

Model Binding in ASP.NET Core

12/18/2019 • 32 minutes to read •  +11

In this article

[What is Model binding](#)

[Example](#)

[Targets](#)

[Sources](#)

[No source for a model property](#)

[Type conversion errors](#)

[Simple types](#)

[Complex types](#)

[Collections](#)

[Dictionaries](#)

[Constructor binding and record types](#)

[Globalization behavior of model binding route data and query strings](#)

[Special data types](#)

[Input formatters](#)

[Exclude specified types from model binding](#)

[Custom model binders](#)

[Manual model binding](#)

[\[FromServices\] attribute](#)

[Additional resources](#)

This article explains what model binding is, how it works, and how to customize its behavior.

[View or download sample code \(how to download\).](#)

What is Model binding


Controllers and Razor pages work with data that comes from HTTP requests. For example, route data may provide a record key, and posted form fields may provide values for the properties of the model. Writing code to retrieve each of these values and convert them from strings to .NET types would be tedious and error-prone. Model binding automates this process. The model binding system:

- Retrieves data from various sources such as route data, form fields, and query strings.
- Provides the data to controllers and Razor pages in method parameters and public properties.
- Converts string data to .NET types.
- Updates properties of complex types.

Example

Suppose you have the following action method:

C#

 Copy

```
[HttpGet("{id}")]  
public ActionResult<Pet> GetById(int id, bool dogsOnly)
```

And the app receives a request with this URL:

 Copy

```
http://contoso.com/api/pets/2?DogsOnly=true
```

Model binding goes through the following steps after the routing system selects the action method:

- Finds the first parameter of `GetById`, an integer named `id`.

- Looks through the available sources in the HTTP request and finds `id = "2"` in route data.
- Converts the string "2" into integer 2.
- Finds the next parameter of `GetById`, a boolean named `dogsOnly`.
- Looks through the sources and finds "DogsOnly=true" in the query string. Name matching is not case-sensitive.
- Converts the string "true" into boolean `true`.

The framework then calls the `GetById` method, passing in 2 for the `id` parameter, and `true` for the `dogsOnly` parameter.

In the preceding example, the model binding targets are method parameters that are simple types. Targets may also be the properties of a complex type. After each property is successfully bound, [model validation](#) occurs for that property. The record of what data is bound to the model, and any binding or validation errors, is stored in [ControllerBase.ModelState](#) or [PageModel.ModelState](#). To find out if this process was successful, the app checks the [ModelState.IsValid](#) flag.

Targets

Model binding tries to find values for the following kinds of targets:

- Parameters of the controller action method that a request is routed to.
- Parameters of the Razor Pages handler method that a request is routed to.
- Public properties of a controller or `PageModel` class, if specified by attributes.

[BindProperty] attribute

Can be applied to a public property of a controller or `PageModel` class to cause model binding to target that property:

C#

 Copy

```
public class EditModel : InstructorsPageModel
{
```

```
[BindProperty]  
public Instructor Instructor { get; set; }
```

[BindProperties] attribute

Available in ASP.NET Core 2.1 and later. Can be applied to a controller or `PageModel` class to tell model binding to target all public properties of the class:

C#



```
[BindProperties(SupportsGet = true)]  
public class CreateModel : InstructorsPageModel  
{  
    public Instructor Instructor { get; set; }
```

Model binding for HTTP GET requests

By default, properties are not bound for HTTP GET requests. Typically, all you need for a GET request is a record ID parameter. The record ID is used to look up the item in the database. Therefore, there is no need to bind a property that holds an instance of the model. In scenarios where you do want properties bound to data from GET requests, set the `SupportsGet` property to `true`:

C#



```
[BindProperty(Name = "ai_user", SupportsGet = true)]  
public string ApplicationInsightsCookie { get; set; }
```

Sources

By default, model binding gets data in the form of key-value pairs from the following sources in an HTTP request:

1. Form fields
2. The request body (For [controllers that have the \[ApiController\] attribute.](#))
3. Route data
4. Query string parameters
5. Uploaded files

For each target parameter or property, the sources are scanned in the order indicated in the preceding list. There are a few exceptions:

- Route data and query string values are used only for simple types.
- Uploaded files are bound only to target types that implement `IFormFile` or `IEnumerable<IFormFile>`.

