.Net Core Assignment

Install Prerequisites

Ensure you have the following installed on your system.

- Install .NET 8 SDK: Verify with dotnet --version in your terminal.
- Install VS Code
- Install VS Code Extensions (Ctrl + Shift + X):
    - C# Dev Kit (Solution Explorer view).
    - C# (Base language support).
    - GitHub Copilot Chat (Login for IntelliSense)
    - *Tip:* Disable "Auto Save" for the demo (File > Auto Save) so you can control when changes apply.
- Install SQL Server Express

Project Initialization

Create Project Solution Folder

1) Create New Folder for Project Solution and Open Terminal
2) Go to Project Solution Folder.

```
C:\> cd “C:\Project\Demo”
```

3) Run Scaffold

```
C:\Project\Demo> dotnet new mvc -n WebAppName -f net8.0
```

If error, then check dotnet --version

4) Open Project

```
C:\Project\Demo> cd WebAppName
C:\Project\Demo\WebAppName> code .
```

Note: When VS Code asks "Required assets to build and debug are missing...", click **Yes**.

5) Run Project

```
C:\Project\Demo\WebAppName> dotnet watch
```

Note: after this you should able see hosting link like http://localhots:0000/

6) Add Packages

```
C:\Project\Demo\WebAppName> dotnet add package System.Data.SqlClient
C:\Project\Demo\WebAppName> dotnet new nugetconfig
```

Note: run only for first time

7) Add Solution and Project

```
C:\Project\Demo\WebAppName> cd ..
C:\Project\Demo> dotnet new sln -n DemoSolution
C:\Project\Demo> dotnet sln add WebAppName/WebAppName.csproj
C:\Project\Demo> code .
```

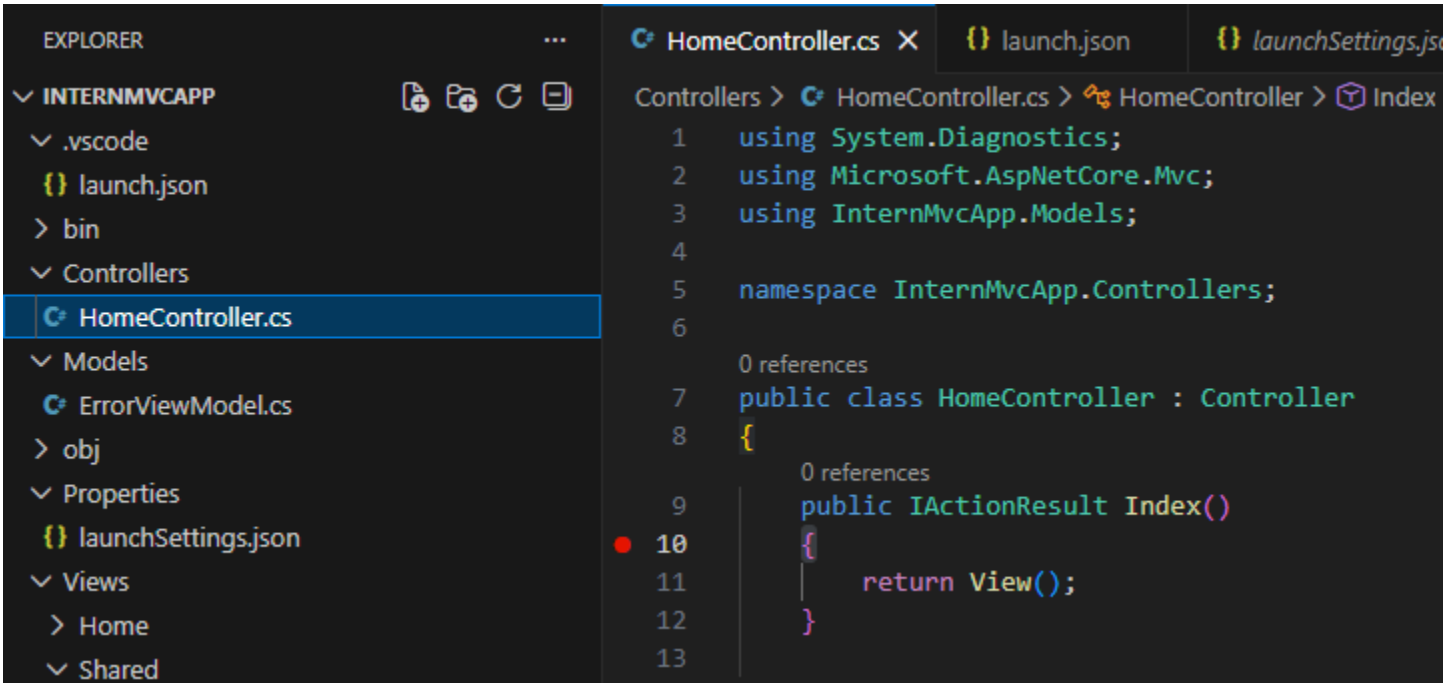
(Optional)Take your time to understand all commands

```
C:\Project\Demo\WebAppName> dotnet --help
```

