Software Craftsmanship Guild

526 S. Main St, Suite 609

Akron, OH 44311

Model Binding

Software Craftsmanship Guild

Lesson Goals

Learn how ASP.NET MVC creates .NET objects using data sent by the browser in an HTTP request.

Any time we create an ActionMethod that takes a parameter, the parameter is created via model binding.

Technique 1 – Request Object

The Request object is provided by the controller inheritable and contains all the values in the Request from the browser for your use.

This technique does not use model binding at all.

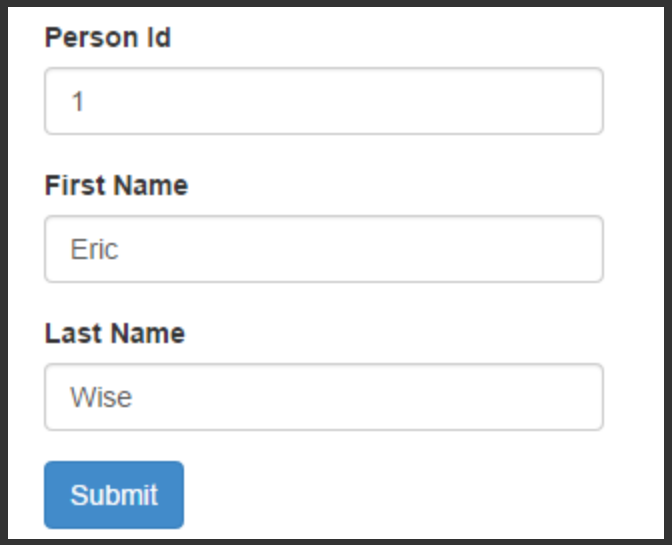
Here's the View

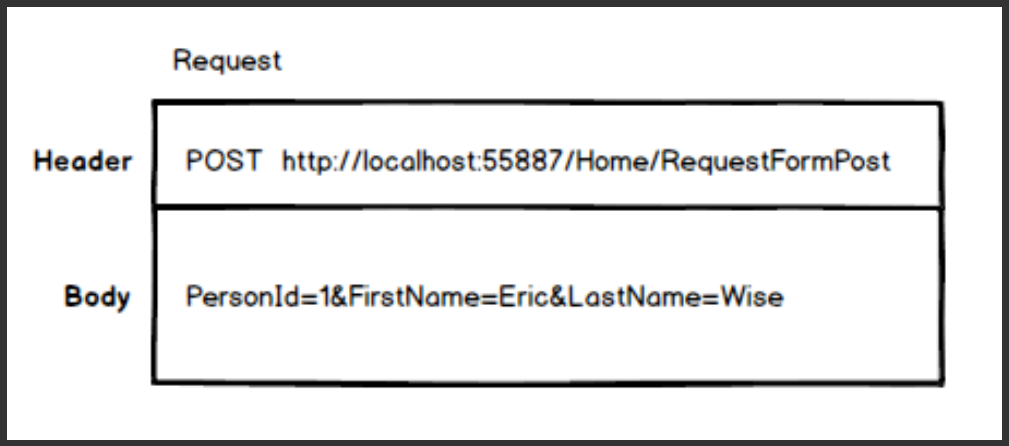
```
<h2>Standard Posting Using Request.Form in the Post</h2>

@using(Html.BeginForm("RequestFormPost", "Home", FormMethod.Post))
{
    <div class="form-group">
        <label>Person Id</label>
        <input type="text" class="form-control" name="PersonId"/>
    </div>
    <div class="form-group">
        <label>First Name</label>
        <input type="text" class="form-control" name="FirstName" />
    </div>
    <div class="form-group">
        <label>Last Name</label>
        <input type="text" class="form-control" name="LastName" />
    </div>
    <button class="btn btn-primary">Submit</button>
}
```

SOFTWARE CRAFTSMANSHIP GUILD

What the Browser Does…





What the Server Does…

```
[HttpPost]
public ActionResult RequestFormPost()
{
    var p = new Person();
    p.PersonId = int.Parse(Request.Form["PersonId"]);
    p.FirstName = Request.Form["FirstName"];
    p.LastName = Request.Form["LastName"];

    return View("Result", p);
}
```

