NATIONAL ECONOMICS UNIVERSITY
SCHOOL OF INFORMATION TECHNOLOGY AND DIGITAL ECONOMICS

CHAPTER IV

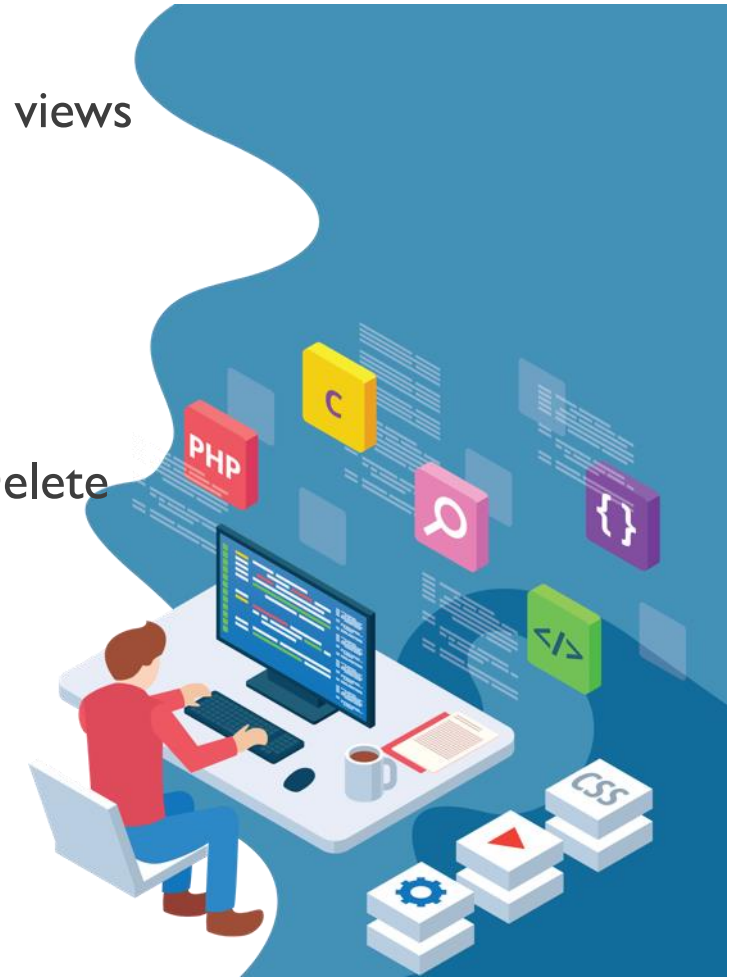
ASP.NET WEB APP (MVC)

PHAM THAO

OUTLINE

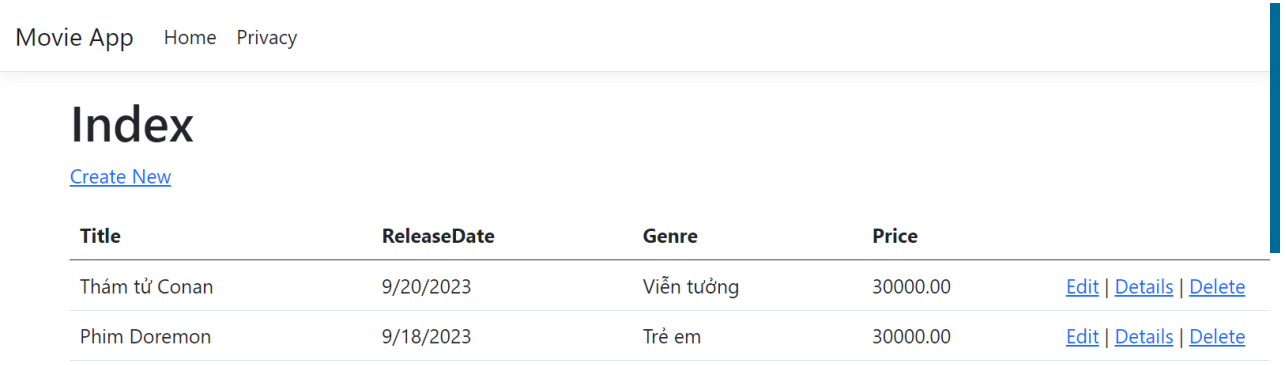
ASP.NET Core Web App MVC

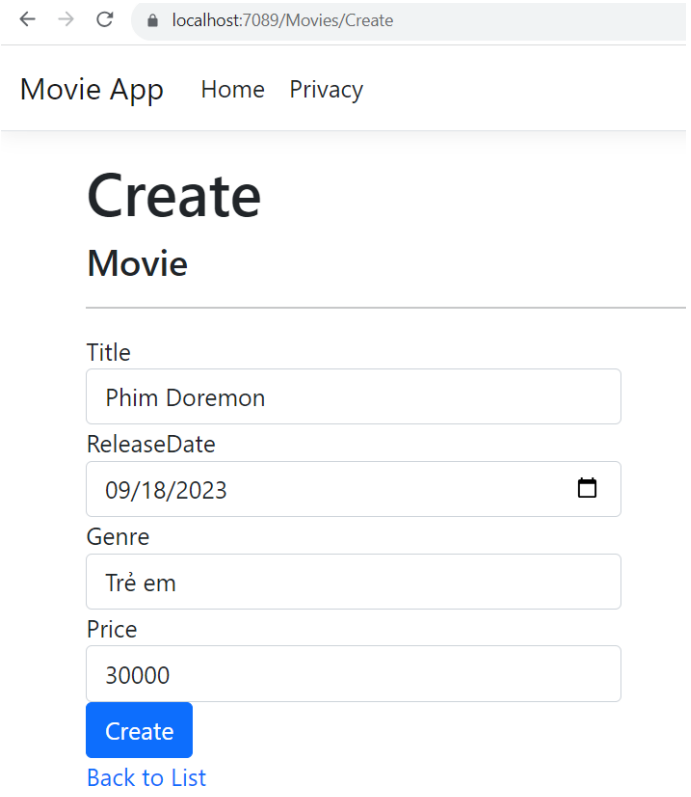
- The Model-View-Controller (MVC) architectural
- Add a controller
  - Change Method
- Add a view
  - Change Layout
  - Passing Data from the Controller to the View
- Add a model
  - Scaffolding movie Pages
  - Initial migration
- Run MovieWebsite
- appsettings.json
- Work with a database
- Controller actions and views
- Add search
- Add a new field
- Add validation
- Examine Details and Delete

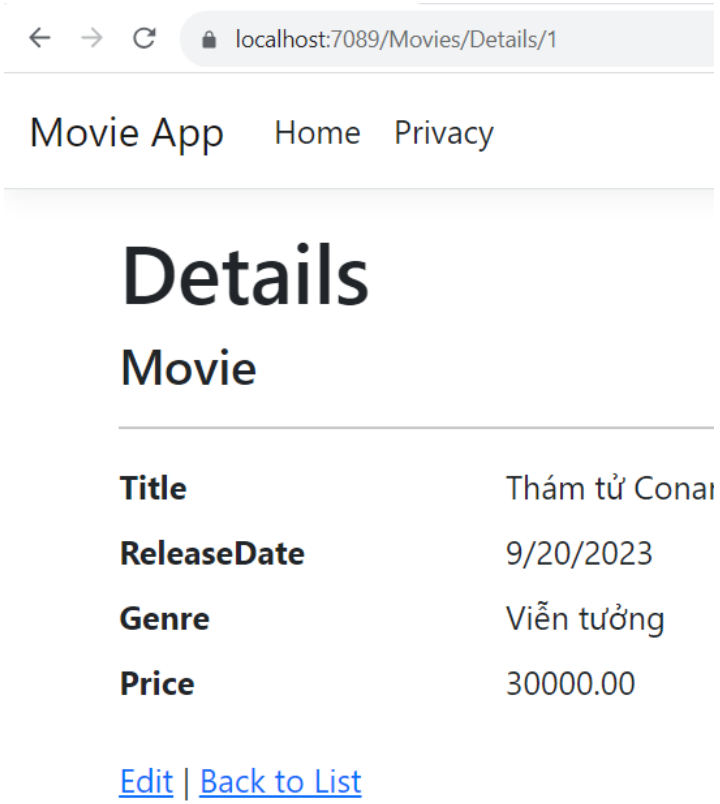


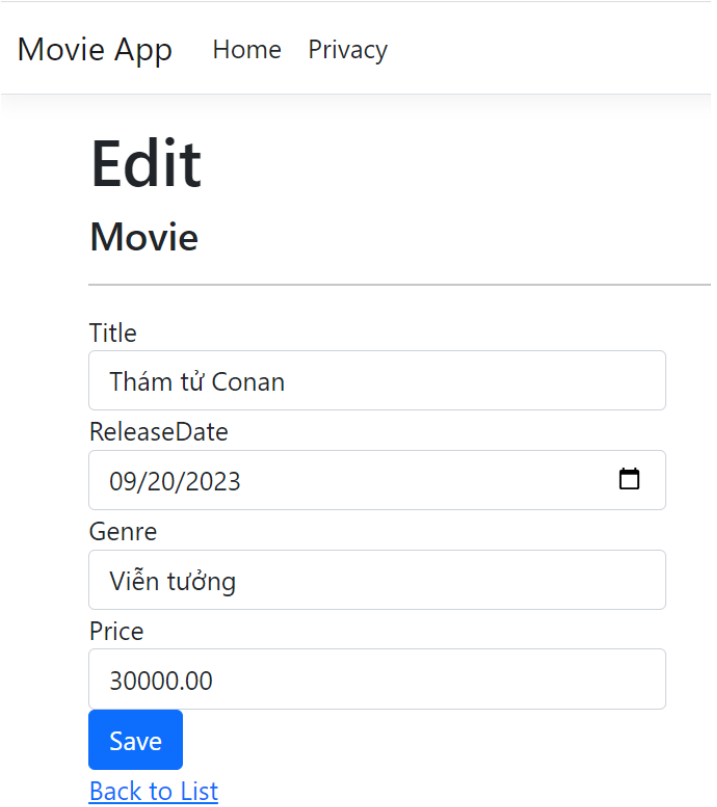
THE OBJECTIVES

ASP.NET Core Web App (MVC)

