# Model validation in ASP.NET Core MVC and Razor Pages

Article • 08/30/2024 • 25 contributors

Feedback

In this article



This article explains how to validate user input in an ASP.NET Core MVC or Razor Pages app.

View or download sample code (how to download).

## Model state

Model state represents errors that come from two subsystems: model binding and model validation. Errors that originate from model binding are generally data conversion errors. For example, an "x" is entered in an integer field. Model validation occurs after model binding and reports errors where data doesn't conform to business rules. For example, a 0 is entered in a field that expects a rating between 1 and 5.

Both model binding and model validation occur before the execution of a controller action or a

Razor Pages handler method. For web apps, it's the app's responsibility to inspect `ModelState.IsValid` and react appropriately. Web apps typically redisplay the page with an error message, as shown in the following Razor Pages example:

C# Copy

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

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

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

For ASP.NET Core MVC with controllers and views, the following example shows how to check `ModelState.IsValid` inside of a controller action:

C# Copy

```
public async Task<IActionResult> Create(Movie movie)
{
    if (!ModelState.IsValid)
    {
        return View(movie);
    }

    _context.Movies.Add(movie);
    await _context.SaveChangesAsync();

    return RedirectToAction(nameof(Index));
}
```

Web API controllers don't have to check `ModelState.IsValid` if they have the [ApiController] attribute. In that case, an automatic HTTP 400 response containing error details is returned when model state is invalid. For more information, see Automatic HTTP 400 responses.

# Rerun validation

Validation is automatic, but you might want to repeat it manually. For example, you might compute a value for a property and want to rerun validation after setting the property to the

computed value. To rerun validation, call ModelStateDictionary.ClearValidationState to clear validation specific to the model being validated followed by `TryValidateModel`:

C# Copy

```
public async Task<IActionResult> OnPostTryValidateAsync()
{
    var modifiedReleaseDate = DateTime.Now.Date;
    Movie.ReleaseDate = modifiedReleaseDate;

    ModelState.ClearValidationState(nameof(Movie));
    if (!TryValidateModel(Movie, nameof(Movie)))
    {
        return Page();
    }

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

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

# Validation attributes

Validation attributes let you specify validation rules for model properties. The following example from the sample app shows a model class that is annotated with validation attributes. The `[ClassicMovie]` attribute is a custom validation attribute and the others are built in. Not shown is `[ClassicMovieWithClientValidator]`, which shows an alternative way to implement a custom attribute.

C# Copy

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

    [Required]
    [StringLength(100)]
    public string Title { get; set; } = null!;

    [ClassicMovie(1960)]
    [DataType(DataType.Date)]
    [Display(Name = "Release Date")]
    public DateTime ReleaseDate { get; set; }

    [Required]
    [StringLength(1000)]
```

```
    public string Description { get; set; } = null!;

    [Range(0, 999.99)]
    public decimal Price { get; set; }

    public Genre Genre { get; set; }

    public bool Preorder { get; set; }
}
```

# Built-in attributes

Here are some of the built-in validation attributes:

- [ValidateNever]: Indicates that a property or parameter should be excluded from validation.
- [CreditCard]: Validates that the property has a credit card format. Requires jQuery Validation Additional Methods.
- [Compare]: Validates that two properties in a model match.
- [EmailAddress]: Validates that the property has an email format.
- [Phone]: Validates that the property has a telephone number format.
- [Range]: Validates that the property value falls within a specified range.
- [RegularExpression]: Validates that the property value matches a specified regular expression.
- [Required]: Validates that the field isn't null. See [Required] attribute for details about this attribute's behavior.
- [StringLength]: Validates that a string property value doesn't exceed a specified length limit.
- [Url]: Validates that the property has a URL format.
- [Remote]: Validates input on the client by calling an action method on the server. See [Remote] attribute for details about this attribute's behavior.

A complete list of validation attributes can be found in the System.ComponentModel.DataAnnotations namespace.

