ASP.NET Core 2.2

Version

3.0 Preview 2

2.2

2.1

2.0

1.1

1.0

Model Binding in ASP.NET Core

11/13/2018 7 minutes to read Contributors all

In this article

By Rachel Appel

Introduction to 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.

How model binding works

When MVC receives an HTTP request, it routes it to a specific action method of a controller. It determines which action method to run based on what is in the route data, then it binds values from the HTTP request to that action method's parameters. For example, consider the following URL:

`http://contoso.com/movies/edit/2`

Since the route template looks like this, `{controller=Home}/{action=Index}/{id?}`, `movies/edit/2` routes to the `Movies` controller, and its `Edit` action method. It also accepts an optional parameter called `id`. The code for the action method should look something like this:

C# Copy

```csharp
public IActionResult Edit(int? id)
```

Note: The strings in the URL route are not case sensitive.

MVC will try to bind request data to the action parameters by name. MVC will look for values for each parameter using the parameter name and the names of its public settable properties. In the above example, the only action parameter is named `id`, which MVC binds to the value with the same name in the route values. In addition to route values MVC will bind data from various parts of the request and it does so in a set order. 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
3. `Query strings`: The query string part of the URI.

Note: Form values, route data, and query strings are all stored as name-value pairs.

Since model binding asked for a key named `id` and there's nothing named `id` in the form values, it moved on to the route values looking for that key. In our example, it's a match. Binding happens, and the value is converted to the integer 2. The same request using Edit(string id) would convert to the string "2".

So far the example uses simple types. In MVC simple types are any .NET primitive type or type with a string type converter. If the action method's parameter were a class such as the `Movie` type, which contains both simple and complex types as properties, MVC's model binding will still handle it nicely. It uses reflection and recursion to traverse the properties of complex types looking for matches. Model binding looks for the pattern *parameter_name.property_name* to bind values to properties. If it doesn't find matching values of this form, it will attempt to bind using just the property name. For those types such as `Collection` types, model binding looks for matches to *parameter_name[index]* or just *[index]*. Model binding treats `Dictionary` types similarly, asking for *parameter_name[key]* or just *[key]*, as long as the keys are simple types. Keys that are supported match the field names HTML and tag helpers generated for the same model type. This enables round-tripping values so that the form fields remain filled with the user's input for their convenience, for example, when bound data from a create or edit didn't pass validation.

To make model binding possible, the class must have a public default constructor and public writable properties to bind. When model binding occurs, the class is instantiated using the public default constructor, then the properties can be set.

When a parameter is bound, model binding stops looking for values with that name and it moves on to bind the next parameter. Otherwise, the default model binding behavior sets parameters to their default values depending on their type:

- `T[]`: With the exception of arrays of type `byte[]`, binding sets parameters of type `T[]` to `Array.Empty<T>()`. Arrays of type `byte[]` are set to `null`.
- Reference Types: Binding creates an instance of a class with the default constructor without setting properties. However, model binding sets `string` parameters to `null`.
- Nullable Types: Nullable types are set to `null`. In the above example, model binding sets `id` to `null` since it's of type `int?`.
- Value Types: Non-nullable value types of type `T` are set to `default(T)`. For example, model binding will set a parameter `int id` to 0. Consider using model validation or nullable types rather than relying on default values.

If binding fails, MVC doesn't throw an error. Every action which accepts user input should check the `ModelState.IsValid` property.

Note: Each entry in the controller's `ModelState` property is a `ModelStateEntry` containing an `Errors` property. It's rarely necessary to query this collection yourself. Use `ModelState.IsValid` instead.

Additionally, there are some special data types that MVC must consider when performing model binding:

- `IFormFile`, `IEnumerable<IFormFile>`: One or more uploaded files that are part of the HTTP request.
- `CancellationToken`: Used to cancel activity in asynchronous controllers.

These types can be bound to action parameters or to properties on a class type.

Once model binding is complete, Validation occurs. Default model binding works great for the vast majority of development scenarios. It's also extensible so if you have unique needs you can customize the built-in behavior.

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 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.

Attributes are very helpful tools when you need to override the default behavior of model binding.

Customize model binding and validation globally

The model binding and validation system's behavior is driven by ModelMetadata that describes:

- How a model is to be bound.
- How validation occurs on the type and its properties.

Aspects of the system's behavior can be configured globally by adding a details provider to MvcOptions.ModelMetadataDetailsProviders. MVC has a few built-in details providers that allow configuring behavior such as disabling model binding or validation for certain types.

To disable model binding on all models of a certain type, add an ExcludeBindingMetadataProvider in `Startup.ConfigureServices`. For example, to disable model binding on all models of type `System.Version`:

C# Copy

```
services.AddMvc().AddMvcOptions(options =>
    options.ModelMetadataDetailsProviders.Add(
        new ExcludeBindingMetadataProvider(typeof(System.Version))));
```

To disable validation on properties of a certain type, add a SuppressChildValidationMetadataProvider in `Startup.ConfigureServices`. For example, to disable validation on properties of type `System.Guid`:

C# Copy

```
services.AddMvc().AddMvcOptions(options =>
    options.ModelMetadataDetailsProviders.Add(
        new SuppressChildValidationMetadataProvider(typeof(System.Guid))));
```

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.

Note

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.

Note

The `JsonInputFormatter` is the default formatter and is based on Json.NET.

ASP.NET Core selects input formatters based on the Content-Type 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. Your startup code should look something like this:

C# Copy

```csharp
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .AddXmlSerializerFormatters();
   }
```

Code in the *Startup.cs* file contains a `ConfigureServices` method with a `services` argument you can use to build up services for your ASP.NET Core app. In the sample, we are adding an XML formatter as a service that MVC will provide for this app. The `options` argument passed into the `AddMvc` method allows you to add and manage filters, formatters, and other system options from MVC upon app startup. Then apply the `Consumes` attribute to controller classes or action methods to work with the format you want.

Custom Model Binding

You can extend model binding by writing your own custom model binders. Learn more about custom model binding.