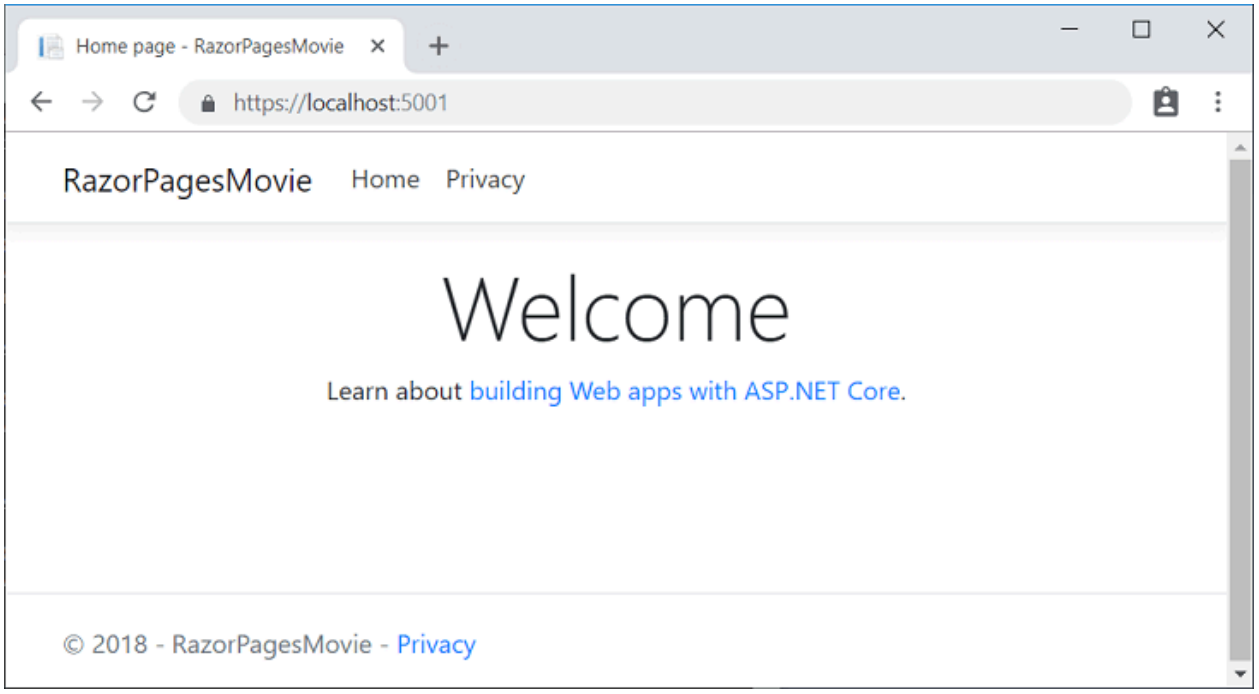
Response compression

ASP.NET Core 2.2 can compress responses with the Brotli compression format.

For more information, see Response Compression Middleware supports Brotli compression.

Project templates

ASP.NET Core web project templates were updated to Bootstrap 4 and Angular 6. The new look is visually simpler and makes it easier to see the important structures of the app.



Validation performance

MVC's validation system is designed to be extensible and flexible, allowing you to determine on a per request basis which validators apply to a given model. This is great for authoring complex validation providers. However, in the most common case an application only uses the built-in validators and don't require this extra flexibility. Built-in validators include DataAnnotations such as [Required] and [StringLength], and `IValidatableObject`.

In ASP.NET Core 2.2, MVC can short-circuit validation if it determines that a given model graph doesn't require validation. Skipping validation results in significant improvements when validating models that can't or don't have any validators. This includes objects

such as collections of primitives (such as `byte[]`, `string[]`, `Dictionary<string, string>`), or complex object graphs without many validators.

HTTP Client performance

In ASP.NET Core 2.2, the performance of `SocketsHttpHandler` was improved by reducing connection pool locking contention. For apps that make many outgoing HTTP requests, such as some microservices architectures, throughput is improved. Under load, `HttpClient` throughput can be improved by up to 60% on Linux and 20% on Windows.

For more information, see the pull request that made this improvement.

Additional information

For the complete list of changes, see the ASP.NET Core 2.2 Release Notes.

What's new in ASP.NET Core 2.1

Article • 08/30/2024

This article highlights the most significant changes in ASP.NET Core 2.1, with links to relevant documentation.

SignalR

SignalR has been rewritten for ASP.NET Core 2.1.

ASP.NET Core SignalR includes a number of improvements:

- A simplified scale-out model.
- A new JavaScript client with no jQuery dependency.
- A new compact binary protocol based on MessagePack.
- Support for custom protocols.
- A new streaming response model.
- Support for clients based on bare WebSockets.

For more information, see ASP.NET Core SignalR.

Razor class libraries

ASP.NET Core 2.1 makes it easier to build and include Razor-based UI in a library and share it across multiple projects. The new Razor SDK enables building Razor files into a class library project that can be packaged into a NuGet package. Views and pages in libraries are automatically discovered and can be overridden by the app. By integrating Razor compilation into the build:

- The app startup time is significantly faster.
- Fast updates to Razor views and pages at runtime are still available as part of an iterative development workflow.

For more information, see Reusable Razor UI in class libraries with ASP.NET Core.

Identity UI library & scaffolding

ASP.NET Core 2.1 provides ASP.NET Core Identity as a Razor class library. Apps that include Identity can apply the new Identity scaffolder to selectively add the source code contained in the Identity Razor class library (RCL). You might want to generate source

code so you can modify the code and change the behavior. For example, you could instruct the scaffolder to generate the code used in registration. Generated code takes precedence over the same code in the Identity RCL.

Apps that do **not** include authentication can apply the Identity scaffolder to add the RCL Identity package. You have the option of selecting Identity code to be generated.

For more information, see Scaffold Identity in ASP.NET Core projects.

HTTPS

With the increased focus on security and privacy, enabling HTTPS for web apps is important. HTTPS enforcement is becoming increasingly strict on the web. Sites that don't use HTTPS are considered insecure. Browsers (Chromium, Mozilla) are starting to enforce that web features must be used from a secure context. GDPR requires the use of HTTPS to protect user privacy. While using HTTPS in production is critical, using HTTPS in development can help prevent issues in deployment (for example, insecure links). ASP.NET Core 2.1 includes a number of improvements that make it easier to use HTTPS in development and to configure HTTPS in production. For more information, see Enforce HTTPS.

On by default

To facilitate secure website development, HTTPS is now enabled by default. Starting in 2.1, Kestrel listens on `https://localhost:5001` when a local development certificate is present. A development certificate is created:

- As part of the .NET Core SDK first-run experience, when you use the SDK for the first time.
- Manually using the new `dev-certs` tool.

Run `dotnet dev-certs https --trust` to trust the certificate.

HTTPS redirection and enforcement

Web apps typically need to listen on both HTTP and HTTPS, but then redirect all HTTP traffic to HTTPS. In 2.1, specialized HTTPS redirection middleware that intelligently redirects based on the presence of configuration or bound server ports has been introduced.

Use of HTTPS can be further enforced using HTTP Strict Transport Security Protocol (HSTS). HSTS instructs browsers to always access the site via HTTPS. ASP.NET Core 2.1

adds HSTS middleware that supports options for max age, subdomains, and the HSTS preload list.

Configuration for production

In production, HTTPS must be explicitly configured. In 2.1, default configuration schema for configuring HTTPS for Kestrel has been added. Apps can be configured to use:

- Multiple endpoints including the URLs. For more information, see Kestrel web server implementation: Endpoint configuration.
- The certificate to use for HTTPS either from a file on disk or from a certificate store.

GDPR

ASP.NET Core provides APIs and templates to help meet some of the EU General Data Protection Regulation (GDPR) requirements. For more information, see GDPR support in ASP.NET Core. A sample app shows how to use and lets you test most of the GDPR extension points and APIs added to the ASP.NET Core 2.1 templates.

Integration tests

A new package is introduced that streamlines test creation and execution. The Microsoft.AspNetCore.Mvc.Testing package handles the following tasks:

- Copies the dependency file (*.*deps*) from the tested app into the test project's *bin* folder.
- Sets the content root to the tested app's project root so that static files and pages/views are found when the tests are executed.
- Provides the WebApplicationFactory<TEntryPoint> class to streamline bootstrapping the tested app with TestServer.

The following test uses xUnit to check that the Index page loads with a success status code and with the correct Content-Type header:

C#

```
public class BasicTests
    : IClassFixture<WebApplicationFactory<RazorPagesProject.Startup>>
{
    private readonly HttpClient _client;

    public BasicTests(WebApplicationFactory<RazorPagesProject.Startup>
factory)
```

```
    {
        _client = factory.CreateClient();
    }

    [Fact]
    public async Task GetHomePage()
    {
        // Act
        var response = await _client.GetAsync("/");

        // Assert
        response.EnsureSuccessStatusCode(); // Status Code 200-299
        Assert.Equal("text/html; charset=utf-8",
            response.Content.Headers.ContentType.ToString());
    }
}
```

For more information, see the Integration tests topic.

[ApiController], ActionResult<T>

ASP.NET Core 2.1 adds new programming conventions that make it easier to build clean and descriptive web APIs. `ActionResult<T>` is a new type added to allow an app to return either a response type or any other action result (similar to IActionResult), while still indicating the response type. The `[ApiController]` attribute has also been added as the way to opt in to Web API-specific conventions and behaviors.

For more information, see Build Web APIs with ASP.NET Core.

IHttpClientFactory

ASP.NET Core 2.1 includes a new `IHttpClientFactory` service that makes it easier to configure and consume instances of `HttpClient` in apps. `HttpClient` already has the concept of delegating handlers that could be linked together for outgoing HTTP requests. The factory:

- Makes registering of instances of `HttpClient` per named client more intuitive.
- Implements a Polly handler that allows Polly policies to be used for Retry, CircuitBreakers, etc.

For more information, see Initiate HTTP Requests.

Kestrel libuv transport configuration

With the release of ASP.NET Core 2.1, Kestrel's default transport is no longer based on Libuv but instead based on managed sockets. For more information, see Kestrel web server implementation: Libuv transport configuration.

Generic host builder

The Generic Host Builder (`HostBuilder`) has been introduced. This builder can be used for apps that don't process HTTP requests (Messaging, background tasks, etc.).

For more information, see .NET Generic Host.

Updated SPA templates

The Single Page Application templates for Angular and React are updated to use the standard project structures and build systems for each framework.

The Angular template is based on the Angular CLI, and the React template is based on create-react-app.

For more information, see:

- Use Angular with ASP.NET Core
- Use React with ASP.NET Core

Razor Pages search for Razor assets

In 2.1, Razor Pages search for Razor assets (such as layouts and partials) in the following directories in the listed order:

1. Current Pages folder.
2. */Pages/Shared/*
3. */Views/Shared/*

Razor Pages in an area

Razor Pages now support areas. To see an example of areas, create a new Razor Pages web app with individual user accounts. A Razor Pages web app with individual user accounts includes */Areas/Identity/Pages*.

MVC compatibility version

The SetCompatibilityVersion method allows an app to opt-in or opt-out of potentially breaking behavior changes introduced in ASP.NET Core MVC 2.1 or later.

For more information, see Compatibility version for ASP.NET Core MVC.

Migrate from 2.0 to 2.1

See Migrate from ASP.NET Core 2.0 to 2.1.

Additional information

For the complete list of changes, see the ASP.NET Core 2.1 Release Notes.

What's new in ASP.NET Core 2.0

Article • 09/18/2024

This article highlights the most significant changes in ASP.NET Core 2.0, with links to relevant documentation.

Razor Pages

Razor Pages is a new feature of ASP.NET Core MVC that makes coding page-focused scenarios easier and more productive.

For more information, see the introduction and tutorial:

- Introduction to Razor Pages
- Get started with Razor Pages

ASP.NET Core metapackage

A new ASP.NET Core metapackage includes all of the packages made and supported by the ASP.NET Core and Entity Framework Core teams, along with their internal and 3rd-party dependencies. You no longer need to choose individual ASP.NET Core features by package. All features are included in the Microsoft.AspNetCore.All package. The default templates use this package.

For more information, see Microsoft.AspNetCore.All metapackage for ASP.NET Core 2.0.

Runtime Store

Applications that use the `Microsoft.AspNetCore.All` metapackage automatically take advantage of the new .NET Core Runtime Store. The Store contains all the runtime assets needed to run ASP.NET Core 2.0 applications. When you use the `Microsoft.AspNetCore.All` metapackage, no assets from the referenced ASP.NET Core NuGet packages are deployed with the application because they already reside on the target system. The assets in the Runtime Store are also precompiled to improve application startup time.

For more information, see Runtime store

.NET Standard 2.0

The ASP.NET Core 2.0 packages target .NET Standard 2.0. The packages can be referenced by other .NET Standard 2.0 libraries, and they can run on .NET Standard 2.0-compliant implementations of .NET, including .NET Core 2.0 and .NET Framework 4.6.1.

The `Microsoft.AspNetCore.All` metapackage targets .NET Core 2.0 only, because it's intended to be used with the .NET Core 2.0 Runtime Store.

Configuration update

An `IConfiguration` instance is added to the services container by default in ASP.NET Core 2.0. `IConfiguration` in the services container makes it easier for applications to retrieve configuration values from the container.

For information about the status of planned documentation, see the GitHub issue.

Logging update

In ASP.NET Core 2.0, logging is incorporated into the dependency injection (DI) system by default. You add providers and configure filtering in the `Program.cs` file instead of in the `Startup.cs` file. And the default `ILoggerFactory` supports filtering in a way that lets you use one flexible approach for both cross-provider filtering and specific-provider filtering.

For more information, see Introduction to Logging.

Authentication update

A new authentication model makes it easier to configure authentication for an application using DI.

New templates are available for configuring authentication for web apps and web APIs using Azure AD B2C.

For information about the status of planned documentation, see the GitHub issue.

Identity update

We've made it easier to build secure web APIs using Identity in ASP.NET Core 2.0. You can acquire access tokens for accessing your web APIs using the Microsoft Authentication Library (MSAL).

For more information on authentication changes in 2.0, see the following resources:

- Account confirmation and password recovery in ASP.NET Core
- Enable QR Code generation for authenticator apps in ASP.NET Core
- Migrate Authentication and Identity to ASP.NET Core 2.0

SPA templates

Single Page Application (SPA) project templates for Angular, Aurelia, Knockout.js, React.js, and React.js with Redux are available. The Angular template has been updated to Angular 4. The Angular and React templates are available by default; for information about how to get the other templates, see Create a new SPA project. For information about how to build a SPA in ASP.NET Core, see The features described in this article are obsolete as of ASP.NET Core 3.0.

Kestrel improvements

The Kestrel web server has new features that make it more suitable as an Internet-facing server. A number of server constraint configuration options are added in the `KestrelServerOptions` class's new `Limits` property. Add limits for the following:

- Maximum client connections
- Maximum request body size
- Minimum request body data rate

For more information, see Kestrel web server implementation in ASP.NET Core.

WebListener renamed to HTTP.sys

The packages `Microsoft.AspNetCore.Server.WebListener` and `Microsoft.Net.Http.Server` have been merged into a new package `Microsoft.AspNetCore.Server.HttpSys`. The namespaces have been updated to match.

For more information, see HTTP.sys web server implementation in ASP.NET Core.

Enhanced HTTP header support

When using MVC to transmit a `FileStreamResult` or a `FileContentResult`, you now have the option to set an `ETag` or a `LastModified` date on the content you transmit. You can set these values on the returned content with code similar to the following:

C#

```
var data = Encoding.UTF8.GetBytes("This is a sample text from a binary
array");
var entityTag = new EntityTagHeaderValue("\"MyCalculatedEtagValue\"");
return File(data, "text/plain", "downloadName.txt", lastModified:
DateTime.UtcNow.AddSeconds(-5), entityTag: entityTag);
```

The file returned to your visitors has the appropriate HTTP headers for the `ETag` and `LastModified` values.

If an application visitor requests content with a Range Request header, ASP.NET Core recognizes the request and handles the header. If the requested content can be partially delivered, ASP.NET Core appropriately skips and returns just the requested set of bytes. You don't need to write any special handlers into your methods to adapt or handle this feature; it's automatically handled for you.

Hosting startup and Application Insights

Hosting environments can now inject extra package dependencies and execute code during application startup, without the application needing to explicitly take a dependency or call any methods. This feature can be used to enable certain environments to "light-up" features unique to that environment without the application needing to know ahead of time.

In ASP.NET Core 2.0, this feature is used to automatically enable Application Insights diagnostics when debugging in Visual Studio and (after opting in) when running in Azure App Services. As a result, the project templates no longer add Application Insights packages and code by default.

For information about the status of planned documentation, see the GitHub issue.

Automatic use of antiforgery tokens

ASP.NET Core has always helped HTML-encode content by default, but with the new version an extra step is taken to help prevent cross-site request forgery (XSRF) attacks. ASP.NET Core will now emit antiforgery tokens by default and validate them on form POST actions and pages without extra configuration.

For more information, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core.

Automatic precompilation

Razor view pre-compilation is enabled during publish by default, reducing the publish output size and application startup time.

For more information, see Razor view compilation and precompilation in ASP.NET Core.

Razor support for C# 7.1

The Razor view engine has been updated to work with the new Roslyn compiler. That includes support for C# 7.1 features like Default Expressions, Inferred Tuple Names, and Pattern-Matching with Generics. To use C# 7.1 in your project, add the following property in your project file and then reload the solution:

XML

```
<LangVersion>latest</LangVersion>
```

For information about the status of C# 7.1 features, see the Roslyn GitHub repository.

Other documentation updates for 2.0

- Visual Studio publish profiles for ASP.NET Core app deployment
- Key Management
- Configure Facebook authentication
- Configure Twitter authentication
- Configure Google authentication
- Configure Microsoft Account authentication

Migration guidance

For guidance on how to migrate ASP.NET Core 1.x applications to ASP.NET Core 2.0, see the following resources:

- Migrate from ASP.NET Core 1.x to ASP.NET Core 2.0
- Migrate Authentication and Identity to ASP.NET Core 2.0

Additional Information

For the complete list of changes, see the ASP.NET Core 2.0 Release Notes.

To connect with the ASP.NET Core development team's progress and plans, tune in to the ASP.NET Community Standup.

What's new in ASP.NET Core 1.1

Article • 06/03/2022

ASP.NET Core 1.1 includes the following new features:

- URL Rewriting Middleware
- Response Caching Middleware
- View Components as Tag Helpers
- Middleware as MVC filters
- Cookie-based TempData provider
- Azure App Service logging provider
- Azure Key Vault configuration provider
- Azure and Redis Storage Data Protection Key Repositories
- WebListener Server for Windows
- WebSockets support

Choosing between versions 1.0 and 1.1 of ASP.NET Core

ASP.NET Core 1.1 has more features than ASP.NET Core 1.0. In general, we recommend you use the latest version.

Additional Information

- ASP.NET Core 1.1.0 Release Notes
- To connect with the ASP.NET Core development team's progress and plans, tune in to the ASP.NET Community Standup .

ASP.NET Core documentation - what's new?

Welcome to what's new in ASP.NET Core docs. Use this page to quickly find the latest changes.

Find ASP.NET Core docs updates

WHAT'S NEW

June 2024

May 2024

April 2024

March 2024

February 2024

January 2024

Get involved - contribute to ASP.NET Core docs

OVERVIEW

ASP.NET Core docs repository

Project structure and labels for issues and pull requests

CONCEPT

Contributor guide

ASP.NET Core docs contributor guide

ASP.NET Core API reference docs contributor guide

Community

WHAT'S NEW

Community

Related what's new pages

WHAT'S NEW

Xamarin docs updates

.NET Core release notes

ASP.NET Core release notes

C# compiler (Roslyn) release notes

Visual Studio release notes

Visual Studio for Mac release notes

Visual Studio Code release notes

Choose an ASP.NET Core web UI

Article • 10/21/2024

ASP.NET Core is a complete UI framework. Choose which functionalities to combine that fit the app's web UI needs.

For new project development, we recommend ASP.NET Core Blazor.

ASP.NET Core Blazor

Blazor is a full-stack web UI framework and is recommended for most web UI scenarios.

Benefits of using Blazor:

- Reusable component model.
- Efficient diff-based component rendering.
- Flexibly render components from the server or client via WebAssembly.
- Build rich interactive web UI components in C#.
- Render components statically from the server.
- Progressively enhance server rendered components for smoother navigation and form handling and to enable streaming rendering.
- Share code for common logic on the client and server.
- Interop with JavaScript.
- Integrate components with existing MVC, Razor Pages, or JavaScript based apps.

For a complete overview of Blazor, its architecture and benefits, see ASP.NET Core Blazor and ASP.NET Core Blazor hosting models. To get started with your first Blazor app, see Build your first Blazor app.

ASP.NET Core Razor Pages

Razor Pages is a page-based model for building server rendered web UI. Razor pages UI are dynamically rendered on the server to generate the page's HTML and CSS in response to a browser request. The page arrives at the client ready to display. Support for Razor Pages is built on ASP.NET Core MVC.

Razor Pages benefits:

- Quickly build and update UI. Code for the page is kept with the page, while keeping UI and business logic concerns separate.
- Testable and scales to large apps.

- Keep your ASP.NET Core pages organized in a simpler way than ASP.NET MVC:
  - View specific logic and view models can be kept together in their own namespace and directory.
  - Groups of related pages can be kept in their own namespace and directory.

To get started with your first ASP.NET Core Razor Pages app, see Tutorial: Get started with Razor Pages in ASP.NET Core. For a complete overview of ASP.NET Core Razor Pages, its architecture and benefits, see: Introduction to Razor Pages in ASP.NET Core.

ASP.NET Core MVC

ASP.NET Core MVC renders UI on the server and uses a Model-View-Controller (MVC) architectural pattern. The MVC pattern separates an app into three main groups of components: models, views, and controllers. User requests are routed to a controller. The controller is responsible for working with the model to perform user actions or retrieve results of queries. The controller chooses the view to display to the user and provides it with any model data it requires.

ASP.NET Core MVC benefits:

- Based on a scalable and mature model for building large web apps.
- Clear separation of concerns for maximum flexibility.
- The Model-View-Controller separation of responsibilities ensures that the business model can evolve without being tightly coupled to low-level implementation details.

To get started with ASP.NET Core MVC, see Get started with ASP.NET Core MVC. For an overview of ASP.NET Core MVC's architecture and benefits, see Overview of ASP.NET Core MVC.

ASP.NET Core Single Page Applications (SPA) with frontend JavaScript frameworks

Build client-side logic for ASP.NET Core apps using popular JavaScript frameworks, like Angular, React, and Vue. ASP.NET Core provides project templates for Angular, React, and Vue, and it can be used with other JavaScript frameworks as well.

Benefits of ASP.NET Core SPA with JavaScript Frameworks, in addition to the client rendering benefits previously listed:

- The JavaScript runtime environment is already provided with the browser.
- Large community and mature ecosystem.

- Build client-side logic for ASP.NET Core apps using popular JS frameworks, like Angular, React, and Vue.

Downsides:

- More coding languages, frameworks, and tools required.
- Difficult to share code so some logic may be duplicated.

To get started, see:

- Create an ASP.NET Core app with Angular
- Create an ASP.NET Core app with React
- Create an ASP.NET Core app with Vue
- JavaScript and TypeScript in Visual Studio

Combine multiple web UI solutions: ASP.NET Core MVC or Razor Pages plus Blazor

MVC, Razor Pages, and Blazor are part of the ASP.NET Core framework and are designed to be used together. Razor components can be integrated into Razor Pages and MVC apps. When a view or page is rendered, components can be prerendered at the same time.

Benefits for MVC or Razor Pages plus Blazor, in addition to MVC or Razor Pages benefits:

- Prerendering executes Razor components on the server and renders them into a view or page, which improves the perceived load time of the app.
- Add interactivity to existing views or pages with the Component Tag Helper.

To get started with ASP.NET Core MVC or Razor Pages plus Blazor, see Integrate ASP.NET Core Razor components into ASP.NET Core apps.

Next steps

For more information, see:

- ASP.NET Core Blazor
- ASP.NET Core Blazor hosting models
- Integrate ASP.NET Core Razor components into ASP.NET Core apps
- Compare gRPC services with HTTP APIs

Tutorial: Create a Razor Pages web app with ASP.NET Core

Article • 07/01/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

This series of tutorials explains the basics of building a Razor Pages web app.

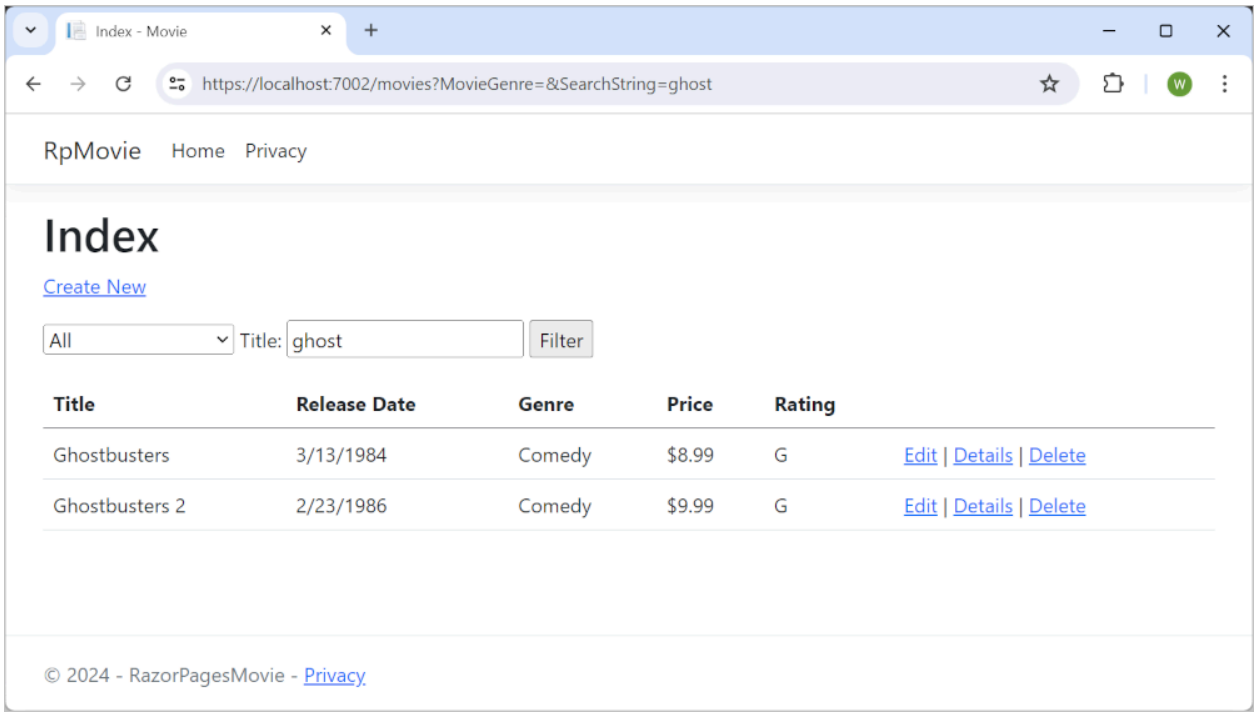
For a more advanced introduction aimed at developers who are familiar with controllers and views, see Introduction to Razor Pages in ASP.NET Core.

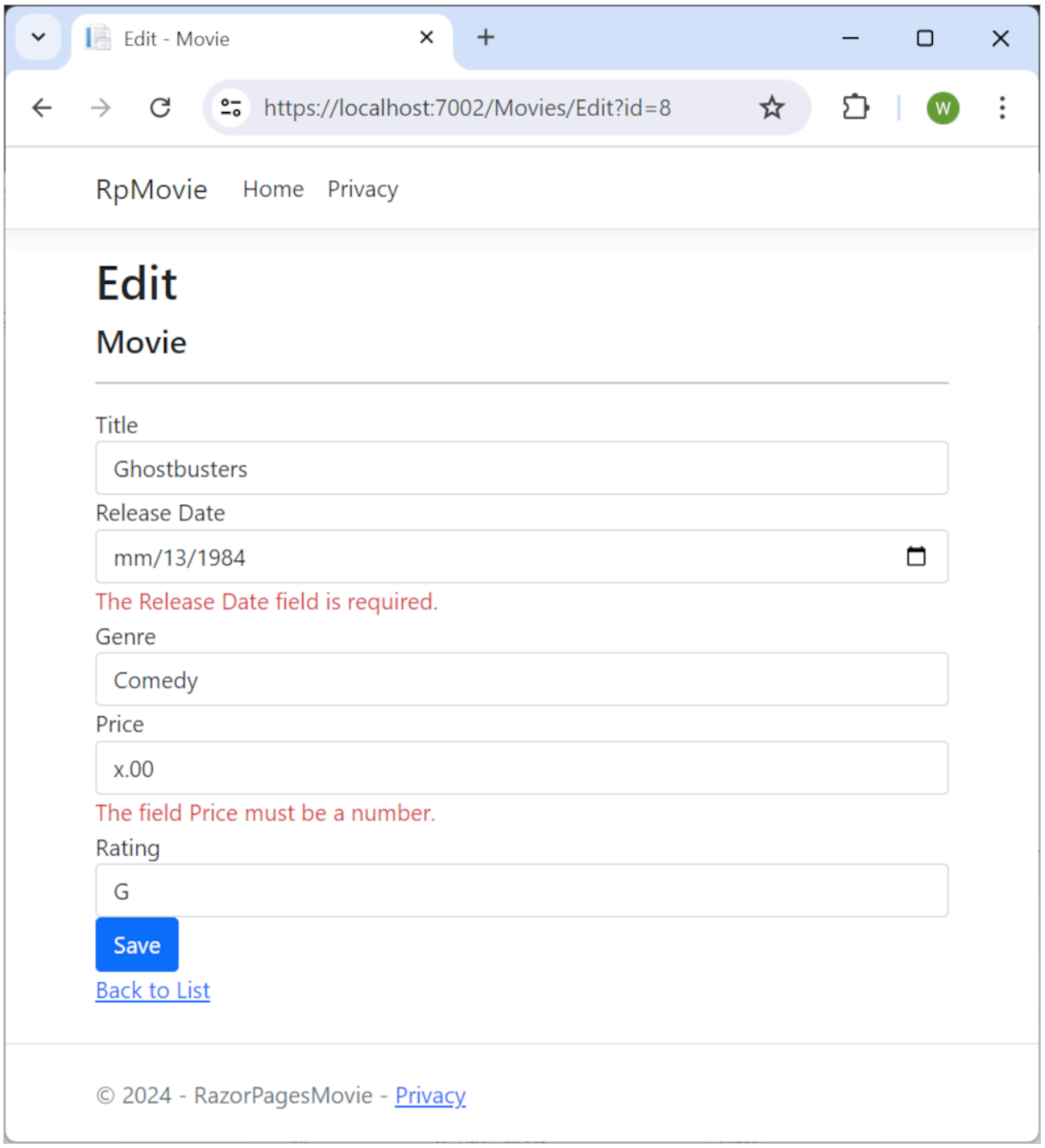
If you're new to ASP.NET Core development and are unsure of which ASP.NET Core web UI solution will best fit your needs, see Choose an ASP.NET Core UI.

This series includes the following tutorials:

1. Create a Razor Pages web app
2. Add a model to a Razor Pages app
3. Scaffold (generate) Razor pages
4. Work with a database
5. Update Razor pages
6. Add search
7. Add a new field
8. Add validation

At the end, you'll have an app that can display and manage a database of movies.


Index - Movie
https://localhost:7002/movies?MovieGenre=&SearchString=ghost
RpMovie
Home
Privacy
Index
Create New
All
Title:
ghost
Filter
Title
Release Date
Genre
Price
Rating
Ghostbusters
3/13/1984
Comedy
$8.99
G
Edit | Details | Delete
Ghostbusters 2
2/23/1986
Comedy
$9.99
G
Edit | Details | Delete
© 2024 - RazorPagesMovie - Privacy


Edit - Movie
https://localhost:7002/Movies/Edit?id=8
RpMovie
Home
Privacy
Edit
Movie
Title
Ghostbusters
Release Date
mm/13/1984
The Release Date field is required.
Genre
Comedy
Price
x.00
The field Price must be a number.
Rating
G
Save
Back to List
© 2024 - RazorPagesMovie - Privacy

Tutorial: Get started with Razor Pages in ASP.NET Core

Article • 08/05/2024

> **ⓘ Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

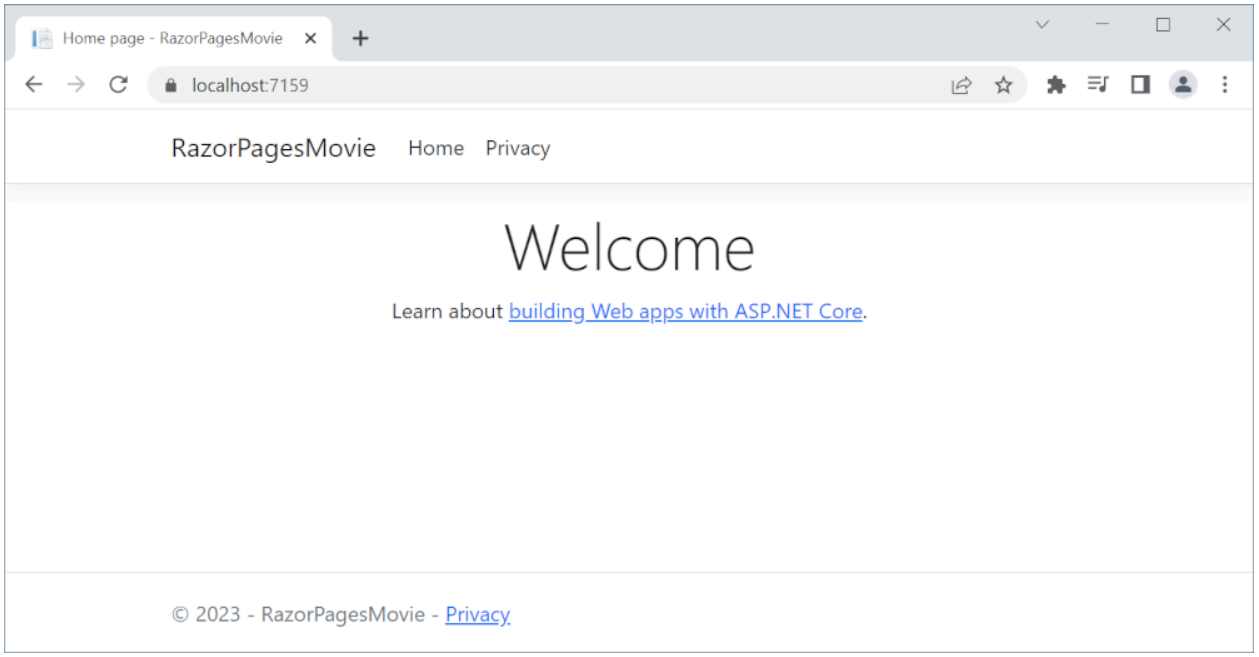
By Rick Anderson

This is the first tutorial of a series that teaches the basics of building an ASP.NET Core Razor Pages web app.

For a more advanced introduction aimed at developers who are familiar with controllers and views, see Introduction to Razor Pages. For a video introduction, see Entity Framework Core for Beginners.

If you're new to ASP.NET Core development and are unsure of which ASP.NET Core web UI solution will best fit your needs, see Choose an ASP.NET Core UI.

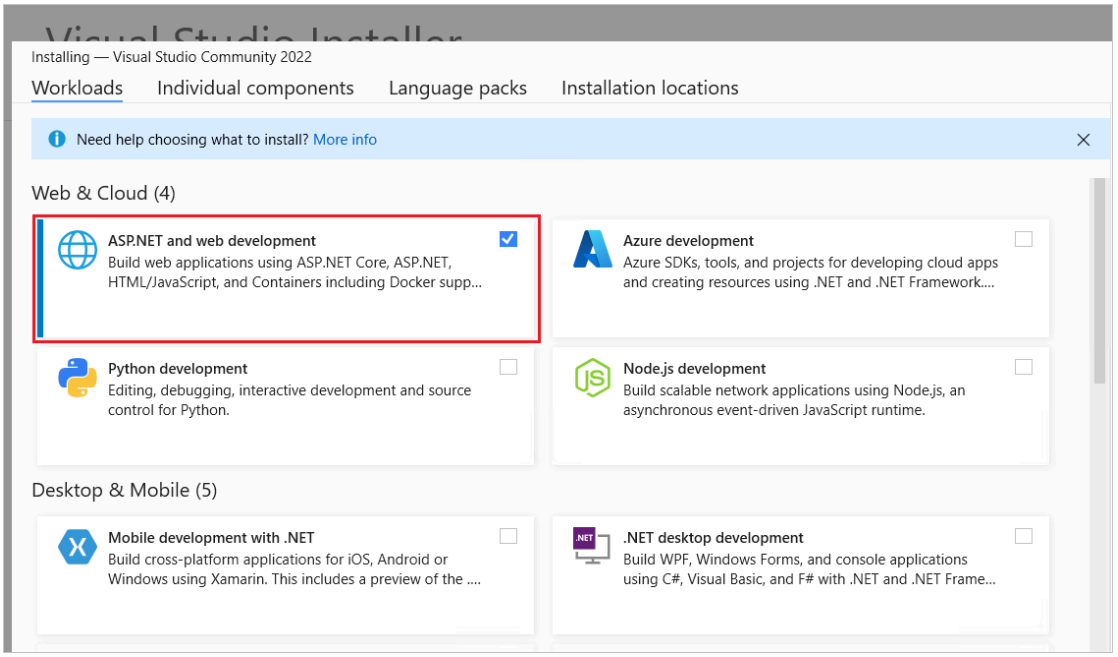
At the end of this tutorial, you'll have a Razor Pages web app that manages a database of movies.



Prerequisites

Visual Studio

- Visual Studio 2022 Preview with the **ASP.NET and web development** workload.