## Error messages

Validation attributes let you specify the error message to be displayed for invalid input. For example:

C# Copy

```
[StringLength(8, ErrorMessage = "Name length can't be more than 8.")]
```

Internally, the attributes call String.Format with a placeholder for the field name and sometimes additional placeholders. For example:

C# Copy

```
[StringLength(8, ErrorMessage = "{0} length must be between {2} and {1}.", MinimumLen
```

When applied to a `Name` property, the error message created by the preceding code would be "Name length must be between 6 and 8.".

To find out which parameters are passed to `String.Format` for a particular attribute's error message, see the DataAnnotations source code.

## Use JSON property names in validation errors

By default, when a validation error occurs, model validation produces a ModelStateDictionary with the property name as the error key. Some apps, such as single page apps, benefit from using JSON property names for validation errors generated from Web APIs. The following code configures validation to use the SystemTextJsonValidationMetadataProvider to use JSON property names:

C# Copy

```
using Microsoft.AspNetCore.Mvc.ModelBinding.Metadata;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers(options =>
{
    options.ModelMetadataDetailsProviders.Add(new SystemTextJsonValidationMetadataPro
});

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

The following code configures validation to use the NewtonsoftJsonValidationMetadataProvider to use JSON property name when using Json.NET:

```
C#
using Microsoft.AspNetCore.Mvc.NewtonsoftJson;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers(options =>
{
    options.ModelMetadataDetailsProviders.Add(new NewtonsoftJsonValidationMetadataPro
}).AddNewtonsoftJson();

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

For an example of the policy to use camel-casing, see Program.cs on GitHub.

# Non-nullable reference types and [Required] attribute

The validation system treats non-nullable parameters or bound properties as if they had a `[Required(AllowEmptyStrings = true)]` attribute. By enabling Nullable contexts, MVC implicitly starts validating non-nullable properties or parameters as if they had been attributed with the `[Required(AllowEmptyStrings = true)]` attribute. Consider the following code:

```
C#
public class Person
{
    public string Name { get; set; }
}
```

If the app was built with `<Nullable>enable</Nullable>`, a missing value for `Name` in a JSON or

form post results in a validation error. This may seem contradictory since the `[Required(AllowEmptyStrings = true)]` attribute is implied, but this is expected behavior because empty strings are converted to null by default. Use a nullable reference type to allow null or missing values to be specified for the `Name` property:

C# Copy

```
public class Person
{
    public string? Name { get; set; }
}
```

This behavior can be disabled by configuring SuppressImplicitRequiredAttributeForNonNullableReferenceTypes in `Program.cs`:

C# Copy

```
builder.Services.AddControllers(
    options => options.SuppressImplicitRequiredAttributeForNonNullableReferenceTypes
```

## [Required] validation on the server

On the server, a required value is considered missing if the property is null. A non-nullable field is always valid, and the `[Required]` attribute's error message is never displayed.

However, model binding for a non-nullable property may fail, resulting in an error message such as `The value '' is invalid`. To specify a custom error message for server-side validation of non-nullable types, you have the following options:

- Make the field nullable (for example, `decimal?` instead of `decimal`). Nullable<T> value types are treated like standard nullable types.
- Specify the default error message to be used by model binding, as shown in the following example:

  C# Copy

  ```
  builder.Services.AddRazorPages()
      .AddMvcOptions(options =>
      {
          options.MaxModelValidationErrors = 50;
          options.ModelBindingMessageProvider.SetValueMustNotBeNullAccessor(
              _ => "The field is required.");
  ```

```
    });

builder.Services.AddSingleton
    <IValidationAttributeAdapterProvider, CustomValidationAttributeAdapterProvid
```

For more information about model binding errors that you can set default messages for, see DefaultModelBindingMessageProvider.

## [Required] validation on the client

Non-nullable types and strings are handled differently on the client compared to the server. On the client:

- A value is considered present only if input is entered for it. Therefore, client-side validation handles non-nullable types the same as nullable types.
- Whitespace in a string field is considered valid input by the jQuery Validation required method. Server-side validation considers a required string field invalid if only whitespace is entered.

As noted earlier, non-nullable types are treated as though they had a `[Required(AllowEmptyStrings = true)]` attribute. That means you get client-side validation even if you don't apply the `[Required(AllowEmptyStrings = true)]` attribute. But if you don't use the attribute, you get a default error message. To specify a custom error message, use the attribute.

