SE Lab 7 Contents (ASP.Net Core – DB First Approach)

ASP.NET Core. What’s new?

ASP.NET Web Applications using the .NET Framework are still in use, though their popularity has shifted somewhat with the advent of newer technologies like ASP.NET Core. ASP.NET Web Forms, a part of the .NET Framework, is considered more legacy technology but is still supported and used in many existing applications

Many developers and organizations continue to use ASP.NET for its robustness, security features, and integration with the .NET Framework. However, for new projects, ASP.NET Core is often preferred due to its cross-platform capabilities, improved performance, and modern development practices

If you're working on maintaining or upgrading an existing ASP.NET Web Application, it's still a viable and supported option. For new projects, exploring ASP.NET Core might be beneficial

ASP.NET Core offers several advantages over its predecessor, ASP.NET, making it a popular choice for modern web development. Here are some key benefits:

1. Cross-Platform:
    - ASP.NET Core can run on Windows, macOS, and Linux, allowing developers to build and deploy applications on different operating systems
2. High Performance:
    - ASP.NET Core is designed for high performance and scalability. It is faster and more efficient than ASP.NET, thanks to its modular and lightweight architecture.
3. Open Source:
    - ASP.NET Core is open source, which means it benefits from community contributions and transparency. Developers can access the source code, contribute to its development, and use it without licensing costs
4. Unified Framework:
    - ASP.NET Core provides a unified framework for building web APIs and web applications, simplifying the development process and reducing the need for multiple frameworks
5. Modern Development Practices:
    - It supports modern development practices such as dependency injection, asynchronous programming, and built-in support for popular client-side frameworks like Angular, React, and Vue.js
6. Cloud-Ready:
    - ASP.NET Core is designed with cloud deployment in mind. It includes features like environment-based configuration and built-in support for Docker, making it easier to deploy applications to the cloud

7. Improved Security:
    - ASP.NET Core includes enhanced security features, such as built-in support for HTTPS, data protection, and authentication/authorization mechanisms
8. Side-by-Side Versioning:
    - It allows multiple versions of ASP.NET Core to run side-by-side on the same machine, enabling developers to upgrade applications independently
9. Lightweight and Modular:
    - The framework is modular, allowing developers to include only the necessary components, which reduces the application's footprint and improves performance
10. Tooling and Productivity:
    - ASP.NET Core is supported by a rich set of development tools, including Visual Studio and Visual Studio Code, which enhance developer productivity with features like IntelliSense, debugging, and integrated testing

These advantages make ASP.NET Core a powerful and flexible framework for building modern, high-performance web applications.

ASP.NET Core supports the Database First approach

ASP.NET Core supports the Database First approach through Entity Framework Core (EF Core).

The Database First approach allows you to start with an existing database and generate the data access layer of your application based on the database schema. This is particularly useful when working with a pre-existing database or when the database design is managed by a separate team.

1. Steps of Database First approach in ASP.NET Core

1.1. Install EF Core Tools:

- Ensure you have the necessary EF Core tools installed. You can install them via NuGet Package Manager in Visual Studio or using the .NET CLI:
- dotnet add package Microsoft.EntityFrameworkCore.Design
- dotnet add package Microsoft.EntityFrameworkCore.SqlServer

1.2. Scaffold the Database Context and Entities:

- Use the Scaffold-DbContext command to generate the DbContext and entity classes from your existing database. For example:
- Scaffold-DbContext "YourConnectionString" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models

- Replace "YourConnectionString" with your actual database connection string and Models with the desired output directory for the generated classes.

1.3. Configure the DbContext:

- In your ASP.NET Core project, configure the generated DbContext in the Startup.cs file:

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddDbContext<YourDbContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("YourConnectionString")));
  // Other service configurations
}
```

1.4. Use the Generated Models:

- You can now use the generated DbContext and entity classes to interact with your database. For example, you can perform CRUD operations using LINQ queries:

```
using (var context = new YourDbContext())
{
  var data = context.YourEntity.ToList();
}
```

In ASP.NET Core 6.0 and later, the Startup.cs file has been integrated into the Program.cs file, simplifying the setup process. This new approach uses a top-level statement to configure the application, making the code more concise and easier to manage.

In this tutorial, DB First Approach with 2 ASP.NET framework : ASP.NET Core Web Application using Razor Pages and ASP.NET Core Web Application with MVC

2. ASP.NET Core Web Application using Razor Pages

Razor Pages is a page-based programming model that makes building web UI easier and more productive. It is designed to simplify the development of web applications by focusing on individual pages.

Key Features:

1. Page-Based Model:

   - Each Razor Page consists of a .cshtml file (for the view) and a .cshtml.cs file (for the page model or code-behind).

- This structure keeps the code for each page together, making it easier to manage and understand.

2. Simplified Routing:
    - Razor Pages use a convention-based routing system, where the URL path maps directly to the file path in the Pages folder.
    - This reduces the need for explicit routing configurations.
3. Built-In Support for Common Patterns:
    - Razor Pages support model binding, validation, and tag helpers, making it easy to implement common web application patterns.
4. Ideal for Simple Applications:
    - Razor Pages are great for applications where the focus is on individual pages rather than complex interactions between multiple controllers and views.

Example:

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddRazorPages();
var app = builder.Build();

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();
app.UseAuthorization();
app.MapRazorPages();

app.Run();
```

3. ASP.NET Core Web Application with MVC

MVC (Model-View-Controller) is a design pattern that separates an application into three main components: the model, the view, and the controller. This separation helps manage the complexity of large applications by dividing the responsibilities.

Key Features:

1. Separation of Concerns:

- The MVC pattern divides the application into three interconnected components, making it easier to manage and test.
- Model: Represents the application's data and business logic.
- View: Displays the data and the user interface.
- Controller: Handles user input and interactions, updating the model and view accordingly.

2. Flexible Routing:
    - MVC provides a flexible routing system that allows for complex URL patterns and custom route configurations.
3. Rich Ecosystem:
    - MVC has a rich set of features, including filters, areas, and scaffolding, which help in building robust and maintainable applications.
4. Ideal for Complex Applications:
    - MVC is well-suited for applications with complex interactions and multiple views, where the separation of concerns is crucial.

Example:

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllersWithViews();
var app = builder.Build();

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();
app.UseAuthorization();
app.MapControllerRoute(
  name: "default",
  pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();
```

Choosing Between Razor Pages and MVC

- Razor Pages: Best for page-focused applications where each page is a self-contained unit. It simplifies the development process and is easier to learn for beginners.
- MVC: Best for applications that require a clear separation of concerns and have complex interactions between different parts of the application. It provides more flexibility and control over the application's architecture.

Both approaches are powerful and have their own advantages. The choice depends on the specific requirements and complexity of your project.

Exercise 1: Develop a School Management Web App with ASP.NET Core Web Application using Razor Pages

This exercise aims to create an ASP.NET Core Web Application using Razor Pages with a comprehensive database schema that includes Students, Courses, Instructors, Enrollments, and Departments. Here are the steps to guide you through the process:

Step1: Setting Up the Database

Objective: Create a database with tables for Students, Courses, Instructors, Enrollments, and Departments.

Steps:

1. Create a Database:
    - Open SQL Server Management Studio (SSMS).
    - Create a new database named SchoolDB.
2. Create Tables:
    - Execute the following SQL scripts to create the necessary tables:

```
CREATE TABLE Departments (
  DepartmentID INT PRIMARY KEY IDENTITY(1,1),
  Name NVARCHAR(50) NOT NULL,
  Budget DECIMAL(18, 2) NOT NULL,
  StartDate DATE NOT NULL
);

CREATE TABLE Instructors (
  InstructorID INT PRIMARY KEY IDENTITY(1,1),
  FirstName NVARCHAR(50) NOT NULL,
  LastName NVARCHAR(50) NOT NULL,
  HireDate DATE NOT NULL
);

CREATE TABLE Courses (
  CourseID INT PRIMARY KEY IDENTITY(1,1),
  Title NVARCHAR(100) NOT NULL,
  Credits INT NOT NULL,
  DepartmentID INT FOREIGN KEY REFERENCES Departments(DepartmentID)
);

CREATE TABLE Students (
  StudentID INT PRIMARY KEY IDENTITY(1,1),
  FirstName NVARCHAR(50) NOT NULL,
  LastName NVARCHAR(50) NOT NULL,
  EnrollmentDate DATE NOT NULL
);

