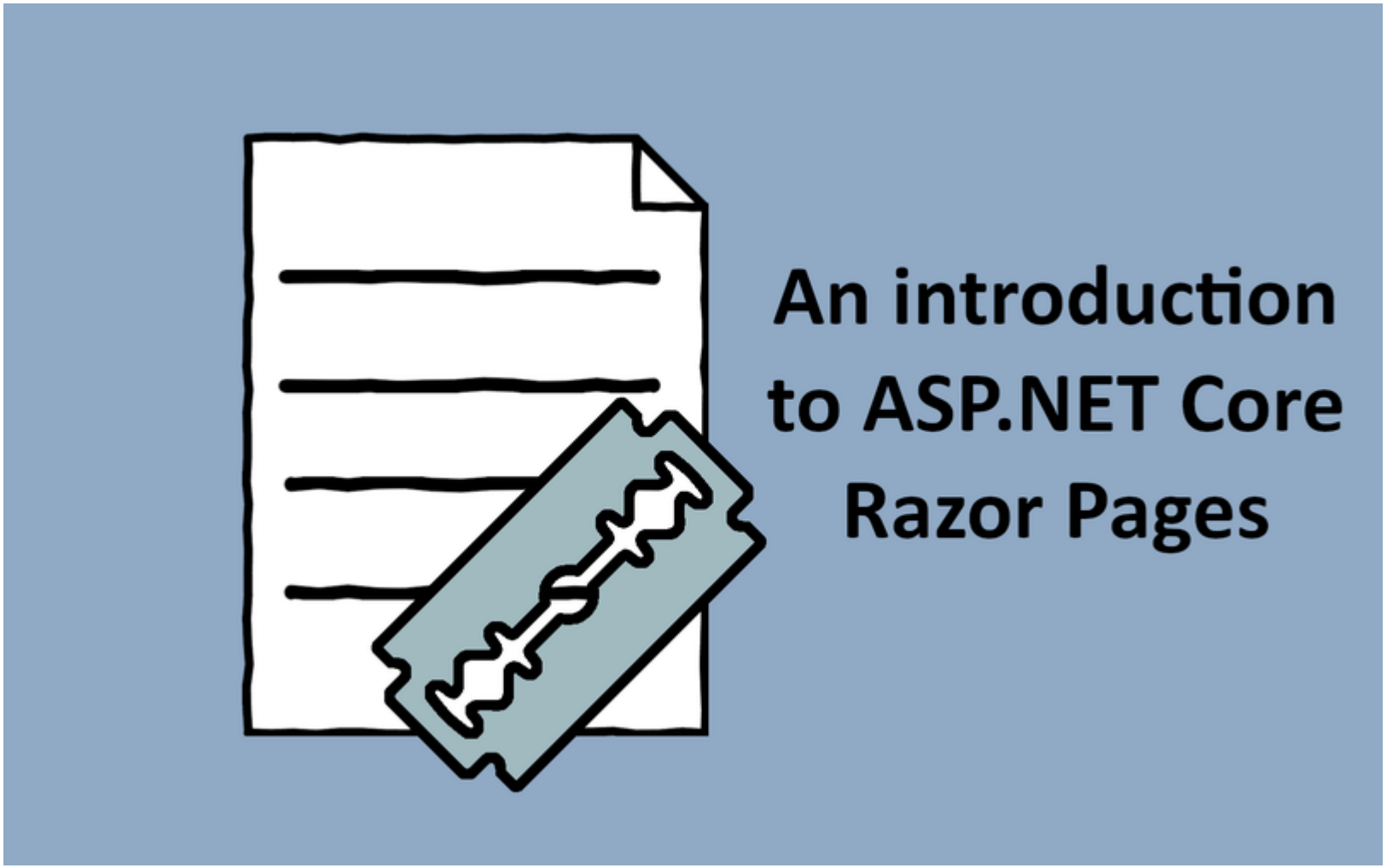
BY ANDREW LOCK • 2019-01-09

TWITTER FACEBOOK LINKEDIN

# Getting Started with ASP.NET Core Razor Pages




Razor Pages is a new aspect of ASP.NET Core MVC introduced in ASP.NET Core 2.0. It offers a "page-based" approach for building server-side rendered apps in ASP.NET Core and can coexist with "traditional" MVC or Web API controllers. In this post I provide an introduction to Razor Pages, the basics of getting started, and how Razor Pages differs from MVC.