# [Remote] attribute

The [Remote] attribute implements client-side validation that requires calling a method on the server to determine whether field input is valid. For example, the app may need to verify whether a user name is already in use.

To implement remote validation:

1. Create an action method for JavaScript to call. The jQuery Validation remote method expects a JSON response:

    - `true` means the input data is valid.
    - `false`, `undefined`, or `null` means the input is invalid. Display the default error message.
    - Any other string means the input is invalid. Display the string as a custom error message.

Here's an example of an action method that returns a custom error message:

```
C#
[AcceptVerbs("GET", "POST")]
public IActionResult VerifyEmail(string email)
{
    if (!_userService.VerifyEmail(email))
    {
        return Json($"Email {email} is already in use.");
    }

    return Json(true);
}
```

2. In the model class, annotate the property with a `[Remote]` attribute that points to the validation action method, as shown in the following example:

```
C#
[Remote(action: "VerifyEmail", controller: "Users")]
public string Email { get; set; } = null!;
```

Server side validation also needs to be implemented for clients that have disabled JavaScript.

## Additional fields

The AdditionalFields property of the `[Remote]` attribute lets you validate combinations of fields against data on the server. For example, if the `User` model had `FirstName` and `LastName` properties, you might want to verify that no existing users already have that pair of names. The following example shows how to use `AdditionalFields`:

```
C#
[Remote(action: "VerifyName", controller: "Users", AdditionalFields = nameof(LastName
[Display(Name = "First Name")]
public string FirstName { get; set; } = null!;

[Remote(action: "VerifyName", controller: "Users", AdditionalFields = nameof(FirstNam
[Display(Name = "Last Name")]
public string LastName { get; set; } = null!;
```

`AdditionalFields` could be set explicitly to the strings "FirstName" and "LastName", but using

the nameof operator simplifies later refactoring. The action method for this validation must accept both `firstName` and `lastName` arguments:

C# Copy

```
[AcceptVerbs("GET", "POST")]
public IActionResult VerifyName(string firstName, string lastName)
{
    if (!_userService.VerifyName(firstName, lastName))
    {
        return Json($"A user named {firstName} {lastName} already exists.");
    }

    return Json(true);
}
```

When the user enters a first or last name, JavaScript makes a remote call to see if that pair of names has been taken.

To validate two or more additional fields, provide them as a comma-delimited list. For example, to add a `MiddleName` property to the model, set the `[Remote]` attribute as shown in the following example:

C# Copy

```
[Remote(action: "VerifyName", controller: "Users",
    AdditionalFields = nameof(FirstName) + "," + nameof(LastName))]
public string MiddleName { get; set; }
```

`AdditionalFields`, like all attribute arguments, must be a constant expression. Therefore, don't use an interpolated string or call Join to initialize `AdditionalFields`.

# Alternatives to built-in attributes

If you need validation not provided by built-in attributes, you can:

- Create custom attributes.
- Implement IValidatableObject.

# Custom attributes

For scenarios that the built-in validation attributes don't handle, you can create custom validation attributes. Create a class that inherits from ValidationAttribute, and override the

IsValid method.

The `IsValid` method accepts an object named *value*, which is the input to be validated. An overload also accepts a ValidationContext object, which provides additional information, such as the model instance created by model binding.

The following example validates that the release date for a movie in the *Classic* genre isn't later than a specified year. The `[ClassicMovie]` attribute:

- Is only run on the server.
- For Classic movies, validates the release date:

C# Copy

```
public class ClassicMovieAttribute : ValidationAttribute
{
    public ClassicMovieAttribute(int year)
        => Year = year;

    public int Year { get; }

    public string GetErrorMessage() =>
        $"Classic movies must have a release year no later than {Year}.";

    protected override ValidationResult? IsValid(
        object? value, ValidationContext validationContext)
    {
        var movie = (Movie)validationContext.ObjectInstance;
        var releaseYear = ((DateTime)value!).Year;

        if (movie.Genre == Genre.Classic && releaseYear > Year)
        {
            return new ValidationResult(GetErrorMessage());
        }

        return ValidationResult.Success;
    }
}
```

The `movie` variable in the preceding example represents a `Movie` object that contains the data from the form submission. When validation fails, a ValidationResult with an error message is returned.

