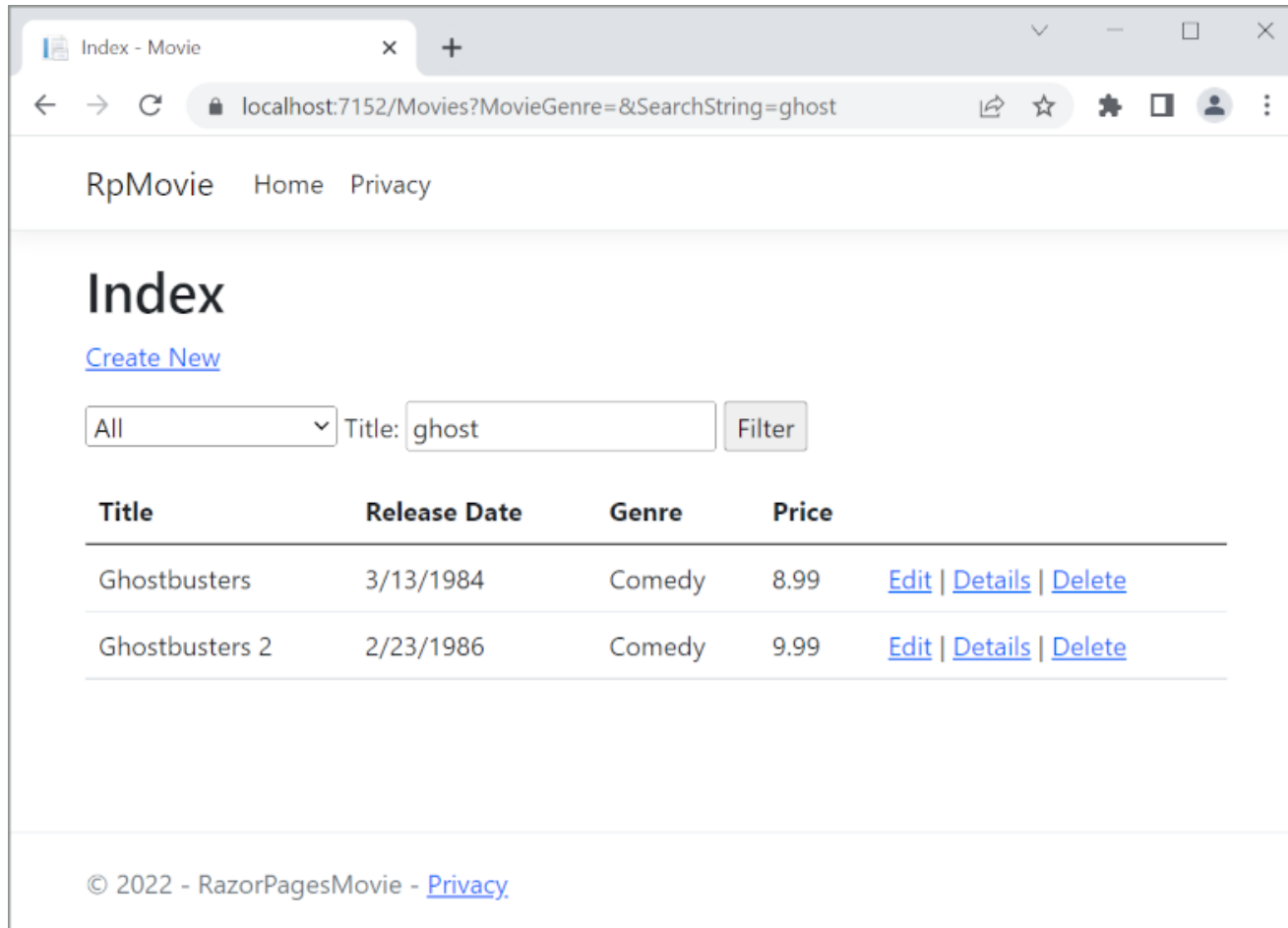


# Развој на серверски WEB апликации

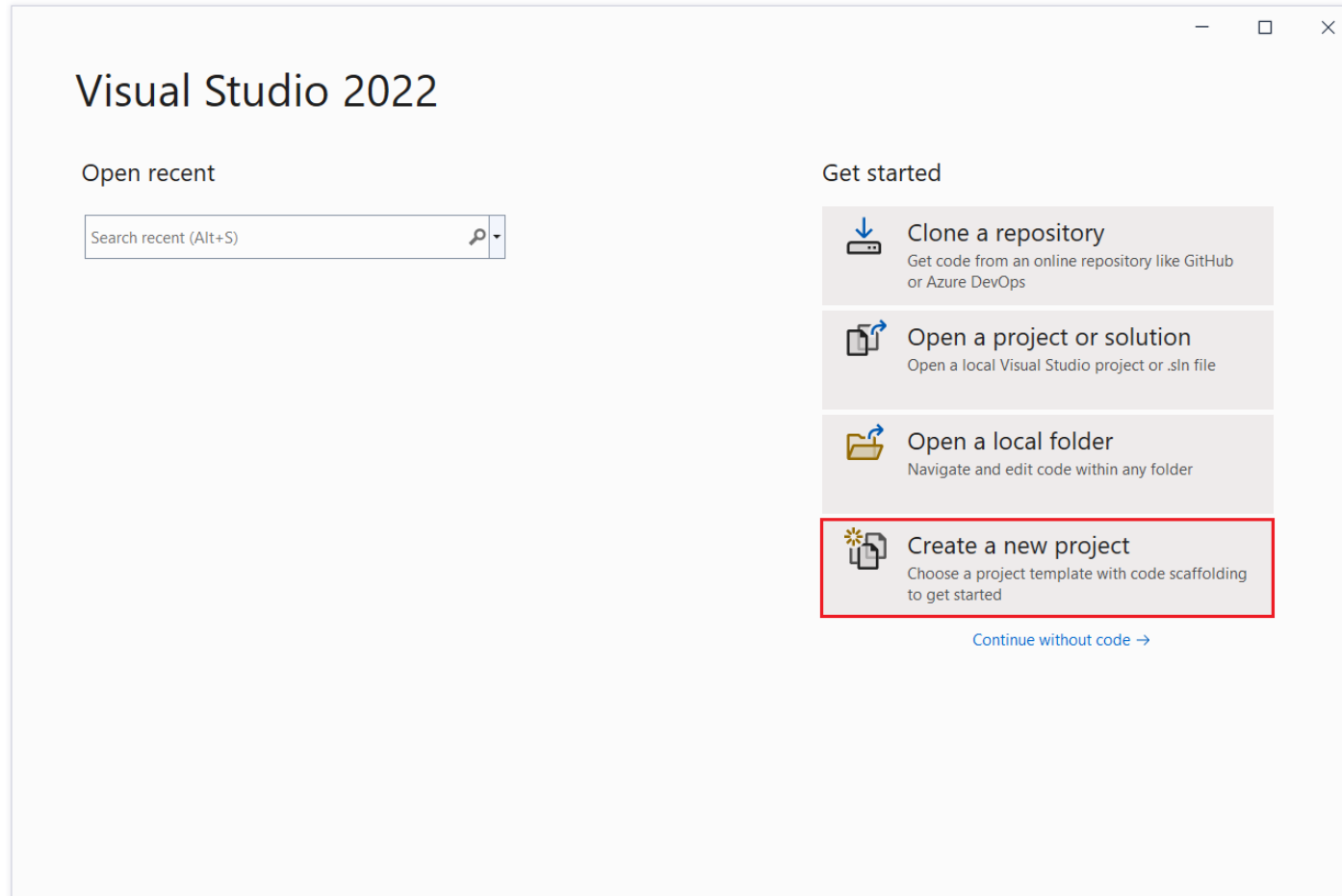
Проф. д-р Перо Латкоски

# Пример на Razor Pages web app with ASP.NET Core

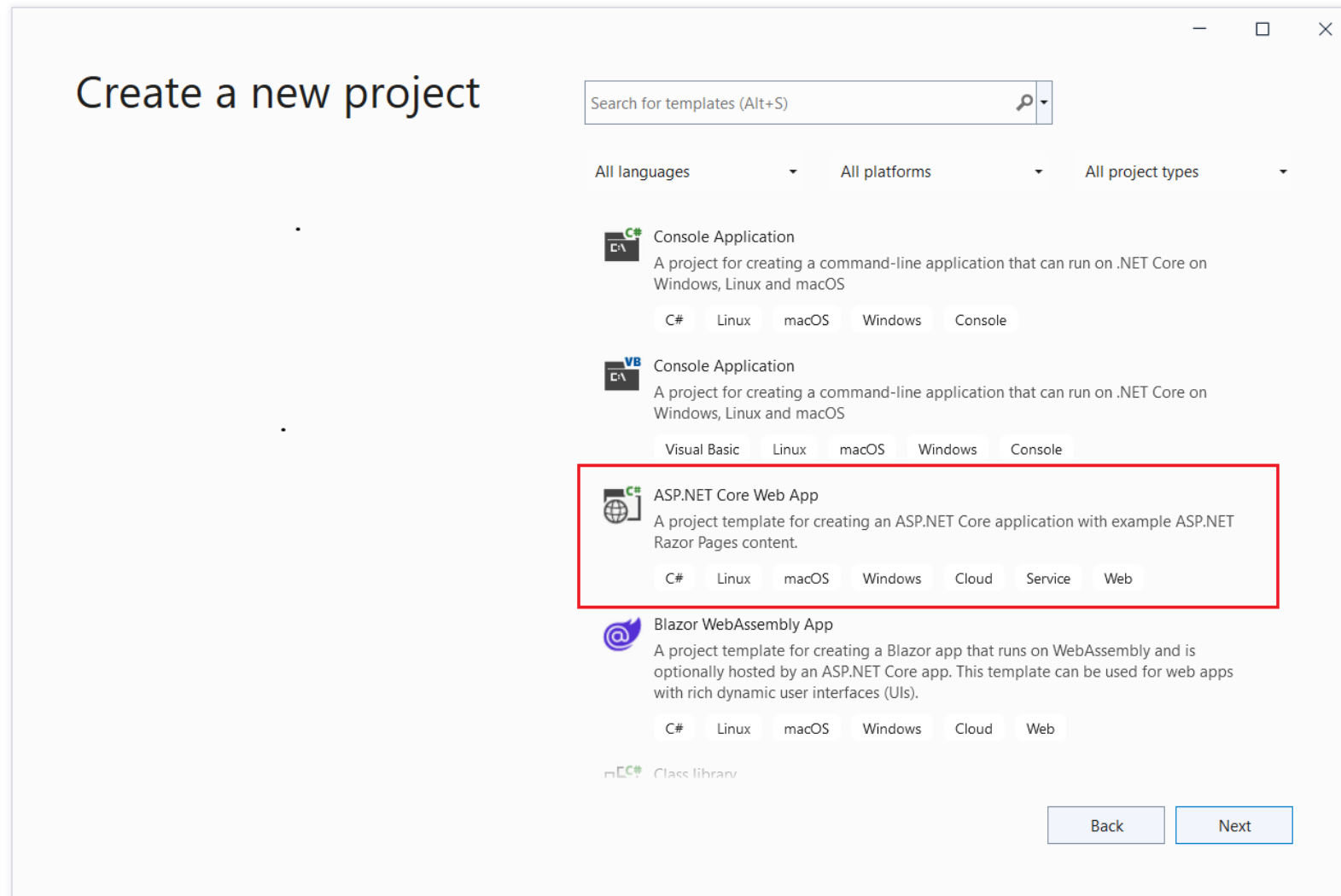