CREATE TABLE Enrollments (
  EnrollmentID INT PRIMARY KEY IDENTITY(1,1),
  CourseID INT FOREIGN KEY REFERENCES Courses(CourseID),
  StudentID INT FOREIGN KEY REFERENCES Students(StudentID),
  Grade DECIMAL(3, 2)
);
```

3. Insert Sample Data:
    - Insert sample data into the tables:

```
INSERT INTO Departments (Name, Budget, StartDate) VALUES ('Computer Science',
120000.00, '2023-01-01');
```

INSERT INTO Instructors (FirstName, LastName, HireDate) VALUES ('John', 'Doe', '2020-08-15');

INSERT INTO Courses (Title, Credits, DepartmentID) VALUES ('Introduction to Programming', 3, 1);

INSERT INTO Students (FirstName, LastName, EnrollmentDate) VALUES ('Jane', 'Smith', '2023-09-01');

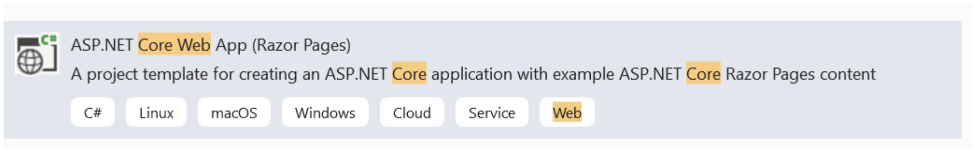
INSERT INTO Enrollments (CourseID, StudentID, Grade) VALUES (1, 1, 3.5);

Step 2: Creating the ASP.NET Core Web Application

Objective: Create a new ASP.NET Core Web Application with Razor Pages and set up the necessary packages.

Step 1: Setting up a new Project

1. Create a New Project:
   - Open Visual Studio and create a new ASP.NET Core Web Application project.



ASP.NET Core Web App (Razor Pages)
A project template for creating an ASP.NET Core application with example ASP.NET Core Razor Pages content

   - Name your project SchoolAppCore and choose a location to save it.
   - Select the Web Application template and ensure Razor Pages is selected. Click Create.
2. Install Entity Framework Core:
   - Right-click on your project in the Solution Explorer and select Manage NuGet Packages.
   - Search for Microsoft.EntityFrameworkCore.SqlServer and install it.
   - Search for Microsoft.EntityFrameworkCore.Tools and install it.

Step 3: Modify the Menu in _Layout.cshtml

1. Open the _Layout.cshtml File:
   - In Visual Studio, navigate to the Views/Shared folder for an MVC application or the Pages/Shared folder for a Razor Pages application.
   - Open the _Layout.cshtml file.

2. Locate the Navigation Section:
   - Find the section in the _Layout.cshtml file where the navigation menu is defined. This is typically within a <nav> element.

3. Add Navigation Links:

    - Add links for Students, Courses, Instructors, Enrollments, and Departments to the navigation menu. Use the asp-area and asp-controller or asp-page attributes to

4. Example Code for Razor Pages Application:

```
<nav class="navbar navbar-expand-sm navbar-light bg-light">
    <a class="navbar-brand" asp-page="/Index">SchoolApp</a>
    <button class="navbar-toggler" type="button" data-toggle="collapse" data-
target="#navbarNav" aria-controls="navbarNav" aria-expanded="false" aria-
label="Toggle navigation">
        <span class="navbar-toggler-icon"></span>
    </button>
    <div class="collapse navbar-collapse" id="navbarNav">
        <ul class="navbar-nav">
            <li class="nav-item">
                <a class="nav-link" asp-page="/Index">Home</a>
            </li>
            <li class="nav-item">
                <a class="nav-link" asp-page="/Students/Index">Students</a>
            </li>
            <li class="nav-item">
                <a class="nav-link" asp-page="/Courses/Index">Courses</a>
            </li>
            <li class="nav-item">
                <a class="nav-link" asp-page="/Instructors/Index">Instructors</a>
            </li>
            <li class="nav-item">
                <a class="nav-link" asp-page="/Enrollments/Index">Enrollments</a>
            </li>
            <li class="nav-item">
                <a class="nav-link" asp-page="/Departments/Index">Departments</a>
            </li>
        </ul>
    </div>
</nav>
```

Step 4: Setting Up the Database Connection

Objective: Configure the connection to the SchoolDB database using Entity Framework Core.

Steps:

1. Define the Connection String in appsettings.json:

    - Open the appsettings.json file in your project.
    - Add the connection string within the ConnectionStrings section:

```
{
  "ConnectionStrings": {
    "SchoolDBConnectionString":
"Server=(local)\\SQLEXPRESS;Database=SchoolDB;Trusted_Connection=True;T
rustServerCertificate=True"},
    "Logging": {
      "LogLevel": {
        "Default": "Information",
        "Microsoft.AspNetCore": "Warning"
      }
    },
    "AllowedHosts": "*" }
```

2. Create the Entity Framework Core Model:
    - Right-click on your project in the Solution Explorer, select Add > New Folder, and name it Models.
    - Right-click on the Models folder, select Add > Class, and name it SchoolContext.cs.
    - Add the following code to the **SchoolContext.cs** file:

```
using Microsoft.EntityFrameworkCore;

namespace SchoolAppCore.Models

{

  public class SchoolContext : DbContext

  {

    public SchoolContext(DbContextOptions<SchoolContext> options) :
base(options) { }

    public DbSet<Department> Departments { get; set; }

    public DbSet<Instructor> Instructors { get; set; }

    public DbSet<Course> Courses { get; set; }

    public DbSet<Student> Students { get; set; }

    public DbSet<Enrollment> Enrollments { get; set; }

  }

}
```

3. Create the Entity Classes:
    - Create classes for Department, Instructor, Course, Student, and Enrollment in the Models folder. Here is an example for the Department class:

```
namespace SchoolAppCore.Models

{
    public class Department
    {
        public int DepartmentID { get; set; }
        public string Name { get; set; }
        public decimal Budget { get; set; }
        public DateTime StartDate { get; set; }
    }
}
```

- Repeat similar steps for the other entity classes above **(SchoolContext.cs)**

4. Configure **the DbContext in Startup.cs / Program.cs:**

   - Open the Startup.cs / **Program.cs (for .NET 6+)** file

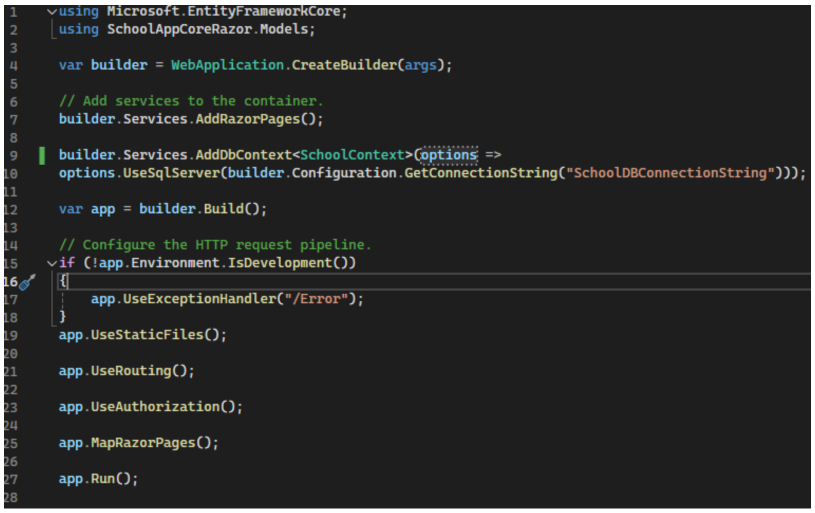
     If Startup.cs then add the following code in the ConfigureServices method:

```
public void ConfigureServices(IServiceCollection services){
  services.AddDbContext<SchoolContext>(options =>
options.UseSqlServer(Configuration.GetConnectionString("SchoolDBConnectionString")));
  services.AddRazorPages();
}
```

If no Startup.cs then open **Program.cs (for .NET 6+)** **.** Please add:

```
using Microsoft.EntityFrameworkCore;
using SchoolAppCoreRazor.Models;
...
Builder.Services.AddRazorPages();
builder.Services.AddDbContext<SchoolContext>(options =>
options.UseSqlServer(builder.Configuration.GetConnectionString("SchoolDBConnection
String")));
........
```

Notes: SchoolDBConnectionString ~ check in **appsettings.json**



```
using Microsoft.EntityFrameworkCore;
using SchoolAppCoreRazor.Models;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();

builder.Services.AddDbContext<SchoolContext>(options =>
options.UseSqlServer(builder.Configuration.GetConnectionString("SchoolDBConnectionString")));

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
}
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

File Program.cs

Step 5: Creating Razor Pages

Objective: Create Razor Pages to manage Students, Courses, Instructors, Enrollments, and Departments.

