**Unit 3: HTTP and ASP.NET Core (3 Hours)**

**HTTP Basics**

**1. HTTP Request/Response Message Format**

* **Description**: HTTP (Hypertext Transfer Protocol) is the foundation of communication for the web. A client (browser) sends a request to the server, and the server responds with a message containing the requested resources.
* **Syntax**: **Request Message**:

GET /index.html HTTP/1.1

Host: www.example.com

User-Agent: Mozilla/5.0

Accept: text/html

**Response Message**:

HTTP/1.1 200 OK

Content-Type: text/html

Content-Length: 123

<html>

<body>

<h1>Hello, World!</h1>

</body>

</html>

* **Uses**:
  + **GET**: Retrieve data from the server.
  + **POST**: Submit data to the server.
  + **PUT**: Update data on the server.
  + **DELETE**: Remove data from the server.
* **Example**:
  + A browser sends an HTTP GET request for a webpage.
  + The server responds with an HTML page.
* **Project Example**:
  + Create a simple ASP.NET Core web application where a browser requests data, and the server responds with "Hello, World!" in plain text.

**Steps to Create a Simple ASP.NET Core Web Application**

This project demonstrates a basic ASP.NET Core web application where a browser requests data, and the server responds with "Hello, World!" in plain text.

**Step 1: Create a New ASP.NET Core Project**

1. **Open Visual Studio**:
   * Go to File > New > Project.
2. **Select the Project Template**:
   * Choose **ASP.NET Core Empty**.
   * Click **Next**.
3. **Configure the Project**:
   * Enter a name for your project, e.g., HelloWorldApp.
   * Select the desired location to save the project.
   * Click **Next**.
4. **Select the Target Framework**:
   * Choose the latest **.NET Core** or **.NET** version (e.g., .NET 6 or .NET 7).
   * Click **Create**.

**Step 2: Add Minimal API Logic in Program.cs**

1. Open the Program.cs file (the default entry point of an ASP.NET Core application).
2. Replace the content of Program.cs with the following code:

using Microsoft.AspNetCore.Builder;

using Microsoft.AspNetCore.Http;

var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

// Handle all HTTP GET requests and respond with "Hello, World!"

app.MapGet("/", async context =>

{

await context.Response.WriteAsync("Hello, World!");

});

app.Run();

**Code Explanation**

1. **WebApplication.CreateBuilder**:
   * Creates the application builder, which is used to configure the app.
2. **app.MapGet**:
   * Maps the GET HTTP requests to the root path ("/") to a handler that sends a plain text response.
3. **context.Response.WriteAsync**:
   * Writes the response as plain text ("Hello, World!").
4. **app.Run()**:
   * Starts the application.

**Step 3: Run the Application**

1. Press Ctrl + F5 or click **Run** to start the application.
2. Once the application starts, it will open a browser window with the following URL:

http://localhost:<port>/

Replace <port> with the port number shown in the console.

1. You should see the message "Hello, World!" displayed in the browser.

**Step 4: Test the Application**

1. Open your browser and navigate to:

http://localhost:<port>/

1. Use tools like **Postman** or **curl** to test the response:
   * **Postman**:
     + Send a GET request to the URL.
   * **curl**:

curl http://localhost:<port>/

**Optional: Add More Routes**

If you'd like to handle additional paths, you can modify Program.cs:

app.MapGet("/hello", async context =>

{

await context.Response.WriteAsync("Welcome to the Hello endpoint!");

});

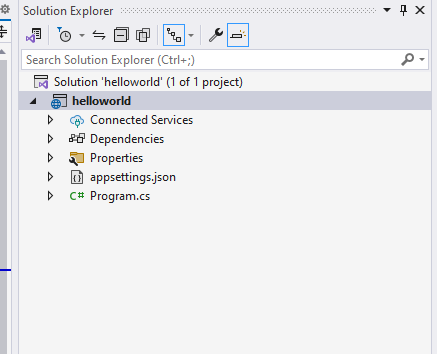
app.MapGet("/goodbye", async context =>

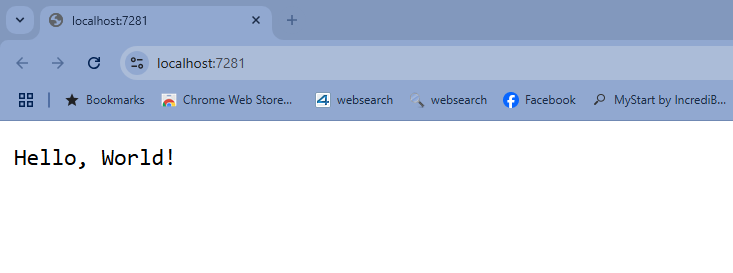
{

await context.Response.WriteAsync("Goodbye, World!");

