

08/10/2015 • 19 minutes to read

In this article

[ASP.NET MVC - The Features and Foibles of ASP.NET MVC Model Binding](#)

[Model Binding Basics](#)

[Binding to Complex Objects](#)

[Decomposing Model Binding](#)

[Value Providers](#)

[Custom Value Providers](#)

[Default Model Binder](#)

[Recursive Model Binding](#)

[Where Model Binding Seems to Fall Down](#)

[Model Binder Selection](#)

[Adorning Models with Custom Attributes](#)

[Ask the Binders!](#)

[Key Extensibility Points](#)

February 2012

Volume 27 Number 02

ASP.NET MVC - The Features and Foibles of ASP.NET MVC Model Binding

By [Jess Chadwick](#) | February 2012

ASP.NET MVC model binding simplifies controller actions by introducing an abstraction layer that automatically populates controller action parameters, taking care of the mundane property mapping and type conversion code typically involved in working with ASP.NET request data. Though model binding seems simple, it's actually a relatively complex framework composed of a number of parts that work together to create and populate the objects that your controller actions require.

This article will take you deep into the heart of the ASP.NET MVC model binding subsystem, showing each layer of the model binding framework and the various ways you can extend the model binding logic to meet your application's needs. Along the way, you'll see a few frequently overlooked model binding techniques as well as how to avoid some of the most common model binding mistakes.

Model Binding Basics


To understand what model binding is, first take a look at a typical way to populate an object from request values in an ASP.NET application, shown in **Figure 1**.

Figure 1 Retrieving Values Directly from the Request

XML	 Copy
<pre>public ActionResult Create() { var product = new Product() { AvailabilityDate = DateTime.Parse(Request["availabilityDate"]), CategoryId = Int32.Parse(Request["categoryId"]), Description = Request["description"], Kind = (ProductKind)Enum.Parse(typeof(ProductKind), Request["kind"]), Name = Request["name"], UnitPrice = Decimal.Parse(Request["unitPrice"]), UnitsInStock = Int32.Parse(Request["unitsInStock"]), }; // ... }</pre>	

Then compare the action in **Figure 1** with **Figure 2**, which leverages model binding to produce the same result.

Figure 2 Model Binding to Primitive Values


XML	 Copy
<pre>public ActionResult Create(DateTime availabilityDate, int categoryId, string description, ProductKind kind, string name, decimal unitPrice, int unitsInStock) { var product = new Product() { AvailabilityDate = availabilityDate, CategoryId = categoryId, Description = description, Kind = kind, Name = name, UnitPrice = unitPrice, UnitsInStock = unitsInStock, }; // ... }</pre>	

Though the two examples both achieve the same thing—a populated `Product` instance—the code in **Figure 2** relies on ASP.NET MVC to convert the values from the request into strongly typed values. With model binding, controller actions can be focused on providing business value and avoid wasting time with mundane request mapping and parsing.

Binding to Complex Objects

Although model binding to even simple, primitive types can make a pretty big impact, many controller actions rely on more than just a couple of parameters. Luckily, ASP.NET MVC handles complex types just as well as primitive types.

The following code takes one more pass at the Create action, skipping the primitive values and binding directly to the Product class:

XML	 Copy
<pre>public ActionResult Create(Product product) { // ... }</pre>	

Once again, this code produces the same result as the actions in **Figure 1** and **Figure 2**, only this time no code was involved at all—the complex ASP.NET MVC model binding eliminated all of the boilerplate code required to create and populate a new Product instance. This code exemplifies the true power of model binding.

Decomposing Model Binding

Now that you've seen model binding in action, it's time to break down the pieces that make up the model binding framework.

Model binding is broken down into two distinct steps: collecting values from the request and populating models with those values. These steps are accomplished by value providers and model binders, respectively.

Value Providers

ASP.NET MVC includes value provider implementations that cover the most common sources of request values such as querystring parameters, form fields and route data. At run time, ASP.NET MVC uses the value providers registered in the ValueProviderFactories class to evaluate request values that the model binders can use.