1. Create Razor Pages:

   - Right-click on the **Pages** folder, select **Add** > **New Folder**, and name it **Departments**.
   - Right-click on the Departments folder, **select Add > Razor Page, and choose Razor Pages using Entity Framework (CRUD).** (
   - Select **Department as the Model class** and **SchoolContext as the data context class**. (*Unchecked all Optiops, or just checked Reference cript libraries*) **Click Add.**

2. **Repeat** 1. Create Razor Pages the process for Students, Courses, Instructors, and Enrollments.

Step 6: Testing the Application

Objective: Run the application and test the functionality.

1. Run the Application:

   - Press F5 or click the Start button to run the application.
   - Navigate to the respective URLs (e.g., /Departments, /Students) to access the list of entities.

2. Verify the Output:

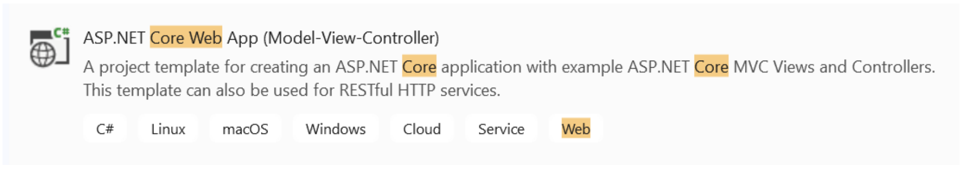
   - Check the lists displayed on the pages to verify that the data is being retrieved from the database correctly.

Exercise 2: Develop a School Management Web App with ASP.NET Core Web App (MVC)

This exercise aims to create an ASP.NET Core Web App(Model -View-Controller) with a comprehensive database schema that includes Students, Courses, Instructors, Enrollments, and Departments. Here are the steps to guide you through the process:

Step 1: Setting up an environment

1. Create a New Project:
    - Open Visual Studio and create a new ASP.NET Core Web Application project.



ASP.NET Core Web App (Model-View-Controller)

A project template for creating an ASP.NET Core application with example ASP.NET Core MVC Views and Controllers. This template can also be used for RESTful HTTP services.

    - Name your project SchoolAppCoreMVC and choose a location to save it.
    - Select the Web Application (Model-View-Controller) template and click Create.
2. Install Entity Framework Core:
    - Right-click on your project in the Solution Explorer and select Manage NuGet Packages.
    - Search for Microsoft.EntityFrameworkCore.SqlServer and install it.
    - Search for Microsoft.EntityFrameworkCore.Tools and install it.

Step 2: Modify the Menu in _Layout.cshtml

2. Open the _Layout.cshtml File:
    - In Visual Studio, navigate to the Views/Shared folder for an MVC application or the Pages/Shared folder for a Razor Pages application.
    - Open the _Layout.cshtml file.
3. Locate the Navigation Section:
    - Find the section in the _Layout.cshtml file where the navigation menu is defined. This is typically within a <nav> element.
4. Add Navigation Links:
    - Add links for Students, Courses, Instructors, Enrollments, and Departments to the navigation menu. Use the asp-area and asp-controller or asp-page attributes to generate the correct URLs.
5. Example Code for MVC Application

```
<nav class="navbar navbar-expand-sm navbar-light bg-light">
    <a class="navbar-brand" asp-area="" asp-controller="Home" asp-
action="Index">SchoolApp</a>
    <button class="navbar-toggler" type="button" data-toggle="collapse" data-
target="#navbarNav" aria-controls="navbarNav" aria-expanded="false" aria-
label="Toggle navigation">
        <span class="navbar-toggler-icon"></span>
    </button>
    <div class="collapse navbar-collapse" id="navbarNav">
        <ul class="navbar-nav">
            <li class="nav-item">
                <a class="nav-link" asp-area="" asp-controller="Home" asp-
action="Index">Home</a>
            </li>
            <li class="nav-item">
                <a class="nav-link" asp-area="" asp-controller="Students"
asp-action="Index">Students</a>
            </li>
            <li class="nav-item">
                <a class="nav-link" asp-area="" asp-controller="Courses" asp-
action="Index">Courses</a>
            </li>
            <li class="nav-item">
                <a class="nav-link" asp-area="" asp-controller="Instructors"
asp-action="Index">Instructors</a>
            </li>
            <li class="nav-item">
                <a class="nav-link" asp-area="" asp-controller="Enrollments"
asp-action="Index">Enrollments</a>
            </li>
            <li class="nav-item">
                <a class="nav-link" asp-area="" asp-controller="Departments"
asp-action="Index">Departments</a>
            </li>
        </ul>
    </div>
</nav>
```

Step 3: Setting Up the Database Connection

Objective: Configure the connection to the SchoolDB database using Entity Framework Core.