});

* Visiting /hello will respond with "Welcome to the Hello endpoint!".
* Visiting /goodbye will respond with "Goodbye, World!".





**ASP.NET Core Architecture**

**2. Overview**

* **Description**: ASP.NET Core is a cross-platform framework for building modern web applications. It is modular, high-performance, and lightweight.
* **Core Components**:
  + **Middleware**: Processes HTTP requests in a pipeline.
  + **Routing**: Maps incoming requests to appropriate controllers or endpoints.
  + **Dependency Injection**: Provides services to controllers and other components.
* **Syntax**: Example of configuring middleware in Startup.cs:

public class Startup

{

public void Configure(IApplicationBuilder app)

{

app.UseRouting();

app.UseEndpoints(endpoints =>

{

endpoints.MapGet("/", async context =>

{

await context.Response.WriteAsync("Hello, World!");

});

});

}

}

* **Uses**:
  + Middleware allows you to process requests, handle authentication, and modify responses.
  + Routing directs incoming HTTP requests to specific controllers or endpoints.
* **Example**:
  + A pipeline with middleware that logs requests and serves a response.
* **Project Example**:
  + Create an ASP.NET Core app with custom middleware to log each HTTP request and return a custom message.

**Steps to Create an ASP.NET Core App with Custom Middleware**

This project demonstrates how to build an ASP.NET Core application that uses custom middleware to log each HTTP request and return a custom response.

**Step 1: Create a New ASP.NET Core Project**

1. **Open Visual Studio**:
   * Go to File > New > Project.
2. **Select the Project Template**:
   * Choose **ASP.NET Core Empty**.
   * Click **Next**.
3. **Configure the Project**:
   * Enter a name for your project, e.g., RequestLoggerApp.
   * Select the desired location to save the project.
   * Click **Next**.
4. **Select the Target Framework**:
   * Choose the latest **.NET Core** or **.NET** version (e.g., .NET 6 or .NET 7).
   * Click **Create**.

**Step 2: Add Custom Middleware**

1. Open the Program.cs file.
2. Replace the content of Program.cs with the following code:

using Microsoft.AspNetCore.Builder;

using Microsoft.AspNetCore.Http;

using Microsoft.Extensions.Logging;

using System.Threading.Tasks;

var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

// Custom Middleware to log HTTP requests

app.Use(async (context, next) =>

{

// Log Request Information

Console.WriteLine($"HTTP Request: {context.Request.Method} {context.Request.Path}");

// Continue to the next middleware

await next.Invoke();

// Log Response Information

Console.WriteLine($"HTTP Response: {context.Response.StatusCode}");

});

// Endpoint to return a custom response

app.MapGet("/", async context =>

{

await context.Response.WriteAsync("Welcome! Your request has been logged.");

});

app.Run();

**Code Explanation**

1. **Custom Middleware**:
   * **app.Use**:
     + Middleware to intercept each HTTP request.
     + Logs the HTTP request method (e.g., GET, POST) and path (e.g., /).
     + After processing the request, it logs the HTTP response status code.
2. **Request Logging**:
   * context.Request.Method: Logs the HTTP method (e.g., GET).
   * context.Request.Path: Logs the requested URL path.
3. **Response Logging**:
   * context.Response.StatusCode: Logs the status code returned to the client.
4. **Next Middleware**:
   * next.Invoke() passes the request to the next middleware in the pipeline.
5. **Custom Endpoint**:
   * **app.MapGet** defines a route for the root URL (/) that returns a custom message: "Welcome! Your request has been logged."

