

# Parameter Binding in ASP.NET Web API

07/11/2013 • 11 minutes to read • Contributors      all

## In this article

[Using \[FromUri\]](#)

[Using \[FromBody\]](#)

[Type Converters](#)

[Model Binders](#)

[Value Providers](#)

[HttpParameterBinding](#)

[IActionValueBinder](#)

[Additional Resources](#)


by [Mike Wasson](#)

When Web API calls a method on a controller, it must set values for the parameters, a process called *binding*. This article describes how Web API binds parameters, and how you can customize the binding process.

By default, Web API uses the following rules to bind parameters:

- If the parameter is a "simple" type, Web API tries to get the value from the URI. Simple types include the .NET [primitive types](#) (**int**, **bool**, **double**, and so forth), plus **TimeSpan**, **DateTime**, **Guid**, **decimal**, and **string**, *plus* any type with a type converter that can convert from a string. (More about type converters later.)
- For complex types, Web API tries to read the value from the message body, using a [media-type formatter](#).

For example, here is a typical Web API controller method:

|  |  |
|--|--|
| C#   |  Copy |
| <pre>HttpResponseMessage Put(int id, Product item) { ... }</pre> |  |

The *id* parameter is a "simple" type, so Web API tries to get the value from the request URI. The *item* parameter is a complex type, so Web API uses a media-type formatter to read the

value from the request body.

To get a value from the URI, Web API looks in the route data and the URI query string. The route data is populated when the routing system parses the URI and matches it to a route. For more information, see [Routing and Action Selection](#).

In the rest of this article, I'll show how you can customize the model binding process. For complex types, however, consider using media-type formatters whenever possible. A key principle of HTTP is that resources are sent in the message body, using content negotiation to specify the representation of the resource. Media-type formatters were designed for exactly this purpose.

## Using [FromUri]

To force Web API to read a complex type from the URI, add the **[FromUri]** attribute to the parameter. The following example defines a `GeoPoint` type, along with a controller method that gets the `GeoPoint` from the URI.

C#



```
public class GeoPoint
{
    public double Latitude { get; set; }
    public double Longitude { get; set; }
}

public ValuesController : ApiController
{
    public HttpResponseMessage Get([FromUri] GeoPoint location) { ... }
}
```

The client can put the Latitude and Longitude values in the query string and Web API will use them to construct a `GeoPoint`. For example:

```
http://localhost/api/values/?Latitude=47.678558&Longitude=-122.130989
```

## Using [FromBody]

To force Web API to read a simple type from the request body, add the **[FromBody]** attribute to the parameter:

C#

 Copy

```
public HttpResponseMessage Post([FromBody] string name) { ... }
```

In this example, Web API will use a media-type formatter to read the value of *name* from the request body. Here is an example client request.

console

 Copy

```
POST http://localhost:5076/api/values HTTP/1.1
User-Agent: Fiddler
Host: localhost:5076
Content-Type: application/json
Content-Length: 7
"Alice"
```

When a parameter has [FromBody], Web API uses the Content-Type header to select a formatter. In this example, the content type is "application/json" and the request body is a raw JSON string (not a JSON object).

At most one parameter is allowed to read from the message body. So this will not work:

C#

 Copy

```
// Caution: Will not work!
public HttpResponseMessage Post([FromBody] int id, [FromBody] string name) {
    ... }
```

The reason for this rule is that the request body might be stored in a non-buffered stream that can only be read once.

## Type Converters

You can make Web API treat a class as a simple type (so that Web API will try to bind it from the URI) by creating a **TypeConverter** and providing a string conversion.

The following code shows a `GeoPoint` class that represents a geographical point, plus a **TypeConverter** that converts from strings to `GeoPoint` instances. The `GeoPoint` class is decorated with a `[TypeConverter]` attribute to specify the type converter. (This example was inspired by Mike Stall's blog post [How to bind to custom objects in action signatures in MVC/WebAPI](https://mikestall.com/2013/05/20/how-to-bind-to-custom-objects-in-action-signatures-in-mvc/webapi/).)