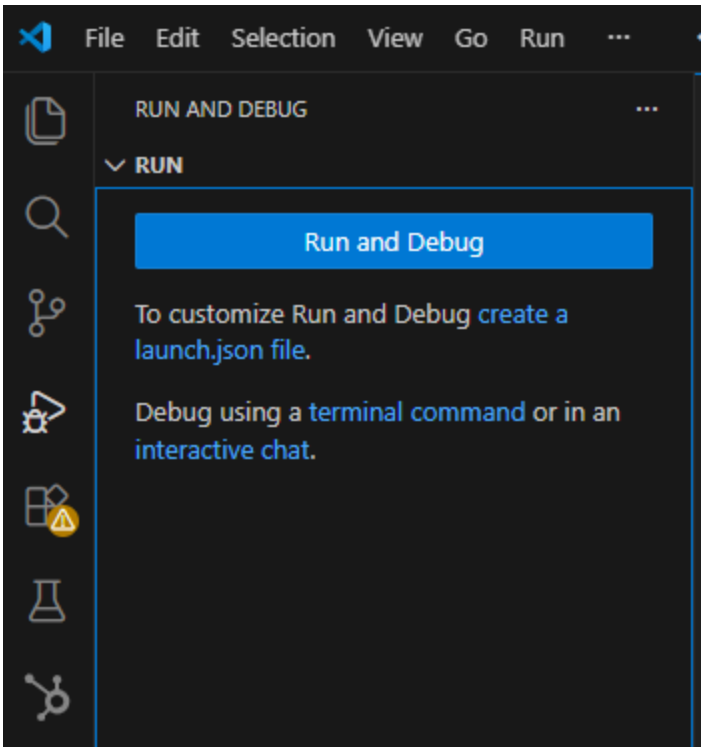
Project Debugging

Debugging isn't automatic like in Visual Studio; you need to tell VS Code *how* to launch the project in launch.json file

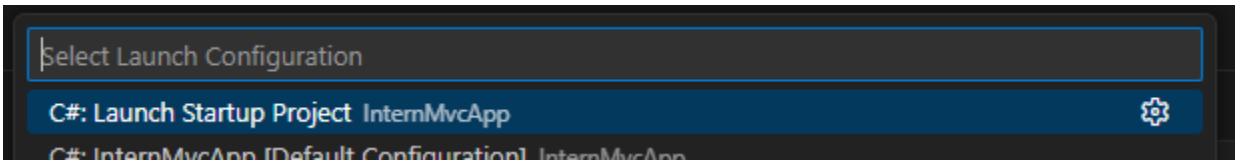
1) Set a Breakpoint:
    - Go to **`Controllers/HomeController.cs`**. Click in the left margin next to the code line A red dot will appear.



