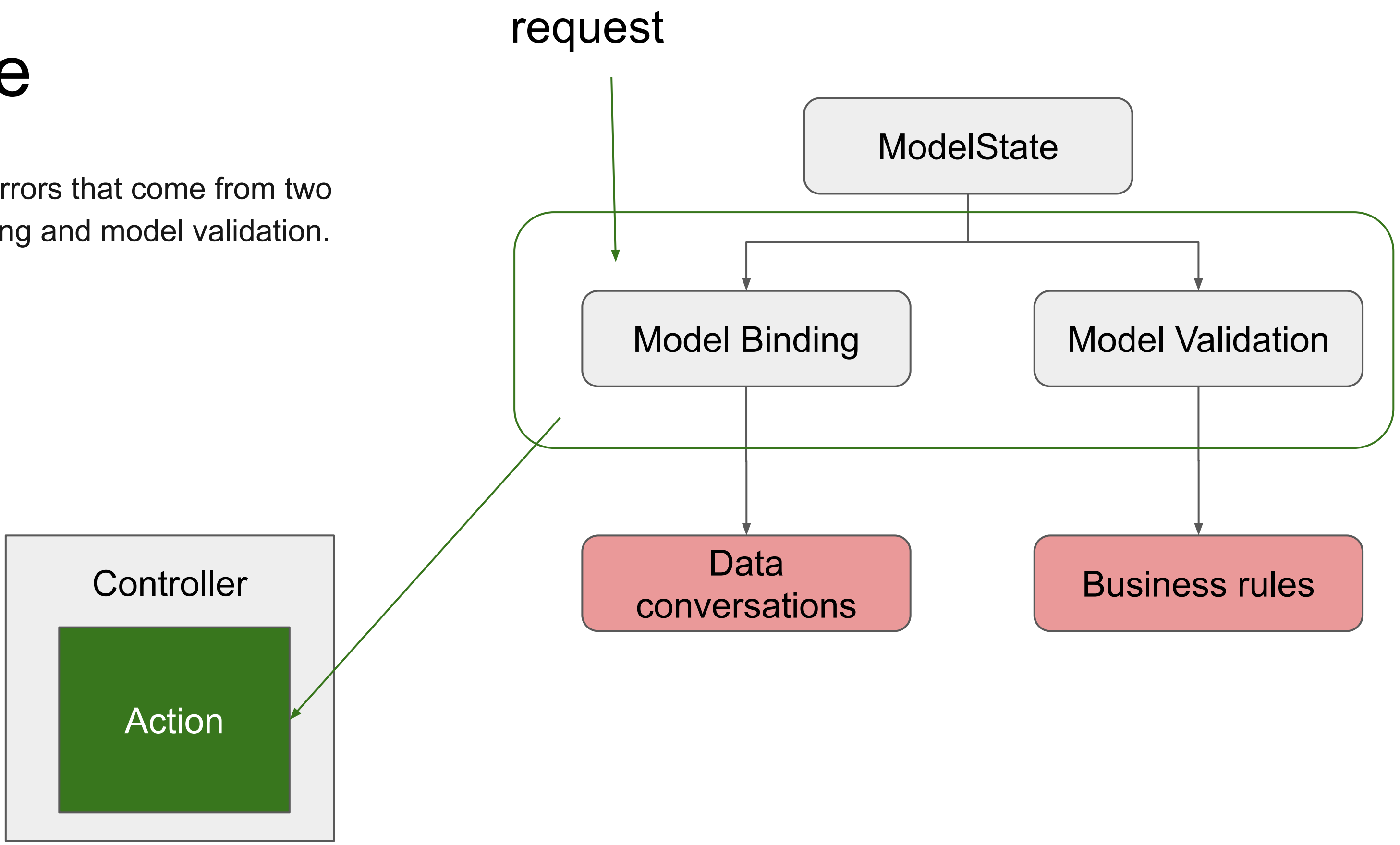
Validations

Asp.net core

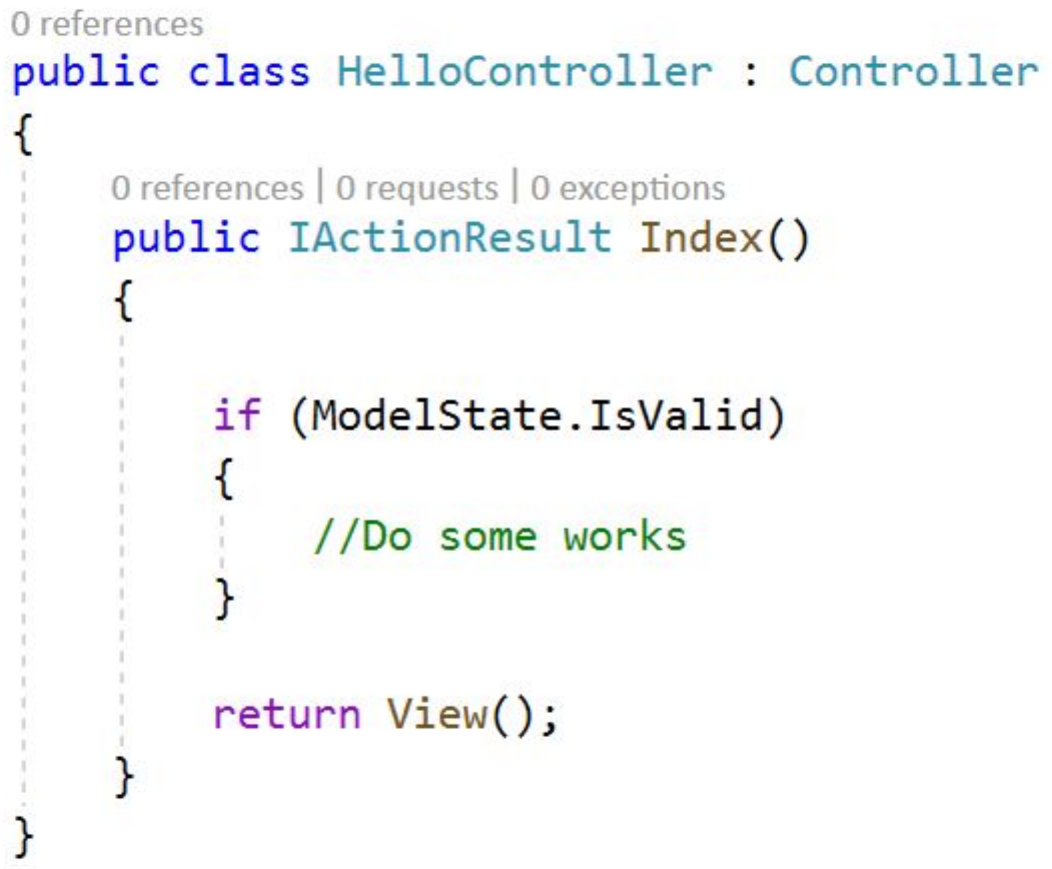
Model state

Model state represents errors that come from two subsystems: model binding and model validation.



Model state

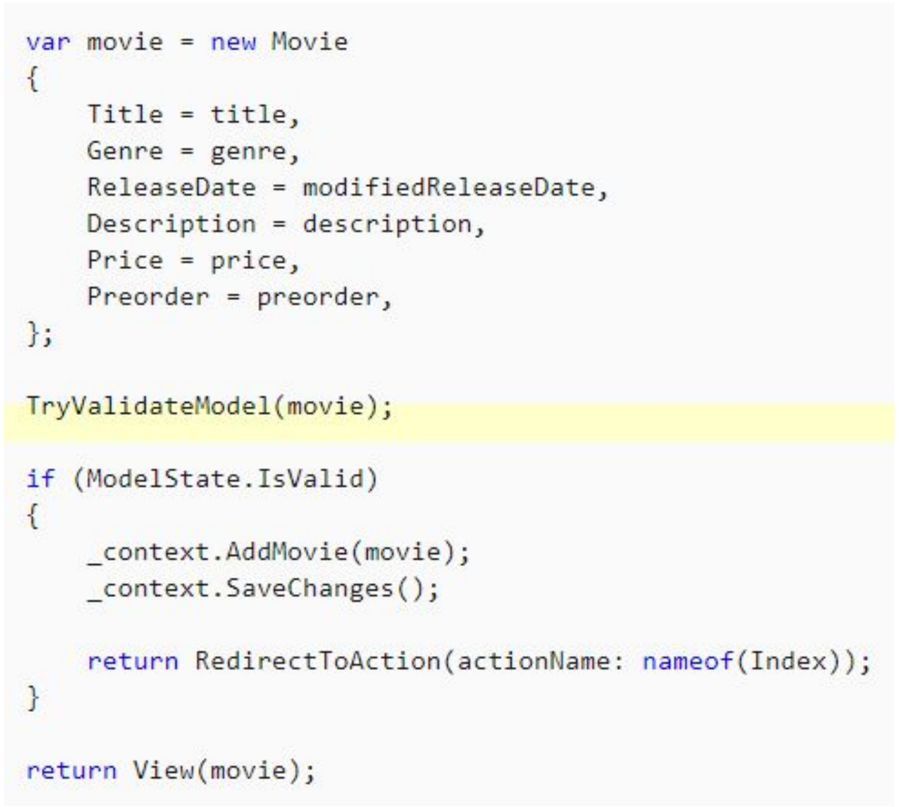
For web apps, it's the app's responsibility to inspect `ModelState.IsValid`and react appropriately. Web apps typically redisplay the page with an error message



```
0 references
public class HelloController : Controller
{
    0 references | 0 requests | 0 exceptions
    public IActionResult Index()
    {

        if (ModelState.IsValid)
        {
            //Do some works
        }

        return View();
    }
}
```

Rerun validation

Validation is automatic, but you might want to repeat it manually.

