# LAB 0 – .NET & EF Core Environment Setup

**Topic: Setting Up the Development Environment**

## Lab Objectives

Before starting any Lab on ASP.NET Core MVC and Entity Framework Core (EF Core), students MUST have a development environment that is correctly installed and configured.

This Lab 0 will guide you to:

1. Install **Visual Studio 2022** with the correct Workloads.
2. Ensure the **.NET 8 SDK** (Software Development Kit) is installed.
3. Confirm the installation of **SQL Server LocalDB** (to run the Database).
4. Install the **Entity Framework Core** command-line tools (Tools).

## Part 1: Install Visual Studio 2022

This is our main Integrated Development Environment (IDE).

1. **Download:** Go to the Visual Studio homepage and download **Visual Studio 2022 Community**. This version is free for students and individual developers.
    - Link: https://visualstudio.microsoft.com/downloads/
2. **Install Workloads (IMPORTANT):**
    - After running the installer, the Visual Studio Installer will appear.
    - In the **Workloads** tab, you MUST check the following 2 workloads:
        - **ASP.NET and web development**: Provides everything for ASP.NET Core web development, including the .NET SDK.
        - **Data storage and processing**: Provides tools for working with databases, including SQL Server Express LocalDB.
    - (Optional) You should also select **.NET desktop development** if you are learning WinForms/WPF programming.
3. **Complete Installation:** Click **Install** (or Modify if already installed) and wait for the process to finish.

## Part 2: Check .NET SDK

Usually, the "ASP.NET" workload above has already installed the .NET 8 SDK. We will double-check.

1. Open **Command Prompt** (type cmd in the Start Menu).
2. Type the following command and press Enter:
   dotnet --version

3. **Result:** If you see a version starting with 8.0.x (e.g., 8.0.300), you have successfully installed it.
4. **If it fails:** If the command doesn't run or shows an old version, you need to install the .NET 8 SDK manually from the link below (choose the SDK, not the Runtime):
    - Link: https://dotnet.microsoft.com/download/dotnet/8.0

# Part 3: Check SQL Server LocalDB

To run the labs, we need a database. LocalDB is a lightweight version of SQL Server, installed with Visual Studio (when selecting the "Data storage" workload).

1. Open **Visual Studio 2022**.
2. On the menu bar, select **View → SQL Server Object Explorer**.
3. A new window will appear (usually on the left).
4. You will see a SQL Server item. Expand it.
5. **Result:** You MUST see a server named **(localdb)\mssqllocaldb**.
    - This is the server we will use in appsettings.json (e.g., Server=(localdb)\\mssqllocaldb;...).
    - If you don't see it, you need to go back to the Visual Studio Installer (in Part 1) and ensure you have selected the "Data storage and processing" workload.

# Part 4: Install EF Core Tools

Entity Framework Core (EF Core) needs command-line tools to perform tasks like Add-Migration (create DB blueprint) and Update-Database (apply the blueprint).

There are 2 toolsets we need to know:

## 4.1. Global Tools (dotnet-ef)

This tool runs in the Command Prompt (cmd) or Terminal.

1. Open **Command Prompt**.
2. Run the following command to install (or update if already installed):
   dotnet tool install --global dotnet-ef

   If it's already installed, you can run the following command to update to the latest version:
   dotnet tool update --global dotnet-ef

3. **Check:** Type the command dotnet-ef --version. If it shows a version (e.g., 8.0.0), you are successful.

## 4.2. Package Manager Console Tools

These are tools that run inside Visual Studio. We don't need to *install* them in the environment,

but we need to *install* the NuGet packages into **each project** where we want to use EF Core.

When you do **Lab 1** or **Lab 2**, you will be asked to install these packages into your project:

1. Microsoft.EntityFrameworkCore.SqlServer: Library for EF Core to communicate with SQL Server.
2. Microsoft.EntityFrameworkCore.Tools: Provides commands for the **Package Manager Console (PMC)** inside Visual Studio (e.g., Add-Migration, Update-Database).
3. Microsoft.AspNetCore.Identity.EntityFrameworkCore: (Used for Lab 1) Provides the tables for the User Management system (Identity).

You just need to be aware of this. You will practice installing them in Lab 1.

## 9. Completion Checklist (Mandatory)

Make sure you have completed all the following items before starting Lab 1:

- [ ] Successfully installed **Visual Studio 2022 Community**.
- [ ] Selected the **"ASP.NET..."** and **"Data storage..."** workloads during installation.
- [ ] Running dotnet --version shows version **8.0.x**.
- [ ] Running dotnet-ef --version shows a version (e.g., **8.0.x**).
- [ ] Opening **SQL Server Object Explorer** in Visual Studio shows **(localdb)\mssqllocaldb**.

If you have completed these 5 items, you are READY for the next labs!

*End of Lab 0*

# LAB 1 Practice – ASP.NET Core Identity & Role Management

**Topic: Authentication (Login) & Authorization (Admin Role Management)**

## Lab Objectives

After completing this lab, students MUST be able to:

1. Create a new ASP.NET Core MVC project with **Individual Accounts** (ASP.NET Core Identity).
2. Customize the ApplicationUser model to add custom fields (e.g., FullName).
3. Seed 3 Roles (Admin, Faculty, Student) and a default Admin user on startup.
4. Implement a secure **Login** page for all users (public).
5. Implement an **Admin-only** section to **Register** new users.
6. Implement **Validation** for Login and Register forms using **ViewModels**.

