ADVANCED WEB APPLICATION DEVELOPMENT

Abstract

Advanced Web application development involves creating application programs that are hosted on remote servers and accessed by users over the Internet. Unlike traditional applications, web apps do not require downloading and can be accessed through a network. Users can interact with web applications using popular web browsers like Google Chrome, Safari, or Mozilla Firefox. Microsoft offers ASP.NET MVC as a web application framework, designed to empower programmers in constructing dynamic websites. This framework enables developers to utilize a comprehensive programming language like C# to easily build web applications. In most cases, web applications are written using JavaScript, Cascading Style Sheets (CSS), and HTML5, providing

FACULTY OF INFORMATION TECHNOLOGY
Internal circulation, 2023

TABLE OF CONTENTS

LAB 1
HTML CSS JAVASCRIPT

I. TARGET(+)

- ✓ Understand and apply HTML, CSS, JS to build interfaces for web applications.

II. REFERENCES

- ✓ HTML Tutorial: https://www.w3schools.com/html/default.asp
- ✓ CSS Tutorial: https://www.w3schools.com/css/default.asp
- ✓ JavaScript Tutorial: https://www.w3schools.com/js/default.asp
- ✓ Bootstrap 5 Tutorial: https://www.w3schools.com/bootstrap5/index.php

III. REQUIREMENTS(+)

- ✓ Create account github: https://github.com
- ✓ Install Visual Studio Code 2022: https://learn.microsoft.com/en-us/visualstudio/install/install-visual-studio?view=vs-2022 or https://www.youtube.com/watch?v=JPZsB_6yHVo

IV. PREPARATION

- ✓ Add your project to source control https://www.c-sharpcorner.com/article/how to-use-github-invisual-studio-2022/
- ✓ Using Git source control in VS Code https://code.visualstudio.com/docs/sourcecontrol/overview
- ✓ All of LAB in this course, please submit your github/gitlab/other project link

V. IN LAB(ACTIVITIES)

- ✓ 1. Read: DFT3013 WEB DESIGN TECHNOLOGIES https://prezi.com/25qw6flmrpjm/dft3013-web-design-technologies-introduction
- ✓ 2. Write your first C# code https://learn.microsoft.com/en-us/training/modules/csharp-write-first/
- ✓ 3. Get started with web development using Visual Studio Code https://learn.microsoft.com/en-us/training/modules/get-started-with-web-development/

- ✓ 4. Learn the basics of web accessibility
  https://learn.microsoft.com/enus/training/modules/web-development-101-accessibility/

SUBMIT YOUR CODE TO GIT!

- - - END LAB 01 - - -

LAB 2
ASP.NET CORE MVC

I. TARGET

- ✓ ASP.NET Core MVC web development with controllers and views

II. REQUIREMENTS

- ✓ Create a web app.
- ✓ Add and scaffold a model
- ✓ Install Visual Studio 2022: https://learn.microsoft.com/en-us/visualstudio/install/install-visual-studio?view=vs-2022

III. IN LAB

1. Create 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. It's important to name the project MvcMovie. Capitalization needs to match each namespace when code is copied.
- Select **Next**.
- In the Additional information dialog, select **.NET 6.0 (Long-term support)**.
- Select **Create.**

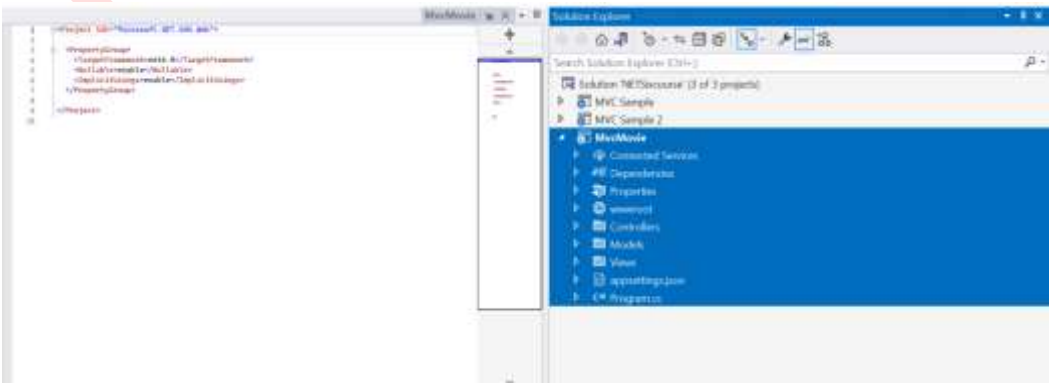


Figure 1.1: Create a new project

- 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:

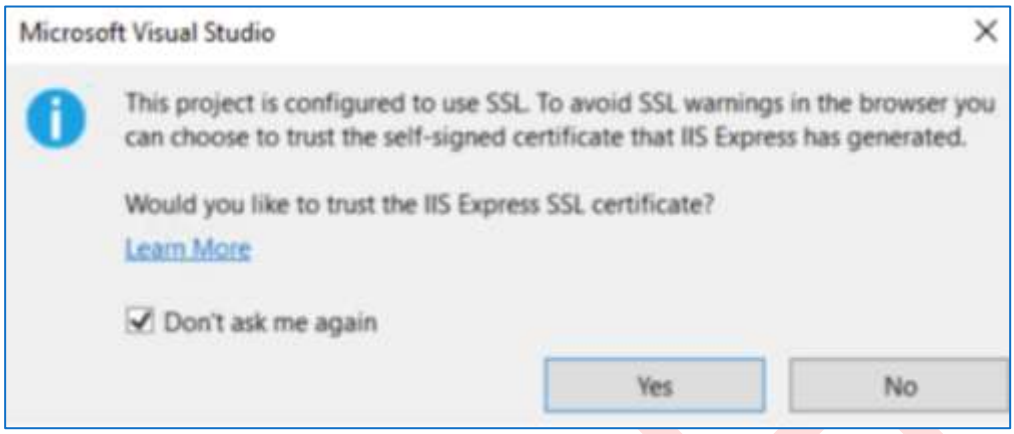


*Figure 1.2 Project is **not** yet configured to use SSL*

- Select Yes if you trust the IIS Express SSL certificate. The following dialog is displayed:

Figure 1.3: IIS Express Self-Signed Certificate not trusted using Visual Studio 2022

- Select Yes if you agree to trust the development certificate.

Visual Studio runs the app and opens the default browser. The address bar shows localhost:<port#> and not something like example.com. The standard hostname for your local computer is localhost. When Visual Studio creates a web project, a random port is used for the web server.

Launching the app without debugging by selecting Ctrl+F5 allows you to:

– Make code changes.

– Save the file.

– Quickly refresh the browser and see the code changes.

You can launch the app in debug or non-debug mode from the Debug menu:

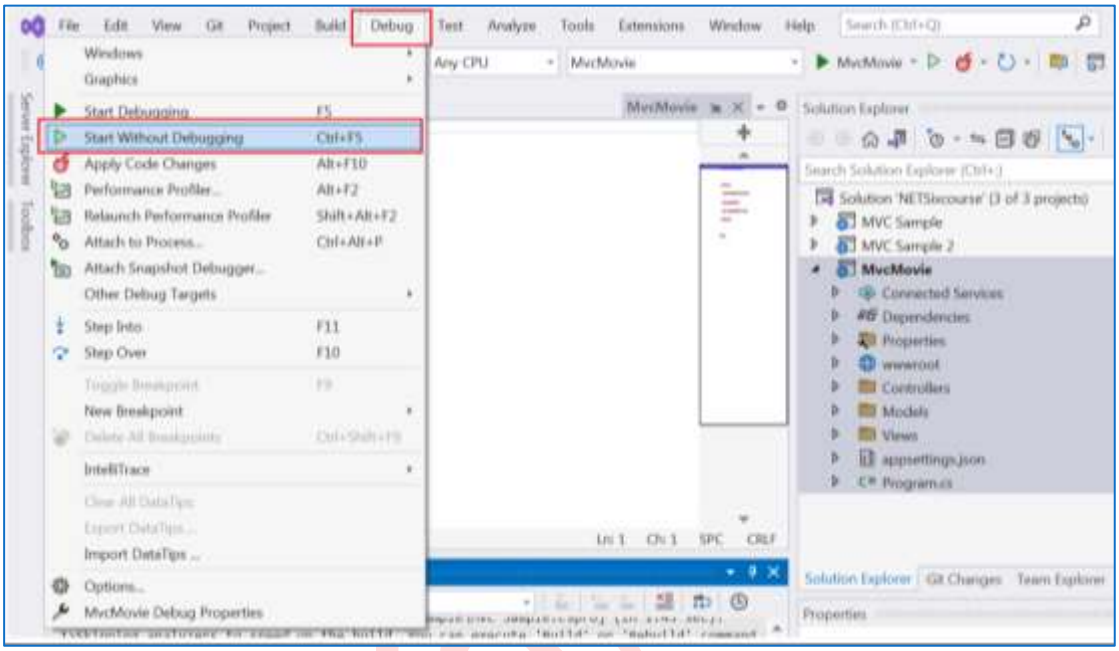


Figure 1.4: Launch the app in debug or non-debug mode from the Debug menu

You can debug the app by selecting the MvcMovie button in the toolbar:

The following image shows the app:

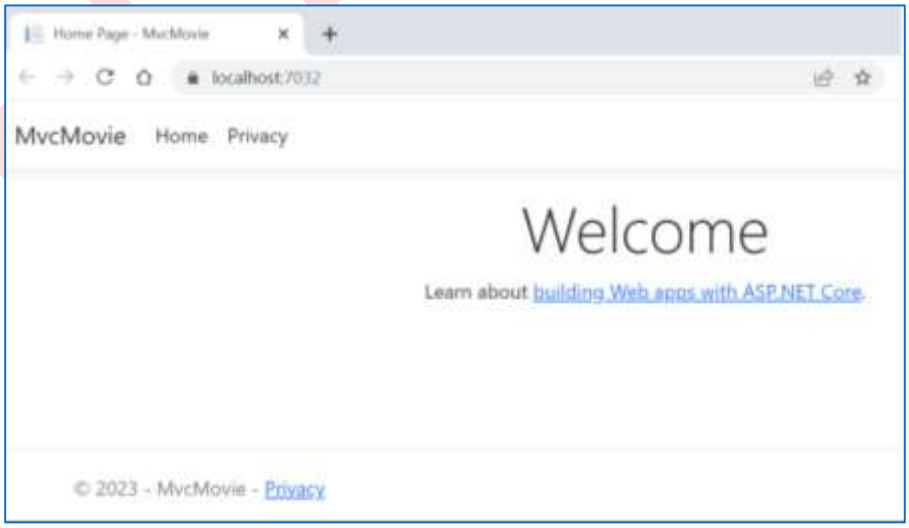


Figure 1.5 App demo

2. Add a controller to an ASP.NET Core MVC app

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. 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 Lab, 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.

In an MVC application, the role of the view is to solely present information. On the other hand, the controller is responsible for managing and responding to user input and interactions. For instance, the controller deals with URL segments and query-string values, and then forwards these values to the model. Subsequently, the model can utilize these values to perform database queries. For example:

- 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, which are detailed later in the Lab.

The MVC architectural pattern divides an application into three primary groups of components: Models, Views, and Controllers. This pattern facilitates the separation of concerns, where UI logic resides in the view, input logic resides in the controller, and business logic resides in the model. This separation aids in handling complexity during the app development process by allowing focused work on specific implementation

aspects without affecting the code of others. For instance, one can modify the view code without relying on the business logic code.

The MVC project contains folders for the Controllers and Views. In **Solution Explorer**, right-click **Controllers > Add > Controller**.

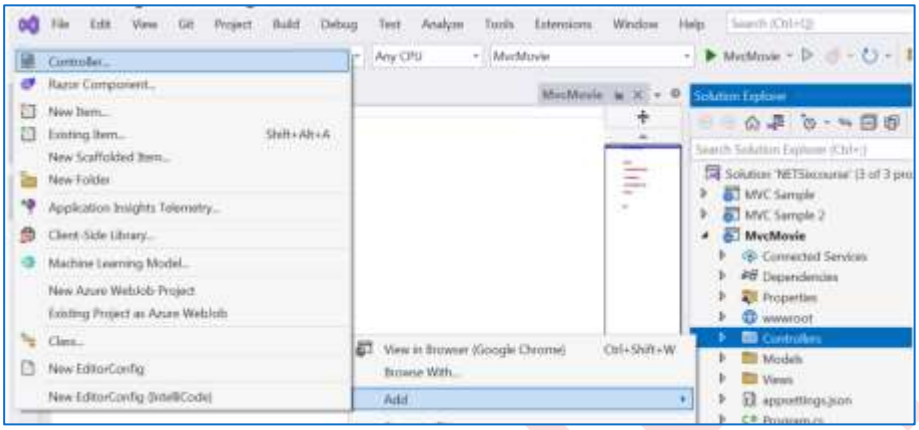


Figure 2.1: The MVC project contains folders for the Controllers and Views.

In the **Add New Scaffolded** Item dialog box, select **MVC Controller - Empty > Add**.

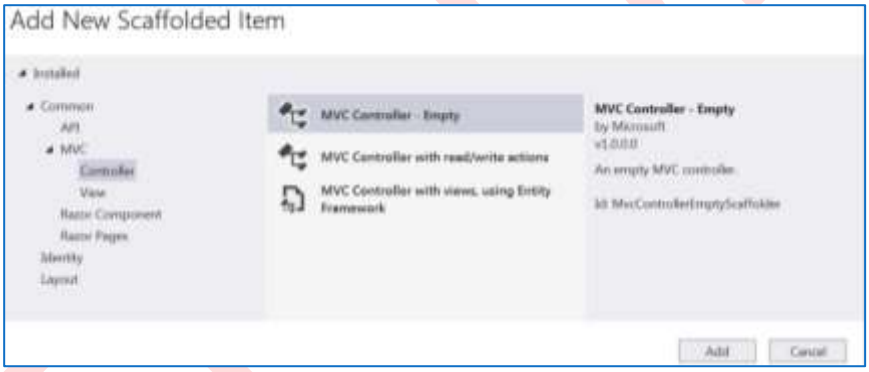


Figure 2.2: Add New Scaffolded Item dialog box

In the **Add New Item - MvcMovie** dialog, enter **HelloWorldController.cs** and select **Add**.

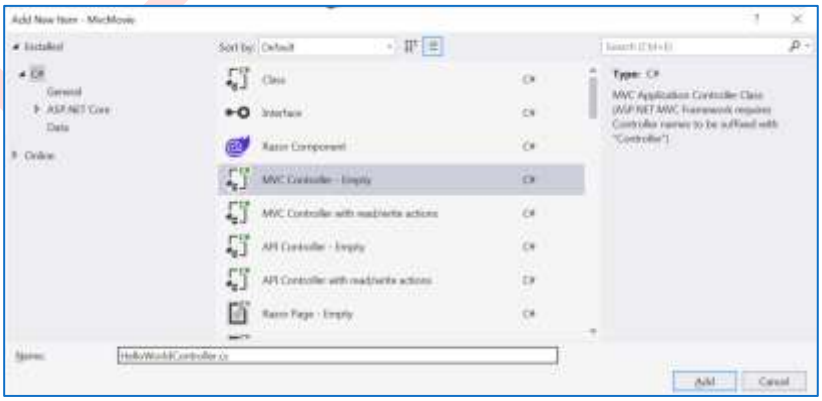


Figure 2.3: Add New Item - MvcMovie dialog

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

```
using Microsoft.AspNetCore.Mvc;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        //public IActionResult Index()
        //{
        //    return View();
        //}

        //
        // GET: /HelloWorld/

        public string Index()
        {
            return "This is my default action...";
        }

        //
        // GET: /HelloWorld/Welcome/

        public string Welcome()
        {
            return "This is the Welcome action method...";
        }
    }
}
```

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:

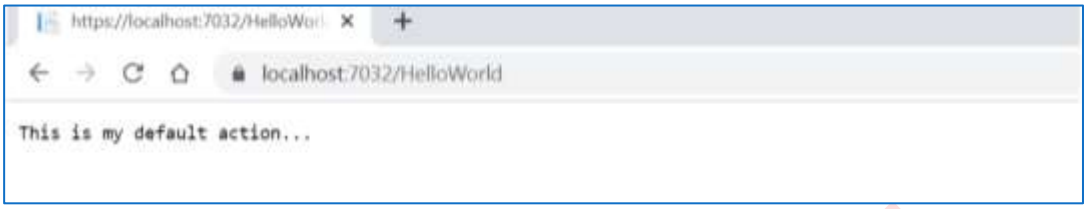
- Is a targetable URL in the web application, such as https://localhost:5001/HelloWorld
- Combines:
  - ✓ The protocol used: HTTPS.
  - ✓ The network location of the web server, including the TCP port: localhost:5001.
  - ✓ The target URI: HelloWorld.

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 lab, 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.



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]**

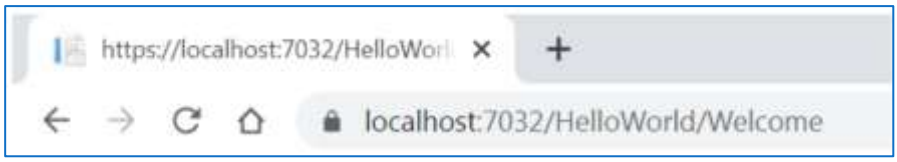
The routing format is set in the **Program.cs** file.

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

When you browse to the app and don't supply any URL segments, it defaults to the "Home" controller and the "Index" method specified in the template line highlighted above. In the preceding URL segments:

- The first URL segment determines the controller class to run. So localhost:5001/HelloWorld maps to the **HelloWorld Controller class.**
- The second part of the URL segment determines the action method on the class. So **localhost:5001/HelloWorld/Index** causes the **Index** method of the **HelloWorldController** class to run. Notice that you only had to browse to **localhost:5001/HelloWorld** and the **Index** method was called by default. **Index** is the default method that will be called on a controller if a method name isn't explicitly specified.
- The third part of the URL segment **(id)** is for route data. Route data is explained later in the Lab.

Browse to: https://localhost:{PORT}/HelloWorld/Welcome. Replace {PORT} with your port number. The Welcome method runs and returns the string **This is the Welcome action method....** For this URL, the controller is **HelloWorld** and **Welcome** is the action method. You haven't used the **[Parameters]** part of the URL yet.



Modify the code to pass some parameter information from the URL to the controller. For example, **/HelloWorld/Welcome?name=Rick&numtimes=4.**

Change the **Welcome** method to include two parameters as shown in the following code.

```
//
// GET: /HelloWorld/Welcome/
// Requires using System.Text.Encodings.Web;
public string Welcome(string name, int numTimes = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, NumTimes is: {numTimes}");
}
```

The preceding code:

- Uses the C# optional-parameter feature to indicate that the **numTimes** parameter defaults to 1 if no value is passed for that parameter.
- Uses **HtmlEncoder.Default.Encode** to protect the app from malicious input, such as through JavaScript.
- Uses **Interpolated Strings in $"Hello {name}, NumTimes is: {numTimes}".**

Run the app and browse to:

https://localhost:{PORT}/HelloWorld/Welcome?name=Rick&numtimes=4.

Replace {PORT} with your port number.

Try different values for **name** and **numtimes** in the URL. The MVC model binding system automatically maps the named parameters from the query string to parameters in the method.

In the previous image:

- The URL segment Parameters isn't used.
- The **name** and **numTimes** parameters are passed in the query string.
- The **? (question mark)** in the above URL is a separator, and the query string follows.

– The **&** character separates field-value pairs.

Replace the Welcome method with the following code:

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

Run the app and enter the following URL:

https://localhost:{PORT}/HelloWorld/Welcome/3?name=Rick

In the preceding URL:

– The third **URL** segment matched the route parameter **id**.

– The **Welcome** method contains a parameter **id** that matched the **URL** template in the **MapControllerRoute** method.

– The trailing ? starts the query string.

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

Figure 2.4 MapContronllerRouter example

In the preceding example:

– The third URL segment matched the route parameter id.

– The **Welcome** method contains a parameter **id** that matched the **URL** template in the **MapControllerRoute** method.

– The trailing **? (in id?)** indicates the **id** parameter is optional.

3. Add a view to an ASP.NET Core MVC app

In this section, you modify the **HelloWorldController** class to use **Razor** view files. This cleanly encapsulates the process of generating HTML responses to a client.

View templates are created using Razor. Razor-based view templates:

– Have a .cshtml file extension.

– Provide an elegant way to create HTML output with C#.

Currently the **Index** method returns a string with a message in the controller class. In the **HelloWorldController** class, replace the **Index** method with the following code:

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

The preceding code:

- Calls the controller's View method.
- Uses a view template to generate an HTML response.

Controller methods:

- Are referred to as action methods. For example, the **Index** action method in the preceding code.
- Generally return an **IActionResult** or a class derived from **ActionResult**, not a type like **string**.

Right-click on the **Views** folder, and then **Add > New Folder** and name the folder HelloWorld. **Right-click** on the **Views/HelloWorld folder**, and then **Add > New Item.**

In the **Add New Item - MvcMovie** dialog:

- In the search box in the upper-right, enter view
- Select **Razor View – Empty**
- Keep the **Name** box value, **Index.cshtml.**
- Select **Add**

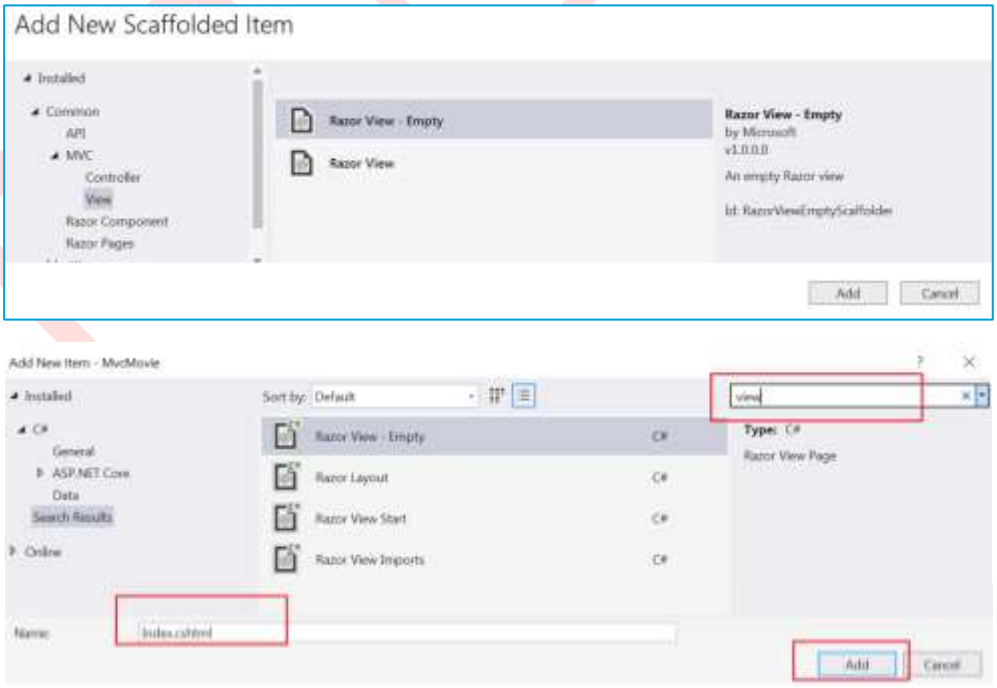


Figure 3.1: Add New Item - MvcMovie dialog

Replace the contents of the Views/HelloWorld/Index.cshtml Razor view file with the following:

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

<h2>Index</h2>

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

Navigate to **https://localhost:{PORT}/HelloWorld**:

- The **Index** method in the **HelloWorldController** ran the statement **return View();**, which specified that the method should use a view template file to render a response to the browser.
- A view template file name wasn't specified, so MVC defaulted to using the default view file. When the view file name isn't specified, the default view is returned. The default view has the same name as the action method, **Index** in this example. The view template **/Views/HelloWorld/Index.cshtml** is used.
- The following image shows the string "Hello from our View Template!" hard-coded in the view as *Figure 3.2*

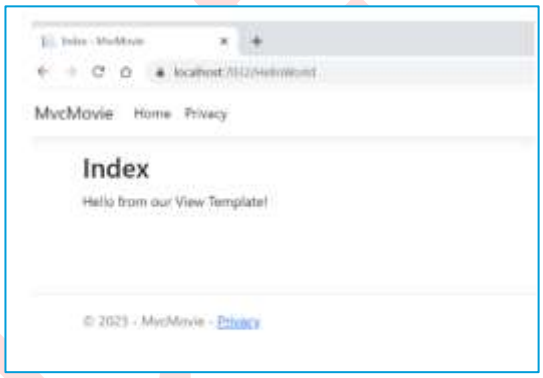


Figure 3.2

4. Change views and layout pages

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 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 the **title, footer, and menu link in the layout file**. Replace the content of the **Views/Shared/_Layout.cshtml** file with the following markup. The changes are highlighted:

```
<title>@ViewData["Title"] - Movie App</title>
…
<a class="navbar-brand" asp-area="" asp-controller="Movies" asp-action="Index">Movie App</a>
…
  &copy; 2023 - Movie App - <a asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
```

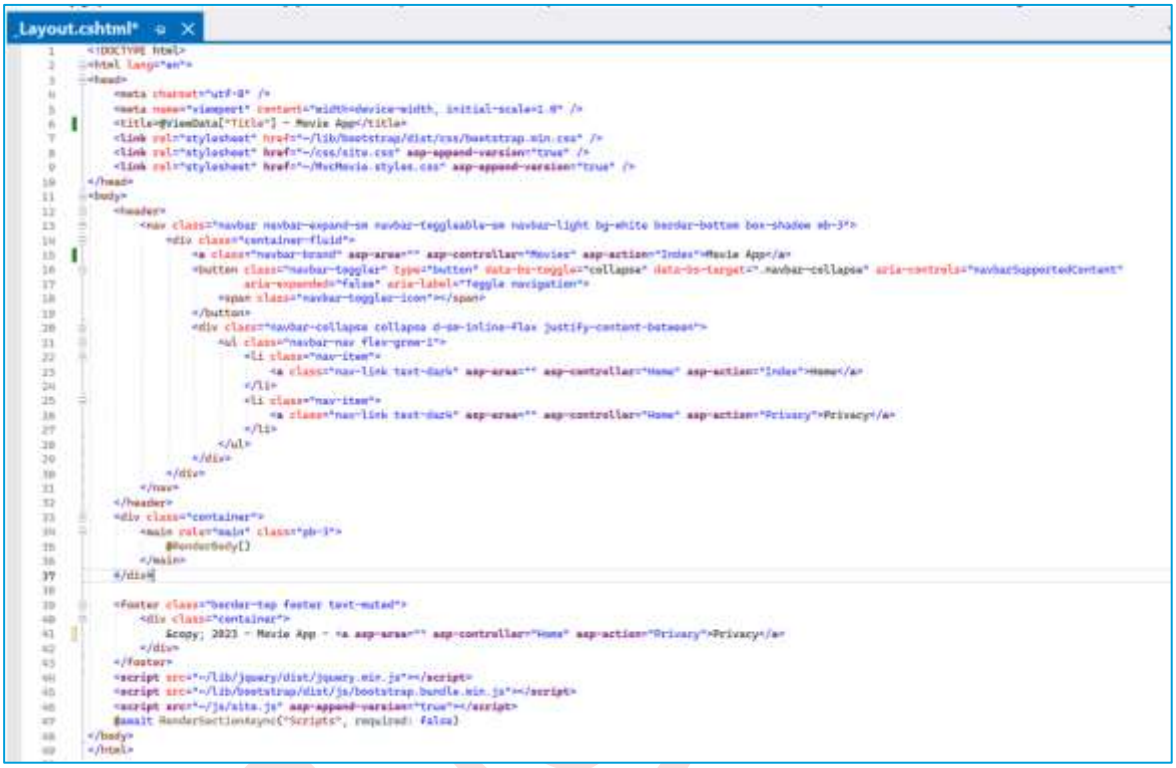


Figure 4.1 Layout page

The preceding markup made the following changes:

- Three occurrences of MvcMovie to Movie App.
- The anchor element MvcMovie to <aclass="navbar-brand" asp-controller="Movies" asp-action="Index">Movie App

In the preceding markup, the asp-area="" anchor Tag Helper attribute and attribute value was omitted because this app isn't using Areas.

Note: The **Movies** controller hasn't been implemented. At this point, the **Movie App** link isn't functional.

Save the changes and select the **Privacy** link. Notice how the title on the browser tab displays **Privacy Policy - Movie App** instead of **Privacy Policy - MvcMovie**

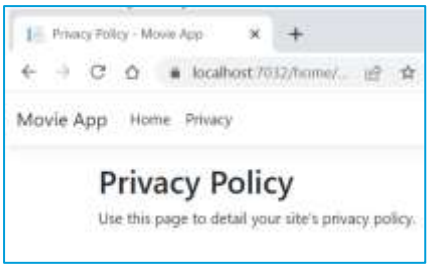


Figure 4.2: Privacy Policy view on the browser.

Select the Home link. Notice that the title and anchor text display Movie App. The changes were made once in the layout template and all pages on the site reflect the new link text and new title:

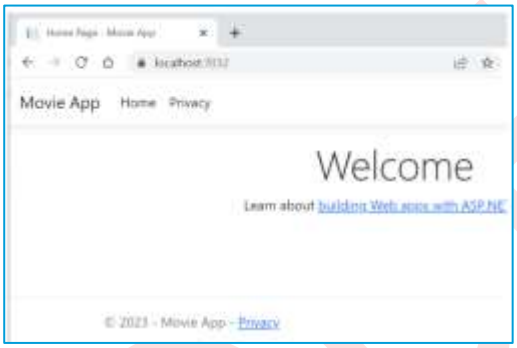


Figure 4.3: Home view on the browser

Examine the Views/_ViewStart.cshtml file: @{ Layout = "_Layout"; }

The **Views/_ViewStart.cshtml** file brings in the **Views/Shared/_Layout.cshtml** file to each view. The **Layout** property can be used to set a different layout view, or set it to null so no layout file will be used.

Open the **Views/HelloWorld/Index.cshtml** view file.

Change the title and <h2> element as highlighted in the following:

```
@{
    ViewData["Title"] = "Movie List";
}

<h2>My Movie List</h2>

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

The title <h2> and element are slightly different so it's clear which part of the code changes the display. **ViewData["Title"] = "Movie List";** in the code above sets the **Title** property of the **ViewData** dictionary to "Movie List". The **Title** property is used in the **<title>** HTML element in the layout page (_Layout) **<title>@ViewData["Title"] - Movie App</title>**

Save the change and navigate to **https://localhost:{PORT}/HelloWorld**. Notice that the following have changed:

- Browser title.
- Primary heading.

The content in the Index.cshtml view template is combined with the Views/Shared/_Layout.cshtml view template, resulting in a single HTML response sent to the browser. The utilization of layout templates simplifies the process of implementing changes that affect all pages within an application.

However, the "Hello from our View Template!" message, which serves as a small piece of data, is statically coded. It's worth noting that the MVC application currently lacks a model component, represented by the "M" in MVC, but features both a view ("V") and a controller ("C").

5. Passing Data from the Controller to the View

In response to an incoming URL request, controller actions are triggered. A controller class contains the code responsible for managing browser requests. It retrieves data from a data source and determines the appropriate response to send back to the browser. To generate and format an HTML response for the browser, view templates can be utilized within a controller. Thus, controllers play a crucial role in supplying the necessary data for rendering a response in a view template.

View templates should not:

- Do business logic
- Interact with a database directly

A view template should work only with the data that's provided to it by the controller. Maintaining this "separation of concerns" helps keep the code:

- Clean.
- Testable.
- Maintainable.

At present, the **Welcome** method within the **HelloWorldController** class receives a name and an ID parameter and directly displays their values on the browser. Instead of rendering this response as a string within the controller, it is recommended to modify the

controller to utilize a view template. By using a view template, a dynamic response can be generated. This requires passing the necessary data from the controller to the view in order to generate the response. To accomplish this, the controller should store the dynamic data (parameters) that the view template requires in a ViewData dictionary. Subsequently, the view template can access the dynamic data from the dictionary.

In **HelloWorldController.cs**, change the **Welcome** method to add a **Message** and **NumTimes** value to the **ViewData** dictionary.

The **ViewData** dictionary serves as a flexible container that can hold objects of any type. Initially, the **ViewData** object doesn't have any predefined properties until items are added to it. When it comes to the MVC model binding system, it automatically associates the named parameters "name" and "numTimes" from the query string with the corresponding parameters in the method. This behavior can be observed in the complete **HelloWorldController**.

```
public IActionResult Welcome(string name, int numTimes = 1)
{
            ViewData["Message"] = "Hello " + name;
            ViewData["NumTimes"] = numTimes;

            return View();
}
```

The **ViewData** dictionary object contains data that will be passed to the view. Create a **Welcome** view template named **Views/HelloWorld/Welcome.cshtml**. You'll create a loop in the **Welcome.cshtml** view template that displays "Hello" NumTimes. Replace the contents of **Views/HelloWorld/Welcome.cshtml** with the following:

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

<h2>Welcome</h2>

<ul>
    @for (int i = 0; i < (int)ViewData["NumTimes"]!; i++)
    {
        <li>@ViewData["Message"]</li>
    }
</ul>
```

Save your changes and browse to the following URL: **https://localhost:{PORT}/HelloWorld/Welcome?name=Rick&numtimes=4**

Data is taken from the URL and passed to the controller using the **MVC model binder**. The controller packages the data into a **ViewData** dictionary and passes that object to the view. The view then renders the data as HTML to the browser:

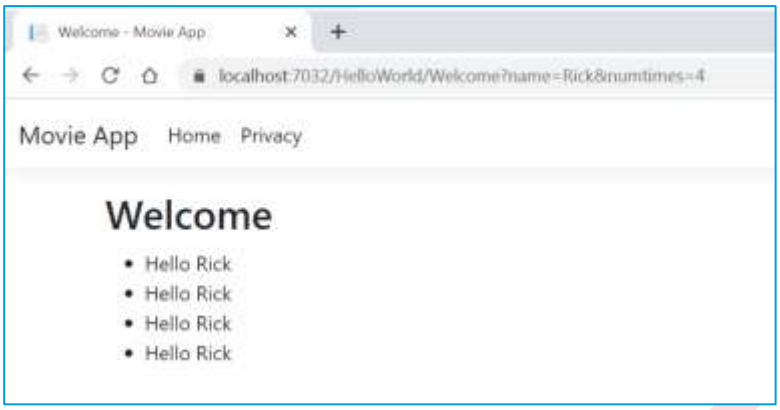


Figure 5.1: Renders the data from VIewData as HTML to the browser

In the previous example, data was passed from the controller to a view using the ViewData dictionary. However, in the later part of the Lab, a view model is utilized instead to transmit data from a controller to a view. The view model approach for data passing is considered preferable over the use of the ViewData dictionary.

In the next lab, a database of movies is created.

SUBMIT YOUR CODE TO GIT!

- - - END LAB 02 - - -

LAB 3
ASP.NET Core MVC (cont)

I. TARGET

- ✓ ASP.NET Core MVC web development
- ✓ Add a model to an ASP.NET Core MVC app

II. REFERENCES

- ✓ Entity Framework Core for Beginners, https://www.youtube.com/playlist?list=PLdo4fOcmZ0oXCPdC3fTFA3Z79-eVH3K-s
- ✓ Understanding AddTransient Vs AddScoped Vs AddSingleton In ASP.NET Core https://www.c-sharpcorner.com/article/understanding-addtransient-vs-addscoped-vs-addsingletonin-asp-net-core/

III. REQUIREMENTS

- ✓ Install SQL Server 2022 Express
- ✓ Install SQL Server Management Studio (SSMS)
- ✓ SQL Server tools and connectors, https://www.microsoft.com/en-us/sql-server/sql-server-downloads
- ✓ Install SQL Server 2022 and Install SQL Server Management Studio (SSMS) https://www.youtube.com/watch?v=iNXUxqy1svM

IV. IN LAB

In this Lab, classes are added for managing movies in a database. These classes are the "Model" part of the MVC 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.

The model classes created are known as POCO classes, from Plain Old CLR Objects. POCO classes don't have any dependency on EF Core. They only define the properties of the data to be stored in the database.

In this Lab, **model classes are created first**, and using EF Core to create the database.

