request. If the admin confirms, the system will:
    - Create a permanent **Registration** record for each approved course. This involves moving data from Pending to Registration: the student ID, the course offering ID, attempt number, etc., into the Registrations table. A new RegistrationID is generated for each. The Registration record represents the student's official enrollment in that course for the semester. It also likely copies the timestamp of approval.
    - Update the PendingRegistration record's Status to "Approved" (or some final status) and set its ApprovalDate to now (this field now clearly becomes the date of admin approval). This marks the pending entry as completed. The design keeps the PendingRegistration for history (with status updated to distinguish it from still-pending ones). Alternatively, the system could delete the pending entry, but maintaining it with a status is useful for audit trail.
    - If the admin **rejects** at this stage (perhaps due to capacity issues or discovering an error), the PendingRegistration's Status is set to "Rejected" (and likely IsApprovedByAdvisor stays "Yes" but now final status is rejected). No Registration record is created in that case.
    - Any rejection or approval here would trigger notifications: e.g., if approved, a notification to the student like "Your course selection has been approved by the AR Office" (and possibly to the advisor as well, to close the loop); if rejected, a notification to the student explaining the admin did not approve (with reason if provided). These would use the Notification system mentioned earlier.

After admin confirmation, the workflow for those courses is complete: students can now see the courses in their *Registered Courses* list as officially registered. The Registration records will later be

used to track outcomes (grades). The system also updates any related counts (e.g., how many students registered for a course, which the coordinator can view). If the registration session is still open and within deadlines, students could submit another request (for additional courses or a new attempt if one was rejected). The design thus supports iterative submissions within the open window. Finally, when the session's EndDate passes, the **CloseExpiredSessionsService** will mark it closed (IsOpen=false), preventing further submissions. The multi-level approval workflow ensures that advisors vet the academic suitability of course selections first, and the AR office then finalizes enrollment – matching the intended business process of the faculty. (The system monitors both stages – advisor and AR – as noted in requirements: *"AR staff should be able to monitor whether student advisors have approved or rejected registrations"* .)

2.4.3 Notification Processing

Notification handling is a supportive workflow that runs in parallel to the main processes, ensuring that users are informed of important events. The application uses a **Notification** entity/table to record messages. Each notification record can include: a receiver (Student or Staff) with a message tailored to them, an optional sender and even a third-party related entity, plus flags for read/unread status. This flexible schema allows the system to capture different scenarios: one-to-one messages (e.g., advisor to student), one-to-many (e.g., admin to all students), and even multi-party contexts (like a notification visible to a student and their advisor regarding the same event). Key points of notification processing:

- **Triggers:** Notifications are created by the backend logic when certain actions occur. For example, when a **student submits** a course registration, the system might create a notification for the advisor: "Student X has submitted a registration request for your approval." Similarly, when an **advisor makes a decision**, the system generates a notification for the student: either "Your course selection was approved by Advisor Y and is awaiting final approval" or "Advisor Y has rejected your course selection – please revise and resubmit." Finally, when the **admin confirms or rejects**, the student gets notified of the outcome. These notifications ensure that at each stage, the relevant party doesn't need to constantly check a status – they have a message log to refer to. The code supports different sender/receiver types (the Notification table has fields for SenderStudentID, SenderStaffID, ReceiverID, and even ThirdParty IDs to cover all combinations). For instance, in an advisor rejection notification: *SenderStaffID = [AdvisorID]*, *ReceiverID = [StudentID]*, *ThirdPartyType = "Staff"* (or similar to reference the advisor for display). In an admin approval notification: *SenderStaffID = [AdminID]*, *ReceiverID = [StudentID]*, and perhaps *ThirdPartyStaffID = [AdvisorID]* to cc the advisor.*
- **Viewing Notifications:** Each role has a notification page (StudentNotification.cshtml, AdvisorNotification.cshtml, AdminNotification.cshtml) accessible from their dashboard. When loaded, the controller fetches relevant notifications for that user. For example, StudentController.Notification() will query notifications where ReceiverID matches the student's ID or a broadcast flag is set (e.g., IsForAllStudents) (DB_Schema_1.pdf). It then passes them to the view for display. The notifications are typically listed in reverse chronological order with an indicator if they are unread (IsRead flag). When the user opens

the page, the system may mark them as read. This provides an in-app messaging hub for users to review any actions taken on their registrations or other related events.

- **Internal Implementation:** While the structure is in place, the exact creation of notifications in code is subtle. The controllers currently prepare the views (setting ViewData with the current user or counts), but the insertion of Notification records likely occurs inside the workflow methods (which might not be fully shown in the code snapshot). Given the design, whenever Status or approval fields change on a PendingRegistration, the system could create a new Notification. For example, in AdvisorController.SubmitApprovals, after setting a pending record to "Rejected", a new Notification might be added: Notification.Type = "RejectNotice", ReceiverID = studentId, SenderStaffID = advisorId, RelatedPendingRegistrationID = pendingId, Message = "Advisor X rejected your course Y." and so on. If multiple courses are rejected, the system might combine them or send separate notifications. The presence of fields like RelatedPendingRegistrationID allows linking a notification to the specific request record for easy reference.
- **Email/SMS (future):** The project does not show external email or SMS sending, but the Notification system could be extended to send emails for critical updates (especially to students). Currently, it functions as an in-app notification only.

In practice, the notification module keeps users informed without requiring them to manually check each status. This is crucial in a multi-step workflow – a student knows immediately when an advisor or admin has acted on their request. Advisors can also see a history of what they approved or rejected, and admins can see system-wide notifications (for example, confirmations that a registration session was created or closed). Though some implementation details are still basic (e.g. the Notification views are present but the creation logic might be incomplete), the design aligns with the system's needs for communication.

2.5 Data Access Layer

2.5.1 Database Models (ORM)

The system's data model is implemented via **Entity Framework Core** using code-first migration. The main entities and their relationships are illustrated in the UML diagram below, which reflects the structure of the database:

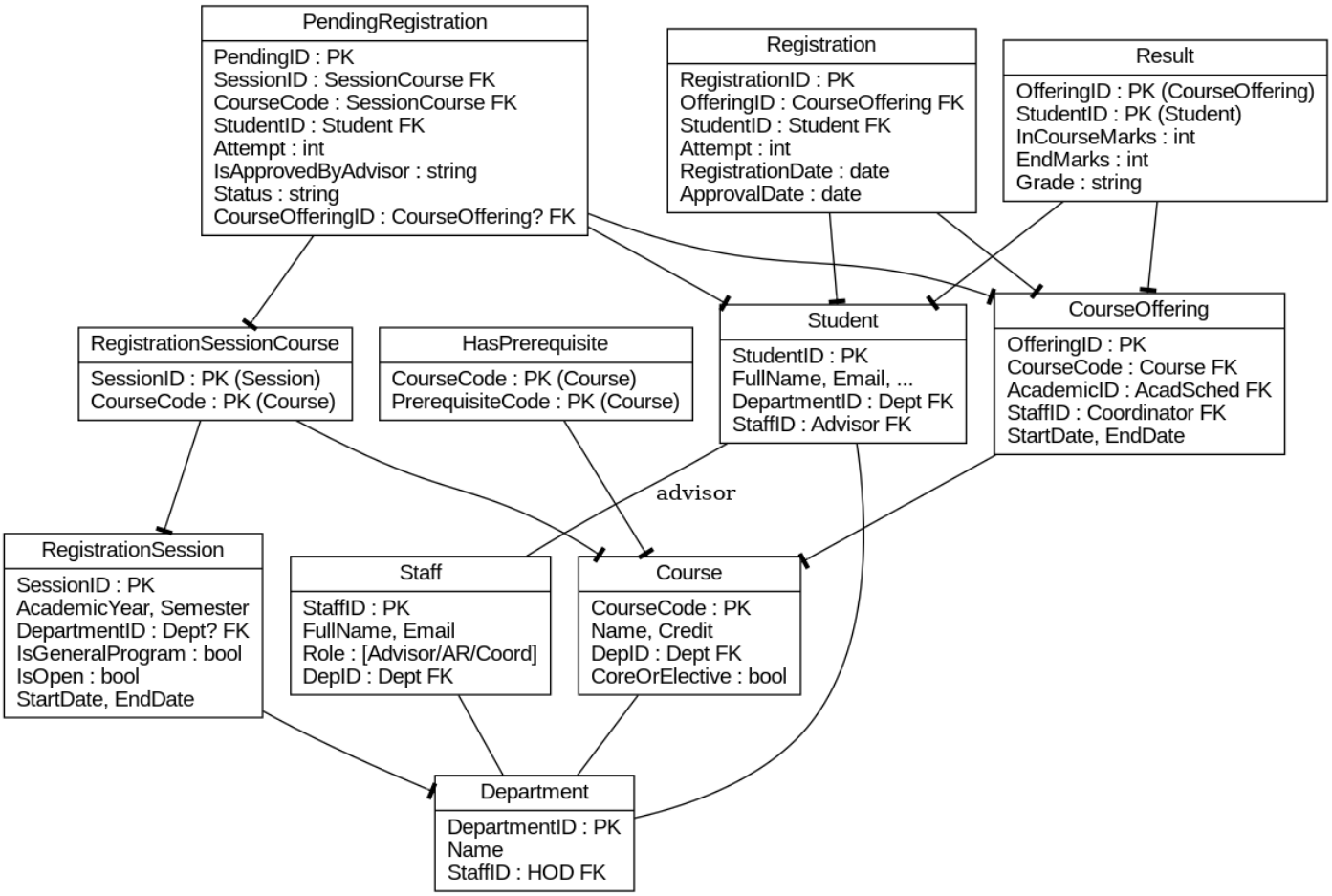


Figure 2.5.1: Simplified ER/UML diagram of the Course Registration System data model (primary keys, foreign keys, and relationships).

As shown above, the database is centered around Students, Staff (faculty/administrative users), Courses, and the registration workflow tables. Key models include:

- **Student:** Represents a student user with fields for personal information (Email, FullName, etc.) and academic info. Each Student has a DepartmentID (link to the Department they belong to) and a StaffID which references their academic advisor (a Staff member). In EF Core, this is set up as a one-to-many: one Staff (advisor) has many Student advisees. The Student model also contains properties like AcademicYear (year of study) and possibly GPA (computed on the fly from results).
- **Staff:** Represents faculty or administrative staff users (advisors, course coordinators, AR officers). Staff has login credentials (Email, Password) and personal details, as well as a DepID (department they belong to). A crucial field is Role which indicates their role in the system – e.g., "Advisor", "Coordinator", "AR" (academic registrar). This Role field is used

for authorization (e.g., [Authorize(Roles="Advisor")] on advisor controller). Staff may also have relationships: a Staff member can advise many students (inverse of the Student's advisor relation), and a Staff member can coordinate many course offerings.

- **Department:** Represents an academic department. It has a DepartmentID and a name. There is also a reference to a staff member as the Head of Department (HOD) – in the data model, Department has a StaffID that links to a Staff who is the HOD. Departments are used to categorize students, staff, and courses. A student's department and a staff's department are typically the same (e.g., an advisor in the CS department advises CS students).
- **Course:** Represents a course offered by the faculty. The primary key is a CourseCode (string). Each course has a name, credit value, and is associated with a department (DepID) and a flag for Core/Elective. The Semester field in Course indicates which semester(s) it's normally offered (though the registration session ultimately decides if it's offered in a given term). Courses serve as templates – actual offerings per term are separate.
- **HasPrerequisite:** A join table (many-to-many self-relationship for Course) that records prerequisite relationships between courses. It has a composite key: CourseCode and PrerequisiteCode (both foreign keys to Course). Each entry means "Course X has prerequisite Y". EF Core is configured to enforce cascading rules such that deleting a course will delete related prerequisite entries, etc., with restricted deletes to preserve referential integrity. This table is used when validating student registration requests to ensure the student has passed all prerequisites of a chosen course.
- **AcademicSchedule:** (Not shown above, but present in the DB) Represents academic calendar entries. Fields include AcademicYear, Semester, start/end dates. This is likely used by CourseOffering to link an offering to a specific term's schedule (AcademicID in CourseOffering corresponds to a ScheduleID). In newer design, RegistrationSession overlaps in purpose (storing year, semester, dates as well), so AcademicSchedule might be legacy or used only for reference.
- **RegistrationSession:** Represents a registration period for a specific semester and (optionally) department. Key fields: SessionID, AcademicYear, Semester, DepartmentID (nullable for sessions that are general to all students), a boolean IsGeneralProgram (true if the session is for general courses open to all departments), StartDate/EndDate, and an IsOpen flag indicating if students can currently register. When an admin "opens" course registration, they create a RegistrationSession. This model is central to scoping which courses students see: a student will query available courses from the sessions where AcademicYear matches their year (and either DepartmentID matches theirs or is general) and that are open. Only one session per dept per semester might be active at a time.
- **RegistrationSessionCourse:** A join table linking Courses to a RegistrationSession (composite key of SessionID + CourseCode). This indicates that a particular course is offered in that session. Essentially, when the AR officer assigns courses to a session, entries are created here. It's used when students fetch available courses: the query finds all CourseCode in the open session(s) for that student's year/department. The model also has navigation back to Course

and RegistrationSession. EF Core configures a composite foreign key here and cascades deletions if a session is removed (it will remove the linked courses).

- **CourseOffering:** Represents a specific offering of a course in a semester, potentially with an instructor. It has a generated OfferingID as primary key. It links to a Course via CourseCode, and ideally to an AcademicSchedule (AcademicID) and Staff (StaffID for the coordinator/instructor). It may duplicate some info from RegistrationSession (year/semester) or tie into it – for example, the system might create a CourseOffering for each combination of (Course + RegistrationSession) when assigning coordinators. In the database, CourseOffering includes StartDate and EndDate (which might mirror the semester dates). For simplicity, one can think of CourseOffering as "a class of a course in a specific term, taught by a specific instructor". This model is used to link final registrations and results to a class and to allow coordinators to view their class roster.
- **PendingRegistration:** Represents a student's **registration request** for a course, awaiting approval. It has a surrogate primary key PendingID. It also carries a composite foreign key (SessionID + CourseCode) linking to the RegistrationSessionCourse that the student is attempting. This effectively ties the request to which session and which course offering it relates to. Other fields include StudentID (the student who made the request), Status (string indicating status: typically "Pending", and later "Approved" or "Rejected"), RegistrationDate (when student submitted), ApprovalDate (when finalized by admin), Attempt (attempt number for that course) (DB_Schema_1.pdf), IsApprovedByAdvisor (tracks advisor's decision – "No" or "Yes" as strings in this design), and an optional CourseOfferingID. The CourseOfferingID is null until the admin approves; at approval time, the admin can assign the specific Offering (especially if multiple sections) or the system could auto-link if only one offering exists. The PendingRegistration is the heart of the workflow state machine: it starts as a new entry when student submits (Status=Pending, IsApprovedByAdvisor=No), gets updated by advisor (setting IsApprovedByAdvisor=Yes or Status=Rejected), and finally updated by admin (Status=Approved or Rejected with ApprovalDate). Business logic ensures only valid combinations occur (e.g., an admin will only approve if advisor field is "Yes" or if advisor step is bypassed).
- **Registration:** Represents an **approved enrollment** of a student in a course. This table is essentially the finalized registration record that can be used for class rosters and grade recording. It has an RegistrationID (PK), StudentID, OfferingID (linking to CourseOffering), Attempt, RegistrationDate, ApprovalDate, etc. Once a PendingRegistration is approved by the admin, a Registration entry is created. The Registration.Date and ApprovalDate likely mirror the Pending's dates (when it was requested and approved). If a student is registering in the general (common) session vs department session, the Semester and Department can be inferred via the linked CourseOffering or RegistrationSession, so those may not be separate fields here (the schema shows a Semester field, which might have been an earlier design choice). Each Registration corresponds to one course in one semester for one student.
- **Result:** Represents the **grade/outcome** of a student in a course offering. It has a composite key of (OfferingID, StudentID) – so one result per student per offering. Fields include InCourseMarks (continuous assessment scores), EndMarks (final exam score), and Grade

(final letter grade). This is populated at the end of the semester (likely by course coordinators or AR staff). Results are used to calculate GPA and determine if a student passes or must reattempt a course. When computing GPA, the system converts the Grade to grade points (via a method in StudentController, e.g., "A" => 4.0, etc.) and then averages based on credits. Students can view their Results (grade report), and advisors/coordinators can also view their advisees' or students' results.

EF Core is configured via ApplicationDbContext.OnModelCreating to establish all these relationships with the correct cardinalities and cascade rules. For example, the fluent API ensures that deleting a Department will cascade to its Staff and Students or be restricted if references exist, as appropriate. The Student–Advisor relationship is set as one-to-many with no cascade on delete (so deleting a staff won't delete students). The composite keys (like for HasPrerequisite, RegistrationSessionCourse, Result) are defined with HasKey(...) calls. Also, alternate keys and indexes can be inferred (e.g., ensuring uniqueness of certain combinations).

Overall, the **data layer** provides a normalized schema that captures the requirements: it separates the *transactional workflow data* (pending registrations and final registrations) from *master data* (students, staff, courses, sessions). This makes queries efficient and maintains data integrity. The system uses **code-first migrations** to build this schema – migrations such as *AddRegistrationSession*, *AddAttemptToPendingRegistrations*, etc., incrementally evolved the DB to meet new requirements during development. For instance, an early migration introduced the RegistrationSession tables, and another added the Attempt field once the reattempt logic was implemented. The final state of the schema (as depicted above) aligns with the project's functionality, supporting multi-role operations and ensuring each piece of data is properly related.

2.5.2 Data Access Components

Data is primarily accessed through the ApplicationDbContext using EF Core's DbSet properties. Each model has a corresponding DbSet<T> in the context (e.g., DbSet<Student> Students, DbSet<PendingRegistration> PendingRegistrations, etc.). Controllers use dependency injection to get the context (e.g., via constructor injection of ApplicationDbContext) and then use LINQ queries to retrieve or modify data. Some examples of data access in the code:

- The AccountController checks login credentials by querying context.Students and context.Staffs for a matching email/password. It uses EF's async LINQ (FirstOrDefaultAsync) to find the user.
- When listing pending requests for an advisor, the AdvisorController might query context.PendingRegistrations with a filter on StaffID and Status to count how many are waiting for that advisor. Similarly, AdminController queries PendingRegistrations to show counts on the admin dashboard (e.g., total pending count).
- More complex queries involve includes and groupings: for example, before a student's submission is saved, the code loads all past results and groups them by course to calculate attempt counts. It does context.Results.GroupBy(r => new { r.StudentID, r.CourseOffering.CourseCode })… which demonstrates EF Core's ability to navigate relationships (from Result to CourseOffering to Course) and perform aggregation.

- The context's OnModelCreating configures join tables and foreign keys so that navigation properties can be used. For instance, the PendingRegistration model likely has a navigation RegistrationSessionCourse (with composite FK) and Student. Thus, one could include Include(pr => pr.RegistrationSessionCourse).ThenInclude(rsc => rsc.Course) to get course details for each pending request. The code indeed shows usage of .Include for related data, e.g., including Course info when needed.
- **Transactions:** EF Core by default wraps SaveChanges() in a transaction. In cases where multiple tables are updated (e.g., creating Registration records and updating PendingRegistration status), all those changes are saved together so either all succeed or all fail, keeping data consistent.
- **Stored Data vs Computed:** Most data needed is stored explicitly, but some values (like current GPA) are computed on the fly. The controllers compute GPA by retrieving all of a student's results and calculating the average (the code uses a helper to convert grades to points and sums credits). This avoids storing GPA as a field, preventing inconsistency and ensuring it's updated whenever results are updated.

No custom repository pattern is implemented – the controllers call the DbContext directly. Given the project's scope, this is sufficient. The design relies on EF Core change tracking to handle updates: e.g., loading a PendingRegistration, modifying its properties (pr.Status = "Rejected"), then calling SaveChanges – EF Core generates the appropriate UPDATE statement.

2.5.3 Database Migration & Seeding

During development, the team used EF Core migrations to evolve the schema. The migration files (e.g., 20250319050935_initial.cs, 20250320063436_AddRegistrationSession.cs, etc.) show how tables were added and altered. For example, the initial migration would have created base tables like Student, Staff, Course, etc., while later migrations added the RegistrationSession and PendingRegistration tables as the workflow was fleshed out. The presence of an *AddAttemptToPendingRegistrations* migration indicates a new column was added to PendingRegistration to store the attempt number once the reattempt feature was introduced. Each migration corresponds to a point in time (the timestamp in the filename) during the project. The context's model snapshot (ApplicationDbContextModelSnapshot.cs) reflects the final schema for consistency checks.

Seeding: The project likely seeds some initial data for demonstration – for instance, some Department entries, a set of Courses, and a few Staff and Student accounts. Seeding could be done in the OnModelCreating or via a separate initializer. While the code excerpts don't explicitly show seeding, the included database backup (DB .bacpac) and the ER diagram suggest that sample data exists (the ER diagram shows several sample values). In a development environment, they might have hardcoded some users (for login credentials) and courses to allow immediate testing.

For example, they may seed an admin staff (AR Officer) account and a couple of advisors, plus one student with known credentials for demo. The *Presentation.pptx* hints at user IDs like *2021/E/190* which could be student IDs. It's reasonable the system expects numeric StudentID/StaffID but uses those as display (e.g., student roll numbers). If so, seeding ensures at least one advisor is linked to

the student, etc. There is also mention of a *DB5.bacpac* in the repository which likely contains the populated database schema and data for testing.

In production, such seeding would not be needed beyond administrative setup (like creating initial staff accounts and populating the course catalog). The majority of data (students, courses, departments) would come from integration with university systems or manual input by admins. For now, the focus is on ensuring that the schema supports all required data. The migrations confirm that all needed tables (including Notifications, which might have been added outside EF) are present in the final database.

In summary, the data access layer successfully maps the complex relationships of the course registration domain into a relational schema, and EF Core is used to interact with this schema in a convenient way. The combination of **code-first EF models** and **LINQ queries** makes the data layer robust yet easy to work with, allowing the developers to focus on business logic while EF Core handles SQL generation and database state.