**Step 3: Run the Application**

1. Press Ctrl + F5 or click **Run** to start the application.
2. Once the application starts, it will open a browser window with the following URL:

http://localhost:<port>/

1. Check the console output for logs.

**Step 4: Test the Middleware**

1. Open a browser and navigate to:

http://localhost:<port>/

1. **Console Output**:
   * When you visit the page, you will see logs like:

HTTP Request: GET /

HTTP Response: 200

1. Use tools like **Postman** or **curl** to test:
   * **Postman**:
     + Send a GET request to http://localhost:<port>/.
   * **curl**:

curl http://localhost:<port>/

**Optional: Add More Custom Middleware**

You can add additional middleware to handle other scenarios, such as custom error handling or request validation.

// Custom Middleware for Error Handling

app.Use(async (context, next) =>

{

try

{

await next.Invoke();

}

catch (Exception ex)

{

Console.WriteLine($"Error: {ex.Message}");

context.Response.StatusCode = 500;

await context.Response.WriteAsync("An error occurred while processing your request.");

}

});

**Optional: Add Another Route**

You can add more routes to test different paths:

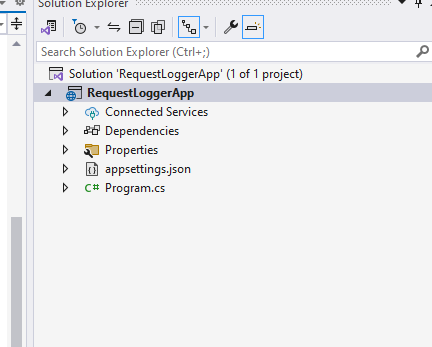
app.MapGet("/about", async context =>

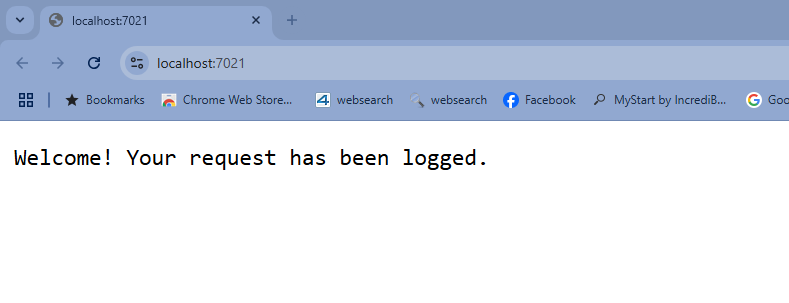
{

await context.Response.WriteAsync("This is the About page!");

});

* Visiting /about will return "This is the About page!".





**MVC Pattern and Its Implementation**

**3. MVC Pattern**

* **Description**: MVC (Model-View-Controller) is a design pattern that separates an application into three components:
  + **Model**: Represents data and business logic.
  + **View**: Handles the presentation layer (UI).
  + **Controller**: Manages user input and updates the model and view.
* **Syntax**: Example of a Controller in ASP.NET Core MVC:

public class HomeController : Controller

{

public IActionResult Index()

{

return View();

}

}

Example of a View (Index.cshtml):

<h1>Welcome to ASP.NET Core MVC!</h1>

Example of a Model:

public class Student

{

public int Id { get; set; }

public string Name { get; set; }

}

* **Uses**:
  + MVC organizes code for better scalability and maintainability.
  + Separates concerns, making it easier to update individual components.
* **Example**:
  + A blog application where users can view posts (View), submit comments (Controller), and store data in a database (Model).
* **Project Example**:
  + Build a basic CRUD (Create, Read, Update, Delete) app for managing students using ASP.NET Core MVC.

**Steps to Build a Basic CRUD App for Managing Students Using ASP.NET Core MVC**

This project demonstrates how to build a basic ASP.NET Core MVC application for managing students. The app will allow users to create, view, edit, and delete student records.

**Step 1: Create a New ASP.NET Core MVC Project**

1. **Open Visual Studio**:
   * Go to File > New > Project.
2. **Select the Project Template**:
   * Choose **ASP.NET Core Web App (Model-View-Controller)**.
   * Click **Next**.
3. **Configure the Project**:
   * Enter a name for your project, e.g., StudentManagementMVC.
   * Select the desired location to save the project.
   * Click **Next**.
4. **Select the Target Framework**:
   * Choose the latest **.NET Core** or **.NET** version (e.g., .NET 6 or .NET 7).
   * Click **Create**.