By default, the value provider collection evaluates values from the various sources in the following order:

1. Previously bound action parameters, when the action is a child action

2. Form fields (Request.Form)
3. The property values in the JSON Request body (Request.InputStream), but only when the request is an AJAX request
4. Route data (RouteData.Values)
5. Querystring parameters (Request.QueryString)
6. Posted files (Request.Files)

The value providers collection, like the Request object, is really just a glorified dictionary—an abstraction layer of key/value pairs that model binders can use without needing to know where the data came from. However, the value provider framework takes this abstraction a step further than the Request dictionary, giving you complete control over how and where the model binding framework gets its data. You can even create your own custom value providers.

Custom Value Providers

The minimum requirement to create a custom value provider is pretty straightforward: Create a new class that implements the `System.Web.Mvc.ValueProviderFactory` interface.

For example, **Figure 3** demonstrates a custom value provider that retrieves values from the user's cookies.

Figure 3 Custom Value Provider Factory that Inspects Cookie Values

XML

 Copy

```
public class CookieValueProviderFactory : ValueProviderFactory
{
    public override IValueProvider GetValueProvider
    (
        ControllerContext controllerContext
    )
    {
        var cookies = controllerContext.HttpContext.Request.Cookies;

        var cookieValues = new NameValueCollection();
        foreach (var key in cookies.AllKeys)
```

```
{  
    cookieValues.Add(key, cookies[key].Value);  
}  
  
return new NameValueCollectionValueProvider(  
    cookieValues, CultureInfo.CurrentCulture);  
}  
}
```

Notice how simple the `CookieValueProviderFactory` is. Instead of building a brand-new value provider from the ground up, the `CookieValueProviderFactory` simply retrieves the user's cookies and leverages the `NameValueCollectionValueProvider` to expose those values to the model binding framework.

After you've created a custom value provider, you'll need to add it to the list of value providers via the `ValueProviderFactories.Factories` collection:

XML

 Copy

```
var factory = new CookieValueProviderFactory();  
ValueProviderFactories.Factories.Add(factory);
```

It's pretty easy to create custom value providers, but be cautious in doing so. The set of value providers that ASP.NET MVC ships out of the box exposes most of the data available in the `HttpRequest` (with the exception of cookies, perhaps) pretty well and generally provides enough data to satisfy most scenarios.

To determine whether creating a new value provider is the right thing to do for your particular scenario, ask the following question: Does the set of information provided by the existing value providers contain all the data I need (albeit perhaps not in the proper format)?

If the answer is no, then adding a custom value provider is probably the right way to address the void. However, when the answer is yes—as it usually is—consider how you can fill in the missing pieces by customizing the model binding behavior to access the data being provided by the value providers. The rest of this article shows you how to do just that.

The main component of the ASP.NET MVC model binding framework responsible for creating and populating models using values provided by value providers is called the model binder.

Default Model Binder

The ASP.NET MVC framework includes default model binder implementation named the `DefaultModelBinder`, which is designed to effectively bind most model types. It does this by using relatively simple and recursive logic for each property of the target model:

1. Examine the value providers to see if the property was discovered as a simple type or a complex type by checking to see if the property name is registered as a prefix. Prefixes are simply the HTML form field name "dot notation" used to represent whether a value is a property of a complex object. The prefix pattern is `[ParentProperty].[Property]`. For example, the form field with the name `UnitPrice.Amount` contains the value for the `Amount` field of the `UnitPrice` property.
2. Get the `ValueProviderResult` from the registered value providers for the property's name.
3. If the value is a simple type, try to convert it to the target type. The default conversion logic leverages the property's `TypeConverter` to convert from the source value of type string to the target type.
4. Otherwise, the property is a complex type, so perform a recursive binding.

Recursive Model Binding

Recursive model binding effectively starts the whole model binding process over again but uses the name of the target property as the new prefix. Using this approach, the `DefaultModelBinder` is able to traverse entire complex object graphs and populate even deeply nested property values.

To see recursive binding in action, change `Product.UnitPrice` from a simple decimal type to the custom type `Currency`. **Figure 4** shows both classes.