3.1 API Overview

The Course Registration System provides a set of REST-style API endpoints for managing course offerings, student registrations, approvals, and academic records. These endpoints are organized by functionality and user role (Student, Student Advisor, Course Coordinator, AR Officer). All APIs return JSON responses and expect requests in either JSON or standard form data format as noted. Authentication is required for most endpoints (using the application's session cookie after login), and access is controlled via roles:

- **Student** – course enrollment and viewing own records
- **Advisor** – approve/reject advisee course registrations, view advisee records
- **Coordinator** – view courses they coordinate and enrolled students, view student records
- **AR Officer (Admin)** – create course offering sessions, manage registrations, final approvals, view all students

By convention, endpoints (except the Auth API) are exposed under the controller name (e.g. /Admin/… for AR Officer functions, /Student/… for student functions, etc.). The base path is the application root. The **Auth API** endpoints are prefixed with /api/ as they are intended for programmatic login.

3.2 Authentication & Authorization

Authentication Method: The system uses form-based authentication with ASP.NET Core Identity cookies. Users (students and staff) log in via the Login API to establish a session. On successful login (via form or API), a session cookie is issued which must be present in subsequent requests to authorized endpoints. There is no OAuth/JWT token issuance in the current implementation – the APIs rely on the session cookie for authentication.

Auth API – Login:

- **POST /api/auth/login** – Validates user credentials and starts a session.
    - **Request Body:** JSON object with:
        - email (string) – User's email (username)
        - password (string) – User's password
    - **Response:** On success, returns HTTP 200 with a JSON body containing user info and role. For example:

```
{
  "message": "Login successful",
  "role": "Student",
  "userId": 10001234,
  "fullName": "John Doe",
  "academicYear": 3
```

}

The role field will be one of "Student", "Advisor", "Coordinator", or "AR" (for admin). Staff logins respond with their role accordingly. **No JWT token is returned** – the client is expected to use the established cookie session for subsequent calls.

- **Error Responses:**
  - **400 Bad Request** – If email or password is missing in the request (e.g. {"message": "Email and password are required"}).
  - **401 Unauthorized** – If credentials are invalid (wrong email/password). Returns a JSON error message ({"message": "Invalid credentials"}) and no session is created.
  - **500 Server Error** – On unexpected server error during login (returns JSON with an error message).

Session & Authorization: After login, the user's session cookie must be included in subsequent requests. Endpoints are protected by role-based authorization: each API will only execute if the user's role matches the required role:

- Endpoints under /Student/* require the **Student** role.
- Endpoints under /Advisor/* require **Advisor** role.
- Endpoints under /Coordinator/* require **Coordinator** role.
- Endpoints under /Admin/* require **AR Officer** role.

If an authenticated user without the required role attempts access, the request will be denied (HTTP 403 Forbidden). If an unauthenticated request is made to a protected endpoint, the system will redirect to the login page (or return 401/403 in API contexts).

3.3 Course and Offering APIs

These endpoints allow administrators (AR Officers) to manage course offerings for registration, and allow course coordinators to view their offered courses. All endpoints in this section require authentication.

Admin: Filter Courses by Semester/Department – GET /Admin/GetFilteredCourses

Description: Retrieves the list of courses for a given semester, optionally filtered by department. Used by AR staff when preparing a registration session (to choose which courses to offer). **Auth:** Requires AR Officer role.
Request Parameters:

- semester (int, query) – Semester number to filter by (e.g. 1 or 2).
- departmentId (int, query, optional) – Department ID to filter by. If provided, only courses from that department are returned; if omitted or null, courses from all departments for the given semester are returned.

Response: JSON array of **Course** objects. Each course object includes:

- CourseCode (string) – Course code/ID.
- CourseName (string) – Name of the course.
- Credit (int) – Credit value of the course.
- DepartmentName (string) – Name of the department offering the course (or "Unknown" if not found).

Example:

GET /Admin/GetFilteredCourses?semester=1&departmentId=3

Response:


```
[
  {
    "CourseCode": "CS101",
    "CourseName": "Introduction to Computer Science",
    "Credit": 3,
    "DepartmentName": "Computer Science"
  },
  {
    "CourseCode": "CS102",
    "CourseName": "Data Structures",
    "Credit": 4,
    "DepartmentName": "Computer Science"
  }
]
```


Admin: Create Registration Session – POST /Admin/CreateRegistrationSession

Description: Creates a new course registration session (offering) for one or more student batches (academic years) in a given semester. This defines a set of courses that will be open for registration until a specified closing date.
Auth: Requires AR Officer role.
Request: This endpoint expects form data (URL-encoded or form-data). (**Note:** It is typically invoked via a form submit in the admin UI.) The form fields are:

- academicYears (array of string) – Academic year(s) of students for whom this registration session is opened. For example, ["3","4"] might indicate third and fourth-year students.
- semester (int) – Semester number for which registration is open (e.g. 1 or 2).

- departmentId (int, optional) – Department ID if this session is department-specific. For general (faculty-wide) sessions, this can be omitted or left null.
- closingDate (DateTime) – The end date of the registration period. After this date, the session will close for student registrations. (Format example: 2025-05-30T00:00:00 or as appropriate; the API expects a valid date/time format.)
- selectedCourses (list of string) – The list of course codes to offer in this session. These courses should correspond to the given semester (and department if specified).

Behavior: For each academic year provided, the system will create a new registration session record (if one does not already exist for the same academic year, semester, and department). All specified courses will be linked to the session. The session is initially marked as open. (If a session for that year/semester already exists, it will not create duplicates – the implementation checks for existing sessions and would reject or skip duplicates.)

Response: JSON object indicating success or failure. On success:

{ "success": true, "message": "Registration session created successfully" }

- A success: true indicates the session(s) and course offerings were created. The message provides confirmation. On validation failure (e.g. missing fields or closing date in the past), returns **400 Bad Request** with a JSON message, for example:

{ "success": false, "message": "Please fill all fields and ensure the closing date is not in the past." }

Example: An AR officer wants to open registration for **Semester 1** courses for **Year 3** students in the **Computer Science** department, allowing them to register for courses CS301 and CS302, until 2025-09-01.

Request:

POST /Admin/CreateRegistrationSession

Content-Type: application/x-www-form-urlencoded

academicYears=3

&semester=1

&departmentId=3

&closingDate=2025-09-01

&selectedCourses=CS301

&selectedCourses=CS302

Response:

{ "success": true, "message": "Registration session created successfully" }

Admin: List Registration Sessions – GET /Admin/GetRegistrationSessions

Description: Retrieves all existing registration sessions (course offering sessions) created by the AR office. This is used to review sessions, their status, and included courses. **Auth:** Requires AR Officer role.

Response: JSON array of **RegistrationSession** objects. Each session object includes:

- SessionID (int) – Unique identifier of the registration session.
- AcademicYear (string) – Academic year of students that the session is for (as stored, e.g. "3" for third-year).
- Semester (string) – Semester number for the session (stored as string in the system, e.g. "1").
- DepartmentName (string) – Department name if session is department-specific, or "General Program" if it is a general (faculty-wide) session.
- StartDate (string) – The date the session started (typically when it was created).
- EndDate (string) – The closing date of the session (registration deadline), in YYYY-MM-DD format.
- Status (string) – "Open" if the session is currently open for registration, or "Closed" if the session has been closed. (This is derived from an internal boolean IsOpen flag.)
- Courses (array of string) – List of course names offered in this session.

Example response snippet:


```
[
  {
    "SessionID": 5,
    "AcademicYear": "3",
    "Semester": "1",
    "DepartmentName": "Computer Science",
    "StartDate": "2025-08-01",
    "EndDate": "2025-09-01",
    "Status": "Open",
    "Courses": ["Advanced Algorithms", "Database Systems"]
  },
  {
    "SessionID": 6,
    "AcademicYear": "4",
```

"Semester": "1",

"DepartmentName": "General Program",

"StartDate": "2025-08-02",

"EndDate": "2025-09-01",

"Status": "Open",

"Courses": ["Project Management", "Entrepreneurship"]

}

]

This indicates two open sessions: one for Year 3 CS students, and one general session for Year 4 covering general courses.

Admin: Update Registration Session – POST /Admin/UpdateRegistrationSession

Description: Updates an existing registration session's end date or open/closed status. The AR officer can extend or close a session and change its status.
Auth: Requires AR Officer role.

Request: Form data or query parameters (the implementation binds simple types directly from form/query):

- sessionID (int) – The ID of the registration session to update.
- newEndDate (DateTime) – The new closing date for the session.
- newStatus (string) – The new status for the session, either "Open" or "Closed". (Setting to "Closed" will mark the session as closed; "Open" will re-open if it was closed.)

Response: On success, returns a JSON message, for example:

{ "message": "Session updated successfully!" }

If the specified session ID is not found, returns **404 Not Found**:

{ "message": "Session not found." }

Example:

POST /Admin/UpdateRegistrationSession?sessionID=5&newEndDate=2025-09-15&newStatus=Open

This extends session #5's closing date to 15th Sept 2025 and (ensures it's open). Successful response:

{ "message": "Session updated successfully!" }

Admin: Delete Registration Session – POST /Admin/DeleteRegistrationSession

Description: Permanently deletes a registration session and all its associated course offerings. This is typically used if a session was created in error or canceled.

Auth: Requires AR Officer role.

Request: Form/query parameter:

- sessionID (int) – ID of the registration session to delete.

Response: On success (session found and removed), returns:

{ "message": "Session deleted successfully!" }

If the session ID does not exist, returns **404 Not Found** with message "Session not found.".

Note: Deleting a session will also remove any student registration requests associated with that session (if any exist, since the session and its courses are no longer offered). This action is irreversible.

Admin: Export Session Registrations – GET /Admin/ExportSession

Description: Exports the details of a registration session (including student registrations) as a PDF report. This generates a PDF file for offline records or reporting.
Auth: Requires AR Officer role.

Request Parameters:

- sessionID (int, query) – The ID of the registration session to export.

Response: A PDF file download. The content type is application/pdf, and the filename will be in the format Session_{sessionID}.pdf. The PDF contains a list of all students and courses for that session, including any relevant details (e.g., student names, IDs, and the courses they registered for, and possibly their status).

Example: Navigating to or calling /Admin/ExportSession?sessionID=5 will prompt a download of **Session_5.pdf** containing the report for session #5.

Admin: Utility – List Departments – GET /Admin/GetStudentDepartments

Description: Returns a list of department names of students (excluding administrative departments). This is used to populate filters or dropdowns in the UI when managing course offerings.
Auth: Requires AR Officer role.

Response: JSON array of department names (strings). Example:

["Computer Science", "Electrical Engineering", "Mechanical Engineering"]

This list is derived from the departments associated with students in the system and is distinct and sorted. (The "Administration" department is excluded.)

Admin: Utility – List Academic Years – GET /Admin/GetAcademicYears

Description: Returns a list of academic year values of students. Used for filtering by year (e.g., Year 1, Year 2, etc.).

Auth: Requires AR Officer role.

Response: JSON array of academic year identifiers. These are typically integers or strings representing the year of study. Example:

[4, 3, 2, 1]

(The list is distinct and sorted in descending order, meaning Year 4 down to Year 1 in this example.)

Coordinator: List Coordinated Courses – GET /Coordinator/GetCoordinatedCourses

Description: Retrieves the courses (offerings) that the logged-in Course Coordinator is responsible for. Each entry includes course details and enrollment statistics.

Auth: Requires Coordinator role.

Response: JSON array of **CourseOffering** objects for the coordinator. Each object includes:

- CourseCode (string) – Code of the course the coordinator oversees.
- CourseName (string) – Name of that course.
- Semester (string) – Semester in which the course is offered.
- OfferingID (int) – The internal offering ID (if any) for that course instance.
- StudentCount (int) – Number of students currently enrolled (approved registrations) in this course.

For example:


```
[
  {
    "CourseCode": "CS301",
    "CourseName": "Advanced Algorithms",
    "Semester": "1",
    "OfferingID": 27,
    "StudentCount": 45
  },
  {
    "CourseCode": "CS302",
    "CourseName": "Database Systems",
    "Semester": "1",
    "OfferingID": 28,
    "StudentCount": 50
  }
]
```

This indicates the coordinator is responsible for CS301 and CS302 in semester 1, with 45 and 50 students enrolled respectively.

Note: A course will appear in this list if the coordinator's staff ID has been associated with that course's offering (i.e., the coordinator is assigned as the course coordinator/instructor). If a course has no assigned coordinator or no enrollments yet, it may not show up here.

Coordinator: List Students in a Course – GET /Coordinator/GetStudentsByCourse

Description: Retrieves the list of students enrolled in a specific course coordinated by the logged-in coordinator. This allows the coordinator to view details of each student in their class. **Auth:** Requires Coordinator role.

Request Parameter:

- courseCode (string, query) – The course code for which to fetch enrolled students. This must be a course that the coordinator coordinates (otherwise the list will be empty or the request may be unauthorized).

Response: JSON array of **Student** objects enrolled in that course. Each student entry includes:

- StudentID – University ID/registration number of the student.
- StudentName – Full name of the student.
- Department – Department of the student.
- AcademicYear – The academic year of the student (e.g., 3 for third-year).
- (Additional fields like GPA may be omitted here; the coordinator can retrieve detailed results with the next endpoint if needed.)

Example:

GET /Coordinator/GetStudentsByCourse?courseCode=CS301

Response:


```
[
  {
    "StudentID": 20210045,
    "StudentName": "Alice Wong",
    "Department": "Computer Science",
    "AcademicYear": 3
  },
  {
    "StudentID": 20210078,
    "StudentName": "Bob Li",
```

"Department": "Computer Science",

"AcademicYear": 3

}

]

3.4 Student Registration APIs

These endpoints are used by students to view available courses and submit course registration requests. All require authentication as a Student.

Student: Get Available Courses – GET /Student/GetAvailableCourses

Description: Retrieves the list of courses that the logged-in student is eligible to register for in the current open registration session(s). This takes into account the student's academic year, department, completed courses, and any registration sessions that are currently open. **Auth:** Requires Student role.

Request: No parameters (the student is inferred from the session login).

Response: JSON object with two properties:

- courses – an array of **AvailableCourse** objects representing each course the student can enroll in.
- semesters – an array of semester numbers (or identifiers) that are applicable to the available courses list. This helps the front-end filter or group courses by semester. For example, if courses from Semester 1 and 2 are available, this might be [1, 2].

Each **AvailableCourse** in courses includes:

- CourseCode (string) – The course code.
- CourseName (string) – Name of the course.
- Credit (int) – Credit value of the course.
- Semester (string) – Semester number of the course offering (e.g., "1" or "2").
- ClosingDate (string) – The registration closing date for this course's session (format YYYY-MM-DD). After this date, the course can no longer be registered by the student.
- Coordinator (string) – Name of the coordinator or instructor for the course. (If no specific coordinator is assigned yet, this is "Not Assigned".)
- Attempt (int) – The attempt number if the student is retaking the course. 1 means first attempt, 2 means the student has taken (and failed) this course once before (so this is a reattempt), etc. This value is calculated as previous attempts + 1.
- Prerequisites (array of string) – List of course codes that are prerequisites for this course, if any. (If empty, the course has no prerequisites.)
- CourseOfferingID (int) – Internal identifier for the course offering (if an offering record exists). If no specific offering ID is assigned yet, this may be 0.

The logic behind the scenes filters out courses the student **cannot** register for, such as:

- Courses the student has already passed (achieved a passing grade in a previous attempt).
- Courses the student is already registered for (including those pending approval – the system does not strictly exclude pending by code, but typically the UI would not allow duplicate selection).
- Courses for which the student has exhausted the maximum attempt limit (the system enforces a maximum of 3 attempts for a course; a 4th attempt will not be offered if three failures are already recorded).

Example: Student (ID 20210045, Year 3 CS) calls GET /Student/GetAvailableCourses. Suppose there is an open registration session for Year 3 CS students in Semester 1. The student has failed CS301 once before and passed CS201 previously. The response might be:


```
{
  "courses": [
    {
      "CourseCode": "CS301",
      "CourseName": "Advanced Algorithms",
      "Credit": 3,
      "Semester": "1",
      "ClosingDate": "2025-09-01",
      "Coordinator": "Not Assigned",
      "Attempt": 2,
      "Prerequisites": ["CS201"],
      "CourseOfferingID": 0
    },
    {
      "CourseCode": "CS302",
      "CourseName": "Database Systems",
      "Credit": 4,
      "Semester": "1",
      "ClosingDate": "2025-09-01",
      "Coordinator": "Dr. Smith",
      "Attempt": 1,
```

```
    "Prerequisites": [],
    "CourseOfferingID": 0
  }
 ],
 "semesters": [1]
}
```


In this example:

- The student can take **CS301** again (Attempt 2, since they failed it before) even though it has a prerequisite CS201, because the student has passed CS201 already.
- The student can take **CS302** for the first time.
- Both are semester 1 courses and share the same closing date for registration.

Student: Submit Course Registration – POST /Student/SubmitCourseRegistration

Description: Submits the student's selected courses for enrollment in the current open session(s). This creates one or more pending registration requests that must be approved by the student's advisor and then by the AR office.

Auth: Requires Student role.#

Request Body: JSON array of course selections. Each element is a JSON object with:

- CourseCode (string) – The code of the course the student wants to register for.
- CourseOfferingID (int) – The offering ID of the course (if known/applicable). The front-end typically passes this as received from the GetAvailableCourses response. If no specific offering ID is provided, the backend will determine the correct session and offering for the course.

Example request:


```
[
 { "CourseCode": "CS301", "CourseOfferingID": 0 },
 { "CourseCode": "CS302", "CourseOfferingID": 0 }
]
```


Behavior: The server will identify the open registration session corresponding to each course (based on the student's year/department and course's semester). For each course, a **PendingRegistration** record is created linking the student and the course under the appropriate session. These pending records indicate the student's intent to enroll, awaiting approvals. The system ensures that duplicates are not created (if the student already has a pending or approved registration for the course, it won't add another). Prerequisites are **not** automatically enforced by the API at submission (it is assumed the available courses list already filtered out ineligible courses).

Response: On success, returns HTTP 200 with a confirmation message:

{ "message": "Course registration submitted successfully!" }

At this point, the student's request is recorded and will be visible to their advisor for approval. If the student had no open registration session or input was invalid, an error is returned:

- **400 Bad Request** if the request payload is empty or malformed. For example:
- { "message": "No courses selected." }
- **401 Unauthorized** if the student is not properly authenticated (session expired or not logged in).

Example: As in the previous example, the student submits CS301 and CS302:

POST /Student/SubmitCourseRegistration

Content-Type: application/json

[

{ "CourseCode": "CS301", "CourseOfferingID": 0 },

{ "CourseCode": "CS302", "CourseOfferingID": 0 }

]

Response:

{ "message": "Course registration submitted successfully!" }

After this call, the student's selected courses are pending advisor approval. The student can typically see these pending courses in their portal (marked as pending). (There isn't a separate API endpoint in this codebase to fetch the student's pending list, but the UI would reflect the submission.)

3.5 User and Academic Records APIs

These endpoints allow retrieval of user information and academic records, such as lists of students and their results. Different roles have access to different scopes of data (e.g., AR can see all students, advisors can see their advisees, coordinators can see students in their courses).

Admin: Get All Students & Advisors – GET /Admin/GetAdvisorStudentList

Description: Retrieves a master list of students along with their assigned advisor and other details. This provides the AR office a quick view of which advisor is responsible for which student, and basic academic info.

Auth: Requires AR Officer role.
Response: JSON array of student entries. Each entry includes:

- regNo (string) – The student's registration number or ID.
- studentName (string) – Full name of the student.

- advisorName (string) – Full name of the student's advisor.
- department (string) – Department name of the student.
- year (int or string) – The student's current academic year.

Example:


```
[
 {
  "regNo": "2021/E/045",
  "studentName": "Alice Wong",
  "advisorName": "Dr. Robert Brown",
  "department": "Computer Science",
  "year": 3
 },
 {
  "regNo": "2021/E/078",
  "studentName": "Bob Li",
  "advisorName": "Dr. Robert Brown",
  "department": "Computer Science",
  "year": 3
 },
 {
  "regNo": "2021/E/112",
  "studentName": "Charlie Kumar",
  "advisorName": "Prof. Jane Doe",
  "department": "Electrical Engineering",
  "year": 2
 }
]
```


In this example, Alice and Bob have the same advisor, etc. This list is comprehensive across departments. The AR officer can use this for oversight of advisor assignments and to lookup students quickly.

Advisor: Get Student Academic Record – GET /Advisor/GetStudentResults

Description: Allows a student advisor to retrieve the academic results (transcript) of one of their advisees (students under their supervision). This helps advisors review academic performance, including past course grades and GPA.

Auth: Requires Advisor role.

Request Parameter:

- studentId (int, query) – The Student ID (registration number) of the advisee whose results are requested. The advisor must be assigned to this student, otherwise the data may be incomplete or the request unauthorized.

Response: JSON array of **ResultRecord** objects, each representing the outcome of a course the student has taken in the past. Each record includes:

- CourseCode (string) – Code of the course taken.
- CourseName (string) – Name of the course.
- Semester (string) – Semester in which the course was taken.
- AcademicYear (string) – The academic year (session) in which the course was taken (if available).
- Grade (string) – The grade achieved (e.g., letter grade or pass/fail notation).
- Average (number) – The numeric score or percentage the student earned in the course.

Example:

GET /Advisor/GetStudentResults?studentId=20210045

Response:


```
[
 {
  "CourseCode": "CS201",
  "CourseName": "Data Structures",
  "Semester": "1",
  "AcademicYear": "2023/2024",
  "Grade": "B+",
  "Average": 78.5
 },
 {
  "CourseCode": "CS202",
  "CourseName": "Algorithms",
```

```
  "Semester": "2",
  "AcademicYear": "2023/2024",
  "Grade": "A",
  "Average": 85.0
 },
 {
  "CourseCode": "CS301",
  "CourseName": "Advanced Algorithms",
  "Semester": "1",
  "AcademicYear": "2024/2025",
  "Grade": "F",
  "Average": 45.0
 }
]
```


This shows the advisee's performance in several courses. (CS301 was failed in the current year, presumably prompting a reattempt.) The advisor can calculate or view the student's GPA from this data. (The API does not directly compute GPA, but the data needed for it – grades/credits – is provided. Typically, the front-end or advisor would compute current GPA as needed.)

Coordinator: Get Student Academic Record – GET /Coordinator/GetStudentResults

Description: Allows a course coordinator to view the academic record of a student (usually one enrolled in their course). This is similar to the advisor's view, but a coordinator might use it to understand a student's background or performance in prerequisite courses, etc. **Auth:** Requires Coordinator role.

Request Parameter:

- studentId (int, query) – The ID of the student whose results are requested. The coordinator is expected to use this for students in their courses (the system does not strictly enforce that the student is enrolled in one of the coordinator's courses for this query, but that is the intended use-case).