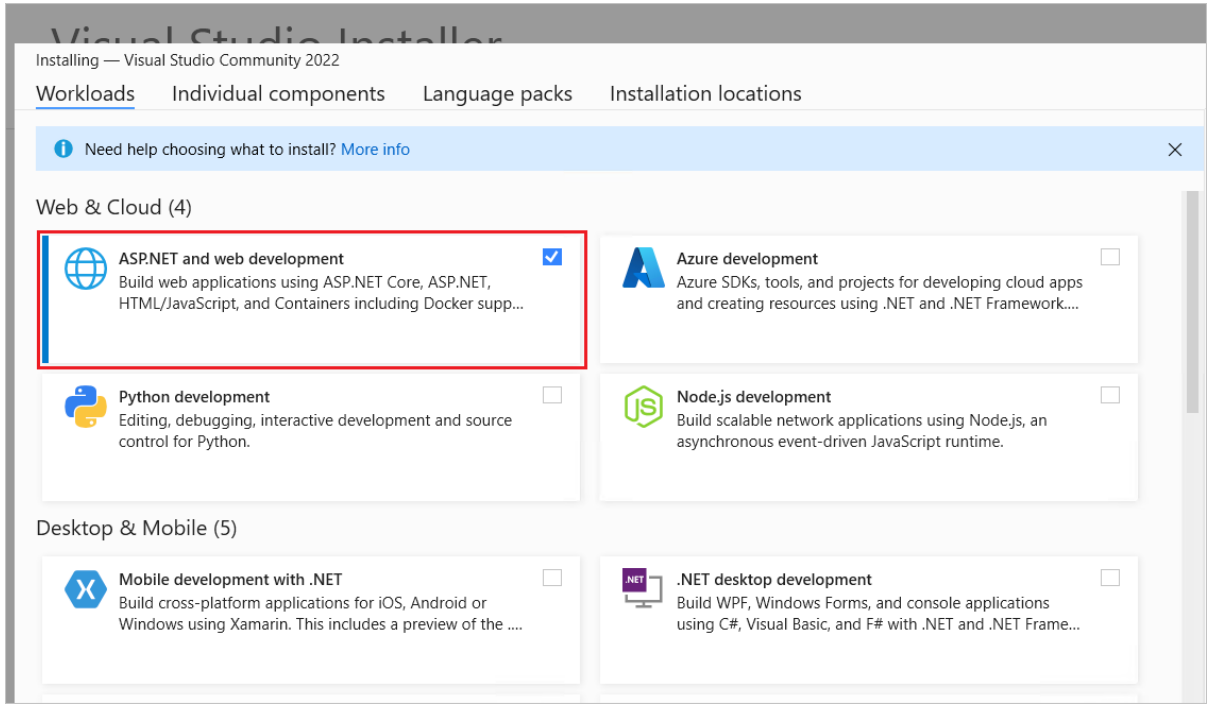








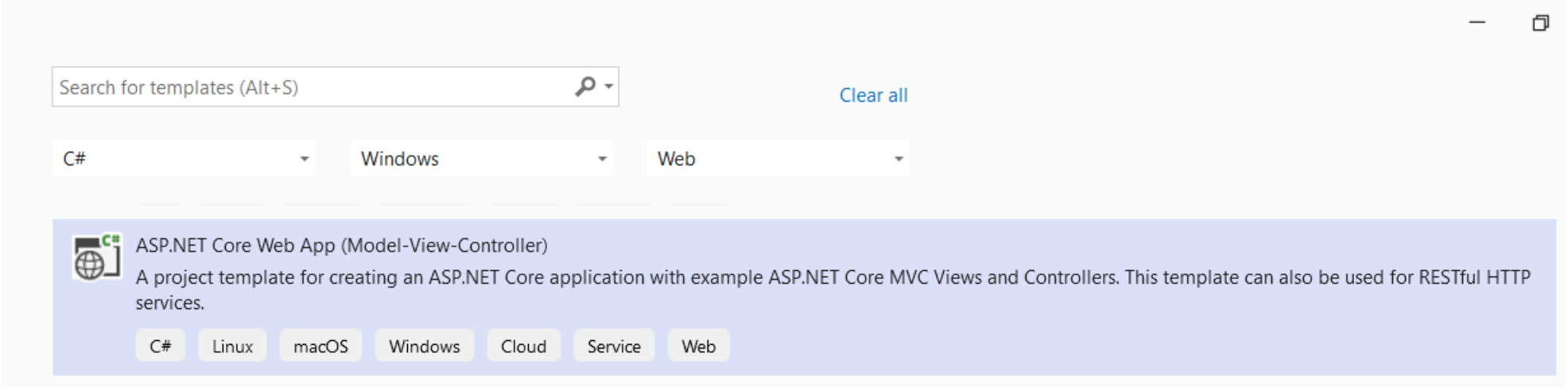
SETUP ASP.NET AND WEB DEVELOPMENT



- ASP.NET is a popular web-development framework for building web apps on the .NET platform.
- ASP.NET Core is the open-source version of ASP.NET, that runs on macOS, Linux, and Windows. ASP.NET Core was first released in 2016 and is a re-design of earlier Windows-only versions of ASP.NET.
  - ASP.NET Core is designed to allow runtime components, APIs, compilers, and languages evolve quickly, while still providing a stable and supported platform to keep apps running.
  - Multiple versions of ASP.NET Core can exist side by side on the same server. Meaning one app can adopt the latest version, while other apps keep running on the version they were tested on.
  - ASP.NET Core provides various support lifecycle options to meet the needs of your app.

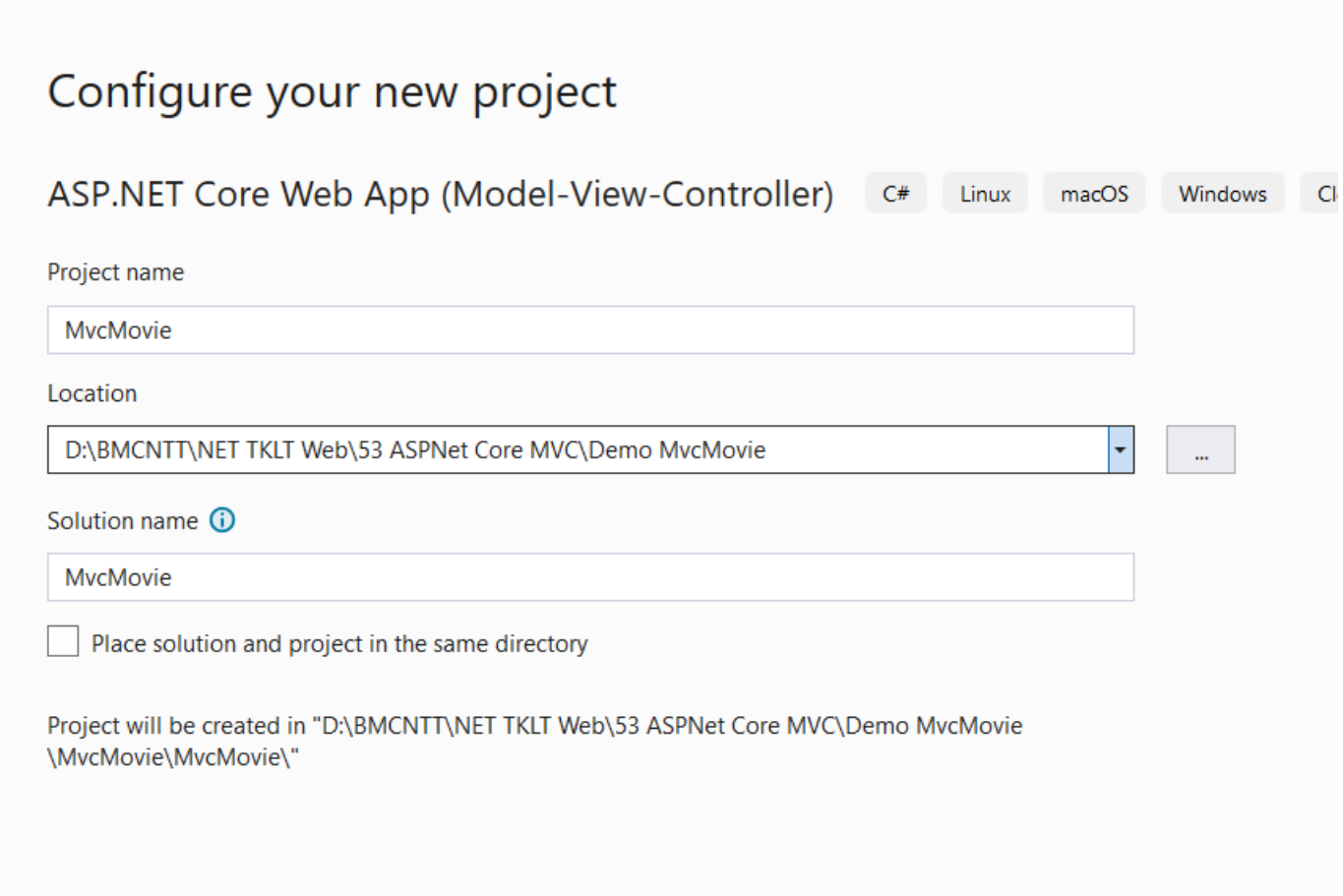
https://learn.microsoft.com/vi-vn/aspnet/core/tutorials/first-mvc-app/start-mvc?view=aspnetcore-7.0&tabs=visual-studio

CREATE A NEW PROJECT



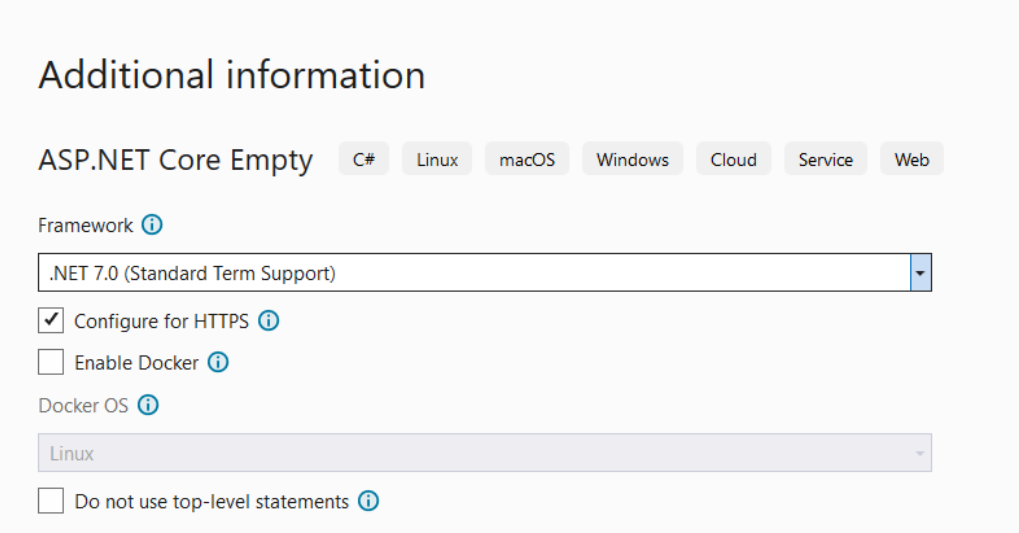
- ASP.NET Core Web App (MVC):
  - Similar to the "ASP.NET Core Web App" template, but explicitly emphasizes the use of the Model-View-Controller (MVC) architectural pattern.
  - Provides a structured way to build web applications where data, presentation, and logic are separated into models, views, and controllers.
  - Suitable for developers familiar with MVC and who prefer to build applications following this pattern.
  - Offers a balanced approach between server-rendered views and API endpoints for data access.