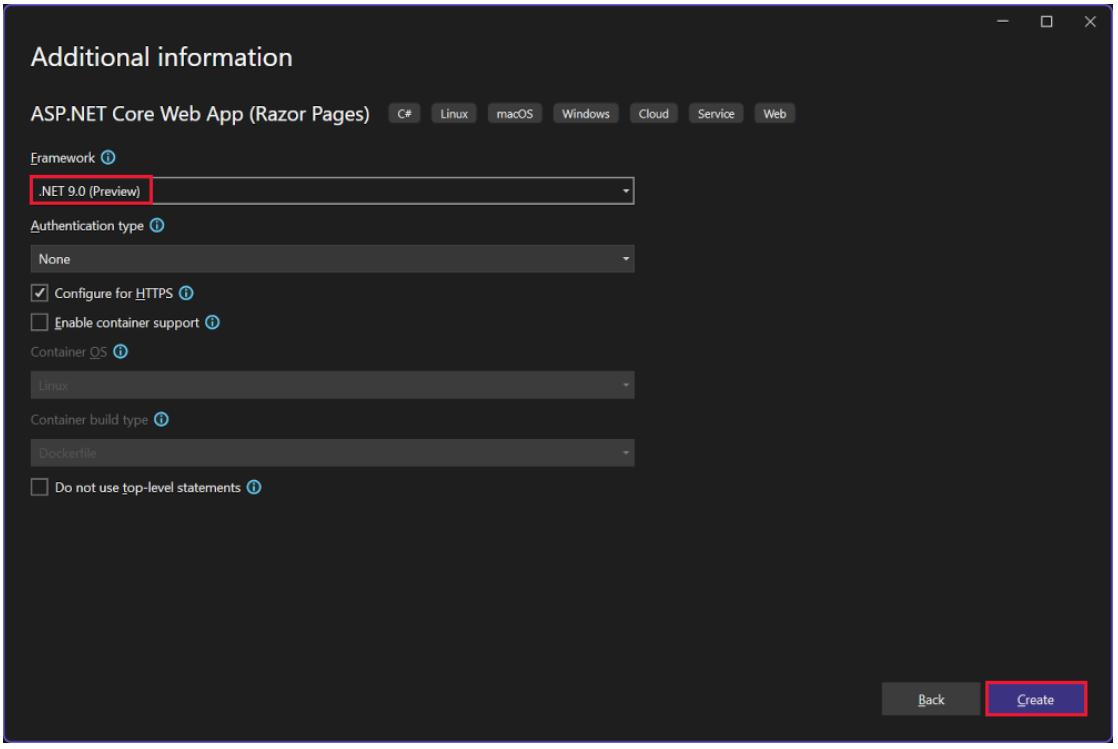
Create a Razor Pages web app

Visual Studio

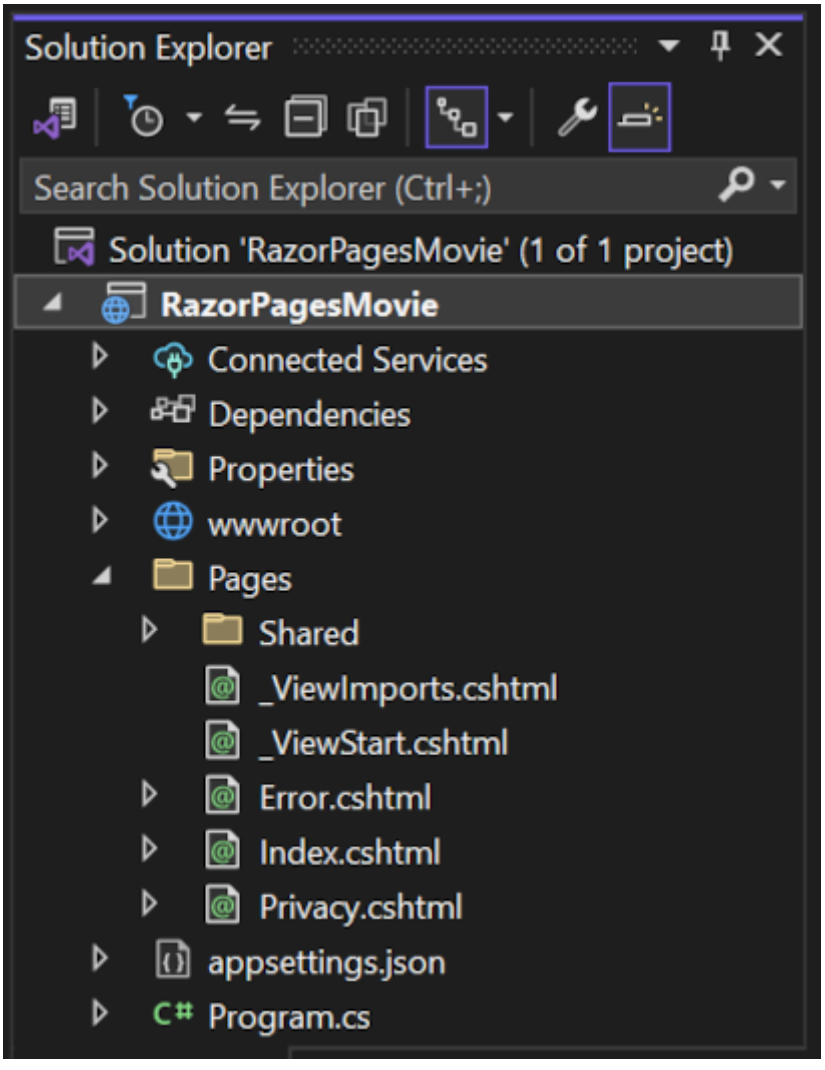
- Start Visual Studio and select **New project**.
- In the **Create a new project** dialog, select **ASP.NET Core Web App (Razor Pages)** > **Next**.
- In the **Configure your new project** dialog, enter `RazorPagesMovie` for **Project name**. It's important to name the project **RazorPagesMovie**, including matching the capitalization, so the namespaces will match when you copy and paste example code.
- Select **Next**.
- In the **Additional information** dialog:
  - Select **.NET 9.0 (Preview)**.

- Verify: **Do not use top-level statements** is unchecked.

- Select **Create**.



The following starter project is created:



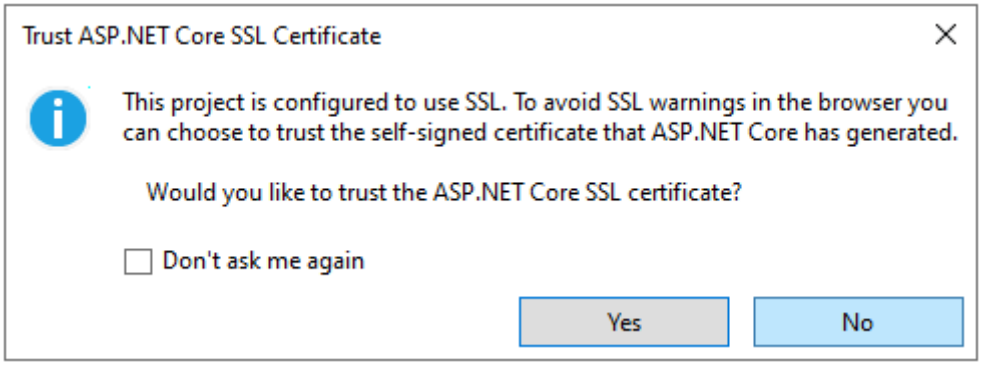
For alternative approaches to create the project, see Create a new project in Visual Studio.

Run the app

Visual Studio

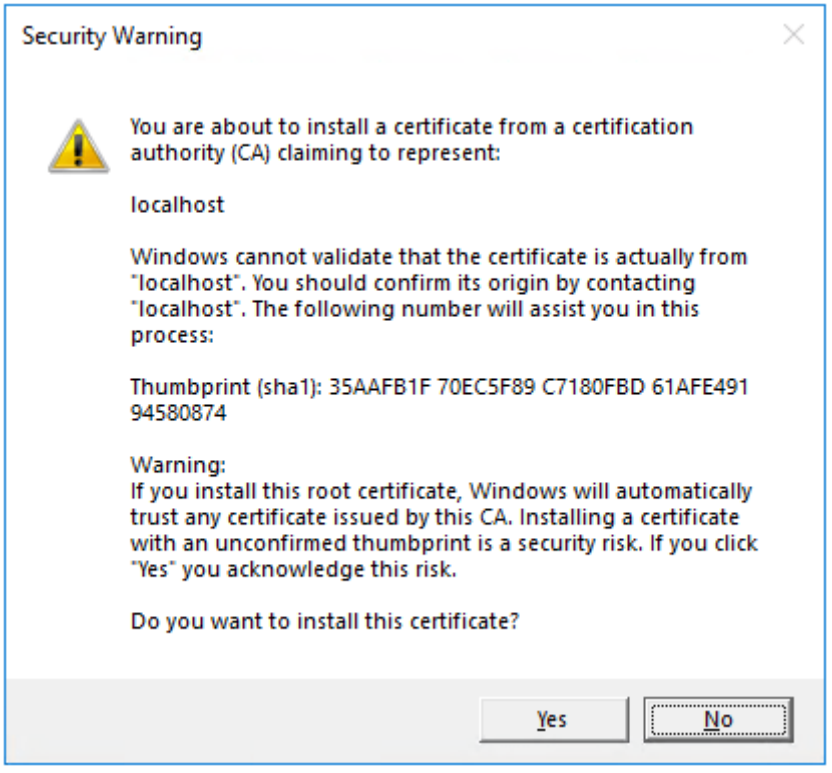
Select **RazorPagesMovie** in **Solution Explorer**, and then press `Ctrl`+`F5` to run the app without the debugger.

Visual Studio displays the following dialog when a project is not yet configured to use SSL:



Select **Yes** if you trust the IIS Express SSL certificate.

The following dialog is displayed:



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

For information on trusting the Firefox browser, see Firefox SEC_ERROR_INADEQUATE_KEY_USAGE certificate error.

Visual Studio:

- Runs the app, which launches the Kestrel server.
- Launches the default browser at `https://localhost:<port>`, which displays the apps UI. `<port>` is the random port that is assigned when the app was created.

Close the browser window.

Examine the project files

The following sections contain 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 has HTML markup with C# code using Razor syntax.
- A `.cshtml.cs` file that has 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. `_Layout.cshtml` sets up the navigation menu at the top of the page and the copyright notice at the bottom of the page. For more information, see Layout in ASP.NET Core.

wwwroot folder

Contains static assets, like HTML files, JavaScript files, and CSS files. For more information, see Static files in ASP.NET Core.

`appsettings.json`

Contains configuration data, like connection strings. For more information, see Configuration in ASP.NET Core.

Program.cs

Contains the following code:

```
C#
```

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

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

var app = builder.Build();

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

app.UseHttpsRedirection();

app.UseRouting();

app.UseAuthorization();

app.MapStaticAssets();
app.MapRazorPages();

app.Run();
```

The following lines of code in this file create a `WebApplicationBuilder` with preconfigured defaults, add Razor Pages support to the Dependency Injection (DI) container, and builds the app:

```
C#
```

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

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

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

The developer exception page is enabled by default and provides helpful information on exceptions. Production apps should not be run in development mode because the developer exception page can leak sensitive information.

The following code sets the exception endpoint to `/Error` and enables HTTP Strict Transport Security Protocol (HSTS) when the app is ***not*** running in development mode:

```
C#
```

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

For example, the preceding code runs when the app is in production or test mode. For more information, see Use multiple environments in ASP.NET Core.

The following code enables various Middleware:

- `app.UseHttpsRedirection();` : Redirects HTTP requests to HTTPS.
- `app.UseRouting();` : Adds route matching to the middleware pipeline. For more information, see Routing in ASP.NET Core.
- `app.UseAuthorization();` : Authorizes a user to access secure resources. This app doesn't use authorization, therefore this line could be removed.
- `app.MapRazorPages();`: Configures endpoint routing for Razor Pages.
- `app.MapStaticAssets();` : Optimize the delivery of static assets in an app such as such as HTML, CSS, images, and JavaScript to be served. For more information, see What's new in ASP.NET Core 9.0.
- `app.Run();` : Runs the app.

Troubleshooting with the completed sample

If you run into a problem you can't resolve, compare your code to the completed project. View or download completed project (how to download).

Next steps

Next: Add a model

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

Article • 10/29/2024

> **ⓘ Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

In this tutorial, classes are added for managing movies in a database. The app's model classes use Entity Framework Core (EF Core) to work with the database. EF Core is an object-relational mapper (O/RM) that simplifies data access. You write the model classes first, and EF Core creates the database.

The model classes are known as POCO classes (from "**P**lain-**O**ld **C**LR **O**bjects") because they don't have a dependency on EF Core. They define the properties of the data that are stored in the database.

Add a data model

Visual Studio

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

C#

```
using System.ComponentModel.DataAnnotations;

namespace RazorPagesMovie.Models;

public class Movie
{
    public int Id { get; set; }
```

```
    public string? Title { get; set; }
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }
    public string? Genre { get; set; }
    public decimal Price { get; set; }
}
```

The `Movie` class contains:

- The `ID` field is required by the database for the primary key.
- A [DataType] attribute that specifies the type of data in the `ReleaseDate` property. With this attribute:
  - The user isn't required to enter time information in the date field.
  - Only the date is displayed, not time information.
- The question mark after `string` indicates that the property is nullable. For more information, see Nullable reference types.

DataAnnotations are covered in a later tutorial.

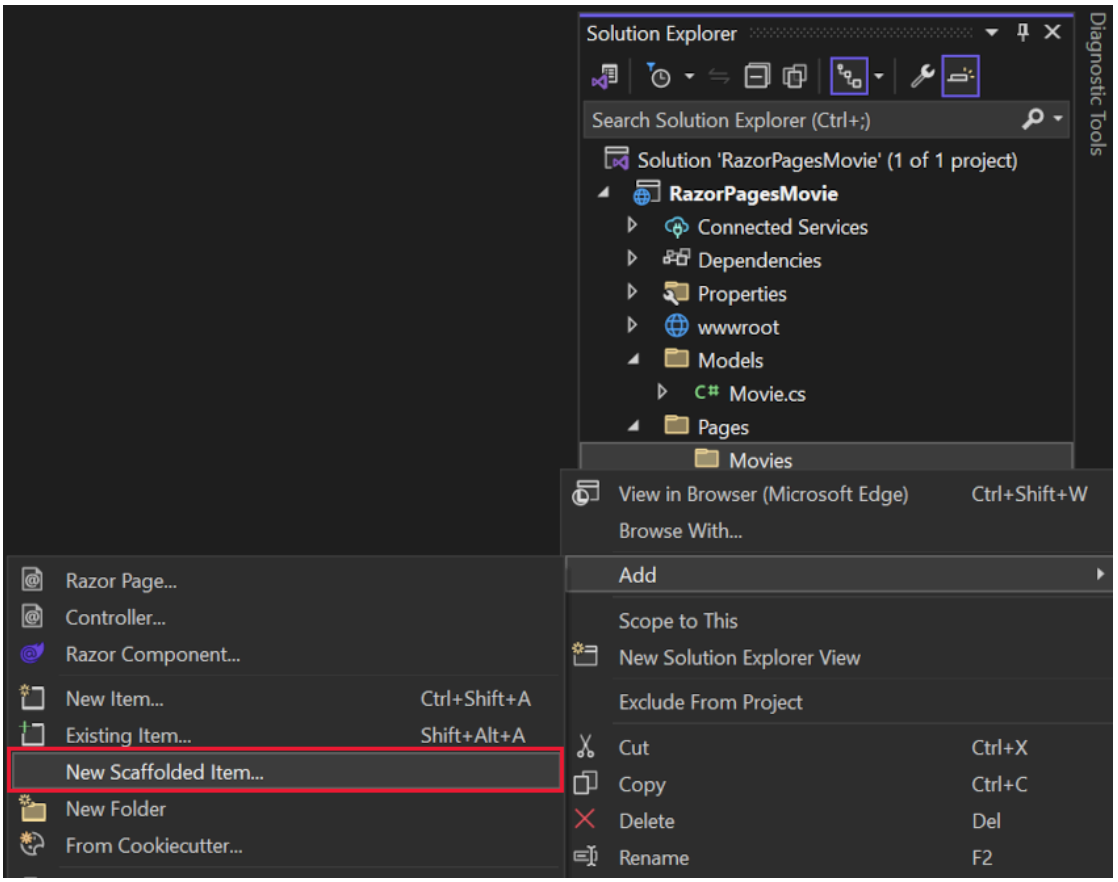
Build the project to verify there are no compilation errors.

Scaffold the movie model

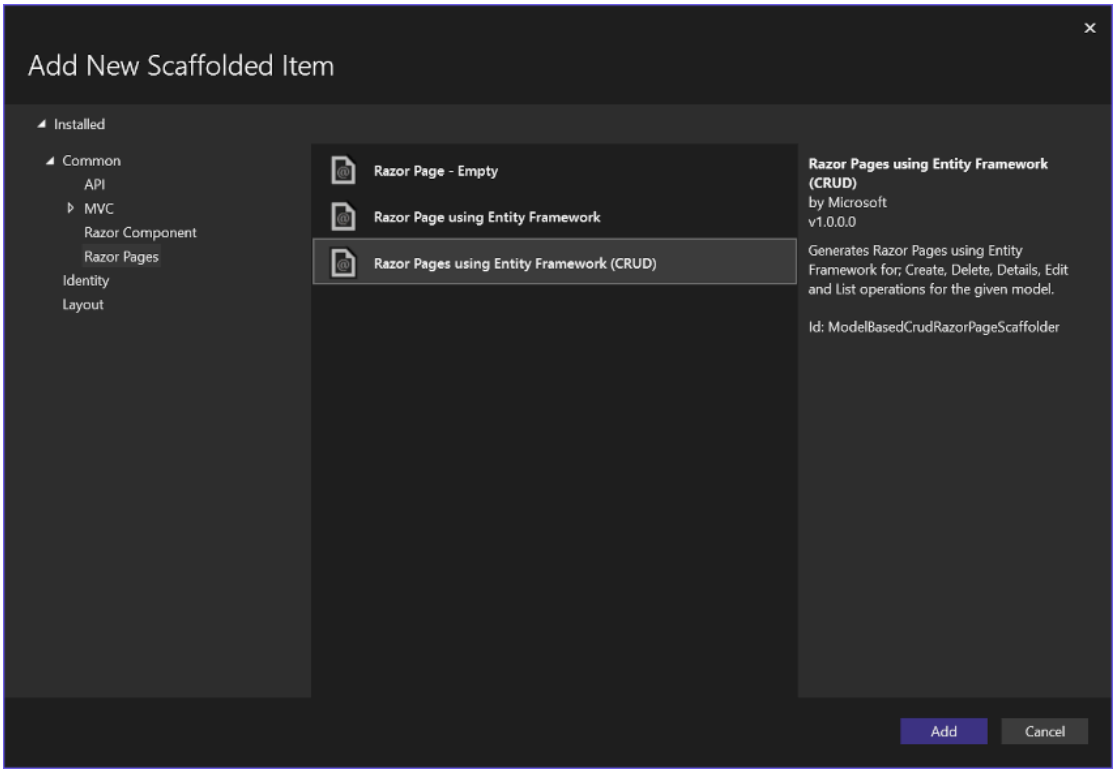
In this section, the movie model is scaffolded. That is, the scaffolding tool produces pages for Create, Read, Update, and Delete (CRUD) operations for the movie model.

Visual Studio

1. Create the *Pages/Movies* folder:
   a. Right-click on the *Pages* folder > **Add** > **New Folder**.
   b. Name the folder *Movies*.
2. Right-click on the *Pages/Movies* folder > **Add** > **New Scaffolded Item**.



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

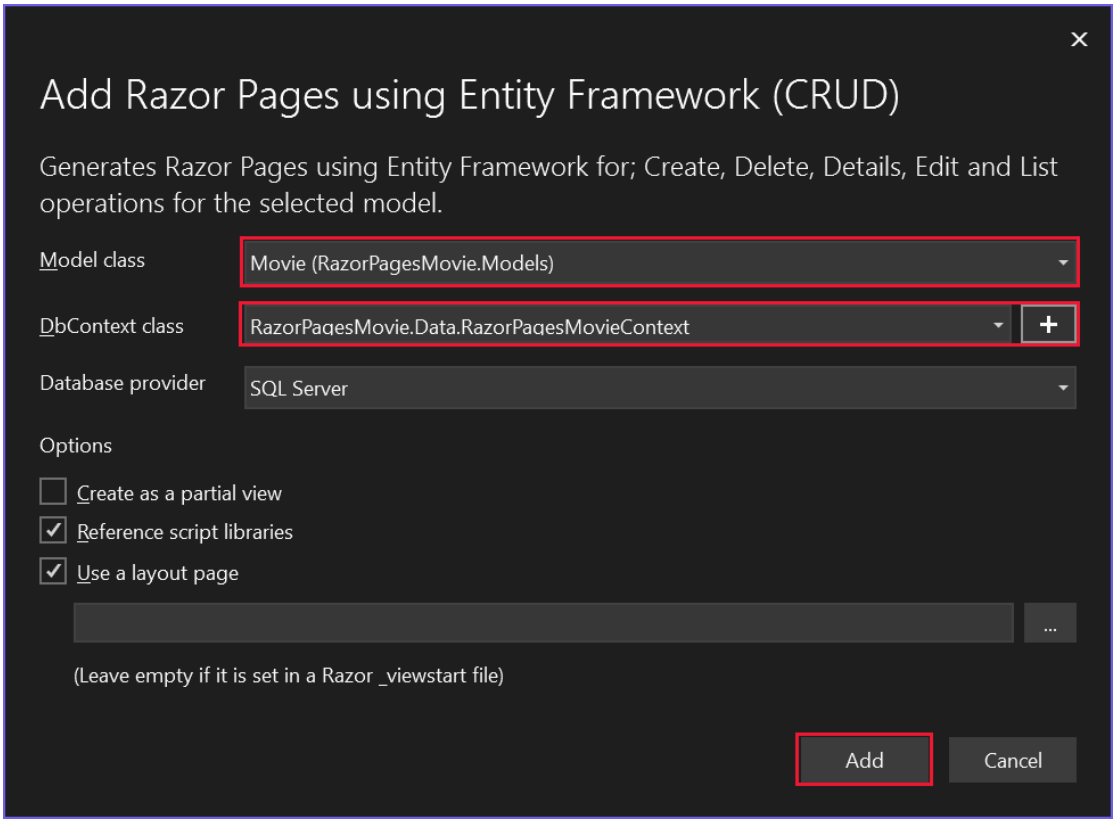


4. Complete the **Add Razor Pages using Entity Framework (CRUD)** dialog:
   a. In the **Model class** drop down, select **Movie (RazorPagesMovie.Models)**.
   b. In the **Data context class** row, select the + (plus) sign.

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

ii. In the **Database provider** drop down, select **SQL Server**.

c. Select **Add**.



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

> ⚠ **Warning**
>
> This article uses a local database that doesn't require the user to be authenticated. Production apps should use the most secure authentication flow available. For more information on authentication for deployed test and production apps, see Secure authentication flows.

Files created and updated

The scaffold process creates the following files:

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

The created files are explained in the next tutorial.

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

Visual Studio

C#

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

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddDbContext<RazorPagesMovieContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("RazorPag
esMovieContext") ?? throw new InvalidOperationException("Connection
string 'RazorPagesMovieContext' not found.")));

var app = builder.Build();

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

app.UseHttpsRedirection();

app.UseRouting();

app.UseAuthorization();

app.MapStaticAssets();
app.MapRazorPages();

app.Run();
```

The `Program.cs` changes are explained later in this tutorial.

Create the initial database schema using EF's migration feature

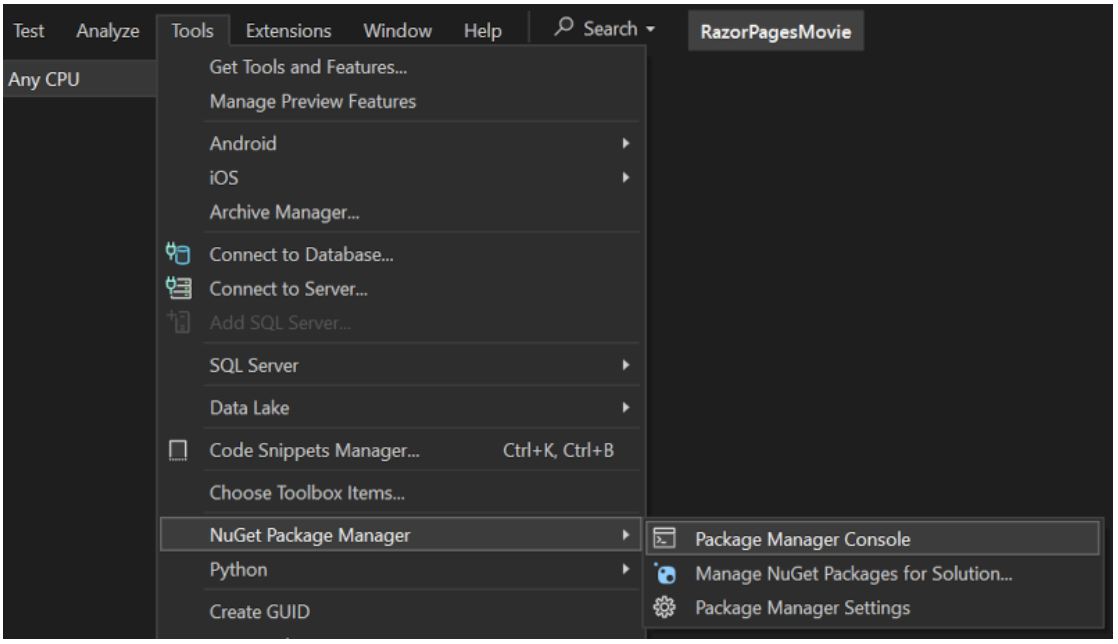
The migrations feature in Entity Framework Core provides a way to:

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

Visual Studio

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

- Add an initial migration.
- Update the database with the initial migration.
- From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console**.



- In the PMC, enter the following command:

```
PowerShell

Add-Migration InitialCreate
```

- The `Add-Migration` 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 migration. Any name can be used, but by convention a name is selected that describes the migration.

The following warning is displayed, which is addressed in a later step:

> 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()'.

- In the PMC, enter the following command:

```
PowerShell
```

```
Update-Database
```

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

The data context `RazorPagesMovieContext`:

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

The `RazorPagesMovieContext` class in the generated file `Data/RazorPagesMovieContext.cs`:

```
C#
```

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

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

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

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

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

Test the app

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

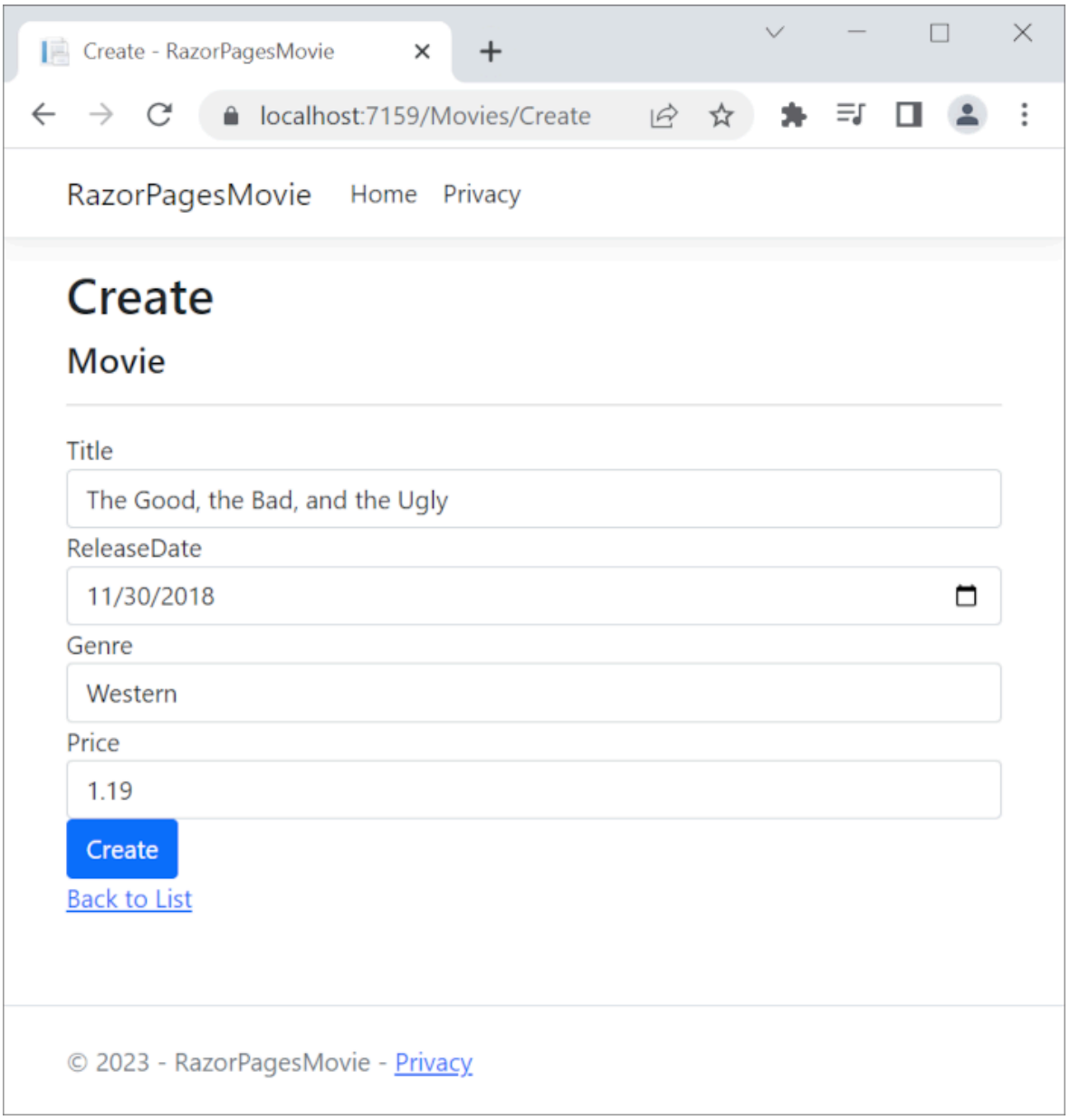
   If you receive the following error:

   Console

   ```
   SqlException: Cannot open database "RazorPagesMovieContext-GUID"
   requested by the login. The login failed.
   Login failed for user 'User-name'.
   ```

   You missed the migrations step.

2. Test the **Create New** link.



> **Note**
>
> You may not be able to enter decimal commas in the `Price` field. To support jQuery validation for non-English locales that use a comma (",") for a decimal point and for non US-English date formats, the app must be globalized. For globalization instructions, see this GitHub issue.

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

The next tutorial explains the files created by scaffolding.

Examine the context registered with dependency injection

ASP.NET Core is built with dependency injection. Services, such as the EF Core database context, are registered with dependency injection during application startup. Components that require these services (such as Razor Pages) are provided via constructor parameters. The constructor code that gets a database context instance is shown later in the tutorial.

The scaffolding tool automatically created a database context and registered it with the dependency injection container. The following highlighted code is added to the `Program.cs` file by the scaffolder:

Visual Studio

C#

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

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddDbContext<RazorPagesMovieContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("RazorPag
esMovieContext") ?? throw new InvalidOperationException("Connection
string 'RazorPagesMovieContext' not found.")));

var app = builder.Build();

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

app.UseHttpsRedirection();

app.UseRouting();

app.UseAuthorization();

app.MapStaticAssets();
app.MapRazorPages();

app.Run();
```

Troubleshooting with the completed sample

If you run into a problem you can't resolve, compare your code to the completed project. View or download completed project (how to download).

Next steps

Previous: Get Started Next: Scaffolded Razor Pages

Part 3, scaffolded Razor Pages in ASP.NET Core

Article • 07/01/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By Rick Anderson

This tutorial examines the Razor Pages created by scaffolding in the previous tutorial.

The Create, Delete, Details, and Edit pages

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

C#

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

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

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

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

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

Razor Pages are derived from PageModel. By convention, the `PageModel` derived class is named `PageNameModel`. For example, the Index page is named `IndexModel`.

The constructor uses dependency injection to add the `RazorPagesMovieContext` to the page:

C#

```
public class IndexModel : PageModel
{
    private readonly RazorPagesMovie.Data.RazorPagesMovieContext _context;

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

See Asynchronous code for more information on asynchronous programming with Entity Framework.

When a `GET` request is made for the page, the `OnGetAsync` method returns a list of movies to the Razor Page. On a Razor Page, `OnGetAsync` or `OnGet` is called to initialize the state of the page. In this case, `OnGetAsync` gets a list of movies and displays them.

When `OnGet` returns `void` or `OnGetAsync` returns `Task`, no return statement is used. For example, examine the Privacy Page:

C#

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace RazorPagesMovie.Pages
{
    public class PrivacyModel : PageModel
    {
        private readonly ILogger<PrivacyModel> _logger;
```

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

        public void OnGet()
        {
        }
    }

}
```

When the return type is IActionResult or `Task<IActionResult>`, a return statement must be provided. For example, the `Pages/Movies/Create.cshtml.cs OnPostAsync` method:

C#

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Movie.Add(Movie);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

Examine the `Pages/Movies/Index.cshtml` Razor Page:

CSHTML

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

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

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Title)
```

```
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model.Movie) {
        <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-page="./Edit" asp-route-id="@item.Id">Edit</a> |
                <a asp-page="./Details" asp-route-id="@item.Id">Details</a>
|
                <a asp-page="./Delete" asp-route-id="@item.Id">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

Razor can transition from HTML into C# or into Razor-specific markup. When an `@` symbol is followed by a Razor reserved keyword, it transitions into Razor-specific markup, otherwise it transitions into C#.

The @page directive

The `@page` Razor directive makes the file an MVC action, which means that it can handle requests. `@page` must be the first Razor directive on a page. `@page` and `@model` are examples of transitioning into Razor-specific markup. See Razor syntax for more information.

The @model directive

```cshtml
@page
@model RazorPagesMovie.Pages.Movies.IndexModel
```

The `@model` directive specifies the type of the model passed to the Razor Page. In the preceding example, the `@model` line makes the `PageModel` derived class available to the Razor Page. The model is used in the `@Html.DisplayNameFor` and `@Html.DisplayFor` HTML Helpers on the page.

Examine the lambda expression used in the following HTML Helper:

```cshtml
@Html.DisplayNameFor(model => model.Movie[0].Title)
```

The DisplayNameFor HTML Helper inspects the `Title` property referenced in the lambda expression to determine the display name. The lambda expression is inspected rather than evaluated. That means there is no access violation when `model`, `model.Movie`, or `model.Movie[0]` is `null` or empty. When the lambda expression is evaluated, for example, with `@Html.DisplayFor(modelItem => item.Title)`, the model's property values are evaluated.

The layout page

Select the menu links **RazorPagesMovie**, **Home**, and **Privacy**. Each page shows the same menu layout. The menu layout is implemented in the `Pages/Shared/_Layout.cshtml` file.

Open and examine the `Pages/Shared/_Layout.cshtml` file.

Layout templates allow the HTML container layout to be:

- Specified in one place.
- Applied in multiple pages in the site.

Find the `@RenderBody()` line. `RenderBody` is a placeholder where all the page-specific views show up, *wrapped* in the layout page. For example, select the **Privacy** link and the `Pages/Privacy.cshtml` view is rendered inside the `RenderBody` method.

ViewData and layout

Consider the following markup from the `Pages/Movies/Index.cshtml` file:

CSHTML

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

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

The preceding highlighted markup is an example of Razor transitioning into C#. The `{` and `}` characters enclose a block of C# code.

The `PageModel` base class contains a `ViewData` dictionary property that can be used to pass data to a View. Objects are added to the `ViewData` dictionary using a ***key value*** pattern. In the preceding sample, the `Title` property is added to the `ViewData` dictionary.

The `Title` property is used in the `Pages/Shared/_Layout.cshtml` file. The following markup shows the first few lines of the `_Layout.cshtml` file.

CSHTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - RazorPagesMovie</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" asp-append-version="true"
/>
    <link rel="stylesheet" href="~/RazorPagesMovie.styles.css" asp-append-
version="true" />
```

Update the layout

1. Change the `<title>` element in the `Pages/Shared/_Layout.cshtml` file to display **Movie** rather than **RazorPagesMovie**.

   CSHTML

   ```
   <!DOCTYPE html>
   <html lang="en">
   <head>
   ```

```
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-
scale=1.0" />
    <title>@ViewData["Title"] - Movie</title>
```

2. Find the following anchor element in the `Pages/Shared/_Layout.cshtml` file.

```
CSHTML

<a class="navbar-brand" asp-area="" asp-
page="/Index">RazorPagesMovie</a>
```

3. Replace the preceding element with the following markup:

```
CSHTML

<a class="navbar-brand" asp-page="/Movies/Index">RpMovie</a>
```

   The preceding anchor element is a Tag Helper. In this case, it's the Anchor Tag Helper. The `asp-page="/Movies/Index"` Tag Helper attribute and value creates a link to the `/Movies/Index` Razor Page. The `asp-area` attribute value is empty, so the area isn't used in the link. See Areas for more information.

4. Save the changes and test the app by selecting the **RpMovie** link. See the _Layout.cshtml file in GitHub if you have any problems.

5. Test the **Home**, **RpMovie**, **Create**, **Edit**, and **Delete** links. Each page sets the title, which you can see in the browser tab. When you bookmark a page, the title is used for the bookmark.

> **Note**
>
> You may not be able to enter decimal commas in the `Price` field. To support jQuery validation for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize the app. See this GitHub issue 4076 for instructions on adding decimal comma.

The `Layout` property is set in the `Pages/_ViewStart.cshtml` file:

```
CSHTML

@{
    Layout = "_Layout";
```

```
}
```

The preceding markup sets the layout file to `Pages/Shared/_Layout.cshtml` for all Razor files under the *Pages* folder. See Layout for more information.

The Create page model

Examine the `Pages/Movies/Create.cshtml.cs` page model:

C#

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

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

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

        public IActionResult OnGet()
        {
            return Page();
        }

        [BindProperty]
        public Movie Movie { get; set; } = default!;

        // To protect from overposting attacks, see
https://aka.ms/RazorPagesCRUD
        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }
```

```
            _context.Movie.Add(Movie);
            await _context.SaveChangesAsync();

            return RedirectToPage("./Index");
        }
    }
}
```

The `OnGet` method initializes any state needed for the page. The Create page doesn't have any state to initialize, so `Page` is returned. Later in the tutorial, an example of `OnGet` initializing state is shown. The `Page` method creates a `PageResult` object that renders the `Create.cshtml` page.

The `Movie` property uses the [BindProperty] attribute to opt-in to model binding. When the Create form posts the form values, the ASP.NET Core runtime binds the posted values to the `Movie` model.

The `OnPostAsync` method is run when the page posts form data:

C#

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Movie.Add(Movie);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

If there are any model errors, the form is redisplayed, along with any form data posted. Most model errors can be caught on the client-side before the form is posted. An example of a model error is posting a value for the date field that cannot be converted to a date. Client-side validation and model validation are discussed later in the tutorial.

If there are no model errors:

- The data is saved.
- The browser is redirected to the Index page.

The Create Razor Page

Examine the `Pages/Movies/Create.cshtml` Razor Page file:

CSHTML

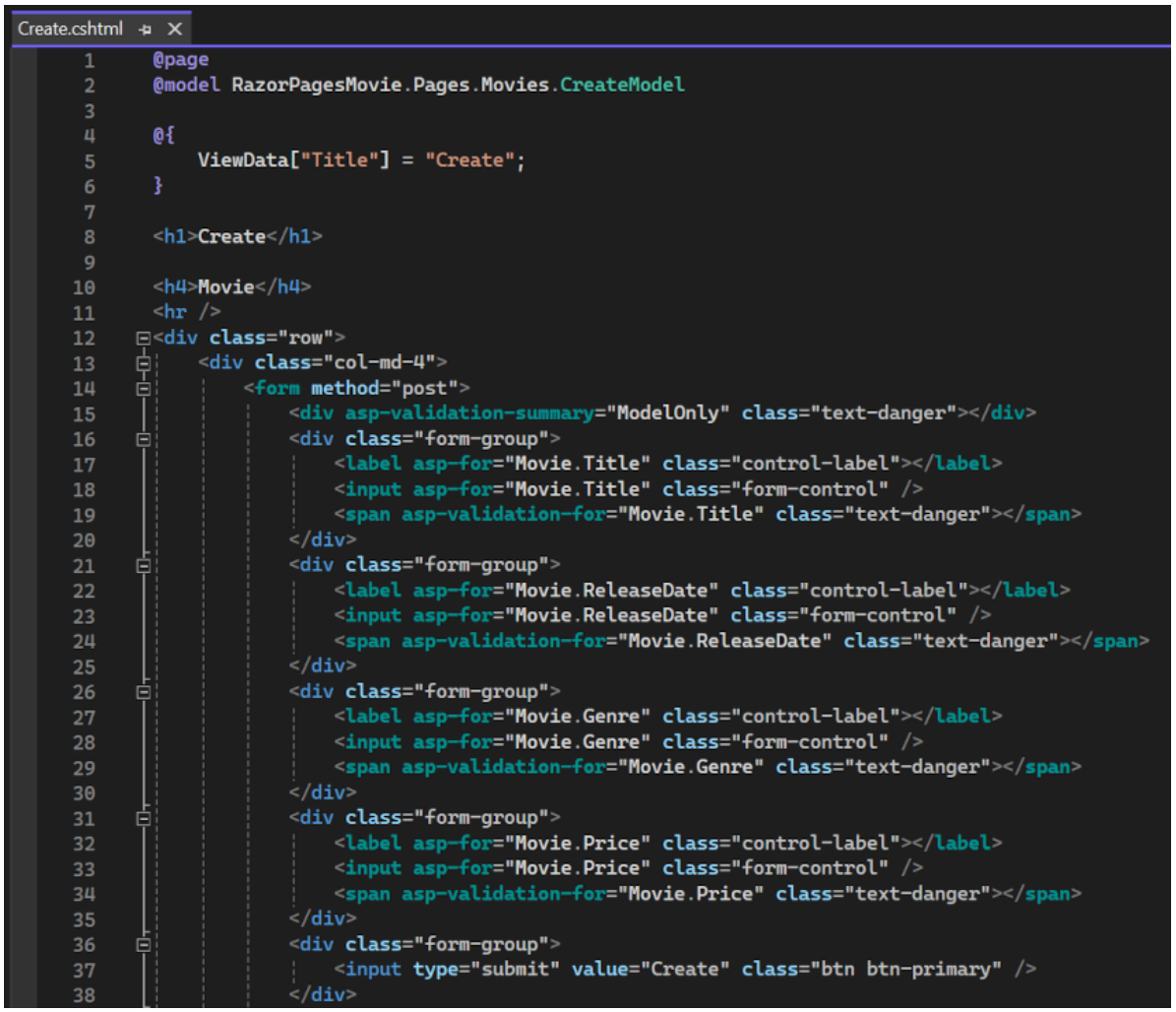
```
@page
@model RazorPagesMovie.Pages.Movies.CreateModel

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

<h1>Create</h1>

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

<div>
```

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

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

Visual Studio

Visual Studio displays the following tags in a distinctive bold font used for Tag Helpers:

- `<form method="post">`
- `<div asp-validation-summary="ModelOnly" class="text-danger"></div>`
- `<label asp-for="Movie.Title" class="control-label"></label>`
- `<input asp-for="Movie.Title" class="form-control" />`
- `<span asp-validation-for="Movie.Title" class="text-danger"></span>`



The `<form method="post">` element is a Form Tag Helper. The Form Tag Helper automatically includes an antiforgery token.