If the default source is not correct, use one of the following attributes to specify the source:

- [\[FromQuery\]](#) - Gets values from the query string.
- [\[FromRoute\]](#) - Gets values from route data.
- [\[FromForm\]](#) - Gets values from posted form fields.
- [\[FromBody\]](#) - Gets values from the request body.
- [\[FromHeader\]](#) - Gets values from HTTP headers.

These attributes:

- Are added to model properties individually (not to the model class), as in the following example:

```
C# Copy  
  
public class Instructor  
{  
    public int ID { get; set; }  
}
```

```
[FromQuery(Name = "Note")]  
public string NoteFromQueryString { get; set; }
```

- Optionally accept a model name value in the constructor. This option is provided in case the property name doesn't match the value in the request. For instance, the value in the request might be a header with a hyphen in its name, as in the following example:

C#

 Copy

```
public void OnGet([FromHeader(Name = "Accept-Language")] string language)
```

[FromBody] attribute

Apply the `[FromBody]` attribute to a parameter to populate its properties from the body of an HTTP request. The ASP.NET Core runtime delegates the responsibility of reading the body to an input formatter. Input formatters are explained [later in this article](#).

When `[FromBody]` is applied to a complex type parameter, any binding source attributes applied to its properties are ignored. For example, the following `Create` action specifies that its `pet` parameter is populated from the body:

C#

 Copy

```
public ActionResult<Pet> Create([FromBody] Pet pet)
```

The `Pet` class specifies that its `Breed` property is populated from a query string parameter:

C#

 Copy

```
public class Pet  
{
```

```
public string Name { get; set; }

[FromQuery] // Attribute is ignored.
public string Breed { get; set; }
}
```

In the preceding example:

- The `[FromQuery]` attribute is ignored.
- The `Breed` property is not populated from a query string parameter.

Input formatters read only the body and don't understand binding source attributes. If a suitable value is found in the body, that value is used to populate the `Breed` property.

Don't apply `[FromBody]` to more than one parameter per action method. Once the request stream is read by an input formatter, it's no longer available to be read again for binding other `[FromBody]` parameters.

Additional sources

Source data is provided to the model binding system by *value providers*. You can write and register custom value providers that get data for model binding from other sources. For example, you might want data from cookies or session state. To get data from a new source:

- Create a class that implements `IValueProvider`.
- Create a class that implements `IValueProviderFactory`.
- Register the factory class in `Startup.ConfigureServices`.

The sample app includes a [value provider](#) and [factory](#) example that gets values from cookies. Here's the registration code in `Startup.ConfigureServices`:

C#



```
services.AddRazorPages()  
    .AddMvcOptions(options =>  
{  
    options.ValueProviderFactories.Add(new CookieValueProviderFactory());  
    options.ModelMetadataDetailsProviders.Add(  
        new ExcludeBindingMetadataProvider(typeof(System.Version)));  
    options.ModelMetadataDetailsProviders.Add(  
        new SuppressChildValidationMetadataProvider(typeof(System.Guid)));  
    })  
    .AddXmlSerializerFormatters();
```

The code shown puts the custom value provider after all the built-in value providers. To make it the first in the list, call `Insert(0, new CookieValueProviderFactory())` instead of `Add`.

No source for a model property

By default, a model state error isn't created if no value is found for a model property. The property is set to null or a default value:

- Nullable simple types are set to `null`.
- Non-nullable value types are set to `default(T)`. For example, a parameter `int id` is set to 0.
- For complex Types, model binding creates an instance by using the default constructor, without setting properties.
- Arrays are set to `Array.Empty<T>()`, except that `byte[]` arrays are set to `null`.

If model state should be invalidated when nothing is found in form fields for a model property, use the [\[BindRequired\]](#) attribute.

Note that this [\[BindRequired\]](#) behavior applies to model binding from posted form data, not to JSON or XML data in a request body. Request body data is handled by [input formatters](#).

Type conversion errors

If a source is found but can't be converted into the target type, model state is flagged as invalid. The target parameter or property is set to null or a default value, as noted in the previous section.

In an API controller that has the `[ApiController]` attribute, invalid model state results in an automatic HTTP 400 response.