[Tutorial: Create a Razor Pages web app with ASP.NET Core | Microsoft Learn](#)

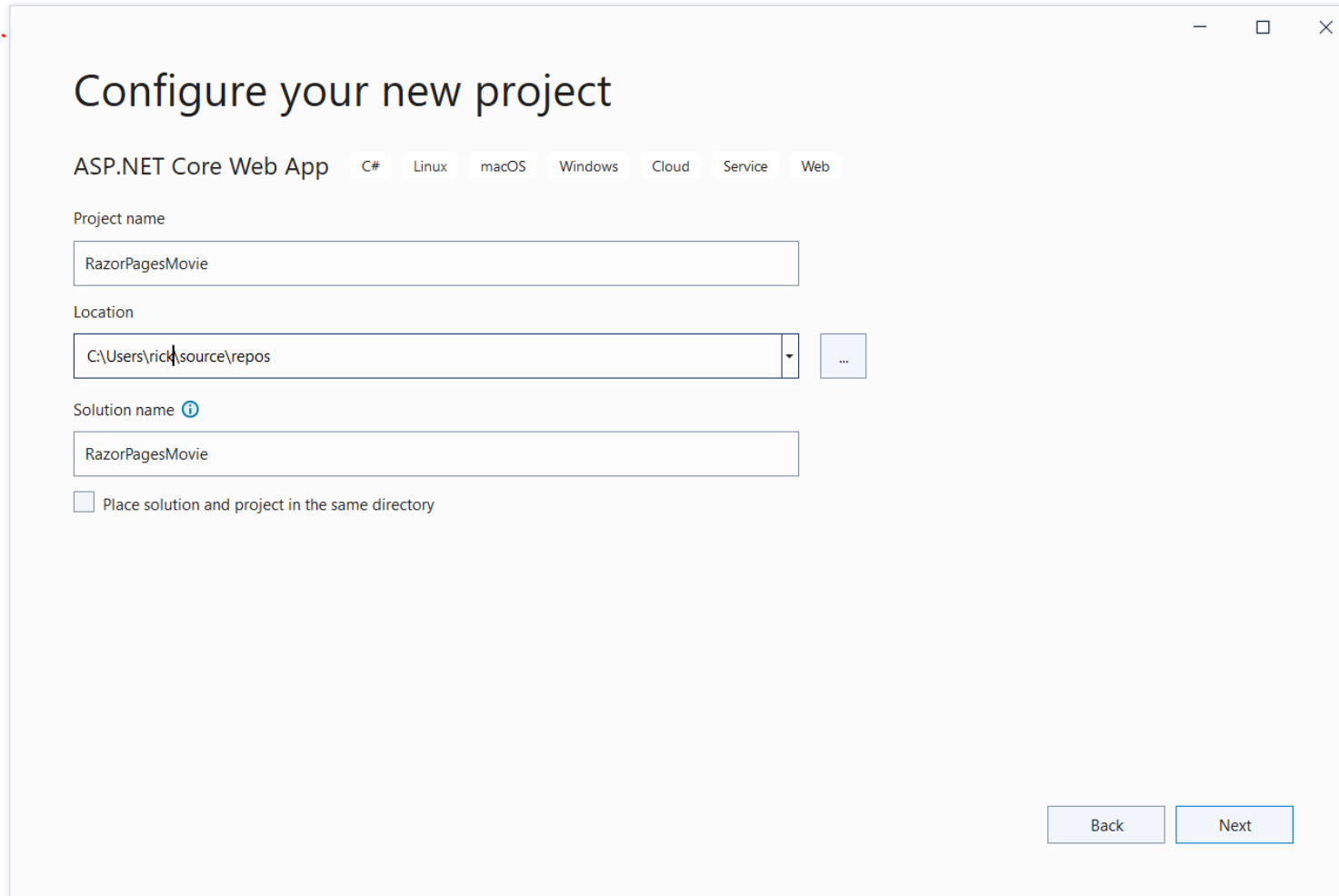
# Start Visual Studio 2022 and select **Create a new project**.



