

client errors to problem details.

## Additional resources

- [View or download sample code](#) <sup>↗</sup> (how to download)
- [Troubleshoot ASP.NET Core on Azure App Service and IIS](#)
- [Common error troubleshooting for Azure App Service and IIS with ASP.NET Core](#)
- [Handle errors in ASP.NET Core controller-based web APIs](#)
- [Handle errors in minimal APIs.](#)

# Make HTTP requests using `IHttpClientFactory` in ASP.NET Core

Article • 07/26/2024

## 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 [Kirk Larkin](#), [Steve Gordon](#), [Glenn Condrón](#), and [Ryan Nowak](#).

An `IHttpClientFactory` can be registered and used to configure and create `HttpClient` instances in an app. `IHttpClientFactory` offers the following benefits:

- Provides a central location for naming and configuring logical `HttpClient` instances. For example, a client named *github* could be registered and configured to access [GitHub](#). A default client can be registered for general access.
- Codifies the concept of outgoing middleware via delegating handlers in `HttpClient`. Provides extensions for Polly-based middleware to take advantage of delegating handlers in `HttpClient`.
- Manages the pooling and lifetime of underlying `HttpClientMessageHandler` instances. Automatic management avoids common DNS (Domain Name System) problems that occur when manually managing `HttpClient` lifetimes.
- Adds a configurable logging experience (via `ILogger`) for all requests sent through clients created by the factory.

The sample code in this topic version uses `System.Text.Json` to deserialize JSON content returned in HTTP responses. For samples that use `Json.NET` and `ReadAsStringAsync<T>`, use the version selector to select a 2.x version of this topic.

## Consumption patterns

There are several ways `IHttpClientFactory` can be used in an app:

- [Basic usage](#)
- [Named clients](#)

- [Typed clients](#)
- [Generated clients](#)

The best approach depends upon the app's requirements.

## Basic usage

Register `IHttpClientFactory` by calling `AddHttpClient` in `Program.cs`:

```
C#  
  
var builder = WebApplication.CreateBuilder(args);  
  
// Add services to the container.  
builder.Services.AddHttpClient();
```

An `IHttpClientFactory` can be requested using [dependency injection \(DI\)](#). The following code uses `IHttpClientFactory` to create an `HttpClient` instance:

```
C#  
  
public class BasicModel : PageModel  
{  
    private readonly IHttpClientFactory _httpClientFactory;  
  
    public BasicModel(IHttpClientFactory httpClientFactory) =>  
        _httpClientFactory = httpClientFactory;  
  
    public IEnumerable<GitHubBranch>? GitHubBranches { get; set; }  
  
    public async Task OnGet()  
    {  
        var httpRequestMessage = new HttpRequestMessage(  
            HttpMethod.Get,  
            "https://api.github.com/repos/dotnet/AspNetCore.Docs/branches")  
        {  
            Headers =  
            {  
                { HeaderNames.Accept, "application/vnd.github.v3+json" },  
                { HeaderNames.UserAgent, "HttpRequestsSample" }  
            }  
        };  
    };  
  
    var httpClient = _httpClientFactory.CreateClient();  
    var httpResponseMessage = await  
    httpClient.SendAsync(httpRequestMessage);  
  
    if (httpResponseMessage.IsSuccessStatusCode)  
    {  
        using var contentStream =
```

```

        await httpResponseMessage.Content.ReadAsStreamAsync();

        GitHubBranches = await JsonSerializer.DeserializeAsync
            <IEnumerable<GitHubBranch>>(contentStream);
    }
}
}

```

Using `IHttpClientFactory` like in the preceding example is a good way to refactor an existing app. It has no impact on how `HttpClient` is used. In places where `HttpClient` instances are created in an existing app, replace those occurrences with calls to [CreateClient](#).

## Named clients

Named clients are a good choice when:

- The app requires many distinct uses of `HttpClient`.
- Many `HttpClient`s have different configuration.

Specify configuration for a named `HttpClient` during its registration in `Program.cs`:

C#

```

builder.Services.AddHttpClient("GitHub", httpClient =>
{
    httpClient.BaseAddress = new Uri("https://api.github.com/");

    // using Microsoft.Net.Http.Headers;
    // The GitHub API requires two headers.
    httpClient.DefaultRequestHeaders.Add(
        HeaderNames.Accept, "application/vnd.github.v3+json");
    httpClient.DefaultRequestHeaders.Add(
        HeaderNames.UserAgent, "HttpRequestsSample");
});

```

In the preceding code the client is configured with:

- The base address `https://api.github.com/`.
- Two headers required to work with the GitHub API.

## CreateClient

Each time [CreateClient](#) is called:

- A new instance of `HttpClient` is created.

- The configuration action is called.

To create a named client, pass its name into `CreateClient`:

C#

```
public class NamedClientModel : PageModel
{
    private readonly IHttpConnectionFactory _httpClientFactory;

    public NamedClientModel(IHttClientFactory httpClientFactory) =>
        _httpClientFactory = httpClientFactory;

    public IEnumerable<GitHubBranch>? GitHubBranches { get; set; }

    public async Task OnGet()
    {
        var httpClient = _httpClientFactory.CreateClient("GitHub");
        var httpResponseMessage = await httpClient.GetAsync(
            "repos/dotnet/AspNetCore.Docs/branches");

        if (httpResponseMessage.IsSuccessStatusCode)
        {
            using var contentStream =
                await httpResponseMessage.Content.ReadAsStreamAsync();

            GitHubBranches = await JsonSerializer.DeserializeAsync<
                IEnumerable<GitHubBranch>>(contentStream);
        }
    }
}
```

In the preceding code, the request doesn't need to specify a hostname. The code can pass just the path, since the base address configured for the client is used.

## Typed clients

Typed clients:

- Provide the same capabilities as named clients without the need to use strings as keys.
- Provides IntelliSense and compiler help when consuming clients.
- Provide a single location to configure and interact with a particular `HttpClient`. For example, a single typed client might be used:
  - For a single backend endpoint.
  - To encapsulate all logic dealing with the endpoint.
- Work with DI and can be injected where required in the app.

A typed client accepts an `HttpClient` parameter in its constructor:

```
C#

public class GitHubService
{
    private readonly HttpClient _httpClient;

    public GitHubService(HttpClient httpClient)
    {
        _httpClient = httpClient;

        _httpClient.BaseAddress = new Uri("https://api.github.com/");

        // using Microsoft.Net.Http.Headers;
        // The GitHub API requires two headers.
        _httpClient.DefaultRequestHeaders.Add(
            HeaderNames.Accept, "application/vnd.github.v3+json");
        _httpClient.DefaultRequestHeaders.Add(
            HeaderNames.UserAgent, "HttpRequestsSample");
    }

    public async Task<IEnumerable<GitHubBranch>?>
    GetAspNetCoreDocsBranchesAsync() =>
        await _httpClient.GetFromJsonAsync<IEnumerable<GitHubBranch>>(
            "repos/dotnet/AspNetCore.Docs/branches");
}
```

In the preceding code:

- The configuration is moved into the typed client.
- The provided `HttpClient` instance is stored as a private field.

API-specific methods can be created that expose `HttpClient` functionality. For example, the `GetAspNetCoreDocsBranches` method encapsulates code to retrieve docs GitHub branches.

The following code calls `AddHttpClient` in `Program.cs` to register the `GitHubService` typed client class:

```
C#

builder.Services.AddHttpClient<GitHubService>();
```

The typed client is registered as transient with DI. In the preceding code, `AddHttpClient` registers `GitHubService` as a transient service. This registration uses a factory method to:

1. Create an instance of `HttpClient`.

2. Create an instance of `GitHubService`, passing in the instance of `HttpClient` to its constructor.

The typed client can be injected and consumed directly:

C#

```
public class TypedClientModel : PageModel
{
    private readonly GitHubService _gitHubService;

    public TypedClientModel(GitHubService gitHubService) =>
        _gitHubService = gitHubService;

    public IEnumerable<GitHubBranch>? GitHubBranches { get; set; }

    public async Task OnGet()
    {
        try
        {
            GitHubBranches = await
                _gitHubService.GetAspNetCoreDocsBranchesAsync();
        }
        catch (HttpRequestException)
        {
            // ...
        }
    }
}
```

The configuration for a typed client can also be specified during its registration in `Program.cs`, rather than in the typed client's constructor:

C#

```
builder.Services.AddHttpClient<GitHubService>(httpClient =>
{
    httpClient.BaseAddress = new Uri("https://api.github.com/");

    // ...
});
```

## Generated clients

`IHttpClientFactory` can be used in combination with third-party libraries such as [Refit](#). Refit is a REST library for .NET. It converts REST APIs into live interfaces. Call `AddRefitClient` to generate a dynamic implementation of an interface, which uses `HttpClient` to make the external HTTP calls.

A custom interface represents the external API:

```
C#

public interface IGitHubClient
{
    [Get("/repos/dotnet/AspNetCore.Docs/branches")]
    Task<IEnumerable<GitHubBranch>> GetAspNetCoreDocsBranchesAsync();
}
```

Call `AddRefitClient` to generate the dynamic implementation and then call `ConfigureHttpClient` to configure the underlying `HttpClient`:

```
C#

builder.Services.AddRefitClient<IGitHubClient>()
    .ConfigureHttpClient(httpClient =>
    {
        httpClient.BaseAddress = new Uri("https://api.github.com/");

        // using Microsoft.Net.Http.Headers;
        // The GitHub API requires two headers.
        httpClient.DefaultRequestHeaders.Add(
            HeaderNames.Accept, "application/vnd.github.v3+json");
        httpClient.DefaultRequestHeaders.Add(
            HeaderNames.UserAgent, "HttpRequestsSample");
    });
```

Use DI to access the dynamic implementation of `IGitHubClient`:

```
C#

public class RefitModel : PageModel
{
    private readonly IGitHubClient _gitHubClient;

    public RefitModel(IGitHubClient gitHubClient) =>
        _gitHubClient = gitHubClient;

    public IEnumerable<GitHubBranch>? GitHubBranches { get; set; }

    public async Task OnGet()
    {
        try
        {
            GitHubBranches = await
                _gitHubClient.GetAspNetCoreDocsBranchesAsync();
        }
        catch (ApiException)
        {
        }
    }
}
```



```
    }  
    }  
}
```

## Make POST, PUT, and DELETE requests

In the preceding examples, all HTTP requests use the GET HTTP verb. `HttpClient` also supports other HTTP verbs, including:

- POST
- PUT
- DELETE
- PATCH

For a complete list of supported HTTP verbs, see [HttpMethod](#).

The following example shows how to make an HTTP POST request:

C#

```
public async Task CreateItemAsync(TodoItem todoItem)  
{  
    var todoItemJson = new StringContent(  
        JsonSerializer.Serialize(todoItem),  
        Encoding.UTF8,  
        Application.Json); // using static System.Net.Mime.MediaTypeNames;  
  
    using var httpResponseMessage =  
        await _httpClient.PostAsync("/api/TodoItems", todoItemJson);  
  
    httpResponseMessage.EnsureSuccessStatusCode();  
}
```

In the preceding code, the `CreateItemAsync` method:

- Serializes the `TodoItem` parameter to JSON using `System.Text.Json`.
- Creates an instance of `StringContent` to package the serialized JSON for sending in the HTTP request's body.
- Calls `PostAsync` to send the JSON content to the specified URL. This is a relative URL that gets added to the `HttpClient.BaseAddress`.
- Calls `EnsureSuccessStatusCode` to throw an exception if the response status code doesn't indicate success.

`HttpClient` also supports other types of content. For example, [MultipartContent](#) and [StreamContent](#). For a complete list of supported content, see [HttpContent](#).

The following example shows an HTTP PUT request:

```
C#

public async Task SaveItemAsync(TodoItem todoItem)
{
    var todoItemJson = new StringContent(
        JsonSerializer.Serialize(todoItem),
        Encoding.UTF8,
        Application.Json);

    using var httpResponseMessage =
        await _httpClient.PutAsync($" /api/TodoItems/{todoItem.Id}",
        todoItemJson);

    httpResponseMessage.EnsureSuccessStatusCode();
}
```

The preceding code is similar to the POST example. The `SaveItemAsync` method calls [PutAsync](#) instead of `PostAsync`.

The following example shows an HTTP DELETE request:

```
C#

public async Task DeleteItemAsync(long itemId)
{
    using var httpResponseMessage =
        await _httpClient.DeleteAsync($" /api/TodoItems/{itemId}");

    httpResponseMessage.EnsureSuccessStatusCode();
}
```

In the preceding code, the `DeleteItemAsync` method calls [DeleteAsync](#). Because HTTP DELETE requests typically contain no body, the `DeleteAsync` method doesn't provide an overload that accepts an instance of `HttpContent`.

To learn more about using different HTTP verbs with `HttpClient`, see [HttpClient](#).

## Outgoing request middleware

`HttpClient` has the concept of delegating handlers that can be linked together for outgoing HTTP requests. `IHttpClientFactory`:

- Simplifies defining the handlers to apply for each named client.
- Supports registration and chaining of multiple handlers to build an outgoing request middleware pipeline. Each of these handlers is able to perform work before and after the outgoing request. This pattern:
  - Is similar to the inbound middleware pipeline in ASP.NET Core.
  - Provides a mechanism to manage cross-cutting concerns around HTTP requests, such as:
    - caching
    - error handling
    - serialization
    - logging

To create a delegating handler:

- Derive from [DelegatingHandler](#).
- Override [SendAsync](#). Execute code before passing the request to the next handler in the pipeline:

C#

```
public class ValidateHeaderHandler : DelegatingHandler
{
    protected override async Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {
        if (!request.Headers.Contains("X-API-KEY"))
        {
            return new HttpResponseMessage(HttpStatusCode.BadRequest)
            {
                Content = new StringContent(
                    "The API key header X-API-KEY is required.")
            };
        }

        return await base.SendAsync(request, cancellationToken);
    }
}
```

The preceding code checks if the `X-API-KEY` header is in the request. If `X-API-KEY` is missing, [BadRequest](#) is returned.

More than one handler can be added to the configuration for an `HttpClient` with [Microsoft.Extensions.DependencyInjection.HttpClientBuilderExtensions.AddHttpClientHandler](#):

C#

```
builder.Services.AddTransient<ValidateHeaderHandler>();

builder.Services.AddHttpClient("HttpMessageHandler")
    .AddHttpMessageHandler<ValidateHeaderHandler>();
```

In the preceding code, the `ValidateHeaderHandler` is registered with DI. Once registered, `AddHttpMessageHandler` can be called, passing in the type for the handler.

Multiple handlers can be registered in the order that they should execute. Each handler wraps the next handler until the final `HttpClientHandler` executes the request:

```
C#

builder.Services.AddTransient<SampleHandler1>();
builder.Services.AddTransient<SampleHandler2>();

builder.Services.AddHttpClient("MultipleHttpMessageHandlers")
    .AddHttpMessageHandler<SampleHandler1>()
    .AddHttpMessageHandler<SampleHandler2>();
```

In the preceding code, `SampleHandler1` runs first, before `SampleHandler2`.

## Use DI in outgoing request middleware

When `IHttpClientFactory` creates a new delegating handler, it uses DI to fulfill the handler's constructor parameters. `IHttpClientFactory` creates a **separate** DI scope for each handler, which can lead to surprising behavior when a handler consumes a *scoped* service.

For example, consider the following interface and its implementation, which represents a task as an operation with an identifier, `OperationId`:

```
C#

public interface IOperationScoped
{
    string OperationId { get; }
}

public class OperationScoped : IOperationScoped
{
    public string OperationId { get; } = Guid.NewGuid().ToString()[^4..];
}
```

As its name suggests, `IOperationScoped` is registered with DI using a *scoped* lifetime:

C#

```
builder.Services.AddScoped<IOperationScoped, OperationScoped>();
```

The following delegating handler consumes and uses `IOperationScoped` to set the `X-OPERATION-ID` header for the outgoing request:

C#

```
public class OperationHandler : DelegatingHandler
{
    private readonly IOperationScoped _operationScoped;

    public OperationHandler(IOperationScoped operationScoped) =>
        _operationScoped = operationScoped;

    protected override async Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {
        request.Headers.Add("X-OPERATION-ID", _operationScoped.OperationId);

        return await base.SendAsync(request, cancellationToken);
    }
}
```

In the [HttpRequestsSample download](#), navigate to `/operation` and refresh the page. The request scope value changes for each request, but the handler scope value only changes every 5 seconds.

Handlers can depend upon services of any scope. Services that handlers depend upon are disposed when the handler is disposed.

Use one of the following approaches to share per-request state with message handlers:

- Pass data into the handler using [HttpRequestMessage.Options](#).
- Use [IHttpContextAccessor](#) to access the current request.
- Create a custom [AsyncLocal<T>](#) storage object to pass the data.

## Use Polly-based handlers

`IHttpClientFactory` integrates with the third-party library [Polly](#). Polly is a comprehensive resilience and transient fault-handling library for .NET. It allows developers to express policies such as Retry, Circuit Breaker, Timeout, Bulkhead Isolation, and Fallback in a fluent and thread-safe manner.

Extension methods are provided to enable the use of Polly policies with configured `HttpClient` instances. The Polly extensions support adding Polly-based handlers to clients. Polly requires the [Microsoft.Extensions.Http.Polly](#) NuGet package.

## Handle transient faults

Faults typically occur when external HTTP calls are transient. [AddTransientHttpErrorPolicy](#) allows a policy to be defined to handle transient errors. Policies configured with `AddTransientHttpErrorPolicy` handle the following responses:

- [HttpRequestException](#)
- HTTP 5xx
- HTTP 408

`AddTransientHttpErrorPolicy` provides access to a `PolicyBuilder` object configured to handle errors representing a possible transient fault:

C#

```
builder.Services.AddHttpClient("PollyWaitAndRetry")
    .AddTransientHttpErrorPolicy(policyBuilder =>
        policyBuilder.WaitAndRetryAsync(
            3, retryNumber => TimeSpan.FromMilliseconds(600)));
```

In the preceding code, a `WaitAndRetryAsync` policy is defined. Failed requests are retried up to three times with a delay of 600 ms between attempts.

## Dynamically select policies

Extension methods are provided to add Polly-based handlers, for example, [AddPolicyHandler](#). The following `AddPolicyHandler` overload inspects the request to decide which policy to apply:

C#

```
var timeoutPolicy = Policy.TimeoutAsync<HttpResponseMessage>(
    TimeSpan.FromSeconds(10));
var longTimeoutPolicy = Policy.TimeoutAsync<HttpResponseMessage>(
    TimeSpan.FromSeconds(30));

builder.Services.AddHttpClient("PollyDynamic")
    .AddPolicyHandler(httpRequestMessage =>
        httpRequestMessage.Method == HttpMethod.Get ? timeoutPolicy :
        longTimeoutPolicy);
```

In the preceding code, if the outgoing request is an HTTP GET, a 10-second timeout is applied. For any other HTTP method, a 30-second timeout is used.

## Add multiple Polly handlers

It's common to nest Polly policies:

C#

```
builder.Services.AddHttpClient("PollyMultiple")
    .AddTransientHttpErrorPolicy(policyBuilder =>
        policyBuilder.RetryAsync(3))
    .AddTransientHttpErrorPolicy(policyBuilder =>
        policyBuilder.CircuitBreakerAsync(5, TimeSpan.FromSeconds(30)));
```

In the preceding example:

- Two handlers are added.
- The first handler uses `AddTransientHttpErrorPolicy` to add a retry policy. Failed requests are retried up to three times.
- The second `AddTransientHttpErrorPolicy` call adds a circuit breaker policy. Further external requests are blocked for 30 seconds if 5 failed attempts occur sequentially. Circuit breaker policies are stateful. All calls through this client share the same circuit state.

## Add policies from the Polly registry

An approach to managing regularly used policies is to define them once and register them with a `PolicyRegistry`. For example:

C#

```
var timeoutPolicy = Policy.TimeoutAsync<HttpResponseMessage>(
    TimeSpan.FromSeconds(10));
var longTimeoutPolicy = Policy.TimeoutAsync<HttpResponseMessage>(
    TimeSpan.FromSeconds(30));

var policyRegistry = builder.Services.AddPolicyRegistry();

policyRegistry.Add("Regular", timeoutPolicy);
policyRegistry.Add("Long", longTimeoutPolicy);

builder.Services.AddHttpClient("PollyRegistryRegular")
    .AddPolicyHandlerFromRegistry("Regular");
```

```
builder.Services.AddHttpClient("PollyRegistryLong")
    .AddPolicyHandlerFromRegistry("Long");
```

In the preceding code:

- Two policies, `Regular` and `Long`, are added to the Polly registry.
- `AddPolicyHandlerFromRegistry` configures individual named clients to use these policies from the Polly registry.

For more information on `IHttpClientFactory` and Polly integrations, see the [Polly wiki](#).

## HttpClient and lifetime management

A new `HttpClient` instance is returned each time `CreateClient` is called on the `IHttpClientFactory`. An `HttpMessageHandler` is created per named client. The factory manages the lifetimes of the `HttpMessageHandler` instances.

`IHttpClientFactory` pools the `HttpMessageHandler` instances created by the factory to reduce resource consumption. An `HttpMessageHandler` instance may be reused from the pool when creating a new `HttpClient` instance if its lifetime hasn't expired.

Pooling of handlers is desirable as each handler typically manages its own underlying HTTP connections. Creating more handlers than necessary can result in connection delays. Some handlers also keep connections open indefinitely, which can prevent the handler from reacting to DNS (Domain Name System) changes.

The default handler lifetime is two minutes. The default value can be overridden on a per named client basis:

C#

```
builder.Services.AddHttpClient("HandlerLifetime")
    .SetHandlerLifetime(TimeSpan.FromMinutes(5));
```

`HttpClient` instances can generally be treated as .NET objects **not** requiring disposal. Disposal cancels outgoing requests and guarantees the given `HttpClient` instance can't be used after calling `Dispose`. `IHttpClientFactory` tracks and disposes resources used by `HttpClient` instances.

Keeping a single `HttpClient` instance alive for a long duration is a common pattern used before the inception of `IHttpClientFactory`. This pattern becomes unnecessary after migrating to `IHttpClientFactory`.



# Alternatives to IHttpConnectionFactory

Using `IHttpConnectionFactory` in a DI-enabled app avoids:

- Resource exhaustion problems by pooling `HttpMessageHandler` instances.
- Stale DNS problems by cycling `HttpMessageHandler` instances at regular intervals.

There are alternative ways to solve the preceding problems using a long-lived `SocketsHttpHandler` instance.

- Create an instance of `SocketsHttpHandler` when the app starts and use it for the life of the app.
- Configure `PooledConnectionLifetime` to an appropriate value based on DNS refresh times.
- Create `HttpClient` instances using `new HttpClient(handler, disposeHandler: false)` as needed.

The preceding approaches solve the resource management problems that `IHttpConnectionFactory` solves in a similar way.

- The `SocketsHttpHandler` shares connections across `HttpClient` instances. This sharing prevents socket exhaustion.
- The `SocketsHttpHandler` cycles connections according to `PooledConnectionLifetime` to avoid stale DNS problems.

## Logging

Clients created via `IHttpConnectionFactory` record log messages for all requests. Enable the appropriate information level in the logging configuration to see the default log messages. Additional logging, such as the logging of request headers, is only included at trace level.

The log category used for each client includes the name of the client. A client named *MyNamedClient*, for example, logs messages with a category of "System.Net.Http.HttpClient.**MyNamedClient**.LogicalHandler". Messages suffixed with *LogicalHandler* occur outside the request handler pipeline. On the request, messages are logged before any other handlers in the pipeline have processed it. On the response, messages are logged after any other pipeline handlers have received the response.

Logging also occurs inside the request handler pipeline. In the *MyNamedClient* example, those messages are logged with the log category "System.Net.Http.HttpClient.**MyNamedClient**.ClientHandler". For the request, this occurs

after all other handlers have run and immediately before the request is sent. On the response, this logging includes the state of the response before it passes back through the handler pipeline.

Enabling logging outside and inside the pipeline enables inspection of the changes made by the other pipeline handlers. This may include changes to request headers or to the response status code.

Including the name of the client in the log category enables log filtering for specific named clients.

## Configure the `HttpMessageHandler`

It may be necessary to control the configuration of the inner `HttpMessageHandler` used by a client.

An `IHttpClientBuilder` is returned when adding named or typed clients. The [ConfigurePrimaryHttpMessageHandler](#) extension method can be used to define a delegate. The delegate is used to create and configure the primary `HttpMessageHandler` used by that client:

C#

```
builder.Services.AddHttpClient("ConfiguredHttpMessageHandler")
    .ConfigurePrimaryHttpMessageHandler(() =>
        new HttpClientHandler
        {
            AllowAutoRedirect = true,
            UseDefaultCredentials = true
        });
```

## Cookies

The pooled `HttpMessageHandler` instances results in `CookieContainer` objects being shared. Unanticipated `CookieContainer` object sharing often results in incorrect code. For apps that require cookies, consider either:

- Disabling automatic cookie handling
- Avoiding `IHttpClientFactory`

Call [ConfigurePrimaryHttpMessageHandler](#) to disable automatic cookie handling:

C#

```
builder.Services.AddHttpClient("NoAutomaticCookies")
    .ConfigurePrimaryHttpMessageHandler(() =>
        new HttpClientHandler
        {
            UseCookies = false
        });
```

## Use IHttpClientFactory in a console app

In a console app, add the following package references to the project:

- [Microsoft.Extensions.Hosting](#) ↗
- [Microsoft.Extensions.Http](#) ↗

In the following example:

- [IHttpClientFactory](#) and `GitHubService` are registered in the [Generic Host's](#) service container.
- `GitHubService` is requested from DI, which in-turn requests an instance of `IHttpClientFactory`.
- `GitHubService` uses `IHttpClientFactory` to create an instance of `HttpClient`, which it uses to retrieve docs GitHub branches.