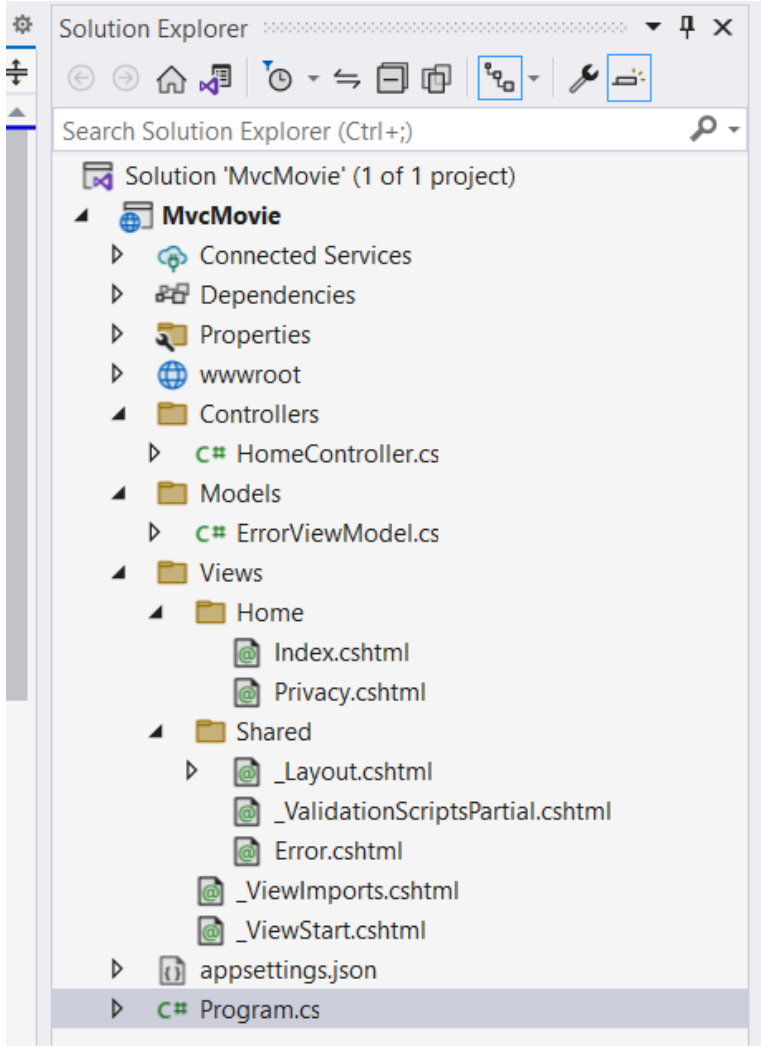
CREATE A NEW PROJECT



- Start Visual Studio and select Create a new project.
- In the Create a new project dialog, select ASP.NET Core Web App (Model-View-Controller) > Next.
- In the Configure your new project dialog, enter MvcMovie for Project name.
- Select Next.

CREATE A NEW PROJECT

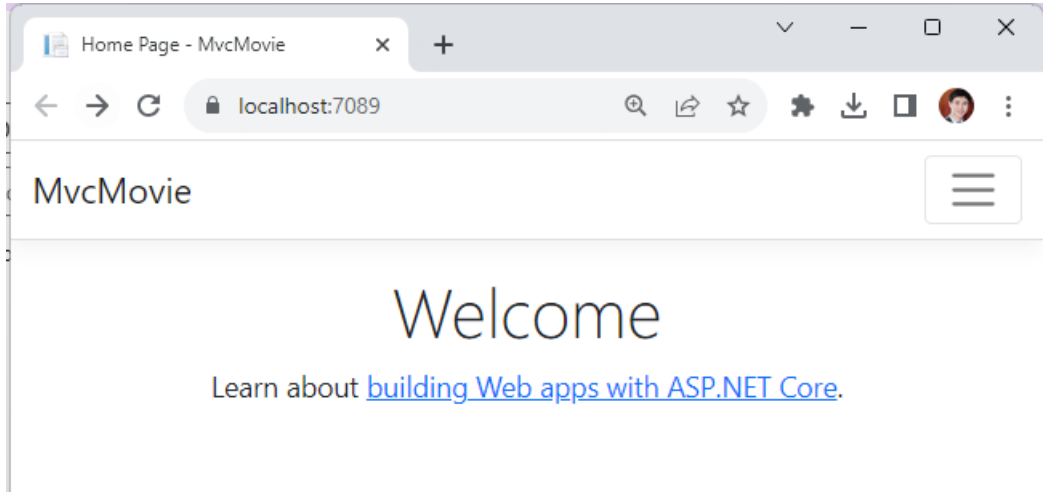


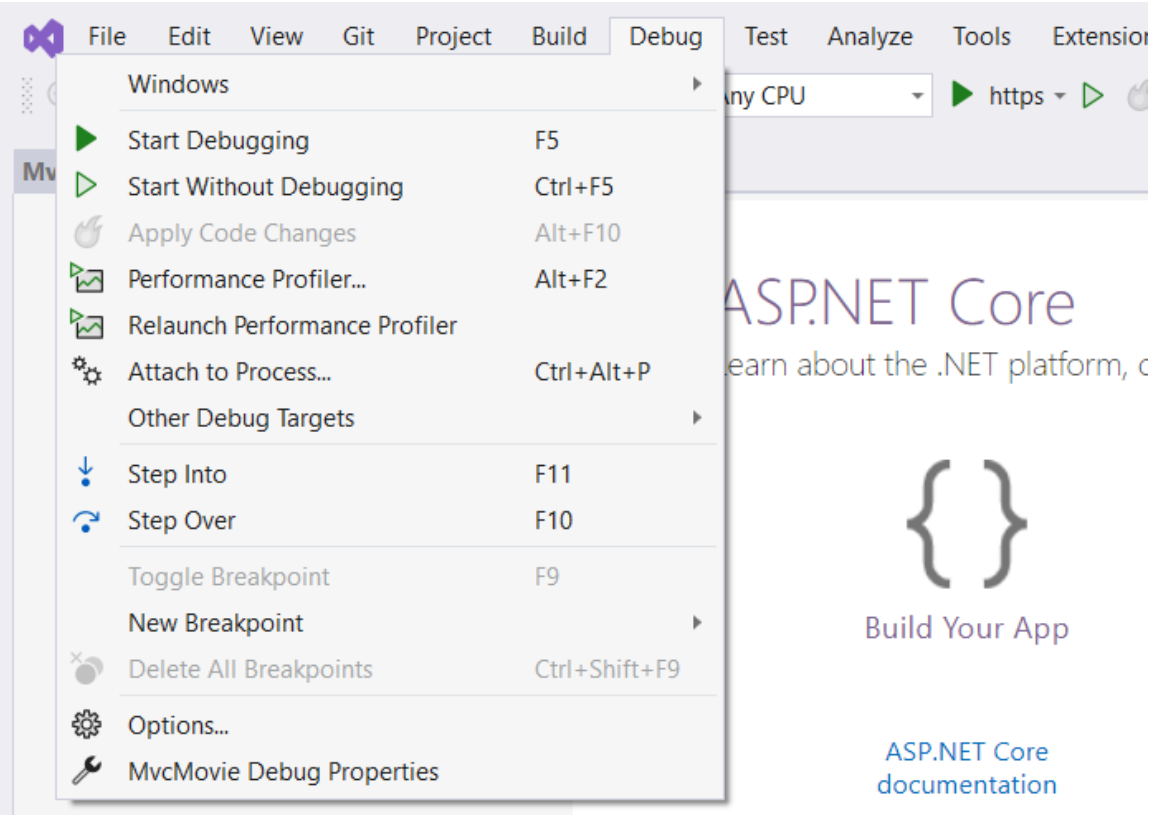


- In the Additional information dialog:
- Select .NET 7.0.
- Verify that Do not use top-level statements is unchecked.
- Select Create.

CREATE A NEW PROJECT

- Select Ctrl+F5 to run the app without the debugger.
- Visual Studio displays the following dialog when a project is not yet configured to use SSL:



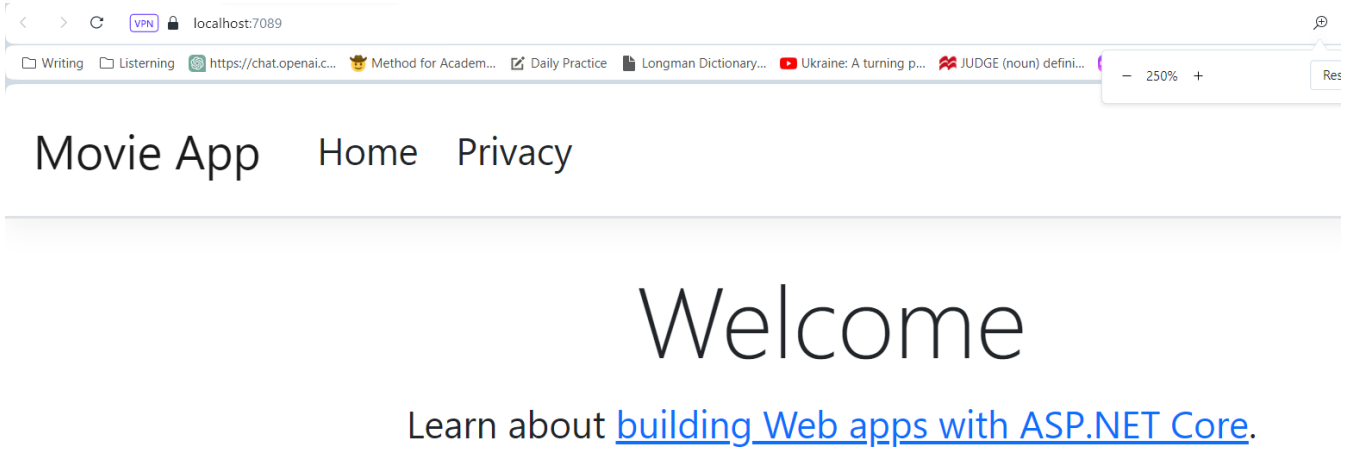


CREATE A NEW PROJECT

- Default
  - Home
  - Action: Index

```
3 references
public class HomeController : Controller
{
    private readonly ILogger<HomeController> _logger;

    0 references
    public HomeController(ILogger<HomeController> logger)
    {
        _logger = logger;
    }

    0 references
    public IActionResult Index()
    {
        return View();
    }
```