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

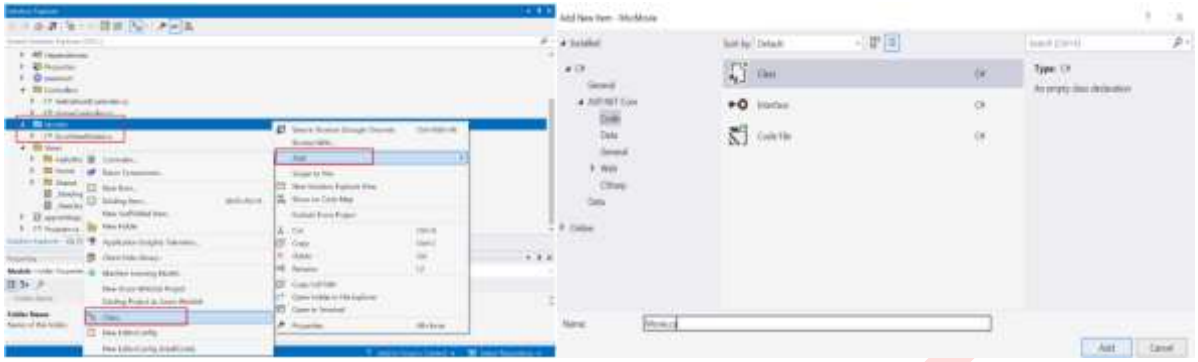


Figure 1.1: Add new model class

Update the Models/Movie.cs file with the following code:

```
using System.ComponentModel.DataAnnotations;
namespace MvcMovie.Models
{
    public class Movie
    {
        public int Id { get; set; }
        public string? Title { get; set; }

        [DataType(DataType.Date)] //<= Using Data Anotations
        public DateTime ReleaseDate { get; set; }
        public string? Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

The Movie class includes an Id field that serves as the primary key required by the database. The ReleaseDate field is decorated with the DataType attribute, specifying that it stores data of type Date. Utilizing this attribute allows for efficient handling of the ReleaseDate data:

– The user isn't required to enter time information in the date field.

– Only the date is displayed, not time information.

DataAnnotations are covered in a later Lab!. The question mark after string indicates that the property is nullable.

1. Add NuGet packages

From the Tools menu, select NuGet Package Manager > Package Manager Console (PMC).

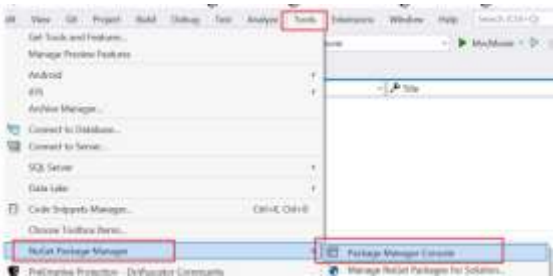
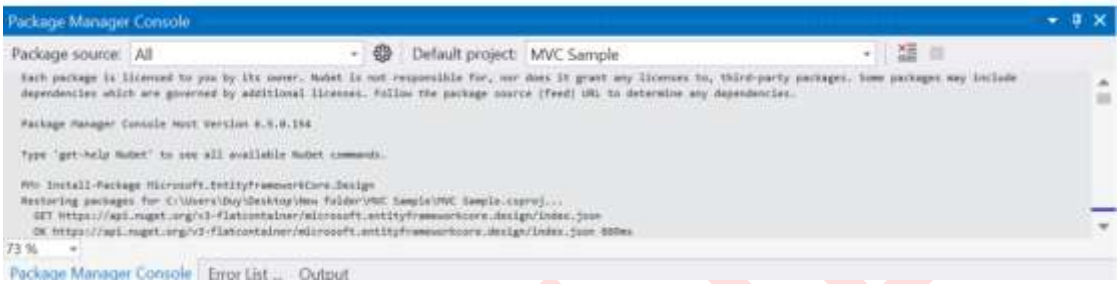


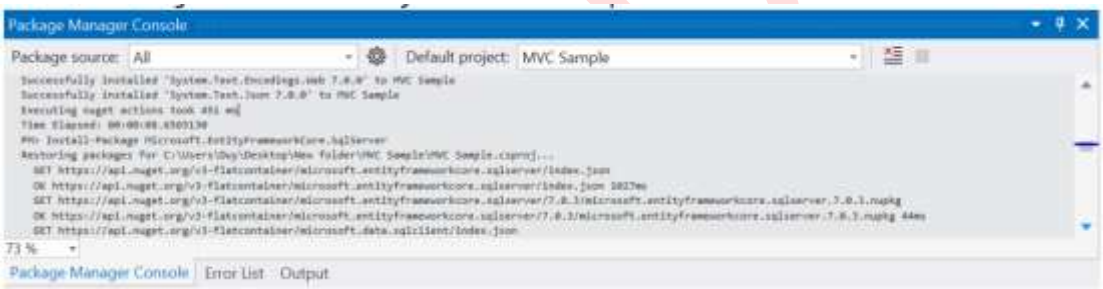
Figure 1.2 1. Add NuGet packages

In the PMC, run the following command:

– ***PM> Install-Package Microsoft.EntityFrameworkCore.Design***



– ***PM> Install-Package Microsoft.EntityFrameworkCore.SqlServer***

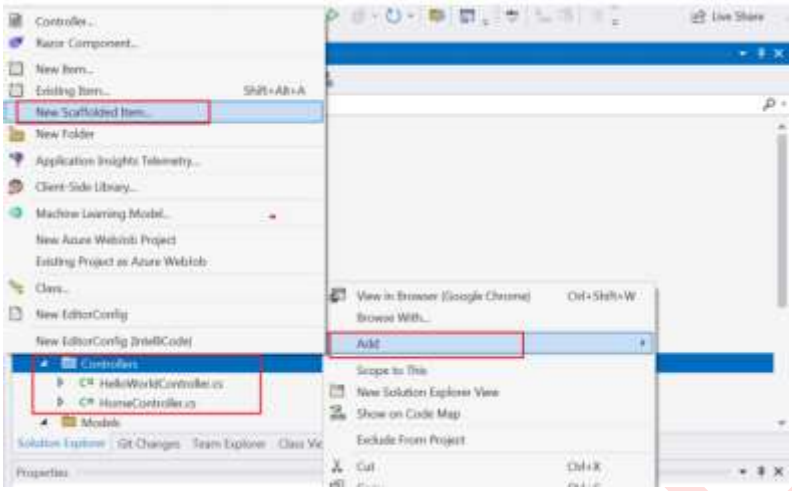


The EF Core SQL Server provider is responsible for handling interactions with SQL Server. When you install the provider package, it automatically includes the EF Core package as a dependency. During the scaffolding step in the Lab, the required utilities will be installed automatically. Make sure to build the project to identify any compiler errors and ensure a successful execution.

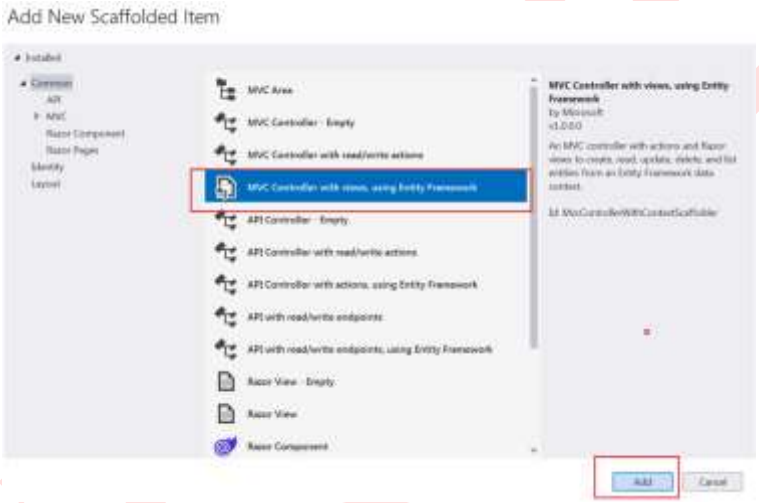
2. Scaffold movie pages

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**

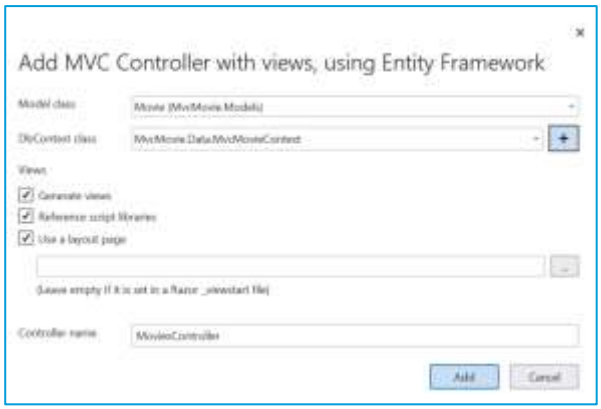


In the Add New Scaffold dialog, select **MVC Controller with views, using Entity Framework > Add**

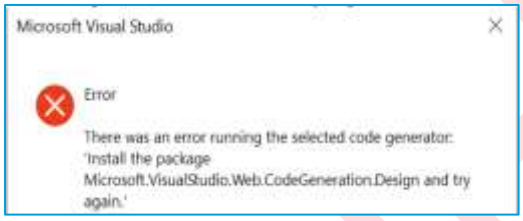


Complete the Add MVC Controller with views, using Entity Framework dialog:

- In the **Model class** drop down, select Movie (MvcMovie.Models).
- In the **DbContext** class row, select the + (plus) sign.
  - ✓ In the **Add Data Context dialog**, the class name MvcMovie.Data.MvcMovieContext is generated.
  - ✓ Select **Add**.
- Views and **Controller** name: Keep the default.
- Select **Add**.



If you get an error message, select Add a second time to try it again



If you get an error message again, Scaffolding updates the following: **Inserts required package references** in the MvcMovie.csproj project file:

```
<ItemGroup>
    <PackageReference Include="Microsoft.EntityFrameworkCore" Version="6.0.14" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="6.0.14" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="6.0.14">
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers;
buildtransitive</IncludeAssets>
    </PackageReference>
    <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design"
Version="6.0.12" />
  </ItemGroup>
```

Try again! (Add MVC Controller with views, using Entity Framework). If not, Add a database connection string to the appsettings.json file:

OR

```
"ConnectionStrings": {
      "ToDoItemsDatabase": "Server=JOHANDRE\\SQL2019; Database=ToDoItems; User=xxx;
Password=xxx; Trusted_Connection=True;MultipleActiveResultSets=true"
  },
```

Scaffolding creates the following:

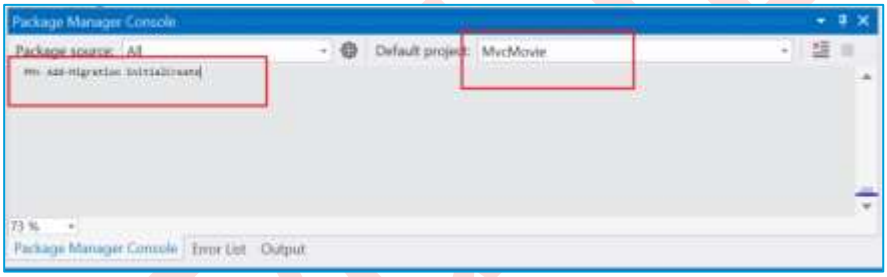
- A movies controller: **Controllers/MoviesController.cs**
- Razor view files for **Create, Delete, Details, Edit**, and **Index** pages:
- **Views/Movies/*.cshtml**
- A database context class: **Data/MvcMovieContext.cs**

The automatic creation of these files and file updates is known as scaffolding. The scaffolded pages can't be used yet because the database doesn't exist!

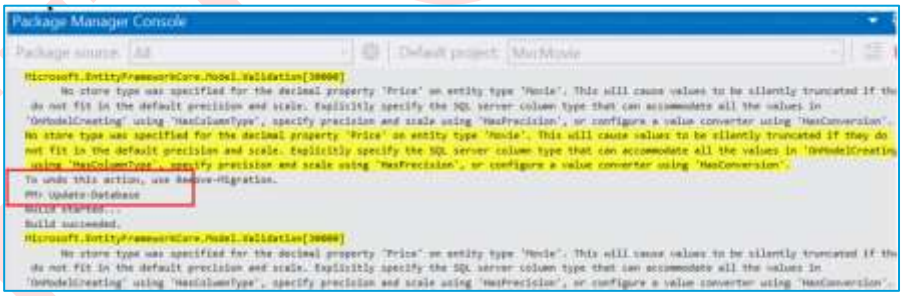
3. Initial migration

From the Tools menu, select **NuGet Package Manager > Package Manager Console** . In the Package Manager Console (PMC), enter the following commands:

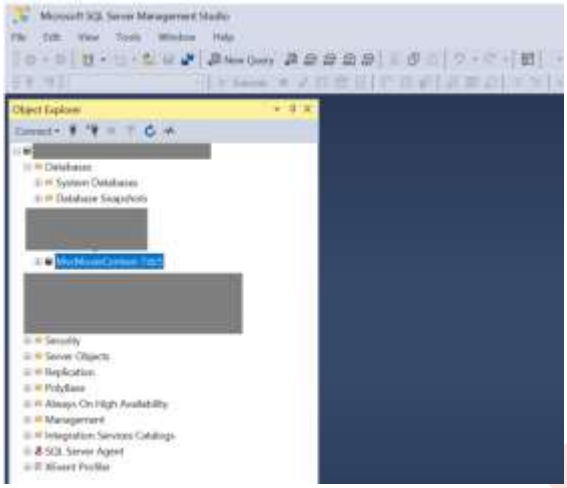
3.1. Add-Migration InitialCreate



3.2. Update-Database



- **Result:**



Add-Migration InitialCreate: The command **"Add-Migration InitialCreate"** generates a migration file named **"{timestamp}_InitialCreate.cs"** within the Migrations directory. The argument "InitialCreate" serves as the name for the migration. While any name can be chosen, it is customary to select a descriptive name for the migration. As this is the initial migration, the generated class includes code for creating the database schema. This schema is built upon the model defined in the **MvcMovieContext** class.

Update-Database: The database is updated to the most recent migration using this command, which was generated by the previous step. By executing this command, the Up method in the Migrations/{time-stamp}_InitialCreate.cs file is triggered, resulting in the creation of the database.

The **Update-Database** command generates the following **warning**:

"No store type was specified for the decimal property 'Price' on entity type 'Movie'. This will cause values to be silently truncated if they do not fit in the default precision and scale. Explicitly specify the SQL server column type that can accommodate all the values in 'OnModelCreating' using 'HasColumnType', specify precision and scale using 'HasPrecision', or configure a value converter using 'HasConversion'." Ignore the preceding warning, it's fixed in a later

3.3. The generated database context class and registration

EF Core enables data access through the utilization of a model, which consists of entity classes and a context object responsible for managing interactions with the database. The context object facilitates data querying and saving operations. Derived from Microsoft.EntityFrameworkCore.DbContext, the database context defines the entities to be incorporated into the data model.

Scaffolding creates the **Data/MvcMovieContext.cs** database context class:

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

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

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

The preceding code creates a **DbSet<Movie>** property that represents the movies in the database.

3.4. Dependency injection

In ASP.NET Core, dependency injection (DI) plays a crucial role. Program.cs is where services, such as the database context, are registered using DI. By injecting these registered services as constructor parameters, components gain access to the necessary dependencies.

In the **Controllers/MoviesController.cs** file, the constructor uses Dependency Injection to inject the **MvcMovieContext** database context into the controller. The database context is used in each of the CRUD methods in the controller.

```
public MoviesController(MvcMovieContext context)
{
      _context = context;
}
```

Scaffolding generated the following code in **Program.cs:**

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using MvcMovie.Data;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<MvcMovieContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("MvcMovieContext") ?? throw
new InvalidOperationException("Connection string 'MvcMovieContext' not found.")));
```

3.5. The generated database connection string

Scaffolding added a connection string to the **appsettings.json** file:

```
{
  "ConnectionStrings": {
    "MvcMovieContext": "Server=(localdb)\\mssqllocaldb;Database=MvcMovieContext-
7dc5;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

For local development, the ASP.NET Core configuration system reads the ConnectionString key from the **appsettings.json** file.

3.6. The InitialCreate class

Examine the Migrations/{timestamp}_InitialCreate.cs migration file:

```
using System;
using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace MvcMovie.Migrations
{
    public partial class InitialCreate : Migration
    {
        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);
                });
        }

        protected override void Down(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.DropTable(
                name: "Movie");
        }
    }
}
```

In the preceding code:

- InitialCreate.Up creates the Movie table and configures Id as the primary key.
- InitialCreate.Down reverts the schema changes made by the Up migration.

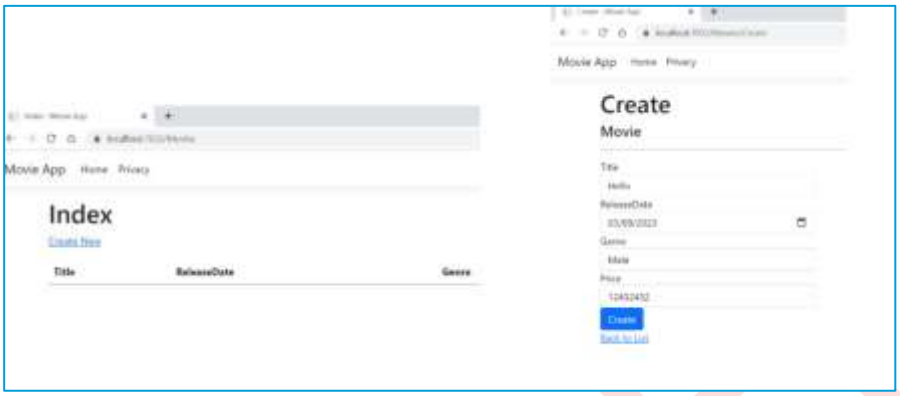
4. Dependency injection in the controller

Open the **Controllers/MoviesController.cs** file and examine the constructor:

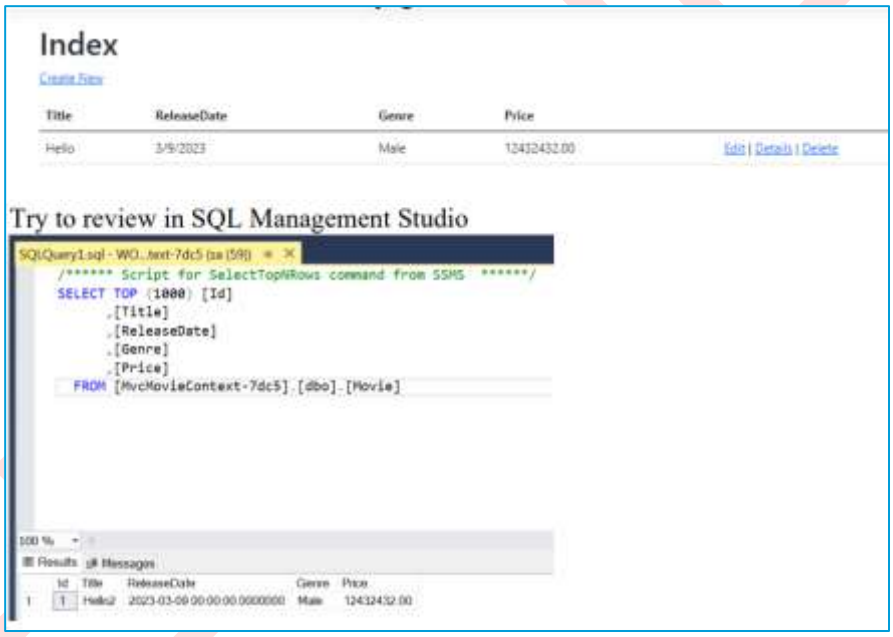
```
public class MoviesController : Controller
{
    private readonly MvcMovieContext _context;

    public MoviesController(MvcMovieContext context)
    {
        _context = context;
    }
}
```

The constructor uses **Dependency Injection** to inject the database context **(MvcMovieContext)** into the controller. The database context is used in each of the CRUD methods in the controller.

– Test the **Create** page. Enter and submit data:



– Test the **Edit**, **Details**, and **Delete** pages:



Strongly typed models and the **@model** directive:

During the earlier part of this Lab, you were introduced to the concept of passing data or objects from a controller to a view using the ViewData dictionary. This dictionary acts as a dynamic object, providing a flexible way to convey information to a view. In addition to this, the MVC framework offers a more robust approach by allowing the passing of strongly typed model objects to views. By using this approach, compile-time code checking becomes possible, enhancing the reliability of the system. This mechanism was

utilized in the MoviesController class and associated views through the scaffolding process.

Examine the generated Details method in the Controllers/MoviesController.cs file:

```
// GET: Movies/Details/5
        public async Task<IActionResult> Details(int? id)
        {
            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);
        }
```

The **id** parameter is generally passed as route data. For example, **https://localhost:5001/movies/details/1** sets:

- The controller to the movies controller, the first URL segment.
- The action to details, the second URL segment.
- The id to 1, the last URL segment.

The **id** can be passed in with a query string, as in the following example: **https://localhost:5001/movies/details?id=1**

The **id** parameter is defined as a **nullable type (int?)** in cases when the id value isn't provided. A lambda expression is passed in to the **FirstOrDefaultAsync** method to select movie entities that match the route data or query string value:

```
var movie = await _context.Movie
                .FirstOrDefaultAsync(m => m.Id == id);
```

If a movie is found, an instance of the Movie model is passed to the Details view: return View(movie);

Examine the contents of the **Views/Movies/Details.cshtml** file:

```
@model MvcMovie.Models.Movie

@{
    ViewData["Title"] = "Details";
}

<h1>Details</h1>

<div>
    <h4>Movie</h4>
    <hr />
    <dl class="row">
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Title)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Title)
```

---The end---

The **@model** statement at the top of the view file specifies the type of object that the view expects. When the movie controller was created, the following **@model** statement was included: **@model MvcMovie.Models.Movie**

This **@model** directive allows access to the movie that the controller passed to the view. The **Model** object is strongly typed. For example, in the **Details.cshtml** view, the code passes each movie field to the **DisplayNameFor** and DisplayFor HTML Helpers with the strongly typed **Model** object. The **Create** and **Edit** methods and views also pass a Movie model object.

Examine the **Index.cshtml** view and the **Index** method in the Movies controller. Notice how the code creates a List object when it calls the **View** method. The code passes this **Movies** list from the **Index** action method to the view:

```
// GET: Movies
 public async Task<IActionResult> Index()
 {
       return View(await _context.Movie.ToListAsync());
 }
```

When the movies controller was created, scaffolding included the following **@model** statement at the top of the **Index.cshtml** file:

@model IEnumerable<MvcMovie.Models.Movie>

The **@model** directive allows access to the list of movies that the controller passed to the view by using a Model object that's strongly typed. For example, in the **Index.cshtml** view, the code loops through the movies with a **foreach** statement over the strongly typed **Model** object: @model IEnumerable<MvcMovie.Models.Movie>

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

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.Id">Details</a> |
                <a asp-action="Delete" asp-route-id="@item.Id">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

Because the **Model** object is strongly typed as an **IEnumerable<Movie>** object, each item in the loop is typed as **Movie**. Among other benefits, the compiler validates the types used in the code. Run project and delete all record! *(Delete all the records in the database. You can do this with the delete links in the browser or from SQL Management studio).*

5. Seed the database

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

```
using Microsoft.EntityFrameworkCore;
using MvcMovie.Data;

namespace MvcMovie.Models
{
    public static class SeedData
    {
        public static void Initialize(IServiceProvider serviceProvider)
        {
            using (var context = new MvcMovieContext(
                serviceProvider.GetRequiredService<
                    DbContextOptions<MvcMovieContext>>()))
            {
                // Look for any movies.
                if (context.Movie.Any())
                {
                    return;   // DB has been seeded
                }

                context.Movie.AddRange(
                    new Movie
                    {
                        Title = "When Harry Met Sally",
                        ReleaseDate = DateTime.Parse("1989-2-12"),
                        Genre = "Romantic Comedy",
                        Price = 7.99M
                    },

                    new Movie
                    {
                        Title = "Ghostbusters ",
                        ReleaseDate = DateTime.Parse("1984-3-13"),
                        Genre = "Comedy",
                        Price = 8.99M
                    },

                    new Movie
                    {
                        Title = "Ghostbusters 2",
                        ReleaseDate = DateTime.Parse("1986-2-23"),
                        Genre = "Comedy",
                        Price = 9.99M
                    },

                    new Movie
                    {
                        Title = "Rio Bravo",
                        ReleaseDate = DateTime.Parse("1959-4-15"),
                        Genre = "Western",
                        Price = 3.99M
                    }
                   );
                   context.SaveChanges();
            }
        }
    }
}
```

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

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

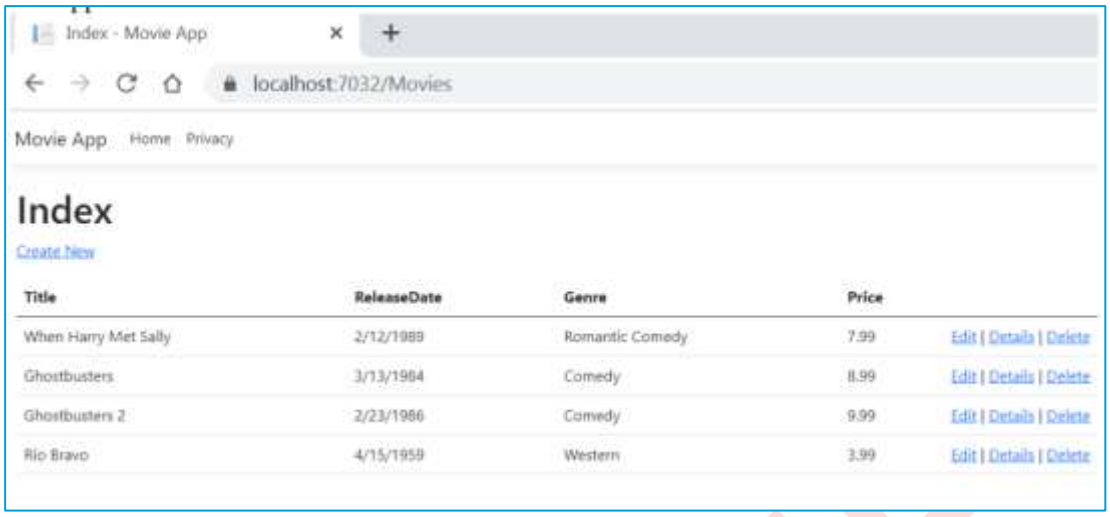
6. Add the seed initializer

Replace the contents of Program.cs with the following code. The new code is highlighted.

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using MvcMovie.Data;
using MvcMovie.Models;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<MvcMovieContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("MvcMovieContext") ?? throw
new InvalidOperationException("Connection string 'MvcMovieContext' not found.")));

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

var app = builder.Build();

using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;

    SeedData.Initialize(services);
}
// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    // The default HSTS value is 30 days. You may want to change this for production
scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

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

app.UseRouting();

app.UseAuthorization();

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

app.Run();
```

Test the app. Force the app to initialize, calling the code in the **Program.cs file**, so the seed method runs. To force initialization, close the command prompt window that Visual Studio opened before, and restart by pressing **Ctrl+F5.**

The app shows the seeded data:



We have a good start to the movie app, but the presentation isn't ideal, for example, ReleaseDate should be two words (Release Date). Open the Models/Movie.cs file and add the highlighted lines shown below:

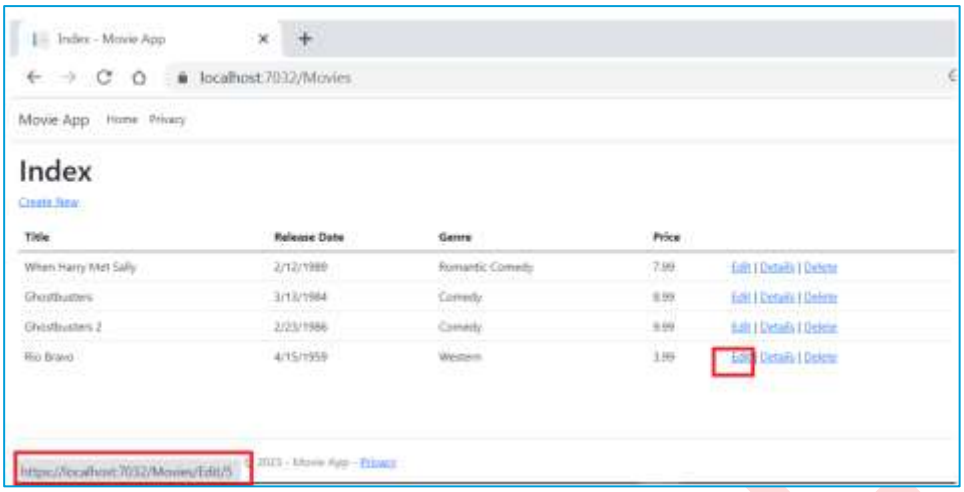
```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
namespace MvcMovie.Models
{
    public class Movie
    {
        public int Id { get; set; }
        public string? Title { get; set; }

        [Display(Name = "Release Date")]
        [DataType(DataType.Date)] //<= Using Data Anotations
        public DateTime ReleaseDate { get; set; }
        public string? Genre { get; set; }

        [Column(TypeName = "decimal(18, 2)")]
        public decimal Price { get; set; }
    }
}
```

The **Display** attribute specifies what to display for the name of a field (in this case "Release Date" instead of "ReleaseDate").

The **DataType** attribute specifies the type of the data (Date), so the time information stored in the field isn't displayed.

The **[Column(TypeName = "decimal(18, 2)")]** data annotation is required so Entity Framework Core can correctly map Price to currency in the database.

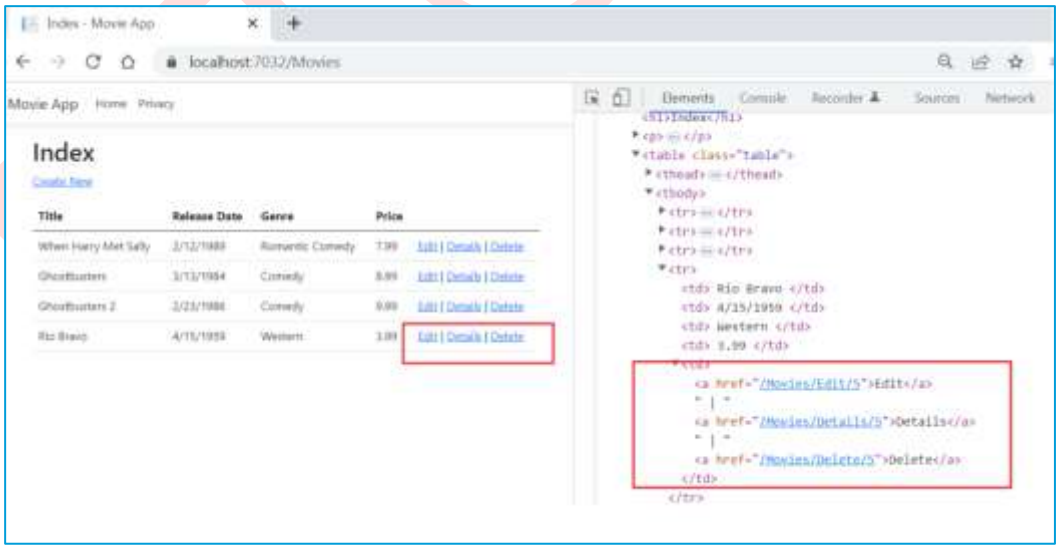
Run project and Browse to the Movies controller and hold the mouse pointer over an Edit link to see the target URL.



The **Edit**, **Details**, and **Delete** links are generated by the **Core MVC Anchor Tag Helper** in the **Views/Movies/Index.cshtml** file.

```
<td>
    <a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
    <a asp-action="Details" asp-route-id="@item.Id">Details</a> |
    <a asp-action="Delete" asp-route-id="@item.Id">Delete</a>
</td>
```

Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. In the code above, the **AnchorTagHelper** dynamically generates the HTML href attribute value from the controller action method and route id. You use **View Source** from your favorite browser or use the developer tools to examine the generated markup. A portion of the generated HTML is shown below:



Recall the format for routing set in the **Program.cs file**

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

ASP.NET Core translates **https://localhost:5001/Movies/Edit/4** into a request to the **Edit** action method of the **Movies** controller with the parameter **Id** of 4. (Controller methods are also known as action methods.)

Tag Helpers are a popular feature in ASP.NET Core. Open the **Movies** controller and examine the two **Edit** action methods. The following code shows the **HTTP GET Edit** method, which fetches the movie and populates the edit form generated by the **Edit.cshtml** Razor file.

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

            var movie = await _context.Movie.FindAsync(id);
            if (movie == null)
            {
                return NotFound();
            }
            return View(movie);
        }
```

The following code shows the HTTP POST Edit method, which processes the posted movie values: Notice the second **Edit** action method is preceded by the **[HttpPost]** attribute.

```
// POST: Movies/Edit/5
        // To protect from overposting attacks, enable the specific properties you want to
bind to.
        // For more details, see http://go.microsoft.com/fwlink/?LinkId=317598.
        [HttpPost]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult> Edit(int id,
[Bind("Id,Title,ReleaseDate,Genre,Price")] Movie movie)
        {
            if (id != movie.Id)
            {
                return NotFound();
            }

            if (ModelState.IsValid)
            {
                try
                {
                    _context.Update(movie);
                    await _context.SaveChangesAsync();
                }
                catch (DbUpdateConcurrencyException)
                {
                    if (!MovieExists(movie.Id))
                    {
                        return NotFound();
                    }
                    else
                    {
```

```
                throw;
            }
        }
        return RedirectToAction(nameof(Index));
    }
    return View(movie);
}
```

The **[Bind]** attribute is one way to protect against over-posting. You should only include properties in the **[Bind]** attribute that you want to change. ViewModels provide an alternative approach to prevent over-posting. (https://rachelappel.com/2014/09/02/use-viewmodels-to-manage-data-amporganize-code-in-asp-net-mvc-applications/)

The **HttpPost** attribute specifies that this **Edit** method can be invoked only for **POST** requests. You could apply the [**HttpGet**] attribute to the first edit method, but that's not necessary because [**HttpGet**] is the default.

The **ValidateAntiForgeryToken** attribute is used to **prevent forgery of a request** and is paired up with an anti-forgery token generated in the edit view file **(Views/Movies/Edit.cshtml)**. The edit view file generates the anti-forgery token with the Form Tag Helper.

<form asp-action="Edit">

The Form Tag Helper generates a hidden anti-forgery token that must match the **[ValidateAntiForgeryToken]** generated anti-forgery token in the **Edit** method of the Movies controller.

The **HttpGet Edit** method takes the movie **ID** parameter, looks up the movie using the Entity Framework **FindAsync** method, and returns the selected movie to the Edit view. If a movie cannot be found, **NotFound** (HTTP 404) is returned.

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

    var movie = await _context.Movie.FindAsync(id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

When the scaffolding system created the Edit view, it examined the **Movie** class and created code to render **<label>** and **<input>** elements for each property of the class. The following example shows the Edit view that was generated by the Visual Studio scaffolding system:

```
@model MvcMovie.Models.Movie

@{
    ViewData["Title"] = "Edit";
}

<h1>Edit</h1>

<h4>Movie</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Edit">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Id" />
            <div class="form-group">
                <label asp-for="Title" class="control-label"></label>
                <input asp-for="Title" class="form-control" />
                <span asp-validation-for="Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="ReleaseDate" class="control-label"></label>
                <input asp-for="ReleaseDate" class="form-control" />
                <span asp-validation-for="ReleaseDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Genre" class="control-label"></label>
                <input asp-for="Genre" class="form-control" />
                <span asp-validation-for="Genre" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Price" class="control-label"></label>
                <input asp-for="Price" class="form-control" />
                <span asp-validation-for="Price" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Notice how the view template has a **@model MvcMovie.Models.Movie** statement at the top of the file. **@model MvcMovie.Models.Movie** specifies that the view expects the model for the view template to be of type **Movie.**

The scaffolded code uses several Tag Helper methods to streamline the HTML markup. The **Label Tag Helper** displays the name of the field ("Title", "ReleaseDate", "Genre", or "Price"). The Input Tag Helper renders an HTML **<input>** element. The **Validation Tag Helper** displays any validation messages associated with that property.

Run the application and navigate to the **/Movies** URL. Click an **Edit** link. In the browser, view the source for the page. The generated HTML for the **<form>** element is shown below.

```
<form action="/Movies/Edit/5" method="post" novalidate="novalidate">

        <input type="hidden" data-val="true" data-val-required="The Id field is required." id="Id" name="Id"
value="5">
        <div class="form-group">
            <label class="control-label" for="Title">Title</label>
            <input class="form-control" type="text" id="Title" name="Title" value="Rio Bravo">
            <span    class="text-danger    field-validation-valid"    data-valmsg-for="Title"    data-valmsg-
replace="true"></span>
        </div>
        <div class="form-group">
            <label class="control-label" for="ReleaseDate">Release Date</label>
            <input class="form-control" type="date" data-val="true" data-val-required="The Release Date field
is required." id="ReleaseDate" name="ReleaseDate" value="1959-04-15">
            <span  class="text-danger  field-validation-valid"  data-valmsg-for="ReleaseDate"  data-valmsg-
replace="true"></span>
        </div>
        <div class="form-group">
            <label class="control-label" for="Genre">Genre</label>
            <input class="form-control" type="text" id="Genre" name="Genre" value="Western">
            <span    class="text-danger    field-validation-valid"    data-valmsg-for="Genre"    data-valmsg-
replace="true"></span>
        </div>
        <div class="form-group">
            <label class="control-label" for="Price">Price</label>
            <input class="form-control" type="text" data-val="true" data-val-number="The field Price must be
a number." data-val-required="The Price field is required." id="Price" name="Price" value="3.99">
            <span    class="text-danger    field-validation-valid"    data-valmsg-for="Price"    data-valmsg-
replace="true"></span>
        </div>
        <div class="form-group">
            <input type="submit" value="Save" class="btn btn-primary">
        </div>
    <input                          name="__RequestVerificationToken"                          type="hidden"
value="CfDJ8FntYJHbTUBBoVqXehbEVqv8HO9u36tUAg6A5yOa7P30Bdkevs2tX3souXPG5f7no_x8xgygWPmmmfcd4ktJ7ZSFxV6mDDP4_mIY
n5wtYq2Su4yNB9iCJmLX5UsS5EHmabiSLsRe9vzFkYjqhr69lvo"></form>
```

The <**input**> elements are in an **HTML <form>** element whose **action** attribute is set to post to the **/Movies/Edit/id** URL. The form data will be posted to the server when the **Save** button is clicked. The last line before the closing **</form>** element shows the hidden "Cross-site request forgery (also known as XSRF or CSRF)" token generated by the Form Tag Helper.

7. Processing the POST Request

The following listing shows the **[HttpPost]** version of the Edit action method.

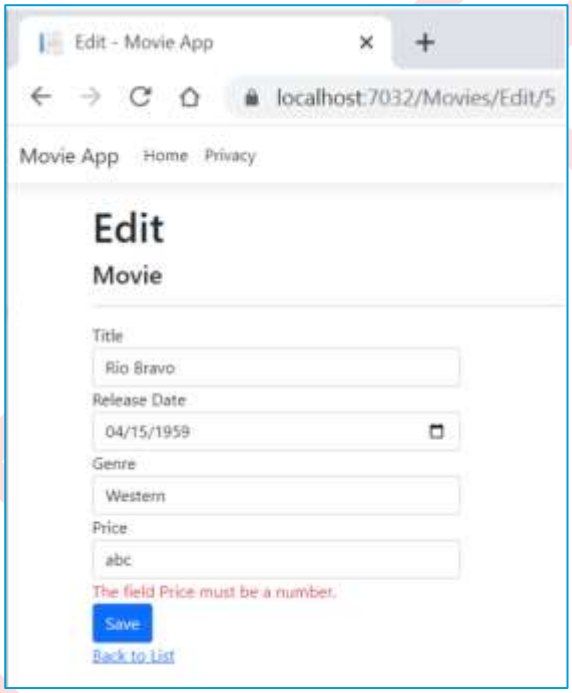
```
// POST: Movies/Edit/5
. . .
    return View(movie);
}
```

The **[ValidateAntiForgeryToken]** attribute validates the hidden XSRF token generated by the anti-forgery token generator in the Form Tag Helper. The **model binding** system takes the posted form values and creates a **Movie** object that's passed as the movie parameter. The **ModelState.IsValid** property verifies that the data submitted in the form can be used to modify (edit or update) a **Movie** object. If the data is valid, it's saved. The updated (edited) movie data is saved to the database by calling the **SaveChangesAsync** method of database context. After saving the data, the code redirects

the user to the Index action method of the MoviesController class, which displays the movie collection, including the changes just made.

Before the form is posted to the server, client-side validation checks any validation rules on the fields. If there are any validation errors, an error message is displayed and the form isn't posted. If JavaScript is disabled, you won't have client-side validation but the server will detect the posted values that are not valid, and the form values will be redisplayed with error messages.

Later in the Lab we examine **Model Validation** in more detail. The **Validation Tag Helper** in the **Views/Movies/Edit.cshtml** view template takes care of displaying appropriate error messages.



All the **HttpGet** methods in the movie controller follow a similar pattern. They get a movie object (or list of objects, in the case of **Index**), and pass the object (model) to the view. The Create method passes an empty movie object to the **Create** view.