7. Build an **Admin-only** page to manage users and **Assign Roles**.

# 0. Prerequisites

- Completed **Lab 0**.
- Visual Studio 2022.
- .NET 8.
- SQL Server LocalDB.

# Part 1: Create Project with ASP.NET Core Identity

This is the most critical step. We are **not** using Authentication: None.

1. Open Visual Studio → **Create a new project**.
2. Select template: **ASP.NET Core Web App (Model-View-Controller)**.
3. Name it:
    - Project name: StudentManagementMvc
    - Solution name: StudentManagementMvc
4. Click **Next**.
5. On the "Additional information" screen:
    - Framework: **.NET 8.0**
    - **Authentication type: Change from None to Individual Accounts**.
6. Click **Create**.
7. Run the project (Ctrl+F5). You will see "Register" and "Login" links in the header. We will customize this.

# Part 2: Configure and Customize Identity

## 2.1. Configure Connection String

When you created the project with Identity, VS already configured a connection string. Let's verify it.

1. Open appsettings.json.
2. You will see a ConnectionStrings section, typically named DefaultConnection. We can rename the Database to match our plan.
3. **Update** the ConnectionStrings to:

```
{
  "ConnectionStrings": {
    "DefaultConnection":
"Server=(localdb)\\mssqllocaldb;Database=StudentManagementMvcDb;Trusted_Connection=
True;MultipleActiveResultSets=true"
  },
  "Logging": {
    // ...
```

```
  },
  "AllowedHosts": "*"
}
```

## 2.2. Customize the ApplicationUser Model

Let's add a FullName field to the user table.

1. Create a Models folder (if it doesn't exist).
2. Right-click the Models folder → **Add** → **Class...**.
3. Name it ApplicationUser.cs and add this content:

```
using Microsoft.AspNetCore.Identity;
using System.ComponentModel.DataAnnotations;

namespace StudentManagementMvc.Models
{
    // Add profile data for application users by adding properties to the ApplicationUser class
    public class ApplicationUser : IdentityUser
    {
        [Required]
        [StringLength(100)]
        public string FullName { get; set; }
    }
}
```

## 2.3. Update ApplicationDbContext and Program.cs

1. Open Data/ApplicationDbContext.cs.
2. Change it to inherit from IdentityDbContext<ApplicationUser> (our custom class) instead of just IdentityDbContext.

```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
using StudentManagementMvc.Models; // Add this using

namespace StudentManagementMvc.Data
{
    // Change this line
    public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
    {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
            : base(options)
        {
```

```
    }
  }
}
```

3. Open Program.cs.
4. Find the line builder.Services.AddDefaultIdentity.... We need to change this to use our ApplicationUser and add support for Roles.
5. **Replace** builder.Services.AddDefaultIdentity... with:

```
// Use ApplicationUser and add Role services
builder.Services.AddIdentity<ApplicationUser, IdentityRole>(options =>
options.SignIn.RequireConfirmedAccount = false)
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddDefaultTokenProviders()
    .AddDefaultUI(); // Add this to support Razor Pages for Identity
```

6. Still in Program.cs, **remove** the line builder.Services.AddRazorPages(); if it exists (it's now included in AddDefaultUI()).

### 2.4. Run Identity Migration

1. Open **Package Manager Console** (Tools → NuGet Package Manager → PMC).
2. Run Add-Migration to create the migration for our FullName field and Identity tables.

```
Add-Migration InitialIdentitySchema
```

3. Apply the migration to the database:

```
Update-Database
```

4. Check your **SQL Server Object Explorer**. You will see the StudentManagementMvcDb with all the AspNet... tables (AspNetUsers, AspNetRoles, etc.). The AspNetUsers table will have your FullName column.

## Part 3: Seed Roles and Default Admin User

We need to create the 3 roles and an Admin user automatically.

1. Create a new folder Services (or Data).
2. Add a new class DbInitializer.cs:

```
using Microsoft.AspNetCore.Identity;
using StudentManagementMvc.Models;
```

```
namespace StudentManagementMvc.Services
{
    public static class DbInitializer
    {
        public static async Task SeedRolesAndAdminAsync(IServiceProvider service)
        {
            // Get required services
            var userManager = service.GetRequiredService<UserManager<ApplicationUser>>();
            var roleManager = service.GetRequiredService<RoleManager<IdentityRole>>();

            // Define roles
            string[] roleNames = { "Admin", "Faculty", "Student" };

            // Create roles if they don't exist
            foreach (var roleName in roleNames)
            {
                if (!await roleManager.RoleExistsAsync(roleName))
                {
                    await roleManager.CreateAsync(new IdentityRole(roleName));
                }
            }

            // --- Create a default Admin User ---
            var adminUser = await userManager.FindByEmailAsync("admin@example.com");

            if (adminUser == null)
            {
                adminUser = new ApplicationUser
                {
                    UserName = "admin@example.com",
                    Email = "admin@example.com",
                    FullName = "Administrator",
                    EmailConfirmed = true // Confirm email immediately
                };

                // Create user with a password
                var result = await userManager.CreateAsync(adminUser, "Admin@123");

                if (result.Succeeded)
                {
                    // Assign the 'Admin' role to the new user
                    await userManager.AddToRoleAsync(adminUser, "Admin");
                }
```

```
        }
      }
    }
}
```