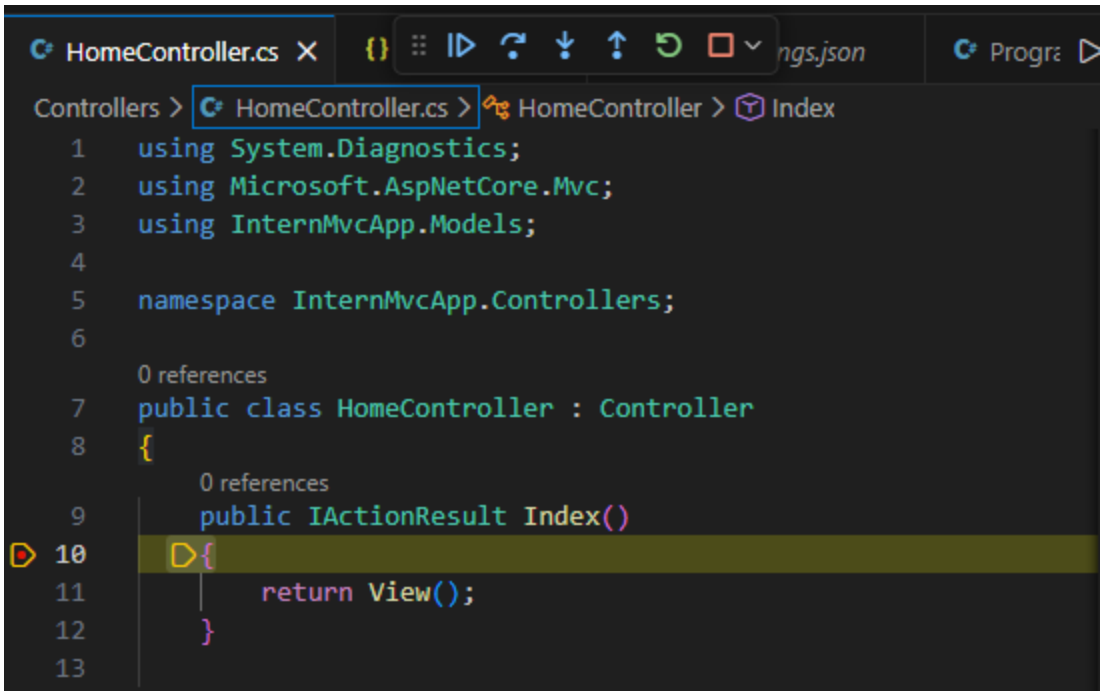
2) Start Debugging
    - Click the "Run and Debug" icon on the left sidebar > Play button



- Then click on C# and select startup project



- After project run you should able to see breakpoint debugger with Debug controller



3) Debugging JavaScript
    - For Debugging JavaScript, I recommend to use browser developer tools (Ctrl+Shift+I)

Add New Library and Use

- Go up one level where .sln file locate
- Create the Class Library Project

```
C:\Project\Demo\WebAppName> cd..
C:\Project\Demo> dotnet new classlib -n WebAppName.Repository -f net8.0
```

- add this new project to your solution

```
dotnet sln add WebAppName.Repository/WebAppName.Repository.csproj
```

- Connect the Library to Web App

```
cd WebAppName
dotnet add reference ../WebAppName.Repository/WebAppName.Repository.csproj
```

Configure Entity Framework

EF with Code First

Make sure you have Entity Framework NuGet configured

- Add EF Packages

```
C:\Project\Demo\WebAppName> dotnet add package Microsoft.EntityFrameworkCore --version 8.0.11
C:\Project\Demo\WebAppName> dotnet add package Microsoft.EntityFrameworkCore.SqlServer  --version
8.0.11
C:\Project\Demo\WebAppName> dotnet add package Microsoft.EntityFrameworkCore.Tools  --version 8.0.11
C:\Project\Demo\WebAppName> dotnet add package Microsoft.EntityFrameworkCore.Design --version 8.0.11
```

- Define Your Model

```
namespace WebAppName.Repository.Models;
public class Intern
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Department { get; set; }
    public DateTime JoinDate { get; set; }
}
```

- Create the DbContext

```
using Microsoft.EntityFrameworkCore;
using WebAppName.Repository.Models;
namespace WebAppName.Repository.Data;
public class ApplicationDbContext : DbContext


{
    // Constructor helps pass configuration options (like connection string)
to the base class
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext>
options)
        : base(options)
    {    }

    // This property represents the "Interns" table in your database
    public DbSet<Intern> Interns { get; set; }
}
```

- Configure the Connection String

```
{
    "ConnectionStrings": {

        "DefaultConnection": "Server=localhost\\;Integrated
Security=True;TrustServerCertificate=True;"

    }
}
```

    - Ensure that dbinstancename matches the instance name of your local SQL Server.
- Register DbContext in Program.cs

```
using Microsoft.EntityFrameworkCore;
using WebAppName.Repository.Data;

// 1. Fetch the connection string from appsettings.json
var connectionString = builder.Configuration.GetConnectionString("DefaultConnection");

// 2. Register the DbContext in the Dependency Injection container
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString, b => b.MigrationsAssembly("WebAppName")));
```