All the methods that create, edit, delete, or otherwise modify data do so in the [**HttpPost**] overload of the method.

Modifying data in an **HTTP GET** method is a security risk. Modifying data in an **HTTP GET** method also violates HTTP best practices and the architectural **REST** pattern, which specifies that GET requests shouldn't change the state of your application.

In other words, performing a GET operation should be a safe operation that has no side effects and doesn't modify your persisted data.

SUBMIT YOUR CODE TO GIT!

- - - END LAB 03 - - -

LAB 4
ASP.NET Core MVC (cont)

I. TARGET

- ✓ Adding search feature to project.
- ✓ Adding new data fields.
- ✓ Add validation rules to model.
- ✓ Examine the Details and Delete methods

II. REFERENCES

- ✓ Linq in C#, https://viblo.asia/p/linq-trong-c-6J3ZgpgxlmB
- ✓ Using LINQ(Language Integrated Query) in .NET CORE, https://www.youtube.com/watch?v=Vu_jFsPOJFw&list=PLwJr0JSP7i8BERdErX9Ird67xTflZkxb-&index=31
- ✓ Professional ASP.NET MVC 5 by Jon Galloway, Brad Wilson, K. Scott Allen, David Matson

III. REQUIREMENTS

- ✓ Add search function to ASP.NET Core Razor Pages
- ✓ Update database using migration

IV. IN LAB

In this Lab, you add search capability to the Index action method that lets you search movies by genre or name. Update the Index method found inside Controllers/MoviesController.cs with the following code:

```
// GET: Movies
        //public async Task<IActionResult> Index()
        //{
        //      return View(await _context.Movie.ToListAsync());
        //}
        public async Task<IActionResult> Index(string searchString)
        {
            var movies = from m in _context.Movie
                         select m;

            if (!String.IsNullOrEmpty(searchString))
            {
                movies = movies.Where(s => s.Title!.Contains(searchString));
            }

            return View(await movies.ToListAsync());
        }
```

The first line of the Index action method creates a LINQ query to select the movies:

```
var movies = from m in _context.Movie
             select m;
```

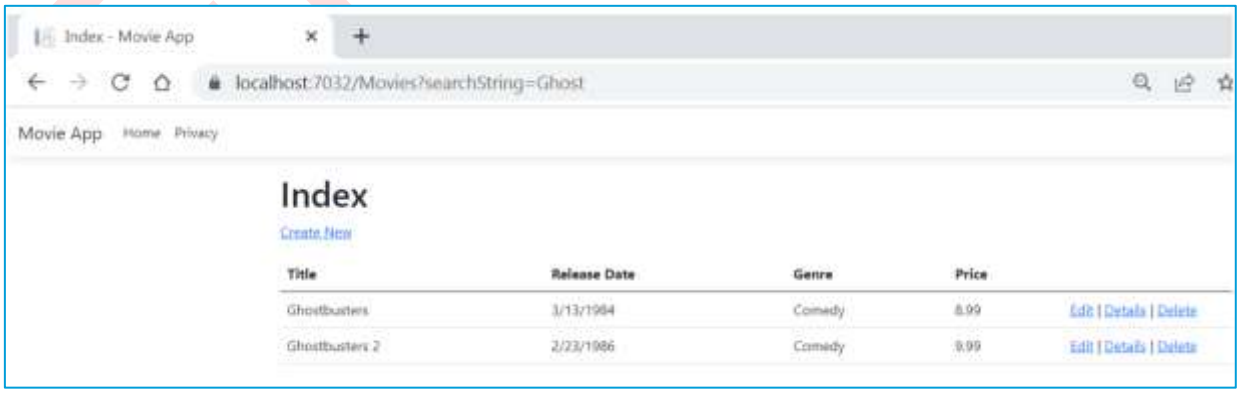
The query is only defined at this point, it has not been run against the database. If the searchString parameter contains a string, the movies query is modified to filter on the value of the search string:

```
if (!String.IsNullOrEmpty(searchString))
            {
                movies = movies.Where(s => s.Title!.Contains(searchString));
            }
```

The **s => s.Title!.Contains(searchString)** code above is a **Lambda Expression**. Lambdas are used in method-based LINQ queries as arguments to standard query operator methods such as the **Where** method or Contains (used in the code above). LINQ queries are not executed when they're defined or when they're modified by calling a method such as **Where**, **Contains**, or **OrderBy**. Rather, query execution is deferred. That means that the evaluation of an expression is delayed until its realized value is actually iterated over or the ToListAsync method is called.

The Contains method is run on the database, not in the c# code shown above. The case sensitivity on the query depends on the database and the collation. On SQL Server, **Contains** maps to **SQL LIKE**, which is case insensitive. In SQLite, with the default collation, it's case sensitive.

Navigate to **/Movies/Index.** Append a query string such as **?searchString=Ghost** to the URL. The filtered movies are displayed.



If you change the signature of the **Index** method to have a parameter named **id**, the **id** parameter will match the optional {id} placeholder for the default routes set in **Program.cs**

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

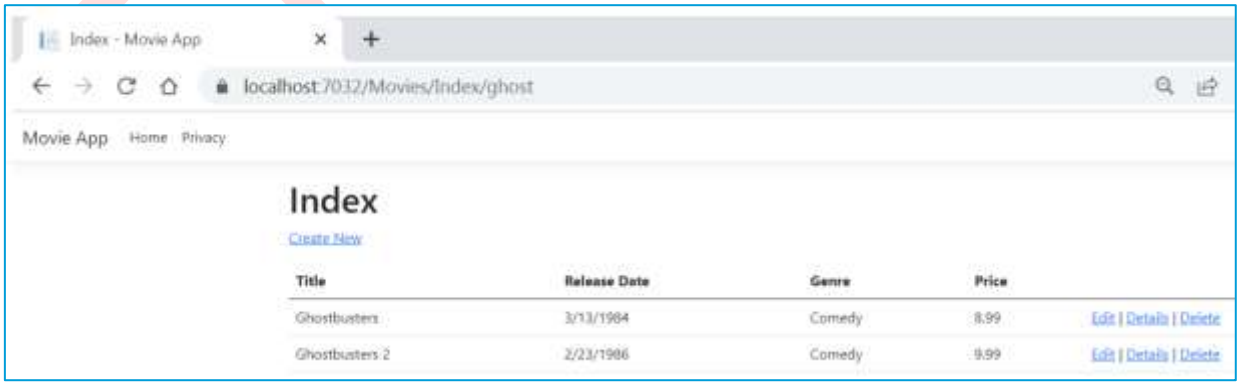
Change the parameter to id and change all occurrences of **searchString** to **id**. The previous Index method:

```
public async Task<IActionResult> Index(string searchString)
        {
            var movies = from m in _context.Movie
                         select m;

            if (!String.IsNullOrEmpty(searchString))
            {
                movies = movies.Where(s => s.Title!.Contains(searchString));
            }

            return View(await movies.ToListAsync());
        }
```

The updated Index method with id parameter:

```
public async Task<IActionResult> Index(string id)
        {
            var movies = from m in _context.Movie
                         select m;

            if (!String.IsNullOrEmpty(id))
            {
                movies = movies.Where(s => s.Title!.Contains(id));
            }

            return View(await movies.ToListAsync());
        }
```

You can now pass the search title as route data (a URL segment) instead of as a query string value:



However, you can't expect users to modify the URL every time they want to search for a movie. So now you'll add UI elements to help them filter movies. If you changed

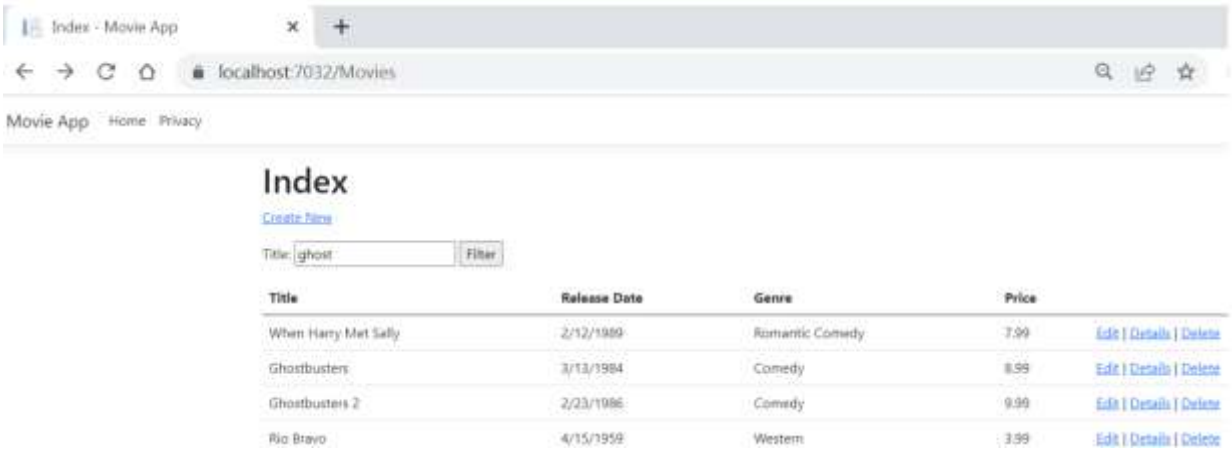
the signature of the Index method to test how to pass the route-bound **ID** parameter, change it back so that it takes a parameter named **searchString**:

```
public async Task<IActionResult> Index(string searchString)
        {
            var movies = from m in _context.Movie
                         select m;

            if (!String.IsNullOrEmpty(searchString))
            {
                movies = movies.Where(s => s.Title!.Contains(searchString));
            }

            return View(await movies.ToListAsync());
        }
```

Open the **Views/Movies/Index.cshtml** file, and add the <**form**> markup highlighted below:

```
@model IEnumerable<MvcMovie.Models.Movie>

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

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
<form asp-controller="Movies" asp-action="Index">
    <p>
        Title: <input type="text" name="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>
```

The HTML <**form**> tag uses the Form Tag Helper, so when you submit the form, the filter string is posted to the **Index** action of the movies controller. Save your changes and then test the filter.

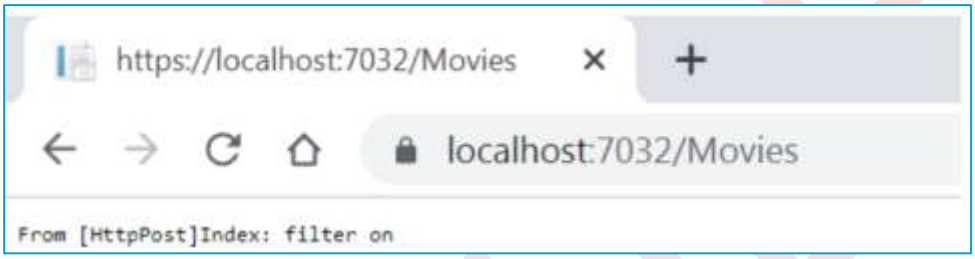


There's no [**HttpPost**] overload of the **Index** method as you might expect. You don't need it, because the method isn't changing the state of the app, just filtering data. You could add the following **[HttpPost] Index** method.

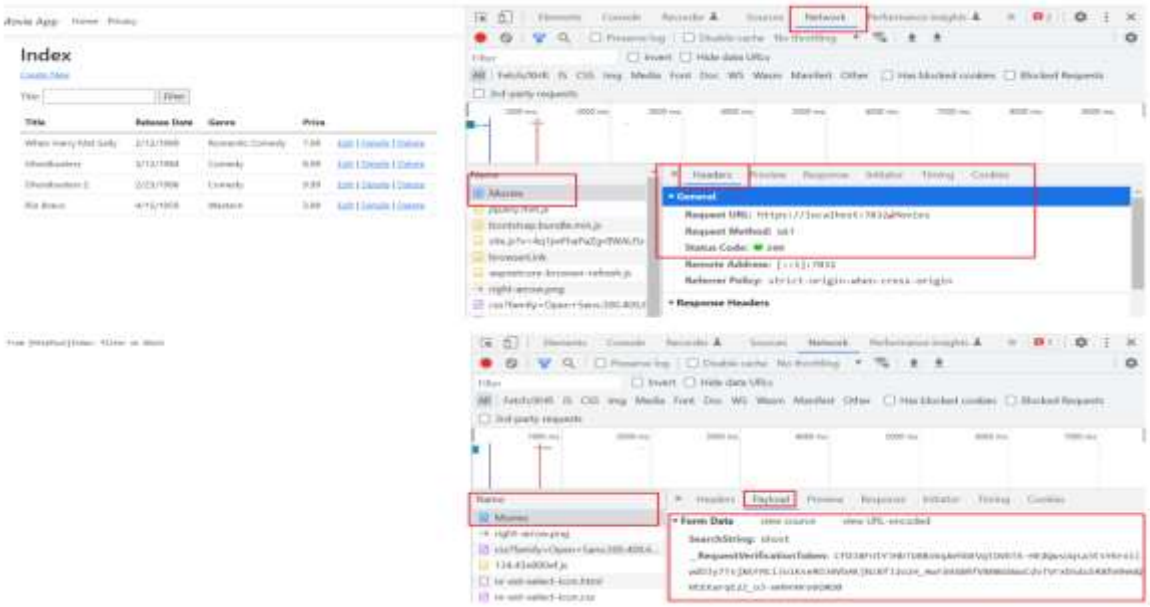
```
[HttpPost]
 public string Index(string searchString, bool notUsed)
 {
     return "From [HttpPost]Index: filter on " + searchString;
 }
```

The **notUsed** parameter is used to create an overload for the Index method. We'll talk about that later in the Lab.

If you add this method, the action invoker would match the **[HttpPost] Index** method, and the **[HttpPost] Index** method would run as shown in the image below. (Type 'Ghost',Click Filter button).



However, even if you add this [**HttpPost**] version of the **Index** method, there's a limitation in how this has all been implemented. Imagine that you want to bookmark a particular search or you want to send a link to friends that they can click in order to see the same filtered list of movies. Notice that the URL for the HTTP POST request is the same as the URL for the GET **request (localhost:{PORT}/Movies/Index)** -- there's no search information in the URL. The search string information is sent to the server as a form field value. You can verify that with the browser Developer tools or the excellent Fiddler tool. The image below shows the Chrome browser Developer tools:

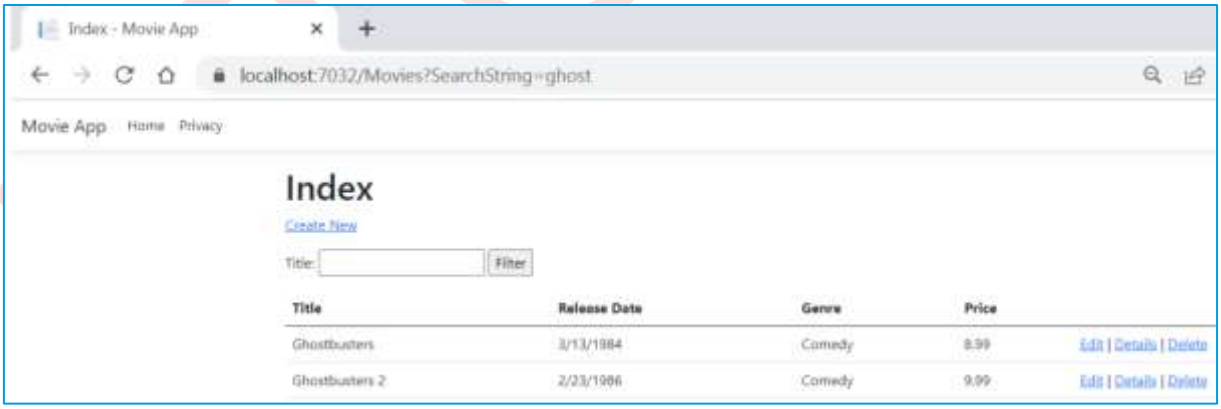


You can see the search parameter and XSRF token (Prevent Cross-Site Request Forgery (XSRF/CSRF)) in the request body. Note, as mentioned in the previous Lab, the Form Tag Helper generates an XSRF anti-forgery token. We're not modifying data, so we don't need to validate the token in the controller method.

Because the search parameter is in the request body and not the URL, you can't capture that search information to bookmark or share with others. Fix this by specifying the request should be HTTP GET found in the Views/Movies/Index.cshtml file.

```
@model IEnumerable<MvcMovie.Models.Movie>

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

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
@*<form asp-controller="Movies" asp-action="Index">*@
<form asp-controller="Movies" asp-action="Index" method="get">
    <p>
        Title: <input type="text" name="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>
<table class="table">
```

Now when you submit a search, the URL contains the search query string. Searching will also go to the **HttpGet Index** action method, even if you have a **HttpPost Index** method.



The following markup shows the change to the form tag:

```
<form asp-controller="Movies" asp-action="Index" method="get">
```

1. Add Search by genre

Add the following MovieGenreViewModel class to the Models folder

```
using Microsoft.AspNetCore.Mvc.Rendering;

namespace MvcMovie.Models
{
    public class MovieGenreViewModel
    {
        public List<Movie>? Movies { get; set; }
        public SelectList? Genres { get; set; }
        public string? MovieGenre { get; set; }
        public string? SearchString { get; set; }
    }
}
```

The movie-genre view model will contain:

- A list of movies.
- A **SelectList** containing the list of genres. This allows the user to select a genre from the list.
- **MovieGenre**, which contains the selected genre.
- **SearchString**, which contains the text users enter in the search text box.

Replace the Index method in MoviesController.cs with the following code:

```
// GET: Movies
 public async Task<IActionResult> Index(string movieGenre, string searchString)
 {
     // Use LINQ to get list of genres.
     IQueryable<string> genreQuery = from m in _context.Movie
                                     orderby m.Genre
                                     select m.Genre;
     var movies = from m in _context.Movie
                  select m;

     if (!string.IsNullOrEmpty(searchString))
     {
         movies = movies.Where(s => s.Title!.Contains(searchString));
     }

     if (!string.IsNullOrEmpty(movieGenre))
     {
         movies = movies.Where(x => x.Genre == movieGenre);
     }
     var movieGenreVM = new MovieGenreViewModel
     {
         Genres = new SelectList(await genreQuery.Distinct().ToListAsync()),
         Movies = await movies.ToListAsync()
     };

     return View(movieGenreVM);
 }
```

The following code is a LINQ query that retrieves all the genres from the database

```
// Use LINQ to get list of genres.
IQueryable<string> genreQuery = from m in _context.Movie
                                orderby m.Genre
                                select m.Genre;
```

The **SelectList** of genres is created by projecting the distinct genres (we don't want our select list to have duplicate genres). When the user searches for the item, the search value is retained in the search box

2. Add search by genre to the Index view

Update Index.cshtml found in Views/Movies/ as follows:

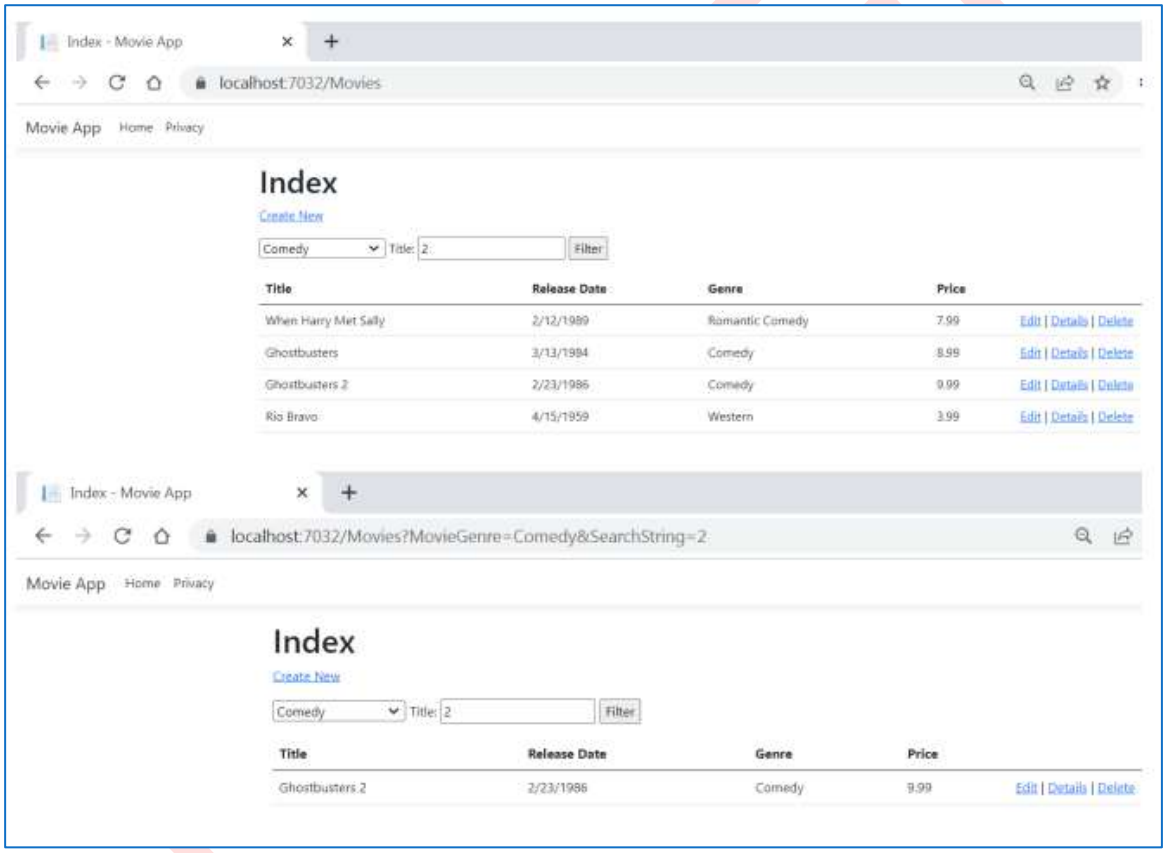
```
@*@model IEnumerable<MvcMovie.Models.Movie>*@
@model MvcMovie.Models.MovieGenreViewModel
@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
@*<form asp-controller="Movies" asp-action="Index">*@
<form asp-controller="Movies" asp-action="Index" method="get">
    <p>
        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>

        Title: <input type="text" asp-for="SearchString" />
      @* Title: <input type="text" name="SearchString" />*@
        <input type="submit" value="Filter" />
    </p>
</form>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movies[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies[0].Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Movies)
        {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.Id">Details</a> |
                <a asp-action="Delete" asp-route-id="@item.Id">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

Examine the lambda expression used in the following HTML Helper:

@Html.DisplayNameFor(model => model.Movies[0].Title)

In the provided passage, the DisplayNameFor HTML Helper method analyzes the Title property within a lambda expression to ascertain the display name. This examination is done without executing the lambda expression, which prevents any access violation when model, model.Movies, or model.Movies[0] contain null or empty values. Once the lambda expression is eventually evaluated, for instance, with **@Html.DisplayFor(modelItem => item.Title),** the actual property values of the model are computed and displayed.

Test the app by searching by genre, by movie title, and by both:



3. Add a Rating Property to the Movie Model

In this section Entity Framework Code First Migrations is used to:

- Add a new field to the model.
- Migrate the new field to the database.

When EF Code First is used to automatically create a database, Code First:

- Adds a table to the database to track the schema of the database.

– Verifies the database is in sync with the model classes it was generated from. If they aren't in sync, EF throws an exception. This makes it easier to find inconsistent database/code issues.

- Add a **Rating** property to **Models/Movie.cs**

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
namespace MvcMovie.Models
{
    public class Movie
    {
        public int Id { get; set; }
        public string? Title { get; set; }

        [Display(Name = "Release Date")]
        [DataType(DataType.Date)] //<= Using Data Anotations
        public DateTime ReleaseDate { get; set; }
        public string? Genre { get; set; }

        [Column(TypeName = "decimal(18, 2)")]
        public decimal Price { get; set; }
        public string? Rating { get; set; }
    }
}
```

Build the app:

– Using the **Ctrl+Shift+B** (Visual studio 2022)

– Using the dotnet build (Visual studio code)

Because you've added a new field to the **Movie** class, you need to update the property binding list so this new property will be included. In **MoviesController.cs,** update the **[Bind]** attribute for both the **Create** and **Edit** action methods to include the **Rating** property:

```
[HttpPost]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult>
Create([Bind("Id,Title,ReleaseDate,Genre,Price,Rating")] Movie movie)
        {

. . .

[HttpPost]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult> Edit(int id,
[Bind("Id,Title,ReleaseDate,Genre,Price,Rating")] Movie movie)
        {

. . .
```

Update the view templates in order to display, create, and edit the new **Rating** property in the browser view.

- Edit the **/Views/Movies/Index.cshtml** file and add a **Rating** field:

```
@*@model IEnumerable<MvcMovie.Models.Movie>*@
@model MvcMovie.Models.MovieGenreViewModel
@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
@*<form asp-controller="Movies" asp-action="Index">*@
<form asp-controller="Movies" asp-action="Index" method="get">
    <p>
        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>

        Title: <input type="text" asp-for="SearchString" />
       @* Title: <input type="text" name="SearchString" />*@
        <input type="submit" value="Filter" />
    </p>
</form>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movies[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies[0].Price)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies[0].Rating)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Movies)
        {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                    @Html.DisplayFor(modelItem => item.Rating)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.Id">Details</a> |
                <a asp-action="Delete" asp-route-id="@item.Id">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

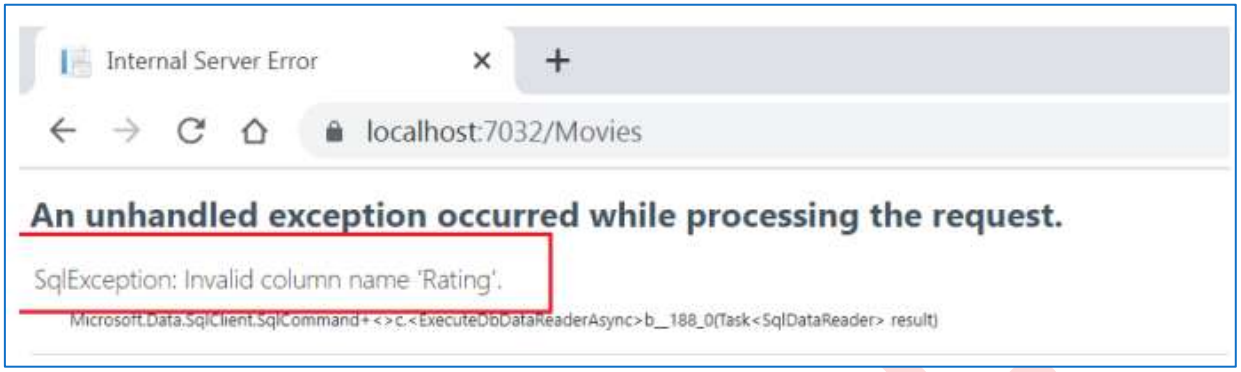
- Update the **/Views/Movies/Create.cshtml** with a **Rating** field.

```
@model MvcMovie.Models.Movie

@{
    ViewData["Title"] = "Create";
}

<h1>Create</h1>

<h4>Movie</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Create">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Title" class="control-label"></label>
                <input asp-for="Title" class="form-control" />
                <span asp-validation-for="Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Rating" class="control-label"></label>
                <input asp-for="Rating" class="form-control" />
                <span asp-validation-for="Rating" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="ReleaseDate" class="control-label"></label>
                <input asp-for="ReleaseDate" class="form-control" />
                <span asp-validation-for="ReleaseDate" class="text-danger"></span>
            </div>
```

Update the remaining templates. Update the SeedData class so that it provides a value for the new column. A sample change is shown below, but you'll want to make this change for each new Movie.

```
context.Movie.AddRange(
    new Movie
    {
        Title = "When Harry Met Sally",
        ReleaseDate = DateTime.Parse("1989-2-12"),
        Genre = "Romantic Comedy",
        Rating = "R",
        Price = 7.99M
    },

    new Movie
    {
        Title = "Ghostbusters ",
        ReleaseDate = DateTime.Parse("1984-3-13"),
        Genre = "Comedy",
        Rating = "R",
        Price = 8.99M
    },

    new Movie
    {
        Title = "Ghostbusters 2",
        ReleaseDate = DateTime.Parse("1986-2-23"),
        Genre = "Comedy",
        Rating = "R",
        Price = 9.99M
    },

    new Movie
    {
        Title = "Rio Bravo",
        ReleaseDate = DateTime.Parse("1959-4-15"),
        Genre = "Western",
        Rating = "R",
        Price = 3.99M
    }
);
context.SaveChanges();
```

The app won't work until the DB is updated to include the new field. If it's run now, the following SqlException is thrown:

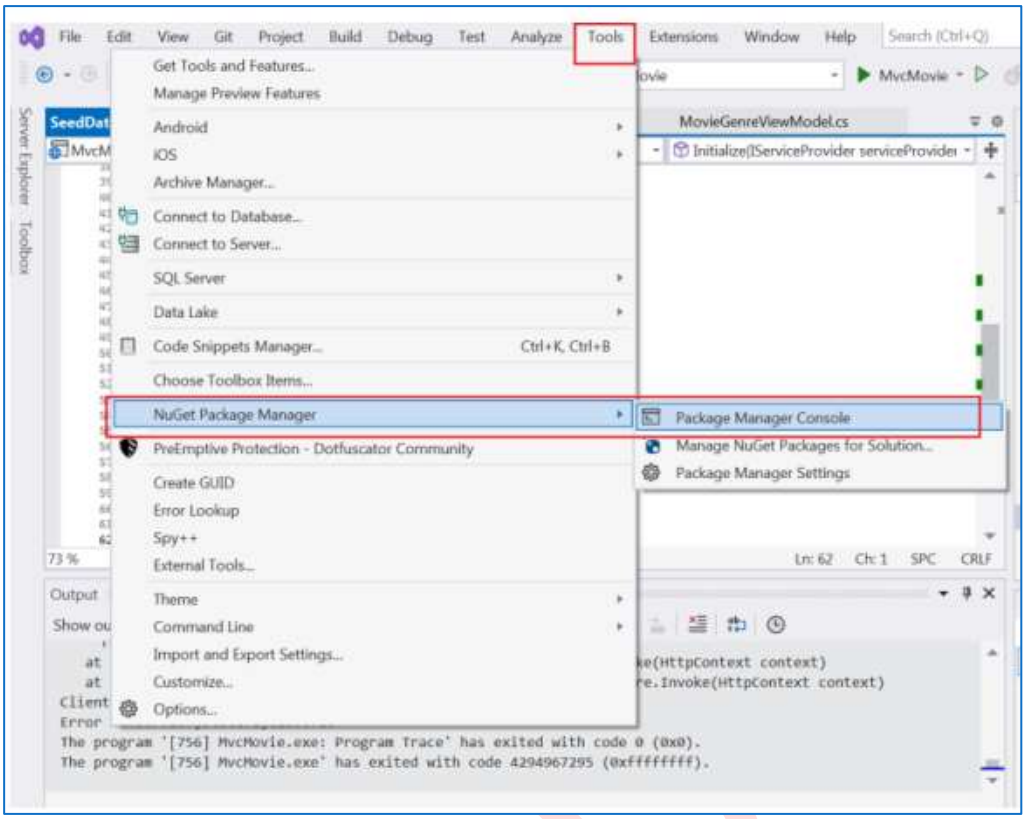


The error arises due to a discrepancy between the updated Movie model class and the schema of the Movie table in the current database. Specifically, the database table lacks a Rating column, leading to the issue.

There are a few approaches to resolving the error:

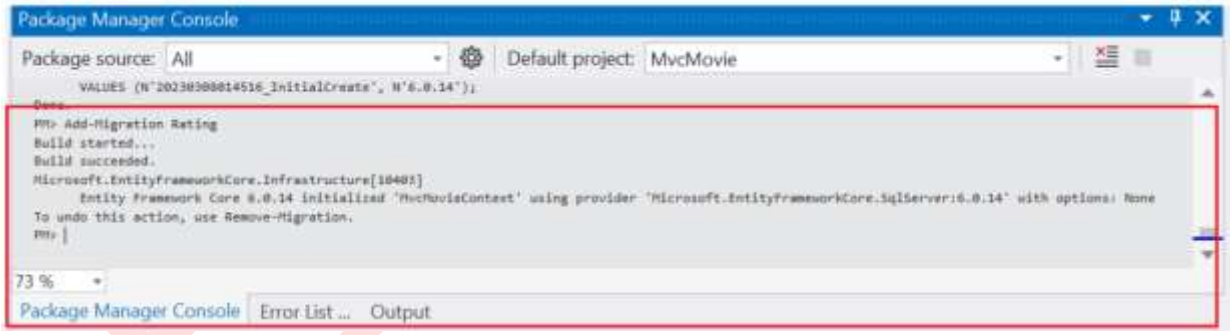
1. Automatically dropping and re-creating the database in Entity Framework based on the new model class schema is convenient for active development on a test database, allowing easy evolution of both model and schema. However, avoid using this approach on a production database, as it will result in data loss. Instead, use an initializer to seed the database with test data, especially during early development and with SQLite.
2. To align the model classes with the existing database, you have two options for explicitly modifying the schema. By doing so, you ensure data consistency. This can be achieved either through manual alterations or by generating a database change script. Both approaches offer the advantage of maintaining your data intact.
3. **Use Code First Migrations** to update the database schema. For this Lab, Code First Migrations is used.

For this Lab, Code First Migrations is used. From the Tools menu, select NuGet Package Manager > Package Manager Console.



In the PMC, enter the following commands:

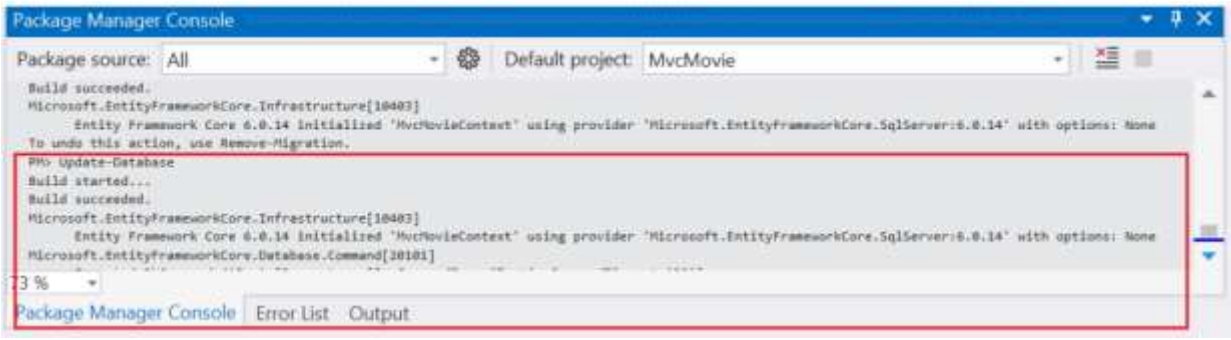
- **Add-Migration Rating**
- **Update-Database**



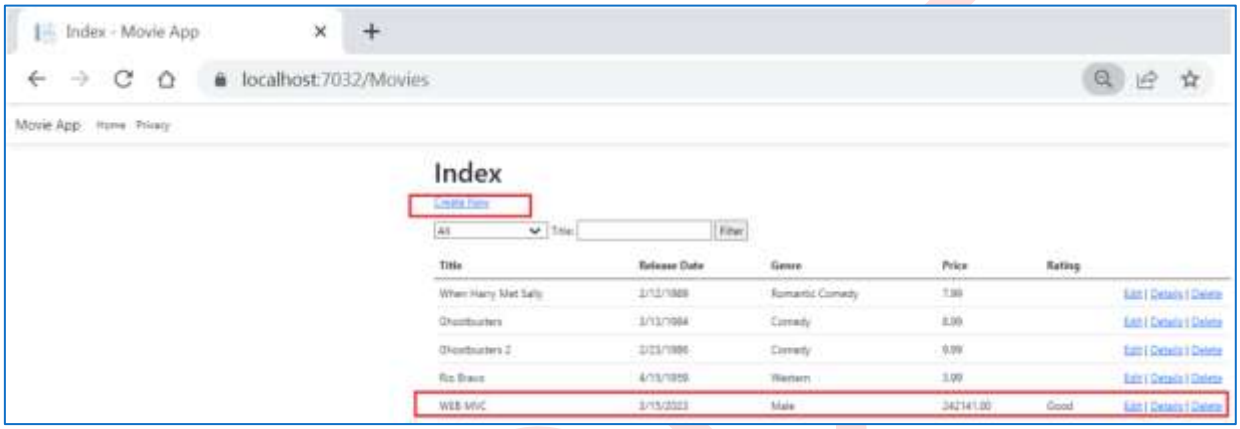
The **Add-Migration** command tells the migration framework to examine the current **Movie** model with the current **Movie DB** schema and create the necessary code to migrate the DB to the new model.

The name "**Rating**" is arbitrary and is used to name the migration file. It's helpful to use a meaningful name for the migration file.

If all the records in the DB are deleted, the initialize method will seed the DB and include the **Rating** field.



Run the app and verify you can create, edit, and display movies with a **Rating** field.



4. Add validation rules to the movie model

- Validation logic is added to the Movie model.
- You ensure that the validation rules are enforced any time a user creates or edits a movie.

ASP.NET Core MVC adheres to the DRY principle ("Don't Repeat Yourself"), reducing code duplication and enhancing maintainability. Validation support in MVC and Entity Framework Core Code First exemplifies DRY by enabling declarative specification of rules in one place (the model class) and enforcing them throughout the app. Utilize built-in validation attributes like Required, StringLength, RegularExpression, and Range to optimize the Movie class for validation.

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
namespace MvcMovie.Models
{
    public class Movie
    {
        public int Id { get; set; }

        [StringLength(60, MinimumLength = 3)]
        [Required]
        public string? Title { get; set; }

        [Display(Name = "Release Date")]
        [DataType(DataType.Date)] //<= Using Data Anotations
        public DateTime ReleaseDate { get; set; }

        [RegularExpression(@"^[A-Z]+[a-zA-Z\s]*$")]
        [Required]
        [StringLength(30)]
        public string? Genre { get; set; }

        [Range(1, 100)]
        [DataType(DataType.Currency)]
        [Column(TypeName = "decimal(18, 2)")]
        public decimal Price { get; set; }

        [RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*$")]
        [StringLength(5)]
        [Required]
        public string? Rating { get; set; }
    }
}
```

The validation attributes specify behavior that you want to enforce on the model properties they're applied to:

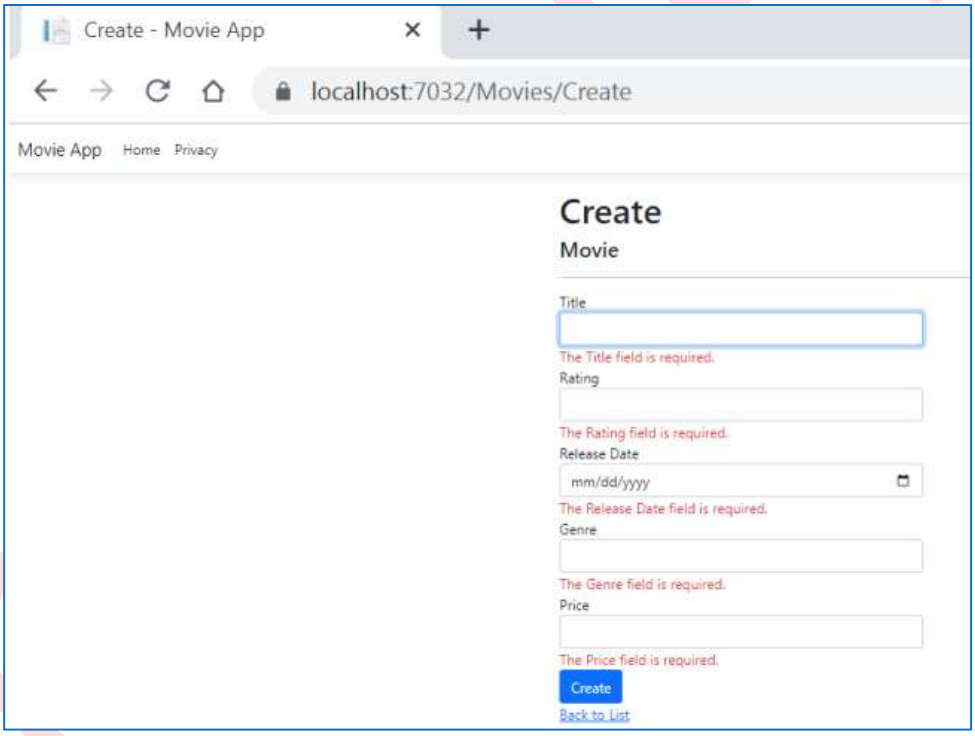
- The Required and MinimumLength attributes indicate that a property must have a value; but nothing prevents a user from entering white space to satisfy this validation.
- The RegularExpression attribute is used to limit what characters can be input. In the preceding code, "Genre":
  - ✓ Must only use letters.
  - ✓ The first letter is required to be uppercase. White spaces are allowed while numbers, and special characters are not allowed
- The RegularExpression "Rating":
  - ✓ Requires that the first character be an uppercase letter.
  - ✓ Allows special characters and numbers in subsequent spaces. "PG-13" is valid for a rating, but fails for a "Genre".
- The **Range** attribute constrains a value to within a specified range.