In the **Create a new project** dialog, select **ASP.NET Core Web App**, and then select **Next**.



In the **Configure your new project** dialog, enter RazorPagesMovie for **Project name**. It's important to name the project *RazorPagesMovie*, including matching the capitalization, so the namespaces will match when you copy and paste example code.



The screenshot shows the 'Configure your new project' dialog box. At the top, the title 'Configure your new project' is displayed. Below the title, there are tabs for different project types: 'ASP.NET Core Web App' (selected), 'C#', 'Linux', 'macOS', 'Windows', 'Cloud', 'Service', and 'Web'. The 'Project name' field contains 'RazorPagesMovie'. The 'Location' field contains 'C:\Users\rick\source\repos' and has a dropdown arrow and a browse button ('...'). The 'Solution name' field, which has a help icon, also contains 'RazorPagesMovie'. At the bottom left, there is a checkbox labeled 'Place solution and project in the same directory' which is currently unchecked. At the bottom right, there are 'Back' and 'Next' buttons.

Configure your new project

ASP.NET Core Web App C# Linux macOS Windows Cloud Service Web

Project name

RazorPagesMovie

Location

C:\Users\rick\source\repos

Solution name ⓘ

RazorPagesMovie

☐ Place solution and project in the same directory

Back Next

Select **Next**.

In the **Additional information** dialog, select **.NET 8.0 (Long-term support)** and then select **Create**.

## Additional information

### ASP.NET Core Web App (Razor Pages)

C#

Linux


macOS

Windows


Cloud

Service


Web


Framework 

.NET 8.0 (Long Term Support)

Authentication type 


None

☒ Configure for HTTPS 

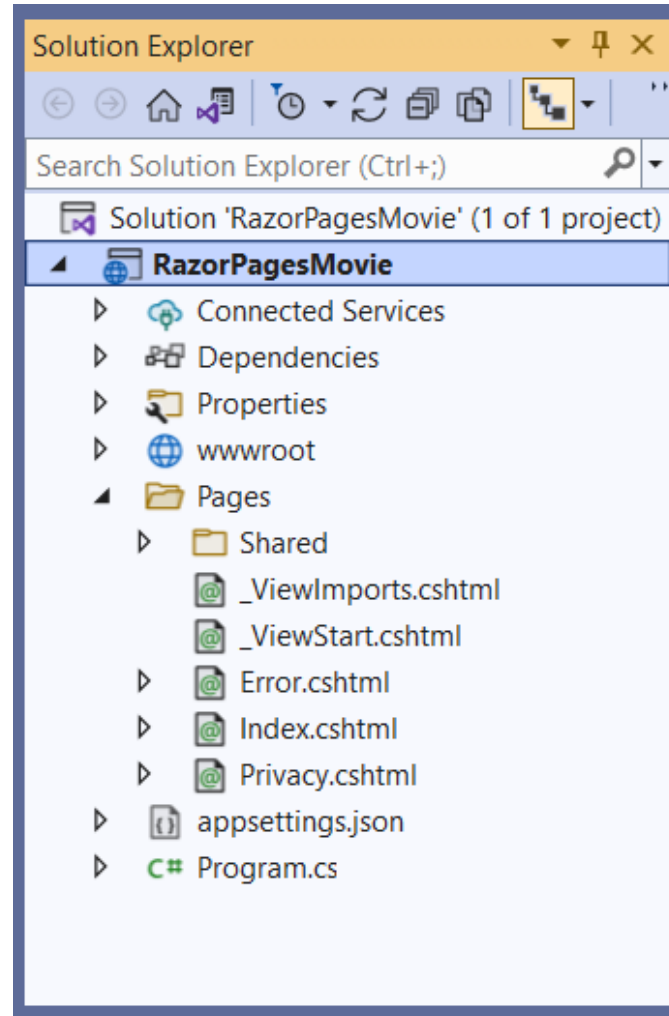
☐ Enable Docker 

Docker OS 

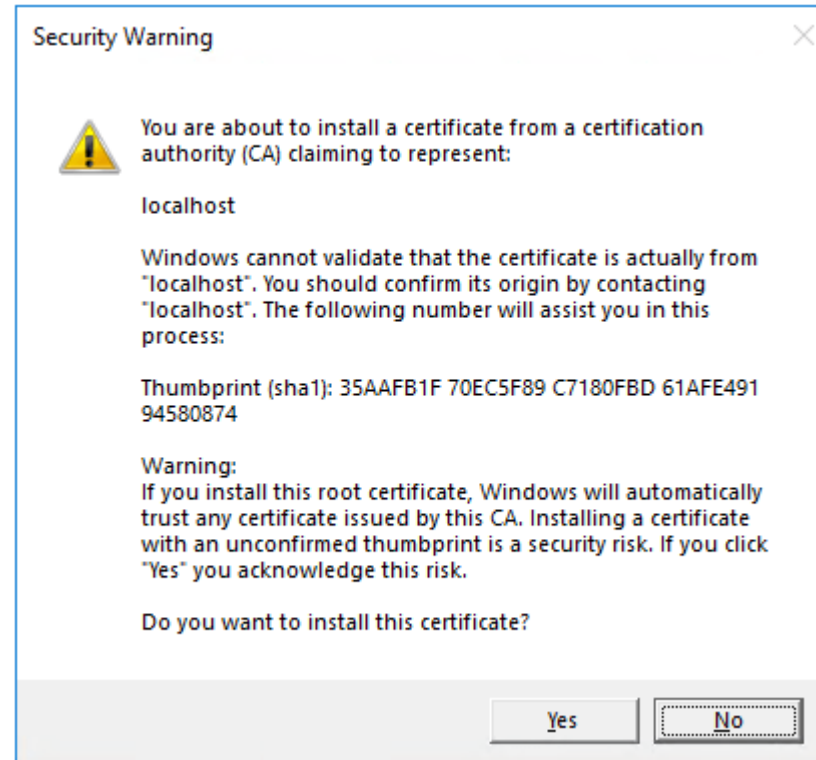
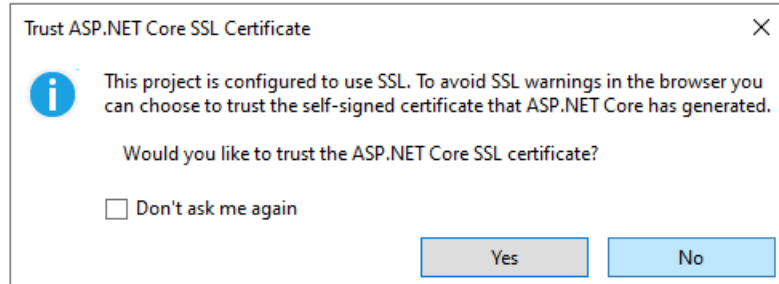
Linux

☒ Do not use top-level statements 

The following starter project is created:



Select **RazorPagesMovie** in **Solution Explorer**, and then press Ctrl+F5 to run without the debugger.





# Program.cs

- The following code enables various Middleware:
- `app.UseHttpsRedirection();` : Redirects HTTP requests to HTTPS.
- `app.UseStaticFiles();` : Enables static files, such as HTML, CSS, images, and JavaScript to be served.
- `app.UseRouting();` : Adds route matching to the middleware pipeline.  
`app.MapRazorPages();` : Configures endpoint routing for Razor Pages.
- `app.UseAuthorization();` : Authorizes a user to access secure resources. This app doesn't use authorization, therefore this line could be removed.
- `app.Run();` : Runs the app.

