Programming Web Applications with ASP.NET MVC 4

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Preface

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

Tip
This icon signifies a tip, suggestion, or general note.

Caution
This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Programming Razor* by Jess Chadwick. Copyright 2011 O'Reilly Media, Inc., 063-6-920-02062-2."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

Safari® Books Online

Note
Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at http://my.safaribooksonline.com.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

`http://www.oreilly.com/catalog/0636920020622/`

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, courses, conferences, and news, see our website at `http://www.oreilly.com`.

Find us on Facebook: `http://facebook.com/oreilly`

Follow us on Twitter: `http://twitter.com/oreillymedia`

Watch us on YouTube: `http://www.youtube.com/oreillymedia`

Chapter 1

Introduction

TODO

Chapter 2

CHP2

TODO

Chapter 3

CHP3

TODO

Chapter 4

CHP4

TODO

Chapter 5

CHP5

TODO

Chapter 6

Models

ASP.NET MVC's model binding functionality makes it possible for you to pass complex types as parameters to your controller actions just as you would any normal .NET method. ASP.NET MVC populates the values in these parameters for you automatically. In this article, I'll show you just how ASP.NET MVC's model binding framework works and how you can extend it to meet your own needs. I'll also walk you through the extensibility points that ASP.NET MVC provides to let you control exactly how and where the data comes from. Along the way, I'll also point out some common model binding mistakes and commonly-missed opportunities. Now, let's bind some models!

6.1 Retrieving Request values

ASP.NET MVC model binding simplifies how you retrieve values from an ASP.NET web request, making them more declarative and easier to work with. To provide you with a frame of reference, Retrieving values directly from the request shows an example of how you might be used to working with web request values.

Retrieving values directly from the request

```
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"]),
   };
   ...
}
```

The controller action in this particular example creates and populates the properties of a new `Product` object with values taken straight from the request. Since `Product`'s properties are defined as various primative, non-`string` types, the action also needs to parse each of the request values into the proper type. This example may seem simple and straight-forward, but it's actually quite frail: if any of the parsing attempts fails, the entire action will fail. Sure, you can use the various `TryParse()` methods to avoid most exceptions, however that just means you need to write even more code.

The side-effect of this approach is that every action is very explicit. The downside to writing such explicit code is that it puts the burden on you, the developer, to perform all the work; and remember to perform this work every time it is required. A larger amount of code also tends to obscure the real goal: in this example, adding a new `Product` to the system.

6.1.1 Model Binding Basics

Out of the box ASP.NET MVC provides you with the ability to declaratively retrieve values from various aspects of the current request. ASP.NET MVC Model Binding makes interacting with values from the `Request` much easier. So easy, in fact, that you longer need to know that they are coming from the `Request`!

To ease into the concept, take a look at Retrieving values directly from the request which shows very basic model binding in action:

Model binding to primitivevalues

```
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,
   };
   ...
}
```

Instead of retrieving the values from the `Request` explicitly, the action method now declares them as parameters. Note that the parameter names are very important, as they still correspond to the keys in the `Request` dictionary. When the ASP.NET MVC framework executes this method it attempts to populate the action's parameters using the same values from the request that the previous example, Retrieving values directly from the request, showed. Now it's up to ASP.NET MVC to perform much of the mundane, boilerplate code so that the logic within the action can concentrate on providing business value. The code that is left much more meaningful, not to mention more readable.

6.1.1.1 Binding to complex objects

Applying the model binding approach even to simple, primitivetypes can make a pretty big impact in making your code more expressive. However, only the most basic scenarios rely on just a couple of parameters. Luckily, ASP.NET MVC supports binding to complex types as well as primitivetypes.

Model binding to complex objects (AKA "models") takes one more pass at the *Create* action, this time skipping the middle-man primitivetypes and binding right to a `Product` instance:

Model binding to complex objects (AKA "models")

```
public ActionResult Create(Product product)
{
   ...
}
```

The action shown in Model binding to complex objects (AKA "models") is equivalent to what you saw previously in Model binding to primitivevalues. That's right - ASP.NET MVC's complex model binding just eliminated *all* of the boilerplate code required to create and populate a new `Product` instance! Model binding to complex objects (AKA "models") shows the true power of model binding.

Now that I've shown you the basics of ASP.NET MVC model binding, let's take a look under the hood and see how it all works.