- The **StringLength** attribute lets you set the maximum length of a string property, and optionally its minimum length.
- Value types (such as **decimal, int, float, DateTime**) are inherently required and don't need the [**Required**] attribute.

Having validation rules automatically enforced by ASP.NET Core helps make your app more robust. It also ensures that you can't forget to validate something and inadvertently let bad data into the database. Validation Error UI:

Run SQL: DELETE FROM [MvcMovieContext-7dc5].[dbo].[Movie];

Run the app and navigate to the Movies controller. Select the Create New link to add a new movie. Fill out the form with some invalid values. As soon as jQuery client side validation detects the error, it displays an error message.



5. Using DataType Attributes

Open the **Movie.cs** file and examine the **Movie** class. The **System.ComponentModel.DataAnnotations** namespace provides formatting attributes in addition to the built-in set of validation attributes.

We've already applied a **DataType** enumeration value to the release date and to the price fields. The following code shows the **ReleaseDate** and **Price** properties with the appropriate **DataType** attribute:

```
[Display(Name = "Release Date")]
        [DataType(DataType.Date)] //<= Using Data Anotations
        public DateTime ReleaseDate { get; set; }

        [RegularExpression(@"^[A-Z]+[a-zA-Z\s]*$")]
        [Required]
        [StringLength(30)]
        public string? Genre { get; set; }

        [Range(1, 100)]
        [DataType(DataType.Currency)]
        [Column(TypeName = "decimal(18, 2)")]
        public decimal Price { get; set; }
```

The DataType attributes only provide hints for the view engine to format the data and supplies elements/attributes such as <a> for URL's and **<a href="mailto:EmailAddress.com">** for email. You can use the **RegularExpression** attribute to validate the format of the data.

The **DataType** attribute is used to specify a data type that's more specific than the database intrinsic type, they're not validation attributes. In this case we only want to keep track of the date, not the time.

The **DataType** Enumeration provides for many data types, such as Date, Time, PhoneNumber, Currency, EmailAddress and more. The DataType attribute can also enable the application to automatically provide type-specific features. For example, a **mailto**: link can be created for **DataType.EmailAddress**, and a date selector can be provided for **DataType.Date** in browsers that support HTML5.

The **DataType** attributes emit HTML 5 data- (pronounced data dash) attributes that HTML 5 browsers can understand. The DataType attributes do not provide any validation.

DataType.Date doesn't specify the format of the date that's displayed. By default, the data field is displayed according to the default formats based on the server's **CultureInfo.** The **DisplayFormat** attribute is used to explicitly specify the date format.

```
[Display(Name = "Release Date")]
[DataType(DataType.Date)] //<= Using Data Anotations
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
public DateTime ReleaseDate { get; set; }
```

The **ApplyFormatInEditMode** setting specifies that the formatting should also be applied when the value is displayed in a text box for editing. (You might not want that for some fields — for example, for currency values, you probably don't want the currency symbol in the text box for editing.)

You can use the **DisplayFormat** attribute by itself, but it's generally a good idea to use the **DataType** attribute. The **DataType** attribute conveys the semantics of the data as opposed to how to render it on a screen, and provides the following benefits that you don't get with DisplayFormat:

- The browser can enable HTML5 features (for example to show a calendar control, the locale-appropriate currency symbol, email links, etc.)
- By default, the browser will render data using the correct format based on your locale.
- The DataType attribute can enable MVC to choose the right field template to render the data (the DisplayFormat if used by itself uses the string template)

You will need to disable jQuery date validation to use the **Range** attribute with **DateTime**. It's generally not a good practice to compile hard dates in your models, so using the **Range** attribute and **DateTime** is discouraged.

The following code shows combining attributes on one line:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
namespace MvcMovie.Models
{
    public class Movie
    {
        public int Id { get; set; }

        [StringLength(60, MinimumLength = 3)]
        public string Title { get; set; }

        [Display(Name = "Release Date"), DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }

        [RegularExpression(@"^[A-Z]+[a-zA-Z\s]*$"), Required, StringLength(30)]
        public string Genre { get; set; }

        [Range(1, 100), DataType(DataType.Currency)]
        [Column(TypeName = "decimal(18, 2)")]
        public decimal Price { get; set; }

        [RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*$"), StringLength(5)]
        public string Rating { get; set; }
    }
}
```

We will review the app and make some improvements to the automatically generated **Details** and **Delete** methods. Open the **Movie controller** and examine the **Details** method:

```
// GET: Movies/Details/5
public async Task<IActionResult> Details(int? id)
{
    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);
}
```

The MVC scaffolding engine that created this action method adds a comment showing an HTTP request that invokes the method. In this case it's a GET request with three URL segments, the **Movies** controller, the **Details** method, and an **id** value. Recall these segments are defined in **Program.cs.**

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

EF simplifies data searching with the **FirstOrDefaultAsync** method, which includes an important security measure. Before performing any actions, the code validates whether the search method successfully locates a movie. This precaution prevents potential manipulation of the site's URLs by hackers, who could introduce errors by modifying the link from, for instance, **http://localhost:{PORT}/Movies/Details/1** to something like **http://localhost:{PORT}/Movies/Details/12345** (a value that doesn't correspond to an actual movie). Without verifying for a null movie, the application would encounter an exception. To understand this further, please examine the Delete and DeleteConfirmed methods.

```
// GET: Movies/Delete/5
public async Task<IActionResult> Delete(int? id)
{
    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);
}
```

```
// POST: Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    if (_context.Movie == null)
    {
        return Problem("Entity set 'MvcMovieContext.Movie'  is null.");
    }
    var movie = await _context.Movie.FindAsync(id);
    if (movie != null)
    {
        _context.Movie.Remove(movie);
    }

    await _context.SaveChangesAsync();
    return RedirectToAction(nameof(Index));
}
```

Please be aware that utilizing the **HTTP GET Delete** method does not actually delete the designated movie. Instead, it provides a view of the movie, allowing you to submit a deletion request using the HTTP POST method. Executing a delete operation in response to a GET request (or any other operation like edit or create that alters data) can create a security vulnerability.

The [**HttpPost**] method that deletes the data is named **DeleteConfirmed** to give the HTTP POST method a unique signature or name. The two method signatures are shown below:

```
// GET: Movies/Delete/5
        public async Task<IActionResult> Delete(int? id)
        {

...

// POST: Movies/Delete/5
        [HttpPost, ActionName("Delete")]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult> DeleteConfirmed(int id)
        {
```

The CLR mandates that overloaded methods must possess a unique parameter signature, denoting the same method name but with a different list of parameters. Nevertheless, in this scenario, there's a requirement for two Delete methods—one for GET and another for POST—that share an identical parameter signature, both accepting a single integer as input.

Two strategies exist to tackle this issue. The first approach involves assigning distinct names to the methods, as demonstrated in the preceding example through the scaffolding mechanism. Nonetheless, this approach poses a minor setback: ASP.NET relies on method names to map segments of a URL to corresponding action methods.

Consequently, renaming a method might disrupt the routing system's ability to locate the intended method.

To overcome this challenge, the solution employed in the provided example includes adding the ActionName("Delete") attribute to the DeleteConfirmed method. This attribute facilitates mapping for the routing system, ensuring that a URL containing "/Delete/" for a POST request will successfully find the DeleteConfirmed method.

An additional workaround frequently employed for methods with matching names and signatures involves artificially modifying the POST method's signature by adding an extra, unused parameter. This approach was previously implemented in a prior post, where we introduced the "notUsed" parameter.

You could do the same thing here for the **[HttpPost] Delete** method:

```
// POST: Movies/Delete/6
        [HttpPost]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult> Delete(int id, bool notUsed)

//This code in Page 43 in LAB 04
       [HttpPost]
        public string Index(string searchString, bool notUsed)
        {
            return "From [HttpPost]Index: filter on " + searchString;
        }

// End - This code in Page 43 in LAB 04
```

SUBMIT YOUR CODE TO GIT!

- - - END LAB 04 - - -

LAB 5
Minimal APIs

I. TARGET

- ✓ Create a minimal API with ASP.NET Core
- ✓ Creating and Using HTTP Client SDKs in .NET 6

II. REFERENCES

- ✓ Introduction to Minimal APIs in .NET 6, https://www.claudiobernasconi.ch/2022/02/23/introduction-to-minimal-apis-in-dotnet6
- ✓ Minimal APIs quick reference, https://learn.microsoft.com/en-us/aspnet/core/fundamentals/minimal-apis?view=aspnetcore-7.0

III. REQUIREMENTS

- ✓ Create a minimal API with ASP.NET Core
- ✓ Creating and Using HTTP Client SDKs in .NET 6

IV. IN LAB

ASP.NET Core supports two API approaches: **controllers (Lab06) and minimal APIs (Lab05).** Controllers use inheritance for endpoints, while minimal APIs utilize lambdas/methods. Minimal APIs prioritize simplicity and hide the host class, while controllers allow dependency injection via constructor/property injection. Both support dependency injection but minimal APIs access the service provider. Choose based on project needs and preferences..

Here's sample code for an **API based on controllers:**

```
namespace APIWithControllers;

public class Program
{
    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);

        builder.Services.AddControllers();
        var app = builder.Build();

        app.UseHttpsRedirection();

        app.MapControllers();

        app.Run();
    }
}
```

And

```
using Microsoft.AspNetCore.Mvc;
namespace APIWithControllers.Controllers;
[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
{
    private static readonly string[] Summaries = new[]
    {
        "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm",
        "Balmy","Hot","Sweltering","Scorching"
    };

    private readonly ILogger<WeatherForecastController> _logger;

    public WeatherForecastController(ILogger<WeatherForecastController> logger)
    {
        _logger = logger;
    }

    [HttpGet(Name = "GetWeatherForecast")]
    public IEnumerable<WeatherForecast> Get()
    {
        return Enumerable.Range(1, 5).Select(index => new WeatherForecast
        {
            Date = DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
            TemperatureC = Random.Shared.Next(-20, 55),
            Summary = Summaries[Random.Shared.Next(Summaries.Length)]
        })
        .ToArray();
    }
}
```

The following code provides the same functionality in a minimal API project. Notice that the minimal API approach involves including the related code in lambda expressions.

```
namespace MinimalAPI;

public class Program
{
    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);

        var app = builder.Build();

        app.UseHttpsRedirection();

        var summaries = new[]
        {
            "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy", "Hot",
            "Sweltering", "Scorching"
        };

        app.MapGet("/weatherforecast", (HttpContext httpContext) =>
        {
            var forecast = Enumerable.Range(1, 5).Select(index =>
                new WeatherForecast
                {
                    Date = DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
                    TemperatureC = Random.Shared.Next(-20, 55),
                    Summary = summaries[Random.Shared.Next(summaries.Length)]
                })
                .ToArray();
            return forecast;
        });

        app.Run();
    }
}
```

Both API projects refer to the following class:

```
namespace APIWithControllers;
public class WeatherForecast
{
    public DateOnly Date { get; set; }

    public int TemperatureC { get; set; }

    public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);

    public string? Summary { get; set; }
}
```

Minimal APIs have many of the same capabilities as controller-based APIs. They support the configuration and customization needed to scale to multiple APIs, handle complex routes, apply authorization rules, and control the content of API responses. There are a few capabilities available with controller-based APIs that are not yet supported or implemented by minimal APIs. These include:

- No built-in support for model binding (IModelBinderProvider, IModelBinder). Support can be added with a custom binding shim.
- No support for binding from forms. This includes binding IFormFile.

- No built-in support for validation (IModelValidator).
- No support for application parts or the application model. There's no way to apply or build your own conventions.
- No built-in view rendering support. We recommend using Razor Pages for rendering views.
- No support for JsonPatch
- No support for Odata

Minimal APIs are a simplified approach for building fast HTTP APIs with ASP.NET Core. You can build fully functioning REST endpoints with minimal code and configuration. Skip traditional scaffolding and avoid unnecessary controllers by fluently declaring API routes and actions. For example, the following code creates an API at the root of the web app (Program.cs) that returns the text, "Hello World!".

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
app.MapGet("/", () => "Hello World!");
app.Run();
```

Most APIs accept parameters as part of the route.

```
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapGet("/users/{userId}/books/{bookId}",
    (int userId, int bookId) => $"The user id is {userId} and book id is
{bookId}");

app.Run();
```

That's all it takes to get started, but it's not all that's available. Minimal APIs support the configuration and customization needed to scale to multiple APIs, handle complex routes, apply authorization rules, and control the content of API responses

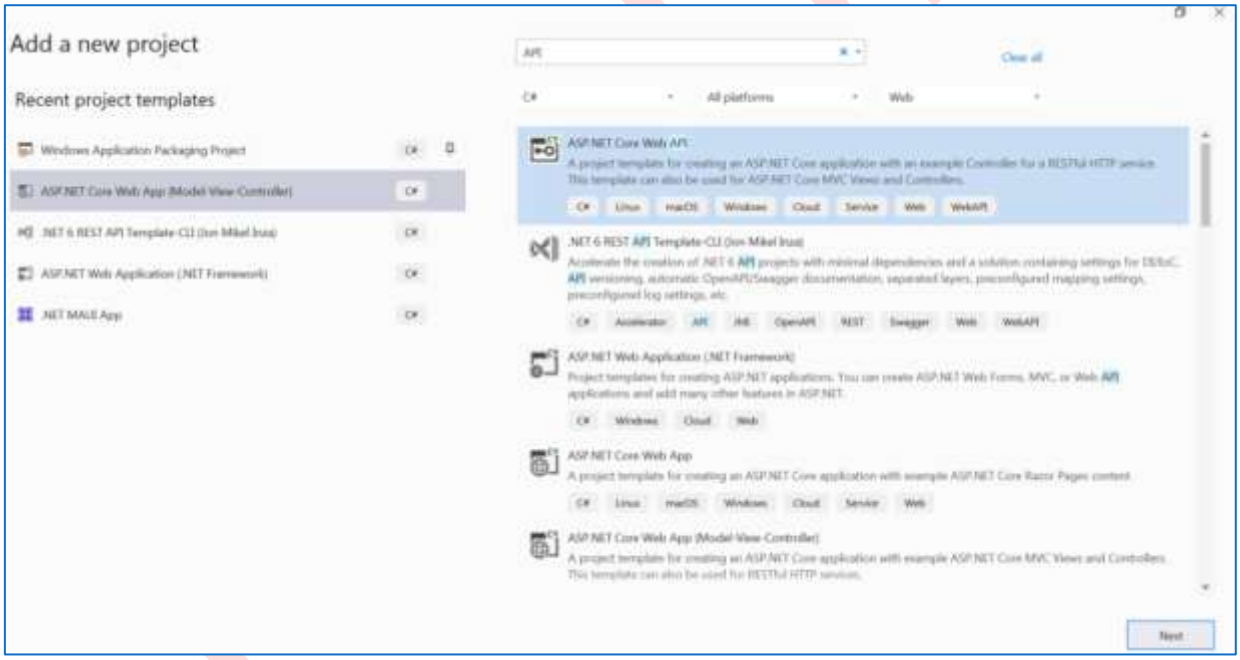
Minimal APIs are architected to create HTTP APIs with minimal dependencies. They are ideal for microservices and apps that want to include only the minimum files, features, and dependencies in ASP.NET Core. This Lab teaches the basics of building a minimal API with ASP.NET Core. This Lab creates the following API (Can use https://www.postman.com/downloads/ for testing):

API	Description	Request body	Response body
GET /	Browser test, "Hello World"	None	Hello World!
GET /todoitems	Get all to-do items	None	Array of to-do items
GET /todoitems/complete	Get completed to-do items	None	Array of to-do items
GET /todoitems/{id}	Get an item by ID	None	To-do item
POST /todoitems	Add a new item	To-do item	To-do item
PUT /todoitems/{id}	Update an existing item	To-do item	None
DELETE /todoitems/{id}	Delete an item	None	None

1. Create a new project

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

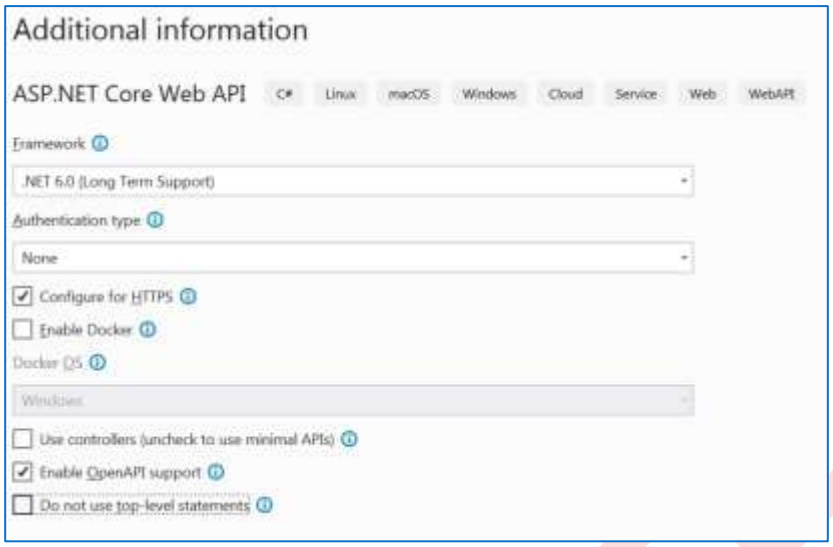
- In the **Create a new project** dialog:
- Enter **API** in the **Search for templates** search box.
- Select the **ASP.NET Core Web API** template and select **Next**.



Name the project **TodoApi** and select **Next**.

In the **Additional information** dialog:

- Select .NET 6.0 (Long-term support)
- Remove Use controllers (uncheck to use minimal APIs)
- Select Create



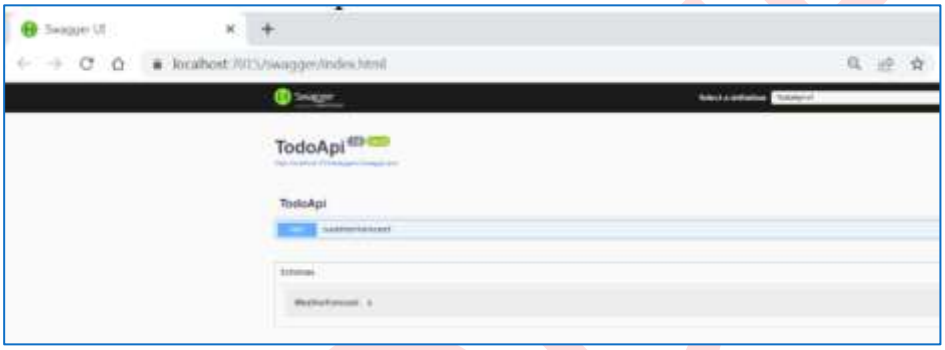
The Program.cs file contains the following code:

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();

var summaries = new[]
{
    "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy", "Hot", "Sweltering",
"Scorching"
};

app.MapGet("/weatherforecast", () =>
{
    var forecast = Enumerable.Range(1, 5).Select(index =>
        new WeatherForecast
        (
            DateTime.Now.AddDays(index),
            Random.Shared.Next(-20, 55),
            summaries[Random.Shared.Next(summaries.Length)]
        ))
        .ToArray();
    return forecast;
})
.WithName("GetWeatherForecast");

app.Run();

internal record WeatherForecast(DateTime Date, int TemperatureC, string? Summary)
{
    public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);
}
```

The project template creates a WeatherForecast API with support for Swagger (Lab06). Swagger is used to generate useful documentation and help pages for APIs. The following highlighted code adds support for Swagger:

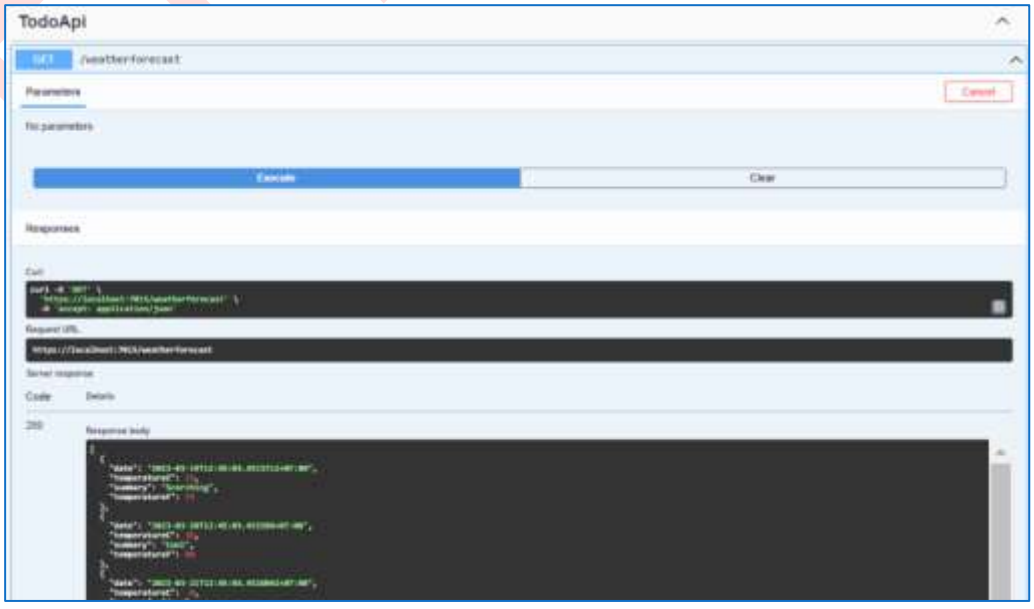
```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}
```

Press Ctrl+F5 to run without the debugger:

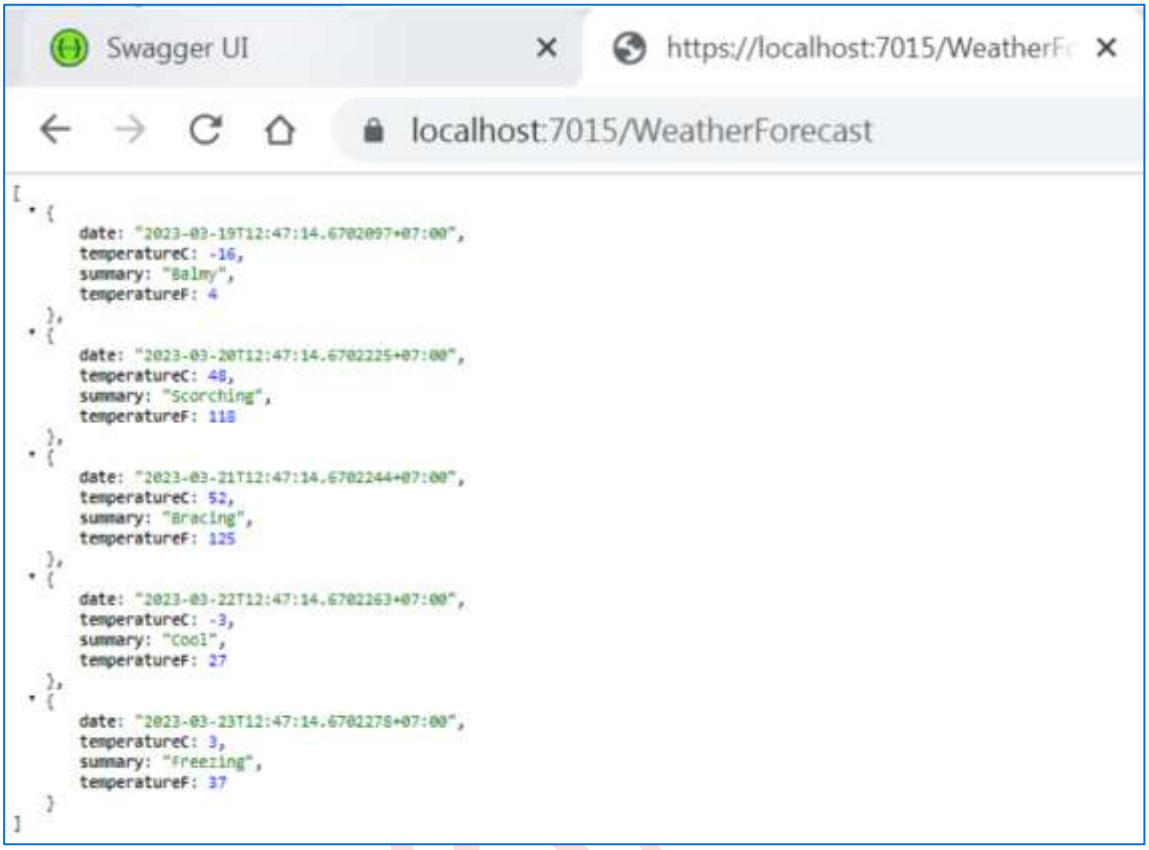


The Swagger page /swagger/index.html is displayed. Select GET > Try it out> Execute. The page displays:

- The Curl command to test the WeatherForecast API.
- The URL to test the WeatherForecast API.
- The response code, body, and headers.
- A drop down list box with media types and the example value and schema.



Copy and paste the Request URL in the browser: https://localhost:/WeatherForecast. JSON similar to the following is returned:



This Lab focuses on creating an API, so we'll delete the Swagger code and the WeatherForecast code. Replace the contents of the Program.cs file with the following:

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

app.MapGet("/", () => "Hello World!");

app.Run();
```

The following highlighted code creates a WebApplicationBuilder and a WebApplication with preconfigured defaults:

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

app.MapGet("/", () => "Hello World!");

app.Run();
```

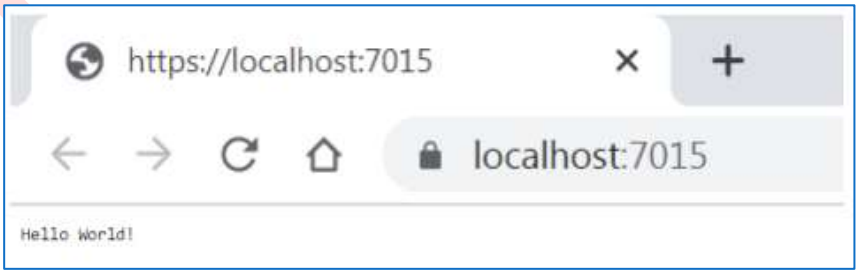
The following code creates an HTTP GET endpoint / which returns Hello World!:

app.MapGet("/", () => "Hello World!"); app.Run(); → runs the app.

Remove the two **"launchUrl": "swagger"**, lines from the **Properties/launchSettings.json** file. When the **launchUrl** isn't specified, the web browser requests the / endpoint.

```
{
  "$schema": "https://json.schemastore.org/launchsettings.json",
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:12077",
      "sslPort": 44377
    }
  },
  "profiles": {
    "TodoApi": {

      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
     // "launchUrl": "swagger",
      "applicationUrl": "https://localhost:7015;http://localhost:5178",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
     // "launchUrl": "swagger",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

Run the app. Hello World! is displayed. The updated Program.cs file contains a minimal but complete app.



- **Add NuGet packages:**
  - From the **Tools** menu, select **NuGet Package Manager > Manage NuGet Packages for Solution**.

- Enter **Microsoft.EntityFrameworkCore.InMemory** in the search box, and then select **Microsoft.EntityFrameworkCore.InMemory.**(Install version 6.0.15)
- Select the **Project** checkbox in the right pane and then select Install. o Follow the preceding instructions to add the **Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore** (version 6.0.15).

Add the API code

Replace the contents of the Program.cs file with the following code:

```
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<TodoDb>(opt => opt.UseInMemoryDatabase("TodoList"));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.MapGet("/todoitems", async (TodoDb db) =>
    await db.Todos.ToListAsync());

app.MapGet("/todoitems/complete", async (TodoDb db) =>
    await db.Todos.Where(t => t.IsComplete).ToListAsync());

app.MapGet("/todoitems/{id}", async (int id, TodoDb db) =>
    await db.Todos.FindAsync(id)
        is Todo todo
            ? Results.Ok(todo)
            : Results.NotFound());

app.MapPost("/todoitems", async (Todo todo, TodoDb db) =>
{
    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todo.Id}", todo);
});

app.MapPut("/todoitems/{id}", async (int id, Todo inputTodo, TodoDb db) =>
{
    var todo = await db.Todos.FindAsync(id);

    if (todo is null) return Results.NotFound();

    todo.Name = inputTodo.Name;
    todo.IsComplete = inputTodo.IsComplete;

    await db.SaveChangesAsync();

    return Results.NoContent();
});

app.MapDelete("/todoitems/{id}", async (int id, TodoDb db) =>
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
        await db.SaveChangesAsync();
        return Results.Ok(todo);
    }

    return Results.NotFound();
});
```

The model and database context classes

The sample app contains the following model:

A model is a class that represents data that the app manages. The model for this app is the Todo class. The sample app contains the following database context class:

```
using Microsoft.EntityFrameworkCore;

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

        public DbSet<Todo> Todos => Set<Todo>();
    }
}
```

The database context is the main class that coordinates Entity Framework functionality for a data model. This class is created by deriving from the **Microsoft.EntityFrameworkCore.DbContext** class.

The following highlighted code adds the database context to the dependency injection (DI) container and enables displaying database-related exceptions:

```
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<TodoDb>(opt => opt.UseInMemoryDatabase("TodoList"));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

var app = builder.Build();

app.MapGet("/", () => "Hello World!");
```

The DI container provides access to the database context and other services. The following code creates an HTTP POST endpoint /todoitems to add data to the in-memory database:

```
app.MapPost("/todoitems", async (Todo todo, TodoDb db) =>
{
    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todo.Id}", todo);
});
```

2. Test Api

2.1. Install Postman to test the app

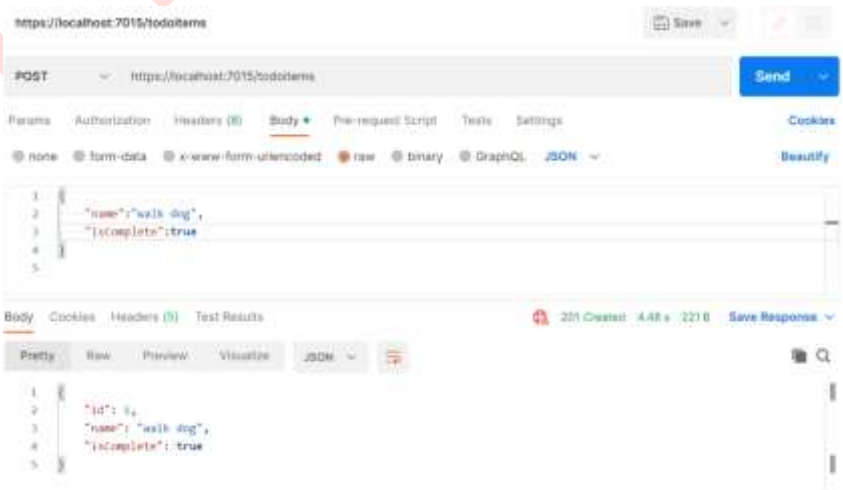
- Install Postman
- Start the web app.
- Start Postman.
- Disable SSL certificate verification → From File > Settings (General tab), disable SSL certificate verification.

2.2. Test posting data

- Create a new HTTP request.
- Set the HTTP method to POST.
- Set the URI to https://localhost:<port>/todoitems.
- For example: https://localhost:5001/todoitems
- Select the Body tab.
- Select raw.
- Set the type to JSON
- In the request body enter JSON for a to-do item:

```
{
    "name":"walk dog",
  "isComplete":true
}
```

- **Select Send**



Examine the GET endpoints

The sample app implements several GET endpoints using calls to **MapGet**:

API	Description	Request body	Response body
GET /	Browser test, "Hello World"	None	Hello World!
GET /todoitems	Get all to-do items	None	Array of to-do items
GET /todoitems/complete	Get completed to-do items	None	Array of to-do items
GET /todoitems/{id}	Get an item by ID	None	To-do item

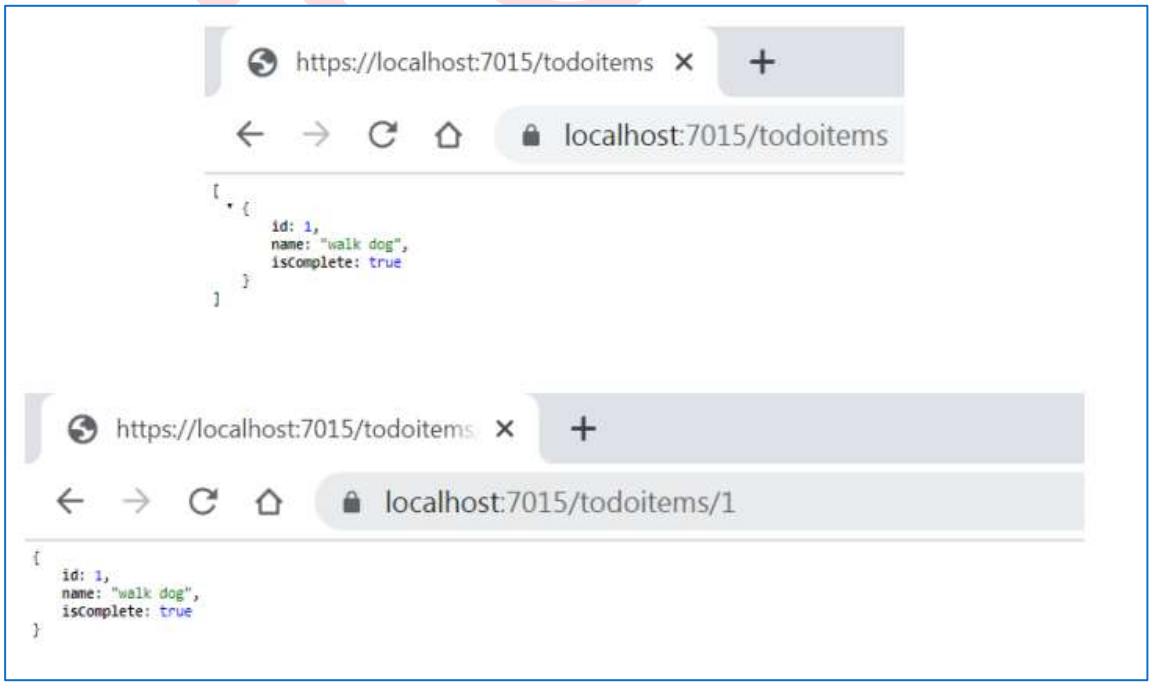
```
app.MapGet("/", () => "Hello World!");

app.MapGet("/todoitems", async (TodoDb db) =>
    await db.Todos.ToListAsync());

app.MapGet("/todoitems/complete", async (TodoDb db) =>
    await db.Todos.Where(t => t.IsComplete).ToListAsync());

app.MapGet("/todoitems/{id}", async (int id, TodoDb db) =>
    await db.Todos.FindAsync(id)
        is Todo todo
            ? Results.Ok(todo)
            : Results.NotFound());
```

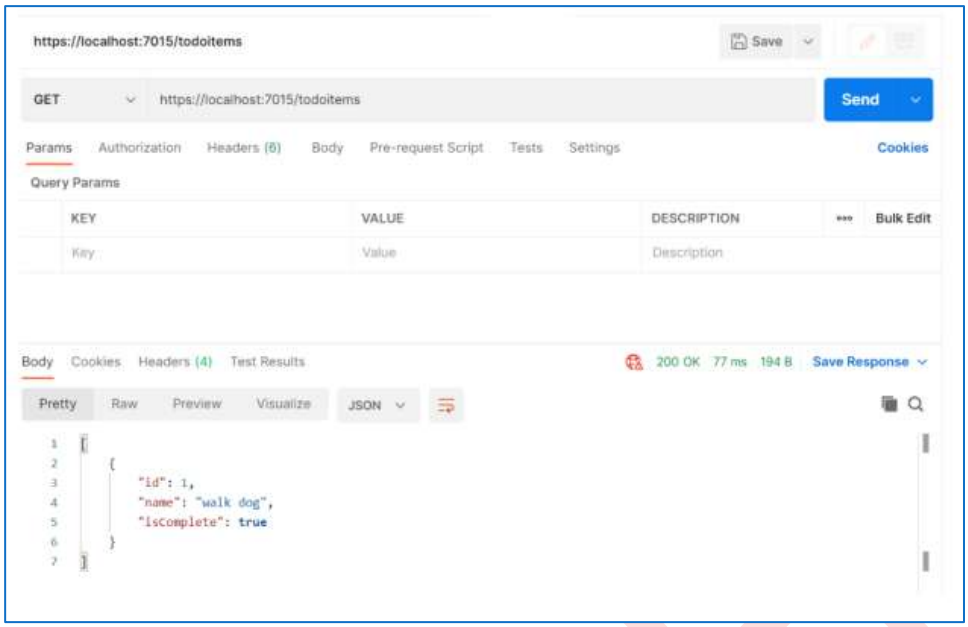
2.3. Test the GET endpoints

Test the app by calling the two endpoints from a browser or Postman. For example:

- **GET** https://localhost:5001/todoitems
- **GET** https://localhost:5001/todoitems/1

The call to **GET /todoitems** produces a response similar to the following:





This app uses an in-memory database. If the app is restarted, the GET request doesn't return any data. If no data is returned, first POST data to the app.