The scaffolding engine creates Razor markup for each field in the model, except the ID, similar to the following:

CSHTML

```
<div asp-validation-summary="ModelOnly" class="text-danger"></div>
<div class="form-group">
    <label asp-for="Movie.Title" class="control-label"></label>
    <input asp-for="Movie.Title" class="form-control" />
    <span asp-validation-for="Movie.Title" class="text-danger"></span>
</div>
```

The Validation Tag Helpers (`<div asp-validation-summary` and `<span asp-validation-for`) display validation errors. Validation is covered in more detail later in this series.

The Label Tag Helper (`<label asp-for="Movie.Title" class="control-label"></label>`) generates the label caption and `[for]` attribute for the `Title` property.

The Input Tag Helper (`<input asp-for="Movie.Title" class="form-control">`) uses the DataAnnotations attributes and produces HTML attributes needed for jQuery Validation on the client-side.

For more information on Tag Helpers such as `<form method="post">`, see Tag Helpers in ASP.NET Core.

Next steps

Previous: Add a model | Next: Work with a database

Part 4 of tutorial series on Razor Pages

Article • 09/17/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By Joe Audette

The `RazorPagesMovieContext` object handles the task of connecting to the database and mapping `Movie` objects to database records. The database context is registered with the Dependency Injection container in `Program.cs`:

Visual Studio

C#

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

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddDbContext<RazorPagesMovieContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("RazorPag
esMovieContext") ?? throw new InvalidOperationException("Connection
string 'RazorPagesMovieContext' not found.")));
```

The ASP.NET Core Configuration system reads the `ConnectionString` key. For local development, configuration gets the connection string from the `appsettings.json` file.

Visual Studio

The generated connection string is similar to the following JSON:

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "RazorPagesMovieContext": "Server=
(localdb)\\mssqllocaldb;Database=RazorPagesMovieContext-f2e0482c-952d-
4b1c-afe9-
a1a3dfe52e55;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

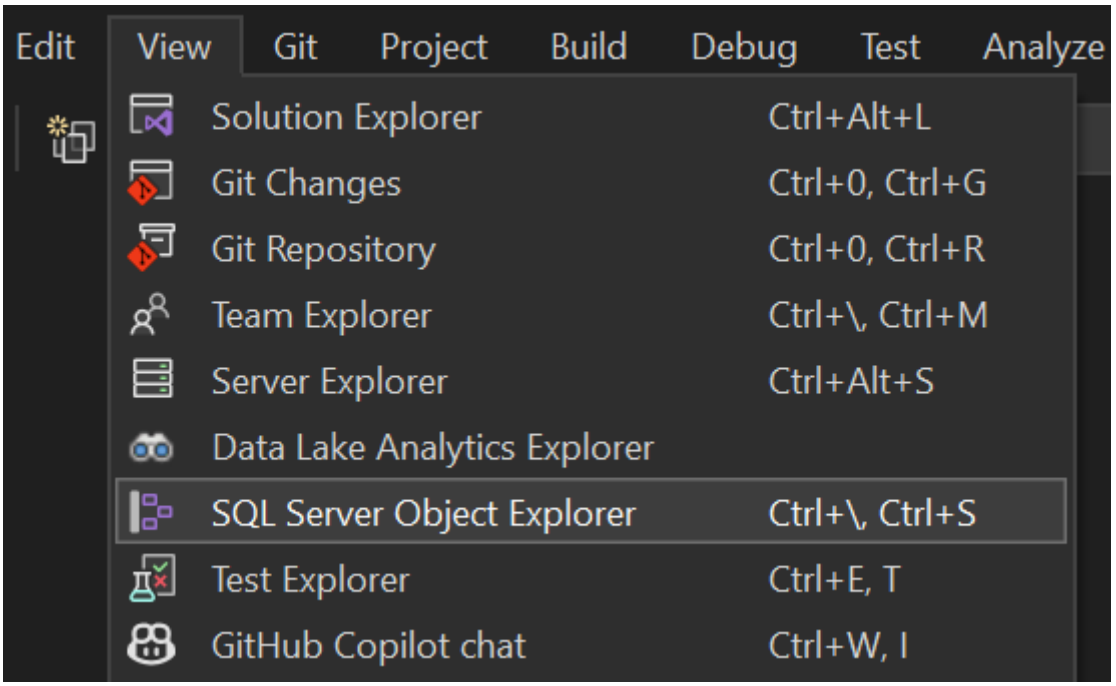
> ⚠ **Warning**
>
> This article uses a local database that doesn't require the user to be authenticated. Production apps should use the most secure authentication flow available. For more information on authentication for deployed test and production apps, see Secure authentication flows.

Visual Studio

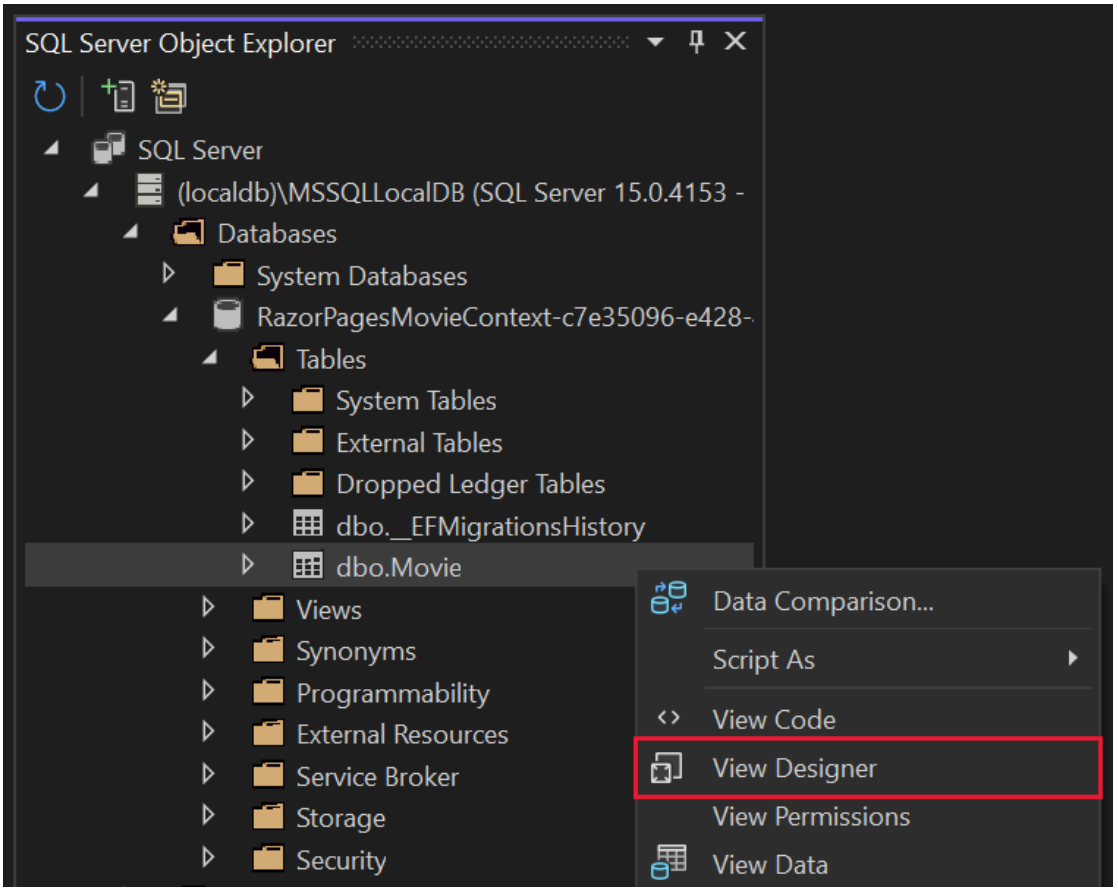
SQL Server Express LocalDB

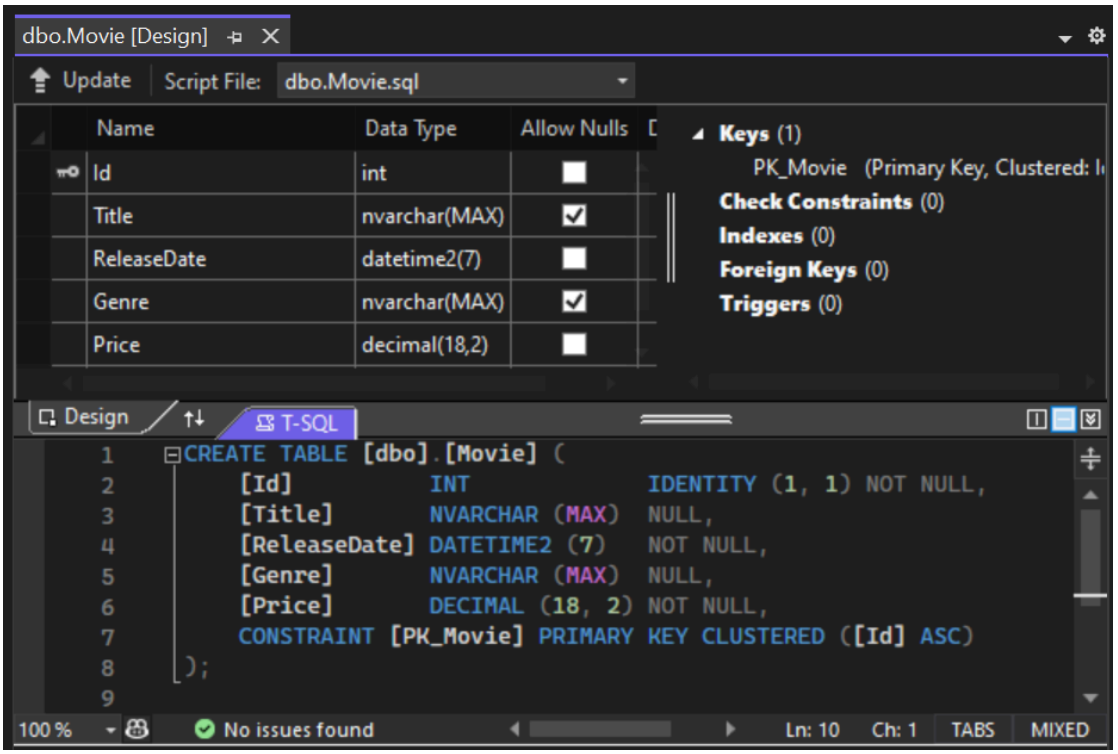
LocalDB is a lightweight version of the SQL Server Express database engine that's targeted for program development. LocalDB starts on demand and runs in user mode, so there's no complex configuration. By default, LocalDB database creates `*.mdf` files in the `C:\Users\<user>\` directory.

1. From the **View** menu, open **SQL Server Object Explorer** (SSOX).



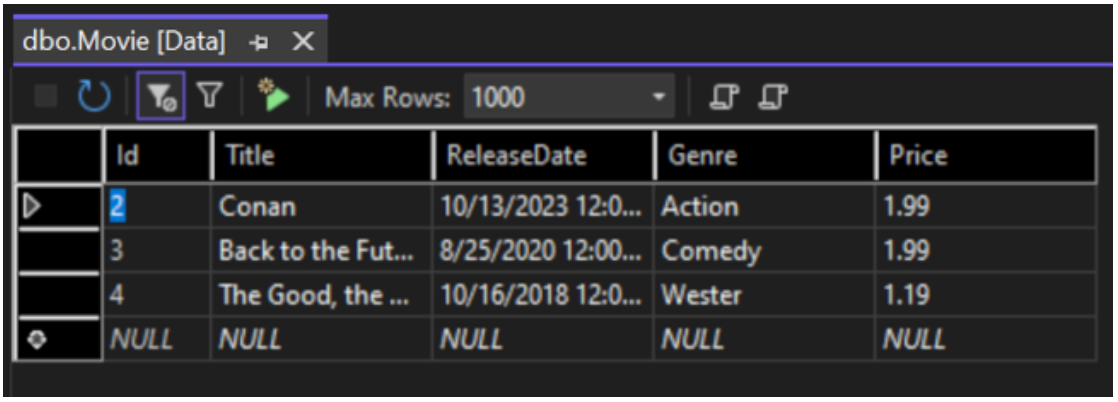
2. Right-click on the `Movie` table and select **View Designer**:





Note the key icon next to `ID`. By default, EF creates a property named `ID` for the primary key.

3. Right-click on the `Movie` table and select **View Data**:



Seed the database

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

C#

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

namespace RazorPagesMovie.Models;

public static class SeedData
```

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

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

            context.Movie.AddRange(
                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.

C#

```
if (context.Movie.Any())
{
    return;
}
```

Add the seed initializer

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

Visual Studio

C#

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

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddDbContext<RazorPagesMovieContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("RazorPag
esMovieContext") ?? throw new InvalidOperationException("Connection
string 'RazorPagesMovieContext' not found.")));

var app = builder.Build();

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

    SeedData.Initialize(services);
}

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}
```

```
app.UseHttpsRedirection();

app.UseRouting();

app.UseAuthorization();

app.MapStaticAssets();
app.MapRazorPages();

app.Run();
```

In the previous code, `Program.cs` has been modified to do the following:

- Get a database context instance from the dependency injection (DI) container.
- Call the `seedData.Initialize` method, passing to it the database context instance.
- Dispose the context when the seed method completes. The using statement ensures the context is disposed.

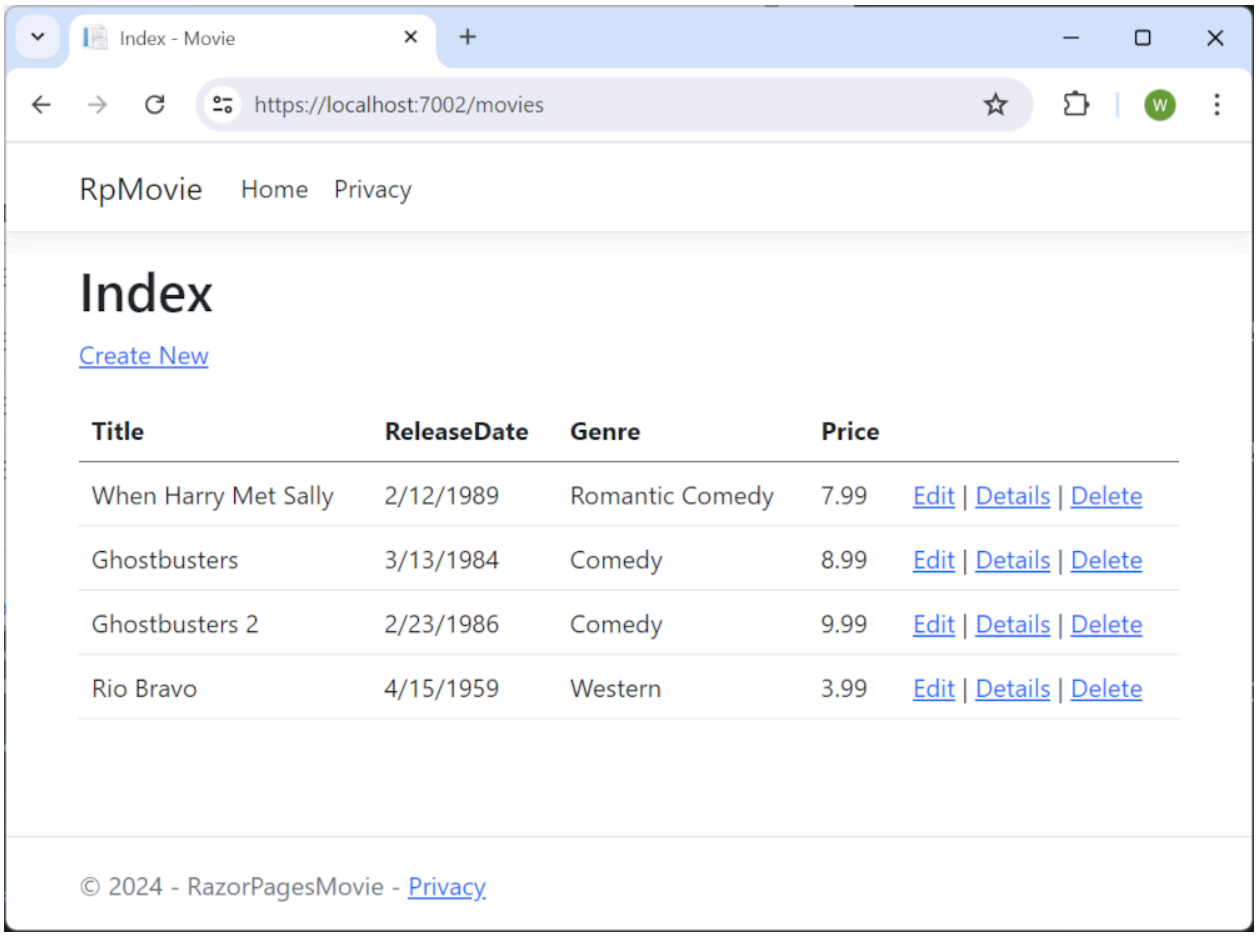
The following exception occurs when `Update-Database` has not been run:

> `SqlException: Cannot open database "RazorPagesMovieContext-" requested by the login. The login failed.` `Login failed for user 'user name'.`

Test the app

Delete all the records in the database so the seed method will run. Stop and start the app to seed the database. If the database isn't seeded, put a breakpoint on `if (context.Movie.Any())` and step through the code.

The app shows the seeded data:



Next steps

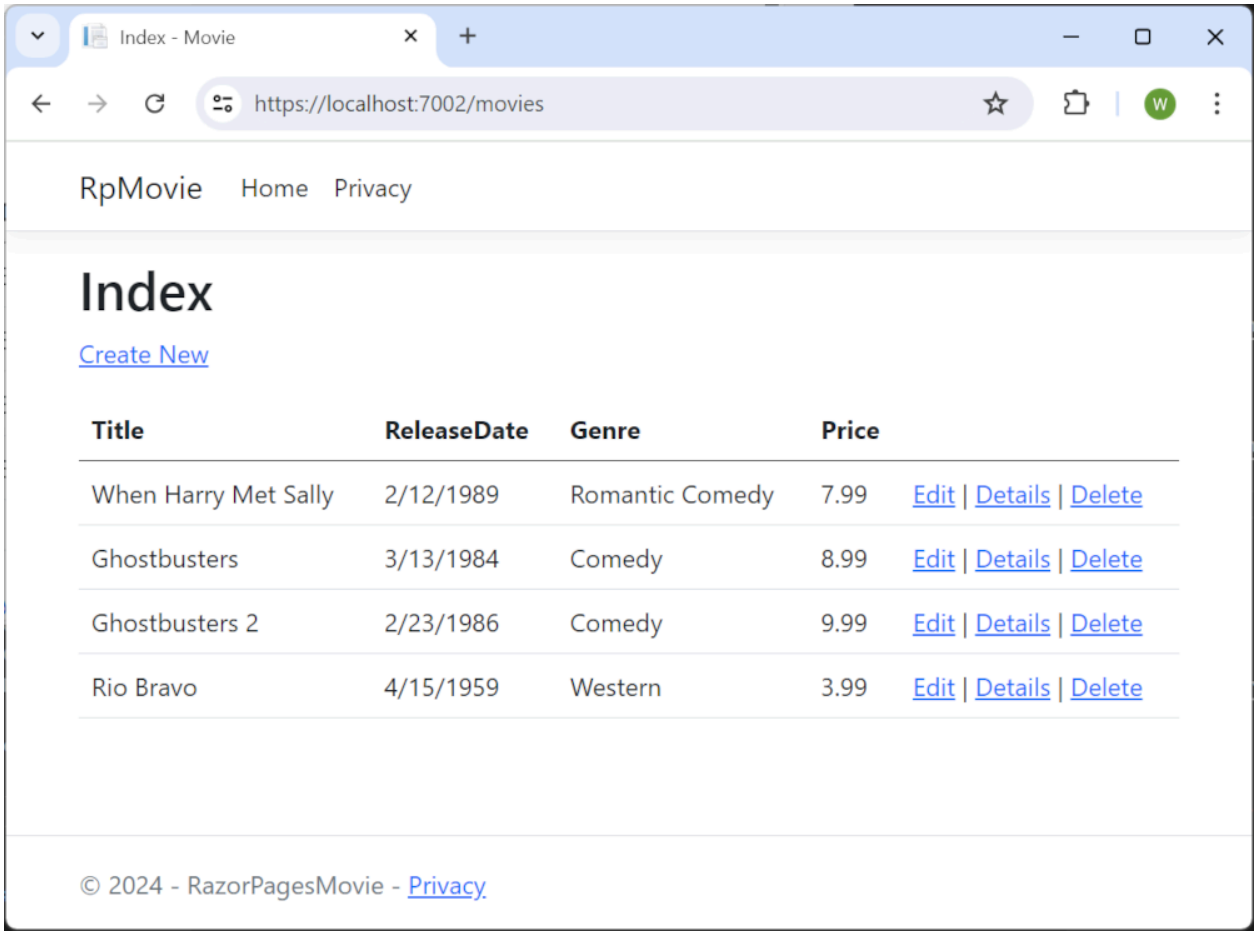
Previous: Scaffolded Razor Pages
Next: Update the pages

Part 5, update the generated pages in an ASP.NET Core app

Article • 07/01/2024

> **ⓘ Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

The scaffolded movie app has a good start, but the presentation isn't ideal. **ReleaseDate** should be two words, **Release Date**.



Update the model

Update `Models/Movie.cs` with the following highlighted code:

C#

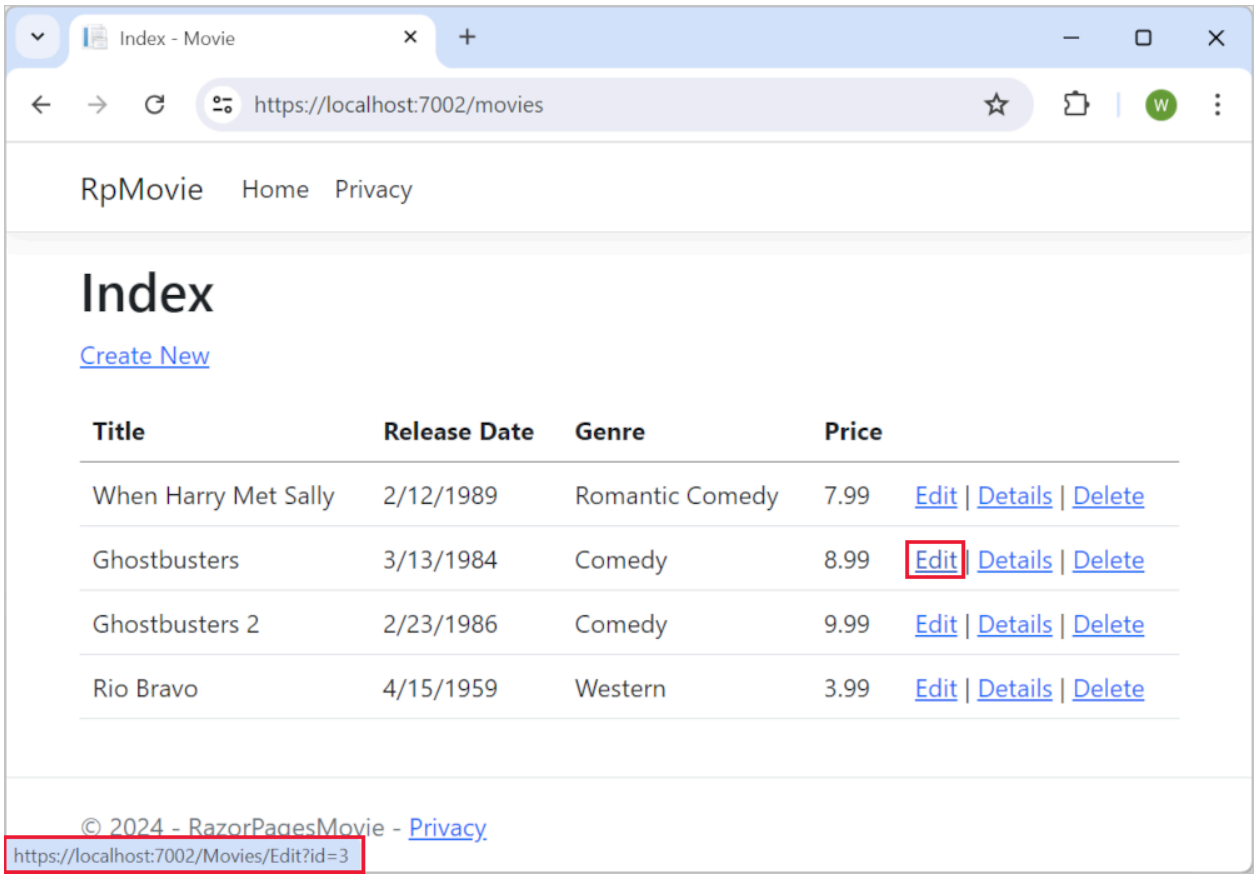
```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace RazorPagesMovie.Models;

public class Movie
{
    public int Id { get; set; }
    public string Title { get; set; } = string.Empty;

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

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

In the previous code:

- The `[Column(TypeName = "decimal(18, 2)")]` data annotation enables Entity Framework Core to correctly map `Price` to currency in the database. For more information, see Data Types.
- The [Display] attribute specifies the display name of a field. In the preceding code, `Release Date` instead of `ReleaseDate`.
- The [DataType] attribute specifies the type of the data (`Date`). The time information stored in the field isn't displayed.

DataAnnotations is covered in the next tutorial.

Browse to *Pages/Movies* and hover over an **Edit** link to see the target URL.



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

CSHTML

```
@foreach (var item in Model.Movie) {
        <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-page="./Edit" asp-route-id="@item.Id">Edit</a> |
                <a asp-page="./Details" asp-route-id="@item.Id">Details</a>
|
                <a asp-page="./Delete" asp-route-id="@item.Id">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files.

In the preceding code, the Anchor Tag Helper dynamically generates the HTML `href` attribute value from the Razor Page (the route is relative), the `asp-page`, and the route identifier (`asp-route-id`). For more information, see URL generation for Pages.

Use **View Source** from a browser to examine the generated markup. A portion of the generated HTML is shown below:

HTML

```
<td>
  <a href="/Movies/Edit?id=1">Edit</a> |
  <a href="/Movies/Details?id=1">Details</a> |
  <a href="/Movies/Delete?id=1">Delete</a>
</td>
```

The dynamically generated links pass the movie ID with a query string. For example, the `?id=1` in `https://localhost:5001/Movies/Details?id=1`.

Add route template

Update the Edit, Details, and Delete Razor Pages to use the `{id:int}` route template. Change the page directive for each of these pages from `@page` to `@page "{id:int}"`. Run the app and then view source.

The generated HTML adds the ID to the path portion of the URL:

HTML

```
<td>
  <a href="/Movies/Edit/1">Edit</a> |
  <a href="/Movies/Details/1">Details</a> |
  <a href="/Movies/Delete/1">Delete</a>
</td>
```

A request to the page with the `{id:int}` route template that does **not** include the integer returns an HTTP 404 (not found) error. For example, `https://localhost:5001/Movies/Details` returns a 404 error. To make the ID optional, append `?` to the route constraint:

CSHTML

```
@page "{id:int?}"
```

Test the behavior of `@page "{id:int?}"`:

1. Set the page directive in `Pages/Movies/Details.cshtml` to `@page "{id:int?}"`.
2. Set a break point in `public async Task<IActionResult> OnGetAsync(int? id)`, in `Pages/Movies/Details.cshtml.cs`.
3. Navigate to `https://localhost:5001/Movies/Details/`.

With the `@page "{id:int}"` directive, the break point is never hit. The routing engine returns HTTP 404. Using `@page "{id:int?}"`, the `OnGetAsync` method returns `NotFound` (HTTP 404):

C#

```
public async Task<IActionResult> OnGetAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

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

Review concurrency exception handling

Review the `OnPostAsync` method in the `Pages/Movies/Edit.cshtml.cs` file:

C#

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }
```

```
    _context.Attach(Movie).State = EntityState.Modified;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!MovieExists(Movie.Id))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return RedirectToPage("./Index");
}

private bool MovieExists(int id)
{
    return _context.Movie.Any(e => e.Id == id);
}
```

The previous code detects concurrency exceptions when one client deletes the movie and the other client posts changes to the movie.

To test the `catch` block:

1. Set a breakpoint on `catch (DbUpdateConcurrencyException)`.
2. Select **Edit** for a movie, make changes, but don't enter **Save**.
3. In another browser window, select the **Delete** link for the same movie, and then delete the movie.
4. In the previous browser window, post changes to the movie.

Production code may want to detect concurrency conflicts. See Handle concurrency conflicts for more information.

Posting and binding review

Examine the `Pages/Movies/Edit.cshtml.cs` file:

C#

```
public class EditModel : PageModel
{
```

```
    private readonly RazorPagesMovie.Data.RazorPagesMovieContext _context;

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

    [BindProperty]
    public Movie Movie { get; set; } = default!;

    public async Task<IActionResult> OnGetAsync(int? id)
    {
        if (id == null)
        {
            return NotFound();
        }

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

    // To protect from overposting attacks, enable the specific properties
you want to bind to.
    // For more details, see https://aka.ms/RazorPagesCRUD.
    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _context.Attach(Movie).State = EntityState.Modified;

        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(Movie.Id))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
```

```
        return RedirectToPage("./Index");
    }

    private bool MovieExists(int id)
    {
        return _context.Movie.Any(e => e.Id == id);
    }
```

When an HTTP GET request is made to the Movies/Edit page, for example, `https://localhost:5001/Movies/Edit/3`:

- The `OnGetAsync` method fetches the movie from the database and returns the `Page` method.
- The `Page` method renders the `Pages/Movies/Edit.cshtml` Razor Page. The `Pages/Movies/Edit.cshtml` file contains the model directive `@model RazorPagesMovie.Pages.Movies.EditModel`, which makes the movie model available on the page.
- The Edit form is displayed with the values from the movie.

When the Movies/Edit page is posted:

- The form values on the page are bound to the `Movie` property. The `[BindProperty]` attribute enables Model binding.

  C#

  ```
  [BindProperty]
  public Movie Movie { get; set; }
  ```

- If there are errors in the model state, for example, `ReleaseDate` cannot be converted to a date, the form is redisplayed with the submitted values.
- If there are no model errors, the movie is saved.

The HTTP GET methods in the Index, Create, and Delete Razor pages follow a similar pattern. The HTTP POST `OnPostAsync` method in the Create Razor Page follows a similar pattern to the `OnPostAsync` method in the Edit Razor Page.

Next steps

Previous: Work with a database | Next: Add search

Part 6, add search to ASP.NET Core Razor Pages

Article • 07/01/2024

> **ⓘ Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By Rick Anderson

In the following sections, searching movies by *genre* or *name* is added.

Add the following highlighted code to `Pages/Movies/Index.cshtml.cs`:

C#

```
public class IndexModel : PageModel
{
    private readonly RazorPagesMovie.Data.RazorPagesMovieContext _context;

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

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

    [BindProperty(SupportsGet = true)]
    public string? SearchString { get; set; }

    public SelectList? Genres { get; set; }

    [BindProperty(SupportsGet = true)]
    public string? MovieGenre { get; set; }
```

In the previous code:

- `SearchString`: Contains the text users enter in the search text box. `SearchString` has the [BindProperty] attribute. `[BindProperty]` binds form values and query

strings with the same name as the property. `[BindProperty(SupportsGet = true)]` is required for binding on HTTP GET requests.

- `Genres`: Contains the list of genres. `Genres` allows the user to select a genre from the list. `SelectList` requires `using Microsoft.AspNetCore.Mvc.Rendering;`
- `MovieGenre`: Contains the specific genre the user selects. For example, "Western".
- `Genres` and `MovieGenre` are used later in this tutorial.

> ⚠ **Warning**
>
> For security reasons, you must opt in to binding `GET` request data to page model properties. Verify user input before mapping it to properties. Opting into `GET` binding is useful when addressing scenarios that rely on query string or route values.
>
> To bind a property on `GET` requests, set the [BindProperty] attribute's `SupportsGet` property to `true`:
>
> C#
>
> ```
> [BindProperty(SupportsGet = true)]
> ```
>
> For more information, see ASP.NET Core Community Standup: Bind on GET discussion (YouTube).

Update the `Movies/Index` page's `OnGetAsync` method with the following code:

C#

```
public async Task OnGetAsync()
{
    var movies = from m in _context.Movie
                 select m;
    if (!string.IsNullOrEmpty(SearchString))
    {
        movies = movies.Where(s => s.Title.Contains(SearchString));
    }

    Movie = await movies.ToListAsync();
}
```

The first line of the `OnGetAsync` method creates a LINQ query to select the movies:

C#

```
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` property is not `null` or empty, the movies query is modified to filter on the search string:

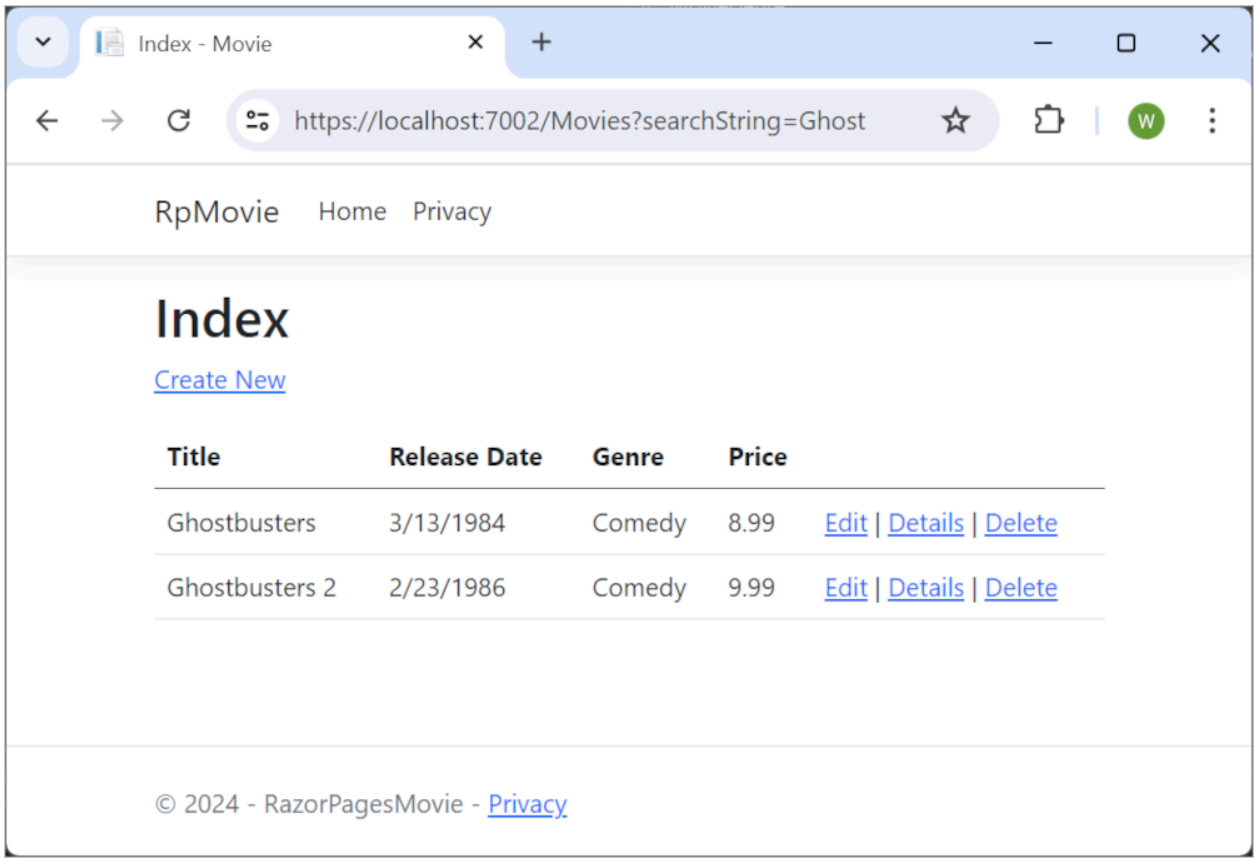
C#

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

The `s => s.Title.Contains()` code 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`. 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. The evaluation of an expression is delayed until its realized value is iterated over or the `ToListAsync` method is called. See Query Execution for more information.

> ⓘ Note
>
> The Contains method is run on the database, not in the C# code. 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. SQLite with the default collation is a mixture of case sensitive and case ***IN***sensitive, depending on the query. For information on making case insensitive SQLite queries, see the following:
>
> - How to use case-insensitive query with Sqlite provider? (dotnet/efcore #11414)
> - How to make a SQLite column case insensitive (dotnet/AspNetCore.Docs #22314)
> - Collations and Case Sensitivity

Navigate to the Movies page and append a query string such as `?searchString=Ghost` to the URL. For example, `https://localhost:5001/Movies?searchString=Ghost`. The filtered movies are displayed.