6.2 Model Binding Lifecycle

Like many things in the .NET Framework, model binding is pretty simple and straightforward to use, however it's powered by a relatively complex and extensible framework. Let's take a look at the pieces that make ASP.NET MVC model binding work, starting with where all of the data comes from.

6.2.1 Value Providers

Previous sections kept referring to the model binder retrieving values from the `Request` dictionary. The bad news is that this is not actually true: ASP.NET MVC model binding does not access the `Request` dictionary directly. The good news, however, is that it is safe for you to continue the model binder gets its values from the same parts of the request that the `Request` uses to populate its dictionary like querystring parameters and form fields. It even accesses these sources in the same order, effectively giving them the same priority which means for most scenarios you can continue to pretend that the model binder retrieving values directly from the `Request` dictionary.

The sources that ASP.NET MVC model binding uses to retrieve its data are called *Value Providers*. Out of the box, ASP.NET MVC defines Value Providers for common scenarios such as querystring parameters, form fields and route data. ASP.NET MVC registers these Value Providers via the `ValueProviderFactories` class.

The default collection evaluates values from the various sources in the following order:

1. If the action is a child action, the action's parameters
2. Form fields (i.e. `Request.Form`)
3. If the request is an AJAX request, the property values in the JSON Request body (`Request.InputStream`)
4. Route data (i.e. `RouteData.Values`)
5. Query string parameters (i.e. `Request.QueryString`)
6. Posted files (i.e. `Request.Files`)

Prior to this section I referred to the binding to values provided by the `Request` dictionary. This reference is relevant because the value provider framework, 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, by giving you complete control over how and where the model binding framework gets its data. Via the `ValueProviderFactories` class you can remove or reorder the default value providers, or even add your own custom providers by modifying the `ValueProviderFactory` instances in the `ValueProviderFactories.Factories` collection.

======= Custom Value Providers Value providers are just another one of ASP.NET MVC's extensibility points that make it pretty easy to add to the sources that contribute to the model binding framework's available values. Custom value provider factory that inspects cookie values demonstrates how you can add the user's cookie values to that list of sources.

Custom value provider factory that inspects cookie values

```
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) ↩
          ;
```

```
    }
}
```

The `CookieValueProviderFactory` is actually not a new value provider implementation at all - it simply leverages one of the built-in value providers, `NameValueCollectionValueProvider`, by copying the values of the user's cookies into a `NameValueCollection` and passing that collection to the `NameValueCollectionValueProvider`.

Use caution when considering whether to build a custom value provider. When the out-of-box model binding experience doesn't seem to support a particular scenario, many developers are eager to fill the data gap with a custom value provider. However, the majority of model binding scenarios rely on data contained in the current request and the set of value providers that ASP.NET MVC ships out of the box expose the important parts of the `HttpRequest` (with the exception of cookies, perhaps) pretty well. In other words, the out-of-box value providers provide the data you need in most scenarios so when there seems to be a gap, it's usually on the model binder end of things.

Ask the following question to help determine whether to create a new value provider: 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. I'll show you several ways to extend the default model binding behavior in the following sections.

6.2.1.1 Model Binders

Unsurprisingly, the main component of the ASP.NET MVC model binding framework is the "model binder", which orchestrates the various pieces of the model binding framework such as value providers and binding contexts to create and populate objects.

True to its extensible nature, the ASP.NET MVC framework allows developers to control which binder is used - the only requirement for a model binder to join in the fun is that it must implement the `System.Web.Mvc.IModelBinder` interface. In fact, ASP.NET MVC's model binding framework does not prescribe a specific model binding implementation. Due to its interface-based design, you have complete control over how model binding is performed in your application. To help get you started, however, the framework does include a general purpose implementation, giving you the option to create your own model binder or use the default binding logic.

======= Default Model Binder The ASP.NET MVC framework includes an `IModelBinder` 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 to be bound:

1. Check with the value provider(s) 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 of the *Amount* field of the *UnitPrice* property.
2. Get the `ValueProviderResult` from the registered value provider(s) 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 binding The recursive binding in the final step effectively starts the whole process over at step 1, only using the property name 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`. Product class with complex UnitPrice property shows both classes:

Product class with complex UnitPrice property

```
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; }
}
```

Now, instead of looking the `decimal` value named *UnitPrice*, the model binder will look for the values named *UnitPrice.Amount* and *UnitPrice.Code* to populate the complex `Product.UnitPrice` property.