Return values

ASP.NET Core automatically serializes the object to JSON and writes the JSON into the body of the response message. The response code for this return type is 200 OK, assuming there are no unhandled exceptions. Unhandled exceptions are translated into 5xx errors.

The return types can represent a wide range of HTTP status codes. For example, **GET /todoitems/{id}** can return two different status values:

- If no item matches the requested ID, the method returns a 404 status NotFound error code.
- Otherwise, the method returns 200 with a JSON response body. Returning item results in an HTTP 200 response.

2.4. Examine the PUT endpoint

The sample app implements a single PUT endpoint using **MapPut:**

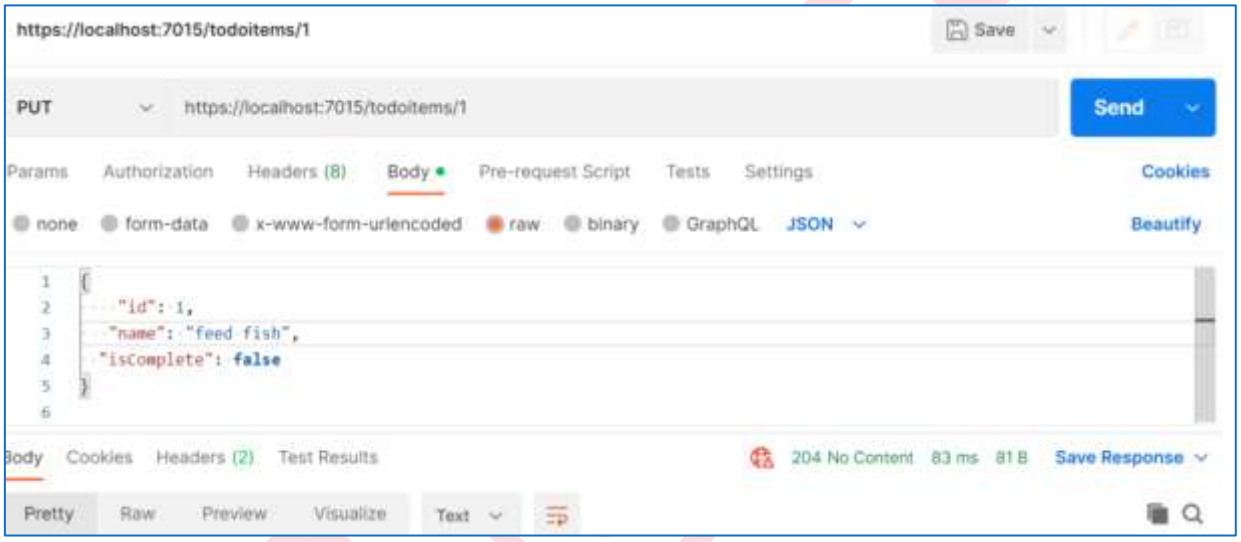
```
app.MapPut("/todoitems/{id}", async (int id, Todo inputTodo, TodoDb db) =>
{
    var todo = await db.Todos.FindAsync(id);

    if (todo is null) return Results.NotFound();

    todo.Name = inputTodo.Name;
    todo.IsComplete = inputTodo.IsComplete;

    await db.SaveChangesAsync();

    return Results.NoContent();
});
```

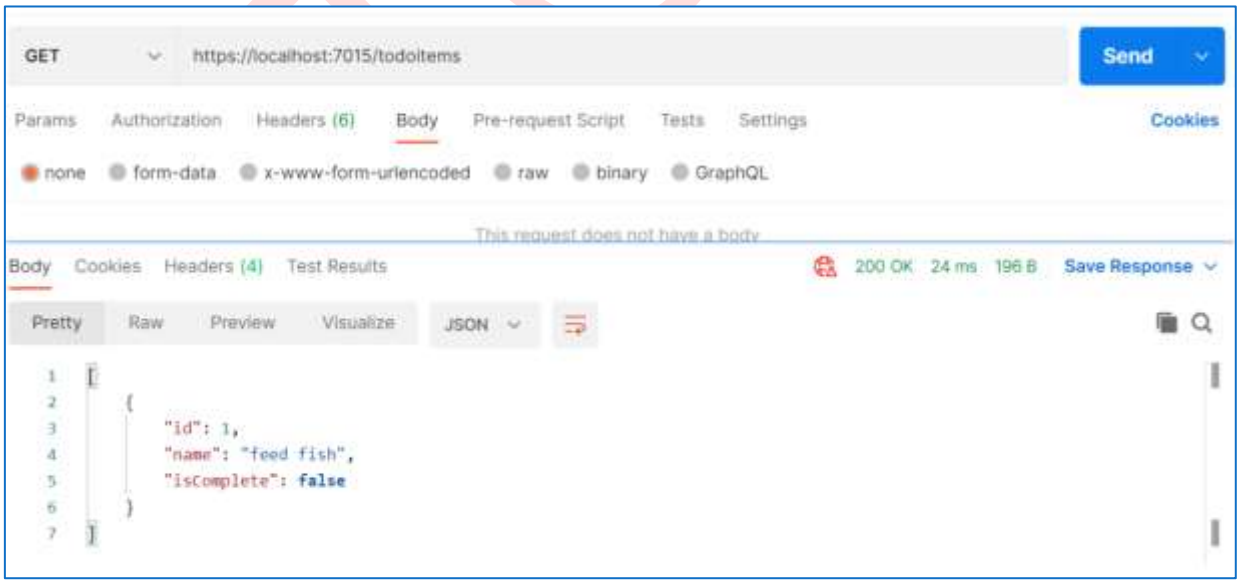
This method is similar to the **MapPost** method, except it uses HTTP PUT. A successful response returns **204 (No Content).** According to the HTTP specification, a PUT request requires the client to send the entire updated entity, not just the changes. To support partial updates, use **HTTP PATCH**.

Test the PUT endpoint

This sample uses an in-memory database that must be initialized each time the app is started. There must be an item in the database before you make a PUT call. Call GET to ensure there's an item in the database before making a PUT call.

Update the to-do item that has **Id = 1** and set its name to **"feed fish":**





2.5. Examine the DELETE endpoint

The sample app implements a single DELETE endpoint using **MapDelete**:

```
app.MapDelete("/todoitems/{id}", async (int id, TodoDb db) =>
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
        await db.SaveChangesAsync();
        return Results.Ok(todo);
    }

    return Results.NotFound();
});
```

Use Postman to delete a to-do item:

- Set the method to **DELETE**.
- Set the URI of the object to delete (for example ***https://localhost:5001/todoitems/1***).
- Select **Send**

3. Prevent over-posting

Currently the sample app exposes the entire Todo object. Production apps typically limit the data that's input and returned using a subset of the model. There are multiple reasons behind this and security is a major one. The subset of a model is usually referred to as a Data Transfer Object (DTO), input model, or view model. DTO is used in this article.

A DTO may be used to:

- Prevent over-posting.
- Hide properties that clients are not supposed to view.
- Omit some properties in order to reduce payload size.
- Flatten object graphs that contain nested objects. Flattened object graphs can be more convenient for clients.

To demonstrate the DTO approach, update the Todo class to include a secret field:

```
namespace TodoApi.Models
{
    public class Todo
    {
        public int Id { get; set; }
        public string? Name { get; set; }
        public bool IsComplete { get; set; }
        public string? Secret { get; set; }
    }
}
```

The secret field needs to be hidden from this app, but an administrative app could choose to expose it. Verify you can post and get the secret field.

3.1. Create a DTO model

```
namespace TodoApi.Models
{
    public class TodoItemDTO
    {
        public int Id { get; set; }
        public string? Name { get; set; }
        public bool IsComplete { get; set; }

        public TodoItemDTO() { }
        public TodoItemDTO(Todo todoItem) =>
        (Id, Name, IsComplete) = (todoItem.Id, todoItem.Name, todoItem.IsComplete);
    }
}
```

3.2. Update the code to use TodoItemDTO

```
using Microsoft.EntityFrameworkCore;
using TodoApi.Models;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
builder.Services.AddDbContext<TodoDb>(opt => opt.UseInMemoryDatabase("TodoList"));
var app = builder.Build();

app.MapGet("/todoitems", async (TodoDb db) =>
    await db.Todos.Select(x => new TodoItemDTO(x)).ToListAsync());

app.MapGet("/todoitems/{id}", async (int id, TodoDb db) =>
    await db.Todos.FindAsync(id)
        is Todo todo
            ? Results.Ok(new TodoItemDTO(todo))
            : Results.NotFound());

app.MapPost("/todoitems", async (TodoItemDTO todoItemDTO, TodoDb db) =>
{
    var todoItem = new Todo
    {
        IsComplete = todoItemDTO.IsComplete,
        Name = todoItemDTO.Name
    };

    db.Todos.Add(todoItem);
    await db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todoItem.Id}", new TodoItemDTO(todoItem));
});

app.MapPut("/todoitems/{id}", async (int id, TodoItemDTO todoItemDTO, TodoDb db) =>
{
    var todo = await db.Todos.FindAsync(id);
```

```
    if (todo is null) return Results.NotFound();

    todo.Name = todoItemDTO.Name;
    todo.IsComplete = todoItemDTO.IsComplete;

    await db.SaveChangesAsync();

    return Results.NoContent();
});

app.MapDelete("/todoitems/{id}", async (int id, TodoDb db) =>
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
        await db.SaveChangesAsync();
        return Results.Ok(new TodoItemDTO(todo));
    }

    return Results.NotFound();
});

app.Run();
```

- **Update:**

```
public class Todo
{
    public int Id { get; set; }
    public string? Name { get; set; }
    public bool IsComplete { get; set; }
    public string? Secret { get; set; }
}

public class TodoItemDTO
{
    public int Id { get; set; }
    public string? Name { get; set; }
    public bool IsComplete { get; set; }

    public TodoItemDTO() { }
    public TodoItemDTO(Todo todoItem) =>
    (Id, Name, IsComplete) = (todoItem.Id, todoItem.Name, todoItem.IsComplete);
}


class TodoDb : DbContext
{
    public TodoDb(DbContextOptions<TodoDb> options)
        : base(options) { }

    public DbSet<Todo> Todos => Set<Todo>();
}
```

Verify you can't post or get the secret field

4. Use JsonOptions

The following code uses JsonOptions

```
using Microsoft.AspNetCore.Http.Json;

var builder = WebApplication.CreateBuilder(args);

// Configure JSON options
builder.Services.Configure<JsonOptions>(options =>
{
    options.SerializerOptions.IncludeFields = true;
});

var app = builder.Build();

app.MapGet("/", () => new Todo { Name = "Walk dog", IsComplete = false });

app.Run();

class Todo
{
    // These are public fields instead of properties.
    public string? Name;
    public bool IsComplete;
}
```

The following code uses JsonSerializerOptions:

```
using System.Text.Json;

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

var options = new JsonSerializerOptions(JsonSerializerDefaults.Web);

app.MapGet("/", () => Results.Json(new Todo
{
    Name = "Walk dog",
    IsComplete = false
}, options));

app.Run();

class Todo
{
    public string? Name { get; set; }
    public bool IsComplete { get; set; }
}
```

The preceding code uses web defaults (Web defaults for JsonSerializerOptions), which converts property names to camel case.

SUBMIT YOUR CODE TO GIT!

- - - END LAB 05 - - -

LAB 6
Controller-based APIs

I. TARGET

- ✓ Creating web APIs using controllers or using minimal APIs.
- ✓ Use controllers for handling web API requests.

II. REFERENCES

- ✓ Migrate Minimal APIs To Controller Based APIs, https://www.binaryintellect.net/articles/78af5609-725a-42c4-b17e-99f0dd6256f4.aspx.
- ✓ Web API Controllers, https://www.tutorialsteacher.com/webapi/web-api-controller,
- ✓ Create web APIs with ASP.NET Core, https://www.tutorialsteacher.com/webapi/web-api-controller

III. REQUIREMENTS

- ✓ Create a web API project.
- ✓ Add a model class and a database context.
- ✓ Scaffold a controller with CRUD methods.
- ✓ Configure routing, URL paths, and return values.
- ✓ Call the web API with http-repl.

IV. IN LAB

ASP.NET Core supports creating web APIs using controllers or using minimal APIs. Controllers in a web API are classes that derive from ControllerBase. This Lab shows how to use controllers for handling web API requests.

1. Overview

1.1. ControllerBase class

A controller-based web API consists of one or more controller classes that derive from ControllerBase.

The web API project template provides a starter controller:

```
[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
```

1.2. Attributes

The Microsoft.AspNetCore.Mvc namespace provides attributes that can be used to configure the behavior of web API controllers and action methods. The following example uses attributes to specify the supported HTTP action verb and any known HTTP status codes that could be returned:

```
[HttpPost]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public ActionResult<Pet> Create(Pet pet)
{
    pet.Id = _petsInMemoryStore.Any() ?
             _petsInMemoryStore.Max(p => p.Id) + 1 : 1;
    _petsInMemoryStore.Add(pet);

    return CreatedAtAction(nameof(GetById), new { id = pet.Id }, pet);
}
```

Here are some more examples of attributes that are available.

Attribute	Notes
[Route]	Specifies URL pattern for a controller or action.
[Bind]	Specifies prefix and properties to include for model binding.
[HttpGet]	Identifies an action that supports the HTTP GET action verb.
[Consumes]	Specifies data types that an action accepts.
[Produces]	Specifies data types that an action returns.

1.3. ApiController attribute

- **Attribute on specific controllers**

The [**ApiController**] attribute can be applied to specific controllers, as in the following example from the project template:

```
[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
```

- **Attribute on multiple controllers**

One approach to using the attribute on more than one controller is to create a custom base controller class annotated with the [ApiController] attribute. The following example shows a custom base class and a controller that derives from it:

```
[ApiController]
public class MyControllerBase : ControllerBase
{
}
```

And

```
[Produces(MediaTypeNames.Application.Json)]
[Route("[controller]")]
public class PetsController : MyControllerBase
```

1.4. Example

The [**Consumes**] attribute allows an action to limit the supported request content types. Apply the [**Consumes**] attribute to an action or controller, specifying one or more content types:

```
[ApiController]
[Route("api/[controller]")]
public class ConsumesController : ControllerBase
{
    [HttpPost]
    [Consumes("application/json")]
    public IActionResult PostJson(IEnumerable<int> values) =>
        Ok(new { Consumes = "application/json", Values = values });

    [HttpPost]
    [Consumes("application/x-www-form-urlencoded")]
    public IActionResult PostForm([FromForm] IEnumerable<int> values) =>
        Ok(new { Consumes = "application/x-www-form-urlencoded", Values = values });
}
```

In the preceding code, **ConsumesController** is configured to handle requests sent to the ***https://localhost:5001/api/Consumes*** URL. Both of the controller's actions, **PostJson** and **PostForm**, handle **POST** requests with the same URL. Without the [**Consumes**] attribute applying a type constraint, an ambiguous match exception is thrown.

The [**Consumes**] attribute is applied to both actions. The **PostJson** action handles requests sent with a **Content-Type** header of application/json. The PostForm action handles requests sent with a **Content-Type** header of **application/x-www-form-urlencoded.**

2. Create a web API with ASP.NET Core

Learn how to:

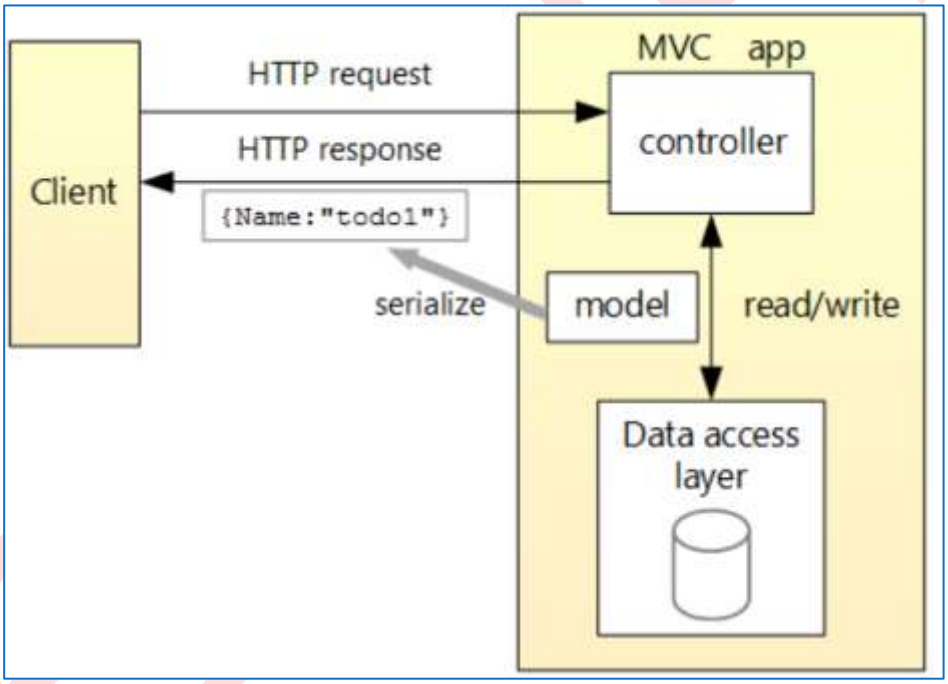
- Create a web API project.
- Add a model class and a database context.

- Scaffold a controller with CRUD methods.
- Configure routing, URL paths, and return values.
- Call the web API with http-repl.

At the end, you have a web API that can manage "to-do" items stored in a database. This section creates the following API:

API	Description	Request body	Response body
GET /api/todoitems	Get all to-do items	None	Array of to-do items
GET /api/todoitems/{id}	Get an item by ID	None	To-do item
POST /api/todoitems	Add a new item	To-do item	To-do item
PUT /api/todoitems/{id}	Update an existing item	To-do item	None
DELETE /api/todoitems/{id}	Delete an item	None	None

The following diagram shows the design of the app.



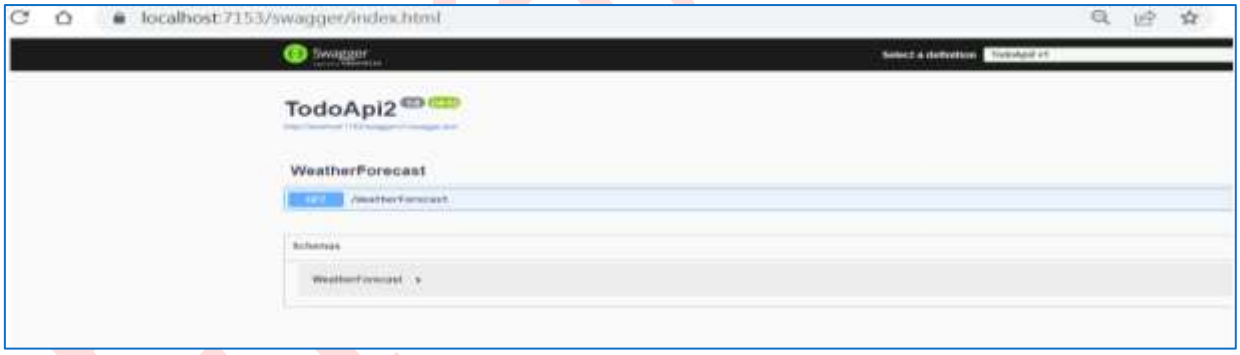
2.1. Create a web project

- From the **File menu**, select **New > Project.**
- Enter **Web API** in the search box.
- Select the **ASP.NET Core Web API** template and select **Next**.
- In the Configure your new project dialog, name the project **TodoApi** and select **Next**.
- In the Additional information dialog:

- ✓ Confirm the **Framework is .NET 6.0 (Long-term support).**
- ✓ Confirm the checkbox for **Use controllers(uncheck to use minimal APIs)** is checked.
- ✓ Select **Create**

2.2. Test the project

Press **Ctrl+F5** to run without the debugger:



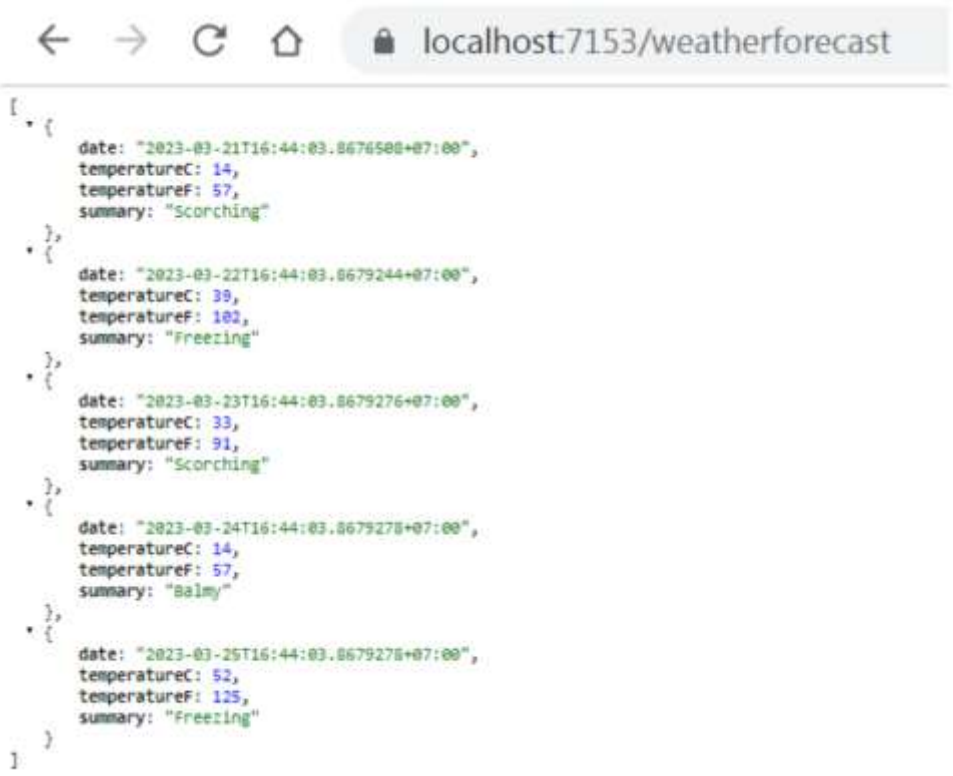
The Swagger page /swagger/index.html is displayed. Select GET > Try it out > Execute. The page displays:

- The Curl command to test the WeatherForecast API.
- The URL to test the WeatherForecast API.
- The response code, body, and headers.
- A drop-down list box with media types and the example value and schema.

➢ Swagger is used to generate useful documentation and help pages for web APIs.

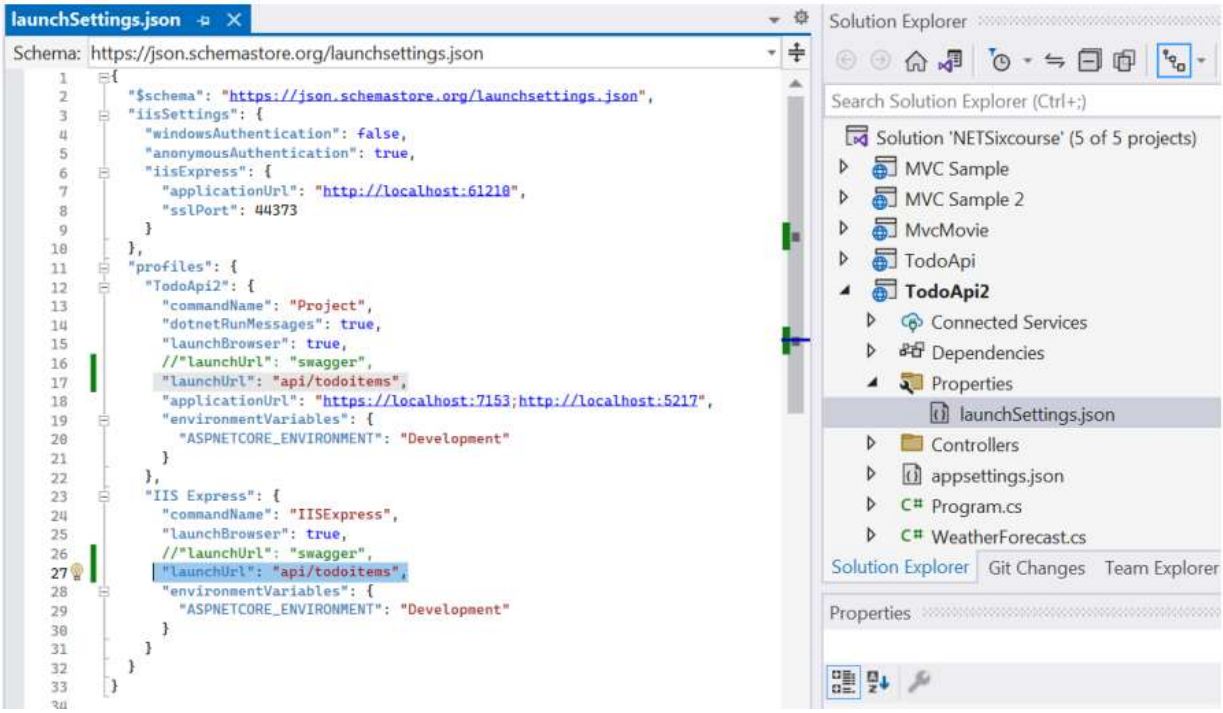
➢ Copy and paste the Request URL in the browser: https://localhost:/weatherforecast

➢ JSON similar to the following example is returned:



2.3. Update the launchUrl

In **Properties\launchSettings.json**, update **launchUrl** from "**swagger**" to "**api/todoitems**":



Because Swagger will be removed, the preceding markup changes the URL that is launched to the GET method of the controller added in the following sections.

2.4. Add a model class

A model is a set of classes that represent the data that the app manages. The model for this app is a single TodoItem class.

- In Solution Explorer, right-click the project. Select Add > New Folder. Name the folder Models.
- Right-click the Models folder and select Add > Class. Name the class TodoItem and select Add.
- Replace the template code with the following:

```
namespace TodoApi2.Models
{
    public class TodoItem
    {
        public long Id { get; set; }
        public string? Name { get; set; }
        public bool IsComplete { get; set; }
    }
}
```

The **Id** property functions as the unique key in a relational database. Model classes can go anywhere in the project, but the **Models** folder is used by convention.

2.5. Add a database context

The database context is the main class that coordinates Entity Framework functionality for a data model. This class is created by deriving from the **Microsoft.EntityFrameworkCore.DbContext** class.

Add NuGet packages

- From the Tools menu, select ***NuGet Package Manager > Manage NuGet Packages for Solution.***
- Select the ***Browse*** tab, and then enter ***Microsoft.EntityFrameworkCore.InMemory*** in the search box.
- Select ***Microsoft.EntityFrameworkCore.InMemory*** in the left pane.
- Select the ***Project checkbox*** in the right pane and then select ***Install*** (version 6.0.15)

Add the TodoContext database context

- Right-click the Models folder and select Add > Class. Name the class TodoContext and click Add.
- Enter the following code:

```
namespace TodoApi2.Models
{
    public class TodoItem
    {
        public long Id { get; set; }
        public string? Name { get; set; }
        public bool IsComplete { get; set; }
    }
}
```

2.6. Register the database context

In ASP.NET Core, services such as the DB context must be registered with the dependency injection (DI) container. The container provides the service to controllers. Update **Program.cs** with the following code:

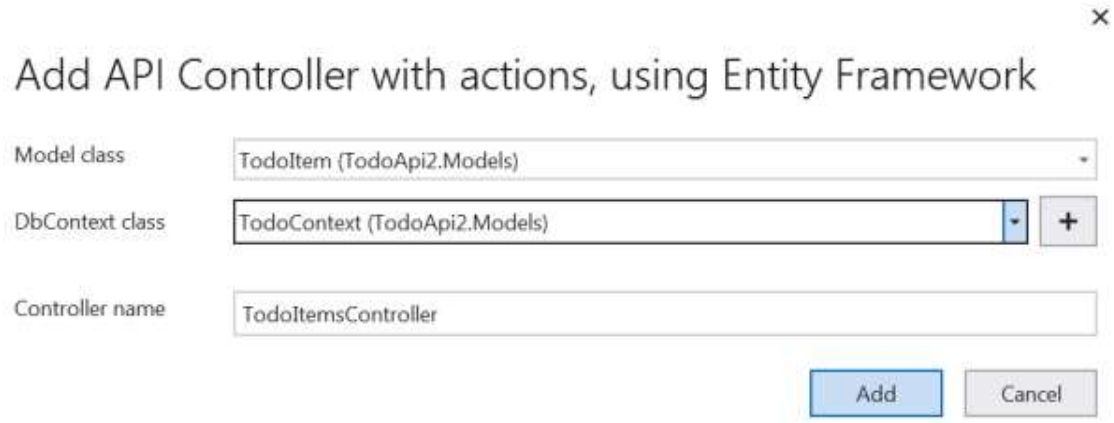
```
using Microsoft.EntityFrameworkCore;
using TodoApi2.Models;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();
// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();

//builder.Services.AddSwaggerGen(c =>
//{
//    c.SwaggerDoc("v1", new() { Title = "TodoApi", Version = "v1" });
//});
builder.Services.AddDbContext<TodoContext>(opt =>
    opt.UseInMemoryDatabase("TodoList"));

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    //app.UseSwagger();
    //app.UseSwaggerUI();
    //or //app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json", "TodoApi v1"));
}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

The preceding code:

- Removes the Swagger calls.
- Removes unused using directives.
- Adds the database context to the DI container.
- Specifies that the database context will use an in-memory database.

2.7. Scaffold a controller

- Right-click the Controllers folder.
- Select Add > New Scaffolded Item.
- Select **API Controller with actions, using Entity Framework**, and then select **Add**.
- In the Add API Controller with actions, using Entity Framework dialog:
  - Select **TodoItem (TodoApi.Models)** in the **Model** class.
  - Select **TodoContext (TodoApi.Models)** in the **Data context** class.
  - Select **Add**.



If the scaffolding operation fails, select Add to try scaffolding a second time. Or Open csproj file to confirm all packages version is 6.x.x, try add again.

The generated code:

- Marks the class with the [ApiController] attribute. This attribute indicates that the controller responds to web API requests.
- Uses DI to inject the database context (TodoContext) into the controller. The database context is used in each of the CRUD methods in the controller

The ASP.NET Core templates for:

- Controllers with views include [**action**] in the route template.
- API controllers don't include [**action**] in the route template.

When the [**action**] token isn't in the route template, the action name is excluded from the route. That is, the action's associated method name isn't used in the matching route.

2.8. Update the PostTodoItem create method

Update the return statement in the **PostTodoItem** to use the nameof operator:

```
[HttpPost]
public async Task<ActionResult<TodoItem>> PostTodoItem(TodoItem todoItem)
{
  if (_context.TodoItems == null)
  {
      return Problem("Entity set 'TodoContext.TodoItems'  is null.");
  }
    _context.TodoItems.Add(todoItem);
    await _context.SaveChangesAsync();

   // return CreatedAtAction("GetTodoItem", new { id = todoItem.Id }, todoItem);
    return CreatedAtAction(nameof(GetTodoItem), new { id = todoItem.Id }, todoItem);
}
```

The preceding code is an **HTTP POST** method, as indicated by the [**HttpPost**] attribute. The method gets the value of the to-do item from the body of the HTTP request.

The CreatedAtAction method:

- Returns an HTTP 201 status code if successful. HTTP 201 is the standard response for an HTTP POST method that creates a new resource on the server.
- Adds a Location header to the response. The **Location** header specifies the URI of the newly created to-do item. For more information, see 10.2.2 201 Created.
- References the **GetTodoItem** action to create the **Location** header's URI. The C# nameof keyword is used to avoid hard-coding the action name in the **CreatedAtAction** call.

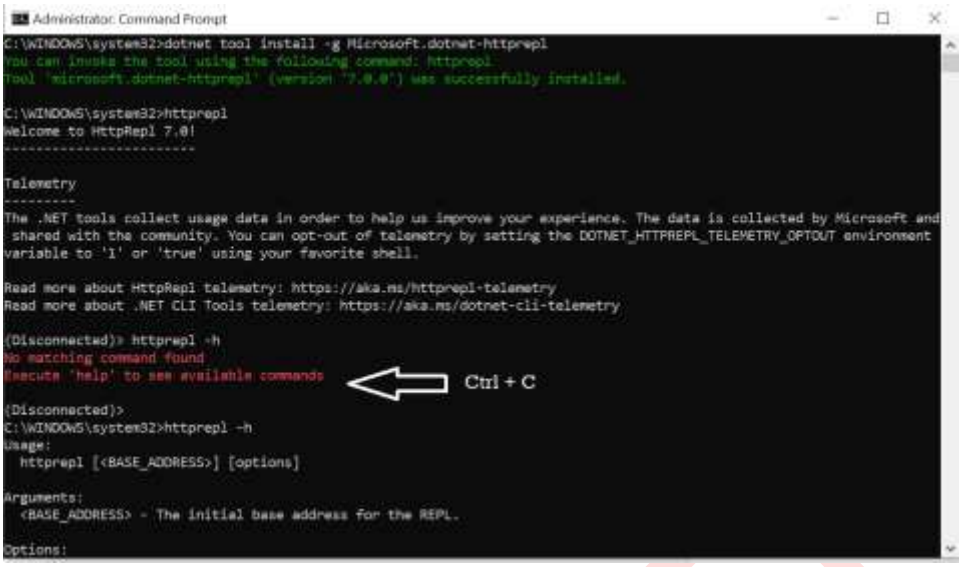
2.9. Test web APIs with the HttpRepl

The HTTP Read-Eval-Print Loop (REPL) is:

- A lightweight, cross-platform command-line tool that's supported everywhere .NET Core is supported.
- Used for making HTTP requests to test ASP.NET Core web APIs (and non-ASP.NET Core web APIs) and view their results.
- Capable of testing web APIs hosted in any environment, including localhost and Azure App Service.

To install the HttpRepl, run the following command (Open CMD (by Administrator)):

dotnet tool install -g Microsoft.dotnet-httprepl

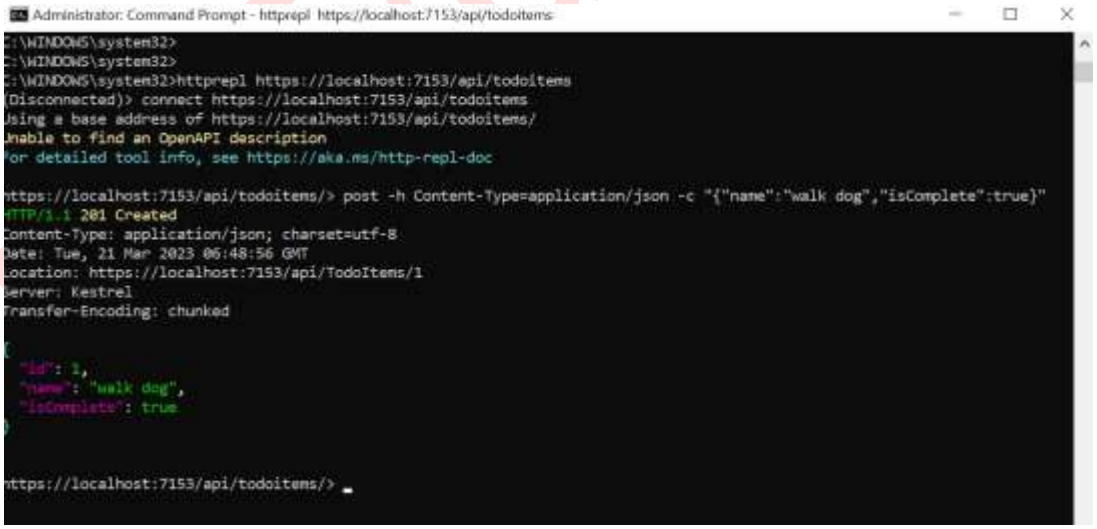


2.10. Test PostTodoItem

Press **Ctrl+F5** to run the app. Open a new terminal window, and run the following commands. If your app uses a different port number, replace 5001 in the httprepl command with your port number.