Response: JSON array of **ResultRecord** objects, identical format to the advisor's GetStudentResults response (course code, name, semester, academic year, grade, average). It will list all courses the student has taken with their grades.

Coordinators can use this to, for example, check the prior performance or GPA of students in their class. (For brevity, the example is similar to the advisor's response above and is omitted here.)

Security Note: While the API does not explicitly restrict which student a coordinator can query, it is implied that this should be used for students under their instruction. In practice, the UI would likely only expose this function for relevant students.

3.6 Notification & Approval APIs

These endpoints handle the **approval workflow** for course registrations – essentially the "notifications" for advisors and admins that there are pending registration requests, and the actions to approve or reject those requests. Advisors receive notifications of new course requests from their advisees, and AR officers see pending requests that require final approval. The APIs allow retrieval of pending requests and submission of approval decisions.

Advisor: Get Pending Registrations – GET /Advisor/GetPendingRegistrations

Description: Retrieves the list of pending course registration requests for a specific student advisee that the advisor has not yet approved or rejected. This allows an advisor to review what courses a student is trying to register for.

Auth: Requires Advisor role.

Request Parameter:

- studentId (int, query) – The ID of the advisee (student) whose pending requests to fetch. The advisor should be assigned to this student.

Response: JSON array of pending course requests (each a simplified **PendingRegistration** record) for that student that are awaiting the advisor's decision. Each entry includes:

- CourseCode (string) – The course the student wants to enroll in.
- CourseName (string) – Name of the course.
- Credits (int) – Credit value of the course.
- Semester (string) – Semester of the course offering (e.g., "Semester 1").

These are the key details an advisor needs to decide approval. The advisor already knows the student in question (from the context of providing the studentId).

Example:

GET /Advisor/GetPendingRegistrations?studentId=20210045

Response:


```
[
  {
    "CourseCode": "CS301",
    "CourseName": "Advanced Algorithms",
    "Credits": 3,
    "Semester": "Semester 1"
```

```
  },
  {
    "CourseCode": "CS302",
    "CourseName": "Database Systems",
    "Credits": 4,
    "Semester": "Semester 1"
  }
]
```


This indicates the student 20210045 has requested CS301 and CS302 and those are pending advisor approval.

Advisor: Approve/Reject Course Requests – POST /Advisor/SubmitApprovals

Description: Submits the advisor's decisions (approve or reject) for one or more pending course registration requests for a student. An advisor can approve or deny each requested course for their advisee. Typically, this might be done in bulk via the UI (selecting approve/reject for each course and submitting once).

Auth: Requires Advisor role.

Request Body: JSON object with the following structure:

- studentId (string or int) – The ID of the student whose requests are being decided.
- courses (array) – A list of course decisions. Each element is an object with:
    - courseCode (string) – The course code of the request.
    - status (string) – The advisor's decision for that course, either "Approved" or "Rejected".

Example request:


```
{
  "studentId": "20210045",
  "courses": [
    { "courseCode": "CS301", "status": "Approved" },
    { "courseCode": "CS302", "status": "Rejected" }
  ]
}
```


Behavior: For each course listed, the system finds the corresponding pending registration record for that student and course. If found, it updates the record:

- If status is "Approved": marks the pending registration as advisor-approved (IsApprovedByAdvisor set to yes, internally). The Status may remain "Pending" (meaning pending final AR approval) but it is now flagged that advisor has approved.
- If status is "Rejected": marks the Status as "Rejected" and effectively halts the request (the AR officer will see it as rejected by advisor and typically would not approve it).

All decisions are processed in one transaction.

Response: A JSON object confirming the operation. On success:

{ "message": "Approvals recorded successfully." }

(If the implementation does not return a custom message, it may simply return 200 OK with no detailed body; but assume a confirmation message as above.)

Errors: If any pending record is not found (for example, an invalid course code or if already decided), the response might be **404** or **400** with an error message. If the advisor is not actually the advisor for that student, the operation will effectively find no records and may return success with no changes or an error (this scenario is prevented by design in the UI).

After this call, any courses the advisor approved will move on for AR final approval, and any rejected will be marked rejected (the student could potentially see the rejection, and the AR will know the advisor rejected it). The advisor will no longer see these in their pending list on subsequent calls (they're either approved or removed).

Admin: Get All Pending Registrations – GET /Admin/GetPendingRegistrations

Description: Retrieves **all** pending course registration requests that require AR officer review. This includes requests that are awaiting advisor action as well as those already advisor-approved but waiting for final AR approval. The AR officer can see the full list of students and courses that are in the pipeline.

Auth: Requires AR Officer role.

Response: JSON array of **PendingRegistration** records. Each object includes comprehensive information about each pending request:

- PendingID (int) – The unique ID of the pending registration record.
- StudentID (int) – The student's ID.
- StudentName (string) – Full name of the student.
- CourseCode (string) – Course code requested.
- CourseName (string) – Course name.
- Semester (string) – Semester of the course offering.
- ClosingDate (string) – The closing date of the registration session for this course (format YYYY-MM-DD).

- Status (string) – The current status of the request (typically "Pending" for those awaiting AR decision; if an advisor rejected it, this might show "Rejected" – AR can still see it was rejected by advisor).
- RegistrationDate (string) – The date the student submitted the registration request (format YYYY-MM-DD).
- Attempt (int) – The attempt number for this course (as explained earlier, 1 for first attempt, 2 for second attempt, etc. If the student failed before, this would be >1 indicating a reattempt).
- IsApprovedByAdvisor (string or boolean) – Indicates whether the student's advisor has approved the request. For example, "Yes" or "No". If "No", it means the request is still waiting for advisor action or was rejected by the advisor. "Yes" means the advisor has approved and it's pending final AR review.

Example (showing one pending record awaiting AR decision and one that an advisor rejected):


```
[
 {
  "PendingID": 12,
  "StudentID": 20210045,
  "StudentName": "Alice Wong",
  "CourseCode": "CS301",
  "CourseName": "Advanced Algorithms",
  "Semester": "1",
  "ClosingDate": "2025-09-01",
  "Status": "Pending",
  "RegistrationDate": "2025-08-20",
  "Attempt": 2,
  "IsApprovedByAdvisor": "Yes"
 },
 {
  "PendingID": 13,
  "StudentID": 20210078,
  "StudentName": "Bob Li",
  "CourseCode": "CS302",
  "CourseName": "Database Systems",
  "Semester": "1",
```

"ClosingDate": "2025-09-01",

"Status": "Rejected",

"RegistrationDate": "2025-08-22",

"Attempt": 1,

"IsApprovedByAdvisor": "No"

}

]

In this example, Alice's request for CS301 was approved by her advisor and is pending AR's decision, while Bob's request for CS302 was rejected by his advisor (so AR likely will not approve it – it's effectively terminated). The AR officer can review all such entries.

Admin: Approve/Reject Pending Registration – POST /Admin/UpdatePendingStatus

Description: Submits the AR officer's decision on a pending registration request. The AR can approve a request (finalizing the enrollment) or reject it.

Auth: Requires AR Officer role.

Request Body: JSON object (sent in the request body) with:

- PendingID (int) – The ID of the pending registration record (as listed in GetPendingRegistrations).
- Status (string) – The decision, either "Approved" or "Rejected".

Example:

{ "PendingID": 12, "Status": "Approved" }

Behavior: The system will locate the pending registration by ID. If not found, returns 404. If found:

- If the AR **approves**:
    - The pending record's Status is updated to "Approved" (and an ApprovalDate may be recorded internally).
    - A new **Registration** record is created in the system to officially enroll the student in the course. (This moves the student from "pending" to "enrolled".) Specifically, the code will create an entry in the Registrations table linking the student to an official course offering. If a corresponding **CourseOffering** (course instance) does not exist yet for that course in that semester, it will create one at this time and link it. The new registration record will contain the student ID, the course offering ID, and attempt number, etc.
    - The pending request is now resolved. (It may remain in the database with status updated, but it will no longer appear in the "pending" list since it's not pending anymore.)
- If the AR **rejects**:

- The pending record's Status is updated to "Rejected" (and ApprovalDate set). No new enrollment is created. This effectively cancels the request.
- (If the advisor had approved it, they will see later that it was rejected by AR; the student will also see the rejection.)

Response: On success, a confirmation message:

{ "message": "Registration request processed successfully." }

(The code returns "Session updated successfully!" for session updates; for pending updates, it returns a generic message or just 200. We can assume a message as above for documentation clarity.) If the PendingID is invalid, it returns **404 Not Found** ({"message": "Session not found."} – note: the wording might be a minor oversight in the implementation using "Session" instead of "Request" in the message).

Example:

POST /Admin/UpdatePendingStatus

Content-Type: application/json

{ "PendingID": 12, "Status": "Approved" }

Response:

{ "message": "Registration request processed successfully." }

After approving, the student with PendingID 12 (Alice Wong for CS301 in the earlier example) is now officially enrolled in CS301. A new course offering (if not already present) and registration record are created with an assigned OfferingID, etc. The AR officer would no longer see this entry in pending requests. If the AR had rejected it instead, the student's request would be marked rejected (and the student and advisor would know it was denied).

Coordinator: (Optional) Pending Approvals – GET /Coordinator/GetPendingRegistrations and POST /Coordinator/SubmitApprovals

Note: In the current system, course coordinators generally are **not** in the approval chain for student registrations (only advisors and AR are). These endpoints exist in the code but mirror the advisor's functionality, possibly intended for a scenario where coordinators also approve course enrollments. By default, they may not be actively used. For completeness:

- GET /Coordinator/GetPendingRegistrations?studentId=X would list pending requests for a given student (similar to the advisor's view).
- POST /Coordinator/SubmitApprovals would accept a JSON of decisions similar to the advisor's SubmitApprovals format.

These require Coordinator role. In a standard workflow, coordinators do not need to approve registrations, so these would typically return an empty list or not be called. The presence of these

endpoints suggests they were implemented for consistency or future use. In typical usage, you can ignore these for the registration approval process (advisors and AR handle approvals).

3.7 API Usage Notes

- **Authentication Required:** All endpoints (except the Auth API login) require a valid authenticated session. Ensure the user logs in via the form or Auth API, and include the session cookie in subsequent requests. If not authenticated, protected endpoints will not return data (they will redirect or send an unauthorized response). There is no separate token-based auth in this API; it relies on cookies.
- **Role-Based Access:** Endpoints are strictly divided by user roles. A user will only have access to the endpoints for their role. For example, a Student cannot call /Admin/GetRegistrationSessions (they would be forbidden), and an Advisor cannot call student endpoints, etc. Attempting to do so will result in a 403 Forbidden or a redirect to login. Always use credentials for the appropriate role when testing endpoints.
- **Data Formats:**
    - **Dates:** Dates in responses are typically formatted as YYYY-MM-DD strings. Date-time inputs (e.g., closingDate for sessions) should be provided in a standard ISO format.
    - **Strings vs Numbers:** Some fields that are numeric in concept are treated as strings in the database and API (e.g., Semester and AcademicYear often appear as strings in responses). The client should be prepared to handle either string or number for such fields. In practice, you can treat "1" and 1 as equivalent for semester, etc.
    - **Request Body Formats:** Pay attention to whether an endpoint expects JSON or form data:
        - JSON: For Auth API login, student course submission, advisor approvals, etc. (Content-Type: application/json).
        - Form Data: For admin session creation/update endpoints (they use [FromForm] binding). In Postman or clients, send those as form fields or as query parameters.
    - **File Download:** The ExportSession endpoint returns a binary PDF file; it will not be JSON. The client should handle it as a file download (the response includes a Content-Disposition with filename).
- **Error Handling:** The API returns error messages in a consistent JSON structure with a message field describing the error. For example:
    - 400 Bad Request for missing/invalid input (e.g., login without password, or session creation with invalid date) – the JSON will contain a message explaining the issue.
    - 404 Not Found if an entity ID is not found (e.g., updating a session or pending request that doesn't exist).
    - 500 Internal Server Error if something unexpected goes wrong on the server. In such cases, a generic error message or stack trace might be returned (depending on

development mode). The system prints diagnostic messages to the console (for developers) for debugging (e.g., logging course fetching, registration creation, etc.), but those are not exposed via the API.

- **Business Rules & Validation:**
  - **Max Course Attempts:** The system enforces a maximum of 3 attempts for any course. This means if a student has failed a course three times (recorded in results), the course will no longer appear in their available list, and they cannot register for it again via the API. This is checked when listing available courses.
  - **Prerequisites:** Prerequisite relationships are included in the data (Prerequisites field for courses), but the API does not prevent a student from registering a course if they haven't met the prereqs – it assumes the available courses query already filters out courses where prerequisites aren't satisfied (or at least the student/advisor will notice and handle it). The AR officer should not offer courses to students who haven't completed prerequisites, but currently the system does not automatically cross-check a student's transcript for prerequisites at submission time. It relies on advisors to catch such issues (advisors can see the student's results and the course's prerequisites).
  - **Concurrent Sessions:** It's possible to have multiple registration sessions open concurrently (e.g., a general session and a department-specific session). The GetAvailableCourses endpoint will aggregate courses from all open sessions relevant to the student. The semesters array in its response helps the front-end separate courses by semester if needed.
  - **Duplicate Requests:** The system avoids creating duplicate pending registrations. If a student submits a course that is already pending or already approved, the Create registration API will ignore or prevent duplicate entries. Also, if a session already exists for a given year/semester/department, the admin cannot create another identical one (the API checks for existing session and will not duplicate it).
  - **Data Consistency:** When the AR approves a registration, the system ensures a corresponding Course Offering and Registration exist. The CourseOffering ID is used in the Results table for storing grades. The design is such that each unique course/semester offering gets an OfferingID (with potentially multiple students). The code, upon final approval, finds or creates the CourseOffering if it wasn't created at session setup, and then adds a Registration linking the student to that offering. This ensures that later, when results are recorded for that offering, each registered student is accounted for.
- **Notifications to Users:** The system currently does not send out email or push notifications on approvals/rejections. "Notifications" are provided via the UI: students and advisors would have to check the system to see the status of requests. For example, after AR's decision, the student can see in their course list whether the course was approved or rejected. An extension could be to send an email to the student upon approval or rejection, but that is not implemented in the given code.
- **Example Workflow:** To illustrate usage, consider a full cycle:

AR creates a session: Use CreateRegistrationSession to open Semester X for Year Y students with certain courses.

Student fetches courses: The student calls GetAvailableCourses and gets the list of courses open for them.

Student submits registration: The student selects courses and calls SubmitCourseRegistration. The system returns success.

Advisor notified: The advisor (seeing their dashboard or via GetPendingRegistrations for that student) now has pending requests. They retrieve and see the courses, then use SubmitApprovals to approve or reject them.

AR final approval: The AR checks GetPendingRegistrations (now the advisor-approved ones are marked accordingly) and calls UpdatePendingStatus for each to approve. Once approved, the student is officially enrolled; if rejected, the process ends for that request.

Student outcome: The student can now see the course in their registered courses (if approved) or see that it was not approved. (The actual API for the student to list current registrations isn't explicitly defined, but the UI would show it, possibly using the same data now moved to Registrations/Results.)

- **API Idempotence:** Some endpoints like CreateRegistrationSession and SubmitCourseRegistration are designed to be called once per intended action. If called twice with the same data, they may create duplicates or return an error if duplicates are detected. For instance, creating a session twice for the same semester/year will likely return a success message both times, but the second time it may just find the existing session and not create a new one (or it might inadvertently create a duplicate if the check misses something like different course lists). It's best to avoid duplicate calls and check existing sessions with GetRegistrationSessions if unsure.

3.8 API Data Models

This section summarizes the key data models and JSON structures used in the API. These correspond to the objects returned in responses or expected in requests. All fields listed are as implemented in the codebase (no extra fields beyond these are present).

- **Course**: Represents a course offered by the faculty.
    - CourseCode (string): The unique code of the course (e.g., "CS101").
    - CourseName (string): The name of the course.
    - Credit (int): Credit value of the course.
    - DepartmentName (string): Name of the department offering the course.
- **RegistrationSession**: Represents a registration session (an offering period for certain students to register for certain courses).
    - SessionID (int): Unique identifier of the session.

- AcademicYear (string): The academic year of students eligible (e.g., "3" for third-year students).
- Semester (string): Semester number for the session (e.g., "1" or "2").
- DepartmentName (string): Department name if session is department-specific, or "General Program" if it's a general session for all departments.
- StartDate (string): Date the session started (YYYY-MM-DD).
- EndDate (string): Date the session closes (YYYY-MM-DD).
- IsOpen (bool): **(Internal)** Indicates if the session is open. (In API responses, this is conveyed via the Status field as "Open"/"Closed").

- **PendingRegistration**: Represents a student's course registration request that is pending approval.
  - PendingID (int): Unique ID of the pending request.
  - StudentID (int): ID of the student who made the request.
  - StudentName (string): Full name of the student.
  - CourseCode (string): Course code requested.
  - CourseName (string): Course name.
  - Semester (string): Semester of the course offering.
  - ClosingDate (string): Registration closing date for the session (YYYY-MM-DD).
  - RegistrationDate (string): Date the student submitted the request (YYYY-MM-DD).
  - Status (string): Current status of the request. Expected values: "Pending" (awaiting approval), "Approved" (approved by AR – final), "Rejected" (rejected by an advisor or AR).
  - Attempt (int): The attempt number for the student taking this course (1 = first time, 2 = second attempt, etc.).
  - IsApprovedByAdvisor (string/bool): Advisor's decision state. "No" (or false) if not yet approved by advisor, "Yes" (or true) if advisor has approved. (If advisor rejected, Status will be "Rejected" and this may still be "No" since advisor didn't approve.)
- **Registration**: Represents a finalized enrollment of a student in a course offering (after all approvals). *(This model is mostly internal; it's not directly exposed via an endpoint in this API, but worth noting)*
  - RegistrationID (int): Unique ID of the registration record.
  - StudentID (int): Student who is enrolled.
  - OfferingID (int): The CourseOffering ID that the student is enrolled in.

- (Other fields may include enrollment date, etc., but they are not exposed via API directly.)

- **CourseOffering**: Represents an instance of a course being offered in a specific academic term (ties together Course, semester, academic year, and instructor).
  - OfferingID (int): Unique ID of the course offering.
  - CourseCode (string): Course code of the offering.
  - AcademicID (int): Reference to an academic term (year/semester) – links to AcademicSchedule, which defines the actual calendar year and semester.
  - Semester (string): Semester number of the offering (often duplicative of AcademicSchedule's semester, but stored here too).
  - StartDate (DateTime): Start date of the course offering (term start).
  - EndDate (DateTime): End date of the course offering (term end).
  - StaffID (int): The staff member coordinating/teaching the course (if assigned).
  - DepartmentID (int): Department offering the course.

In the API context, CourseOffering data appears in coordinator endpoints: for example, GetCoordinatedCourses returns some fields of CourseOffering (CourseCode, CourseName, Semester, OfferingID, StudentCount). Not all fields are exposed, only those relevant to the coordinator.

- **AvailableCourse**: (Composite model used in GetAvailableCourses for students) – Represents a course available for a student to register, including context like attempt and prerequisites.
  - CourseCode (string)
  - CourseName (string)
  - Credit (int)
  - Semester (string)
  - ClosingDate (string, date)
  - Coordinator (string)
  - Attempt (int)
  - Prerequisites (array of string)
  - CourseOfferingID (int)
    (See section 3.4 for detailed explanation of each field.)

- **ResultRecord**: Represents a student's result in a course (used in advisor's and coordinator's GetStudentResults).
  - CourseCode (string)

- CourseName (string)
- Semester (string)
- AcademicYear (string) – academic year or session in which the course was taken
- Grade (string) – grade or outcome (e.g., letter grade)
- Average (number) – numeric score/percentage obtained

- **LoginRequest (LoginViewModel)**: Credentials for logging in.
  - Email (string) – User's email/username.
  - Password (string) – User's password.

    (This is sent as JSON to the Auth API login.)
- **LoginResponse**: Information returned upon successful login.
  - message (string) – Success message.
  - role (string) – User's role ("Student", "Advisor", "Coordinator", or "AR").
  - userId (int) – User's ID (StudentID for students, StaffID for staff).
  - fullName (string) – User's full name.
  - academicYear (int, students only) – Student's academic year (for staff, this field may not be present).

    (If login fails, an error message is returned instead.)
- **CourseSelectionDto**: Course selection item used in student's registration submission.
  - CourseCode (string) – The course code the student wants to register.
  - CourseOfferingID (int) – The offering ID (if any).
    (This is part of the array in SubmitCourseRegistration request.)
- **ApprovalSubmission**: Structure for advisor/coordinator approvals submission.
  - studentId (int) – ID of the student whose requests are being approved/rejected.
  - courses (array of { courseCode: string, status: string }) – Decisions.
    (Used in Advisor/SubmitApprovals and similarly in coordinator's, if used.)
- **UpdatePendingStatusRequest**: Structure for admin's final approval action.
  - PendingID (int) – ID of the pending registration to act on.
  - Status (string) – "Approved" or "Rejected".
    (Used in Admin/UpdatePendingStatus request body.)

Each of these data models corresponds to concrete classes or anonymous types in the code. The response payloads in this API are flat JSON, meaning complex object relations are usually broken down into simple fields (e.g., DepartmentName is given instead of a nested Department object). This makes it easier for the front-end (JavaScript or other clients) to directly use the data.

Conclusion: This API documentation covers all endpoints of the Course Registration System's ASP.NET Core MVC backend. The endpoints are designed for internal use in the faculty's web application (with potential extension to external clients via the Auth API). All examples above align with the actual implementation logic and data structures in the system, ensuring that developers working with this API can rely on this documentation for accurate integration.

Course Registration System API Endpoints

Below is a comprehensive list of API endpoints for the Course Registration System backend, organized by category. Each entry includes the endpoint path, HTTP method, a brief description, required user role, any request parameters or body, and a summary of the response structure. These details are directly based on the project's controller/route code and data models.

Auth APIs

This category covers authentication-related endpoints, primarily for logging in users (students or staff).

