Agenda: Understanding Model Binders

- Html Form behavior
- Model Binder Overview
- DefaultModelBinder
- Binding to Complex Classes
- FormCollection Model Binding
- HttpPostedFileBase Model Binder
- Bind Attribute
- UpdateModel and TryUpdateModel
- Writing Custom Model Binders

Html Form Behavior

1. Form tag cannot be nested inside another form tag.
2. Input type's **button** and **reset** and any **disabled** element name and value pair is not submitted to the server.
3. Input type="button" is used only for writing client side JavaScript code.
4. Input type="checkbox" submits name and value pair only if the checkbox is checked otherwise nothing is submitted. If value is not provided the default value posted is "on"
5. To group radio buttons, **same name** must be given to them and only the name-value pair of the radio button selected will be submitted to server.
6. For select, name of select and value of the option selected is submitted to server.
7. If multiple options are selected in Listbox (Select tag), then with same name different values (multiple pairs) are submitted to server
8. Input type="image" behaves as submit button whereas <img> tag renders static image. Position of mouse cursor is submitted to server.
9. A form can have more than one submit button, but only the name-value pair of the submit button used to post the form is included with the request.
10. If the form has input type="file" to upload a file along with request then its enctype="multipart/form-data" and method="post".

Model Binders Overview

Model Binders is responsible for mapping a browser request into an object. Your action methods need data, and the incoming HTTP request carries the data you need and this is embedded into POST-ed form values, and possibly the URL itself.

Model binders allow your controller code to remain cleanly separated from the dirtiness of interrogating the request and its associated environment.

DefaultModelBinder magically converts form values and route data into objects.

Default Model Binder: It creates following types of objects from the browser request

a) Primitive types such as string, int, decimal, DateTime etc…

b) A class such as Employee or Department…

c) An array such as string or Employee

d) A collection such as IEnumerable<T>, ICollection<T> (When elements like checkbox have same name)

View :

```
@using (Html.BeginForm())
{
  <input type="text" name="name" value=" " />
  <input type="checkbox" name="hobbies" value="Football" /><span>Football</span>
  <input type="checkbox" name="hobbies" value="Cricket" /><span>Cricket</span>
  <input type="checkbox" name="hobbies" value="Table Tennis" /><span>Table Tennis</span>
  <input type="checkbox" name="hobbies" value="Badminton" /><span>Badminton</span>
  <br />
  <input type="submit" name="btnSubmit" value="Submit" />
}
```

Controller:

```
public class HobbiesController : Controller
{
  public ActionResult Index()
  {
    return View();
  }
  [HttpPost]
  public ActionResult Index(string[] hobbies, string name)
  {
    string str = "Hobbies of " + name + " are<br>";
    foreach (string s in hobbies)
      str += s + "<br>";
    return Content(str);
  }
}
```

Note: In the above action method parameter can be string[], IEnumerable<string> or List<string>

Binding to Complex Classes

This is situation where Model is a class has another class as one of its members. For example: Employee has Address as its member.

Model Classes

```
public class Address
```

```
{
   public string Street
   { get; set; }
   public string City
   { get; set; }
}
public class Employee
{
   public string Name
   { get; set; }
   public Address ResAddress
   { get; set; }
   public Address OffAddress
   { get; set; }
}
```

Controller:

```
public class HomeController : Controller
 {
    public ActionResult Index()
    {
       return View();
    }
    [HttpPost]
    public ActionResult Index(Employee emp)
    {
       string str = "Name: " + emp.Name + "<br>";
       str += "Res Street: " + emp.ResAddress.Street + "<br>";
       str += "Res City: " + emp.ResAddress.City + "<br>";
       return Content(str);
    }
 }
```

View:

```
@using (Html.BeginForm())
{
   <label>Name</label><input type="text" name="name" value=" " /><br />
   <label>Res Street</label><input type="text" name="ResAddress.Street" value=" " /><br />
   <label>Res City</label><input type="text" name="ResAddress.City" value=" " /><br />
```

```
    <label>Off Street</label><input type="text" name="OffAddress.Street" value=" " /><br />
    <label>Off City</label><input type="text" name="OffAddress.City" value=" " /><br />
    <input type="submit" name="submit" value="Submit" />
}
```