# IValidatableObject

The preceding example works only with `Movie` types. Another option for class-level validation

is to implement IValidatableObject in the model class, as shown in the following example:

C# Copy

```
public class ValidatableMovie : IValidatableObject
{
    private const int _classicYear = 1960;

    public int Id { get; set; }

    [Required]
    [StringLength(100)]
    public string Title { get; set; } = null!;

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

    [Required]
    [StringLength(1000)]
    public string Description { get; set; } = null!;

    [Range(0, 999.99)]
    public decimal Price { get; set; }

    public Genre Genre { get; set; }

    public bool Preorder { get; set; }

    public IEnumerable<ValidationResult> Validate(ValidationContext validationContext
    {
        if (Genre == Genre.Classic && ReleaseDate.Year > _classicYear)
        {
            yield return new ValidationResult(
                $"Classic movies must have a release year no later than {_classicYear
                new[] { nameof(ReleaseDate) });
        }
    }
}
```

# Custom validation

The following code shows how to add a model error after examining the model:

C# Copy

```
if (Contact.Name == Contact.ShortName)
{
    ModelState.AddModelError("Contact.ShortName",
```

```
                                "Short name can't be the same as Name.");
}
```

The following code implements the validation test in a controller:

C# Copy

```
if (contact.Name == contact.ShortName)
{
    ModelState.AddModelError(nameof(contact.ShortName),
                                "Short name can't be the same as Name.");
}
```

The following code verifies the phone number and email are unique:

C# Copy

```
public async Task<IActionResult> OnPostAsync()
{
    // Attach Validation Error Message to the Model on validation failure.

    if (Contact.Name == Contact.ShortName)
    {
        ModelState.AddModelError("Contact.ShortName",
                                 "Short name can't be the same as Name.");
    }

    if (_context.Contact.Any(i => i.PhoneNumber == Contact.PhoneNumber))
    {
        ModelState.AddModelError("Contact.PhoneNumber",
                                  "The Phone number is already in use.");
    }
    if (_context.Contact.Any(i => i.Email == Contact.Email))
    {
        ModelState.AddModelError("Contact.Email", "The Email is already in use.");
    }

    if (!ModelState.IsValid || _context.Contact == null || Contact == null)
    {
        // if model is invalid, return the page with the model state errors.
        return Page();
    }
    _context.Contact.Add(Contact);
    await _context.SaveChangesAsync();

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

The following code implements the validation test in a controller:

C# Copy

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create([Bind("Id,Name,ShortName,Email,PhoneNumber")]
{
    // Attach Validation Error Message to the Model on validation failure.
    if (contact.Name == contact.ShortName)
    {
        ModelState.AddModelError(nameof(contact.ShortName),
                                 "Short name can't be the same as Name.");
    }

    if (_context.Contact.Any(i => i.PhoneNumber == contact.PhoneNumber))
    {
        ModelState.AddModelError(nameof(contact.PhoneNumber),
                                  "The Phone number is already in use.");
    }
    if (_context.Contact.Any(i => i.Email == contact.Email))
    {
        ModelState.AddModelError(nameof(contact.Email), "The Email is already in use.
    }

    if (ModelState.IsValid)
    {
        _context.Add(contact);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    return View(contact);
}
```

Checking for a unique phone number or email is typically also done with remote validation.

