**model binding**

Model binding in ASP.NET Core MVC maps data from HTTP requests to action method parameters. The parameters may be simple types such as strings, integers, or floats, or they may be complex types. This is a great feature of MVC because mapping incoming data to a counterpart is an often repeated scenario, regardless of size or complexity of the data. MVC solves this problem by abstracting binding away so developers don't have to keep rewriting a slightly different version of that same code in every app. Writing your own text to type converter code is tedious, and error prone.

Below is a list of the data sources in the order that model binding looks through them:

1. Form values: These are form values that go in the HTTP request using the POST method. (including jQuery POST requests).
2. Route values: The set of route values provided by [Routing](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/routing?view=aspnetcore-2.1)
3. Query strings: The query string part of the URI.

## Customize model binding behavior with attributes

MVC contains several attributes that you can use to direct its default model binding behavior to a different source. For example, you can specify whether binding is required for a property, or if it should never happen at all by using the [BindRequired] or [BindNever] attributes. Alternatively, you can override the default data source, and specify the model binder's data source. Below is a list of model binding attributes:

* [BindRequired]: This attribute adds a model state error if binding cannot occur.
* [BindNever]: Tells the model binder to never bind to this parameter.
* [FromHeader], [FromQuery], [FromRoute], [FromForm]: Use these to specify the exact binding source you want to apply.
* [FromServices]: This attribute uses [dependency injection](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-2.1) to bind parameters from services.
* [FromBody]: Use the configured formatters to bind data from the request body. The formatter is selected based on content type of the request.
* [ModelBinder]: Used to override the default model binder, binding source and name.

## Bind formatted data from the request body

Request data can come in a variety of formats including JSON, XML and many others. When you use the [FromBody] attribute to indicate that you want to bind a parameter to data in the request body, MVC uses a configured set of formatters to handle the request data based on its content type. By default MVC includes a JsonInputFormatter class for handling JSON data, but you can add additional formatters for handling XML and other custom formats.

There can be at most one parameter per action decorated with [FromBody]. The ASP.NET Core MVC run-time delegates the responsibility of reading the request stream to the formatter. Once the request stream is read for a parameter, it's generally not possible to read the request stream again for binding other [FromBody] parameters.

The JsonInputFormatter is the default formatter and is based on [**Json.NET**](https://www.newtonsoft.com/json).

ASP.NET Core selects input formatters based on the [Content-Type](https://www.w3.org/Protocols/rfc1341/4_Content-Type.html) header and the type of the parameter, unless there's an attribute applied to it specifying otherwise. If you'd like to use XML or another format you must configure it in the Startup.cs file, but you may first have to obtain a reference to Microsoft.AspNetCore.Mvc.Formatters.Xml using NuGet.

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc()

.AddXmlSerializerFormatters();

}

## Model Validation Attributes

Validation attributes are a way to configure model validation so it's similar conceptually to validation on fields in database tables. This includes constraints such as assigning data types or required fields. Other types of validation include applying patterns to data to enforce business rules, such as a credit card, phone number, or email address. Validation attributes make enforcing these requirements much simpler and easier to use.

Simply reading through the model reveals the rules about data for this app, making it easier to maintain the code. Below are several popular built-in validation attributes:

* [CreditCard]: Validates the property has a credit card format.
* [Compare]: Validates two properties in a model match.
* [EmailAddress]: Validates the property has an email format.
* [Phone]: Validates the property has a telephone format.
* [Range]: Validates the property value falls within the given range.
* [RegularExpression]: Validates that the data matches the specified regular expression.
* [Required]: Makes a property required.
* [StringLength]: Validates that a string property has at most the given maximum length.
* [Url]: Validates the property has a URL format.

There may be instances where you need more features than built-in attributes provide. For those times, you can create custom validation attributes by deriving from ValidationAttribute or changing your model to implement IValidatableObject.

## Model State

Model state represents validation errors in submitted HTML form values.

MVC will continue validating fields until reaches the maximum number of errors (200 by default). You can configure this number by inserting the following code into the ConfigureServices method in the Startup.cs file:

services.AddMvc(options => options.MaxModelValidationErrors = 50);

## Custom validation

Validation attributes work for most validation needs. However, some validation rules are specific to your business. Your rules might not be common data validation techniques such as ensuring a field is required or that it conforms to a range of values. For these scenarios, custom validation attributes are a great solution. Creating your own custom validation attributes in MVC is easy. Just inherit from the ValidationAttribute, and override the IsValid method. The IsValid method accepts two parameters, the first is an object named value and the second is a ValidationContext object named validationContext. Value refers to the actual value from the field that your custom validator is validating.

public class ClassicMovieAttribute : ValidationAttribute, IClientModelValidator

{

private int \_year;

public ClassicMovieAttribute(int year)

{

\_year = year;

}

protected override ValidationResult IsValid(object value, ValidationContext validationContext)

{

Movie movie = (Movie)validationContext.ObjectInstance;

if (movie.Genre == Genre.Classic && movie.ReleaseDate.Year > \_year)

{

return new ValidationResult(GetErrorMessage());

}

return ValidationResult.Success;

}

Alternatively, this same code could be placed in the model by implementing the Validatemethod on the IValidatableObject interface. While custom validation attributes work well for validating individual properties, implementing IValidatableObject can be used to implement class-level validation as seen here.