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

var host = new HostBuilder()
    .ConfigureServices(services =>
    {
        services.AddHttpClient();
        services.AddTransient<GitHubService>();
    })
    .Build();

try
{
    var gitHubService = host.Services.GetRequiredService<GitHubService>();
    var gitHubBranches = await
gitHubService.GetAspNetCoreDocsBranchesAsync();

    Console.WriteLine($"{gitHubBranches?.Count() ?? 0} GitHub Branches");

    if (gitHubBranches is not null)
```

```

    {
        foreach (var gitHubBranch in gitHubBranches)
        {
            Console.WriteLine($"- {gitHubBranch.Name}");
        }
    }
}
catch (Exception ex)
{
    host.Services.GetRequiredService<ILogger<Program>>()
        .LogError(ex, "Unable to load branches from GitHub.");
}

public class GitHubService
{
    private readonly IHttpClientFactory _httpClientFactory;

    public GitHubService(IHttpClientFactory httpClientFactory) =>
        _httpClientFactory = httpClientFactory;

    public async Task<IEnumerable<GitHubBranch>?>
    GetAspNetCoreDocsBranchesAsync()
    {
        var httpRequestMessage = new HttpRequestMessage(
            HttpMethod.Get,
            "https://api.github.com/repos/dotnet/AspNetCore.Docs/branches")
        {
            Headers =
            {
                { "Accept", "application/vnd.github.v3+json" },
                { "User-Agent", "HttpRequestsConsoleSample" }
            }
        };

        var httpClient = _httpClientFactory.CreateClient();
        var httpResponseMessage = await
            httpClient.SendAsync(httpRequestMessage);

        httpResponseMessage.EnsureSuccessStatusCode();

        using var contentStream =
            await httpResponseMessage.Content.ReadAsStreamAsync();

        return await JsonSerializer.DeserializeAsync
            <IEnumerable<GitHubBranch>>(contentStream);
    }
}

public record GitHubBranch(
    [property: JsonPropertyName("name")] string Name);

```

## Header propagation middleware

Header propagation is an ASP.NET Core middleware to propagate HTTP headers from the incoming request to the outgoing `HttpClient` requests. To use header propagation:

- Install the [Microsoft.AspNetCore.HeaderPropagation](#) [↗](#) package.
- Configure the `HttpClient` and middleware pipeline in `Program.cs`:

```
C#

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

builder.Services.AddHttpClient("PropagateHeaders")
    .AddHeaderPropagation();

builder.Services.AddHeaderPropagation(options =>
{
    options.Headers.Add("X-TraceId");
});

var app = builder.Build();

// Configure the HTTP request pipeline.
app.UseHttpsRedirection();

app.UseHeaderPropagation();

app.MapControllers();
```

- Make outbound requests using the configured `HttpClient` instance, which includes the added headers.

## Additional resources

- [View or download sample code](#) [↗](#) (how to download)
- [Use HttpClientFactory to implement resilient HTTP requests](#)
- [Implement HTTP call retries with exponential backoff with HttpClientFactory and Polly policies](#)
- [Implement the Circuit Breaker pattern](#)
- [How to serialize and deserialize JSON in .NET](#)

# Static files in ASP.NET Core

Article • 11/05/2024

## ❗ 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](#) 

Static files, such as HTML, CSS, images, and JavaScript, are assets an ASP.NET Core app serves directly to clients by default.

For Blazor static files guidance, which adds to or supersedes the guidance in this article, see [ASP.NET Core Blazor static files](#).

## Serve static files

Static files are stored within the project's [web root](#) directory. The default directory is `{content root}/wwwroot`, but it can be changed with the [UseWebRoot](#) method. For more information, see [Content root](#) and [Web root](#).

The [CreateBuilder](#) method sets the content root to the current directory:

C#

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

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var app = builder.Build();

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

app.UseHttpsRedirection();
app.MapStaticAssets();
```

```
app.UseAuthorization();

app.MapDefaultControllerRoute().WithStaticAssets();
app.MapRazorPages().WithStaticAssets();

app.Run();
```

Static files are accessible via a path relative to the [web root](#). For example, the **Web Application** project templates contain several folders within the `wwwroot` folder:

- `wwwroot`
  - `css`
  - `js`
  - `lib`

Consider an app with the `wwwroot/images/MyImage.jpg` file. The URI format to access a file in the `images` folder is `https://<hostname>/images/<image_file_name>`. For example, `https://localhost:5001/images/MyImage.jpg`

## MapStaticAssets

Creating performant web apps requires optimizing asset delivery to the browser. Possible optimizations include:

- Serve a given asset once until the file changes or the browser clears its cache. Set the [ETag](#) header.
- Prevent the browser from using old or stale assets after an app is updated. Set the [Last-Modified](#) header.
- Set up proper [caching headers](#).
- Use [caching middleware](#).
- Serve [compressed](#) versions of the assets when possible.
- Use a [CDN](#) to serve the assets closer to the user.
- Minimize the size of assets served to the browser. This optimization doesn't include minification.

[MapStaticAssets](#) are routing endpoint conventions that optimize the delivery of static assets in an app. It's designed to work with all UI frameworks, including Blazor, Razor Pages, and MVC.

[UseStaticFiles](#) also serves static files, but it doesn't provide the same level of optimization as `MapStaticAssets`. For a comparison of `UseStaticFiles` and `MapStaticAssets`, see [Optimizing static web asset delivery](#).

## Serve files in web root

The default web app templates call the `MapStaticAssets` method in `Program.cs`, which enables static files to be served:

C#

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

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var app = builder.Build();

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

app.UseHttpsRedirection();
app.MapStaticAssets();

app.UseAuthorization();

app.MapDefaultControllerRoute().WithStaticAssets();
app.MapRazorPages().WithStaticAssets();

app.Run();
```

The parameterless `UseStaticFiles` method overload marks the files in [web root](#) as servable. The following markup references `wwwroot/images/MyImage.jpg`:

HTML

```

```

In the preceding markup, the tilde character `~` points to the [web root](#).

## Serve files outside of web root

Consider a directory hierarchy in which the static files to be served reside outside of the [web root](#):

- `wwwroot`
  - `css`
  - `images`



- js
- MyStaticFiles
  - images
    - red-rose.jpg

A request can access the `red-rose.jpg` file by configuring the Static File Middleware as follows:

C#

```
using Microsoft.Extensions.FileProviders;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var app = builder.Build();

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

app.UseHttpsRedirection();

app.UseStaticFiles(); //Serve files from wwwroot
app.UseStaticFiles(new StaticFileOptions
{
    FileProvider = new PhysicalFileProvider(
        Path.Combine(builder.Environment.ContentRootPath,
            "MyStaticFiles")),
    RequestPath = "/StaticFiles"
});

app.UseAuthorization();

app.MapDefaultControllerRoute().WithStaticAssets();
app.MapRazorPages().WithStaticAssets();

app.Run();
```

In the preceding code, the *MyStaticFiles* directory hierarchy is exposed publicly via the *StaticFiles* URI segment. A request to `https://<hostname>/StaticFiles/images/red-rose.jpg` serves the `red-rose.jpg` file.

The following markup references `MyStaticFiles/images/red-rose.jpg`:

HTML

```

```

To serve files from multiple locations, see [Serve files from multiple locations](#).

## Set HTTP response headers

A [StaticFileOptions](#) object can be used to set HTTP response headers. In addition to configuring static file serving from the [web root](#), the following code sets the [Cache-Control](#) [header](#):

C#

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

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var app = builder.Build();

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

app.UseHttpsRedirection();

var cacheMaxAgeOneWeek = (60 * 60 * 24 * 7).ToString();
app.UseStaticFiles(new StaticFileOptions
{
    OnPrepareResponse = ctx =>
    {
        ctx.Context.Response.Headers.Append(
            "Cache-Control", $"public, max-age={cacheMaxAgeOneWeek}");
    }
});

app.UseAuthorization();

app.MapDefaultControllerRoute().WithStaticAssets();
app.MapRazorPages().WithStaticAssets();

app.Run();
```

The preceding code makes static files publicly available in the local cache for one week.

# Static file authorization

The ASP.NET Core templates call [MapStaticAssets](#) before calling [UseAuthorization](#). Most apps follow this pattern. When the Static File Middleware is called before the authorization middleware:

- No authorization checks are performed on the static files.
- Static files served by the Static File Middleware, such as those under `wwwroot`, are publicly accessible.

To serve static files based on authorization:

- Store them outside of `wwwroot`.
- Call `UseStaticFiles`, specifying a path, after calling `UseAuthorization`.
- Set the [fallback authorization policy](#).

C#

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.FileProviders;
using StaticFileAuth.Data;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddRazorPages();

builder.Services.AddAuthorization(options =>
{
    options.FallbackPolicy = new AuthorizationPolicyBuilder()
        .RequireAuthenticatedUser()
        .Build();
});

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
```

```

else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

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

app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

app.UseStaticFiles(new StaticFileOptions
{
    FileProvider = new PhysicalFileProvider(
        Path.Combine(builder.Environment.ContentRootPath,
            "MyStaticFiles")),
    RequestPath = "/StaticFiles"
});

app.MapRazorPages();

app.Run();

```

In the preceding code, the fallback authorization policy requires *all* users to be authenticated. Endpoints such as controllers, Razor Pages, etc that specify their own authorization requirements don't use the fallback authorization policy. For example, Razor Pages, controllers, or action methods with `[AllowAnonymous]` or `[Authorize(PolicyName="MyPolicy")]` use the applied authorization attribute rather than the fallback authorization policy.

`RequireAuthenticatedUser` adds `DenyAnonymousAuthorizationRequirement` to the current instance, which enforces that the current user is authenticated.

Static assets under `wwwroot` are publicly accessible because the default Static File Middleware (`app.UseStaticFiles();`) is called before `UseAuthentication`. Static assets in the *MyStaticFiles* folder require authentication. The [sample code](#) demonstrates this.

An alternative approach to serve files based on authorization is to:

- Store them outside of `wwwroot` and any directory accessible to the Static File Middleware.
- Serve them via an action method to which authorization is applied and return a `FileResult` object:

C#

```
[Authorize]
public class BannerImageModel : PageModel
{
    private readonly IWebHostEnvironment _env;

    public BannerImageModel(IWebHostEnvironment env) =>
        _env = env;

    public PhysicalFileResult OnGet()
    {
        var filePath = Path.Combine(
            _env.ContentRootPath, "MyStaticFiles", "images", "red-
            rose.jpg");

        return PhysicalFile(filePath, "image/jpeg");
    }
}
```

The preceding approach requires a page or endpoint per file. The following code returns files or uploads files for authenticated users:

C#

```
app.MapGet("/files/{fileName}", IActionResult (string fileName) =>
{
    var filePath = GetOrCreateFilePath(fileName);

    if (File.Exists(filePath))
    {
        return TypedResults.PhysicalFile(filePath, fileDownloadName: $"
{fileName}");
    }

    return TypedResults.NotFound("No file found with the supplied file
name");
})
.WithName("GetFileByName")
.RequireAuthorization("AuthenticatedUsers");

app.MapPost("/files",
    async (IFormFile file, LinkGenerator linker, HttpContext context) =>
    {
        // Don't rely on the file.FileName as it is only metadata that can
        be
        // manipulated by the end-user. See the `Utilities.IsFileValid`
        method that
        // takes an IFormFile and validates its signature within the
        // AllowedFileSignatures

        var fileSaveName = Guid.NewGuid().ToString("N")
    })
    .WithName("PostFile")
    .RequireAuthorization("AuthenticatedUsers")
    .RequireFormFile(1)
    .RequireFormFile(2);
```

```

        + Path.GetExtension(file.FileName);
        await SaveFileWithCustomFileName(file, fileSaveName);

        context.Response.Headers.Append("Location",
                                         linker.GetPathByName(context,
"GetFileByName",
                                         new { fileName = fileSaveName}));
        return TypedResults.Ok("File Uploaded Successfully!");
    })
    .RequireAuthorization("AdminsOnly");

app.Run();

```

IFormFile in the preceding sample uses memory buffer for uploading. For handling large file use streaming. See [Upload large files with streaming](#).

See the [StaticFileAuth](#) [GitHub](#) folder for the complete sample.

## Directory browsing

Directory browsing allows directory listing within specified directories.

Directory browsing is disabled by default for security reasons. For more information, see [Security considerations for static files](#).

Enable directory browsing with [AddDirectoryBrowser](#) and [UseDirectoryBrowser](#):

```

C#

using Microsoft.AspNetCore.StaticFiles;
using Microsoft.Extensions.FileProviders;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.AddDirectoryBrowser();

var app = builder.Build();

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

app.UseHttpsRedirection();

app.MapStaticAssets();

```

```

var fileProvider = new
PhysicalFileProvider(Path.Combine(builder.Environment.WebRootPath,
"images"));
var requestPath = "/MyImages";

// Enable displaying browser links.
app.UseStaticFiles(new StaticFileOptions
{
    FileProvider = fileProvider,
    RequestPath = requestPath
});

app.UseDirectoryBrowser(new DirectoryBrowserOptions
{
    FileProvider = fileProvider,
    RequestPath = requestPath
});

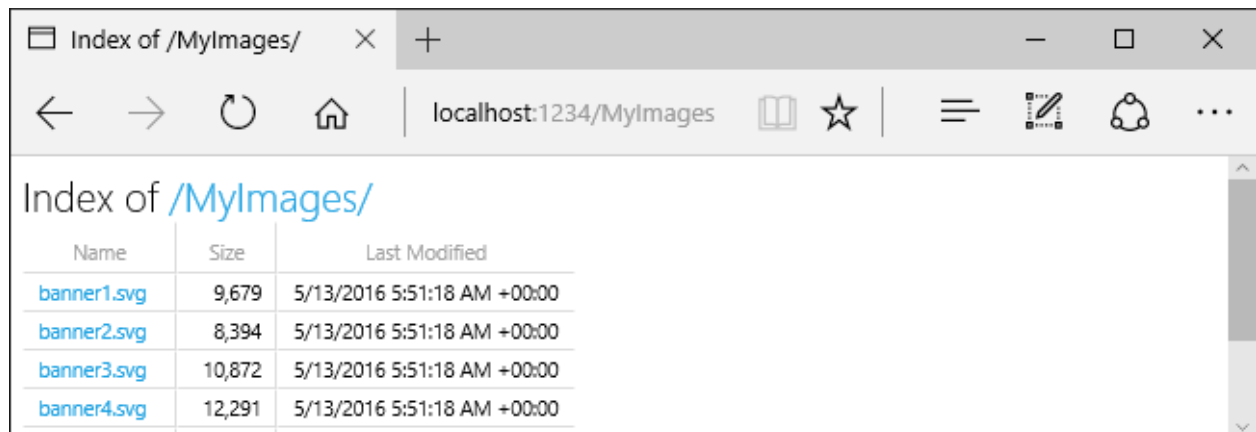
app.UseAuthorization();

app.MapDefaultControllerRoute().WithStaticAssets();
app.MapRazorPages().WithStaticAssets();

app.Run();

```

The preceding code allows directory browsing of the *wwwroot/images* folder using the URL `https://<hostname>/MyImages`, with links to each file and folder:



Name	Size	Last Modified
<a href="#">banner1.svg</a>	9,679	5/13/2016 5:51:18 AM +00:00
<a href="#">banner2.svg</a>	8,394	5/13/2016 5:51:18 AM +00:00
<a href="#">banner3.svg</a>	10,872	5/13/2016 5:51:18 AM +00:00
<a href="#">banner4.svg</a>	12,291	5/13/2016 5:51:18 AM +00:00

`AddDirectoryBrowser` [adds services](#) required by the directory browsing middleware, including `HtmlEncoder`. These services may be added by other calls, such as `AddRazorPages`, but we recommend calling `AddDirectoryBrowser` to ensure the services are added in all apps.

## Serve default documents

Setting a default page provides visitors a starting point on a site. To serve a default file from `wwwroot` without requiring the request URL to include the file's name, call the [UseDefaultFiles](#) method:

C#

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

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var app = builder.Build();

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

app.UseHttpsRedirection();

app.UseDefaultFiles();

app.UseStaticFiles();
app.UseAuthorization();

app.MapDefaultControllerRoute().WithStaticAssets();
app.MapRazorPages().WithStaticAssets();

app.Run();
```

`UseDefaultFiles` must be called before `UseStaticFiles` to serve the default file.

`UseDefaultFiles` is a URL rewriter that doesn't serve the file.

With `UseDefaultFiles`, requests to a folder in `wwwroot` search for:

- `default.htm`
- `default.html`
- `index.htm`
- `index.html`

The first file found from the list is served as though the request included the file's name. The browser URL continues to reflect the URI requested. For example, in the [sample app](#), a request to `https://localhost:<port>/def/` serves `default.html` from `wwwroot/def`.

The following code changes the default file name to `mydefault.html`:



C#

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

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var app = builder.Build();

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

app.UseHttpsRedirection();

var options = new DefaultFilesOptions();
options.DefaultFileNames.Clear();
options.DefaultFileNames.Add("mydefault.html");
app.UseDefaultFiles(options);

app.UseStaticFiles();

app.UseAuthorization();

app.MapDefaultControllerRoute().WithStaticAssets();
app.MapRazorPages().WithStaticAssets();

app.Run();
```

## UseFileServer for default documents

[UseFileServer](#) combines the functionality of `UseStaticFiles`, `UseDefaultFiles`, and optionally `UseDirectoryBrowser`.

Call `app.UseFileServer` to enable the serving of static files and the default file. Directory browsing isn't enabled:

C#

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

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
```

```

    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();

app.UseFileServer();

app.UseAuthorization();

app.MapDefaultControllerRoute().WithStaticAssets();
app.MapRazorPages().WithStaticAssets();

app.Run();

```

The following code enables the serving of static files, the default file, and directory browsing:

```

C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.AddDirectoryBrowser();

var app = builder.Build();

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

app.UseHttpsRedirection();

app.UseFileServer(enableDirectoryBrowsing: true);

app.UseRouting();

app.UseAuthorization();

app.MapDefaultControllerRoute().WithStaticAssets();
app.MapRazorPages().WithStaticAssets();

app.Run();

```

Consider the following directory hierarchy:

- `wwwroot`

- `css`
- `images`
- `js`
- `MyStaticFiles`
  - `defaultFiles`
    - `default.html`
    - `image3.png`
  - `images`
    - `MyImage.jpg`

The following code enables the serving of static files, the default file, and directory browsing of `MyStaticFiles`:

C#

```
using Microsoft.Extensions.FileProviders;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.AddDirectoryBrowser();

var app = builder.Build();

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

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseFileServer(new FileServerOptions
{
    FileProvider = new PhysicalFileProvider(
        Path.Combine(builder.Environment.ContentRootPath,
            "MyStaticFiles")),
    RequestPath = "/StaticFiles",
    EnableDirectoryBrowsing = true
});

app.UseAuthorization();

app.MapDefaultControllerRoute().WithStaticAssets();
app.MapRazorPages().WithStaticAssets();
```

```
app.Run();
```

[AddDirectoryBrowser](#) must be called when the `EnableDirectoryBrowsing` property value is `true`.

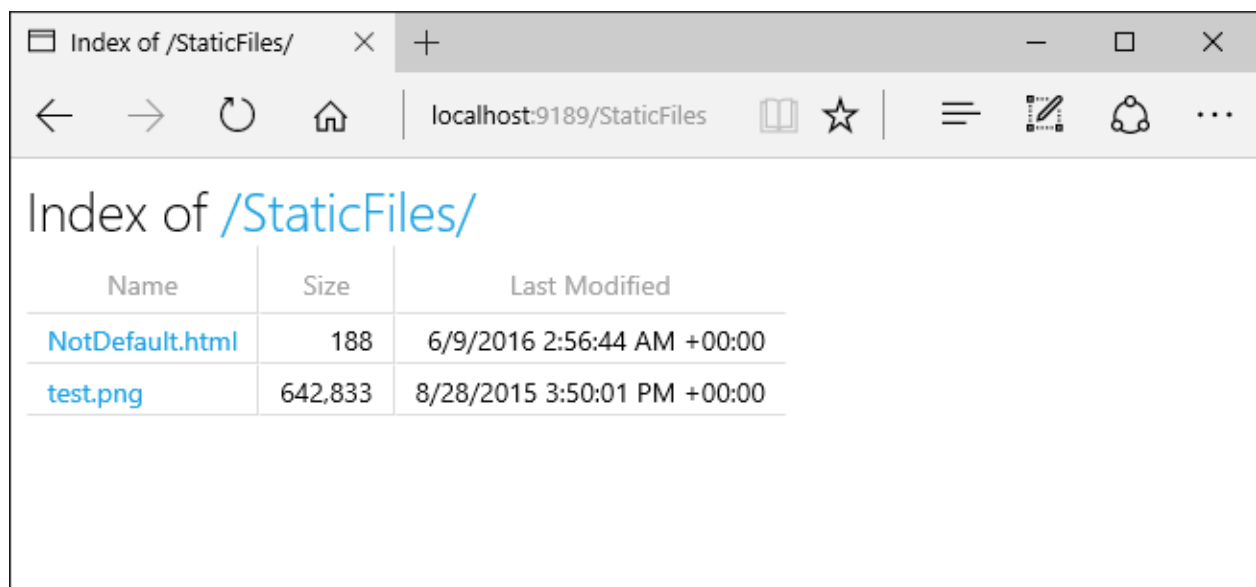
Using the preceding file hierarchy and code, URLs resolve as follows:

[Expand table](#)

URI	Response
<code>https://&lt;hostname&gt;/StaticFiles/images/MyImage.jpg</code>	<code>MyStaticFiles/images/MyImage.jpg</code>
<code>https://&lt;hostname&gt;/StaticFiles</code>	directory listing
<code>https://&lt;hostname&gt;/StaticFiles/defaultFiles</code>	<code>MyStaticFiles/defaultFiles/default.html</code>
<code>https://&lt;hostname&gt;/StaticFiles/defaultFiles/image3.png</code>	<code>MyStaticFiles/defaultFiles//image3.png</code>

If no default-named file exists in the *MyStaticFiles* directory,

`https://<hostname>/StaticFiles` returns the directory listing with clickable links:



The screenshot shows a web browser window with the address bar set to `localhost:9189/StaticFiles`. The page title is "Index of /StaticFiles/". Below the title is a table with three columns: "Name", "Size", and "Last Modified". The table contains two entries: "NotDefault.html" with a size of 188 and a last modified date of 6/9/2016 2:56:44 AM +00:00, and "test.png" with a size of 642,833 and a last modified date of 8/28/2015 3:50:01 PM +00:00. Both entries are hyperlinks.

Name	Size	Last Modified
<a href="#">NotDefault.html</a>	188	6/9/2016 2:56:44 AM +00:00
<a href="#">test.png</a>	642,833	8/28/2015 3:50:01 PM +00:00

[UseDefaultFiles](#) and [UseDirectoryBrowser](#) perform a client-side redirect from the target URI without a trailing `/` to the target URI with a trailing `/`. For example, from `https://<hostname>/StaticFiles` to `https://<hostname>/StaticFiles/`. Relative URLs within the *StaticFiles* directory are invalid without a trailing slash (`/`) unless the [RedirectToAppendTrailingSlash](#) option of [DefaultFilesOptions](#) is used.

## FileExtensionContentTypeProvider

The `FileExtensionContentTypeProvider` class contains a `Mappings` property that serves as a mapping of file extensions to MIME content types. In the following sample, several file extensions are mapped to known MIME types. The `.rtf` extension is replaced, and `.mp4` is removed:

C#

```
using Microsoft.AspNetCore.StaticFiles;
using Microsoft.Extensions.FileProviders;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var app = builder.Build();

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

app.UseHttpsRedirection();

// Set up custom content types - associating file extension to MIME type
var provider = new FileExtensionContentTypeProvider();
// Add new mappings
provider.Mappings[".myapp"] = "application/x-msdownload";
provider.Mappings[".htm3"] = "text/html";
provider.Mappings[".image"] = "image/png";
// Replace an existing mapping
provider.Mappings[".rtf"] = "application/x-msdownload";
// Remove MP4 videos.
provider.Mappings.Remove(".mp4");

app.UseStaticFiles(new StaticFileOptions
{
    ContentTypeProvider = provider
});

app.UseAuthorization();

app.MapDefaultControllerRoute().WithStaticAssets();
app.MapRazorPages().WithStaticAssets();

app.Run();
```

See [MIME content types](#).

# Non-standard content types

The Static File Middleware understands almost 400 known file content types. If the user requests a file with an unknown file type, the Static File Middleware passes the request to the next middleware in the pipeline. If no middleware handles the request, a *404 Not Found* response is returned. If directory browsing is enabled, a link to the file is displayed in a directory listing.

