

Add validation to an ASP.NET Core MVC app

📅 04/13/2017 ⌚ 10 minutes to read Contributors 

In this article

[Keeping things DRY](#)

[Adding validation rules to the movie model](#)

[Validation Error UI in MVC](#)

[How validation works](#)

[Using DataType Attributes](#)

[Additional resources](#)

By [Rick Anderson](#)

In this section you'll add validation logic to the `Movie` model, and you'll ensure that the validation rules are enforced any time a user creates or edits a movie.

Keeping things DRY

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

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

Adding validation rules to the movie model

Open the *Movie.cs* file. DataAnnotations provides a built-in set of validation attributes that you apply declaratively to any class or property. (It also contains formatting attributes like `DataType` that help with formatting and don't provide any validation.)

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

C# Copy

```
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)]
    public decimal Price { get; set; }

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

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

The validation attributes specify behavior that you want to enforce on the model properties they're applied to. The `Required` and `MinimumLength` attributes indicates 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 code above, `Genre` and `Rating` must use only letters (First letter uppercase, white space, numbers and special characters are not allowed). The `Range` attribute constrains a value to within a specified range. The `StringLength` attribute lets you set the maximum length of a string property, and optionally its minimum length. Value types (such as `decimal`, `int`, `float`, `DateTime`) are inherently required and don't need the `[Required]` attribute.

Having validation rules automatically enforced by ASP.NET Core helps make your app more robust. It also ensures that you can't forget to validate something and inadvertently let bad data into the database.

Validation Error UI in MVC

Run the app and navigate to the Movies controller.

Tap the **Create New** link to add a new movie. Fill out the form with some invalid values. As soon as jQuery client side validation detects the error, it displays an error message.

The screenshot shows a web browser window with the title 'Create - Movie App'. The address bar shows 'localhost:5000/Movies/Cre'. The page has a dark header with 'MvcMovie' and a hamburger menu icon. The main content area is titled 'Create Movie'. It contains five form fields, each with a validation error message in red text:

- Title:** The field Title must be a string with a minimum length of 3 and a maximum length of 60.
- Release Date:** The field contains the placeholder 'mm/dd/yyyy'.
- Genre:** The Genre field is required.
- Price:** The field contains the character 'z'. The field Price must be a number.
- Rating:** The field is empty. The field Rating must match the regular expression `^[A-Z]+[a-zA-Z]*$`.

At the bottom of the form, there is a 'Create' button and a 'Back to List' link. The footer shows '© 2017 - MvcMovie'.

ⓘ Note

You may not be able to enter decimal commas in the field. To support [jQuery validation](#) for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize your app. This [GitHub issue 4076](#) for instructions on adding decimal comma.

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

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

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

How validation works

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

```
C# Copy

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

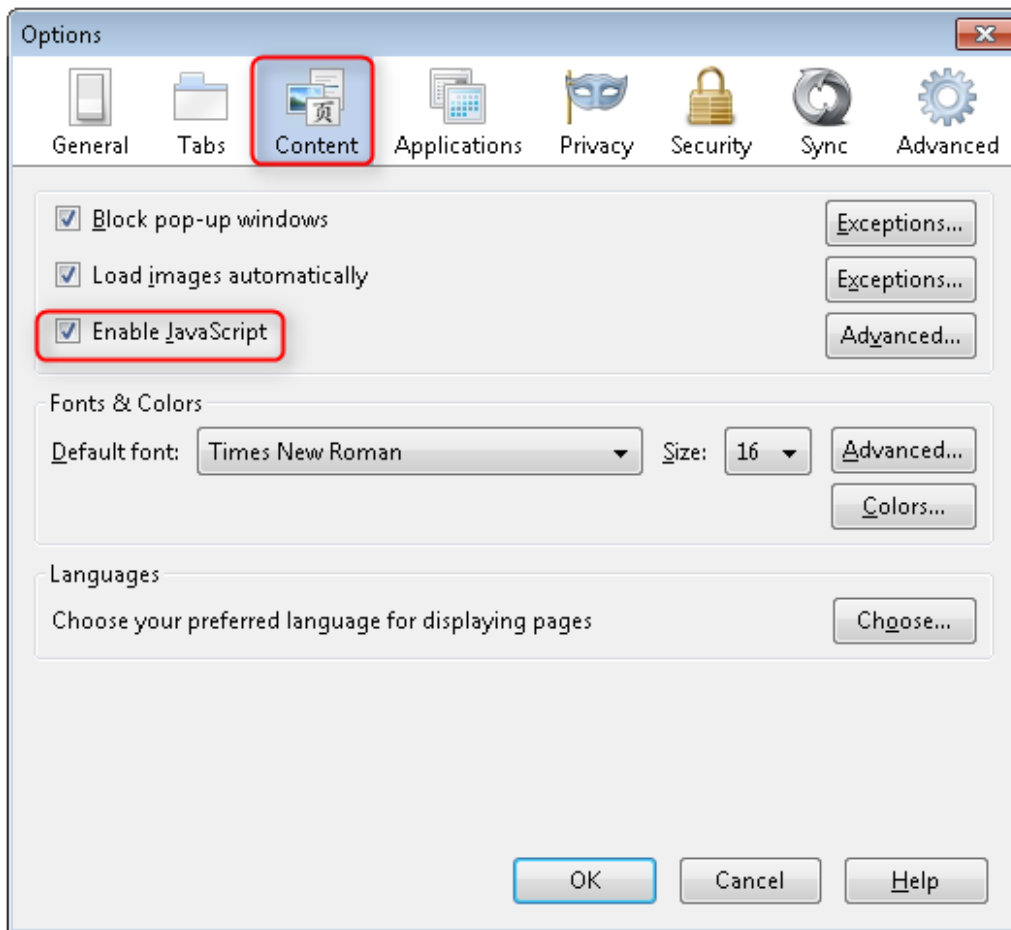
// POST: Movies/Create
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(
    [Bind("ID,Title,ReleaseDate,Genre,Price, Rating")] Movie movie)
{
    if (ModelState.IsValid)
    {
        _context.Add(movie);
        await _context.SaveChangesAsync();
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