End point	**Method**	**Description**	**Required Role**	**Request Parameters**	**Request Body**	**Response**
/Auth/ Login	POST	Authenticate a user (student or staff) with credentials and start a session (issue auth token). Checks the provided email/ID against the Student or Staff records and verifies the password. On success, returns user details and role.	*None* (Public)	*(none)*	JSON with **Email** and **Password** (login credentials)	JSON object with user info (e.g. UserID, Name, Role) and an authentication token or session identifier.
/Auth/ Logout	POST	Log out the current user, ending their session or invalidating their token. (May be implemented if using session cookies; not needed for stateless JWT except for client-side disposal.)	*Authenticated* (Any role)	*(none)*	*(none)*	Success status (e.g. HTTP 200 OK) indicating the session was terminated.

Course and Offering APIs

These endpoints allow the Academic Registry (Admin) to manage courses and registration sessions (semesters) and assign course offerings to sessions. They also provide data about courses and departments needed for the registration process.

End point	Method	Description	Required Role	Request Parameters	Request Body	Response
/Admin/GetCourses	GET	Retrieve the master list of all courses available in the system, including details like course code, name, credit value, semester offered, etc. Used by Admin when assigning courses to a session.	Admin	*(none)*	*(none)*	Array of course objects (each with fields such as **CourseCode**, **CourseName**, **Credit**, **Semester**, etc.).
/Admin/GetDepartments	GET	Retrieve the list of all academic departments. Typically used to filter courses or designate department-specific sessions.	Admin	*(none)*	*(none)*	Array of department objects (e.g. **DepartmentID**, **DepartmentName**)
/Admin/GetCoordinators	GET	Get a list of staff who can serve as course coordinators. Optionally may filter by department. Used when assigning a coordinator to a course offering.	Admin	Department Id (optional, query) to filter coordinators by department.	*(none)*	Array of staff entries (each with **StaffID**, name, etc.) for coordinators (Role = "Coordinator").
/Admin/CreateRegistrationSession	POST	Create a new course registration session (semester term). Allows Admin to specify the academic year, semester, whether it's general or department-specific, the department (if applicable), and the registration start/end dates. Sets the session as open or closed for registration.	Admin	*(none)*	JSON with session details: **AcademicYear**, **Semester**, **IsGeneralProgram** (bool), **DepartmentID** (if not general), **StartDate**, **EndDate**.	Session object just created (with assigned **SessionID** and the provided fields), or a success message.

/Admin/AddCourseOffering	POST	Assign a course to a registration session (create a course offering for that term). This endpoint links a course to an active session and optionally assigns a coordinator (instructor) to that offering. It creates a new Course Offering recordfile	Admin	*(none)*	JSON with offering details: **SessionID** (the target registration session), **CourseCode** (which course to offer), and **StaffID** (coordinator for the course).	The created offering record, including **OfferingID** and the provided details (course, session, coordinator).
/Admin/GetCourseOfferings	GET	Retrieve the list of courses offered in a specific registration session. Allows Admin to review which courses are assigned to that term (and who the coordinator is for each).	Admin	*sessionId (query or path) to specify the session of interest.*	(none)	Array of offered course objects for the session. Each includes OfferingID, CourseCode (and course name), session info, and assigned StaffID (coordinator) if any.

Note: A *Course Offering* represents a course being offered in a particular semester session. The CourseOfferings table includes an OfferingID, CourseCode, reference to the session (AcademicID/SessionID), and the StaffID of the coordinator assigned. Creating an offering uses these fields. The system also tracks departments for courses and sessions, which is why Department data is available.

Student Registration APIs

These endpoints are used by students to view available courses for the current registration period, submit course registrations, and review their registrations. They ensure that students can only perform actions on their own record.

End point	Method	Description	Required Role	Request Parameters	Request Body	Response
Student/ Get Available Courses	GET	Retrieve the list of courses available for the student to enroll in for the current open registration session. The system determines the relevant session (e.g. based on current date and student's	Student	*(none)* (context is current open session for that student's	*(none)*	Array of offered course objects that the student can register for. Each includes course details (code, name, credit, etc.) and offering/session

		department) and returns courses offered in that session that the student has not already registered for. This respects any prerequisites (only courses for which the student meets prerequisites will be shown).		program)		info. Prerequisite requirements are implicitly considered (courses with unmet prereqs would not appear).
/Student/ Register Course	POST	Submit a course registration request for the current semester. Creates a new **pending registration** record for the student. The request will await approval from the student's advisor (and subsequently the AR officer). The student is typically registering for a specific course offering.	Student	*(none)*	JSON with registration info: **OfferingID** (the course offering the student wants to enroll in). Optionally includes attempt number if it's a repeat attempt (otherwise attempt is determined by the system).	A pending registration object confirming the request. This will include a **PendingID** and echo details like course, session, **StudentID**, attempt number, timestamp, etc., along with an initial **Status** (e.g. "Pending Advisor Approval").
/Student/ Get My Registratio ns	GET	View the student's current registration requests for the active semester. This returns all **pending registrations** the student has submitted that are in progress (waiting for approval or recently approved). Each entry includes the course and its approval status.	Student	*(none)*	*(none)*	Array of the student's pending registration records. Fields include **CourseCode** (and name), **Status** (e.g. pending, approved, or rejected), and flags for advisor approval and final approval. If a request has been advisor-approved or fully approved, the status reflects that.

/Student/ Get Registration History	GET	Retrieve the student's course registration history from past semesters. This serves as the student's academic record of courses taken (including reattempts). Each record typically includes the term, course, attempt number, and grade earned.	Student	*(none)*	*(none)*	Array of past registration records for the student. Each entry includes **AcademicYear/Semester** (term), **CourseCode** (and name), **Attempt** number, and final **Grade** or outcome for that course. (Data is drawn from finalized registrations and results tables.)
/Student/ Get GPA	GET	Calculate and return the student's current Grade Point Average. The GPA is computed based on the grades of all courses the student has completed (using the Results of each course and their credit values).	Student	*(none)*	*(none)*	JSON object with the student's **GPA** (e.g. a numeric value to two decimal points). This is derived from the student's results (grades) and course credits.

Notes: When a student registers for a course, a record is created in a PendingRegistrations table with the student, course, session, and an initial status. It also tracks if the advisor has approved it (IsApprovedByAdvisor) and other info like attempt number for repeat courses. Once fully approved, it moves to a final Registrations record. The student's **registration history** is essentially all their finalized registrations with associated results. The **GPA** is computed from the **Results** table, which stores each course's grade (and components like in-class and final marks) combined with the course credit values.

Academic Records APIs

These endpoints allow retrieval of academic records and performance data. Students can view their academic results, advisors can view their advisees' academic details, and coordinators can view performance of students in the courses they coordinate.

End point	Method	Description	Required Role	Request Parameters	Request Body	Response
/Student/ Get Registration History	GET	*(Same as above in Student Registration APIs – listed here for completeness)* View the student's past course registrations and results. This is effectively the student's	Student	*(none)*	*(none)*	List of past courses with term, course name, and grade/outcome. (Only courses the

		transcript of courses taken, including grades.				student completed are included, typically those in the **Registrations** and **Results** tables.)
/Student/ GetGPA	GET	*(Same as above)* Retrieve the student's cumulative GPA up to the current date.	Student	*(none)*	*(none)*	Current GPA value (computed from all completed courses' grades and credits).
/Advisor/ GetAdvisees	GET	Get the list of students (advisees) assigned to the logged-in advisor. Each student in the list is one whose record has the advisor's StaffID as their advisor. This allows the advisor to select a specific advisee for further detail.	Advisor	*(none)*	*(none)*	Array of student summary objects (each with **StudentID**, name, department, academic year, etc.). These are the advisor's advisees.
/Advisor/ GetAdviseeProfile/{studentId}	GET	Retrieve personal and academic profile details for a specific advisee (student). The advisor can see the student's personal info (e.g. full name, contact info, addresses, photo) and academic info (department, current year, etc.).	Advisor	**studentId** (in URL/path)	*(none)*	A detailed student profile object containing fields such as **FullName**, **Gender**, **Contact info** (phone, addresses), **DepartmentID & name**, **AcademicYear**, etc..
/Advisor/ GetAdviseeResults/{studentId}	GET	Retrieve the academic results (grades) of a specific advisee. This returns the list of courses the student has completed along with the grade for each (and possibly the term taken). It allows an advisor to review the academic performance of their advisee.	Advisor	**studentId** (in URL or query)	*(none)*	Array of course results for that student. Each entry includes **CourseCode** (and name), the term taken, and the **Grade** achieved. (Derived from the Results records for the student.)

/Advisor/GetAdviseeGPA/{studentId}	GET	Get the current GPA of a specific advisee. This calculates the student's GPA similarly to how a student would see their own, but accessible to the advisor for monitoring purposes.	Advisor	**studentId** (in URL/query)	*(none)*	GPA value for the student (e.g. a decimal representing their cumulative grade point average).
/Coordinator/GetMyCourses	GET	Retrieve the list of course offerings that the coordinator (logged-in staff) is responsible for. These are typically the courses for which the staff member is designated as coordinator in the current or recent sessions. (This queries course offerings where StaffID equals the coordinator's ID.)	Coordinator	**(none)**	*(none)*	Array of course offerings coordinated by the user. Each includes OfferingID, CourseCode (and course name), and session/term information. This list lets the coordinator pick a course to view student details.
/Coordinator/GetStudentsInCourse/{offeringId}	GET	For a given course offering (that the coordinator manages), retrieve the list of students enrolled and their in-class performance. This allows the coordinator to see how students are doing in coursework. It typically shows each student's continuous assessment score ("in-class marks") for that course, and possibly their final exam score if available, along with basic info. It also inherently shows the number of students (the list length is the enrollment count).	Coordinator	**offeringId (course offering ID for the class)**	*(none)*	Array of student performance entries for the course. Each entry includes StudentID, name, and their in-course assessment marks (e.g. midterm/assignment total). It may also include the final exam marks or final letter Grade if those are recorded, but primarily focuses on in-class marks as per requirements.
/Coordinator/GetStudentGPA/{studentId}	GET	Retrieve the current GPA of a particular student who is enrolled in one of the coordinator's courses. This allows the coordinator to understand the academic standing of students in their class (as an indicator of	Coordinator	**studentId (of a student in the coordinator's course)**	*(none)*	GPA value for the specified student. (Same calculation as the student's own GPA view.)

		overall performance). The coordinator should only use this for students they teach.				

Notes: Advisors and coordinators have restricted access to academic records. An **advisor** can only fetch data for their assigned students (advisees). Each student record links to an advisor via a StaffID field, enabling endpoints like GetAdvisees. Advisors can view an advisee's personal details and academic results (transcripts) to provide guidance. A **course coordinator** can view data for students in the courses they coordinate. The GetStudentsInCourse endpoint returns in-class marks (continuous assessment) from the Results table (the **InCourse** field) for each student in the class, so the coordinator can monitor progress. Coordinators can also fetch a student's GPA to identify if the student is high-performing or at risk. All GPA and grade information comes from the official Results records which include course grades.

Notification and Approval APIs

These endpoints handle the approval workflow for course registrations and the notification system that alerts users (students, advisors, AR staff) about registration events. When a student registers for a course, their advisor must approve it, followed by final approval from the AR officer. Notifications are generated at each step to inform the relevant parties.

End point	Method	Description	Required Role	Request Parameters	Request Body	Response
/Advisor/GetPendingRegistrations	GET	List all pending course registration requests from the advisor's advisees that are awaiting the advisor's decision. Each entry corresponds to a student's request for a course, which has not yet been approved or rejected by the advisor (IsApprovedByAdvisor is false). This allows the advisor to see which registrations need their attention.	Advisor	*(none)*	*(none)*	Array of pending registration entries (for the advisor's students). Each includes **PendingID**, the student's name/ID, course details, request date, and current status (e.g. "Pending Advisor Approval").

/Advisor/ ApproveRegistration/{pendingId}	POST	Approve a student's pending course registration request. The advisor confirms that the student can take the course. This sets the advisor approval flag and updates the PendingRegistration status (e.g. to "Advisor Approved"). A notification is usually sent to the next party (the AR officer) indicating the advisor's approval.	Advisor	**pendingId** (the ID of the pending registration request)	*(none)*	Success response (HTTP 200 OK or similar). The pending registration's status is updated (advisor approval recorded). A notification will be created to inform the AR officer of this approval.
/Advisor/ RejectRegistration/{pendingId}	POST	Reject a student's pending registration request. The advisor declines the course request (for example, if the course load is too high or prerequisites not met). The record is marked as rejected by advisor (status updated accordingly) and the student will be notified of the rejection.	Advisor	**pendingId**	*(none)*	Success response indicating the request was rejected. The pending entry's status is set to a rejected state, and a notification to the student is generated.
/Admin/ GetPendingRegistrations	GET	View all pending course registration requests that have been submitted by students. This includes those waiting for advisor approval and those awaiting final AR approval. The AR officer can **monitor the status of each request**, seeing whether each has been advisor-approved or notfile. Typically, the AR officer will focus on those already approved by advisors for final processing.	Admin	*(none)*	*(none)*	Array of pending registration entries for all students. Each entry includes student info, course info, and current status (whether advisor-approved or still pending advisor). This allows the AR staff to identify which requests are ready for final approval.
/Admin/ Approve Registration/{pendingId}	POST	Final approval of a course registration. The AR officer approves a pending request that has (usually) been advisor-approved. This action confirms the	Admin	**pendingId**	*(none)*	Success response. The registration is finalized (the student is officially enrolled). The pending request

		student's enrollment in the course. The system will move the record from pending to the official Registrations table and mark it as fully approved. A notification is sent to the student (and possibly the advisor) that the registration is approved.				entry is resolved/removed, and the student's registration record is now in the permanent list. A notification is generated to inform the student of the approval.
/Admin/RejectRegistration/{pendingId}	POST	Final rejection of a course registration. The AR officer denies the pending request (for example, if it was not approved by the advisor or due to administrative reasons). This marks the request as rejected at the final stage. A notification is sent to the student to inform them.	Admin	**pendingId**	*(none)*	Success response. The pending request is marked rejected/finalized as not approved. A notification is generated for the student (and advisor) about the rejection.
/Notification/GetNotifications	GET	Retrieve all notifications for the current user (student or staff). Notifications are generated by events in the registration process (and possibly other events) and can include messages such as a request being approved or a new request needing action. This endpoint filters notifications where the user is the intended receiver. It also allows the UI to display which notifications have been read or are new (IsRead flag).	Student/ Advisor/ Coordinator/Admin (any authenticated user)	*(none)*	*(none)*	Array of notification objects for the user. Each includes details like a message, type of notification (e.g. approval, reminder), timestamp, and a flag indicating if it has been read.
/Notification/MarkAsRead/{notificationId}	POST	Mark a specific notification as read. This updates the notification's status so that it is not shown as new/unread in the user's notification list. It is typically called when a user views or	Student/ Advisor/ Coordinator/Admin (any, for their own notifications)	*notificationId*	*(none)*	Success status (e.g. 200 OK). The notification with the given ID is updated in the database (its IsRead field set to true).

		acknowledges a notification.				

Notes: The approval process involves two stages. When a student registers, a pending record is created (status "Pending"). The advisor's approval endpoints update IsApprovedByAdvisor and status in that pending record. Once the advisor approves, the AR officer can see that the request is advisor-approved and then use the final approve endpoint to complete it. Final approval causes the record to be recorded in the permanent **Registrations** table. Throughout this process, the system generates **Notification** records to inform the relevant parties. The Notification table includes fields to tailor messages for receivers and track read status, as well as links to related entities like the pending registration ID. Users can fetch their notifications and mark them as read via the notifications API. This ensures that students know when their request is approved/rejected, advisors know when a new request arrives, and AR staff know when an advisor has approved a request.

4. Testing

4.1 Functional Testing

4.1.1 Unit Testing (xUnit)

4.1.1 Authentication and Access Control

- **ACC01 –** *SignInUser_ShouldCreateCorrectClaims*
  Verifies role-based claims creation during login.
- **AUTH01 –** *Login_WithValidStudentCredentials_ReturnsSuccessResponse*
  Confirms successful student login via API.
- **AUTH01 –** *Login_WithInvalidCredentials_ReturnsUnauthorized*
  Ensures 401 Unauthorized response for invalid credentials.
- **DB01 –** *AdminDashboard_ShouldReturnViewResult_WhenUserIsAuthorized*
  Validates authorized access to Admin Dashboard.

4.1.2 Registration Management

- **AC01 –** *ExportSession_ShouldReturnPDFFile*
  Validates generation of session registration reports in PDF.
- **AC02 –** *CreateRegistrationSession_ShouldInsertSessionAndOfferings*
  Confirms successful insertion of session and related courses.
- **AC03 –** *GetPendingRegistrations_ShouldReturnData*
  Ensures retrieval of student pending course registrations.
- **ST01 –** *SubmitCourseRegistration_ShouldInsertPendingRegistration*
  Tests successful creation of pending course registration.

4.1.3 Advisor and Coordinator Functionalities

- **ADV01 –** *SubmitApprovals_ShouldUpdatePendingRegistration*
  Verifies advisor approval logic for pending registrations.
- **ADV02 –** *GetStudentResults_ShouldReturnCorrectData*
  Checks accurate fetch of student results for advisors.
- **COORD01 –** *GetStudentResults_ShouldReturnCorrectData*
  Validates student result access for coordinators.

4.1.4 Supporting Services

- **CS01 –** *CloseExpiredSessions_ShouldCloseAll*
  Tests logic for closing all expired registration sessions.
- **CT01 –** *GetRegisteredCredits_ShouldReturnCorrectCredit*
  Ensures accurate calculation of pending semester credits.
- **SS01 –** *GetCurrentSemester_ShouldReturnCorrectSemester_WhenWithinRange*
  Tests current semester identification logic.

Code	Test Method	Description	Outcome
ACC01	SignInUser_ShouldCreateCorrectClaims	Verifies claims during login	☑ Passed
AC01	ExportSession_ShouldReturnPDFFile	Checks PDF export	☑ Passed
AC02	CreateRegistrationSession_ShouldInsertSessionAndOfferings	Inserts sessions and course offerings	☑ Passed
AC03	GetPendingRegistrations_ShouldReturnData	Validates pending registrations fetch	☑ Passed
ADV01	SubmitApprovals_ShouldUpdatePendingRegistration	Advisor approves pending	☑ Passed
ADV02	GetStudentResults_ShouldReturnCorrectData	Advisor result fetch	☑ Passed
AUTH01	Login_WithValidStudentCredentials_ReturnsSuccessResponse	Valid Student API login	☑ Passed
AUTH01	Login_WithInvalidCredentials_ReturnsUnauthorized	Invalid login returns 401	☑ Passed
COORD01	GetStudentResults_ShouldReturnCorrectData	Coordinator result fetch	☑ Passed
DB01	AdminDashboard_ShouldReturnViewResult_WhenUserIsAuthorized	Admin dashboard access	☑ Passed
ST01	SubmitCourseRegistration_ShouldInsertPendingRegistration	Submits new pending	☑ Passed
CS01	CloseExpiredSessions_ShouldCloseAll	Service to close expired sessions	☑ Passed
CT01	GetRegisteredCredits_ShouldReturnCorrectCredit	Tracks credit limit	☑ Passed
SS01	GetCurrentSemester_ShouldReturnCorrectSemester_WhenWithinRange	Calculates semester	☑ Passed

Table 4.1 – Unit test cases and their execution outcomes.

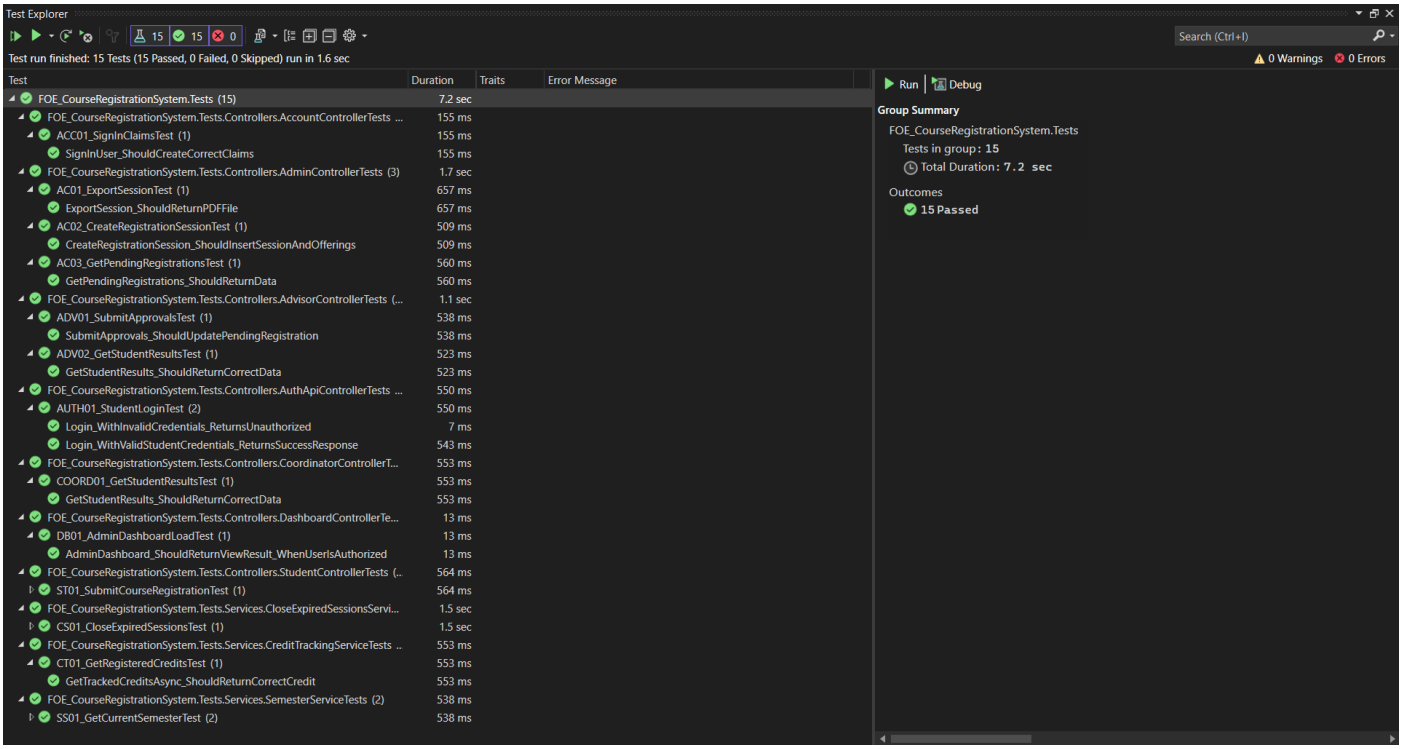


Fig 4.1.1 – Execution summary of xUnit unit tests showing all 15 test cases passed successfully.