## Part 2, add a model to a Razor Pages app in ASP.NET Core

- In this tutorial, classes are added for [managing movies in a database](#).
- The app's model classes use [Entity Framework Core \(EF Core\)](#) to work with the [database](#).
  - EF Core is an object-relational mapper (O/RM) that simplifies data access. You write the model classes first, and EF Core creates the database.
- The model classes are known as POCO classes (from "**P**lain-**O**ld **CLR** **O**bjects") because they don't have a dependency on EF Core. They define the properties of the data that are stored in the database.
  - Each POCO class maps to a table in the database, and each property of the class represents a column in the table.
  - The ORM framework is responsible for generating SQL queries to map the objects to the database and vice versa.

# Add a data model

1. In **Solution Explorer**, right-click the *RazorPagesMovie* project > **Add** > **New Folder**. Name the folder **Models**.
2. Right-click the *Models* folder. Select **Add** > **Class**. Name the class **Movie**.
3. Add the following properties to the *Movie* class:

```
using System.ComponentModel.DataAnnotations;
```

```
namespace RazorPagesMovie.Models
```

```
{
```

```
    public class Movie
```

```
    {
```

```
        public int ID { get; set; }
```

```
        public string Title { get; set; } = string.Empty;
```

```
        [DataType(DataType.Date)]
```

```
        public DateTime ReleaseDate { get; set; }
```

```
        public string Genre { get; set; } = string.Empty;
```

```
        public decimal Price { get; set; }
```

```
    }
```

```
}
```

The Movie class contains:

The ID field is required by the database for the primary key.

A [DataType] attribute that specifies the type of data in the ReleaseDate property. With this attribute:

- The user isn't required to enter time information in the date field.
- Only the date is displayed, not time information.

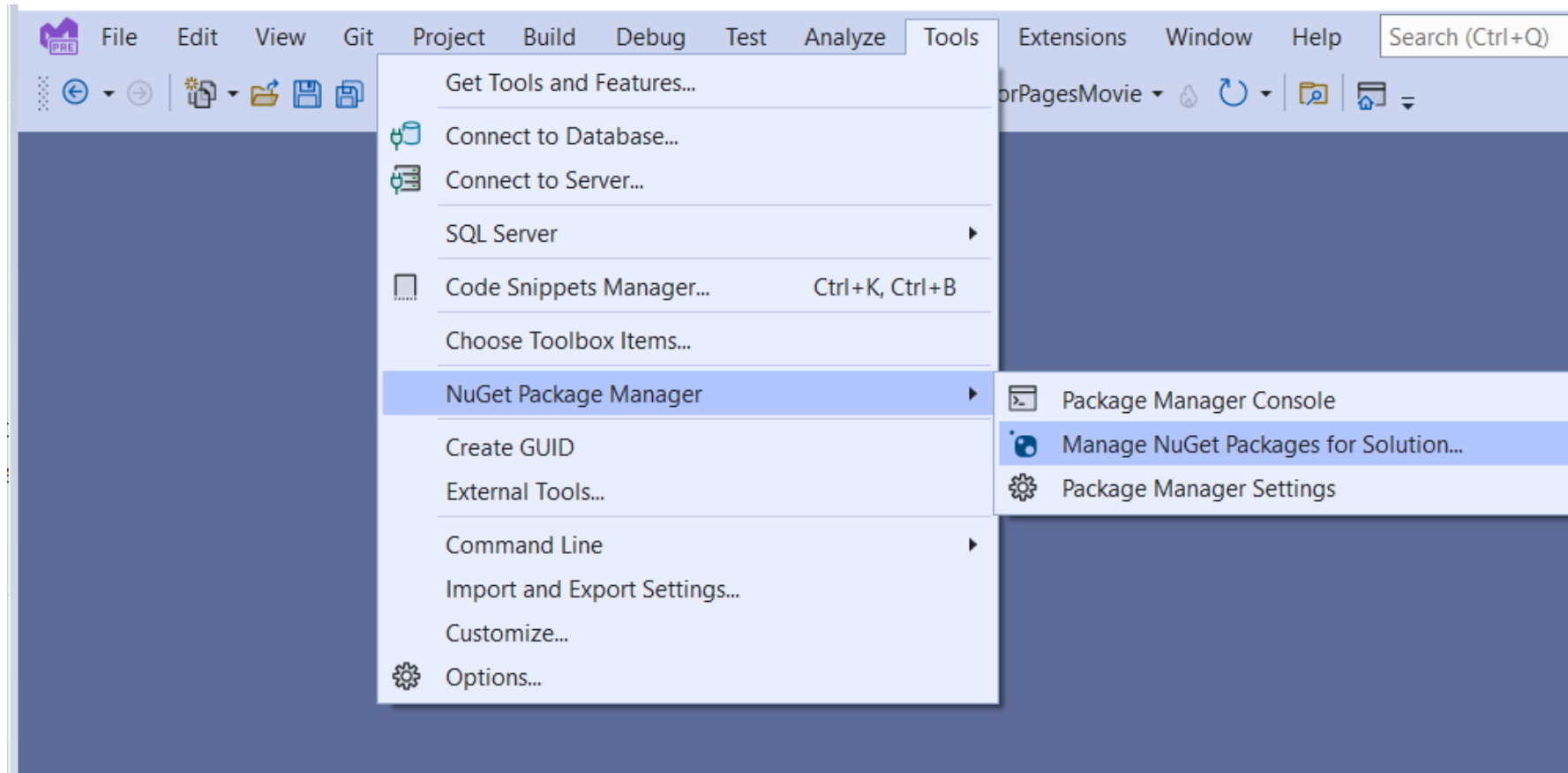
Build the project to verify there are no compilation errors.

# Scaffold the movie model

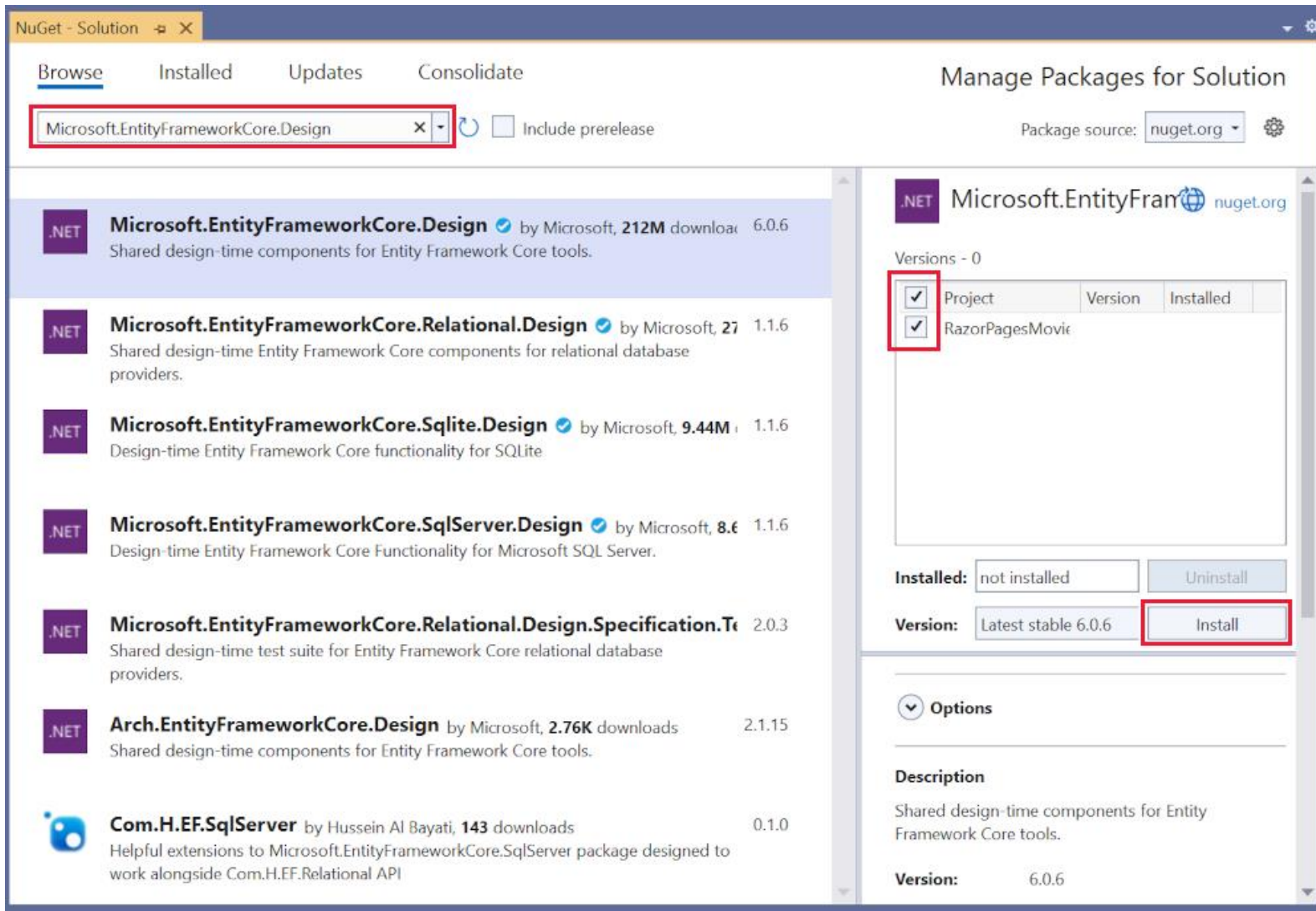
In this section, the movie model is scaffolded. That is, the scaffolding tool produces **pages for Create, Read, Update, and Delete (CRUD) operations** for the movie data model.