The `DefaultModelBinder`'s 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 like it was child's play. To demonstrate to true power of recursiving binding, give the `Product` class a new property named "Child" with the same type, `Product`:

```
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:

```
<input type="text" name="Child.Child.Child.Child.Child.Child.Name" />
```

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 similar to the following:

```
new Product {
   Child = new Product {
      Child = new Product {
         Child = new Product {
            Child = new Product {
               Child = new Product {
                  Child = new Product { Name = "MADNESS!" }
               }
            }
         }
      }
   }
}
```

Even though this contrived example sets the value of just a single property, it serves to show that the `DefaultModelBinder` is able 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!

6.2.1.2 Binding context

6.2.1.3 Updating existing models

Up to now, the entire discussion has shown how the model binding framework can create and bind controller action parameters. While it does a great job at creating new objects against which to bind, you can also use the `Controller.TryUpdate()` method to execute the model binding logic against an existing object.

Updating an existing model shows the `Controller.TryUpdate()` method in action, updating an existing `Product` instance.

Updating an existing model

```
public ActionResult Update(int id)
{
   Product existingProduct;

   // Retrieve product from the database, etc.

   TryUpdateModel(existingProduct);

   ...
}
```

The `Controller.TryUpdateModel()` method accepts an instance of a model and optional lists of property names to include or exclude from binding. The method then calls the same model binding logic that the framework calls to bind values to controller action parameters. Other than the fact that the `Controller.TryUpdateModel()` method doesn't create a new model instance, the two approaches are actually the same.

6.2.1.4 Validation

As the `DefaultModelBinder` attempts to bind each property of an object, it also validates that that property meets the validation criteria that's been applied to it. It does this by requesting the validators for the property from the `ModelValidatorProviders.` collection and executing each validator against the populated property. When a validator fails, the `DefaultModelBinder` adds the details of that validation failure to the `ModelState` dictionary, which acts as a central container for all of the errors and messages that apply to the model.

Following standard ASP.NET MVC fashion, the model validator framework is fully extensible, however it does ship with an implementation that validates models marked up with the .NET DataAnnotations namespace. This namespace contains a slew of useful attributes that you can use to mark up your objects, declaratively applying business rules without the need for significant amounts of imperative validation code. For example, Product class with Data Annotations shows a subset of the `Product` class with some basic DataAnnotation attributes applied.

Product class with Data Annotations

```
public class Product : IProduct
{
   [Required, Range(0, 100)]
   public int CategoryId { get; set; }

   [Required, StringLength(200)]
   public string Name { get; set; }

   [Required, Range(0, int.MaxValue)]
   public int UnitsInStock { get; set; }

   ...
}
```

With these attributes in place, a validation error will be added to the `ModelState` dictionary for each property value that does not pass the validation rules specified in the DataAnnotation attributes. Thus, when `ModelState.IsValid()` returns true, I know that I can safely work with the model-bound instance. However, when `ModelState.IsValid()` returns false, however, it is time to skip whatever logic I had planned - such as saving the data to a database - and deal with the validation errors.

In most cases, "dealing with the validation errors" just means sending the user back to the original form to correct their mistakes and try the form post again. In addition to the `ModelState` object, ASP.NET MVC also offers a few helper methods that can help render validation messages in the UI to indicate which fields failed validation and why. Form displaying model validation errors shows a snippet with these helpers - `Html.ValidationSummary()` and `Html.ValidationMessageFor<T>()` - in action.

Form displaying model validation errors

```
<div class="editor-label">
   @Html.LabelFor(model => model.Name)
</div>
<div class="editor-field">
   @Html.EditorFor(model => model.Name)
   @Html.ValidationMessageFor(model => model.Name)
</div>