**Step 2: Create the Student Model**

1. Add a folder named Models in the root of your project.
2. Add a Student.cs file inside the Models folder and define the Student class:

namespace StudentManagementMVC.Models

{

public class Student

{

public int Id { get; set; }

public string Name { get; set; }

public double Marks { get; set; }

}

}

**Step 3: Add a StudentController**

1. Add a folder named Controllers (if it doesn’t already exist).
2. Right-click the Controllers folder > **Add > Controller** > **MVC Controller - Empty**.
3. Name the controller StudentController.
4. Replace the content of StudentController.cs with the following code:

using Microsoft.AspNetCore.Mvc;

using StudentManagementMVC.Models;

using System.Collections.Generic;

using System.Linq;

namespace StudentManagementMVC.Controllers

{

public class StudentController : Controller

{

// Static list to simulate a database

private static List<Student> students = new List<Student>

{

new Student { Id = 1, Name = "Alice", Marks = 85 },

new Student { Id = 2, Name = "Bob", Marks = 92 }

};

// Index: Display all students

public IActionResult Index()

{

return View(students);

}

// Create (GET): Show the create form

[HttpGet]

public IActionResult Create()

{

return View();

}

// Create (POST): Add a new student

[HttpPost]

public IActionResult Create(Student student)

{

student.Id = students.Count > 0 ? students.Max(s => s.Id) + 1 : 1;

students.Add(student);

return RedirectToAction("Index");

}

// Edit (GET): Show the edit form

[HttpGet]

public IActionResult Edit(int id)

{

var student = students.FirstOrDefault(s => s.Id == id);

if (student == null) return NotFound();

return View(student);

}

// Edit (POST): Update an existing student

[HttpPost]

public IActionResult Edit(Student updatedStudent)

{

var student = students.FirstOrDefault(s => s.Id == updatedStudent.Id);

if (student == null) return NotFound();

student.Name = updatedStudent.Name;

student.Marks = updatedStudent.Marks;

return RedirectToAction("Index");

}

// Delete: Remove a student

public IActionResult Delete(int id)

{

var student = students.FirstOrDefault(s => s.Id == id);

if (student != null)

{

students.Remove(student);

}

return RedirectToAction("Index");

}

}

}

**Step 4: Create Views**

1. **Add a Folder for Views**:
   * Add a folder named Student under the Views folder.
2. **Create Views**:
   * Add the following Razor views for the Student controller.

**1. Index.cshtml (List Students)**

@model List<StudentManagementMVC.Models.Student>

<h1>Student List</h1>

<a href="/Student/Create" class="btn btn-primary">Add New Student</a>

<table class="table">

<thead>

<tr>

<th>ID</th>

<th>Name</th>

<th>Marks</th>

<th>Actions</th>

</tr>

</thead>

<tbody>

@foreach (var student in Model)

{

<tr>

<td>@student.Id</td>

<td>@student.Name</td>

<td>@student.Marks</td>

<td>

<a href="/Student/Edit/@student.Id" class="btn btn-warning">Edit</a>

<a href="/Student/Delete/@student.Id" class="btn btn-danger">Delete</a>

</td>

</tr>

}

</tbody>

</table>

**2. Create.cshtml (Add Student)**

@model StudentManagementMVC.Models.Student

<h1>Add New Student</h1>

<form method="post" action="/Student/Create">

<div class="form-group">

<label for="Name">Name</label>

<input type="text" id="Name" name="Name" class="form-control" required />

</div>

<div class="form-group">

<label for="Marks">Marks</label>

<input type="number" id="Marks" name="Marks" class="form-control" required />

</div>

<button type="submit" class="btn btn-primary">Save</button>

</form>

**3. Edit.cshtml (Edit Student)**

@model StudentManagementMVC.Models.Student

<h1>Edit Student</h1>

<form method="post" action="/Student/Edit">

<input type="hidden" name="Id" value="@Model.Id" />

<div class="form-group">

<label for="Name">Name</label>

<input type="text" id="Name" name="Name" value="@Model.Name" class="form-control" required />

</div>

<div class="form-group">

<label for="Marks">Marks</label>

<input type="number" id="Marks" name="Marks" value="@Model.Marks" class="form-control" required />