Add the NuGet package Microsoft.EntityFrameworkCore.Design, which is required for the scaffolding tool.

From the **Tools** menu, select **NuGet Package Manager > Manage NuGet Packages for Solution**



1. Select the **Browse** tab.
2. Enter `Microsoft.EntityFrameworkCore.Design` and select it from the list.
3. Check **Project** and then Select **Install**
4. Select **I Accept** in the **License Acceptance** dialog.

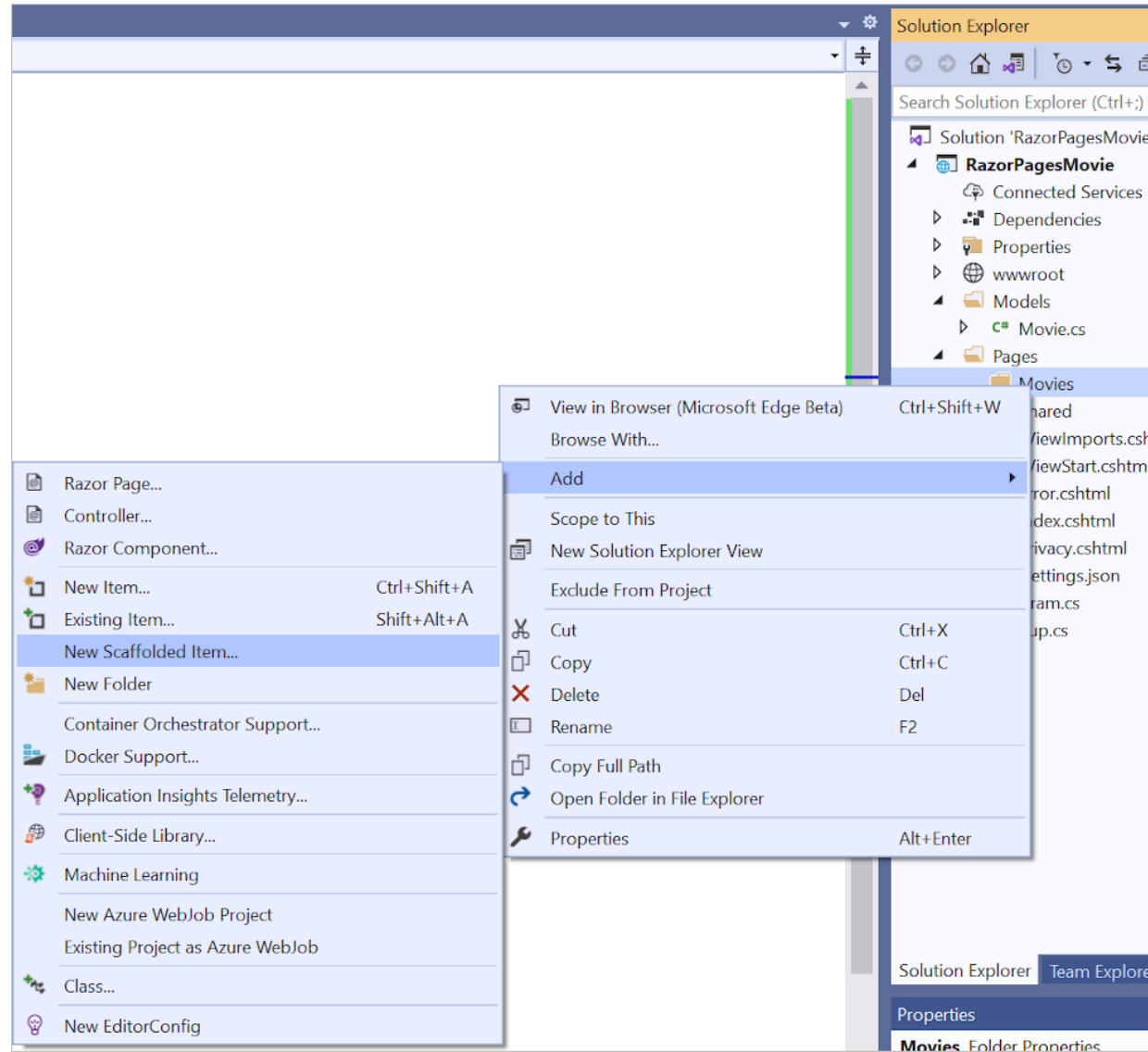


Create the *Pages/Movies* folder:

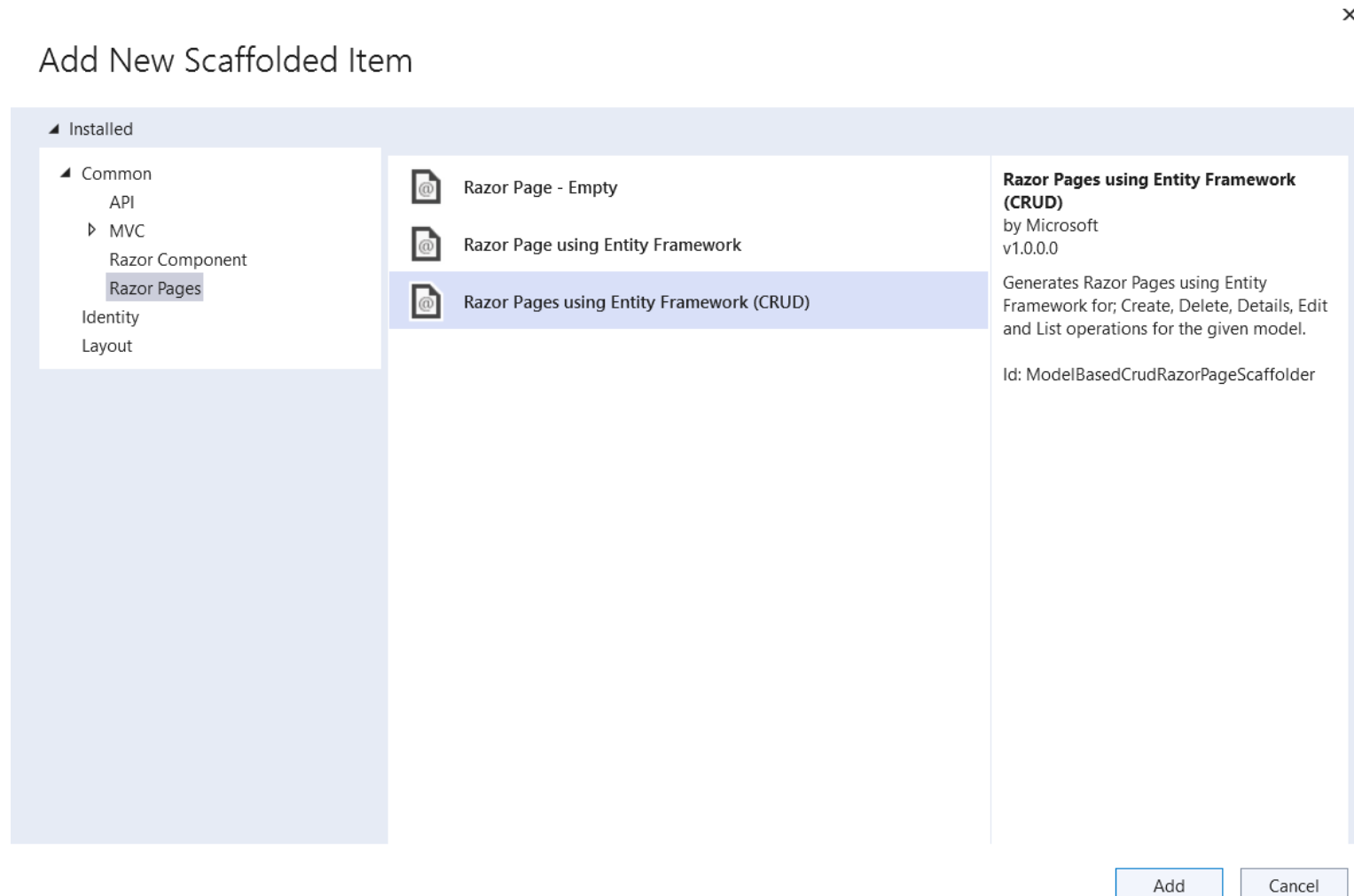
Right-click on the *Pages* folder > **Add** > **New Folder**.

Name the folder *Movies*.

Right-click on the *Pages/Movies* folder > **Add** > **New Scaffolded Item**.



In the **Add New Scaffold** dialog, select **Razor Pages using Entity Framework (CRUD)** > Add.



Complete the **Add Razor Pages using Entity Framework (CRUD)** dialog:

1.In the **Model class** drop down, select **Movie (RazorPagesMovie.Models)**.

2.In the **Data context class** row, select the + (plus) sign.

1.In the **Add Data Context** dialog, the class name `RazorPagesMovie.Data.RazorPagesMovieContext` is generated.

3.Select **Add**.

The screenshot shows a dialog box titled "Add Razor Pages using Entity Framework (CRUD)". It contains the following elements:

- A description: "Generates Razor Pages using Entity Framework for; Create, Delete, Details, Edit and List operations for the selected model."
- "Model class:" dropdown menu with "Movie (RazorPagesMovie.Models)" selected.
- "Data context class:" dropdown menu with "RazorPagesMovie.Data.RazorPagesMovieContext" selected, and a "+" button to the right.
- "Options:" section with three checkboxes:
  - ☐ Create as a partial view
  - ☒ Reference script libraries
  - ☒ Use a layout page:
- A text input field for a layout page, with a "..." button to its right.
- A note: "(Leave empty if it is set in a Razor \_viewstart file)"
- "Add" and "Cancel" buttons at the bottom right.

If you get an error message that says you need to install the Microsoft.EntityFrameworkCore.SqlServer package, repeat the steps starting with Add > New Scaffolded Item.

The appsettings.json file is updated with the connection string used to connect to a local database.



## Files created and updated

The scaffold process creates the following files:

- *Pages/Movies*: Create, Delete, Details, Edit, and Index.
- *Data/RazorPagesMovieContext.cs*

The scaffold process adds the following highlighted code to the Program.cs file:

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using RazorPagesMovie.Data;
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddDbContext<RazorPagesMovieContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("RazorPagesMovieContext")) ?

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this for production scenarios
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

add a DbContext to the dependency injection container. DbContext class serves as the central hub for interacting with your database. It acts as a bridge between your application's data model (represented by POCO classes) and the underlying database engine.

The options parameter is a lambda expression that configures the DbContext to use SQL Server as the database provider, and reads the connection string from the appsettings.json file.

## Create the initial database schema using EF's migration feature

The **migrations feature** in Entity Framework Core provides a way to:

Create the initial database schema.

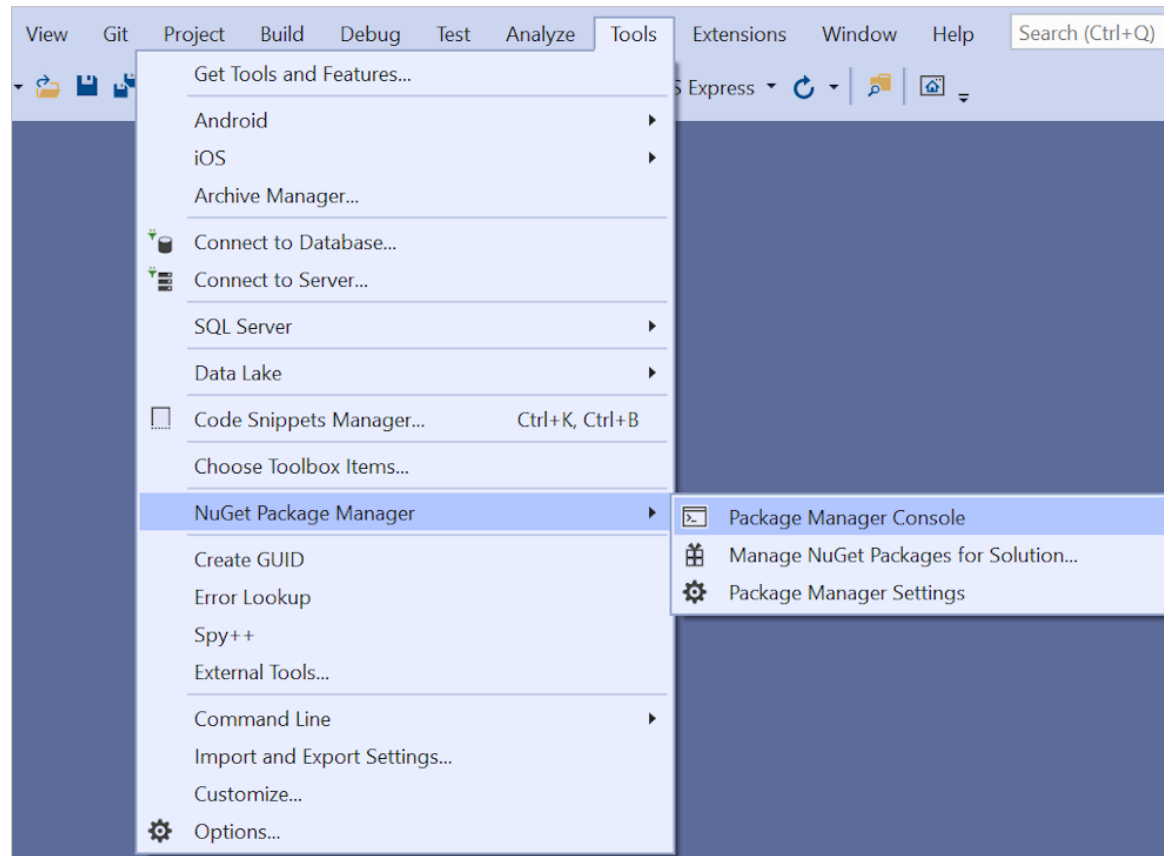
Incrementally update the database schema to keep it in sync with the app's data model. Existing data in the database is preserved.