- Create the Database (Migrations)
    - Install dotnet-ef (only first time)

```
C:\Project\Demo\WebAppName> dotnet tool install --global dotnet-ef
```

    - Add migrations

```
C:\Project\Demo\WebAppName> dotnet ef migrations add InitialCreate
```

    - Tell EF to update database

```
C:\Project\Demo\WebAppName> dotnet ef database update
```

    - Now, You should able to see new DB & table in SSMS
    - After this If you modify table like add new property then

```
C:\Project\Demo\WebAppName> dotnet ef migrations add AddPropToInterns
C:\Project\Demo\WebAppName> dotnet ef database update
```

- Update the Controller to Use DB
    - Inject on Controller constructor

```
public InternsController(ApplicationDbContext context)

{ _context = context; }
```

now you have _context for communicate with DB for CRUD operation.

- Follow Code first approach in Entity Framework for steps

ADO.NET

Please follow Introduction To ADO.NET

Dependency Injection

Coming soon after testing

- Create the Interface

```
namespace InternManagement.Repository;

public interface IInternRepository
{
    Task<List<Intern>> GetAllInternsAsync();
    Task AddInternAsync(Intern intern);
}
```

- Create the Implementation

```
using Microsoft.EntityFrameworkCore;
using InternManagement.Repository.Data;
using InternManagement.Repository.Models;

namespace InternManagement.Repository;

public class InternRepository : IInternRepository
{
    private readonly ApplicationDbContext _context;

    public InternRepository(ApplicationDbContext context)
    {
        _context = context;
    }

    public async Task<List<Intern>> GetAllInternsAsync()
    {
        return await _context.Interns.ToListAsync();
    }
```

```
    public async Task AddInternAsync(Intern intern)
    {
        _context.Interns.Add(intern);
        await _context.SaveChangesAsync();
    }
}
```

- Register the Service in Program.cs

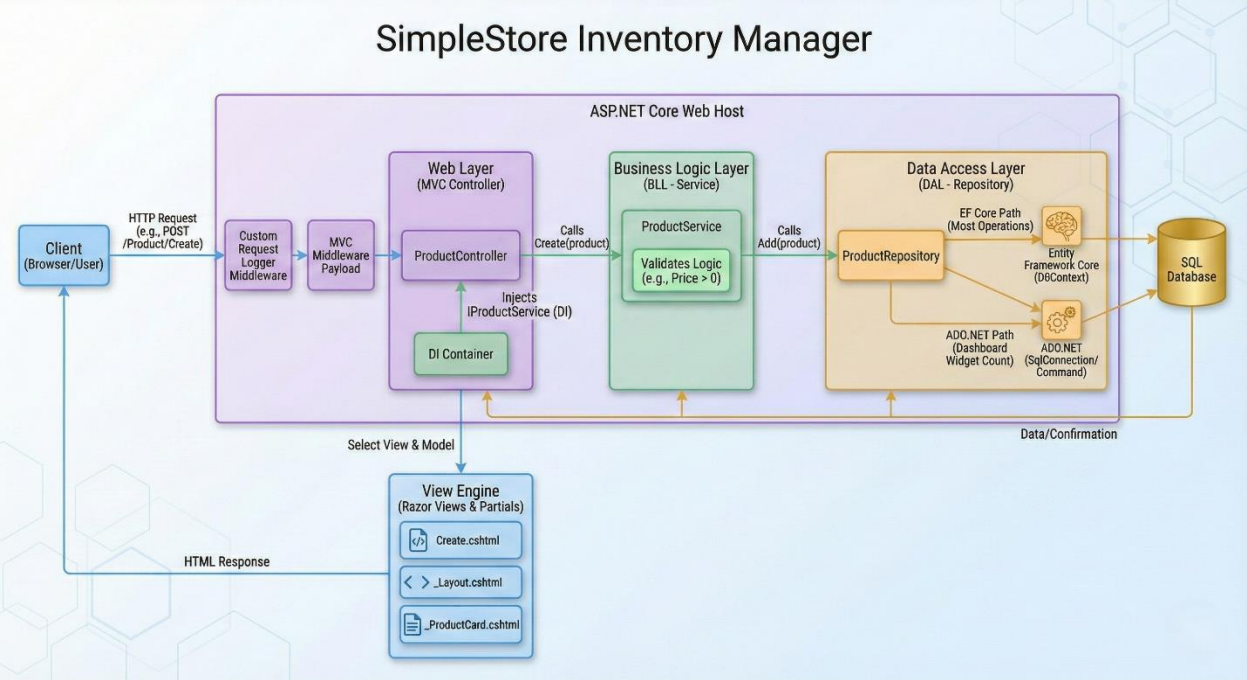
```
// Register the Interface and its Implementation
builder.Services.AddScoped<IInternRepository,
InternRepository>();
```