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
app.Run();
```



THE MODEL-VIEW-CONTROLLER (MVC) ARCHITECTURAL PATTERN

- The Model-View-Controller (MVC) architectural pattern separates an app into three main components:
- Model, View, and Controller.
- The MVC pattern helps you create apps that are more testable and easier to update than traditional monolithic apps.
- This pattern helps to achieve separation of concerns:
- The UI logic belongs in the view.
  - Input logic belongs in the controller.
- Business logic belongs in the model.
- This separation helps manage complexity when building an app, because it enables work on one aspect of the implementation at a time without impacting the code of another.
- For example, you can work on the view code without depending on the business logic code.

THE MODEL-VIEW-CONTROLLER (MVC) ARCHITECTURAL PATTERN

MVC-based apps contain:

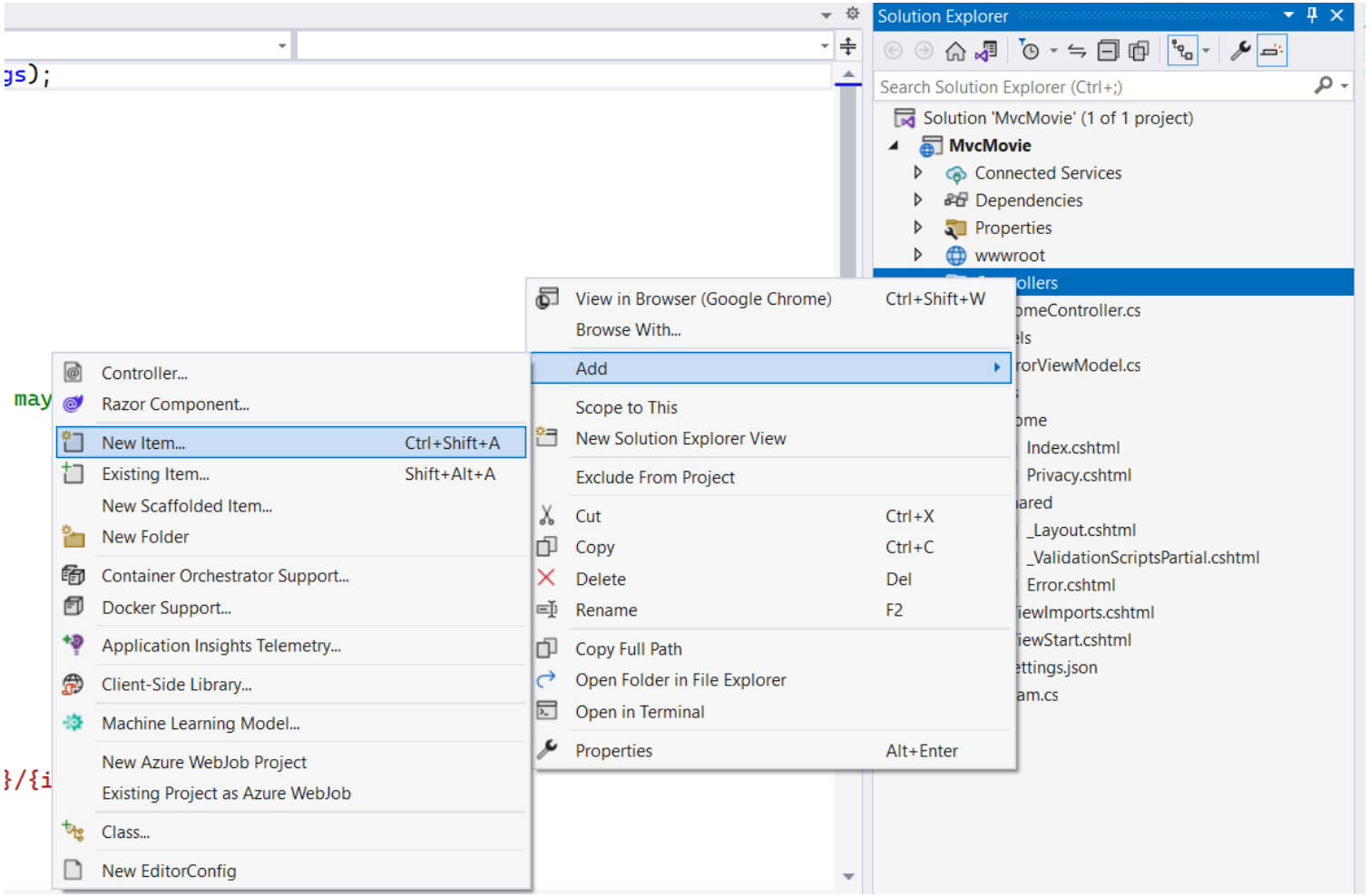
- Models:
  - Classes that represent the data of the app.
  - The model classes use validation logic to enforce business rules for that data.
  - Typically, model objects retrieve and store model state in a database.
  - In this tutorial, a Movie model retrieves movie data from a database, provides it to the view or updates it.
  - Updated data is written to a database.
- Views:
  - Views are the components that display the app's user interface (UI).
  - Generally, this UI displays the model data.
- Controllers: Classes that:
  - Handle browser requests.
  - Retrieve model data.
  - Call view templates that return a response.

THE MODEL-VIEW-CONTROLLER (MVC) ARCHITECTURAL PATTERN

- In an MVC app, the view only displays information. The controller handles and responds to user input and interaction.
- For example, the controller handles URL segments and query-string values, and passes these values to the model.
- The model might use these values to query the database.
- https://localhost:5001/Home/Privacy: specifies the Home controller and the Privacy action.
- https://localhost:5001/Movies/Edit/5: is a request to edit the movie with ID=5 using the Movies controller and the Edit action,

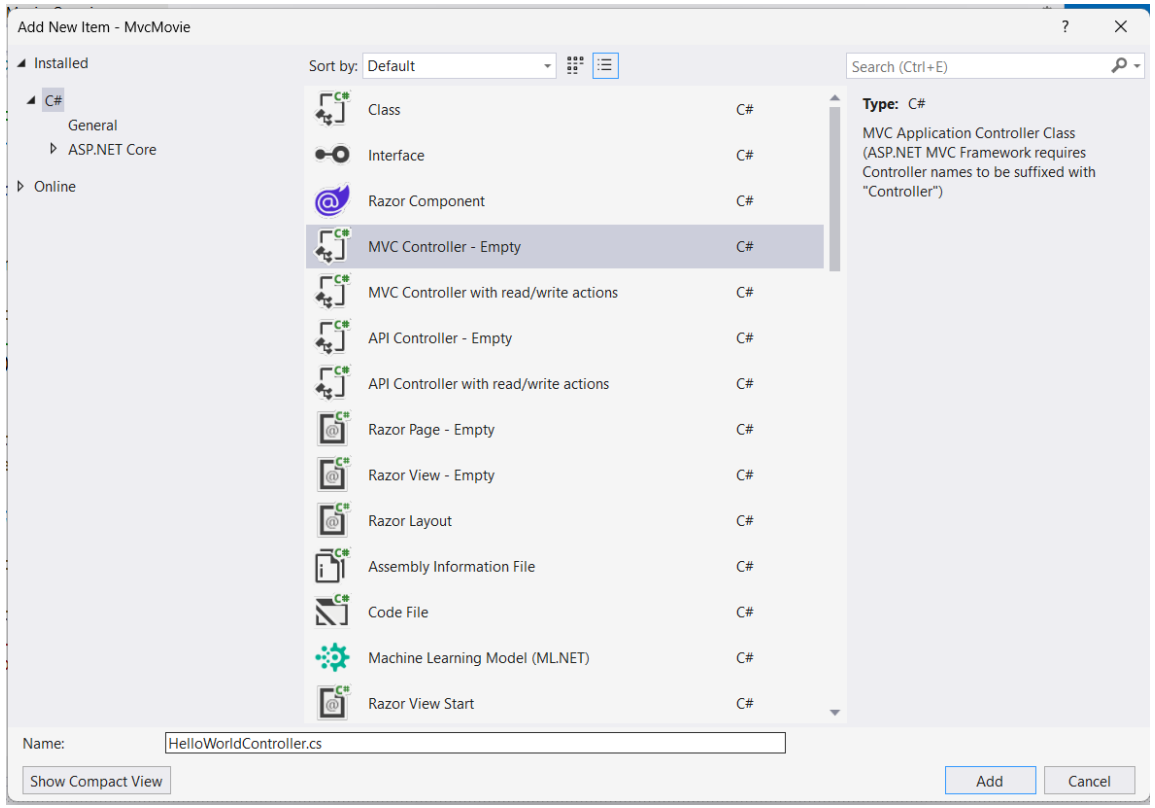
ADD CONTROLLER

- In Solution Explorer, right-click Controllers > Add > Controller.



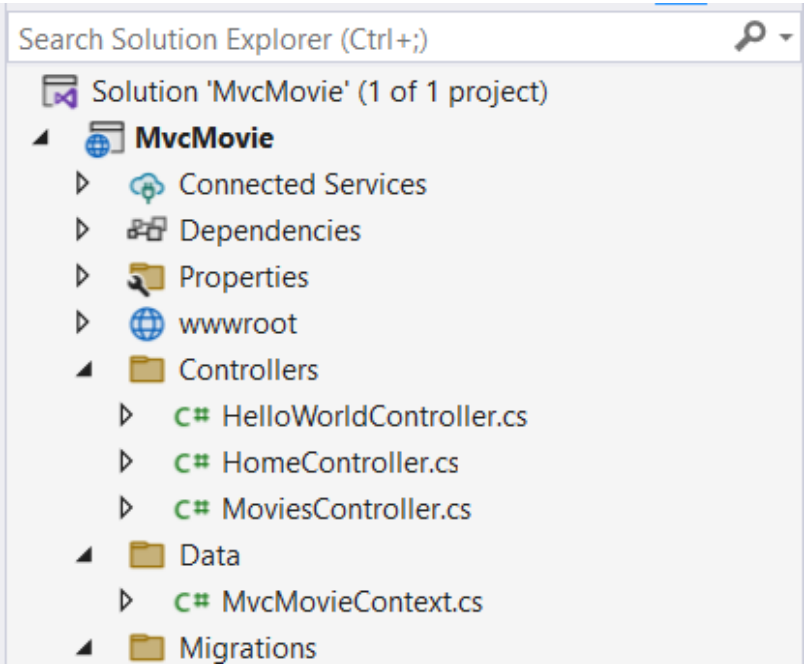
ADD CONTROLLER

- In the **Add New Scaffolded Item** dialog box, select **MVC Controller - Empty** > **Add**.
- In the Add New Item - MvcMovie dialog, enter HelloWorldController.cs and select Add.



ADD CONTROLLER

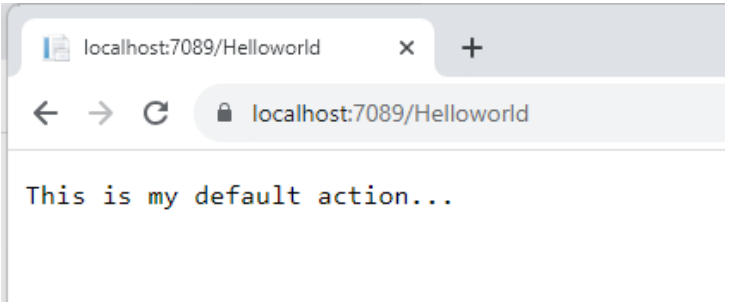
- Replace the contents of Controllers/HelloWorldController.cs with the following code:



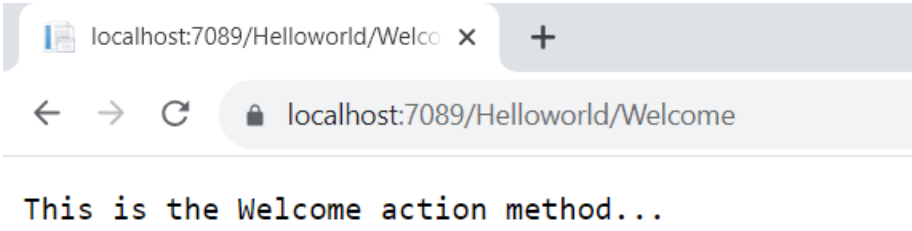

```
0 references
public class HelloWorldController : Controller
{
    /*
    public IActionResult Index()
    {
        return View();
    }
    */

    // GET: /HelloWorld/
    0 references
    public string Index()
    {
        return "This is my default action...";
    }
    //
    // GET: /HelloWorld/Welcome/
    0 references
    public string Welcome()
    {
        return "This is the Welcome action method...";
    }