# ValidationResult

Consider the following custom `ValidateNameAttribute`:

C# Copy

```
public class ValidateNameAttribute : ValidationAttribute
{
    public ValidateNameAttribute()
    {
        const string defaultErrorMessage = "Error with Name";
        ErrorMessage ??= defaultErrorMessage;
    }
```

```
    protected override ValidationResult? IsValid(object? value,
                                         ValidationContext validationContext)
    {
        if (value == null || string.IsNullOrWhiteSpace(value.ToString()))
        {
            return new ValidationResult("Name is required.");
        }

        if (value.ToString()!.ToLower().Contains("zz"))
        {

            return new ValidationResult(
                        FormatErrorMessage(validationContext.DisplayName));
        }

        return ValidationResult.Success;
    }
}
```

In the following code, the custom `[ValidateName]` attribute is applied:

C# Copy

```
public class Contact
{
    public Guid Id { get; set; }

    [ValidateName(ErrorMessage = "Name must not contain `zz`")]
    public string? Name { get; set; }
    public string? Email { get; set; }
    public string? PhoneNumber { get; set; }
}
```

When the model contains `zz`, a new ValidationResult is returned.

# Top-level node validation

Top-level nodes include:

- Action parameters
- Controller properties
- Page handler parameters
- Page model properties

Model-bound top-level nodes are validated in addition to validating model properties. In the following example from the sample app, the `VerifyPhone` method uses the

RegularExpressionAttribute to validate the `phone` action parameter:

```
C#
[AcceptVerbs("GET", "POST")]
public IActionResult VerifyPhone(
    [RegularExpression(@"^\d{3}-\d{3}-\d{4}$")] string phone)
{
    if (!ModelState.IsValid)
    {
        return Json($"Phone {phone} has an invalid format. Format: ###-###-####");
    }

    return Json(true);
}
```

Top-level nodes can use BindRequiredAttribute with validation attributes. In the following example from the sample app, the `CheckAge` method specifies that the `age` parameter must be bound from the query string when the form is submitted:

```
C#
[HttpPost]
public IActionResult CheckAge([BindRequired, FromQuery] int age)
{
```

In the Check Age page (`CheckAge.cshtml`), there are two forms. The first form submits an `Age` value of `99` as a query string parameter: `https://localhost:5001/Users/CheckAge?Age=99`.

When a properly formatted `age` parameter from the query string is submitted, the form validates.

The second form on the Check Age page submits the `Age` value in the body of the request, and validation fails. Binding fails because the `age` parameter must come from a query string.

# Maximum errors

Validation stops when the maximum number of errors is reached (200 by default). You can configure this number with the following code in `Program.cs`:

```
C#
builder.Services.AddRazorPages()
    .AddMvcOptions(options =>
```

```
    {
        options.MaxModelValidationErrors = 50;
        options.ModelBindingMessageProvider.SetValueMustNotBeNullAccessor(
            _ => "The field is required.");
    });

builder.Services.AddSingleton
    <IValidationAttributeAdapterProvider, CustomValidationAttributeAdapterProvider>()
```

# Maximum recursion

ValidationVisitor traverses the object graph of the model being validated. For models that are deep or are infinitely recursive, validation may result in stack overflow. MvcOptions.MaxValidationDepth provides a way to stop validation early if the visitor recursion exceeds a configured depth. The default value of `MvcOptions.MaxValidationDepth` is 32.

# Automatic short-circuit

Validation is automatically short-circuited (skipped) if the model graph doesn't require validation. Objects that the runtime skips validation for include collections of primitives (such as `byte[]`, `string[]`, `Dictionary<string, string>`) and complex object graphs that don't have any validators.

# Client-side validation

Client-side validation prevents submission until the form is valid. The Submit button runs JavaScript that either submits the form or displays error messages.

Client-side validation avoids an unnecessary round trip to the server when there are input errors on a form. The following script references in `_Layout.cshtml` and `_ValidationScriptsPartial.cshtml` support client-side validation:

CSHTML Copy

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.6.0/jquery.js"></script>
```

CSHTML Copy

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery-validate/1.19.3/jquery.val
```

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery-validation-unobtrusive/3.2
```

The jQuery Unobtrusive Validation script is a custom Microsoft front-end library that builds on the popular jQuery Validation plugin. Without jQuery Unobtrusive Validation, you would have to code the same validation logic in two places: once in the server-side validation attributes on model properties, and then again in client-side scripts. Instead, Tag Helpers and HTML helpers use the validation attributes and type metadata from model properties to render HTML 5 `data-` attributes for the form elements that need validation. jQuery Unobtrusive Validation parses the `data-` attributes and passes the logic to jQuery Validation, effectively "copying" the server-side validation logic to the client. You can display validation errors on the client using tag helpers as shown here:

CSHTML Copy