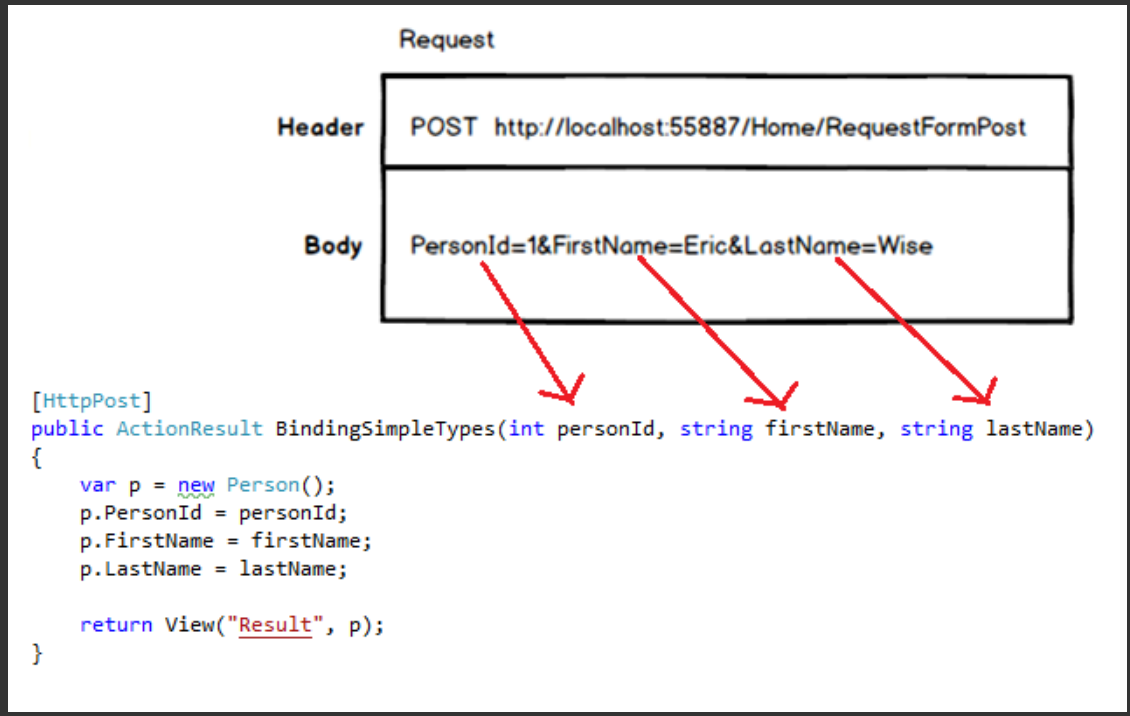
SOFTWARE CRAFTSMANSHIP GUILD

Technique 2 – Simple Types

We can create method parameters for each of our form fields.

The ASP .NET Model Binder will examine the names of the form fields and the names of the method parameters.

If it finds a match it will convert the data into the parameter. If you do not have a match the parameter will receive its default value (0 for numbers, false for Boolean, null for string, etc).




```
[HttpPost]
public ActionResult BindingSimpleTypes(int personId, string firstName, string lastName)
{
    var p = new Person();
    p.PersonId = personId;
    p.FirstName = firstName;
    p.LastName = lastName;

    return View("Result", p);
}
```

So the whole point of Model Binding…

ASP.NET examines everything in the request header and body (HTTP – HyperText Transfer Protocol) and attempts to map it automatically to your method parameters.

It will examine (in this order)

1. Form (body) elements
2. URL (Route / RouteConfig.cs) elements
3. QueryString Elements

Binding to a Class Type

Let's provide an empty person model to the view and use Razor this time.

```
public ActionResult BindingClassTypes()
{
    return View(new Person());
}

[HttpPost]
public ActionResult BindingClassTypes(Person p)
{
    return View("Result", p);
}
```