If you would like to see a full integration of Twilio APIs in a .NET Core application then checkout this free 5-part video series. It's separate from this blog post tutorial but will give you a full run down of many APIs at once.

# Razor Pages vs MVC

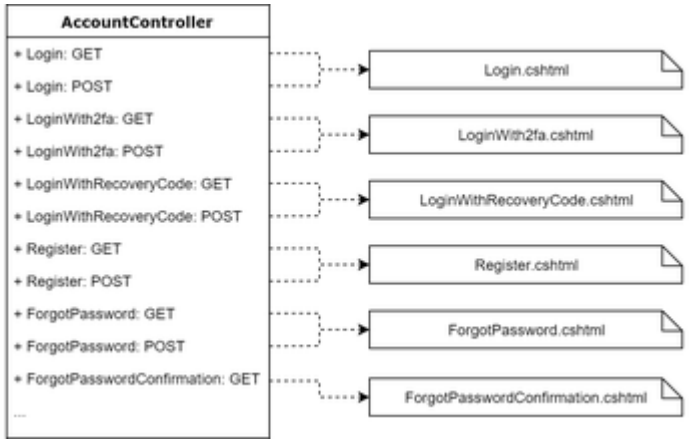
If you've used ASP.NET Core for building server-side rendered apps, then you'll be familiar with the traditional Model-View-Controller (MVC) pattern. Razor Pages provides an abstraction over the top of MVC, which can make it better suited to some page-based apps.

In MVC, **controllers** are used to group similar **actions** together. When a request is received, **routing** directs the request to a single action method. This method typically performs some processing, and returns an `IActionResult`, commonly a `ViewResult` or a `RedirectResult`. If a `ViewResult` is returned, a Razor **view** is rendered using the provided **view model**.

MVC provides a lot of flexibility, so the grouping of actions into controllers can be highly discretionary, but you commonly group actions that are related in some way, such as by URL route or by function. For example, you might group by domain components so that in an ecommerce app actions related to "products" would be in the `ProductController`, while "cart" actions would be in the `CartController`. Alternatively, actions may be grouped based on technical aspects; for example, where all actions on a controller share a common set of authorization requirements.

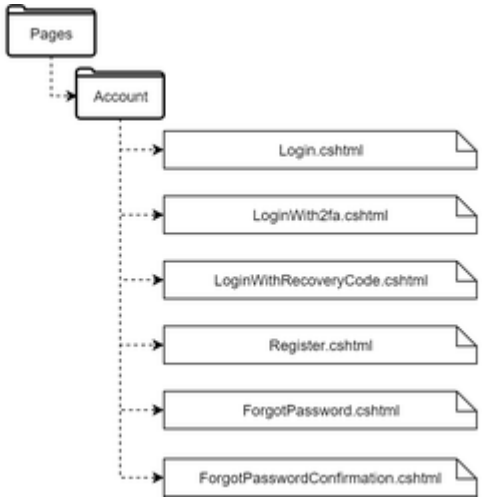
A common pattern you'll find is to have pairs of related actions inside a controller. This is especially true where you are using HTML forms, where you would typically have one action to handle the initial `GET` request, and another action for the `POST` request. Both of these actions use the same URL route and the same Razor view. From the point of view of a user (or the developer), they're logically two aspects of the same "page".

In some cases, you may find that your controllers are filled with these action method pairs. For example, the default ASP.NET Core Identity `AccountController` for an MVC app contains many such pairs:




The GET and POST pair of actions are highly coupled, as they both return the same view model, may need similar initialization logic, and use the same Razor view. The pair of actions are also related to the overall controller in which they're located (they're all related to identity and accounts), but they're more closely related to each other.

Razor Pages offers much the same functionality as traditional MVC, but using a slightly different model by taking advantage of this pairing. Each route (each pair of actions) becomes a separate Razor Page, instead of grouping many similar actions together under a single controller. That page can have multiple handlers that each respond to a different HTTP verb, but use the same view. The same Identity `AccountController` from above could therefore be rewritten in Razor Pages as shown below. In fact, as of ASP.NET Core 2.1, the new project templates use Razor Pages for Identity, even in an MVC app.