```
<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>
```

The preceding tag helpers render the following HTML:

HTML Copy

```
<div class="form-group">
    <label class="control-label" for="Movie_ReleaseDate">Release Date</label>
    <input class="form-control" type="date" data-val="true"
        data-val-required="The Release Date field is required."
        id="Movie_ReleaseDate" name="Movie.ReleaseDate" value="">
    <span class="text-danger field-validation-valid"
        data-valmsg-for="Movie.ReleaseDate" data-valmsg-replace="true"></span>
</div>
```

Notice that the `data-` attributes in the HTML output correspond to the validation attributes for the `Movie.ReleaseDate` property. The `data-val-required` attribute contains an error message to display if the user doesn't fill in the release date field. jQuery Unobtrusive Validation passes this value to the jQuery Validation required() method, which then displays that message in the accompanying **** element.

Data type validation is based on the .NET type of a property, unless that is overridden by a [DataType] attribute. Browsers have their own default error messages, but the jQuery Validation Unobtrusive Validation package can override those messages. `[DataType]` attributes and

subclasses such as [EmailAddress] let you specify the error message.

# Unobtrusive validation

For information on unobtrusive validation, see this GitHub issue.

## Add Validation to Dynamic Forms

jQuery Unobtrusive Validation passes validation logic and parameters to jQuery Validation when the page first loads. Therefore, validation doesn't work automatically on dynamically generated forms. To enable validation, tell jQuery Unobtrusive Validation to parse the dynamic form immediately after you create it. For example, the following code sets up client-side validation on a form added via AJAX.

JavaScript Copy

```javascript
$.get({
    url: "https://url/that/returns/a/form",
    dataType: "html",
    error: function(jqXHR, textStatus, errorThrown) {
        alert(textStatus + ": Couldn't add form. " + errorThrown);
    },
    success: function(newFormHTML) {
        var container = document.getElementById("form-container");
        container.insertAdjacentHTML("beforeend", newFormHTML);
        var forms = container.getElementsByTagName("form");
        var newForm = forms[forms.length - 1];
        $.validator.unobtrusive.parse(newForm);
    }
})
```

The `$.validator.unobtrusive.parse()` method accepts a jQuery selector for its one argument. This method tells jQuery Unobtrusive Validation to parse the `data-` attributes of forms within that selector. The values of those attributes are then passed to the jQuery Validation plugin.

## Add Validation to Dynamic Controls

The `$.validator.unobtrusive.parse()` method works on an entire form, not on individual dynamically generated controls, such as `<input>` and `<select/>`. To reparse the form, remove the validation data that was added when the form was parsed earlier, as shown in the following example:

JavaScript Copy

```
$.get({
    url: "https://url/that/returns/a/control",
    dataType: "html",
    error: function(jqXHR, textStatus, errorThrown) {
        alert(textStatus + ": Couldn't add control. " + errorThrown);
    },
    success: function(newInputHTML) {
        var form = document.getElementById("my-form");
        form.insertAdjacentHTML("beforeend", newInputHTML);
        $(form).removeData("validator")    // Added by jQuery Validation
               .removeData("unobtrusiveValidation");   // Added by jQuery Unobtrusive
        $.validator.unobtrusive.parse(form);
    }
})
```

# Custom client-side validation

Custom client-side validation is done by generating `data-` HTML attributes that work with a custom jQuery Validation adapter. The following sample adapter code was written for the `[ClassicMovie]` and `[ClassicMovieWithClientValidator]` attributes that were introduced earlier in this article:

JavaScript Copy

```
$.validator.addMethod('classicmovie', function (value, element, params) {
    var genre = $(params[0]).val(), year = params[1], date = new Date(value);

    // The Classic genre has a value of '0'.
    if (genre && genre.length > 0 && genre[0] === '0') {
        // The release date for a Classic is valid if it's no greater than the given
        return date.getUTCFullYear() <= year;
    }

    return true;
});