}
```

ADD CONTROLLER

- Every public method in a controller is callable as an HTTP endpoint. In the sample above, both methods return a string. Note the comments preceding each method.
- An HTTP endpoint: Is a targetable URL in the web application, such as https://localhost:5001/HelloWorld.
- Combines: The protocol used: HTTPS.
- The network location of the web server, including the TCP port: localhost:5001.
- The target URI: HelloWorld.
- The first comment states this is an HTTP GET method that's invoked by appending /HelloWorld/ to the base URL.



The second comment specifies an HTTP GET method that's invoked by appending /HelloWorld/Welcome/ to the URL. Later on in the tutorial, the scaffolding engine is used to generate HTTP POST methods, which update data.
Run the app without the debugger.
Append /**HelloWorld** to the path in the address bar. The Index method returns a string.



ADD CONTROLLER

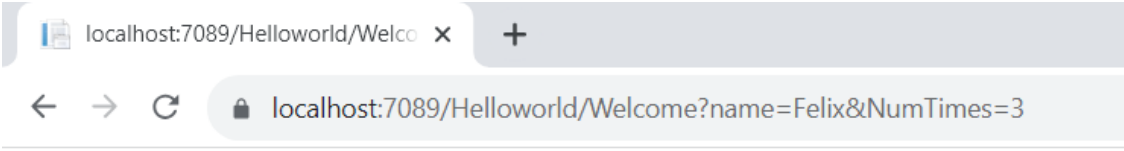
- MVC invokes controller classes, and the action methods within them, depending on the incoming URL.
- The default URL routing logic used by MVC, uses a format like this to determine what code to invoke:
- /[Controller]/[ActionName]/[Parameters]
- The routing format is set in the Program.cs file.

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
app.Run();
```

ADD CONTROLLER - MODIFICATION

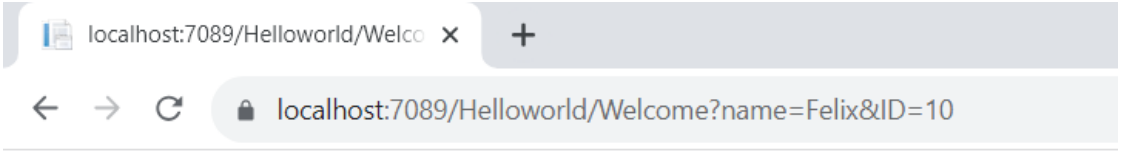
- Modify Welcome method and Run

```
//Modify the welcome method to

0 references
public string Welcome(string name, int numTimes = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, NumTimes is: {numTimes}");
}
```



Hello Felix, NumTimes is: 3

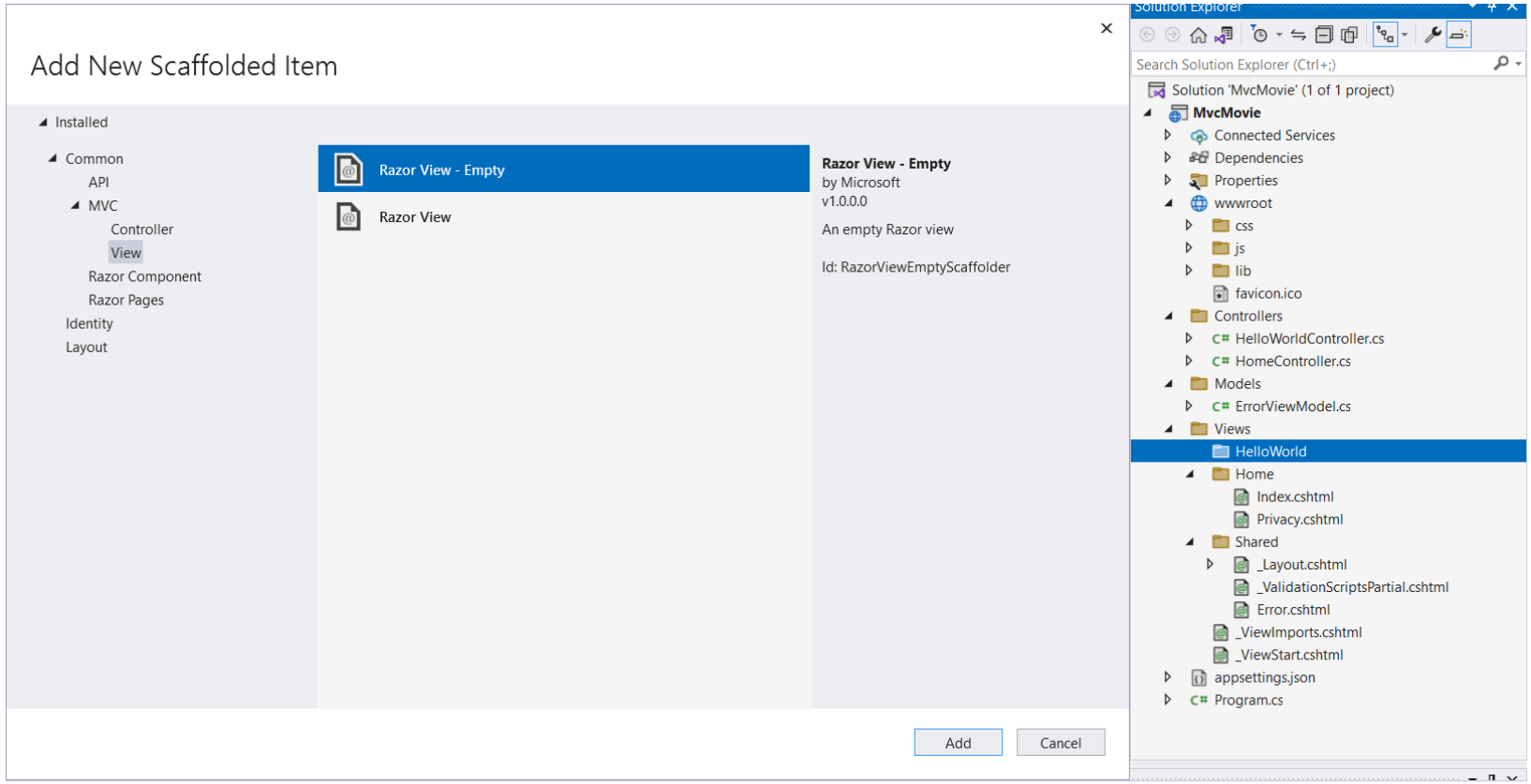
```
// GET: /HelloWorld/Welcome/
// Requires using System.Text.Encodings.Web;
//HelloWorld/Welcome? name = Felix & ID = 10

0 references
public string Welcome(string name, int ID = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, ID: {ID}");
}
```



Hello Felix, ID: 10

ADD A VIEW

- Right-click on 'Views' and add a folder named 'HelloWorld'.
- Add a new Razor View – Empty to the 'HelloWorld' folder in the 'Views'.



ADD A VIEW

- Replace the contents of the Views/HelloWorld/Index.cshtml Razor view file with the following
- Change the code in Index Method of

```
0 references
public IActionResult Index()
{
    return View();
}
```

localhost:7089/Helloworld

localhost:7089/Helloworld

```
This is my default action...
```

Program.cs | Index.cshtml | Privacy.cshtml | ErrorViewModel.cs | HomeController

```
@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>Hello from our View Template!</p>
```

Index - MvcMovie

localhost:7089/Helloworld

MvcMovie Home Privacy

Index

Hello from our View Template!

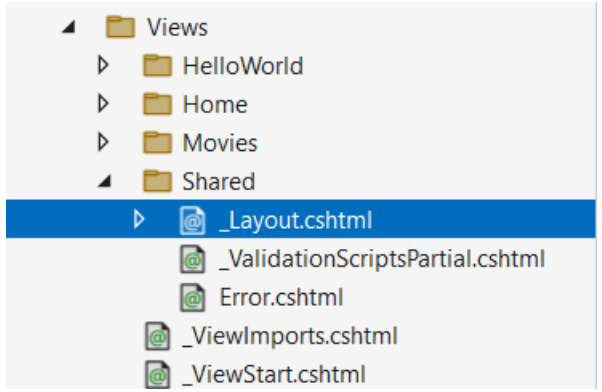
ADD A VIEW - CHANGE VIEWS AND LAYOUT PAGES

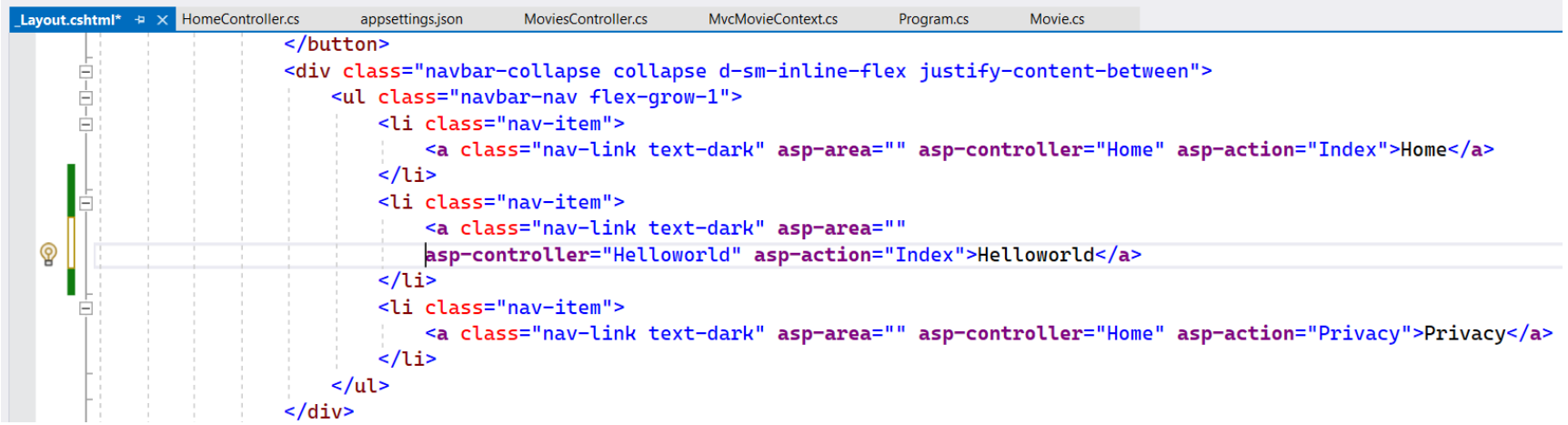
- Default layout
  - Select the menu links MvcMovie, Home, and Privacy. Each page shows the same menu layout. The menu layout is implemented in the Views/Shared/_Layout.cshtml file.
- Open the Views/Shared/_Layout.cshtml file.

```
_Layout.cshtml   Index.cshtml
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Movie App</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
    <link rel="stylesheet" href="~/css/site.css" asp-append-version="true" />
    <link rel="stylesheet" href="~/MvcMovie.styles.css" asp-append-version="true" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-shadow mb-3">
            <div class="container-fluid">
                <a class="navbar-brand" asp-area="" asp-controller="Movies" asp-action="Index">Movie App</a>
```