The first (HTTP GET) `Create` action method displays the initial Create form. The second (`[HttpPost]`) version handles the form post. The second `Create` method (The `[HttpPost]` version) calls `ModelState.IsValid` to check whether the movie has any validation errors. Calling this method evaluates

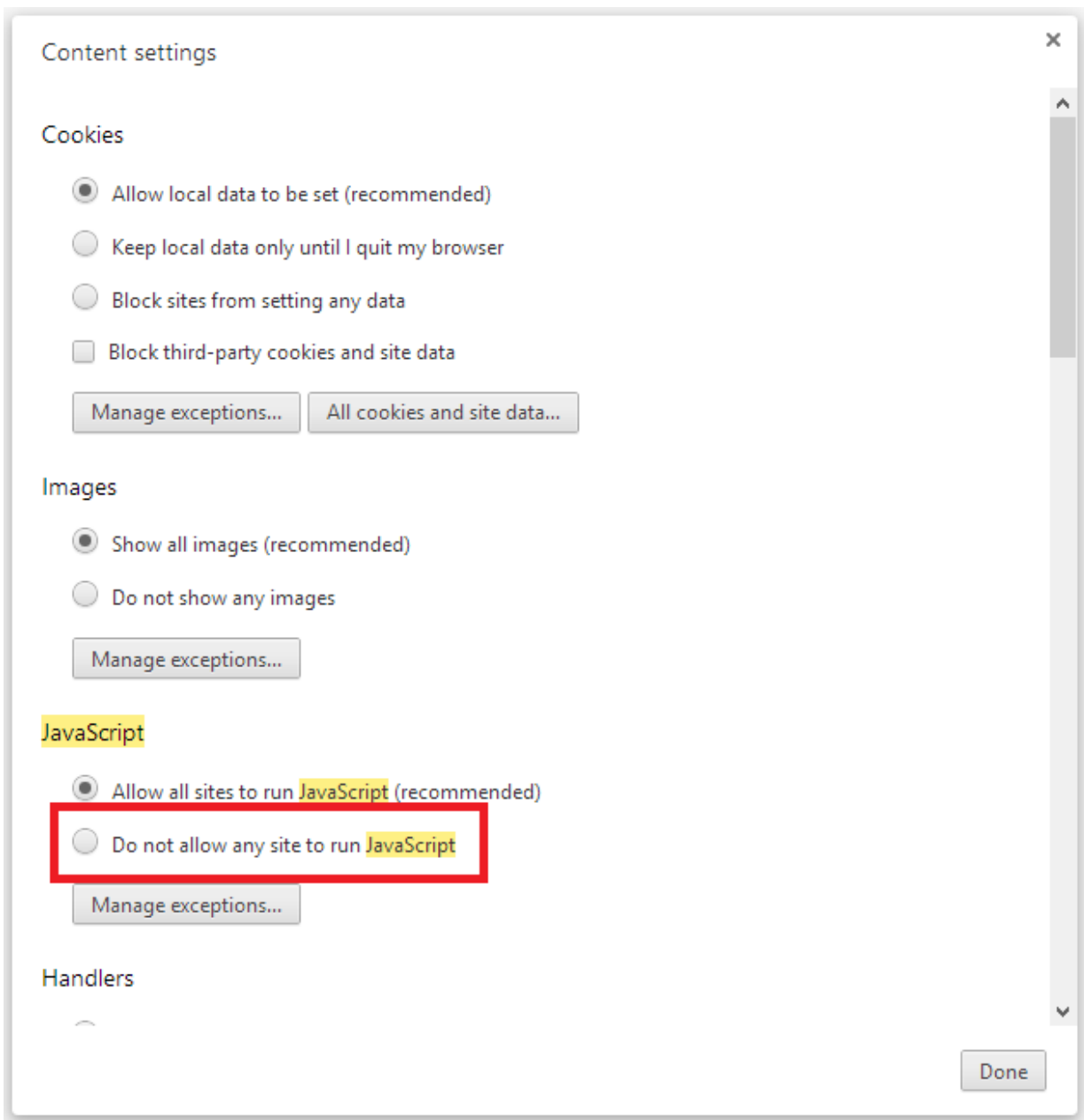
any validation attributes that have been applied to the object. If the object has validation errors, the `Create` method re-displays the form. If there are no errors, the method saves the new movie in the database. In our movie example, the form isn't posted to the server when there are validation errors detected on the client side; the second `Create` method is never called when there are client side validation errors. If you disable JavaScript in your browser, client validation is disabled and you can test the HTTP POST `Create` method `ModelState.IsValid` detecting any validation errors.