The following code enables serving unknown types and renders the unknown file as an image:

C#

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

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var app = builder.Build();

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

app.UseHttpsRedirection();

app.UseStaticFiles(new StaticFileOptions
{
    ServeUnknownFileTypes = true,
    DefaultContentType = "image/png"
});

app.UseAuthorization();

app.MapDefaultControllerRoute().WithStaticAssets();
app.MapRazorPages().WithStaticAssets();

app.Run();
```

With the preceding code, a request for a file with an unknown content type is returned as an image.

## Warning

Enabling [ServeUnknownFileTypes](#) is a security risk. It's disabled by default, and its use is discouraged. [FileExtensionContentTypeProvider](#) provides a safer alternative

to serving files with non-standard extensions.

## Serve files from multiple locations

Consider the following Razor page which displays the `/MyStaticFiles/image3.png` file:

C#HTML

```
@page
```

```
<p> Test /MyStaticFiles/image3.png</p>
```

```

```

`UseStaticFiles` and `UseFileServer` default to the file provider pointing at `wwwroot`.

Additional instances of `UseStaticFiles` and `UseFileServer` can be provided with other file providers to serve files from other locations. The following example calls

`UseStaticFiles` twice to serve files from both `wwwroot` and `MyStaticFiles`:

C#

```
app.UseStaticFiles();
app.UseStaticFiles(new StaticFileOptions
{
    FileProvider = new PhysicalFileProvider(
        Path.Combine(builder.Environment.ContentRootPath, "MyStaticFiles"))
});
```

Using the preceding code:

- The `/MyStaticFiles/image3.png` file is displayed.
- The [Image Tag Helpers AppendVersion](#) is not applied because the Tag Helpers depend on [WebRootFileProvider](#). `WebRootFileProvider` has not been updated to include the `MyStaticFiles` folder.

The following code updates the `WebRootFileProvider`, which enables the Image Tag Helper to provide a version:

C#

```
var webRootProvider = new
PhysicalFileProvider(builder.Environment.WebRootPath);
var newPathProvider = new PhysicalFileProvider(
    Path.Combine(builder.Environment.ContentRootPath, "MyStaticFiles"));
```

```
var compositeProvider = new CompositeFileProvider(webRootProvider,  
                                                newPathProvider);  
  
// Update the default provider.  
app.Environment.WebRootFileProvider = compositeProvider;  
  
app.MapStaticAssets();
```

### ⓘ Note

The preceding approach applies to Razor Pages and MVC apps. For guidance that applies to Blazor Web Apps, see [ASP.NET Core Blazor static files](#).

## Security considerations for static files

### ⚠ Warning

`UseDirectoryBrowser` and `UseStaticFiles` can leak secrets. Disabling directory browsing in production is highly recommended. Carefully review which directories are enabled via `UseStaticFiles` or `UseDirectoryBrowser`. The entire directory and its sub-directories become publicly accessible. Store files suitable for serving to the public in a dedicated directory, such as `<content_root>/wwwroot`. Separate these files from MVC views, Razor Pages, configuration files, etc.

- The URLs for content exposed with `UseDirectoryBrowser`, `UseStaticFiles`, and `MapStaticAssets` are subject to the case sensitivity and character restrictions of the underlying file system. For example, Windows is case insensitive, but macOS and Linux aren't.
- ASP.NET Core apps hosted in IIS use the [ASP.NET Core Module](#) to forward all requests to the app, including static file requests. The IIS static file handler isn't used and has no chance to handle requests.
- Complete the following steps in IIS Manager to remove the IIS static file handler at the server or website level:
  1. Navigate to the **Modules** feature.
  2. Select **StaticFileModule** in the list.
  3. Click **Remove** in the **Actions** sidebar.



### ⚠ Warning

If the IIS static file handler is enabled **and** the ASP.NET Core Module is configured incorrectly, static files are served. This happens, for example, if the *web.config* file isn't deployed.

- Place code files, including `.cs` and `.cshtml`, outside of the app project's [web root](#). A logical separation is therefore created between the app's client-side content and server-based code. This prevents server-side code from being leaked.

## Serve files outside wwwroot by updating `IWebHostEnvironment.WebRootPath`

When `IWebHostEnvironment.WebRootPath` is set to a folder other than `wwwroot`:

- In the development environment, static assets found in both `wwwroot` and the updated `IWebHostEnvironment.WebRootPath` are served from `wwwroot`.
- In any environment other than development, duplicate static assets are served from the updated `IWebHostEnvironment.WebRootPath` folder.

Consider a web app created with the empty web template:

- Containing an `Index.html` file in `wwwroot` and `wwwroot-custom`.
- With the following updated `Program.cs` file that sets `WebRootPath = "wwwroot-custom"`:

```
C#  
  
var builder = WebApplication.CreateBuilder(new WebApplicationOptions  
{  
    Args = args,  
    // Look for static files in "wwwroot-custom"  
    WebRootPath = "wwwroot-custom"  
});  
  
var app = builder.Build();  
  
app.UseDefaultFiles();  
app.MapStaticAssets();  
  
app.Run();
```

In the preceding code, requests to `/`:

- In the development environment return `wwwroot/Index.html`
- In any environment other than development return `wwwroot-custom/Index.html`

To ensure assets from `wwwroot-custom` are returned, use one of the following approaches:

- Delete duplicate named assets in `wwwroot`.
- Set `"ASPNETCORE_ENVIRONMENT"` in `Properties/launchSettings.json` to any value other than `"Development"`.

- Completely disable static web assets by setting `<StaticWebAssetsEnabled>false</StaticWebAssetsEnabled>` in the project file.

**WARNING, disabling static web assets disables [Razor Class Libraries](#).**

- Add the following XML to the project file:

XML

```
<ItemGroup>
  <Content Remove="wwwroot\*" />
</ItemGroup>
```

The following code updates `IWebHostEnvironment.WebRootPath` to a non development value, guaranteeing duplicate content is returned from `wwwroot-custom` rather than `wwwroot`:

C#

```
var builder = WebApplication.CreateBuilder(new WebApplicationOptions
{
    Args = args,
    // Examine Hosting environment: logging value
    EnvironmentName = Environments.Staging,
    WebRootPath = "wwwroot-custom"
});

var app = builder.Build();

app.Logger.LogInformation("ASPNETCORE_ENVIRONMENT: {env}",
    Environment.GetEnvironmentVariable("ASPNETCORE_ENVIRONMENT"));

app.Logger.LogInformation("app.Environment.IsDevelopment(): {env}",
    app.Environment.IsDevelopment().ToString());

app.UseDefaultFiles();
```

```
app.MapStaticAssets();
```

```
app.Run();
```

## Additional resources

- [View or download sample code](#) [↗](#) (how to download)
- [Middleware](#)
- [Introduction to ASP.NET Core](#)
- [ASP.NET Core Blazor file uploads](#)
- [ASP.NET Core Blazor file downloads](#)

# dotnet-scaffold telemetry

Article • 11/11/2024

The `dotnet-scaffold` command includes a telemetry feature that collects usage data. This feature helps the `dotnet-scaffold` team understand how the tool is used so it can be improved.

## How to opt out

The `dotnet-scaffold` telemetry feature is enabled by default. To opt out of the telemetry feature, set the `DOTNET_SCAFFOLD_TELEMETRY_OPTOUT` environment variable to `1` or `true`.

## Disclosure

When you run the `dotnet-scaffold` tool the first time, it displays output similar to the following example. The text may vary slightly depending on the version of the tool you're running. This "first run" experience is how Microsoft notifies you about data collection.

Console

```
dotnet-scaffold collects usage data in order to help us improve your
experience. The data is collected by Microsoft and shared with the
community. You can opt-out of telemetry by setting the
DOTNET_SCAFFOLD_TELEMETRY_OPTOUT environment variable to '1' or 'true' using
your favorite shell.
```

```
Read more about dotnet-scaffold telemetry:
https://aka.ms/dotnet-scaffold/telemetry
Read more about .NET CLI Tools telemetry:
https://aka.ms/dotnet-cli-telemetry
```

To suppress the "first run" experience text, set the `DOTNET_SCAFFOLD_SKIP_FIRST_TIME_EXPERIENCE` environment variable to `1` or `true`.

## Data points

The telemetry feature doesn't collect:

- Personal data, such as usernames, email addresses, or URLs.
- Any project data.

The data is sent securely to Microsoft servers and held under restricted access.

Protecting your privacy is important to us. If you suspect the telemetry feature is collecting sensitive data or the data is being insecurely or inappropriately handled, take one of the following actions:


- File an issue in the [dotnet/scaffolding](#) repository.
- Send an email to [dotnet@microsoft.com](mailto:dotnet@microsoft.com) for investigation.

The telemetry feature collects the following data.

 Expand table

.NET SDK versions	Data
>=8.0	Timestamp of invocation.
>=8.0	Three-octet IP address used to determine the geographical location.
>=8.0	Operating system and version.
>=8.0	Runtime ID (RID) the tool is running on.
>=8.0	Whether the tool is running in a container.
>=8.0	Hashed Media Access Control (MAC) address: a cryptographically (SHA256) hashed and unique ID for a machine.
>=8.0	Kernel version.
>=8.0	dotnet-scaffold version.
>=8.0	Hashed tool information (tool name, tool version, tool package name, tool package version, chosen scaffolder category, related scaffolding categories)
>=8.0	Tool level (global or local tool)
>=8.0	Hashed command invoked (for example, <code>mvccontroller</code> ) and whether it succeeded.
>=8.0	dotnet-scaffold-aspnet scaffolder name, step names, and whether they succeeded.
>=8.0	dotnet-scaffold-aspire scaffolder name and whether it succeeded.
>=8.0	dotnet-scaffold aspnet scaffolder validation method name and whether they succeed.
>=8.0	dotnet-scaffold aspire scaffolder validation method name and whether they succeed.

# Additional resources

- [.NET Core SDK telemetry](#)
- [.NET CLI telemetry data](#) 

# 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](#)

# Introduction to Razor Pages in ASP.NET Core

Article • 09/27/2024

By [Rick Anderson](#), [Dave Brock](#), and [Kirk Larkin](#)

## 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](#).

Razor Pages can make coding page-focused scenarios easier and more productive than using controllers and views.

If you're looking for a tutorial that uses the Model-View-Controller approach, see [Get started with ASP.NET Core MVC](#).

This document provides an introduction to Razor Pages. It's not a step by step tutorial. If you find some of the sections too advanced, see [Get started with Razor Pages](#). For an overview of ASP.NET Core, see the [Introduction to ASP.NET Core](#).

## Prerequisites

### Visual Studio

- [Visual Studio 2022](#) with the **ASP.NET and web development** workload.
- [.NET 6.0 SDK](#)

## Create a Razor Pages project

### Visual Studio

See [Get started with Razor Pages](#) for detailed instructions on how to create a Razor Pages project.

# Razor Pages

Razor Pages is enabled in `Program.cs`:

C#

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

builder.Services.AddRazorPages();

var app = builder.Build();

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

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

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

In the preceding code:

- [AddRazorPages](#) adds services for Razor Pages to the app.
- [MapRazorPages](#) adds endpoints for Razor Pages to the [IEndpointRouteBuilder](#).

Consider a basic page:

C#HTML

`@page`

```
<h1>Hello, world!</h1>
<h2>The time on the server is @DateTime.Now</h2>
```

The preceding code looks a lot like a [Razor view file](#) used in an ASP.NET Core app with controllers and views. What makes it different is the `@page` directive. `@page` makes the file into an MVC action, which means that it handles requests directly, without going through a controller. `@page` must be the first Razor directive on a page. `@page` affects the behavior of other [Razor](#) constructs. Razor Pages file names have a `.cshtml` suffix.

A similar page, using a `PageModel` class, is shown in the following two files. The `Pages/Index2.cshtml` file:

CHTML

```
@page
@using RazorPagesIntro.Pages
@model Index2Model

<h2>Separate page model</h2>
<p>
    @Model.Message
</p>
```

The `Pages/Index2.cshtml.cs` page model:

C#

```
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.Extensions.Logging;
using System;

namespace RazorPagesIntro.Pages
{
    public class Index2Model : PageModel
    {
        public string Message { get; private set; } = "PageModel in C#";

        public void OnGet()
        {
            Message += $" Server time is { DateTime.Now }";
        }
    }
}
```

By convention, the `PageModel` class file has the same name as the Razor Page file with `.cs` appended. For example, the previous Razor Page is `Pages/Index2.cshtml`. The file containing the `PageModel` class is named `Pages/Index2.cshtml.cs`.

The associations of URL paths to pages are determined by the page's location in the file system. The following table shows a Razor Page path and the matching URL:

[Expand table](#)

File name and path	matching URL
<code>/Pages/Index.cshtml</code>	<code>/</code> or <code>/Index</code>

File name and path	matching URL
/Pages/Contact.cshtml	/Contact
/Pages/Store/Contact.cshtml	/Store/Contact
/Pages/Store/Index.cshtml	/Store or /Store/Index

Notes:

- The runtime looks for Razor Pages files in the *Pages* folder by default.
- `Index` is the default page when a URL doesn't include a page.

## Write a basic form

Razor Pages is designed to make common patterns used with web browsers easy to implement when building an app. [Model binding](#), [Tag Helpers](#), and HTML helpers work with the properties defined in a Razor Page class. Consider a page that implements a basic "contact us" form for the `Contact` model:

For the samples in this document, the `DbContext` is initialized in the [Program.cs](#) file.

The in memory database requires the `Microsoft.EntityFrameworkCore.InMemory` NuGet package.

C#

```
using Microsoft.EntityFrameworkCore;
using RazorPagesContacts.Data;
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddDbContext<CustomerDbContext>(options =>
    options.UseInMemoryDatabase("name"));

var app = builder.Build();

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

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

app.UseRouting();
```

```
app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

The data model:

```
C#

using System.ComponentModel.DataAnnotations;

namespace RazorPagesContacts.Models
{
    public class Customer
    {
        public int Id { get; set; }

        [Required, StringLength(10)]
        public string? Name { get; set; }
    }
}
```

The db context:

```
C#

using Microsoft.EntityFrameworkCore;

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

        public DbSet<RazorPagesContacts.Models.Customer> Customer =>
Set<RazorPagesContacts.Models.Customer>();
    }
}
```

The `Pages/Customers/Create.cshtml` view file:

```
CHTML
```

```

@page
@model RazorPagesContacts.Pages.Customers.CreateModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<p>Enter a customer name:</p>

<form method="post">
    Name:
    <input asp-for="Customer!.Name" />
    <input type="submit" />
</form>

```

The `Pages/Customers/Create.cshtml.cs` page model:

```

C#

public class CreateModel : PageModel
{
    private readonly Data.CustomerDbContext _context;

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

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

    [BindProperty]
    public Customer? Customer { get; set; }

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

        if (Customer != null) _context.Customer.Add(Customer);
        await _context.SaveChangesAsync();

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

```

By convention, the `PageModel` class is called `<PageName>Model` and is in the same namespace as the page.

The `PageModel` class allows separation of the logic of a page from its presentation. It defines page handlers for requests sent to the page and the data used to render the page. This separation allows:

- Managing of page dependencies through [dependency injection](#).
- [Unit testing](#)

The page has an `OnPostAsync` *handler method*, which runs on `POST` requests (when a user posts the form). Handler methods for any HTTP verb can be added. The most common handlers are:

- `OnGet` to initialize state needed for the page. In the preceding code, the `OnGet` method displays the `Create.cshtml` Razor Page.
- `OnPost` to handle form submissions.

The `Async` naming suffix is optional but is often used by convention for asynchronous functions. The preceding code is typical for Razor Pages.

If you're familiar with ASP.NET apps using controllers and views:

- The `OnPostAsync` code in the preceding example looks similar to typical controller code.
- Most of the MVC primitives like [model binding](#), [validation](#), and action results work the same with Controllers and Razor Pages.

The previous `OnPostAsync` method:

C#

```
[BindProperty]
public Customer? Customer { get; set; }

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

    if (Customer != null) _context.Customer.Add(Customer);
    await _context.SaveChangesAsync();

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

The basic flow of `OnPostAsync`:



Check for validation errors.

- If there are no errors, save the data and redirect.
- If there are errors, show the page again with validation messages. In many cases, validation errors would be detected on the client, and never submitted to the server.

The `Pages/Customers/Create.cshtml` view file:

CSSHTML

```
@page
@model RazorPagesContacts.Pages.Customers.CreateModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<p>Enter a customer name:</p>

<form method="post">
    Name:
    <input asp-for="Customer!.Name" />
    <input type="submit" />
</form>
```

The rendered HTML from `Pages/Customers/Create.cshtml`:

HTML

```
<p>Enter a customer name:</p>

<form method="post">
    Name:
    <input type="text" data-val="true"
        data-val-length="The field Name must be a string with a maximum
length of 10."
        data-val-length-max="10" data-val-required="The Name field is
required."
        id="Customer_Name" maxlength="10" name="Customer.Name" value=""
    />
    <input type="submit" />
    <input name="__RequestVerificationToken" type="hidden"
        value="<Antiforgery token here>" />
</form>
```

In the previous code, posting the form:

- With valid data:
  - The `OnPostAsync` handler method calls the `RedirectToPage` helper method. `RedirectToPage` returns an instance of `RedirectToPageResult`. `RedirectToPage`:

- Is an action result.
- Is similar to `RedirectToAction` or `RedirectToRoute` (used in controllers and views).
- Is customized for pages. In the preceding sample, it redirects to the root Index page (`/Index`). `RedirectToPage` is detailed in the [URL generation for Pages](#) section.
- With validation errors that are passed to the server:
  - The `OnPostAsync` handler method calls the `Page` helper method. `Page` returns an instance of `PageResult`. Returning `Page` is similar to how actions in controllers return `View`. `PageResult` is the default return type for a handler method. A handler method that returns `void` renders the page.
  - In the preceding example, posting the form with no value results in `ModelState.IsValid` returning false. In this sample, no validation errors are displayed on the client. Validation error handling is covered later in this document.

```
C#

[BindProperty]
public Customer? Customer { get; set; }

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

    if (Customer != null) _context.Customer.Add(Customer);
    await _context.SaveChangesAsync();

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

- With validation errors detected by client side validation:
  - Data is **not** posted to the server.
  - Client-side validation is explained later in this document.

The `Customer` property uses `[BindProperty]` attribute to opt in to model binding:

```
C#

[BindProperty]
public Customer? Customer { get; set; }
```

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

    if (Customer != null) _context.Customer.Add(Customer);
    await _context.SaveChangesAsync();

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

`[BindProperty]` should **not** be used on models containing properties that should not be changed by the client. For more information, see [Overposting](#).

Razor Pages, by default, bind properties only with non-`GET` verbs. Binding to properties removes the need to writing code to convert HTTP data to the model type. Binding reduces code by using the same property to render form fields (`<input asp-for="Customer.Name">`) and accept the input.

### ⚠ 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 \(YouTube\)](#).<sup>↗</sup>

Reviewing the `Pages/Customers/Create.cshtml` view file:

CSHTML

```
@page
@model RazorPagesContacts.Pages.Customers.CreateModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

```
<p>Enter a customer name:</p>
```

```
<form method="post">
```

```
    Name:
```

```
    <input asp-for="Customer!.Name" />
```

```
    <input type="submit" />
```

```
</form>
```

- In the preceding code, the **input tag helper** `<input asp-for="Customer.Name" />` binds the HTML `<input>` element to the `Customer.Name` model expression.
- `@addTagHelper` makes Tag Helpers available.

## The home page

`Index.cshtml` is the home page:

CSHTML

```
@page
```

```
@model RazorPagesContacts.Pages.Customers.IndexModel
```

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

```
<h1>Contacts home page</h1>
```

```
<form method="post">
```

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

```
        <thead>
```

```
            <tr>
```

```
                <th>ID</th>
```

```
                <th>Name</th>
```

```
            <th></th>
```

```
        </tr>
```

```
    </thead>
```

```
    <tbody>
```

```
    @if (Model.Customers != null)
```

```
    {
```

```
        foreach (var contact in Model.Customers)
```

```
        {
```

```
            <tr>
```

```
                <td> @contact.Id </td>
```

```
                <td>@contact.Name</td>
```

```
                <td>
```

```
                    <!-- <snippet_Edit> -->
```

```
                    <a asp-page="./Edit" asp-route-
```

```
id="@contact.Id">Edit</a> |
```

```
                    <!-- </snippet_Edit> -->
```

```
                    <!-- <snippet_Delete> -->
```

```
                    <button type="submit" asp-page-handler="delete" asp-  
route-id="@contact.Id">delete</button>
```

```
                    <!-- </snippet_Delete> -->
```

```
                </td>
```

```

        </tr>
    }
}
</tbody>
</table>
<a asp-page="Create">Create New</a>
</form>

```

The associated `PageModel` class (`Index.cshtml.cs`):

```

C#

public class IndexModel : PageModel
{
    private readonly Data.CustomerDbContext _context;
    public IndexModel(Data.CustomerDbContext context)
    {
        _context = context;
    }

    public IList<Customer>? Customers { get; set; }

    public async Task OnGetAsync()
    {
        Customers = await _context.Customer.ToListAsync();
    }

    public async Task<IActionResult> OnPostDeleteAsync(int id)
    {
        var contact = await _context.Customer.FindAsync(id);

        if (contact != null)
        {
            _context.Customer.Remove(contact);
            await _context.SaveChangesAsync();
        }

        return RedirectToPage();
    }
}

```

The `Index.cshtml` file contains the following markup:

```

CSHTML

<a asp-page="./Edit" asp-route-id="@contact.Id">Edit</a> |

```

The `<a /a>` [Anchor Tag Helper](#) used the `asp-route-{value}` attribute to generate a link to the Edit page. The link contains route data with the contact ID. For example,

`https://localhost:5001/Edit/1`. [Tag Helpers](#) enable server-side code to participate in creating and rendering HTML elements in Razor files.

The `Index.cshtml` file contains markup to create a delete button for each customer contact:

CSHTML

```
<button type="submit" asp-page-handler="delete" asp-route-id="@contact.Id">delete</button>
```

The rendered HTML:

HTML

```
<button type="submit" formaction="/Customers?id=1&handler=delete">delete</button>
```

When the delete button is rendered in HTML, its [formaction](#) includes parameters for:

- The customer contact ID, specified by the `asp-route-id` attribute.
- The `handler`, specified by the `asp-page-handler` attribute.

When the button is selected, a form `POST` request is sent to the server. By convention, the name of the handler method is selected based on the value of the `handler` parameter according to the scheme `OnPost[handler]Async`.

Because the `handler` is `delete` in this example, the `OnPostDeleteAsync` handler method is used to process the `POST` request. If the `asp-page-handler` is set to a different value, such as `remove`, a handler method with the name `OnPostRemoveAsync` is selected.

C#

```
public async Task<IActionResult> OnPostDeleteAsync(int id)
{
    var contact = await _context.Customer.FindAsync(id);

    if (contact != null)
    {
        _context.Customer.Remove(contact);
        await _context.SaveChangesAsync();
    }

    return RedirectToPage();
}
```

The `OnPostDeleteAsync` method:

- Gets the `id` from the query string.
- Queries the database for the customer contact with `FindAsync`.
- If the customer contact is found, it's removed and the database is updated.
- Calls `RedirectToPage` to redirect to the root Index page (`/Index`).

## The Edit.cshtml file

CSHTML

```
@page "{id:int}"
@model RazorPagesContacts.Pages.Customers.EditModel

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

<h1>Edit</h1>

<h4>Customer</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger">
            </div>
            <input type="hidden" asp-for="Customer!.Id" />
            <div class="form-group">
                <label asp-for="Customer!.Name" class="control-label">
                </label>
                <input asp-for="Customer!.Name" class="form-control" />
                <span asp-validation-for="Customer!.Name" 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-page="./Index">Back to List</a>
</div>

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

The first line contains the `@page "{id:int}"` directive. The routing constraint `"{id:int}"` tells the page to accept requests to the page that contain `int` route data. If a request to the page doesn't contain route data that can be converted to an `int`, the runtime returns an HTTP 404 (not found) error. To make the ID optional, append `?` to the route constraint:

C#HTML

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

The `Edit.cshtml.cs` file:

C#

```
public class EditModel : PageModel
{
    private readonly RazorPagesContacts.Data.CustomerDbContext _context;

    public EditModel(RazorPagesContacts.Data.CustomerDbContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Customer? Customer { get; set; }

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

        Customer = await _context.Customer.FirstOrDefaultAsync(m => m.Id ==
id);

        if (Customer == null)
        {
            return NotFound();
        }
        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();
        }
    }
}
```



```

    }

    if (Customer != null)
    {
        _context.Attach(Customer).State = EntityState.Modified;

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

    return RedirectToPage("./Index");
}

private bool CustomerExists(int id)
{
    return _context.Customer.Any(e => e.Id == id);
}
}

```

## Validation

Validation rules:

- Are declaratively specified in the model class.
- Are enforced everywhere in the app.

The [System.ComponentModel.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.

Consider the `Customer` model:

C#

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

```
namespace RazorPagesContacts.Models
{
    public class Customer
    {
        public int Id { get; set; }

        [Required, StringLength(10)]
        public string? Name { get; set; }
    }
}
```

Using the following `Create.cshtml` view file:

CSHTML

```
@page
@model RazorPagesContacts.Pages.Customers.CreateModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<p>Validation: customer name:</p>

<form method="post">
    <div asp-validation-summary="ModelOnly"></div>
    <span asp-validation-for="Customer!.Name"></span>
    Name:
    <input asp-for="Customer!.Name" />
    <input type="submit" />
</form>

<script src="~/lib/jquery/dist/jquery.js"></script>
<script src="~/lib/jquery-validation/dist/jquery.validate.js"></script>
<script src="~/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.js"></script>
```

The preceding code:

- Includes jQuery and jQuery validation scripts.
- Uses the `<div />` and `<span />` [Tag Helpers](#) to enable:
  - Client-side validation.
  - Validation error rendering.
- Generates the following HTML:

HTML

```
<p>Enter a customer name:</p>

<form method="post">
    Name:
```

```

        <input type="text" data-val="true"
            data-val-length="The field Name must be a string with a
maximum length of 10."
            data-val-length-max="10" data-val-required="The Name field
is required."
            id="Customer_Name" maxlength="10" name="Customer.Name"
value="" />
        <input type="submit" />
        <input name="__RequestVerificationToken" type="hidden"
            value="<Antiforgery token here>" />
    </form>

    <script src="/lib/jquery/dist/jquery.js"></script>
    <script src="/lib/jquery-validation/dist/jquery.validate.js"></script>
    <script src="/lib/jquery-validation-
unobtrusive/jquery.validate.unobtrusive.js"></script>

```

Posting the Create form without a name value displays the error message "The Name field is required." on the form. If JavaScript is enabled on the client, the browser displays the error without posting to the server.

The `[StringLength(10)]` attribute generates `data-val-length-max="10"` on the rendered HTML. `data-val-length-max` prevents browsers from entering more than the maximum length specified. If a tool such as [Fiddler](#) is used to edit and replay the post:

- With the name longer than 10.
- The error message "The field Name must be a string with a maximum length of 10." is returned.