Steps:

1. Define the Connection String in appsettings.json:
    - Open the appsettings.json file in your project.
    - Add the connection string within the ConnectionStrings section:

```
{
  "ConnectionStrings": {
    "SchoolDBConnectionString":
"Server=(local)\\SQLEXPRESS;Database=SchoolDB;Trusted_Connection=True;TrustServerCertificate=True"
  },
    "Logging": {
      "LogLevel": {
        "Default": "Information",
        "Microsoft.AspNetCore": "Warning"
      }
    },
    "AllowedHosts": "*" }
```

2. Create the Entity Framework Core Model:

   - Right-click on your project in the Solution Explorer, select Add > New Folder, and name it Models.
   - Right-click on the Models folder, select Add > Class, and name it SchoolContext.cs.
   - Add the following code to the SchoolContext.cs file:

```
using Microsoft.EntityFrameworkCore;


namespace SchoolAppCoreMVC.Models
{
  public class SchoolContext : DbContext
  {
    public SchoolContext(DbContextOptions<SchoolContext> options) :
base(options) { }


    public DbSet<Department> Departments { get; set; }
    public DbSet<Instructor> Instructors { get; set; }
    public DbSet<Course> Courses { get; set; }
    public DbSet<Student> Students { get; set; }
    public DbSet<Enrollment> Enrollments { get; set; }
  }
```

```
        }
```

3. Create the Entity Classes:

   - Create classes for Department, Instructor, Course, Student, and Enrollment in the Models folder. Here is an example for the Department class:

```
namespace SchoolAppCoreMVC.Models
{
  public class Department
  {
    public int DepartmentID { get; set; }
    public string Name { get; set; }
    public decimal Budget { get; set; }
    public DateTime StartDate { get; set; }
  }
}
```

   - Repeat similar steps for the other entity classes.

2. Configure **the DbContext in Startup.cs / Program.cs:**

   - Open the Startup.cs / **Program.cs (for .NET 6+)** file

     If Startup.cs then add the following code in the ConfigureServices method:

```
public void ConfigureServices(IServiceCollection services){
  services.AddDbContext<SchoolContext>(options =>
options.UseSqlServer(Configuration.GetConnectionString("SchoolDBConnectionString")));
  services.AddRazorPages();
}
```

     If no Startup.cs then open **Program.cs (for .NET 6+)** **.** Please add:

```
using Microsoft.EntityFrameworkCore;
using SchoolAppCoreRazor.Models;
...
Builder.Services.AddRazorPages();
builder.Services.AddDbContext<SchoolContext>(options =>
options.UseSqlServer(builder.Configuration.GetConnectionString("SchoolDBConnection
String")));
```

Step 4: Creating Controllers and Views

Objective: Create controllers and views to manage Students, Courses, Instructors, Enrollments, and Departments.

Steps:

1. Create Controllers:
    - Right-click on the Controllers folder, select Add > Controller, and choose MVC Controller with views, using Entity Framework. Name it DepartmentsController.
    - Select Department as the model class and SchoolContext as the data context class. Click Add.
    - Repeat the process for Students, Courses, Instructors, and Enrollments.
2. Modify the Views:
    - Open the views for each controller (e.g., Views/Departments/Index.cshtml) and ensure they display the relevant data.

Step 5: Testing the Application

Objective: Run the application and test the functionality.

Steps:

1. Run the Application:
    - Press F5 or click the Start button to run the application.
    - Navigate to the respective URLs (e.g., /Departments, /Students) to access the list of entities.
2. Verify the Output:
    - Check the lists displayed on the pages to verify that the data is being retrieved from the database correctly.

Exercise 3: Map Project of Exercise 1 & 2 to GitHub repository

1. Initialize a Local Git Repository:
    - In Visual Studio, go to View > Git Changes to open the Git Changes window.
    - Click the Initialize Repository button to create a local Git repository for your project.
2. Create a GitHub Repository:
    - Go to your GitHub profile and click on the Repositories tab.