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

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



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



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

```

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

```

Below is portion of the *Create.cshtml* view template that you scaffolded earlier in the tutorial. It's used by the action methods shown above both to display the initial form and to redisplay it in the event of an error.

HTML	Copy
<pre> <form asp-action="Create"> <div class="form-horizontal"> <h4>Movie</h4> <hr /> <div asp-validation-summary="ModelOnly" class="text-danger"></div> <div class="form-group"> <label asp-for="Title" class="col-md-2 control-label"></label> <div class="col-md-10"> <input asp-for="Title" class="form-control" /> </div> </div> @*Markup removed for brevity.*@ </div> </form> </pre>	

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

What's really nice about this approach is that neither the controller nor the `Create` view template knows anything about the actual validation rules being enforced or about the specific error messages displayed. The validation rules and the error strings are specified only in the `Movie` class. These same validation rules

are automatically applied to the `Edit` view and any other views templates you might create that edit your model.

When you need to change validation logic, you can do so in exactly one place by adding validation attributes to the model (in this example, the `Movie` class). You won't have to worry about different parts of the application being inconsistent with how the rules are enforced — all validation logic will be defined in one place and used everywhere. This keeps the code very clean, and makes it easy to maintain and evolve. And it means that you'll be fully honoring the DRY principle.

Using DataType Attributes

Open the `Movie.cs` file and examine the `Movie` class. The `System.ComponentModel.DataAnnotations` namespace provides formatting attributes in addition to the built-in set of validation attributes. We've already applied a `DataType` enumeration value to the release date and to the price fields. The following code shows the `ReleaseDate` and `Price` properties with the appropriate `DataType` attribute.

C#

 Copy

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

[Range(1, 100)]
[DataType(DataType.Currency)]
public decimal Price { get; set; }
```

The `DataType` attributes only provide hints for the view engine to format the data (and supplies elements/attributes such as `<a>` for URL's and `` for email. You can use the `RegularExpression` attribute to validate the format of the data. The `DataType` attribute is used to specify a data type that's more specific than the database intrinsic type, they're not validation attributes. In this case we only want to keep track of the date, not the time. The `DataType` Enumeration provides for many data types, such as `Date`, `Time`, `PhoneNumber`, `Currency`, `EmailAddress` and more. The `DataType` attribute can also enable the application to automatically provide type-specific features. For example, a `mailto:` link can be created for `DataType.EmailAddress`, and a date selector can be provided for `DataType.Date` in browsers that support HTML5. The `DataType` attributes emit HTML 5 `data-` (pronounced data dash) attributes that HTML 5 browsers can understand. The `DataType` attributes do **not** provide any validation.

`DataType.Date` doesn't specify the format of the date that's displayed. By default, the data field is displayed according to the default formats based on the server's `CultureInfo`.

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

C#	
<pre>[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)] public DateTime ReleaseDate { get; set; }</pre>	


The `ApplyFormatInEditMode` setting specifies that the formatting should also be applied when the value is displayed in a text box for editing. (You might not want that for some fields — for example, for currency values, you probably don't want the currency symbol in the text box for editing.)

You can use the `DisplayFormat` attribute by itself, but it's generally a good idea to use the `DataType` attribute. The `DataType` attribute conveys the semantics of the data as opposed to how to render it on a screen, and provides the following benefits that you don't get with `DisplayFormat`:

- The browser can enable HTML5 features (for example to show a calendar control, the locale-appropriate currency symbol, email links, etc.)
- By default, the browser will render data using the correct format based on your locale.
- The `DataType` attribute can enable MVC to choose the right field template to render the data (the `DisplayFormat` if used by itself uses the string template).


Note

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

C#	
<pre>[Range(typeof(DateTime), "1/1/1966", "1/1/2020")]</pre>	

You will need to disable jQuery date validation to use the `Range` attribute with `DateTime`. It's generally not a good practice to compile hard dates in your models, so using the `Range` attribute and `DateTime` is discouraged.

The following code shows combining attributes on one line:

C#	
<pre>public class Movie { public int ID { get; set; }</pre>	

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

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

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

[Range(1, 100), DataType(DataType.Currency)]
public decimal Price { get; set; }

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

In the next part of the series, we'll review the application and make some improvements to the automatically generated `Details` and `Delete` methods.

Additional resources

- [Working with Forms](#)
- [Globalization and localization](#)
- [Introduction to Tag Helpers](#)
- [Author Tag Helpers](#)

[Previous - Add a field](#)

[Next - Examine the Details and Delete methods](#)