Consider the following `Movie` model:

C#

```

using System;
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; }

        [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 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 space, 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 sets 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.

The Create page for the `Movie` model shows displays errors with invalid values:

The screenshot shows a web browser window with the address bar displaying `https://localhost:5001/Movies/Create`. The page title is 'Create - Movie'. The navigation bar includes 'RpMovie', 'Home', and 'Privacy'. The main heading is 'Create Movie'. The form contains the following fields and validation messages:

- Title:** Input field contains 'a'. Validation message: 'The field Title must be a string with a minimum length of 3 and a maximum length of 60.'
- Release Date:** Input field contains '00/01/0001'. Validation message: 'The Release Date field is required.'
- Genre:** Input field contains 'a'. Validation message: 'The field Genre must match the regular expression `^[A-Z]+[a-zA-Z0-9"\s-]*$`.'
- Price:** Input field contains 'Dog'. Validation message: 'The field Price must be a number.'
- Rating:** Input field contains 'z'. Validation message: 'The field Rating must match the regular expression `^[A-Z]+[a-zA-Z0-9"\s-]*$`.'

At the bottom of the form, there is a blue 'Create' button and a link labeled 'Back to List'.

For more information, see:

- [Add validation to the Movie app](#)
- [Model validation in ASP.NET Core](#).

## CSS isolation

Isolate CSS styles to individual pages, views, and components to reduce or avoid:

- Dependencies on global styles that can be challenging to maintain.
- Style conflicts in nested content.

To add a *scoped CSS file* for a page or view, place the CSS styles in a companion

`.cshtml.css` file matching the name of the `.cshtml` file. In the following example, an

`Index.cshtml.css` file supplies CSS styles that are only applied to the `Index.cshtml` page or view.

Pages/Index.cshtml.css (Razor Pages) or Views/Index.cshtml.css (MVC):

CSS

```
h1 {  
    color: red;  
}
```

CSS isolation occurs at build time. The framework rewrites CSS selectors to match markup rendered by the app's pages or views. The rewritten CSS styles are bundled and produced as a static asset, `{APP ASSEMBLY}.styles.css`. The placeholder `{APP ASSEMBLY}` is the assembly name of the project. A link to the bundled CSS styles is placed in the app's layout.

In the `<head>` content of the app's `Pages/Shared/_Layout.cshtml` (Razor Pages) or `Views/Shared/_Layout.cshtml` (MVC), add or confirm the presence of the link to the bundled CSS styles:

HTML

```
<link rel="stylesheet" href="~/ {APP ASSEMBLY}.styles.css" />
```

In the following example, the app's assembly name is `WebApp`:

HTML

```
<link rel="stylesheet" href="WebApp.styles.css" />
```

The styles defined in a scoped CSS file are only applied to the rendered output of the matching file. In the preceding example, any `h1` CSS declarations defined elsewhere in the app don't conflict with the `Index`'s heading style. CSS style cascading and inheritance rules remain in effect for scoped CSS files. For example, styles applied directly to an `<h1>` element in the `Index.cshtml` file override the scoped CSS file's styles in `Index.cshtml.css`.

#### ⓘ Note

In order to guarantee CSS style isolation when bundling occurs, importing CSS in Razor code blocks isn't supported.

CSS isolation only applies to HTML elements. CSS isolation isn't supported for [Tag Helpers](#).

Within the bundled CSS file, each page, view, or Razor component is associated with a scope identifier in the format `b-{STRING}`, where the `{STRING}` placeholder is a ten-character string generated by the framework. The following example provides the style for the preceding `<h1>` element in the `Index` page of a Razor Pages app:

CSS

```
/* /Pages/Index.cshtml.rz.scp.css */  
h1[b-3xxtam6d07] {  
    color: red;  
}
```

In the `Index` page where the CSS style is applied from the bundled file, the scope identifier is appended as an HTML attribute:

HTML

```
<h1 b-3xxtam6d07>
```

The identifier is unique to an app. At build time, a project bundle is created with the convention `{STATIC WEB ASSETS BASE PATH}/Project.lib.scp.css`, where the placeholder `{STATIC WEB ASSETS BASE PATH}` is the static web assets base path.

If other projects are utilized, such as NuGet packages or [Razor class libraries](#), the bundled file:

- References the styles using CSS imports.
- Isn't published as a static web asset of the app that consumes the styles.

## CSS preprocessor support

CSS preprocessors are useful for improving CSS development by utilizing features such as variables, nesting, modules, mixins, and inheritance. While CSS isolation doesn't natively support CSS preprocessors such as Sass or Less, integrating CSS preprocessors is seamless as long as preprocessor compilation occurs before the framework rewrites the CSS selectors during the build process. Using Visual Studio for example, configure existing preprocessor compilation as a **Before Build** task in the Visual Studio Task Runner Explorer.

Many third-party NuGet packages, such as [AspNetCore.SassCompiler](#) [↗](#), can compile SASS/SCSS files at the beginning of the build process before CSS isolation occurs, and no additional configuration is required.

# CSS isolation configuration

CSS isolation permits configuration for some advanced scenarios, such as when there are dependencies on existing tools or workflows.

## Customize scope identifier format

*In this section, the `{Pages|Views}` placeholder is either `Pages` for Razor Pages apps or `Views` for MVC apps.*

By default, scope identifiers use the format `b-{STRING}`, where the `{STRING}` placeholder is a ten-character string generated by the framework. To customize the scope identifier format, update the project file to a desired pattern:

XML

```
<ItemGroup>
  <None Update="{Pages|Views}/Index.cshtml.css" CssScope="custom-scope-
  identifier" />
</ItemGroup>
```

In the preceding example, the CSS generated for `Index.cshtml.css` changes its scope identifier from `b-{STRING}` to `custom-scope-identifier`.

Use scope identifiers to achieve inheritance with scoped CSS files. In the following project file example, a `BaseView.cshtml.css` file contains common styles across views. A `DerivedView.cshtml.css` file inherits these styles.

XML

```
<ItemGroup>
  <None Update="{Pages|Views}/BaseView.cshtml.css" CssScope="custom-scope-
  identifier" />
  <None Update="{Pages|Views}/DerivedView.cshtml.css" CssScope="custom-
  scope-identifier" />
</ItemGroup>
```

Use the wildcard (\*) operator to share scope identifiers across multiple files:

XML

```
<ItemGroup>
  <None Update="{Pages|Views}/*.cshtml.css" CssScope="custom-scope-
```



```
identifier" />
</ItemGroup>
```

## Change base path for static web assets

The scoped CSS file is generated at the root of the app. In the project file, use the `StaticWebAssetBasePath` property to change the default path. The following example places the scoped CSS file, and the rest of the app's assets, at the `_content` path:

XML

```
<PropertyGroup>
  <StaticWebAssetBasePath>_content/${PackageId}</StaticWebAssetBasePath>
</PropertyGroup>
```

## Disable automatic bundling

To opt out of how framework publishes and loads scoped files at runtime, use the `DisableScopedCssBundling` property. When using this property, other tools or processes are responsible for taking the isolated CSS files from the `obj` directory and publishing and loading them at runtime:

XML

```
<PropertyGroup>
  <DisableScopedCssBundling>true</DisableScopedCssBundling>
</PropertyGroup>
```

## Razor class library (RCL) support

When a [Razor class library \(RCL\)](#) provides isolated styles, the `<link>` tag's `href` attribute points to `{STATIC WEB ASSET BASE PATH}/{PACKAGE ID}.bundle.scoped.css`, where the placeholders are:

- `{STATIC WEB ASSET BASE PATH}`: The static web asset base path.
- `{PACKAGE ID}`: The library's [package identifier](#). The package identifier defaults to the project's assembly name if the package identifier isn't specified in the project file.

In the following example:

- The static web asset base path is `_content/ClassLib`.
- The class library's assembly name is `ClassLib`.

`Pages/Shared/_Layout.cshtml` (Razor Pages) or `Views/Shared/_Layout.cshtml` (MVC):

HTML

```
<link href="_content/ClassLib/ClassLib.bundle.scp.css" rel="stylesheet">
```

For more information on RCLs, see the following articles:

- [Reusable Razor UI in class libraries with ASP.NET Core](#)
- [Consume ASP.NET Core Razor components from a Razor class library \(RCL\)](#)

For information on Blazor CSS isolation, see [ASP.NET Core Blazor CSS isolation](#).

## Handle HEAD requests with an OnGet handler fallback

`HEAD` requests allow retrieving the headers for a specific resource. Unlike `GET` requests, `HEAD` requests don't return a response body.

Ordinarily, an `OnHead` handler is created and called for `HEAD` requests:

C#

```
public void OnHead()
{
    HttpContext.Response.Headers.Add("Head Test", "Handled by OnHead!");
}
```

Razor Pages falls back to calling the `OnGet` handler if no `OnHead` handler is defined.

## XSRF/CSRF and Razor Pages

Razor Pages are protected by [Antiforgery validation](#). The `FormTagHelper` injects antiforgery tokens into HTML form elements.

## Using Layouts, partials, templates, and Tag Helpers with Razor Pages

Pages work with all the capabilities of the Razor view engine. Layouts, partials, templates, Tag Helpers, `_ViewStart.cshtml`, and `_ViewImports.cshtml` work in the same way they do for conventional Razor views.

Let's declutter this page by taking advantage of some of those capabilities.

Add a [layout page](#) to `Pages/Shared/_Layout.cshtml`:

CSSHTML

```
<!DOCTYPE html>
<html>
<head>
    <title>RP Sample</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
</head>
<body>
    <a asp-page="/Index">Home</a>
    <a asp-page="/Customers/Create">Create</a>
    <a asp-page="/Customers/Index">Customers</a> <br />

    @RenderBody()
    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/jquery-validation/dist/jquery.validate.js"></script>
    <script src="~/lib/jquery-validation-
unobtrusive/jquery.validate.unobtrusive.js"></script>
</body>
</html>
```

The [Layout](#):

- Controls the layout of each page (unless the page opts out of layout).
- Imports HTML structures such as JavaScript and stylesheets.
- The contents of the Razor page are rendered where `@RenderBody()` is called.

For more information, see [layout page](#).

The [Layout](#) property is set in `Pages/_ViewStart.cshtml`:

CSSHTML

```
@{
    Layout = "_Layout";
}
```

The layout is in the *Pages/Shared* folder. Pages look for other views (layouts, templates, partials) hierarchically, starting in the same folder as the current page. A layout in the *Pages/Shared* folder can be used from any Razor page under the *Pages* folder.

The layout file should go in the *Pages/Shared* folder.

We recommend you **not** put the layout file in the *Views/Shared* folder. *Views/Shared* is an MVC views pattern. Razor Pages are meant to rely on folder hierarchy, not path conventions.

View search from a Razor Page includes the *Pages* folder. The layouts, templates, and partials used with MVC controllers and conventional Razor views *just work*.

Add a `Pages/_ViewImports.cshtml` file:

CSHTML

```
@namespace RazorPagesContacts.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

`@namespace` is explained later in the tutorial. The `@addTagHelper` directive brings in the [built-in Tag Helpers](#) to all the pages in the *Pages* folder.

The `@namespace` directive set on a page:

CSHTML

```
@page
@namespace RazorPagesIntro.Pages.Customers

@model NamespaceModel

<h2>Name space</h2>
<p>
    @Model.Message
</p>
```

The `@namespace` directive sets the namespace for the page. The `@model` directive doesn't need to include the namespace.

When the `@namespace` directive is contained in `_ViewImports.cshtml`, the specified namespace supplies the prefix for the generated namespace in the Page that imports the `@namespace` directive. The rest of the generated namespace (the suffix portion) is the dot-separated relative path between the folder containing `_ViewImports.cshtml` and the folder containing the page.

For example, the `PageModel` class `Pages/Customers/Edit.cshtml.cs` explicitly sets the namespace:

C#

```

namespace RazorPagesContacts.Pages
{
    public class EditModel : PageModel
    {
        private readonly ApplicationDbContext _db;

        public EditModel(ApplicationDbContext db)
        {
            _db = db;
        }

        // Code removed for brevity.
    }
}

```

The `Pages/_ViewImports.cshtml` file sets the following namespace:

CSHTML

```

@namespace RazorPagesContacts.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

```

The generated namespace for the `Pages/Customers/Edit.cshtml` Razor Page is the same as the `PageModel` class.

`@namespace` also works with conventional Razor views.

Consider the `Pages/Customers/Create.cshtml` view file:

CSHTML

```

@page
@model RazorPagesContacts.Pages.Customers.CreateModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<p>Validation: customer name:</p>

<form method="post">
    <div asp-validation-summary="ModelOnly"></div>
    <span asp-validation-for="Customer!.Name"></span>
    Name:
    <input asp-for="Customer!.Name" />
    <input type="submit" />
</form>

<script src="~/lib/jquery/dist/jquery.js"></script>
<script src="~/lib/jquery-validation/dist/jquery.validate.js"></script>
<script src="~/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.js"></script>

```

The updated `Pages/Customers/Create.cshtml` view file with `_ViewImports.cshtml` and the preceding layout file:

CSSHTML

```
@page
@model CreateModel

<p>Enter a customer name:</p>

<form method="post">
    Name:
    <input asp-for="Customer!.Name" />
    <input type="submit" />
</form>
```

In the preceding code, the `_ViewImports.cshtml` imported the namespace and Tag Helpers. The layout file imported the JavaScript files.

The [Razor Pages starter project](#) contains the `Pages/_ValidationScriptsPartial.cshtml`, which hooks up client-side validation.

For more information on partial views, see [Partial views in ASP.NET Core](#).

## URL generation for Pages

The `Create` page, shown previously, uses `RedirectToPage`:

C#

```
public class CreateModel : PageModel
{
    private readonly Data.CustomerDbContext _context;

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

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

    [BindProperty]
    public Customer? Customer { get; set; }

    public async Task<IActionResult> OnPostAsync()
    {

```

```

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

        if (Customer != null) _context.Customer.Add(Customer);
        await _context.SaveChangesAsync();

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

```

The app has the following file/folder structure:

- */Pages*
  - Index.cshtml
  - Privacy.cshtml
  - */Customers*
    - Create.cshtml
    - Edit.cshtml
    - Index.cshtml

The `Pages/Customers/Create.cshtml` and `Pages/Customers/Edit.cshtml` pages redirect to `Pages/Customers/Index.cshtml` after success. The string `./Index` is a relative page name used to access the preceding page. It is used to generate URLs to the `Pages/Customers/Index.cshtml` page. For example:

- `Url.Page("./Index", ...)`
- `<a asp-page="./Index">Customers Index Page</a>`
- `RedirectToPage("./Index")`

The absolute page name `/Index` is used to generate URLs to the `Pages/Index.cshtml` page. For example:

- `Url.Page("/Index", ...)`
- `<a asp-page="/Index">Home Index Page</a>`
- `RedirectToPage("/Index")`

The page name is the path to the page from the root `/Pages` folder including a leading `/` (for example, `/Index`). The preceding URL generation samples offer enhanced options and functional capabilities over hard-coding a URL. URL generation uses [routing](#) and

can generate and encode parameters according to how the route is defined in the destination path.

URL generation for pages supports relative names. The following table shows which Index page is selected using different `RedirectToPage` parameters in `Pages/Customers/Create.cshtml`.

 Expand table

<b>RedirectToPage(x)</b>	<b>Page</b>
<code>RedirectToPage("/Index")</code>	<i>Pages/Index</i>
<code>RedirectToPage("../Index");</code>	<i>Pages/Customers/Index</i>
<code>RedirectToPage("./Index")</code>	<i>Pages/Index</i>
<code>RedirectToPage("Index")</code>	<i>Pages/Customers/Index</i>

`RedirectToPage("Index")`, `RedirectToPage("../Index")`, and `RedirectToPage("./Index")` are *relative names*. The `RedirectToPage` parameter is *combined* with the path of the current page to compute the name of the destination page.

Relative name linking is useful when building sites with a complex structure. When relative names are used to link between pages in a folder:

- Renaming a folder doesn't break the relative links.
- Links are not broken because they don't include the folder name.

To redirect to a page in a different [Area](#), specify the area:

C#

```
RedirectToPage("/Index", new { area = "Services" });
```

For more information, see [Areas in ASP.NET Core](#) and [Razor Pages route and app conventions in ASP.NET Core](#).

## ViewData attribute

Data can be passed to a page with [ViewDataAttribute](#). Properties with the `[ViewData]` attribute have their values stored and loaded from the [ViewDataDictionary](#).



In the following example, the `AboutModel` applies the `[ViewData]` attribute to the `Title` property:

C#

```
public class AboutModel : PageModel
{
    [ViewData]
    public string Title { get; } = "About";

    public void OnGet()
    {
    }
}
```

In the About page, access the `Title` property as a model property:

C#HTML

```
<h1>@Model.Title</h1>
```

In the layout, the title is read from the ViewData dictionary:

C#HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>@ViewData["Title"] - WebApplication</title>
    ...
</head>
</html>
```

## TempData

ASP.NET Core exposes the `TempData`. This property stores data until it's read. The `Keep` and `Peek` methods can be used to examine the data without deletion. `TempData` is useful for redirection, when data is needed for more than a single request.

The following code sets the value of `Message` using `TempData`:

C#

```
public class CreateDotModel : PageModel
{
    private readonly ApplicationDbContext _db;

    public CreateDotModel(ApplicationDbContext db)
```

```

{
    _db = db;
}

[TempData]
public string Message { get; set; }

[BindProperty]
public Customer Customer { get; set; }

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

    _db.Customers.Add(Customer);
    await _db.SaveChangesAsync();
    Message = $"Customer {Customer.Name} added";
    return RedirectToPage("./Index");
}
}

```

The following markup in the `Pages/Customers/Index.cshtml` file displays the value of `Message` using `TempData`.

CSHTML

```
<h3>Msg: @Model.Message</h3>
```

The `Pages/Customers/Index.cshtml.cs` page model applies the `[TempData]` attribute to the `Message` property.

C#

```

[TempData]
public string Message { get; set; }

```

For more information, see [TempData](#).

## Multiple handlers per page

The following page generates markup for two handlers using the `asp-page-handler` Tag Helper:

CSHTML

```

@page
@model CreateFATHModel

<html>
<body>
    <p>
        Enter your name.
    </p>
    <div asp-validation-summary="All"></div>
    <form method="POST">
        <div><label>Name: <input asp-for="Customer.Name" /></label></div>
        <!-- <snippet_Handlers> -->
        <input type="submit" asp-page-handler="JoinList" value="Join" />
        <input type="submit" asp-page-handler="JoinListUC" value="JOIN UC"
    />
        <!-- </snippet_Handlers> -->
    </form>
</body>
</html>

```

The form in the preceding example has two submit buttons, each using the `FormActionTagHelper` to submit to a different URL. The `asp-page-handler` attribute is a companion to `asp-page`. `asp-page-handler` generates URLs that submit to each of the handler methods defined by a page. `asp-page` isn't specified because the sample is linking to the current page.

The page model:

```

C#

using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesContacts.Data;

namespace RazorPagesContacts.Pages.Customers
{
    public class CreateFATHModel : PageModel
    {
        private readonly ApplicationDbContext _db;

        public CreateFATHModel(ApplicationDbContext db)
        {
            _db = db;
        }

        [BindProperty]
        public Customer Customer { get; set; }

        public async Task<IActionResult> OnPostJoinListAsync()

```

```

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

        _db.Customers.Add(Customer);
        await _db.SaveChangesAsync();
        return RedirectToPage("/Index");
    }

    public async Task<IActionResult> OnPostJoinListUCAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }
        Customer.Name = Customer.Name?.ToUpperInvariant();
        return await OnPostJoinListAsync();
    }
}

```

The preceding code uses *named handler methods*. Named handler methods are created by taking the text in the name after `On<HTTP Verb>` and before `Async` (if present). In the preceding example, the page methods are `OnPostJoinListAsync` and `OnPostJoinListUCAsync`. With `OnPost` and `Async` removed, the handler names are `JoinList` and `JoinListUC`.

CSHTML

```

<input type="submit" asp-page-handler="JoinList" value="Join" />
<input type="submit" asp-page-handler="JoinListUC" value="JOIN UC" />

```

Using the preceding code, the URL path that submits to `OnPostJoinListAsync` is `https://localhost:5001/Customers/CreateFATH?handler=JoinList`. The URL path that submits to `OnPostJoinListUCAsync` is `https://localhost:5001/Customers/CreateFATH?handler=JoinListUC`.

## Custom routes

Use the `@page` directive to:

- Specify a custom route to a page. For example, the route to the About page can be set to `/Some/Other/Path` with `@page "/Some/Other/Path"`.

- Append segments to a page's default route. For example, an "item" segment can be added to a page's default route with `@page "item"`.
- Append parameters to a page's default route. For example, an ID parameter, `id`, can be required for a page with `@page "{id}"`.

A root-relative path designated by a tilde (~) at the beginning of the path is supported. For example, `@page "~/Some/Other/Path"` is the same as `@page "/Some/Other/Path"`.

If you don't like the query string `?handler=JoinList` in the URL, change the route to put the handler name in the path portion of the URL. The route can be customized by adding a route template enclosed in double quotes after the `@page` directive.

Cshtml

```
@page "{handler?}"
@model CreateRouteModel

<html>
<body>
  <p>
    Enter your name.
  </p>
  <div asp-validation-summary="All"></div>
  <form method="POST">
    <div><label>Name: <input asp-for="Customer.Name" /></label></div>
    <input type="submit" asp-page-handler="JoinList" value="Join" />
    <input type="submit" asp-page-handler="JoinListUC" value="JOIN UC" />
  </form>
</body>
</html>
```

Using the preceding code, the URL path that submits to `OnPostJoinListAsync` is `https://localhost:5001/Customers/CreateFATH/JoinList`. The URL path that submits to `OnPostJoinListUCAsync` is `https://localhost:5001/Customers/CreateFATH/JoinListUC`.

The `?` following `handler` means the route parameter is optional.

## Collocation of JavaScript (JS) files

Collocation of JavaScript (JS) files for pages and views is a convenient way to organize scripts in an app.

Collocate JS files using the following filename extension conventions:

- Pages of Razor Pages apps and views of MVC apps: `.cshtml.js`. Examples:

- `Pages/Index.cshtml.js` for the `Index` page of a Razor Pages app at `Pages/Index.cshtml`.
- `Views/Home/Index.cshtml.js` for the `Index` view of an MVC app at `Views/Home/Index.cshtml`.

Collocated JS files are publicly addressable using the *path to the file in the project*:

- Pages and views from a collocated scripts file in the app:

```
{PATH}/{PAGE, VIEW, OR COMPONENT}.{EXTENSION}.js
```

- The `{PATH}` placeholder is the path to the page, view, or component.
- The `{PAGE, VIEW, OR COMPONENT}` placeholder is the page, view, or component.
- The `{EXTENSION}` placeholder matches the extension of the page, view, or component, either `razor` or `cshtml`.

Razor Pages example:

A JS file for the `Index` page is placed in the `Pages` folder (`Pages/Index.cshtml.js`) next to the `Index` page (`Pages/Index.cshtml`). In the `Index` page, the script is referenced at the path in the `Pages` folder:

razor

```
@section Scripts {
    <script src="~/Pages/Index.cshtml.js"></script>
}
```

The default layout `Pages/Shared/_Layout.cshtml` can be configured to include collocated JS files, eliminating the need to configure each page individually:

razor

```
<script asp-src-include="@((ViewContext.View.Path).js)"></script>
```

The [sample download](#) <sup>↗</sup> uses the preceding code snippet to include collocated JS files in the default layout.

When the app is published, the framework automatically moves the script to the web root. In the preceding example, the script is moved to `bin\Release\{TARGET FRAMEWORK MONIKER}\publish\wwwroot\Pages\Index.cshtml.js`, where the `{TARGET FRAMEWORK MONIKER}` placeholder is the [Target Framework Moniker \(TFM\)](#). No change is required to the script's relative URL in the `Index` page.

When the app is published, the framework automatically moves the script to the web root. In the preceding example, the script is moved to `bin\Release\{TARGET FRAMEWORK MONIKER}\publish\wwwroot\Components\Pages\Index.razor.js`, where the `{TARGET FRAMEWORK MONIKER}` placeholder is the [Target Framework Moniker \(TFM\)](#). No change is required to the script's relative URL in the `Index` component.

- For scripts provided by a Razor class library (RCL):

```
_content/{PACKAGE ID}/{PATH}/{PAGE, VIEW, OR COMPONENT}.{EXTENSION}.js
```

- The `{PACKAGE ID}` placeholder is the RCL's package identifier (or library name for a class library referenced by the app).
- The `{PATH}` placeholder is the path to the page, view, or component. If a Razor component is located at the root of the RCL, the path segment isn't included.
- The `{PAGE, VIEW, OR COMPONENT}` placeholder is the page, view, or component.
- The `{EXTENSION}` placeholder matches the extension of page, view, or component, either `razor` or `cshtml`.

## Advanced configuration and settings

The configuration and settings in following sections is not required by most apps.

To configure advanced options, use the [AddRazorPages](#) overload that configures [RazorPagesOptions](#):

C#

```
using Microsoft.EntityFrameworkCore;
using RazorPagesContacts.Data;
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages(options =>
{
    options.RootDirectory = "/MyPages";
    options.Conventions.AuthorizeFolder("/MyPages/Admin");
});

builder.Services.AddDbContext<CustomerDbContext>(options =>
    options.UseInMemoryDatabase("name"));

var app = builder.Build();

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

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

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

Use the [RazorPagesOptions](#) to set the root directory for pages, or add application model conventions for pages. For more information on conventions, see [Razor Pages authorization conventions](#).

To precompile views, see [Razor view compilation](#).

## Specify that Razor Pages are at the content root

By default, Razor Pages are rooted in the */Pages* directory. Add [WithRazorPagesAtContentRoot](#) to specify that your Razor Pages are at the [content root](#) ([ContentRootPath](#)) of the app:

C#

```
using Microsoft.EntityFrameworkCore;
using RazorPagesContacts.Data;
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages(options =>
{
    options.Conventions.AuthorizeFolder("/MyPages/Admin");
})
    .WithRazorPagesAtContentRoot();

builder.Services.AddDbContext<CustomerDbContext>(options =>
    options.UseInMemoryDatabase("name"));

var app = builder.Build();

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

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



```
app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

## Specify that Razor Pages are at a custom root directory

Add [WithRazorPagesRoot](#) to specify that Razor Pages are at a custom root directory in the app (provide a relative path):