```
httprepl https://localhost:5001/api/todoitems
post -h Content-Type=application/json -c "{"name":"walk dog","isComplete":true}"
```

Here's an example of the output from the command:



2.11. Test the location header URI

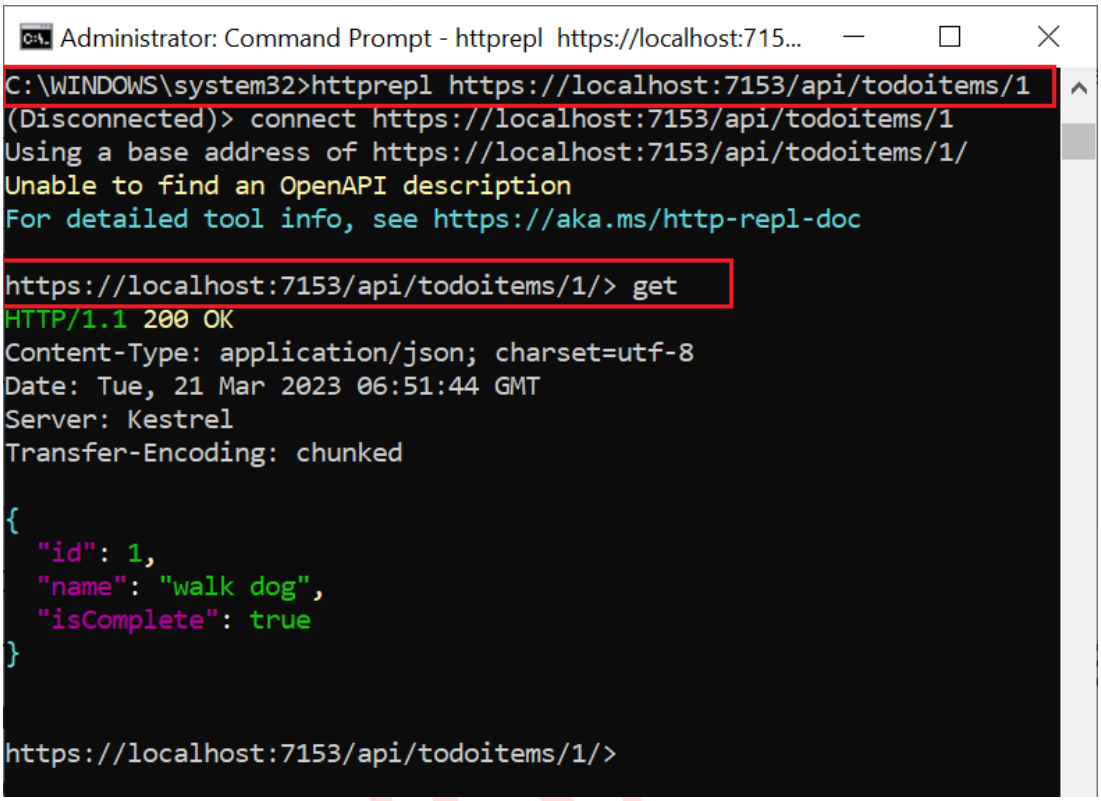
To test the location header, copy and paste it into an httprepl get command.

The following example assumes that you're still in an httprepl session. If you ended the previous httprepl session, replace connect with httprepl in the following commands:

connect https://localhost:5001/api/todoitems/1

get

Here's an example of the output from the command:



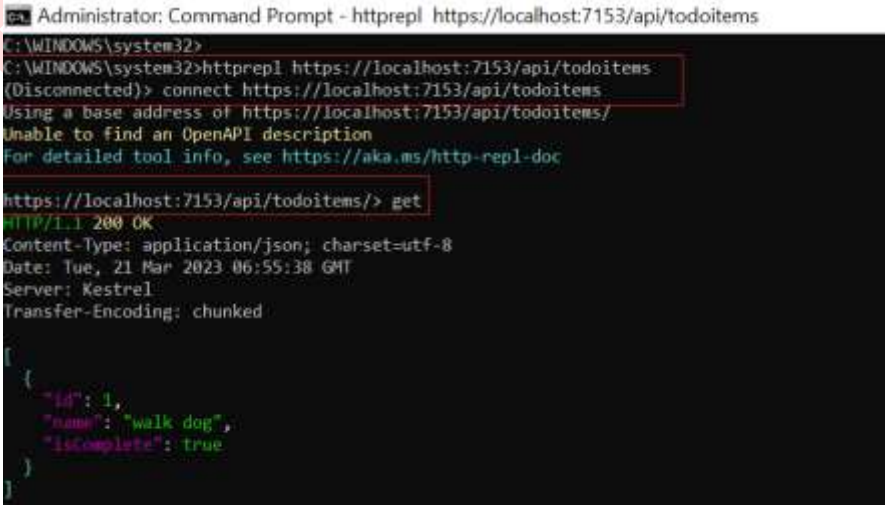
2.12. Examine the GET methods

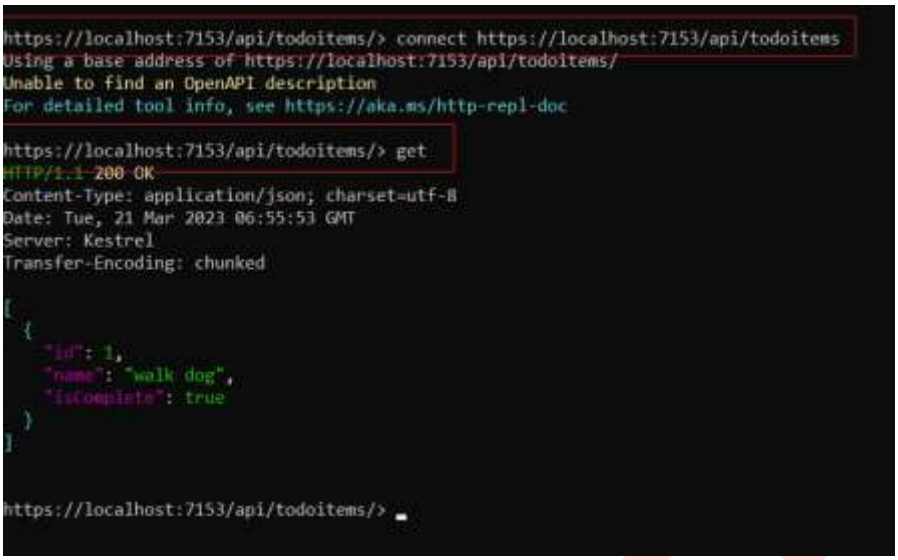
Two GET endpoints are implemented:

✓ GET /api/todoitems

✓ GET /api/todoitems/{id}

You just saw an example of the /api/todoitems/{id} route. Test the /api/todoitems route





This time, the JSON returned is an array of one item.

This app uses an in-memory database. If the app is stopped and started, the preceding GET request will not return any data. If no data is returned, POST data to the app.

2.13. Routing and URL paths

The [HttpGet] attribute denotes a method that responds to an HTTP GET request. The URL path for each method is constructed as follows:

- Start with the template string in the controller's Route attribute:

```
[Route("api/[controller]")]
[ApiController]
public class TodoItemsController : ControllerBase
```

- Replace [**controller**] with the name of the controller, which by convention is the controller class name minus the "Controller" suffix. For this sample, the controller class name is **TodoItems**Controller, so the controller name is "TodoItems". ASP.NET Core routing is case insensitive.
- If the [**HttpGet**] attribute has a route template (for example, **[HttpGet("products")]),** append that to the path. This sample doesn't use a template.

In the following **GetTodoItem** method, "{**id**}" is a placeholder variable for the unique identifier of the to-do item. When **GetTodoItem** is invoked, the value of "{**id**}" in the URL is provided to the method in its id parameter

```
// GET: api/TodoItems/5
  [HttpGet("{id}")]
  public async Task<ActionResult<TodoItem>> GetTodoItem(long id)
  {
    if (_context.TodoItems == null)
    {
        return NotFound();
    }
      var todoItem = await _context.TodoItems.FindAsync(id);

      if (todoItem == null)
      {
          return NotFound();
      }

      return todoItem;
  }
```

- **Return values**

The return type of the **GetTodoItems** and **GetTodoItem** methods is ActionResult<T> type.

ASP.NET Core automatically serializes the object to JSON and writes the JSON into the body of the response message. The response code for this return type is 200 OK, assuming there are no unhandled exceptions. Unhandled exceptions are translated into 5xx errors.

ActionResult return types can represent a wide range of HTTP status codes. For example, **GetTodoItem** can return two different status values:

- If no item matches the requested ID, the method returns a 404 status NotFound error code.
- Otherwise, the method returns 200 with a JSON response body. Returning item results in an HTTP 200 response.

2.14. The PutTodoItem method

- Examine the **PutTodoItem** method:

```
[HttpPut("{id}")]
  public async Task<IActionResult> PutTodoItem(long id, TodoItem todoItem)
  {
      if (id != todoItem.Id)
      {
          return BadRequest();
      }

      _context.Entry(todoItem).State = EntityState.Modified;

      try
      {
          await _context.SaveChangesAsync();
      }
```

```
    catch (DbUpdateConcurrencyException)
    {
        if (!TodoItemExists(id))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return NoContent();
}
```

PutTodoItem is similar to **PostTodoItem**, except it uses HTTP PUT. The response is 204 (No Content). According to the HTTP specification, a PUT request requires the client to send the entire updated entity, not just the changes. To support partial updates, use HTTP PATCH.

If you get an error calling PutTodoItem in the following section, call GET to ensure there's an item in the database.

- **Test the PutTodoItem method**

This sample uses an in-memory database that must be initialized each time the app is started. There must be an item in the database before you make a PUT call. Call GET to ensure there's an item in the database before making a PUT call.

Update the to-do item that has Id = 1 and set its name to "feed fish":

```
connect https://localhost:5001/api/todoitems/1
put -h Content-Type=application/json -c "{"id":1,"name":"feed fish","isComplete":true}"
```

Here's an example of the output from the command:

```
HTTP/1.1 204 No Content
Date: Tue, 14 Mar 2022 21:20:47 GMT
Server: Kestrel
```

2.15. The DeleteTodoItem method

- Examine the **DeleteTodoItem** method:

```
[HttpDelete("{id}")]
public async Task<IActionResult> DeleteTodoItem(long id)
{
    if (_context.TodoItems == null)
    {
        return NotFound();
    }
```

```
    var todoItem = await _context.TodoItems.FindAsync(id);
    if (todoItem == null)
    {
        return NotFound();
    }

    _context.TodoItems.Remove(todoItem);
    await _context.SaveChangesAsync();

    return NoContent();
}
```

- **Test the DeleteTodoItem method**

Delete the to-do item that has Id = 1:

```
connect https://localhost:5001/api/todoitems/1
delete
```

Here's an example of the output from the command:

```
HTTP/1.1 204 No Content
Date: Tue, 14 Mar 2023 21:43:00 GMT
Server: Kestrel
```

2.16. Prevent over-posting

Currently the sample app exposes the entire **TodoItem** object. Production apps typically limit the data that's input and returned using a subset of the model. There are multiple reasons behind this, and security is a major one. The subset of a model is usually referred to as a Data Transfer Object (DTO), input model, or view model. **DTO** is used in this Lab.

A DTO may be used to:

- ✓ Prevent over-posting.
- ✓ Hide properties that clients are not supposed to view.
- ✓ Omit some properties in order to reduce payload size.
- ✓ Flatten object graphs that contain nested objects. Flattened object graphs can be more convenient for clients.

To demonstrate the DTO approach, update the **TodoItem** class to include a secret field:

```
namespace TodoApi2.Models
{
    public class TodoItem
    {
        public long Id { get; set; }
        public string? Name { get; set; }
        public bool IsComplete { get; set; }
        public string? Secret { get; set; }
    }
}
```

The secret field needs to be hidden from this app, but an administrative app could choose to expose it.

Verify you **can** post and get the secret field.

- **Create a DTO model**

```
namespace TodoApi2.Models
{
    public class TodoItemDTO
    {
        public long Id { get; set; }
        public string? Name { get; set; }
        public bool IsComplete { get; set; }
    }
}
```

Update the **TodoItemsController** to use **TodoItemDTO**:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using TodoApi2.Models;

namespace TodoApi2.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class TodoItemsController : ControllerBase
    {
        private readonly TodoContext _context;

        public TodoItemsController(TodoContext context)
        {
            _context = context;
        }

        // GET: api/TodoItems
        [HttpGet]
        public async Task<ActionResult<IEnumerable<TodoItemDTO>>> GetTodoItems()
        {
          if (_context.TodoItems == null)
          {
              return NotFound();
          }
            //return await _context.TodoItems.ToListAsync();
            return await _context.TodoItems
                .Select(x => ItemToDTO(x))
                .ToListAsync();
        }
```

```
// GET: api/TodoItems/5
[HttpGet("{id}")]
public async Task<ActionResult<TodoItemDTO>> GetTodoItem(long id)
{
  if (_context.TodoItems == null)
  {
      return NotFound();
  }
    var todoItem = await _context.TodoItems.FindAsync(id);
    if (todoItem == null)
    {
        return NotFound();
    }

    //return todoItem;
    return ItemToDTO(todoItem);
}

// PUT: api/TodoItems/5
// To protect from overposting attacks, see https://go.microsoft.com/fwlink/?linkid=2123754
[HttpPut("{id}")]
public async Task<IActionResult> PutTodoItem(long id, TodoItem todoItem)
{
    if (id != todoItem.Id)
    {
        return BadRequest();
    }

    _context.Entry(todoItem).State = EntityState.Modified;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!TodoItemExists(id))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return NoContent();
}

// POST: api/TodoItems
// To protect from overposting attacks, see https://go.microsoft.com/fwlink/?linkid=2123754
[HttpPost]
public async Task<ActionResult<TodoItem>> PostTodoItem(TodoItem todoItem)
{
  if (_context.TodoItems == null)
  {
      return Problem("Entity set 'TodoContext.TodoItems'  is null.");
  }
    _context.TodoItems.Add(todoItem);
    await _context.SaveChangesAsync();

    // return CreatedAtAction("GetTodoItem", new { id = todoItem.Id }, todoItem);
    return CreatedAtAction(nameof(GetTodoItem), new { id = todoItem.Id }, todoItem);
}

// PUT: api/TodoItems/5
// To protect from overposting attacks, see https://go.microsoft.com/fwlink/?linkid=2123754
[HttpPut("{id}")]
public async Task<IActionResult> UpdateTodoItem(long id, TodoItemDTO todoItemDTO)
{
    if (id != todoItemDTO.Id)
    {
        return BadRequest();
    }
```

```
            var todoItem = await _context.TodoItems.FindAsync(id);
            if (todoItem == null)
            {
                return NotFound();
            }

            todoItem.Name = todoItemDTO.Name;
            todoItem.IsComplete = todoItemDTO.IsComplete;

            try
            {
                await _context.SaveChangesAsync();
            }
            catch (DbUpdateConcurrencyException) when (!TodoItemExists(id))
            {

                return NotFound();
            }

            return NoContent();
        }
        // POST: api/TodoItems
        // To protect from overposting attacks, see https://go.microsoft.com/fwlink/?linkid=2123754
        [HttpPost]
        public async Task<ActionResult<TodoItemDTO>> CreateTodoItem(TodoItemDTO todoItemDTO)
        {
            var todoItem = new TodoItem
            {
                IsComplete = todoItemDTO.IsComplete,
                Name = todoItemDTO.Name
            };

            _context.TodoItems.Add(todoItem);
            await _context.SaveChangesAsync();

            return CreatedAtAction(
                nameof(GetTodoItem),
                new { id = todoItem.Id },
                ItemToDTO(todoItem));
        }
        // DELETE: api/TodoItems/5
        [HttpDelete("{id}")]
        public async Task<IActionResult> DeleteTodoItem(long id)
        {
            if (_context.TodoItems == null)
            {
                return NotFound();
            }
            var todoItem = await _context.TodoItems.FindAsync(id);
            if (todoItem == null)
            {
                return NotFound();
            }

            _context.TodoItems.Remove(todoItem);
            await _context.SaveChangesAsync();

            return NoContent();
        }
        private bool TodoItemExists(long id)
        {
            return (_context.TodoItems?.Any(e => e.Id == id)).GetValueOrDefault();
        }
        private static TodoItemDTO ItemToDTO(TodoItem todoItem) =>
            new TodoItemDTO
            {
                Id = todoItem.Id,
                Name = todoItem.Name,
                IsComplete = todoItem.IsComplete
            };
    }
}
```

Verify you **can't** post or get the secret field.

3. Call the web API with JavaScript

In this section, you'll add an HTML page containing forms for creating and managing to-do items. Event handlers are attached to elements on the page. The event handlers result in HTTP requests to the web API's action methods. The Fetch API's fetch function initiates each HTTP request.

The **fetch** function returns a Promise object, which contains an HTTP response represented as a Response object. A common pattern is to extract the JSON **response** body by invoking the **json** function on the **Response** object. JavaScript updates the page with the details from the web API's response.

The simplest fetch call accepts a single parameter representing the route. A second parameter, known as the init object, is optional. init is used to configure the HTTP request.

Configure the app to serve static files and enable default file mapping. The following highlighted code is needed in **Program.cs**:

```
using Microsoft.EntityFrameworkCore;
using TodoApi2.Models;


var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();
// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();

//builder.Services.AddSwaggerGen(c =>
//{
//    c.SwaggerDoc("v1", new() { Title = "TodoApi", Version = "v1" });
//});
builder.Services.AddDbContext<TodoContext>(opt =>
    opt.UseInMemoryDatabase("TodoList"));

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    //app.UseSwagger();
    //app.UseSwaggerUI();
    //or //app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json", "TodoApi v1"));
}

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

app.UseAuthorization();

app.MapControllers();

app.Run();
```

- Create a wwwroot folder in the project root.
- Create a css folder inside of the wwwroot folder.

- Create a js folder inside of the wwwroot folder.
- Add an HTML file named **index.html** to the wwwroot folder (Add →New Item →HTML Page). Replace the contents of **index.html** with the following markup:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>To-do CRUD</title>
    <link rel="stylesheet" href="css/site.css" />
</head>
<body>
    <h1>To-do CRUD</h1>
    <h3>Add</h3>
    <form action="javascript:void(0);" method="POST" onsubmit="addItem()">
        <input type="text" id="add-name" placeholder="New to-do">
        <input type="submit" value="Add">
    </form>

    <div id="editForm">
        <h3>Edit</h3>
        <form action="javascript:void(0);" onsubmit="updateItem()">
            <input type="hidden" id="edit-id">
            <input type="checkbox" id="edit-isComplete">
            <input type="text" id="edit-name">
            <input type="submit" value="Save">
            <a onclick="closeInput()" aria-label="Close">&#10006;</a>
        </form>
    </div>

    <p id="counter"></p>

    <table>
        <tr>
            <th>Is Complete?</th>
            <th>Name</th>
            <th></th>
            <th></th>
        </tr>
        <tbody id="todos"></tbody>
    </table>

    <script src="js/site.js" asp-append-version="true"></script>
    <script type="text/javascript">
        getItems();
    </script>
</body>
</html>
```

Add a CSS file named **site.css** to the wwwroot/css folder. Replace the contents of **site.css** with the following styles:

```
input[type='submit'], button, [aria-label] {
    cursor: pointer;
}

#editForm {
    display: none;
}

table {
    font-family: Arial, sans-serif;
    border: 1px solid;
    border-collapse: collapse;
}
```

```
th {
    background-color: #f8f8f8;
    padding: 5px;
}

td {
    border: 1px solid;
    padding: 5px;
}
```

Add a JavaScript file named **site.js** to the wwwroot/js folder. Replace the contents of **site.js** with the following code:

```
const uri = 'api/todoitems';
let todos = [];

function getItems() {
    fetch(uri)
        .then(response => response.json())
        .then(data => _displayItems(data))
        .catch(error => console.error('Unable to get items.', error));
}

function addItem() {
    const addNameTextbox = document.getElementById('add-name');

    const item = {
        isComplete: false,
        name: addNameTextbox.value.trim()
    };

    fetch(uri, {
        method: 'POST',
        headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(item)
    })
        .then(response => response.json())
        .then(() => {
            getItems();
            addNameTextbox.value = '';
        })
        .catch(error => console.error('Unable to add item.', error));
}

function deleteItem(id) {
    fetch('${uri}/${id}', {
        method: 'DELETE'
    })
        .then(() => getItems())
        .catch(error => console.error('Unable to delete item.', error));
}

function displayEditForm(id) {
    const item = todos.find(item => item.id === id);

    document.getElementById('edit-name').value = item.name;
    document.getElementById('edit-id').value = item.id;
    document.getElementById('edit-isComplete').checked = item.isComplete;
    document.getElementById('editForm').style.display = 'block';
}
```

```
function updateItem() {
    const itemId = document.getElementById('edit-id').value;
    const item = {
        id: parseInt(itemId, 10),
        isComplete: document.getElementById('edit-isComplete').checked,
        name: document.getElementById('edit-name').value.trim()
    };

    fetch('${uri}/${itemId}', {
        method: 'PUT',
        headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(item)
    })
        .then(() => getItems())
        .catch(error => console.error('Unable to update item.', error));

    closeInput();

    return false;
}

 function closeInput() {
     document.getElementById('editForm').style.display = 'none';
 }

 function _displayCount(itemCount) {
     const name = (itemCount === 1) ? 'to-do' : 'to-dos';

     document.getElementById('counter').innerText = '${itemCount} ${name}';
 }

 function _displayItems(data) {
     const tBody = document.getElementById('todos');
     tBody.innerHTML = '';

     _displayCount(data.length);

     const button = document.createElement('button');

     data.forEach(item => {
         let isCompleteCheckbox = document.createElement('input');
         isCompleteCheckbox.type = 'checkbox';
         isCompleteCheckbox.disabled = true;
         isCompleteCheckbox.checked = item.isComplete;

         let editButton = button.cloneNode(false);
         editButton.innerText = 'Edit';
         editButton.setAttribute('onclick', 'displayEditForm(${item.id})');

         let deleteButton = button.cloneNode(false);
         deleteButton.innerText = 'Delete';
         deleteButton.setAttribute('onclick', 'deleteItem(${item.id})');

         let tr = tBody.insertRow();

         let td1 = tr.insertCell(0);
         td1.appendChild(isCompleteCheckbox);
```

```
        let td2 = tr.insertCell(1);
        let textNode = document.createTextNode(item.name);
        td2.appendChild(textNode);

        let td3 = tr.insertCell(2);
        td3.appendChild(editButton);

        let td4 = tr.insertCell(3);
        td4.appendChild(deleteButton);
    });

    todos = data;
}
```

A change to the ASP.NET Core project's launch settings may be required to test the HTML page locally:

Open Properties\launchSettings.json.

Remove the **launchUrl** property to force the app to open at **index.html**—the project's default file.

This sample calls all of the CRUD methods of the web API. Following are explanations of the web API requests.

3.1. Get a list of to-do items

In the following code, an HTTP GET request is sent to the api/todoitems route:

```
function getItems() {
    fetch(uri)
        .then(response => response.json())
        .then(data => _displayItems(data))
        .catch(error => console.error('Unable to get items.', error));
}
```

When the web API returns a successful status code, the **_displayItems** function is invoked. Each to-do item in the array parameter accepted by **_displayItems** is added to a table with Edit and Delete buttons. If the web API request fails, an error is logged to the browser's console.

3.2. Add a to-do item

In the following code:

- An **item** variable is declared to construct an object literal representation of the to-do item.
- A Fetch request is configured with the following options:
  - ✓ **method**—specifies the POST HTTP action verb.

- ✓ **body**—specifies the JSON representation of the request body. The JSON is produced by passing the object literal stored in **item** to the JSON.stringify function.
- ✓ **headers**—specifies the **Accept** and **Content-Type** HTTP request headers. Both headers are set to **application/json** to specify the media type being received and sent, respectively.

– An HTTP POST request is sent to the api/todoitems route.

```
function addItem() {
    const addNameTextbox = document.getElementById('add-name');

    const item = {
        isComplete: false,
        name: addNameTextbox.value.trim()
    };

    fetch(uri, {
        method: 'POST',
        headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(item)
    })
        .then(response => response.json())

        .then(() => {
            getItems();
            addNameTextbox.value = '';
        })
        .catch(error => console.error('Unable to add item.', error));
}
```

When the web API returns a successful status code, the **getItems** function is invoked to update the HTML table. If the web API request fails, an error is logged to the browser's console.

3.3. Update a to-do item

Updating a to-do item is similar to adding one; however, there are two significant differences:

– The route is suffixed with the unique identifier of the item to update. For example, api/todoitems/1.

– The HTTP action verb is PUT, as indicated by the **method** option.

```
function updateItem() {
    const itemId = document.getElementById('edit-id').value;
    const item = {
        id: parseInt(itemId, 10),
        isComplete: document.getElementById('edit-isComplete').checked,
        name: document.getElementById('edit-name').value.trim()
    };

    fetch('${uri}/${itemId}', {
        method: 'PUT',
        headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(item)
    })
        .then(() => getItems())
        .catch(error => console.error('Unable to update item.', error));

    closeInput();

    return false;
}
```

3.4. Delete a to-do item

To delete a to-do item, set the request's **method** option to **DELETE** and specify the item's unique identifier in the URL.

```
function deleteItem(id) {
    fetch('${uri}/${id}', {
        method: 'DELETE'
    })
        .then(() => getItems())
        .catch(error => console.error('Unable to delete item.', error));
}
```

4. Differences between minimal APIs and APIs with controllers

- No support for filters: For example, no support for IAsyncAuthorizationFilter, IAsyncActionFilter, IAsyncExceptionFilter, IAsyncResultFilter, and IAsyncResourceFilter.
- No support for model binding, i.e. IModelBinderProvider, IModelBinder. Support can be added with a custom binding shim.
- No support for binding from forms. This includes binding IFormFile. We plan to add support for IFormFile in the future.
- No built-in support for validation, i.e. IModelValidator
- No support for application parts or the application model. There's no way to apply or build your own conventions.
- No built-in view rendering support. We recommend using Razor Pages for rendering views.

- No support for JsonPatch
- No support for OData
- No support for ApiVersioning. See this issue for more details.

5. Add authentication support to a web API (Optional)

ASP.NET Core Identity adds user interface (UI) login functionality to ASP.NET Core web apps. To secure web APIs and SPAs, use one of the following:

- Azure Active Directory
- Azure Active Directory B2C (Azure AD B2C)
- Duende Identity Server

Duende Identity Server is an OpenID Connect and OAuth 2.0 framework for ASP.NET Core. Duende Identity Server enables the following security features:

- Authentication as a Service (AaaS)
- Single sign-on/off (SSO) over multiple application types
- Access control for APIs
- Federation Gateway

Take a look by yourself:

https://viblo.asia/p/cung-tim-hieu-ve-http-request-methods-djeZ1xBoKWz

https://swagger.io/tools/swagger-ui/

https://docs.duendesoftware.com/identityserver/v6/overview/

SUBMIT YOUR CODE TO GIT!

- - - END LAB 06 - - -

LAB 7
ASP.NET Core MVC with EF Core

I. TARGET

- ✓ The Contoso University exemplar web application showcases the process of building an ASP.NET Core MVC web application using Entity Framework (EF) Core in conjunction with Visual Studio.
- ✓ The presented application represents a website dedicated to Contoso University, a fictional educational institution. Its features encompass student admissions, course generation, and instructor assignments.

II. REFERENCES

- ✓ ASP.NET Core MVC with EF Core - tutorial series, https://learn.microsoft.com/en-us/aspnet/core/data/ef-mvc/?view=aspnetcore-7.0
- ✓ Tutorial: Get started with EF Core in an ASP.NET MVC web app, https://learn.microsoft.com/en-us/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/creating-an-entity-framework-data-model-for-an-asp-net-mvc-application
- ✓ Code First Approach In ASP.NET Core MVC With EF Core Migration, https://www.c-sharpcorner.com/article/code-first-approach-in-asp-net-core-mvc-with-ef-core-migration/

III. REQUIREMENTS

- ✓ Create a project ASP.NET Core MVC with EF Core
- ✓ Create, update and sync the database with your model classes.
- ✓ Building some function Search, report...

IV. IN LAB

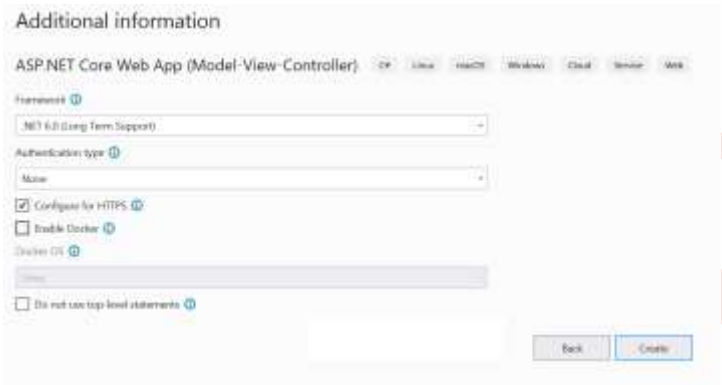
Installing the tools:

You have the option to install dotnet ef as either a global or local tool. The majority of developers choose to install it globally, and you can do this by executing the following command in CMD (Windows) or Terminal: dotnet tool install --global dotnet-ef

Verify installation: dotnet ef

1. Get started

Start Visual Studio → In the Add a new project dialog, select ASP.NET Core Web App (Model-View-Controller) → Next → Enter project name ContosoUniversity → Next.



Open **Views/Shared/_Layout.cshtml** and make the following changes:

- Change each occurrence of ContosoUniversity to Contoso University.
- Add menu entries for About, Students, Courses, Instructors, and Departments, and delete the Privacy menu entry.

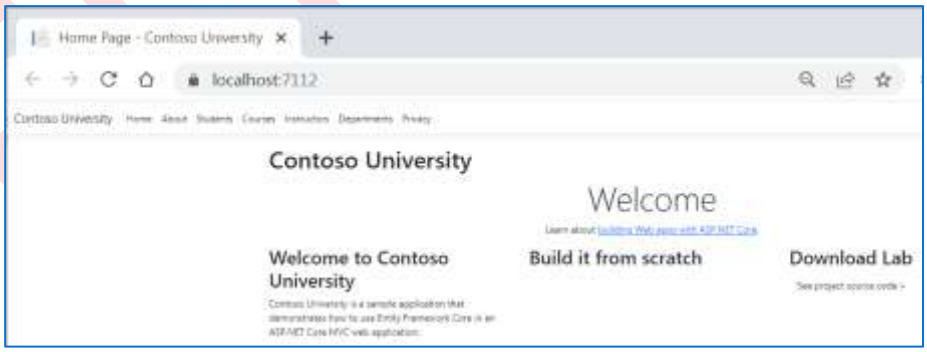
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Contoso University</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
    <link rel="stylesheet" href="~/css/site.css" asp-append-version="true" />
    <link rel="stylesheet" href="~/ContosoUniversity.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="Home" asp-action="Index">Contoso University</a>
                <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target=".navbar-collapse" aria-
controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Index">Home</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="About">About</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Students" asp-action="Index">Students</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Courses" asp-action="Index">Courses</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Instructors" asp-
action="Index">Instructors</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Departments" asp-
action="Index">Departments</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
                        </li>
```

```
                    </ul>
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2023 - Contoso University - <a asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
        </div>
    </footer>
    <script src="~/lib/jquery/dist/jquery.min.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
    @await RenderSectionAsync("Scripts", required: false)
</body>
</html>
```

In Views/Home/Index.cshtml, replace the contents of the file with the following markup:

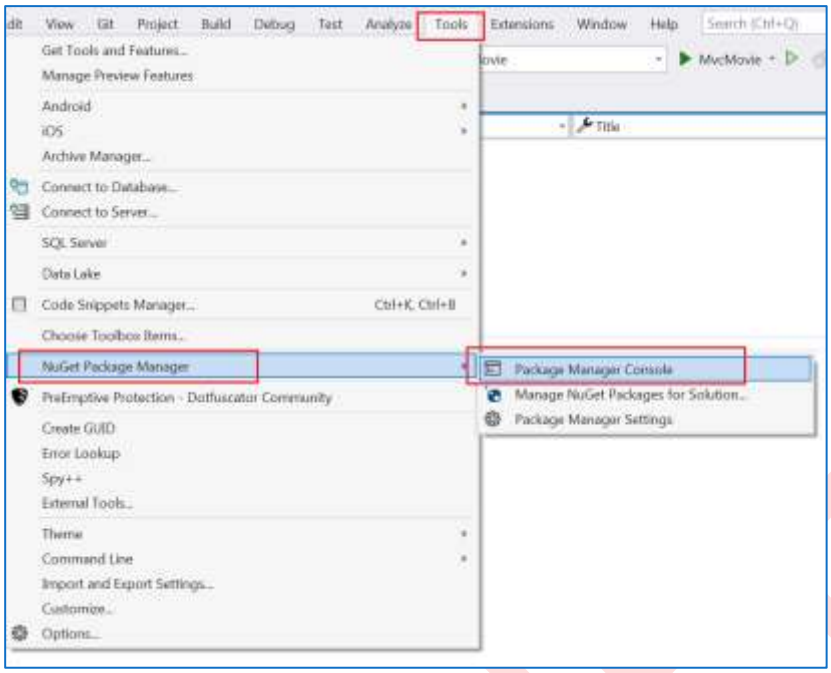
```
@{
    ViewData["Title"] = "Home Page";
}

<div class="jumbotron">
    <h1>Contoso University</h1>
</div>
<div class="text-center">
    <h1 class="display-4">Welcome</h1>
    <p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building Web apps with ASP.NET Core</a>.</p>
</div>

<div class="row">
    <div class="col-md-4">
        <h2>Welcome to Contoso University</h2>
        <p>
            Contoso University is a sample application that
            demonstrates how to use Entity Framework Core in an
            ASP.NET Core MVC web application.
        </p>
    </div>
    <div class="col-md-4">
        <h2>Build it from scratch</h2>

    </div>
    <div class="col-md-4">
        <h2>Download Lab</h2>
        <p><a class="btn btn-default" href="https://google.com">See project source code &raquo;</a></p>
    </div>
</div>
```

Press **CTRL+F5** to run the project or choose Debug > Start Without Debugging from the menu. The home page is displayed with tabs for the pages created in this Lab.



1.1. Add NuGet packages (Using CMD or GUI)

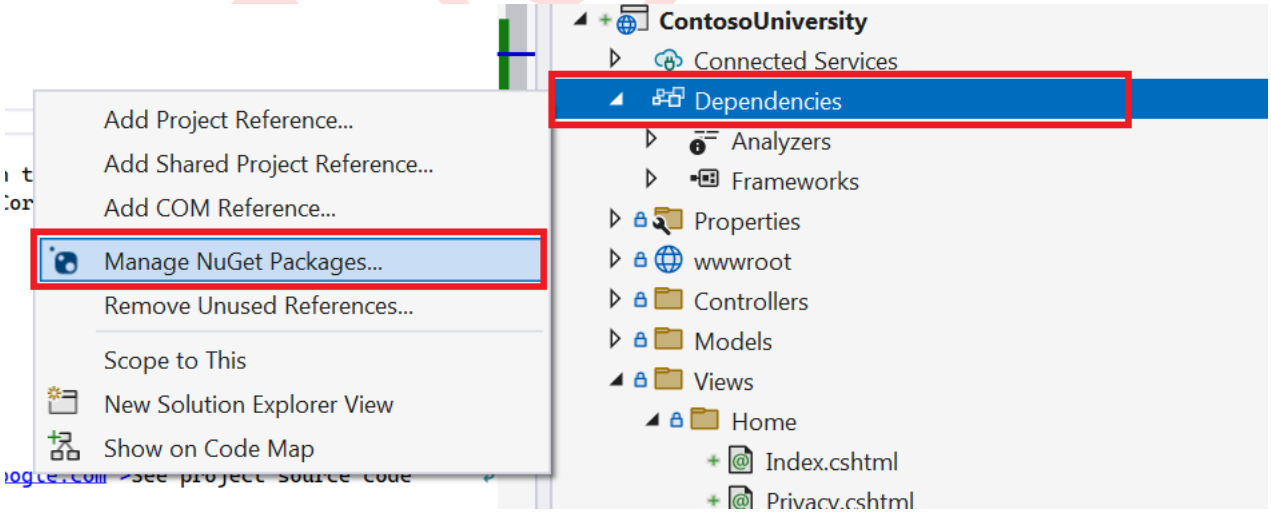
Using CMD: From the Tools menu, select ***NuGet Package Manager > Package Manager Console (PMC).***



In the PMC, run the following command:

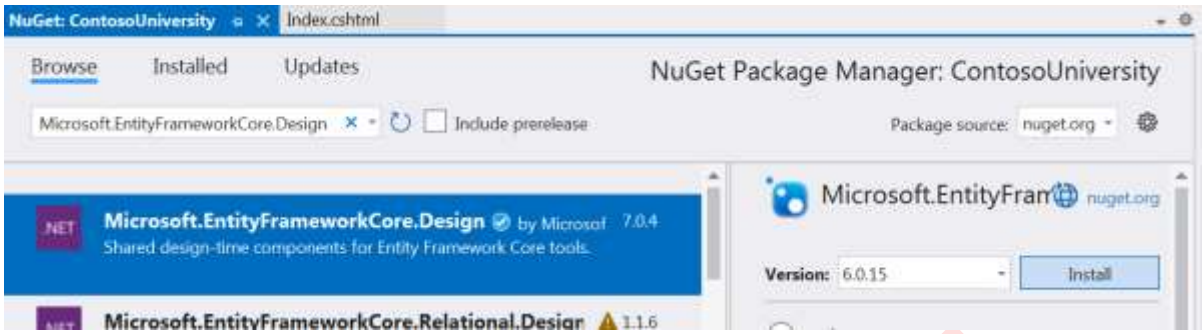
- *PM> **Install-Package Microsoft.EntityFrameworkCore.Design***
- *PM> **Install-Package Microsoft.EntityFrameworkCore.SqlServer***
- *PM> **Install-Package Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore***
- *PM> **Install-Package Microsoft.EntityFrameworkCore.Tools***

1.2. Using GUI

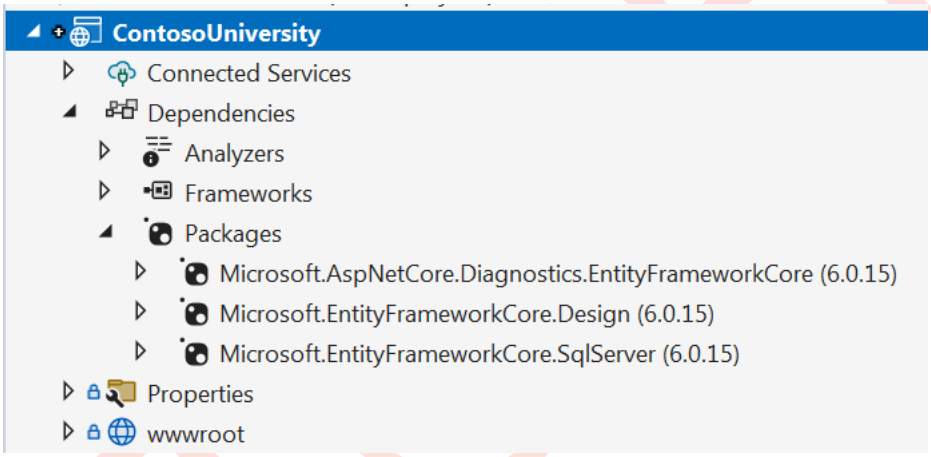


Enter following packages in Browse and Install them (version 6.x.xx)

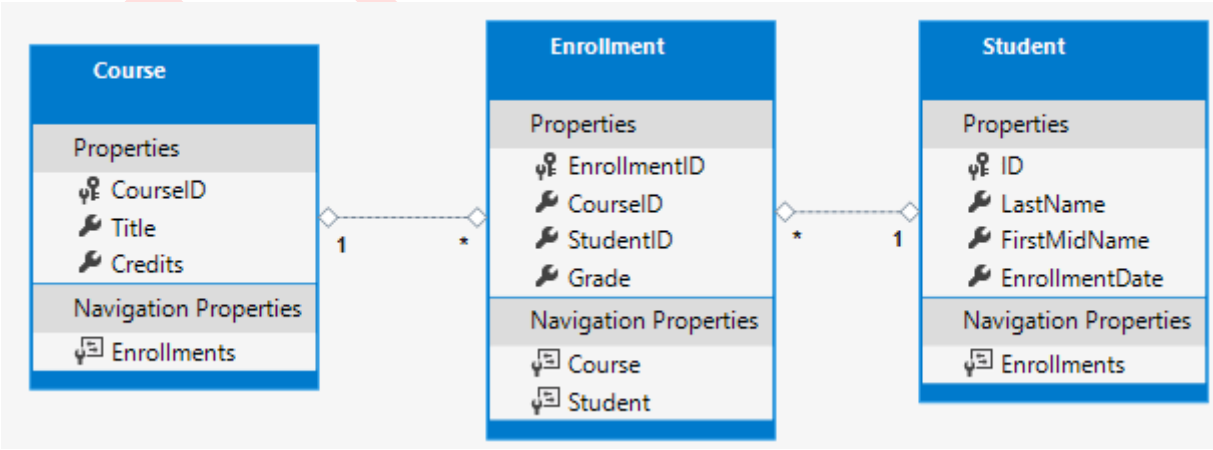
- *Microsoft.EntityFrameworkCore.Design*
- *Microsoft.EntityFrameworkCore.SqlServer*
- *Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore*
- *Microsoft.EntityFrameworkCore.Tools*



The ASP.NET Core middleware for EF Core error pages is facilitated by the Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore NuGet package. Its primary function is to identify and analyze errors related to EF Core migrations, aiding in their detection and diagnosis.



