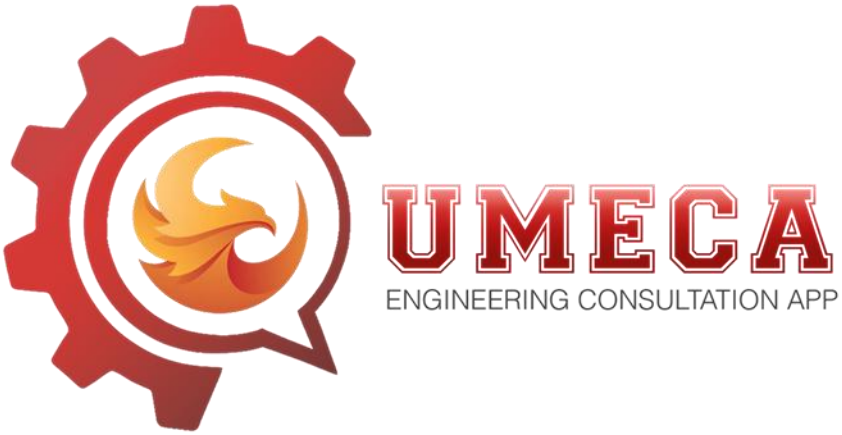
DESKTOP PROGRAMMERS MANUAL



University of Mindanao Engineering Consultation App

CPE223/L (7599)

Author: John Lee S. Cuenca

j.cuenca.540732@umindanao.edu.ph

Engr. Jay Al Gallenero

October 10, 2025

B. Table of Contents

C. INTRODUCTION

I. Purpose of The Manual

This programming manual serves as a comprehensive technical reference for the University of Mindanao Engineering Consultation App (UMECA) . It provides detailed documentation of the system's architecture, codebase structure, and implementation guidelines. The manual is designed to facilitate:

- Understanding of the system's technical architecture and design patterns
- Onboarding of new development team members
- Maintenance and enhancement of existing features
- Troubleshooting and debugging of system issues
- Consistent coding practices and standards across the development team

This documentation is intended for software developers, system architects, quality assurance engineers, and technical personnel involved in the development, maintenance, and deployment of the consultation management application.

II. System Overview

UMECA (University of Mindanao Engineering Consultation Application) is a comprehensive academic consultation management system designed to streamline and enhance the consultation process between students and faculty members. The desktop application serves as a core interface for faculty and administrators, providing efficient access to consultation data, scheduling, and management functions in an offline-capable and stable environment.

The system implements a multi-layered architecture with clear separation of concerns following Domain-Driven Design (DDD) and MVVM (Model-View-ViewModel) principles. This structure promotes modularity, scalability, and maintainability across both desktop and mobile platforms.

Core Features:

- Secure user authentication and role-based access control for students, faculty, and administrators
- Consultation request management and faculty scheduling tools for managing academic sessions
- Administrative dashboard for system oversight and user monitoring
- Consultation history tracking, reporting, and analytics
- Real-time notifications and status updates for consultations
- User management with encrypted credentials and data protection mechanisms

The desktop UMECA is structured into several distinct projects representing key architectural layers:

- **Consultation.App** – Acts as the presentation layer, implementing the MVVM pattern for the desktop environment. It handles user interactions, interface logic, and visualization of consultation data.
- **Consultation.BackEndCRUD** – Manages core data operations (Create, Read, Update, Delete) and acts as the service layer, ensuring business rules and data validation are properly applied.
- **Consultation.Domain** – Defines the core domain models and business logic, representing entities such as consultations, users, and schedules.
- **Consultation.Infrastructure** – Serves as the data access layer, managing database connections and persistence operations through repository patterns.
- **Consultation.Desktop.Test** – Contains testing and debugging tools to ensure system reliability and performance in desktop deployment.
- **Enum** – Provides enumerations and shared constants used throughout the application to maintain consistent value definitions across modules.

The desktop application connects seamlessly with the institution's central database (Azure MySQL) and shares common domain logic with the mobile version, ensuring consistency across platforms. This modular structure allows the desktop version to operate independently while remaining fully integrated into the overall UMECA ecosystem.

III. Technologies Used

i. LANGUAGES

- **C# 10.0**: Primary programming language for all application layers
- **SQL**: Database query language for data manipulation and retrieval
- **XAML**: Markup language for WPF user interface definition (if applicable)

ii. FRAMEWORKS

- **NET 8.0**: Core application framework
- **Entity Framework Core**: Object-Relational Mapping (ORM) framework for database operations
- **Microsoft.AspNetCore.Identity.EntityFrameworkCore 8.0.15:** Authentication framework
- **Entity Framework Core SQL Server 9.0.9: SQL Server provider** In-memory database for testing

- **Pomelo.EntityFrameworkCore.MySql 9.0.0**: MySQL provider
- **Windows Presentation Foundation (WPF)** or **Windows Forms**: Desktop application framework (based on Form inheritance observed)
- **xUnit/NUnit**: Unit testing framework for automated testing

iii. DBMS

- **Microsoft SQL Server (LocalDB)**: Primary database management system
- **Connection String Configuration**:
  - Data Source: (localdb)\\MSSQLLocalDB
  - Initial Catalog: ConsultationDatabase
  - Integrated Security: True
  - Connection Timeout: 30 seconds
  - Encrypt: False
  - Trust Server Certificate: False
  - Application Intent: ReadWrite
  - Multi Subnet Failover: False

iv. TOOLS

- **Visual Studio 2022**: Integrated Development Environment (IDE)
- **Git**: Version control system
- **GitHub**: Remote repository hosting service (https://github.com/tortolsaur/proto-sd-project.git)
- **Entity Framework Core Migrations**: Database schema versioning and management
- **NuGet Package Manager**: Dependency management
- **SQL Server Management Studio (SSMS)**: Database administration and query tool

D. SYSTEM ARCHITECTURE

I. Overview Of Software Architecture

The **UMECA** is designed using a **layered architecture**, pattern combined with **Domain-Driven Design (DDD)** principles, ensuring clear separation of concerns and maintainability. Each project within the solution represents a distinct layer or module, handling a specific responsibility in the overall system workflow.

The solution is composed of the following main components:

1. **Presentation Layer (Consultation.App)**
   This serves as the **application layer** or **entry point** of the system, handling user interactions and coordinating between the user interface and backend services. It communicates with the backend logic to execute user requests and display results.
   - Implements Model-View-Presenter (MVP) or Model-View-ViewModel (MVVM) pattern
   - Contains Views (Forms) such as Login.cs, implementing interfaces like ILoginView
   - Manages user interface components including TextBox controls (txtEmail, txtPassword), Labels (resultLabel1, ErrorPassLabel)
   - Handles user input validation and event-driven interactions through EventHandler delegates (e.g., LoginEvent)
   - Implements View interfaces for loose coupling between presentation andbusiness logic

2. **Application/Service Layer (Consultation.BackEndCRUD)**
   This module manages the **core data operations**: Create, Read, Update, and Delete (CRUD). It acts as the service layer, ensuring that business rules are properly applied before interacting with the database or domain entities.
   - Contains business logic and application services
   - Includes AuthService for authentication operations with dependency injection of AppDbContext
   - Implements service interfaces and their concrete implementations
   - Manages ViewModels such as EditConsultationViewModel with properties:
     - studentName: string
     - facultyName: string
     - courseCode: string
     - studentUMID: string
     - concernDescription: string
     - dateSchedule: DateTime

3. **Consultation.Domain**

   The domain layer defines the **core business logic and entities** of the application. It contains the object models, validation rules, and core functionalities that represent the heart of the system's business process.

   - Contains domain entities and business rules
   - Implements entity classes such as Admin with:
     - Primary key: AdminID (int)
     - Properties: AdminName (string), UsersID (string)
     - Navigation properties: Users collection
     - Data annotations: [Key], [ForeignKey]
   - Includes enumerations such as DaysOfWeek:
     - Monday = 1, Tuesday = 2, Wednesday = 3, Thursday = 4, Friday = 5, Saturday = 6, Sunday = 7
   - Defines domain models for:
     - Students, Faculty, Departments, EnrolledCourses
     - Consultations, Bulletins, Notifications
     - Programs, SchoolYears, Semesters, UserTypes, Status

4. **Consultation.Infrastructure**

   This serves as the **data access and persistence layer**. It manages connections to the database, repositories, and external data sources. It ensures that data is stored, retrieved, and updated efficiently and securely.

   - Implements data access using Entity Framework Core
   - Contains AppDbContext for database context management
   - Manages database migrations with timestamp-based naming (e.g., 20250712134553_addingmigration)
   - Implements repository pattern for data operations
   - Handles database seeding with UpdateData operations for AspNetUsers table
   - Stores hashed passwords using secure hashing algorithms

5. **Consultation.Desktop.Test**

   This project is primarily used for **testing and validation** of system components in a desktop environment. It ensures that each function, particularly within the backend, operates correctly and meets expected performance.

   - Implements unit tests using test fixtures with [TestFixture] attribute
   - Contains test classes such as AuthServiceTests with:
     - Private fields: _context (AppDbContext), _authService (AuthService)
     - Setup methods decorated with [SetUp] attribute
     - Test methods for service validation including:
       - Setup(): void
       - Login_WithExistingUser_ReturnsUser
       - TearDown(): void
   - DbContextOptionsBuilder configuration for in-memory testing

6. **Enum**

   This project stores **enumeration types and constants** that are shared across different modules. It promotes consistency and reusability by providing a centralized definition of fixed values used throughout the system.

Consultation.App --> Consultation.BackEndCRUD --> Consultation.Infrastructure

↑

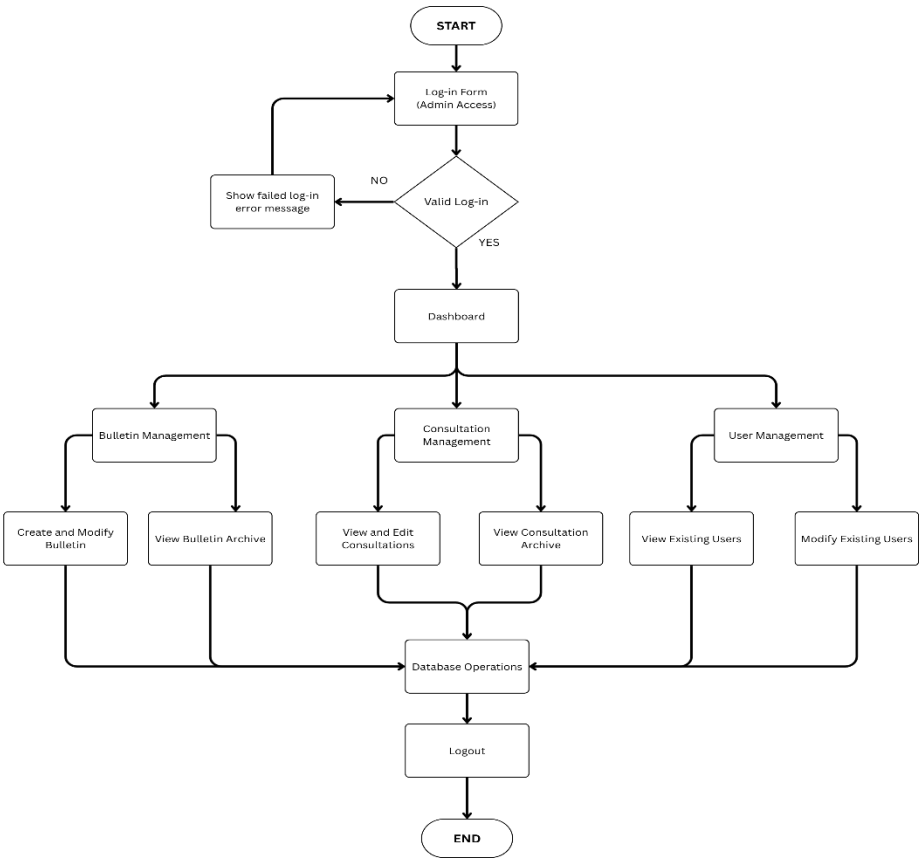
Consultation.Domain

Enum (shared by all)

Architectural Patterns:

- **Dependency Injection**: Used throughout for loose coupling
- **Repository Pattern**: Abstracts data access operations
- **Unit of Work**: Manages database transactions
- **DTO/ViewModel Pattern**: Separates domain models from presentation models

II. Flow Diagram



The flowchart shows the admin system process starting from the login form. If the login fails, an error message appears; if successful, the admin accesses the dashboard. From there, the admin can manage bulletins, consultations, and users each allowing actions like creating, viewing, or editing records. All operations involve database updates, and the process ends with logout.

AUTHENTICATION FLOW:
User Input → Login View → LoginEvent → AuthService.Login() →
AppDbContext Query → Database Validation → Return User Object →
Update UI/Navigate to Dashboard

CONSULTATION MANAGEMENT FLOW:
User Request → Consultation View → Service Layer →
Domain Validation → Repository Pattern → AppDbContext →
Database Transaction → Migration Support → Data Persistence

E. INSTALLATION INSTRUCTIONS

I. Environment Setup

System Requirements:

- Operating System: Windows 10/11 (64-bit)
- Processor: Intel Core i5 or equivalent (minimum), Intel Core i7 or better (recommended)
- RAM: 8 GB minimum, 16 GB recommended
- Storage: 10 GB free disk space for IDE, SDK, and project files
- Internet Connection: Required for package restoration and initial setup

Development Environment Prerequisites:

1. Install Visual Studio 2022 (Community, Professional, or Enterprise Edition)
2. Install .NET 6.0 SDK or later from https://dotnet.microsoft.com/download
3. Install SQL Server LocalDB (included with Visual Studio or SQL Server Express)
4. Install Git for Windows from https://git-scm.com/download/win
5. Configure Git credentials for GitHub access

II. Software Requirements

Required Software Components:

1. **Visual Studio 2022**
   - Workload: ASP.NET and web development
   - Workload: .NET desktop development
   - Individual Component: SQL Server Express LocalDB
   - Individual Component: Entity Framework 6 tools
2. **.NET SDK 6.0 or Higher**
   - Runtime: ASP.NET Core Runtime
   - Runtime: .NET Desktop Runtime
3. **SQL Server LocalDB**
   - Instance Name: MSSQLLocalDB
   - Database Engine Services
4. **Git Version Control**
   - Git Bash
   - Git Credential Manager
5. **NuGet Package Manager**
   - Entity Framework Core (6.x or 7.x)
   - Entity Framework Core Tools
   - Entity Framework Core Design
   - Microsoft.AspNetCore.Identity.EntityFrameworkCore
   - Testing frameworks (xUnit/NUnit)

III. Step-By-Step Installation Process

Step 1: Clone the Repository

https://github.com/tortolsaur/proto-sd-project.git

Step 2: Open Solution in Visual Studio

1. Launch Visual Studio 2022
2. Select "Open a project or solution"
3. Navigate to the cloned repository folder
4. Open the solution file: proto-sd-project.sln
5. Wait for Visual Studio to load all projects and restore NuGet packages