C#

```
using Microsoft.EntityFrameworkCore;
using RazorPagesContacts.Data;
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages(options =>
{
    options.Conventions.AuthorizeFolder("/MyPages/Admin");
})
.WithRazorPagesRoot("/path/to/razor/pages");

builder.Services.AddDbContext<CustomerDbContext>(options =>
    options.UseInMemoryDatabase("name"));

var app = builder.Build();

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

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


app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

## Additional resources

- See [Get started with Razor Pages](#), which builds on this introduction.
- [Authorize attribute and Razor Pages](#)
- [Download or view sample code](#) 
- [Overview of ASP.NET Core](#)
- [Razor syntax reference for ASP.NET Core](#)
- [Areas in ASP.NET Core](#)
- [Tutorial: Get started with Razor Pages in ASP.NET Core](#)
- [Razor Pages authorization conventions in ASP.NET Core](#)
- [Razor Pages route and app conventions in ASP.NET Core](#)
- [Razor Pages unit tests in ASP.NET Core](#)
- [Partial views in ASP.NET Core](#)

# 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

☆ | W | ⋮

RpMovie   Home   Privacy

# Index

[Create New](#)

All ▾

Title:

Filter

Title	Release Date	Genre	Price	Rating	
Ghostbusters	3/13/1984	Comedy	\$8.99	G	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Ghostbusters 2	2/23/1986	Comedy	\$9.99	G	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>

© 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](#).

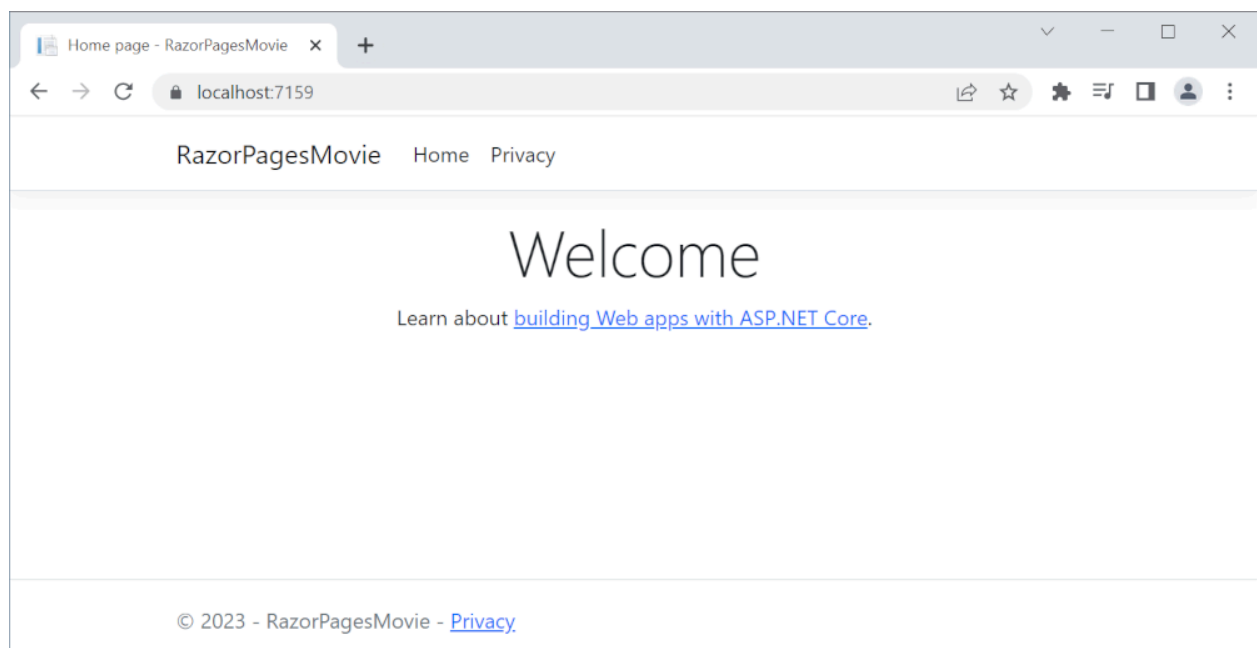
By [Rick Anderson](#) 

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](#).

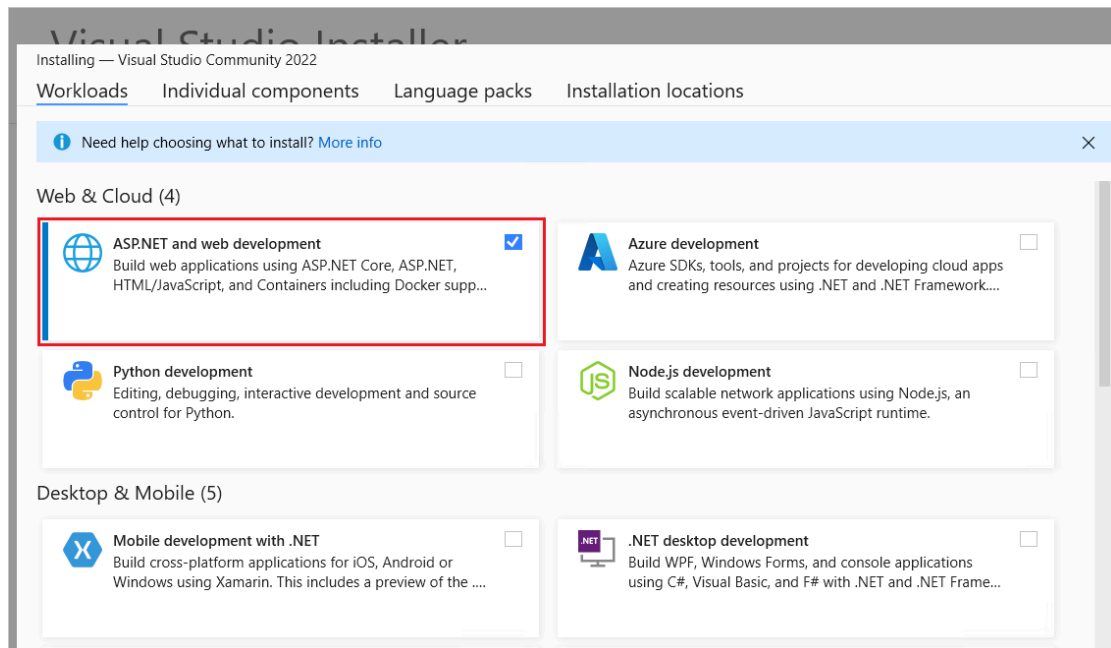
At the end of this tutorial, you'll have a Razor Pages web app that manages a database of movies.



# 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**.

Additional information

ASP.NET Core Web App (Razor Pages) C# Linux macOS Windows Cloud Service Web

Framework ⓘ  
.NET 9.0 (Preview)

Authentication type ⓘ  
None

☒ Configure for HTTPS ⓘ  
☐ Enable container support ⓘ

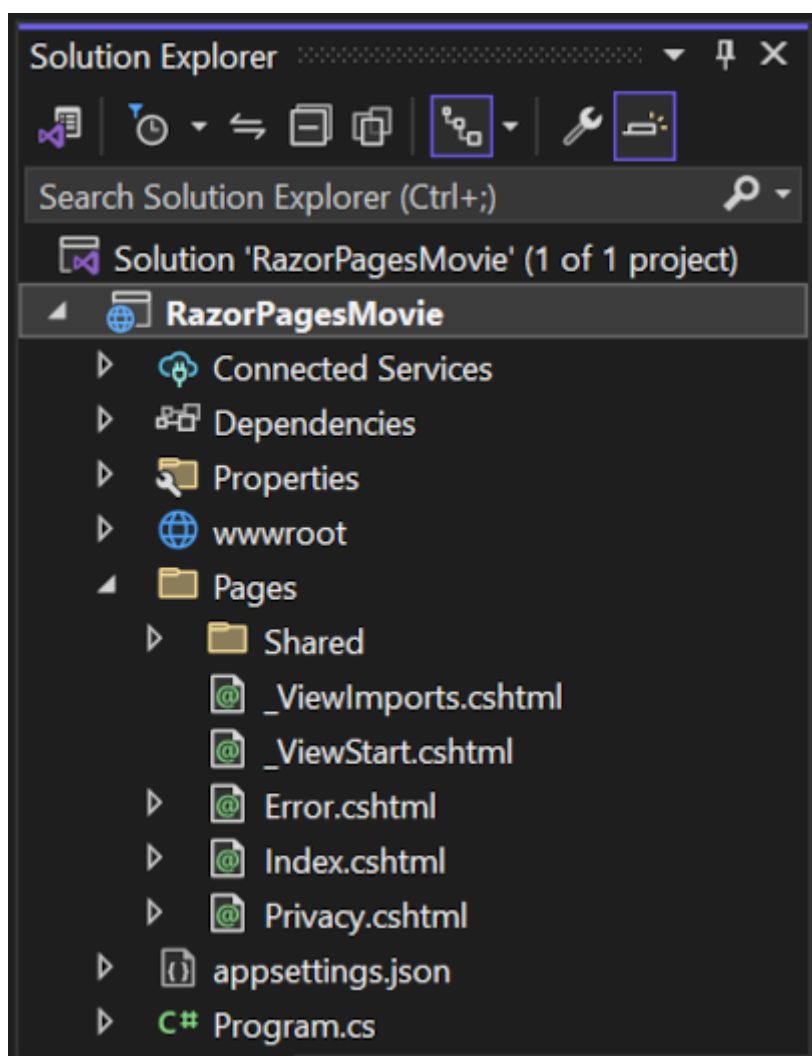
Container OS ⓘ  
Linux

Container build type ⓘ  
Dockerfile

☐ Do not use top-level statements ⓘ

Back Create

The following starter project is created:





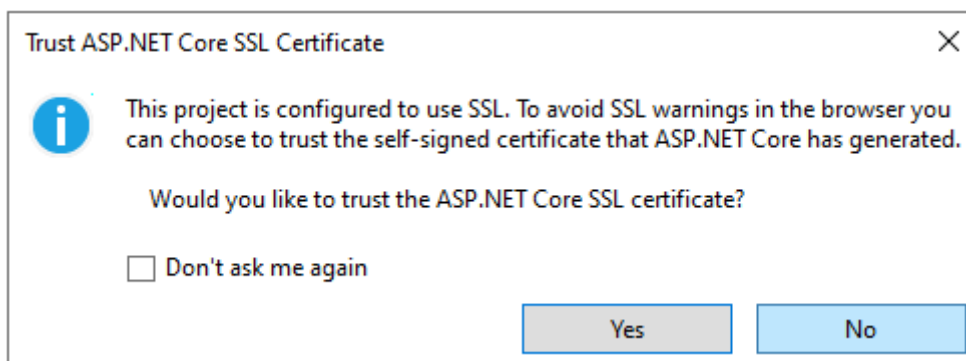
For alternative approaches to create the project, see [Create a new project in Visual Studio](#).

## Run the app

Visual Studio

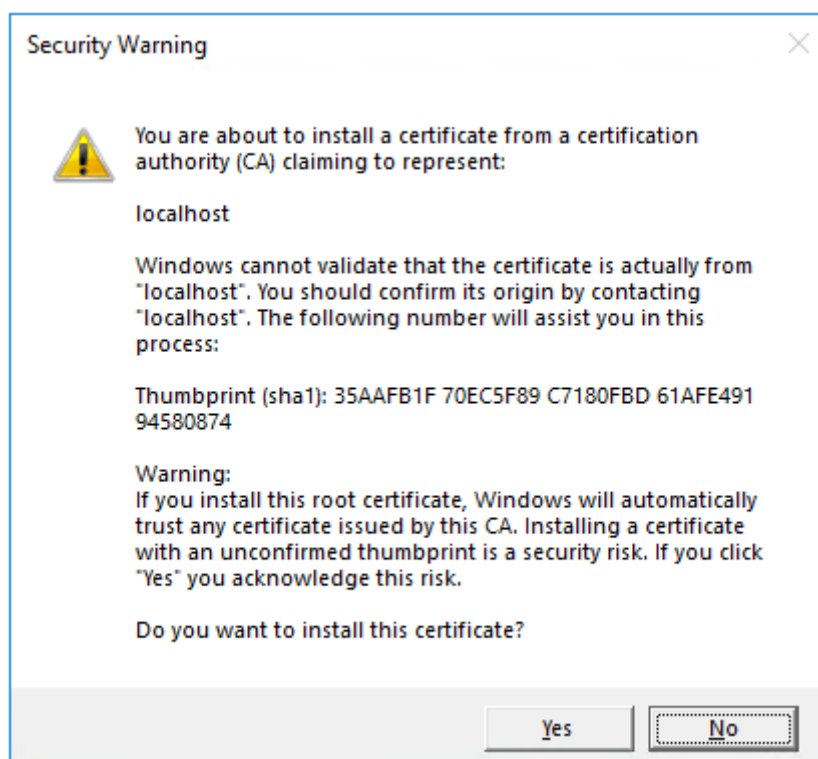
Select **RazorPagesMovie** in **Solution Explorer**, and then press **Ctrl** + **F5** to run the app without the debugger.

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 "Plain-Old CLR Objects") 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.

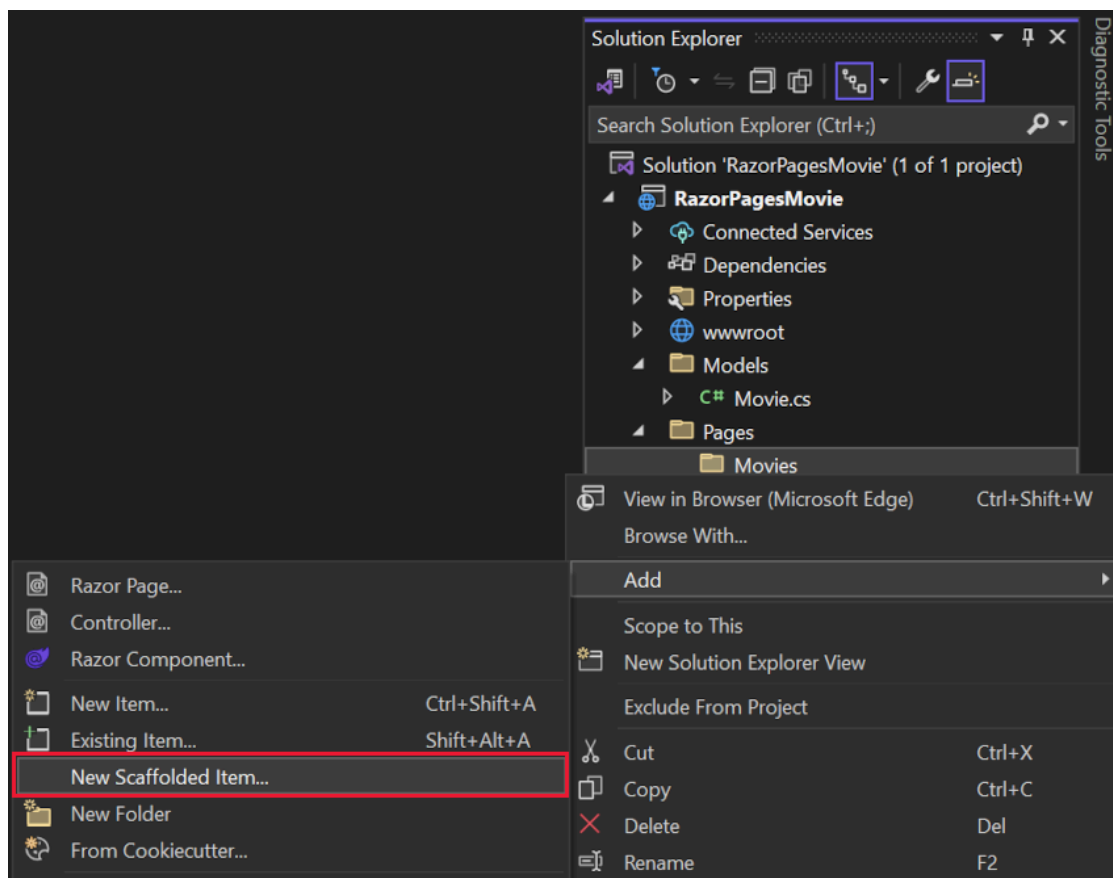
Build the project to verify there are no compilation errors.

## Scaffold the movie model

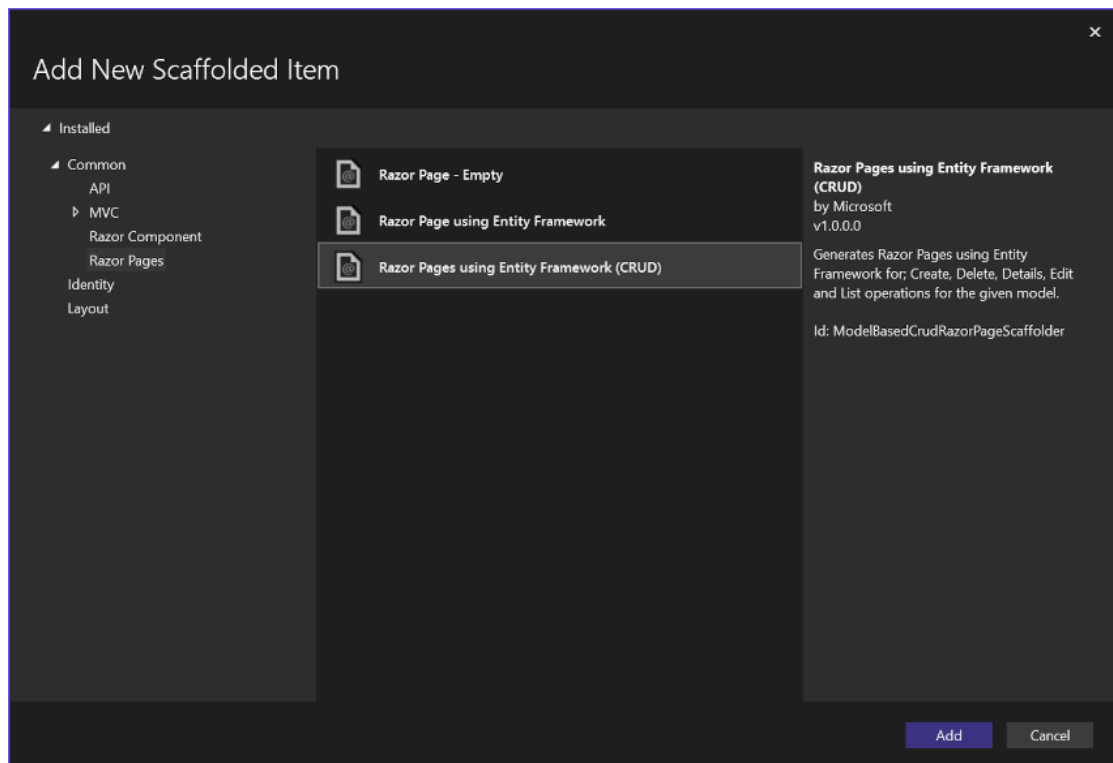
In this section, the movie model is scaffolded. That is, the scaffolding tool produces pages for Create, Read, Update, and Delete (CRUD) operations for the movie model.

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**.

**Add Razor Pages using Entity Framework (CRUD)**

Generates Razor Pages using Entity Framework for; Create, Delete, Details, Edit and List operations for the selected model.

**Model class**: `Movie (RazorPagesMovie.Models)`

**DbContext class**: `RazorPagesMovie.Data.RazorPagesMovieContext` +

**Database provider**: `SQL Server`

**Options**

- ☐ Create as a partial view
- ☒ Reference script libraries
- ☒ Use a layout page

\_\_\_\_\_ ...

(Leave empty if it is set in a Razor \_viewstart file)

**Add** **Cancel**

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("RazorPagesMovieContext") ?? 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

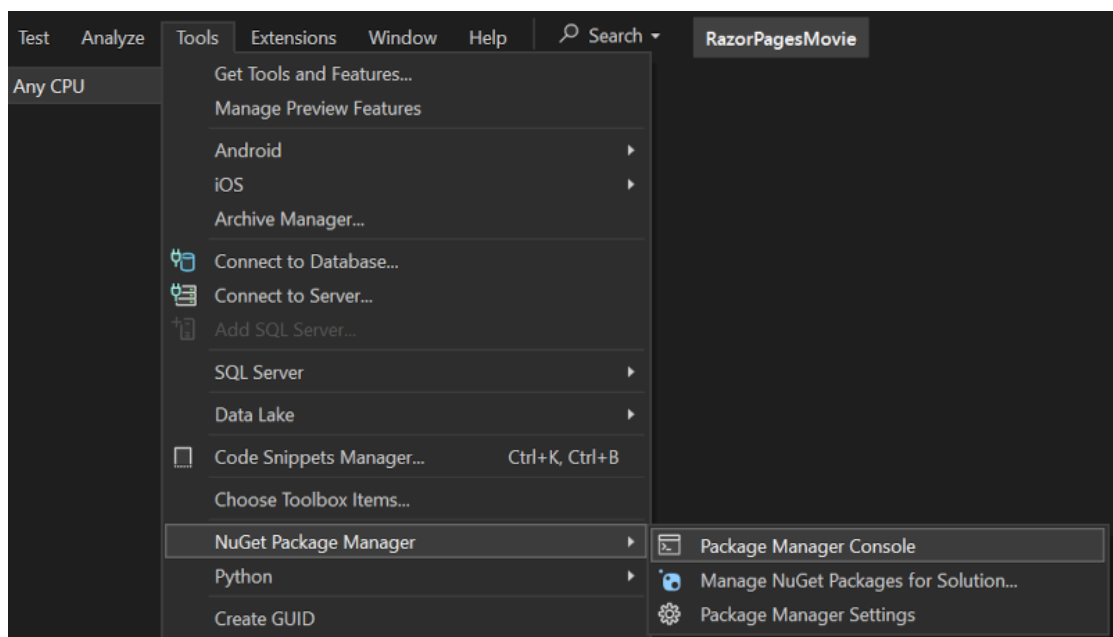
The migrations feature in Entity Framework Core provides a way to:

- 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.

Create - RazorPagesMovie

localhost:7159/Movies/Create

RazorPagesMovie Home Privacy

## Create Movie

Title

The Good, the Bad, and the Ugly

ReleaseDate

11/30/2018

Genre

Western

Price

1.19

Create

[Back to List](#)

© 2023 - RazorPagesMovie - [Privacy](#)

### ⚠ 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("RazorPagesMovieContext") ?? 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](#) <sup>↗</sup> ([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 `ActionResult` or `Task<ActionResult>`, a return statement must be provided. For example, the `Pages/Movies/Create.cshtml.cs OnPostAsync` method:

```

C#

public async Task<ActionResult> 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

C#HTML

```
@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:

C#HTML

```
@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>
```

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

CSSHTML

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

- Replace the preceding element with the following markup:

CSSHTML

```
<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.

- Save the changes and test the app by selecting the **RpMovie** link. See the [\\_Layout.cshtml](#) file in GitHub if you have any problems.
- 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:

CSSHTML

```
@{
    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>`

```

1  @page
2  @model RazorPagesMovie.Pages.Movies.CreateModel
3
4  @{
5      ViewData["Title"] = "Create";
6  }
7
8  <h1>Create</h1>
9
10 <h4>Movie</h4>
11 <hr />
12 <div class="row">
13     <div class="col-md-4">
14         <form method="post">
15             <div asp-validation-summary="ModelOnly" class="text-danger"></div>
16             <div class="form-group">
17                 <label asp-for="Movie.Title" class="control-label"></label>
18                 <input asp-for="Movie.Title" class="form-control" />
19                 <span asp-validation-for="Movie.Title" class="text-danger"></span>
20             </div>
21             <div class="form-group">
22                 <label asp-for="Movie.ReleaseDate" class="control-label"></label>
23                 <input asp-for="Movie.ReleaseDate" class="form-control" />
24                 <span asp-validation-for="Movie.ReleaseDate" class="text-danger"></span>
25             </div>
26             <div class="form-group">
27                 <label asp-for="Movie.Genre" class="control-label"></label>
28                 <input asp-for="Movie.Genre" class="form-control" />
29                 <span asp-validation-for="Movie.Genre" class="text-danger"></span>
30             </div>
31             <div class="form-group">
32                 <label asp-for="Movie.Price" class="control-label"></label>
33                 <input asp-for="Movie.Price" class="form-control" />
34                 <span asp-validation-for="Movie.Price" class="text-danger"></span>
35             </div>
36             <div class="form-group">
37                 <input type="submit" value="Create" class="btn btn-primary" />
38             </div>

```

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:

CSSHTML

```
<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("RazorPagesMovieContext") ?? 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

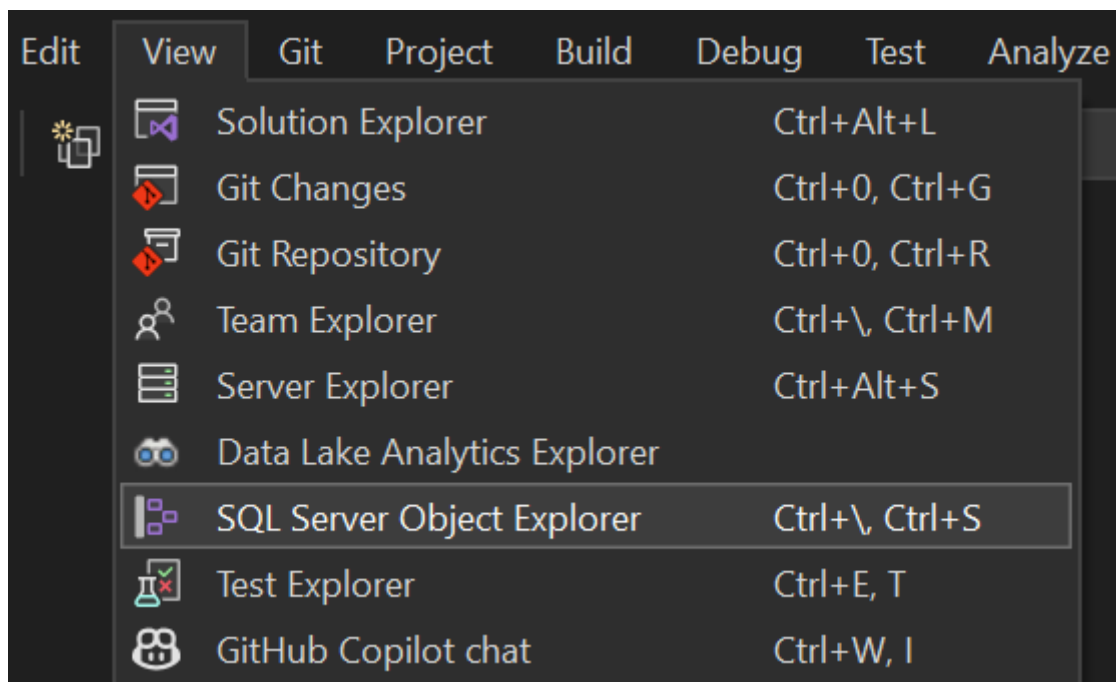
This article uses a local database that doesn't require the user to be authenticated. Production apps should use the most secure authentication flow available. For more information on authentication for deployed test and production apps, see [Secure authentication flows](#).

Visual Studio

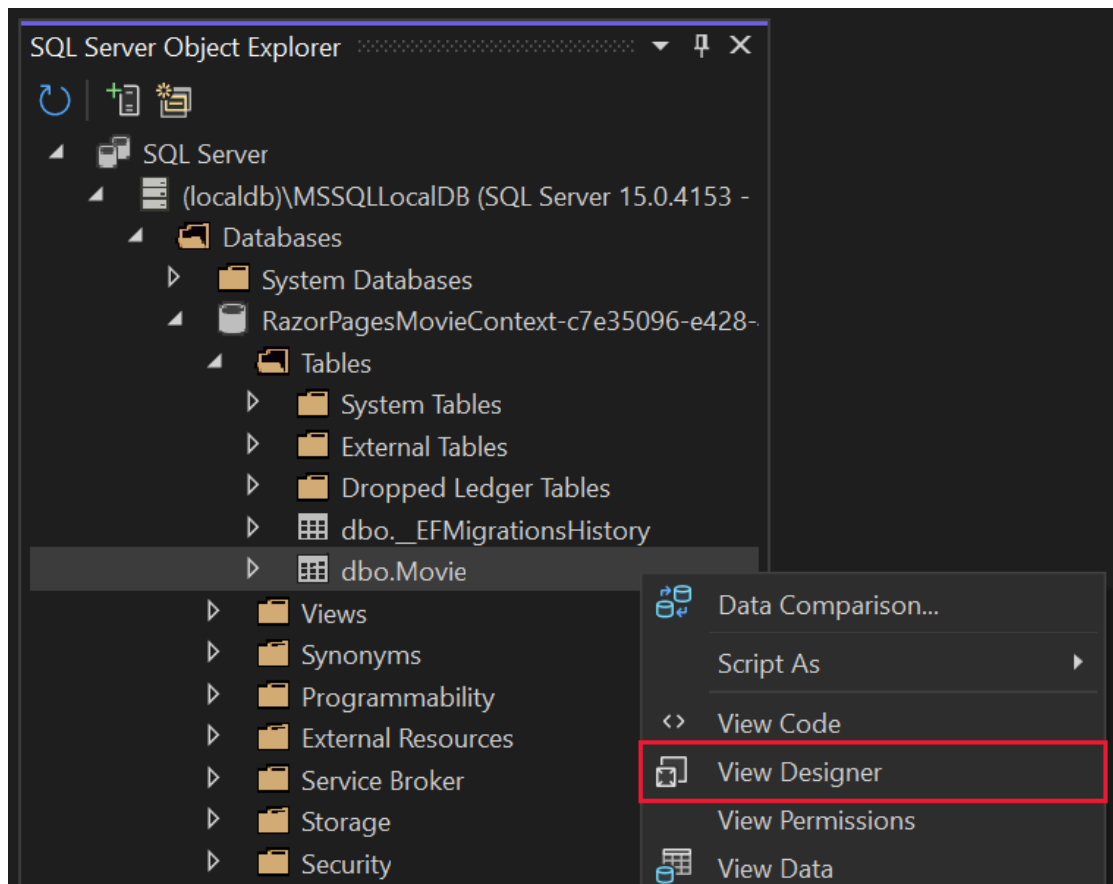
## SQL Server Express LocalDB

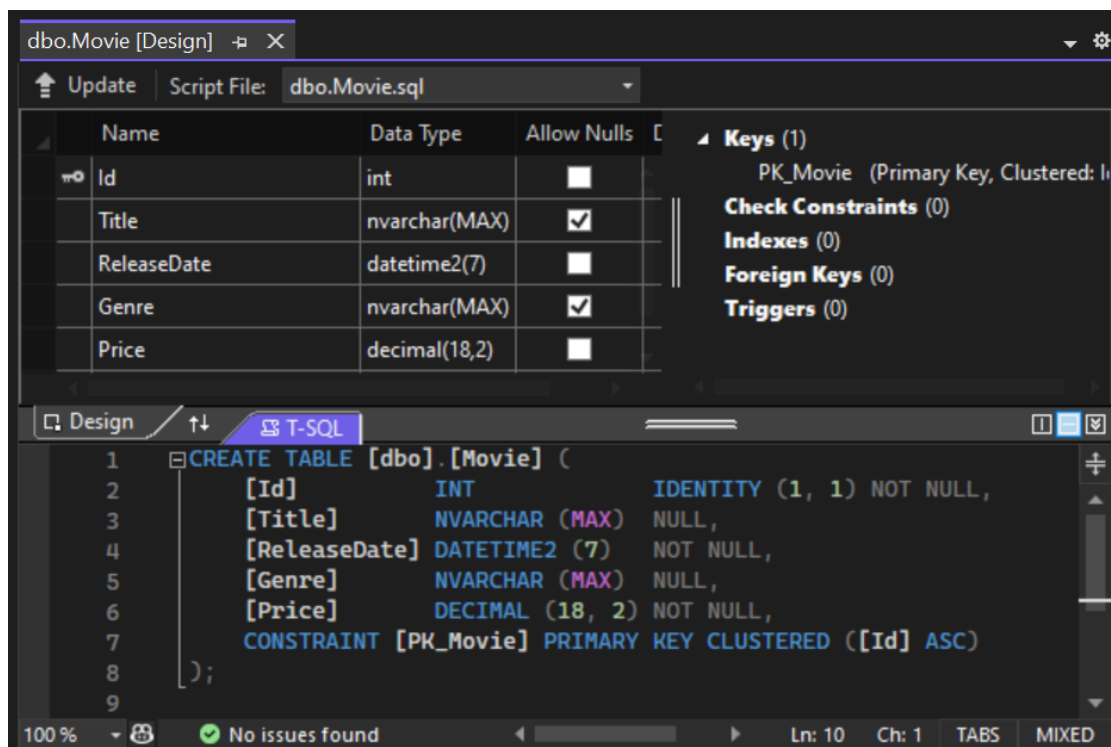
LocalDB is a lightweight version of the SQL Server Express database engine that's targeted for program development. LocalDB starts on demand and runs in user mode, so there's no complex configuration. By default, LocalDB database creates \*.mdf files in the C:\Users\<user>\ directory.