Figure 4 Product Class with Complex Unitprice Property

XML

 Copy

```
public class Product
{
    public DateTime AvailabilityDate { get; set; }
    public int CategoryId { get; set; }
    public string Description { get; set; }
    public ProductKind Kind { get; set; }
    public string Name { get; set; }
    public Currency UnitPrice { get; set; }
    public int UnitsInStock { get; set; }
}

public class Currency
{
    public float Amount { get; set; }
    public string Code { get; set; }
}
```

With this update in place, the model binder will look for the values named `UnitPrice.Amount` and `UnitPrice.Code` to populate the complex `Product.UnitPrice` property.

The `DefaultModelBinder` recursive binding logic can effectively populate even the most complex object graphs. So far, you've seen a complex object that resided only one level deep in the object hierarchy, which the `DefaultModelBinder` handled with ease. To demonstrate the true power of recursive model binding, add a new property named `Child` to `Product` with the same type, `Product`:

XML

 Copy

```
public class Product {
    public Product Child { get; set; }
    // ...
}
```


Then, add a new field to the form and—applying the dot notation to indicate each level—create as many levels as you’d like. For example:

XML	 Copy
<pre><input type="text" name="Child.Child.Child.Child.Child.Child.Name"/></pre>	

This form field will result in six levels of Products! For each level, the `DefaultModelBinder` will dutifully create a new `Product` instance and dive right into binding its values. When the binder is all done, it will have created an object graph that looks like the code in **Figure 5**.

Figure 5 An Object Graph Created from Recursive Model Binding

XML	 Copy
<pre>new Product { Child = new Product { Child = new Product { Child = new Product { Child = new Product { Child = new Product { Child = new Product { Name = "MADNESS!" } } } } } } }</pre>	

Even though this contrived example sets the value of just a single property, it’s a great demonstration on how the `DefaultModelBinder` recursive model binding functionality allows it to support some very complex object graphs right out of


the box. With recursive model binding, if you can create a form field name to represent the value to populate, it doesn't matter where in the object hierarchy that value lives—the model binder will find it and bind it.

Where Model Binding Seems to Fall Down


It's true: There are some models that the `DefaultModelBinder` simply won't be able to bind. However, there are also quite a few scenarios in which the default model binding logic may not seem to work but in fact works just fine as long as you use it appropriately.

Following are a few of the most common scenarios that developers often assume the `DefaultModelBinder` can't handle and how you can implement them using the `DefaultModelBinder` and nothing else.

Complex Collections The out-of-the-box ASP.NET MVC value providers treat all request field names as if they're form post values. Take, for example, a collection of primitive values in a form post, in which each value requires its own unique index (whitespace added for readability):

XML	 Copy
<pre>MyCollection[0]=one & MyCollection[1]=two & MyCollection[2]=three</pre>	

The same approach can also be applied to collections of complex objects. To demonstrate this, update the `Product` class to support multiple currencies by changing the `UnitPrice` property to a collection of `Currency` objects:

XML	 Copy
<pre>public class Product : IProduct { public IEnumerable<Currency> UnitPrice { get; set; }</pre>	

```
// ...  
}
```

With this change, the following request parameters are required to populate the updated UnitPrice property:

XML

 Copy

```
UnitPrice[0].Code=USD &  
UnitPrice[0].Amount=100.00 &  
  
UnitPrice[1].Code=EUR &  
UnitPrice[1].Amount=73.64
```

Pay close attention to the naming syntax of the request parameters required to bind collections of complex objects. Notice the indexers used to identify each unique item in the area, and that each property for each instance must contain the full, indexed reference to that instance. Just keep in mind that the model binder expects property names to follow the form post naming syntax, regardless of whether the request is a GET or a POST.