3. Call this initializer from Program.cs. Find the line var app = builder.Build(); and add the following code block *after* it, and *before* app.Run():

```
// --- Start Seed Data code ---
using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;
    try
    {
        // Call the DbInitializer
        await
StudentManagementMvc.Services.DbInitializer.SeedRolesAndAdminAsync(services);
    }
    catch (Exception ex)
    {
        var logger = services.GetRequiredService<ILogger<Program>>();
        logger.LogError(ex, "An error occurred seeding the DB.");
    }
}
// --- End Seed Data code ---

// ... app.Use...
app.Run();
```

4. Run the project (Ctrl+F5). The app will start, create the 3 roles, and the admin@example.com user (Password: Admin@123).

# Part 4: Scaffold and Customize Identity Pages

By default, the Identity pages are compiled in a library. We need to scaffold them to customize them.

1. **Stop** the project.
2. In **Command Prompt** (or **Package Manager Console**), run this command to scaffold the Identity files into your project:

```
dotnet aspnet-codegenerator identity -dc
StudentManagementMvc.Data.ApplicationDbContext
```

3. This will create a new Areas/Identity folder in your project.
4. **Crucial Requirement:** The Admin manages registration. So, we must **delete** the public registration pages.
    - In Solution Explorer, go to Areas/Identity/Pages/Account/.
    - **Delete** Register.cshtml.
    - **Delete** Register.cshtml.cs.
    - **Delete** RegisterConfirmation.cshtml.
    - **Delete** RegisterConfirmation.cshtml.cs.
5. Now, if you run the app, the "Register" link will be broken (which is what we want). Let's fix the layout.

# Part 5: Update Layout and Login Page

## 5.1. Clean up _Layout.cshtml

1. Open Views/Shared/_Layout.cshtml.
2. Find the _LoginPartial.cshtml. It handles showing "Register" and "Login" links.

## 5.2. Clean up _LoginPartial.cshtml

1. Open Views/Shared/_LoginPartial.cshtml.
2. **Delete** the <li> element for "Register" as we no longer have a public registration page.

```
@using Microsoft.AspNetCore.Identity
@using StudentManagementMvc.Models
@inject SignInManager<ApplicationUser> SignInManager
@inject UserManager<ApplicationUser> UserManager

<ul class="navbar-nav">
@if (SignInManager.IsSignedIn(User))
{
    <li class="nav-item">
        <a  class="nav-link text-dark" asp-area="Identity" asp-page="/Account/Manage/Index"
title="Manage">Hello @(((ApplicationUser)User).FullName)!</a>
    </li>
    <li class="nav-item">
        <form  class="form-inline" asp-area="Identity" asp-page="/Account/Logout"
asp-route-returnUrl="@Url.Action("Index", "Home", new { area = "" })">
            <button  type="submit" class="nav-link btn btn-link text-dark">Logout</button>
        </form>
    </li>
}
else
```

```
{
    <!-- THIS IS THE ITEM TO DELETE
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="Identity"
asp-page="/Account/Register">Register</a>
    </li>
    -->
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="Identity" asp-page="/Account/Login">Login</a>
    </li>
}
</ul>
```

*Note: I also updated the "Hello" message to use FullName.*

3. **Test Login:** Run the app. Login with admin@example.com and Admin@123. It should work.

# Part 6: Create Admin-Only User Management (ViewModels)

Now, we create the Admin's section to Register and Manage users.

## 6.1. Create ViewModels

1. Create a new folder ViewModels.
2. Add class RegisterViewModel.cs:

```
using System.ComponentModel.DataAnnotations;

namespace StudentManagementMvc.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        public string Email { get; set; }

        [Required]
        [Display(Name = "Full Name")]
        public string FullName { get; set; }

        [Required]
        [DataType(DataType.Password)]
```

```
        public string Password { get; set; }

        [DataType(DataType.Password)]
        [Display(Name = "Confirm Password")]
        [Compare("Password", ErrorMessage = "The password and confirmation password do not
match.")]
        public string ConfirmPassword { get; set; }

        [Required]
        [Display(Name = "Role")]
        public string SelectedRole { get; set; }
    }
}
```

3. Add class UserListViewModel.cs:

```
namespace StudentManagementMvc.ViewModels
{
    public class UserListViewModel
    {
        public string UserId { get; set; }
        public string Email { get; set; }
        public string FullName { get; set; }
        public string Roles { get; set; } // Comma-separated list of roles
    }
}
```

4. Add class ManageUserRolesViewModel.cs:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace StudentManagementMvc.ViewModels
{
    public class ManageUserRolesViewModel
    {
        public string UserId { get; set; }
        public string Email { get; set; }