And the view…

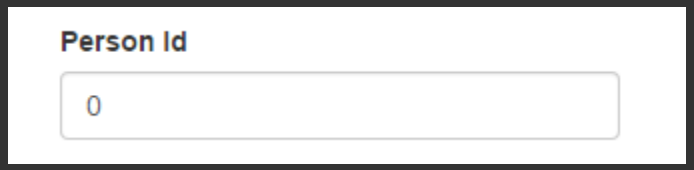
```
@model MvcModelBinding.Models.Person

@{
    ViewBag.Title = "BindingClassTypes";
}

<h2>Standard Posting Using a Class Type in the Post</h2>
<p>We will also switch to Razor</p>
@using (Html.BeginForm("BindingClassTypes", "Home", FormMethod.Post))
{
    <div class="form-group">
        <label>Person Id</label>
        @Html.TextBoxFor(m => m.PersonId, new { @class = "form-control" })
        @*<input type="text" class="form-control" name="PersonId" />*@
    </div>
    <div class="form-group">
        <label>First Name</label>
        @Html.TextBoxFor(m => m.FirstName, new { @class = "form-control" })
        @*<input type="text" class="form-control" name="FirstName" />*@
    </div>
    <div class="form-group">
        <label>Last Name</label>
        @Html.TextBoxFor(m => m.LastName, new { @class = "form-control" })
        @*<input type="text" class="form-control" name="LastName" />*@
    </div>
    <button class="btn btn-primary">Submit</button>
}
```

Notice the Person Id

The default of int is 0, so now the textbox pre-fills with a zero.



This is where nullable types come in handy.

```
public class Person
{
    public int? PersonId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

Classes Within Classes

Let's add a HomeAddress to our Person

```
public class Address
{
    public string City { get; set; }
    public string State { get; set; }
    public string Zipcode { get; set; }
}
```

```
public class Person
{
    public int? PersonId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public Address HomeAddress { get; set; }
}
```

Add a new Controller / views

```
public ActionResult BindingComplexClassTypes()
{
    return View(new Person() { HomeAddress = new Address()});
}

[HttpPost]
public ActionResult BindingComplexClassTypes(Person p)
{
    return View("Result2", p);
}
```

Add fields to the new form

```
<div class="form-group">
    <label>City</label>
    @Html.TextBoxFor(m => m.HomeAddress.City, new { @class = "form-control" })
</div>
<div class="form-group">
    <label>State</label>
    @Html.TextBoxFor(m => m.HomeAddress.State, new { @class = "form-control" })
</div>
<div class="form-group">
    <label>Zipcode</label>
    <input type="text" name="HomeAddress.Zipcode" class="form-control"/>
</div>
<button class="btn btn-primary">Submit</button>
```

Binding a List of Objects

Binding a list of objects is as simple as putting an array indexer on the front of the field.

Usually we will mix in some jQuery with a button that just adds more rows to the form with the right indexer. Then you get a nice “list building” effect.

```
<div class="form-group">
    <label>Address 1 city</label>
    <input type="text" name="[0].City" class="form-control" />
</div>
<div class="form-group">
    <label>Address 1 state</label>
    <input type="text" name="[0].State" class="form-control" />
</div>
<div class="form-group">
    <label>Address 1 zipcode</label>
    <input type="text" name="[0].Zipcode" class="form-control" />
</div>
<hr />
<div class="form-group">
    <label>Address 2 city</label>
    <input type="text" name="[1].City" class="form-control" />
</div>
<div class="form-group">
    <label>Address 2 state</label>
    <input type="text" name="[1].State" class="form-control" />
</div>
```

Multiple Object Binding on the Controller

Nothing surprising here. Any collection type will work (list, array, etc)