C#



```
[TypeConverter(typeof(GeoPointConverter))]
public class GeoPoint
{
    public double Latitude { get; set; }
    public double Longitude { get; set; }

    public static bool TryParse(string s, out GeoPoint result)
    {
        result = null;

        var parts = s.Split(',');
        if (parts.Length != 2)
        {
            return false;
        }

        double latitude, longitude;
        if (double.TryParse(parts[0], out latitude) &&
            double.TryParse(parts[1], out longitude))
        {
            result = new GeoPoint() { Longitude = longitude, Latitude =
latitude };
            return true;
        }
        return false;
    }
}

class GeoPointConverter : TypeConverter
{
    public override bool CanConvertFrom(ITypeDescriptorContext context, Type
sourceType)
    {
        if (sourceType == typeof(string))
        {
            return true;
        }
        return base.CanConvertFrom(context, sourceType);
    }

    public override object ConvertFrom(ITypeDescriptorContext context,
CultureInfo culture, object value)
    {
        if (value is string)
        {
            GeoPoint point;
            if (GeoPoint.TryParse((string)value, out point))
            {
                return point;
            }
        }
    }
}
```

```
    }  
    }  
    return base.ConvertFrom(context, culture, value);  
}  
}
```

Now Web API will treat `GeoPoint` as a simple type, meaning it will try to bind `GeoPoint` parameters from the URI. You don't need to include `[FromUri]` on the parameter.

C#

 Copy

```
public HttpResponseMessage Get(GeoPoint location) { ... }
```

The client can invoke the method with a URI like this:

```
http://localhost/api/values/?location=47.678558,-122.130989
```

## Model Binders

A more flexible option than a type converter is to create a custom model binder. With a model binder, you have access to things like the HTTP request, the action description, and the raw values from the route data.

To create a model binder, implement the **IMoelBinder** interface. This interface defines a single method, **BindModel**:

C#

 Copy

```
bool BindModel(HttpContext actionContext, ModelBindingContext  
bindingContext);
```

Here is a model binder for `GeoPoint` objects.

C#

 Copy

```
public class GeoPointModelBinder : IMoelBinder  
{  
    // List of known locations.  
    private static ConcurrentDictionary<string, GeoPoint> _locations  
        = new ConcurrentDictionary<string, GeoPoint>  
        (StringComparer.OrdinalIgnoreCase);  
  
    static GeoPointModelBinder()  
    {  
        // ...  
    }  
}
```

```
{
    _locations["redmond"] = new GeoPoint() { Latitude = 47.67856, Longitude
= -122.131 };
    _locations["paris"] = new GeoPoint() { Latitude = 48.856930, Longitude
= 2.3412 };
    _locations["tokyo"] = new GeoPoint() { Latitude = 35.683208, Longitude
= 139.80894 };
}

public bool BindModel(HttpContext actionContext, ModelBindingContext
bindingContext)
{
    if (bindingContext.ModelType != typeof(GeoPoint))
    {
        return false;
    }

    ValueProviderResult val = bindingContext.ValueProvider.GetValue(
        bindingContext.ModelName);
    if (val == null)
    {
        return false;
    }

    string key = val.RawValue as string;
    if (key == null)
    {
        bindingContext.ModelState.AddModelError(
            bindingContext.ModelName, "Wrong value type");
        return false;
    }

    GeoPoint result;
    if (_locations.TryGetValue(key, out result) || GeoPoint.TryParse(key,
out result))
    {
        bindingContext.Model = result;
        return true;
    }

    bindingContext.ModelState.AddModelError(
        bindingContext.ModelName, "Cannot convert value to GeoPoint");
    return false;
}
}
```

A model binder gets raw input values from a *value provider*. This design separates two distinct functions:

- The value provider takes the HTTP request and populates a dictionary of key-value pairs.
- The model binder uses this dictionary to populate the model.

The default value provider in Web API gets values from the route data and the query string. For example, if the URI is `http://localhost/api/values/1?location=48,-122`, the value provider creates the following key-value pairs:

- `id = "1"`
- `location = "48,-122"`


(I'm assuming the default route template, which is `"api/{controller}/{id}"`.)

The name of the parameter to bind is stored in the **ModelBindingContext.ModelName** property. The model binder looks for a key with this value in the dictionary. If the value exists and can be converted into a `GeoPoint`, the model binder assigns the bound value to the **ModelBindingContext.Model** property.


Notice that the model binder is not limited to a simple type conversion. In this example, the model binder first looks in a table of known locations, and if that fails, it uses type conversion.

## Setting the Model Binder

There are several ways to set a model binder. First, you can add a **[ModelBinder]** attribute to the parameter.

|   |   |
|---|---|
| C#  |  |
| <pre>public HttpResponseMessage Get([ModelBinder(typeof(GeoPointModelBinder))]     GeoPoint location)</pre> |   |

You can also add a **[ModelBinder]** attribute to the type. Web API will use the specified model binder for all parameters of that type.

|   |   |
|---|---|
| C#  |  |
| <pre>[ModelBinder(typeof(GeoPointModelBinder))] public class GeoPoint {     // .... }</pre> |   |

Finally, you can add a model-binder provider to the **HttpConfiguration**. A model-binder provider is simply a factory class that creates a model binder. You can create a provider by deriving from the [ModelBinderProvider](#) class. However, if your model binder handles a single type, it's easier to use the built-in **SimpleModelBinderProvider**, which is designed for this purpose. The following code shows how to do this.