        // We will populate this list with all roles
        public List<RoleViewModel> Roles { get; set; } = new List<RoleViewModel>();
    }
```

```
    public class RoleViewModel
    {
        public string RoleName { get; set; }
        public bool IsSelected { get; set; }
    }
}
```

# Part 7: Create Admin Controller and Views

## 7.1. Create AdminController.cs

1. Right-click Controllers → **Add** → **Controller...**.
2. Select **MVC Controller - Empty**.
3. Name it AdminController.cs.
4. Add the [Authorize(Roles = "Admin")] attribute to the class.
5. Inject UserManager, RoleManager.

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using StudentManagementMvc.Models;
using StudentManagementMvc.ViewModels;
using System.Data;

namespace StudentManagementMvc.Controllers
{
    // This entire controller is only accessible by users in the "Admin" role
    [Authorize(Roles = "Admin")]
    public class AdminController : Controller
    {
        private readonly UserManager<ApplicationUser> _userManager;
        private readonly RoleManager<IdentityRole> _roleManager;

        public AdminController(UserManager<ApplicationUser> userManager,
RoleManager<IdentityRole> roleManager)
        {
            _userManager = userManager;
            _roleManager = roleManager;
        }

        // GET: Admin/Index (List all users)
```

```
public async Task<IActionResult> Index()
{
    var users = await _userManager.Users.ToListAsync();
    var userViewModels = new List<UserListViewModel>();

    foreach (var user in users)
    {
        var roles = await _userManager.GetRolesAsync(user);
        userViewModels.Add(new UserListViewModel
        {
            UserId = user.Id,
            Email = user.Email,
            FullName = user.FullName,
            Roles = string.Join(", ", roles)
        });
    }
    return View(userViewModels);
}

// GET: Admin/Register
public async Task<IActionResult> Register()
{
    // Pass roles to the view
    ViewBag.Roles = await _roleManager.Roles.Select(r => r.Name).ToListAsync();
    return View(new RegisterViewModel());
}

// POST: Admin/Register
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser
        {
            UserName = model.Email,
            Email = model.Email,
            FullName = model.FullName,
            EmailConfirmed = true // Admin creates, so auto-confirm
        };

        var result = await _userManager.CreateAsync(user, model.Password);
```

```
        if (result.Succeeded)
        {
            // Add to selected role
            if (await _roleManager.RoleExistsAsync(model.SelectedRole))
            {
                await _userManager.AddToRoleAsync(user, model.SelectedRole);
            }
            return RedirectToAction("Index");
        }
        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(string.Empty, error.Description);
        }
    }

    // If we got this far, something failed, redisplay form
    ViewBag.Roles = await _roleManager.Roles.Select(r => r.Name).ToListAsync();
    return View(model);
}

// GET: Admin/ManageRoles/GUID
public async Task<IActionResult> ManageRoles(string userId)
{
    var user = await _userManager.FindByIdAsync(userId);
    if (user == null)
    {
        return NotFound();
    }

    var viewModel = new ManageUserRolesViewModel
    {
        UserId = user.Id,
        Email = user.Email
    };

    var allRoles = await _roleManager.Roles.ToListAsync();
    foreach (var role in allRoles)
    {
        viewModel.Roles.Add(new RoleViewModel
        {
            RoleName = role.Name,
            IsSelected = await _userManager.IsInRoleAsync(user, role.Name)
```

```
        });
    }

    return View(viewModel);
}

// POST: Admin/ManageRoles
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> ManageRoles(ManageUserRolesViewModel model)
{
    var user = await _userManager.FindByIdAsync(model.UserId);
    if (user == null)
    {
        return NotFound();
    }

    // Get all current roles for the user
    var userRoles = await _userManager.GetRolesAsync(user);

    // Loop through the roles submitted in the form
    foreach (var roleViewModel in model.Roles)
    {
        if (roleViewModel.IsSelected)
        {
            // If selected and user is not already in this role, add them
            if (!await _userManager.IsInRoleAsync(user, roleViewModel.RoleName))
            {
                await _userManager.AddToRoleAsync(user, roleViewModel.RoleName);
            }
        }
        else
        {
            // If not selected and user is in this role, remove them
            if (await _userManager.IsInRoleAsync(user, roleViewModel.RoleName))
            {
                await _userManager.RemoveFromRoleAsync(user, roleViewModel.RoleName);
            }
        }
    }

    return RedirectToAction("Index");
}
```

```
    }
}
```

## 7.2. Create Admin Views

1. Create a folder Views/Admin.
2. Add Views/Admin/Index.cshtml (List Users):

```
@model IEnumerable<StudentManagementMvc.ViewModels.UserListViewModel>

@{
    ViewData["Title"] = "User Management";
}

<h1>User Management</h1>

<p>
    <a asp-action="Register" class="btn btn-primary">Create New User</a>
</p>

<table class="table">
    <thead>
        <tr>
            <th>Email</th>
            <th>Full Name</th>
            <th>Roles</th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var user in Model)
        {
            <tr>
                <td>@user.Email</td>
                <td>@user.FullName</td>
                <td>@user.Roles</td>
                <td>
                    <a asp-action="ManageRoles" asp-route-userId="@user.UserId" class="btn
btn-secondary btn-sm">Manage Roles</a>
                    <!-- Add Edit/Delete user links here if needed -->
                </td>
            }
        }
```

```
    </tbody>
</table>
```