$.validator.unobtrusive.adapters.add('classicmovie', ['year'], function (options) {
    var element = $(options.form).find('select#Movie_Genre')[0];

    options.rules['classicmovie'] = [element, parseInt(options.params['year'])];
    options.messages['classicmovie'] = options.message;
});
```

For information about how to write adapters, see the jQuery Validation documentation.

The use of an adapter for a given field is triggered by `data-` attributes that:

- Flag the field as being subject to validation (`data-val="true"`).
- Identify a validation rule name and error message text (for example, `data-val-rulename="Error message."`).
- Provide any additional parameters the validator needs (for example, `data-val-rulename-param1="value"`).

The following example shows the `data-` attributes for the sample app's `ClassicMovie` attribute:

HTML Copy

```
<input class="form-control" type="date"
    data-val="true"
    data-val-classicmovie="Classic movies must have a release year no later than 1960
    data-val-classicmovie-year="1960"
    data-val-required="The Release Date field is required."
    id="Movie_ReleaseDate" name="Movie.ReleaseDate" value="">
```

As noted earlier, Tag Helpers and HTML helpers use information from validation attributes to render `data-` attributes. There are two options for writing code that results in the creation of custom `data-` HTML attributes:

- Create a class that derives from AttributeAdapterBase<TAttribute> and a class that implements IValidationAttributeAdapterProvider, and register your attribute and its adapter in DI. This method follows the single responsibility principle in that server-related and client-related validation code is in separate classes. The adapter also has the advantage that since it's registered in DI, other services in DI are available to it if needed.
- Implement IClientModelValidator in your ValidationAttribute class. This method might be appropriate if the attribute doesn't do any server-side validation and doesn't need any services from DI.

## AttributeAdapter for client-side validation

This method of rendering `data-` attributes in HTML is used by the `ClassicMovie` attribute in the sample app. To add client validation by using this method:

1. Create an attribute adapter class for the custom validation attribute. Derive the class from AttributeAdapterBase<TAttribute>. Create an `AddValidation` method that adds `data-` attributes to the rendered output, as shown in this example:

C# Copy

```
public class ClassicMovieAttributeAdapter : AttributeAdapterBase<ClassicMovieAtt
{
    public ClassicMovieAttributeAdapter(
        ClassicMovieAttribute attribute, IStringLocalizer? stringLocalizer)
        : base(attribute, stringLocalizer)
    {

    }

    public override void AddValidation(ClientModelValidationContext context)
    {
        MergeAttribute(context.Attributes, "data-val", "true");
        MergeAttribute(context.Attributes, "data-val-classicmovie", GetErrorMess

        var year = Attribute.Year.ToString(CultureInfo.InvariantCulture);
        MergeAttribute(context.Attributes, "data-val-classicmovie-year", year);
    }

    public override string GetErrorMessage(ModelValidationContextBase validation
        => Attribute.GetErrorMessage();
}
```

2. Create an adapter provider class that implements IValidationAttributeAdapterProvider. In the GetAttributeAdapter method pass in the custom attribute to the adapter's constructor, as shown in this example:

C# Copy

```
public class CustomValidationAttributeAdapterProvider : IValidationAttributeAdap
{
    private readonly IValidationAttributeAdapterProvider baseProvider =
        new ValidationAttributeAdapterProvider();

    public IAttributeAdapter? GetAttributeAdapter(
        ValidationAttribute attribute, IStringLocalizer? stringLocalizer)
    {
        if (attribute is ClassicMovieAttribute classicMovieAttribute)
        {
            return new ClassicMovieAttributeAdapter(classicMovieAttribute, strin
        }

        return baseProvider.GetAttributeAdapter(attribute, stringLocalizer);
    }
}
```

3. Register the adapter provider for DI in `Program.cs`:

```
C#
builder.Services.AddRazorPages()
    .AddMvcOptions(options =>
    {
        options.MaxModelValidationErrors = 50;
        options.ModelBindingMessageProvider.SetValueMustNotBeNullAccessor(
            _ => "The field is required.");
    });

builder.Services.AddSingleton
    <IValidationAttributeAdapterProvider, CustomValidationAttributeAdapterProvid
```

## IClientModelValidator for client-side validation

This method of rendering `data-` attributes in HTML is used by the `ClassicMovieWithClientValidator` attribute in the sample app. To add client validation by using this method:

- In the custom validation attribute, implement the IClientModelValidator interface and create an AddValidation method. In the `AddValidation` method, add `data-` attributes for validation, as shown in the following example:

```
C#
public class ClassicMovieWithClientValidatorAttribute :
    ValidationAttribute, IClientModelValidator
{
    public ClassicMovieWithClientValidatorAttribute(int year)
        => Year = year;

    public int Year { get; }

    public void AddValidation(ClientModelValidationContext context)
    {
        MergeAttribute(context.Attributes, "data-val", "true");
        MergeAttribute(context.Attributes, "data-val-classicmovie", GetErrorMess

        var year = Year.ToString(CultureInfo.InvariantCulture);
        MergeAttribute(context.Attributes, "data-val-classicmovie-year", year);
    }

    public string GetErrorMessage() =>
        $"Classic movies must have a release year no later than {Year}.";
```

```
    protected override ValidationResult? IsValid(
        object? value, ValidationContext validationContext)
    {
        var movie = (Movie)validationContext.ObjectInstance;
        var releaseYear = ((DateTime)value!).Year;

        if (movie.Genre == Genre.Classic && releaseYear > Year)
        {
            return new ValidationResult(GetErrorMessage());
        }

        return ValidationResult.Success;
    }

    private static bool MergeAttribute(IDictionary<string, string> attributes, s
    {
        if (attributes.ContainsKey(key))
        {
            return false;
        }

        attributes.Add(key, value);
        return true;
    }
}
```

# Disable client-side validation

The following code disables client validation in Razor Pages:

C# Copy

```
builder.Services.AddRazorPages()
    .AddViewOptions(options =>
    {
        options.HtmlHelperOptions.ClientValidationEnabled = false;
    });
```

Other options to disable client-side validation:

- Comment out the reference to `_ValidationScriptsPartial` in all the `.cshtml` files.
- Remove the contents of the *Pages\Shared_ValidationScriptsPartial.cshtml* file.

The preceding approach won't prevent client-side validation of ASP.NET Core Identity Razor class library. For more information, see Scaffold Identity in ASP.NET Core projects.

# Problem details

Problem Details are not the only response format to describe an HTTP API error, however, they are commonly used to report errors for HTTP APIs.

The problem details service implements the IProblemDetailsService interface, which supports creating problem details in ASP.NET Core. The AddProblemDetails(IServiceCollection) extension method on IServiceCollection registers the default `IProblemDetailsService` implementation.

In ASP.NET Core apps, the following middleware generates problem details HTTP responses when `AddProblemDetails` is called, except when the Accept request HTTP header doesn't include one of the content types supported by the registered IProblemDetailsWriter (default: `application/json`):

- ExceptionHandlerMiddleware: Generates a problem details response when a custom handler is not defined.
- StatusCodePagesMiddleware: Generates a problem details response by default.
- DeveloperExceptionPageMiddleware: Generates a problem details response in development when the `Accept` request HTTP header doesn't include `text/html`.

# Additional resources

- System.ComponentModel.DataAnnotations
- Model Binding

# Additional resources

## Documentation

**Validation with the Data Annotation Validators (C#)**

Take advantage of the Data Annotation Model Binder to perform validation within an ASP.NET MVC application. Learn how to use the different types of validator... (C#)

**Model Binding in ASP.NET Core**

Learn how model binding in ASP.NET Core works and how to customize its behavior.

**Model Validation in ASP.NET Web API - ASP.NET 4.x**

Overview of model validation in ASP.NET Web API for ASP.NET 4.x.

Show 2 more

## Training

Module
**Improve how forms and validation work in Blazor web apps - Training**

Learn how to use DOM events, forms, and validation in a Blazor app

## Events

**Be one of the first to start using SQL database in Fabric**

**Dec 4, 8 AM - Dec 13, 8 AM**

Get all your questions answered in live sessions starting December 3rd.
Register now