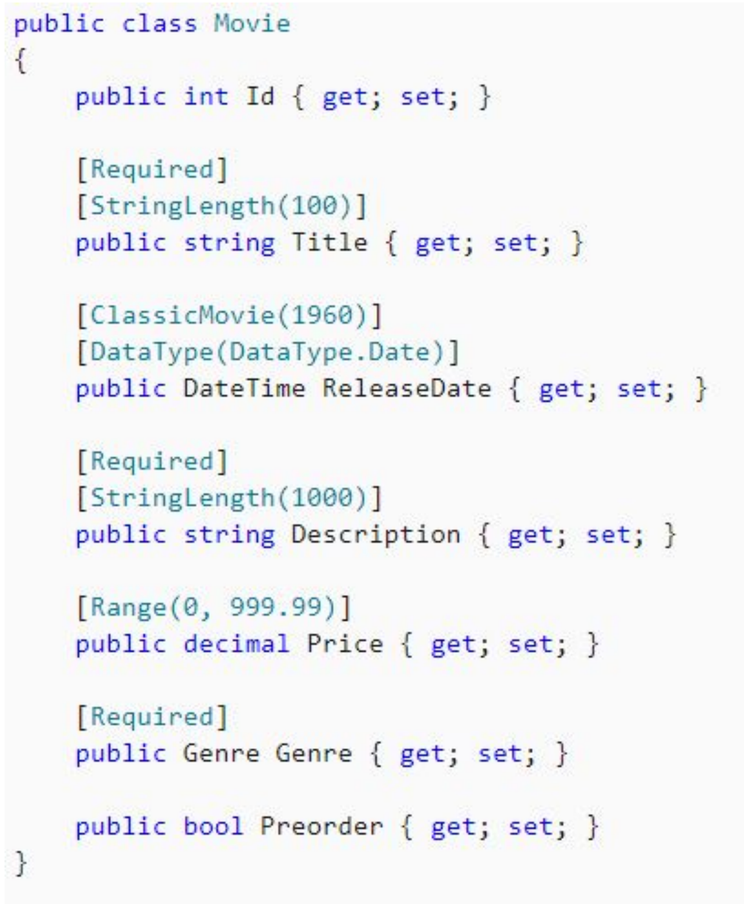


```
var movie = new Movie
{
    Title = title,
    Genre = genre,
    ReleaseDate = modifiedReleaseDate,
    Description = description,
    Price = price,
    Preorder = preorder,
};

TryValidateModel(movie);

if (ModelState.IsValid)
{
    _context.AddMovie(movie);
    _context.SaveChanges();

    return RedirectToAction(actionName: nameof(Index));
}

return View(movie);
```

Validation attributes

Validation attributes let you specify validation rules for model properties

The `[ClassicMovie]` attribute is a custom validation attribute and the others are built-in



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

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

    [ClassicMovie(1960)]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

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

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

    [Required]
    public Genre Genre { get; set; }

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

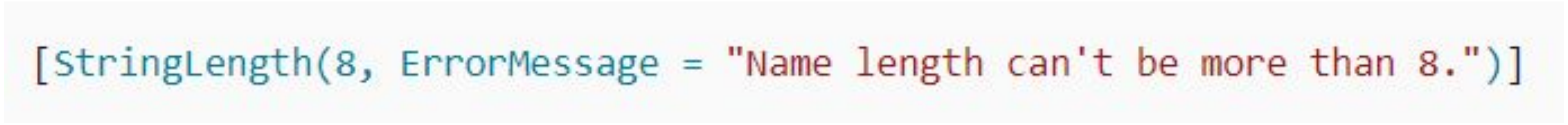
Built-in attributes

Here are some of the built-in validation attributes:

- `[CreditCard]`: Validates that the property has a credit card format.
- `[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 is not 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.

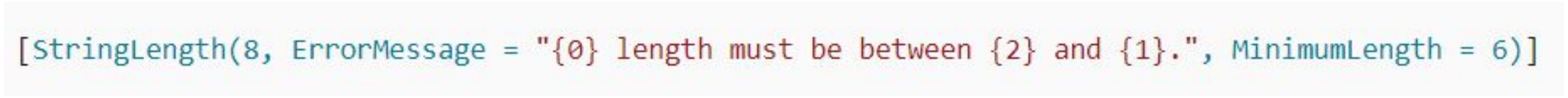
Error messages

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



```
[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