Razor Pages have the advantage of being highly cohesive. Everything related to a given page in your app is in one place. Contrast that with MVC controllers where some actions are highly correlated, but the controller *as a whole* is less cohesive.

Another good indicator for using Razor Pages is when your MVC controllers are just returning a Razor view with no significant processing required. A classic example is the `HomeController` from the ASP.NET Core 2.0 templates, which includes four actions:

```
1  public class HomeController : Controller
2  {
3      public IActionResult Index()
4      {
5          return View();
6      }
7
8      public IActionResult About()
9      {
10         ViewData["Message"] = "Your application description page.";
```

```
11
12          return View();
13      }
14
15      public IActionResult Contact()
16      {
17          ViewData["Message"] = "Your contact page.";
18
19          return View();
20      }
21
22      public IActionResult Error()
23      {
24          return View(new ErrorViewModel { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier })
25      }
26  }
```

These actions are not really related, but every action needs a controller, and the `HomeController` is a somewhat convenient location to put them. The Razor Pages equivalent places the `Index` (Home), `About`, `Contact`, and `Error` pages in the root directory, removing the implicit links between them. As an added bonus, everything related to the `About` page (for example) can be found in the files `About.cshtml` and `About.cshtml.cs`, which are located together on disk and in your solution explorer. That is in contrast to the MVC approach where controllers, view model, and view files are often located in completely different folders.

However, both of these approaches are *functionally* identical, so which one should you choose?

## When should you use Razor Pages?

It's important to realize that you don't have to go all-in with Razor Pages. Razor Pages uses exactly the same infrastructure as traditional MVC, so you can mix Razor Pages with MVC and Web API controllers all in the same app. Razor Pages also uses the same ASP.NET Core primitives as traditional MVC, so you still get model binding, validation, and action results.

From a maintainability point of view, I find the extra cohesion afforded by Razor Pages makes it preferable for new development. Not having to jump back and forth between the controller, view model, and view files is surprisingly refreshing!

Having said that, there are some situations where it may be preferable to stick with traditional MVC controllers:

- When you have a lot of MVC action filters on your controllers. You can use filters in Razor Pages too, but they generally provide less fine-grained control than for traditional MVC. For example, you can't apply Razor Page filters to individual handlers (for example, `GET` vs `POST` handlers) within a Razor Page.
- When your MVC controllers aren't rendering views. Razor Pages are focused around a "page" model, where you're rendering a view for the user. If your controllers are either Web API controllers, or aren't designed to provide pages a user navigates through, then Razor Pages don't really make sense.
- When your controller is already highly cohesive, and it makes sense to centralize the action methods in one file.

On the other hand, there are some situations where Razor Pages really shines:

- When your action methods have little or no logic and are just returning views (for example, the `HomeController` shown previously).
- When you have HTML forms with pairs of `GET` and `POST` actions. Razor Pages makes each pair a cohesive page, which I find requires less cognitive overhead when developing, rather than having to jump between multiple files.

- When you were previously using ASP.NET Web Pages (WebMatrix). This framework provided a lightweight page-based model, but it was completely separate from ASP.NET. In contrast, Razor Pages has a similar level of simplicity, but also the full power of ASP.NET Core when required.

Now you've seen the high-level differences between MVC and Razor Pages, it's time to dive into the specifics. How do you create a Razor Page, and how does it differ from a traditional Razor View?

## The Razor Page Model

Razor Pages are built on top of MVC, but they use a slightly different paradigm than the MVC pattern. With MVC the controller typically provides the logic and behavior for an action, ultimately producing a view model which contains data that is used to render the view. Razor Pages takes a slightly different approach, by using a **Page Model**.