Fig 4.1.2 – Detailed xUnit test report listing all 15 test cases executed with 100% pass rate, including individual method names and associated test classes.

4.2 API Testing (Postman)

4.2.1 Authentication and Access Control

- **AUTH01 –** Login_WithValidStudentCredentials_ReturnsSuccessResponse
  Confirms successful student login via API.
- **AUTH02 –** Login_WithInvalidCredentials_ReturnsUnauthorized
  Ensures 401 Unauthorized response for invalid login attempts.

4.2.2 Registration Management

- **AC03 –** GetPendingRegistrations_ShouldReturnData
  Fetches pending course registrations for students by Admin.

4.2.3 Advisor Functionalities

- **ADV02 –** GetStudentResults_ShouldReturnCorrectData
  Verifies advisor's access to advisee student results via secure API.

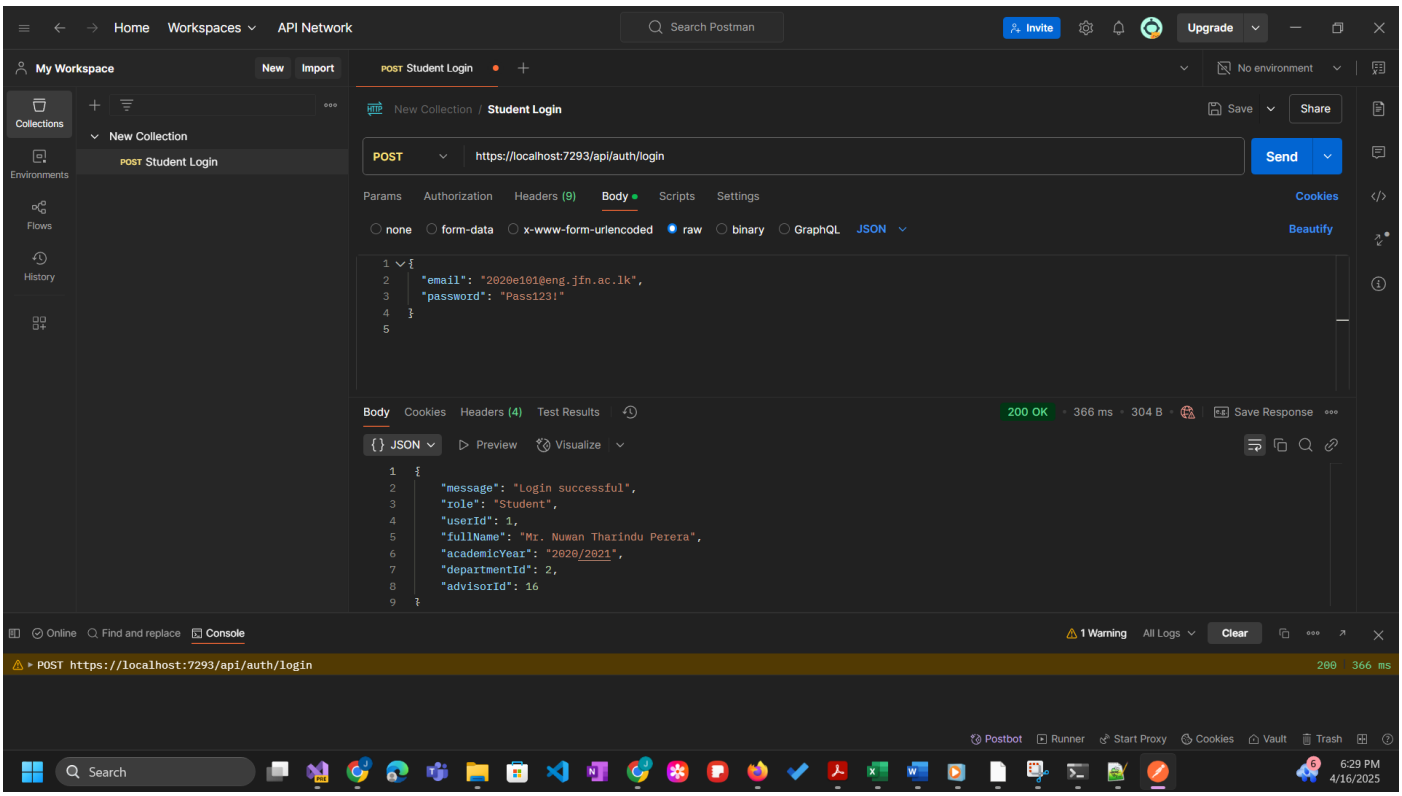


Fig 4.2.1. – Valid Login API tested successfully via Postman showing 200 OK response.

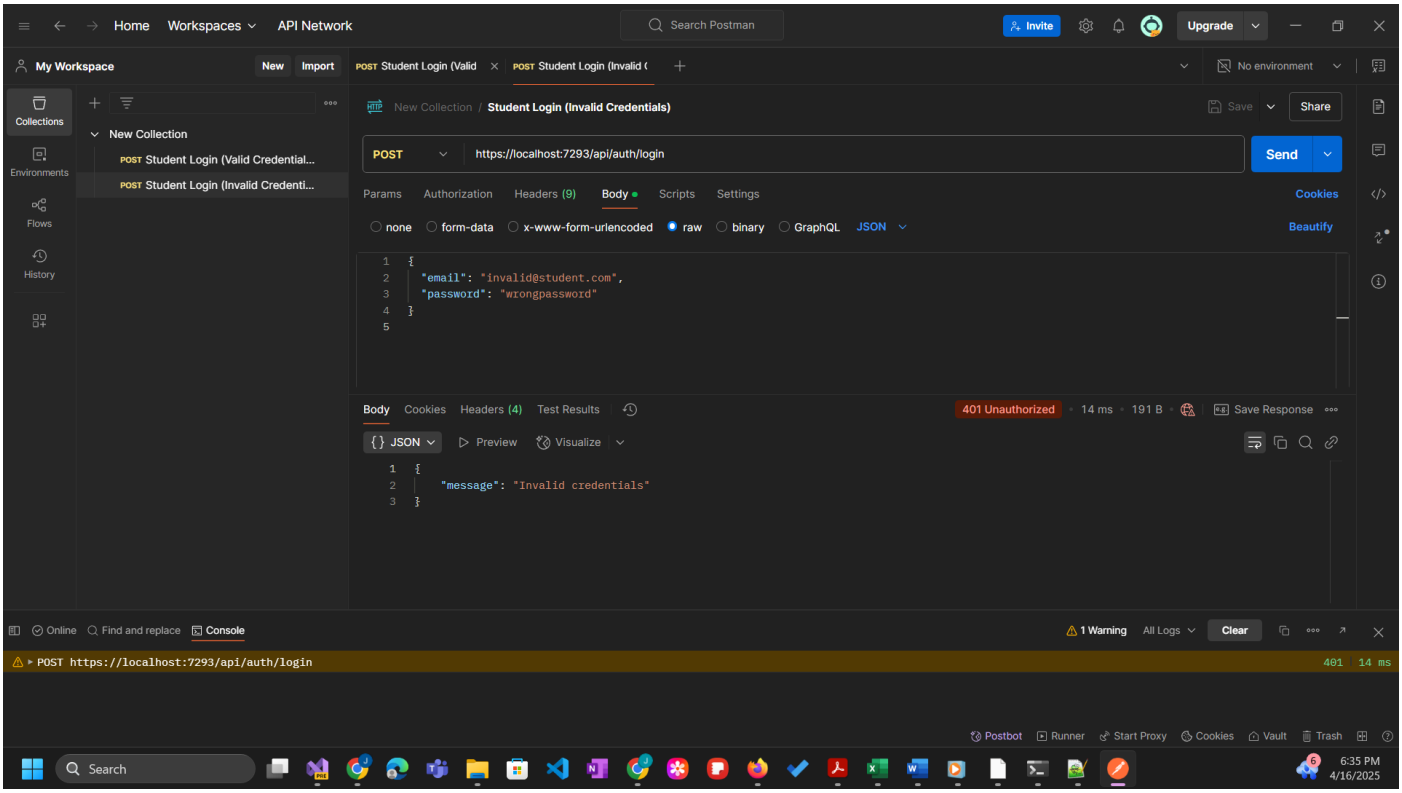


Fig 4.1.2 – Invalid Login API returns 401 Unauthorized with correct error message.

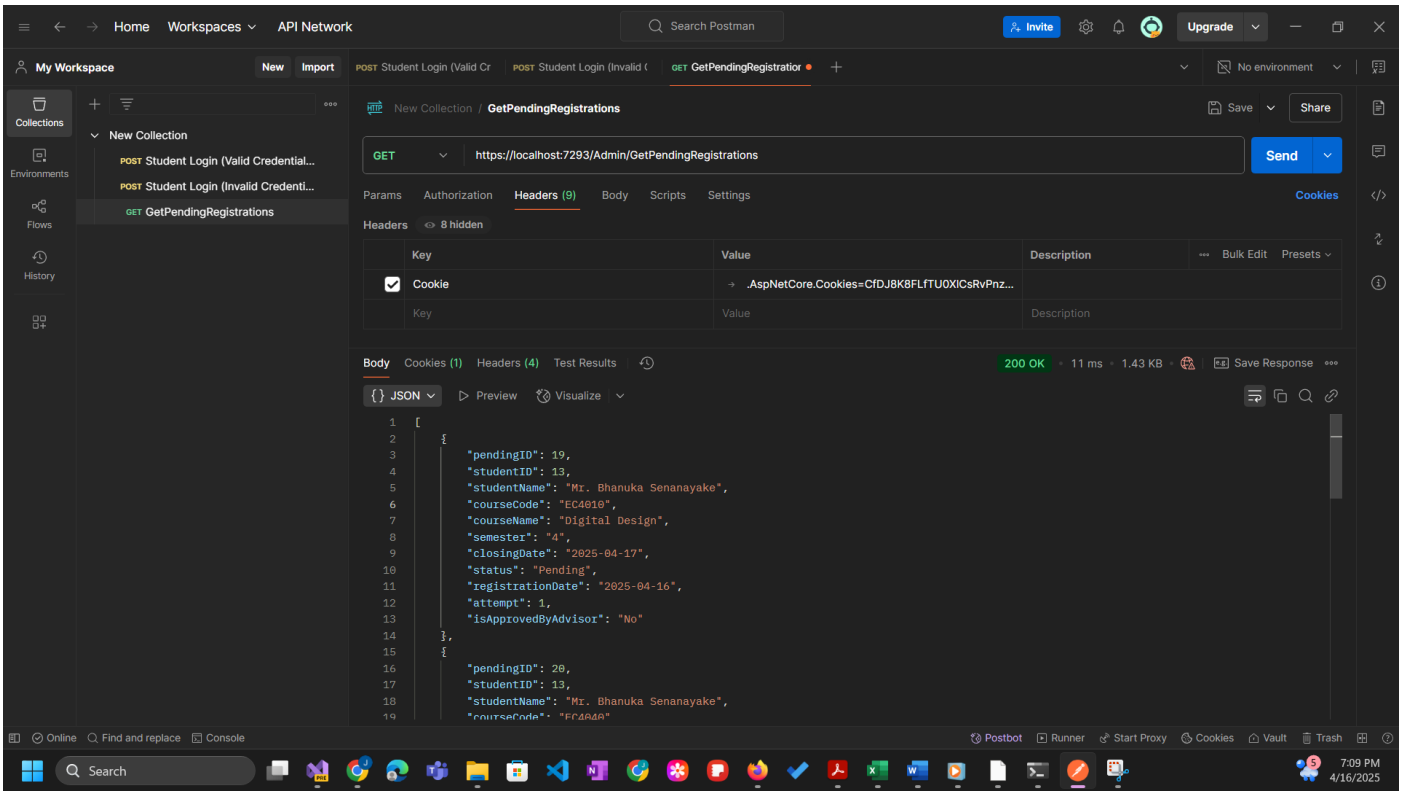


Fig 4.2.3 – Admin fetching pending course registrations via GET API using session cookie.

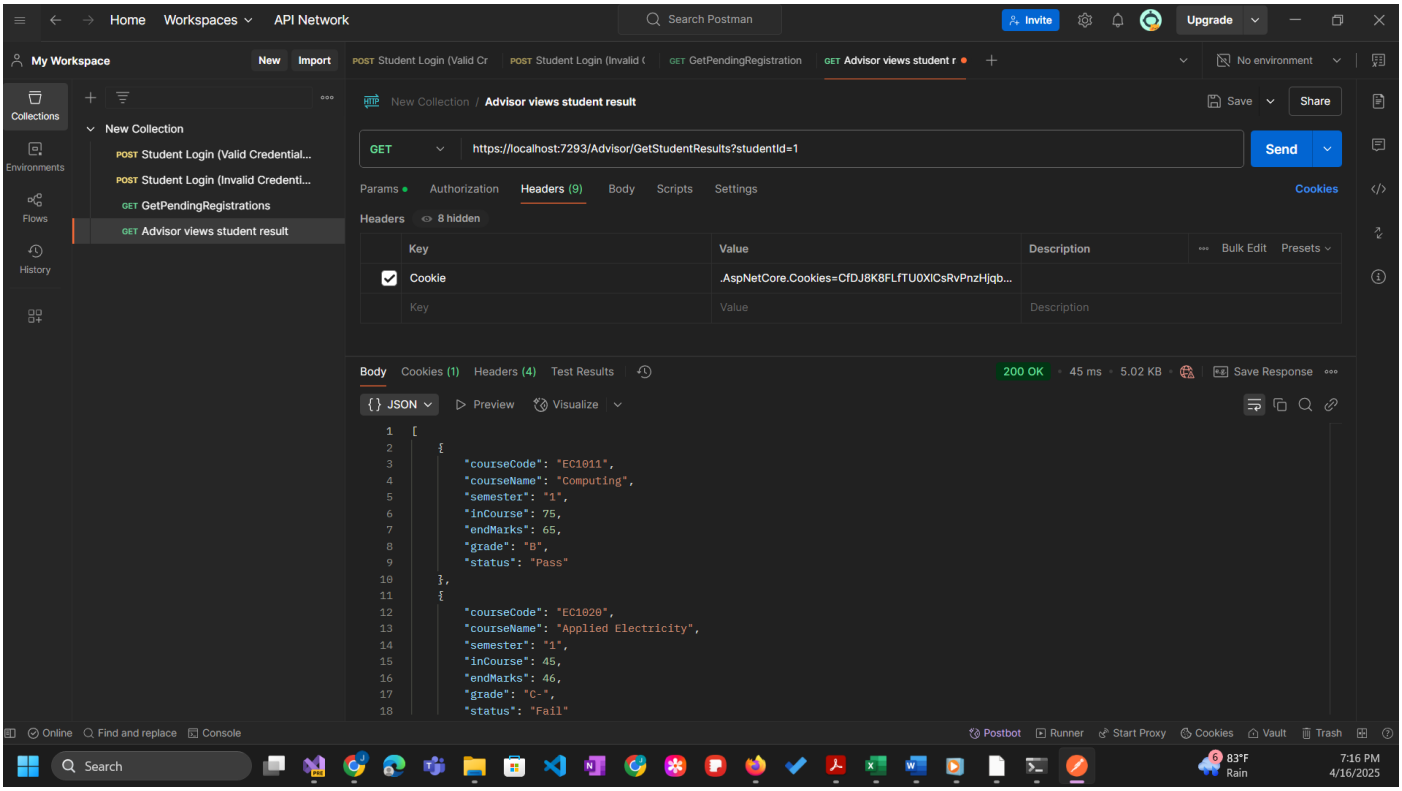


Fig 4.2.4 – Advisor successfully retrieves result data of advisee using secure GET API.

4.3 UI Testing (Selenium)

4.3.1 Authentication Flow

- **Login Page Loads Properly**
  Ensures the login UI renders with form fields and labels.
- **Valid Login Test**
  Simulates login with correct credentials and checks dashboard load.
- **Invalid Login Test**
  Enters wrong credentials and verifies error message display.
- **Logout Test**
  Confirms user is redirected to login page after logout.

4.3.2 Navigation Flow

- **Student Dashboard Navigation**
  Confirms links to Register Course, Results, Registered Courses, and Notifications are working.
- **Admin Panel Links**
  Tests navigation from sidebar (Dashboard, Course Offering, Session Management, Advisor List).

4.3.3 Registration Form Validation

- **Empty Field Submission**
  Tests course registration form submission without selecting courses.

- **Valid Course Selection**
  Confirms registration goes through when valid courses are selected.

4.3.4 Dashboard Data Verification

- **Student Info Displayed**
  Checks that name, GPA, credits, and advisor info are loaded on the student dashboard.
- **Admin Statistics Display**
  Verifies lecturer, student, and session stats render correctly.

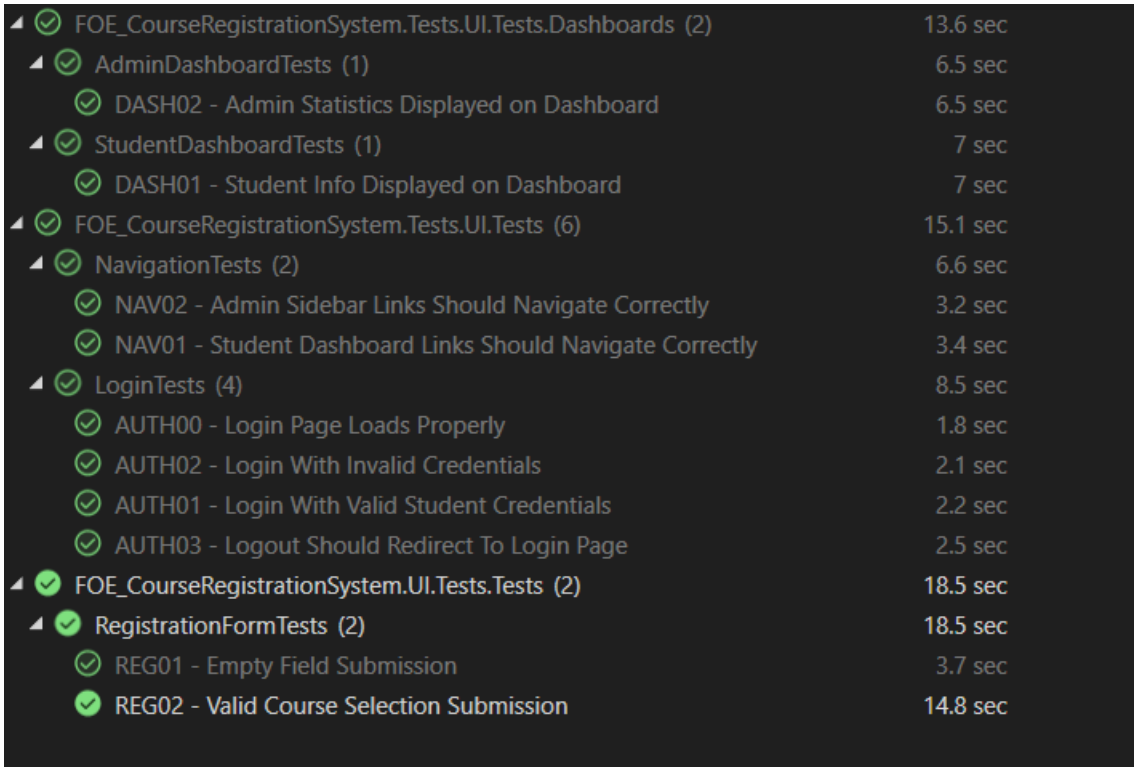


Fig 4.3.1 – Selenium UI test results showing successful execution of authentication, navigation, dashboard, and registration form validations.

4.3 UI Testing Summary (Selenium, xUnit)

Section	Test Case ID	Test Description	Status	Test File
4.3.1 Authentication Flow	AUTH00	Login Page Loads Properly	☑ Passed	LoginTests.cs
	AUTH01	Login With Valid Student Credentials	☑ Passed	LoginTests.cs
	AUTH02	Login With Invalid Credentials	☑ Passed	LoginTests.cs
	AUTH03	Logout Should Redirect To Login Page	☑ Passed	LoginTests.cs
4.3.2 Navigation Flow	NAV01	Student Dashboard Links Should Navigate Correctly	☑ Passed	NavigationTests.cs
	NAV02	Admin Sidebar Links Should Navigate Correctly	☑ Passed	NavigationTests.cs
4.3.3 Registration Form Validation	REG01	Empty Field Submission	☑ Passed	RegistrationFormTests.cs
	REG02	Valid Course Selection Submission	☑ Passed	RegistrationFormTests.cs
4.3.4 Dashboard Data Verification	DASH01	Student Info Displayed on Dashboard	☑ Passed	StudentDashboardTests.cs
	DASH02	Admin Statistics Displayed on Dashboard	☑ Passed	AdminDashboardTests.cs

Table 4.3 – Selenium-based UI test scenarios summary.