```
[StringLength(8, ErrorMessage = "{0} length must be between {2} and {1}.", MinimumLength = 6)]
```

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

[Required] attribute

By default, the validation system treats non-nullable parameters or properties as if they had a `[Required]` attribute.

[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.

[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.

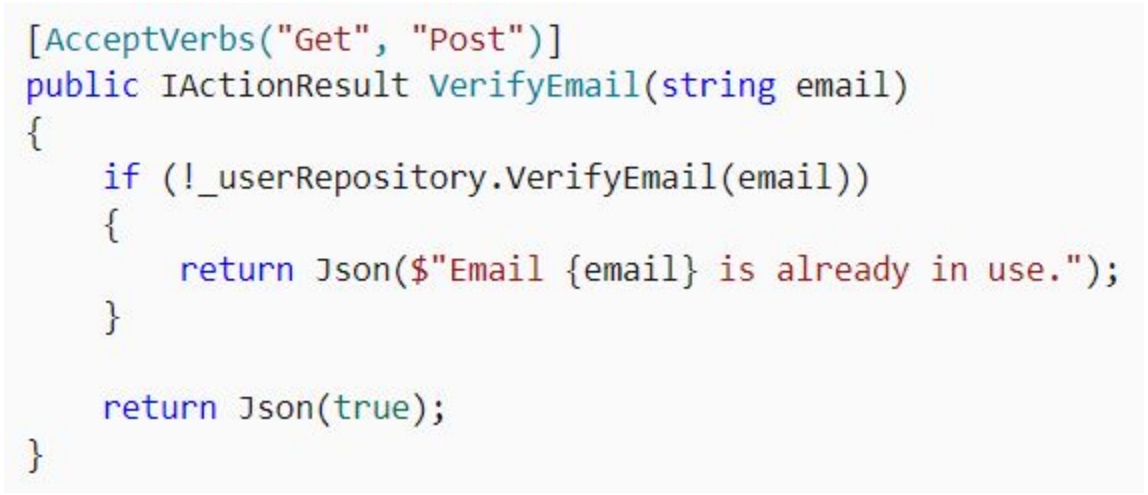
As noted earlier, non-nullable types are treated as though they had a `[Required]` attribute. That means you get client-side validation even if you don't apply the `[Required]` attribute. But if you don't use the attribute, you get a default error message. To specify a custom error message, use the attribute.

[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

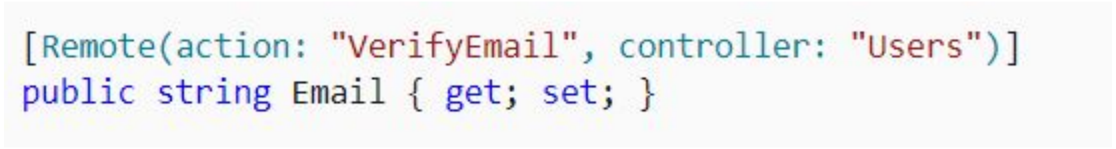
1. Create an action method for JavaScript to call. The jQuery Validate remote method expects a JSON response:
   - `"true"` means the input data is valid.
   - `"false"`, `undefined`, or `null` means the input is invalid. Display the default error message.
   - Any other string means the input is invalid. Display the string as a custom error message.



```
[AcceptVerbs("Get", "Post")]
public IActionResult VerifyEmail(string email)
{
    if (!_userRepository.VerifyEmail(email))
    {
        return Json($"Email {email} is already in use.");
    }