<div class="editor-label">
   @Html.LabelFor(model => model.UnitsInStock)
</div>
<div class="editor-field">
   @Html.EditorFor(model => model.UnitsInStock)
   @Html.ValidationMessageFor(model => model.UnitsInStock)
</div>
```

6.3 Where Model Binding Seems to Fall Down

While it's true that the `DefaultModelBinder` doesn't support every kind of model, there are quite a few scenarios that the default logic may not *seem* to work, but in fact works great if you approach it the right way. In this section, I'll show you a few of the most common scenarios that developers often assume the `DefaultModelBinder` cannot handle and how you can make them work using the `DefaultModelBinder` and nothing else. Then, I'll show a few scenarios that are truly beyond the reach of the `DefaultModelBinder` and how to extend the framework to support those scenarios as well.

6.3.1 Complex Collections

Even during GET requests the out-of-the-box ASP.NET MVC value providers treat all of the request field names as if they are form post values. Take, for example, a collection of primitivevalues in a form post, in which each value requires its own unique index (whitespace added for readability):

```
MyCollection[0]=one &
MyCollection[1]=two &
MyCollection[2]=three
```

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

Product class that supports multiple currencies

```
public class Product {
   public IEnumerable<Currency> UnitPrice { get; set; }
   ...
}
```

Then, to populate the updated `UnitPrice` property the request would need to contain the following set of parameters:

```
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 object. 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.

The complex object collection scenario is perhaps one of the most widely problematic scenarios that developers run into because the syntax is not 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.

6.3.1.1 Binding Directly to Business Models

Many model binding tutorials bind directly to "business models", that is classes that you use throughout your application that represent application concepts and data models. In fact, even the examples throughout this very article show model binding directly to the `Product` business model. The examples even apply data annotations to the `Product` class to show off the model binder’s integrated validation.

Binding directly to business models can introduce many problems, but at the core of them all is the fact that the model binding framework tries to populate and validate the model a property at a time. In order to accomplish this, the model binder expects the target object to be not much more than a set of writable fields - a simple data transfer object.

The disconnect derives from the fact that business models following object-oriented principles generally do not expose a simple set of readable and writable properties. Many times, write access to many fields is either controlled through the use of methods or disallowed completely.

Take, for example, the `Product.UnitPrice` property, which contains a list of `Currency` objects. As it stands, anyone can come along and completely change anything they like. For example:

```
var product = GetProduct(id);
product.UnitPrice[1].Code = "GBP";
```

The currency’s `Code` has been changed from Euros to British Pounds, yet the `Amount` stays the same. Certainly, 100 Euros are not equivalent to 100 British Pounds! `Currency` values are a combination of amount and currency code and the two values together should treated as read-only.

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

   public Currency(float amount, string code) {
      Amount = amount;
      Code = code;
   }
}
```

[?screen] shows the updated `Currency` class, changing both `Amount` and `Code` to read-only properties. After this update, the default model binder starts to complain that it does not know how to create and bind the `Currency` class, throwing an exception with the error: `No parameterless constructor defined for this object.`

Of course, one way around this particular scenario is to create a custom model binder that knows how to create and "bind" `Currency` models (more on that in the next section). However, the larger problem is that, as business model logic grows more and more restrictive, the default model binder is less and less able to directly leverage those models and the number of model binding issues such as this one continues to increase.

====== Request Models One of the most effective ways to avoid the complications that come along with binding directly to business models is to introduce a `Request Model`. Just as many developers like to create "view models" - custom-tailored classes that specifically address the needs of the view layer - the `Request Model` approach introduces a new model whose only purpose in life is to act as an intermediary between the model binder and the business model.

[?screen] shows what the request model for the `Currency` object might look like. And, since our examples have shown the `Currency` class in the context of a `Product`, [?screen] also includes a corresponding request model for the `Product` class.

```
public class CurrencyRequest
{
    [Required]
    public float Amount { get; set; }

    [Required]
    public string Code  { get; set; }
}

public class CreateProductRequest
{
    [Required]
    public int CategoryId { get; set; }

    [Required, StringLength(200)]
    public string Name { get; set; }

    public string Description { get; set; }

    public ProductKind Kind { get; set; }

    [Required]
    public IEnumerable<CurrencyRequest> UnitPrice { get; set; }
}
```

This first thing to notice about the request models is that all of their properties are both readable and writable, making them very accessible to model binding. Keep in mind that the primary focus for request models is to accept - not restrict - input from the user.

The next thing to notice is that request objects - like view models - are often custom-tailored for a specific scenario. In this case, the `CreateProductRequest` object contains properties and validation logic specific to the *creation* of a new `Product`, as opposed to updating or deleting an existing one. Likewise, the example shows that you can continue to apply validation logic to request models. After all, they are still "models"!

The final step after the model binder has populated and validated the request model is to map the model-bound values into the business model. This step is heavily dependant on the specifics of each scenario, but generally speaking you have a few options: write your own explicit mapping logic, use a code generation tool to generate the logic, or use one of the many great mapping frameworks such as AutoMapper.

[?screen] contains explicit mapping logic that copies values from the `CurrencyRequest` and `CreateProductRequest` request models to their corresponding business models.