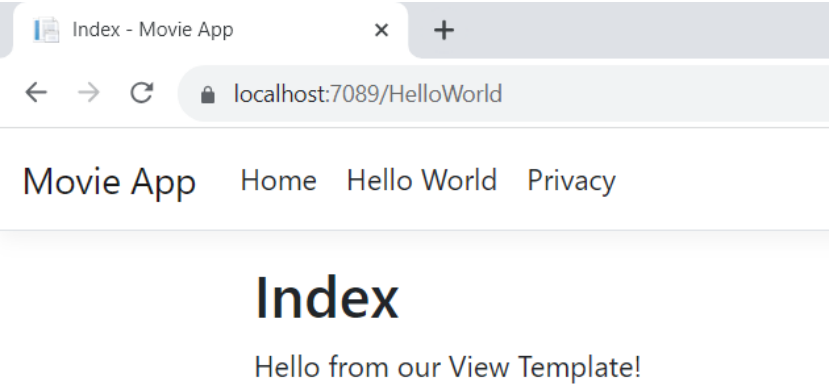
- Layout templates allow:
  - Specifying the HTML container layout of a site in one place.
  - Applying the HTML container layout across multiple pages in the site.
  - Find the @RenderBody() line. RenderBody is a placeholder where all the view-specific pages you create show up, wrapped in the layout page. For example, if you select the Privacy link, the Views/Home/Privacy.cshtml view is rendered inside the RenderBody method.
- Change share layout

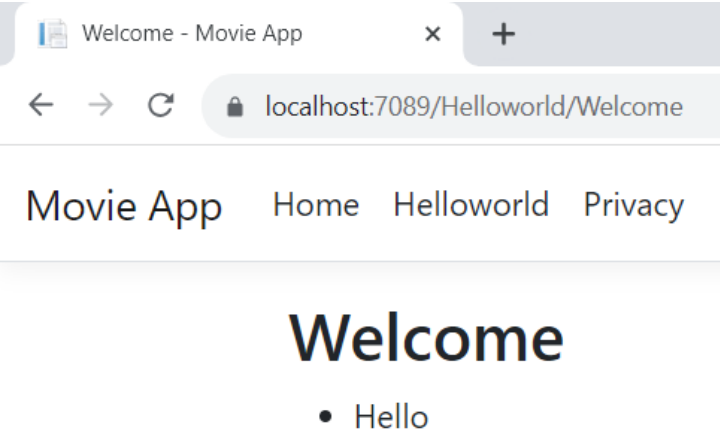
ADD A VIEW - CHANGE VIEWS AND LAYOUT PAGES





- Add Menu
  - Add hyperlink in li tag → Helloworld





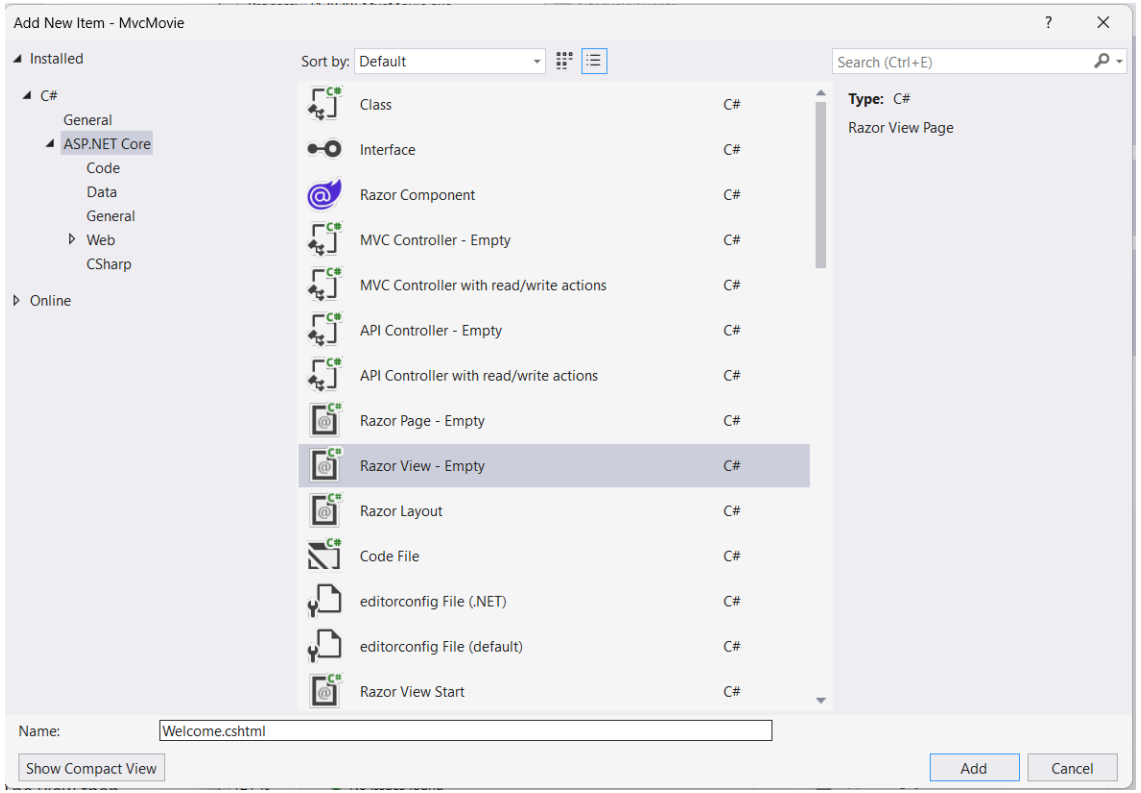
ADD A VIEW - PASSING DATA FROM THE CONTROLLER TO THE VIEW

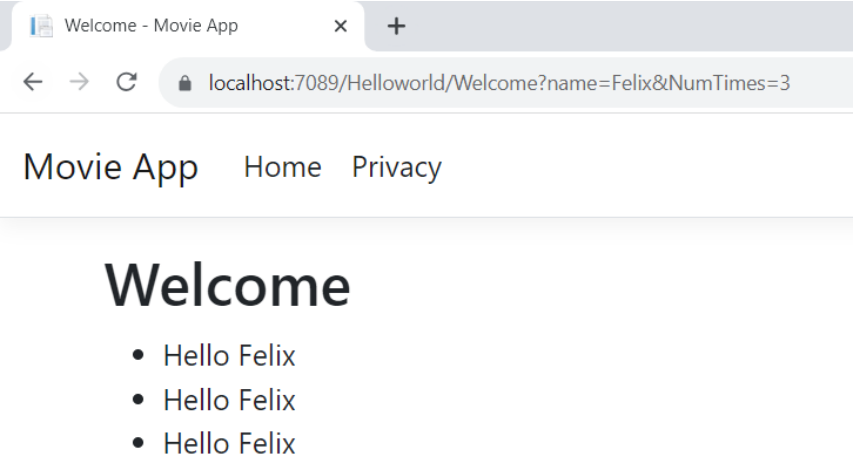
- HelloWorldController.cs
  - Replace the 'Welcome' method as follows:

```
/// <summary>
/// Bổ sung thêm một action trong Controller
/// The ViewData dictionary object contains data that will be passed to the view.
/// </summary>
/// <param name="name"></param>
/// <param name="numTimes"></param>
/// <returns></returns>
0 references
public IActionResult Welcome(string? name, int? numTimes = 1)
{
    if (name != null)
    {
        ViewData["Message"] = "Hello " + name;
    }
    if (numTimes != null)
    {
        ViewData["NumTimes"] = numTimes;
    }
    return View();
}
```

ADD A VIEW - PASSING DATA FROM THE CONTROLLER TO THE VIEW

- Add new View



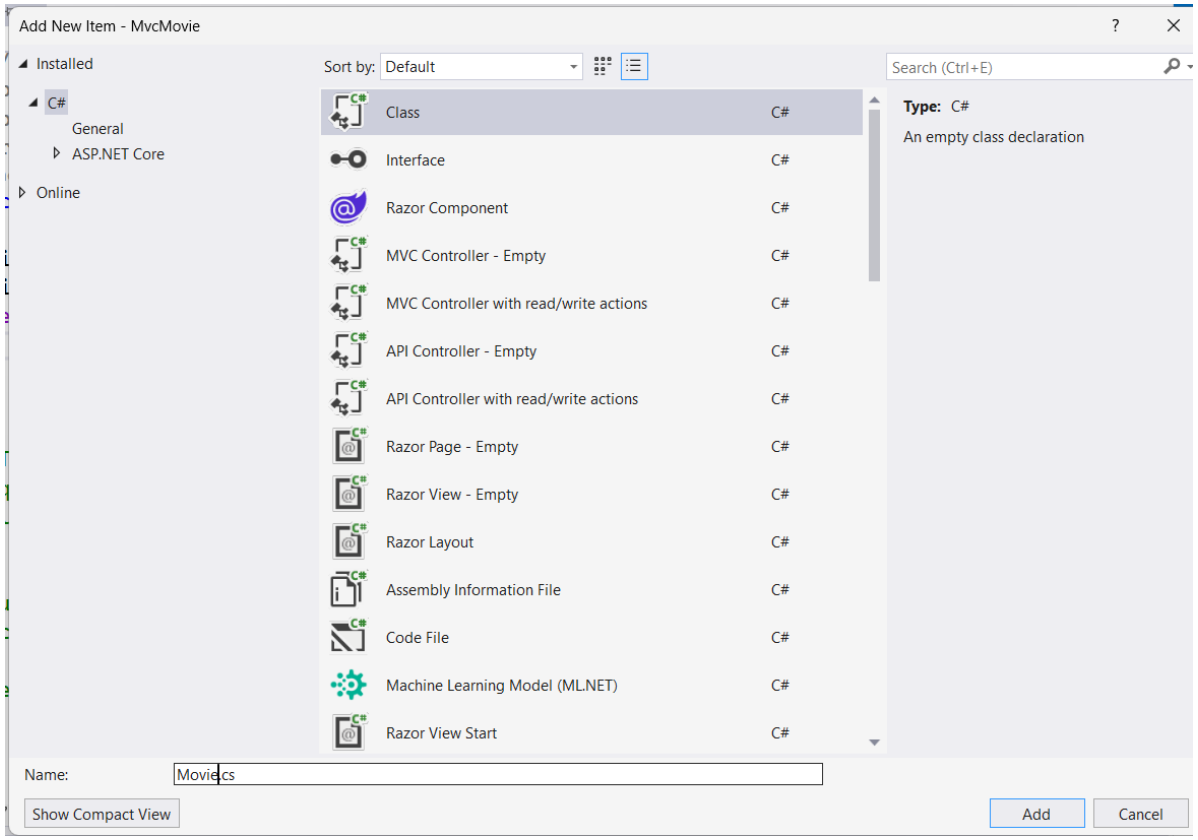


MODEL

- In the preceding sample, the ViewData dictionary was used to pass data from the controller to a view.
- a view model is used to pass data from a controller to a view.
- The view model approach to passing data is preferred over the ViewData dictionary approach.
- Classes are added for managing movies in a database. These classes are the "**M**odel" part of the **M**VC app.
- These model classes are used with Entity Framework Core (EF Core) to work with a database. EF Core is an object-relational mapping (ORM) framework that simplifies the data access code that you have to write.