    return Json(true);
}
```

To implement remote validation

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:



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

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

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

To validate two or more additional fields, provide them as a comma-delimited list

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

Alternatives to built-in attributes

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

- Create custom attributes.
- Implement IValidatableObject.

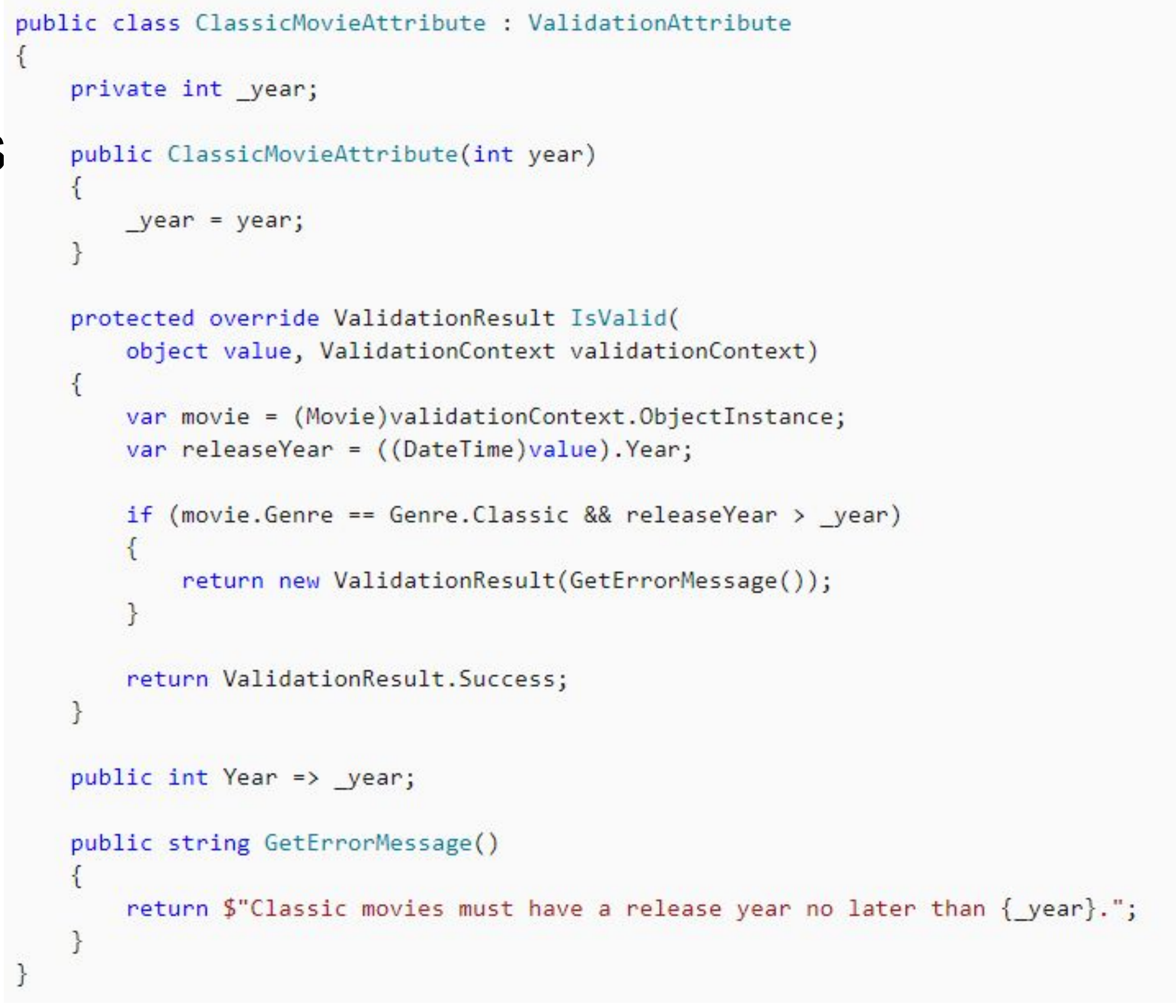
Custom attributes

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.

Custom attributes

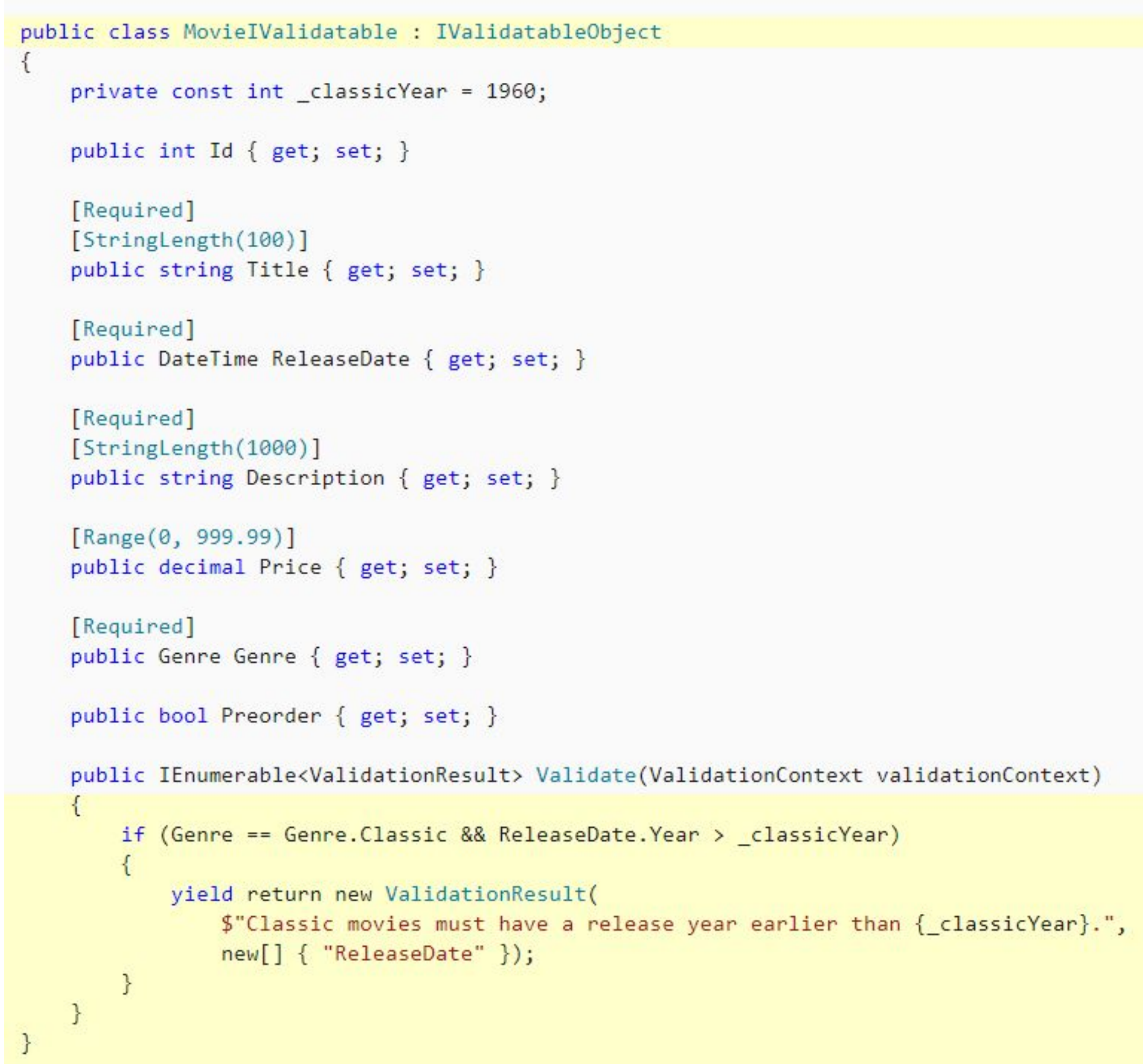
The following example validates that the release date for a movie in the *Classic* genre isn't later than a specified year.