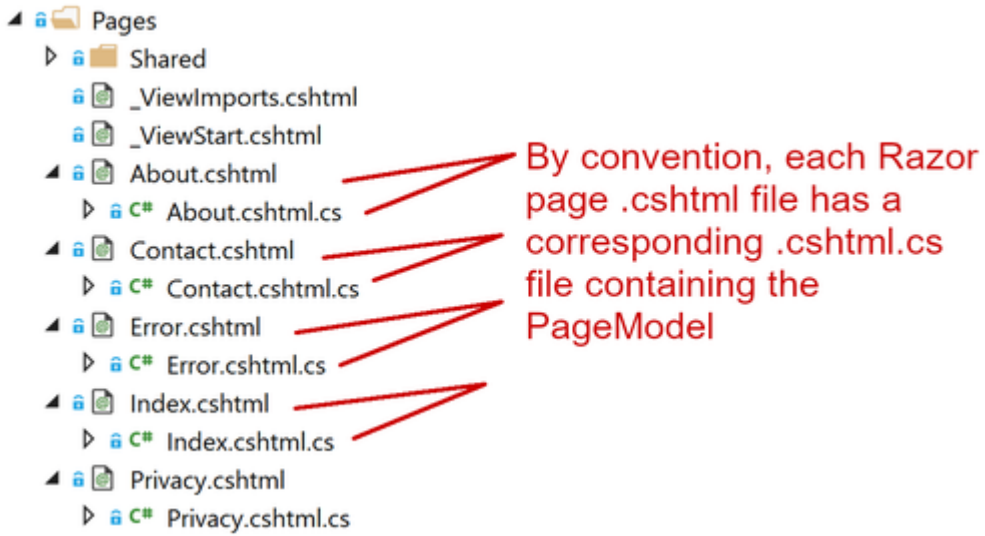
Compared to MVC, the page model acts as both a mini-controller and the view model for the view. It's responsible for both the behavior of the page and for exposing the data used to generate the view. This pattern is closer to the Model-View-ViewModel (MVVM) pattern used in some desktop and mobile frameworks, especially if the business logic is pushed out of the page model and into your "business" model.

In technical terms a Razor Page is very similar to a Razor view, except it has an `@page` directive at the top of the file:

```
1 @page
2
3 <div>The time is @DateTime.Now</div>
```

As with Razor views, any HTML in the Razor page is rendered to the client, and you can use the `@` symbol to render C# values or use C# control structures. See the documentation for a complete reference guide to Razor syntax.

Adding `@page` is all that's required to expose your page, but this page doesn't use a page model yet. More typically you create a class that derives from `PageModel` and associate it with your `cshtml` file. You can include your `PageModel` and Razor view in the same `cshtml` file if you wish, but best practice is to keep the `PageModel` in a "code-behind" file, and only include presentational data in the `cshtml` file. By convention, if your razor page is called `MyPage.cshtml`, the code-behind file should be named `MyPage.cshtml.cs`:




The `PageModel` class is the page model for the Razor view. When the Razor page is rendered, properties exposed on the `PageModel` are available in the `.cshtml` view. For example, you could expose a property for the current time in your `Index.cshtml.cs` file:

```
1  using System;
2  using Microsoft.AspNetCore.Mvc.RazorPages;
3
4  public class IndexModel: PageModel
5  {
6      public DateTime CurrentTime => DateTime.UtcNow;
7  }
```

And render it in your `Index.cshtml` file using standard Razor syntax:

```
1 @page
2 @model IndexModel
3
4 <div>The current time is @Model.CurrentTime.ToShortTimeString()</div>
```

If you're familiar with Razor views, this should be quite familiar. You declare the type of the model using the `@model` directive, which is exposed on the `Model` property. The difference is that instead of your MVC controller passing in a View Model of type `IndexModel`, the `PageModel` itself is exposed as the `Model` property.

## Routing in Razor Pages

Razor Pages, like MVC, uses a mixture of conventions, configuration, and declarative directives to control how your app behaves. They use the same routing infrastructure as MVC under the hood; the difference is in how routing is configured.

- For MVC and Web API, you use either attribute or convention-based routing to match an incoming URL to a controller and action.
- For Razor Pages, the path to the file on disk is used to calculate the URL at which the page can be reached. By convention, all Razor Pages are nested in the `Pages` directory.

For example, if you create a Razor Page in your app at `/Pages/MyFolder/Test.cshtml`, it would be exposed at the URL `/MyFolder/Test`. This is definitely a positive feature for working with Razor Pages—navigating and visualizing the URLs exposed by your app is as easy as viewing the file structure.