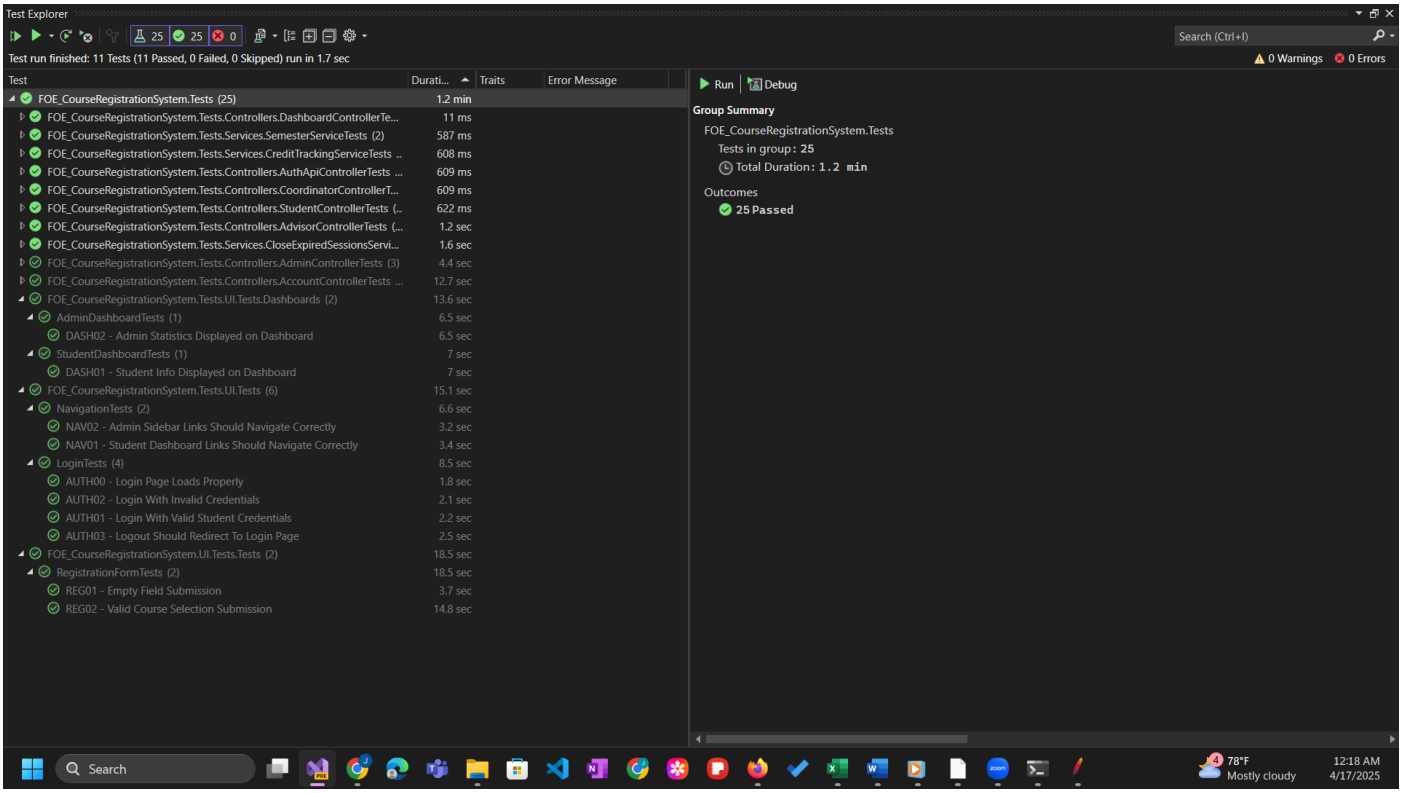


Fig 4.3.2 – Complete xUnit and Selenium UI test suite execution showing all 25 test cases passed successfully across unit, controller, and UI flows.

4.4. Non-Functional Testing

4.4.1 Performance Testing

4.4.1 Load Testing (Apache JMeter)

Objective

Simulate 50 concurrent users logging into the FOE Course Registration System to test how the system handles load and response times.

Tool Used

- Apache JMeter 5.6.3

Test Configuration

Setting	Value
Number of Threads (Users)	50
Ramp-up Period (Seconds)	10
Loop Count	10
Protocol	https
Server Name or IP	localhost
Port Number	7293
Path	/Account/Login

Request Method	POST
Parameters Sent	Email, Password
Headers	Content-Type: application/x-www-form-urlencoded

Table 4.4.1 – Apache JMeter configuration for simulating 50 concurrent user login requests to the /Account/Login endpoint.

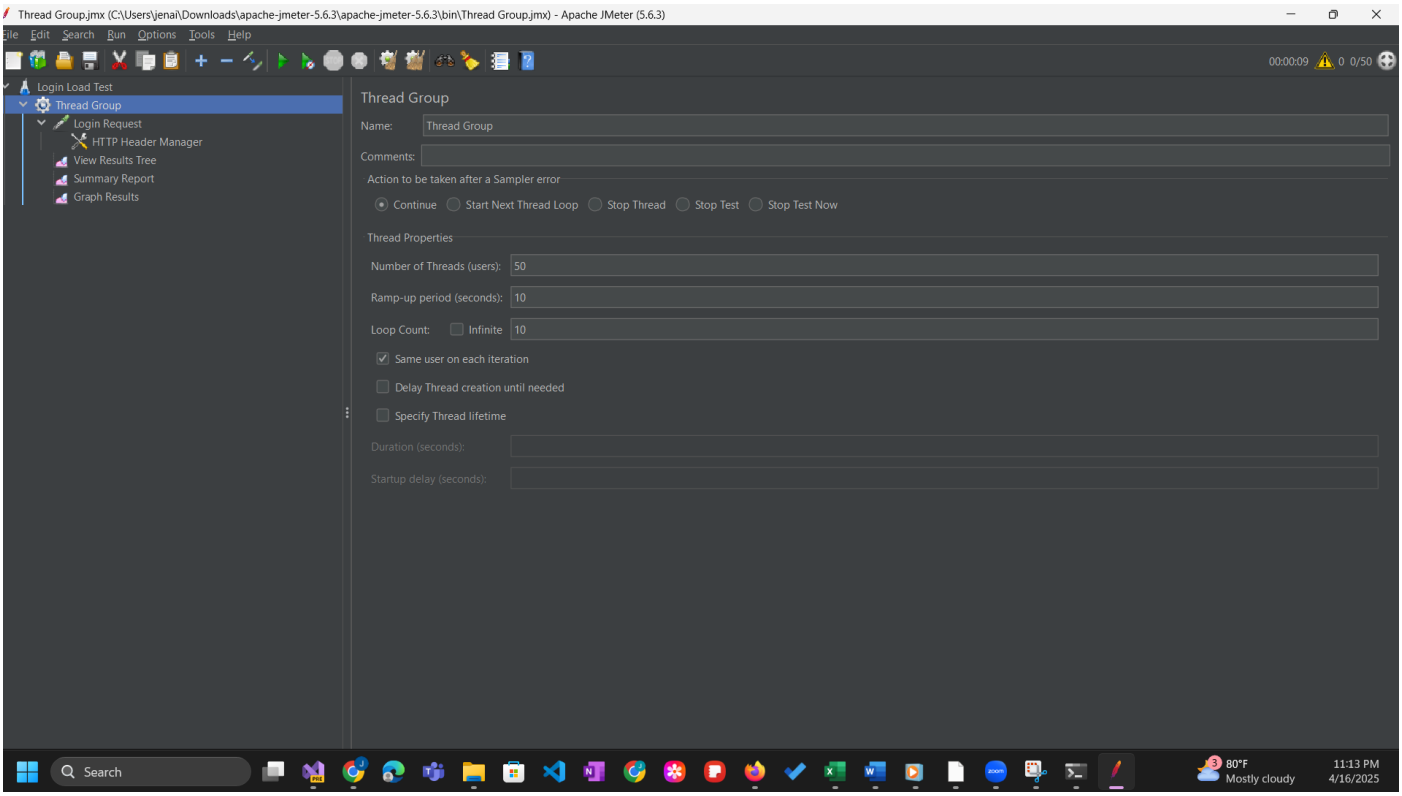


Fig 4.4.1.1 – Thread Group Setup

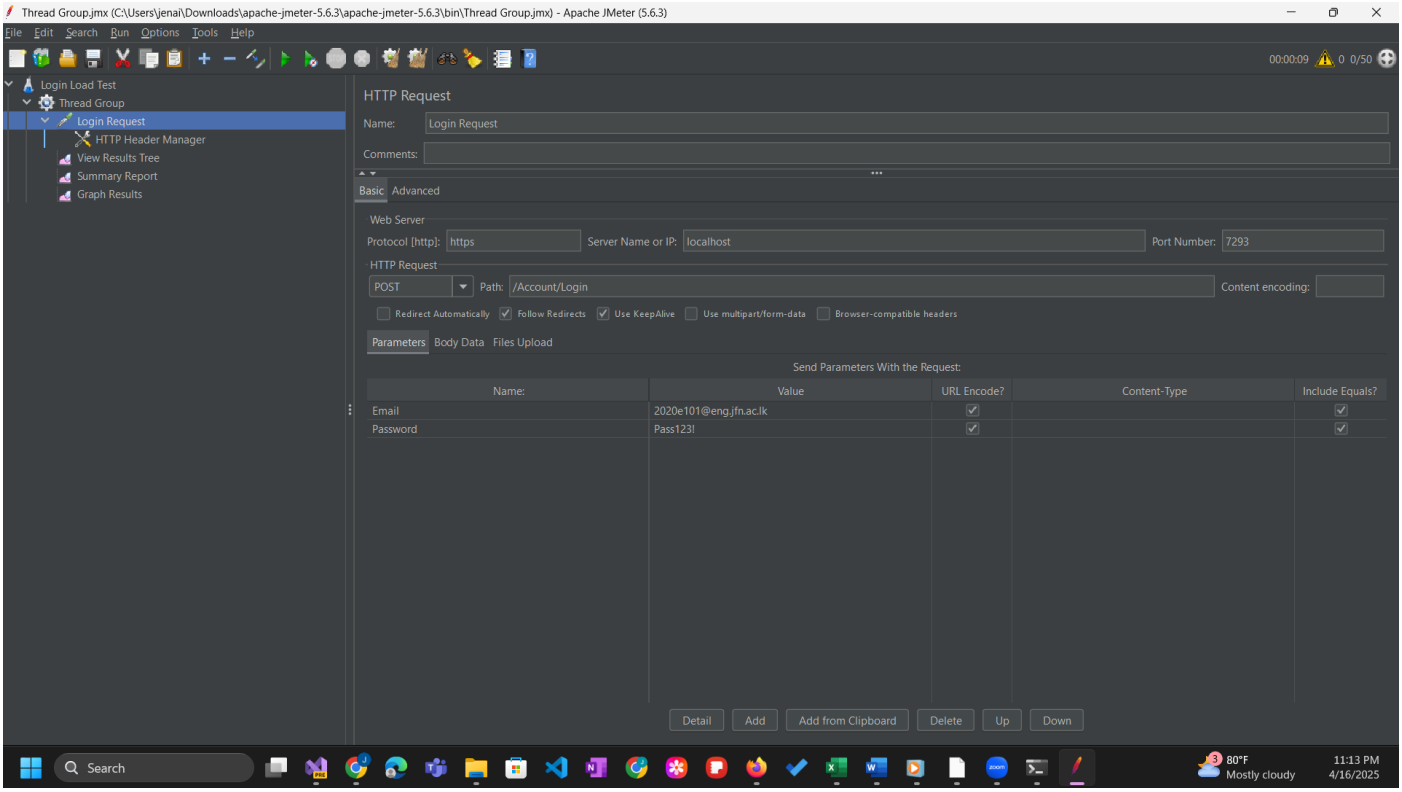


Fig 4.4.1.2 – HTTP Request

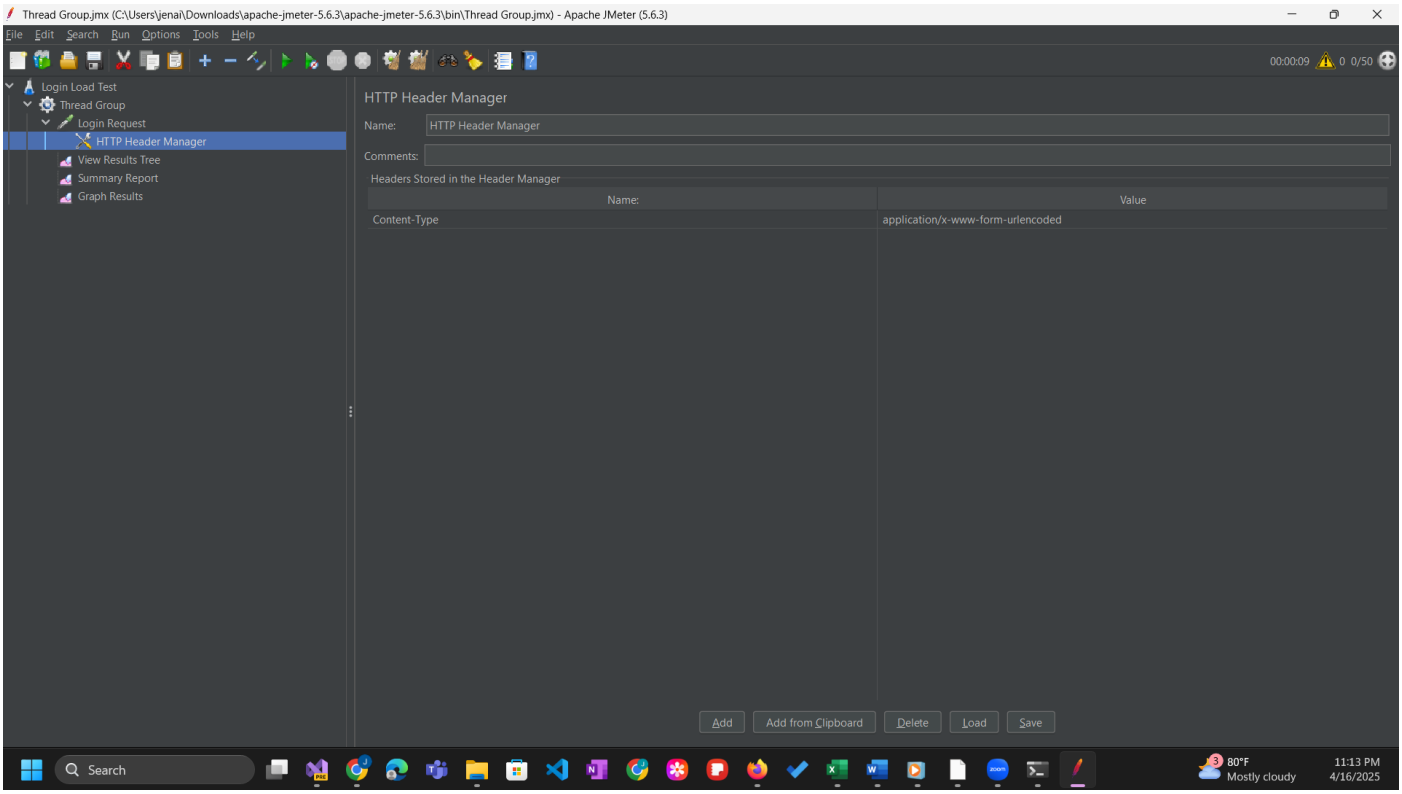


Fig 4.4.1.3 – Header Manager

Listeners Used for Result Visualization

Listener Name	Description
View Results Tree	To view each request and corresponding response
Summary Report	Displays aggregated test metrics
Graph Results	Visual representation of response time and load

Table 4.4.1.2 – JMeter listeners used to monitor, visualize, and summarize login request performance results.

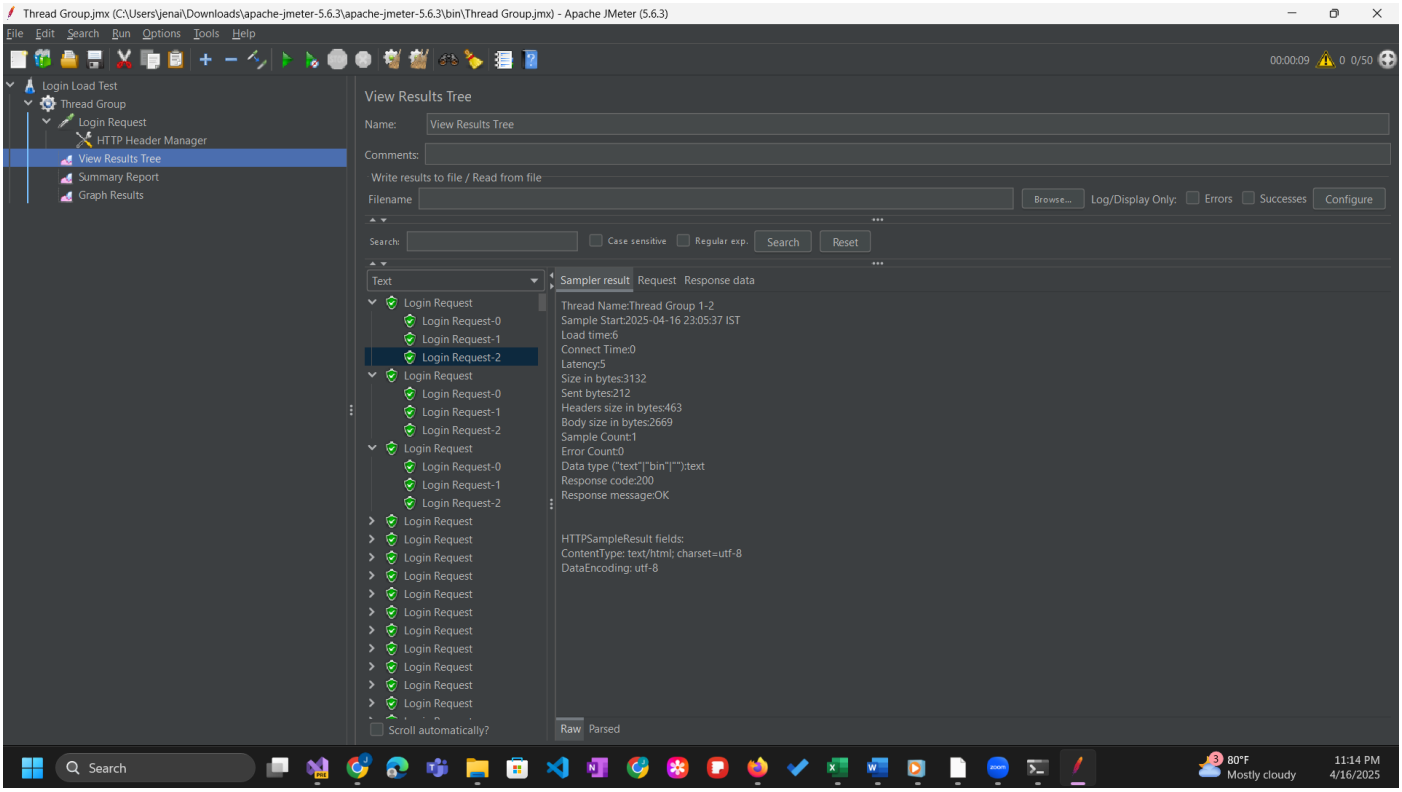


Fig 4.4.1.4 – View Results Tree

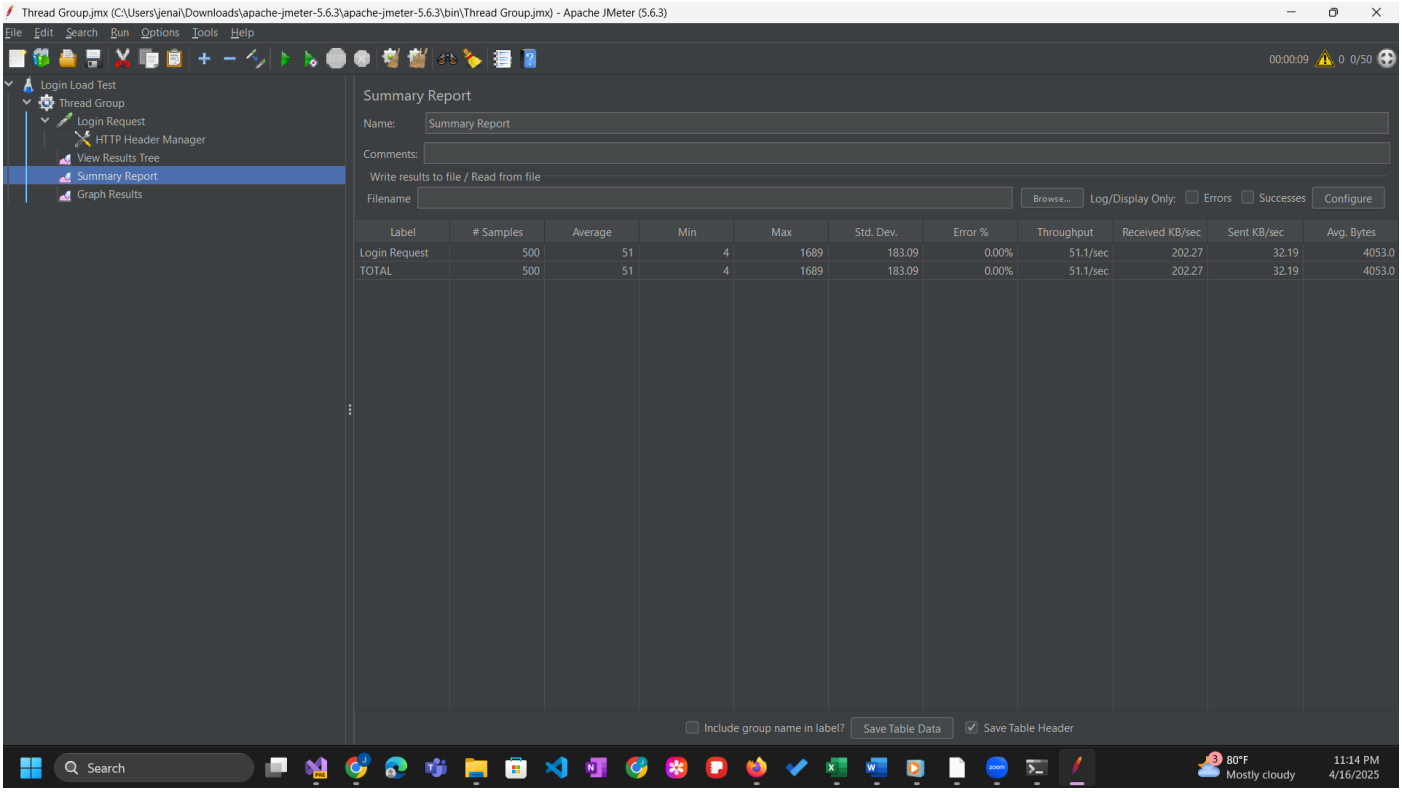


Fig 4.4.1.5 – Summary Report

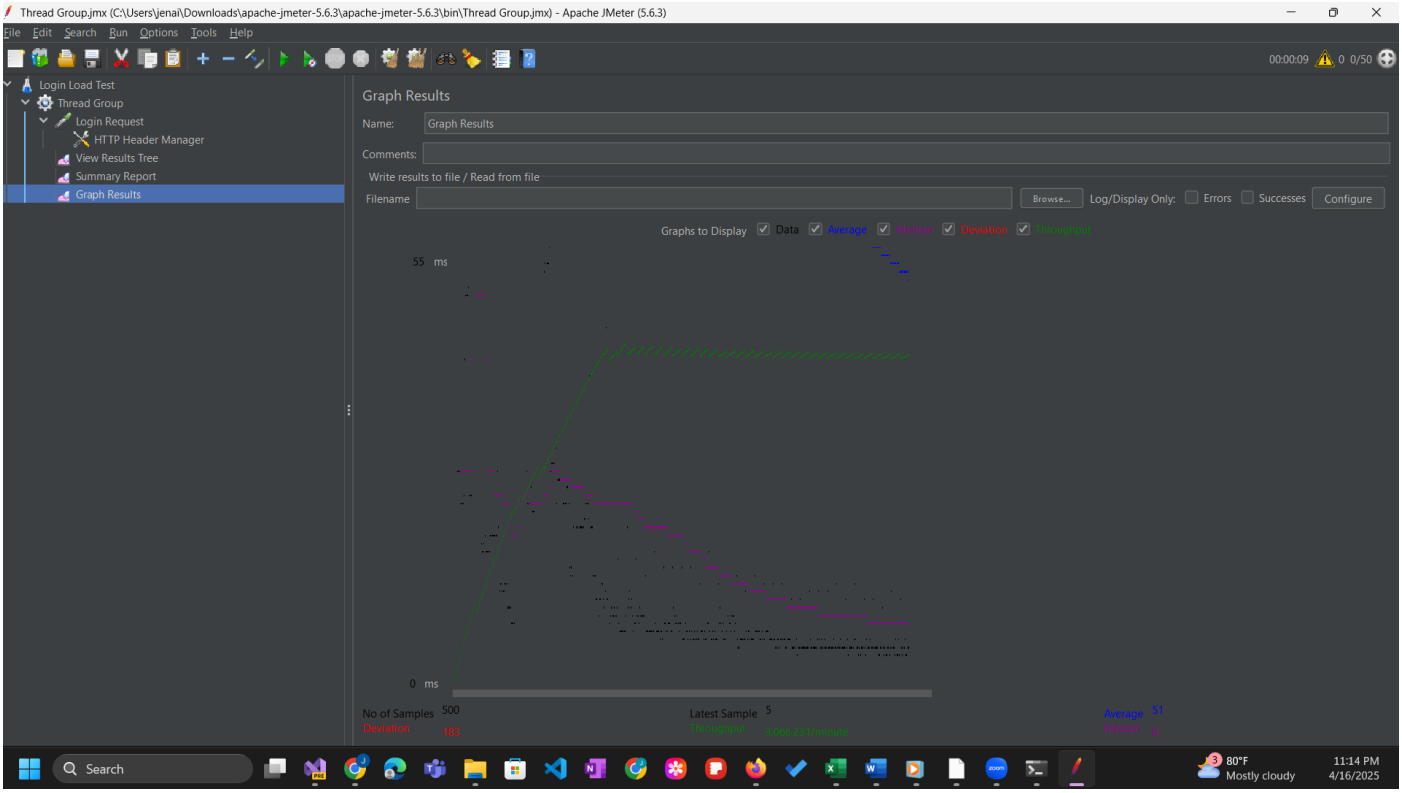


Fig 4.4.1.6 – Graph Results

Summary of Results

Metric	Value
Samples Executed	500
Average Response Time	51 ms
Minimum Response Time	4 ms
Maximum Response Time	1689 ms
Error Rate	0.00%
Throughput	51.1 requests/second
Received KB/sec	202.27
Sent KB/sec	32.19

Table 4.4.1.3 – Summary of load test results showing response times, error rate, and throughput for 500 simulated login requests.