Step 3: Restore NuGet Packages

Step 4: Configure Database Connection

1. Locate appsettings.json or connection string configuration file
2. Verify the connection string points to LocalDB:

Step 5: Apply Database Migrations

Step 6: Verify Database Creation

1. Open SQL Server Object Explorer in Visual Studio
2. Expand (localdb)\MSSQLLocalDB
3. Verify ConsultationDatabase exists
4. Check tables including:
    - AspNetUsers
    - Admins
    - Students
    - Faculty
    - Consultations
    - Bulletins
    - Departments

Step 8: Run the Application

1. Set Consultation.App as the startup project (right-click → Set as Startup Project)
2. Press F5 or click "Start Debugging"
3. The application should launch and display the Login screen

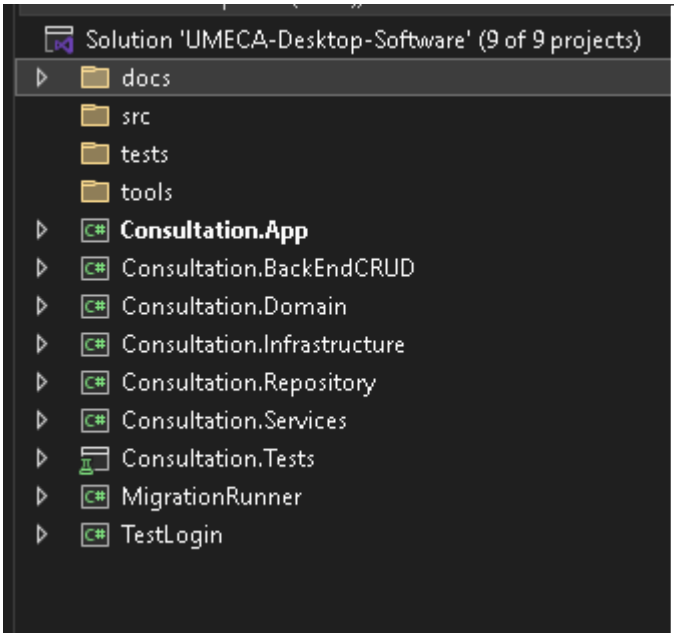
Step 9: Verify Installation

1. Test database connectivity by attempting login
2. Check that all forms load correctly
3. Verify navigation between different views
4. Check error log for any initialization issues

Troubleshooting Installation Issues:

- If NuGet packages fail to restore, clear NuGet cache: dotnet nuget locals all --clear
- If database migration fails, delete the database and run Update-Database again
- If LocalDB is not available, install SQL Server Express with LocalDB feature
- Ensure all project references are correctly loaded (check Solution Explorer for warning icons).

F. SOURCE CODE STRUCTURE



Consultation Management System - Solution Architecture Documentation

Solution Name: UMECA- Desktop-Software

Total Projects: 9 projects

Architecture Pattern: Clean Architecture / Layered Architecture

Solution Structure:

I. Folder Organization:

- **docs**

Contains project documentation, specifications, and technical references for the consultation management system.

- **src**

Houses all source code projects organized by architectural layers and responsibilities.

- **tests**

Contains test projects for unit testing, integration testing, and quality assurance.

- **tools**

Includes utility projects and helper tools supporting the development and deployment process.

II. Core Projects:

- **Consultation.App**

Layer: Application Layer

Purpose: Contains application logic, use cases, and orchestration. Implements CQRS patterns, command/query handlers, and application services that coordinate domain operations.

- **Consultation.BackEndCRUD**

Layer: Presentation/API Layer

Purpose: Provides backend API endpoints for Create, Read, Update, and Delete operations. Serves as the entry point for client applications to interact with the consultation system.

- **Consultation.Domain**

Layer: Domain Layer (Core)

Purpose: Contains core business entities, domain models, and business logic. Represents the heart of the application with entities like Student, Faculty, ConsultationRequest, and domain rules.

- **Consultation.Infrastructure**

Layer: Infrastructure Layer

Purpose: Implements external concerns such as database access, file systems, email services, and third-party integrations. Contains concrete implementations of repository interfaces.

- **Consultation.Repository**

Layer: Data Access Layer

Purpose: Defines repository interfaces and implementations following the Repository Pattern. Provides abstraction over data persistence operations for domain entities.

- **Consultation.Services**

Layer: Domain Services Layer

Purpose: Contains domain services that implement business logic spanning multiple entities or complex operations that don't naturally fit within a single entity.

- **Consultation.Tests**

Layer: Testing Layer

Purpose: Houses unit tests, integration tests, and automated test suites ensuring code quality, business logic validation, and system reliability.

III. Supporting Projects:

- **MigrationRunner**

Purpose: Manages database schema migrations, versioning, and deployment scripts. Ensures database structure stays synchronized with domain model changes across different environments.

- **TestLogin**

Purpose: Provides authentication and login testing utilities. Facilitates testing of user authentication flows, role-based access control, and security features.

IV. Architecture Principles:

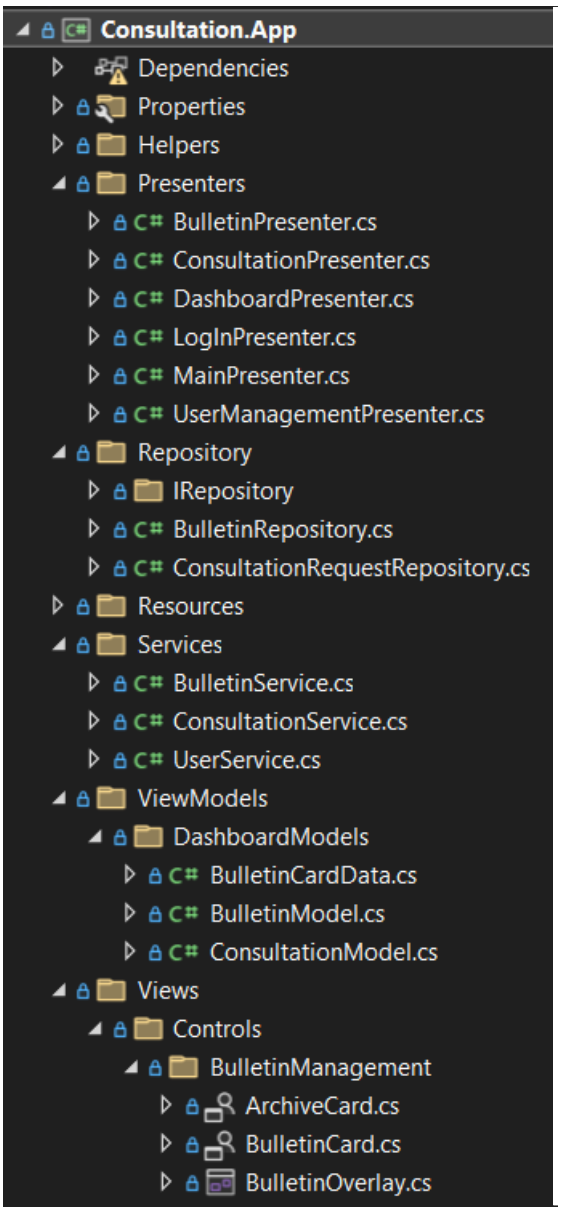
This solution follows Clean Architecture principles:

- Dependency Rule: Dependencies point inward toward the domain core
- Separation of Concerns: Each layer has distinct responsibilities
- Independence: Domain layer is independent of frameworks and external concerns
- Testability: Core business logic can be tested without external dependencies
- Flexibility: Easy to swap implementations without affecting business logic

The architecture ensures maintainability, scalability, and clear separation between business rules and technical implementation details for the university consultation management system.

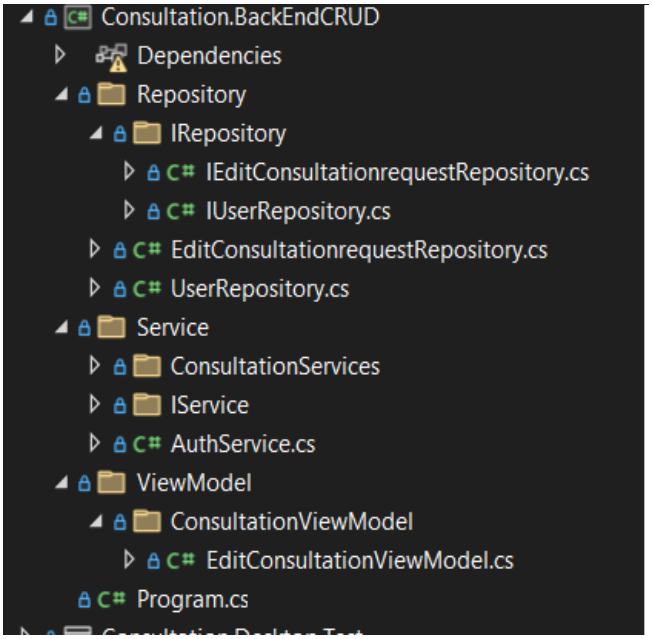
I. Project Folder Layout

The solution follows a multi-project architecture organized as follows:



Project Name: Consultation.App
Architectural Role: Application Layer (Business Logic & Coordination)
Folder Structure:

- **Dependencies**
  Contains project references and NuGet dependencies required by the application.
- **Properties**
  Holds configuration files and assembly information.
- **Helpers**
  Includes utility or support classes used across the project.
- **Presenters**
  Contains classes responsible for handling presentation logic and communication between the view and business layers.
  Files:
  - BulletinPresenter.cs
  - ConsultationPresenter.cs
  - DashboardPresenter.cs
  - LoginPresenter.cs
  - MainPresenter.cs
  - UserManagementPresenter.cs
  - BulletinService.cs
  - ConsultationService.cs
  - UserService.cs
- **ViewModels**
  Manages the data structure and state binding between the views and the presenters.
  Subfolder: DashboardModels
  - BulletinCardData.cs, BulletinModel.cs, ConsultationModel.cs
- **Views**
  Contains the user interface components for the application.
  Subfolder: Controls
  - **BulletinManagement**
    - ArchiveCard.cs
  - BulletinCard.cs, BulletinOverlay.cs, BulletinPublishedreportArea.cs



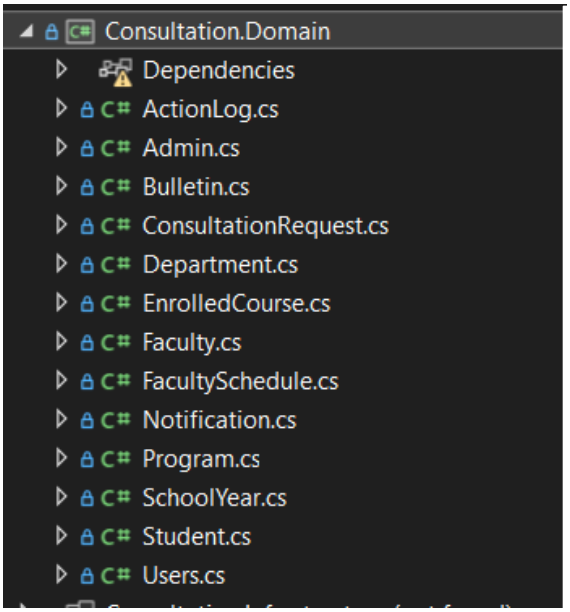
Project Name: Consultation.BackEndCRUD
Architectural Role: Data Access and CRUD Layer (Backend Operations)

Folder Structure:

- **Dependencies**
  Contains external references and NuGet packages required by the backend project.
- **Repository**
  This folder manages all data-related transactions and communication between the application and the data source. It defines interfaces and implementations for data access.
- **Subfolders and Files:**
  - IRepository
    - IEditConsultationrequestRepository.cs
    - IUserRepository.cs
  - Repository Implementations:
    - EditConsultationrequestRepository.cs
    - UserRepository.cs
- **Service**
  Contains the service layer responsible for executing business rules and handling backend logic.
  Files:
  - ConsultationServices.cs
  - AuthService.cs
  - IService.cs
- **ViewModel**
  Handles data transformation and organization between backend services and presentation components.
  Subfolder:
  - ConsultationViewModel
  - EditConsultationViewModel.cs
- **Program.cs**
  Serves as the project's entry point, initializing the backend processes and configurations.

Root Files:

- Program.cs (Main entry point)



Project Name: Consultation.Domain
Architectural Role: Domain Layer (Core Business Entities and Models)

Folder Structure:

- **Dependencies**
  Contains essential references and shared dependencies required by the domain entities.
- **ActionLog.cs**
  Represents system logs that track user or system actions within the application.

• **Admin.cs**
Defines properties and behavior of administrative users who manage consultation-related processes.

• **Bulletin.cs**
Represents the bulletin entity containing announcements, updates, or public notices within the system.

• **ConsultationRequest.cs**
Defines the model for handling consultation requests between students and faculty.

• **Department.cs**
Represents academic departments and their relationships with other entities such as faculty and courses.

• **EnrolledCourse.cs**
Stores data related to student course enrollments for each academic term.

• **Faculty.cs**
Represents faculty member data, including identification, department affiliation, and consultation responsibilities.

• **FacultySchedule.cs**
Manages faculty members' consultation schedules, availability, and assigned time slots.

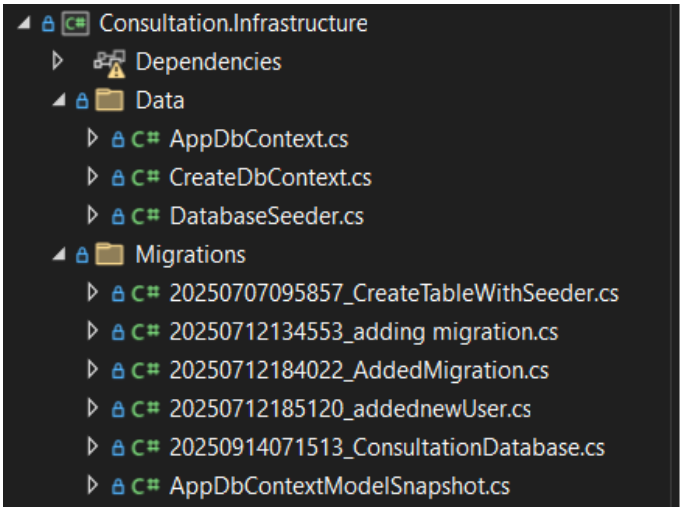
• **Notification.cs**
Defines notifications sent to users, including alerts, consultation updates, and announcements.

• **Program.cs**
Represents academic programs and their structures within the institution.

• **SchoolYear.cs**
Defines academic year data, including start and end terms used across other entities.

• **Student.cs**
Represents student-related data, including profile details, enrolled courses, and consultation requests.

• **Users.cs**
Serves as the base user entity that defines shared properties across all user roles (admin, faculty, student).