Though it's somewhat counterintuitive, JSON requests have the same requirements—they, too, must adhere to the form post naming syntax. Take, for example, the JSON payload for the previous UnitPrice collection. The pure JSON array syntax for this data would be represented as:

```
[  
  { "Code": "USD", "Amount": 100.00 },  
  { "Code": "EUR", "Amount": 73.64 }  
]
```

 Copy

However, the default value providers and model binders require the data to be represented as a JSON form post:

 Copy

```
{
  "UnitPrice[0].Code": "USD",
  "UnitPrice[0].Amount": 100.00,

  "UnitPrice[1].Code": "EUR",
  "UnitPrice[1].Amount": 73.64
}
```

The complex object collection scenario is perhaps one of the most widely problematic scenarios that developers run into because the syntax isn't necessarily evident to all developers. However, once you learn the relatively simple syntax for posting complex collections, these scenarios become much easier to deal with.

Generic Custom Model Binders Though the `DefaultModelBinder` is powerful enough to handle almost anything you throw at it, there are times when it just doesn't do what you need. When these scenarios occur, many developers jump at the chance to take advantage of the model binding framework's extensibility model and build their own custom model binder.

For example, even though the Microsoft .NET Framework provides excellent support for object-oriented principles, the `DefaultModelBinder` offers no support for binding to abstract base classes and interfaces. To demonstrate this shortcoming, refactor the `Product` class so that it derives from an interface—named `IProduct`—that consists of read-only properties. Likewise, update the `Create` controller action so that it accepts the new `IProduct` interface instead of the concrete `Product` implementation, as shown in **Figure 6**.

Figure 6 Binding to an Interface

XML

 Copy

```
public interface IProduct
{
    DateTime AvailabilityDate { get; }
    int CategoryId { get; }
    string Description { get; }
    ProductKind Kind { get; }
    string Name { get; }
    decimal UnitPrice { get; }
```

```
int UnitsInStock { get; }  
}  
  
public ActionResult Create(IProduct product)  
{  
    // ...  
}
```

The updated Create action shown in **Figure 6**—while perfectly legitimate C# code—causes the `DefaultModelBinder` to throw the exception: “Cannot create an instance of an interface.” It’s quite understandable that the model binder throws this exception, considering that `DefaultModelBinder` has no way of knowing what concrete type of `IProduct` to create.

The simplest way to solve this issue is to create a custom model binder that implements the `IModelBinder` interface. **Figure 7** shows `ProductModelBinder`, a custom model binder that knows how to create and bind an instance of the `IProduct` interface.

Figure 7 `ProductModelBinder`—a Tightly Coupled Custom Model Binder

XML

 Copy

```
public class ProductModelBinder : IModelBinder  
{  
    public object BindModel  
    (  
        ControllerContext controllerContext,  
        ModelBindingContext bindingContext  
    )  
    {  
        var product = new Product() {  
            Description = GetValue(bindingContext, "Description"),  
            Name = GetValue(bindingContext, "Name"),  
        };  
  
        string availabilityDateValue =  
            GetValue(bindingContext, "AvailabilityDate");  
  
        if(availabilityDateValue != null)
```

```
{
    DateTime date;
    if (DateTime.TryParse(availabilityDateValue, out date))
        product.AvailabilityDate = date;
}

string categoryIdValue =
    GetValue(bindingContext, "CategoryId");

if (categoryIdValue != null)
{
    int categoryId;
    if (Int32.TryParse(categoryIdValue, out categoryId))
        product.CategoryId = categoryId;
}

// Repeat custom binding code for every property
// ...

return product;
}

private string GetValue(
    ModelBindingContext bindingContext, string key)
{
    var result = bindingContext.ValueProvider.GetValue(key);
    return (result == null) ? null : result.AttemptedValue;
}
}
```

The downside to creating custom model binders that implement the `IModelBinder` interface directly is that they often duplicate much of the `DefaultModelBinder` just to modify a few areas of logic. It's also common for these custom binders to focus on specific model classes, creating a tight coupling between the framework and the business layer and limiting reuse to support other model types.

To avoid all these issues in your custom model binders, consider deriving from `DefaultModelBinder` and overriding specific behaviors to suit your needs. This approach often provides the best of both worlds.