```
public ActionResult BindingMultipleObjects()
{
    return View();
}

[HttpPost]
public ActionResult BindingMultipleObjects(List<Address> addresses)
{
    return View("Result3", addresses);
}
```

Custom Binding

Like most things in ASP.NET, the model binder is customizable, so if you don't like what it is doing or have a special case, you can roll your own binding logic.

To do this you need to create a class that implements IModelBinder and write your code in the BindModel() method.

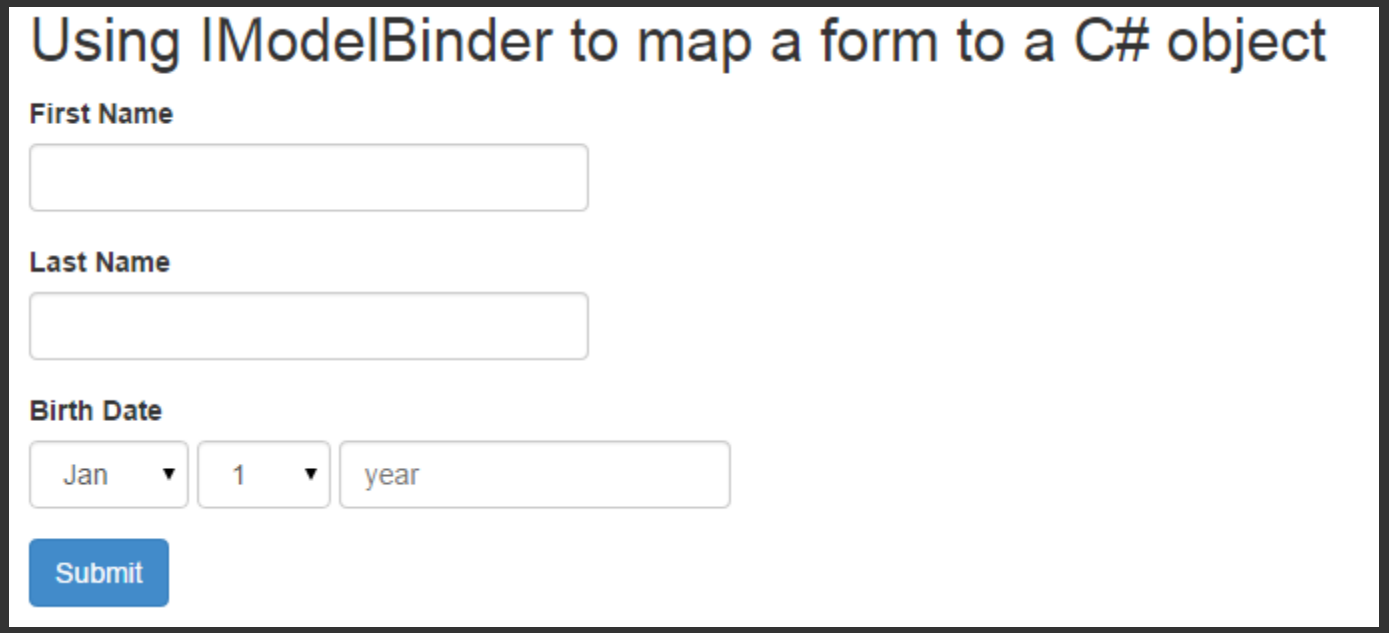
Form for Custom Binding

Let’s add a new class that contains a birthdate.

```
public class BirthdayPerson
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime BirthDay { get; set; }
}
```

View

Let’s make a view that doesn’t map directly to a DateTime by splitting month/day/year.



Binding Class

Implement IModelBinder and we're back to using the request object.

```
public class BirthdayPersonBinder : IModelBinder
{
    public object BindModel(ControllerContext controllerContext, ModelBindingContext bindingContext)
    {
        var person = new BirthdayPerson();

        person.FirstName = controllerContext.HttpContext.Request.Form["FirstName"];
        person.LastName = controllerContext.HttpContext.Request.Form["LastName"];

        int month = int.Parse(controllerContext.HttpContext.Request.Form["month"]);
        int day = int.Parse(controllerContext.HttpContext.Request.Form["day"]);
        int year = int.Parse(controllerContext.HttpContext.Request.Form["year"]);

        person.BirthDay = new DateTime(year, month, day);

        return person;
    }
}
```

Action Method

Two ways of doing this. For a one-off, you can tell it which model binder as an attribute in the method like so:

```
[HttpPost]
public ActionResult CustomModelBinding([ModelBinder(typeof(BirthdayPersonBinder))]BirthdayPerson person)
{
    return View("Result4", person);
}
```

Or you can register it in the global.asax so all BirthdayPerson bindings use this:

```
ModelBinders.Binders[typeof(BirthdayPerson)] = new BirthdayPersonBinder();
```

Gut Check

- Where in the request will the default model binder look to find values to populate models with? (Bonus points for correct order)
  1. Form fields in a POST body
  2. URL segment values
  3. QueryString parameters

Gut Check

- What character in a form field name allows binding to a child object's property?
  - A period
  - <input name="HomeAddress.ZipCode" />
- What characters in a form field name allows binding to items within a list?
  - Square brackets with an index
  - <input name="[0].FirstName" />

Gut Check

- What should you Google when you have a situation that the default model binder can't handle?
  - "ASP.NET MVC custom model binder"
  - (You're not expected to master these right now!)

Conclusion

The default model binder is capable of handling most of the common scenarios for working with web form data.

File away in your mind that you can override and write your own binder for very complex scenarios, but most of the time the default one is sufficient.