**Part 9, add validation to an ASP.NET Core MVC app**

* Article
* 07/26/2024
* 12 contributors

Feedback

**In this article**

1. [Keeping things DRY](https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/validation?view=aspnetcore-8.0#keeping-things-dry)
2. [Delete the previously edited data](https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/validation?view=aspnetcore-8.0#delete-the-previously-edited-data)
3. [Add validation rules to the movie model](https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/validation?view=aspnetcore-8.0#add-validation-rules-to-the-movie-model)
4. [Validation Error UI](https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/validation?view=aspnetcore-8.0#validation-error-ui)

Show 3 more

By [Rick Anderson](https://twitter.com/RickAndMSFT)

In this section:

* Validation logic is added to the Movie model.
* You 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](https://wikipedia.org/wiki/Don%27t_repeat_yourself) ("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.

**Delete the previously edited data**

In the next step, validation rules are added that don't allow null values. Run the app, navigate to /Movies/Index, delete all listed movies, and stop the app. The app will use the seed data the next time it is run.

**Add validation rules to the movie model**

The 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.

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

C#Copy

using System;

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

namespace MvcMovie.Models;

public class Movie

{

public int Id { get; set; }

[StringLength(60, MinimumLength = 3)]

[Required]

public string? Title { get; set; }

[Display(Name = "Release Date")]

[DataType(DataType.Date)]

public DateTime ReleaseDate { get; set; }

[Range(1, 100)]

[DataType(DataType.Currency)]

[Column(TypeName = "decimal(18, 2)")]

public decimal Price { get; set; }

[RegularExpression(@"^[A-Z]+[a-zA-Z\s]\*$")]

[Required]

[StringLength(30)]

public string? Genre { get; set; }

[RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]\*$")]

[StringLength(5)]

[Required]

public string? Rating { get; set; }

}

The validation attributes specify behavior that you want to enforce on the model properties they're applied to:

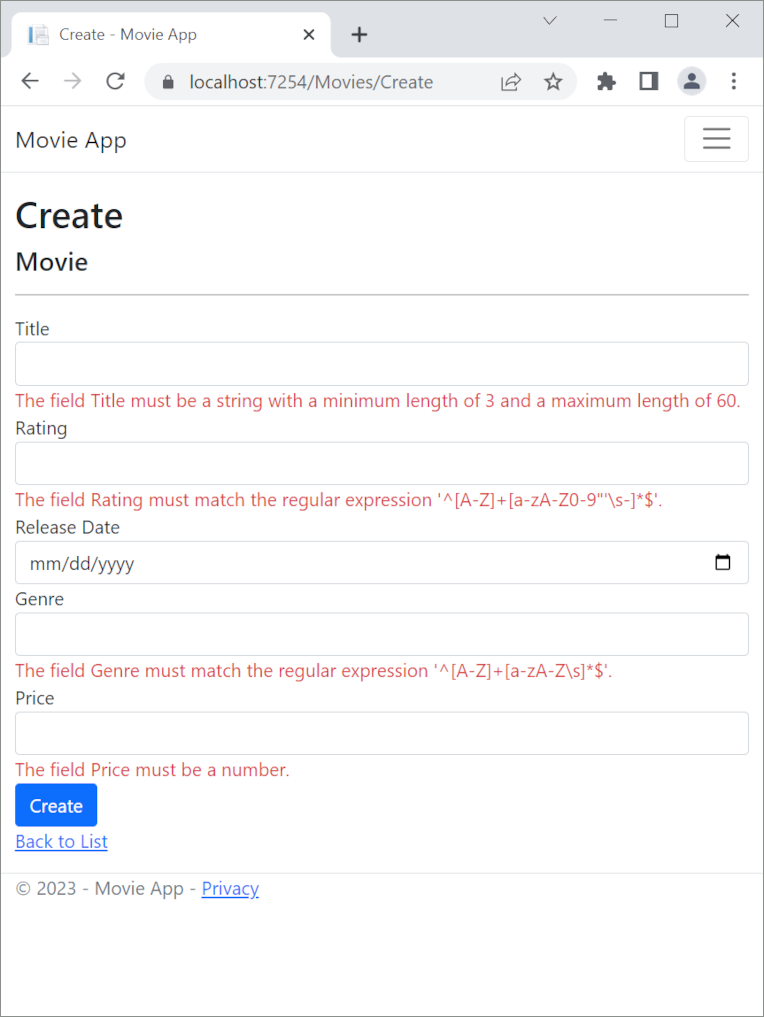
* The Required and MinimumLength attributes indicate that a property must have a value; but nothing prevents a user from entering white space to satisfy this validation.
* The RegularExpression attribute is used to limit what characters can be input. In the preceding code, "Genre":
  + Must only use letters.
  + The first letter is required to be uppercase. White spaces are allowed while 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 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**

Run the app and navigate to the Movies controller.

Select 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.



**Note**

You may not be able to enter decimal commas in decimal fields. To support [**jQuery validation**](https://jqueryvalidation.org/) for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize your app. [**See this GitHub comment 4076**](https://github.com/dotnet/AspNetCore.Docs/issues/4076#issuecomment-1153254062) 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](https://www.telerik.com/fiddler) , or the [F12 Developer tools](https://learn.microsoft.com/en-us/microsoft-edge/devtools-guide).