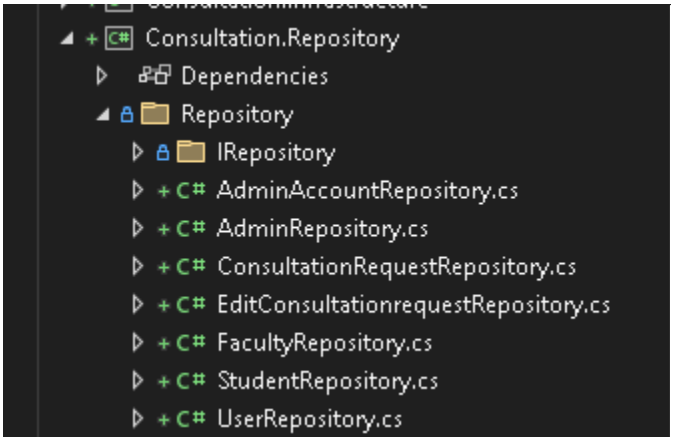
Project Name:
Consultation.Infrastructure
Architectural Role: Infrastructure Layer (Database Configuration and Data Persistence)
Folder Structure:

- **Dependencies**
  Contains external libraries and Entity Framework dependencies required for data access and database operations.

- **Data**
  Handles the configuration and initialization of the database context used throughout the system.
  Files:
  - AppDbContext.cs — Defines the Entity Framework Core context that manages entity sets and database interactions.
  - CreateDbContext.cs — Provides functionality for creating and initializing the database context, particularly for migrations and runtime configuration.
  - DatabaseSeeder.cs — Seeds the database with initial data such as admin users, roles, or test data during setup.
- **Migrations**
  Contains Entity Framework migration files that track and manage database schema changes.
  Files:
  - 20250707095857_CreateTableWithSeeder.cs — Initial migration that creates core database tables and seeds initial data.
  - 20250712134553_adding migration.cs — Migration that adds new fields or modifies existing tables.
  - 20250712184022_AddedMigration.cs — Subsequent schema update, possibly including refinements or new relationships.
  - 20250712185120_addednewUser.cs — Adds user-related tables or fields.
  - 20250914071513_ConsultationDatabase.cs — Major update to include consultation-related tables and relationships.
  - AppDbContextModelSnapshot.cs — Maintains the current database model snapshot used by Entity Framework to track changes.

Root Files:

- All database-related configuration and migration files are contained within the Data and Migrations folders.



Project Name: Consultation.Repository
Architectural Role: Data Access Layer (Repository Pattern Implementation)

Folder Structure:

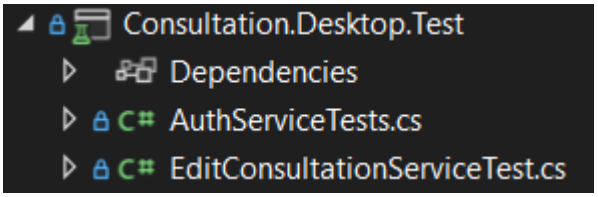
- **Dependencies**
  Contains essential references and shared dependencies required by the repository layer for data access operations.
- **Repository Folder**
  Houses all repository implementations that handle data persistence and retrieval operations.

- **Repository Classes:**
  - **AdminAccountRepository.cs**
    Manages data access operations for administrative account entities, including CRUD operations for admin authentication and account management.
  - **AdminRepository.cs**
    Handles data persistence and retrieval operations for administrator entities, managing admin- specific business data and relationships.
  - **ConsultationRequestRepository.cs**
    Implements data access logic for consultation request entities, handling the storage, retrieval, and manipulation of consultation-related data between students and faculty.
  - **EditConsultationRequestRepository.cs**
    Provides specialized repository methods for editing and updating existing consultation requests, managing modification operations and tracking changes.
  - **FacultyRepository.cs**
    Manages data operations for faculty entities, including faculty profile information, department associations, and consultation-related data persistence.
  - **StudentRepository.cs**
    Handles data access operations for student entities, managing student profiles, enrolled courses, and consultation request history.
  - **UserRepository.cs**
    Implements the base repository for user entities, providing common data access methods shared across all user types (admin, faculty, student).

Architecture Pattern:

This repository layer implements the Repository Pattern, which:

- Abstracts data access logic from business logic
- Provides a collection-like interface for domain entities
- Encapsulates database queries and operations
- Enables unit testing through dependency injection
- Maintains separation of concerns between layers



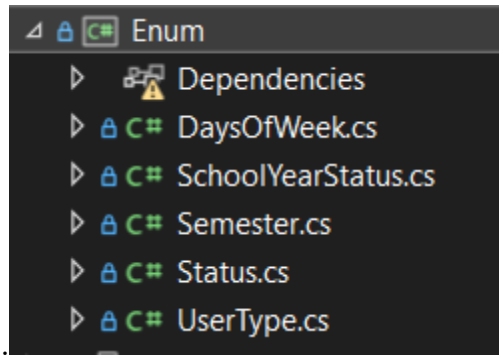
Project Name: Consultation.Desktop.Test
Architectural Role: Testing Layer (Unit and Integration Testing)

Folder Structure:

- **Dependencies**
  Contains project references and testing libraries (e.g., xUnit, NUnit, MSTest) required to execute test cases.
- **AuthServiceTests.cs**
  Contains unit tests designed to validate the functionality, security, and reliability of the AuthService component from the backend or application layer.
- **EditConsultationServiceTest.cs**
  Includes test cases verifying the behavior, accuracy, and robustness of the EditConsultationService logic, ensuring that CRUD and validation operations function correctly.

Root Files:

- AuthServiceTests.cs
- EditConsultationServiceTest.cs



Project Name: Enum
Architectural Role: Enumeration Layer (Constant Definitions and Categorical Values)

Folder Structure:

- **Dependencies**
  Contains references and shared dependencies required by the enumeration files.
- **DaysOfWeek.cs**
  Defines an enumeration representing the days of the week, used for scheduling and consultation availability logic.
- **SchoolYearStatus.cs**
  Contains enumeration values describing the status of an academic school year (e.g., Active, Inactive, Upcoming).
- **Semester.cs**
  Defines the enumeration for academic semesters or terms (e.g., FirstSemester, SecondSemester, Summer).
- **Status.cs**
  Represents a general-purpose enumeration used for marking entity states such as Active, Pending, or Archived.
- **UserType.cs**
  Defines user classification types (e.g., Admin, Faculty, Student), used for role-based access control and logic handling.

Root Files:

- All C# files represent distinct enumerations that categorize and standardize data values used throughout the system

II. Key Packages/Modules/Classes

Presentation Layer (Consultation.App)

- **Login.cs**: Main login form implementing ILoginView interface
    - Properties: useremail, password (getter-only)
    - Events: LoginEvent (EventHandler)
    - Controls: txtEmail, txtPassword, resultLabel1, ErrorPassLabel
- **CSWindowPresenter.cs**: Main window presenter
- **LoginPresenter.cs**: Handles login presentation logic
- **DashboardPresenter.cs**: Dashboard presentation logic
- **BulletinPresenter.cs**: Bulletin board presenter
- **MainPresenter.cs**: Main application presenter
- **UserManagementPresenter.cs**: User management presenter

Service Layer (Consultation.BackEndCRUD)

- **EditConsultationViewModel.cs**: ViewModel for consultation editing
    - Properties: studentName, facultyName, courseCode, studentUMID, concernDescription, dateSchedule
- **AuthService.cs**: Authentication service with dependency injection
    - Dependencies: AppDbContext _context
    - Methods: User authentication and validation
- **IService Interfaces**: Contract definitions for services
- **Repository Pattern**: Abstracts data access operations

Domain Layer (Consultation.Domain)

- **Admin.cs**: Admin domain entity
    - [Key] public int AdminID { get; set; }
    - public string AdminName { get; set; }
    - public Users Users { get; set; } // Navigation property
    - [ForeignKey(nameof(Users))] public string UsersID { get; set; }

DaysOfWeek.cs: Enumeration for days

```
public enum DaysOfWeek
{
Monday =1, Tuesday = 2, Wednesday = 3, Thursday = 4, Friday = 5, Saturday = 6, Sunday =7
}
```

- **Other Entities**: Student, Faculty, Bulletin, ConsultationRequest, Department, EnrolledCourse, FacultySchedule, Notification, Program, SchoolYear, Semester, Status, UserType

Infrastructure Layer (Consultation.Infrastructure)

- **AppDbContext.cs**: Entity Framework Core database context
    - Manages DbSet collections for all entities
    - Configures entity relationships and constraints

- **Migrations**: Database schema versioning
    - addingmigration (20250712134553): Initial migration with AspNetUsers UpdateData
    - Password hashing using secure algorithms
    - Subsequent migrations for schema updates

Testing Layer (Consultation.Desktop.Test)

- **AuthServiceTests.cs**: Unit tests for authentication

Helper Classes (Consultation.App/Helpers)

- **CustomMessageBox.cs:** useful for ensuring a consistent, user-friendly message box style throughout the application.

III. Naming Conventions

General Conventions:

- **PascalCase**: Used for class names, method names, properties, and public members
    - Examples: Admin, EditConsultationViewModel, LoginPresenter, AuthService
- **camelCase**: Used for private fields, local variables, and method parameters
    - Private fields with underscore prefix: _context, _authService
    - Local variables: option, migrationBuilder, studentName

Specific Naming Patterns:

1. **Classes:**
    - Entity classes: Noun form (e.g., Admin, Student, Faculty)
    - Service classes: Noun + "Service" (e.g., AuthService, EditConsultationService)
    - Presenter classes: Noun + "Presenter" (e.g., LoginPresenter, DashboardPresenter)
    - ViewModel classes: Noun + "ViewModel" (e.g., EditConsultationViewModel)
    - Test classes: Class name + "Tests" (e.g., AuthServiceTests)
2. **Interfaces:**
    - Prefix with "I" (e.g., ILoginView, IAuthService, IUserRepository)
3. **Properties:**
    - PascalCase for public properties (e.g., AdminID, AdminName, UsersID)
    - camelCase for ViewModel properties (e.g., studentName, facultyName, courseCode)

4. **Methods:**
    - Verb or verb phrase (e.g., Setup(), Login(), TearDown(), UpdateData())
    - Test methods: MethodName_Scenario_ExpectedResult (e.g., Login_WithExistingUser_ReturnsUser)

5. **Controls (UI Elements):**
    - Prefix with control type abbreviation + description
    - TextBox: txt prefix (e.g., txtEmail, txtPassword)
    - Label: lbl or full word (e.g., ErrorPassLabel, resultLabel1)

6. **Database Tables:**
    - Plural form for entity tables (e.g., AspNetUsers, Students, Consultations)

7. **Migration Files:**
    - Format: YYYYMMDDHHMMSS_DescriptiveName.cs
    - Examples: 20250712134553_addingmigration.cs, 20250712184022_AddedMigration.cs

8. **Enumerations:**
    - PascalCase for enum name (e.g., DaysOfWeek, Status, Semester)
    - PascalCase for enum values (e.g., Monday, Tuesday)

9. **Namespaces:**
    - Follow folder structure: Consultation.App.Views.Login
    - Pattern: ProjectName.LayerName.FeatureName

10. **Event Handlers:**
    - Suffix with "Event" (e.g., LoginEvent)

Code Organization Conventions:

- One class per file
- File name matches class name exactly
- Designer files use .Designer.cs suffix
- Partial classes for form designers
- Region blocks for organizing code sections (implied from standard practices)

G. FUNCTION DOCUMENTATION

I. Method Signatures And Purpose

Presentation Layer (Consultation.App.Views)

```
// Login.cs - ILoginView Implementation
public partial class Login : Form, ILoginView
{
    // Property: Gets the email entered by user
    public string useremail => txtEmail.Text;

    // Property: Gets the password entered by user
    public string password => txtPassword.Text;

    // Event: Triggered when login button is clicked
    public event EventHandler LoginEvent;

    // Constructor: Initializes login form components
    public Login()

    // Method: Handles form initialization (auto-generated)
    private void InitializeComponent()
}
```

Service Layer (Consultation.BackEndCRUD)

```
// AuthService.cs - Authentication Service
public class AuthService
{
    private readonly AppDbContext _context;

    // Constructor: Initializes service with database context
    // Purpose: Sets up dependency injection for database operations
    public AuthService(AppDbContext context)

    // Method: Authenticates user credentials
    // Purpose: Validates user email and password against database
    public async Task<Users> Login(string email, string password)

    // Method: Validates user credentials
    // Purpose: Checks if user exists and password matches
    public bool ValidateUser(string email, string password)

    // Method: Hashes password for secure storage
    // Purpose: Encrypts plain text password using secure algorithm
    private string HashPassword(string password)
}
```

```
// EditConsultationService.cs - Consultation Management
public class EditConsultationService
{
    private readonly AppDbContext _context;

    // Constructor: Initializes service with database context
    public EditConsultationService(AppDbContext context)

    // Method: Retrieves consultation by ID
    // Purpose: Fetches consultation details for editing
    public async Task<ConsultationRequest> GetConsultationById(int consultationId)

    // Method: Updates consultation details
    // Purpose: Saves modified consultation information to database
    public async Task<bool> UpdateConsultation(EditConsultationViewModel model)

    // Method: Validates consultation data
    // Purpose: Ensures all required fields are properly filled
    private bool ValidateConsultationData(EditConsultationViewModel model)
}
```

Domain Layer (Consultation.Domain)

```
// Admin.cs - Admin Entity
public class Admin
{
    // Property: Primary key for Admin entity
    [Key]
    public int AdminID { get; set; }

    // Property: Name of the administrator
    public string AdminName { get; set; }

    // Property: Navigation property to Users entity
    public Users Users { get; set; }

    // Property: Foreign key reference to Users
    [ForeignKey(nameof(Users))]
    public string UsersID { get; set; }
}
```

Infrastructure Layer (Consultation.Infrastructure)

```
// AppDbContext.cs - Database Context
public class AppDbContext : DbContext
{
    // Constructor: Initializes context with options
    // Purpose: Configures database connection and settings
    public AppDbContext(DbContextOptions<AppDbContext> options) :
base(options)

    // Method: Configures entity models and relationships
    // Purpose: Defines database schema and constraints
    protected override void OnModelCreating(ModelBuilder modelBuilder)

    // DbSet: Manages Admin entities
    public DbSet<Admin> Admins { get; set; }

    // DbSet: Manages Student entities
    public DbSet<Student> Students { get; set; }

    // DbSet: Manages Faculty entities
    public DbSet<Faculty> Faculty { get; set; }

    // DbSet: Manages Consultation entities
    public DbSet<ConsultationRequest> Consultations { get; set; }
}}

// Migration: addingmigration (20250712134553)
public partial class addingmigration : Migration
{
    // Method: Applies migration changes to database
    // Purpose: Creates or modifies database schema
    protected override void Up(MigrationBuilder migrationBuilder)

    // Method: Reverts migration changes
    // Purpose: Rolls back database schema changes
    protected override void Down(MigrationBuilder migrationBuilder)
`
```