</div>

<button type="submit" class="btn btn-primary">Update</button>

</form>

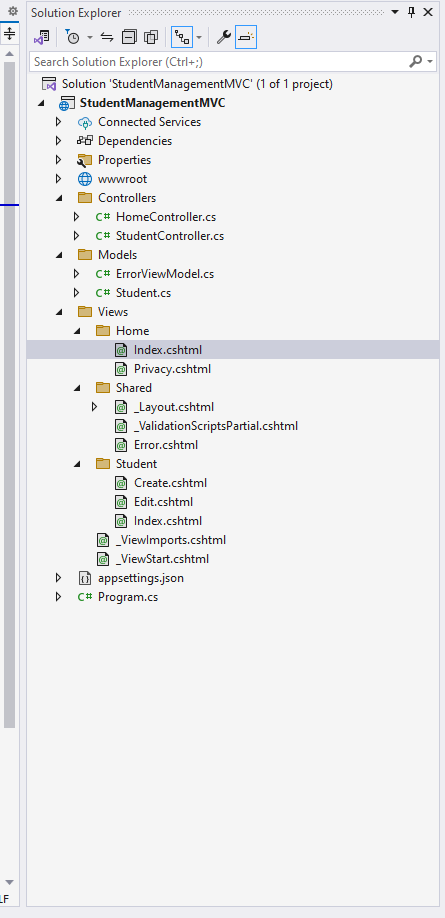
**Step 5: Run the Application**

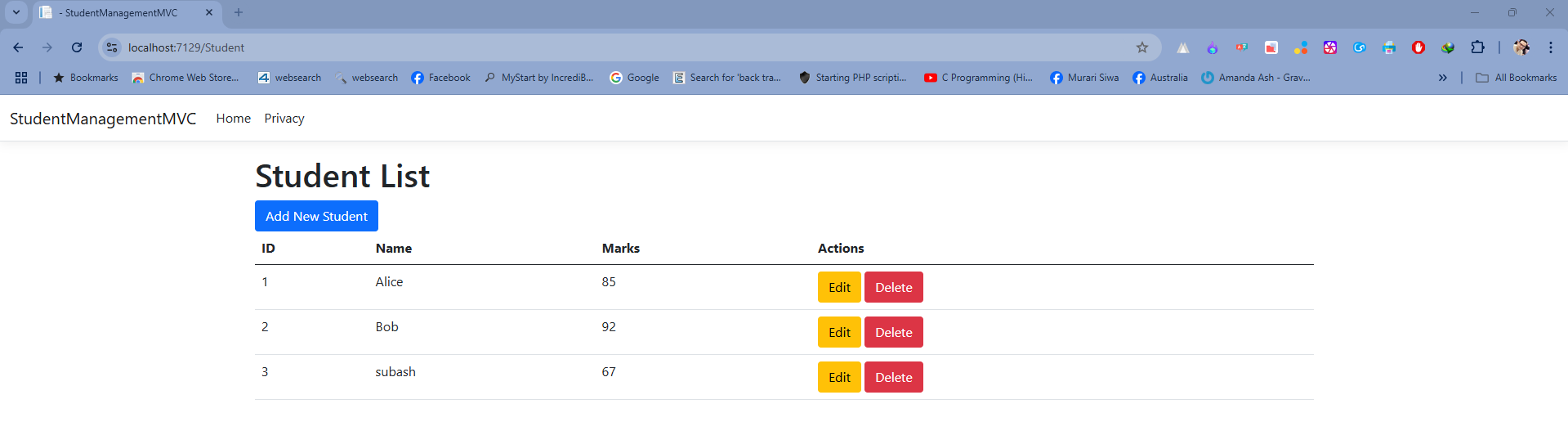
1. Press Ctrl + F5 or click **Run**.
2. Navigate to:

http://localhost:<port>/Student

**Application Features**

1. **Index Page**:
   * Displays all students with options to edit or delete.
2. **Create Page**:
   * Allows adding a new student.
3. **Edit Page**:
   * Allows updating student details.
4. **Delete**:
   * Removes a student from the list.