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.

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

The screenshot shows the 'View Data' view of the 'dbo.Movie' table. The table contains 4 rows of data. The first row has Id=2, Title='Conan', ReleaseDate='10/13/2023 12:0...', Genre='Action', and Price='1.99'. The second row has Id=3, Title='Back to the Fut...', ReleaseDate='8/25/2020 12:00...', Genre='Comedy', and Price='1.99'. The third row has Id=4, Title='The Good, the ...', ReleaseDate='10/16/2018 12:0...', Genre='Wester', and Price='1.19'. The fourth row is NULL.

	Id	Title	ReleaseDate	Genre	Price
▶	2	Conan	10/13/2023 12:0...	Action	1.99
	3	Back to the Fut...	8/25/2020 12:00...	Comedy	1.99
	4	The Good, the ...	10/16/2018 12:0...	Wester	1.19
⚙	NULL	NULL	NULL	NULL	NULL

## 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("RazorPagesMovieContext") ?? 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.

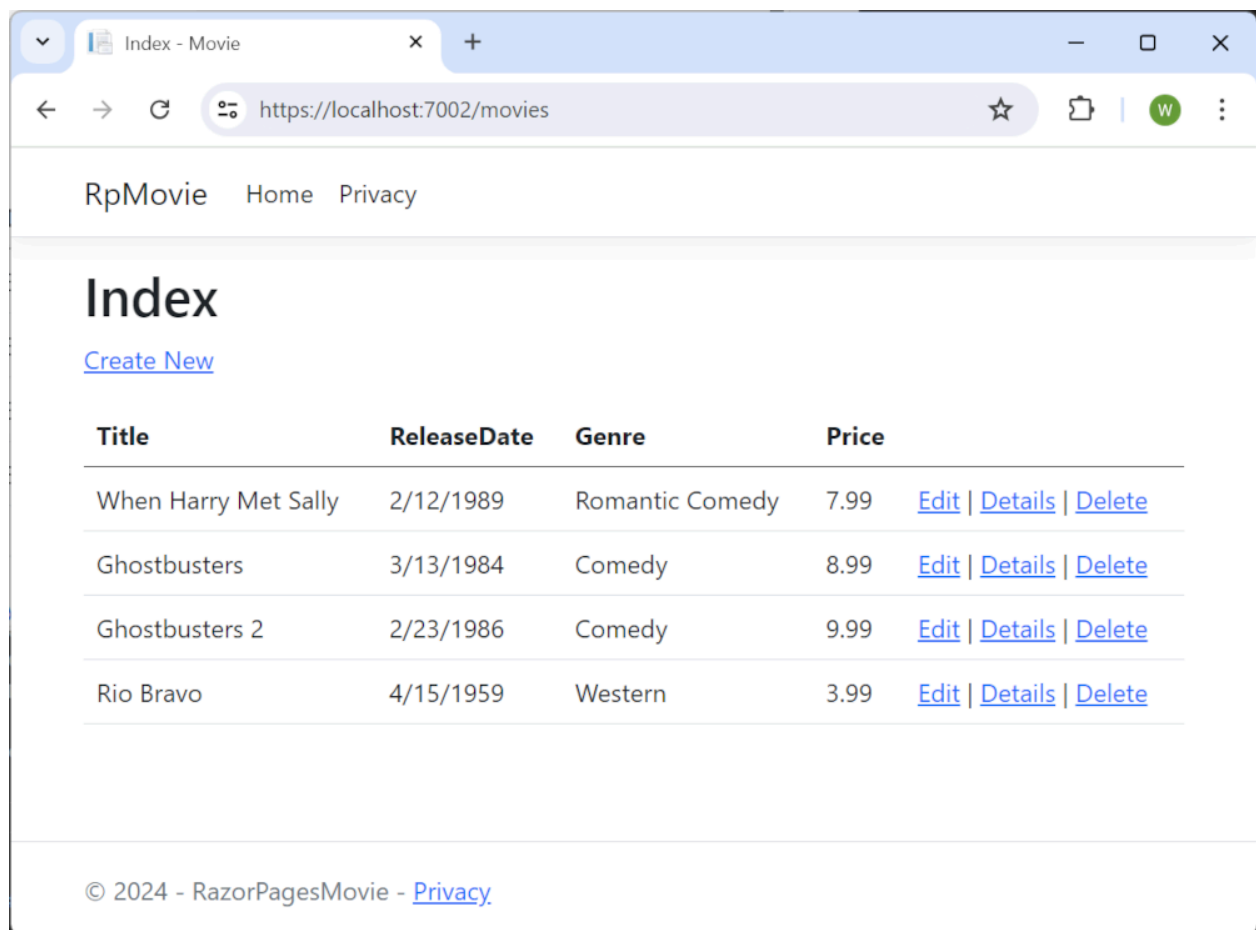
The following exception occurs when `Update-Database` has not been run:

```
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

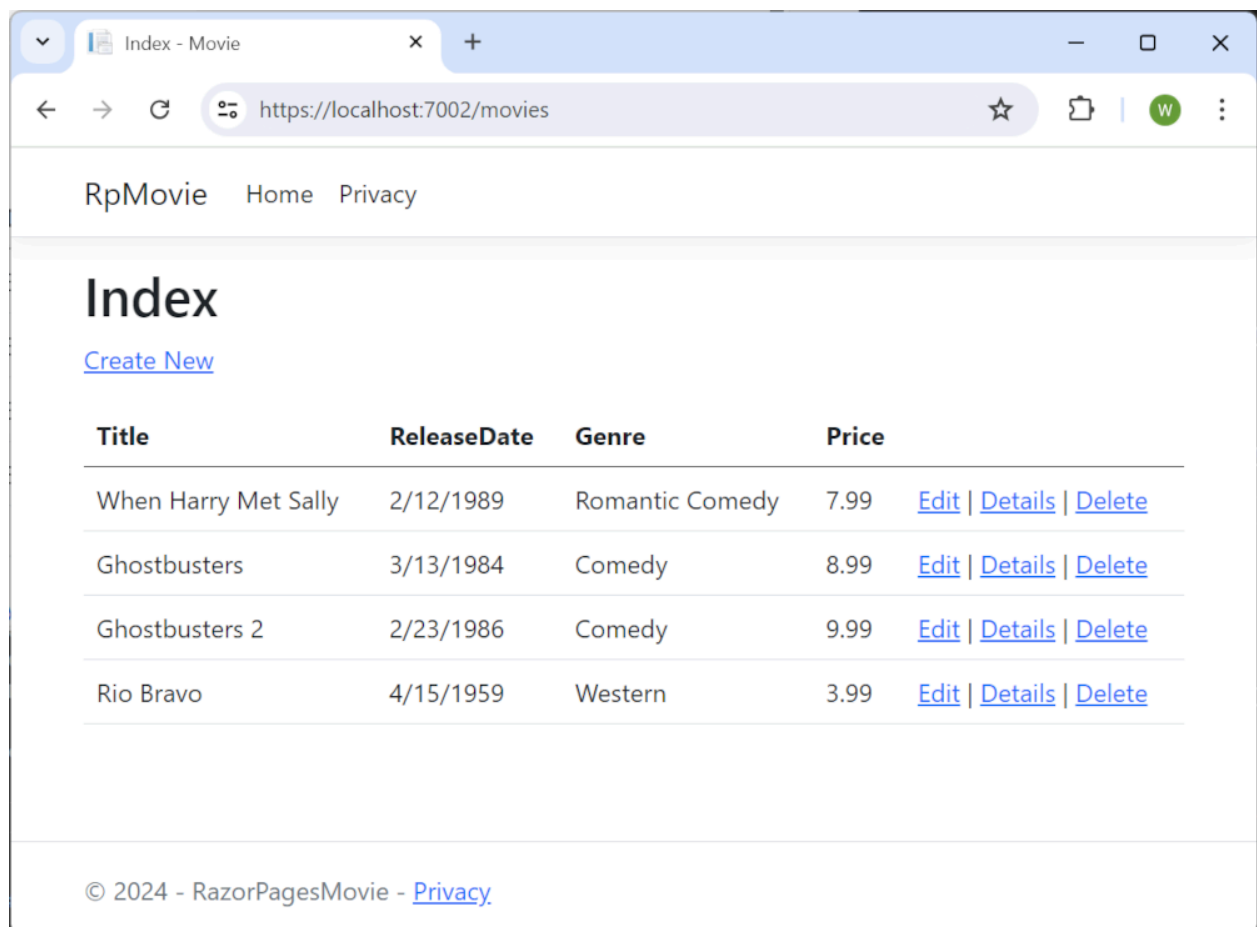
Article • 07/01/2024

## 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.

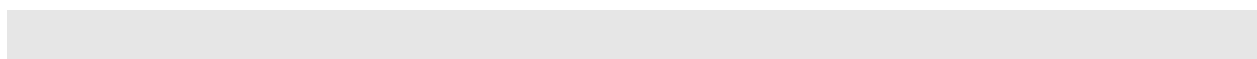
For the current release, see the [.NET 9 version of this article](#).

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;

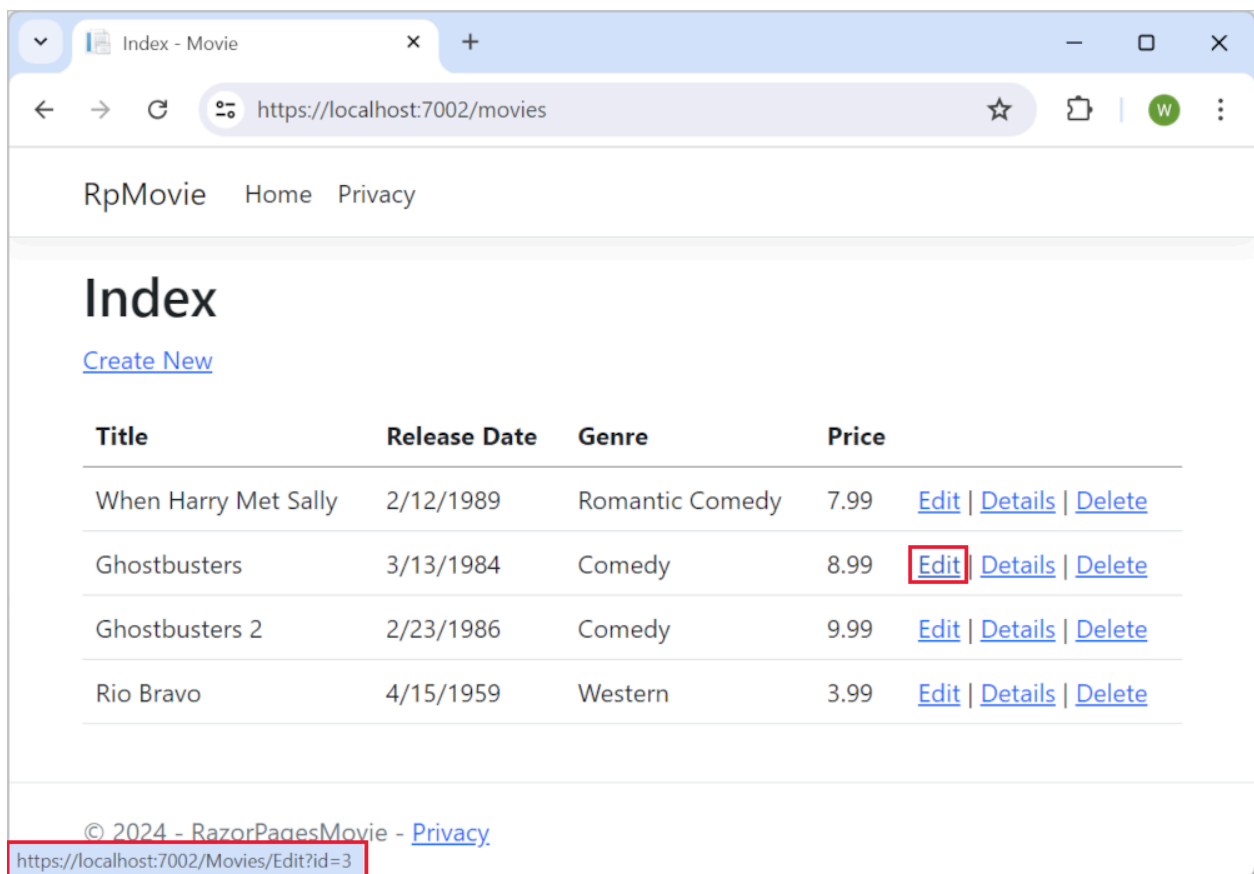
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }
}
```

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 \(YouTube\)](#).<sup>↗</sup>

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`.

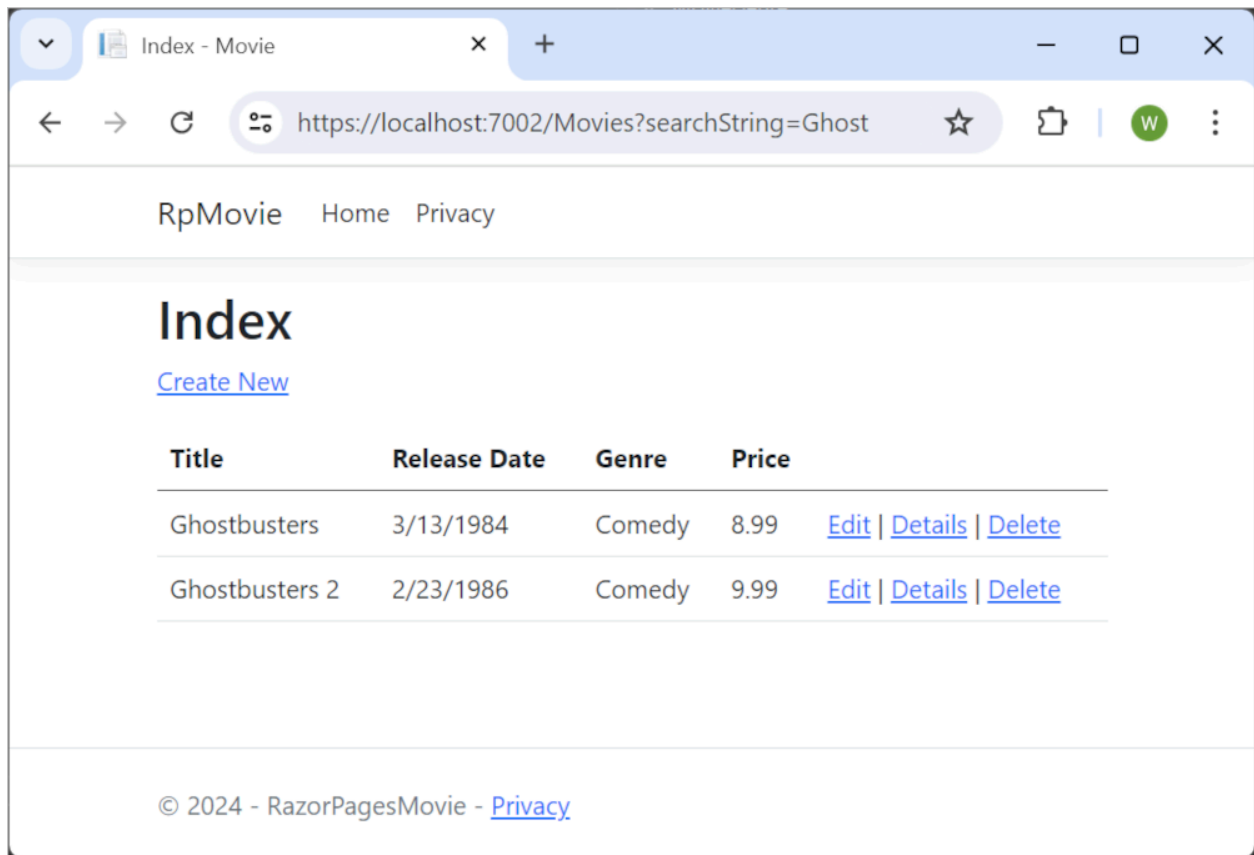
Rather, query execution is deferred. The evaluation of an expression is delayed until its realized value is iterated over or the `ToListAsync` method is called. See [Query Execution](#) for more information.

#### ⓘ 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? \(dotnet/efcore #11414\)](#).<sup>↗</sup>
- [How to make a SQLite column case insensitive \(dotnet/AspNetCore.Docs #22314\)](#).<sup>↗</sup>
- [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.

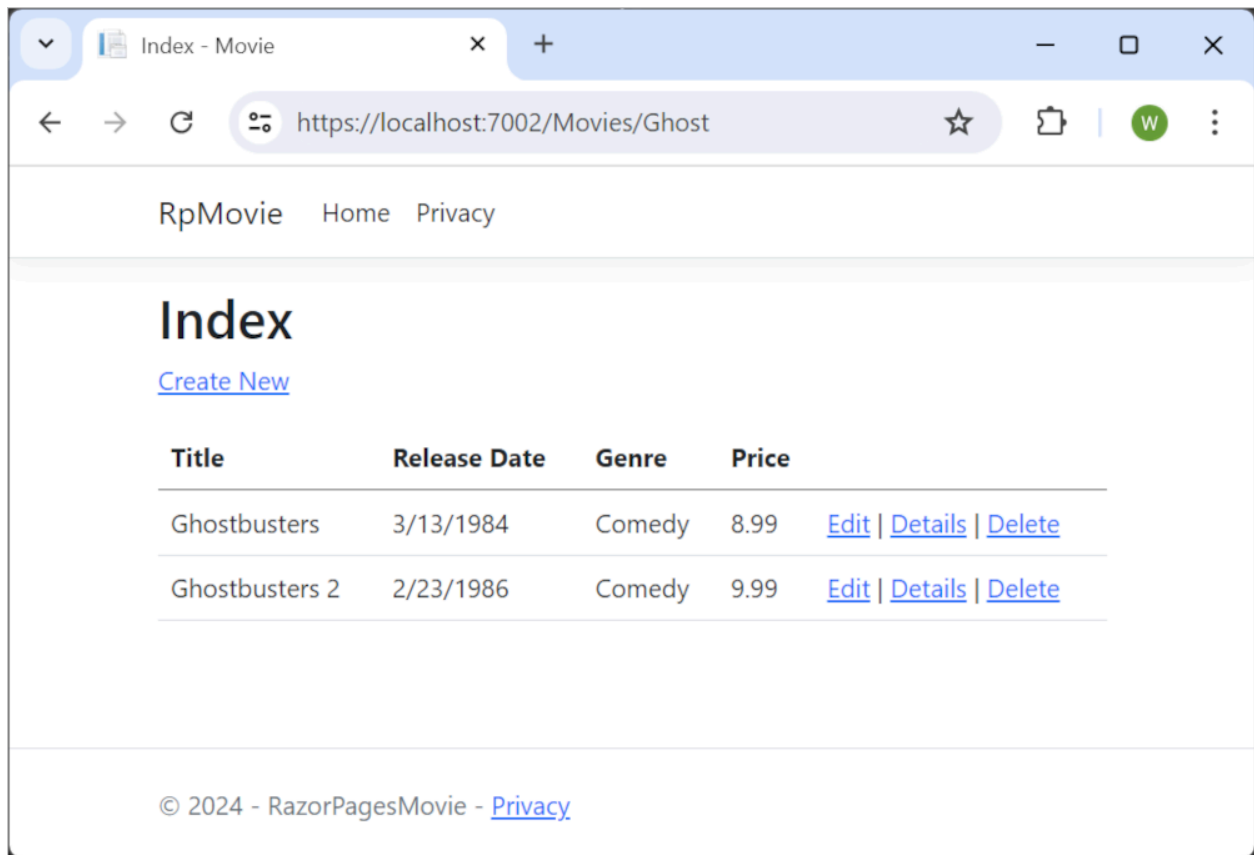


If the following route template is added to the Index page, the search string can be passed as a URL segment. For example, `https://localhost:5001/Movies/Ghost`.

CSSHTML

```
@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.

Index - Movie

localhost:7152/Movies?SearchString=ghost

RpMovie Home Privacy

## Index

[Create New](#)

Title:

Title	Release Date	Genre	Price	
Ghostbusters	3/13/1984	Comedy	8.99	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Ghostbusters 2	2/23/1986	Comedy	9.99	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>

© 2022 - RazorPagesMovie - [Privacy](#)

## 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";
}

```

```

}

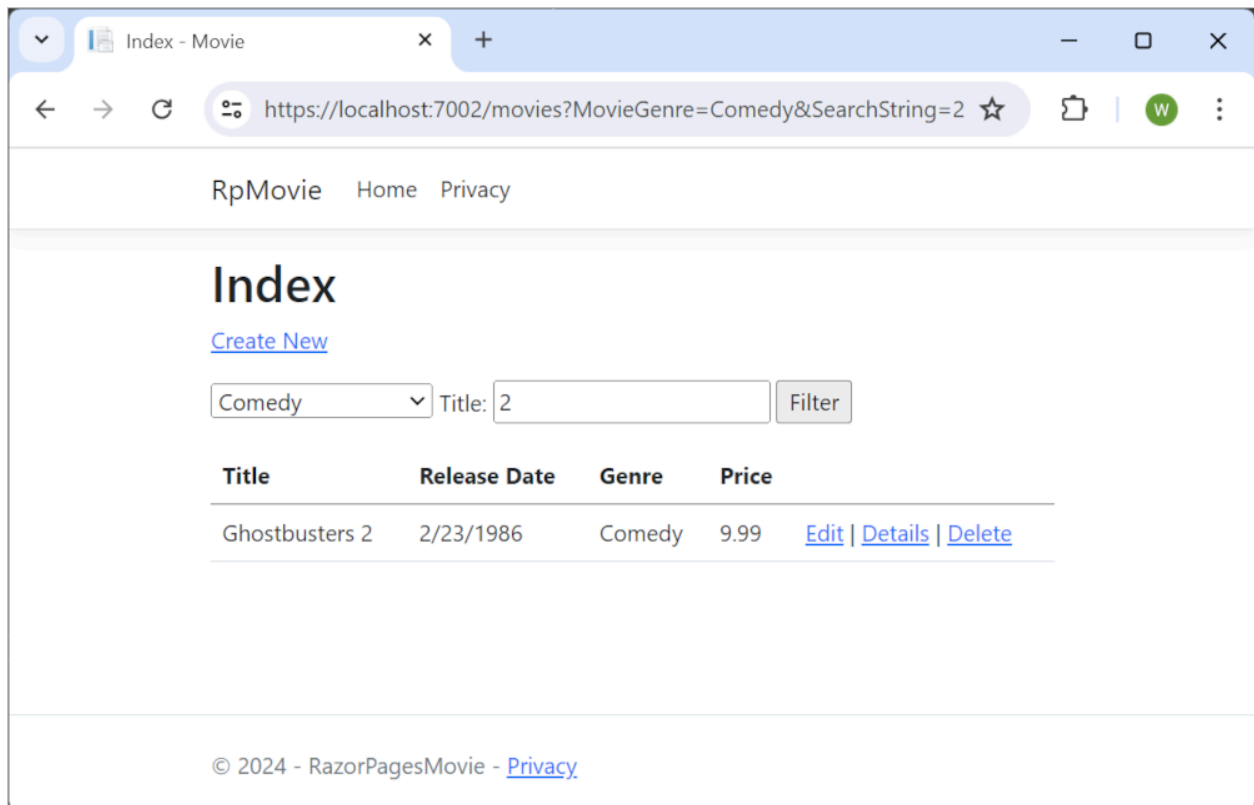
<h1>Index</h1>

<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 synchronized, all without the developer having to switch contexts to and from a database 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](#)  ("Don't Repeat Yourself"). 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.

The screenshot shows a web browser window with the address bar displaying `https://localhost:5001/Movies/Create`. The page title is "Create - Movie". The navigation bar includes "RpMovie", "Home", and "Privacy". The main heading is "Create Movie". The form contains the following fields and validation messages:

- Title:** Input field contains "a". Error message: "The field Title must be a string with a minimum length of 3 and a maximum length of 60."
- Release Date:** Date picker shows "00/01/0001". Error message: "The Release Date field is required."
- Genre:** Input field contains "a". Error message: "The field Genre must match the regular expression '^[A-Z]+[a-zA-Z0-9\\s]\*\$'."
- Price:** Input field contains "Dog". Error message: "The field Price must be a number."
- Rating:** Input field contains "z". Error message: "The field Rating must match the regular expression '^[A-Z]+[a-zA-Z0-9\\s]\*\$'."

At the bottom of the form is a blue "Create" button and a link "Back to List".

### ❗ 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;

    [DisplayName = "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](#)

# Filter methods for Razor Pages in ASP.NET Core

Article • 04/10/2024

By [Rick Anderson](#) 

Razor Page filters [IPageFilter](#) and [IAsyncPageFilter](#) allow Razor Pages to run code before and after a Razor Page handler is run. Razor Page filters are similar to [ASP.NET Core MVC action filters](#), except they can't be applied to individual page handler methods.

Razor Page filters:

- Run code after a handler method has been selected, but before model binding occurs.
- Run code before the handler method executes, after model binding is complete.
- Run code after the handler method executes.
- Can be implemented on a page or globally.
- Cannot be applied to specific page handler methods.
- Can have constructor dependencies populated by [Dependency Injection](#) (DI). For more information, see [ServiceFilterAttribute](#) and [TypeFilterAttribute](#).

While page constructors and middleware enable executing custom code before a handler method executes, only Razor Page filters enable access to [HttpContext](#) and the page. Middleware has access to the `HttpContext`, but not to the "page context". Filters have a [FilterContext](#) derived parameter, which provides access to `HttpContext`. Here's a sample for a page filter: [Implement a filter attribute](#) that adds a header to the response, something that can't be done with constructors or middleware. Access to the page context, which includes access to the instances of the page and it's model, are only available when executing filters, handlers, or the body of a Razor Page.

[View or download sample code](#)  ([how to download](#))

Razor Page filters provide the following methods, which can be applied globally or at the page level:

- Synchronous methods:
  - [OnPageHandlerSelected](#) : Called after a handler method has been selected, but before model binding occurs.
  - [OnPageHandlerExecuting](#) : Called before the handler method executes, after model binding is complete.

- `OnPageHandlerExecuted` : Called after the handler method executes, before the action result.
- Asynchronous methods:
  - `OnPageHandlerSelectionAsync` : Called asynchronously after the handler method has been selected, but before model binding occurs.
  - `OnPageHandlerExecutionAsync` : Called asynchronously before the handler method is invoked, after model binding is complete.

Implement **either** the synchronous or the async version of a filter interface, **not** both. The framework checks first to see if the filter implements the async interface, and if so, it calls that. If not, it calls the synchronous interface's method(s). If both interfaces are implemented, only the async methods are called. The same rule applies to overrides in pages, implement the synchronous or the async version of the override, not both.

## Implement Razor Page filters globally

The following code implements `IAsyncPageFilter`:

C#