Testing Layer (Consultation.Desktop.Test)

```
// AuthServiceTests.cs - Authentication Service Tests
[TestFixture]
public class AuthServiceTests
{
    private AppDbContext _context;
    private AuthService _authService;

    // Method: Initializes test environment before each test
    // Purpose: Sets up in-memory database and service instance
    [SetUp]
    public void Setup()

    // Method: Tests successful user login
    // Purpose: Verifies that valid credentials return user object
    [Test]
    public void Login_WithExistingUser_ReturnsUser()

    // Method: Tests login with invalid credentials
    // Purpose: Verifies that invalid credentials return null
    [Test]
    public void Login_WithInvalidCredentials_ReturnsNull()

    // Method: Cleans up test environment after each test
    // Purpose: Disposes database context and releases resources
    [TearDown]
    public void TearDown()
}
```

Helper Classes

```
// CustomMessageBox.cs - Custom MessageBox Helper
public static class CustomMessageBox
{
    // Method: Displays custom message box with larger font
    // Purpose: Provide consistent UI experience with 14pt font for better readability
    public static DialogResult Show(
        string message,
        string title,
        MessageBoxButtons buttons,
        MessageBoxIcon icon,
        MessageBoxDefaultButton defaultButton = MessageBoxDefaultButton.Button1)
)
```

Repository Methods

```
// IBulletinRepository.cs - Bulletin Repository Interface
public interface IBulletinRepository
{
    // Method: Retrieves all bulletins from database
    // Purpose: Get complete list of bulletins ordered by date published
    Task<List<Bulletin>> GetAllBulletins();

    // Method: Retrieves only active bulletins
    // Purpose: Get bulletins that are currently visible to users
    Task<List<Bulletin>> GetActiveBulletins();

    // Method: Retrieves archived bulletins
    // Purpose: Get bulletins that have been archived
    Task<List<Bulletin>> GetArchivedBulletins();

    // Method: Gets count of active bulletins
    // Purpose: Display statistics on dashboard
    Task<int> GetActiveBulletinCount();

    // Method: Gets count of archived bulletins
    // Purpose: Display archive statistics
    Task<int> GetArchivedBulletinCount();

    // Method: Retrieves single bulletin by ID
    // Purpose: Get specific bulletin for editing or viewing
    Task<Bulletin> GetBulletinById(int id);

    // Method: Adds new bulletin to database
    // Purpose: Create new bulletin entry
    Task<bool> AddBulletin(Bulletin bulletin);

    // Method: Updates existing bulletin
    // Purpose: Modify bulletin information
    Task<bool> UpdateBulletin(Bulletin bulletin);

    // Method: Deletes bulletin permanently
    // Purpose: Remove bulletin from system
    Task<bool> DeleteBulletin(int id);

    // Method: Archives bulletin (soft delete)
    // Purpose: Move bulletin to archive without permanent deletion
    Task<bool> ArchiveBulletin(int id);
```

Service Methods

```
// BulletinService.cs - Singleton Bulletin Service
public class BulletinService
{
    // Property: Gets singleton instance
    // Purpose: Ensure single instance across application
    public static BulletinService Instance { get; }

    // Event: Fired when new bulletin is published
    public event EventHandler<BulletinPublishedEventArgs> BulletinPublished;

    // Event: Fired when bulletins are modified
    public event EventHandler BulletinsChanged;

    // Event: Fired when bulletin is archived
    public event EventHandler BulletinArchived;

    // Event: Fired when bulletin is restored
    public event EventHandler BulletinRestored;

    // Method: Retrieves all bulletins
    // Purpose: Access bulletin data across application
    public async Task<List<Bulletin>> GetAllBulletins()

    // Method: Adds new bulletin
    // Purpose: Create bulletin and notify subscribers
    public async Task<bool> AddBulletin(Bulletin bulletin)

    // Method: Updates bulletin
    // Purpose: Modify bulletin and notify subscribers
    public async Task<bool> UpdateBulletin(Bulletin bulletin)

    // Method: Archives bulletin
    // Purpose: Move to archive and notify subscribers
    public async Task<bool> ArchiveBulletin(int bulletinId)
}
```

II. Inputs

Authentication Service Methods:

```
// AuthService.Login()
Inputs:
  - email: string (format: valid email address, e.g., "user@example.com")
  - password: string (plain text password entered by user)

// AuthService.ValidateUser()
Inputs:
  - email: string (user's email address)
  - password: string (plain text password)
```

CustomMessageBox.Show()

```
Inputs:
  - message: string (text to display in message box)
  - title: string (dialog title)
  - buttons: MessageBoxButtons (button configuration)
  - icon: MessageBoxIcon (icon type: Information, Warning, Error, Question)
  - defaultButton: MessageBoxDefaultButton (default button selection, optional)
```

BulletinRepository Methods

```
// GetBulletinById()
Inputs:
  - id: int (bulletin primary key)
// AddBulletin()
Inputs:
  - bulletin: Bulletin object
    - Title: string
    - Content: string
    - DatePublished: DateTime
    - IsActive: bool
    - PostedBy: string (user ID)
// UpdateBulletin()
Inputs:
  - bulletin: Bulletin object (with updated properties)

// DeleteBulletin(), ArchiveBulletin(), RestoreBulletin()
Inputs:
  - id: int (bulletin ID)
```

ConsultationRequestRepository Methods

// UpdateConsultationStatus()
Inputs:
- consultationId: int (consultation primary key)
- status: Domain.Enum.Status (new status value)

// DeleteConsultation()
Inputs:
- consultationId: int (consultation ID to delete)

Consultation Service Methods:

// EditConsultationService.GetConsultationById()
Inputs:
- consultationId: int (primary key of consultation record)

// EditConsultationService.UpdateConsultation()
Inputs:
- model: EditConsultationViewModel
  - studentName: string (full name of student)
  - facultyName: string (full name of faculty member)
  - courseCode: string (course identifier, e.g., "CS101")
  - studentUMID: string (unique student identifier)
  - concernDescription: string (detailed description of consultation topic)
  - dateSchedule: DateTime (scheduled consultation date and time)

Database Context Methods:

// AppDbContext.OnModelCreating()
Inputs:
- modelBuilder: ModelBuilder (EF Core model configuration object)

// Migration.UpdateData()
Inputs:
- table: string (table name, e.g., "AspNetUsers")
- keyColumn: string (primary key column name, e.g., "Id")
- keyValue: object (primary key value for record identification)
- column: string (column name to update, e.g., "PasswordHash")
- value: object (new value for column, e.g., hashed password string)

Test Setup Methods:

// AuthServiceTests.Setup()
Inputs:
- None (initializes using DbContextOptionsBuilder configuration)

// Login_WithExistingUser_ReturnsUser()
Inputs:
- Test data seeded in Setup() method:
  - Email: "testuser@example.com"
  - Password: "TestPassword123"

UI Control Inputs:

// Login.cs Properties
Inputs (from UI controls):
- txtEmail.Text: string (user-entered email)
- txtPassword.Text: string (user-entered password)

// LoginEvent Handler
Inputs:
- sender: object (event source)
- e: EventArgs (event arguments)

III. Outputs

CustomMessageBox.Show()

Output:
- DialogResult (OK, Cancel, Yes, No, etc.)
- Shows modal dialog with custom formatting

BulletinRepository Methods

// GetAllBulletins(), GetActiveBulletins(), GetArchivedBulletins()
Output:
- Task<List<Bulletin>> (asynchronous)
- Returns: List of Bulletin objects ordered by DatePublished descending
- Returns: Empty list if error occurs

// GetActiveBulletinCount(), GetArchivedBulletinCount()
Output:
- Task<int> (asynchronous)
- Returns: Count of bulletins matching criteria

// GetBulletinById()
Output:
- Task<Bulletin> (asynchronous)
- Returns: Bulletin object if found
- Returns: null if not found or error occurs

// AddBulletin(), UpdateBulletin(), DeleteBulletin(), ArchiveBulletin(), RestoreBulletin()
Output:
- Task<bool> (asynchronous)
- Returns: true if operation successful
- Returns: false if operation fails

Service Methods

// Instance property
Output:
- BulletinService (singleton instance)

// GetAllBulletins()
Output:
- Task<List<Bulletin>>
- Delegates to repository

// AddBulletin(), UpdateBulletin()
Output:
- Task<bool>
- Side effect: Fires BulletinPublished or BulletinsChanged event

// ArchiveBulletin()
Output:
- Task<bool>
- Side effect: Fires BulletinArchived event

Authentication Service Methods:

```
// AuthService.Login()
Output:
  - Task<Users> (asynchronous)
  - Returns: Users object with populated properties if authentication successful
  - Returns: null if authentication fails
  - Properties in Users object:
    - UserId: string
    - Email: string
    - UserName: string
    - PasswordHash: string
    - UserType: UserType enum
    - Created: DateTime

// AuthService.ValidateUser()
Output:
  - bool
  - Returns: true if credentials are valid
  - Returns: false if credentials are invalid or user doesn't exist

// AuthService.HashPassword()
Output:
  - string (hashed password)
  - Format: Base64-encoded hash string
```

UI Property Outputs:

```
// Login.useremail property
Output:
  - string (text content of txtEmail TextBox)

// Login.password property
Output:
  - string (text content of txtPassword TextBox)
```

Consultation Service Methods:

// EditConsultationService.GetConsultationById()
Output:
- Task<ConsultationRequest> (asynchronous)
- Returns: ConsultationRequest object if found
- Returns: null if consultation ID doesn't exist
- Properties in ConsultationRequest:
  - ConsultationID: int
  - StudentID: string
  - FacultyID: string
  - CourseCode: string
  - ConcernDescription: string
  - DateSchedule: DateTime
  - Status: Status enum

// EditConsultationService.UpdateConsultation()
Output:
- Task<bool> (asynchronous)
- Returns: true if update successful
- Returns: false if update fails or validation errors occur

// ValidateConsultationData()
Output:
- bool
- Returns: true if all validation checks pass
- Returns: false if any validation fails

Database Context Methods:

// AppDbContext.OnModelCreating()
Output:
- void
- Side effect: Configures entity relationships and constraints in model

// AppDbContext.SaveChangesAsync()
Output:
- Task<int>
- Returns: Number of state entries written to database

Migration Methods:

// Migration.Up()
Output:
- void
- Side effect: Applies schema changes to database
- Creates/modifies tables, columns, indexes, constraints

// Migration.Down()
Output:
- void
- Side effect: Reverts schema changes from database

// UpdateData()
Output:
- void
- Side effect: Updates specific records in database table
- Example: Updates AspNetUsers with new password hashes

Test Methods:

// AuthServiceTests.Setup()
Output:
- void
- Side effect: Initializes _context and _authService fields

// Login_WithExistingUser_ReturnsUser()
Output:
- void
- Assertion: Verifies returned Users object is not null
- Assertion: Verifies user properties match expected values

// AuthServiceTests.TearDown()
Output:
- void
- Side effect: Disposes database context, releases resources

IV. SIDE EFFECTS

CustomMessageBox.Show()

Side Effects:
- UI Blocking: Displays modal dialog, blocks thread until user responds
- Form Creation: Creates temporary Form object with custom styling
- Memory Allocation: Allocates resources for form and controls
- Resource Cleanup: Disposes form after dialog closes

BulletinRepository Methods

// GetAllBulletins(), GetActiveBulletins(), GetArchivedBulletins()
Side Effects:
- Database Query: Executes SELECT with OrderByDescending
- Entity Tracking: Loads entities into EF Core change tracker
- Connection: Opens database connection
- Logging: Writes error to console if exception occurs

// AddBulletin()
Side Effects:
- Database Write: INSERT operation on Bulletin table
- Transaction: Opens and commits transaction
- Auto-increment: Generates new BulletinID
- Audit: Sets DatePublished to current DateTime

// UpdateBulletin()
Side Effects:
- Database Write: UPDATE operation
- Concurrency: May trigger concurrency exception
- Change Tracking: Marks entity as modified

// ArchiveBulletin()
Side Effects:
- Database Write: Updates IsActive = false
- Does not physically delete record (soft delete)

// RestoreBulletin()
Side Effects:
- Database Write: Updates IsActive = true

BulletinService Methods

```
// AddBulletin()
Side Effects:
  - Database Write: Via repository
  - Event Firing: Raises BulletinPublished event
  - UI Updates: Subscribers refresh their views
  - Notification: May trigger notification creation

// UpdateBulletin()
Side Effects:
  - Database Write: Via repository
  - Event Firing: Raises BulletinsChanged event
  - UI Refresh: All listening views update

// ArchiveBulletin()
Side Effects:
  - Database Write: Via repository
  - Event Firing: Raises BulletinArchived event
  - View Updates: Dashboard and bulletin views refresh
```

Authentication Service Side Effects:

```
// AuthService.Login()
Side Effects:
  - Database Query: Executes SELECT query against AspNetUsers table
  - Performance: Database connection opened and query executed
  - Logging: May log authentication attempts (if implemented)
  - No modifications to database state

// AuthService.ValidateUser()
Side Effects:
  - Database Query: Reads user record from database
  - Memory: Loads user entity into memory for validation
  - No persistent state changes
```

Consultation Service Side Effects:

```
// EditConsultationService.UpdateConsultation()
Side Effects:
  - Database Modification: Updates consultation record in database
  - Transaction: Opens database transaction
  - Commit/Rollback: Commits changes if successful, rolls back on failure
  - Change Tracking: EF Core tracks entity state changes
  - Audit Trail: May trigger audit logging (if implemented)
  - Notification: May trigger notification creation for related users

// GetConsultationById()
Side Effects:
  - Database Query: Reads consultation data
  - Entity Tracking: Loads entity into EF Core change tracker
  - Memory Allocation: Allocates memory for entity object
```

Database Context Side Effects:

```
// AppDbContext.SaveChangesAsync()
Side Effects:
  - Database Write: Commits all pending changes to database
  - Transaction Management: Opens, commits, or rolls back transaction
  - Change Tracking: Resets change tracker state after save
  - Concurrency Checks: Validates optimistic concurrency tokens
  - Triggers: May execute database triggers (if defined)

// OnModelCreating()
Side Effects:
  - Model Configuration: Defines entity relationships in memory
  - Constraint Definition: Sets up foreign keys, indexes
  - Schema Validation: Validates model against database schema
  - No runtime database changes (only affects schema generation)
```

Migration Side Effects:

```
// Migration.Up()
Side Effects:
  - Schema Changes: Creates/alters tables, columns, indexes
  - Data Migration: Executes UpdateData operations
  - Database Lock: May lock tables during schema modifications
  - History Update: Records migration in __EFMigrationsHistory table
  - Password Updates: Modifies AspNetUsers.PasswordHash values
  - Irreversible Changes: Some migrations cannot be rolled back

// Migration.Down()
Side Effects:
  - Schema Rollback: Reverts tables, columns, indexes
  - Data Loss: May delete data if not properly handled
  - History Update: Removes migration from __EFMigrationsHistory
```