ADD MODEL

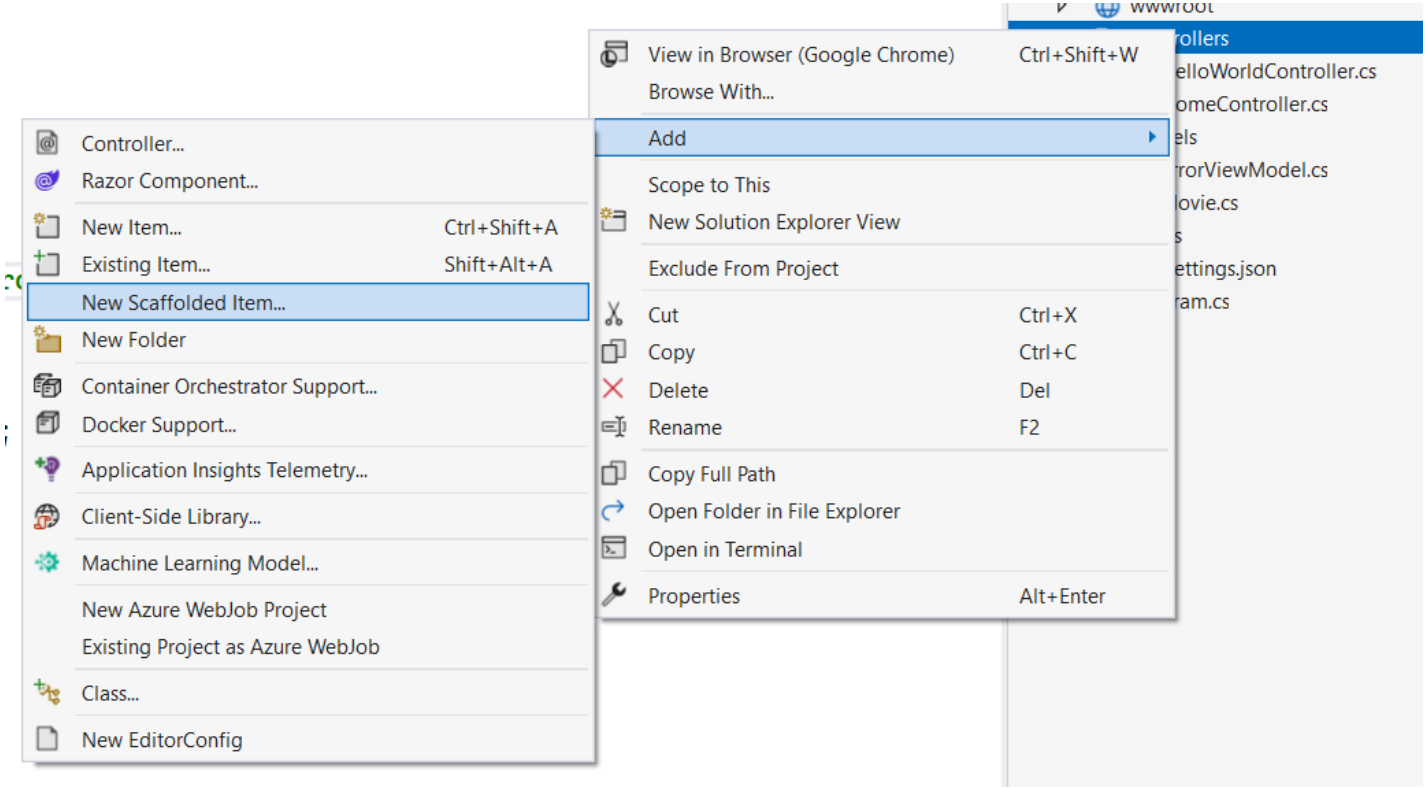
- Right-click the Models folder > Add > Class. Name the file Movie.cs.




```
using System.ComponentModel.DataAnnotations;

namespace MvcMovie.Models
{
    0 references
    public class Movie
    {
        0 references
        public int Id { get; set; }
        0 references
        public string? Title { get; set; }

        /// <summary>
        /// Chỉ cung cấp thông tin phần Date trong trường này
        /// </summary>
        [DataType(DataType.Date)]
        0 references
        public DateTime ReleaseDate { get; set; }
        0 references
        public string? Genre { get; set; }
        0 references
        public decimal Price { get; set; }
    }
}
```

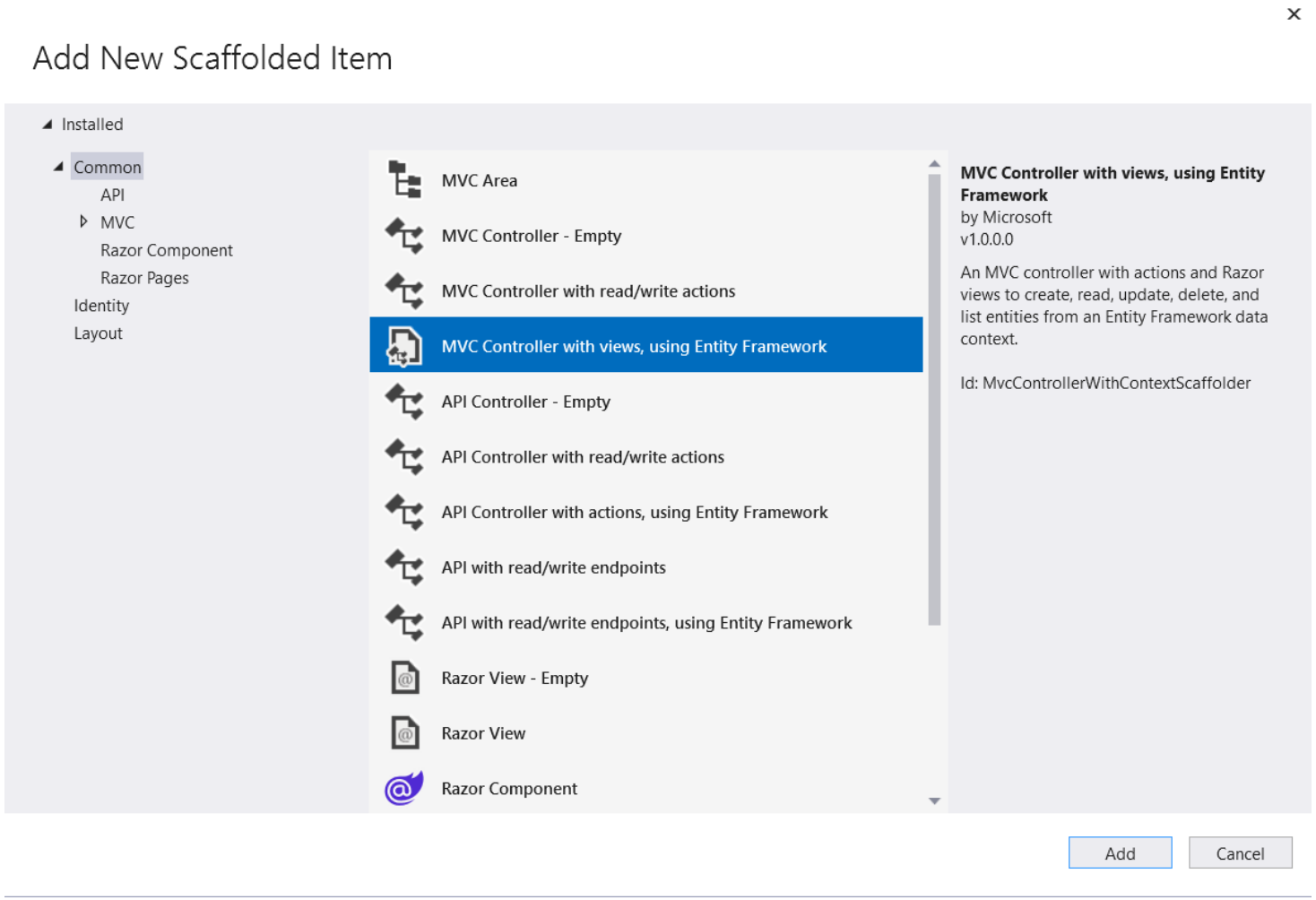
SCAFFOLD MOVIE PAGES

- Use the scaffolding tool to produce Create, Read, Update, and Delete (CRUD) pages for the movie model.
- In **Solution Explorer**, right-click the *Controllers* folder and select **Add > New Scaffolded Item**.



SCAFFOLD MOVIE PAGES

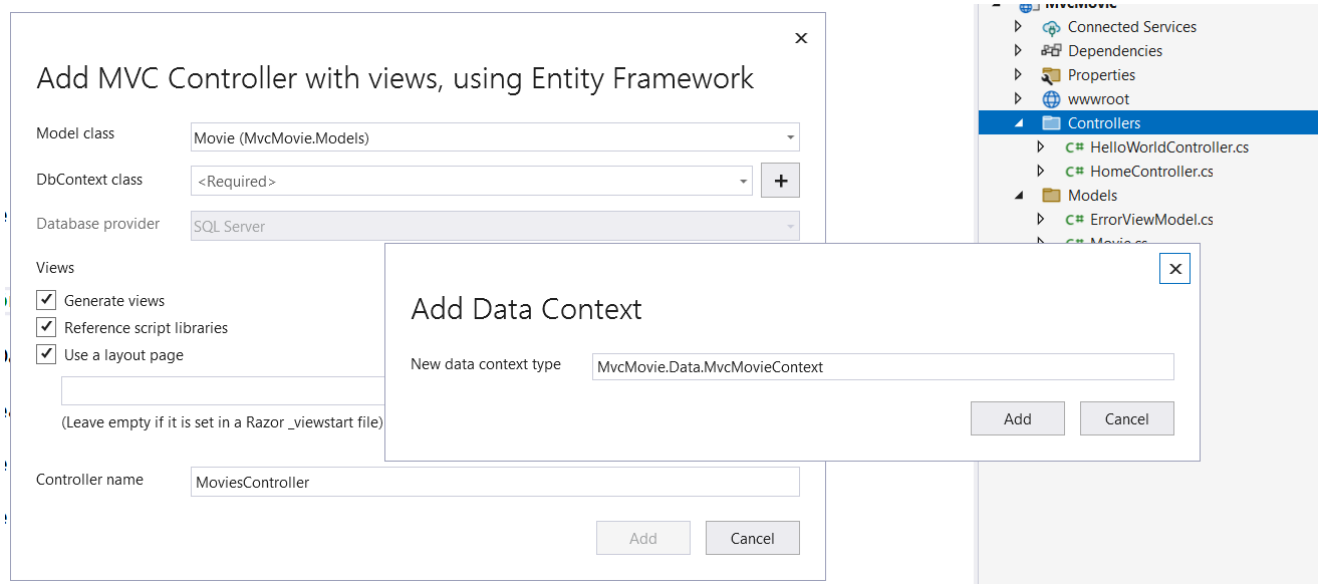
- In the **Add New Scaffolded Item** dialog:
- In the left pane, select **Installed** > **Common** > **MVC**.
- Select **MVC Controller with views, using Entity Framework**.
- Select **Add**.



SCAFFOLD MOVIE PAGES

- Complete the **Add MVC Controller with views, using Entity Framework** dialog:
- In the **Model class** drop down, select **Movie (MvcMovie.Models)**.
- In the **Data context class** row, select the **+** (plus) sign.
  - In the **Add Data Context** dialog, the class name *MvcMovie.Data.MvcMovieContext* is generated.
  - Select **Add**.

•In the **Database provider** drop down, select **SQL Server**.
•**Views** and **Controller name**: Keep the default.
•Select **Add**.