```
public class SampleAsyncPageFilter : IAsyncPageFilter
{
    private readonly IConfiguration _config;

    public SampleAsyncPageFilter(IConfiguration config)
    {
        _config = config;
    }

    public Task OnPageHandlerSelectionAsync(PageHandlerSelectedContext context)
    {
        var key = _config["UserAgentID"];
        context.HttpContext.Request.Headers.TryGetValue("user-agent",
            out StringValues value);
        ProcessUserAgent.Write(context.ActionDescriptor.DisplayName,
            "SampleAsyncPageFilter.OnPageHandlerSelectionAsync",
            value, key.ToString());

        return Task.CompletedTask;
    }

    public async Task OnPageHandlerExecutionAsync(PageHandlerExecutingContext context,
        PageHandlerExecutionDelegate next)
    {
        // ...
    }
}
```

```

    {
        // Do post work.
        await next.Invoke();
    }
}

```

In the preceding code, `ProcessUserAgent.Write` is user supplied code that works with the user agent string.

The following code enables the `SampleAsyncPageFilter` in the `Startup` class:

```

C#

public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages()
        .AddMvcOptions(options =>
        {
            options.Filters.Add(new SampleAsyncPageFilter(Configuration));
        });
}

```

The following code calls `AddFolderApplicationModelConvention` to apply the `SampleAsyncPageFilter` to only pages in `/Movies`:

```

C#

public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages(options =>
    {
        options.Conventions.AddFolderApplicationModelConvention(
            "/Movies",
            model => model.Filters.Add(new
SampleAsyncPageFilter(Configuration)));
    });
}

```

The following code implements the synchronous `IPageFilter`:

```

C#

public class SamplePageFilter : IPageFilter
{
    private readonly IConfiguration _config;

    public SamplePageFilter(IConfiguration config)
    {
        _config = config;
    }
}

```

```

    }

    public void OnPageHandlerSelected(PageHandlerSelectedContext context)
    {
        var key = _config["UserAgentID"];
        context.HttpContext.Request.Headers.TryGetValue("user-agent", out
StringValues value);
        ProcessUserAgent.Write(context.ActionDescriptor.DisplayName,
                             "SamplePageFilter.OnPageHandlerSelected",
                             value, key.ToString());
    }

    public void OnPageHandlerExecuting(PageHandlerExecutingContext context)
    {
        Debug.WriteLine("Global sync OnPageHandlerExecuting called.");
    }

    public void OnPageHandlerExecuted(PageHandlerExecutedContext context)
    {
        Debug.WriteLine("Global sync OnPageHandlerExecuted called.");
    }
}

```

The following code enables the `SamplePageFilter`:

C#

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages()
        .AddMvcOptions(options =>
        {
            options.Filters.Add(new SamplePageFilter(Configuration));
        });
}

```

## Implement Razor Page filters by overriding filter methods

The following code overrides the asynchronous Razor Page filters:

C#

```

public class IndexModel : PageModel
{
    private readonly IConfiguration _config;

    public IndexModel(IConfiguration config)
    {

```



```

        _config = config;
    }

    public override Task
    OnPageHandlerSelectionAsync(PageHandlerSelectedContext context)
    {
        Debug.WriteLine("/IndexModel OnPageHandlerSelectionAsync");
        return Task.CompletedTask;
    }

    public async override Task
    OnPageHandlerExecutionAsync(PageHandlerExecutingContext context,
    PageHandlerExecutionDelegate next)
    {
        var key = _config["UserAgentID"];
        context.HttpContext.Request.Headers.TryGetValue("user-agent", out
        StringValues value);
        ProcessUserAgent.Write(context.ActionDescriptor.DisplayName,
        "/IndexModel-OnPageHandlerExecutionAsync",
        value, key.ToString());

        await next.Invoke();
    }
}

```

## Implement a filter attribute

The built-in attribute-based filter [OnResultExecutionAsync](#) filter can be subclassed. The following filter adds a header to the response:

C#

```

using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Filters;

namespace PageFilter.Filters
{
    public class AddHeaderAttribute : ResultFilterAttribute
    {
        private readonly string _name;
        private readonly string _value;

        public AddHeaderAttribute (string name, string value)
        {
            _name = name;
            _value = value;
        }

        public override void OnResultExecuting(ResultExecutingContext
        context)
    }
}

```

```

        {
            context.HttpContext.Response.Headers.Add(_name, new string[] {
                _value });
        }
    }
}

```

The following code applies the `AddHeader` attribute:

```

C#

using Microsoft.AspNetCore.Mvc.RazorPages;
using PageFilter.Filters;

namespace PageFilter.Movies
{
    [AddHeader("Author", "Rick")]
    public class TestModel : PageModel
    {
        public void OnGet()
        {
        }
    }
}

```

Use a tool such as the browser developer tools to examine the headers. Under **Response Headers**, `author: Rick` is displayed.

See [Overriding the default order](#) for instructions on overriding the order.

See [Cancellation and short circuiting](#) for instructions to short-circuit the filter pipeline from a filter.

## Authorize filter attribute

The `Authorize` attribute can be applied to a `PageModel`:

```

C#

using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace PageFilter.Pages
{
    [Authorize]
    public class ModelWithAuthFilterModel : PageModel

```

```
{  
    public IActionResult OnGet() => Page();  
}
```

# Razor Pages route and app conventions in ASP.NET Core

Article • 04/10/2024

Learn how to use page [route and app model provider conventions](#) to control page routing, discovery, and processing in Razor Pages apps.

To specify a page route, add route segments, or add parameters to a route, use the page's `@page` directive. For more information, see [Custom routes](#).

There are reserved words that can't be used as route segments or parameter names. For more information, see [Routing: Reserved routing names](#).

[View or download sample code](#) [↗](#) ([how to download](#))

[⌵](#) Expand table

Scenario	The sample demonstrates
<a href="#">Model conventions</a>  Conventions.Add <ul style="list-style-type: none"><li>• <a href="#">IPageRouteModelConvention</a></li><li>• <a href="#">IPageApplicationModelConvention</a></li><li>• <a href="#">IPageHandlerModelConvention</a></li></ul>	Add a route template and header to an app's pages.
<a href="#">Page route action conventions</a> <ul style="list-style-type: none"><li>• <a href="#">AddFolderRouteModelConvention</a></li><li>• <a href="#">AddPageRouteModelConvention</a></li><li>• <a href="#">AddPageRoute</a></li></ul>	Add a route template to pages in a folder and to a single page.
<a href="#">Page model action conventions</a> <ul style="list-style-type: none"><li>• <a href="#">AddFolderApplicationModelConvention</a></li><li>• <a href="#">AddPageApplicationModelConvention</a></li><li>• <a href="#">ConfigureFilter</a> (filter class, lambda expression, or filter factory)</li></ul>	Add a header to pages in a folder, add a header to a single page, and configure a <a href="#">filter factory</a> to add a header to an app's pages.

Razor Pages conventions are configured using an [AddRazorPages](#) overload that configures [RazorPagesOptions](#). The following convention examples are explained later in this topic:

C#

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

builder.Services.AddRazorPages(options =>
{
    options.Conventions.Add( ... );
    options.Conventions.AddFolderRouteModelConvention(
        "/OtherPages", model => { ... });
    options.Conventions.AddPageRouteModelConvention(
        "/About", model => { ... });
    options.Conventions.AddPageRoute(
        "/Contact", "TheContactPage/{text?}");
    options.Conventions.AddFolderApplicationModelConvention(
        "/OtherPages", model => { ... });
    options.Conventions.AddPageApplicationModelConvention(
        "/About", model => { ... });
    options.Conventions.ConfigureFilter(model => { ... });
    options.Conventions.ConfigureFilter( ... );
});
}
```

## Route order

Routes specify an [Order](#) for processing (route matching).

 Expand table

Route order	Behavior
-1	The route is processed before other routes are processed.
0	Order isn't specified (default value). Not assigning <code>Order</code> ( <code>Order = null</code> ) defaults the route <code>Order</code> to 0 (zero) for processing.
1, 2, ... n	Specifies the route processing order.

Route processing is established by convention:

- Routes are processed in sequential order (-1, 0, 1, 2, ... n).
- When routes have the same `Order`, the most specific route is matched first followed by less specific routes.
- When routes with the same `Order` and the same number of parameters match a request URL, routes are processed in the order that they're added to the [PageConventionCollection](#).

If possible, avoid depending on an established route processing order. Generally, routing selects the correct route with URL matching. If you must set route `Order` properties to route requests correctly, the app's routing scheme is probably confusing to clients and fragile to maintain. Seek to simplify the app's routing scheme. The sample app requires an explicit route processing order to demonstrate several routing scenarios using a single app. However, you should attempt to avoid the practice of setting route `Order` in production apps.

Razor Pages routing and MVC controller routing share an implementation. Information on route order in the MVC topics is available at [Routing to controller actions: Ordering attribute routes](#).

## Model conventions

Add a delegate for `IPageConvention` to add [model conventions](#) that apply to Razor Pages.

### Add a route model convention to all pages

Use [Conventions](#) to create and add an `IPageRouteModelConvention` to the collection of `IPageConvention` instances that are applied during page route model construction.

The sample app contains the `GlobalTemplatePageRouteModelConvention` class to add a `{globalTemplate?}` route template to all of the pages in the app:

C#

```
using Microsoft.AspNetCore.Mvc.ApplicationModels;
namespace SampleApp.Conventions;

public class GlobalTemplatePageRouteModelConvention :
    IPageRouteModelConvention
{
    public void Apply(PageRouteModel model)
    {
        var selectorCount = model.Selectors.Count;
        for (var i = 0; i < selectorCount; i++)
        {
            var selector = model.Selectors[i];
            model.Selectors.Add(new SelectorModel
            {
                AttributeRouteModel = new AttributeRouteModel
                {
                    Order = 1,
```

```

        Template = AttributeRouteModel.CombineTemplates(
            selector.AttributeRouteModel!.Template,
            "{globalTemplate?}"),
    }
    });
}
}
}
}
}

```

In the preceding code:

- The [PageRouteModel](#) is passed to the [Apply](#) method.
- The [PageRouteModel.Selectors](#) gets the count of selectors.
- A new [SelectorModel](#) is added which contains a [AttributeRouteModel](#)

Razor Pages options, such as adding [Conventions](#), are added when Razor Pages is added to the service collection. For an example, see the [sample app](#).

C#

```

using Microsoft.AspNetCore.Mvc.ApplicationModels;
using Microsoft.EntityFrameworkCore;
using SampleApp.Conventions;
using SampleApp.Data;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<AppDbContext>(options =>

options.UseInMemoryDatabase("InMemoryDb"));

builder.Services.AddRazorPages(options =>
{
    options.Conventions.Add(new
GlobalTemplatePageRouteModelConvention());

    options.Conventions.AddFolderRouteModelConvention("/OtherPages",
model =>
{
    var selectorCount = model.Selectors.Count;
    for (var i = 0; i < selectorCount; i++)
    {
        var selector = model.Selectors[i];
        model.Selectors.Add(new SelectorModel
        {
            AttributeRouteModel = new AttributeRouteModel
            {
                Order = 2,
                Template = AttributeRouteModel.CombineTemplates(
                    selector.AttributeRouteModel!.Template,
                    "{otherPagesTemplate?}"),
            }
        }
    }
}
}

```

```

        });
    }
});

options.Conventions.AddPageRouteModelConvention("/About", model =>
{
    var selectorCount = model.Selectors.Count;
    for (var i = 0; i < selectorCount; i++)
    {
        var selector = model.Selectors[i];
        model.Selectors.Add(new SelectorModel
        {
            AttributeRouteModel = new AttributeRouteModel
            {
                Order = 2,
                Template = AttributeRouteModel.CombineTemplates(
                    selector.AttributeRouteModel!.Template,
                    "{aboutTemplate?}"),
            }
        });
    }
});

});

});

var app = builder.Build();

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

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseAuthorization();
app.MapRazorPages();
app.Run();

```

Consider the `GlobalTemplatePageRouteModelConvention` class:

```

C#

using Microsoft.AspNetCore.Mvc.ApplicationModels;
namespace SampleApp.Conventions;

public class GlobalTemplatePageRouteModelConvention :
    IPageRouteModelConvention
{
    public void Apply(PageRouteModel model)
    {
        var selectorCount = model.Selectors.Count;
        for (var i = 0; i < selectorCount; i++)

```



```

{
    var selector = model.Selectors[i];
    model.Selectors.Add(new SelectorModel
    {
        AttributeRouteModel = new AttributeRouteModel
        {
            Order = 1,
            Template = AttributeRouteModel.CombineTemplates(
                selector.AttributeRouteModel!.Template,
                "{globalTemplate?}"),
        }
    });
}
}
}
}

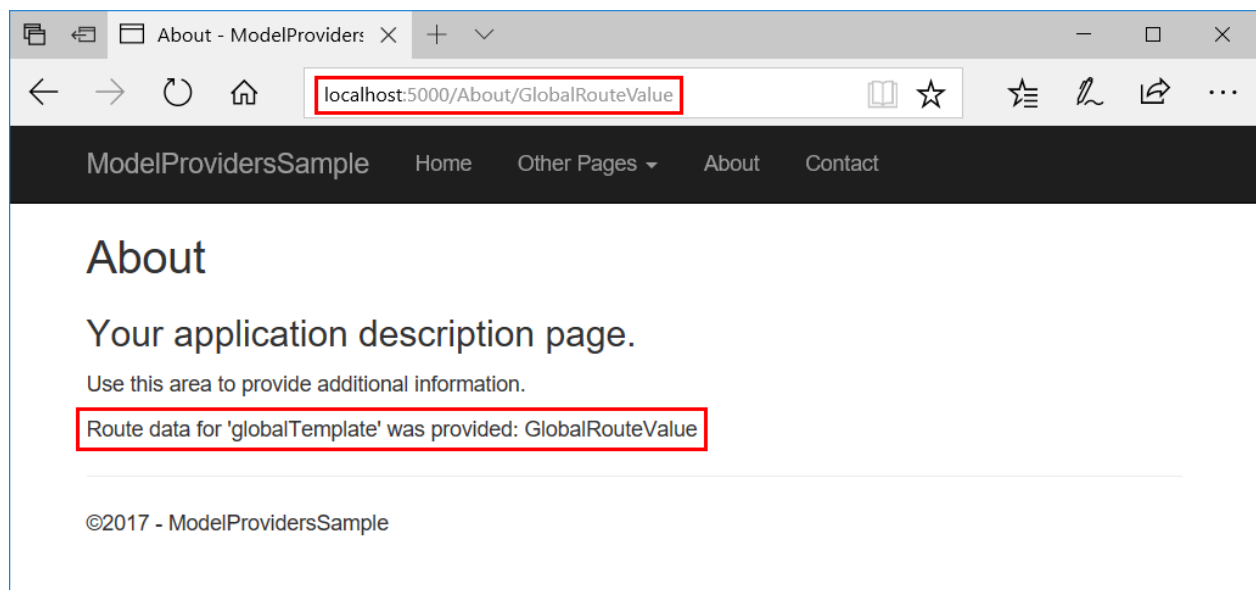
```

The `Order` property for the `AttributeRouteModel` is set to `1`. This ensures the following route matching behavior in the sample app:

- A route template for `TheContactPage/{text?}` is added later in this topic. The `Contact Page` route has a default order of `null` (`Order = 0`), so it matches before the `{globalTemplate?}` route template which has `Order = 1`.
- The `{aboutTemplate?}` route template is shown in the preceding code. The `{aboutTemplate?}` template is given an `order` of `2`. When the About page is requested at `/About/RouteDataValue`, "RouteDataValue" is loaded into `RouteData.Values["globalTemplate"]` (`Order = 1`) and not `RouteData.Values["aboutTemplate"]` (`Order = 2`) due to setting the `Order` property.
- The `{otherPagesTemplate?}` route template is shown in the preceding code. The `{otherPagesTemplate?}` template is given an `order` of `2`. When any page in the *Pages/OtherPages* folder is requested with a route parameter:
- For example, `/OtherPages/Page1/xyz`
- The route data value "xyz" is loaded into `RouteData.Values["globalTemplate"]` (`Order = 1`).
- `RouteData.Values["otherPagesTemplate"]` with (`Order = 2`) is not loaded due to the `Order` property `2` having a higher value.

**When possible, don't set the `order`.** When `order` is not set, it defaults to `Order = 0`. Rely on routing to select the correct route rather than the `Order` property.

Request the sample's `About` page at `localhost:{port}/About/GlobalRouteValue` and inspect the result:



The sample app uses the [Rick.Docs.Samples.RouteInfo](#) NuGet package to display routing information in the logging output. Using `localhost:{port}/About/GlobalRouteValue`, the logger displays the request, the `order`, and the template used:

.NET CLI

```
info: SampleApp.Pages.AboutModel[0]
      /About/GlobalRouteValue Order = 1 Template =
About/{globalTemplate?}
```

## Add an app model convention to all pages

Use [Conventions](#) to create and add an [IPageApplicationModelConvention](#) to the collection of [IPageConvention](#) instances that are applied during page app model construction.

To demonstrate this and other conventions later in the topic, the sample app includes an `AddHeaderAttribute` class. The class constructor accepts a `name` string and a `values` string array. These values are used in its `OnResultExecuting` method to set a response header. The full class is shown in the [Page model action conventions](#) section later in the topic.

The sample app uses the `AddHeaderAttribute` class to add a header, `GlobalHeader`, to all of the pages in the app:

C#

```
public class GlobalHeaderPageApplicationModelConvention
    : IPageApplicationModelConvention
```

```
{
    public void Apply(PageApplicationModel model)
    {
        model.Filters.Add(new AddHeaderAttribute(
            "GlobalHeader", new string[] { "Global Header Value" }));
    }
}
```

Program.cs:

C#

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

builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseInMemoryDatabase("InMemoryDb"));

builder.Services.AddRazorPages(options =>
{
    options.Conventions.Add(new
GlobalTemplatePageRouteModelConvention());

    options.Conventions.Add(new
GlobalHeaderPageApplicationModelConvention());
});
```

Request the sample's About page at `localhost:{port}/About` and inspect the headers to view the result:

▼ **Response Headers** [view source](#)

**AboutHeader:** About Header Value  
**Content-Type:** text/html; charset=utf-8  
**Date:** Thu, 19 Oct 2017 21:09:07 GMT  
**FilterFactoryHeader:** Filter Factory Header Value 1  
**FilterFactoryHeader:** Filter Factory Header Value 2  
**GlobalHeader:** Global Header Value  
**Server:** Kestrel  
**Transfer-Encoding:** chunked

## Add a handler model convention to all pages

Use [Conventions](#) to create and add an [IPageHandlerModelConvention](#) to the collection of [IPageConvention](#) instances that are applied during page handler model construction.

C#

```
public class GlobalPageHandlerModelConvention
    : IPageHandlerModelConvention
```

```

{
    public void Apply(PageHandlerModel model)
    {
        // Access the PageHandlerModel
    }
}

```

## Page route action conventions

The default route model provider that derives from [IPageRouteModelProvider](#) invokes conventions which are designed to provide extensibility points for configuring page routes.

## Folder route model convention

Use [AddFolderRouteModelConvention](#) to create and add an [IPageRouteModelConvention](#) that invokes an action on the [PageRouteModel](#) for all of the pages under the specified folder.

The sample app uses [AddFolderRouteModelConvention](#) to add an `{otherPagesTemplate?}` route template to the pages in the *OtherPages* folder:

C#

```

options.Conventions.AddFolderRouteModelConvention("/OtherPages", model =>
{
    var selectorCount = model.Selectors.Count;
    for (var i = 0; i < selectorCount; i++)
    {
        var selector = model.Selectors[i];
        model.Selectors.Add(new SelectorModel
        {
            AttributeRouteModel = new AttributeRouteModel
            {
                Order = 2,
                Template = AttributeRouteModel.CombineTemplates(
                    selector.AttributeRouteModel!.Template,
                    "{otherPagesTemplate?}"),
            }
        });
    }
});
}
});

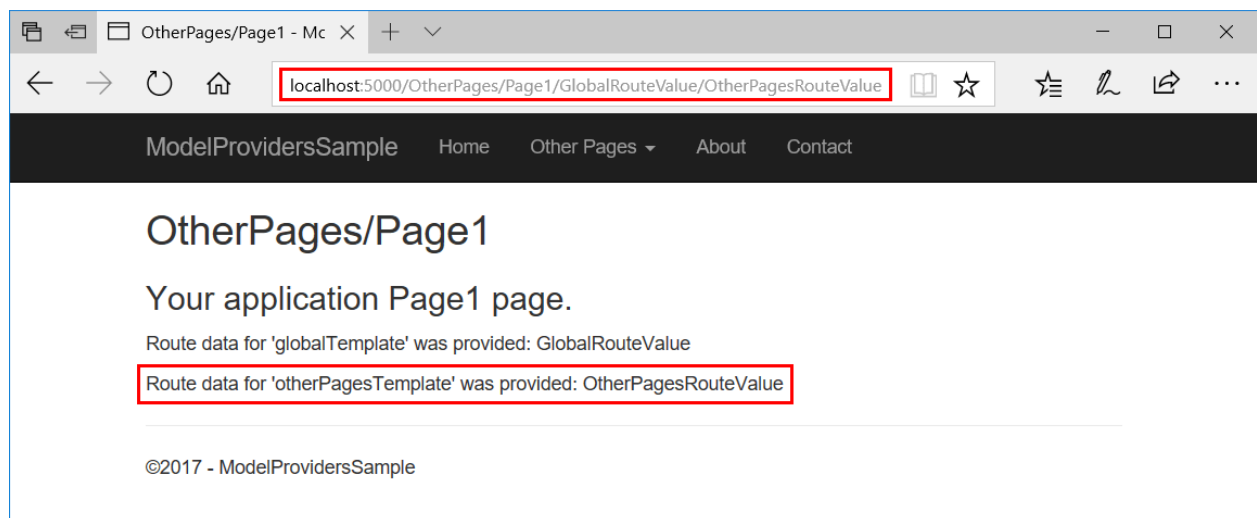
```

The [Order](#) property for the [AttributeRouteModel](#) is set to `2`. This ensures that the template for `{globalTemplate?}` (set earlier in the topic to `1`) is given priority for the first route data value position when a single route value is provided. If a page in the

`Pages/OtherPages` folder is requested with a route parameter value (for example, `/OtherPages/Page1/RouteDataValue`), `"RouteDataValue"` is loaded into `RouteData.Values["globalTemplate"]` (`Order = 1`) and not `RouteData.Values["otherPagesTemplate"]` (`Order = 2`) due to setting the `Order` property.

Wherever possible, don't set the `Order`, which results in `Order = 0`. Rely on routing to select the correct route.

Request the sample's `Page1` page at `localhost:5000/OtherPages/Page1/GlobalRouteValue/OtherPagesRouteValue` and inspect the result:



## Page route model convention

Use [AddPageRouteModelConvention](#) to create and add an [IPageRouteModelConvention](#) that invokes an action on the [PageRouteModel](#) for the page with the specified name.

The sample app uses `AddPageRouteModelConvention` to add an `{aboutTemplate?}` route template to the About page:

C#

```
options.Conventions.AddPageRouteModelConvention("/About", model =>
{
    var selectorCount = model.Selectors.Count;
    for (var i = 0; i < selectorCount; i++)
    {
        var selector = model.Selectors[i];
        model.Selectors.Add(new SelectorModel
        {
            AttributeRouteModel = new AttributeRouteModel
            {
                Order = 2,
                Template = AttributeRouteModel.CombineTemplates(
```

```

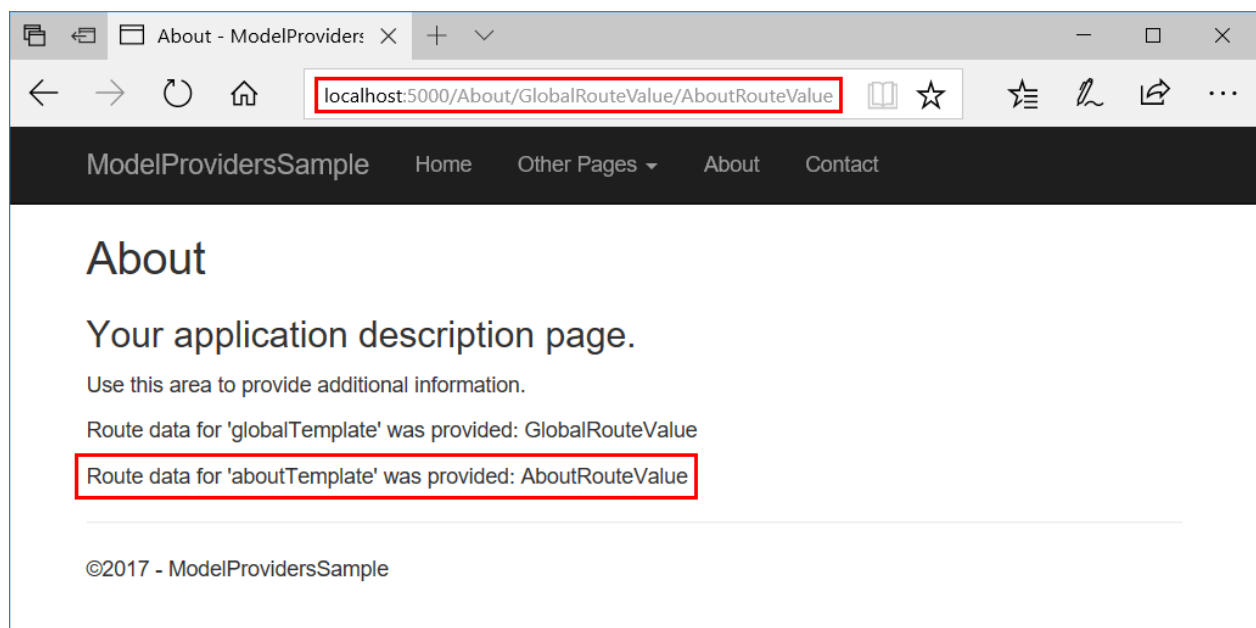
        selector.AttributeRouteModel!.Template,
        "{aboutTemplate?}"),
    }
});
}
});

```

The `Order` property for the `AttributeRouteModel` is set to `2`. This ensures that the template for `{globalTemplate?}` (set earlier in the topic to `1`) is given priority for the first route data value position when a single route value is provided. If the About page is requested with a route parameter value at `/About/RouteDataValue`, "RouteDataValue" is loaded into `RouteData.Values["globalTemplate"]` (`Order = 1`) and not `RouteData.Values["aboutTemplate"]` (`Order = 2`) due to setting the `Order` property.