```
public class ClassicMovieAttribute : ValidationAttribute
{
    private int _year;

    public ClassicMovieAttribute(int year)
    {
        _year = 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;
    }

    public int Year => _year;

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

IValidatableObject

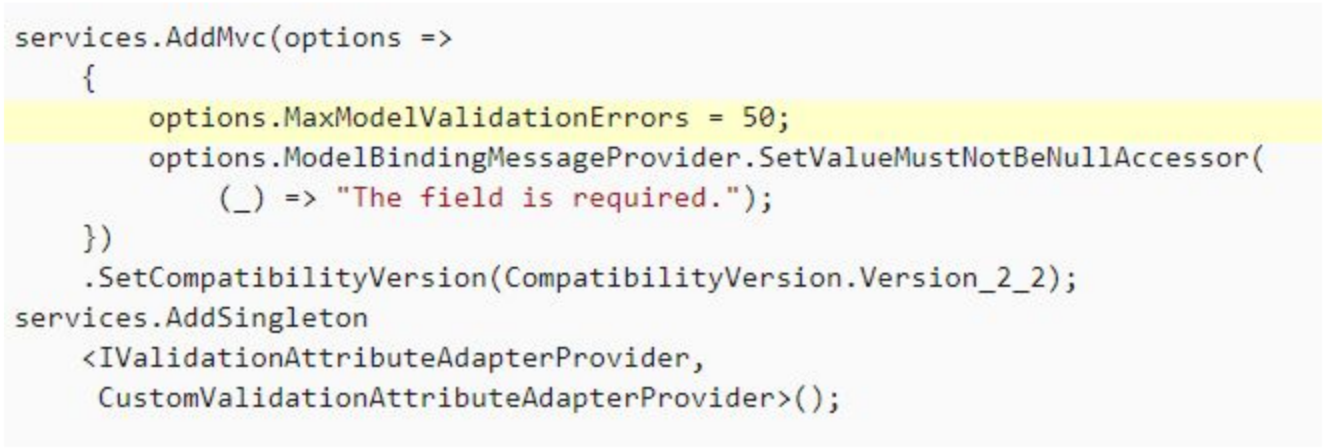
The preceding example works only with `Movie` types. Another option for class-level validation is to implement `IValidatableObject` in the model class



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

    public int Id { get; set; }

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

    [Required]
    public DateTime ReleaseDate { get; set; }

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

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

    [Required]
    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 earlier than {_classicYear}.",
                new[] { "ReleaseDate" });
        }
    }
}
```

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 `Startup.ConfigureServices`



```
services.AddMvc(options =>
    {
        options.MaxModelValidationErrors = 50;
        options.ModelBindingMessageProvider.SetValueMustNotBeNullAccessor(
            (_) => "The field is required.");
    })
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
services.AddSingleton
    <IValidationAttributeAdapterProvider,
     CustomValidationAttributeAdapterProvider>();
```

Disable validation

```
public class NullObjectModelValidator : IObjectModelValidator
{
    public void Validate(
        ActionContext actionContext,
        ValidationStateDictionary validationState,
        string prefix,
        object model)
    {
    }
}
```