```
public ActionResult Create(CreateProductRequest request)
{
   var product = new Product {
      CategoryId = request.CategoryId,
      Description = request.Description,
      Name = request.Name,
      UnitPrice = request.UnitPrice.Select(x =>
         new Currency(x.Amount, x.Code)
      )
   };

   ...
}
```

When used correctly, request models can be a great tool way to encourage even greater separation of concerns in your application, further disconnecting your UI layer from your business model.

6.4 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. Unfortunately, many of these custom model binders tend to implement model binding from scratch by deriving directly from the `IModelBinder` interface. It is 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.

For example, even though the .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, I'll refactor the `Product` class so that it derives from a interface, named `IProduct`, that consists of read-only properties. Likewise, I'll update the *Create* controller action so that it accepts the new `IProduct` interface instead of the concrete implementation, `Product`.

Binding to an interface

```
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 Binding to an interface - while perfectly legitimate C# code - causes the `DefaultModelBinder` to throw the exception: "Cannot create an instance of an interface". The fact that this exception occurs is quite understandable, considering that `DefaultModelBinder` has no way of knowing what concrete type of `IProduct` to create. Regardless, the binder diligently attempts to create an instance of `IProduct` by calling `Activator.CreateInstance(typeof(IProduct))`, but this call - of course - throws the aforementioned exception.

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

ProductModelBinder: a tightly-coupled custom model binder

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

        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;
    }
}
```

6.4.1 Abstract Model Binder

Since the new binder is a direct implementation of the `IModelBinder` interface, it is responsible for all of the model binding logic. Not only does this demand a lot of imperative assignment, the model binder has to change any time the model is updated so the model and its binder remain synchronized. It also has to duplicate much of the logic that the `DefaultModelBinder` already implements. As such, it might be a better idea to analyze how to leverage the `DefaultModelBinder` so it can handle as much work as possible and simply augment its logic where necessary.

In the scenario involving model binding to an interface, the only problem 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. If you tell `DefaultModelBinder` how to determine the proper type, you can not only address the specific `IProduct` scenario, you can actually create a general-purpose model binder that can handle most other interface hierarchies as well!

A general purpose abstract model binder

```
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 modelTypeName = providerResult.AttemptedValue;

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

The example shown in A general purpose abstract model binder requests the "*type*" key from the request's value providers. I prefer to use the value providers for this kind of data because it provides flexibility as to 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. Also note, the request field that defines the type name in this example is very important, and may possibly conflict with your model; consider what might happen if the target model also had a property named "*Type*". This example uses the "*type*" key because it is unique enough that it does not cause a conflict in this application. However, when applying this technique to your application you may need to determine an alternative approach that works for you.

Next, the `AbstractModelBinder` takes the concrete model type name and attempts to locate a reference to the type. It then uses the ASP.NET `BuildManager` class to locate the specified type, only looking at concrete types referenced in the application that derive from the non-concrete model type specified as the controller action parameter. Once again, the `BuildManager` is just a technique I am using in this example - feel free to use your favorite (and more performant) method to locate the concrete type by name.

Finally, once the `AbstractModelBinder` finds the desired concrete type, it replaces the existing `ModelMetadata` property on the *bindingContext* by generating a new set of metadata using the discovered concrete class. This effectively causes the model binder to forget that it was initially bound to a non-concrete type. Once the model metadata is updated and the trick has been played, the `DefaultModelBinder` now has all the information it needs to begin working again, so simply release control back to the base `DefaultModelBinder` logic and 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.

6.4.1.1 JSON Model Binder

JSON is a very compact and efficient way to transfer data. It's also a great lightweight alternative for posting complex collections. While ASP.NET MVC supports JSON request data via the `JsonValueProviderFactory`, it is implemented as an all-or-nothing approach. That is, the factory expects the *entire* response to include a single JSON object and does not support the ability for individual fields to deliver data in JSON format.

