# ASP.NET CORE

ASP.NET Core is a **cross-platform**, high-performance, [open-source](https://github.com/aspnet/home) framework for building modern, cloud-based, Internet-connected applications. With ASP.NET Core, you can:

* Build web apps and services, [IoT](https://www.microsoft.com/internet-of-things/) apps, and mobile backends.
* Use your favorite development tools on Windows, macOS, and Linux.
* Deploy to the cloud or on-premises.
* Run on [.NET Core or .NET Framework](https://docs.microsoft.com/en-us/dotnet/articles/standard/choosing-core-framework-server).

Cross-Platform: What is cross platform mean: Runs on macOS, Linux, and Windows.

**Choosing between .NET Core and .NET Framework for server apps:**

There are two supported implementations for building server-side applications with .NET:

.NET Framework and .NET Core. Both share many of the same components and you can share code across the two. However, there are fundamental differences between the two and your choice depends on what you want to accomplish. This article provides guidance on when to use each.

Use .NET Core for your server application when:

* You have cross-platform needs.
* You are targeting microservices.
* You are using Docker containers.
* You need high-performance and scalable systems.
* You need side-by-side .NET versions per application.
* Use .NET Framework for your server application when:
* Your app currently uses .NET Framework (recommendation is to extend instead of migrating).
* Your app uses third-party .NET libraries or NuGet packages not available for .NET Core.
* Your app uses .NET technologies that aren't available for .NET Core.
* Your app uses a platform that doesn’t support .NET Core. Windows, macOS, and Linux support .NET Core.

We recommend the following sequence of tutorials and articles for an introduction to developing ASP.NET Core apps:

Follow a tutorial for the type of app you want to develop or maintain:

| App type | Scenario | Tutorial |
| --- | --- | --- |
| Web app | For new development | [Get started with Razor Pages](https://docs.microsoft.com/en-us/aspnet/core/tutorials/razor-pages/razor-pages-start?view=aspnetcore-3.1) |
| Web app | For maintaining an MVC app | [Get started with MVC](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc?view=aspnetcore-3.1) |
| Web API |  | [Create a web API](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-3.1)\* |
| Real-time app |  | [Get started with SignalR](https://docs.microsoft.com/en-us/aspnet/core/tutorials/signalr?view=aspnetcore-3.1) |
| Blazor app |  | [Get started with Blazor](https://docs.microsoft.com/en-us/aspnet/core/blazor/get-started?view=aspnetcore-3.1) |
| Remote Procedure Call app |  | [Get started with a gRPC service](https://docs.microsoft.com/en-us/aspnet/core/tutorials/grpc/grpc-start?view=aspnetcore-3.1) |

Follow a tutorial that shows how to do basic data access:

| **Scenario** | **Tutorial** |
| --- | --- |
| For new development | [Razor Pages with Entity Framework Core](https://docs.microsoft.com/en-us/aspnet/core/data/ef-rp/intro?view=aspnetcore-3.1) |
| For maintaining an MVC app | [MVC with Entity Framework Core](https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/intro?view=aspnetcore-3.1) |

What is "managed code"?

https://docs.microsoft.com/en-us/dotnet/standard/clr

‎

When working with .NET Framework, you will often encounter the term "managed code". This document will explain what this term means and additional information around it.

To put it very simply, managed code is just that: code whose execution is managed by a runtime. In this case, the runtime in question is called the **Common Language Runtime** or CLR, regardless of the implementation ([Mono](https://www.mono-project.com/) or .NET Framework or .NET Core). CLR is in charge of taking the managed code, compiling it into machine code and then executing it. On top of that, runtime provides several important services such as automatic memory management, security boundaries, type safety etc.

Contrast this to the way you would run a C/C++ program, also called "unmanaged code". In the unmanaged world, the programmer is in charge of pretty much everything. The actual program is, essentially, a binary that the operating system (OS) loads into memory and starts. Everything else, from memory management to security considerations are a burden of the programmer.

Managed code is written in one of the high-level languages that can be run on top of .NET, such as C#, Visual Basic, F# and others. When you compile code written in those languages with their respective compiler, you don’t get machine code. You get **Intermediate Language** code which the runtime then compiles and executes. C++ is the one exception to this rule, as it can also produce native, unmanaged binaries that run on Windows.

Intermediate Language & execution

What is "Intermediate Language" (or IL for short)? It is a product of compilation of code written in high-level .NET languages. Once you compile your code written in one of these languages, you will get a binary that is made out of IL. It is important to note that the IL is independent from any specific language that runs on top of the runtime; there is even a separate specification for it that you can read if you’re so inclined.

Once you produce IL from your high-level code, you will most likely want to run it. This is where the CLR takes over and starts the process of **Just-In-Time** compiling, or **JIT-ing** your code from IL to machine code that can actually be run on a CPU. In this way, the CLR knows exactly what your code is doing and can effectively manage it.

Intermediate Language is sometimes also called Common Intermediate Language (CIL) or Microsoft Intermediate Language (MSIL).