Connection String Side Effects (from Infrastructure):

```
// UseSqlServer() Configuration
Side Effects:
  - Connection Pool: Establishes connection pool to LocalDB
  - Initial Catalog: Connects to ConsultationDatabase
  - Security Context: Uses integrated Windows authentication
  - Timeout Settings: Sets 30-second connection timeout
  - Encryption: Disables connection encryption (Encrypt=False)
  -
```

Connection String Side Effects (from Infrastructure):

```
// UseSqlServer() Configuration
Side Effects:
  - Connection Pool: Establishes connection pool to LocalDB
  - Initial Catalog: Connects to ConsultationDatabase
  - Security Context: Uses integrated Windows authentication
  - Timeout Settings: Sets 30-second connection timeout
  - Encryption: Disables connection encryption (Encrypt=False)
  - Certificate Trust: Disables server certificate validation
```

UI Event Handler Side Effects:

// Login.LoginEvent
Side Effects:
- UI Update: May disable login button during processing
- Label Updates: Sets resultLabel1 or ErrorPassLabel text
- Form Navigation: May close login form and open main window
- Error Display: Shows error messages for invalid credentials
- Session Management: Establishes user session on successful login

// InitializeComponent()
Side Effects:
- Control Creation: Instantiates UI controls (TextBox, Label, Button)
- Event Binding: Attaches event handlers to controls
- Layout Configuration: Sets control positions and properties
- Resource Loading: Loads form resources (if any)

Test Method Side Effects:

// AuthServiceTests.Setup()
Side Effects:
- Memory Allocation: Creates in-memory database
- Test Data Seeding: Inserts test user records
- Service Instantiation: Creates AuthService instance
- Context Initialization: Configures DbContext with test options

// Login_WithExistingUser_ReturnsUser()
Side Effects:
- Database Query: Reads from in-memory test database
- No persistent state changes (in-memory database discarded after test)

// TearDown()
Side Effects:
- Resource Cleanup: Disposes database context
- Memory Release: Frees allocated memory
- Connection Closure: Closes database connections

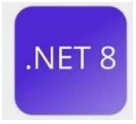
H. DATABASE DESIGN

I. Database Using Entity Framework Core

System Requirements

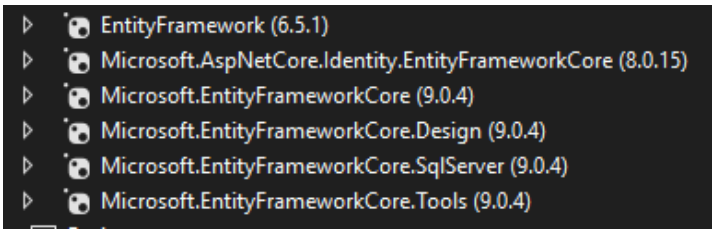
To build and run the EF Core database project, the following tools and frameworks are required:

- **.NET SDK (version 8.0)**



- **IDE: Visual Studio 2022**

- **EF Core Packages:**



  - Microsoft.AspNetCore.Identity.EntityFrameworkCore(8.0.15)
  - Micosoft.EntityFramewordCore(9.0.4)
  - Micosoft.EntityFramewordCore.Design(9.0.4)
  - Micosoft.EntityFramewordCore.SqlServer(9.0.4)
  - Micosoft.EntityFramewordCore.Tools(9.0.4)

- **Database Engine:**

  - Microsoft SQL Server Management
  - MySQL Workbench

II. Database Design

The database is designed to efficiently manage and organize the core operations of the school system by integrating data related to students, faculty, courses, programs, and administrative activities. Its main purpose is to provide a centralized platform that ensures accurate record-keeping, smooth coordination between departments, and easy access to information. The system helps reduce redundancy and improve efficiency in tracking student progress, managing course enrollments, scheduling consultations, and handling notifications or announcements. By defining clear relationships among entities such as students, faculty, programs, and school years, the database establishes a well-structured framework that supports both academic and administrative functions while promoting data consistency and reliability across the entire school system.

Core Tables

These contain essential information about main entities:

- **Student**: Stores student information and links to programs, courses, consultation requests, and user accounts.
- **Faculty**: Contains faculty member details and connects to schedules, consultation requests, enrolled courses, and user accounts.
- **SchoolYear**: Represents academic years and semesters, linking students and faculty to enrollment periods.
- **Users**: Manages system users, extending identity management to students, faculty, and administrators.
- **Admin**: Stores administrator information and links to user accounts.

Master Tables

These store reference or static data used by core tables:

- **Program**: Lists academic programs under departments.
- **Department**: Contains academic department details.
- **Notification:** Stores messages or alerts sent to students and faculty, such as

reminders, updates, or important announcements related to academic activities or events.

- **Bulletin**: Stores announcements and bulletins.
- **FacultySchedule**: Defines faculty availability with day and time slots.

Transactional Tables

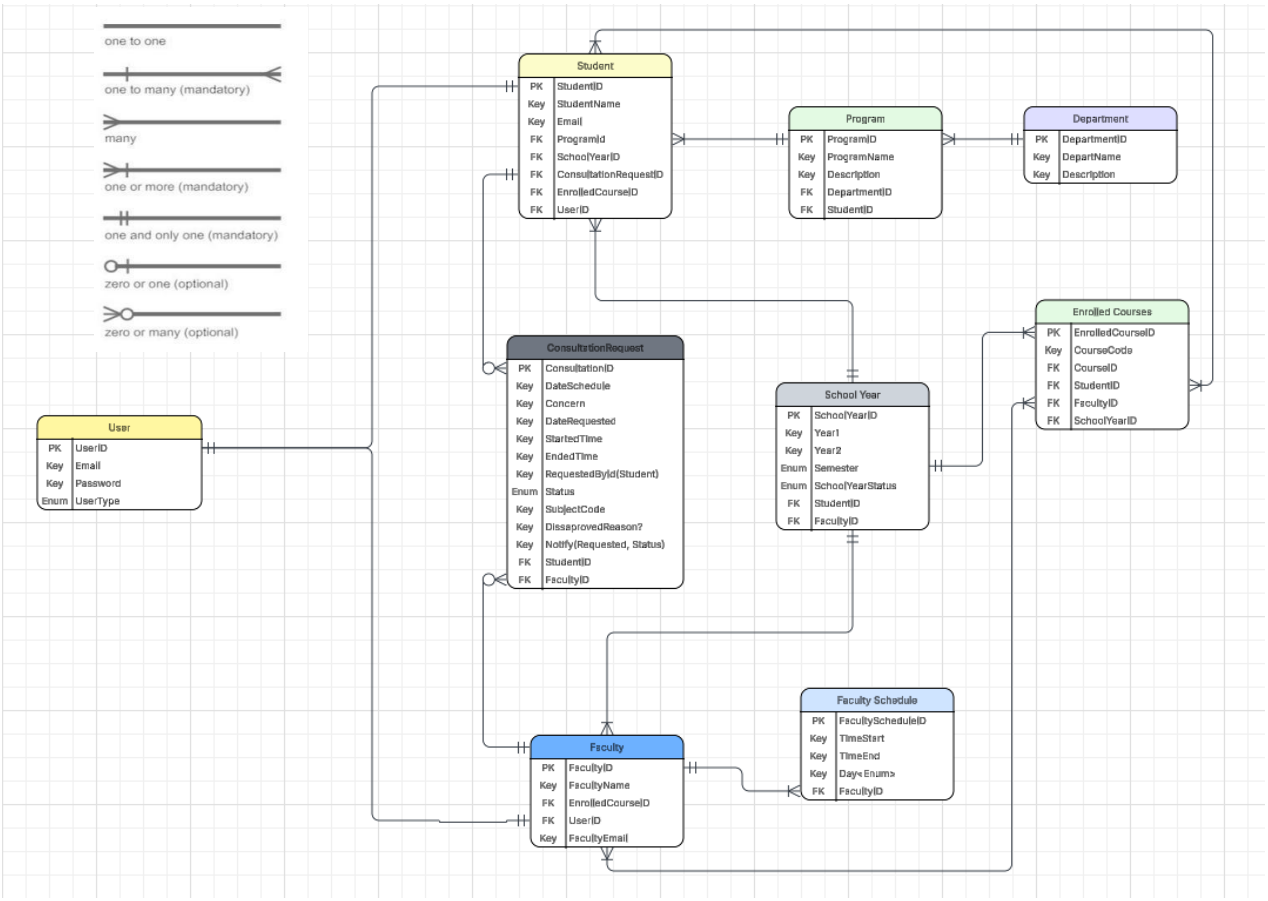
These capture dynamic activities and interactions:

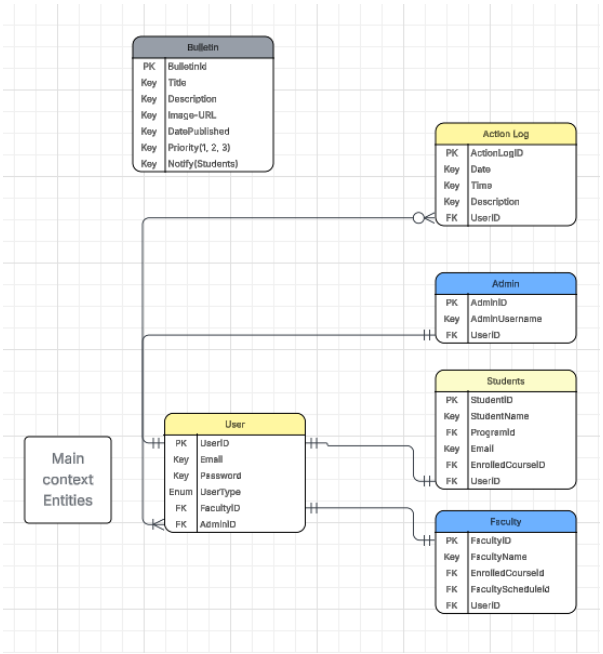
- **EnrolledCourses**: Maps students to courses for specific school years and faculties.
- **ConsultationRequest**: Records requests for student-faculty consultations with scheduling and status.
- **ActionLog**: Audits user actions within the system.

Entity Relationships

- A Student is enrolled in one Program and can register in many Courses through EnrolledCourses.
- Each Course belongs to one Program.
- A Faculty member can teach many courses and has a schedule recorded in FacultySchedule.
- ConsultationRequests link students and faculty for advising purposes.
- Users are linked one-to-one with Student, Faculty, or Admin accounts to manage authentication and roles.
- The SchoolYear entity links students and faculty to specific academic years and semesters, affecting course enrollment and schedules.

III. Erd (Entity Relationship Diagram





IV. Entity Classes

Database is built with 13 entities:

1. Student

Type: Core Table / Primary Table

Purpose: Stores main student information.

Relations:

- Linked to **Program** → **ProgramID** ***(FK)***
- Linked to **SchoolYear**→ **ProgramID** ***(FK)***→ via **SchoolYears** list ***(FK)***
- Linked to **ConsultationRequest** → via **ConsultationRequests** list ***(FK)***
- Linked to **Users** → One-to-one user account mapping
- Linked to **EnrolledCourse** → via **EnrolledCourses** list ***(FK)***

Attributes:

- **StudentID** ***(int)***: Unique identifier for each student.
- **StudentUMID** ***(string)***: University's unique matriculation ID for the student.
- **StudentName** ***(string)***: Full name of the student.
- **Email** ***(string)***: Student's email address.
- **ProgramID** ***(int)***: Foreign key referencing the academic program.
- **Program** ***(Program)***: Navigation property to access program details.
- **ConsultationRequests** *(List<**ConsultationRequest**>)*: All consultation records linked to the student.
- **SchoolYears** *(List<**SchoolYear**>)*: Academic years the student has been enrolled in.
- **Users** ***(Users)***: Associated system user account.
- **EnrolledCourses** *(List<**EnrolledCourse**>)*: List of courses the student is enrolled in.

```
public class Student
{
    [Key]
    2 references
    public int StudentID { get; set; }

    1 reference
    public string StudentUMID { get; set; }

    1 reference
    public string StudentName { get; set; }

    0 references
    public string Email { get; set; }

    [ForeignKey(nameof(ProgramID))]
    1 reference
    public int ProgramID { get; set; }
    1 reference
    public virtual Program Program { get; set; }

    [InverseProperty(nameof(Users.Student))]
    2 references
    public Users Users { get; set; }

    3 references
    public List<ConsultationRequest> ConsultationRequests { get; set; }
    4 references
    public List<SchoolYear> SchoolYears { get; set; }

    3 references
    public List<EnrolledCourse> EnrolledCourses { get; set; }
}
```

2. Program

Type: Master Table

Purpose: Lists academic programs offered by departments.

Relations:

- Linked to **Department → DepartmentID *(FK)***
- Used in **Student → via ProgramID *(FK)***
- Used in **Courses → via ProgramID *(FK)***

Attributes:

- **ProgramID *(int)*:** Unique identifier for each program.
- **ProgramName *(string)*:** Name of the program (e.g., BSIT, BSCPE).
- **Description *(string)*:** Brief detail about the program.
- **DepartmentID *(int)*:** Foreign key referring to the department offering the program.
- **Department *(Department)*:** Navigation property to access department details.

```
public class Program
{
    [Key]
    0 references
    public int ProgramID { get; set; }

    0 references
    public string  ProgramName { get; set; }

    0 references
    public string Description { get; set; }

    [ForeignKey(nameof(DepartmentID))]
    1 reference
    public int  DepartmentID { get; set; }
    0 references
    public virtual Department Department { get; set; }


}
```

3. Department

Type: Master Table

Purpose: Stores academic departments.

Relations:

- Used in **Program → via DepartmentID *(FK)* Attributes:**
- **DepartmentID *(int)*:** Unique department identifier.
- **DepartmentName *(string)*:** Name of the department.
- **Description *(string)*:** Description of the department.

```
public class Department
{
    [Key]
    0 references
    public int DepartmentID { get; set; }

    0 references
    public string DepartmentName { get; set; }

    0 references
    public string Description { get; set; }

    // many programs to one department ...
}
```

4. **EnrolledCourses**

Type: Transactional Table

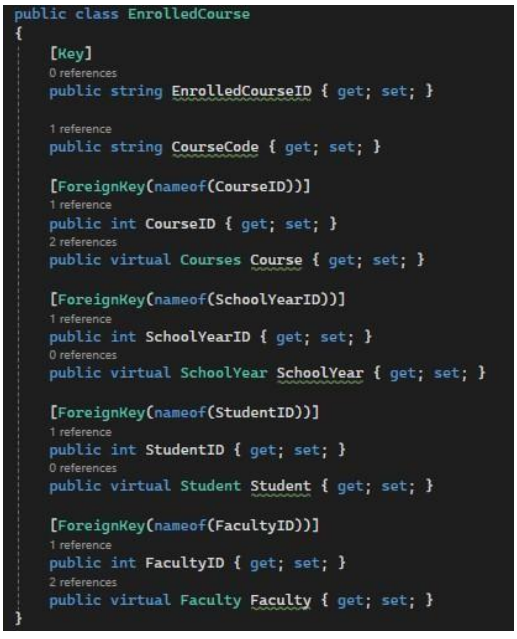
Purpose: Stores course enrollment records, mapping students to courses for specific academic years and assigned faculty.

