Introduction to Razor Pages in ASP.NET Core

Sursa:

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

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 (link: https://docs.microsoft.com/en-us/aspnet/core/tutorials/razor-pages/razor-pages-start?view=aspnetcore-3.1&tabs=visual-studio).

For an overview of ASP.NET Core, see the Introduction to ASP.NET Core.

Prerequisites

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac
  - Visual Studio 2019 with the **ASP.NET and web development** workload
  - .NET Core 3.0 SDK or later

Create a Razor Pages project

See Get started with Razor Pages for detailed instructions on how to create a Razor Pages project.

Razor Pages

Razor Pages is enabled in *Startup.cs*:

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }
```

```
    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

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

        app.UseRouting();

        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
        });
    }
}
```

Consider a basic page:

```
@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:

```
@page
@using RazorPagesIntro.Pages   <!-- Namespace for Index2Model class -->
@model Index2Model

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

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

```
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:

TABLE 1

File name and path	matching URL
/Pages/Index.cshtml	`/` or `/Index`
/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 all *just 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 Startup.cs file.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<CustomerDbContext>(options =>
                    options.UseInMemoryDatabase("name"));
    services.AddRazorPages();
}
```

The data model:

```
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:

```
using Microsoft.EntityFrameworkCore;
using RazorPagesContacts.Models;

namespace RazorPagesContacts.Data
{
    public class CustomerDbContext : DbContext
```

```
    {
        public CustomerDbContext(DbContextOptions options)
            : base(options)
        {
        }
        public DbSet<Customer> Customers { get; set; }
    }
}
```

The *Pages/Create.cshtml* view file:

```
@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/Create.cshtml.cs* page model:

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

namespace RazorPagesContacts.Pages.Customers
{
    public class CreateModel : PageModel
    {
        private readonly CustomerDbContext _context;

        public CreateModel(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();
            }

            _context.Customers.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 *CreateModel.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:

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

    _context.Customers.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/Create.cshtml* view file:

```
@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/Create.cshtml*:

```
<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 handing is covered later in this document.

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

    _context.Customers.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:

```
public class CreateModel : PageModel
{
```

```
    private readonly CustomerDbContext _context;

    public CreateModel(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();
        }

        _context.Customers.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`:

```
[BindProperty(SupportsGet = true)]
```

For more information, see **ASP.NET Core Community Standup: Bind on GET discussion (YouTube)**.

Reviewing the *Pages/Create.cshtml* view file:

```
@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:

```
@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>
            </tr>
        </thead>
        <tbody>
            @foreach (var contact in Model.Customer)
            {
                <tr>
                    <td> @contact.Id  </td>
```

```
                    <td>@contact.Name</td>
                    <td>
                        <a asp-page="./Edit" asp-route-id="@contact.Id">Edit</a> |
                        <button type="submit" asp-page-handler="delete"
                                asp-route-id="@contact.Id">delete
                        </button>
                    </td>
                </tr>
            }
        </tbody>
    </table>
    <a asp-page="Create">Create New</a>
</form>
```

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

```
public class IndexModel : PageModel
{
    private readonly CustomerDbContext _context;

    public IndexModel(CustomerDbContext context)
    {
        _context = context;
    }

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

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

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

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

        return RedirectToPage();
    }
}
```

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

```
<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:

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

The rendered HTML:

```
<button type="submit" formaction="/Customers?id=1&amp;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.

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

    if (contact != null)
    {
        _context.Customers.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

```
@page "{id:int}"
@model RazorPagesContacts.Pages.Customers.EditModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<h1>Edit Customer - @Model.Customer.Id</h1>
<form method="post">
    <div asp-validation-summary="All"></div>
    <input asp-for="Customer.Id" type="hidden" />
    <div>
        <label asp-for="Customer.Name"></label>
        <div>
            <input asp-for="Customer.Name" />
            <span asp-validation-for="Customer.Name"></span>
        </div>
    </div>

    <div>
        <button type="submit">Save</button>
    </div>
</form>
```

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:

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

The *Edit.cshtml.cs* file:

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

    public EditModel(CustomerDbContext context)
    {
        _context = context;
    }
```

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

    public async Task<IActionResult> OnGetAsync(int id)
    {
        Customer = await _context.Customers.FindAsync(id);

        if (Customer == null)
        {
            return RedirectToPage("./Index");
        }

        return Page();
    }

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

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

        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            throw new Exception($"Customer {Customer.Id} not found!");
        }

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

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:

```
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:

```
@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:

```
<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:

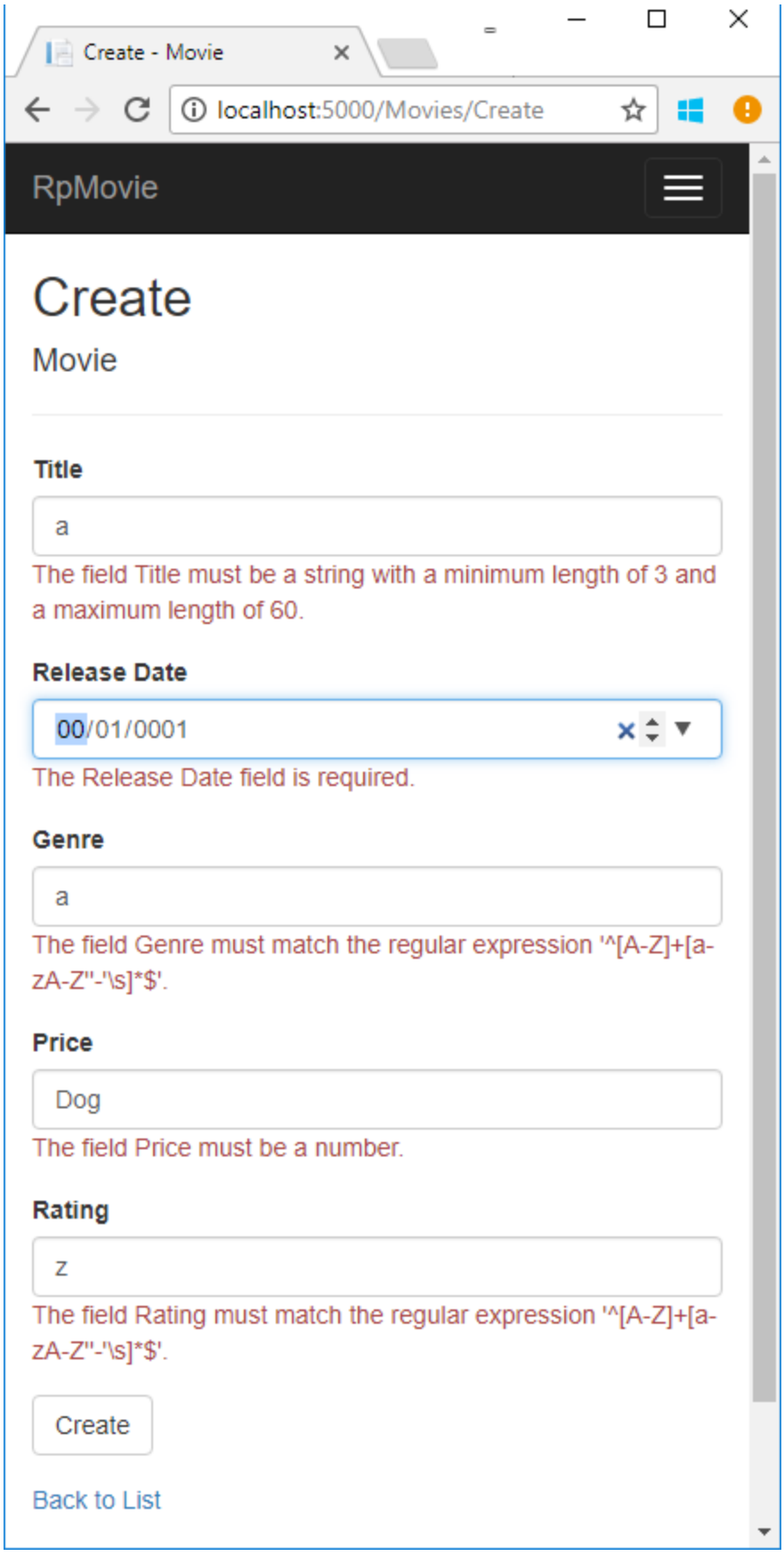
```
public class Movie
{
    public int ID { get; set; }

    [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:


Create - Movie
localhost:5000/Movies/Create
RpMovie
Create
Movie
Title
a
The field Title must be a string with a minimum length of 3 and a maximum length of 60.
Release Date
00/01/0001
The Release Date field is required.
Genre
a
The field Genre must match the regular expression '^[A-Z]+[a-zA-Z''-'\s]*$'.
Price
Dog
The field Price must be a number.
Rating
z
The field Rating must match the regular expression '^[A-Z]+[a-zA-Z''-'\s]*$'.
Create
Back to List

For more information, see:

- Add validation to the Movie app
- Model validation in ASP.NET Core.

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:

```
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*:

```
<!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*:

```
@{
    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:

```
@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:

```
@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:

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

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

        // Code removed for brevity.
```

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

```
@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/Create.cshtml* view file:

```
@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/Create.cshtml* view file with *_ViewImports.cshtml* and the preceding layout file:

```
@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`:

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

    public CreateModel(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();
        }

        _context.Customers.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*.

TABLE 2

RedirectToPage(x)	Page
RedirectToPage("/Index")	*Pages/Index*
RedirectToPage("./Index");	*Pages/Customers/Index*
RedirectToPage("../Index")	*Pages/Index*
RedirectToPage("Index")	*Pages/Customers/Index*

`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:

```
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:

```
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:

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

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

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

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`:

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

```
    public CreateDotModel(AppDbContext 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`.

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

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

```
[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:

```
@page
@model CreateFATHModel

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

```
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 AppDbContext _db;

        public CreateFATHModel(AppDbContext 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 OnPost**JoinList**Async and OnPost**JoinListUC**Async.
With *OnPost* and *Async* removed, the handler names are `JoinList` and `JoinListUC`.

```
<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.

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

<html>
<body>
    <p>
        Enter your name.
    </p>
    <div asp-validation-summary="All"></div>
    <form method="POST">
        <div>Name: <input asp-for="Customer.Name" /></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.

Advanced configuration and settings

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

To configure advanced options, use the extension method AddRazorPagesOptions:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages()
        .AddRazorPagesOptions(options =>
        {
            options.RootDirectory = "/MyPages";
            options.Conventions.AuthorizeFolder("/MyPages/Admin");
        });
}
```

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:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages()
        .AddRazorPagesOptions(options =>
        {
            options.Conventions.AuthorizeFolder("/MyPages/Admin");
        })
        .WithRazorPagesAtContentRoot();
}
```

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):

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages()
        .AddRazorPagesOptions(options =>
        {
            options.Conventions.AuthorizeFolder("/MyPages/Admin");
        })
        .WithRazorPagesRoot("/path/to/razor/pages");
}
```