To demonstrate, revisit the previous complex collection request made in [?screen]:

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

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

Here is how that same request would be represented in JSON format:

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

Not only is the JSON array much cleaner, simpler, and smaller, it's also much easier to build and manage using Javascript in the browser. This client-side ease of use is particularly helpful when building a form dynamically, e.g. allowing the user to add any number of currencies to a `Product`'s `UnitPrice`. However, in order to represent the `UnitPrice` field as JSON using the `DefaultModelBinder`, you'd need to post the entire object as a JSON object. For example:

```
{
   AvailabilityDate: "2/15/2012 12:00:00 AM",
   CategoryId: 3,
   Kind: "Computers",
   Name: "19 inch Monitor",
   UnitPrice:
      [
         { "Code":"USD", "Amount":100.00 },
         { "Code":"EUR", "Amount":73.64 }
      ],
   UnitsInStock: 500
}
```

On the surface, posting JSON objects to be used for model binding looks like a great idea. However, this approach has a number of downsides. First, the client must build the entire request dynamically and this logic must know how to explicitly manage every field that must be sent down. In other words, the HTML form ceases to become a form and simply becomes a way for the Javascript logic to collect data from the user. Then, on the server side, the JSON value provider ignores all HTTP requests except those with the "Content Type" header set to "application/json", so this approach will not work for standard browser (GET) requests - only AJAX requests that contain the correct header. Finally, when the default validation logic fails for even just one field, the model binder considers the *entire object* to be invalid!

A custom JSON model binder offers an alternative to the built-in JSON value provider factory that helps reduce these downsides: a custom JSON model binder. The difference between the JSON value provider factory is that the `JsonModelBinder` allows each individual field to contain JSON, eliminating the need to send the entire request as a single JSON object. Since the model binder binds each property separately, you can mix and match which fields contain simple values and which fields support JSON data. As with most custom model binders, the JSON model binder derives from the `DefaultModelBinder` so that it can fall back to the default binding logic when fields do not contain JSON data.

A custom JSON model binder

```
public class JsonModelBinder : DefaultModelBinder
{
   public override object BindModel
      (
         ControllerContext controllerContext,
         ModelBindingContext bindingContext
      )
   {
      string json = string.Empty;

      var provider = bindingContext.ValueProvider;
      var providerValue = provider.GetValue(bindingContext.ModelName);

      if (providerValue != null)
         json = providerValue.AttemptedValue;

      // Basic expression to make sure the string starts and ends
      // with JSON object ( {} ) or array ( [] ) characters
      if (Regex.IsMatch(json, @"^(\[.*\]|{.*})$"))
      {
            return new JavaScriptSerializer()
                    .Deserialize(json, bindingContext.ModelType);
        }

        return base.BindModel(controllerContext, bindingContext);
    }
}
```

6.5 Model Binder Selection

Given ASP.NET MVC's focus on extensibility, it is probably no surprise to find out that there are quite a few ways to specify which model binder should be used for any given model. In fact, the comments in the source code for the `ModelBinderDictionary.` method literally spell out how the framework discovers the appropriate model binder for each type:

```
private IModelBinder GetBinder(Type modelType, IModelBinder fallbackBinder) {

    // Try to look up a binder for this type. We use this order of precedence:
    // 1. Binder returned from provider
    // 2. Binder registered in the global table
    // 3. Binder attribute defined on the type
    // 4. Supplied fallback binder
```

I'll tackle this list from the bottom up.

6.5.1 Replacing the default (fallback) binder

With no additional configuration, ASP.NET MVC will bind all models using the `DefaultModelBinder`. You can replace this global default handler by setting the `ModelBinders.Binders.DefaultBinder` property to a new model binder. For example:

```
protected void Application_Start()
{
    ModelBinders.Binders.DefaultBinder = new JsonModelBinder();
    ...
}
```

With this setting in place, the instance of `JsonModelBinder` will act as the new fallback binder, handling the binding of all models that haven't specified otherwise.

6.5.2 Adorning models with custom attributes

My favorite approach for specifying model binders is to use the abstract `System.Web.Mvc.CustomModelBinderAttribute` to decorate both classes and individual properties in a nicely declarative way. Though you can apply this approach to any model that you wish to bind, it is best combined with the request model approach because the model binding is the sole reason the request model exists!