- Click the New button to create a new repository.
- Name the repository exercise 1 (ASPNETCoreRazorPageSchoolApp) & exercise 2 (ASPNETCoreMVCSchoolApp)
- Add a description (optional) and choose whether the repository should be public or private.
- Do not initialize the repository with a README file, .gitignore, or license.
- Click Create repository.

3. Map the Local Repository to GitHub:
    - In Visual Studio, go to the Git Changes window.
    - Click the Settings icon (gear icon) and select Repository Settings.
    - Under Remotes, click Add.
    - Enter origin as the remote name and paste the URL of your GitHub repository.
    - Click Save.
4. Commit and Push Changes:
    - Go to the Git Changes window in Visual Studio.
    - Stage all changes by clicking the + button.
    - Enter a commit message, e.g., "Initial commit of ASP.NET Core MVC Student App".
    - Click Commit All and then Push to push the changes to GitHub.

Exercise 4: ASP.Net Report .rdlc

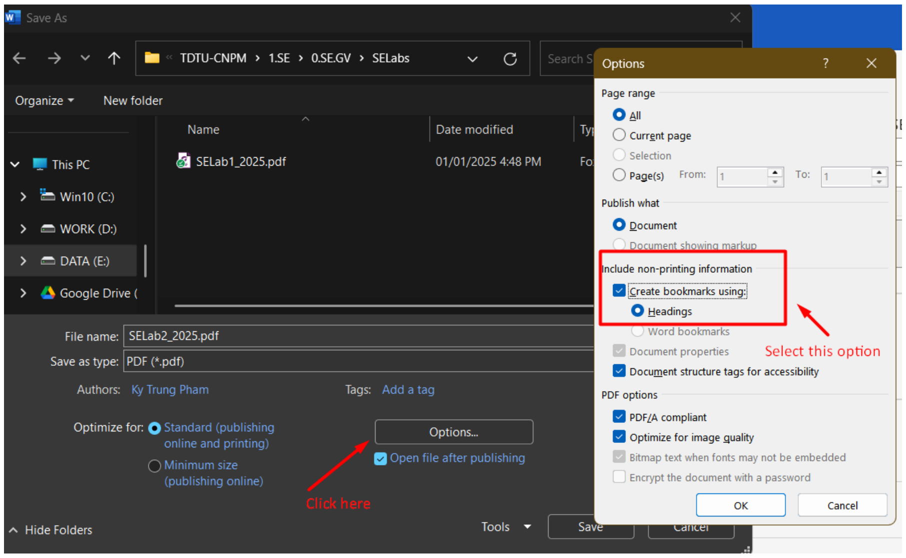
- Make a form and report

Submit on eLearning: Summary Report

- Introduction: Briefly describe the purpose of the exercises.
- Exercises: List each exercise with a brief description of what you did and learned.
- Screenshots: Include the screenshots of every step for each exercise and steps (both Visual Studio and Github) (Index each step with sequential number) and the output when press F5.
- Conclusion: Summarize your overall learning experience and any challenges you faced.

Submit 2 files on eLearning:

- StudentID_StudentName.zip (contain Windows Web Project & .sql file) for both Class Activity and Homework.
- StudentID_StudentName.pdf (require to create bookmarks)



TDTU-CNPM › 1.SE › 0.SE.GV › SELabs

Name	Date modified
SELab1_2025.pdf	01/01/2025 4:48 PM

File name: SELab2_2025.pdf
Save as type: PDF (*.pdf)
Authors: Ky Trung Pham
Tags: Add a tag
Optimize for: [✓ Standard (publishing online and printing)] Minimum size (publishing online)
Options...
[✓] Open file after publishing

Page range: [✓ All] Current page Selection Page(s) From: 1 To: 1
Publish what: [✓ Document] Document showing markup
Include non-printing information
[✓] Create bookmarks using: [✓ Headings] Word bookmarks
[✓] Document properties
[✓] Document structure tags for accessibility
PDF options
[✓] PDF/A compliant
[✓] Optimize for image quality
[✓] Bitmap text when fonts may not be embedded
[ ] Encrypt the document with a password

Notes: There may be a typo/mistake during creating this document. Please correct it by yourself.

Enjoy ☺ Email: tg_phamthaikytrung@tdtu.edu.vn