All login requests completed successfully without errors. The system responded consistently under concurrent load.

Conclusion

The system successfully handled 50 concurrent login users. The login functionality maintained consistent throughput and response time under load. This indicates the login module is scalable and performs reliably under typical usage scenarios.

4.4.2 Security Testing

4.4.2.1 Authentication & Authorization Validation

- **Objective:**
  Ensure that only authenticated users can access protected pages such as student dashboards and admin panels.
- **Approach:**
  - Logged out and manually attempted to access protected URLs (e.g., /Student/Dashboard, /Admin/AdminDashboard).
  - Checked if unauthenticated users were redirected to the login page or shown an appropriate unauthorized access message.
  - Logged in with different roles (Student, Admin) and confirmed that role-specific pages were not accessible to other roles.
- **Result:**
  All protected routes correctly enforced authentication. Unauthenticated users were

redirected to the login page, and unauthorized role-based access was denied as expected. The test was manually verified and passed.

4.4.2.2 Role Escalation Attempts

- **Objective:**
  Ensure that users cannot escalate their privileges by manually accessing URLs meant for other roles (e.g., a student trying to access admin or staff functionalities).
- **Approach:**
  - Logged in as a **Student** and manually entered URLs such as:
    - /Admin/AdminDashboard
    - /Admin/CourseOffering
    - /Admin/AdvisorDetails
  - Observed the system response.
  - Repeated the process with other roles to attempt cross-role access.
- **Result:**
  All unauthorized access attempts were blocked. The system correctly restricted students from accessing admin-specific pages. Role-based authorization checks are functioning correctly. This test was **manually performed and passed**.

4.4.2.3 SQL Injection (Login / Input Forms)

- **Objective:**
  Ensure that input fields such as login and course-related queries are protected against SQL injection attempts.
- **Approach:**
  - Manually tested SQL injection payloads like ' OR 1=1--, '; DROP TABLE Students-- on:
    - **Login form**
    - **Course-related pages**
    - **URL parameters**, e.g., https://localhost:7293/Advisor/AdviseeStudentsResult?studentId=1
      - Modified the studentId value to test unauthorized access and potential injection vectors.
  - Observed application responses for:
    - Unauthorized access
    - SQL errors or exceptions
    - Bypassed validation

- **Security Implementation:**
  The method AdviseeStudentsResult(int studentId) includes logic to verify that the student belongs to the currently logged-in advisor, preventing access via direct URL manipulation:

var student = await _context.Students

.FirstOrDefaultAsync(s => s.StudentID == studentId && s.StaffID == advisor.StaffID);

This logic filters and validates using parameters inside **Entity Framework Core LINQ queries**, which **automatically parameterizes queries**, significantly reducing the risk of SQL injection.

- **Note:**
  Most database operations in the system are built using **Entity Framework Core (EF Core)**. EF Core handles query parameterization internally, offering robust protection against SQL injection by default.
- **Result:**
  All manual SQL injection attempts were blocked. The system responded securely without exposing stack traces or sensitive error messages. No data was accessed or modified improperly.
  This test was manually performed and passed.

4.4.2.4 Password Encryption & Storage

- **Objective:**
  Verify that user passwords are securely stored in the database in hashed form, not as plain text.
- **Tool Used:**
  SQL Server Management Studio (SSMS) – executed direct queries on the Students and Staffs tables.
- **Approach:**
  - Queried user records in the database to inspect stored values in the Password column.
  - Checked for:
    - Presence of hashed or encrypted strings.
    - Use of recognizable hashing algorithms (e.g., bcrypt, SHA256).
    - Absence of plain text passwords.
- **Observation:**
  - **Current passwords are stored in plain text** in the database.
  - This is acceptable for **demonstration and academic purposes only**.
  - The project acknowledges this and **plans to implement secure password hashing** in future updates using techniques such as:
    - Hashing with salt (e.g., using ASP.NET Identity's built-in password hasher)

        - Storing hashed passwords instead of raw strings.

- **Security Note:**
  For production-ready deployment, it is essential to:
    - Hash all passwords before storing.
    - Use a cryptographically secure hashing algorithm (e.g., bcrypt, PBKDF2).
    - Never log or transmit raw passwords.
- **Result:**
  Manually tested – Passwords are currently not hashed, but this is a known and accepted limitation for the demo.
  Security improvement recommended before public deployment.

4.4.3 Compatibility Testing

4.4.3.1 Browser Compatibility

- **Objective:**
  Ensure the web application's UI renders correctly across major modern browsers.
- **Browsers Tested:**
    - Google Chrome
    - Mozilla Firefox
    - Microsoft Edge
- **Approach:**
    - Manually opened and interacted with key pages including Login, Student Dashboard, Admin Dashboard, Course Registration, and Result Pages in each browser.
    - Checked layout consistency, element alignment, and font/icon rendering.
- **Result:**
  Pass — The UI appeared consistent, functional, and seamless across all three browsers tested on the desktop.

4.4.3.2 Device Compatibility

- **Objective:**
  Verify that the application layout and content are responsive and user-friendly on different screen sizes (desktop, tablet, mobile).
- **Tool Used:**
  Chrome DevTools → Device Emulation Mode (Responsive, iPad, iPhone, Galaxy)
- **Observation:**
    - **Desktop:** Fully responsive and visually consistent across screen resolutions.

- **Tablet and Mobile:** Some elements are not responsive (e.g., sidebars, tables, font resizing).
- This is **due to the use of custom static CSS** instead of modern responsive frameworks (like Bootstrap/Flexbox/Grid).

- **Conclusion:**
  - Desktop experience is smooth and consistent.
  - Mobile and tablet responsiveness needs improvement.
  - The issue is acknowledged and **planned to be fixed in future UI updates** using responsive design practices.

- **Result:**
  Partially Pass — Fully compatible on desktop; improvements needed for mobile and tablet views.

5. Installation Guide

5.1 System Requirements

Before setting up the Course Registration System locally, ensure your development machine meets the following requirements:

- **Operating System:** Windows 10 (64-bit) or higher (Windows 11 recommended) for compatibility with Visual Studio 2022.
- **Hardware:** A modern **x64 processor** (quad-core or better recommended) with at least **4 GB of RAM** (16 GB is recommended for smooth performance) (Visual Studio 2022 System Requirements | Microsoft Learn). Ensure you have sufficient disk space – Visual Studio and SQL Server can require around **20–50 GB of free space** for a typical installation (Visual Studio 2022 System Requirements | Microsoft Learn). An SSD is recommended for faster performance.
- **Software: Visual Studio 2022** (Community or higher edition) with the ASP.NET and web development workload. The project targets **ASP.NET Core** (requires .NET 6 or .NET 7 SDK). Install the appropriate .NET SDK (if not already installed with Visual Studio). You will also need **Microsoft SQL Server** (Developer or Express Edition) and **SQL Server Management Studio (SSMS)** version 19 or 20 for database management.
- **User Permissions:** Administrator rights on the machine to install software and set up services (Visual Studio and SQL Server installers require admin privileges (Visual Studio 2022 System Requirements | Microsoft Learn) (Visual Studio 2022 System Requirements | Microsoft Learn)). An active internet connection is needed initially to download installers and any NuGet packages the project might require.

5.2 Prerequisites Setup

Make sure to set up the following prerequisites before installing and running the application:

5.2.1 Database Server – Installing Microsoft SQL Server and SSMS

1. **Install SQL Server:** Download the installer for **Microsoft SQL Server** (Developer Edition is free for development use) from the official Microsoft website. Run the installer and follow the setup wizard. For a quick setup, you can choose the **Basic installation** option, which installs the SQL Server Database Engine with default settings. If you choose the **Custom installation**, ensure that the **Database Engine Services** component is selected. During setup, you may be prompted to configure authentication mode:
    - You can use **Windows Authentication (Trusted Connection)** which uses your Windows credentials (no need to set a SQL username/password).
    - Or enable **Mixed Mode** to set an sa admin password (and allow SQL Authentication). If you opt for Mixed Mode, provide a strong password for the sa account when prompted. (By default, the sa login is disabled on new installations – it must be enabled to use it (SQL Server Management Studio unable to connect to local instance - Server Fault).)

2. **Install SSMS:** After SQL Server is installed, download and install **SQL Server Management Studio (SSMS)** (a separate download from Microsoft). SSMS will allow you to manage the SQL Server instance through a graphical interface. Accept the default installation options for SSMS.

3. **Verify SQL Server is running:** Launch the **SQL Server Configuration Manager** (installed with SQL Server) or use Windows Services to ensure the SQL Server service is running. It typically starts automatically after installation. For Developer Edition with default instance, the service name is usually **MSSQLSERVER** (for Express, it might be **SQLEXPRESS**).

4. **Connect using SSMS:** Open **SSMS** and in the **Connect to Server** dialog:

    - **Server type:** Database Engine
    - **Server name:** If you installed the **Default Instance**, you can simply use (local) or . (dot) as the server name to connect to the local SQL Server (SQL Server Management Studio unable to connect to local instance - Server Fault). For a **named instance** (e.g., SQLEXPRESS), use .\SQLEXPRESS or the name you assigned.
    - **Authentication:** Choose **Windows Authentication** (if using your Windows account) or **SQL Server Authentication** (and enter the sa username and password you set, or another SQL login if you created one). Click **Connect**. If the connection succeeds, you will see the SQL Server instance in Object Explorer.

5. **Create a new database:** In SSMS Object Explorer, right-click on **Databases**, select **New Database...**, and enter a name for the project's database (for example, **CourseRegDB**). Leave other settings at defaults and click **OK** (Create a database - SQL Server | Microsoft Learn). The new database will appear under the Databases node.

If a database initialization script is provided with the project (for example, an .sql file containing table definitions), you can instead execute that script: open it in SSMS (File → Open → File, then execute) to create the schema. Ensure you run the script on the new database (select the database in the dropdown before executing).

5.2.2 .NET SDK/Runtime – Installing .NET 6/7 SDK

1. **Install .NET SDK:** Download and install the **.NET 6 SDK** (Long Term Support) or **.NET 7 SDK** from Microsoft's .NET website. The SDK includes everything needed to build and run ASP.NET Core applications (Download .NET 6.0 (Linux, macOS, and Windows) - Microsoft). Choose the version that matches the project's target framework (if unsure, .NET 6 is a safe choice as it's LTS). Run the installer and complete the setup with default options.

2. **Verify .NET installation:** After installation, you can verify the SDK is installed by opening a **Command Prompt** or **PowerShell** and running dotnet --version. This should output the installed .NET SDK version (e.g., 6.x or 7.x). Visual Studio 2022 can also be used to verify – it will detect the installed SDKs.

3. **Ensure Visual Studio has the ASP.NET workload:** If you haven't already, launch the **Visual Studio Installer**, modify your Visual Studio 2022 installation, and confirm that **ASP.NET**

and web development workload is installed (this ensures Visual Studio has templates and tools for .NET Core web projects). This step is usually done if you selected it during the initial VS installation, but it's good to verify.

5.2.3 Source Code or Package Acquisition – Cloning or Downloading the Project

1. **Obtain the source code:** Acquire the Course Registration System project code by either:
    - **Cloning the repository:** If the project is hosted on a version control platform (like GitHub), use git clone <repository_url> to clone the repository onto your local machine.
      OR
    - **Downloading the archive:** Download the project as a ZIP archive (for example, from a provided link or repository release). Once downloaded, right-click the ZIP file, choose **Extract All**, and extract the files to a convenient location (e.g., C:\Projects\CourseRegistrationSystem\).
2. **Locate the solution file:** In the extracted project folder, find the **Visual Studio solution file** (it will have a .sln extension, e.g., CourseRegistrationSystem.sln). This solution file groups all the project's components (such as the ASP.NET Core web project and any class libraries).
3. **Open the project in Visual Studio:** Double-click the .sln file, or open Visual Studio 2022, go to **File → Open → Project/Solution...** and select the solution file. Visual Studio will load the project. If prompted by Visual Studio to restore NuGet packages (project dependencies), allow it to do so. It may automatically restore packages on project open or build.

5.3 Installation Procedure

Now that prerequisites are in place, follow these steps to build the application and set up all necessary components.

5.3.1 Database Setup – Creating the Database Schema

With the database created in section 5.2.1, the next step is to set up the schema (tables, relationships, and initial data):

- **Using Entity Framework Migrations (Code-First):** If the project uses Entity Framework Core code-first migrations, you can auto-generate the database schema:
    1. In Visual Studio, open the **Package Manager Console** (PMC) via **Tools → NuGet Package Manager → Package Manager Console**.
    2. Ensure the **Default Project** in PMC is set to the EF Core project (usually the main web project if the DbContext is defined there).
    3. Run the update command:
    4. Update-Database

This will apply all pending EF Core migrations to the database, creating tables and seed data as defined by the project. (This command updates the database to the latest migration state (Applying

Migrations - EF Core | Microsoft Learn).) You should see a success message in the console after it runs.

5. In SSMS, refresh the database node to verify that tables have been created under your database's **Tables** folder.

- **Using SQL Script (Database-First):** If the project came with an SQL script to set up the database schema (or if you prefer manual setup):
    1. Open SSMS, connect to your SQL Server, and ensure the new database (created in 5.2.1) is selected.
    2. Click **New Query** and paste the SQL commands from the provided script (or open the .sql file directly in SSMS).
    3. Execute the script (press F5 or the Execute button). This will create the necessary tables, relationships, and possibly seed data.
    4. Verify the tables are created by expanding the **Tables** in Object Explorer for your database.
- **Confirm Schema:** After either method, verify that all expected tables and relationships exist (you might refer to the provided database schema diagram to cross-check table names and columns).

5.3.2 Configure Application – Editing appsettings.json for DB Connection

Next, configure the application to communicate with the local database:

1. **Open configuration file:** In Visual Studio, find and open the file **appsettings.json** (usually located in the root of the web project). This file contains configuration settings for the application, including database connection strings.
2. **Locate Connection Strings:** Find the "ConnectionStrings" section in appsettings.json. There will be an entry for the database connection (commonly named "DefaultConnection" or something similar). It will look like a JSON key-value pair, for example:

   "ConnectionStrings": {
   "DefaultConnection":"Server=...;Database=...;UserId=...;Password=...;Trusted_Connection=...;"}
3. **Modify the connection string:** Update the connection string to point to your local SQL Server instance and database:
    - **Server:** If you installed a default instance of SQL Server, use Server=(local) or Server=.. For a named instance (e.g., Express), use Server=.\SQLEXPRESS (replace SQLEXPRESS with your instance name as needed).
    - **Database:** Set this to the name of the database you created (e.g., CourseRegDB).
    - **Credentials:** If you are using Windows Authentication, include Trusted_Connection=True; and **omit** User Id/Password (also ensure Integrated Security=True if it appears – this is equivalent to Trusted_Connection). This will use

your Windows login to authenticate (SQL Server Management Studio unable to connect to local instance - Server Fault). If you prefer SQL Authentication, specify the User Id and Password for a SQL login (for example, the sa user if enabled, or another user you created) and set Trusted_Connection=False;. For example:

- Windows Auth example:
  "Server=(local);Database=CourseRegDB;Trusted_Connection=True;MultipleActiveResultSets=true;"
- SQL Auth example:
  "Server=.;Database=CourseRegDB;User Id=sa;Password=<YourPassword>;MultipleActiveResultSets=true;"
  Ensure **only one** of these methods is used (if both Trusted_Connection and User Id are present, Trusted Connection will typically override and use Windows auth).

4. **Save changes:** Save the appsettings.json file. If the project has environment-specific config (like appsettings.Development.json), also verify if a connection string is overridden there when running in development mode, and update it similarly.

5.3.3 Build and Deploy Application – Restoring Packages, Building, and Running

Now you will compile the project and ensure everything is ready to run:

1. **Restore NuGet packages:** If Visual Studio hasn't done so already, restore the project's NuGet dependencies. Usually, opening the solution or building will trigger an auto-restore. To manually restore, right-click the **Solution** in Solution Explorer and select **Restore NuGet Packages**, or use **Build → Restore NuGet Packages** from the menu. This will download any required libraries.
2. **Build the solution:** Go to **Build → Build Solution** (or press **Ctrl+Shift+B**). Visual Studio will compile the project. Watch the **Output** window for progress – it should eventually show **Build succeeded** with zero errors. If there are errors, fix them before proceeding (common issues might be missing SDK or wrong target framework – ensure the correct .NET SDK is installed if you see errors about target framework compatibility).
3. **Resolve build issues (if any):**
   - If you encounter errors about missing .NET SDK targeting .NET 6.0 or .NET 7.0 (e.g., error NETSDK1045), make sure the corresponding .NET SDK is installed (see section 5.2.2).
   - If there are package restore errors, ensure you have an internet connection and try restoring again. In some cases, you might need to update package versions to match your environment, but this is unlikely if using the same SDK version.
4. **Prepare for run:** Ensure that the **Course Registration System** web project is set as the Startup Project (in Solution Explorer, the project name should be bold, or right-click it and choose **Set as StartUp Project** if not). This ensures that pressing Run will launch the correct project. At this stage, the application is built and ready to be launched.

5.3.4 Initial Data Seeding – Setting Up Initial Accounts or Data

Depending on the project, you may need to create some initial data (like an administrator account or some reference data) to start using the system:

- **Administrator Account:** Check if the system already has a default admin user created by the migrations or included in the database script. If not, you will need to create one:
    - One approach is to **register a new user** through the application's interface once it's running (if the application has a registration page and doesn't restrict it to admins only). After creating a user, you can use SSMS to promote that user to an admin role if the database has roles setup (for example, insert an entry into an **Admin** or **Roles** table linking your user, or update a column like IsAdmin to true, depending on how the project handles admin users).
    - If there is no user registration available to the public, you might insert a user manually into the database. This requires knowing the password hashing method used. If the project uses ASP.NET Core Identity, manually inserting requires hashing a password using the same algorithm, so using the UI is easier. Alternatively, the team may have provided a script or instructions to create an initial admin user.
    - Another possibility: the code might automatically seed an admin user on first run (commonly with a default email like admin@example.com and a known password). Check the project documentation or ask the project authors if a default login exists. If one exists, you can use that for the initial login (see section 5.4.2).
- **Reference Data (Departments/Courses):** The Course Registration System might require some initial set of departments, courses, or academic terms. If the database was built via a script or migrations, some of this data may already be present. If not, you can add them:
    - Sometimes a project provides seed data via EF Core migrations or a JSON import. Ensure you ran all migrations as in 5.3.1, which might have inserted some default records.
    - If no data is present, you can use the application's admin interface after logging in to create entries (e.g., add new departments, courses, etc.). Alternatively, you could use SSMS to insert rows into the relevant tables, but using the application UI ensures all business rules are applied.
- **Verification of seeding:** After seeding, use SSMS or the application UI to verify that the data appears (e.g., the admin user exists in the Users table, the departments list is populated, etc.).

5.4 Running the Application

With the application built and configured, you can now run it locally and perform a test login to verify everything is working.

5.4.1 Starting the Web Server – Run via Visual Studio (IIS Express or Kestrel)

1. **Start the application:** In Visual Studio, press **F5** (Start Debugging) or click the green **"Start"** button. This will compile (if there were changes) and launch the web application. By default,

Visual Studio will use the profile specified in **launchSettings.json** – typically either **IIS Express** or **Kestrel**. Either is fine for local testing:

- If using **IIS Express**, Visual Studio will launch the app under the IIS Express web server (you'll see an IIS Express icon in the system tray).
- If using **Kestrel** (the built-in .NET web server), a console window may open showing the application logs.