Having said that, Razor Page routes are fully customizable; if you need to expose your page at a route that *doesn't* correspond to it's path on disk, simply provide a route template in the `@page` directive. This can also include other route parameters, as shown below:

```
@page "/customroute/customized/{id?}"
```

This page would be exposed at the URL `/customroute/customized`, and it could also bind an optional `id` segment in the URL, `/customroute/customized/123`, for example. The `id` value can be bound to a `PageModel` property using model binding.

## Model binding in Razor Pages

In MVC the method parameters of an action in a controller are bound to the incoming request by matching values in the URL, query string, or body of the request as appropriate (see the documentation for details). In Razor Pages the incoming request is bound to the properties of the `PageModel` instead.

For security reasons, you have to explicitly opt-in to the properties to bind, by decorating properties to bind with the `[BindProperty]` attribute:

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

public class IndexModel : PageModel
{
    [BindProperty]
    public string Search { get;set; }

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

In this example, the `Search` property would be bound to the request as it's decorated with `[BindProperty]`, but `CurrentTime` would not be bound. For GET requests, you have to go one step further and set the `SupportsGet` property on the attribute:

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

public class IndexModel : PageModel
{
    [BindProperty(SupportsGet = true)]
    public string Search { get;set; }
}
```

If you're binding complex models, such as a form post-back, then adding the `[BindProperty]` attribute everywhere could get tedious. Instead I like to create a single property as the "input model" and decorate this with `[BindProperty]`. This keeps your `PageModel` public surface area explicit and controlled. A common extension to this approach is to make your input model a nested class. This often makes sense as you commonly don't want to use your UI-layer models elsewhere in your app:

```
using System.ComponentModel.DataAnnotations;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

public class IndexModel : PageModel
{
    public bool IsEmailConfirmed { get; set; }

    [BindProperty]
    public InputModel Input { get; set; }

    public class InputModel
    {
```

```
14          [Required, EmailAddress]
15          public string Email { get; set; }
16
17          [Required, Phone, Display(Name = "Phone number")]
18          public string PhoneNumber { get; set; }
19      }
20  }
```

In this example only the property `Input` is bound to the incoming request. This uses the nested `InputModel` class, to define all the expected values to bind. If you need to bind another value, you can add another property to `InputModel`.

## Handling multiple HTTP verbs with Razor Page Handlers

One of the main selling points of Razor Pages is the extra cohesion they can bring over using MVC controllers. A single Razor Page contains all of the UI code associated with a given Razor view by using **page handlers** to respond to requests.

Page handlers are analogous to action methods in MVC. When a Razor Page receives a request, a single handler is selected to run, based on the incoming request and the handler names. Handlers are matched via a naming convention, `On{Verb}[Async]`, where `{Verb}` is the HTTP method, and `[async]` is optional. For example:

- `OnPost` and `OnPostAsync` run in response to `POST` requests
- `OnGet` and `OnGetAsync` run in response to `GET` requests (and optionally `HEAD` requests in ASP.NET Core 2.1+).

For HTML forms, it's very common to have an `OnGet` handler that displays the initial empty form, and an `OnPost` handler which handles the `POST` back from the client. For example the following form shows the code-behind for a Razor Page that updates a user's display name.

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;

public class UpdateDisplayNameModel : PageModel
{
    private readonly IUserService _userService;
    public IndexModel(IUserService userService)
    {
        _userService = userService;
    }

    [BindProperty]
    public InputModel Input { get; set; }

    public void OnGet()
    {
        Input.DisplayName = _userService.GetDefaultDisplayName();
    }

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

        await _userService.UpdateDisplayName(User, Input.DisplayName);
        return RedirectToPage("/Index");
    }
```

```
34
35      public class InputModel
36      {
37          [Required, StringLength(50)]
38          public string DisplayName { get; set; }
39      }
40  }
```

This Razor Page uses a fictional `IUserService` which is injected into the `PageModel` constructor. The `OnGet` page handler runs when the form is initially requested, and sets the default display name to a value obtained from the `IUserService`. The form is sent to the client, who fills in the details and posts it back.