3. Add Views/Admin/Register.cshtml (Register User Form):

```
@model StudentManagementMvc.ViewModels.RegisterViewModel
@{
    ViewData["Title"] = "Register New User";
    var roles = ViewBag.Roles as List<string>;
}

<h1>@ViewData["Title"]</h1>

<div class="row">
    <div class="col-md-4">
        <form asp-action="Register" method="post">
            <h4>Create a new account.</h4>
            <hr />
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>

            <div class="form-floating mb-3">
                <input asp-for="Email" class="form-control" />
                <label asp-for="Email"></label>
                <span asp-validation-for="Email" class="text-danger"></span>
            </div>
            <div class="form-floating mb-3">
                <input asp-for="FullName" class="form-control" />
                <label asp-for="FullName"></label>
                <span asp-validation-for="FullName" class="text-danger"></span>
            </div>
            <div class="form-floating mb-3">
                <input asp-for="Password" class="form-control" />
                <label asp-for="Password"></label>
                <span asp-validation-for="Password" class="text-danger"></span>
            </div>
            <div class="form-floating mb-3">
                <input asp-for="ConfirmPassword" class="form-control" />
                <label asp-for="ConfirmPassword"></label>
                <span asp-validation-for="ConfirmPassword" class="text-danger"></span>
            </div>

            <div classs="form-floating mb-3">
                <label asp-for="SelectedRole"></label>
```

```
            <select asp-for="SelectedRole" class="form-control"
                    asp-items="@(new SelectList(roles))">
                <option value="">-- Select Role --</option>
            </select>
            <span asp-validation-for="SelectedRole" class="text-danger"></span>
        </div>

        <button type="submit" class="w-100 btn btn-lg btn-primary mt-3">Register</button>
    </form>
  </div>
</div>

@section Scripts {
  <partial name="_ValidationScriptsPartial" />
}
```

4. Add Views/Admin/ManageRoles.cshtml (Assign Roles Form):

```
@model StudentManagementMvc.ViewModels.ManageUserRolesViewModel
@{
  ViewData["Title"] = "Manage Roles";
}

<h1>Manage Roles for @Model.Email</h1>

<div class="row">
  <div class="col-md-6">
    <form asp-action="ManageRoles" method="post">
      <div asp-validation-summary="ModelOnly" class="text-danger"></div>
      <input type="hidden" asp-for="UserId" />

      <h4>Assign Roles:</h4>

      @for (int i = 0; i < Model.Roles.Count; i++)
      {
        <div class="form-check m-1">
          <input type="hidden" asp-for="Roles[i].RoleName" />
          <input asp-for="Roles[i].IsSelected" class="form-check-input" />
          <label class="form-check-label" asp-for="Roles[i].IsSelected">
            @Model.Roles[i].RoleName
          </label>
        </div>
      }
```

```
            <div class="form-group mt-3">
                <input type="submit" value="Save" class="btn btn-primary" />
                <a asp-action="Index" class="btn btn-secondary">Cancel</a>
            </div>
        </form>
    </div>
</div>
```

## Part 8: Add Admin Link to Layout

Finally, let's add a link to our new Admin section in the main layout, visible only to Admins.

1. Open Views/Shared/_Layout.cshtml.
2. Find the <ul class="navbar-nav flex-grow-1"> section.
3. Add the "Admin" link, wrapped in an if check.

```
<div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
    <ul class="navbar-nav flex-grow-1">
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="" asp-controller="Home"
asp-action="Index">Home</a>
        </li>
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="" asp-controller="Home"
asp-action="Privacy">Privacy</a>
        </li>

        <!-- Add this Admin link -->
        @if (User.IsInRole("Admin"))
        {
            <li class="nav-item">
                <a class="nav-link text-danger" asp-area="" asp-controller="Admin"
asp-action="Index">Admin Management</a>
            </li>
        }
    </ul>

    <!-- This should already be here -->
    <partial name="_LoginPartial" />
</div>
```

## 9. Completion Checklist (Mandatory)

Review your work against the following checklist:

- [ ] Project created with **Authentication: Individual Accounts**.
- [ ] ApplicationUser class created with FullName.
- [ ] ApplicationDbContext inherits from IdentityDbContext<ApplicationUser>.
- [ ] Program.cs registers Identity<ApplicationUser, IdentityRole>.
- [ ] Ran Add-Migration and Update-Database successfully.
- [ ] DbInitializer created and seeds 3 Roles (Admin, Faculty, Student) and 1 Admin user.
- [ ] Public Register pages are **deleted**.
- [ ] _LoginPartial.cshtml no longer shows the "Register" link.
- [ ] Can **Login** as admin@example.com (Password: Admin@123).
- [ ] **Admin Management** link is visible *only* when logged in as Admin.
- [ ] Admin/Index page lists all users and their roles.
- [ ] Admin/Register page successfully creates a new user (e.g., a "Student") with a selected role and **validation works**.
- [ ] Admin/ManageRoles page successfully updates the roles for a user.
- [ ] RegisterViewModel, UserListViewModel, and ManageUserRolesViewModel are created and used.