```
// There is only one `IObjectModelValidator` object,
// so AddSingleton replaces the default one.
services.AddSingleton<IObjectModelValidator>(new NullObjectModelValidator());
```

You might still see model state errors that originate from model binding

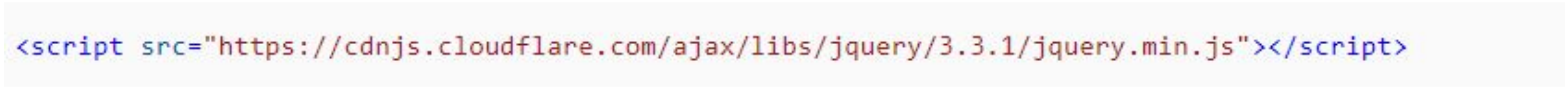
Client-side validation

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

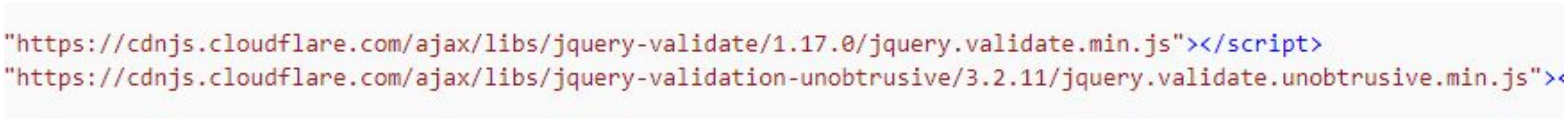
Client-side validation avoids an unnecessary round trip to the server when there are input errors on a form

Js

The following script references in _*Layout.cshtml* and _*ValidationScriptsPartial.cshtml* support client-side validation:



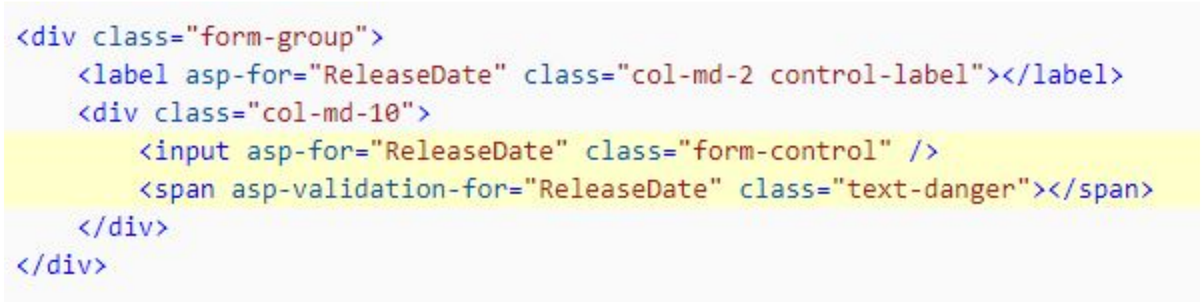
```
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
```



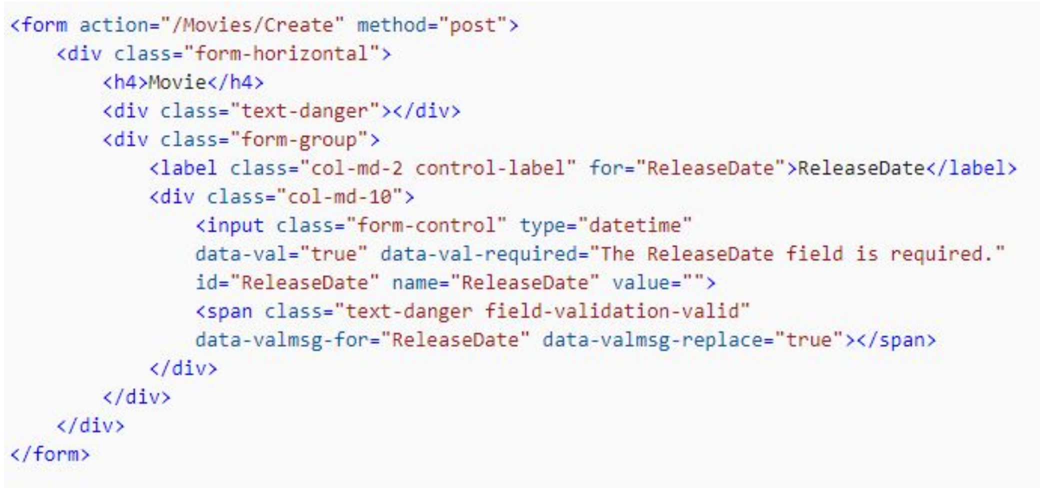
```
"https://cdnjs.cloudflare.com/ajax/libs/jquery-validate/1.17.0/jquery.validate.min.js"></script>
"https://cdnjs.cloudflare.com/ajax/libs/jquery-validation-unobtrusive/3.2.11/jquery.validate.unobtrusive.min.js"><
```

The [jQuery Unobtrusive Validation] script is a custom Microsoft front-end library t7hat builds on the popular [jQuery Validate] 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 Validate, effectively "copying" the server-side validation logic to the client.

How it works

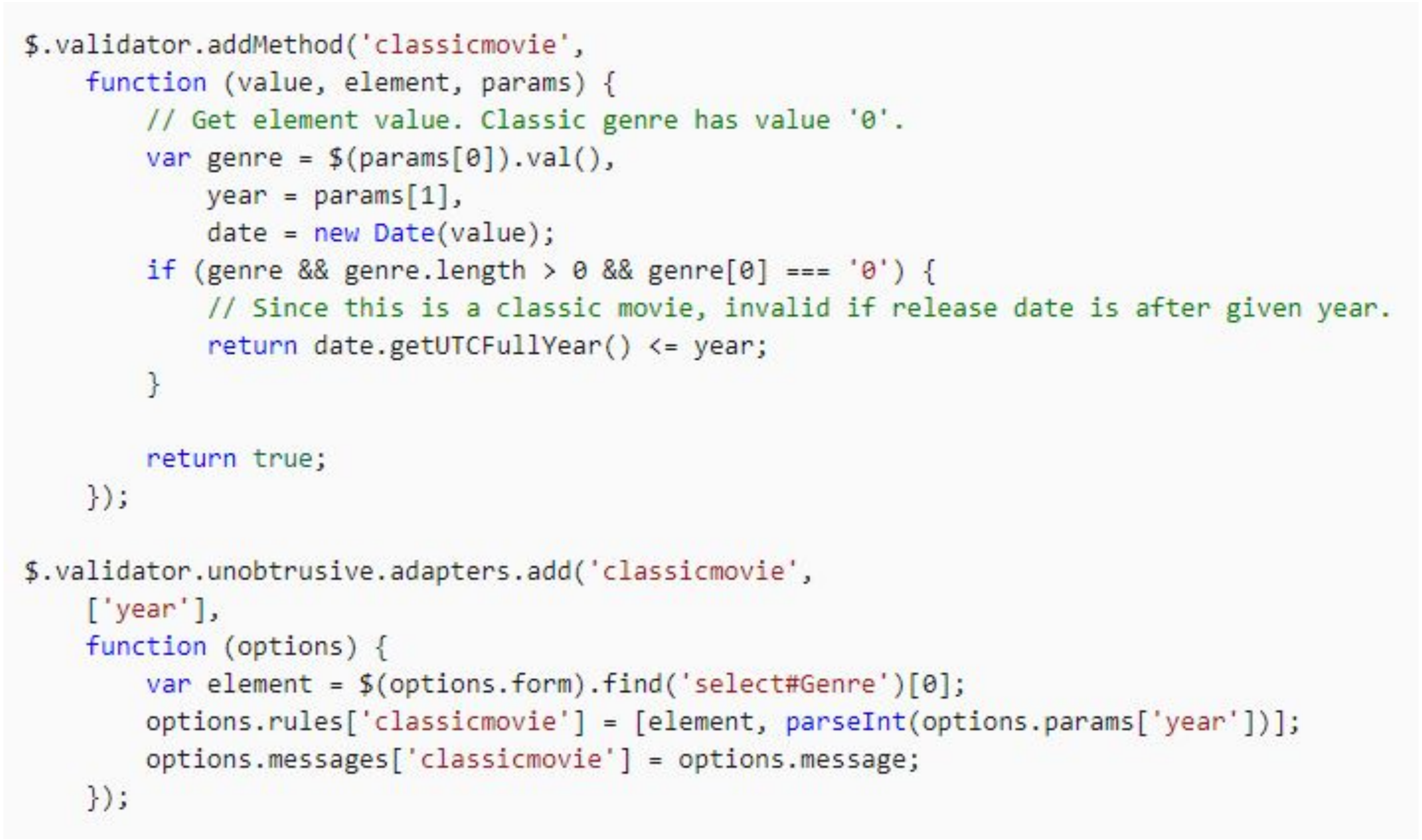