Using FormCollection Model Binding

It's an **un-typed** collection of form fields. In the following case ModelState object is not created.

```
[HttpPost]
public ActionResult Index(FormCollection fc)
{
    string str = "Id=" + fc["Id"] + "<br>";
    str += "Name: " + fc["Name"] + "<br>";
    str += "Street: " + fc["ResAddress.Street"] + "<br>";
    str += "City: " + fc["ResAddress.City"] + "<br>";
    return Content(str);
}
```

Using the HTTP Posted File Base Model Binder

This is used for uploading file(s)

Model:

```
public class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Photo { get; set; }
}
```

View:

```
@using (Html.BeginForm("Index", " Home", FormMethod.Post, new { enctype = "multipart/form-data" }))
{
     <label>Id</label><input type="text" name="Id" value="1" /><br />
     <label>Name</label><input type="text" name="Name" value="a1" /><br />
     <label>Photo</label><input type="file" name="Photo" value="" /><br />
     <input type="submit" name="submit" value="Submit" />
}
```

Controller:

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View();
    }
    [HttpPost]
    public ActionResult Index(HttpPostedFileBase photo, Employee emp)
    {
        string str = "Id=" + emp.Id + "<br>";
        str += "Name: " + emp.Name + "<br>";
        string fn = photo.FileName;
        photo.SaveAs(Server.MapPath("~/uploads/" + fn));
        str += fn + " is saved";
        return Content(str);
    }
}
```

Note:

1. If more than one file upload is used then parameter must be: **IEnumerable<HttpPostedFileBase>**
2. For large files in **web.config**: `<httpRuntime maxRequestLength="65536" /> <!--64MB-->`

Using the Bind Attribute

The form post is matched to a strongly typed object. This can save you time initially but it can also provide a way for a malicious user to exploit vulnerabilities in your code if you're not careful. You can optionally override what values to bind and what not to bind automatically.

Bind attribute is used to control how a model converts a request into an object.

BindAttribute can be used either on model class or parameters of action method only.

Following are its properties

1. **Exclude**: To exclude a comma separated list of properties from binding
2. **Include**: To include a comma separated list of properties from binding
3. **Prefix**: To associate a parameter with a particular form field prefix.

Example of Exclude Property: We can provide either to Method Parameter or to Model Class.

This is useful when validations are performed on Model using ModalState.IsValid

```
[HttpPost]
public ActionResult Index([Bind(Exclude = "Id")]Employee emp)
{. . .}
```

or

```
[Bind(Exclude="Id")]
public class Employee
{. . .}
```

Example of Prefix Property: We won't need to use the Bind attribute to map prefixes to parameters because default model binder is smart enough to do the correct mapping automatically.

Model:

```
public class Address
{
   public string Street
   { get; set; }
   public string City
   { get; set; }
}
```

View:

```
@using (Html.BeginForm())
{
   <div>
      Billing Address<br />
      Street<input type="text" name="Billing.Street" /><br />
      City<input type="text" name="Billing.City" /><br />
   </div>
   <div>
      Shipping Address<br />
      Street<input type="text" name="Shipping.Street" /><br />
      City <input type="text" name="Shipping.City"/><br />
   </div>
   <input type="submit" name="submit" value="Submit" />
}
```

Controller:

```
public class HomeController : Controller
{
   public ActionResult Index()
   {
      return View();
   }
   [HttpPost]
   public ActionResult Index([Bind(Prefix="billing")]Address b, Address shipping)
   {
      string str = "";
      str += "Billing: " + b.Street + ", " + b.City + "<br>";
      str += "Shipping: " + shipping.Street + ", " + shipping.City + "<br>";
      return Content(str);
```

```
    }
}
```

UpdateModel and TryUpdateModel

UpdateModel Method (9 overloaded forms): Updates the specified model instance using values from the value provider, a prefix, a list of properties to exclude, and a list of properties to include.