C#

 Copy

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        var provider = new SimpleModelBinderProvider(
            typeof(GeoPoint), new GeoPointModelBinder());
        config.Services.Insert(typeof(ModelBinderProvider), 0, provider);

        // ...
    }
}
```

With a model-binding provider, you still need to add the **[ModelBinder]** attribute to the parameter, to tell Web API that it should use a model binder and not a media-type formatter. But now you don't need to specify the type of model binder in the attribute:

C#

 Copy

```
public HttpResponseMessage Get([ModelBinder] GeoPoint location) { ... }
```

## Value Providers

I mentioned that a model binder gets values from a value provider. To write a custom value provider, implement the **IValueProvider** interface. Here is an example that pulls values from the cookies in the request:

C#

 Copy

```
public class CookieValueProvider : IValueProvider
{
    private Dictionary<string, string> _values;

    public CookieValueProvider(HttpContext actionContext)
    {
        if (actionContext == null)
```



```
{
    throw new ArgumentNullException("actionContext");
}

_values = new Dictionary<string, string>
(StringComparer.OrdinalIgnoreCase);
foreach (var cookie in actionContext.Request.Headers.GetCookies())
{
    foreach (CookieState state in cookie.Cookies)
    {
        _values[state.Name] = state.Value;
    }
}

public bool ContainsPrefix(string prefix)
{
    return _values.Keys.Contains(prefix);
}

public ValueProviderResult GetValue(string key)
{
    string value;
    if (_values.TryGetValue(key, out value))
    {
        return new ValueProviderResult(value, value,
CultureInfo.InvariantCulture);
    }
    return null;
}
}
```

You also need to create a value provider factory by deriving from the **ValueProviderFactory** class.

C#

 Copy

```
public class CookieValueProviderFactory : ValueProviderFactory
{
    public override IValueProvider GetValueProvider(HttpContext
actionContext)
    {
        return new CookieValueProvider(actionContext);
    }
}
```

Add the value provider factory to the **HttpConfiguration** as follows.

C#

 Copy

```
public static void Register(HttpConfiguration config)
{
    config.Services.Add(typeof(ValueProviderFactory), new
    CookieValueProviderFactory());

    // ...
}
```

Web API composes all of the value providers, so when a model binder calls **ValueProvider.GetValue**, the model binder receives the value from the first value provider that is able to produce it.

Alternatively, you can set the value provider factory at the parameter level by using the **ValueProvider** attribute, as follows:

C#

 Copy

```
public HttpResponseMessage Get(
    [ValueProvider(typeof(CookieValueProviderFactory))] GeoPoint location)
```

This tells Web API to use model binding with the specified value provider factory, and not to use any of the other registered value providers.

## HttpParameterBinding

Model binders are a specific instance of a more general mechanism. If you look at the **[ModelBinder]** attribute, you will see that it derives from the abstract **ParameterBindingAttribute** class. This class defines a single method, **GetBinding**, which returns an **HttpParameterBinding** object:

C#

 Copy

```
public abstract class ParameterBindingAttribute : Attribute
{
    public abstract HttpParameterBinding GetBinding(HttpParameterDescriptor
    parameter);
}
```

An **HttpParameterBinding** is responsible for binding a parameter to a value. In the case of **[ModelBinder]**, the attribute returns an **HttpParameterBinding** implementation that uses

an **IModelBinder** to perform the actual binding. You can also implement your own **HttpParameterBinding**.

For example, suppose you want to get ETags from `if-match` and `if-none-match` headers in the request. We'll start by defining a class to represent ETags.

C#

 Copy

```
public class ETag
{
    public string Tag { get; set; }
}
```

We'll also define an enumeration to indicate whether to get the ETag from the `if-match` header or the `if-none-match` header.

C#

 Copy

```
public enum ETagMatch
{
    IfMatch,
    IfNoneMatch
}
```

Here is an **HttpParameterBinding** that gets the ETag from the desired header and binds it to a parameter of type ETag:

C#

 Copy