Relations:

- Linked to **Course → CourseID** ***(FK)***
- Linked to **SchoolYear → SchoolYearID** ***(FK)***
- Linked to **Student → StudentID** ***(FK)***
- Linked to **Faculty → FacultyID** ***(FK)***

Attributes:

- **EnrolledCourseID** ***(string)***: Unique ID per enrollment record.
- **CourseCode** ***(string)***: Code assigned to the course (e.g., 6802).
- **CourseID** ***(int)***: Foreign key referring to the actual course.
- **Course** ***(Courses)***: Navigation property to access course details.
- **SchoolYearID** ***(int)***: Foreign key referring to the academic year.
- **SchoolYear** ***(SchoolYear)***: Navigation property to access the academic year.
- **StudentID** ***(int)***: Foreign key referring to the enrolled student.
- **Student** ***(Student)***: Navigation property to access student information.
- **FacultyID** ***(int)***: Foreign key referring to the faculty teaching the course.
- **Faculty** ***(Faculty)***: Navigation property to access faculty details.



5. SchoolYear

Type: Core Table

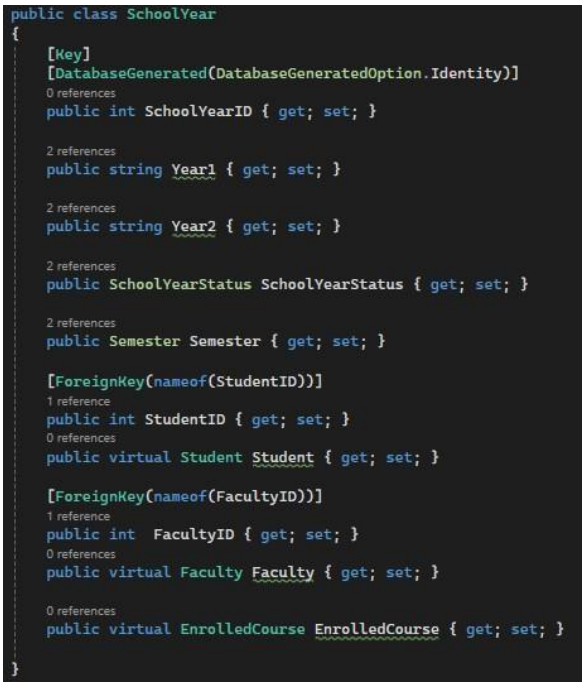
Purpose: Represents an academic year and semester, linking students and faculty to their enrollment periods.

Relations:

- Linked to **Student → StudentID *(FK)***
- Linked to **Faculty → FacultyID *(FK)***
- Linked to **EnrolledCourse → via EnrolledCourse navigation property**

Attributes:

- **SchoolYearID *(int)*:** Unique identifier for the school year (auto-generated).
- **Year1 *(string)*:** Starting year of the academic year (e.g., "2024").
- **Year2 *(string)*:** Ending year of the academic year (e.g., "2025").
- **SchoolYearStatus *(SchoolYearStatus)*:** Status of the school year (e.g., active, inactive).
- **Semester *(Semester)*:** Semester within the academic year (e.g., Fall, Spring).
- **StudentID *(int)*:** Foreign key referring to the student for this school year.
- **Student *(Student)*:** Navigation property to access the student details.
- **FacultyID *(int)*:** Foreign key referring to the faculty for this school year.
- **Faculty *(Faculty)*:** Navigation property to access the faculty details.
- **EnrolledCourse *(EnrolledCourse)*:** Navigation property for the enrolled course(s) in this school year.



```
public class SchoolYear
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    0 references
    public int SchoolYearID { get; set; }

    2 references
    public string Year1 { get; set; }

    2 references
    public string Year2 { get; set; }

    2 references
    public SchoolYearStatus SchoolYearStatus { get; set; }

    2 references
    public Semester Semester { get; set; }

    [ForeignKey(nameof(StudentID))]
    1 reference
    public int StudentID { get; set; }
    0 references
    public virtual Student Student { get; set; }

    [ForeignKey(nameof(FacultyID))]
    1 reference
    public int  FacultyID { get; set; }
    0 references
    public virtual Faculty Faculty { get; set; }

    0 references
    public virtual EnrolledCourse EnrolledCourse { get; set; }
}
```

6. ConsultationRequest

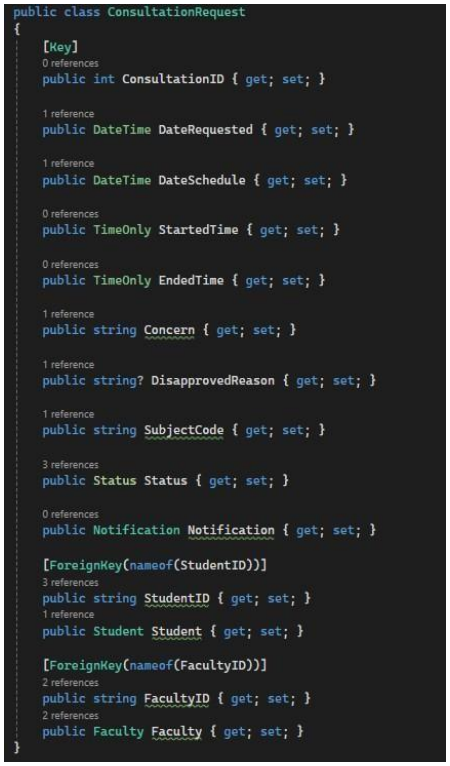
Type: Transactional Table

Purpose: Records requests for consultations between students and faculty, including scheduling and status information.

Relations:

- **Linked to Student → via navigation property Student**
- **Linked to Faculty → FacultyID *(FK)***

Attributes:



```
public class ConsultationRequest
{
    [Key]
    0 references
    public int ConsultationID { get; set; }

    1 reference
    public DateTime DateRequested { get; set; }

    1 reference
    public DateTime DateSchedule { get; set; }

    0 references
    public TimeOnly StartedTime { get; set; }

    0 references
    public TimeOnly EndedTime { get; set; }

    1 reference
    public string Concern { get; set; }

    1 reference
    public string? DisapprovedReason { get; set; }

    1 reference
    public string SubjectCode { get; set; }

    3 references
    public Status Status { get; set; }

    0 references
    public Notification Notification { get; set; }

    [ForeignKey(nameof(StudentID))]
    3 references
    public string StudentID { get; set; }
    1 reference
    public Student Student { get; set; }

    [ForeignKey(nameof(FacultyID))]
    2 references
    public string FacultyID { get; set; }
    2 references
    public Faculty Faculty { get; set; }
}
```

- **ConsultationID *(int)*:** Unique identifier for each consultation request.
- **DateRequested *(DateTime)*:** Date when the consultation was requested.
- **DateSchedule *(DateTime)*:** Scheduled date for the consultation.
- **StartedTime *(TimeOnly)*:** Start time of the consultation.
- **EndedTime *(TimeOnly)*:** End time of the consultation.
- **Concern *(string)*:** Description of the issue or topic to be discussed.
- **DisapprovedReason *(string?)*:** Optional reason for disapproval if the consultation was rejected.
- **SubjectCode *(string)*:** Code of the related subject/course.
- **Status *(Status)*:** Current status of the consultation request (e.g., pending, approved).
- **Notification *(Notification)*:** Associated notification related to this consultation.
- **FacultyID *(int)*:** Foreign key referring to the faculty member handling the consultation.
- **Faculty *(Faculty)*:** Navigation property to access faculty details.
- **Student *(Student)*:** Navigation property to access the requesting student's details.

7. Faculty

Type: Core Table

Purpose: Stores information about faculty members, including their personal details and academic roles.

Relations:

- Linked to **Users → via Users navigation property (one-to-one user account mapping)**
- Linked to **EnrolledCourse → via EnrolledCourseID *(FK)***
- Linked to **FacultySchedule → via FacultyScheduleID *(FK)***
- Linked to **ConsultationRequest → via ConsultationRequests collection**
- Linked to **SchoolYear → via SchoolYears collection**

```
public class Faculty
{
    [Key]
    0 references
    public int FacultyID { get; set; }

    1 reference
    public string FacultyUMID { get; set; }

    3 references
    public string FacultyName { get; set; }

    [ForeignKey(nameof(EnrolledCourseID))]
    1 reference
    public int EnrolledCourseID { get; set; }
    0 references
    public virtual EnrolledCourse EnrolledCourse { get; set; }

    [InverseProperty(nameof(Users.Faculty))]
    2 references
    public Users Users { get; set; }

    [ForeignKey(nameof(FacultyScheduleID))]
    1 reference
    public int FacultyScheduleID { get; set; }

    0 references
    public virtual FacultySchedule FacultySchedule { get; set; }

    0 references
    public List<ConsultationRequest> ConsultationRequests { get; set; }
    0 references
    public ICollection<SchoolYear> SchoolYears { get; set; }
}
```

Attributes:

- **FacultyID (int):** Unique identifier for each faculty member.
- **FacultyUMID (string):** University matriculation ID for the faculty member.
- **FacultyName (string):** Full name of the faculty member.
- **Users (Users):** Navigation property to the associated user account.
- **EnrolledCourseID (int):** Foreign key referring to the enrolled course (optional, may not be necessary depending on usage).
- **EnrolledCourse (EnrolledCourse):** Navigation property to access the enrolled course details.
- **FacultyScheduleID (int):** Foreign key referring to the faculty schedule.
- **FacultySchedule (FacultySchedule):** Navigation property to access the faculty schedule.
- **ConsultationRequests** (List<ConsultationRequest>): Collection of consultation requests involving the faculty.
- **SchoolYears (ICollection<SchoolYear>):** Collection of school years associated with the faculty.

8. FacultySchedule

Type: Core Table
Purpose: Defines the schedule availability of faculty members, including days and time slots.

Relations:

- Linked to **Faculty → FacultyID *(FK)***

Attributes:

- **FacultyScheduleID *(int)*:** Unique identifier for each faculty schedule entry.
- **TimeStart *(TimeOnly)*:** Start time of the scheduled period.
- **TimeEnd *(TimeOnly)*:** End time of the scheduled period**.**
- **Day *(DaysOfWeek)*:** Day of the week for the schedule (e.g., Monday, Tuesday).
- **FacultyID *(int)*:** Foreign key referring to the faculty member.
- **Faculty *(Faculty)*:** Navigation property to access the faculty details.

```
public class FacultySchedule
{
    [Key]
    0 references
    public int FacultyScheduleID { get; set; }

    0 references
    public DateTime TimeStart { get; set; }

    0 references
    public DateTime TimeEnd { get; set; }

    0 references
    public DaysOfWeek Day { get; set; }

    [ForeignKey(nameof(Faculty))]
    0 references
    public int FacultyID { get; set; }
    1 reference
    public virtual Faculty Faculty { get; set; }
}
```

9. **Users**

Type: Core Table / Identity User

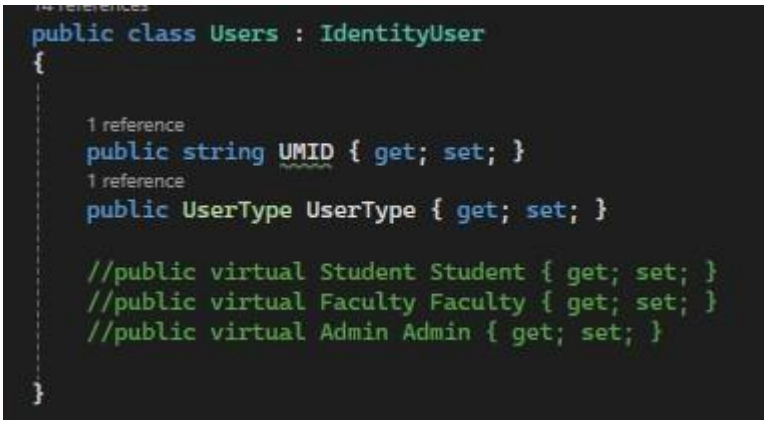
Purpose: Represents system users, extending ASP.NET IdentityUser for authentication and authorization, with additional user-specific information.

Relations:

- **Potentially linked to Student, Faculty, and Admin entities (commented out, can be enabled if needed for navigation).**

Attributes:

- **UMID *(string)*:** Unique matriculation or user ID tied to the user.
- **UserType *(UserType)*:** Enum representing the role or type of user (e.g., Student, Faculty, Admin).
- Inherits all standard properties from IdentityUser (such as **Id**, **UserName**, **Email**, **PasswordHash**, etc.).



10. Bulletin

Type: Master Table

Purpose: Stores information about bulletins or announcements to be displayed in the system.

Relations:

- **None**

Attributes:

- **BulletinID *(int)*:** Unique identifier for each bulletin.
- **Title *(string)*:** Title of the bulletin or announcement.
- **Description *(string)*:** Detailed content or description of the bulletin.
- **ImageURL *(string)*:** URL path to an image associated with the bulletin.
- **DatePublished *(DateTime)*:** The date when the bulletin was published.
- **Priority *(int)*:** Numeric value indicating the importance or display priority of the bulletin.
- **Notify *(string)*:** Notification message or flag related to the bulletin.

```
public class Bulletin
{

    [Key]
    0 references
    public int BulletinID { get; set; }

    0 references
    public string Title { get; set; }

    0 references
    public string Description { get; set; }

    0 references
    public string ImageURL { get; set; }

    0 references
    public DateTime DatePublished { get; set; }

    0 references
    public int Priority { get; set; }

    0 references
    public string Notify { get; set; }
}
```

11. ActionLog

Type: Transactional Table / Audit Log

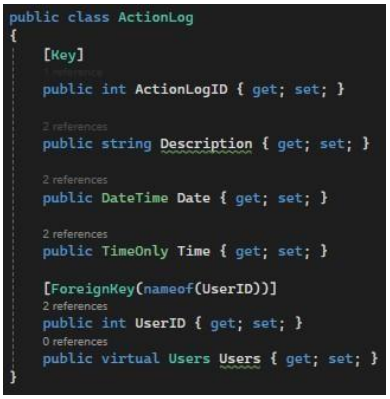
Purpose: Records user actions and activities for auditing and tracking purposes.

Relations:

- Linked to **Users via navigation property.**

Attributes:

- **ActionLogID *(int)*:** Unique identifier for each log entry.
- **Description *(string)*:** Details about the action performed by the user.
- **Date *(DateTime)*:** The date when the action occurred.
- **Time *(TimeOnly)*:** The specific time the action took place.
- **Users *(Users)*:** Navigation property to the user who performed the action.



12. Admin Type: Core Table

Purpose: Stores information about system administrators.

Relations:

2. Linked to **Users via navigation property for authentication and user details.**

Attributes:

3. **AdminID *(int)*:** Unique identifier for each admin.
4. **AdminName *(string)*:** Full name of the administrator.
5. **Users *(Users)*:** Navigation property to the associated system user account.

V. Dbcontext Configuration

Configuring a DbContext class in an ASP.NET Core application using Entity Framework Core. The DbContext acts as a bridge between your domain classes (entities) and the database.