```
protected internal void UpdateModel<TModel>(
            TModel model,
            string prefix,
            string[] includeProperties,
            string[] excludeProperties,
            IValueProvider valueProvider
)
```

This method is generally used when an existing object has to be initialized with the data submitted along with the request.

Example of UpdateModel and TryUpdateModel:

```
    [HttpPost]
    public ActionResult Index(FormCollection fc)
    {
      Address shipping = new Address();
      Address billing = new Address();
      string str = "";
      UpdateModel(billing, "billing"); //Will throw exception if validation fails
      bool b2 = TryUpdateModel(shipping, "shipping"); //Return false when validation fails.

      str += "Billing: " + billing.Street + ", " + billing.City + "<br>";
      str += "Shipping: " + shipping.Street + ", " + shipping.City + "<br>";
      return Content(str + " "  + " " + b2);
    }
```

- UpdateModel() throws an exception if validation fails.
- **TryUpdateModel** will try to update the model with the given value for Address. If the update fails validation then TryUpdateModel will pass update the **ViewData.ModelState** with validation errors and your view will display the validation errors.

Custom Model Binders

In situations DefaultModelBinder is not suitable we will have to write Custom Binder.

1. Add the following class to the project:

```
    public class Address
    {
      public string Street
```

```
        { get; set; }
        public string City
        { get; set; }
    }
    public class Order
    {
        public int OrderId { get; set; }
        public Address BillingAddress { get; set; }
        public Address ShippingAddress { get; set; }
    }
```

2. Add the following to the controller

```
    public ActionResult Index()
    {
        return View();
    }
    [HttpPost]
    public ActionResult Index(Order order)
    {
        string str = "";
        str += "Billing: " + order.BillingAddress.Street + ", " + order.BillingAddress.City + "<br>";
        str += "Shipping: " + order.ShippingAddress.Street + ", " + order.ShippingAddress.City + "<br>";
        return Content(Request.Form + " " +  str);
    }
```

3. Add the following to the View

```
    @using (Html.BeginForm())
    {
        <div>
            Order Id: <input type="text" name="orderId" value="1" />
        </div>
        <div>
            Billing Address<br />
            Street<input type="text" name="billingStreet" /><br />
            City<input type="text" name="billingCity" /><br />
        </div>
        <div>
            Shipping Address<br />
            Street<input type="text" name="shippingStreet" /><br />
            City <input type="text" name="shippingCity"/><br />
        </div>
    <input type="submit" name="submit" value="Submit" />
  }
```

4. Add the Custom Model Provider class to Project:

```
public class CustomOrderBinder : IModelBinder
{
  public object BindModel(ControllerContext controllerContext, ModelBindingContext bindingContext)
  {
    Order order;
    if (bindingContext.Model == null) //If Order is Parameter to Action Method
    {
      order = new Order();
      order.BillingAddress = new Address();
      order.ShippingAddress = new Address();
    }
    else
    {
      //Usefull if UpdateModel(order) or TryUpdateModel(order) is used.
      order = (Order)bindingContext.Model;
    }
    order.OrderId = GetValue<int>(bindingContext, "orderid");
    order.BillingAddress.City = GetValue<string>(bindingContext, "billingCity");
    order.BillingAddress.Street = GetValue<string>(bindingContext, "billingStreet");
    order.ShippingAddress.City = GetValue<string>(bindingContext, "shippingCity");
    order.ShippingAddress.Street = GetValue<string>(bindingContext, "shippingStreet");
    return order;
  }
  private T GetValue<T>(ModelBindingContext bindingContext, string key)
  {
    //At runtime the MVC framework populates the ValueProvider with values it finds in the request's
form, route, and query string collections.
    ValueProviderResult valueResult = bindingContext.ValueProvider.GetValue(key);
    bindingContext.ModelState.SetModelValue(key, valueResult); //**
    return (T)valueResult.ConvertTo(typeof(T));
    //**
    //One of the side-effects of model binding is that binding the model should put model values into
ModelState. When an HTML helper sees there is a ModelState error for "Name", it assumes it will also find
the "attempted value" that the user entered. The helper uses attempted values to repopulate inputs and
allow the user to fix any errors.
  }
```

```
    }
```

5. Add the following to Application_Start in **global.asax**

   ```
   ModelBinders.Binders.Add(typeof(Order), new CustomOrderBinder());
   ```

   OR

   Add the attribute to Order parameter of Action Method.

   ```
   [ModelBinder(typeof(CustomOrderBinder))]
   ```