The **Package Manager Console (PMC)** window is used to:

Add an initial migration.

Update the database with the initial migration.

From the **Tools** menu, select **NuGet Package Manager > Package Manager Console**.



In the PMC, enter the following commands:

```
Add-Migration InitialCreate  
Update-Database
```

The preceding commands install the Entity Framework Core tools and run the migrations command to generate code that creates the initial database schema.

The **migrations** command generates code to create the initial database schema. The schema is based on the model specified in **DbContext**. The **InitialCreate** argument is used to name the migrations. Any name can be used, but by convention a name is selected that describes the migration.

The **update** command runs the **Up** method in migrations that have not been applied. In this case, **update** runs the **Up** method in the **Migrations/<timestamp>\_InitialCreate.cs** file, which creates the database.

# Data context class

- In Entity Framework, a data context class is a class that inherits from the DbContext class. It provides a bridge between the application and the database, and is responsible for managing the connection to the database, querying and updating data, and tracking changes to entities.
- The data context class contains a set of properties that represent the database tables and views, called DbSets, which can be queried and manipulated in the application.

The data context `RazorPagesMovieContext`:

- Derives from [Microsoft.EntityFrameworkCore.DbContext](#).
- Specifies which entities are included in the data model.
- Coordinates EF Core functionality, such as Create, Read, Update and Delete, for the `Movie` model.

```
using
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;
using RazorPagesMovie.Models;

namespace RazorPagesMovie.Data
{
    public class RazorPagesMovieContext : DbContext
    {
        public RazorPagesMovieContext (DbContextOptions<RazorPagesMovieContext> options)
            : base(options)
        {
        }

        public DbSet<RazorPagesMovie.Models.Movie> Movie { get; set; } = default!;
    }
}
```

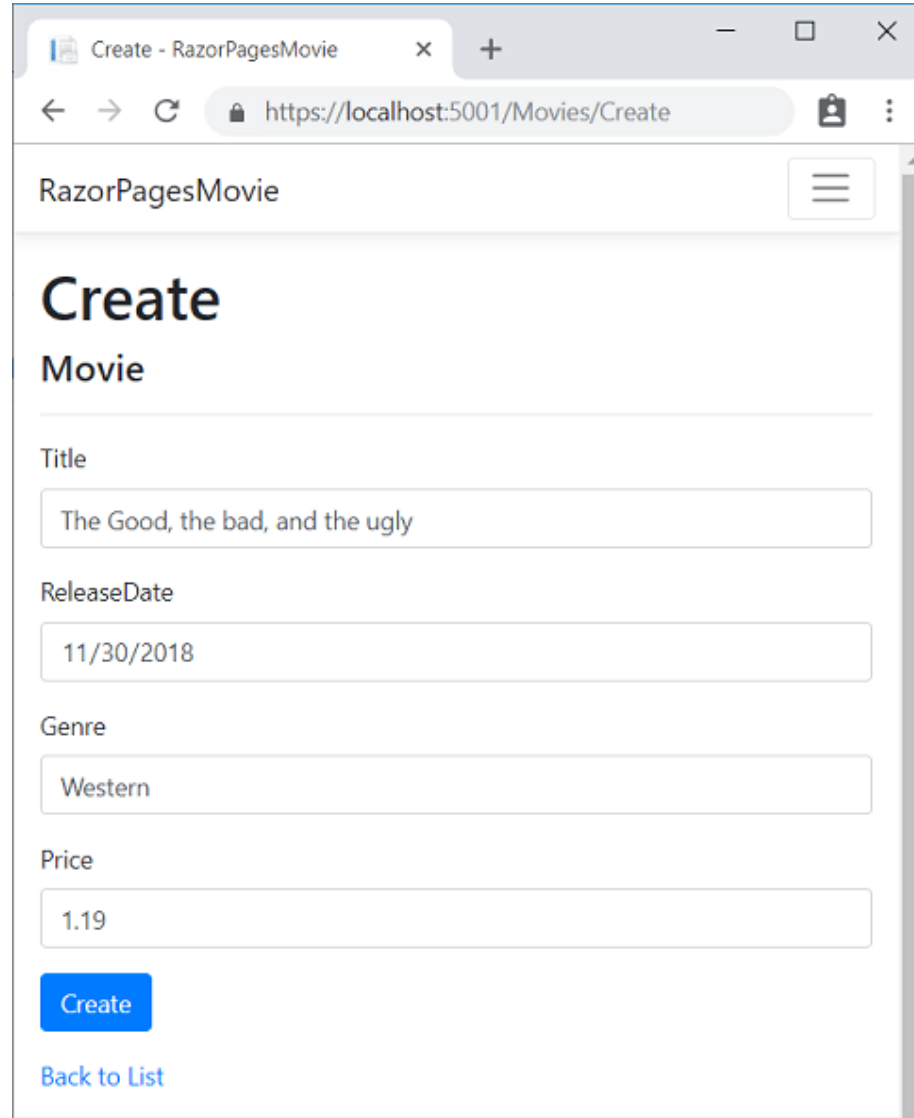
The code creates a `DbSet<Movie>` property for the entity set. In Entity Framework terminology, an entity set typically corresponds to a database table. An entity corresponds to a row in the table.

The name of the connection string is passed in to the context by calling a method on a `DbContextOptions` object. For local development, the Configuration system reads the connection string from the `appsettings.json` file.

## Test the app

1.Run the app and append /Movies to the URL in the browser (http://localhost:port/movies).

Test the Create New link.



The screenshot shows a web browser window with the title 'Create - RazorPagesMovie'. The address bar displays 'https://localhost:5001/Movies/Create'. The page header shows 'RazorPagesMovie' and a hamburger menu icon. The main content area is titled 'Create Movie' and contains a form with the following fields:

- Title:** A text input field containing 'The Good, the bad, and the ugly'.
- ReleaseDate:** A date input field containing '11/30/2018'.
- Genre:** A text input field containing 'Western'.
- Price:** A text input field containing '1.19'.

Below the form is a blue 'Create' button and a blue link labeled 'Back to List'.

Test the **Edit**, **Details**, and **Delete** links.

# Seed the database

Create a new class named SeedData in the Models folder with the following code:

```
using Microsoft.EntityFrameworkCore;
using RazorPagesMovie.Data;

namespace RazorPagesMovie.Models;

public static class SeedData
{
    public static void Initialize(IServiceProvider serviceProvider)
    {
        using (var context = new RazorPagesMovieContext(
            serviceProvider.GetRequiredService<
                DbContextOptions<RazorPagesMovieContext>>()))
        {
            if (context == null || context.Movie == null)
            {
                throw new ArgumentNullException("Null RazorPagesMovieContext");
            }

            // Look for any movies.
            if (context.Movie.Any())
            {
                return; // DB has been seeded

                context.Movie.AddRange(
```

If there are any movies in the database, the seed initializer returns and no movies are added.

## Add the seed initializer

Update the `Program.cs` with the following highlighted code:

```
var app = builder.Build();

using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;
    SeedData.Initialize(services);
}
```

Get a database context instance from the dependency injection (DI) container.

Call the `seedData.Initialize` method, passing to it the database context instance.