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" });

}

}

## Client side validation

Client side validation is a great convenience for users. It saves time they would otherwise spend waiting for a round trip to the server. In business terms, even a few fractions of seconds multiplied hundreds of times each day adds up to be a lot of time, expense, and frustration. Straightforward and immediate validation enables users to work more efficiently and produce better quality input and output.

You must have a view with the proper JavaScript script references in place for client side validation to work as you see here.

<script src="https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.2.0.min.js"></script>

<script src="https://ajax.aspnetcdn.com/ajax/jquery.validate/1.16.0/jquery.validate.min.js"></script>

<script src="https://ajax.aspnetcdn.com/ajax/jquery.validation.unobtrusive/3.2.6/jquery.validate.unobtrusive.min.js"></script>

MVC's [Tag Helpers](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/intro?view=aspnetcore-2.1) and [HTML helpers](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/overview?view=aspnetcore-2.1) are able to use the validation attributes and type metadata from model properties to render HTML 5 [data- attributes](http://w3c.github.io/html/dom.html#embedding-custom-non-visible-data-with-the-data-attributes) in the form elements that need validation. MVC generates the data- attributes for both built-in and custom attributes. jQuery Unobtrusive Validation then parses the data- attributes and passes the logic to jQuery Validate, effectively "copying" the server side validation logic to the client.

### **Add Validation to Dynamic Forms**

Because jQuery Unobtrusive Validation passes validation logic and parameters to jQuery Validate when the page first loads, dynamically generated forms won't automatically exhibit validation. Instead, you must tell jQuery Unobtrusive Validation to parse the dynamic form immediately after creating it. For example, the code below shows how you might set up client side validation on a form added via AJAX.

$.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);

}

})

### **Add Validation to Dynamic Controls**

You can also update the validation rules on a form when individual controls, such as <input/>s and <select/>s, are dynamically generated. You cannot pass selectors for these elements to the parse() method directly because the surrounding form has already been parsed and won't update. Instead, you first remove the existing validation data, then reparse the entire form, as shown below:

$.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 Validate

.removeData("unobtrusiveValidation"); // Added by jQuery Unobtrusive Validation

$.validator.unobtrusive.parse(form);

}

})

**Remote validation**

Remote validation is a great feature to use when you need to validate data on the client against data on the server. For example, your app may need to verify whether an email or user name is already in use, and it must query a large amount of data to do so. Downloading large sets of data for validating one or a few fields consumes too many resources. It may also expose sensitive information. An alternative is to make a round-trip request to validate a field.

You can implement remote validation in a two step process. First, you must annotate your model with the [Remote] attribute. The [Remote] attribute accepts multiple overloads you can use to direct client side JavaScript to the appropriate code to call. The example below points to the VerifyEmail action method of the Users controller.

[Remote(action: "VerifyEmail", controller: "Users")]

public string Email { get; set; }

The second step is putting the validation code in the corresponding action method as defined in the [Remote] attribute. According to the jQuery Validate [remote](https://jqueryvalidation.org/remote-method/) method documentation, the server response must be a JSON string that's either:

* "true" for valid elements.
* "false", undefined, or null for invalid elements, using the default error message.

If the server response is a string (for example, "That name is already taken, try peter123 instead"), the string is displayed as a custom error message in place of the default string.

[AcceptVerbs("Get", "Post")]

public IActionResult VerifyEmail(string email)

{

if (!\_userRepository.VerifyEmail(email))

{

return Json($"Email {email} is already in use.");

}

return Json(true);

}

The AdditionalFields property of the [Remote] attribute is useful for validating combinations of fields against data on the server. For example, if the User model from above had two additional properties called FirstName and LastName, 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; }

AdditionalFields could've been set explicitly to the strings "FirstName" and "LastName", but using the [nameof](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/nameof) operator like this simplifies later refactoring. The action method to perform the validation must then accept two arguments, one for the value of FirstName and one for the value of LastName.

[AcceptVerbs("Get", "Post")]

public IActionResult VerifyName(string firstName, string lastName)

{

if (!\_userRepository.VerifyName(firstName, lastName))

{

return Json(data: $"A user named {firstName} {lastName} already exists.");

}

return Json(data: true);

}

If you need to validate two or more additional fields with the [Remote] attribute, you 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 code:

[Remote(action: "VerifyName", controller: "Users", AdditionalFields = nameof(FirstName) + "," + nameof(LastName))]

public string MiddleName { get; set; }