Abstract Model Binder The only problem with trying to apply model binding to an interface with the `DefaultModelBinder` is that it doesn't know how to determine the concrete model type. Consider the higher-level goal: the ability to develop controller actions against a non-concrete type and dynamically determine the concrete type for each request.

By deriving from `DefaultModelBinder` and overriding only the logic that determines the target model type, you can not only address the specific `IProduct` scenario, but also actually create a general-purpose model binder that can handle most other interface hierarchies as well. **Figure 8** shows an example of a general-purpose model abstract model binder.

Figure 8 A General-Purpose Abstract Model Binder

XML

 Copy

```
public class AbstractModelBinder : DefaultModelBinder
{
    private readonly string _typeNameKey;

    public AbstractModelBinder(string typeNameKey = null)
    {
        _typeNameKey = typeNameKey ?? "__type__";
    }

    public override object BindModel
    (
        ControllerContext controllerContext,
        ModelBindingContext bindingContext
    )
    {
        var providerResult =
            bindingContext.ValueProvider.GetValue(_typeNameKey);

        if (providerResult != null)
        {
            var modelName = providerResult.AttemptedValue;
```

```
var modelType =
    BuildManager.GetReferencedAssemblies()
        .Cast<Assembly>()
        .SelectMany(x => x.GetExportedTypes())
        .Where(type => !type.IsInterface)
        .Where(type => !type.IsAbstract)
        .Where(bindingContext.ModelType.IsAssignableFrom)
        .FirstOrDefault(type =>
            string.Equals(type.Name, modelName,
                StringComparison.OrdinalIgnoreCase));

if (modelType != null)
{
    var metaData =
        ModelMetadataProviders.Current
            .GetMetadataForType(null, modelType);

    bindingContext.ModelMetadata = metaData;
}

// Fall back to default model binding behavior
return base.BindModel(controllerContext, bindingContext);
}
```

To support model binding to an interface, the model binder must first translate the interface into a concrete type. To accomplish this, `AbstractModelBinder` requests the “__type__” key from the request’s value providers. Leveraging value providers for this kind of data provides flexibility as far as where the “__type__” value is defined. For example, the key could be defined as part of the route (in the route data), specified as a querystring parameter or even represented as a field in the form post data.

Next, the `AbstractModelBinder` uses the concrete type name to generate a new set of metadata that describes the details of the concrete class. `AbstractModelBinder` uses this new metadata to replace the existing `ModelMetadata` property that

described the initial abstract model type, effectively causing the model binder to forget that it was ever bound to a non-concrete type to begin with.

After `AbstractModelBinder` replaces the model metadata with all the information needed to bind to the proper model, it simply releases control back to the base `DefaultModelBinder` logic to let it handle the rest of the work.


The `AbstractModelBinder` is an excellent example that shows how you can extend the default binding logic with your own custom logic without reinventing the wheel, by deriving directly from the `IModelBinder` interface.

Model Binder Selection


Creating custom model binders is just the first step. To apply custom model binding logic in your application, you must also register the custom model binders. Most tutorials show you two ways to register custom model binders.

The Global ModelBinders Collection The generally recommended way to override the model binder for specific types is to register a type-to-binder mapping to the `ModelBinders.Binders` dictionary.

The following code snippet tells the framework to use the `AbstractModelBinder` to bind `Currency` models:

XML	 Copy
<pre>ModelBinders.Binders.Add(typeof(Currency), new AbstractModelBinder());</pre>	

Overriding the Default Model Binder Alternatively, to replace the global default handler, you can assign a model binder to the `ModelBinders.Binders.DefaultBinder` property. For example:

XML	 Copy
<pre>ModelBinders.Binders.DefaultBinder = new AbstractModelBinder();</pre>	

Although these two approaches work well for many scenarios, there are two more ways that ASP.NET MVC lets you register a model binder for a type: attributes and providers.

Adorning Models with Custom Attributes

In addition to adding a type mapping to the ModelBinders dictionary, the ASP.NET MVC framework also offers the abstract `System.Web.Mvc.CustomModelBinderAttribute`, an attribute that allows you to dynamically create a model binder for each class or property to which the attribute is applied. **Figure 9** shows a `CustomModelBinderAttribute` implementation that creates an `AbstractModelBinder`.