**Project Conventions**

**4. ASP.NET Core Project Conventions**

* **Description**: Conventions in ASP.NET Core define how files and folders should be structured for ease of development and to follow best practices.
* **Conventions**:
  + **Controllers**:
    - Stored in the Controllers folder.
    - Should inherit from Controller or ControllerBase.
  + **Views**:
    - Stored in the Views folder.
    - Should have a subfolder for each controller (e.g., Views/Home).
  + **Models**:
    - Stored in the Models folder.
    - Should contain classes that represent data and business logic.
* **Syntax**: Example folder structure for an ASP.NET Core MVC project:

MyApp/

├── Controllers/

│ └── HomeController.cs

├── Models/

│ └── Student.cs

├── Views/

│ └── Home/

│ └── Index.cshtml

├── wwwroot/

│ └── css/

│ └── js/

└── Program.cs

* **Uses**:
  + Conventions simplify development and ensure consistency.
  + Helps the framework automatically locate views, controllers, and other resources.
* **Example**:
  + A HomeController uses a view located in Views/Home/Index.cshtml by default.
* **Project Example**:
  + Create an ASP.NET Core MVC app with the above conventions and demonstrate routing between controllers and views.

**Creating an ASP.NET Core MVC App with Conventions and Routing**

This project demonstrates how to create an ASP.NET Core MVC application that follows default conventions and routes between controllers and views.

**Step 1: Create a New ASP.NET Core MVC Project**

1. **Open Visual Studio**:
   * Go to File > New > Project.
2. **Select the Project Template**:
   * Choose **ASP.NET Core Web App (Model-View-Controller)**.
   * Click **Next**.
3. **Configure the Project**:
   * Enter the project name, e.g., RoutingDemoApp.
   * Choose the location to save the project.
   * Click **Next**.
4. **Select the Target Framework**:
   * Choose the latest **.NET Core** or **.NET** version (e.g., .NET 6 or .NET 7).
   * Click **Create**.

**Step 2: Add the StudentController**

1. **Add a Controller**:
   * Right-click the **Controllers** folder in the Solution Explorer.
   * Select **Add > Controller > MVC Controller - Empty**.
   * Name the controller StudentController.
2. **Define Actions in StudentController**: Add the following actions to the StudentController:

using Microsoft.AspNetCore.Mvc;

namespace RoutingDemoApp.Controllers

{

public class StudentController : Controller

{

// Action for Index (Home Page)

public IActionResult Index()

{

return View();

}

// Action for About Page

public IActionResult About()

{

ViewBag.Message = "This is the About page for Students.";

return View();

}

// Action for Details Page

public IActionResult Details(int id)

{

ViewBag.StudentId = id;

return View();

}

}

}

**Step 3: Create Views for StudentController**

1. **Create the Student Folder in Views**:
   * Right-click the Views folder > **Add > New Folder**.
   * Name the folder **Student**.
2. **Add Views**: Add the following views to the Views/Student folder:

**a. Index.cshtml**:

<h1>Welcome to the Student Management System</h1>

<p><a href="/Student/About">Go to About Page</a></p>

<p><a href="/Student/Details/1">View Details for Student 1</a></p>

**b. About.cshtml**:

<h1>About Students</h1>

<p>@ViewBag.Message</p>

**c. Details.cshtml**:

<h1>Student Details</h1>

<p>Student ID: @ViewBag.StudentId</p>

**Step 4: Configure Routing (Optional)**

By default, ASP.NET Core MVC uses the **convention-based routing** defined in Program.cs:

app.MapControllerRoute(

name: "default",

pattern: "{controller=Home}/{action=Index}/{id?}");

**Explanation:**

* **{controller=Home}**: Specifies the default controller (HomeController) if none is provided.
* **{action=Index}**: Specifies the default action (Index) if none is provided.
* **{id?}**: The id is optional and can be used for parameters.

You don’t need to modify this unless you want to add custom routes.

**Step 5: Run the Application**

1. **Start the App**:
   * Press Ctrl + F5 or click **Run**.
2. **Test Routes**:
   * **Index Page**: Navigate to http://localhost:<port>/Student.
     + Displays the Index.cshtml view.
   * **About Page**: Navigate to http://localhost:<port>/Student/About.
     + Displays the About.cshtml view with the message from ViewBag.
   * **Details Page**: Navigate to http://localhost:<port>/Student/Details/1.
     + Displays the Details.cshtml view with Student ID: 1.