*End of Lab 1 Practice*

# LAB 2 – Integrating Student Management (CRUD)

**Topic: Student Management (Search + Filter + Pagination)**

**Requirement:** This lab is built **DIRECTLY** on top of the **Lab 1** project. We will add student management functionality to the application that already has an existing Role system.

## 0. Prerequisites

- **Completed Lab 1** (The StudentManagementMvc project already has Identity, Roles).
- Logged into the application as admin@example.com.

## 1. Update Model and DbContext

### 1.1. Create Student Model

1. In Solution Explorer, right-click the Models folder → **Add** → **Class...**.
2. Name the file Student.cs and add the following content:

using System;

```
using System.ComponentModel.DataAnnotations;

namespace StudentManagementMvc.Models
{
    public class Student
    {
        public int Id { get; set; }  // Primary Key

        [Required(ErrorMessage = "Full name is required")]
        [StringLength(50)]
        public string FullName { get; set; }

        [Required(ErrorMessage = "Email is required")]
        [EmailAddress]
        public string Email { get; set; }

        [Required(ErrorMessage = "Birth date is required")]
        [DataType(DataType.Date)]
        public DateTime BirthDate { get; set; }

        [Range(0, 4.0, ErrorMessage = "GPA must be between 0.0 and 4.0")]
        public double GPA { get; set; }

        [Required(ErrorMessage = "Class name is required")]
        [StringLength(20)]
        public string ClassName { get; set; } // Used for filtering
    }
}
```

## 1.2. Update ApplicationDbContext

1. Open Data/ApplicationDbContext.cs.
2. **Add the DbSet<Student>** to the file. This file already contains the Identity tables from Lab 1.

```
// using...
using StudentManagementMvc.Models; // Ensure you are using the Models folder

namespace StudentManagementMvc.Data
{
    // This file already exists from Lab 1
    public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
    {
```

```
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }

    // Add this line to declare the Students table
    public DbSet<Student> Students { get; set; }
  }
}
```

### 1.3. Run Migration for Student Table

1. Open **Package Manager Console** (PMC).
2. Run the Add-Migration command to add the Students table. (The InitialIdentitySchema migration already exists from Lab 1).

```
Add-Migration AddStudentTable
```

3. Wait for VS to build and create the migration file. Then, run Update-Database:

```
Update-Database
```

4. Check **SQL Server Object Explorer**, you will see the new Students table alongside the AspNet... tables from Lab 1.

## 2. Create CRUD Automatically using Scaffold

1. Right-click the Controllers folder → **Add** → **New Scaffolded Item...**.
2. Select: **MVC Controller with views, using Entity Framework** → Click **Add**.
3. In the Add Controller window:
   - **Model class**: Select Student (StudentManagementMvc.Models).
   - **Data context**: Select ApplicationDbContext (StudentManagementMvc.Data).
   - **Views**: Leave as default.
   - **Controller name**: Leave as default StudentsController.
4. Click **Add**.
   - VS will automatically create StudentsController.cs and the Views/Students folder.

## 3. Integrate Authorization

This is the step that **connects** Lab 1 and Lab 2. We will only allow Admin and Faculty to manage students.

1. Open the newly created Controllers/StudentsController.cs file.
2. Add the using for Authorization:

using Microsoft.AspNetCore.Authorization;

3. Add the [Authorize] attribute to the **top** of the StudentsController class. We specify the 2 roles allowed.

```
using Microsoft.AspNetCore.Authorization; // Add this using
using Microsoft.AspNetCore.Mvc;
// ... other usings

namespace StudentManagementMvc.Controllers
{
    [Authorize(Roles = "Admin, Faculty")] // Only Admin and Faculty can enter
    public class StudentsController : Controller
    {
        private readonly ApplicationDbContext _context;

        // ... (rest of the controller remains the same)
    }
}
```

# 4. Seed Sample Data (30 Students)

1. Open Program.cs.
2. Find the "Seed Data" code block from Lab 1. We will **add** the Student seed code inside the try block, *after* seeding Admin/Roles.

```
// --- Start Seed Data code ---
using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;
    try
    {
        // Seed Roles & Admin (from Lab 1)
        await
StudentManagementMvc.Services.DbInitializer.SeedRolesAndAdminAsync(services);

        // --- Start Seed Students (NEW) ---
        var context = services.GetRequiredService<ApplicationDbContext>();
        context.Database.EnsureCreated(); // Ensure DB exists

        // Check if Student data exists
        if (!context.Students.Any())
        {
```

```
            var classes = new[] { "SE1830", "SE1831", "GD001", "GD002" };
            var rnd = new Random();
            var list = new List<Student>();

            for (int i = 1; i <= 30; i++)
            {
                list.Add(new Student
                {
                    FullName = $"Student {i}",
                    Email = $"student{i}@example.com",
                    BirthDate = new DateTime(2004, 1, 1).AddDays(rnd.Next(1, 365 * 2)), // Born
2004-2005
                    GPA = Math.Round(2 + rnd.NextDouble() * 2, 2), // GPA from 2.0 to 4.0
                    ClassName = classes[rnd.Next(classes.Length)]
                });
            }

            context.Students.AddRange(list);
            context.SaveChanges();
        }
        // --- End Seed Students ---
    }
    catch (Exception ex)
    {
        var logger = services.GetRequiredService<ILogger<Program>>();
        logger.LogError(ex, "An error occurred seeding the DB.");
    }
}
// --- End Seed Data code ---
```

4. Run the project again. The 30 students will be added.

# 5. Validation (Check)

This part was automatically done by Scaffolding and Data Annotations.