Unmanaged code interoperability

Of course, the CLR allows passing the boundaries between managed and unmanaged world, and there is a lot of code that does that, even in the [Base Class Libraries](https://docs.microsoft.com/en-us/dotnet/standard/framework-libraries). This is called **interoperability** or just **interop** for short. These provisions would allow you to, for example, wrap up an unmanaged library and call into it. However, it is important to note that once you do this, when the code passes the boundaries of the runtime, the actual management of the execution is again in the hand of unmanaged code, and thus falls under the same restrictions.

Similar to this, C# is one language that allows you to use unmanaged constructs such as pointers directly in code by utilizing what is known as **unsafe context** which designates a piece of code for which the execution is not managed by the CLR.

**Create a web UI with ASP.NET Core**

<https://docs.microsoft.com/en-us/aspnet/core/tutorials/razor-pages/page?view=aspnetcore-3.1&tabs=visual-studio>

1. ASP.NET Core supports creating webpages using a built-in templating engine called Razor. In this module, you'll learn how to create webpages using Razor with ASP.NET Core.

In this module, you will:

1. Understand when and why to use Razor Pages for your ASP.NET Core application.
2. Create a new Razor page and its paired C# PageModel class file.
3. Complete the projects Create, Read Update and Delete (CRUD) operations by adding a Create form using Razor syntax.
4. Add built-in client-side form input validation using Razor's Input Tag Helper.
5. Add built-in server-side model validation using Data Annotations.
6. Add a reference to a Data Transfer Object (DTO).
7. Consume a RESTful service from your PageModel.
8. Walk through the life cycle of an example HTTP request.
9. Deploy & Test.

Objective: To build a cross-platform ASP.NET Core Razor Pages web application with .NET Core and C#. The web application will consume an existing web API which supports CRUD operations on product data.

RESTful service concepts and HTTP action verbs, such as GET, POST, PUT, and DELETE

When and why to use Razor Pages:

## What Razor Pages are and the benefit they provide

Razor Pages is a server-side, page-centric programming model for building rich web UI with ASP.NET Core. Razor Pages makes it easy to get started building dynamic web applications when all you need is to define UI logic using a combination of HTML, CSS, and C#. Razor Pages provides a productivity advantage over the more complex **M**odel-**V**iew-**C**ontroller (MVC) application model. Razor Pages encourages organization of files by feature, therefore easing maintenance of your application. Razor Pages can be broadly described as an HTML file where you can work with markup as you are used to, but you also have the advantage of adding server-side C# code by using Razor syntax. Razor pages have the extension .cshtml.

**Use** **Razor Pages in your ASP.NET Core application when:**

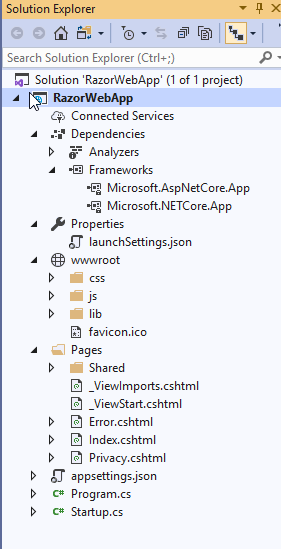
* You want to generate dynamic UI for a browser from your ASP.NET Core application.
* You prefer a page-focused approach to developing web applications, where the page markup and *PageModel* are in close proximity.
* You want your page-focused ASP.NET Core application to use shared common HTML elements across your site and reusable partial views.
* Razor Pages allow you to keep your ASP.NET Core pages organized in a simpler way:
* All view (page) specific logic and page properties defined in the Razor page's (*PageModel*) can be kept together in their own namespace and directory.
* Groups of related pages can be kept in their own namespace and directory

## Create a Razor Pages web app

* From the Visual Studio **File** menu, select **New** > **Project**.
* Create a new ASP.NET Core Web Application and select **Next**.
* Select **ASP.NET Core 3.1** in the dropdown, **Web Application**, and then select **Create**.
* Run the App
* Select **Yes** if you trust the IIS Express SSL certificate.
* Visual Studio starts [IIS Express](https://docs.microsoft.com/en-us/iis/extensions/introduction-to-iis-express/iis-express-overview) and runs the app. The address bar shows localhost:port# and not something like example.com. That's because localhost is the standard hostname for the local computer. Localhost only serves web requests from the local computer. When Visual Studio creates a web project, a random port is used for the web server.

## Examine the project files

Here's an overview of the main project folders and files that you'll work with in later tutorials.



**Pages folder**

Contains Razor pages and supporting files. Each Razor page is a pair of files:

A .cshtml file that contains HTML markup with C# code using Razor syntax.

A .cshtml.cs file that contains C# code that handles page events.

Supporting files have names that begin with an underscore. For example, the \_Layout.cshtml file configures UI elements common to all pages. This file sets up the navigation menu at the top of the page and the copyright notice at the bottom of the page

**wwwroot folder**

Contains static files, such as HTML files, JavaScript files, and CSS files

**appSettings.json**

Contains configuration data, such as connection strings.