Figure 9 A `CustomModelBinderAttribute` Implementation

XML

 Copy

```
[AttributeUsage(
    AttributeTargets.Class | AttributeTargets.Enum |
    AttributeTargets.Interface | AttributeTargets.Parameter |
    AttributeTargets.Struct | AttributeTargets.Property,
    AllowMultiple = false, Inherited = false
)]
public class AbstractModelBinderAttribute : CustomModelBinderAttribute
{
    public override IModelBinder GetBinder()
    {
        return new AbstractModelBinder();
    }
}
```

You can then apply the `AbstractModelBinderAttribute` to any model class or property, like so:

XML

 Copy

```
public class Product
{
    [AbstractModelBinder]
    public IEnumerable<CurrencyRequest> UnitPrice { get; set; }
    // ...
}
```

Now when the model binder attempts to locate the appropriate binder for `Product.UnitPrice`, it will discover the `AbstractModelBinderAttribute` and use the `AbstractModelBinder` to bind the `Product.UnitPrice` property.

Leveraging custom model binder attributes is a great way to achieve a more declarative approach to configuring model binders while keeping the global model binder collection simple. Also, the fact that custom model binder attributes can be applied to both entire classes and individual properties means you have fine-grain control over the model binding process.

Ask the Binders!

Model binder providers offer the ability to execute arbitrary code in real time to determine the best possible model binder for a given type. As such, they provide an excellent middle ground among explicit model binder registration for individual model types, static attribute-based registration and a set default model binder for all types.

The following code shows how to create an `IModelBinderProvider` that provides an `AbstractModelBinder` for all interfaces and abstract types:

XML

 Copy

```
public class AbstractModelBinderProvider : IModelBinderProvider
{
    public IModelBinder GetBinder(Type modelType)
    {
        if (modelType.IsAbstract || modelType.IsInterface)
            return new AbstractModelBinder();
    }
}
```

```
        return null;
    }
}
```

The logic dictating whether the `AbstractModelBinder` applies to a given model type is relatively straightforward: Is it a non-concrete type? If so, the `AbstractModelBinder` is the appropriate model binder for the type, so instantiate the model binder and return it. If the type is a concrete type, then `AbstractModelBinder` is not appropriate; return a null value to indicate that the model binder isn't a match for this type.

An important thing to keep in mind when implementing the `.GetBinder` logic is that the logic will be executed for every property that's a candidate for model binding, so be sure to keep it lightweight or you can easily introduce performance issues into your application.

In order to begin using a model binder provider, add it to the list of providers maintained in the `ModelBinderProviders.BinderProviders` collection. For example, register the `AbstractModelBinder` like so:

XML

 Copy

```
var provider = new AbstractModelBinderProvider();
ModelBinderProviders.BinderProviders.Add(provider);
```

And that easily, you've added model binding support for non-concrete types throughout your entire application.

The model binding approach makes model binding selection much more dynamic by taking the burden of determining the proper model binder away from the framework and placing it in the most appropriate place: the model binders themselves.

Key Extensibility Points

Like any other method, the ASP.NET MVC model binding allows controller actions to accept complex object types as parameters. Model binding also encourages better separation of concerns by separating the logic of populating objects from the logic that uses the populated objects.

I've explored some key extensibility points in the model binding framework that can help you leverage it to the fullest. Taking the time to understand ASP.NET MVC model binding and how to use it properly can have a large impact, even on the simplest of applications.

Jess Chadwick is an independent software consultant specializing in Web technologies. He has more than a decade of development experience ranging from embedded devices in startups to enterprise-scale Web farms at Fortune 500 companies. He's an ASPInsider, Microsoft MVP in ASP.NET, and book and magazine author. He is actively involved in the development community, regularly speaking at user groups and conferences as well as leading the NJDOTNET Central New Jersey .NET user group.

Thanks to the following technical expert for reviewing this article: **Phil Haack**

Is this page helpful?

 Yes  No