```
public class ETagParameterBinding : HttpParameterBinding
{
    ETagMatch _match;

    public ETagParameterBinding(HttpParameterDescriptor parameter, ETagMatch match)
        : base(parameter)
    {
        _match = match;
    }

    public override Task ExecuteBindingAsync(ModelMetadataProvider metadataProvider,
        HttpContext actionContext, CancellationToken cancellationToken)
    {
        EntityTagHeaderValue etagHeader = null;
        switch (_match)
        {
```

```
{
    case ETagMatch.IfNoneMatch:
        etagHeader =
actionContext.Request.Headers.IfNoneMatch.FirstOrDefault();
        break;

    case ETagMatch.IfMatch:
        etagHeader =
actionContext.Request.Headers.IfMatch.FirstOrDefault();
        break;
}

ETag etag = null;
if (etagHeader != null)
{
    etag = new ETag { Tag = etagHeader.Tag };
}
actionContext.ActionArguments[Descriptor.ParameterName] = etag;

var tsc = new TaskCompletionSource<object>();
tsc.SetResult(null);
return tsc.Task;
}
```

The **ExecuteBindingAsync** method does the binding. Within this method, add the bound parameter value to the **ActionArgument** dictionary in the **HttpContext**.

#### ❗ Note

If your **ExecuteBindingAsync** method reads the body of the request message, override the **WillReadBody** property to return true. The request body might be an unbuffered stream that can only be read once, so Web API enforces a rule that at most one binding can read the message body.

To apply a custom **HttpParameterBinding**, you can define an attribute that derives from **ParameterBindingAttribute**. For `ETagParameterBinding`, we'll define two attributes, one for `if-match` headers and one for `if-none-match` headers. Both derive from an abstract base class.

C#

 Copy

```
public abstract class ETagMatchAttribute : ParameterBindingAttribute
{
    private ETagMatch _match;
```

```
public ETagMatchAttribute(ETagMatch match)
{
    _match = match;
}

public override HttpParameterBinding GetBinding(HttpParameterDescriptor
parameter)
{
    if (parameter.ParameterType == typeof(ETag))
    {
        return new ETagParameterBinding(parameter, _match);
    }
    return parameter.BindAsError("Wrong parameter type");
}

public class IfMatchAttribute : ETagMatchAttribute
{
    public IfMatchAttribute()
        : base(ETagMatch.IfMatch)
    {
    }
}

public class IfNoneMatchAttribute : ETagMatchAttribute
{
    public IfNoneMatchAttribute()
        : base(ETagMatch.IfNoneMatch)
    {
    }
}
```

Here is a controller method that uses the `[IfNoneMatch]` attribute.

C#

 Copy

```
public HttpResponseMessage Get([IfNoneMatch] ETag etag) { ... }
```

Besides **ParameterBindingAttribute**, there is another hook for adding a custom **HttpParameterBinding**. On the **HttpConfiguration** object, the **ParameterBindingRules** property is a collection of anonymous functions of type (**HttpParameterDescriptor** -> **HttpParameterBinding**). For example, you could add a rule that any ETag parameter on a GET method uses `ETagParameterBinding` with `if-none-match`:

C#

 Copy

```
config.ParameterBindingRules.Add(p =>
{
    if (p.ParameterType == typeof(ETag) &&
        p.ActionDescriptor.SupportedHttpMethods.Contains(HttpMethod.Get))
    {
        return new ETagParameterBinding(p, ETagMatch.IfNoneMatch);
    }
    else
    {
        return null;
    }
});
```

The function should return `null` for parameters where the binding is not applicable.

## IActionValueBinder

The entire parameter-binding process is controlled by a pluggable service, **IActionValueBinder**. The default implementation of **IActionValueBinder** does the following:

1. Look for a **ParameterBindingAttribute** on the parameter. This includes **[FromBody]**, **[FromUri]**, and **[ModelBinder]**, or custom attributes.
2. Otherwise, look in **HttpConfiguration.ParameterBindingRules** for a function that returns a non-null **HttpParameterBinding**.
3. Otherwise, use the default rules that I described previously.
  - If the parameter type is "simple" or has a type converter, bind from the URI. This is equivalent to putting the **[FromUri]** attribute on the parameter.
  - Otherwise, try to read the parameter from the message body. This is equivalent to putting **[FromBody]** on the parameter.

If you wanted, you could replace the entire **IActionValueBinder** service with a custom implementation.

## Additional Resources

[Custom Parameter Binding Sample](#)

Mike Stall wrote a good series of blog posts about Web API parameter binding:

- [How Web API does Parameter Binding](#)
- [MVC Style parameter binding for Web API](#)
- [How to bind to custom objects in action signatures in MVC/Web API](#)
- [How to create a custom value provider in Web API](#)
- [Web API Parameter binding under the hood](#)