**Step 6: Add Links for Navigation**

To demonstrate routing between controllers and views, update Index.cshtml to include links to other actions dynamically:

<h1>Welcome to the Student Management System</h1>

<p><a href="@Url.Action("About", "Student")">Go to About Page</a></p>

<p><a href="@Url.Action("Details", "Student", new { id = 1 })">View Details for Student 1</a></p>

**Project Structure**

After implementing the above steps, your project structure should look like this:

RoutingDemoApp/

├── Controllers/

│ └── StudentController.cs

├── Views/

│ ├── Shared/

│ │ └── \_Layout.cshtml

│ └── Student/

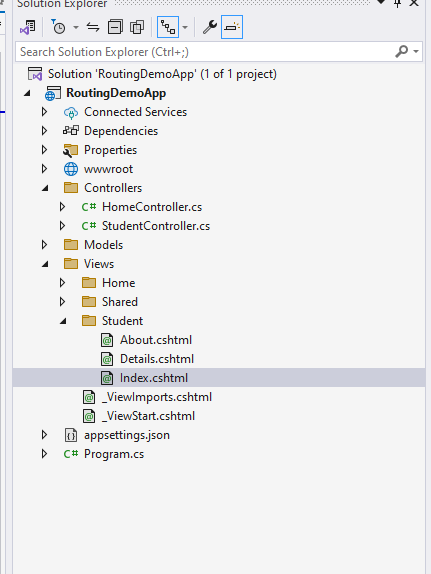
│ ├── Index.cshtml

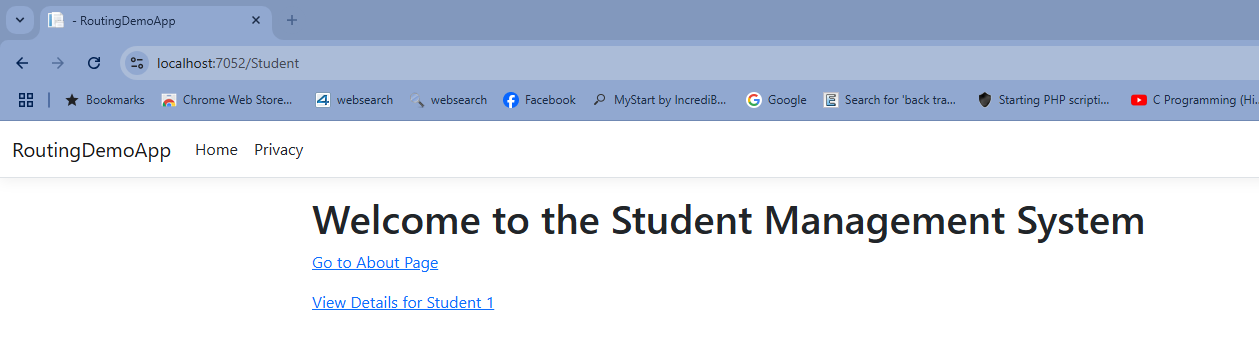
│ ├── About.cshtml

│ └── Details.cshtml

├── wwwroot/

└── Program.cs





**Final Project Example**

**Simple ASP.NET Core MVC Application**

**Goal**: Create an ASP.NET Core MVC application to display and manage student data.

1. **Setup**:
   * Create a new ASP.NET Core MVC project in Visual Studio.
   * Add a Student model in the Models folder.
2. **Student Model**:

public class Student

{

public int Id { get; set; }

public string Name { get; set; }

public double Marks { get; set; }

}

1. **Student Controller**:

public class StudentController : Controller

{

private static List<Student> students = new List<Student>

{

new Student { Id = 1, Name = "Alice", Marks = 85 },

new Student { Id = 2, Name = "Bob", Marks = 92 }

};

public IActionResult Index()

{

return View(students);

}

}

1. **Student View** (Index.cshtml):

<h1>Student List</h1>

<table>

<tr>

<th>ID</th>

<th>Name</th>

<th>Marks</th>

</tr>

@foreach (var student in Model)

{

<tr>

<td>@student.Id</td>

<td>@student.Name</td>

<td>@student.Marks</td>

</tr>

}

</table>

1. **Run the App**:
   * Start the app.
   * Navigate to /Student to view the list of students.