The `OnPostAsync` handler runs in response to the `POST`, and follows a similar pattern to MVC action methods. You should first check that the `PageModel` passes model validation using `ModelState.IsValid`, and if not re-display the form using `Page()`. `Page()` is semantically equivalent to the `View()` method in MVC controllers; it's used to render the view and return a response.

If the `PageModel` is valid, the form uses the provided `DisplayName` value to update the name of the currently logged in user, `User`. The `PageModel` provides access to many of the same properties that the `Controller` base class does, such as `HttpContext`, `Request`, and in this case, `User`. Finally, the handler redirects to another Razor Page using the `RedirectToPage()` method. This is functionally equivalent to the MVC `RedirectToAction()` method.

The `OnGet` and `OnPost` pair of handlers are common when a form only has a single possible role, but it's also possible to have a single Razor Page with *multiple* handlers for the same verb. To create a named handler, use the naming convention `On{Verb}{Handler}[Async]`. For example, maybe we want to add a handler to the `UpdateDisplayName` Razor page that allows users to reset their username to the default. We could add the following `ResetName` handler to the existing Razor page:

```
1  public async Task<IActionResult> OnPostResetNameAsync()
2  {
```

```
    await _userService.ResetDisplayName(User);
    return RedirectToPage("/Index");
}
```

To invoke the handler, you pass the handler name in the query string for the `POST`, for example `?handler=resetName`. This ensures the named handler is invoked instead of the default `OnPostAsync` handler. If you don't like the use of query strings here, you can use a custom route and include the handler name in a path segment instead.

This section showed handlers for `GET` and `POST` but it's also possible to have page handlers for other HTTP verbs like `DELETE`, `PUT`, and `PATCH`. These verbs are generally not used by HTML forms, so will not commonly be required in a page-oriented Razor Pages app. However, they follow the same naming conventions and behavior as other page handlers should you need them to be called by an API for some reason.

## Using tag helpers in Razor Pages

When you're working with MVC actions and views, ASP.NET Core provides various tag helpers such as `asp-action` and `asp-controller` for generating links to your actions from Razor views. Razor Pages have equivalent tag helpers, where you can use `asp-page` to generate the path to a specific Razor page based, and `asp-page-handler` to set a specific handler.

For example, you could create a "reset name" button in your form using both the `asp-page` and `asp-page-handler` tags:

```
<button asp-page="/Index" asp-page-handler="ResetName" type="submit">Reset Display Name</button>
```

## Summary

Razor Pages are a new aspect of ASP.NET Core MVC that were introduced in ASP.NET Core 2.0. They are built on top of existing ASP.NET Core primitives and provide the same overall functionality as traditional MVC, but with a page-based model. For many

apps, the page-based approach using a `PageModel` can result in more cohesive code than traditional MVC. Razor Pages can be used seamlessly in the same app as traditional MVC or Web API controllers, so you only need to use it where it is a good fit.

If you're creating a new app using Razor, I strongly suggest considering Razor Pages as the default approach. It may feel strange for experienced MVC developers at first, but I've been pleasantly surprised by the improved development experience. For existing MVC apps, adding new Razor Pages is easy—but it's unlikely to be worth migrating a whole MVC app to use them. They're functionally identical to MVC, so the main benefit is a workflow that's more convenient for common development tasks.

## Additional Resources

Learn Razor Pages is a tutorial website setup by Microsoft MVP Mike Brind. It's a great resource that I highly recommend in addition to the official documentation.

Andrew Lock is a Microsoft MVP and author of ASP.NET Core in Action by Manning. He can be reached on Twitter at @andrewlocknet, or via his blog at https://andrewlock.net.

JAVA .NET RUBY PHP PYTHON SWIFT ARDUINO JAVASCRIPT

## POSTS BY PRODUCT

SMS AUTHY VOICE TWILIO CLIENT MMS VIDEO TASK ROUTER FLEX SIP IOT PROGRAMMABLE CHAT STUDIO

## CATEGORIES

Code, Tutorials and Hacks

Customer Highlights

Developers Drawing The Owl

News

Stories From The Road

The Owl's Nest: Inside Twilio

TWITTER FACEBOOK

## Tutorials

Sample applications that cover common use cases in a variety of languages. Download, test drive, and tweak them yourself.

Get started