The following entity classes are created for this app:



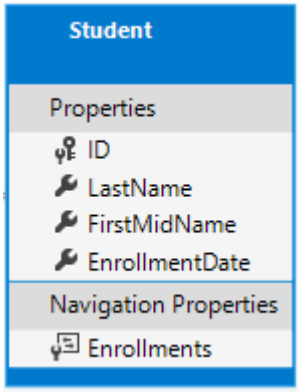
The preceding entities have the following relationships:

– The relationship between the Student and Enrollment entities is one-to-many, meaning that a single student can be enrolled in multiple courses simultaneously.

– Course and Enrollment have a one-to-many relationship, allowing any number of students to be enrolled in a course.

In the following sections, a class is created for each of these entities.

The Student entity



In the Models folder, create the Student class with the following code:

```
namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment>? Enrollments { get; set; }
    }
}
```

The ID property represents the primary key (PK) column of the database table associated with this class. In its default behavior, EF considers a property named either ID or classnameID as the primary key. For instance, the primary key could be denoted as StudentID instead of just ID.

The Enrollments property serves as a navigation property, holding additional entities associated with the main entity. In the context of the Student entity, it encapsulates all Enrollment entities connected to that particular student. For instance, if a Student entry in the database is linked to two Enrollment entries, the Enrollments navigation property of that Student entity will include these two Enrollment entities. In the Enrollment rows, a student's primary key (PK) value is stored in the foreign key (FK) column called StudentID.

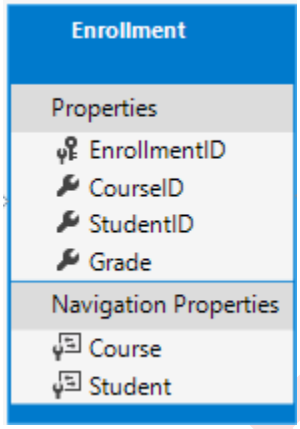
If a navigation property can hold multiple entities:

– The type must be a list, such as **ICollection<T>, List<T>**, or **HashSet<T>.**

– Entities can be **added**, **deleted**, and **updated**.

Many-to-many and one-to-many navigation relationships can contain multiple entities. When **ICollection<T>** is used, EF creates a **HashSet<T>** collection by default.

The Enrollment entity



In the Models folder, create the **Enrollment** class with the following code:

```
namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

The primary key for this entity is the EnrollmentID property, following the classnameID pattern instead of a standalone ID. While the Student entity adheres to the ID pattern, some developers advocate for maintaining uniformity throughout the data model. However, this Lab demonstrates that both patterns can be effectively utilized. In a subsequent Lab, we explore how using ID without the classname simplifies the implementation of inheritance in the data model.

The Grade property is defined as an enum with the Grade type declaration followed by a question mark, indicating its nullable nature. When the Grade is set to null, it signifies that the grade is either unknown or hasn't been assigned yet, distinct from a zero grade.

The StudentID property serves as a foreign key (FK) with its corresponding navigation property named Student. Each Enrollment entity is linked to one and only one Student entity, allowing the property to store just a single Student entity. It's essential to note that

this contrasts with the Student.Enrollments navigation property, which can accommodate multiple Enrollment entities.

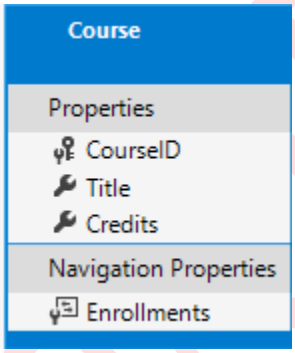
The CourseID property serves as a foreign key (FK) and is linked to the Course navigation property. Each Enrollment entity is connected to precisely one Course entity.

If a property in Entity Framework is named in the format of "<navigation property name><primary key property name>", it will be recognized as a foreign key property.

*For example, **StudentID** for the **Student** navigation property since the **Student** entity's PK is **ID**. FK properties can also be named < primary key property name >.*

*For example, **CourseID** because the **Course** entity's PK is CourseID.*

The Course entity



In the Models folder, create the **Course** class with the following code:

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

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        public int CourseID { get; set; }

        public string Title { get; set; }
        public int Credits { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The Enrollments property represents a navigation property, allowing a Course entity to establish associations with multiple Enrollment entities..

The utilization of the DatabaseGenerated attribute enables the manual entry of the primary key for the course, instead of relying on the database to automatically generate it.

1.3. Create the database context

The pivotal entity responsible for orchestrating EF capabilities concerning a specific data model is the DbContext database context class..

The SchoolContext class is crafted by inheriting from the Microsoft.EntityFrameworkCore.DbContext class. Within this DbContext derived class, the data model incorporates specific entities. Notably, various EF behaviors can be tailored to suit the project's requirements, empowering the SchoolContext to efficiently manage interactions with the underlying database.

In the project folder, create a folder named Data.

In the Data folder create a **SchoolContext** class with the following code:

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public   SchoolContext(DbContextOptions<SchoolContext>   options)   :
base(options)
        {

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

The preceding code creates a **DbSet** property for each entity set. In EF terminology:

- An entity set typically corresponds to a database table.
- An entity corresponds to a row in the table.

Omitting the DbSet<Enrollment> and DbSet<Course> statements would yield the same result, as EF would implicitly include them.:

- The **Student** entity references the **Enrollment** entity.
- The **Enrollment** entity references the **Course** entity.

After setting up the database, EF automatically generates tables with names identical to the DbSet properties. It's worth noting that property names for collections generally adopt a plural form.

As an example, students instead of student, and developers have differing opinions regarding whether table names should be pluralized. In these cases, the default behavior

can be overridden by specifying singular table names in the DbContext. To achieve this, simply add the highlighted code after the last DbSet property in your code.

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {

        }
        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
        }
    }
}
```

1.4. Register the SchoolContext

ASP.NET Core uses dependency injection where services like the EF database context are registered during app startup. Components like MVC controllers receive these services through constructor parameters. The controller constructor code that obtains a context instance is demonstrated later in this Lab.

To register SchoolContext as a service, add the highlighted lines in Program.cs, inside the **ConfigureServices** method.

```
using ContosoUniversity.Data;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

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

builder.Services.AddDbContext<SchoolContext>(opt                    =>
opt.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnecti
on")));
var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    // The default HSTS value is 30 days. You may want to change this for
production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
```

```
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

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

app.Run();
```

The connection string's name is passed to the context via a method on a DbContextOptionsBuilder object. During local development, it's read from appsettings.json..

Open the **appsettings.json** file and add a connection string as shown in the following markup:

```
{
  /*"ConnectionStrings": {
    "DefaultConnection":
"Server=(localdb)\\mssqllocaldb;Database=ContosoUniversity1;Trusted_Connection=True;MultipleActive
ResultSets=true"
  }, */ // if using local Database
  "ConnectionStrings": {
    "DefaultConnection":    "Server=YourServer;Database=    ContosoUniversity1;    User=xxxxx;
Password=xxxxxxxx;Trusted_Connection=True;MultipleActiveResultSets=true"
  }, //If using your awn server
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

1.5. Add the database exception filter

Include AddDatabaseDeveloperPageExceptionFilter in ConfigureServices as shown below::

builder.Services.AddDatabaseDeveloperPageExceptionFilter();

```
...

var builder = WebApplication.CreateBuilder(args);

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

builder.Services.AddDbContext<SchoolContext>(opt                                    =>
opt.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

builder.Services.AddDatabaseDeveloperPageExceptionFilter();
var app = builder.Build();

. . .
```

The **AddDatabaseDeveloperPageExceptionFilter** offers useful error data during development..

2. Initialize DB with test data

EF creates a blank database, and subsequently, a method is incorporated within this section to populate it with test data once it is created.

The EnsureCreated method automates database creation. In "4.Migrations," you'll learn Code First Migrations for schema changes without dropping and recreating the database.

In the Data folder, create a new class named **DbInitializer** with the following code:

```
using ContosoUniversity.Models;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            context.Database.EnsureCreated();

            // Look for any students.
            if (context.Students.Any())
            {
                return;   // DB has been seeded
            }

            var students = new Student[]
            {
            new
Student{FirstMidName="Carson",LastName="Alexander",EnrollmentDate=DateTime.Parse("2022-09-01")},
            new
Student{FirstMidName="Meredith",LastName="Alonso",EnrollmentDate=DateTime.Parse("2022-09-01")},
            new
Student{FirstMidName="Arturo",LastName="Anand",EnrollmentDate=DateTime.Parse("2023-09-01")},
            new
Student{FirstMidName="Gytis",LastName="Barzdukas",EnrollmentDate=DateTime.Parse("2022-09-01")},
            new   Student{FirstMidName="Yan",LastName="Li",EnrollmentDate=DateTime.Parse("2022-
09-01")},
            new
Student{FirstMidName="Peggy",LastName="Justice",EnrollmentDate=DateTime.Parse("2021-09-01")},
            new
Student{FirstMidName="Laura",LastName="Norman",EnrollmentDate=DateTime.Parse("2023-09-01")},
            new
Student{FirstMidName="Nino",LastName="Olivetto",EnrollmentDate=DateTime.Parse("2022-09-01")}
            };
            foreach (Student s in students)
            {
                context.Students.Add(s);
            }
            context.SaveChanges();

            var courses = new Course[]
            {
            new Course{CourseID=1050,Title="Chemistry",Credits=3},
            new Course{CourseID=4022,Title="Microeconomics",Credits=3},
            new Course{CourseID=4041,Title="Macroeconomics",Credits=3},
            new Course{CourseID=1045,Title="Calculus",Credits=4},
            new Course{CourseID=3141,Title="Trigonometry",Credits=4},
            new Course{CourseID=2021,Title="Composition",Credits=3},
            new Course{CourseID=2042,Title="Literature",Credits=4}
            };
```

```
            foreach (Course c in courses)
            {
                context.Courses.Add(c);
            }
            context.SaveChanges();

            var enrollments = new Enrollment[]
            {
            new Enrollment{StudentID=1,CourseID=1050,Grade=Grade.A},
            new Enrollment{StudentID=1,CourseID=4022,Grade=Grade.C},
            new Enrollment{StudentID=1,CourseID=4041,Grade=Grade.B},
            new Enrollment{StudentID=2,CourseID=1045,Grade=Grade.B},
            new Enrollment{StudentID=2,CourseID=3141,Grade=Grade.F},
            new Enrollment{StudentID=2,CourseID=2021,Grade=Grade.F},
            new Enrollment{StudentID=3,CourseID=1050},
            new Enrollment{StudentID=4,CourseID=1050},
            new Enrollment{StudentID=4,CourseID=4022,Grade=Grade.F},
            new Enrollment{StudentID=5,CourseID=4041,Grade=Grade.C},
            new Enrollment{StudentID=6,CourseID=1045},
            new Enrollment{StudentID=7,CourseID=3141,Grade=Grade.A},
            };
            foreach (Enrollment e in enrollments)
            {
                context.Enrollments.Add(e);
            }
            context.SaveChanges();
        }
    }
}
```

The preceding code checks if the database exists:

- If the database is not found: It is created and loaded with test data. It loads test data into arrays rather than **List<T>** collections to optimize performance.
- If the database is found, it takes no action.

3. Update Program.cs with the following code:

```
using ContosoUniversity.Data;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

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

builder.Services.AddDbContext<SchoolContext>(opt                                    =>
opt.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

builder.Services.AddDatabaseDeveloperPageExceptionFilter();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}
using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;
    try
    {
        var context = services.GetRequiredService<SchoolContext>();
        DbInitializer.Initialize(context);
```

```
    }
    catch (Exception ex)
    {
        var logger = services.GetRequiredService<ILogger<Program>>();
        logger.LogError(ex, "An error occurred creating the DB.");
    }
}
app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

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

app.Run();
```

Program.cs does the following on app startup:

- Get a database context instance from the dependency injection container.
- Call the **DbInitializer**.Initialize method.

The first time the application is run, the database is created and the test data loaded. Whenever the data model changes:

- Delete the database.
- Update the seed method, and start afresh with a new database.

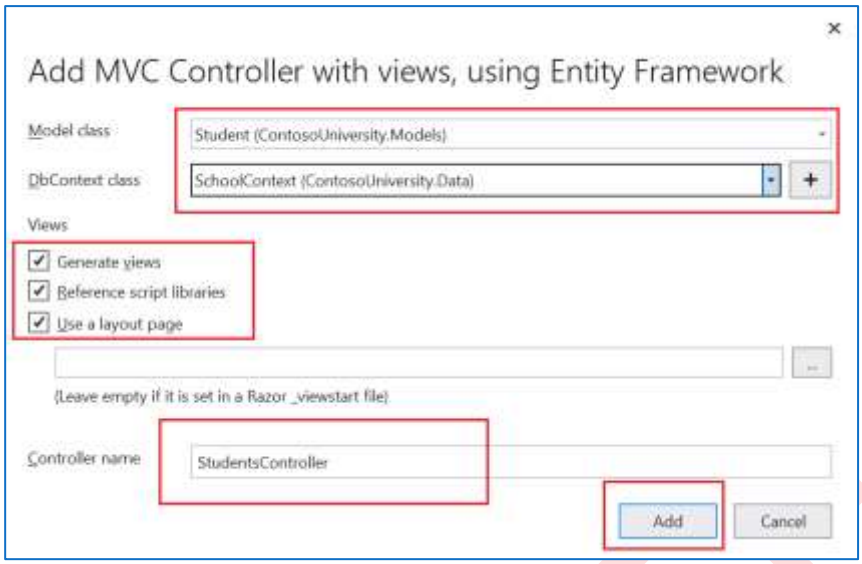
If the data structure changes later in this exercise, the database will be altered without being dropped and recreated. No data is lost when the data structure changes. (4th migration)

4. Create controller and views

Utilize the scaffolding mechanism in Visual Studio to incorporate an MVC controller and views that will employ EF to retrieve and store data.

The automatic creation of CRUD action methods and views is known as scaffolding.

- In Solution Explorer, right-click the Controllers folder and select Add > New Scaffolded Item.
- In the Add New Scaffolded Item dialog box:
  - ✓ Select **MVC controller with views, using Entity Framework**.
  - ✓ Click Add. The Add MVC Controller with views, using Entity Framework dialog box appears:



- ✓ In **Model class**, select **Student**.
- ✓ In Data context class, select **SchoolContext**.
- ✓ Accept the default **StudentsController** as the name.
- ✓ Click **Add**.

The Visual Studio scaffolding tool generates a StudentsController.cs file and a set of views (*.cshtml file) that work with the controller.

Notice the controller takes a SchoolContext as a constructor parameter.

```
namespace ContosoUniversity.Controllers
{
    public class StudentsController : Controller
    {
        private readonly SchoolContext _context;

        public StudentsController(SchoolContext context)
        {
            _context = context;
        }
}
```

ASP.NET Core dependency injection handles the task of providing an instance of SchoolContext to the controller. You set up this configuration in the Program.cs file.

The controller includes an Index action method, which shows all the students in the database. This method retrieves a list of students from the Students entity set by accessing the Students property of the database context instance.:

```
// GET: Students
  public async Task<IActionResult> Index()
  {
        return _context.Students != null ?
                    View(await _context.Students.ToListAsync()) :
                    Problem("Entity set 'SchoolContext.Students'  is null.");
  }
```

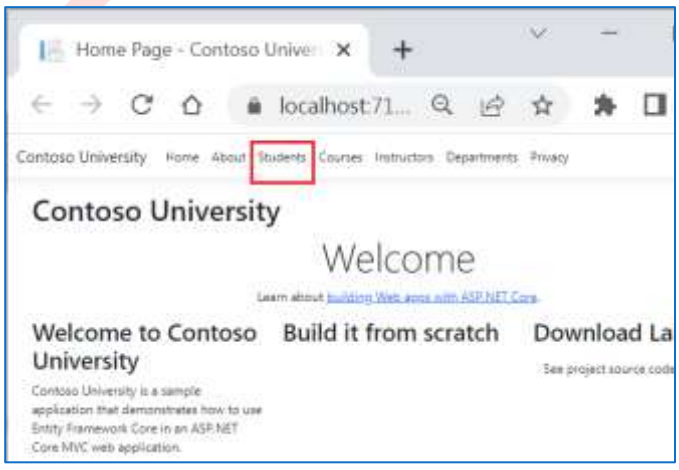
The asynchronous programming elements in this code are explained later.

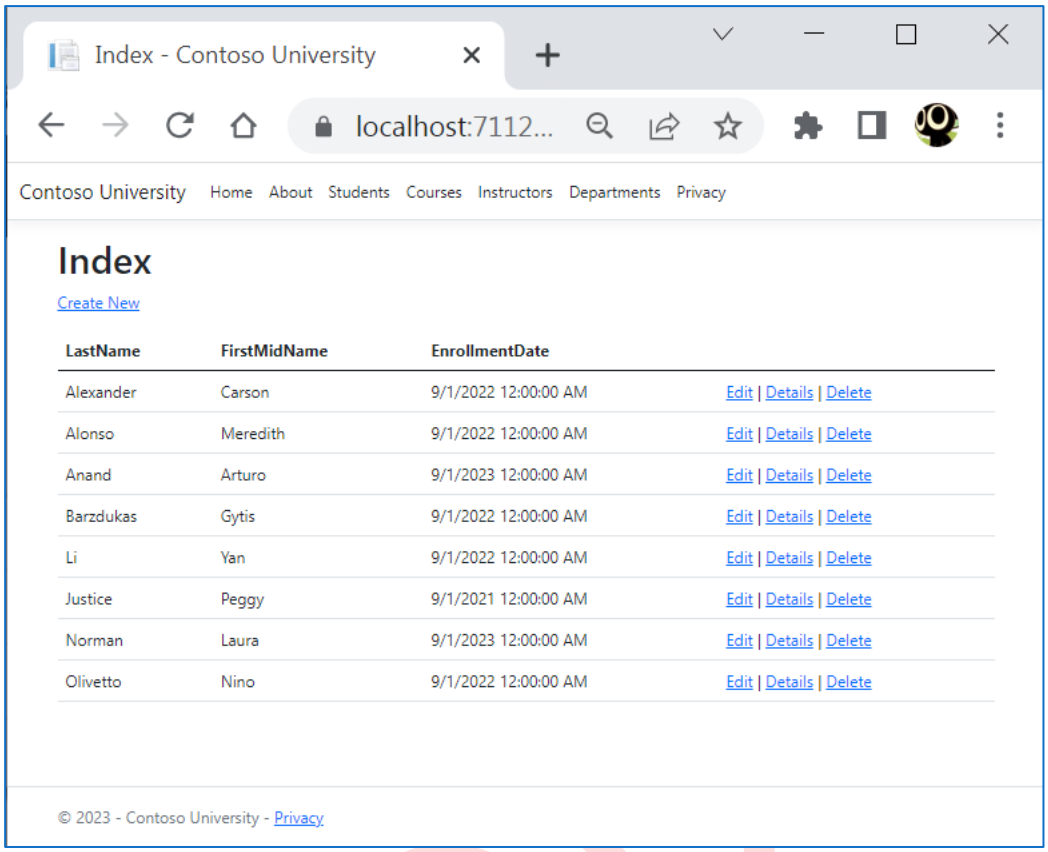
The **Views/Students/Index.cshtml** view displays this list in a table:

```
@model IEnumerable<ContosoUniversity.Models.Student>

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

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.LastName)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.FirstMidName)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.EnrollmentDate)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.LastName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.FirstMidName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
                <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

Press CTRL+F5 to run the project or choose Debug > Start Without Debugging from the menu.

Click the Students tab to see the test data that the **DbInitializer.Initialize** method inserted.



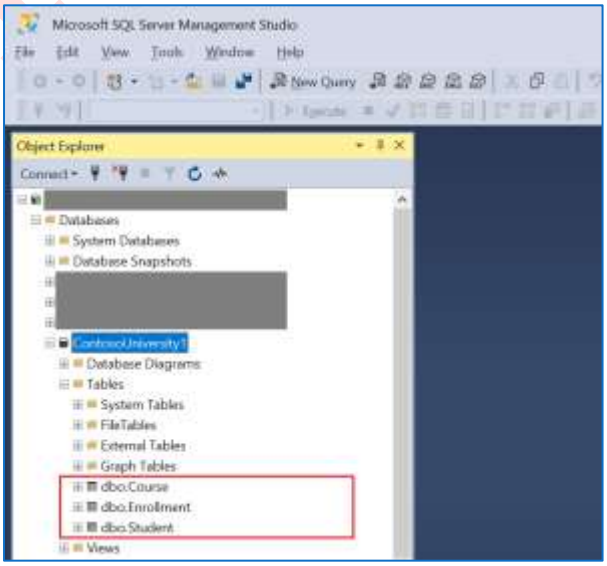


5. View the database

When the app is started, the **DbInitializer.Initialize** method calls EnsureCreated. EF saw that there was no database:

- So it created a database.
- The **Initialize** method code populated the database with data.

Open Microsoft SQL Server Management Studio. Login to your Database Server



Right-click the **Student** table and click **Select Top 1000 Rows** to see the data in the table.

```
/****** Script for SelectTopNRows command from SSMS  ******/
SELECT TOP (1000) [ID]
      ,[LastName]
      ,[FirstMidName]
      ,[EnrollmentDate]
  FROM [ContosoUniversity1].[dbo].[Student]
```

100 %

Results | Messages

	ID	LastName	FirstMidName	EnrollmentDate
1	1	Alexander	Carson	2022-09-01 00:00:00.0000000
2	2	Alonso	Meredith	2022-09-01 00:00:00.0000000
3	3	Anand	Arturo	2023-09-01 00:00:00.0000000
4	4	Barzdukas	Gytis	2022-09-01 00:00:00.0000000
5	5	Li	Yan	2022-09-01 00:00:00.0000000
6	6	Justice	Peggy	2021-09-01 00:00:00.0000000
7	7	Norman	Laura	2023-09-01 00:00:00.0000000
8	8	Olivetto	Nino	2022-09-01 00:00:00.0000000

Because EnsureCreated is called in the initializer method that runs on app start, you could:

- Make a change to the Student class.
- Delete the database.
- Stop, then start the app. The database is automatically re-created to match the change.

For instance, if a Student class is enhanced with an EmailAddress attribute, a fresh EmailAddress column will be created in the table that is being recreated. However, the updated view will not exhibit the added EmailAddress attribute. Therefore, it becomes necessary to utilize Migration during the application development process.

Conventions

The quantity of code written for the EF to generate a comprehensive database is minimal due to the utilization of the conventions EF users:

- The names of **DbSet** properties are used as table names. For entities not referenced by a **DbSet** property, entity class names are used as table names.
- Entity property names are used for column names.

- Entity properties that are named **ID** or **classnameID** are recognized as PK properties.
- A property is interpreted as a FK property if it's named <navigation property name> <PK property name>.
  - ❖ For example, **StudentID** for the **Student** navigation property since the **Student** entity's PK is **ID**. FK properties can also be named <primary key property name>.
  - ❖ For example, **EnrollmentID** since the **Enrollment** entity's PK is **EnrollmentID**.

Standard behavior can be overwritten. For instance, the names of tables can be explicitly indicated, as demonstrated previously in this Lab. Names of columns and any attribute can be designated as a primary key or foreign key.

6. Asynchronous code

Asynchronous programming is the dereliction mode forASP.NET Core and EF Core. A web garçon has a limited number of vestments available, and in high cargo situations all of the available vestments might be in use. When that occurs, the garçon can not handle new requests until the vestments are released. With coetaneous law, multiple vestments may be enthralled indeed though they aren't laboriously performing any tasks because they're staying for I/ O operations to finish. With asynchronous law, when a process is awaiting the completion of an I/ O operation, its thread is freed up for the garçon to use for handling other requests. Accordingly, asynchronous law enables the garçon's coffers to be employed more effectively, allowing it to handle increased business without detainments.

While asynchronous code does add a slight overhead during runtime, its impact on performance is insignificant for low-traffic scenarios. However, in high-traffic situations, the potential for significant performance improvement becomes apparent..

In the following code, **async**, **Task<T>**, **await**, and **ToListAsync** make the code execute asynchronously.

```
// GET: Students
 public async Task<IActionResult> Index()
 {
       return _context.Students != null ?
                   View(await _context.Students.ToListAsync()) :
                   Problem("Entity set 'SchoolContext.Students'  is null.");
 }
```

- By utilizing the async keyword, the compiler generates callbacks for specific segments within the method body, while also automatically producing the Task<IActionResult> object that is ultimately returned..
- The return type **Task<IActionResult>** represents ongoing work with a result of type **IActionResult**.
- When encountering the "await" keyword, the compiler divides the method into two sections. The initial part concludes with the asynchronously initiated operation, while the subsequent part is encapsulated within a callback method that executes upon the completion of the operation.
- **ToListAsync** is the asynchronous version of the ToList extension method.

Some things to be aware of when writing asynchronous code that uses EF:

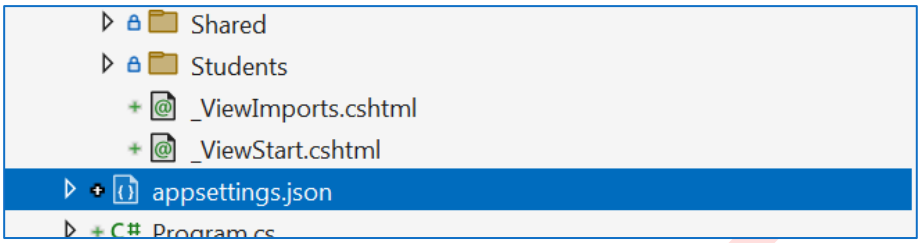
- Only statements causing database queries or commands are executed asynchronously, like ToListAsync, SingleOrDefaultAsync, SaveChangesAsync. Statements that only change an IQueryable, e.g., var students = context.Students.Where(s => s.LastName == "Davolio"), are not executed asynchronously.
- EF context isn't thread-safe, so avoid parallel operations. Always use the await keyword with async EF methods.
- To benefit from async code performance, ensure libraries use async when calling EF methods that trigger database queries.

7. SQL Logging of Entity Framework Core

Logging configuration is commonly provided by the **Logging** section of **appsettings.{Environment}.json** files.

To log SQL statements, add ***"Microsoft.EntityFrameworkCore.Database.Command"***: "**Information**" to the **appsettings.Development.json** file:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "Microsoft.EntityFrameworkCore.Database.Command": "Information"
    }
  }
}
```



8. Create, Read, Update, and Delete

Learn the basic CRUD operations by reviewing and customizing the code automatically created by the MVC scaffolding for controllers and views.

- Customize the Details page
- Update the Create page
- Update the Edit page
- Update the Delete page
- Close database connections

8.1. Customize the Details page:

The scaffolded code for the Students Index page left out the **Enrollments** property, because that property holds a collection. In the Details page, you'll display the contents of the collection in an HTML table.

In **Controllers/StudentsController.cs**, the action method for the Details view uses the **FirstOrDefaultAsync** method to retrieve a single **Student** entity. Add code that calls **Include**. **ThenInclude**, and **AsNoTracking** methods, as shown in the following highlighted code.

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

        //var student = await _context.Students
        //    .FirstOrDefaultAsync(m => m.ID == id);
        var student = await _context.Students
```

```
            .Include(s => s.Enrollments)
            .ThenInclude(e => e.Course)
            .AsNoTracking()
            .FirstOrDefaultAsync(m => m.ID == id);
        if (student == null)
        {
            return NotFound();
        }

        return View(student);
    }
```

The **Include** and **ThenInclude** methods cause the context to load the **Student**.**Enrollments** navigation property, and within each enrollment the **Enrollment**.**Course** navigation property. You'll learn more about these methods in the "6. Read related data".

The **AsNoTracking** method improves performance in scenarios where the entities returned won't be updated in the current context's lifetime. You'll learn more about **AsNoTracking** at the end of this session.

8.2. Route data:

The key value that's passed to the **Details** method comes from route data. Route data is data that the model binder found in a segment of the URL. For example, the default route specifies controller, action, and id segments:

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

In the following URL, the default route maps Instructor as the controller, Index as the action, and 1 as the id; these are route data values.

```
http://localhost:1230/Instructor/Index/1?courseID=2021
```

The last part of the URL ("?courseID=2021") is a query string value. The model binder will also pass the ID value to the Index method id parameter if you pass it as a query string value:

```
http://localhost:1230/Instructor/Index?id=1&CourseID=2021
```

In the Index page, hyperlink URLs are created by tag helper statements in the Razor view. In the following Razor code, the id parameter matches the default route, so id is added to the route data.

```
<a asp-action="Edit" asp-route-id="@item.ID">Edit</a>
```

This generates the following HTML when item.ID is 6:

```
<a href="/Students/Edit/6">Edit</a>
```

In the following Razor code, **studentID** doesn't match a parameter in the default route, so it's added as a query string.

```
<a href="/Students/Edit?studentID=6">Edit</a>
```

8.3. Add enrollments to the Details view:

Open Views/Students/Details.cshtml. Each field is displayed using DisplayNameFor and DisplayFor helpers, as shown in the following example:

```
<dt class = "col-sm-2">
    @Html.DisplayNameFor(model => model.LastName)
</dt>
<dd class = "col-sm-10">
    @Html.DisplayFor(model => model.LastName)
</dd>
```

After the last field and immediately before the closing **</dl>** tag, add the following code to display a list of enrollments:

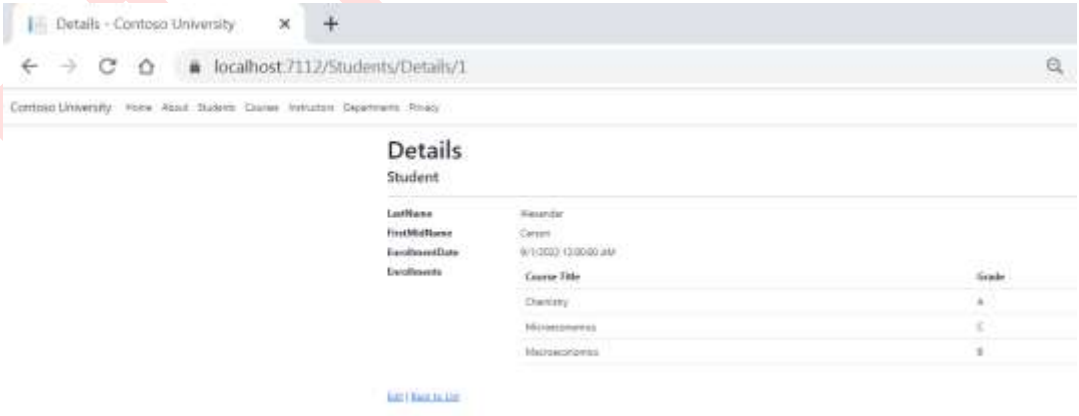
```
@model ContosoUniversity.Models.Student

@{
    ViewData["Title"] = "Details";
}

<h1>Details</h1>

<div>
    <h4>Student</h4>
    <hr />
    <dl class="row">
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.LastName)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.LastName)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.FirstMidName)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.FirstMidName)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.EnrollmentDate)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.EnrollmentDate)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Enrollments)
```

```
        </dt>
        <dd class="col-sm-10">
            <table class="table">
                <tr>
                    <th>Course Title</th>
                    <th>Grade</th>
                </tr>
                @foreach (var item in Model.Enrollments)
                {
                    <tr>
                        <td>
                            @Html.DisplayFor(modelItem => item.Course.Title)
                        </td>
                        <td>
                            @Html.DisplayFor(modelItem => item.Grade)
                        </td>
                    </tr>
                }
            </table>
        </dd>
    </dl>
</div>
<div>
    <a asp-action="Edit" asp-route-id="@Model?.ID">Edit</a> |
    <a asp-action="Index">Back to List</a>
</div>
```

If code indentation is wrong after you paste the code, press CTRL-K-D to correct it.

This code loops through the entities in the **Enrollments** navigation property. For each enrollment, it displays the course title and the grade. The course title is retrieved from the Course entity that's stored in the **Course** navigation property of the Enrollments entity.

Run the app, select the **Students** tab, and click the **Details** link for a student. You see the list of courses and grades for the selected student:



8.4. Update the Create page

In **StudentsController.cs**, modify the HttpPost **Create** method by adding a try-catch block and removing ID from the **Bind** attribute.

```
        [HttpPost]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult> Create([Bind("LastName,FirstMidName,EnrollmentDate")]
Student student)
        {
            try
            {
                if (ModelState.IsValid)
                {
                    _context.Add(student);
                    await _context.SaveChangesAsync();
                    return RedirectToAction(nameof(Index));
                }
            }
            catch (DbUpdateException /* ex */)
            {
                //Log the error (uncomment ex variable name and write a log.
                ModelState.AddModelError("", "Unable to save changes. " +
                    "Try again, and if the problem persists " +
                    "see your system administrator.");
            }
            return View(student);
        }
```

This code adds the Student entity created by the ASP.NET Core MVC model binder to the Students entity set and then saves the changes to the database. (Model binder refers to the ASP.NET Core MVC functionality that makes it easier for you to work with data submitted by a form; a model binder converts posted form values to CLR types and passes them to the action method in parameters. In this case, the model binder instantiates a Student entity for you using property values from the Form collection.)

You removed **ID** from the **Bind** attribute because ID is the primary key value which SQL Server will set automatically when the row is inserted. Input from the user doesn't set the ID value.

Other than the **Bind** attribute, the try-catch block is the only change you've made to the scaffolded code. If an exception that derives from **DbUpdateException** is caught while the changes are being saved, a generic error message is displayed. **DbUpdateException** exceptions are sometimes caused by something external to the application rather than a programming error, so the user is advised to try again. Although not implemented in this sample, a production quality application would log the exception.

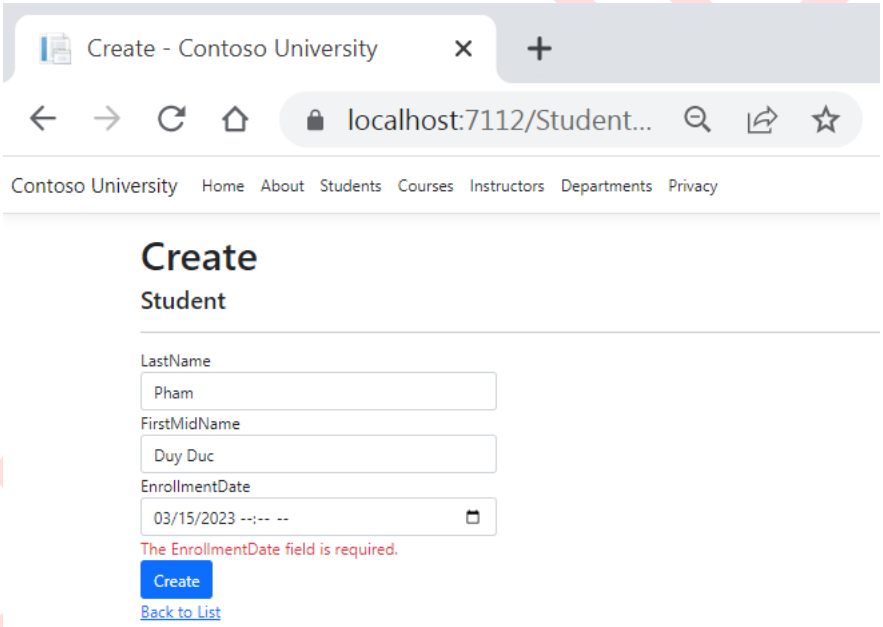
The **ValidateAntiForgeryToken** attribute helps prevent cross-site request forgery (CSRF) attacks. The token is automatically injected into the view by the FormTagHelper and is included when the form is submitted by the user. The token is validated by the **ValidateAntiForgeryToken** attribute.

The code in **Views/Students/Create.cshtml** uses label, input, and span (for validation messages) tag helpers for each field. And Update following code:

```
. . .
<h1>Create</h1>

<h4>Student</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Create" method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">

...
```

Run the app, select the **Students** tab, and click **Create New**.

Enter names and a date. Try entering an invalid date if your browser lets you do that. (Some browsers force you to use a date picker.) Then click Create to see the error message.



This is server-side validation that you get by default. The following highlighted code shows the model validation check in the Create method.

```
if (ModelState.IsValid)
    {
        _context.Add(student);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
```

Change the date to a valid value and click Create to see the new student appear in the Index page.

8.5. Update the Edit page

In **StudentController.cs**, the HttpGet **Edit** method (the one without the **HttpPost** attribute) uses the **FirstOrDefaultAsync** method to retrieve the selected Student entity, as you saw in the **Details** method. You don't need to change this method.

Replace the **HttpPost Edit** action method with the following code.

```
        [HttpPost, ActionName("Edit")]
            [ValidateAntiForgeryToken]
            public async Task<IActionResult> EditPost(int? id)
            {
                if (id == null)
                {
                    return NotFound();
                }
                var studentToUpdate = await _context.Students.FirstOrDefaultAsync(s
=> s.ID == id);
                if (await TryUpdateModelAsync<Student>(studentToUpdate,"",
                    s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
                {
                    try
                    {
                        await _context.SaveChangesAsync();
                        return RedirectToAction(nameof(Index));
                    }
                    catch (DbUpdateException /* ex */)
                    {
                        //Log the error (uncomment ex variable name and write a log.)
                        ModelState.AddModelError("", "Unable to save changes. " +
                            "Try again, and if the problem persists, " +
                            "see your system administrator.");
                    }
                }
                return View(studentToUpdate);
            }
```

These changes implement a security best practice to prevent overposting. The scaffolder generated a **Bind** attribute and added the entity created by the model binder to the entity set with a **Modified** flag. That code isn't recommended for many scenarios because the **Bind** attribute clears out any pre-existing data in fields not listed in the **Include** parameter.