- Follow Dependency Injection and Services in ASP.NET Core for steps

Server-Side Validation

Please follow ASP.NET MVC: Form Validation

Demo Project for Inventory Manager



A "Product Inventory Management System" (let's call it "SimpleStore") is the classic beginner project that perfectly fits every single concept we discussed. It is complex enough to force you to use architecture, but simple enough to finish in a day.

Here is the blueprint for the project, mapping each feature to the concepts you learned.

Project Name: SimpleStore Inventory Manager

Goal: Build a web app where an user can log in, view a list of products, add new ones, edit existing and delete with link checked.

---

1. The Architecture (Layering)

You will not write all code in the Controller. You will strictly enforce N-Tier Architecture.

- DAL (Repository Layer):
    - Create Repository.cs.

    - (optional) First, create a static class for storage and complete the features. Afterward, try to implement Entity Framework with SQL Server for data storage and use that in repositories.
    - Concept Used: Entity Framework (Code First).
    - *Task:* Create a DbContext and a Product class. Use Migrations to create the database.
- BLL (Service Layer):
    - Create Service.cs.
    - Concept Used: BLL Logic.
    - *Task:* Add a rule: "A product name cannot contain "Price must be positive." The Controller calls this Service, not the Repo.
- Web (Controller):
    - Create Controller.cs.
    - Concept Used: Dependency Injection.
    - *Task:* Inject Service into the Controller constructor.

2. The Features & Concepts

Feature A: The "Request Logger" (Middleware)

- What to build: Create a simple class that writes "Request received at [Time]" to the console for every user click.
- Concept Used: Middleware Pipeline. You will inject this before the MVC middleware in Program.cs.

Feature B: The Product List (Read)

- What to build: A page showing all products in a table.
- Concepts Used:
    - MVC Pattern: Controller fetches list -> passes to View.
    - Razor Syntax: Use @foreach to loop through the list and render <tr> rows.
    - Layout View: The page must use _Layout.cshtml so the Navigation Bar (Home, Products) is visible.

Feature C: The "Add Product" Form (Create)

- What to build: A form with Name, Price, Quantity and Category (Dropdown).
- Concepts Used:
    - Razor HTML Helpers: Use @Html.TextBoxFor(m => m.Name) or Tag Helpers <input asp-for="Name">.
    - Validation: Add [Required] to your entity class and display errors using @Html.ValidationMessageFor.

Feature D: The Dashboard Widget (Hybrid Data Access)

- What to build: On the sidebar, show a small box: "Total Products: 50".
- Concept Used: ADO.NET.
    - *Challenge:* Even though the rest of the app uses Entity Framework, write *one* method in your Repository that uses raw SqlConnection and SqlCommand to run SELECT COUNT(*) FROM Products. This proves you understand how to mix both technologies (EF for convenience, ADO for raw speed/reporting).

Feature E: The "Product Card" (Reusability)

- What to build: Instead of a table, switch the view to a "Grid" view.
- Concept Used: Partial Views.
    - Create _ProductCard.cshtml. Use this partial inside your main loop. This keeps your main code clean.

Summary Checklist for You and Evacuator

1. Setup: Create ASP.NET Core MVC project.
2. Middleware: Write a custom logging middleware.
3. Exception Filter: handle common error messages
4. Database: Setup EF Core DbContext and run Migrations.
5. Repo: Create Repository (EF Core).
6. Repo (optional): Add GetTotalCount() using ADO.NET.
7. Service: Create Service with validation logic.

8. DI: Register Service and Repo in Program.cs (builder.Services.AddScoped...).
9. UI: Build the Controller and Razor Views (Index, Create, Edit, Delete (Prompt confirmation before executing)).
10. Style (optional): Use Bootstrap for Responsive UI Design