2. **Trust development certificate:** If this is the first time running an ASP.NET Core app on HTTPS, you might see a prompt about trusting the **developer SSL certificate**. Accept/approve this prompt so that your browser trusts the HTTPS connection. (Visual Studio might prompt to install a developer certificate. You can also manually run dotnet dev-certs https --trust in a command prompt to trust the local HTTPS certificate (ASP.NET Core HTTPS development certificate on Windows · nahid farrokhi).)
3. **Firewall prompts:** On first run, Windows may ask to allow the application through the firewall (especially for Kestrel). It's usually safe to allow private network access so that your browser can connect to the local web server. This doesn't expose the app publicly.
4. **Application startup:** Once running, Visual Studio will open your default web browser to the application's URL. You should see the site's homepage or login page load. The URL will typically be something like:
   - For IIS Express: http://localhost:<port>/ or https://localhost:<port>/ (a random port is assigned by IIS Express, shown in the address bar).
   - For Kestrel: the default URLs are http://localhost:5000 and https://localhost:5001 (unless the project was configured differently). Check the console output or launchSettings for the exact port.
5. **If the browser doesn't open automatically**, you can manually open one and navigate to the appropriate localhost URL as noted in the output/logs.

At this point, the web server is running and the application is live on your machine.

5.4.2 Access and Initial Login – Using the Application

With the application running, follow these steps to access it and perform the initial login:

1. **Navigate to the site:** If not already open, navigate to the application's URL in a web browser (e.g., **Chrome**, **Edge**, or Firefox). Use the https://localhost:<port>/ address shown by Visual Studio. You should see the web application's start page. In many course registration systems, if you are not logged in, it will redirect to a **Login page**.
2. **Login with credentials:** On the Login page, enter the **username/email and password** for an account in the system:
   - If a **default admin account** was provided (or automatically seeded), use those credentials. *(For example, the documentation might specify something like username: admin@example.com and password: Admin@123 – replace with the actual credentials given for the default admin.)*

- If you created an admin user in step 5.3.4, use the email/username and password you set for that user.
- If the system differentiates roles (admin, student, etc.), you can also try logging in as a student or other role if those accounts exist.

3. **Successful login:** After submitting the login form, you should be redirected to the application's post-login landing page (for example, an **Admin Dashboard** or a Student home page depending on the account). Seeing this page indicates that authentication succeeded and the system is connected to the database (since it likely fetched user info and possibly other data).
4. **Default credentials (if login fails):** If you cannot log in (e.g., "invalid username or password"), double-check the credentials or create a new user if possible. For admin access, you may need to use SSMS to set a known password for an account or consult the project team for a default admin password. Ensure that the password is entered correctly and that the database has the user record (you can verify in the Users table via SSMS).

5.4.3 Verification – Checking Key Pages (Admin Dashboard, etc.)

After logging in, it's important to verify that the system is fully functional. Perform the following verifications:

1. **Load the Admin Dashboard:** If logged in as an administrator, navigate to the **Admin Dashboard** or home page for admins. This page might show an overview (counts of users, courses, etc.). Verify that it loads without errors. This confirms that the application can read from the database and render data.
2. **Check data display:** Navigate to a section of the app that should display data from the database. For example:
   - **Manage Users** or **Student List:** See if the list of users or students loads (it might be empty if none exist aside from your admin).
   - **Courses or Departments:** If you seeded courses/departments, check the page that lists courses to ensure they are displayed. If none are in the DB, use the app's feature to add a new course or department and ensure it saves and appears in the list.
   - **Student Registration page:** If logged in as a student, try to view available courses.
3. **Perform a sample action:** To further verify, you can perform a simple create/update action:
   - For example, as admin, try creating a new Department or Course through the UI forms. Save it and see if it appears in the list.
   - As a student, if applicable, attempt to register for a course and see if it succeeds (this might require an open registration period or some initial data setup).
   - Ensure no error messages pop up during these actions.
4. **Email/Notification (if applicable):** If the system is supposed to send emails or notifications, those might not function in a local setup without additional configuration (like SMTP

settings). It's normal for email features to be disabled or require extra setup – this can be verified later and is usually not critical for initial installation verification.

5. **Review logs:** Optionally, check the application console or Visual Studio **Output** (ASP.NET Core Web Server output) for any error messages. If everything is set up correctly, you should not see exceptions. Any exceptions or errors should be investigated (see troubleshooting below).

By completing these steps, you verify that the Course Registration System is installed correctly, can connect to the database, and its core features (login and data display) are working as expected.

5.5 Troubleshooting & FAQs

Even with careful setup, you might encounter some issues. Below are common problems and their solutions:

5.5.1 Connection Issues – Database Unreachable or SQL Login Problems

- **Cannot connect to database / Login failed:** If the application throws errors related to the database connection (e.g., "Cannot open database" or "Login failed for user"):
    - **Check SQL Server service:** Ensure the SQL Server instance is running. You can verify in *Services* or *SQL Server Configuration Manager*.
    - **Verify connection string:** Double-check the server name and instance in your connection string. For a default instance, use (local) or .. For a named instance, the name must match (e.g., .\SQLEXPRESS). A wrong instance name will prevent connection (SQL Server Management Studio unable to connect to local instance - Server Fault).
    - **Authentication mode:** If using Windows Authentication (Trusted_Connection), make sure you're running the application as a Windows user that has access to SQL Server. Usually, your account is an admin on the local SQL by default if you installed it. If using SQL Authentication, ensure that mode is enabled on the server (SQL Server Management Studio unable to connect to local instance - Server Fault) and the user exists with the correct password. By default, the sa user is disabled on new SQL Server installs (SQL Server Management Studio unable to connect to local instance - Server Fault) – you must enable it and set a password in SSMS if you want to use it, or create a new SQL login and give it proper permissions on the database.
    - **Database name mismatch:** If you see an error like "*Database 'XYZ' not found*", it means the database name in the connection string doesn't exist on the server. Ensure you created the database with that name, or update the name in appsettings.json to the actual database name.
    - **Firewall or Network:** For local installations, firewall issues are rare since the app and DB are on the same machine (they often communicate via shared memory or localhost). However, if you installed SQL Server as a named instance, the SQL Server Browser service should be running to resolve the instance name. Ensure **SQL Server Browser** is running (for named instances) (SQL Server Management Studio unable to

connect to local instance - Server Fault). If connecting using TCP (e.g., from another machine or using 127.0.0.1), ensure that TCP/IP protocol is enabled for SQL Server and port 1433 (default) is open in firewall.

- **Timeout errors:** If the app is timing out trying to reach the DB, it's likely the server name or instance is wrong, or the SQL service is stopped. Use SSMS to confirm you can connect with the same server name. If SSMS can connect but the app cannot, the issue is likely in the connection string format or the credentials.
- **SQL Exception details:** To get more info, run the app in Debug mode. When the error occurs, Visual Studio will break (if the exception isn't caught). Examine the exception message in the Output or Exception Helper – it often indicates the cause (e.g., network path not found, login failed, etc.). Adjust settings accordingly as described above.

5.5.2 Migration Errors – Issues during EF Migrations or SQL Script Execution

- **EF Core migration command fails:** If running Update-Database in Package Manager Console throws an error, check the message:
  - "*No executable found matching command 'dotnet-ef'*": This means the EF Core tools are not installed. Install the tools by running dotnet tool install --global dotnet-ef or ensure the project has the Microsoft.EntityFrameworkCore.Design package. Alternatively, use the PMC in Visual Studio (which doesn't require the global tool if the package is present).
  - "*Unable to connect to database*" during migration: This indicates your connection string in the project is likely incorrect or the SQL Server is not accessible. Ensure appsettings is properly configured (the PMC uses the connection from your appsettings by default) and that SQL is running.
  - "*There are pending model changes...*" or "*Migrations have not been applied.*": This might appear if you run the app without applying migrations. The solution is to run Update-Database as in 5.3.1 to apply them.
- **SQL script execution errors:** If you executed an SQL script and got errors:
  - "*Object already exists*" errors: This means you might have run the script twice, or the database already had some tables. If the script is not idempotent, drop the partially created tables or use a fresh database, then run the script again. Alternatively, if using migrations, let EF handle creating tables to avoid conflicts.
  - **Foreign key or dependency errors:** If the script execution order is wrong (e.g., trying to insert data that references a table that isn't created yet), you may need to split or reorder the script. Ensure all CREATE TABLE statements execute before INSERTs that depend on them.
  - **Permissions:** If the script fails due to permissions, ensure your SQL login has rights to create tables or that you are using an account with db_owner privileges on the database. For local dev, using a sysadmin account (your Windows account or sa) is simplest.

- **Migrating existing data:** If you already had a previous version of the database and are applying an update migration, be cautious. It's wise to **backup the database** before running new migrations in case something goes wrong. You can back up by right-clicking the database in SSMS, choose **Tasks → Back Up...** (save a .bak file) as a precaution.
- **No migrations applied (empty database):** If you start the app and the database is empty (no tables), it likely means migrations were not run. You'll get errors whenever the app tries to access a table. Running the migrations or using the provided script (as described in installation) will fix this. In development, you can also enable automatic migration on startup (some projects call Database.Migrate() in code). If that's implemented, the app might create the schema on the fly – check the logs for any migration application on startup.

5.5.3 Server Errors on Launch – Runtime or HTTP 500 Errors

- **White screen or HTTP 500 error on browser:** If you start the app and the browser shows a blank page or a generic "500 Internal Server Error":
    - Run the app in **Development mode** to get detailed error information. In Visual Studio, by default, the ASPNETCORE_ENVIRONMENT is set to Development, which should show detailed error pages. If not, check launchSettings.json to ensure environment is Development when running via VS. Detailed error pages will describe the exception.
    - A common startup error is misconfiguration in appsettings. For instance, a malformed connection string JSON could cause the app to fail at startup. Check that your appsettings.json has valid JSON syntax (no missing commas or braces).
    - **Database connection failure on startup:** If the app tries to connect to the DB at startup (e.g., to run migrations or seed data) and fails, it could crash with an error. In such cases, you'd see an exception in the output (e.g., SqlException). Solve this by fixing the connection string or ensuring the DB is up (see 5.5.1).
    - **Missing dependencies:** If an assembly or NuGet package is missing, the error might mention a file not found. Try cleaning and rebuilding the solution. Ensure all NuGet packages are installed. In some cases, deleting the bin and obj folders and rebuilding can help clear out any old references.
    - **HTTP 500 after login or on certain page:** If the error happens after navigating to a specific page, that could indicate a bug in that page's code (null reference, etc.) or a missing data issue. Check the stack trace on the error page to identify which part of the code caused it. Then fix the underlying issue (could be code expecting certain data to exist – ensure your initial data covers it).
    - **Logging:** Check the application logs. ASP.NET Core might log errors to the console or debug output. Look for any exception messages around the time of failure.
- **Application doesn't start at all:** If Visual Studio shows that the app started and immediately exited (and the browser never opened):
    - There might be an unhandled exception during startup (before the server starts listening). Run without debugging (Ctrl+F5) to see if any console window stays open

with error info. Or check Windows Event Viewer under Application logs for .NET errors.

- Also ensure you didn't accidentally set the project to a console app or something – it should be a web project. Double-check by creating a new ASP.NET Core project to see if that runs, to isolate if it's an environment issue or project-specific.

- **Port conflicts:** If you have another service using the same port that the application wants to use, the app might fail to start. You would see an error about inability to bind to the port. Solve by changing the port in launchSettings or stopping the other service. (For example, if IIS or another instance is using port 5000/5001, pick a different port or use IIS Express which often uses a dynamic port.)

5.5.4 Browser Issues – Mixed Content, SSL, or Loading Problems

- **Browser says "Your connection is not private" or SSL warning:** This is due to the self-signed development certificate used by ASP.NET Core for HTTPS. You have two choices:

  1. Use the **HTTP URL** for local testing (e.g., http://localhost:5000 instead of the https link), if the project allows it. In development, most templates allow both HTTP and HTTPS.

  2. Trust the developer certificate on your machine (recommended). We already mentioned running dotnet dev-certs https --trust (ASP.NET Core HTTPS development certificate on Windows · nahid farrokhi) – if you didn't do that earlier, run it now in a Developer Command Prompt. It will ask for confirmation and then add the certificate to your trusted store. After that, refresh the browser and the HTTPS warning should be gone. If using IIS Express, trusting the IIS Express certificate (which Visual Studio usually sets up) achieves the same result.

- **Mixed content blocked:** If parts of the site (like images or scripts) are not loading and the browser console shows "blocked due to mixed content", it means those resources were requested over HTTP while the page is HTTPS. In a local setup, this could happen if, for example, the site pulls an image from an external http:// URL or you manually navigated to the HTTP version of the site which then loaded something via HTTPS (or vice versa).

  - To fix mixed content issues, ensure you use a consistent scheme. In development, you can use HTTP for everything or HTTPS for everything. If the site's links are configured correctly (they usually are in templates), this shouldn't happen. But if you added custom links, adjust them to use relative URLs or the correct protocol.

  - You can also allow mixed content in browser settings for testing, but it's better to fix the URLs. Modern browsers default to block or warn about mixed content to ensure security.

- **Browser caching issues:** After deployments or code changes, if you find the browser is showing old data or you made a change in code but don't see it, try clearing the cache or doing a hard refresh (Ctrl+F5). Some static files might be cached.

- **UI not loading properly:** If CSS or JS files 404 (check browser developer console for missing file errors), it might be due to incorrect static file paths or a publishing issue. In development, ensure static files are served (ASP.NET Core enables this by default for wwwroot). If you moved files, update references in the layout pages.
- **Different browser behaviors:** Generally, Chrome, Edge, and Firefox should all work with modern ASP.NET Core apps. If something looks off in one browser, test in another to see if it's a browser-specific quirk or a local issue (like cached files or certificate store differences).

If browser issues persist beyond these common scenarios, consider checking the developer console (F12) for errors, and address them based on the messages shown.

5.6 Upgrade and Maintenance

After the initial installation, you should plan for how to handle updates and potential deployment to production environments (such as AWS) in the future:

- **Applying Project Updates:** When new versions of the Course Registration System code are released (or if your team makes changes), you'll need to update your local environment:
    - **Pulling Updates:** If using source control, pull the latest changes from the repository. Rebuild the solution to incorporate any code changes.
    - **Database Migrations:** If the update includes database schema changes (new migration files or updated SQL scripts), apply them to your local database. **Always backup your database before applying updates in production.** For local, it's also wise to take a backup or at least ensure you have the original state saved. Use the Update-Database command for new EF Core migrations, or run any new SQL scripts provided. Confirm that the application still runs and the data is intact after migrations.
    - **Configuration changes:** Check if the appsettings.json or other config files have changed with the update. You may need to reapply your local settings (like connection strings or app secrets) to the new config files if they were overwritten or modified in the update.
- **Regular Backups:** For maintenance, regularly **backup your database** especially before any significant changes. In SQL Server, you can create a .bak backup file (Tasks → Back Up in SSMS) (7.3 Backing Up Databases Before Migration). This ensures that if an update or migration goes wrong, you can restore your data.
- **Monitoring and Logs:** Keep an eye on application logs for errors, especially after updates. ASP.NET Core logging can be configured in appsettings. In development, you see logs in the console or VS output; in production, you might configure logging to files or a monitoring system.
- **Performance considerations:** As you use the system, if you encounter performance issues, you might consider tuning the SQL database (e.g., adding indexes) or adjusting the application (caching, etc.). On a local dev machine, performance issues might be due to limited resources, so ensure your machine meets the recommended specs (see 5.1).
- **Future Deployment to AWS:** If you plan to deploy the system to an AWS environment:

- **Database on AWS:** Consider using **Amazon RDS for SQL Server** to host the database in the cloud. Before migration, backup your local database and restore it to the RDS instance. Update the connection string in the deployed application to point to the RDS endpoint. Remember to also update authentication (AWS RDS might use SQL Authentication; ensure the username/password are set accordingly in the connection string).
- **Application on AWS:** You have multiple options. One straightforward path is to use **AWS Elastic Beanstalk** for an ASP.NET Core application. Elastic Beanstalk can handle deployment either to a Windows IIS environment or a Docker container. You would publish your .NET application (for example, produce a self-contained deployment or a Docker image) and upload it to Elastic Beanstalk. It will provision the necessary infrastructure (EC2 instances, load balancer, etc.). Alternatively, you can manually set up an **EC2 Windows Server** instance with IIS and deploy the application there, or use **AWS App Runner / Amazon ECS** with a containerized .NET app.
- **Configuration for AWS:** In an AWS deployment, you'll likely need to adjust configurations:
  - Store sensitive settings (like DB connection strings) in a secure manner (AWS Parameter Store or Secrets Manager, rather than in plain appsettings deployed).
  - Enable production settings for ASP.NET (e.g., set ASPNETCORE_ENVIRONMENT to Production).
  - Ensure CORS or URL settings are updated if the domain changes.
  - Set up an SMTP service for email if the application sends emails (AWS SES could be used).
- **SSL in production:** You'd use a real SSL certificate (e.g., via AWS Certificate Manager and a Load Balancer or configure IIS with the cert) rather than the dev cert.
- **Scaling and Monitoring:** If the system will be used by many users, plan to monitor performance (CloudWatch on AWS can help) and scale the infrastructure as needed (Elastic Beanstalk can auto-scale instances, etc.).

- **Maintaining the Codebase:** Keep the development environment updated:
  - Update Visual Studio 2022 periodically and the .NET SDK to get the latest fixes and features (especially if the project moves to newer .NET versions over time).
  - Keep dependencies (NuGet packages) updated, but be cautious to test after updates in case of breaking changes.
  - Continue to refer to best practices for ASP.NET Core and MS SQL Server as the project grows (for security patches, performance improvements, etc.).

By following the above guidelines, you'll ensure the Course Registration System remains up-to-date, reliable, and ready for a production deployment when needed. Before any major change (like

moving to AWS or applying a major update), always plan, backup, and test thoroughly. Then, enjoy the streamlined process of course registrations in your system!

5.7 Project Repository and Demonstration Video

For source code access and to observe the system in operation, the following resources are provided:

- **GitHub Repository**
  The full source code for the Course Registration System, including the ASP.NET Core MVC application, database context, and configuration files, is available on GitHub. This repository allows developers and contributors to view, clone, or fork the project for further development or deployment.
  URL: https://github.com/Jenarththan2001/FOE_CourseRegistrationSystem

- **System Demonstration Video**
  A comprehensive video demonstration has been prepared to showcase the functionality of the Course Registration System. The video includes login flows, student registration, advisor and admin approvals, and other core features. This demonstration serves as a visual reference for system behavior during runtime.
  Video Link:https://drive.google.com/file/d/17X55-fXRYbFGx7phcjz0WZI1cNXuISgS/view

These resources are intended to support developers, testers, and evaluators in understanding and verifying the implementation of the Course Registration System.

6. Reference Details

1. IEEE Std 1016-2009, *IEEE Standard for Information Technology – Systems Design – Software Design Descriptions*, IEEE, 2009.
2. IEEE Std 829-1998, *IEEE Standard for Software Test Documentation*, IEEE, 1998.
3. Clements, P., Kazman, R., & Klein, M., *Documenting Software Architectures: Views and Beyond*, 2nd ed., Addison-Wesley, 2010.
4. OpenAPI Initiative, *OpenAPI Specification v3.0*, 2017. [Online]. Available: https://swagger.io/specification/
5. Microsoft Docs, *ASP.NET Core MVC Documentation*, Microsoft, 2025. [Online]. Available: https://learn.microsoft.com/aspnet/core
6. SmartBear, *SwaggerHub Documentation*, 2024. [Online]. Available: https://support.smartbear.com/swaggerhub/
7. Mozilla Developer Network, *HTML & JavaScript DOM Documentation*, Mozilla, 2024.
8. Google Developers, *Chrome Lighthouse Audits*, [Online]. Available: https://developer.chrome.com/docs/lighthouse/
9. OWASP Foundation, *OWASP ZAP – Zed Attack Proxy*, [Online]. Available: https://owasp.org/www-project-zap/
10. SQLMap Developers, *SQLMap: Automatic SQL Injection Tool*, [Online]. Available: https://sqlmap.org/
11. Apache Software Foundation, *Apache JMeter 5.6.3 Documentation*, [Online]. Available: https://jmeter.apache.org/
12. Microsoft Learn, *Entity Framework Core Migrations Guide*, [Online]. Available: https://learn.microsoft.com/ef/core/managing-schemas/migrations/
13. GitHub Docs, *Using Git to Clone and Manage Repositories*, GitHub, 2024.
14. Wikipedia, *Software Design Description (IEEE 1016)*. [Online]. Available: https://en.wikipedia.org/wiki/Software_design_description
15. Wikipedia, *Software Test Documentation (IEEE 829)*. [Online]. Available: https://en.wikipedia.org/wiki/Software_test_documentation
16. Visual Studio Docs, *Installing and Managing Workloads*, Microsoft, 2024.
17. PostgreSQL Docs, *SQL Injection Prevention Guidelines*, PostgreSQL Global Development Group, 2023.
18. Microsoft Security Best Practices, *Password Hashing and Identity Framework*, Microsoft, 2024.
19. Stack Overflow, *EF Core Parameterized Queries Against SQL Injection*, [Online]. Available: https://stackoverflow.com

20. Chrome DevTools, *Device Mode for Responsive Testing*, [Online]. Available: https://developer.chrome.com/docs/devtools/device-mode/
21. Selenium HQ, *Selenium WebDriver Documentation*, [Online]. Available: https://www.selenium.dev/documentation/
22. xUnit.net, *xUnit Testing Framework Guide*, [Online]. Available: https://xunit.net/docs/
23. Postman Labs, *Postman API Testing Tool Guide*, [Online]. Available: https://www.postman.com/
24. Visual Studio Marketplace, *NuGet Package Manager for EF Migrations*, [Online]. Available: https://marketplace.visualstudio.com
25. Razor View Engine Docs, *ASP.NET Razor Pages*, Microsoft, 2024.
26. Microsoft Docs, *Role-based Authorization in ASP.NET Core*, [Online]. Available: https://learn.microsoft.com/aspnet/core/security/authorization/roles
27. JUnit Docs, *Automated Unit Testing Principles*, Oracle, 2024.
28. Entity Framework Core Docs, *DbContext and Data Seeding*, Microsoft, 2024.
29. System.Text.Json Docs, *Working with JSON in ASP.NET Core*, Microsoft, 2024.