**Program.cs**

Contains the entry point for the program.

**Startup.cs**

Contains code that configures app behavior.

# **Add a model to a Razor Pages app in ASP.NET Core**

Entity Framework (EF) Core is an object-relational mapping (ORM) framework that simplifies data access.

The model classes are known as POCO classes (from "plain-old CLR objects") because they don't have any dependency on EF Core. They define the properties of the data that are stored in the database.

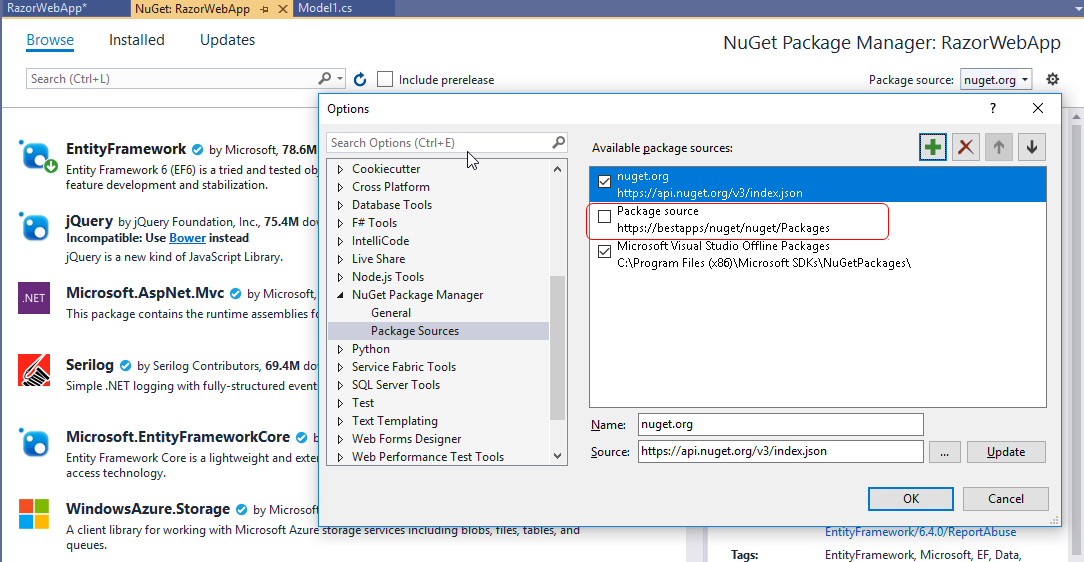
## Add a data model (Models are like classes, where properties can be defined)

* Right-click the **RazorPagesMovie** project > **Add** > **New Folder**. Name the folder *Models*.
* Right click the *Models* folder. Select **Add** > **Class**. Name the class **Movie**.

# Add Entity frameworkCore Nuget package:

Add Entity Frameowrk Nuget Package from Menu Project>Manage Nuget Packages.

**Note**: uncheck the packages from the ambest site(This is not functional yet. So can cause issues)



**ADD Scaffold item model**

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

Initial migration

[Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/razor-pages/model?view=aspnetcore-3.1&tabs=visual-studio#tabpanel_CeZOj-G++Q-3_visual-studio)

[Visual Studio Code](https://docs.microsoft.com/en-us/aspnet/core/tutorials/razor-pages/model?view=aspnetcore-3.1&tabs=visual-studio#tabpanel_CeZOj-G++Q-3_visual-studio-code)

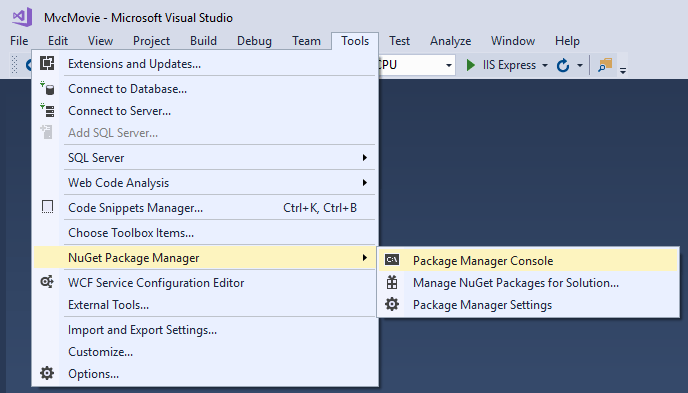
[Visual Studio for Mac](https://docs.microsoft.com/en-us/aspnet/core/tutorials/razor-pages/model?view=aspnetcore-3.1&tabs=visual-studio#tabpanel_CeZOj-G++Q-3_visual-studio-mac)

In this section, the Package Manager Console (PMC) is used to:

Add an initial migration.

Update the database with the initial migration.

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



In the PMC, enter the following commands:

PMC Copy

Add-Migration InitialCreate

Update-Database

The preceding commands generate the following warning: "No type was specified for the decimal column '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 using 'HasColumnType()'."

You can ignore that warning, it will be fixed in a later tutorial.

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

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