SCAFFOLD MOVIE PAGES

- Scaffolding adds the following **packages**:
  - Microsoft.EntityFrameworkCore.SqlServer
  - Microsoft.EntityFrameworkCore.Tools
  - Microsoft.VisualStudio.Web.CodeGeneration .Design
- Scaffolding creates the following:
  - A movies controller: Controllers/MoviesController.cs
  - Razor view files for Create, Delete, Details, Edit, and Index pages: Views/Movies/*.cshtml
  - A database context class: Data/MvcMovieContext.cs
- Scaffolding updates the following:
  - Inserts required package references in the MvcMovie.csproj project file.
  - Registers the database context in the Program.cs file.
  - Adds a database connection string to the appsettings.json file.
  - The automatic creation of these files and file updates is known as scaffolding.
- However, the scaffolded pages can't be used yet because the database doesn't exist.
  - Running the app and selecting the Movie App link results in a Cannot open database or no such table: Movie error message.
  - Build the app to verify that there are no errors.

SCAFFOLD MOVIE PAGES

appsettings.json | MoviesController.cs

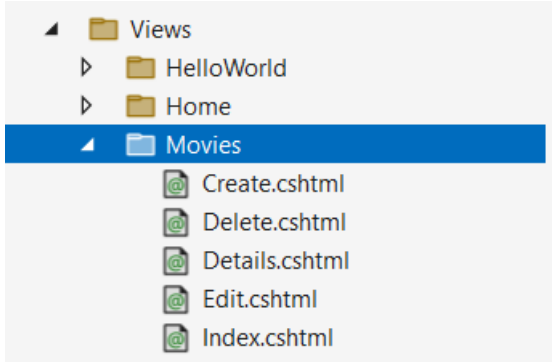
MvcMovie.Controllers.M | Edit(int id, Movie movie)

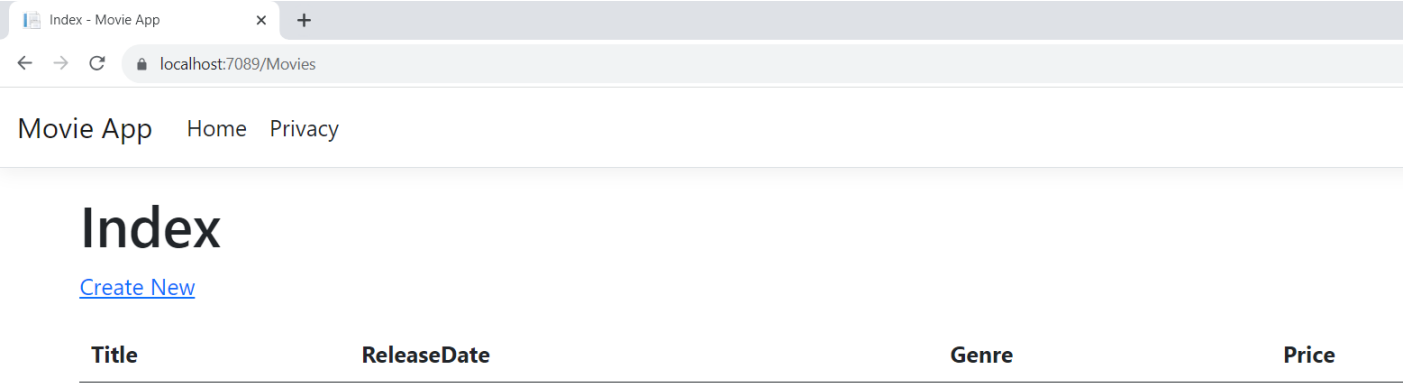
```
    }

    // GET: Movies/Details/5
    0 references
    public async Task<IActionResult> Details(int?
    {
        if (id == null || _context.Movie == null)
        {
            return NotFound();
        }

        var movie = await _context.Movie
            .FirstOrDefaultAsync(m => m.Id == id)
        if (movie == null)
        {
            return NotFound();
        }

        return View(movie);
```

Solution Explorer

Search Solution Explorer (Ctrl+;)

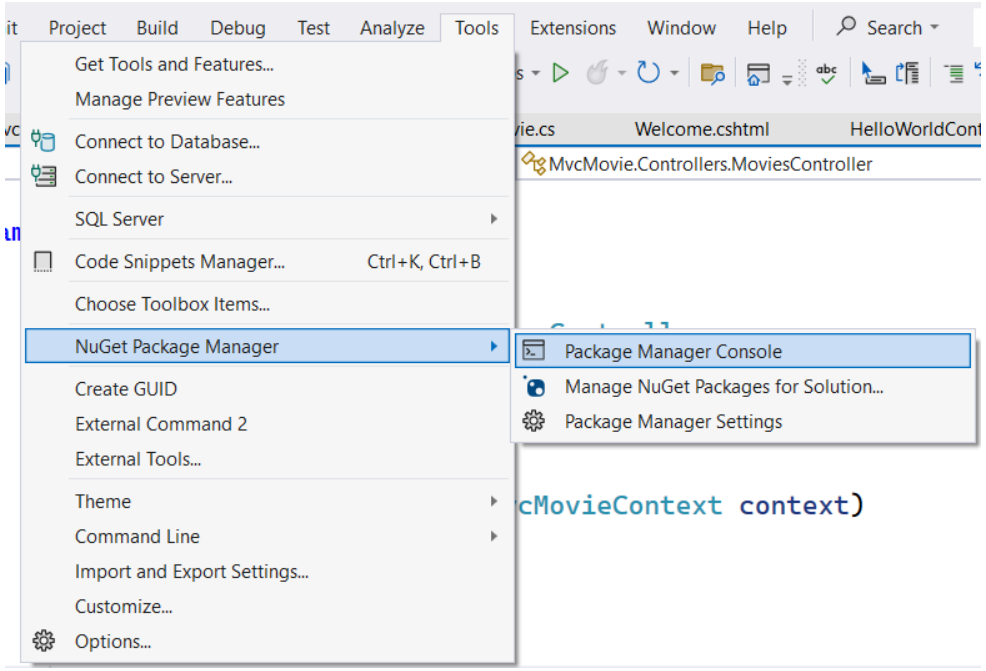
- Properties
- wwwroot
- Controllers
  - HelloWorldController.cs
  - HomeController.cs
  - MoviesController.cs
- Data
  - MvcMovieContext.cs
- Migrations
  - 20230919085421_InitialCreate.cs
  - MvcMovieContextModelSnapshot.cs
- Models
  - ErrorViewModel.cs
  - Movie.cs

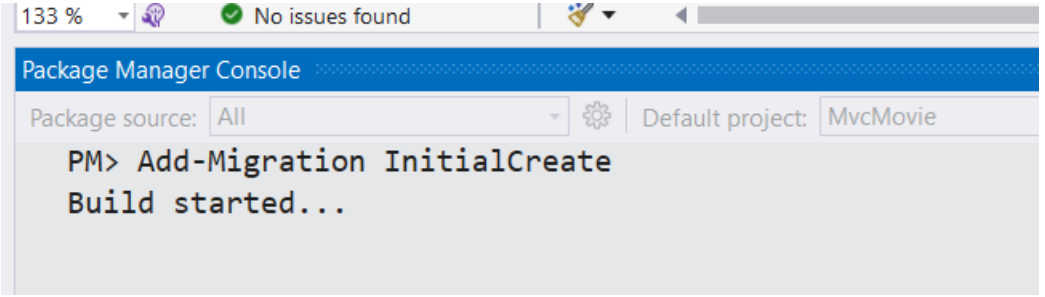




INITIAL MIGRATIONS

- Use the EF Core Migrations feature to create the database. *Migrations* is a set of tools that create and update a database to match the data model.
- From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console** .
- In the Package Manager Console (PMC), enter the following commands:
  - Add-Migration InitialCreate
  - Update-Database





INITIAL MIGRATIONS

- Add-Migration InitialCreate:
  - Generates a Migrations/{timestamp}_InitialCreate.cs migration file.
  - The InitialCreate argument is the migration name. Any name can be used, but by convention, a name is selected that describes the migration.
  - Because this is the first migration, the generated class contains code to create the database schema.
- The database schema is based on the model specified in the MvcMovieContext class.

Update-Database:

Updates the database to the latest migration, which the previous command created.
This command runs the Up method in the Migrations/{time-stamp}_InitialCreate.cs file, which creates the database.

```
public partial class InitialCreate : Migration
{
    /// <inheritdoc />
    0 references
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Movie",
            columns: table => new
            {
                Id = table.Column<int>(type: "int", nullable: false)
                    .Annotation("SqlServer:Identity", "1, 1"),
                Title = table.Column<string>(type: "nvarchar(max)", nullable: true),
                ReleaseDate = table.Column<DateTime>(type: "datetime2", nullable: false),
                Genre = table.Column<string>(type: "nvarchar(max)", nullable: true),
                Price = table.Column<decimal>(type: "decimal(18,2)", nullable: false)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Movie", x => x.Id);
            });
    }

    /// <inheritdoc />
```

ADD A 'MOVIES' MENU

```
</li>
<li class="nav-item">
    <a class="nav-link text-dark" asp-area=""
       asp-controller="Movies" asp-action="Index">Movies</a>
</li>
```

localhost:7089/Movies

Movie App Home Hello World Movies Privacy

Index

Create New

Title	ReleaseDate	Genre	Price	
Thám tử Conan	9/20/2023	Viễn tưởng	30000.00	Edit | Details | Delete
Phim Doremon	9/18/2023	Trẻ em	30000.00	Edit | Details | Delete

CHECK THE CONFIGURATION

appsettings.json

appsettings.json | MoviesController.cs | MvcMovieContext.cs | Program.cs | Movie.cs

Schema: https://json.schemastore.org/appsettings.json

```
1  {
2    "Logging": {
3      "LogLevel": {
4        "Default": "Information",
5        "Microsoft.AspNetCore": "Warning"
6      }
7    },
8    "AllowedHosts": "*",
9    "ConnectionStrings": {
10     "MvcMovieContext": "Server=(localdb)\\mssqllocaldb;Database=MvcMovieContext-f5469
11   }
12 }
```

CHECK THE CONFIGURATION

The InitialCreate class

202309190854...tialCreate.cs | appsettings.json | MoviesController.cs | MvcMovieContext.cs | Program.cs | Movie.cs

MvcMovie | MvcMovie.Migrations.InitialCreate | Up(MigrationBuilder migrationBuil

```
7   {
8       /// <inheritdoc />
        1 reference
9       public partial class InitialCreate : Migration
10      {
11          /// <inheritdoc />
            0 references
12          protected override void Up(MigrationBuilder migrationBuilder)
13          {
14              migrationBuilder.CreateTable(
15                  name: "Movie",
16                  columns: table => new
17                  {
18                      Id = table.Column<int>(type: "int", nullable: false)
19                          .Annotation("SqlServer:Identity", "1, 1"),
20                      Title = table.Column<string>(type: "nvarchar(max)", nullable: true),
21                      ReleaseDate = table.Column<DateTime>(type: "datetime2", nullable: false),
22                      Genre = table.Column<string>(type: "nvarchar(max)", nullable: true),
23                      Price = table.Column<decimal>(type: "decimal(18,2)", nullable: false)
24                  },
25                  constraints: table =>
26                  {
27                      table.PrimaryKey("PK_Movie", x => x.Id);
28                  });
```

CONNECT TO DATABASE