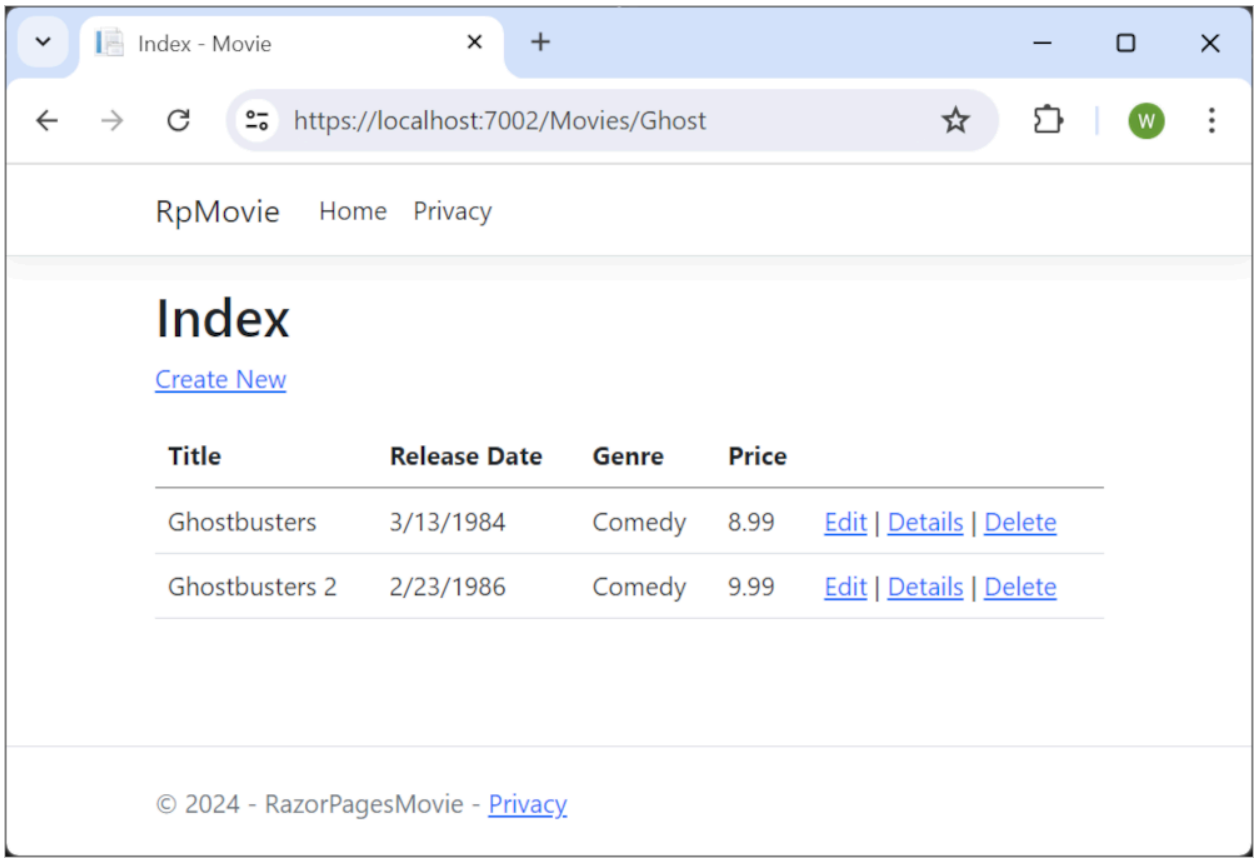


If the following route template is added to the Index page, the search string can be passed as a URL segment. For example, `https://localhost:5001/Movies/Ghost`.

CSHTML

```
@page "{searchString?}"
```

The preceding route constraint allows searching the title as route data (a URL segment) instead of as a query string value. The `?` in `"{searchString?}"` means this is an optional route parameter.



The ASP.NET Core runtime uses model binding to set the value of the `SearchString` property from the query string (`?searchString=Ghost`) or route data (`https://localhost:5001/Movies/Ghost`). Model binding is ***not*** case sensitive.

However, users cannot be expected to modify the URL to search for a movie. In this step, UI is added to filter movies. If you added the route constraint `"{searchString?}"`, remove it.

Open the `Pages/Movies/Index.cshtml` file, and add the markup highlighted in the following code:

CSHTML

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

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

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>
```

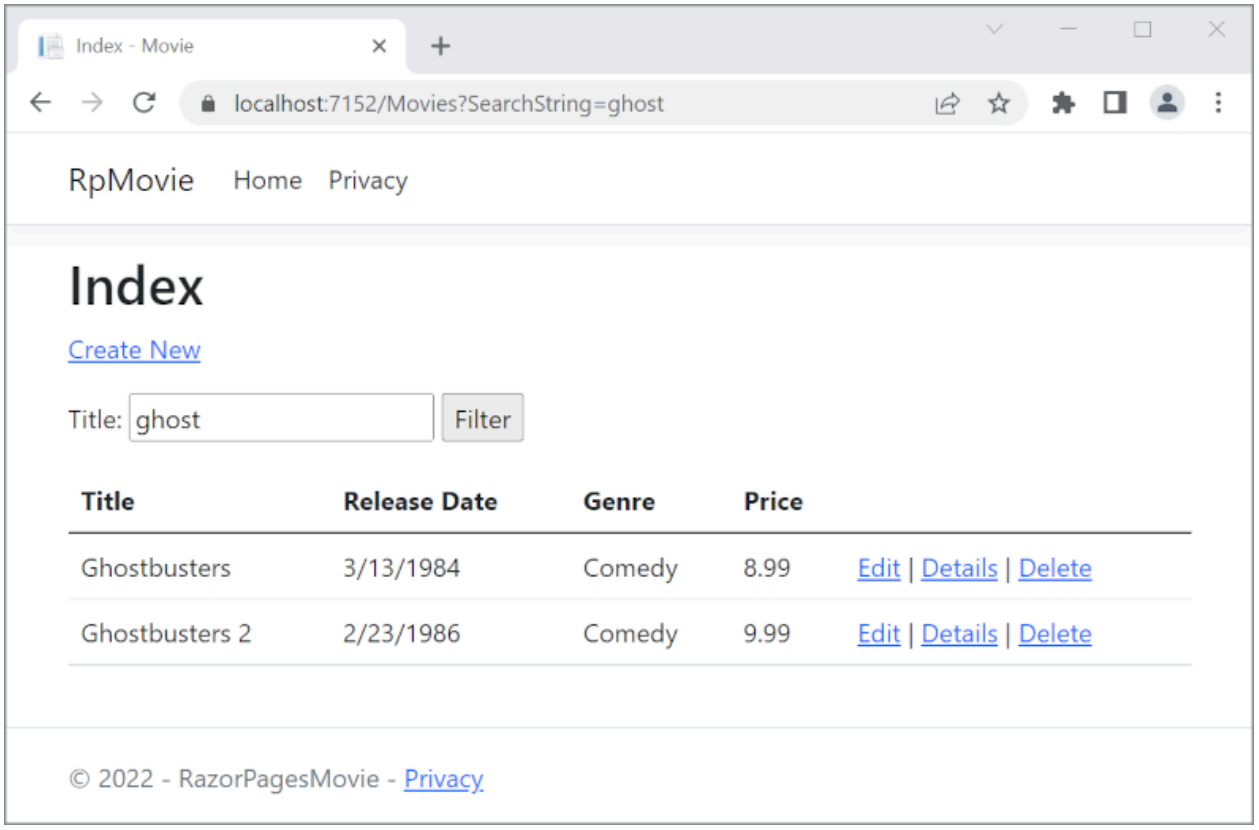
```
<form>
    <p>
        <label>Title: <input type="text" asp-for="SearchString" /></label>
        <input type="submit" value="Filter" />
    </p>
</form>
```

```
<table class="table">
    <thead>
```

The HTML `<form>` tag uses the following Tag Helpers:

- Form Tag Helper. When the form is submitted, the filter string is sent to the *Pages/Movies/Index* page via query string.
- Input Tag Helper

Save the changes and test the filter.



Search by genre

Update the `Movies/Index.cshtml.cs` page `OnGetAsync` method with the following code:

C#

```
public async Task OnGetAsync()
{
    // <snippet_search_linqQuery>
```

```
    IQueryable<string> genreQuery = from m in _context.Movie
                                    orderby m.Genre
                                    select m.Genre;
    // </snippet_search_linqQuery>

    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);
    }

    // <snippet_search_selectList>
    Genres = new SelectList(await genreQuery.Distinct().ToListAsync());
    // </snippet_search_selectList>
    Movie = await movies.ToListAsync();
}
```

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

C#

```
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:

C#

```
Genres = new SelectList(await genreQuery.Distinct().ToListAsync());
```

Add search by genre to the Razor Page

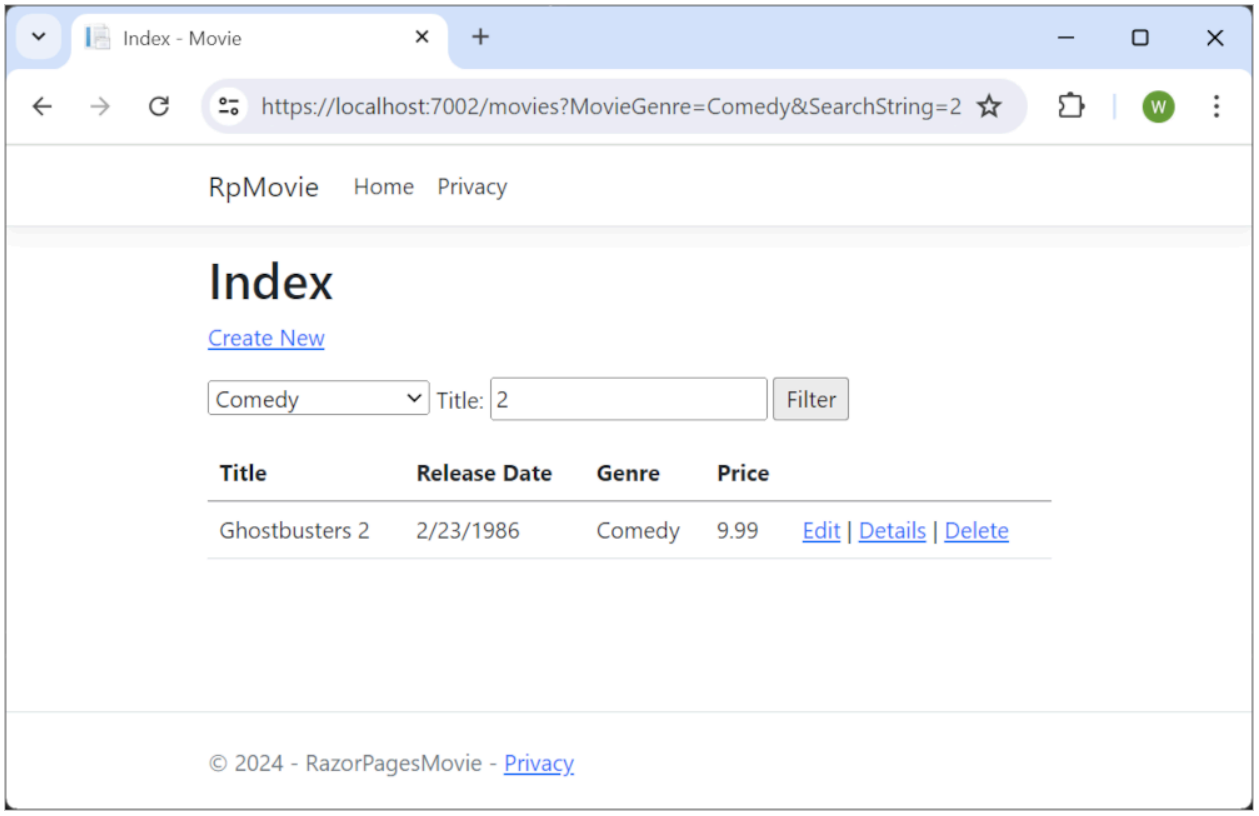
Update the `Index.cshtml` <form> element as highlighted in the following markup:

CSHTML

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

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

```
}

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form>
    <p>
        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>
        <label>Title: <input type="text" asp-for="SearchString" /></label>
        <input type="submit" value="Filter" />
    </p>
</form>
```

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



Next steps

Previous: Update the pages
Next: Add a new field

Part 7, add a new field to a Razor Page in ASP.NET Core

Article • 07/01/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

By Rick Anderson

In this section Entity Framework Core (EF Core) is used to define the database schema based on the app's model class:

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

The EF Core approach allows for a more agile development process. The developer works on the app's data model directly while the database schema is created and then syncronized, all without the developer having to switch contexts to and from a datbase management tool. For an overview of Entity Framework Core and its benefits, see Entity Framework Core.

Using EF Code to automatically create and track a database:

- Adds an __EFMigrationsHistory table to the database to track whether the schema of the database is in sync with the model classes it was generated from.
- Throws an exception if the model classes aren't in sync with the database.

Automatic verification that the schema and model are in sync makes it easier to find inconsistent database code issues.

Adding a Rating Property to the Movie Model

1. Open the `Models/Movie.cs` file and add a `Rating` property:

```
C#
```

```
public class Movie
{
    public int Id { get; set; }
    public string Title { get; set; } = string.Empty;

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

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

2. Edit `Pages/Movies/Index.cshtml`, and add a `Rating` field:

CSHTML

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

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

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form>
    <p>
        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>
        <label>Title: <input type="text" asp-for="SearchString" />
</label>
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">

    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model =>
```

```
model.Movie[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Price)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Rating)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Movie)
        {
            <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-page="./Edit" asp-route-
id="@item.Id">Edit</a> |
                    <a asp-page="./Details" asp-route-
id="@item.Id">Details</a> |
                    <a asp-page="./Delete" asp-route-
id="@item.Id">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

3. Update the following pages with a `Rating` field:

- *Pages/Movies/Create.cshtml*.
- *Pages/Movies/Delete.cshtml*.
- *Pages/Movies/Details.cshtml*.
- *Pages/Movies/Edit.cshtml*.

The app won't work until the database is updated to include the new field. Running the app without an update to the database throws a `SqlException`:

```
SqlException: Invalid column name 'Rating'.
```

The `SqlException` exception is caused by the updated Movie model class being different than the schema of the Movie table of the database. There's no `Rating` column in the database table.

There are a few approaches to resolving the error:

1. Have the Entity Framework automatically drop and re-create the database using the new model class schema. This approach is convenient early in the development cycle, it allows developers to quickly evolve the model and database schema together. The downside is that existing data in the database is lost. Don't use this approach on a production database! Dropping the database on schema changes and using an initializer to automatically seed the database with test data is often a productive way to develop an app.
2. Explicitly modify the schema of the existing database so that it matches the model classes. The advantage of this approach is to keep the data. Make this change either manually or by creating a database change script.
3. Use EF Core Migrations to update the database schema.

For this tutorial, use EF Core Migrations.

Update the `SeedData` class so that it provides a value for the new column. A sample change is shown below, but make this change for each `new Movie` block.

C#

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

See the completed SeedData.cs file.

Build the app

Visual Studio

Press `Ctrl`+`Shift`+`B`

Visual Studio

Add a migration for the rating field

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

2. In the Package Manager Console (PMC), enter the following command:

```
PowerShell

Add-Migration Rating
```

The `Add-Migration` command tells the framework to:

- Compare the `Movie` model with the `Movie` database schema.
- Create code to migrate the database schema 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.

2. In the PMC, enter the following command:

```
PowerShell

Update-Database
```

The `Update-Database` command tells the framework to apply the schema changes to the database and to preserve existing data.

Delete all the records in the database, the initializer will seed the database and include the `Rating` field. Deleting can be done with the delete links in the browser or from Sql Server Object Explorer (SSOX).

Another option is to delete the database and use migrations to re-create the database. To delete the database in SSOX:

1. Select the database in SSOX.

2. Right-click on the database, and select **Delete**.

3. Check **Close existing connections**.

4. Select **OK**.

5. In the PMC, update the database:

```
PowerShell

Update-Database
```

Run the app and verify you can create, edit, and display movies with a `Rating` field. If the database isn't seeded, set a break point in the `SeedData.Initialize` method.

Next steps

Previous: Add Search | Next: Add Validation

Part 8 of tutorial series on Razor Pages

Article • 07/01/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

By Rick Anderson

In this section, validation logic is added to the `Movie` model. The validation rules are enforced any time a user creates or edits a movie.

Validation

A key tenet of software development is called DRY ("**D**on't **R**epeat **Y**ourself"). Razor Pages encourages development where functionality is specified once, and it's reflected throughout the app. DRY can help:

- Reduce the amount of code in an app.
- Make the code less error prone, and easier to test and maintain.

The validation support provided by Razor Pages and Entity Framework is a good example of the DRY principle:

- Validation rules are declaratively specified in one place, in the model class.
- Rules are enforced everywhere in the app.

Add validation rules to the movie model

The System.ComponentModel.DataAnnotations namespace provides:

- A set of built-in validation attributes that are applied declaratively to a class or property.
- Formatting attributes like `[DataType]` that help with formatting and don't provide any validation.

Update the `Movie` class to take advantage of the built-in `[Required]`, `[StringLength]`, `[RegularExpression]`, and `[Range]` validation attributes.

C#

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

namespace RazorPagesMovie.Models;

public class Movie
{
    public int Id { get; set; }

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

    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

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

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

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

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

- The `[Required]` and `[MinimumLength]` attributes indicate that a property must have a value. 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 must be uppercase. White spaces are allowed, while numbers and special characters aren't 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 can set a 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.

The preceding validation rules are used for demonstration, they are not optimal for a production system. For example, the preceding prevents entering a movie with only two chars and doesn't allow special characters in `Genre`.

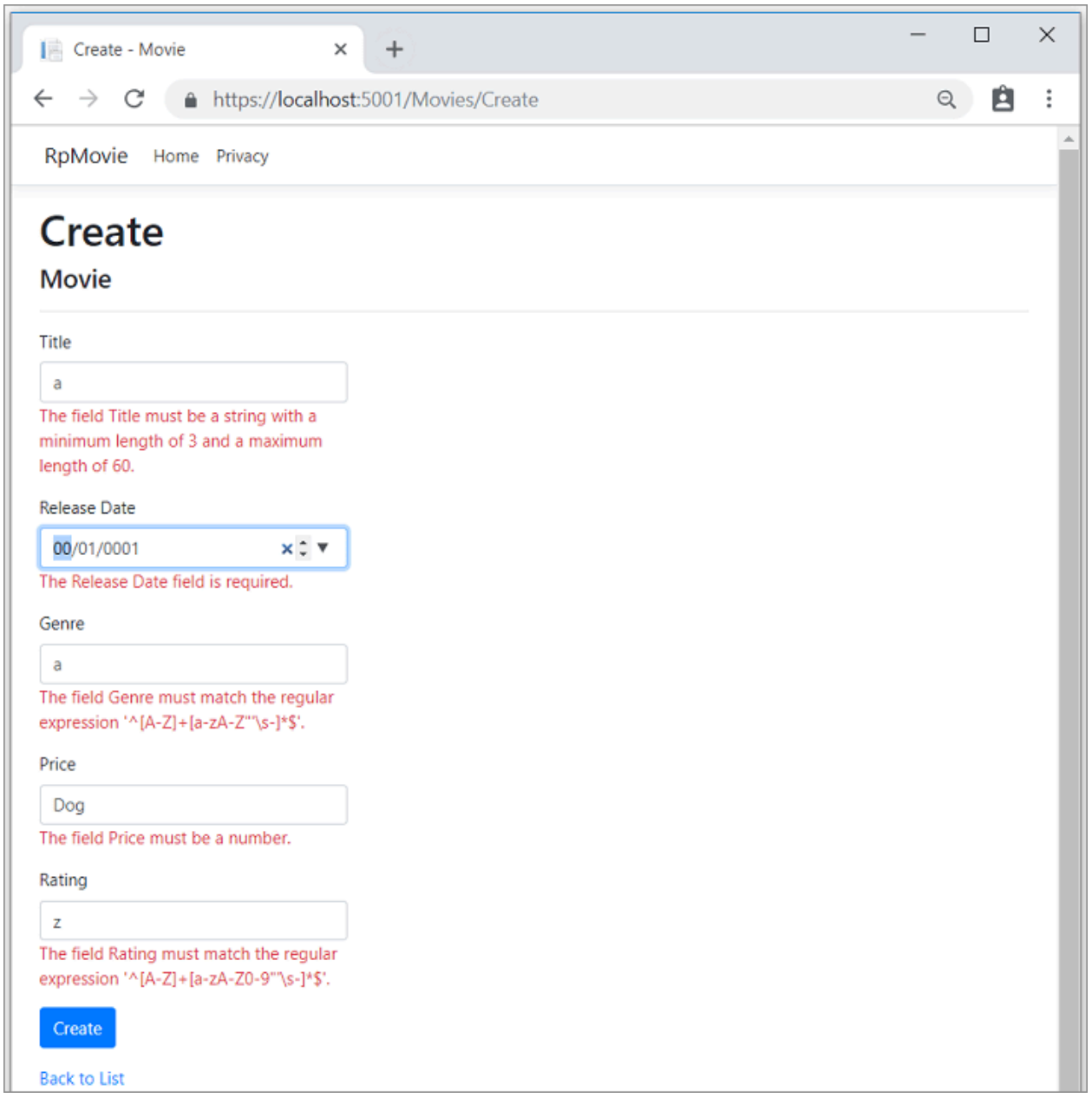
Having validation rules automatically enforced by ASP.NET Core helps:

- Make the app more robust.
- Reduce chances of saving invalid data to the database.

Validation Error UI in Razor Pages

Run the app and navigate to Pages/Movies.

Select the **Create New** link. Complete the form with some invalid values. When jQuery client-side validation detects the error, it displays an error message.



> **Note**
>
> You may not be able to enter decimal commas in decimal fields. To support jQuery validation for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize your app. See this GitHub comment 4076 for instructions on adding decimal comma.

Notice how the form has automatically rendered a validation error message in each field containing an invalid value. The errors are enforced both client-side, using JavaScript and jQuery, and server-side, when a user has JavaScript disabled.

A significant benefit is that **no** code changes were necessary in the Create or Edit pages. Once data annotations were applied to the model, the validation UI was enabled. The Razor Pages created in this tutorial automatically picked up the validation rules, using

validation attributes on the properties of the `Movie` model class. Test validation using the Edit page, the same validation is applied.

The form data isn't posted to the server until there are no client-side validation errors. Verify form data isn't posted by one or more of the following approaches:

- Put a break point in the `OnPostAsync` method. Submit the form by selecting **Create** or **Save**. The break point is never hit.
- Use the Fiddler tool.
- Use the browser developer tools to monitor network traffic.

Server-side validation

When JavaScript is disabled in the browser, submitting the form with errors will post to the server.

Optional, test server-side validation:

1. Disable JavaScript in the browser. JavaScript can be disabled using browser's developer tools. If JavaScript cannot be disabled in the browser, try another browser.
2. Set a break point in the `OnPostAsync` method of the Create or Edit page.
3. Submit a form with invalid data.
4. Verify the model state is invalid:

```
C#
```

```
if (!ModelState.IsValid)
{
    return Page();
}
```

Alternatively, Disable client-side validation on the server.

The following code shows a portion of the `Create.cshtml` page scaffolded earlier in the tutorial. It's used by the Create and Edit pages to:

- Display the initial form.
- Redisplay the form in the event of an error.

```
CSHTML
```

```
<form method="post">
    <div asp-validation-summary="ModelOnly" class="text-danger"></div>
    <div class="form-group">
        <label asp-for="Movie.Title" class="control-label"></label>
        <input asp-for="Movie.Title" class="form-control" />
        <span asp-validation-for="Movie.Title" class="text-danger"></span>
    </div>
```

The Input Tag Helper uses the DataAnnotations attributes and produces HTML attributes needed for jQuery Validation on the client-side. The Validation Tag Helper displays validation errors. See Validation for more information.

The Create and Edit pages have no validation rules in them. The validation rules and the error strings are specified only in the `Movie` class. These validation rules are automatically applied to Razor Pages that edit the `Movie` model.

When validation logic needs to change, it's done only in the model. Validation is applied consistently throughout the app, validation logic is defined in one place. Validation in one place helps keep the code clean, and makes it easier to maintain and update.

Use DataType Attributes

Examine the `Movie` class. The `System.ComponentModel.DataAnnotations` namespace provides formatting attributes in addition to the built-in set of validation attributes. The `[DataType]` attribute is applied to the `ReleaseDate` and `Price` properties.

C#

```
[DataType(DataType.Date)]
public DateTime ReleaseDate { get; set; }

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

The `[DataType]` attributes provide:

- Hints for the view engine to format the data.
- Supplies attributes such as `<a>` for URL's and `<a href="mailto:EmailAddress.com">` for email.

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. `[DataType]` attributes aren't validation attributes. In the sample app, only the date is displayed, without time.

The `DataType` enumeration provides many data types, such as `Date`, `Time`, `PhoneNumber`, `Currency`, `EmailAddress`, and more.

The `[DataType]` attributes:

- Can enable the app to automatically provide type-specific features. For example, a `mailto:` link can be created for `DataType.EmailAddress`.
- Can provide a date selector `DataType.Date` in browsers that support HTML5.
- Emit HTML 5 `data-`, pronounced "data dash", attributes that HTML 5 browsers consume.
- 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 `[Column(TypeName = "decimal(18, 2)")]` data annotation is required so Entity Framework Core can correctly map `Price` to currency in the database. For more information, see Data Types.

The `[DisplayFormat]` attribute is used to explicitly specify the date format:

```
C#

[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode =
true)]
public DateTime ReleaseDate { get; set; }
```

The `ApplyFormatInEditMode` setting specifies that the formatting will be applied when the value is displayed for editing. That behavior may not be wanted for some fields. For example, in currency values, the currency symbol is usually not wanted in the edit UI.

The `[DisplayFormat]` attribute can be used 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. The `[DataType]` attribute provides the following benefits that aren't available 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 renders data using the correct format based on its locale.

- The `[DataType]` attribute can enable the ASP.NET Core framework to choose the right field template to render the data. The `DisplayFormat`, if used by itself, uses the string template.

Note: jQuery validation doesn't work with the `[Range]` attribute and `DateTime`. For example, the following code will always display a client-side validation error, even when the date is in the specified range:

C#

```
[Range(typeof(DateTime), "1/1/1966", "1/1/2020")]
```

It's a best practice to avoid compiling hard dates in models, so using the `[Range]` attribute and `DateTime` is discouraged. Use Configuration for date ranges and other values that are subject to frequent change rather than specifying it in code.

The following code shows combining attributes on one line:

C#

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

namespace RazorPagesMovie.Models;

public class Movie
{
    public int Id { get; set; }

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

    [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; } = string.Empty;

    [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; } = string.Empty;
}
```

Get started with Razor Pages and EF Core shows advanced EF Core operations with Razor Pages.

Apply migrations

The DataAnnotations applied to the class changes the schema. For example, the DataAnnotations applied to the `Title` field:

C#

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

- Limits the characters to 60.
- Doesn't allow a `null` value.

The `Movie` table currently has the following schema:

SQL

```
CREATE TABLE [dbo].[Movie] (
    [ID]          INT             IDENTITY (1, 1) NOT NULL,
    [Title]       NVARCHAR (MAX)  NULL,
    [ReleaseDate] DATETIME2 (7)   NOT NULL,
    [Genre]       NVARCHAR (MAX)  NULL,
    [Price]       DECIMAL (18, 2) NOT NULL,
    [Rating]      NVARCHAR (MAX)  NULL,
    CONSTRAINT [PK_Movie] PRIMARY KEY CLUSTERED ([ID] ASC)
);
```

The preceding schema changes don't cause EF to throw an exception. However, create a migration so the schema is consistent with the model.

Visual Studio

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

PowerShell

```
Add-Migration New_DataAnnotations
Update-Database
```

`Update-Database` runs the `Up` method of the `New_DataAnnotations` class.

Examine the `Up` method:

C#

```
public partial class New_DataAnnotations : Migration
{
    /// <inheritdoc />
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.AlterColumn<string>(
            name: "Title",
            table: "Movie",
            type: "nvarchar(60)",
            maxLength: 60,
            nullable: false,
            oldClrType: typeof(string),
            oldType: "nvarchar(max)");

        migrationBuilder.AlterColumn<string>(
            name: "Rating",
            table: "Movie",
            type: "nvarchar(5)",
            maxLength: 5,
            nullable: false,
            oldClrType: typeof(string),
            oldType: "nvarchar(max)");

        migrationBuilder.AlterColumn<string>(
            name: "Genre",
            table: "Movie",
            type: "nvarchar(30)",
            maxLength: 30,
            nullable: false,
            oldClrType: typeof(string),
            oldType: "nvarchar(max)");
    }
```

The updated `Movie` table has the following schema:

SQL