**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

// To protect from overposting attacks, enable the specific properties you want to bind to.

// For more details, see http://go.microsoft.com/fwlink/?LinkId=317598.

[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(nameof(Index));

}

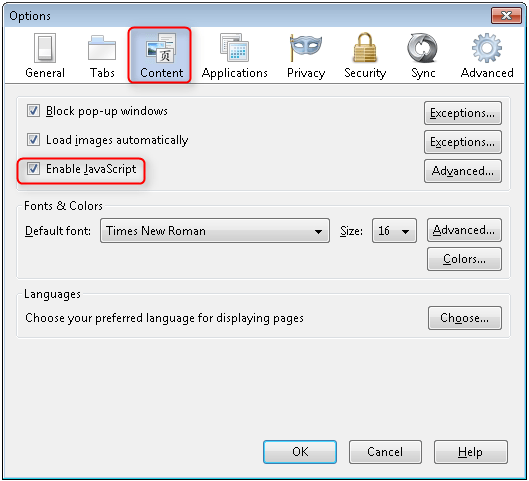
return View(movie);

}

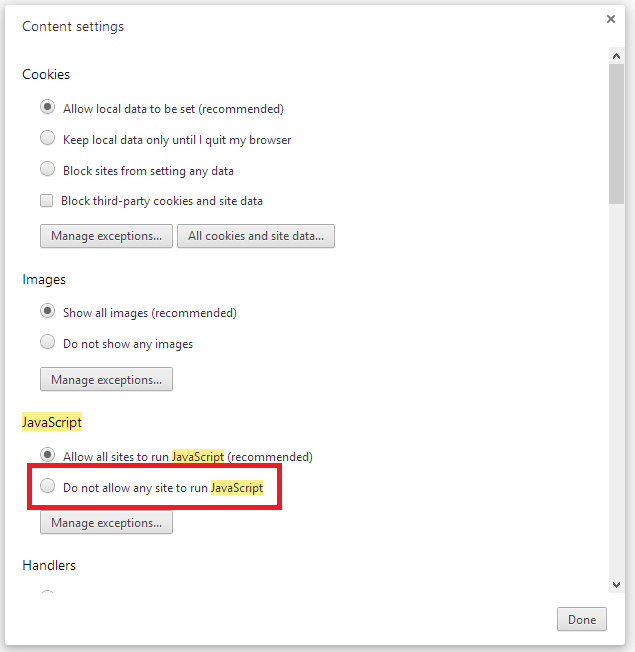
The first (HTTP GET) Create action method displays the initial Create form. The second ([HttpPost]) version handles the form post. The second Create method (The [HttpPost] version) calls ModelState.IsValid to check whether the movie has any validation errors. Calling this method evaluates any validation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

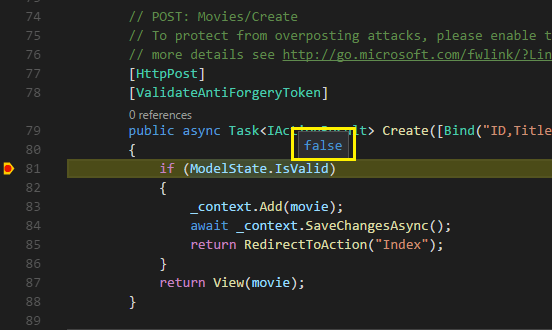
The following image shows how to disable JavaScript in the Firefox browser.



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



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



A portion of the Create.cshtml view template is shown in the following markup:

HTMLCopy

<h4>Movie</h4>

<hr />

<div class="row">

<div class="col-md-4">

<form asp-action="Create">

<div asp-validation-summary="ModelOnly" class="text-danger"></div>

<div class="form-group">

<label asp-for="Title" class="control-label"></label>

<input asp-for="Title" class="form-control" />

<span asp-validation-for="Title" class="text-danger"></span>

</div>

@\*Markup removed for brevity.\*@

The preceding markup is used by the action methods to display the initial form and to redisplay it in the event of an error.

The [Input Tag Helper](https://learn.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-8.0) uses the [DataAnnotations](https://learn.microsoft.com/en-us/aspnet/mvc/overview/older-versions/mvc-music-store/mvc-music-store-part-6) attributes and produces HTML attributes needed for jQuery Validation on the client side. The [Validation Tag Helper](https://learn.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-8.0#the-validation-tag-helpers) displays validation errors. See [Validation](https://learn.microsoft.com/en-us/aspnet/core/mvc/models/validation?view=aspnetcore-8.0) 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)]

[Column(TypeName = "decimal(18, 2)")]

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 <a href="mailto:EmailAddress.com"> 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#Copy

[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]

public DateTime ReleaseDate { get; set; }

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:

[Range(typeof(DateTime), "1/1/1966", "1/1/2020")]

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#Copy

using System;

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

namespace MvcMovie.Models;

public class Movie

{

public int Id { get; set; }

[StringLength(60, MinimumLength = 3)]

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)]

[Column(TypeName = "decimal(18, 2)")]

public decimal Price { get; set; }

[RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]\*$"), StringLength(5)]

public string Rating { get; set; }

}

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