In a Razor page, redisplay the page with an error message:

```
C# Copy  
  
public IActionResult OnPost()  
{  
    if (!ModelState.IsValid)  
    {  
        return Page();  
    }  
  
    _instructorsInMemoryStore.Add(Instructor);  
    return RedirectToPage("./Index");  
}
```

Client-side validation catches most bad data that would otherwise be submitted to a Razor Pages form. This validation makes it hard to trigger the preceding highlighted code. The sample app includes a **Submit with Invalid Date** button that puts bad data in the **Hire Date** field and submits the form. This button shows how the code for redisplaying the page works when data conversion errors occur.

When the page is redisplayed by the preceding code, the invalid input is not shown in the form field. This is because the model property has been set to null or a default value. The invalid input does appear in an error message. But if you want to redisplay the bad data in the form field, consider making the model property a string and doing the data conversion manually.

The same strategy is recommended if you don't want type conversion errors to result in model state errors. In that case, make the model property a string.

Simple types

The simple types that the model binder can convert source strings into include the following:

- [Boolean](#)
- [Byte](#), [SByte](#)
- [Char](#)
- [DateTime](#)
- [DateTimeOffset](#)
- [Decimal](#)
- [Double](#)
- [Enum](#)
- [Guid](#)
- [Int16](#), [Int32](#), [Int64](#)
- [Single](#)
- [TimeSpan](#)
- [UInt16](#), [UInt32](#), [UInt64](#)
- [Uri](#)
- [Version](#)

Complex types

A complex type 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.

For each property of the complex type, model binding looks through the sources for the name pattern *prefix.property_name*. If nothing is found, it looks for just *property_name* without the prefix.

For binding to a parameter, the prefix is the parameter name. For binding to a `PageModel` public property, the prefix is the public property name. Some attributes have a `Prefix` property that lets you override the default usage of parameter or property name.

For example, suppose the complex type is the following `Instructor` class:

C#



```
public class Instructor
{
    public int ID { get; set; }
    public string LastName { get; set; }
    public string FirstName { get; set; }
}
```

Prefix = parameter name

If the model to be bound is a parameter named `instructorToUpdate`:

C#



```
public IActionResult OnPost(int? id, Instructor instructorToUpdate)
```

Model binding starts by looking through the sources for the key `instructorToUpdate.ID`. If that isn't found, it looks for `ID` without a prefix.

Prefix = property name

If the model to be bound is a property named `Instructor` of the controller or `PageModel` class:

C#



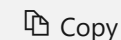
```
[BindProperty]
public Instructor Instructor { get; set; }
```

Model binding starts by looking through the sources for the key `Instructor.ID`. If that isn't found, it looks for `ID` without a prefix.

Custom prefix

If the model to be bound is a parameter named `instructorToUpdate` and a `Bind` attribute specifies `Instructor` as the prefix:

C#



```
public IActionResult OnPost(
    int? id, [Bind(Prefix = "Instructor")] Instructor instructorToUpdate)
```

Model binding starts by looking through the sources for the key `Instructor.ID`. If that isn't found, it looks for `ID` without a prefix.

Attributes for complex type targets

Several built-in attributes are available for controlling model binding of complex types:

- `[Bind]`
- `[BindRequired]`
- `[BindNever]`

⚠ Warning

These attributes affect model binding when posted form data is the source of values. They do **not** affect input formatters, which process posted JSON and XML request bodies. Input formatters are explained **later in this article**.

[Bind] attribute

Can be applied to a class or a method parameter. Specifies which properties of a model should be included in model binding.

[Bind] does **not** affect input formatters.

In the following example, only the specified properties of the `Instructor` model are bound when any handler or action method is called:

C#



```
[Bind("LastName,FirstMidName,HireDate")]  
public class Instructor
```

In the following example, only the specified properties of the `Instructor` model are bound when the `OnPost` method is called:

C#



```
[HttpPost]  
public IActionResult OnPost([Bind("LastName,FirstMidName,HireDate")] Instructor instructor)
```

The [Bind] attribute can be used to protect against overposting in *create* scenarios. It doesn't work well in edit scenarios because excluded properties are set to null or a default value instead of being left unchanged. For defense against

overposting, view models are recommended rather than the `[Bind]` attribute. For more information, see [Security note about overposting](#).