```
CREATE TABLE [dbo].[Movie] (
    [ID]          INT             IDENTITY (1, 1) NOT NULL,
    [Title]       NVARCHAR (60)   NOT NULL,
    [ReleaseDate] DATETIME2 (7)   NOT NULL,
    [Genre]       NVARCHAR (30)   NOT NULL,
    [Price]       DECIMAL (18, 2) NOT NULL,
    [Rating]      NVARCHAR (5)    NOT NULL,
    CONSTRAINT [PK_Movie] PRIMARY KEY CLUSTERED ([ID] ASC)
);
```

Publish to Azure

For information on deploying to Azure, see Tutorial: Build an ASP.NET Core app in Azure with SQL Database.

Thanks for completing this introduction to Razor Pages. Get started with Razor Pages and EF Core is an excellent follow up to this tutorial.

Additional resources

- Tag Helpers in forms in ASP.NET Core
- Globalization and localization in ASP.NET Core
- Tag Helpers in ASP.NET Core
- Author Tag Helpers in ASP.NET Core

Next steps

Previous: Add a new field

Get started with ASP.NET Core MVC

Article • 07/30/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

By Rick Anderson

This tutorial teaches ASP.NET Core MVC web development with controllers and views. If you're new to ASP.NET Core web development, consider the Razor Pages version of this tutorial, which provides an easier starting point. See Choose an ASP.NET Core UI, which compares Razor Pages, MVC, and Blazor for UI development.

This is the first tutorial of a series that teaches ASP.NET Core MVC web development with controllers and views.

At the end of the series, you'll have an app that manages, validates, and displays movie data. You learn how to:

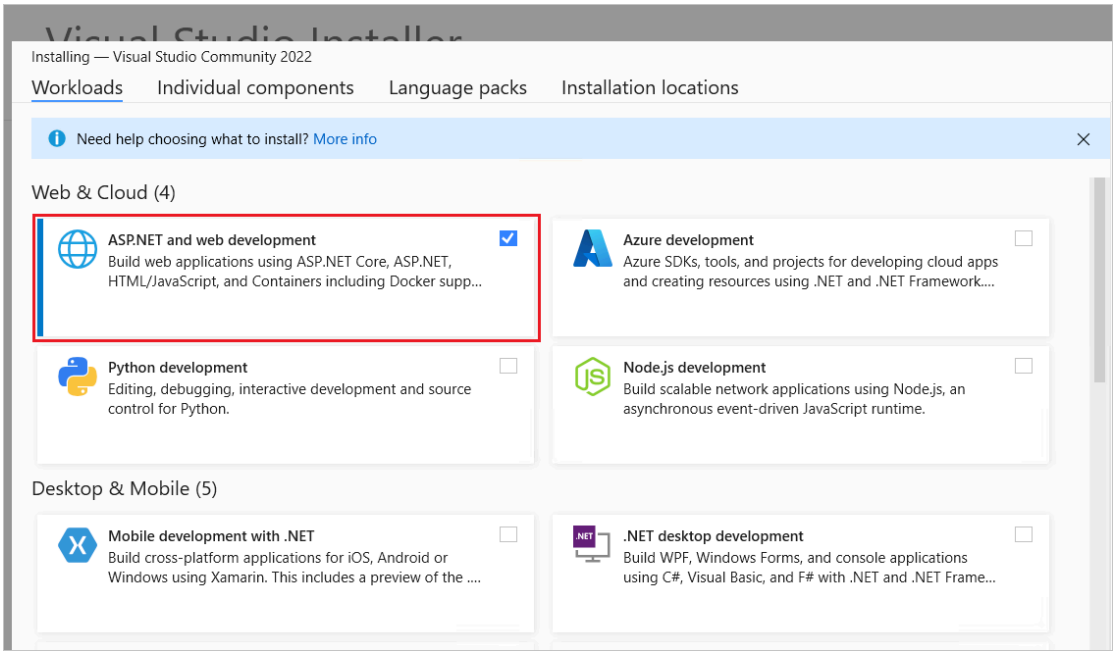
- ✓ Create a web app.
- ✓ Add and scaffold a model.
- ✓ Work with a database.
- ✓ Add search and validation.

View or download sample code (how to download).

Prerequisites

Visual Studio

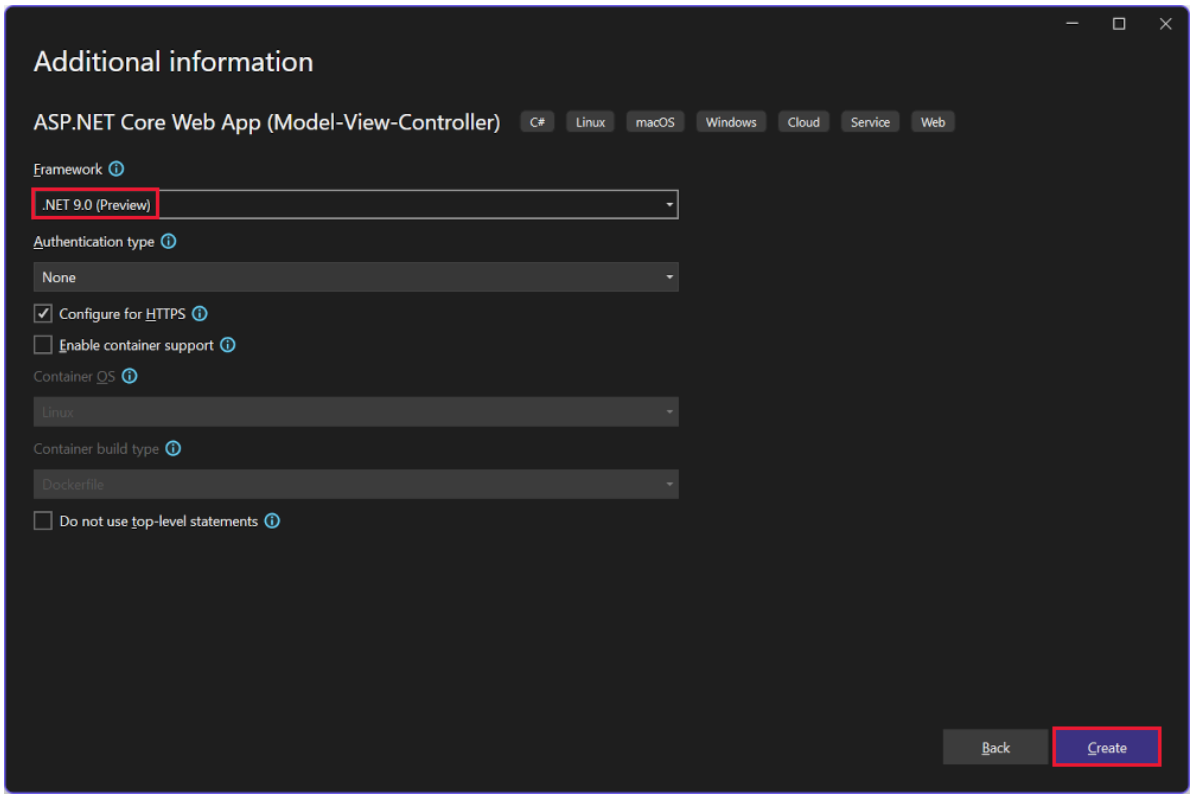
- Visual Studio 2022 Preview with the **ASP.NET and web development** workload.



Create a web app

Visual Studio

- 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.
  - The **Location** for the project can be set to anywhere.
- Select **Next**.
- In the **Additional information** dialog:
  - Select **.NET 9.0 (Preview)**.
  - Verify that **Do not use top-level statements** is unchecked.
- Select **Create**.



For more information, including alternative approaches to create the project, see Create a new project in Visual Studio.

Visual Studio uses the default project template for the created MVC project. The created project:

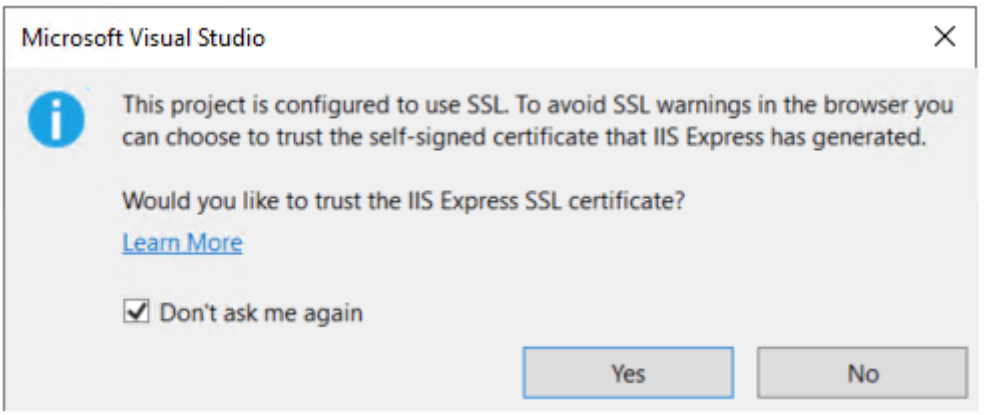
- Is a working app.
- Is a basic starter project.

Run the app

Visual Studio

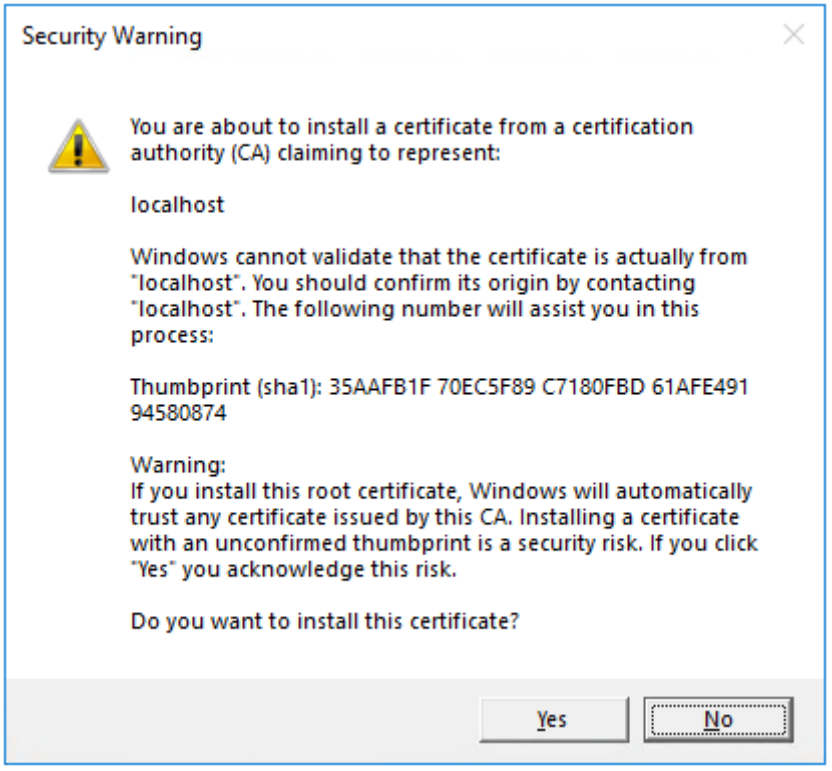
- Press `Ctrl`+`F5` to run the app without the debugger.

  Visual Studio displays the following dialog when a project is not yet configured to use SSL:



Select **Yes** if you trust the IIS Express SSL certificate.

The following dialog is displayed:



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

For information on trusting the Firefox browser, see Firefox SEC_ERROR_INADEQUATE_KEY_USAGE certificate error.

Visual Studio runs the app and opens the default browser.

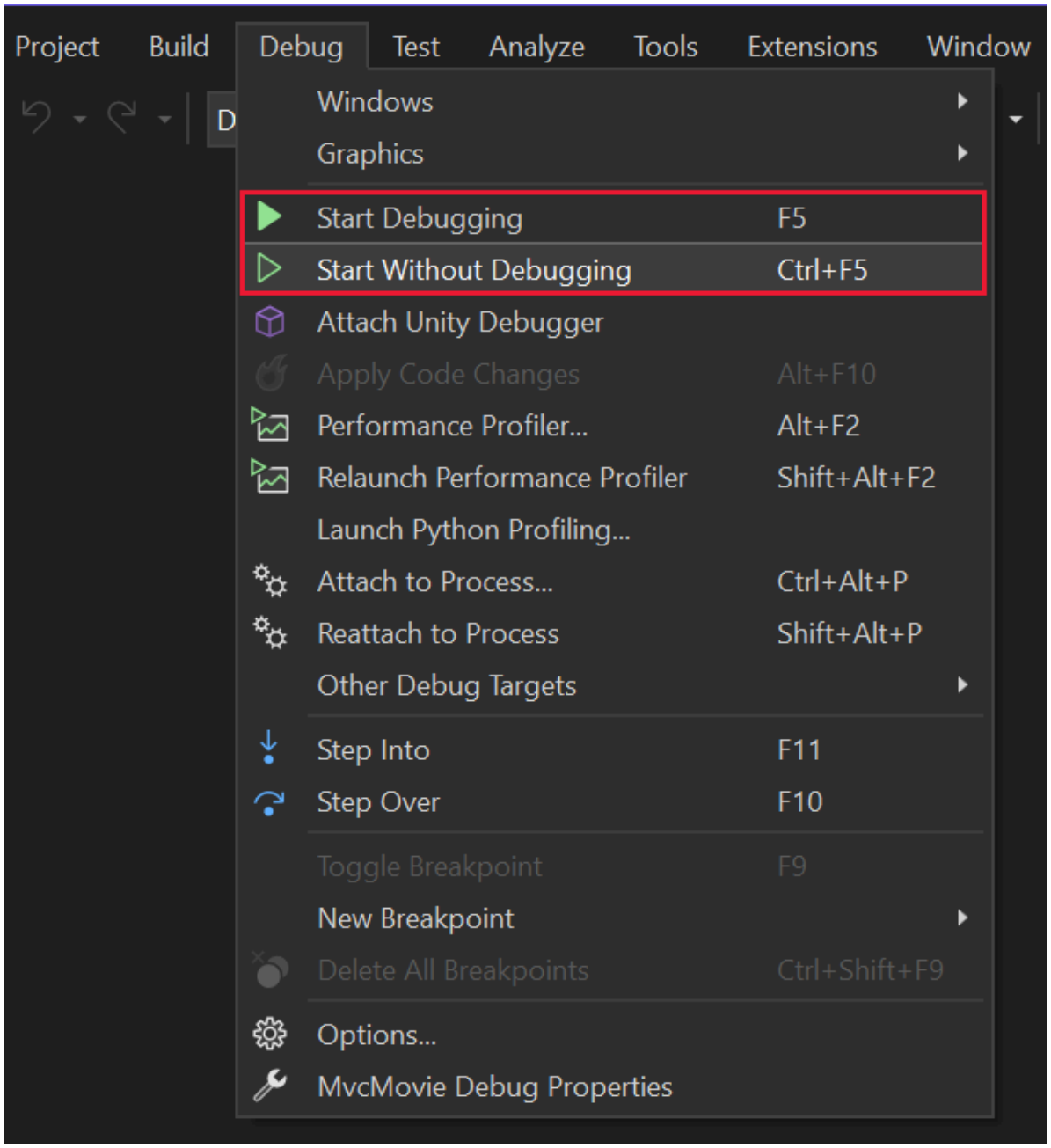
The address bar shows `localhost:<port#>` and not something like `example.com`. The standard hostname for your local computer is `localhost`. When Visual Studio creates a web project, a random port is used for the web server.

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

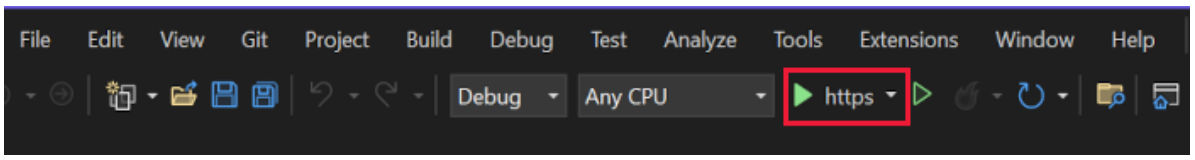
- Make code changes.

- Save the file.
- Quickly refresh the browser and see the code changes.

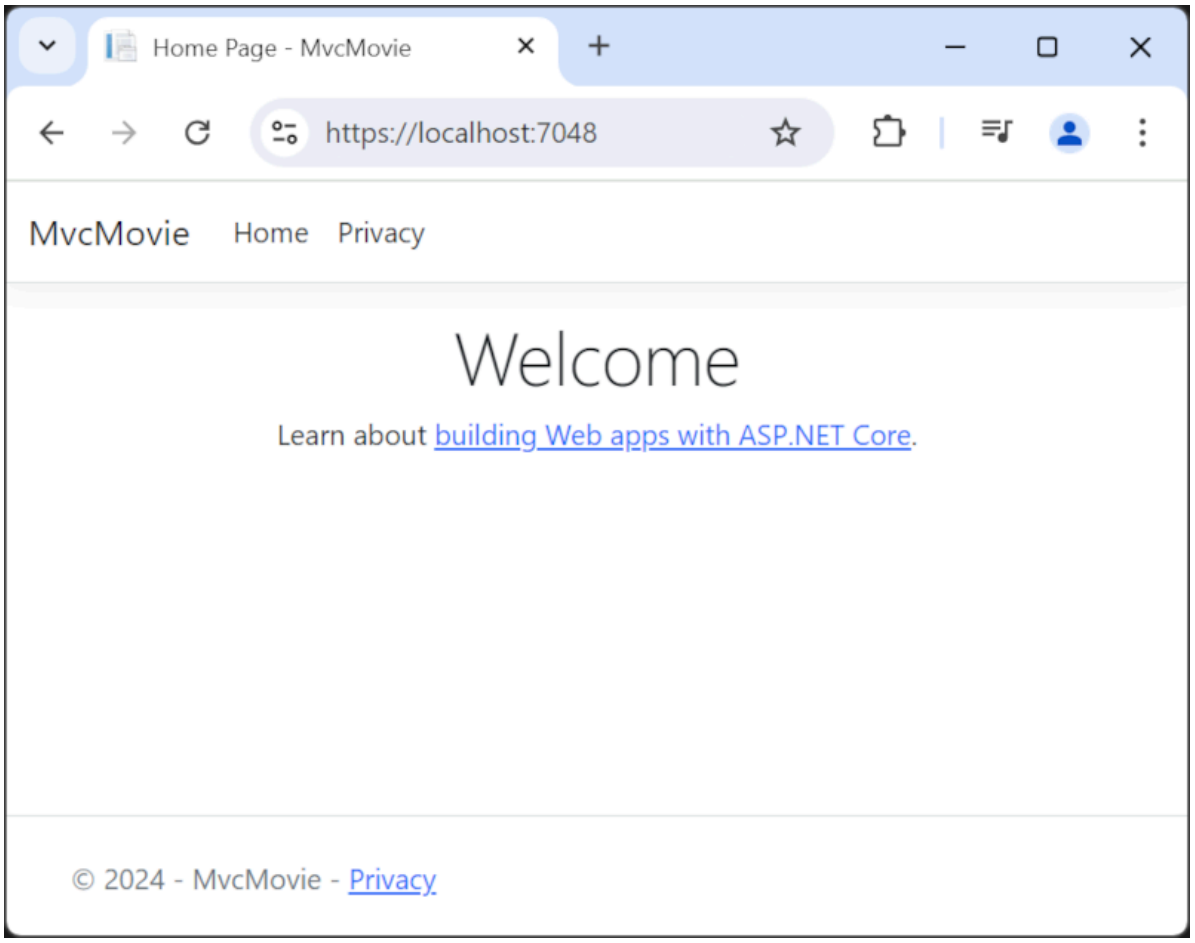
You can launch the app in debug or non-debug mode from the **Debug** menu:



You can debug the app by selecting the **https** button in the toolbar:



The following image shows the app:



- Close the browser window. Visual Studio will stop the application.

Visual Studio

Visual Studio help

- Learn to debug C# code using Visual Studio
- Introduction to the Visual Studio IDE

In the next tutorial in this series, you learn about MVC and start writing some code.

Next: Add a controller

Part 2, add a controller to an ASP.NET Core MVC app

Article • 07/30/2024

> **ⓘ Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By Rick Anderson

The Model-View-Controller (MVC) architectural pattern separates an app into three main components: **M**odel, **V**iew, and **C**ontroller. The MVC pattern helps you create apps that are more testable and easier to update than traditional monolithic apps.

MVC-based apps contain:

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

In an MVC app, the view only displays information. The controller handles and responds to user input and interaction. For example, the controller handles URL segments and query-string values, and passes these values to the model. The model might use these values to query the database. 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 tutorial.

Route data is explained later in the tutorial.

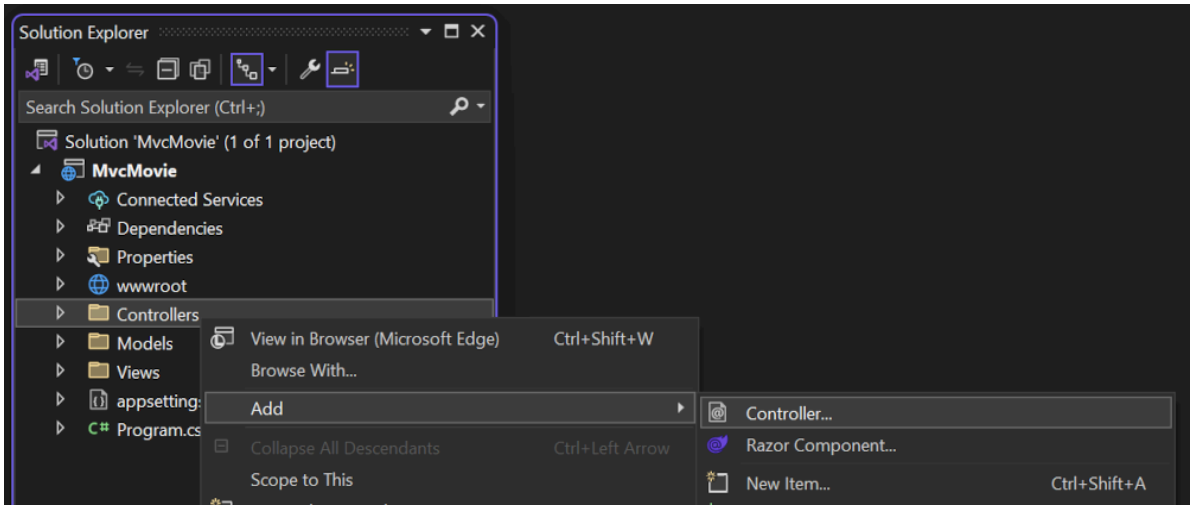
The MVC architectural pattern separates an app into three main groups of components: Models, Views, and Controllers. This pattern helps to achieve separation of concerns: The UI logic belongs in the view. Input logic belongs in the controller. Business logic belongs in the model. This separation helps manage complexity when building an app, because it enables work on one aspect of the implementation at a time without impacting the code of another. For example, you can work on the view code without depending on the business logic code.

These concepts are introduced and demonstrated in this tutorial series while building a movie app. The MVC project contains folders for the *Controllers* and *Views*.

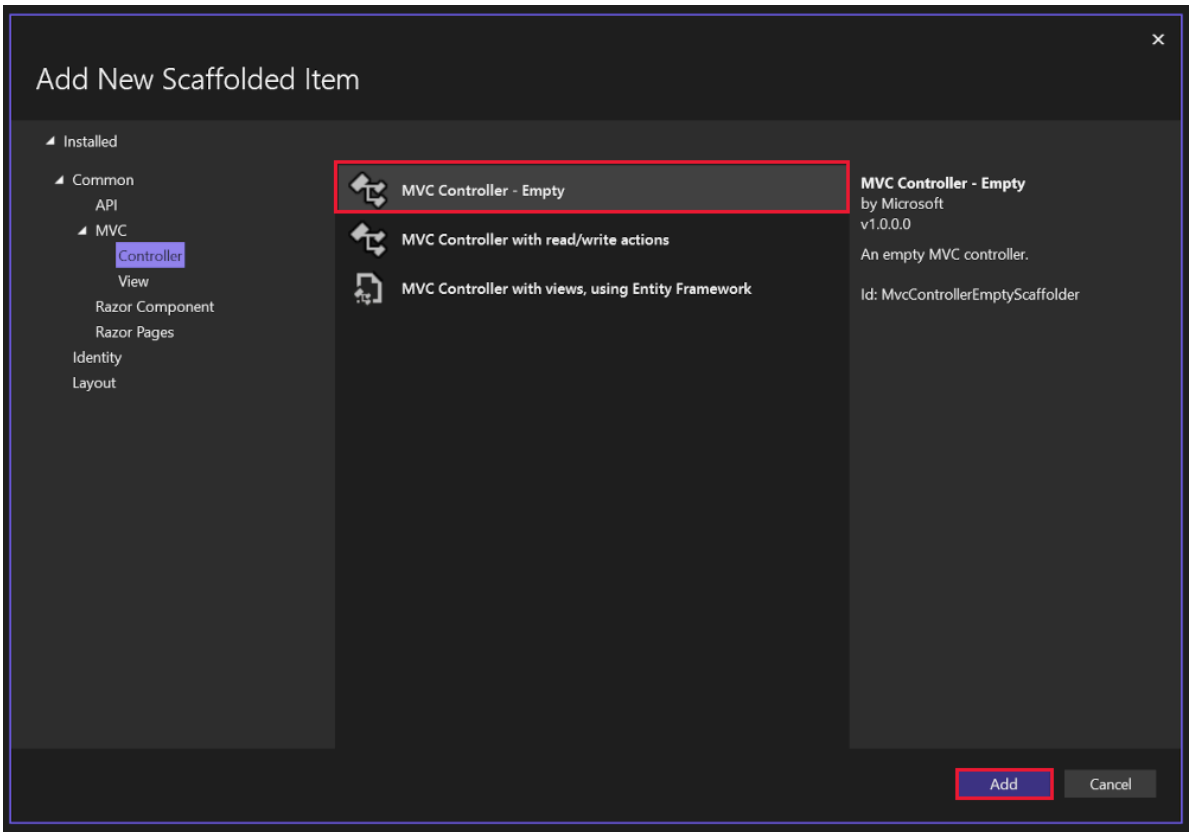
Add a controller

Visual Studio

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



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



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

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

C#

```
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers;

public class HelloWorldController : Controller
{
    //
    // 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:

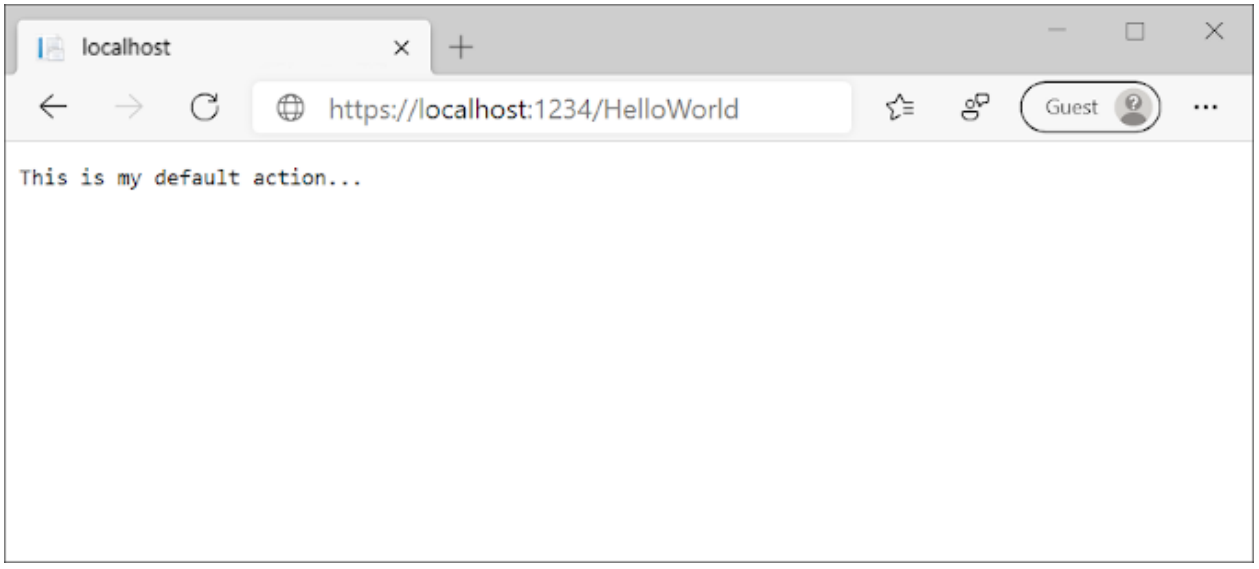
- Is a targetable URL in the web application, such as `https://localhost:5001/HelloWorld`.
- Combines:
  - The protocol used: `HTTPS`.
  - The network location of the web server, including the TCP port: `localhost:5001`.
  - The target URI: `HelloWorld`.

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 tutorial, the scaffolding engine is used to generate `HTTP POST` methods, which update data.

Run the app without the debugger by pressing `Ctrl`+`F5` (Windows) or `⌘`+`F5` (macOS).

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.

C#

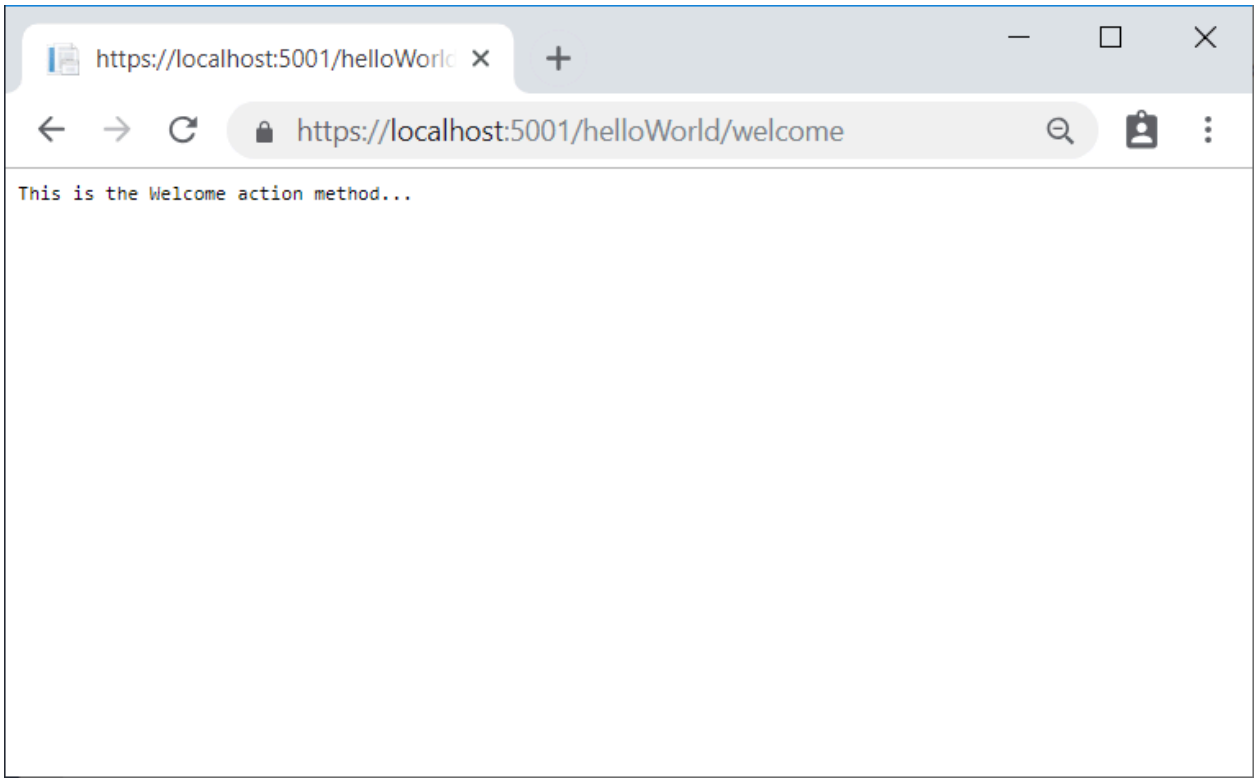
```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

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 tutorial.

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.

C#

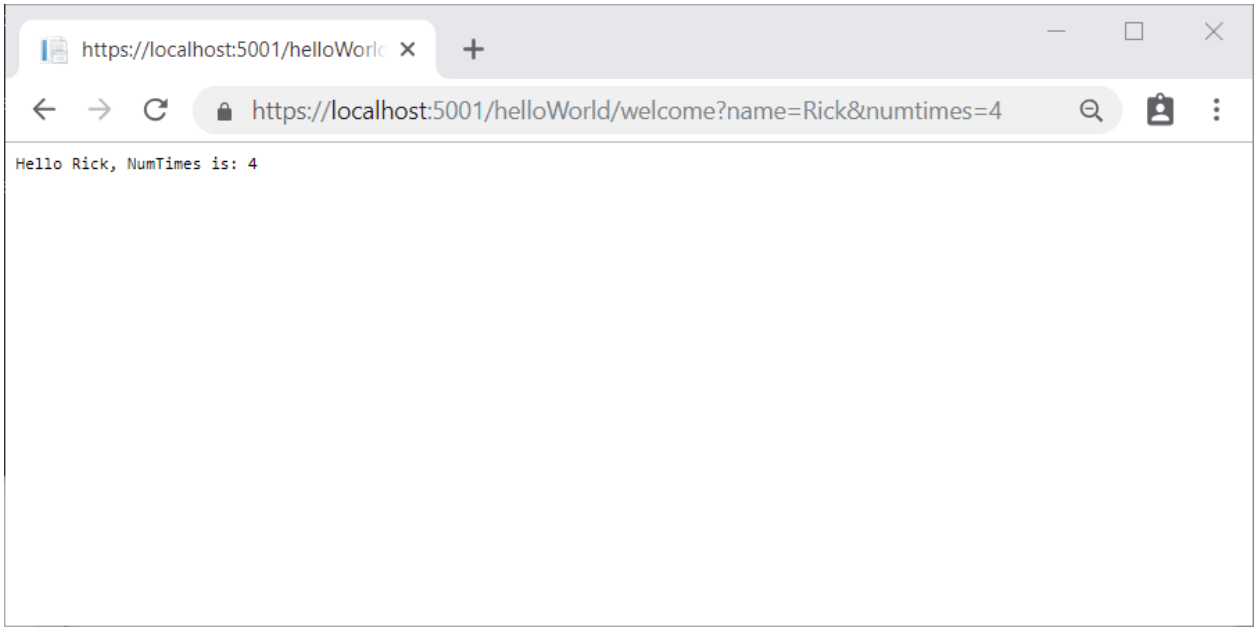
```
// GET: /HelloWorld/Welcome/
// Requires using System.Text.Encodings.Web;
public string Welcome(string name, int numTimes = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, NumTimes is:
{numTimes}");
}
```

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. See Model Binding for more information.



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:

C#

```
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.

C#

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

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.

Previous: Get Started Next: Add a View

Part 3, add a view to an ASP.NET Core MVC app

Article • 07/30/2024

> **ⓘ Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By Rick Anderson

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:

C#

```
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`.

Add a view

Visual Studio

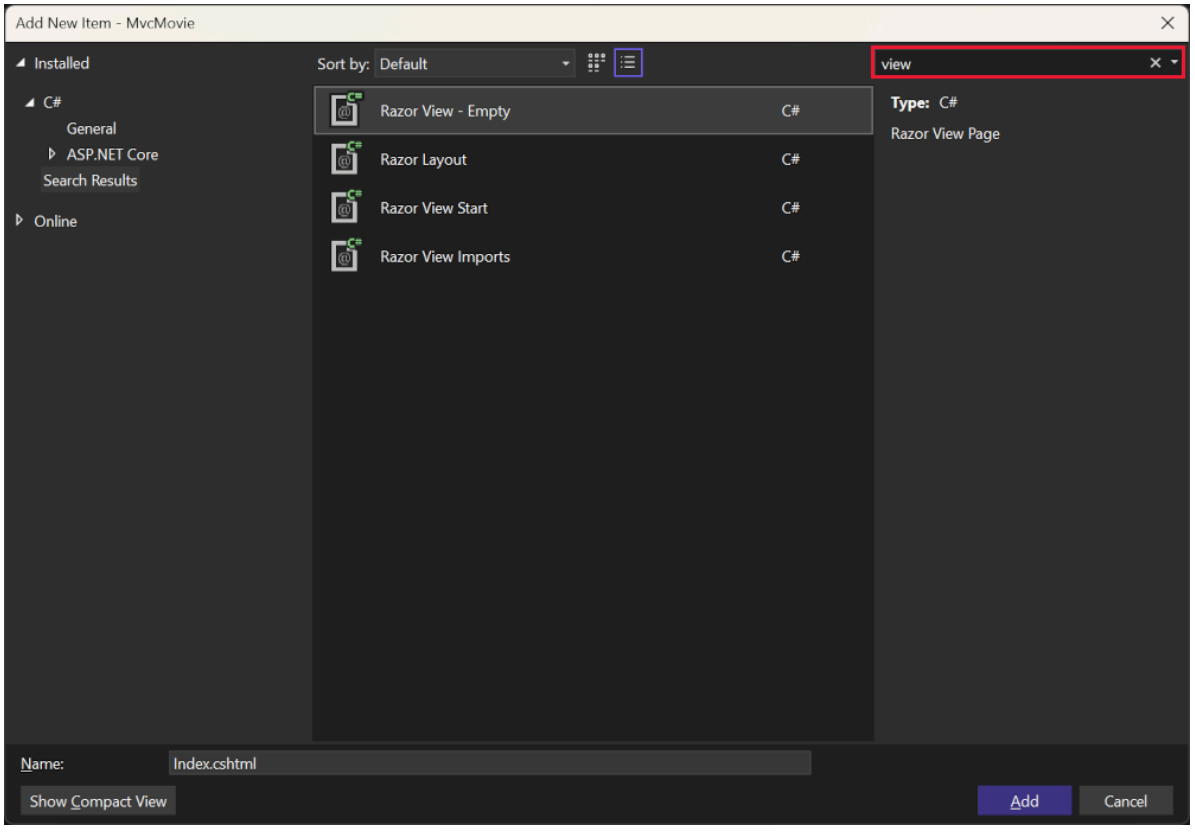
Right-click on the *Views* folder, and then **Add** > **New Folder** and name the folder *HelloWorld*.

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

In the **Add New Item** dialog select **Show All Templates**.

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**



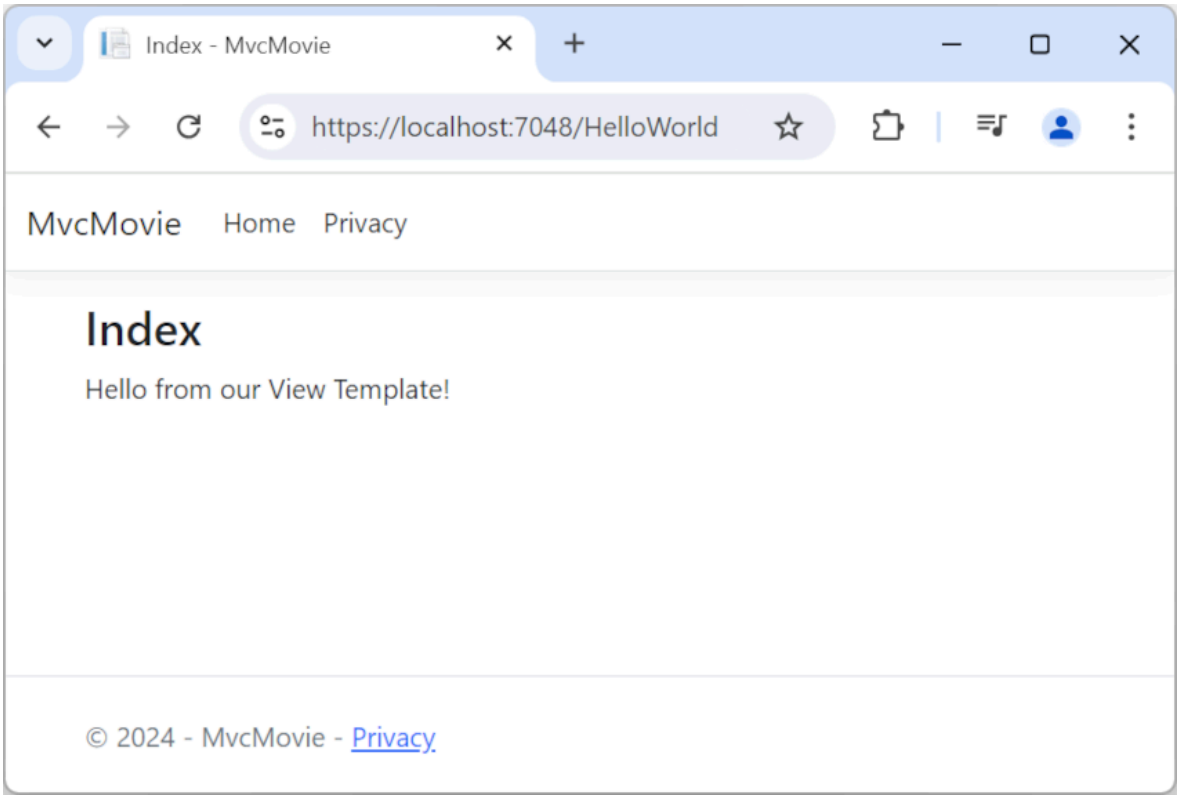
Replace the contents of the `Views/HelloWorld/Index.cshtml` Razor view file with the following:

CSHTML

```
@{
    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:



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:

CSHTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Movie App</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" asp-append-version="true"
/>
    <link rel="stylesheet" href="~/MvcMovie.styles.css" asp-append-
version="true" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-
light bg-white border-bottom box-shadow mb-3">
            <div class="container-fluid">
                <a class="navbar-brand" asp-area="" asp-controller="Movies"
asp-action="Index">Movie App</a>
                <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="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; 2024 - Movie App - <a asp-area="" asp-controller="Home"
asp-action="Privacy">Privacy</a>
        </div>
    </footer>
    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
    @await RenderSectionAsync("Scripts", required: false)
</body>
</html>
```

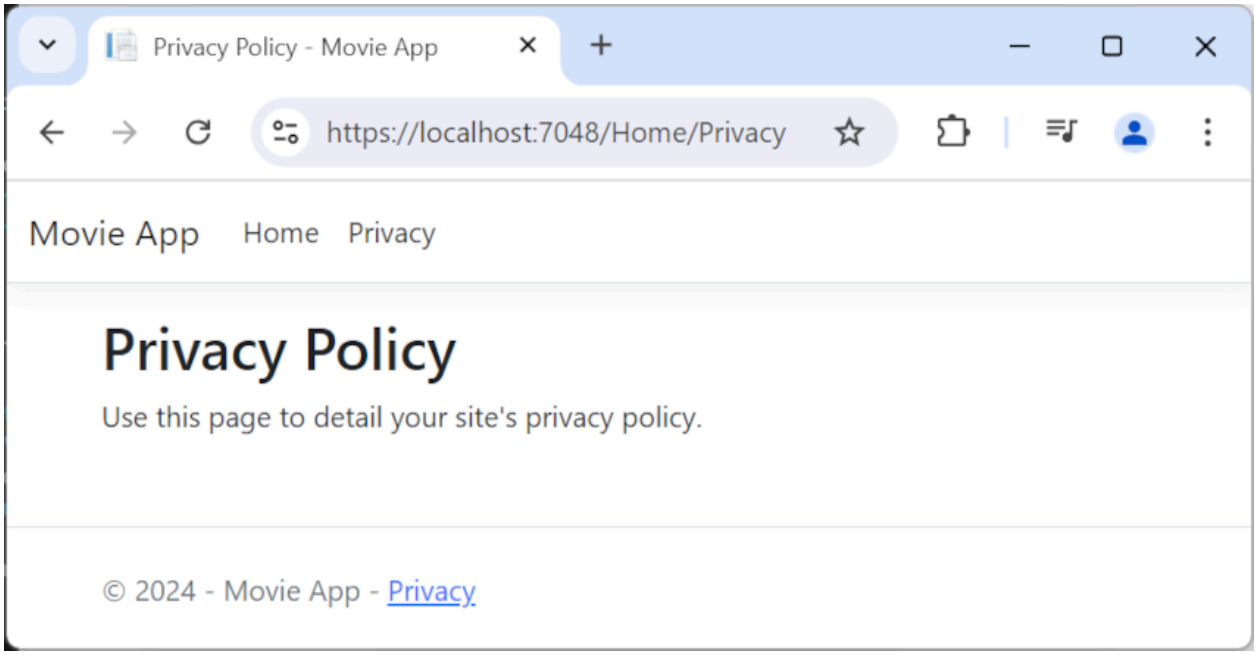
The preceding markup made the following changes:

- Three occurrences of `MvcMovie` to `Movie App`.
- The anchor element `<a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">MvcMovie</a>` to `<a class="navbar-brand" asp-controller="Movies" asp-action="Index">Movie App</a>`.

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**



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.

Examine the `Views/_ViewStart.cshtml` file:

CSHTML

```
@{
    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:

CSHTML

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

<h2>My Movie List</h2>
```

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

The title and `<h2>` 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:

CSHTML

```
<title>@ViewData["Title"] - Movie App</title>
```

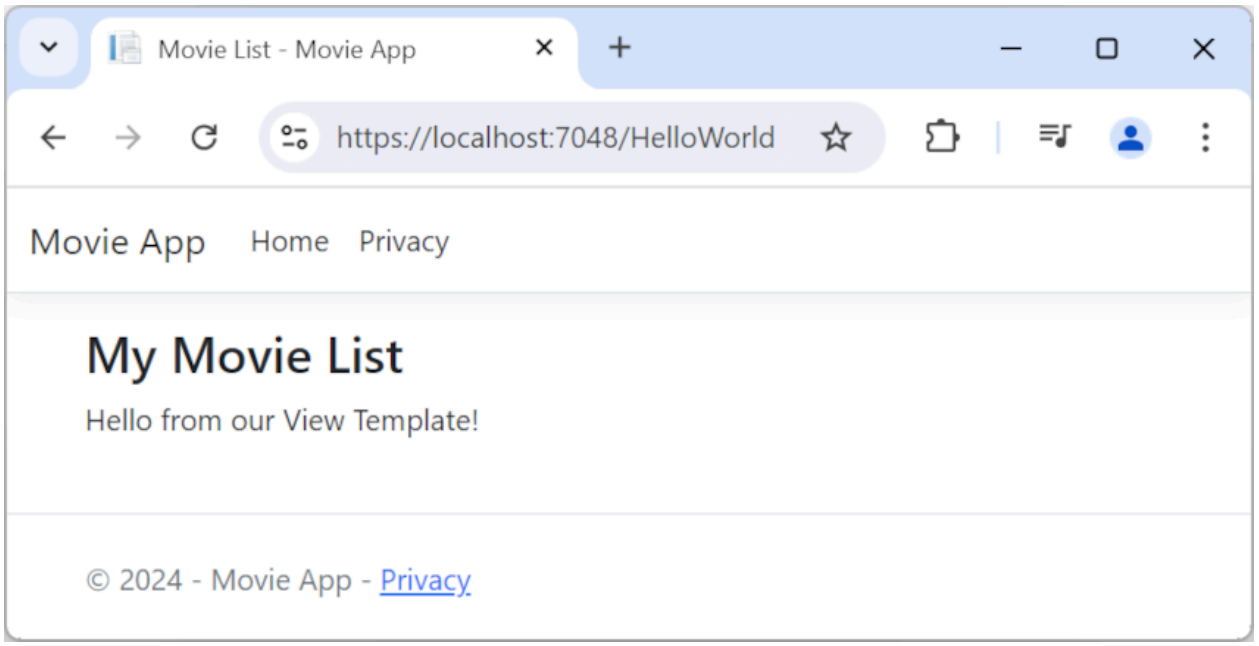
Save the change and navigate to `https://localhost:{PORT}/HelloWorld`.

Notice that the following have changed:

- Browser title.
- Primary heading.
- Secondary headings.

If there are no changes in the browser, it could be cached content that is being viewed. Press Ctrl+F5 in the browser to force the response from the server to be loaded. The browser title is created with `ViewData["Title"]` we set in the `Index.cshtml` view template and the additional "- Movie App" added in the layout file.

The content in the `Index.cshtml` view template is merged with the `Views/Shared/_Layout.cshtml` view template. A single HTML response is sent to the browser. Layout templates make it easy to make changes that apply across all of the pages in an app. To learn more, see Layout.



The small bit of "data", the "Hello from our View Template!" message, is hard-coded however. The MVC application has a "V" (view), a "C" (controller), but no "M" (model) yet.

Passing Data from the Controller to the View

Controller actions are invoked in response to an incoming URL request. A controller class is where the code is written that handles the incoming browser requests. The controller retrieves data from a data source and decides what type of response to send back to the browser. View templates can be used from a controller to generate and format an HTML response to the browser.

Controllers are responsible for providing the data required in order for a view template to render a response.

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.

Currently, the `Welcome` method in the `HelloWorldController` class takes a `name` and an `ID` parameter and then outputs the values directly to the browser.

Rather than have the controller render this response as a string, change the controller to use a view template instead. The view template generates a dynamic response, which means that appropriate data must be passed from the controller to the view to generate the response. Do this by having the controller put the dynamic data (parameters) that the view template needs in a `ViewData` dictionary. The view template can then access the dynamic data.

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

The `ViewData` dictionary is a dynamic object, which means any type can be used. The `ViewData` object has no defined properties until something is added. The MVC model binding system automatically maps the named parameters `name` and `numTimes` from the query string to parameters in the method. The complete `HelloWorldController`:

C#

```
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers;

public class HelloWorldController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
    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:

CSHTML

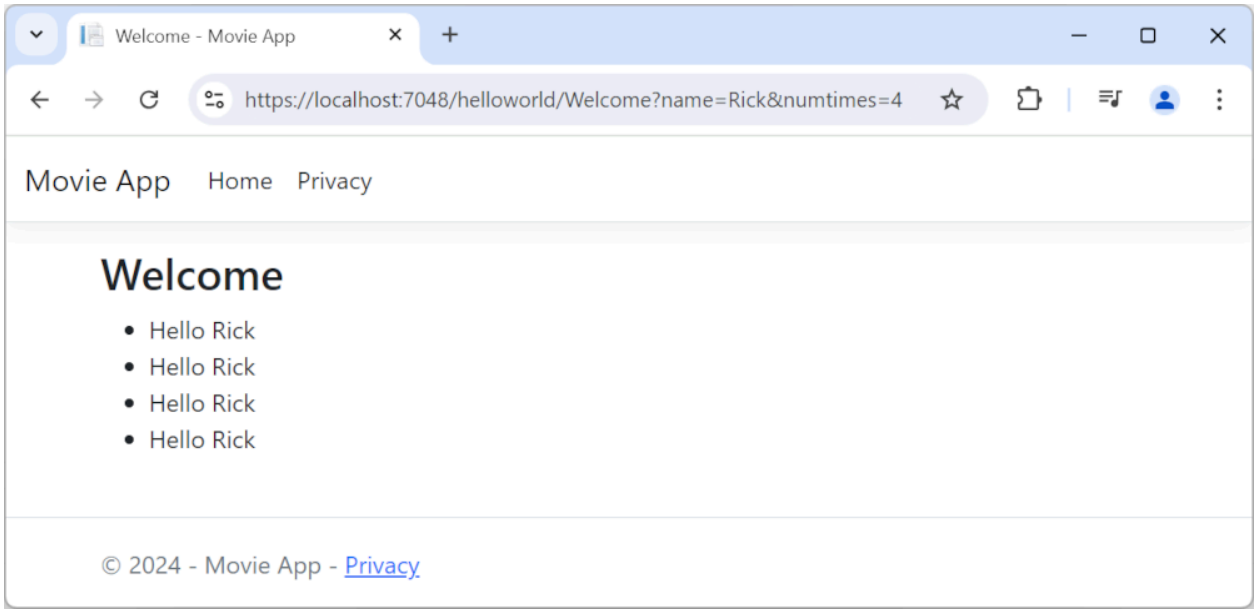
```
@{
    ViewData["Title"] = "Welcome";
```

```
}

<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.



In the preceding sample, the `ViewData` dictionary was used to pass data from the controller to a view. Later in the tutorial, a view model is used to pass data from a controller to a view. The view model approach to passing data is preferred over the `ViewData` dictionary approach.

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

Previous: Add a Controller | Next: Add a Model

Part 4, add a model to an ASP.NET Core MVC app

Article • 07/30/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By Rick Anderson and Jon P Smith.

In this tutorial, classes are added for managing movies in a database. These classes are the "**M**odel" part of the **M**VC app.

These model classes are used with Entity Framework Core (EF Core) to work with a database. EF Core is an object-relational mapping (ORM) framework that simplifies the data access code that you have to write.

The model classes created are known as ***POCO*** classes, from **P**lain **O**ld **C**LR **O**bjects. 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 tutorial, model classes are created first, and EF Core creates the database.

Add a data model class

Visual Studio

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

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

C#