Wherever possible, don't set the `Order`, which results in `Order = 0`. Rely on routing to select the correct route.

Request the sample's About page at `localhost:{port}/About/GlobalRouteValue/AboutRouteValue` and inspect the result:



The logger output displays:

.NET CLI

```

info: SampleApp.Pages.AboutModel[0]
      /About/GlobalRouteValue/AboutRouteValue    Order = 2 Template =
About/{globalTemplate?}/{aboutTemplate?}

```

# Use a parameter transformer to customize page routes

See [Parameter transformers](#).

## Configure a page route

Use [AddPageRoute](#) to configure a route to a page at the specified page path. Generated links to the page use the specified route. [AddPageRoute](#) uses [AddPageRouteModelConvention](#) to establish the route.

The sample app creates a route to `/TheContactPage` for the `Contact` Razor Page:

C#

```
options.Conventions.AddPageRoute("/Contact", "TheContactPage/{text?}");
```

The `Contact` page can also be reached at `/Contact1`` via its default route.

The sample app's custom route to the `Contact` page allows for an optional `text` route segment (`{text?}`). The page also includes this optional segment in its `@page` directive in case the visitor accesses the page at its `/Contact` route:

CSHTML

```
@page "{text?}"
@model ContactModel
@{
    ViewData["Title"] = "Contact";
}

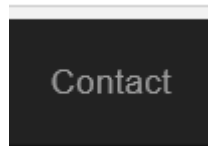
<h1>@ViewData["Title"]</h1>
<h2>@Model.Message</h2>

<address>
    One Microsoft Way<br>
    Redmond, WA 98052-6399<br>
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>

<address>
    <strong>Support:</strong> <a
href="mailto:Support@example.com">Support@example.com</a><br>
    <strong>Marketing:</strong> <a
href="mailto:Marketing@example.com">Marketing@example.com</a>
</address>
```

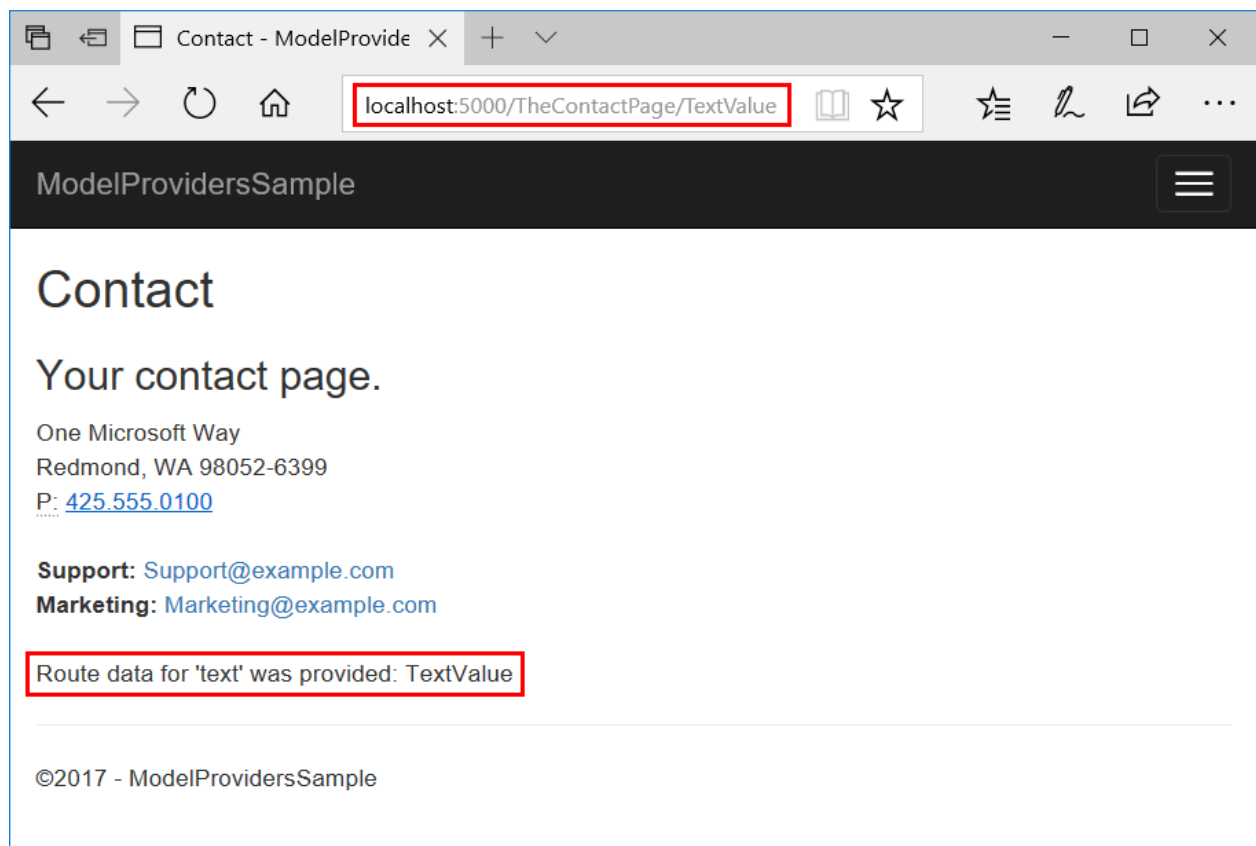
```
<p>@Model.RouteDataTextTemplateValue</p>
```

Note that the URL generated for the **Contact** link in the rendered page reflects the updated route:



```
><li>...</li>
▼<li>
  <a href="/TheContactPage">Contact</a>
</li>
::after
</ul>
```

Visit the `Contact` page at either its ordinary route, `/Contact`, or the custom route, `/TheContactPage`. If you supply an additional `text` route segment, the page shows the HTML-encoded segment that you provide:



## Page model action conventions

The default page model provider that implements `IPageApplicationModelProvider` invokes conventions which are designed to provide extensibility points for configuring



page models. These conventions are useful when building and modifying page discovery and processing scenarios.

For the examples in this section, the sample app uses an `AddHeaderAttribute` class, which is a [ResultFilterAttribute](#), that applies a response header:

C#

```
public class AddHeaderAttribute : ResultFilterAttribute
{
    private readonly string _name;
    private readonly string[] _values;

    public AddHeaderAttribute(string name, string[] values)
    {
        _name = name;
        _values = values;
    }

    public override void OnResultExecuting(ResultExecutingContext context)
    {
        context.HttpContext.Response.Headers.Add(_name, _values);
        base.OnResultExecuting(context);
    }
}
```

Using conventions, the sample demonstrates how to apply the attribute to all of the pages in a folder and to a single page.

### Folder app model convention

Use [AddFolderApplicationModelConvention](#) to create and add an [IPageApplicationModelConvention](#) that invokes an action on [PageApplicationModel](#) instances for all pages under the specified folder.

The sample demonstrates the use of `AddFolderApplicationModelConvention` by adding a header, `OtherPagesHeader`, to the pages inside the *OtherPages* folder of the app:

C#

```
options.Conventions.AddFolderApplicationModelConvention("/OtherPages", model
=>
{
    model.Filters.Add(new AddHeaderAttribute(
        "OtherPagesHeader", new string[] { "OtherPages Header Value" }));
});
```

Request the sample's Page1 page at `localhost:5000/OtherPages/Page1` and inspect the headers to view the result:

---

▼ **Response Headers** [view source](#)

**Content-Type:** text/html; charset=utf-8  
**Date:** Sat, 21 Oct 2017 06:13:16 GMT  
**FilterFactoryHeader:** Filter Factory Header Value 1  
**FilterFactoryHeader:** Filter Factory Header Value 2  
**GlobalHeader:** Global Header Value  
**OtherPagesHeader:** OtherPages Header Value  
**Server:** Kestrel  
**Transfer-Encoding:** chunked

---

## Page app model convention

Use [AddPageApplicationModelConvention](#) to create and add an [IPageApplicationModelConvention](#) that invokes an action on the [PageApplicationModel](#) for the page with the specified name.

The sample demonstrates the use of [AddPageApplicationModelConvention](#) by adding a header, `AboutHeader`, to the About page:

C#

```
options.Conventions.AddPageApplicationModelConvention("/About", model =>
{
    model.Filters.Add(new AddHeaderAttribute(
        "AboutHeader", new string[] { "About Header Value" }));
});
```

Request the sample's About page at `localhost:5000/About` and inspect the headers to view the result:

---

▼ **Response Headers** [view source](#)

**AboutHeader:** About Header Value  
**Content-Type:** text/html; charset=utf-8  
**Date:** Thu, 19 Oct 2017 21:09:07 GMT  
**FilterFactoryHeader:** Filter Factory Header Value 1  
**FilterFactoryHeader:** Filter Factory Header Value 2  
**GlobalHeader:** Global Header Value  
**Server:** Kestrel  
**Transfer-Encoding:** chunked

---

## Configure a filter

[ConfigureFilter](#) configures the specified filter to apply. You can implement a filter class, but the sample app shows how to implement a filter in a lambda expression, which is implemented behind-the-scenes as a factory that returns a filter:

C#

```
options.Conventions.ConfigureFilter(model =>
{
    if (model.RelativePath.Contains("OtherPages/Page2"))
    {
        return new AddHeaderAttribute(
            "OtherPagesPage2Header",
            new string[] { "OtherPages/Page2 Header Value" });
    }
    return new EmptyFilter();
});
```

The page app model is used to check the relative path for segments that lead to the Page2 page in the *OtherPages* folder. If the condition passes, a header is added. If not, the `EmptyFilter` is applied.

`EmptyFilter` is an [Action filter](#). Since Action filters are ignored by Razor Pages, the `EmptyFilter` has no effect as intended if the path doesn't contain `OtherPages/Page2`.

Request the sample's Page2 page at `localhost:5000/OtherPages/Page2` and inspect the headers to view the result:

---

▼ **Response Headers**      [view source](#)

**Content-Type:** text/html; charset=utf-8  
**Date:** Mon, 23 Oct 2017 06:06:52 GMT  
**FilterFactoryHeader:** Filter Factory Header Value 1  
**FilterFactoryHeader:** Filter Factory Header Value 2  
**GlobalHeader:** Global Header Value  
**OtherPagesHeader:** OtherPages Header Value  
**OtherPagesPage2Header:** OtherPages/Page2 Header Value  
**Server:** Kestrel  
**Transfer-Encoding:** chunked

---

## Configure a filter factory

[ConfigureFilter](#) configures the specified factory to apply [filters](#) to all Razor Pages.

The sample app provides an example of using a [filter factory](#) by adding a header, `FilterFactoryHeader`, with two values to the app's pages:

C#

```
options.Conventions.ConfigureFilter(new AddHeaderWithFactory());
```

AddHeaderWithFactory.cs:

C#

```
public class AddHeaderWithFactory : IFilterFactory
{
    // Implement IFilterFactory
    public IFilterMetadata CreateInstance(IServiceProvider serviceProvider)
    {
        return new AddHeaderFilter();
    }

    private class AddHeaderFilter : IResultFilter
    {
        public void OnResultExecuting(ResultExecutingContext context)
        {
            context.HttpContext.Response.Headers.Add(
                "FilterFactoryHeader",
                new string[]
                {
                    "Filter Factory Header Value 1",
                    "Filter Factory Header Value 2"
                });
        }

        public void OnResultExecuted(ResultExecutedContext context)
        {
        }
    }

    public bool IsReusable
    {
        get
        {
            return false;
        }
    }
}
```

Request the sample's About page at `localhost:5000/About` and inspect the headers to view the result:

---

▼ Response Headers	<a href="#">view source</a>
<b>AboutHeader:</b> About Header Value	
<b>Content-Type:</b> text/html; charset=utf-8	
<b>Date:</b> Thu, 19 Oct 2017 21:09:07 GMT	
<b>FilterFactoryHeader:</b> Filter Factory Header Value 1	
<b>FilterFactoryHeader:</b> Filter Factory Header Value 2	
<b>GlobalHeader:</b> Global Header Value	
<b>Server:</b> Kestrel	
<b>Transfer-Encoding:</b> chunked	


---

## MVC Filters and the Page filter (IPageFilter)

MVC [Action filters](#) are ignored by Razor Pages, since Razor Pages use handler methods. Other types of MVC filters are available for you to use: [Authorization](#), [Exception](#), [Resource](#), and [Result](#). For more information, see the [Filters](#) topic.

The Page filter ([IPageFilter](#)) is a filter that applies to Razor Pages. For more information, see [Filter methods for Razor Pages](#).

## Additional resources

- [Razor Pages Routing](#) 
- [Razor Pages authorization conventions in ASP.NET Core](#)
- [Areas in ASP.NET Core](#)

# Overview of ASP.NET Core MVC

Article • 06/17/2024

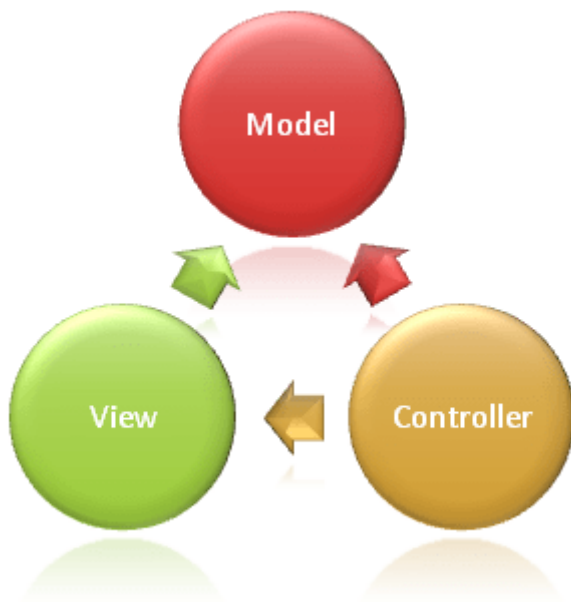
By [Steve Smith](#) 

ASP.NET Core MVC is a rich framework for building web apps and APIs using the Model-View-Controller design pattern.

## MVC pattern

The Model-View-Controller (MVC) architectural pattern separates an application into three main groups of components: Models, Views, and Controllers. This pattern helps to achieve [separation of concerns](#). Using this pattern, user requests are routed to a Controller which is responsible for working with the Model to perform user actions and/or retrieve results of queries. The Controller chooses the View to display to the user, and provides it with any Model data it requires.

The following diagram shows the three main components and which ones reference the others:



This delineation of responsibilities helps you scale the application in terms of complexity because it's easier to code, debug, and test something (model, view, or controller) that has a single job. It's more difficult to update, test, and debug code that has dependencies spread across two or more of these three areas. For example, user interface logic tends to change more frequently than business logic. If presentation code and business logic are combined in a single object, an object containing business logic

must be modified every time the user interface is changed. This often introduces errors and requires the retesting of business logic after every minimal user interface change.

#### ⓘ Note

Both the view and the controller depend on the model. However, the model depends on neither the view nor the controller. This is one of the key benefits of the separation. This separation allows the model to be built and tested independent of the visual presentation.

## Model Responsibilities

The Model in an MVC application represents the state of the application and any business logic or operations that should be performed by it. Business logic should be encapsulated in the model, along with any implementation logic for persisting the state of the application. Strongly-typed views typically use ViewModel types designed to contain the data to display on that view. The controller creates and populates these ViewModel instances from the model.

## View Responsibilities

Views are responsible for presenting content through the user interface. They use the [Razor view engine](#) to embed .NET code in HTML markup. There should be minimal logic within views, and any logic in them should relate to presenting content. If you find the need to perform a great deal of logic in view files in order to display data from a complex model, consider using a [View Component](#), ViewModel, or view template to simplify the view.

## Controller Responsibilities

Controllers are the components that handle user interaction, work with the model, and ultimately select a view to render. In an MVC application, the view only displays information; the controller handles and responds to user input and interaction. In the MVC pattern, the controller is the initial entry point, and is responsible for selecting which model types to work with and which view to render (hence its name - it controls how the app responds to a given request).

#### ⓘ Note

Controllers shouldn't be overly complicated by too many responsibilities. To keep controller logic from becoming overly complex, push business logic out of the controller and into the domain model.

#### Tip

If you find that your controller actions frequently perform the same kinds of actions, move these common actions into [filters](#).

## ASP.NET Core MVC

The ASP.NET Core MVC framework is a lightweight, open source, highly testable presentation framework optimized for use with ASP.NET Core.

ASP.NET Core MVC provides a patterns-based way to build dynamic websites that enables a clean separation of concerns. It gives you full control over markup, supports TDD-friendly development and uses the latest web standards.

## Routing

ASP.NET Core MVC is built on top of [ASP.NET Core's routing](#), a powerful URL-mapping component that lets you build applications that have comprehensible and searchable URLs. This enables you to define your application's URL naming patterns that work well for search engine optimization (SEO) and for link generation, without regard for how the files on your web server are organized. You can define your routes using a convenient route template syntax that supports route value constraints, defaults and optional values.

*Convention-based routing* enables you to globally define the URL formats that your application accepts and how each of those formats maps to a specific action method on a given controller. When an incoming request is received, the routing engine parses the URL and matches it to one of the defined URL formats, and then calls the associated controller's action method.

C#

```
routes.MapRoute(name: "Default", template: "{controller=Home}/{action=Index}/{id?}");
```



*Attribute routing* enables you to specify routing information by decorating your controllers and actions with attributes that define your application's routes. This means that your route definitions are placed next to the controller and action with which they're associated.

C#

```
[Route("api/[controller]")]
public class ProductsController : Controller
{
    [HttpGet("{id}")]
    public IActionResult GetProduct(int id)
    {
        ...
    }
}
```

## Model binding

ASP.NET Core MVC [model binding](#) converts client request data (form values, route data, query string parameters, HTTP headers) into objects that the controller can handle. As a result, your controller logic doesn't have to do the work of figuring out the incoming request data; it simply has the data as parameters to its action methods.

C#

```
public async Task<IActionResult> Login(LoginViewModel model, string
returnUrl = null) { ... }
```

## Model validation

ASP.NET Core MVC supports [validation](#) by decorating your model object with data annotation validation attributes. The validation attributes are checked on the client side before values are posted to the server, as well as on the server before the controller action is called.

C#

```
using System.ComponentModel.DataAnnotations;
public class LoginViewModel
{
    [Required]
    [EmailAddress]
    public string Email { get; set; }
```

```

[Required]
[DataType(DataType.Password)]
public string Password { get; set; }

[Display(Name = "Remember me?")]
public bool RememberMe { get; set; }
}

```

A controller action:

C#

```

public async Task<IActionResult> Login(LoginViewModel model, string
returnUrl = null)
{
    if (ModelState.IsValid)
    {
        // work with the model
    }
    // At this point, something failed, redisplay form
    return View(model);
}

```

The framework handles validating request data both on the client and on the server. Validation logic specified on model types is added to the rendered views as unobtrusive annotations and is enforced in the browser with [jQuery Validation](#).

## Dependency injection

ASP.NET Core has built-in support for [dependency injection \(DI\)](#). In ASP.NET Core MVC, [controllers](#) can request needed services through their constructors, allowing them to follow the [Explicit Dependencies Principle](#).

Your app can also use [dependency injection in view files](#), using the `@inject` directive:

C#HTML

```

@inject SomeService ServiceName

<!DOCTYPE html>
<html lang="en">
<head>
    <title>@ServiceName.GetTitle</title>
</head>
<body>
    <h1>@ServiceName.GetTitle</h1>

```

```
</body>
</html>
```

## Filters

[Filters](#) help developers encapsulate cross-cutting concerns, like exception handling or authorization. Filters enable running custom pre- and post-processing logic for action methods, and can be configured to run at certain points within the execution pipeline for a given request. Filters can be applied to controllers or actions as attributes (or can be run globally). Several filters (such as `Authorize`) are included in the framework.

`[Authorize]` is the attribute that is used to create MVC authorization filters.

```
C#
```

```
[Authorize]
public class AccountController : Controller
```

## Areas

[Areas](#) provide a way to partition a large ASP.NET Core MVC Web app into smaller functional groupings. An area is an MVC structure inside an application. In an MVC project, logical components like Model, Controller, and View are kept in different folders, and MVC uses naming conventions to create the relationship between these components. For a large app, it may be advantageous to partition the app into separate high level areas of functionality. For instance, an e-commerce app with multiple business units, such as checkout, billing, and search etc. Each of these units have their own logical component views, controllers, and models.

## Web APIs

In addition to being a great platform for building web sites, ASP.NET Core MVC has great support for building Web APIs. You can build services that reach a broad range of clients including browsers and mobile devices.

The framework includes support for HTTP content-negotiation with built-in support to [format data](#) as JSON or XML. Write [custom formatters](#) to add support for your own formats.

Use link generation to enable support for hypermedia. Easily enable support for [Cross-Origin Resource Sharing \(CORS\)](#) [↗](#) so that your Web APIs can be shared across multiple

Web applications.

## Testability

The framework's use of interfaces and dependency injection make it well-suited to unit testing, and the framework includes features (like a TestHost and InMemory provider for Entity Framework) that make [integration tests](#) quick and easy as well. Learn more about [how to test controller logic](#).

## Razor view engine

[ASP.NET Core MVC views](#) use the [Razor view engine](#) to render views. Razor is a compact, expressive and fluid template markup language for defining views using embedded C# code. Razor is used to dynamically generate web content on the server. You can cleanly mix server code with client side content and code.

C#HTML

```
<ul>
    @for (int i = 0; i < 5; i++) {
        <li>List item @i</li>
    }
</ul>
```

Using the Razor view engine you can define [layouts](#), [partial views](#) and replaceable sections.

## Strongly typed views

Razor views in MVC can be strongly typed based on your model. Controllers can pass a strongly typed model to views enabling your views to have type checking and IntelliSense support.

For example, the following view renders a model of type `IEnumerable<Product>`:

C#HTML

```
@model IEnumerable<Product>
<ul>
    @foreach (Product p in Model)
    {
        <li>@p.Name</li>
    }
</ul>
```

```
}  
</ul>
```

## Tag Helpers

[Tag Helpers](#) enable server side code to participate in creating and rendering HTML elements in Razor files. You can use tag helpers to define custom tags (for example, `<environment>`) or to modify the behavior of existing tags (for example, `<label>`). Tag Helpers bind to specific elements based on the element name and its attributes. They provide the benefits of server-side rendering while still preserving an HTML editing experience.

There are many built-in Tag Helpers for common tasks - such as creating forms, links, loading assets and more - and even more available in public GitHub repositories and as NuGet packages. Tag Helpers are authored in C#, and they target HTML elements based on element name, attribute name, or parent tag. For example, the built-in `LinkTagHelper` can be used to create a link to the `Login` action of the `AccountsController`:

C#HTML

```
<p>  
    Thank you for confirming your email.  
    Please <a asp-controller="Account" asp-action="Login">Click here to Log  
in</a>.  
</p>
```

The `EnvironmentTagHelper` can be used to include different scripts in your views (for example, raw or minified) based on the runtime environment, such as Development, Staging, or Production:

C#HTML

```
<environment names="Development">  
    <script src="~/lib/jquery/dist/jquery.js"></script>  
</environment>  
<environment names="Staging,Production">  
    <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.1.4.js"  
        asp-fallback-src="~/lib/jquery/dist/jquery.js"  
        asp-fallback-test="window.jQuery">  
    </script>  
</environment>
```

Tag Helpers provide an HTML-friendly development experience and a rich IntelliSense environment for creating HTML and Razor markup. Most of the built-in Tag Helpers

target existing HTML elements and provide server-side attributes for the element.

## View Components


[View Components](#) allow you to package rendering logic and reuse it throughout the application. They're similar to [partial views](#), but with associated logic.

## 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](#).

## Additional resources

- [MyTested.AspNetCore.Mvc - Fluent Testing Library for ASP.NET Core MVC](#)  : Strongly-typed unit testing library, providing a fluent interface for testing MVC and web API apps. *(Not maintained or supported by Microsoft.)*
- [Dependency injection in ASP.NET Core](#)

# 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:

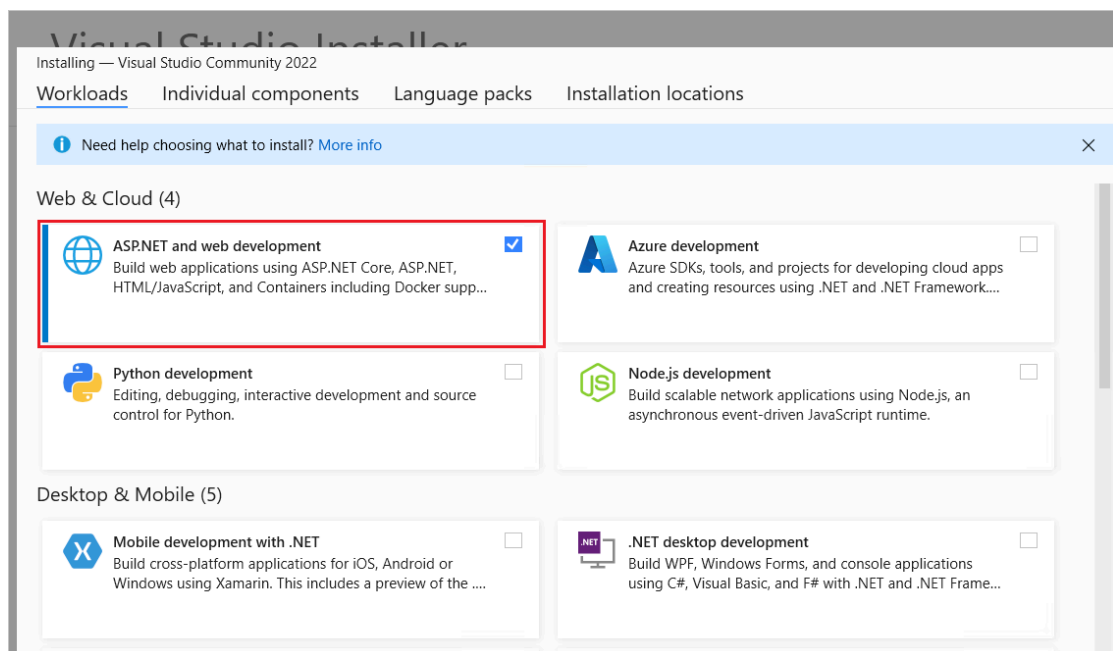
- ✓ Create a web app.
- ✓ Add and scaffold a model.
- ✓ Work with a database.
- ✓ Add search and validation.

[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**.