1. **Test:** Log in with the admin@example.com account.
2. Go to /Students/Create. Click the "Create" button without entering anything → You must see error messages.

# 6. Layout + Menu + Footer

## 6.1. Edit _Layout.cshtml to add "Students" menu

1. Open Views/Shared/_Layout.cshtml.
2. Find <ul class="navbar-nav flex-grow-1">. Add the <li> for "Students" and **protect** it with roles, just like the StudentsController.

```
<div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
  <ul class="navbar-nav flex-grow-1">
    <li class="nav-item">
      <a class="nav-link text-dark" asp-area="" asp-controller="Home"
asp-action="Index">Home</a>
    </li>

    <!-- Admin Link (from Lab 1) -->
    @if (User.IsInRole("Admin"))
    {
      <li class="nav-item">
        <a class="nav-link text-danger" asp-area="" asp-controller="Admin"
asp-action="Index">Admin Management</a>
      </li>
    }

    <!-- Add Students link (Connecting Lab 1 and Lab 2) -->
    @if (User.IsInRole("Admin") || User.IsInRole("Faculty"))
    {
      <li class="nav-item">
        <a class="nav-link text-dark" asp-area="" asp-controller="Students"
asp-action="Index">Students</a>
      </li>
    }
  </ul>
  <partial name="_LoginPartial" />
</div>
```

## 6.2. Create Footer Partial View (If not done)

If you already created _Footer.cshtml in Lab 1, you can skip this.

1. Right-click Views/Shared → **Add** → **View...** → **Razor View - Empty**.
2. View name: _Footer → **Add**.
3. _Footer.cshtml content:

```
<footer class="border-top footer text-muted">
  <div class="container text-center">
    <span>© 2025 - ASP.NET Core MVC Labs</span>
  </div>
```

```
</footer>
```

4. In _Layout.cshtml, add <partial name="_Footer" /> right before the </body> tag.

# 7. Upgrade Index (Search + Filter + Sort + Pagination)

## 7.1. Modify the Index Action in StudentsController

Open StudentsController.cs and replace the Index action (the GET method) with the following code:
(Note: The entire controller must still be wrapped with [Authorize(Roles = "Admin, Faculty")])

```
// GET: Students
public async Task<IActionResult> Index(
    string sortOrder,       // sort parameter
    string searchString,    // search parameter
    string classFilter,     // class filter parameter
    int pageNumber = 1)   // paging parameter
{
    int pageSize = 5; // Number of students per page

    // Store values to use in the View
    ViewData["CurrentSort"] = sortOrder;
    ViewData["NameSortParam"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewData["CurrentFilter"] = searchString;
    ViewData["CurrentClass"] = classFilter;

    // Start the query
    var students = _context.Students.AsQueryable();

    // 1. Search by name (FullName)
    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s => s.FullName.Contains(searchString));
    }

    // 2. Filter by class (ClassName)
    if (!String.IsNullOrEmpty(classFilter))
    {
        students = students.Where(s => s.ClassName == classFilter);
    }

    // 3. Sort by name
    switch (sortOrder)
    {
```

```
        case "name_desc":
            students = students.OrderByDescending(s => s.FullName);
            break;
        default: // Default sort A-Z
            students = students.OrderBy(s => s.FullName);
            break;
    }

    // Get the list of classes for the Dropdown filter
    // Run this query separately, unaffected by other filters
    ViewBag.ClassList = await _context.Students
        .Select(s => s.ClassName)
        .Distinct()
        .OrderBy(c => c)
        .ToListAsync();

    // 4. Pagination
    var totalItems = await students.CountAsync(); // Count total students (after filtering)

    var studentsOnPage = await students
        .Skip((pageNumber - 1) * pageSize) // Skip previous pages
        .Take(pageSize) // Take students for the current page
        .ToListAsync();

    // Pass pagination info to the View
    ViewData["PageNumber"] = pageNumber;
    ViewData["TotalPages"] = (int)Math.Ceiling(totalItems / (double)pageSize);

    // Return the View with the current page's list of students
    return View(studentsOnPage);
}
```