```
using System.ComponentModel.DataAnnotations;

namespace MvcMovie.Models;
```

```
public class Movie
{
    public int Id { get; set; }
    public string? Title { get; set; }
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }
    public string? Genre { get; set; }
    public decimal Price { get; set; }
}
```

The `Movie` class contains an `Id` field, which is required by the database for the primary key.

The DataType attribute on `ReleaseDate` specifies the type of the data (`Date`). With this attribute:

- 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 tutorial.

The question mark after `string` indicates that the property is nullable. For more information, see Nullable reference types.

Add NuGet packages

Visual Studio

Visual Studio automatically installs the required packages.

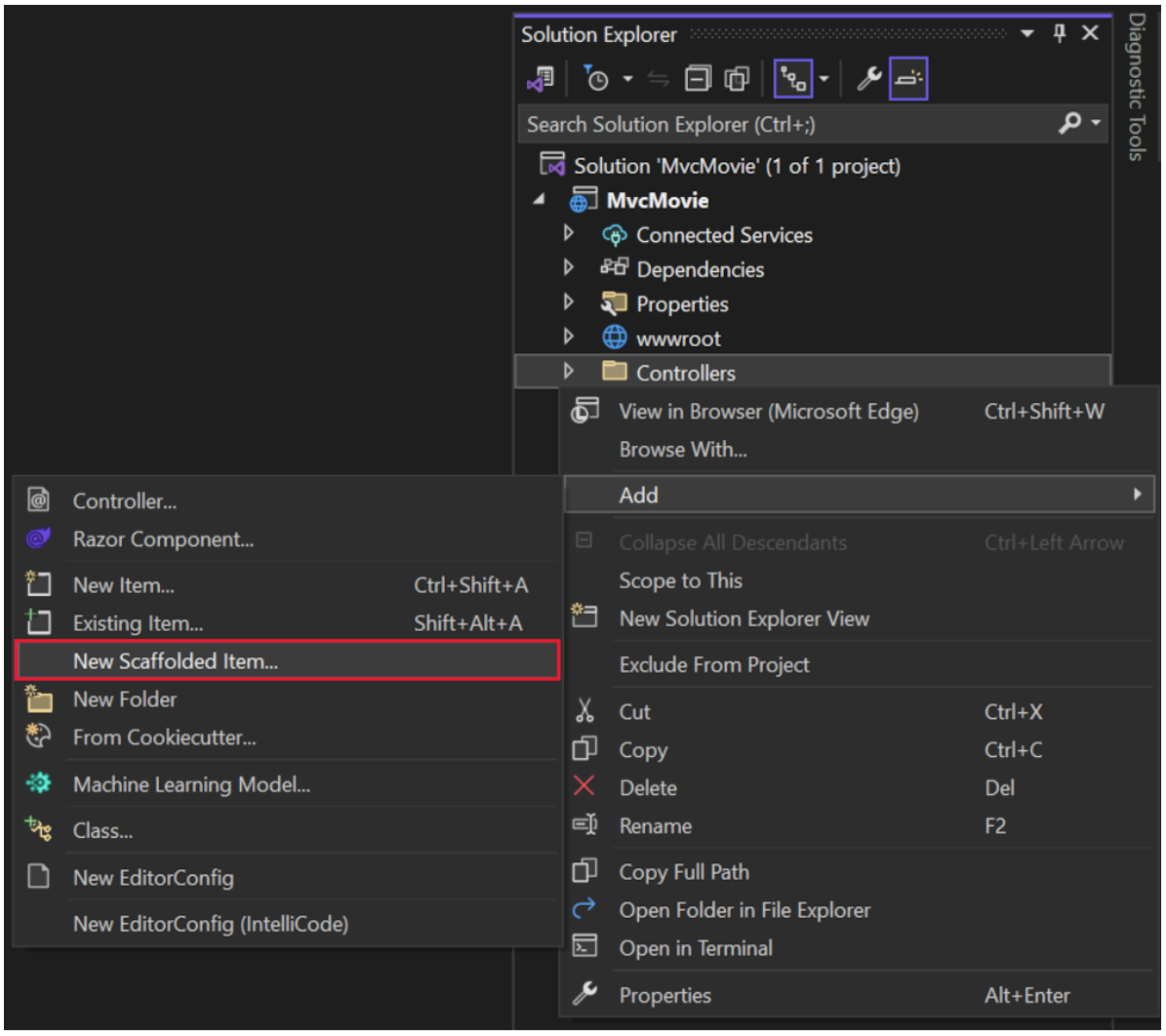
Build the project as a check for compiler errors.

Scaffold movie pages

Use the scaffolding tool to produce `Create`, `Read`, `Update`, and `Delete` (CRUD) pages for the movie model.

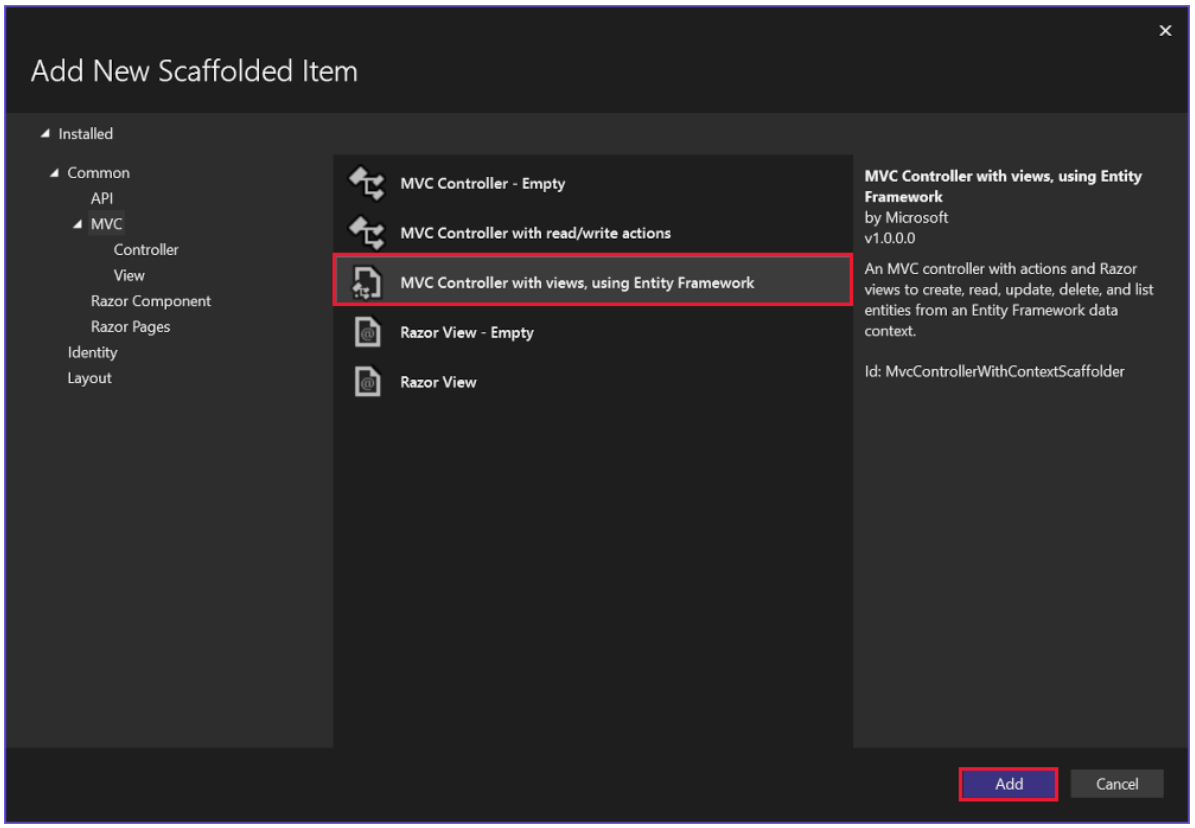
Visual Studio

In **Solution Explorer**, right-click the *Controllers* folder and select **Add** > **New Scaffolded Item**.



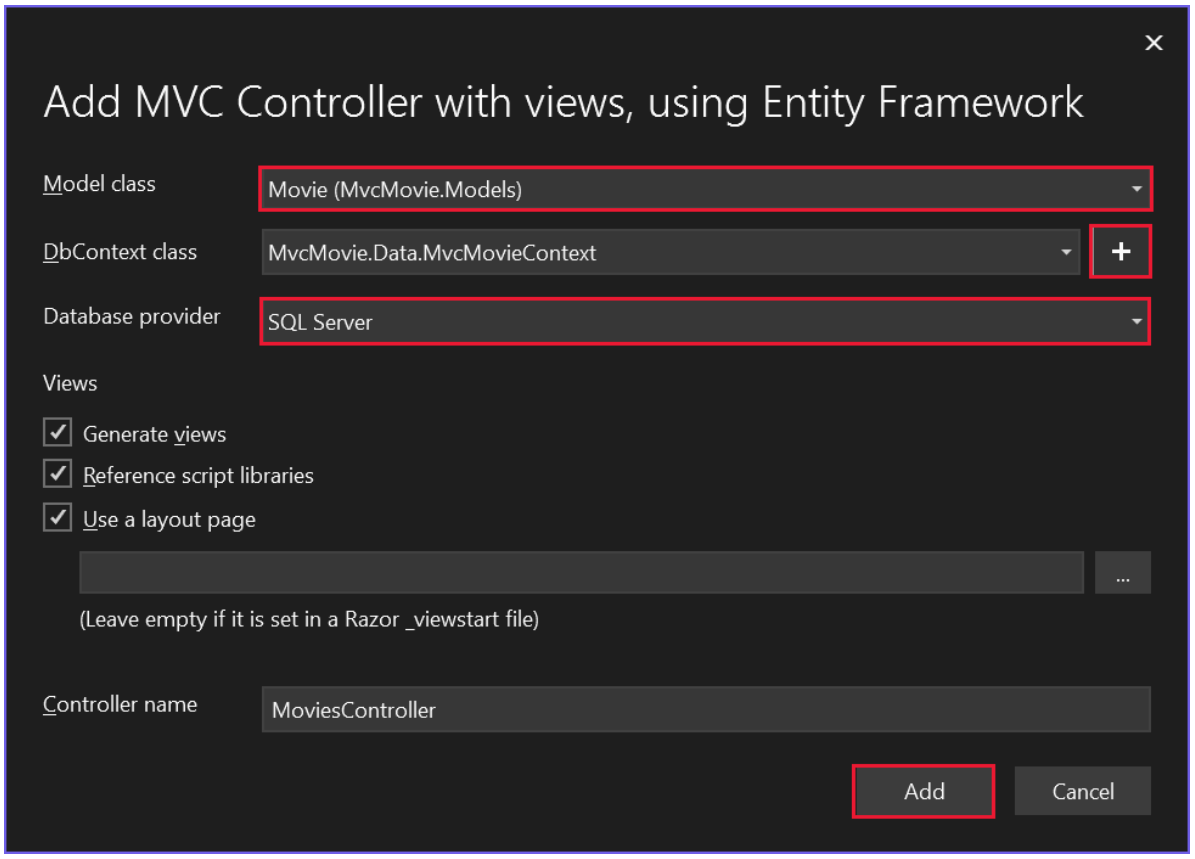
In the **Add New Scaffolded Item** dialog:

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



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

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



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

Scaffolding adds the following packages:

- `Microsoft.EntityFrameworkCore.SqlServer`
- `Microsoft.EntityFrameworkCore.Tools`
- `Microsoft.VisualStudio.Web.CodeGeneration.Design`

Scaffolding creates the following:

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

Scaffolding updates the following:

- Inserts required package references in the `MvcMovie.csproj` project file.
- Registers the database context in the `Program.cs` file.
- Adds a database connection string to the `appsettings.json` file.

The automatic creation of these files and file updates is known as *scaffolding*.

The scaffolded pages can't be used yet because the database doesn't exist. Running the app and selecting the **Movie App** link results in a *Cannot open database* or *no*

such table: Movie error message.

Build the app to verify that there are no errors.

Initial migration

Use the EF Core Migrations feature to create the database. *Migrations* is a set of tools that create and update a database to match the data model.

Visual Studio

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

In the Package Manager Console (PMC), enter the following command:

```
PowerShell

Add-Migration InitialCreate
```

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

The following warning is displayed, which is addressed in a later step:

> 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'.

In the PMC, enter the following command:

```
PowerShell
```

```
Update-Database
```

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

For more information on the PMC tools for EF Core, see EF Core tools reference - PMC in Visual Studio.

Test the app

Visual Studio

Run the app and select the **Movie App** link.

If you get an exception similar to the following, you may have missed the `Update-Database` command in the migrations step:

Console

```
SqlException: Cannot open database "MvcMovieContext-1" requested by the
login. The login failed.
```

Note

You may not be able to enter decimal commas in the `Price` field. To support jQuery validation for non-English locales that use a comma (",") for a decimal point and for non US-English date formats, the app must be globalized. For globalization instructions, see this GitHub issue.

Examine the generated database context class and registration

With EF Core, data access is performed using a model. A model is made up of entity classes and a context object that represents a session with the database. The context object allows querying and saving data. The database context is derived from

Microsoft.EntityFrameworkCore.DbContext and specifies the entities to include in the data model.

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

C#

```
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.

Dependency injection

ASP.NET Core is built with dependency injection (DI). Services, such as the database context, are registered with DI in `Program.cs`. These services are provided to components that require them via constructor parameters.

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.

Scaffolding generated the following highlighted code in `Program.cs`:

Visual Studio

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<MvcMovieContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("MvcMovie
Context") ?? throw new InvalidOperationException("Connection string
'MvcMovieContext' not found.")));
```

The ASP.NET Core configuration system reads the "MvcMovieContext" database connection string.

Examine the generated database connection string

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

Visual Studio

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "MvcMovieContext": "Server=
(localdb)\\mssqllocaldb;Database=MvcMovieContext-4ebefa10-de29-4dea-
b2ad-8a8dc6bcf374;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

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

The `InitialCreate` class

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

C#

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

```
#nullable disable

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

        /// <inheritdoc />
        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.

Dependency injection in the controller

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

```
C#
```

```
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

Earlier in this tutorial, you saw how a controller can pass data or objects to a view using the `ViewData` dictionary. The `ViewData` dictionary is a dynamic object that provides a convenient late-bound way to pass information to a view.

MVC provides the ability to pass strongly typed model objects to a view. This strongly typed approach enables compile time code checking. The scaffolding mechanism passed a strongly typed model in the `MoviesController` class and views.

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

C#

```
// GET: Movies/Details/5
public async Task<IActionResult> Details(int? id)
{
    if (id == 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.

C#

```
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:

C#

```
return View(movie);
```

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

CSHTML

```
@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)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.ReleaseDate)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.ReleaseDate)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Genre)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Genre)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Price)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Price)
        </dd>
    </dl>
</div>
<div>
    <a asp-action="Edit" asp-route-id="@Model.Id">Edit</a> |
    <a asp-action="Index">Back to List</a>
</div>
```

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:

CSHTML

```
@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:

C#

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

The code returns problem details if the `Movie` property of the data context is null.

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

CSHTML

```
@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:

CSHTML

```
@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.

Additional resources

- Entity Framework Core for Beginners
- Tag Helpers
- Globalization and localization

Previous: Adding a View | Next: Working with SQL

Part 5, work with a database in an ASP.NET Core MVC app

Article • 09/10/2024

> **ⓘ Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By Rick Anderson and Jon P Smith.

The `MvcMovieContext` object handles the task of connecting to the database and mapping `Movie` objects to database records. The database context is registered with the Dependency Injection container in the `Program.cs` file:

Visual Studio

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<MvcMovieContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("MvcMovie
Context") ?? throw new InvalidOperationException("Connection string
'MvcMovieContext' not found.")));
```

The ASP.NET Core Configuration system reads the `ConnectionString` key. For local development, it gets the connection string from the `appsettings.json` file:

JSON

```
"ConnectionStrings": {
  "MvcMovieContext": "Server=
(localdb)\\mssqllocaldb;Database=MvcMovieContext-4ebefa10-de29-4dea-
b2ad-8a8dc6bcf374;Trusted_Connection=True;MultipleActiveResultSets=true"
}
```

> **⚠ Warning**
>
> This article uses a local database that doesn't require the user to be authenticated. Production apps should use the most secure authentication flow available. For more information on authentication for deployed test and production apps, see Secure authentication flows.

Visual Studio

SQL Server Express LocalDB

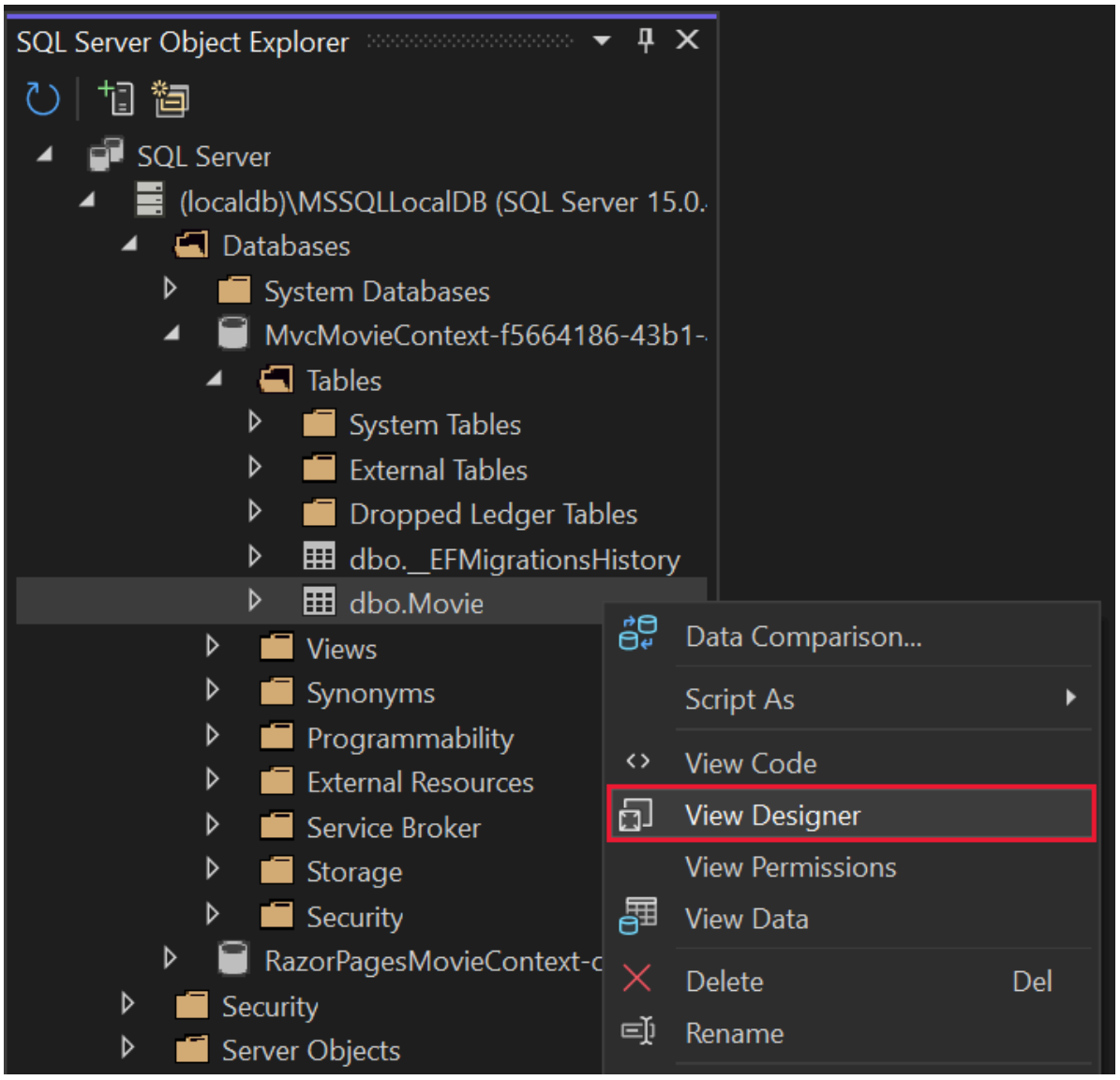
LocalDB:

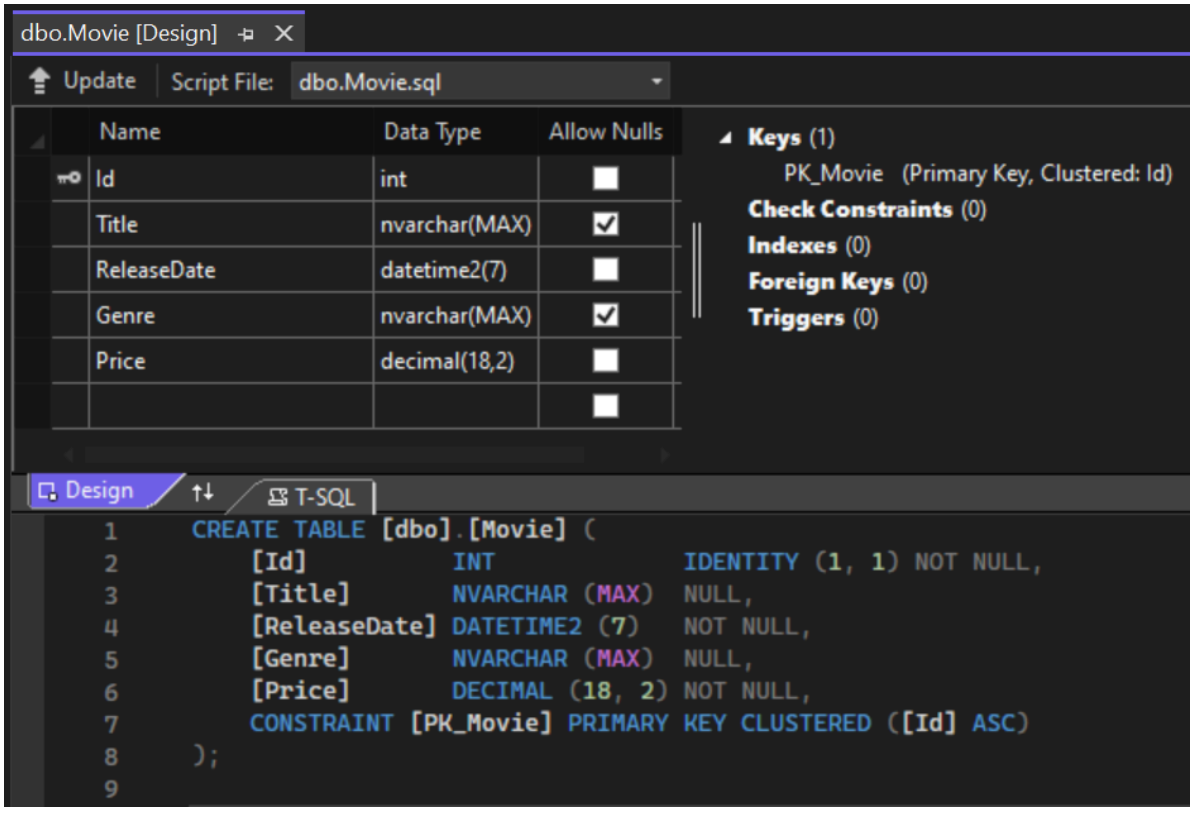
- Is a lightweight version of the SQL Server Express Database Engine, installed by default with Visual Studio.
- Starts on demand by using a connection string.
- Is targeted for program development. It runs in user mode, so there's no complex configuration.
- By default creates *.mdf* files in the *C:/Users/{user}* directory.

Examine the database

From the **View** menu, open **SQL Server Object Explorer** (SSOX).

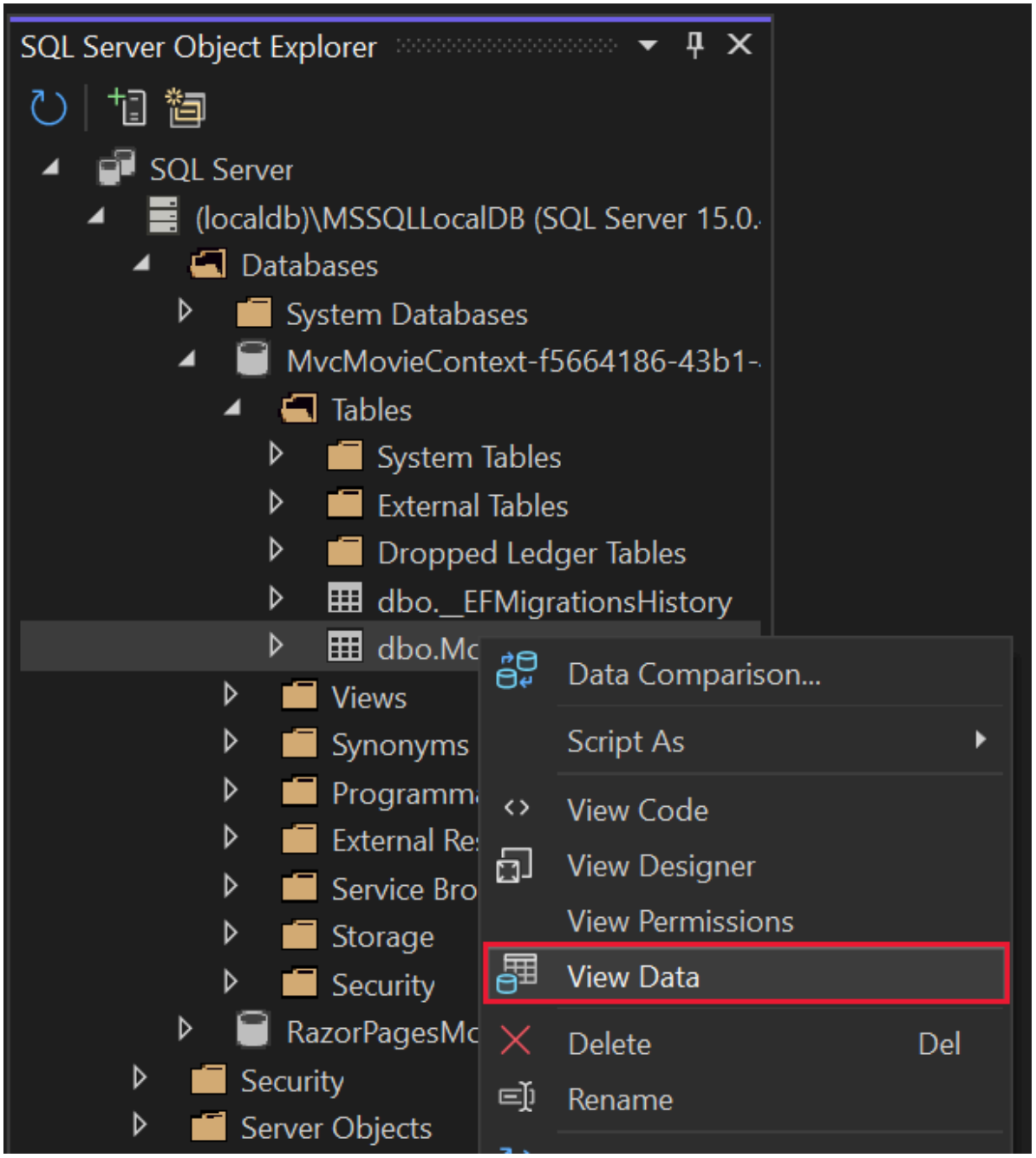
Right-click on the `Movie` table (`dbo.Movie`) > **View Designer**

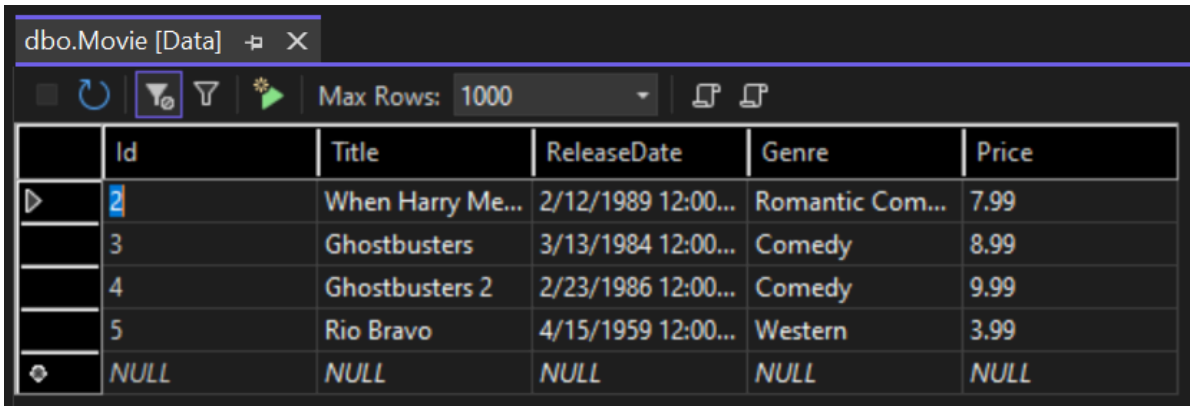

SQL Server Object Explorer
SQL Server
(localdb)\MSSQLLocalDB (SQL Server 15.0.
Databases
System Databases
MvcMovieContext-f5664186-43b1-
Tables
System Tables
External Tables
Dropped Ledger Tables
dbo.__EFMigrationsHistory
dbo.Movie
Views
Synonyms
Programmability
External Resources
Service Broker
Storage
Security
RazorPagesMovieContext-c
Security
Server Objects
Data Comparison...
Script As
View Code
View Designer
View Permissions
View Data
Delete
Del
Rename


dbo.Movie [Design]
Update
Script File: dbo.Movie.sql
Name
Data Type
Allow Nulls
Id
int
Title
nvarchar(MAX)
ReleaseDate
datetime2(7)
Genre
nvarchar(MAX)
Price
decimal(18,2)
Keys (1)
PK_Movie (Primary Key, Clustered: Id)
Check Constraints (0)
Indexes (0)
Foreign Keys (0)
Triggers (0)
Design
T-SQL
CREATE TABLE [dbo].[Movie] (
[Id] INT IDENTITY (1, 1) NOT NULL,
[Title] NVARCHAR (MAX) NULL,
[ReleaseDate] DATETIME2 (7) NOT NULL,
[Genre] NVARCHAR (MAX) NULL,
[Price] DECIMAL (18, 2) NOT NULL,
CONSTRAINT [PK_Movie] PRIMARY KEY CLUSTERED ([Id] ASC)
);

Note the key icon next to `ID`. By default, EF makes a property named `ID` the primary key.

Right-click on the `Movie` table > **View Data**





Seed the database

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

C#

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

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.

C#

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

Add the seed initializer

Visual Studio

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

C#

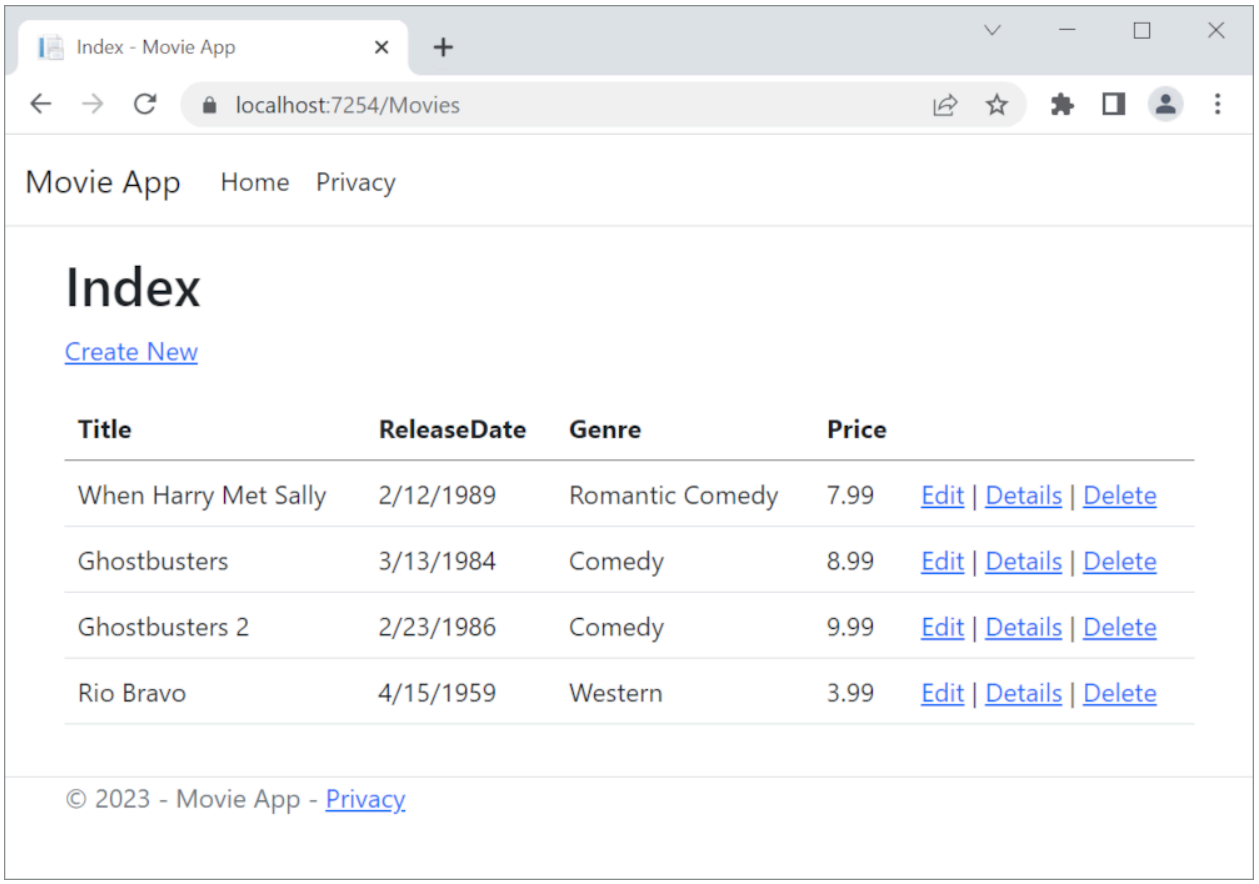
```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using MvcMovie.Data;
using MvcMovie.Models;
var builder = WebApplication.CreateBuilder(args);
```

```
builder.Services.AddDbContext<MvcMovieContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("MvcMovie
Context") ?? 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.UseRouting();

app.UseAuthorization();

app.MapStaticAssets();

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

app.Run();
```

Delete all the records in the database. You can do this with the delete links in the browser or from SSOX.

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, and restart by pressing Ctrl+F5.

The app shows the seeded data.



Previous: Adding a model
Next: Adding controller methods and views

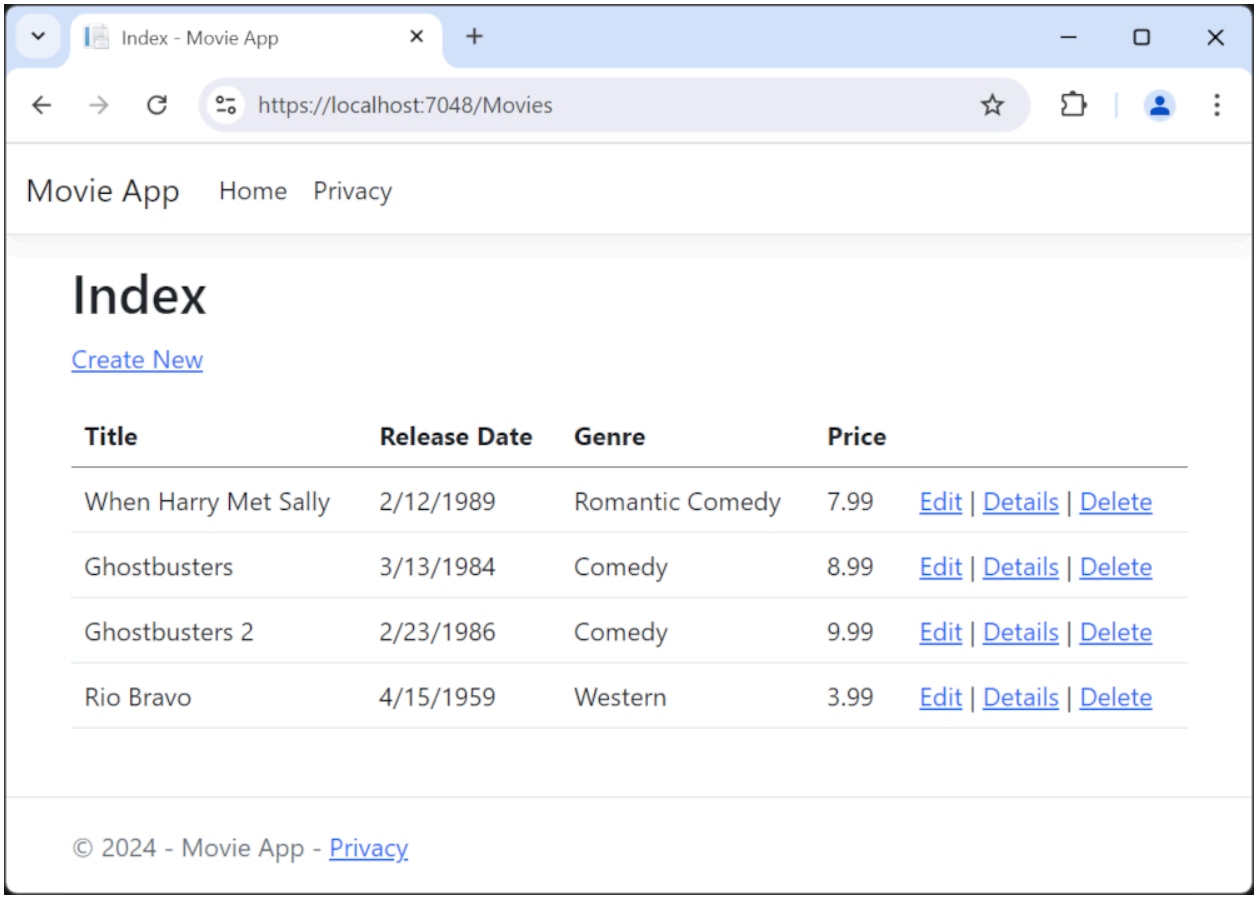
Part 6, controller methods and views in ASP.NET Core

Article • 09/18/2024

> **ⓘ Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By Rick Anderson

We have a good start to the movie app, but the presentation isn't ideal, for example, **ReleaseDate** should be two words.



Open the `Models/Movie.cs` file and add the highlighted lines shown below:

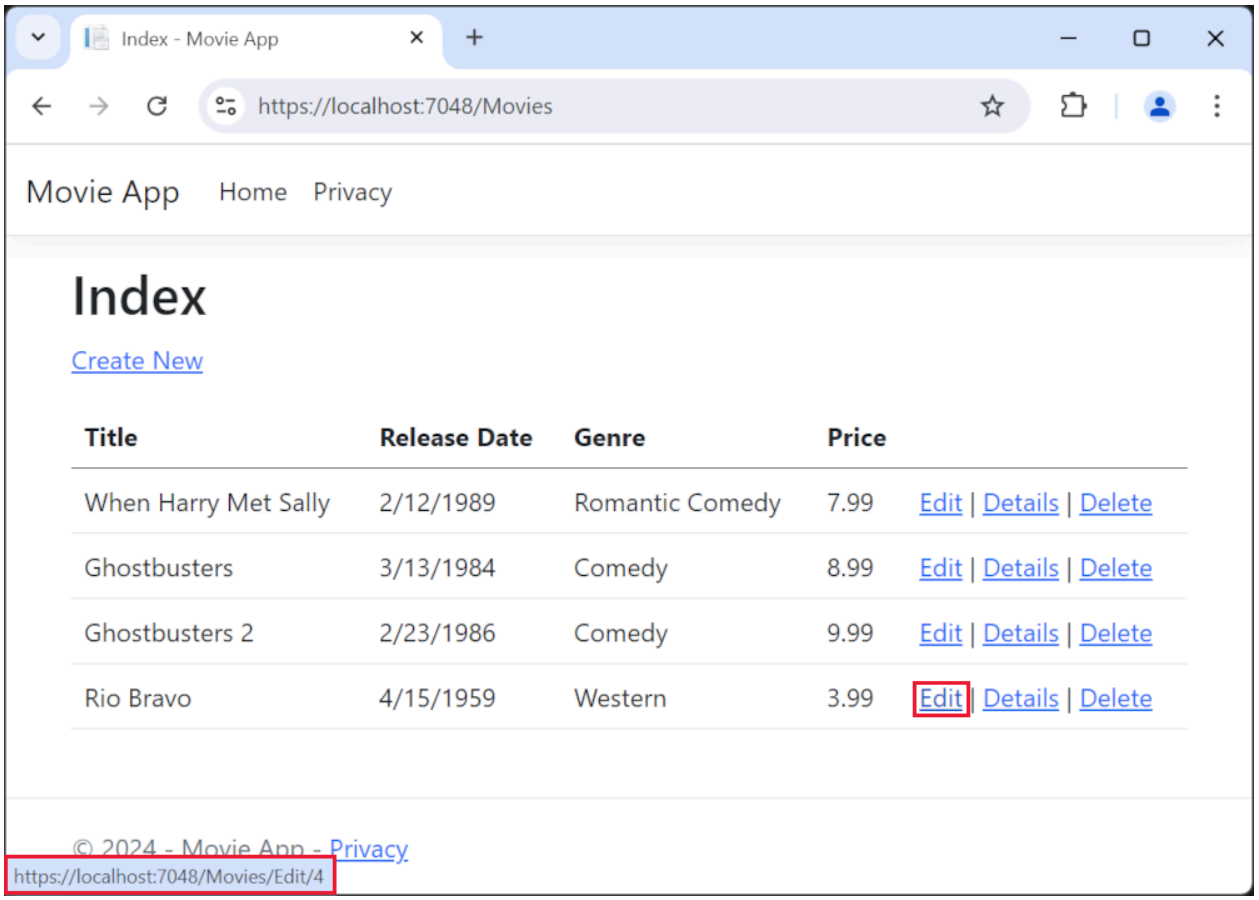
```
C#
```

```
using System;
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)]
    public DateTime ReleaseDate { get; set; }
    public string? Genre { get; set; }
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }
}
```

`DataAnnotations` are explained in the next tutorial. 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. For more information, see Data Types.

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.

CSHTML

```
            <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>
```

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:

HTML

```
 <td>
    <a href="/Movies/Edit/4"> Edit </a> |
    <a href="/Movies/Details/4"> Details </a> |
    <a href="/Movies/Delete/4"> Delete </a>
</td>
```

Recall the format for routing set in the `Program.cs` file:

C#

```
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 one of the most popular new features in ASP.NET Core. For more information, see Additional resources.