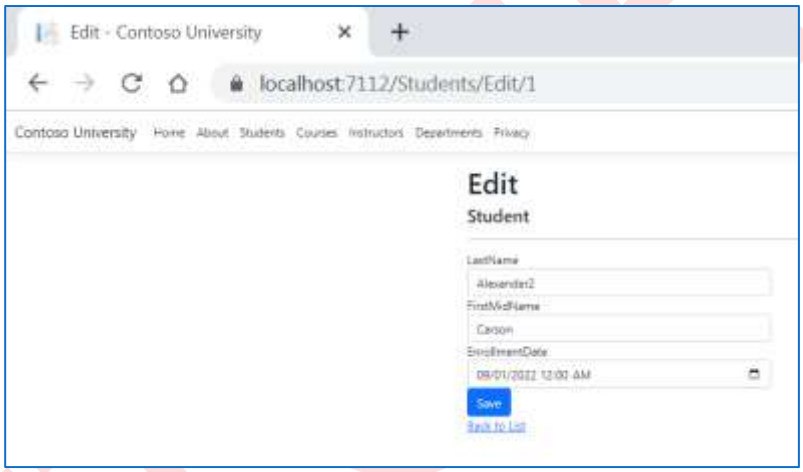
The new code reads the existing entity and calls TryUpdateModel to update fields in the retrieved entity based on user input in the posted form data. The Entity Framework's automatic change tracking sets the Modified flag on the fields that are changed by form input. When the SaveChanges method is called, the Entity Framework creates SQL statements to update the database row. Concurrency conflicts are ignored, and only the table columns that were updated by the user are updated in the database.

As a best practice to prevent overposting, the fields that you want to be updateable by the Edit page are declared in the TryUpdateModel parameters. (The empty string preceding the list of fields in the parameter list is for a prefix to use with the form fields names.) Currently there are no extra fields that you're protecting, but listing the fields that you want the model binder to bind ensures that if you add fields to the data model in the future, they're automatically protected until you explicitly add them here.

As a result of these changes, the method signature of the **HttpPost Edit** method is the same as the HttpGet Edit method; therefore you've renamed the method **EditPost**.

Run the app, select the Students tab, then click an Edit hyperlink.

Change some of the data and click Save. The Index page opens and you see the changed data.



8.6. Update the Delete page

In StudentController.cs, the template code for the **HttpGet Delete** method uses the **FirstOrDefaultAsync** method to retrieve the selected Student entity, as you saw in the Details and Edit methods. However, to implement a custom error message when the call to **SaveChanges** fails, you'll add some functionality to this method and its corresponding view.

As you saw for update and create operations, delete operations require two action methods. The method that's called in response to a GET request displays a view that gives the user a chance to approve or cancel the delete operation. If the user approves it, a POST

request is created. When that happens, the **HttpPost Delete** method is called and then that method actually performs the delete operation.

You'll add a try-catch block to the **HttpPost Delete** method to handle any errors that might occur when the database is updated. If an error occurs, the HttpPost Delete method calls the HttpGet Delete method, passing it a parameter that indicates that an error has occurred. The HttpGet Delete method then redisplays the confirmation page along with the error message, giving the user an opportunity to cancel or try again.

Replace the **HttpGet Delete** action method with the following code, which manages error reporting.

```
// GET: Students/Delete/5
public async Task<IActionResult> Delete(int? id, bool? saveChangesError = false)
{
    if (id == null || _context.Students == null)
    {
        return NotFound();
    }

    var student = await _context.Students
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);
    if (student == null)
    {
        return NotFound();
    }
    if (saveChangesError.GetValueOrDefault())
    {
        ViewData["ErrorMessage"] =
            "Delete failed. Try again, and if the problem persists " +
            "see your system administrator.";
    }
    return View(student);
}
```

This code accepts an optional parameter that indicates whether the method was called after a failure to save changes. This parameter is false when the HttpGet Delete method is called without a previous failure. When it's called by the HttpPost Delete method in response to a database update error, the parameter is true and an error message is passed to the view.

Replace the HttpPost Delete action method (named DeleteConfirmed) with the following code, which performs the actual delete operation and catches any database update errors.

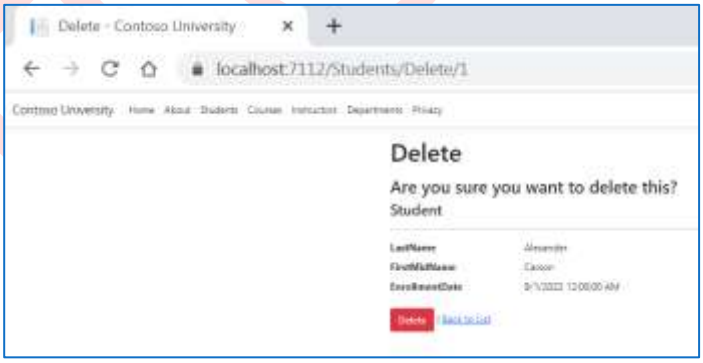
```
// POST: Students/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    if (_context.Students == null)
    {
        return Problem("Entity set 'SchoolContext.Students'  is null.");
    }
    var student = await _context.Students.FindAsync(id);
    try
    {
        if (student != null)
        {
            _context.Students.Remove(student);
        }

        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction(nameof(Delete), new { id = id, saveChangesError = true });
    }
}
```

This code retrieves the selected entity, then calls the Remove method to set the entity's status to Deleted. When SaveChanges is called, a SQL DELETE command is generated.

In Views/Student/Delete.cshtml, add an error message between the h2 heading and the h3 heading, as shown in the following example:

```
<h1>Delete</h1>
<p class="text-danger">@ViewData["ErrorMessage"]</p>
<h3>Are you sure you want to delete this?</h3>
```

Run the app, select the Students tab, and click a Delete hyperlink. Click Delete. The Index page is displayed without the deleted student.



9. Sort, Filter, page, and group

9.1. Add sorting Functionality to the Index method

In StudentsController.cs, replace the Index method with the following code:

```
public async Task<IActionResult> Index(string sortOrder)
    {
        ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
        ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
        var students = from s in _context.Students
                       select s;
        switch (sortOrder)
        {
            case "name_desc":
                students = students.OrderByDescending(s => s.LastName);
                break;
            case "Date":
                students = students.OrderBy(s => s.EnrollmentDate);
                break;
            case "date_desc":
                students = students.OrderByDescending(s => s.EnrollmentDate);
                break;
            default:
                students = students.OrderBy(s => s.LastName);
                break;
        }
        return View(await students.AsNoTracking().ToListAsync());
    }
```

This code receives a sortOrder parameter from the query string in the URL. The query string value is provided by ASP.NET Core MVC as a parameter to the action method. The parameter will be a string that's either "Name" or "Date", optionally followed by an underscore and the string "desc" to specify descending order. The default sort order is ascending.

The first time the Index page is requested, there's no query string. The students are displayed in ascending order by last name, which is the default as established by the fall-through case in the **switch** statement. When the user clicks a column heading hyperlink, the appropriate **sortOrder** value is provided in the query string.

The two **ViewData** elements (NameSortParm and DateSortParm) are used by the view to configure the column heading hyperlinks with the appropriate query string values.

```
ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
```

These are ternary statements. The first one specifies that if the **sortOrder** parameter is null or empty, NameSortParm should be set to "name_desc"; otherwise, it should be set to an empty string. These two statements enable the view to set the column heading hyperlinks as follows:

Current sort order	Last Name Hyperlink	Date Hyperlink
Last Name ascending	descending	ascending
Last Name descending	ascending	ascending
Date ascending	ascending	descending
Date descending	ascending	ascending

The method uses LINQ to Entities to specify the column to sort by. The code creates an **IQueryable** variable before the switch statement, modifies it in the switch statement, and calls the **ToListAsync** method after the **switch** statement. When you create and modify **IQueryable** variables, no query is sent to the database. The query isn't executed until you convert the **IQueryable** object into a collection by calling a method such as **ToListAsync**. Therefore, this code results in a single query that's not executed until the **return View** statement.

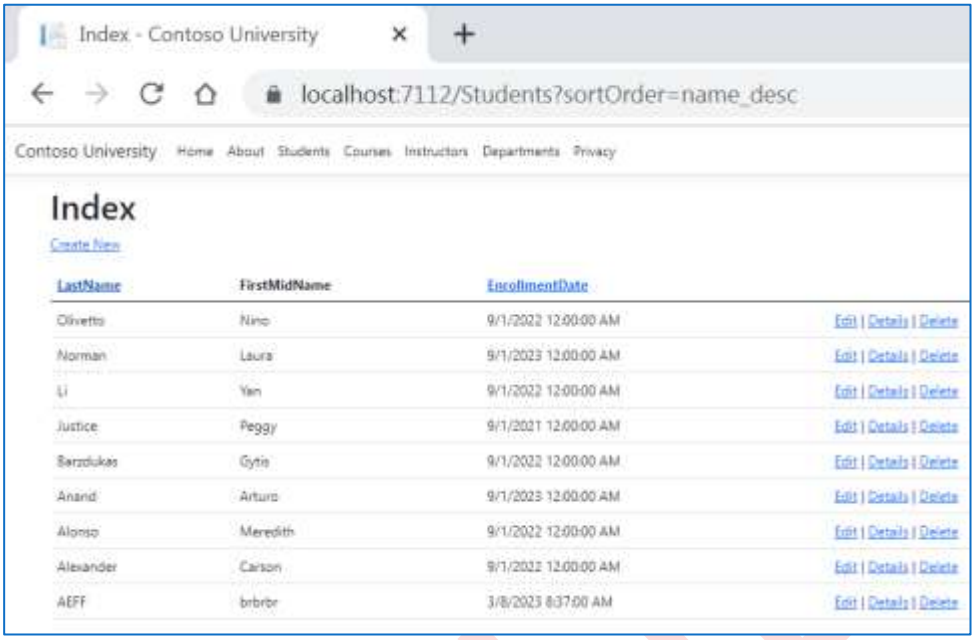
9.2. Add column heading hyperlinks to the Student Index view

Replace the code in Views/Students/Index.cshtml, with the following code to add column heading hyperlinks. The changed lines are highlighted.

```
@model IEnumerable<ContosoUniversity.Models.Student>

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

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-action="Index" asp-route-sortOrder="@ViewData["NameSortParm"]">@Html.DisplayNameFor(model => model.LastName)</a>
            </th>
            <th>
                @Html.DisplayNameFor(model => model.FirstMidName)
            </th>
            <th>
                <a asp-action="Index" asp-route-sortOrder="@ViewData["DateSortParm"]">@Html.DisplayNameFor(model => model.EnrollmentDate)</a>
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.LastName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.FirstMidName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
                <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

This code uses the information in **ViewData** properties to set up hyperlinks with the appropriate query string values.

Run the app, select the **Students** tab, and click the **Last Name** and **Enrollment Date** column headings to verify that sorting works.



Add a Search box - Add filtering functionality to the Index method

In StudentsController.cs, replace the Index method with the following code (the changes are highlighted).

```
public async Task<IActionResult> Index(string sortOrder, string searchString)
        {
            ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
            ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
            ViewData["CurrentFilter"] = searchString;
            var students = from s in _context.Students
                           select s;
            if (!String.IsNullOrEmpty(searchString))
            {
                students = students.Where(s => s.LastName.Contains(searchString)
                                       || s.FirstMidName.Contains(searchString));
            }
            switch (sortOrder)
            {
                case "name_desc":
                    students = students.OrderByDescending(s => s.LastName);
                    break;
                case "Date":
                    students = students.OrderBy(s => s.EnrollmentDate);
                    break;
                case "date_desc":
                    students = students.OrderByDescending(s => s.EnrollmentDate);
                    break;
                default:
                    students = students.OrderBy(s => s.LastName);
                    break;
            }
            return View(await students.AsNoTracking().ToListAsync());
        }
```

You've added a searchString parameter to the Index method. The search string value is received from a text box that you'll add to the Index view. You've also added to the LINQ statement a where clause that selects only students whose first name or last name

contains the search string. The statement that adds the where clause is executed only if there's a value to search for.

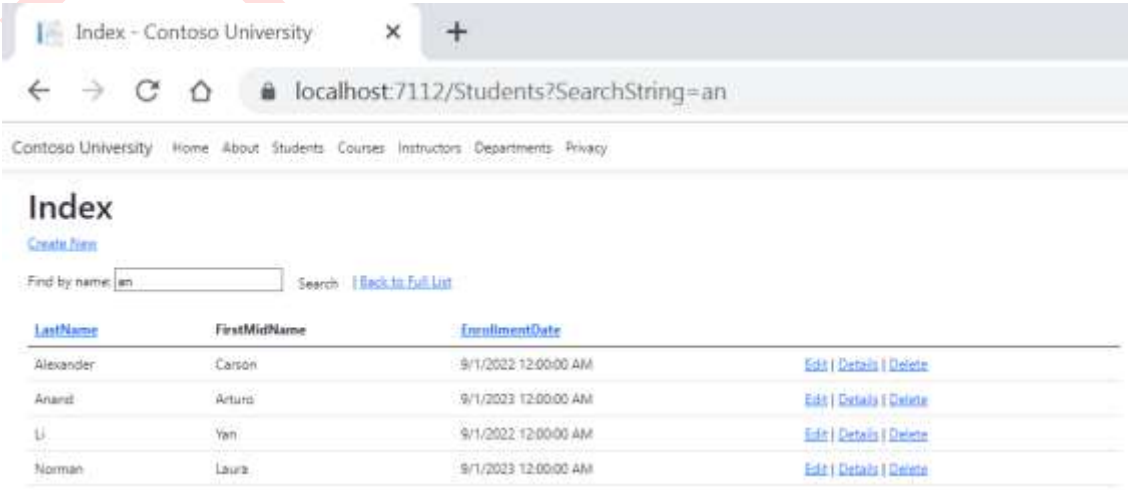
9.3. Add a Search Box to the Student Index View

In **Views/Student/Index.cshtml**, add the highlighted code immediately before the opening table tag in order to create a caption, a text box, and a **Search** button.

```
@model IEnumerable<ContosoUniversity.Models.Student>

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

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
<form asp-action="Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name: <input type="text" name="SearchString" value="@ViewData["CurrentFilter"]" />
            <input type="submit" value="Search" class="btn btn-default" /> |
            <a asp-action="Index">Back to Full List</a>
        </p>
    </div>
</form>
<table class="table">
```

This code uses the **<form>** tag helper to add the search text box and button. By default, the **<form>** tag helper submits form data with a POST, which means that parameters are passed in the HTTP message body and not in the URL as query strings. When you specify HTTP GET, the form data is passed in the URL as query strings, which enables users to bookmark the URL. The W3C guidelines recommend that you should use GET when the action doesn't result in an update.

Run the app, select the Students tab, enter a search string, and click Search to verify that filtering is working.



Notice that the URL contains the search string.

http://localhost:7112/Students?SearchString=an

If you bookmark this page, you'll get the filtered list when you use the bookmark. Adding **method="get"** to the **form** tag is what caused the query string to be generated.

At this stage, if you click a column heading sort link you'll lose the filter value that you entered in the **Search** box. You'll fix that in the next section.

Add paging to Students Index

To add paging to the Students Index page, you'll create a **PaginatedList** class that uses **Skip** and **Take** statements to filter data on the server instead of always retrieving all rows of the table. Then you'll make additional changes in the Index method and add paging buttons to the **Index** view.

In the project folder, create **PaginatedList.cs**, and then replace the template code with the following code.

```
using Microsoft.EntityFrameworkCore;

    namespace ContosoUniversity
    {
        public class PaginatedList<T> : List<T>
        {
            public int PageIndex { get; private set; }
            public int TotalPages { get; private set; }

            public PaginatedList(List<T> items, int count, int pageIndex, int pageSize)
            {
                PageIndex = pageIndex;
                TotalPages = (int)Math.Ceiling(count / (double)pageSize);

                this.AddRange(items);
            }

            public bool HasPreviousPage => PageIndex > 1;

            public bool HasNextPage => PageIndex < TotalPages;

            public static async Task<PaginatedList<T>> CreateAsync(IQueryable<T> source, int pageIndex, int pageSize)
            {
                var count = await source.CountAsync();
                var items = await source.Skip((pageIndex - 1) * pageSize).Take(pageSize).ToListAsync();
                return new PaginatedList<T>(items, count, pageIndex, pageSize);
            }
        }
    }
```

The **CreateAsync** method in this code takes page size and page number and applies the appropriate **Skip** and **Take** statements to the **IQueryable**. When **ToListAsync** is called on the **IQueryable**, it will return a List containing only the requested page. The properties **HasPreviousPage** and **HasNextPage** can be used to enable or disable **Previous** and **Next** paging buttons.

A **CreateAsync** method is used instead of a constructor to create the **PaginatedList<T>** object because constructors can't run asynchronous code.

9.4. Add paging to Index method

In **StudentsController.cs**, replace the **Index** method with the following code.

```
public async Task<IActionResult> Index(
                        string sortOrder,
                        string currentFilter,
                        string searchString,
                        int? pageNumber)
        {
            ViewData["CurrentSort"] = sortOrder;
            ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
            ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
            if (searchString != null)
            {
                pageNumber = 1;
            }
            else
            {
                searchString = currentFilter;
            }
            ViewData["CurrentFilter"] = searchString;
            var students = from s in _context.Students
                           select s;
            if (!String.IsNullOrEmpty(searchString))
            {
                students = students.Where(s => s.LastName.Contains(searchString)
                                       || s.FirstMidName.Contains(searchString));
            }
            switch (sortOrder)
            {
                case "name_desc":
                    students = students.OrderByDescending(s => s.LastName);
                    break;
                case "Date":
                    students = students.OrderBy(s => s.EnrollmentDate);
                    break;
                case "date_desc":
                    students = students.OrderByDescending(s => s.EnrollmentDate);
                    break;
                default:
                    students = students.OrderBy(s => s.LastName);
                    break;
            }
            int pageSize = 3;
            return View(await PaginatedList<Student>.CreateAsync(students.AsNoTracking(), pageNumber ??
1, pageSize));
            //return View(await students.AsNoTracking().ToListAsync());
        }
```

This code adds a page number parameter, a current sort order parameter, and a current filter parameter to the method signature.

```
public async Task<IActionResult> Index(string sortOrder, string currentFilter,
                                       string searchString, int? pageNumber)
```

The first time the page is displayed, or if the user hasn't clicked a paging or sorting link, all the parameters will be null. If a paging link is clicked, the page variable will contain the page number to display.

The **ViewData** element named CurrentSort provides the view with the current sort order, because this must be included in the paging links in order to keep the sort order the same while paging.

The **ViewData** element named CurrentFilter provides the view with the current filter string. This value must be included in the paging links in order to maintain the filter settings during paging, and it must be restored to the text box when the page is redisplayed.

If the search string is changed during paging, the page has to be reset to 1, because the new filter can result in different data to display. The search string is changed when a value is entered in the text box and the Submit button is pressed. In that case, the **searchString** parameter isn't null.

```
if (searchString != null)
{
    pageNumber = 1;
}
else
{
    searchString = currentFilter;
}
```

At the end of the **Index** method, the **PaginatedList.CreateAsync** method converts the student query to a single page of students in a collection type that supports paging. That single page of students is then passed to the view.

```
return View(await PaginatedList<Student>.CreateAsync(students.AsNoTracking(), pageNumber ?? 1, pageSize));
```

The **PaginatedList.CreateAsync** method takes a page number. The two question marks represent the null-coalescing operator. The null-coalescing operator defines a default value for a nullable type; the expression **(pageNumber ?? 1)** means return the value of **pageNumber** if it has a value, or return 1 if **pageNumber** is null.

9.5. Add paging links

In **Views/Students/Index.cshtml**, replace the existing code with the following code. The changes are highlighted.

```
@*@model IEnumerable<ContosoUniversity.Models.Student>*@
@model PaginatedList<ContosoUniversity.Models.Student>
@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
<form asp-action="Index" method="get">
    <div class="form-actions no-color">
        <p>
```

```
                Find by name: <input type="text" name="SearchString"
value="@ViewData["CurrentFilter"]" />
                <input type="submit" value="Search" class="btn btn-default" /> |
                <a asp-action="Index">Back to Full List</a>
            </p>
        </div>
    </form>
    <table class="table">
        <thead>
            <tr>
                @*<th>
                    <a asp-action="Index" asp-route-
sortOrder="@ViewData["NameSortParm"]">@Html.DisplayNameFor(model => model.LastName)</a>
                </th>
                <th>
                    @Html.DisplayNameFor(model => model.FirstMidName)
                </th>
                <th>
                    <a asp-action="Index" asp-route-
sortOrder="@ViewData["DateSortParm"]">@Html.DisplayNameFor(model => model.EnrollmentDate)</a>
                </th>*@
                <th>
                    <a  asp-action="Index"  asp-route-sortOrder="@ViewData["NameSortParm"]"  asp-
route-currentFilter="@ViewData["CurrentFilter"]">Last Name</a>
                </th>
                <th>
                    First Name
                </th>
                <th>
                    <a  asp-action="Index"  asp-route-sortOrder="@ViewData["DateSortParm"]"  asp-
route-currentFilter="@ViewData["CurrentFilter"]">Enrollment Date</a>
                </th>
                <th></th>
            </tr>
        </thead>
        <tbody>
    @foreach (var item in Model) {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
    }
        </tbody>
    </table>
    @{
        var prevDisabled = !Model.HasPreviousPage ? "disabled" : "";
        var nextDisabled = !Model.HasNextPage ? "disabled" : "";
    }

    <a asp-action="Index"
       asp-route-sortOrder="@ViewData["CurrentSort"]"
       asp-route-pageNumber="@(Model.PageIndex - 1)"
       asp-route-currentFilter="@ViewData["CurrentFilter"]"
       class="btn btn-default @prevDisabled">
        Previous
    </a>
    <a asp-action="Index"
       asp-route-sortOrder="@ViewData["CurrentSort"]"
       asp-route-pageNumber="@(Model.PageIndex + 1)"
       asp-route-currentFilter="@ViewData["CurrentFilter"]"
       class="btn btn-default @nextDisabled">
        Next
    </a>
```

The **@model** statement at the top of the page specifies that the view now gets a **PaginatedList<T>** object instead of a **List<T>** object.

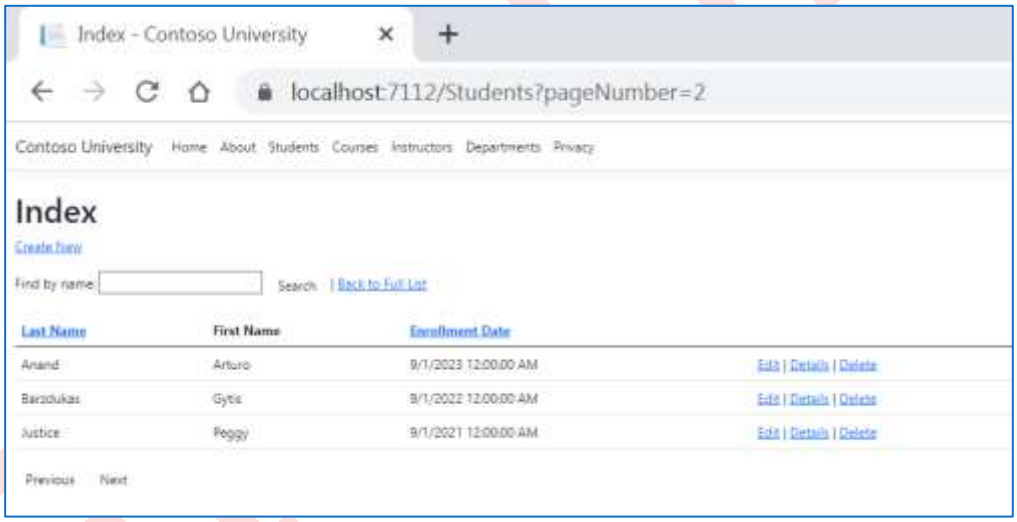
The column header links use the query string to pass the current search string to the controller so that the user can sort within filter results:

```
<a asp-action="Index" asp-route-sortOrder="@ViewData["DateSortParm"]" asp-route-currentFilter="@ViewData["CurrentFilter"]">Enrollment Date</a>
```

The paging buttons are displayed by tag helpers:

```
<a asp-action="Index"
   asp-route-sortOrder="@ViewData["CurrentSort"]"
   asp-route-pageNumber="@(Model.PageIndex - 1)"
   asp-route-currentFilter="@ViewData["CurrentFilter"]"
   class="btn btn-default @prevDisabled">
    Previous
</a>
```

Run the app and go to the Students page.



Click the paging links in different sort orders to make sure paging works. Then enter a search string and try paging again to verify that paging also works correctly with sorting and filtering.

9.6. Create an About page

For the Contoso University website's About page, you'll display how many students have enrolled for each enrollment date. This requires grouping and simple calculations on the groups. To accomplish this, you'll do the following:

- ✓ Create a view model class for the data that you need to pass to the view.
- ✓ Create the About method in the Home controller.

✓ Create the About view.

9.7. Create the view model

Create a SchoolViewModels folder **in** the Models folder.

In the **SchoolViewModels** folder, add a class file **EnrollmentDateGroup.cs** and replace the template code with the following code:

```
using System.ComponentModel.DataAnnotations;
namespace ContosoUniversity.Models.SchoolViewModels
{
    public class EnrollmentDateGroup
    {
        [DataType(DataType.Date)]
        public DateTime? EnrollmentDate { get; set; }

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

Modify the Home Controller

In **HomeController.cs**, add the following using statements at the top of the file:

```
using Microsoft.EntityFrameworkCore;
using ContosoUniversity.Data;
using ContosoUniversity.Models.SchoolViewModels;
using Microsoft.Extensions.Logging;
```

Add a class variable for the database context immediately after the opening curly brace for the class, and get an instance of the context from ASP.NET Core DI:

```
private readonly ILogger<HomeController> _logger;
private readonly SchoolContext _context;

public HomeController(ILogger<HomeController> logger, SchoolContext context)
{
    _logger = logger;
    _context = context;
}
```

Add an **About** method with the following code:

```
public async Task<ActionResult> About()
{
    IQueryable<EnrollmentDateGroup> data =
        from student in _context.Students
        group student by student.EnrollmentDate into dateGroup
        select new EnrollmentDateGroup()
        {
            EnrollmentDate = dateGroup.Key,
            StudentCount = dateGroup.Count()
        };
    return View(await data.AsNoTracking().ToListAsync());
}
```

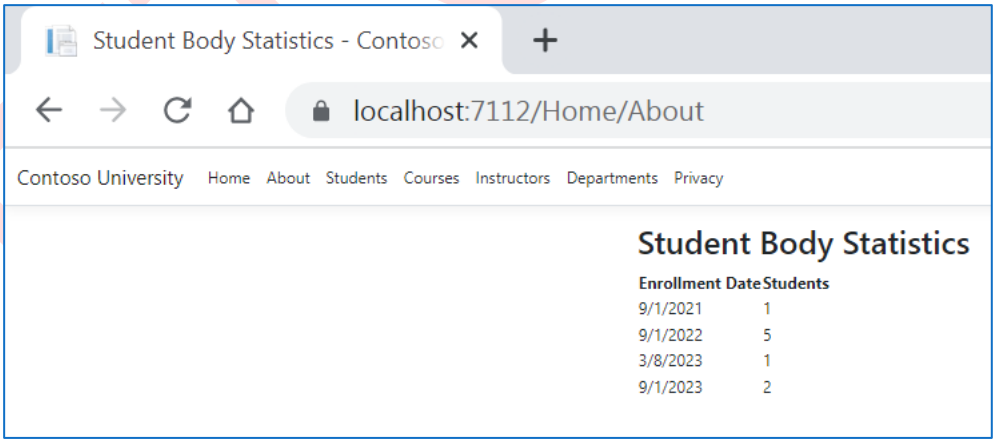
The LINQ statement groups the student entities by enrollment date, calculates the number of entities in each group, and stores the results in a collection of **EnrollmentDateGroup** view model objects.

Create the About View

Add a **Views/Home/About.cshtml** file with the following code:

```
@model IEnumerable<ContosoUniversity.Models.SchoolViewModels.EnrollmentDateGroup>

@{
    ViewData["Title"] = "Student Body Statistics";
}
<h2>Student Body Statistics</h2>

<table>
    <tr>
        <th>
            Enrollment Date
        </th>
        <th>
            Students
        </th>
    </tr>

    @foreach (var item in Model)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                @item.StudentCount
            </td>
        </tr>
    }
</table>
```

Run the app and go to the About page. The count of students for each enrollment date is displayed in a table.



10. Migrations

When developing a new application, the data model undergoes frequent changes, resulting in a mismatch between the model and the database. Initially, you configured the Entity Framework to create the database if it doesn't exist. However, whenever the data

model is modified - by adding, removing, or changing entity classes or updating the DbContext class - you would delete the existing database. The Entity Framework then creates a new database that aligns with the updated model and populates it with test data.

While this approach effectively maintains synchronization between the data model and the database during development, it poses challenges when deploying the application to production. In a production environment, preserving the existing data becomes crucial, and losing all data with each change, such as adding a new column, is undesirable. To address this issue, the EF Core Migrations feature offers a solution by allowing EF to update the database schema instead of creating a brand new database..

To work with migrations, you can use the Package Manager Console (PMC) or the CLI.

10.1. Drop the database

Install EF Core tools as a global tool and delete the database:

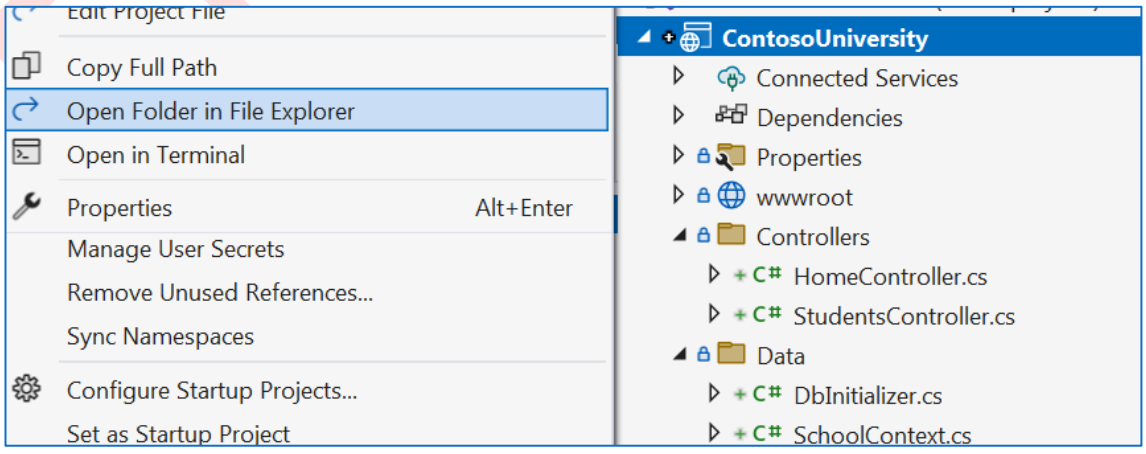
```
dotnet tool install --global dotnet-ef
dotnet ef database drop
```

10.2. Create an initial migration

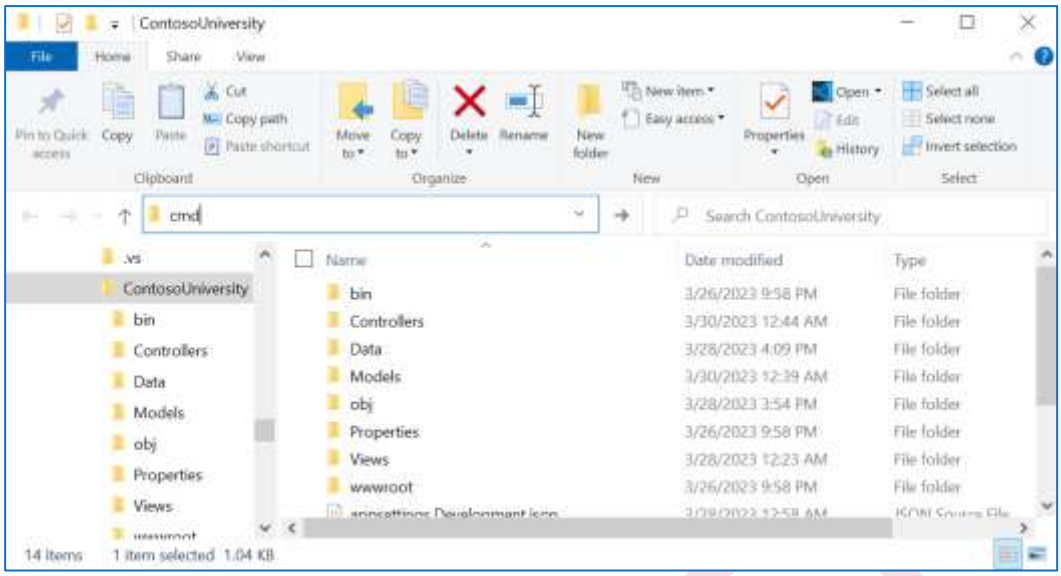
Save your changes and **build** the project. Then open a command window and navigate to the project folder. Here's a quick way to do that:

– In **Solution Explorer**, right-click the project and choose Open Folder in File Explorer from the context menu.

❖



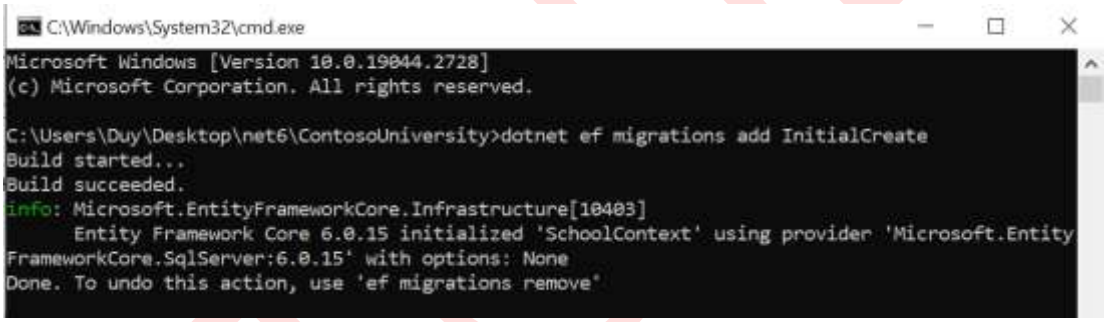
– **Enter "cmd"** in the address bar and press Enter.



Enter the following command in the command window:

```
dotnet ef migrations add InitialCreate
```

In the preceding commands, output similar to the following is displayed:



If you see an error message "cannot access the file ... ContosoUniversity.dll because it is being used by another process.", find the IIS Express icon in the Windows System Tray, and right-click it, then click ContosoUniversity > Stop Site.

10.3. Examine Up and Down methods

When you executed the **migrations add** command, EF generated the code that will create the database from scratch. This code is in the Migrations folder, in the file named **<timestamp>_InitialCreate.cs**. The **Up** method of the **InitialCreate** class creates the database tables that correspond to the data model entity sets, and the **Down** method deletes them, as shown in the following example.

```
public partial class InitialCreate : Migration
    {
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "Course",
                columns: table => new
```

```
                {
                    CourseID = table.Column<int>(type: "int", nullable: false),
                    Title = table.Column<string>(type: "nvarchar(max)", nullable: false),
                    Credits = table.Column<int>(type: "int", nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Course", x => x.CourseID);
                });

        }

        protected override void Down(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.DropTable(
                name: "Enrollment");
    }
}
```

Migrations calls the **Up** method to implement the data model changes for a migration. When you enter a command to roll back the update, Migrations calls the **Down** method.

The provided code pertains to the initial migration generated upon executing the "migrations add InitialCreate" command. The migration name parameter, represented as "InitialCreate" in this instance, serves as the filename and can be customized according to your preference. It is advisable to select a word or phrase that succinctly describes the purpose of the migration. As an illustration, a subsequent migration could be named "AddDepartmentTable".

10.4. The data model snapshot

Migrations creates a snapshot of the current database schema in **Migrations/SchoolContextModelSnapshot.cs**. When you add a migration, EF determines what changed by comparing the data model to the snapshot file.

To remove a migration, you can utilize the "dotnet ef migrations remove" command. This command effectively deletes the migration and ensures that the snapshot is appropriately reset. In the event that the removal process encounters an error, you can try using the "dotnet ef migrations remove -v" command to obtain more detailed information about the failure.

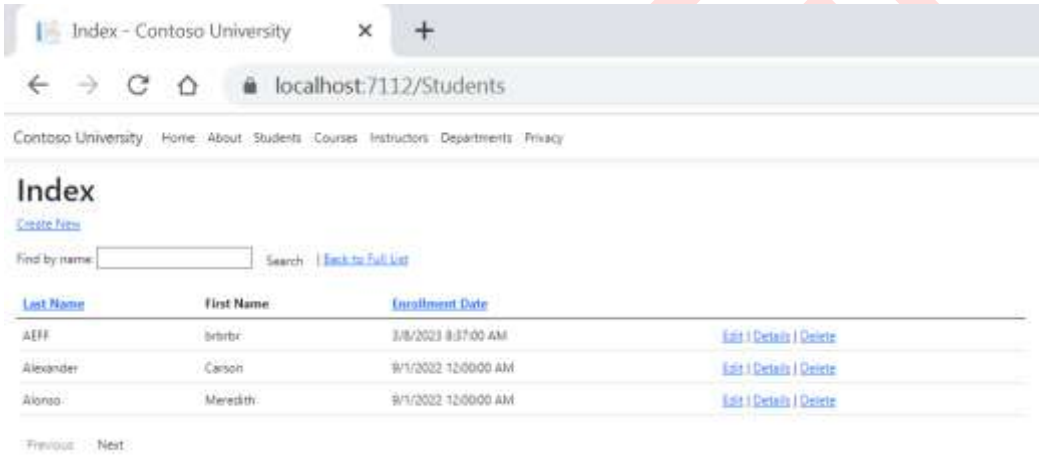
Apply the migration

To establish the database and its corresponding tables, input the provided command within the command window.

dotnet ef database update

The command's output closely resembles the migrations add command, but with additional logs displaying the SQL commands responsible for configuring the database. The sample output provided below excludes most of these logs for brevity. If you wish to suppress such detailed log messages, you can modify the log level within the appsettings.Development.json file.

Run the application to verify that everything still works the same as before.



SUBMIT YOUR CODE TO GIT!

- - - END LAB 06 - - -

REFERENCES

[1]. Hejlsberg A., Torgersen M.(2010), "*The C# Programming Language"*. 4th edition, Addison-Wesley.

[2]. Adam Freeman(2018), "*Pro Entity Framework Core 2 for ASP.NET Core MVC*" 1st ed. Edition.

[3]. HTML Tutorial: https://www.w3schools.com/html/default.asp

[4]. CSS Tutorial: https://www.w3schools.com/css/default.asp

[5]. JavaScript Tutorial: https://www.w3schools.com/js/default.asp

[6]. Bootstrap 5 Tutorial: https://www.w3schools.com/bootstrap5/index.php

[7]. Introduction to Minimal APIs in .NET 6, https://www.claudiobernasconi.ch/2022/02/23/introduction-to-minimal-apis-in-dotnet6

[8]. Minimal APIs quick reference, https://learn.microsoft.com/en-us/aspnet/core/fundamentals/minimal-apis?view=aspnetcore-7.0

[9]. Google Search for key word "*C#"; "WebApp*"; ".NET 6"