Create the AppDbContext Class:

- *Make your class inherit from IdentityDbContext<Users> to include Identity features:*

```
public class AppDbContext : IdentityDbContext<Users>
{
    0 references
    public AppDbContext() : base()
    {
    }

    1 reference
    public AppDbContext(DbContextOptions<AppDbContext> options)
            : base(options)
    {
    }
```

Configure the Database Connection

- *Override the OnConfiguring method to specify the MySQL connection string:*

```
0 references
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    var cs = "Server=consultationdb.mysql.database.azure.com;" +
   "Port=3306;" +
   "Database=umeca_database;" +
   "User Id=ConsultationDb;" +
   "Password=MyServerAdmin123!;" +
   "SslMode=Required;";

    optionsBuilder.UseMySql(cs, new MySqlServerVersion(new Version(8, 0, 36)));

    optionsBuilder.ConfigureWarnings(w => w.Ignore(RelationalEventId.PendingModelChangesWarning));
}
```

Register DbSet Properties

- *Inside the AppDbContext class, define all your database tables as DbSet<TEntity>:*

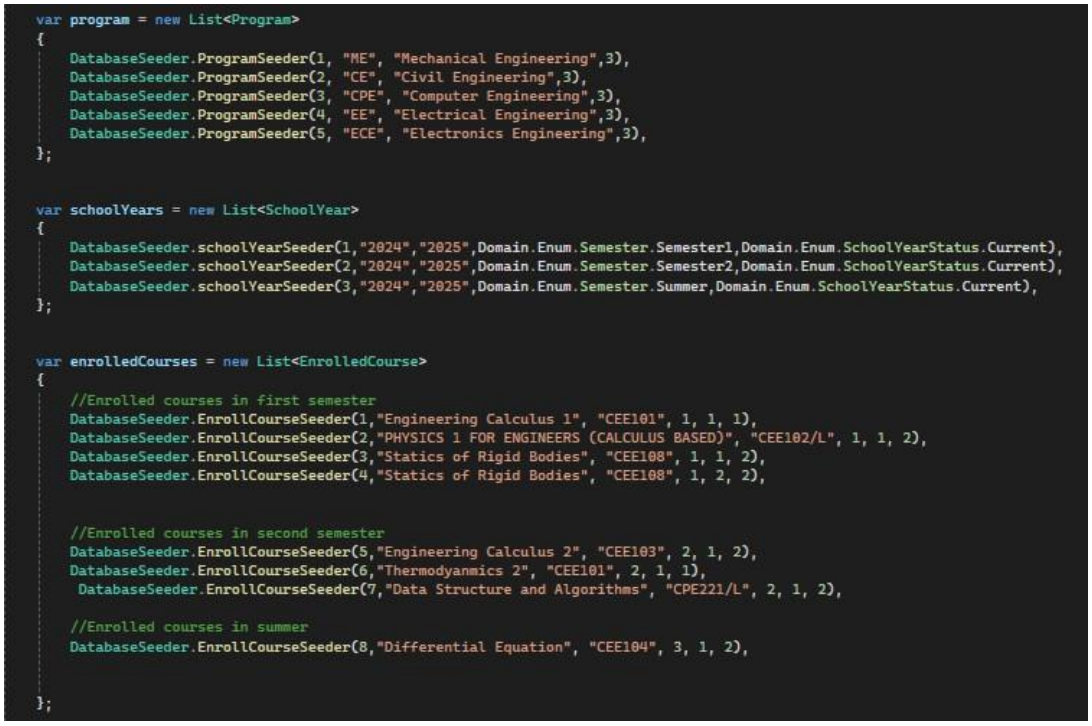
```
2 references
public DbSet<ActionLog> ActionLog { get; set; }
1 reference
public DbSet<Admin> Admin { get; set; }
0 references
public DbSet<Bulletin> Bulletin { get; set; }
3 references
public DbSet<ConsultationRequest> ConsultationRequest { get; set; }
0 references
public DbSet<Department> Department { get; set; }
0 references
public DbSet<EnrolledCourse> EnrolledCourse { get; set; }
2 references
public DbSet<Faculty> Faculty { get; set; }
0 references
public DbSet<Program> Program { get; set; }
0 references
public DbSet<SchoolYear> SchoolYear { get; set; }
2 references
public DbSet<Student> Students { get; set; }
0 references
public DbSet<Users> Users { get; set; }
}
```

Seed Initial Data (Optional but Recommended For Testing)

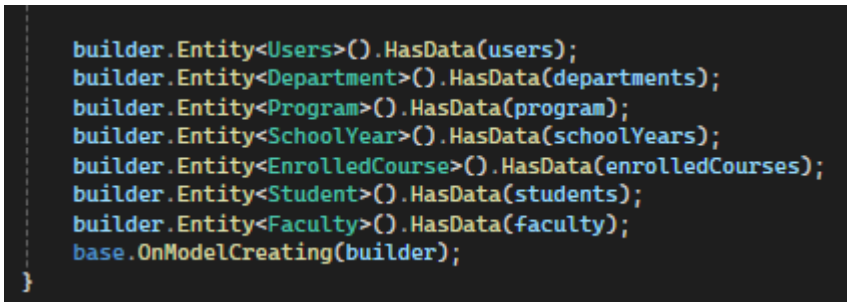
To add default records (students, faculty, programs, etc.), override the OnModelCreating method and use a seeder class:

Figure A

```
0 references
protected override void OnModelCreating(ModelBuilder builder)
{
    var users = new List<Users>
    {
    DatabaseSeeder.UserSeeder("273F528F-5330-411F-9C6B-01543D6249C3", "550200", "Cedric Setimo", "CedricSetimo.550200@umindanao.edu.ph", "MyStudent123!", Domain.Enum.UserType.Student),
    DatabaseSeeder.UserSeeder("53D8F920-EBEC-4DF3-8C53-21F6D123F0D9", "321033", "Rey Mateo", "ReyMateo.550200@umindanao.edu.ph", "MyFaculty123!", Domain.Enum.UserType.Faculty),
    DatabaseSeeder.UserSeeder("6B187E9D-FD71-4F1D-AFDF-EA1D91E818EF", "444533", "Raine Isid", "RaineIsid.550200@umindanao.edu.ph", "MyAdmin123!", Domain.Enum.UserType.Admin),
    DatabaseSeeder.UserSeeder("D0B26692-E380-4374-985F-239B56D06C20", "547343", "Ellaine Musni", "EllaineMusni.550200@umindanao.edu.ph", "MyAdmin123!", Domain.Enum.UserType.Student),
    DatabaseSeeder.UserSeeder("1226920F-9508-44B3-845A-ABABBBCBCF5D", "685043", "Reggie Soylon", "ReggieSoylon.6850@umindanao.edu.ph", "MyStudent123!", Domain.Enum.UserType.Student),
    DatabaseSeeder.UserSeeder("0A52E15B-95E6-40FE-9110-9A44817BFF39", "899812", "Cheley Balsomo", "CheleyBalsomo.8998@umindanao.edu.ph", "MyStudent123!", Domain.Enum.UserType.Student),
    DatabaseSeeder.UserSeeder("78B4AF2A-672F-43C5-B819-5F0B407B7187", "797132", "Jeanelle Labsan", "JeanelleLabsan.7971@umindanao.edu.ph", "MyFaculty123!", Domain.Enum.UserType.Faculty),
    DatabaseSeeder.UserSeeder("59CF8531-68E4-466B-BAEC-45305FE16A14", "924132", "Christopher Destajo", "ChristopherDestajo.9241@umindanao.edu.ph", "MyStudent123!", Domain.Enum.UserType.Student)
    };

    var departments = new List<Department>
    {
        DatabaseSeeder.DepartmentSeeder(1, "CASE", "College of Arts and Sciences Education"),
        DatabaseSeeder.DepartmentSeeder(2, "CBAE", "College of Business Administration Education"),
        DatabaseSeeder.DepartmentSeeder(3, "CEE", "College of Engineering Education")
    };
```



```
    var program = new List<Program>
    {
        DatabaseSeeder.ProgramSeeder(1, "ME", "Mechanical Engineering",3),
        DatabaseSeeder.ProgramSeeder(2, "CE", "Civil Engineering",3),
        DatabaseSeeder.ProgramSeeder(3, "CPE", "Computer Engineering",3),
        DatabaseSeeder.ProgramSeeder(4, "EE", "Electrical Engineering",3),
        DatabaseSeeder.ProgramSeeder(5, "ECE", "Electronics Engineering",3),
    };


    var schoolYears = new List<SchoolYear>
    {
        DatabaseSeeder.schoolYearSeeder(1,"2024","2025",Domain.Enum.Semester.Semester1,Domain.Enum.SchoolYearStatus.Current),
        DatabaseSeeder.schoolYearSeeder(2,"2024","2025",Domain.Enum.Semester.Semester2,Domain.Enum.SchoolYearStatus.Current),
        DatabaseSeeder.schoolYearSeeder(3,"2024","2025",Domain.Enum.Semester.Summer,Domain.Enum.SchoolYearStatus.Current),
    };


    var enrolledCourses = new List<EnrolledCourse>
    {
        //Enrolled courses in first semester
        DatabaseSeeder.EnrollCourseSeeder(1,"Engineering Calculus 1", "CEE101", 1, 1, 1),
        DatabaseSeeder.EnrollCourseSeeder(2,"PHYSICS 1 FOR ENGINEERS (CALCULUS BASED)", "CEE102/L", 1, 1, 2),
        DatabaseSeeder.EnrollCourseSeeder(3,"Statics of Rigid Bodies", "CEE108", 1, 1, 2),
        DatabaseSeeder.EnrollCourseSeeder(4,"Statics of Rigid Bodies", "CEE108", 1, 2, 2),


        //Enrolled courses in second semester
        DatabaseSeeder.EnrollCourseSeeder(5,"Engineering Calculus 2", "CEE103", 2, 1, 2),
        DatabaseSeeder.EnrollCourseSeeder(6,"Thermodyanmics 2", "CEE101", 2, 1, 1),
         DatabaseSeeder.EnrollCourseSeeder(7,"Data Structure and Algorithms", "CPE221/L", 2, 1, 2),

        //Enrolled courses in summer
        DatabaseSeeder.EnrollCourseSeeder(8,"Differential Equation", "CEE104", 3, 1, 2),


    };
```



```
    builder.Entity<Users>().HasData(users);
    builder.Entity<Department>().HasData(departments);
    builder.Entity<Program>().HasData(program);
    builder.Entity<SchoolYear>().HasData(schoolYears);
    builder.Entity<EnrolledCourse>().HasData(enrolledCourses);
    builder.Entity<Student>().HasData(students);
    builder.Entity<Faculty>().HasData(faculty);
    base.OnModelCreating(builder);
}
```

```
var students = new List<Student>
{
    DatabaseSeeder.StudentSeeder(1,"550200","Cedric Setimo","CedricSetimo.550200@umindanao.edu.ph",3,1,
    "273F528F-5330-411F-9C6B-01543D6249C3"),
     DatabaseSeeder.StudentSeeder(2,"547343","Ellaine Musni","EllaineMusni.550200@umindanao.edu.ph",3,1,
    "D0B26692-E380-4374-985F-239B56D06C20"),
};

var faculty = new List<Faculty>()
{
    DatabaseSeeder.FacultySeeder(1, "321033", "Rey Mateo", 1,"53D8F920-EBEC-4DF3-8C53-21F6D123F0D9"),
    DatabaseSeeder.FacultySeeder(2, "797132", "Jeanelle Labsan",2,"78B4AF2A-672F-43C5-B819-5F0B407B7187"),
};
```

Register DbContext in Program.cs

- *Go to your **Program.cs** and register the **AppDbContext** with the service container:*

```
},
"ConnectionStrings": {
  "DefaultConnection": "Server=tcp:consultationserver.database.windows.net,1433;Initial Catalog=ConsultationDatabaseTesting2" +
    ";Persist Security Info=False;User ID=Mymember1;Password=Password123;MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=False;Connection Timeout=30;"
},
```

```
builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));
```

Migrations & Database Update

- ***In Visual Studio (Windows)***
  - *Go to Tools > NuGet Package Manager > Package Manager Console, Type:*

```
Add-Migration InitialCreate
```

  - *After Add-Migration, Create Database, Type:*

```
Update-Database
```

- ***In Visual Studio Code***
  - *Type:*

```
dotnet ef migrations add InitialCreate
```

  - *After Add-Migration, Create Database, Type:*

VI. Data Seeding & Retrieval

Create a new class file and name it, DatabaseSeeder.cs. In the DatabaseSeeder.cs file, create each **static method** for entity of **Users**, **Department**, **Program**, **SchoolYear**, **EnrolledCourse**, **Student**, **Faculty**.

- Example for Users

```
public static Users UserSeeder(string id, string umid, string userName, string email, string password, Domain.Enum.UserType userType)
{
    var Users = new Users
    {
        Id = id,
        UserName = userName,
        Email = email,
        NormalizedUserName = userName.ToUpper(),
        NormalizedEmail = email.ToUpper(),
        EmailConfirmed = true,
        PasswordHash = new PasswordHasher<Users>().HashPassword(null, password),
        SecurityStamp = "5a54c967-0b1f-4c38-bda7-5f94e4c1a3f4",
        ConcurrencyStamp = "8d3ef0d9-b045-4b8f-a18f-15f2cbfa219b",
        UserType = userType,
        UMID = umid
    };
    return Users;
}
```

- Use this Class and its methods in **AppDbContext.cs** for seeding Data. Ex **Figure A.**

```
0 references
protected override void OnModelCreating(ModelBuilder builder)
{
    var users = new List<Users>
    {
    DatabaseSeeder.UserSeeder("273F528F-5330-411F-9C6B-01543D6249C3", "550200", "Cedric Setimo", "CedricSetimo.550200@umindanao.edu.ph", "MyStudent123!", Domain.Enum.UserType.Student),
    DatabaseSeeder.UserSeeder("53D8F920-EBEC-4DF3-8C53-21F6D123F0D9", "321033", "Rey Mateo", "ReyMateo.550200@umindanao.edu.ph", "MyFaculty123!", Domain.Enum.UserType.Faculty),
    DatabaseSeeder.UserSeeder("6B187E9D-FD71-4F1D-AFDF-EA1D91E818EF", "444533", "Raine Isid", "RaineIsid.550200@umindanao.edu.ph", "MyAdmin123!", Domain.Enum.UserType.Admin),
    DatabaseSeeder.UserSeeder("D0B26692-E380-4374-985F-239B56D06C20", "547343", "Ellaine Musni", "EllaineMusni.550200@umindanao.edu.ph", "MyAdmin123!", Domain.Enum.UserType.Student),
    DatabaseSeeder.UserSeeder("1226920F-9508-44B3-845A-ABABBBCBCF5D", "685043", "Reggie Soylon", "ReggieSoylon.6850@umindanao.edu.ph", "MyStudent123!", Domain.Enum.UserType.Student),
    DatabaseSeeder.UserSeeder("0A52E15B-95E6-40FE-9110-9A44817BFF39", "899812", "Cheley Balsomo", "CheleyBalsomo.8998@umindanao.edu.ph", "MyStudent123!", Domain.Enum.UserType.Student),
    DatabaseSeeder.UserSeeder("78B4AF2A-672F-43C5-B819-5F0B407B7187", "797132", "Jeanelle Labsan", "JeanelleLabsan.7971@umindanao.edu.ph", "MyFaculty123!", Domain.Enum.UserType.Faculty),
    DatabaseSeeder.UserSeeder("59CF8531-68E4-466B-BAEC-45305FE16A14", "924132", "Christopher Destajo", "ChristopherDestajo.9241@umindanao.edu.ph", "MyStudent123!", Domain.Enum.UserType.Student)
    };

    var departments = new List<Department>
    {
        DatabaseSeeder.DepartmentSeeder(1, "CASE", "College of Arts and Sciences Education"),
        DatabaseSeeder.DepartmentSeeder(2, "CBAE", "College of Business Administration Education"),
        DatabaseSeeder.DepartmentSeeder(3, "CEE", "College of Engineering Education")
    };
```

```
dotnet ef database update
```

I. TROUBLESHOOTING & DEBUGGING TIPS

I. Common Issues

This section outlines the common issues encountered during the development of the UMECA desktop application and the corresponding solutions applied to resolve them. The primary categories include database connection, authentication, UI controls, Entity Framework, and build or compilation errors.