In order to leverage the `CustomModelBinderAttribute` approach, you first need to create an implementation. [?screen] shows the `JsonCustomModelBinderAttribute` and how it can be applied to the `CreateProductRequest` model.

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

```
   {
      return new JsonModelBinder();
   }
}

public class CreateProductRequest
{
   ...

   [Required]
   [JsonModelBinder]
   public IEnumerable<CurrencyRequest> UnitPrice { get; set; }
}
```

Decorating `CreateProductRequest.UnitPrice` with the `JsonModelBinderAttribute` indicates that the model binder should use the `JsonModelBinder` (created with a call to `JsonModelBinderAttribute.GetBinder()) to bind the +CreateProductRequest.UnitPrice` property.

That is, unless a global handler or model binder provider have been registered for the `CurrencyRequest` type…

6.5.3 Registering a global binder

In much the same way that you can set the default model binder as the fallback for all model types, you can register model binders for individual types as well. Like setting the default model binder, the syntax is very simple.

This example tells the framework to use the `JsonModelBinder` for every `Currency` model it comes across.

```
ModelBinders.Binders.Add(typeof(Currency), new JsonModelBinder());
```

This approach allows you to associate a model binder with a particular type across the entire application in a single line. It's also an effective way to control how business models are bound without having to decorate those models with custom model binder attributes.

6.5.4 Ask the binders!

Yet another approach for locating model binders is to let the model binders themselves dynamically decide whether they should bind a given model. ASP.NET MVC implements this approach by way of the `ModelValidatorProviders` collection which simply iterates through a list of `IModelBinderProvider+s and asks each provider if they can create the appropriate binder for a given type.  Of course, you are free to add and remove the providers present in the +ModelValidatorProviders` collection and even add your own.

In order to see the `IModelBinderProvider` in action, An IModelBinderProvider implementation shows `AbstractModelBinde` updated to implement the `IModelBinderProvider` interface. The logic dictating whether or not the `AbstractModelBinder` should accept responsibility for binding is relatively straight-forward: is the model type a non-concrete type? If so, return a new instance of `AbstractModelBinder` so that it can work its magic. If not, then return a `null` value to indicate that the `AbstractModelBinder` is probably not the appropriate binder for this particular type.

An IModelBinderProvider implementation

```
public class AbstractModelBinderProvider : IModelBinderProvider
{
        public IModelBinder GetBinder(Type modelType)
        {
                var isAbstractType = modelType.IsAbstract || modelType.IsInterface;

                if (!isAbstractType)
                        return null;

                return new AbstractModelBinder();
        }
}
```

Then, in order to begin using the `AbstractModelBinderProvider` you'll need to register it in the list of providers:

```
ModelBinderProviders.BinderProviders.Add(new AbstractModelBinderProvider());
```

The `IModelBinderProvider` approach makes the 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 apporpriate place: the model binders themselves.

6.6 Conclusion

ASP.NET MVC's model binding allows controller actions to accept complex object types as parameters, just like any other method. It is also an incredibly powerful and useful feature that encourages better separation of concerns by keeping the logic of populating objects with data from the request away from the logic that uses the populated objects. Though it readily supports most scenarios, many developers may com across scenarios that the out-of-the-box model binding functionality doesn't support well, or even at all. Luckily, ASP.NET MVC's model binding is powered by a robust framework that provides developers with full control over how object data is populated. This framework provides plenty of extensibility points that allow developers to append, enhance, or even replace core framework functionality with their own custom logic.

In addition to providing an overview of how ASP.NET MVC model binding works, this article exposes some of the common problem scenarios and how to work within and extend the framework to address these scenarios while still getting the most out of what ASP.NET MVC has to offer.

Chapter 7

CHP7

TODO

Chapter 8

CHP8

TODO

Chapter 9

CHP9

TODO

Chapter 10

CHP10

TODO

Chapter 11

CHP11

TODO

Chapter 12

CHP12

TODO

Chapter 13

CHP13

TODO

Chapter 14

CHP14

TODO

Chapter 15

CHP15

TODO

Chapter 16

CHP16

TODO

Chapter 17

CHP17

TODO

Chapter 18

CHP18

TODO

Chapter 19

CHP19

TODO