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.

C#

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

C#

```
// 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,Rating")] 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. For more information, see Protect your controller from over-posting. ViewModels provide an alternative approach to prevent over-posting.

Notice the second `Edit` action method is preceded by the `[HttpPost]` attribute.

C#

```
// 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,Rating")] 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 `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 antiforgery token generated in the edit view file (`Views/Movies/Edit.cshtml`). The edit view file generates the antiforgery token with the Form Tag Helper.

CSHTML

```
<form asp-action="Edit">
```

The Form Tag Helper generates a hidden antiforgery token that must match the `[ValidateAntiForgeryToken]` generated antiforgery token in the `Edit` method of the Movies controller. For more information, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core.

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.

C#

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

CSHTML

```
@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.

HTML

```
<form action="/Movies/Edit/7" method="post">
    <div class="form-horizontal">
        <h4>Movie</h4>
        <hr />
        <div class="text-danger" />
        <input type="hidden" data-val="true" data-val-required="The ID field
is required." id="ID" name="ID" value="7" />
        <div class="form-group">
            <label class="control-label col-md-2" for="Genre" />
            <div class="col-md-10">
```

```
                <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>
        <div class="form-group">
            <label class="control-label col-md-2" for="Price" />
            <div class="col-md-10">
                <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>
        <!-- Markup removed for brevity -->
        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </div>
    </div>
    <input name="__RequestVerificationToken" type="hidden"
value="CfDJ8Inyxgp63fRFqUePGvuI5jGZsloJu1L7X9le1gy7NCIlSduCRx9jDQClrV9pOTTmq
UyXnJBXhmrjcUVDJyDUMm7-
MF_9rK8aAZdRdlOri7FmKVkRe_2v5LIHGKFcTjPrWPYnc9AdSbomkiOSaTEg7RU" />
</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 XSRF token generated by the Form Tag Helper.

Processing the POST Request

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

C#

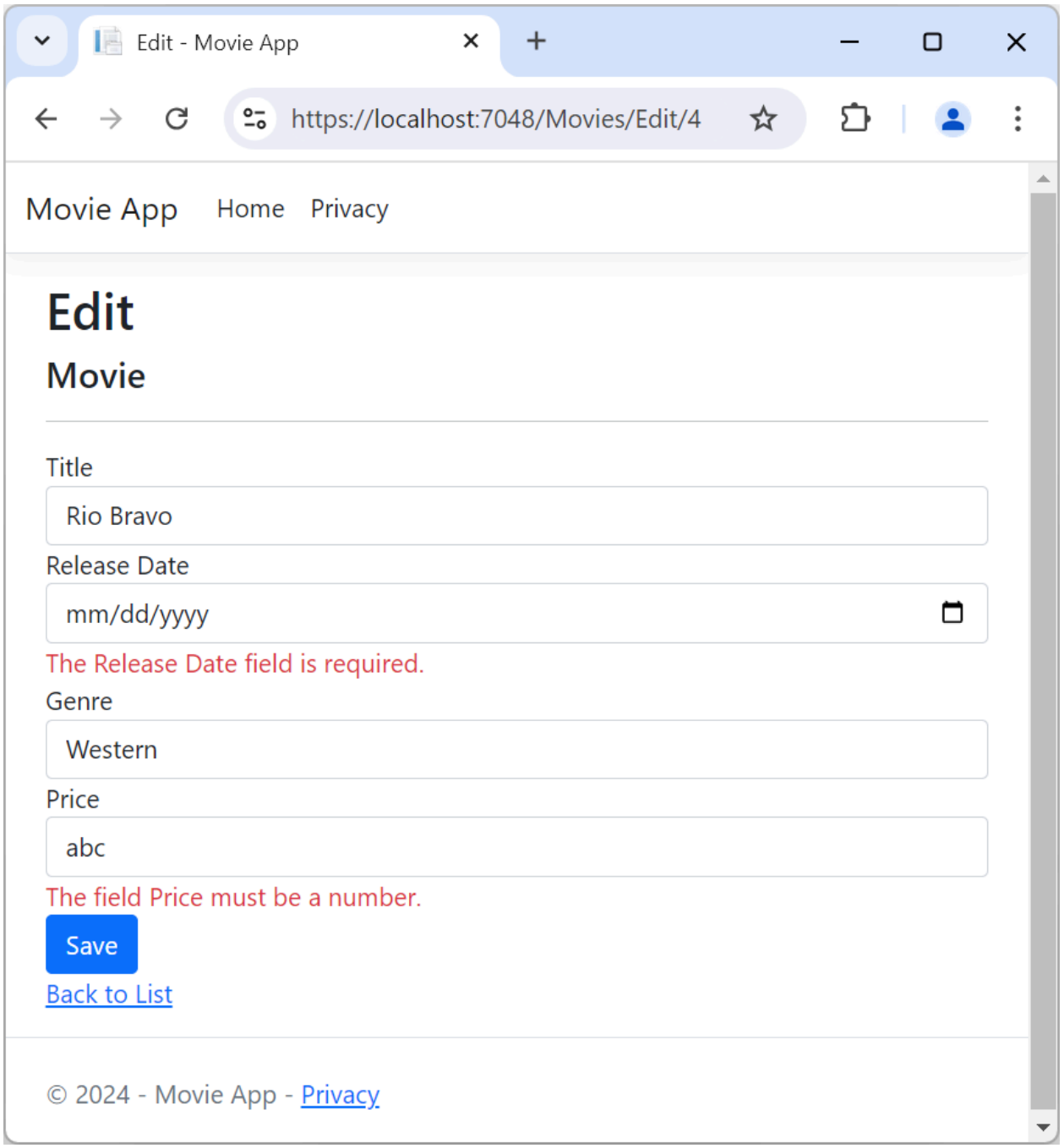
```
// POST: Movies/Edit/5
// To protect from overposting attacks, enable the specific properties you
want to bind to.
// For more details, see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id,
[Bind("Id,Title,ReleaseDate,Genre,Price,Rating")] 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 `[ValidateAntiForgeryToken]` attribute validates the hidden XSRF token generated by the antiforgery 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 tutorial 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.

Additional resources

- Globalization and localization
- Introduction to Tag Helpers
- Author Tag Helpers
- Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core
- Protect your controller from over-posting
- ViewModels
- Form Tag Helper
- Input Tag Helper
- Label Tag Helper
- Select Tag Helper
- Validation Tag Helper

Previous Next

Part 7, add search to an ASP.NET Core MVC app

Article • 09/18/2024

> **ⓘ Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By Rick Anderson

In this section, 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:

C#

```
public async Task<IActionResult> Index(string searchString)
{
    if (_context.Movie == null)
    {
        return Problem("Entity set 'MvcMovieContext.Movie'  is null.");
    }

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

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

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

The following line in the `Index` action method creates a LINQ query to select the movies:

C#

```
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:

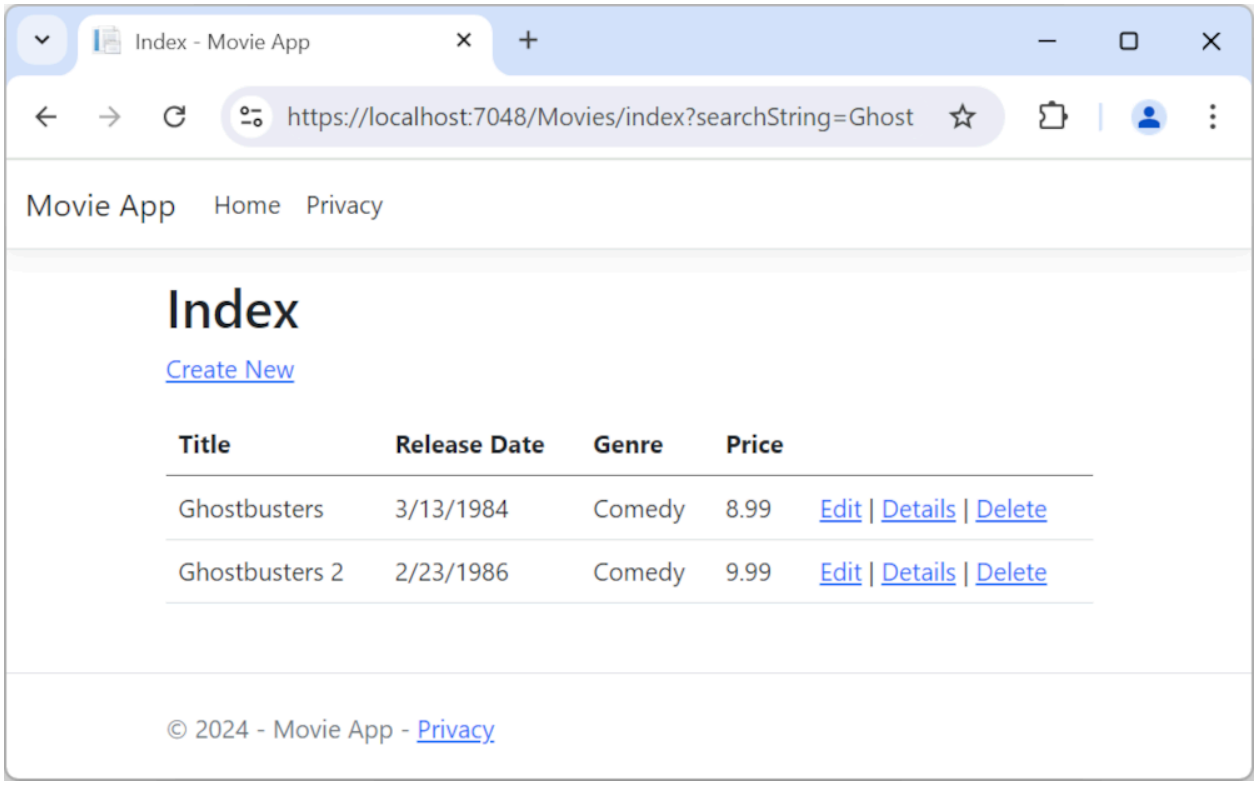
C#

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

The `s => s.Title!.ToUpper().Contains(searchString.ToUpper())` 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. For more information about deferred query execution, see Query Execution.

> ⓘ Note
>
> The Contains method is run on the database, not in the C# code. 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. SQLite with the default collation is a mixture of case sensitive and case ***IN***sensitive, depending on the query. For information on making case insensitive SQLite queries, see the following:
>
> - How to use case-insensitive query with Sqlite provider? (dotnet/efcore #11414)
> - How to make a SQLite column case insensitive (dotnet/AspNetCore.Docs #22314)
> - Collations and Case Sensitivity

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`.

C#

```
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:

C#

```
public async Task<IActionResult> Index(string searchString)
{
    if (_context.Movie == null)
    {
        return Problem("Entity set 'MvcMovieContext.Movie'  is null.");
    }

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

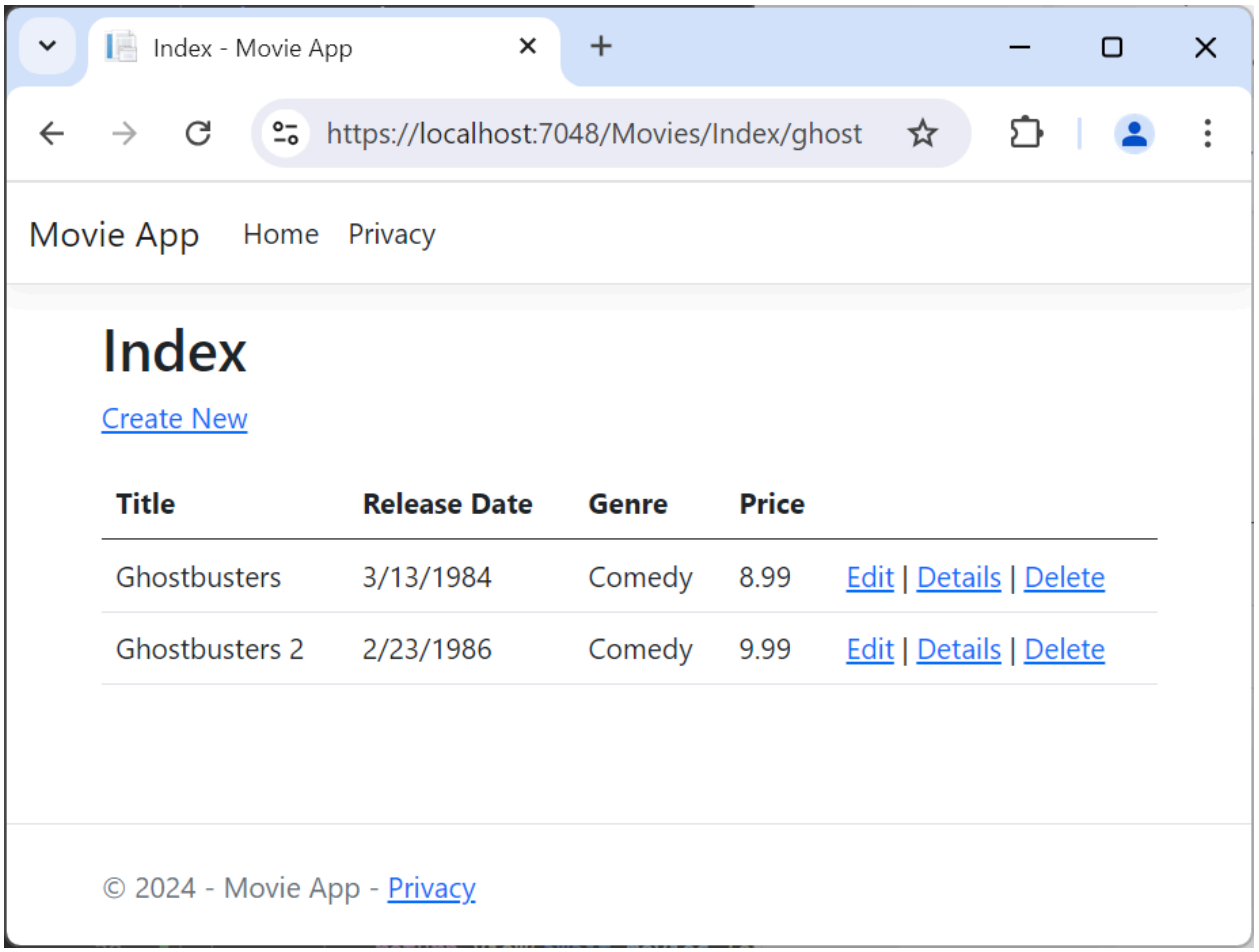
```
    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s =>
s.Title!.ToUpper().Contains(searchString.ToUpper()));
    }

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

The updated `Index` method with `id` parameter:

C#

```
public async Task<IActionResult> Index(string id)
{
    if (_context.Movie == null)
    {
        return Problem("Entity set 'MvcMovieContext.Movie'  is null.");
    }

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

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

    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`:

C#

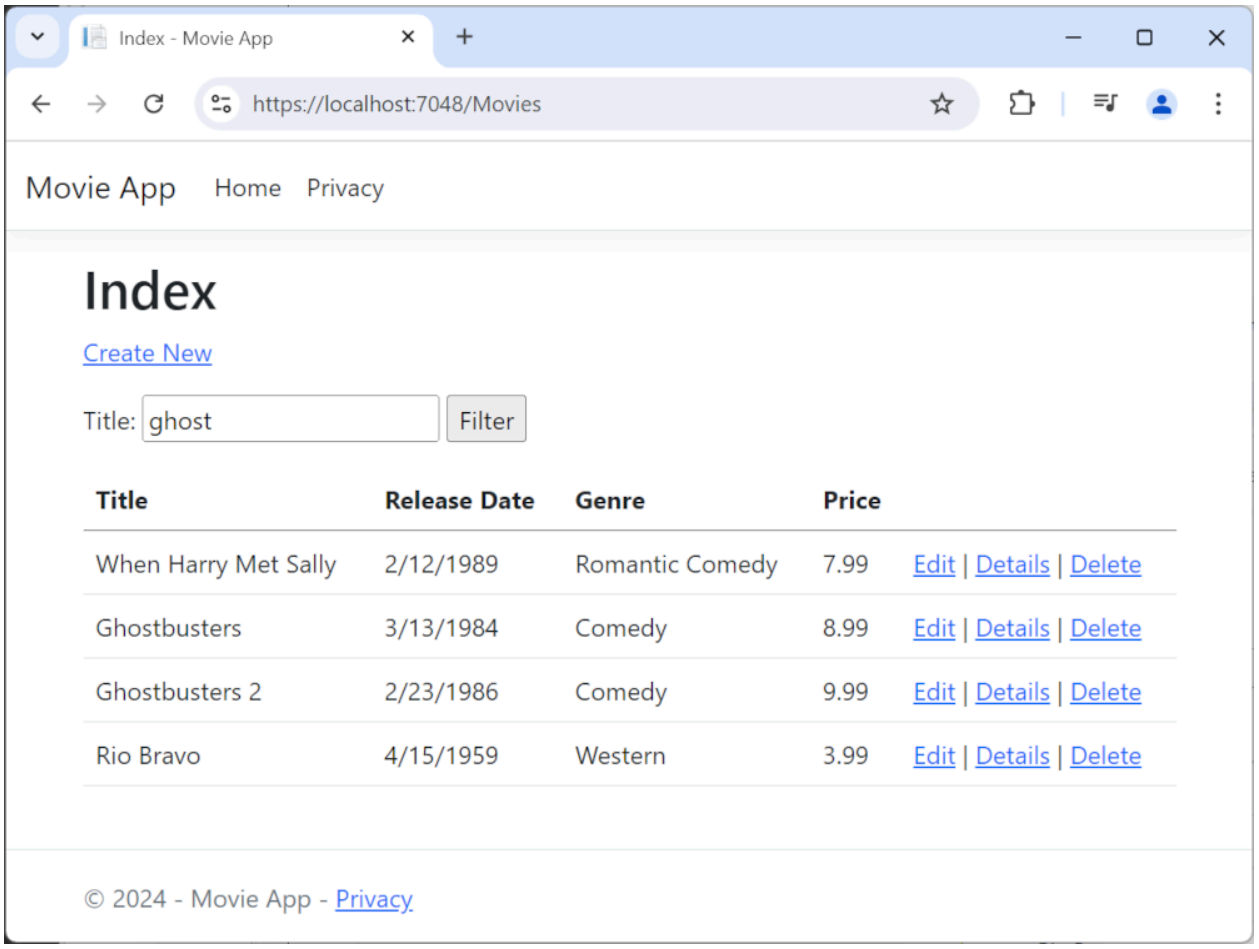
```
public async Task<IActionResult> Index(string searchString)
{
    if (_context.Movie == null)
    {
        return Problem("Entity set 'MvcMovieContext.Movie'  is null.");
    }

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

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

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

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

CSHTML

```
@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>
        <label>Title: <input type="text" name="SearchString" /></label>
        <input type="submit" value="Filter" />
    </p>
</form>
<table class="table">
```

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.

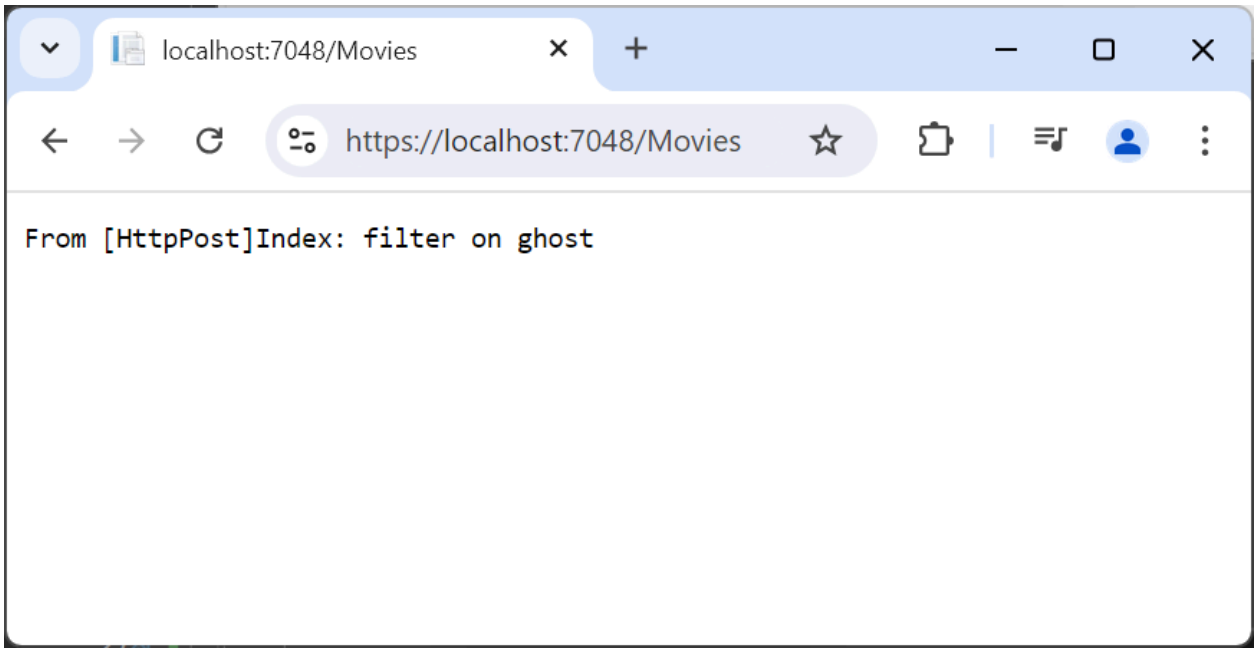
You could add the following `[HttpPost] Index` method.

C#

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

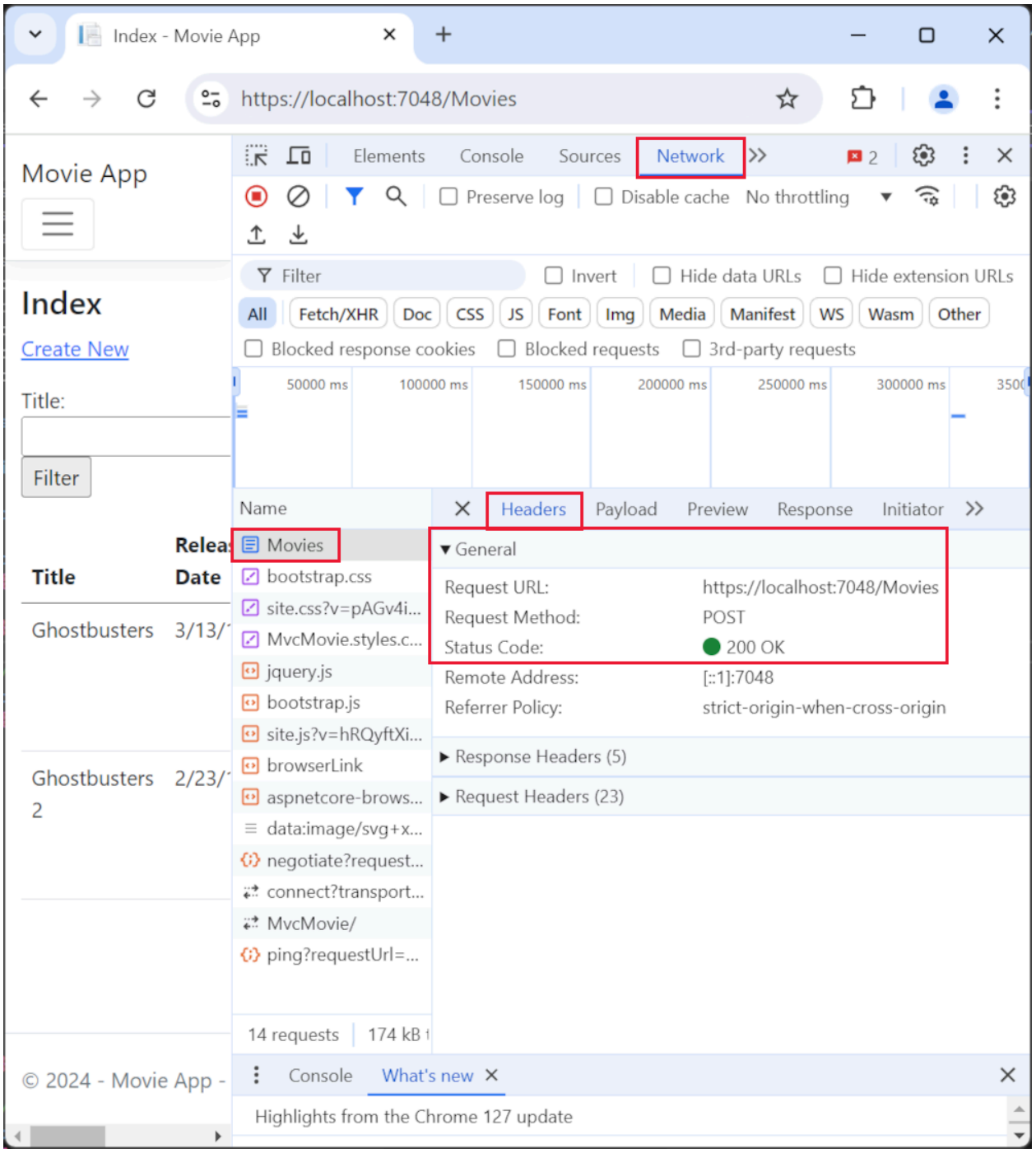
The `notUsed` parameter is used to create an overload for the `Index` method. We'll talk about that later in the tutorial.

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.

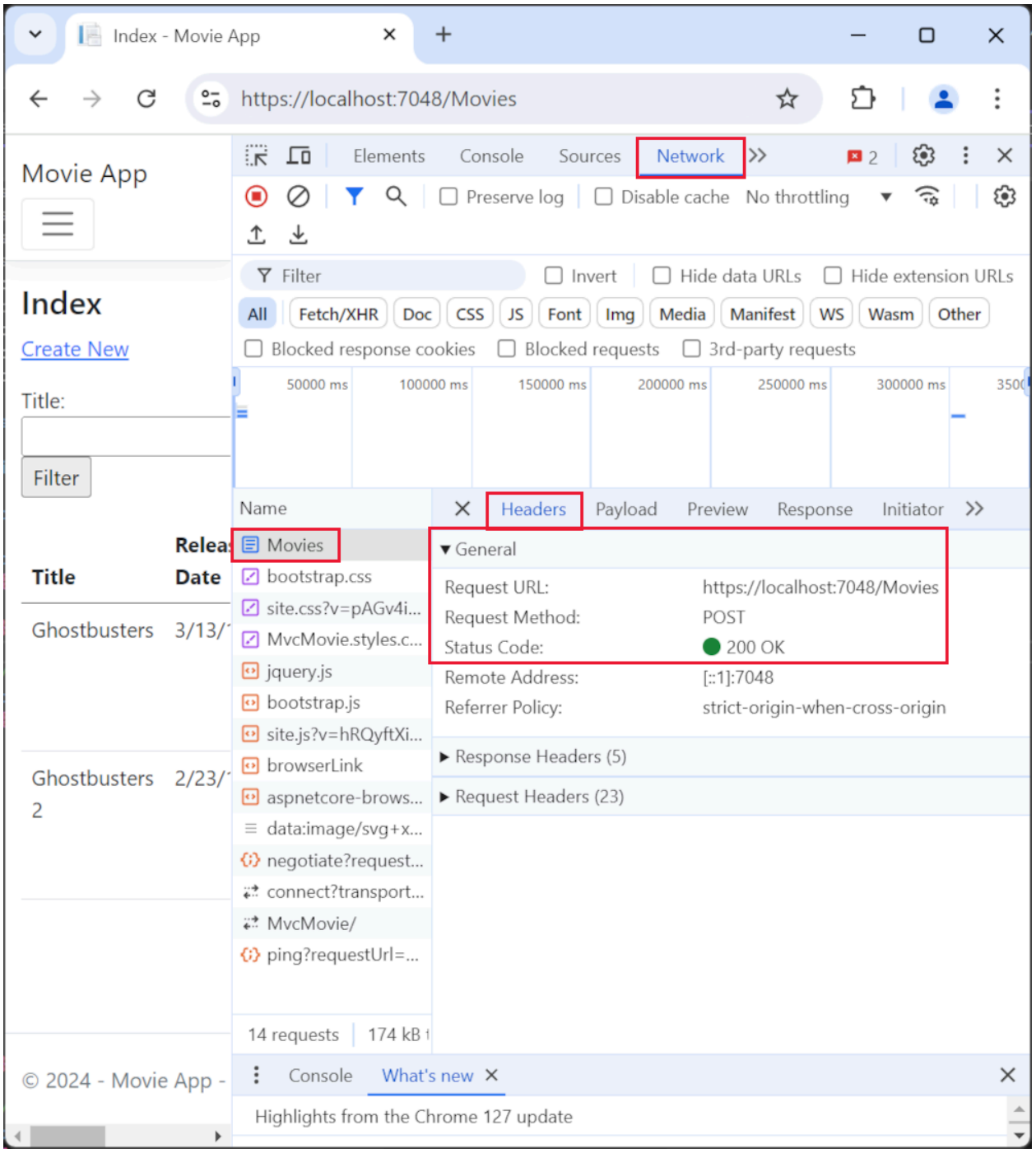


However, even if you add this `[HttpPost]` version of the `Index` method, there's a limitation in how this has all been implemented. Imagine that you want to bookmark a particular search or you want to send a link to friends that they can click in order to see the same filtered list of movies. Notice 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 following image shows the Chrome browser Developer tools with the **Network** and **Headers** tabs selected:



The **Network** and **Payload** tabs are selected to view form data:



You can see the search parameter and XSRF token in the request body. Note, as mentioned in the previous tutorial, the Form Tag Helper generates an XSRF antiforgery token. We're not modifying data, so we don't need to validate the token in the controller method.

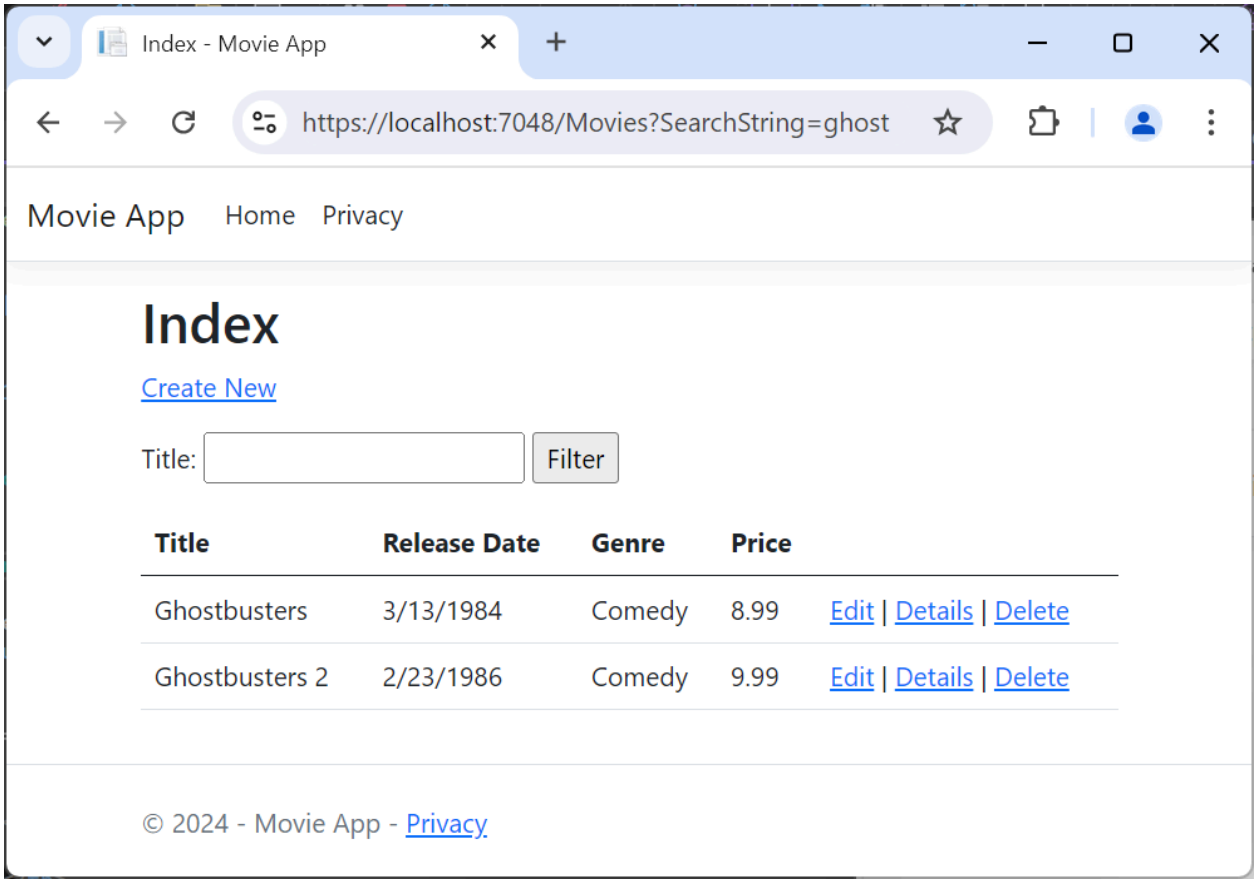
Because the search parameter is in the request body and not the URL, you can't capture that search information to bookmark or share with others. Fix this by specifying the request should be `HTTP GET` in the `form` tag found in the `Views/Movies/Index.cshtml` file.

CSHTML

```
@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" method="get">
    <p>
        <label>Title: <input type="text" name="SearchString" /></label>
        <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.



Add Search by genre

Add the following `MovieGenreViewModel` class to the *Models* folder:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

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:

C#

```
// GET: Movies
public async Task<IActionResult> Index(string movieGenre, string
searchString)
{
    if (_context.Movie == null)
    {
        return Problem("Entity set 'MvcMovieContext.Movie'  is null.");
    }

    // 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!.ToUpper().Contains(searchString.ToUpper()));
    }

    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.

C#

```
// 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.

Add search by genre to the Index view

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

CSHTML

```
@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" method="get">
    <p>

        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>
```

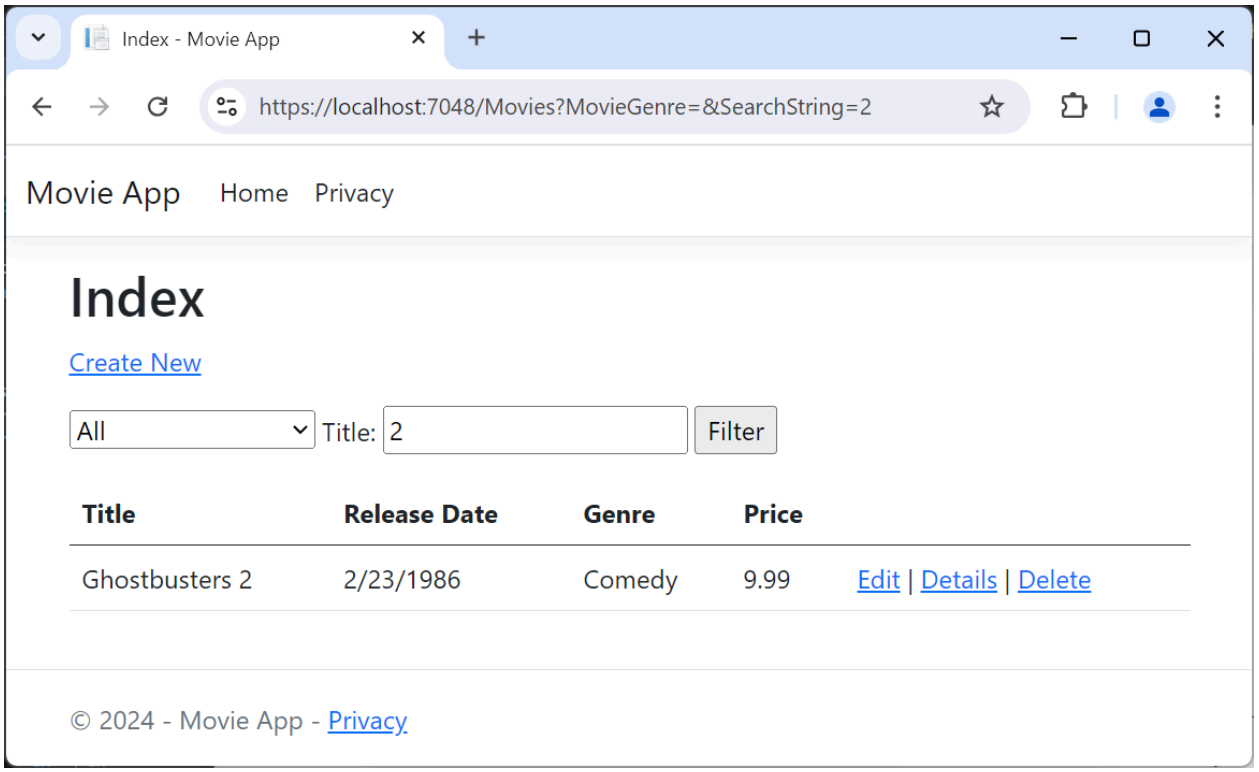
```
        <label>Title: <input type="text" asp-for="SearchString" /></label>
        <input type="submit" value="Filter" />
    </p>
</form>

<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 preceding code, the `DisplayNameFor` HTML Helper inspects the `Title` property referenced in the lambda expression to determine the display name. Since the lambda expression is inspected rather than evaluated, you don't receive an access violation when `model`, `model.Movies`, or `model.Movies[0]` are `null` or empty. When the lambda expression is evaluated (for example, `@Html.DisplayFor(modelItem => item.Title)`), the model's property values are evaluated. The `!` after `model.Movies` is the null-forgiving operator, which is used to declare that `Movies` isn't null.