[ModelBinder] attribute

[ModelBinderAttribute](#) can be applied to types, properties, or parameters. It allows specifying the type of model binder used to bind the specific instance or type. For example:

C#

 Copy

```
[HttpPost]
public IActionResult OnPost([ModelBinder(typeof(MyInstructorModelBinder))] Instructor instructor)
```

The `[ModelBinder]` attribute can also be used to change the name of a property or parameter when it's being model bound:

C#

 Copy

```
public class Instructor
{
    [ModelBinder(Name = "instructor_id")]
    public string Id { get; set; }

    public string Name { get; set; }
}
```

[BindRequired] attribute

Can only be applied to model properties, not to method parameters. Causes model binding to add a model state error if binding cannot occur for a model's property. Here's an example:

C#

 Copy

```
public class InstructorWithCollection
{
    public int ID { get; set; }

    [DataType(DataType.Date)]
    [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
    [Display(Name = "Hire Date")]
    [BindRequired]
    public DateTime HireDate { get; set; }
```

See also the discussion of the `[Required]` attribute in [Model validation](#).

[BindNever] attribute

Can only be applied to model properties, not to method parameters. Prevents model binding from setting a model's property. Here's an example:

C#

 Copy

```
public class InstructorWithDictionary
{
    [BindNever]
    public int ID { get; set; }
```

Collections

For targets that are collections of simple types, model binding looks for matches to *parameter_name* or *property_name*. If no match is found, it looks for one of the supported formats without the prefix. For example:

- Suppose the parameter to be bound is an array named `selectedCourses`:

C#


 Copy

```
public IActionResult OnPost(int? id, int[] selectedCourses)
```

- Form or query string data can be in one of the following formats:

 Copy

```
selectedCourses=1050&selectedCourses=2000
```

 Copy

```
selectedCourses[0]=1050&selectedCourses[1]=2000
```

 Copy

```
[0]=1050&[1]=2000
```

 Copy

```
selectedCourses[a]=1050&selectedCourses[b]=2000&selectedCourses.index=a&selectedCourses.index=b
```

 Copy

```
[a]=1050&[b]=2000&index=a&index=b
```

- The following format is supported only in form data:

 Copy


```
selectedCourses[]=1050&selectedCourses[]=2000
```

- For all of the preceding example formats, model binding passes an array of two items to the `selectedCourses` parameter:
 - `selectedCourses[0]=1050`
 - `selectedCourses[1]=2000`

Data formats that use subscript numbers (... [0] ... [1] ...) must ensure that they are numbered sequentially starting at zero. If there are any gaps in subscript numbering, all items after the gap are ignored. For example, if the subscripts are 0 and 2 instead of 0 and 1, the second item is ignored.

Dictionaries

For `Dictionary` targets, model binding looks for matches to *parameter_name* or *property_name*. If no match is found, it looks for one of the supported formats without the prefix. For example:

- Suppose the target parameter is a `Dictionary<int, string>` named `selectedCourses`:

C#



```
public IActionResult OnPost(int? id, Dictionary<int, string> selectedCourses)
```

- The posted form or query string data can look like one of the following examples:



```
selectedCourses[1050]=Chemistry&selectedCourses[2000]=Economics
```



```
[1050]=Chemistry&selectedCourses[2000]=Economics
```

 Copy

```
selectedCourses[0].Key=1050&selectedCourses[0].Value=Chemistry&  
selectedCourses[1].Key=2000&selectedCourses[1].Value=Economics
```

 Copy

```
[0].Key=1050&[0].Value=Chemistry&[1].Key=2000&[1].Value=Economics
```


- For all of the preceding example formats, model binding passes a dictionary of two items to the `selectedCourses` parameter:
 - `selectedCourses["1050"]="Chemistry"`
 - `selectedCourses["2000"]="Economics"`

Constructor binding and record types

Model binding requires that complex types have a parameterless constructor. Both `System.Text.Json` and `Newtonsoft.Json` based input formatters support deserialization of classes that don't have a parameterless constructor.