Dispose the context when the seed method completes. The `using` statement ensures the context is disposed.

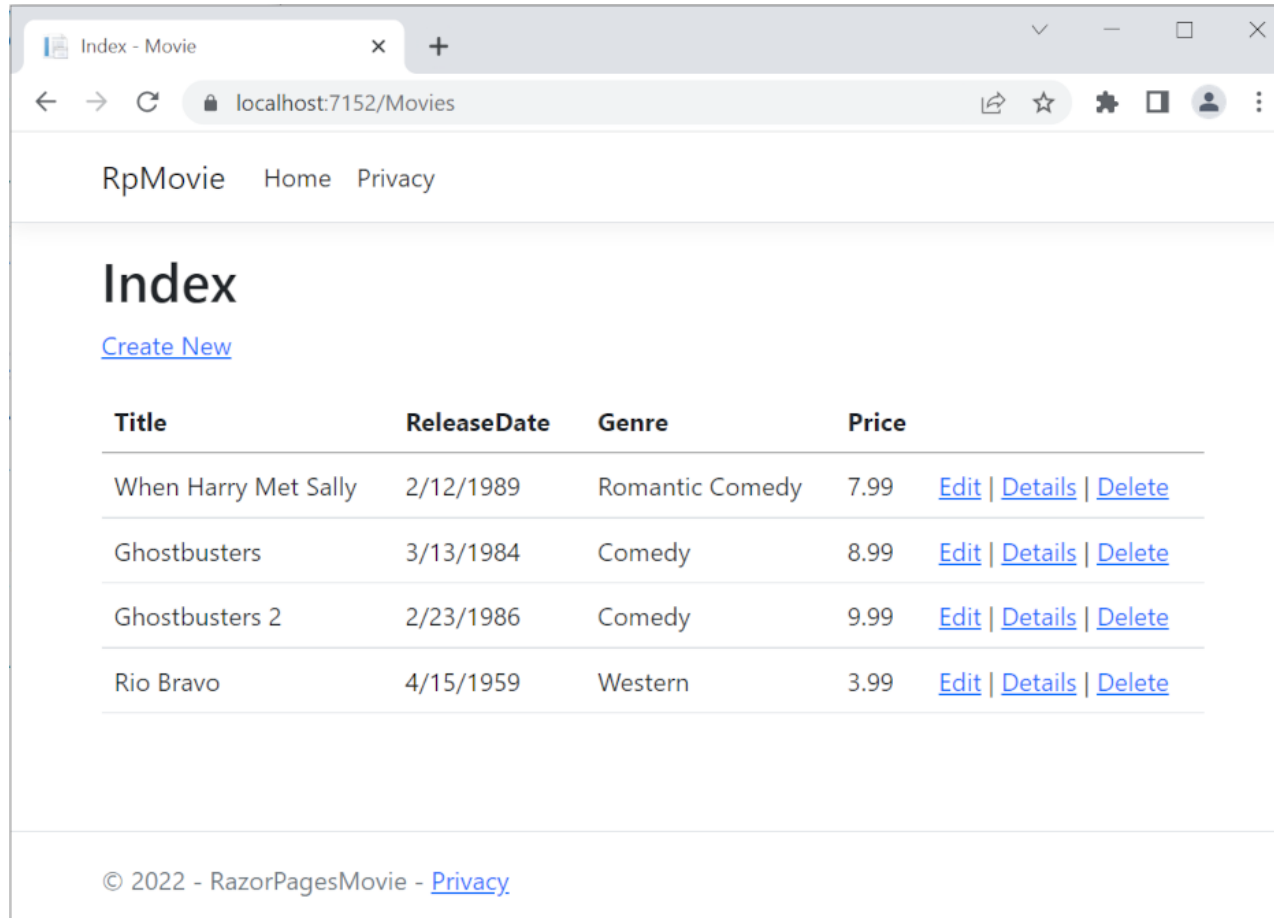
In ASP.NET Core dependency injection, there are three main lifetime options for registered services: singleton, scoped, and transient. Singleton services are created once and reused for the lifetime of the application. Scoped services are created once per scope (e.g. per HTTP request) and are disposed of when the scope is disposed of. Transient services are created each time they are requested and are not reused.



## Test the app

Delete all the records in the database so the seed method will run. Stop and start the app to seed the database.

The app shows the seeded data:



Index				
<a href="#">Create New</a>				
Title	ReleaseDate	Genre	Price	
When Harry Met Sally	2/12/1989	Romantic Comedy	7.99	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Ghostbusters	3/13/1984	Comedy	8.99	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Ghostbusters 2	2/23/1986	Comedy	9.99	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Rio Bravo	4/15/1959	Western	3.99	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>

© 2022 - RazorPagesMovie - [Privacy](#)

## Examine the Pages/Movies/Index.cshtml.cs Page Model:

When a GET request is made for the page, the OnGetAsync method returns a list of movies to the Razor Page.

On a Razor Page, OnGetAsync or OnGet is called to initialize the state of the page. In this case, OnGetAsync gets a list of movies and displays them.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using RazorPagesMovie.Data;
using RazorPagesMovie.Models;

namespace RazorPagesMovie.Pages.Movies
{
    public class IndexModel : PageModel
    {
        private readonly RazorPagesMovie.Data.RazorPagesMovieContext _context;

        public IndexModel(RazorPagesMovie.Data.RazorPagesMovieContext context)
        {
            _context = context;
        }

        public IList<Movie> Movie { get; set; } = default!;

        public async Task OnGetAsync()
        {
            if (_context.Movie != null)
            {
                Movie = await _context.Movie.ToListAsync();
            }
        }
    }
}
```

# Asynchronous EF methods in ASP.NET Core web apps

- Asynchronous programming is the default mode for ASP.NET Core and EF Core.
- A web server has a limited number of threads available, and in high load situations all of the available threads might be in use. When that happens, the server can't process new requests until the threads are freed up.
  - With synchronous code, many threads may be tied up while they aren't doing work because they're waiting for I/O to complete.
  - With asynchronous code, when a process is waiting for I/O to complete, its thread is freed up for the server to use for processing other requests. As a result, asynchronous code enables server resources to be used more efficiently, and the server can handle more traffic without delays.
- Asynchronous code does introduce a small amount of overhead at run time. For low traffic situations, the performance hit is negligible, while for high traffic situations, the potential performance improvement is substantial.

# OnGetAsync()

- In the following code, the `async` keyword, `Task` return value, `await` keyword, and `ToListAsync` method make the code execute asynchronously.

```
public async Task OnGetAsync()
{
    if (_context.Movie != null)
    {
        Movie = await _context.Movie.ToListAsync();
    }
}
```

- The `async` keyword tells the compiler to:
  - Generate callbacks for parts of the method body.
  - Create the `Task` object that's returned.
- The `Task` return type represents ongoing work.
- The `await` keyword causes the compiler to split the method into two parts. The first part ends with the operation that's started asynchronously. The second part is put into a callback method that's called when the operation completes.
- `ToListAsync` is the asynchronous version of the `ToList` extension method.

# The Create page model

- The OnGet method initializes any state needed for the page. The Create page doesn't have any state to initialize, so Page is returned.
  - The Page method creates a `PageResult` object that renders the `Create.cshtml` page.
- The Movie property uses the `[BindProperty]` attribute to take part to model binding. When the Create form posts the form-values, the ASP.NET Core runtime binds the posted values to the Movie model.
- The **OnPostAsync method** is run when the page posts form-data.
- If there are any model errors, the form is redisplayed, along with any form data posted.
  - Most model errors can be caught on the client-side before the form is posted. An example of a model error is posting a value for the date field that cannot be converted to a date.
- If there are no model errors:
  - The data is saved.
  - The browser is redirected to the Index page.