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



Previous Next

Part 8, add a new field to an ASP.NET Core MVC app

Article • 07/30/2024

> **ⓘ Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By Rick Anderson

In this section Entity Framework Migrations is used to:

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

When Entity Framework (EF) is used to automatically create a database from model classes:

- A table is added to the database to track the schema of the database.
- The database is verified to be 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 the Movie Model

Add a `Rating` property to `Models/Movie.cs`:

C#

```
using System;
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)]
    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

Visual Studio

Press Ctrl+Shift+B

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:

C#

```
[Bind("Id,Title,ReleaseDate,Genre,Price,Rating")]
```

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:

CSHTML

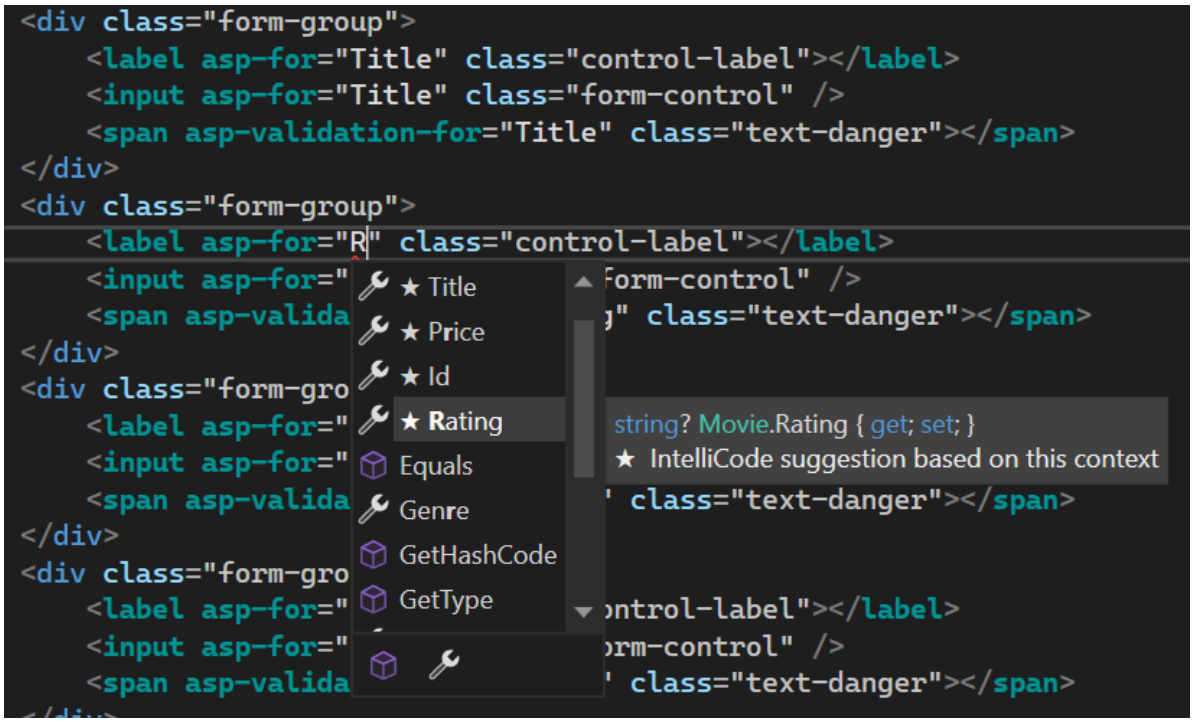
```
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movies![0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies![0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => 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.

Visual Studio

You can copy/paste the previous "form group" and let intelliSense help you update the fields. IntelliSense works with Tag Helpers.



Add the `Rating` property to the remaining `Create.cshtml`, `Delete.cshtml`, `Details.cshtml`, and `Edit.cshtml` view 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`.

C#

```
new Movie
{
    Title = "When Harry Met Sally",
    ReleaseDate = DateTime.Parse("1989-1-11"),
    Genre = "Romantic Comedy",
    Rating = "R",
    Price = 7.99M
},
```

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:

```
SqlException: Invalid column name 'Rating'.
```

This error occurs because the updated Movie model class is different than the schema of the Movie table of the existing database. (There's no `Rating` column in the database table.)

There are a few approaches to resolving the error:

1. Have the Entity Framework automatically drop and re-create the database based on the new model class schema. This approach is very convenient early in the development cycle when you're doing active development on a test database; it allows you to quickly evolve the model and database schema together. The downside, though, is that you lose existing data in the database — so you don't want to use this approach on a production database! Using an initializer to automatically seed a database with test data is often a productive way to develop an application. This is a good approach for early development and when using SQLite.

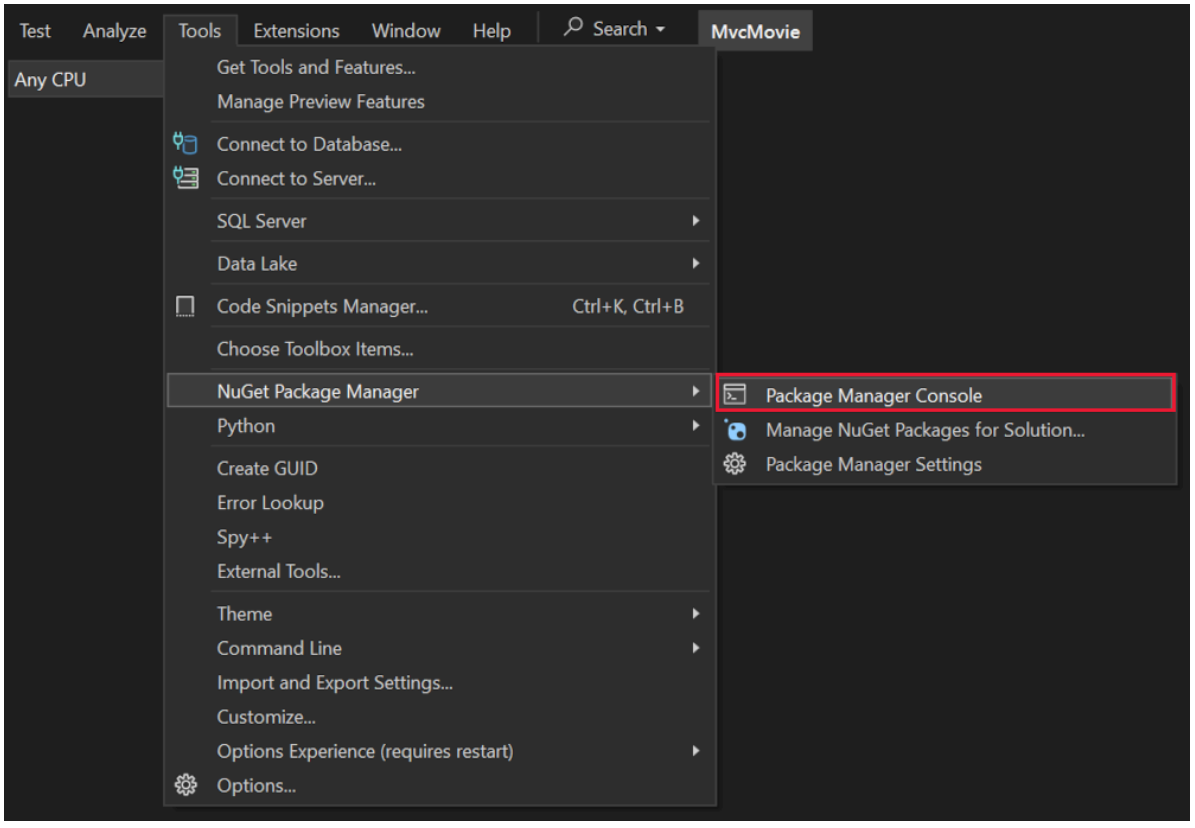
2. Explicitly modify the schema of the existing database so that it matches the model classes. The advantage of this approach is that you keep your data. You can make this change either manually or by creating a database change script.

3. Use Entity Framework Migrations to update the database schema.

For this tutorial, Entity Framework Migrations is used.

Visual Studio

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



In the Package Manager Console, enter the following command:

```
PowerShell
```

```
Add-Migration Rating
```

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.

In the Package Manager Console, enter the following command:

PowerShell

```
Update-Database
```

The Update-Database command runs the Up method in migrations that have not been applied.

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

Previous Next

Part 9, add validation to an ASP.NET Core MVC app

Article • 07/30/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By Rick Anderson

In this section:

- 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.

Keeping things DRY

One of the design tenets of MVC is DRY ("Don't Repeat Yourself"). ASP.NET Core MVC encourages you to specify functionality or behavior only once, and then have it be reflected everywhere in an app. This reduces the amount of code you need to write and makes the code you do write less error prone, easier to test, and easier to maintain.

The validation support provided by MVC and Entity Framework Core is a good example of the DRY principle in action. You can declaratively specify validation rules in one place (in the model class) and the rules are enforced everywhere in the app.

Delete the previously edited data

In the next step, validation rules are added that don't allow null values. Run the app, navigate to `/Movies/Index`, delete all listed movies, and stop the app. The app will use the seed data the next time it is run.

Add validation rules to the movie model

The DataAnnotations namespace provides a set of built-in validation attributes that are applied declaratively to a class or property. DataAnnotations also contains formatting attributes like `DataType` that help with formatting and don't provide any validation.

Update the `Movie` class to take advantage of the built-in validation attributes `Required`, `StringLength`, `RegularExpression`, `Range` and the `DataType` formatting attribute.

C#

```
using System;
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)]
    public DateTime ReleaseDate { get; set; }

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

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

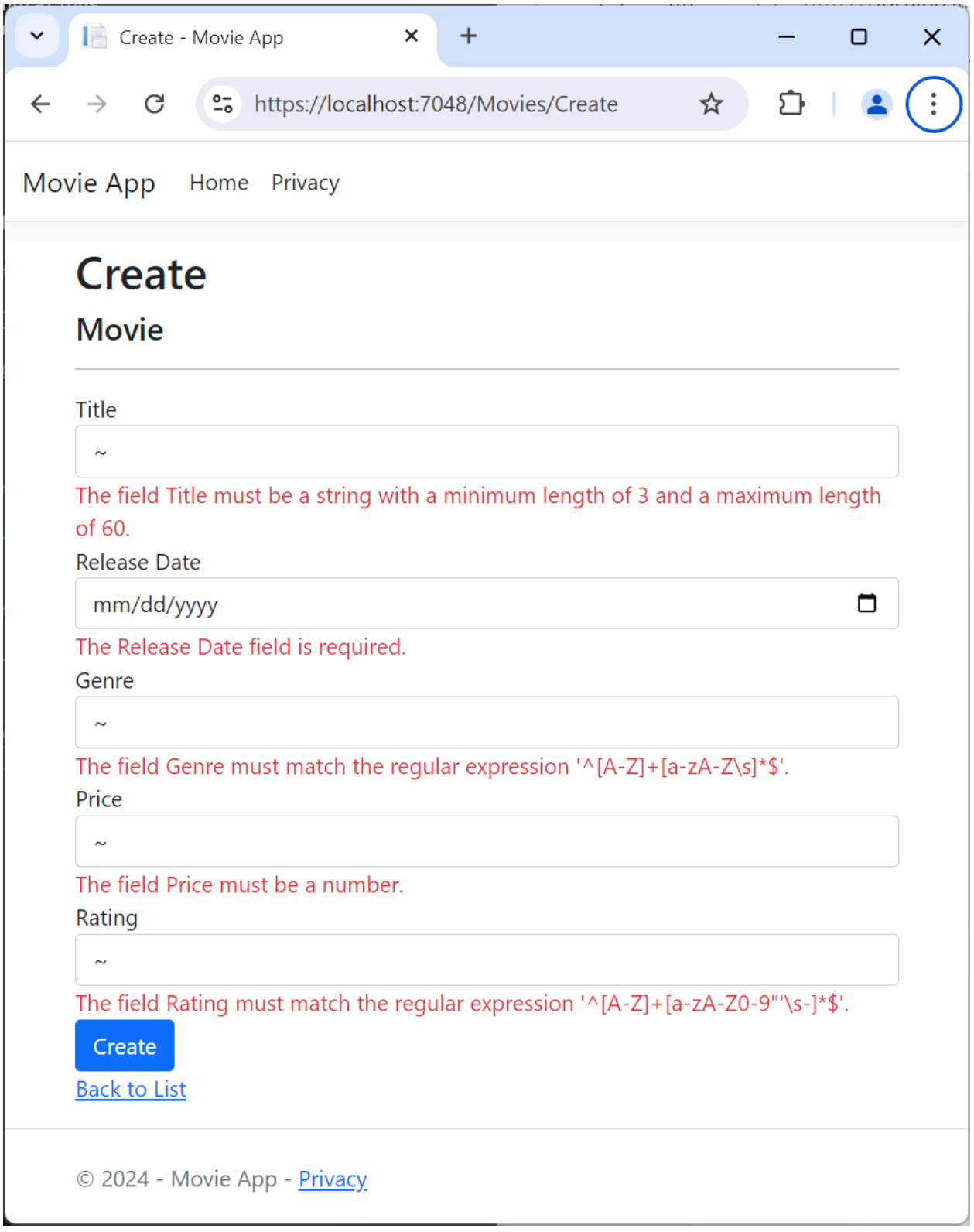
- Value types (such as `decimal`, `int`, `float`, `DateTime`) are inherently required and don't need the `[Required]` attribute.

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 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.



> **Note**
>
> You may not be able to enter decimal commas in decimal fields. To support jQuery validation for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize your app. See this GitHub comment 4076 for instructions on adding decimal comma.

Notice how the form has automatically rendered an appropriate validation error message in each field containing an invalid value. The errors are enforced both client-side (using JavaScript and jQuery) and server-side (in case a user has JavaScript disabled).

A significant benefit is that you didn't need to change a single line of code in the `MoviesController` class or in the `Create.cshtml` view in order to enable this validation UI. The controller and views you created earlier in this tutorial automatically picked up the validation rules that you specified by using validation attributes on the properties of the `Movie` model class. Test validation using the `Edit` action method, and the same validation is applied.

The form data isn't sent to the server until there are no client side validation errors. You can verify this by putting a break point in the `HTTP Post` method, by using the Fiddler tool, or the F12 Developer tools.

How validation works

You might wonder how the validation UI was generated without any updates to the code in the controller or views. The following code shows the two `Create` methods.

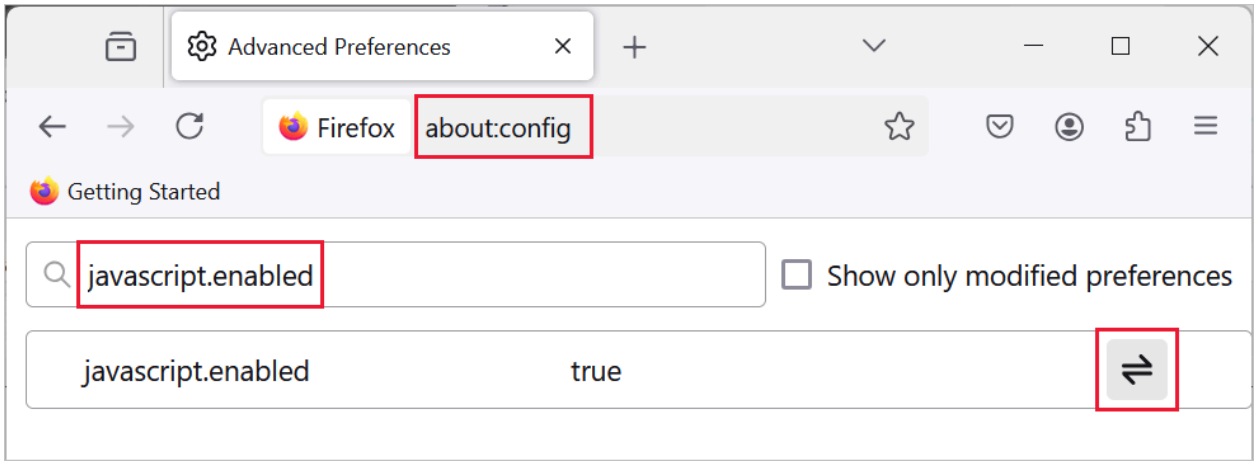
C#

```
// GET: Movies/Create
public IActionResult Create()
{
    return View();
}

// POST: Movies/Create
// 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>
Create([Bind("Id,Title,ReleaseDate,Genre,Price,Rating")] Movie movie)
{
    if (ModelState.IsValid)
    {
        _context.Add(movie);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    return View(movie);
}
```

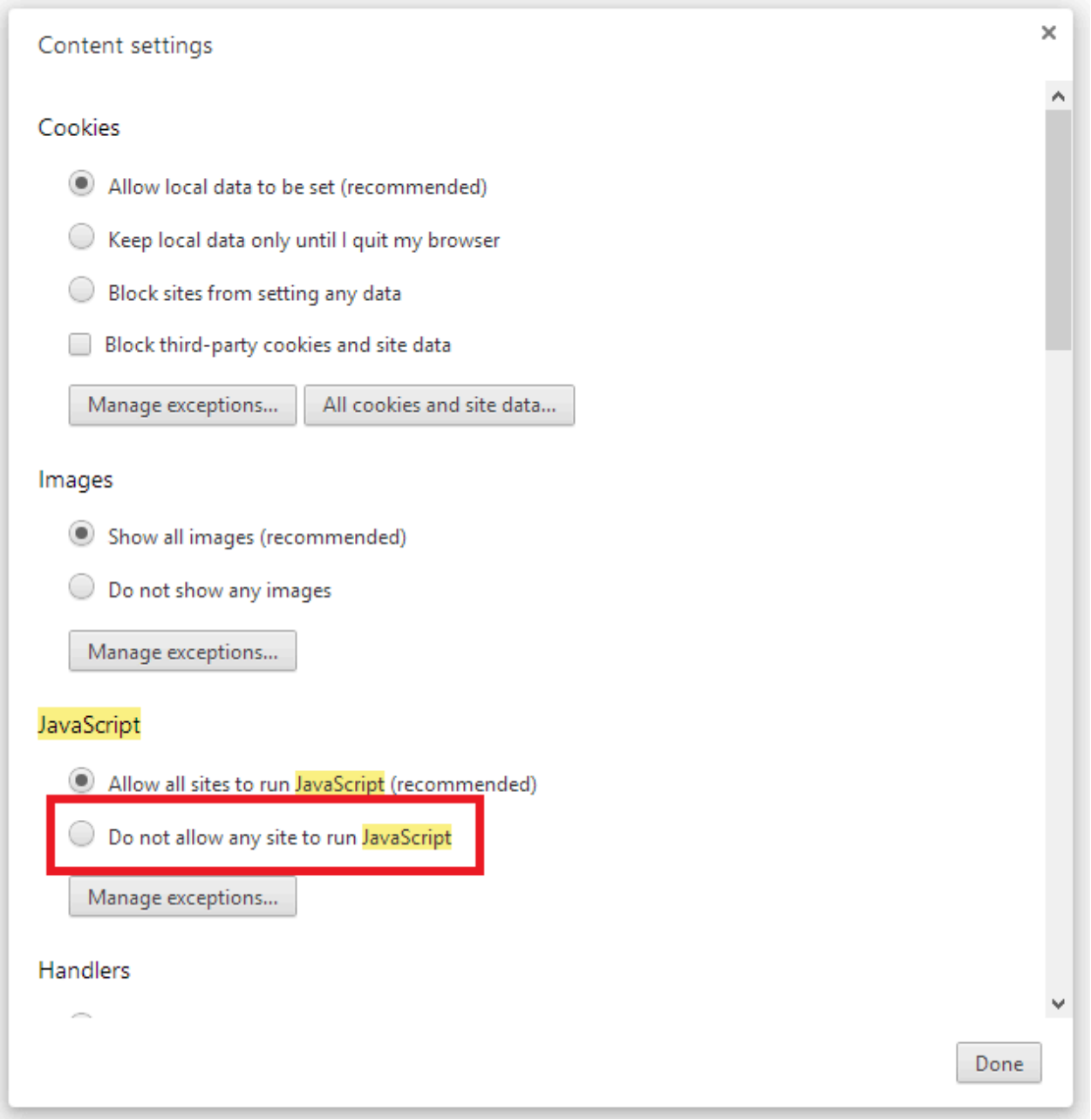
The first (HTTP GET) `Create` action method displays the initial Create form. The second (`[HttpPost]`) version handles the form post. The second `Create` method (The `[HttpPost]` version) calls `ModelState.IsValid` to check whether the movie has any validation errors. Calling this method evaluates any validation attributes that have been applied to the object. If the object has validation errors, the `Create` method re-displays the form. If there are no errors, the method saves the new movie in the database. In our movie example, the form isn't posted to the server when there are validation errors detected on the client side; the second `Create` method is never called when there are client side validation errors. If you disable JavaScript in your browser, client validation is disabled and you can test the HTTP POST `Create` method `ModelState.IsValid` detecting any validation errors.

You can set a break point in the `[HttpPost] Create` method and verify the method is never called, client side validation won't submit the form data when validation errors are detected. If you disable JavaScript in your browser, then submit the form with errors, the break point will be hit. You still get full validation without JavaScript.

The following image shows how to disable JavaScript in the Firefox browser.



The following image shows how to disable JavaScript in the Chrome browser.



After you disable JavaScript, post invalid data and step through the debugger.

```
74          // POST: Movies/Create
75          // To protect from overposting attacks, please enable t
76          // more details see http://go.microsoft.com/fwlink/?Lin
77          [HttpPost]
78          [ValidateAntiForgeryToken]
            0 references
79          public async Task<IAc          t> Create([Bind("ID,Title
80          {                              false
81              if (ModelState.IsValid)
82              {
83                  _context.Add(movie);
84                  await _context.SaveChangesAsync();
85                  return RedirectToAction("Index");
86              }
87              return View(movie);
88          }
89
```

A portion of the `Create.cshtml` view template is shown in the following markup:

HTML

```
<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>

            @*Markup removed for brevity.*@
```

The preceding markup is used by the action methods to display the initial form and to redisplay it in the event of an error.

The Input Tag Helper uses the DataAnnotations attributes and produces HTML attributes needed for jQuery Validation on the client side. The Validation Tag Helper displays validation errors. See Validation for more information.

What's really nice about this approach is that neither the controller nor the `Create` view template knows anything about the actual validation rules being enforced or about the specific error messages displayed. The validation rules and the error strings are specified only in the `Movie` class. These same validation rules are automatically applied to the `Edit` view and any other views templates you might create that edit your model.

When you need to change validation logic, you can do so in exactly one place by adding validation attributes to the model (in this example, the `Movie` class). You won't have to worry about different parts of the application being inconsistent with how the rules are enforced — all validation logic will be defined in one place and used everywhere. This keeps the code very clean, and makes it easy to maintain and evolve. And it means that you'll be fully honoring the DRY principle.

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.

C#

```
[Display(Name = "Release Date")]
[DataType(DataType.Date)]
public DateTime ReleaseDate { 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:

C#

```
[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).

> ⓘ **Note**
>
> jQuery validation doesn't work with the `Range` attribute and `DateTime`. For example, the following code will always display a client side validation error, even when the date is in the specified range:
>
> `[Range(typeof(DateTime), "1/1/1966", "1/1/2020")]`

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:

C#

```
using System;
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; }
}
```

In the next part of the series, we review the app and make some improvements to the automatically generated `Details` and `Delete` methods.

Additional resources

- Working with Forms
- Globalization and localization
- Introduction to Tag Helpers
- Author Tag Helpers

Previous Next

Part 10, examine the Details and Delete methods of an ASP.NET Core app

Article • 08/05/2024

> **ⓘ Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By Rick Anderson

Open the Movie controller and examine the `Details` method:

C#

```
// GET: Movies/Details/5
public async Task<IActionResult> Details(int? id)
{
    if (id == 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`.

C#

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

EF makes it easy to search for data using the `FirstOrDefaultAsync` method. An important security feature built into the method is that the code verifies that the search method has found a movie before it tries to do anything with it. For example, a hacker could introduce errors into the site by changing the URL created by the links from `http://localhost:{PORT}/Movies/Details/1` to something like `http://localhost:{PORT}/Movies/Details/12345` (or some other value that doesn't represent an actual movie). If you didn't check for a null movie, the app would throw an exception.

Examine the `Delete` and `DeleteConfirmed` methods.

C#

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

    if (id == 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)
{
    var movie = await _context.Movie.FindAsync(id);
    if (movie != null)
    {
        _context.Movie.Remove(movie);
    }

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

Note that the `HTTP GET Delete` method doesn't delete the specified movie, it returns a view of the movie where you can submit (HttpPost) the deletion. Performing a delete operation in response to a GET request (or for that matter, performing an edit operation, create operation, or any other operation that changes data) opens up a security hole.

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:

C#

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

C#

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

The common language runtime (CLR) requires overloaded methods to have a unique parameter signature (same method name but different list of parameters). However, here you need two `Delete` methods -- one for GET and one for POST -- that both have the same parameter signature. (They both need to accept a single integer as a parameter.)

There are two approaches to this problem, one is to give the methods different names. That's what the scaffolding mechanism did in the preceding example. However, this introduces a small problem: ASP.NET maps segments of a URL to action methods by name, and if you rename a method, routing normally wouldn't be able to find that method. The solution is what you see in the example, which is to add the `ActionName("Delete")` attribute to the `DeleteConfirmed` method. That attribute performs mapping for the routing system so that a URL that includes /Delete/ for a POST request will find the `DeleteConfirmed` method.

Another common work around for methods that have identical names and signatures is to artificially change the signature of the POST method to include an extra (unused) parameter. That's what we did in a previous post when we added the `notUsed` parameter. You could do the same thing here for the `[HttpPost] Delete` method:

C#

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

Publish to Azure

For information on deploying to Azure, see Tutorial: Build an ASP.NET Core and SQL Database app in Azure App Service.

Reliable web app patterns

See *The Reliable Web App Pattern for.NET* YouTube videos and article for guidance on creating a modern, reliable, performant, testable, cost-efficient, and scalable ASP.NET Core app, whether from scratch or refactoring an existing app.

Previous

ASP.NET Core Blazor tutorials

Article • 11/18/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

The following tutorials provide basic working experiences for building Blazor apps.

For an overview of Blazor, see ASP.NET Core Blazor.

- Build your first Blazor app
- Build a Blazor todo list app (Blazor Web App)
- Build a Blazor movie database app (Overview) (Blazor Web App)
- Use ASP.NET Core SignalR with Blazor (Blazor Web App)
- ASP.NET Core Blazor Hybrid tutorials
- Microsoft Learn
  - Blazor Learning Path
  - Blazor Learn Modules

Tutorial: Create a web API with ASP.NET Core

Article • 08/23/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By Rick Anderson and Kirk Larkin

This tutorial teaches the basics of building a controller-based web API that uses a database. Another approach to creating APIs in ASP.NET Core is to create *minimal APIs*. For help with choosing between minimal APIs and controller-based APIs, see APIs overview. For a tutorial on creating a minimal API, see Tutorial: Create a minimal API with ASP.NET Core.

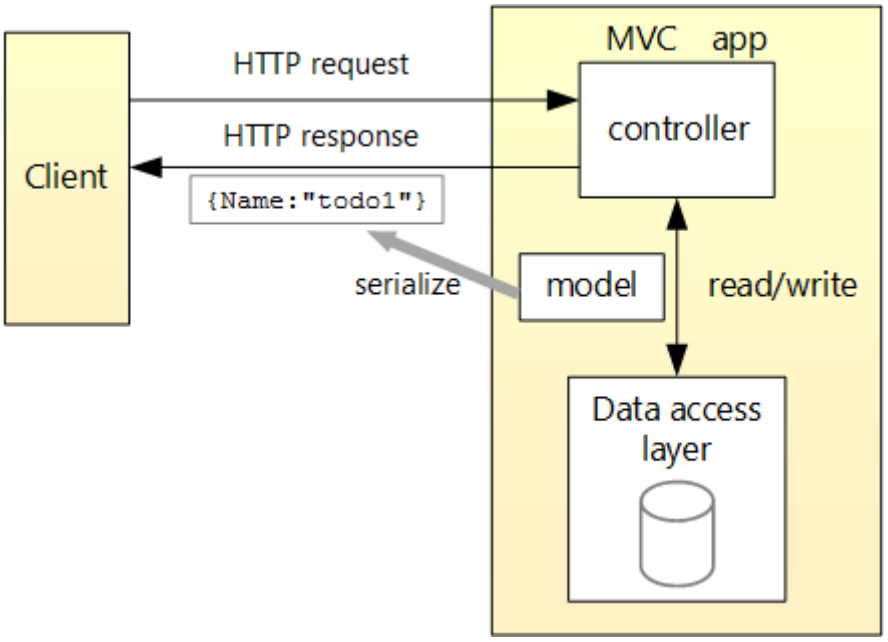
Overview

This tutorial creates the following API:

Expand table

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

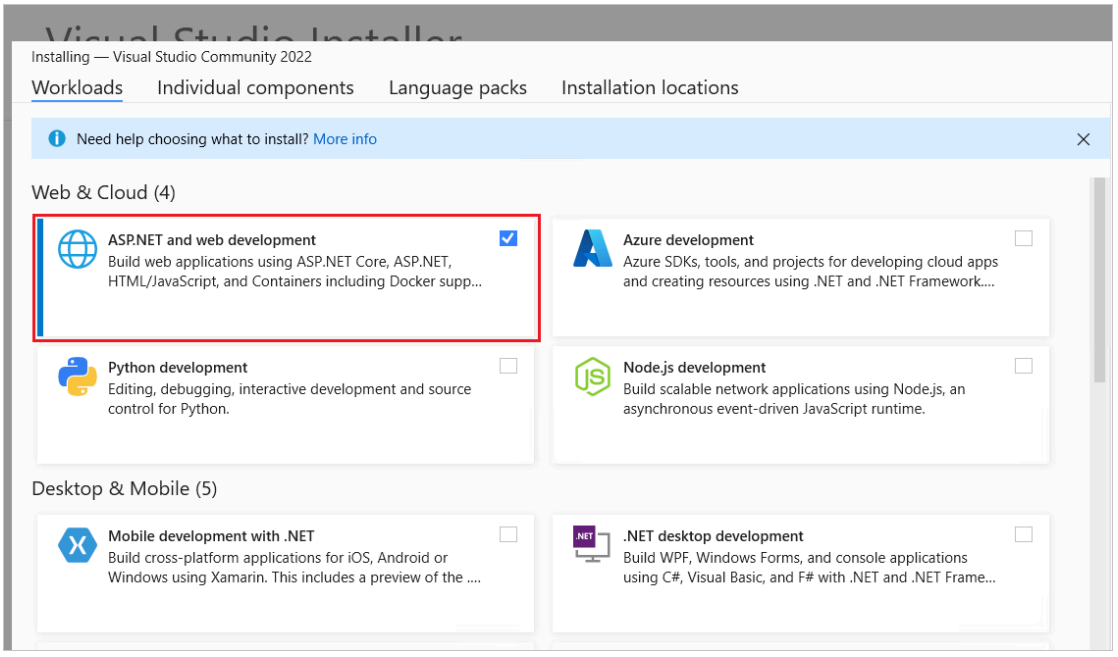
The following diagram shows the design of the app.



Prerequisites

Visual Studio

- Visual Studio 2022 with the **ASP.NET and web development** workload.



Create a web project

Visual Studio

- 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 8.0 (Long Term Support)**.
  - Confirm the checkbox for **Use controllers(uncheck to use minimal APIs)** is checked.
  - Confirm the checkbox for **Enable OpenAPI support** is checked.
  - Select **Create**.

Add a NuGet package

A NuGet package must be added to support the database used in this tutorial.

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

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

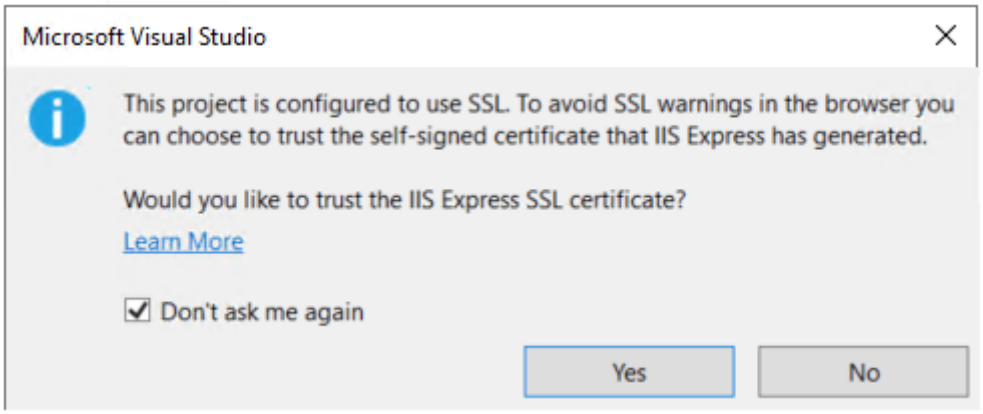
Test the project

The project template creates a `WeatherForecast` API with support for Swagger.

Visual Studio

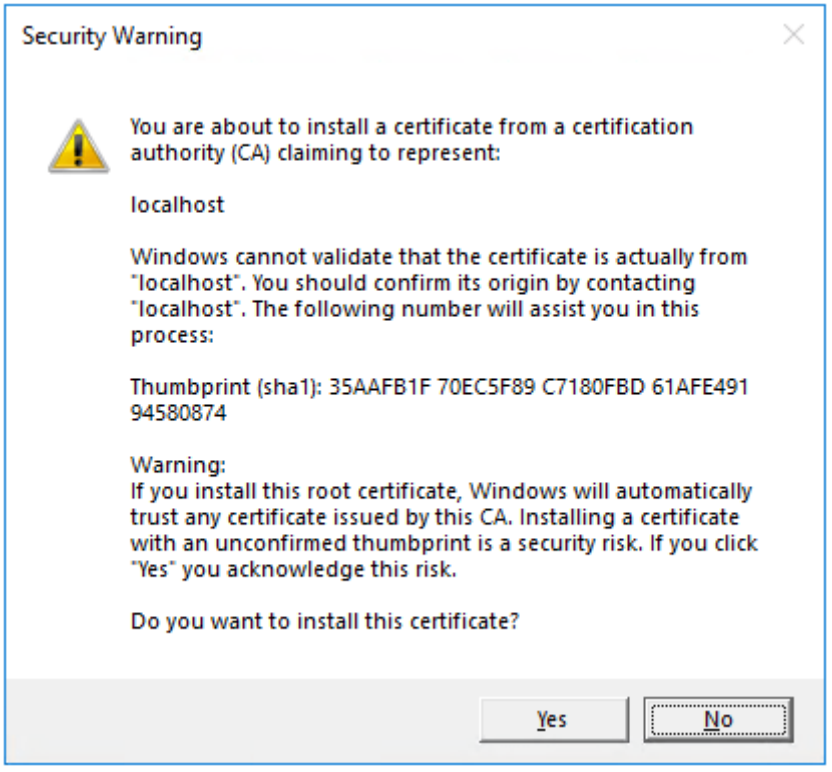
Press Ctrl+F5 to run without the debugger.

Visual Studio displays the following dialog when a project is not yet configured to use SSL:



Select **Yes** if you trust the IIS Express SSL certificate.

The following dialog is displayed:



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

For information on trusting the Firefox browser, see Firefox SEC_ERROR_INADEQUATE_KEY_USAGE certificate error.

Visual Studio launches the default browser and navigates to `https://localhost:<port>/swagger/index.html`, where `<port>` is a randomly chosen port number set at the project creation.

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.

If the Swagger page doesn't appear, see this GitHub issue.

Swagger is used to generate useful documentation and help pages for web APIs. This tutorial uses Swagger to test the app. For more information on Swagger, see ASP.NET Core web API documentation with Swagger / OpenAPI.

Copy and paste the **Request URL** in the browser: `https://localhost:<port>/weatherforecast`

JSON similar to the following example is returned:

JSON


```
[
    {
        "date": "2019-07-16T19:04:05.7257911-06:00",
        "temperatureC": 52,
        "temperatureF": 125,
        "summary": "Mild"
    },
    {
        "date": "2019-07-17T19:04:05.7258461-06:00",
        "temperatureC": 36,
        "temperatureF": 96,
        "summary": "Warm"
    },
    {
        "date": "2019-07-18T19:04:05.7258467-06:00",
        "temperatureC": 39,
        "temperatureF": 102,
        "summary": "Cool"
    },
    {
        "date": "2019-07-19T19:04:05.7258471-06:00",
        "temperatureC": 10,
        "temperatureF": 49,
        "summary": "Bracing"
    },
    {
        "date": "2019-07-20T19:04:05.7258474-06:00",
        "temperatureC": -1,
        "temperatureF": 31,
```

```
        "summary": "Chilly"
    }
]
```

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 the `TodoItem` class.

Visual Studio

- 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:

C#

```
namespace TodoApi.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.

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.

Visual Studio

- Right-click the `Models` folder and select **Add** > **Class**. Name the class *TodoContext* and click **Add**.

- Enter the following code:

```
C#

using Microsoft.EntityFrameworkCore;

namespace TodoApi.Models;

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

    public DbSet<TodoItem> TodoItems { get; set; } = null!;
}
```

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 highlighted code:

```
C#

using Microsoft.EntityFrameworkCore;
using TodoApi.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddDbContext<TodoContext>(opt =>
    opt.UseInMemoryDatabase("TodoList"));
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
```

```
}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

The preceding code:

- Adds `using` directives.
- Adds the database context to the DI container.
- Specifies that the database context will use an in-memory database.

Scaffold a controller

Visual Studio

- 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.

The generated code:

- Marks the class with the [ApiController] attribute. This attribute indicates that the controller responds to web API requests. For information about specific behaviors that the attribute enables, see Create web APIs with ASP.NET Core.
- 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 (method name) isn't included in the endpoint. That is, the action's associated method name isn't used in the matching route.

Update the PostTodoItem create method

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

C#

```
[HttpPost]
public async Task<ActionResult<TodoItem>> PostTodoItem(TodoItem todoItem)
{
    _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 `TodoItem` from the body of the HTTP request.

For more information, see Attribute routing with Http[Verb] attributes.

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.

Test PostTodoItem

- Press Ctrl+F5 to run the app.

- In the Swagger browser window, select **POST /api/TodoItems**, and then select **Try it out**.
- In the **Request body** input window, update the JSON. For example,

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

- Select **Execute**