*(Note: The other actions Create, Edit, Delete, Details remain as scaffolded).*

## 7.2. Replace the content of Views/Students/Index.cshtml

Open Views/Students/Index.cshtml and replace the *entire* content with the code below:

```
@model IEnumerable<StudentManagementMvc.Models.Student>

@{
    ViewData["Title"] = "Students";
    // Get paging and filter data from ViewData/ViewBag
```

```
    var pageNumber = (int)(ViewData["PageNumber"] ?? 1);
    var totalPages = (int)(ViewData["TotalPages"] ?? 1);
    var currentSort = (string)ViewData["CurrentSort"];
    var currentFilter = (string)ViewData["CurrentFilter"];
    var currentClass = (string)ViewData["CurrentClass"];
    var classList = ViewBag.ClassList as List<string>;
}

<h1>Students List</h1>

<!-- Form for Search and Filter -->
<form asp-action="Index" method="get" class="mb-3">
    <div class="row g-3 align-items-end">
        <div class="col-md-4">
            <label class="form-label">Search by name</label>
            <input type="text" name="searchString" value="@currentFilter" class="form-control"
/>
        </div>
        <div class="col-md-3">
            <label class="form-label">Class</label>
            <select name="classFilter" class="form-select">
                <option value="">All Classes</option>
                @if (classList != null)
                {
                    foreach (var cls in classList)
                    {
                        <!-- Keep the old filter value when posting the form -->
                        <option value="@cls" @(currentClass == cls ? "selected" : "")>@cls</option>
                    }
                }
            </select>
        </div>
        <div class="col-md-2">
            <input type="submit" value="Filter / Search" class="btn btn-secondary w-100" />
        </div>
        <div class="col-md-2">
            <a asp-action="Index" class="btn btn-outline-dark w-100">Clear</a>
        </div>
    </div>
</form>

<p>
    <a asp-action="Create" class="btn btn-primary">Create New Student</a>
```

```
</p>

<table class="table table-striped table-hover">
  <thead class="table-dark">
    <tr>
      <th>
        <!-- Sort Link: keep filter state when sorting -->
        <a asp-action="Index"
          asp-route-sortOrder="@ViewData["NameSortParam"]"
          asp-route-searchString="@currentFilter"
          asp-route-classFilter="@currentClass"
          class="text-white text-decoration-none">
           Full Name
           @(currentSort == "name_desc" ? "↓" : (currentSort == "" ? "↑" : ""))
        </a>
      </th>
      <th>Email</th>
      <th>Birth Date</th>
      <th>GPA</th>
      <th>Class</th>
      <th>Actions</th>
    </tr>
  </thead>
  <tbody>
    @foreach (var item in Model)
    {
      <tr>
        <td>@item.FullName</td>
        <td>@item.Email</td>
        <td>@item.BirthDate.ToShortDateString()</td>
        <td>@item.GPA.ToString("0.00")</td>
        <td>@item.ClassName</td>
        <td>
          <a asp-action="Edit" asp-route-id="@item.Id" class="btn btn-sm
btn-outline-primary">Edit</a>
          <a asp-action="Details" asp-route-id="@item.Id" class="btn btn-sm
btn-outline-info">Details</a>
          <a asp-action="Delete" asp-route-id="@item.Id" class="btn btn-sm
btn-outline-danger">Delete</a>
        </td>
      </tr>
    }
  </tbody>
```

```
</table>

<!-- Pagination -->
<nav aria-label="Page navigation">
    <ul class="pagination justify-content-center">

        @{
            // Previous Button
            var prevDisabled = pageNumber <= 1 ? "disabled" : "";
        }
        <li class="page-item @prevDisabled">
            <a class="page-link"
               asp-action="Index"
               asp-route-pageNumber="@(pageNumber - 1)"
               asp-route-searchString="@currentFilter"
               asp-route-classFilter="@currentClass"
               asp-route-sortOrder="@currentSort">
                Previous
            </a>
        </li>

        <!-- Display Page X / Y -->
        <li class="page-item disabled">
            <span class="page-link">
                Page @pageNumber / @totalPages
            </span>
        </li>

        @{
            // Next Button
            var nextDisabled = pageNumber >= totalPages ? "disabled" : "";
        }
        <li class="page-item @nextDisabled">
            <a class="page-link"
               asp-action="Index"
               asp-route-pageNumber="@(pageNumber + 1)"
               asp-route-searchString="@currentFilter"
               asp-route-classFilter="@currentClass"
               asp-route-sortOrder="@currentSort">
                Next
            </a>
        </li>
    </ul>
```

</nav>

# 8. Completion Checklist (Lab 2)

- [ ] Built Lab 2 on top of the Lab 1 platform.
- [ ] Added Student Model and DbSet<Student> to ApplicationDbContext.
- [ ] Successfully ran Add-Migration AddStudentTable + Update-Database.
- [ ] Successfully scaffolded StudentsController + Views.
- [ ] **Connected** Lab 1 & 2: StudentsController is protected by [Authorize(Roles = "Admin, Faculty")].
- [ ] "Students" menu is only visible to Admin and Faculty.
- [ ] Automatically seeded 30 students (alongside seeding Roles/Admin).
- [ ] Students/Index page has all 4 functions working simultaneously:
  - [ ] Search by name.
  - [ ] Filter by class (dropdown).
  - [ ] Sort by name A–Z / Z–A.
  - [ ] Pagination (5 students / page).
- [ ] **(Test)** Logged in as a "Student" (created by Admin) and CANNOT see the "Students" menu or access /Students (is forbidden).

*End of Lab 2*