1. Database Connection Errors

Issue: Unable to connect to LocalDB database

Error: A network-related or instance-specific error occurred while establishing a connection to SQL Server.

To fix LocalDB issues, first verify it's installed (sqllocaldb info) and start the instance (sqllocaldb start MSSQLLocalDB). Then, check the connection string in configuration files and ensure the firewall isn't blocking connections. Confirm the database exists in SQL Server Object Explorer, and if it's corrupted, recreate it by running Update-Database in the Package Manager Console.

Issue: Migration rrors referencing missing projects

Error: Unable to find project 'D:\Downloads\DataBaseUpdated-SQL_Database_Version_2.0\DataBaseUpdated-SQL_Database_Version_2.0\Consultation.Infrastructure\Consultation.Infrastructure.csproj

To fix build or reference issues, clean and rebuild the solution. Verify all project references in Solution Explorer and ensure file paths in the .csproj files are valid. Restore NuGet packages, delete the bin and obj folders, then rebuild the solution. Finally, confirm the Consultation.Infrastructure project is properly loaded.

2. Authentication Issues

Issue: Login fails with valid credentials

To resolve login issues, ensure the password hashing algorithm is consistent for registration and login. Verify the AspNetUsers table has correct PasswordHash values and that email comparison is case-insensitive. Confirm UserType enum values match the database. Check for null references in AuthService, enable detailed logging in AuthService.Login(), and test with known valid credentials from sample data.

Issue: Null reference exception on _context

To fix dependency injection issues, verify it's configured correctly in Program.cs or Startup.cs. Ensure the DbContext is registered in the service container and that AuthService receives an

AppDbContext parameter in its constructor. Also, confirm the correct namespace for AppDbContext is included with a proper using statement.

3. UI Control Errors

Issue: TextBox controls return null or empty values

Error: NullReferenceException when accessing txtEmail.Text

To fix control-related errors, ensure all controls are initialized in InitializeComponent() and their names match the property accessors (e.g., txtEmail, txtPassword). Confirm the form's Load event finishes before accessing controls, verify the controls exist in Login.Designer.cs, and check that the form isn't disposed before being accessed.

Issue: Event handlers not firing
To fix event handling issues, confirm the event handler is attached in the designer or constructor and that its signature matches the EventHandler delegate. Ensure button Click events are properly wired, use breakpoints to verify the handler executes, and check for suppressed exceptions inside try-catch blocks.

4. Entity Framework Errors

Issue: Navigation property returns null

Error: Object reference not set to an instance of an object (Admin.Users is null)

To fix data relationship or loading issues, use **.Include()** for eager loading (e.g., context.Admins.Include(a => a.Users)) or enable lazy loading with proxies. Ensure foreign key values are correctly set and relationships are properly configured in **OnModelCreating()**. If necessary, use explicit loading with context.Entry(admin).Reference(a => a.Users).Load().

Issue: Concurrency conflict during SaveChanges()

Solutions:

- Reload entity from database before updating
- Implement optimistic concurrency with [ConcurrencyCheck] attribute
- Use database transactions for critical updates
- Handle DbUpdateConcurrencyException appropriately

5. Build and Compilation Errors

Issue: 31 Errors in Error List

Solutions:

- Address missing project references first
- Fix broken project file paths in solution
- Restore NuGet packages completely
- Check for circular project dependencies
- Verify all using statements reference correct namespaces
- Clean solution and delete .vs folder, then rebuild

II. Debugging Strategies

During the development and testing phase, several debugging techniques were implemented to ensure the accuracy, performance, and stability of the system. These include:

1. Using Visual Studio Debugger

Breakpoints were strategically placed in critical methods (e.g., AuthService.Login()) to inspect variable states such as email, password, and user. The **Watch Window** was used to monitor real-time data and verify logic execution during runtime.

2. Logging Strategies

Console logging was used to display runtime information such as login attempts and validation results. File-based logging was implemented to record error messages and exceptions for post-run analysis using a custom LogError() method.

3. Database Debugging

Entity Framework (EF) Core query logging was enabled to trace SQL commands executed during runtime. Direct SQL queries were also executed to verify database integrity, relationships, and stored data accuracy.

4. Unit Test Debugging

Unit tests were created to verify individual components, such as the authentication service. Breakpoints were placed in test setup and execution to confirm data seeding, expected results, and test coverage.

5. Presenter/View Debugging

Breakpoints were inserted in the MVP (Model-View-Presenter) layers to trace user input handling, event triggers, and navigation logic between the view and the service.

6. Common Debugging Commands

Common Entity Framework Core debugging commands were used to manage and inspect migrations and database updates:

- Get-Migration — lists migrations
- Script-Migration — generates SQL scripts
- Remove-Migration — rolls back last migration
- Update-Database — updates schema
- Drop-Database — deletes the database

III. Unit Test Location And Usage

Test Project Structure:

```
Consultation.Desktop.Test/
├── AuthServiceTests.cs                # Authentication service tests
├── EditConsultationServiceTest.cs     # Consultation service tests
├── BulletinServiceTests.cs            # Bulletin service tests
├── BulletinRepositoryTests.cs         # Bulletin repository tests
├── ConsultationRepositoryTests.cs     # Consultation repository tests
└── ConsultationView.cs                # View component tests
```

Running Tests:

From Visual Studio:

1. Open Test Explorer: Test → Test Explorer
2. Click "Run All Tests" or right-click specific test
3. View test results in Test Explorer window
4. Double-click failed tests to navigate to code

Test Execution Best Practices:

- Run tests after every significant code change
- Use [SetUp] to initialize test environment consistently
- Use [TearDown] to clean up resources and dispose contexts
- Use in-memory database for unit tests (faster than LocalDB)
- Mock external dependencies for true unit testing
- Keep tests independent - each test should be runnable in isolation
- Use meaningful test names following MethodName_Scenario_ExpectedResult pattern
- Assert both positive and negative test cases
- Verify null checks and edge cases

J. VERSION CONTROL & BUILD INSTRUCTIONS

I. Git Usage Guidelines

Repository Information:

- **Remote URL:** https://github.com/tortolsaur/proto-sd-project.git
- **Version Control System:** Git
- **Hosting Platform:** GitHub

Branch Strategy:

- **Main Branches:**
  - main # *Production-ready code*
  - develop # *Integration branch for features*
- **Feature Branches:**
  - feature/authentication # *Authentication feature*
  - feature/consultation-management # *Consultation management*
  - feature/bulletin-board # *Bulletin board functionality*
  - feature/user-management # *User management features*
- **Bugfix Branches:**
  - bugfix/login-validation # *Fix login validation issues*
  - bugfix/database-connection # *Fix database connection errors*
- **Common Git Commands:**

    1. Initial Setup:

  - git clone https://github.com/tortolsaur/proto-sd-project.git # *Clone repository*
  - cd proto-sd-project # *Navigate to project directory*
  - git config user.name "Your Name" # *Configure user information*
  - git config user.email "your.email@example.com"
  - git config –list # *View current configuration*

2. Daily Workflow:

- git status # *Check current status*
- git pull origin main # *Pull latest changes from remote*
- git checkout -b feature/your-feature-name # *Create new feature branch*
- git branch -a # *View all branches*
- git checkout develop # *Switch to existing branch*
- git add Consultation.App/Views/Login.cs # *Stage specific files*
- git add Consultation.BackEndCRUD/Service/AuthService.cs
- git add . # *Stage all changes*
- git commit -m "Add authentication service with password hashing" # *Commit with descriptive message*
- git push origin feature/your-feature-name # *Push branch to remote*
- git checkout develop # *Merge feature branch into develop*
- git merge feature/your-feature-name
- git branch -d feature/your-feature-name # *Delete local branch after merge*
- git push origin --delete feature/your-feature-name # *Delete remote branch*

- **3. Viewing History:**
  - git log # *View commit history*
  - git log --oneline --graph –all # *View compact history*
  - git show <commit-hash> # *View changes in specific commit*
  - git log --follow Consultation.Domain/Entities/Admin.cs # *View changes in specific file*
  - git blame Consultation.App/Views/Login.cs # *View who changed what*
- **4. Handling Conflicts:**
  - git pull --rebase origin main # *Pull with rebase to avoid merge commits*
  - git status # *If conflicts occur, view conflicted files*
  - git add <resolved-file> # *Edit conflicted files manually, then:*

- git rebase --continue
- git rebase –abort *# Or abort rebase if needed*
- git merge feature/your-feature *# Merge with conflict resolution*
- git add . *# Fix conflicts, then:*
- git commit -m "Resolve merge conflicts"

- **5. Stashing Changes:**
  - git stash save "Work in progress on login feature" *# Stash current changes*
  - git stash list *# View stash list*
  - git stash apply *# Apply most recent stash*
  - git stash pop *# Apply and remove stash*
  - git stash apply stash@{1} *# Apply specific stash*
  - git stash drop stash@{0} *# Drop specific stash*
  - git stash clear *# Clear all stashes*
- **6. Undoing Changes:**
  - git checkout -- Consultation.App/Views/Login.cs *# Discard changes in working directory*
  - git reset HEAD Consultation.Domain/Entities/Admin.cs *# Unstage file*
  - git commit --amend -m "Updated commit message" *# Amend last commit*
  - git revert <commit-hash> *# Revert specific commit (creates new commit)*
  - git reset --hard <commit-hash> *# Reset to specific commit (dangerous - loses history)*
  - git reset --soft HEAD~1 *# Reset but keep changes in working directory*

Best Practices:

- Commit frequently with meaningful messages
- Pull before starting work each day
- Never commit sensitive data (passwords, connection strings)
- Use feature branches for new development
- Keep commits atomic (one logical change per commit)

- Review changes before committing (git diff)
- Write descriptive commit messages
- Use pull requests for code review before merging to main
- Tag releases with semantic versioning (v1.0.0, v1.1.0, etc.)
- Regularly sync with remote repository
- Don't commit generated files (bin, obj, .vs)

II. Build Tools And Commands

1. MSBuild (Visual Studio):

Solution Build:

Build entire solution in Debug mode
msbuild proto-sd-project.sln /p:Configuration=Debug

Build in Release mode
msbuild proto-sd-project.sln /p:Configuration=Release

Clean solution
msbuild proto-sd-project.sln /t:Clean

Rebuild (Clean + Build)
msbuild proto-sd-project.sln /t:Rebuild

Build specific project
msbuild Consultation.App/Consultation.App.csproj /p:Configuration=Debug

Build with maximum CPU cores
msbuild proto-sd-project.sln /m

Build with verbose output
msbuild proto-sd-project.sln /v:detailed

Build and generate log file
msbuild proto-sd-project.sln /fl /flp:logfile=build.log

2. .NET CLI (Cross-platform):

Basic Build Commands:

Build entire solution in Debug mode
msbuild proto-sd-project.sln /p:Configuration=Debug

Build in Release mode
msbuild proto-sd-project.sln /p:Configuration=Release

Clean solution
msbuild proto-sd-project.sln /t:Clean

Rebuild (Clean + Build)
msbuild proto-sd-project.sln /t:Rebuild

Build specific project
msbuild Consultation.App/Consultation.App.csproj /p:Configuration=Debug

Build with maximum CPU cores
msbuild proto-sd-project.sln /m

Build with verbose output
msbuild proto-sd-project.sln /v:detailed

Build and generate log file
msbuild proto-sd-project.sln /fl /flp:logfile=build.log

3. NuGet Package Management:

Package Manager Console:

Install package
Install-Package Microsoft.EntityFrameworkCore

Install specific version
Install-Package Microsoft.EntityFrameworkCore -Version 7.0.0

Update package
Update-Package Microsoft.EntityFrameworkCore

Uninstall package
Uninstall-Package Microsoft.EntityFrameworkCore

Restore all packages
Update-Package -Reinstall

List installed packages
Get-Package

4. .NET CLI for Packages:

Add package reference
dotnet add package Microsoft.EntityFrameworkCore

Add specific version
dotnet add package Microsoft.EntityFrameworkCore --version 7.0.0

Remove package
dotnet remove package Microsoft.EntityFrameworkCore

List packages
dotnet list package

Update packages
dotnet add package Microsoft.EntityFrameworkCore

Restore packages
dotnet restore

5. Database Build Commands:

Add new migration
dotnet ef migrations add MigrationName --project Consultation.Infrastructure

Update database
dotnet ef database update --project Consultation.Infrastructure

Revert to specific migration
dotnet ef database update PreviousMigrationName --project Consultation.Infrastructure

Generate SQL script
dotnet ef migrations script --project Consultation.Infrastructure

Remove last migration
dotnet ef migrations remove --project Consultation.Infrastructure

Drop database
dotnet ef database drop --project Consultation.Infrastructure

Package Manager Console:

Add migration
Add-Migration MigrationName -Project Consultation.Infrastructure

Update database

Update-Database -Project Consultation.Infrastructure

Generate SQL script
Script-Migration -From InitialCreate -To AddNewTable

Remove migration
Remove-Migration -Project Consultation.Infrastructure

Build Best Practices:

- Always restore packages before building
- Use Release configuration for production deployments
- Run tests before pushing code
- Apply database migrations as part of deployment
- Use build scripts for consistency across environments
- Version your builds using semantic versioning
- Keep build logs for troubleshooting
- Use incremental builds during development (/m flag)
- Clean solution when switching configurations
- Verify database migrations before deploying

END OF PROGRAMMING MANUAL

This comprehensive programming manual provides all necessary technical documentation for the UMECA system. For additional support or clarification, please refer to the project repository at https://github.com/tortolsaur/proto-sd-project.git or contact the development team.