C# 9 introduces record types, which are a great way to succinctly represent data over the network. ASP.NET Core adds support for model binding and validating record types with a single constructor:

C#

 Copy

```
public record Person([Required] string Name, [Range(0, 150)] int Age);  
  
public class PersonController  
{
```

```
public IActionResult Index() => View();

[HttpPost]
public IActionResult Index(Person person)
{
    ...
}
```

Person/Index.cshtml:

CSHTML

 Copy

```
@model Person

Name: <input asp-for="Name" />
...
Age: <input asp-for="Age" />
```

When validating record types, the runtime searches for validation metadata specifically on parameters rather than on properties.

Globalization behavior of model binding route data and query strings

The ASP.NET Core route value provider and query string value provider:

- Treat values as invariant culture.
- Expect that URLs are culture-invariant.

In contrast, values coming from form data undergo a culture-sensitive conversion. This is by design so that URLs are shareable across locales.

To make the ASP.NET Core route value provider and query string value provider undergo a culture-sensitive conversion:

- Inherit from [IValueProviderFactory](#)
- Copy the code from [QueryStringValueProviderFactory](#) or [RouteValueProviderFactory](#)
- Replace the [culture value](#) passed to the value provider constructor with [CultureInfo.CurrentCulture](#)
- Replace the default value provider factory in MVC options with your new one:

C#



```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews(options =>
    {
        var index = options.ValueProviderFactories.IndexOf(
            options.ValueProviderFactories.OfType<QueryStringValueProviderFactory>().Single());
        options.ValueProviderFactories[index] = new CulturedQueryStringValueProviderFactory();
    });
}
```

C#



```
public class CulturedQueryStringValueProviderFactory : IValueProviderFactory
{
    public Task CreateValueProviderAsync(ValueProviderFactoryContext context)
    {
        if (context == null)
        {
            throw new ArgumentNullException(nameof(context));
        }

        var query = context.ActionContext.HttpContext.Request.Query;
        if (query != null && query.Count > 0)
        {
            var valueProvider = new QueryStringValueProvider(
                BindingSource.Query,
```

```
        query,
        CultureInfo.CurrentCulture);

    context.ValueProviders.Add(valueProvider);
}

return Task.CompletedTask;
}
```

Special data types

There are some special data types that model binding can handle.

IFormFile and IFormFileCollection

An uploaded file included in the HTTP request. Also supported is `IEnumerable<IFormFile>` for multiple files.

CancellationToken

Used to cancel activity in asynchronous controllers.

FormCollection

Used to retrieve all the values from posted form data.

Input formatters

Data in the request body can be in JSON, XML, or some other format. To parse this data, model binding uses an *input formatter* that is configured to handle a particular content type. By default, ASP.NET Core includes JSON based input formatters for handling JSON data. You can add other formatters for other content types.

ASP.NET Core selects input formatters based on the [Consumes](#) attribute. If no attribute is present, it uses the [Content-Type header](#).

To use the built-in XML input formatters:

- Install the `Microsoft.AspNetCore.Mvc.Formatters.Xml` NuGet package.
- In `Startup.ConfigureServices`, call [AddXmlSerializerFormatters](#) or [AddXmlDataContractSerializerFormatters](#).

```
C# Copy

services.AddRazorPages()
    .AddMvcOptions(options =>
    {
        options.ValueProviderFactories.Add(new CookieValueProviderFactory());
        options.ModelMetadataDetailsProviders.Add(
            new ExcludeBindingMetadataProvider(typeof(System.Version)));
        options.ModelMetadataDetailsProviders.Add(
            new SuppressChildValidationMetadataProvider(typeof(System.Guid)));
    })
    .AddXmlSerializerFormatters();
```

- Apply the `Consumes` attribute to controller classes or action methods that should expect XML in the request body.

```
C# Copy

[HttpPost]
[Consumes("application/xml")]
public ActionResult<Pet> Create(Pet pet)
```

For more information, see [Introducing XML Serialization](#).

Customize model binding with input formatters

An input formatter takes full responsibility for reading data from the request body. To customize this process, configure the APIs used by the input formatter. This section describes how to customize the `System.Text.Json`-based input formatter to understand a custom type named `ObjectId`.