```
<div class="form-group">
    <label asp-for="ReleaseDate" class="col-md-2 control-label"></label>
    <div class="col-md-10">
        <input asp-for="ReleaseDate" class="form-control" />
        <span asp-validation-for="ReleaseDate" class="text-danger"></span>
    </div>
</div>
```



```
<form action="/Movies/Create" method="post">
    <div class="form-horizontal">
        <h4>Movie</h4>
        <div class="text-danger"></div>
        <div class="form-group">
            <label class="col-md-2 control-label" for="ReleaseDate">ReleaseDate</label>
            <div class="col-md-10">
                <input class="form-control" type="datetime"
                data-val="true" data-val-required="The ReleaseDate field is required."
                id="ReleaseDate" name="ReleaseDate" value="">
                <span class="text-danger field-validation-valid"
                data-valmsg-for="ReleaseDate" data-valmsg-replace="true"></span>
            </div>
        </div>
    </div>
</form>
```

Custom client-side validation



```
$.validator.addMethod('classicmovie',
    function (value, element, params) {
        // Get element value. Classic genre has value '0'.
        var genre = $(params[0]).val(),
            year = params[1],
            date = new Date(value);
        if (genre && genre.length > 0 && genre[0] === '0') {
            // Since this is a classic movie, invalid if release date is after given year.
            return date.getUTCFullYear() <= year;
        }

        return true;
    });

$.validator.unobtrusive.adapters.add('classicmovie',
    ['year'],
    function (options) {
        var element = $(options.form).find('select#Genre')[0];
        options.rules['classicmovie'] = [element, parseInt(options.params['year'])];
        options.messages['classicmovie'] = options.message;
    });
```