Consider the following model, which contains a custom `ObjectId` property named `Id`:

C#

 Copy

```
public class ModelWithObjectId
{
    public ObjectId Id { get; set; }
}
```

To customize the model binding process when using `System.Text.Json`, create a class derived from `JsonConverter<T>`:

C#

 Copy

```
internal class ObjectIdConverter : JsonConverter<ObjectId>
{
    public override ObjectId Read(
        ref Utf8JsonReader reader, Type typeToConvert, JsonSerializerOptions options)
    {
        return new ObjectId(JsonSerializer.Deserialize<int>(ref reader, options));
    }

    public override void Write(
        Utf8JsonWriter writer, ObjectId value, JsonSerializerOptions options)
    {
        writer.WriteNumberValue(value.Id);
    }
}
```

```
}  
}
```

To use a custom converter, apply the [JsonConverterAttribute](#) attribute to the type. In the following example, the `ObjectId` type is configured with `ObjectIdConverter` as its custom converter:

C#

 Copy

```
[JsonConverter(typeof(ObjectIdConverter))]  
public struct ObjectId  
{  
    public ObjectId(int id) =>  
        Id = id;  
  
    public int Id { get; }  
}
```

For more information, see [How to write custom converters](#).

Exclude specified types from model binding

The model binding and validation systems' behavior is driven by [ModelMetadata](#). You can customize `ModelMetadata` by adding a details provider to [MvcOptions.ModelMetadataDetailsProviders](#). Built-in details providers are available for disabling model binding or validation for specified types.

To disable model binding on all models of a specified type, add an [ExcludeBindingMetadataProvider](#) in `Startup.ConfigureServices`. For example, to disable model binding on all models of type `System.Version`:

C#

 Copy

```
services.AddRazorPages()  
    .AddMvcOptions(options =>
```



```
{
    options.ValueProviderFactories.Add(new CookieValueProviderFactory());
    options.ModelMetadataDetailsProviders.Add(
        new ExcludeBindingMetadataProvider(typeof(System.Version)));
    options.ModelMetadataDetailsProviders.Add(
        new SuppressChildValidationMetadataProvider(typeof(System.Guid)));
})
.AddXmlSerializerFormatters();
```

To disable validation on properties of a specified type, add a [SuppressChildValidationMetadataProvider](#) in `Startup.ConfigureServices`. For example, to disable validation on properties of type `System.Guid`:

C#



```
services.AddRazorPages()
    .AddMvcOptions(options =>
    {
        options.ValueProviderFactories.Add(new CookieValueProviderFactory());
        options.ModelMetadataDetailsProviders.Add(
            new ExcludeBindingMetadataProvider(typeof(System.Version)));
        options.ModelMetadataDetailsProviders.Add(
            new SuppressChildValidationMetadataProvider(typeof(System.Guid)));
    })
    .AddXmlSerializerFormatters();
```

Custom model binders

You can extend model binding by writing a custom model binder and using the `[ModelBinder]` attribute to select it for a given target. Learn more about [custom model binding](#).

Manual model binding

Model binding can be invoked manually by using the [TryUpdateModelAsync](#) method. The method is defined on both `ControllerBase` and `PageModel` classes. Method overloads let you specify the prefix and value provider to use. The method returns `false` if model binding fails. Here's an example:

C#



```
if (await TryUpdateModelAsync<InstructorWithCollection>(
    newInstructor,
    "Instructor",
    i => i.FirstMidName, i => i.LastName, i => i.HireDate))
{
    _instructorsInMemoryStore.Add(newInstructor);
    return RedirectToPage("./Index");
}
PopulateAssignedCourseData(newInstructor);
return Page();
```

[TryUpdateModelAsync](#) uses value providers to get data from the form body, query string, and route data.

[TryUpdateModelAsync](#) is typically:

- Used with Razor Pages and MVC apps using controllers and views to prevent over-posting.
- Not used with a web API unless consumed from form data, query strings, and route data. Web API endpoints that consume JSON use [Input formatters](#) to deserialize the request body into an object.

For more information, see [TryUpdateModelAsync](#).

[FromServices] attribute

This attribute's name follows the pattern of model binding attributes that specify a data source. But it's not about binding data from a value provider. It gets an instance of a type from the [dependency injection](#) container. Its purpose is to provide an alternative to constructor injection for when you need a service only if a particular method is called.

Additional resources

- [Model validation in ASP.NET Core MVC](#)
- [Custom Model Binding in ASP.NET Core](#)

Is this page helpful?

 Yes  No
