Passing Data From a View to a Controller in ASP.NET MVC 5

John F. Gnazzo

Introduction

The Model-View-Controller (MVC) architectural pattern separates an application into three main components: the Model, the View and the Controller. The ASP.NET MVC framework provides an alternative to the ASP.NET Web Forms pattern for creating web applications. It is a lightweight, highly-testable presentation framework that is being used to easily create large, scalable and highly-performant websites for commercial purposes.

Tantamount to web applications is the collection and processing of user input data. Data is collected in the application View and processed through the Application Controller.

This blog will discuss four (4) common ways to pass data from the view to the controller:

- Passing by Typed Arguments
- Request Object
- Form Collections Object
- Data Binding

Understanding the methods available to handle data contributes to application scalability and robustness.

MVC Application

To demonstrate passing of data from a browser to a controller, a rudimentary application must be created. It should consist of a View, Controller and Mapper function, which in this example behaves as the Model.

Application View

The View is used to collect the data. Figure 1 demonstrates the user interface, or Data Input Form, consisting of a select element, two (2) input elements and a submit button.

The code for the application view is also shown.

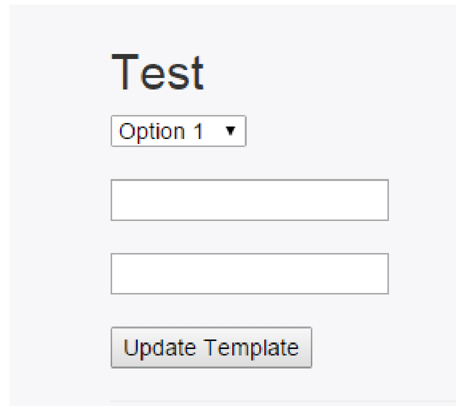

Figure 1- Application View

Application View Code

```
@{
    ViewBag.Title = "Razor View Data Input Form";
}

<h2>Test</h2>

@using (@Html.BeginForm("Compute", "Example", FormMethod.Post))
{
    <select id="updateTemplateSelect" name="option">
        <option value="1">Option 1</option>
        <option value="2">Option 2</option>
        <option value="3">Option 3</option>
    </select>
    <br/><br/>
    <input id="Param1" name="Param1" type="text"/>
    <br/><br/>
    <input id="Param" name="Param2" type="text" />
    <br/> <br/>
    <input id="updateTemplateSubmit" type="submit" value="Update Template" >
    <br />
}
```

Supporting Code

The following code is used to support the application. The Mapper is called by the Controller to process the data and return a result. The Model represents the View Model used to encapsulate the required data for the Output view. The Model is also used for the Data Binding Code Controller Example.

Mapper

```
    public class ComputeMapper
    {
        public ComputeViewModel Compute(string, double param1, double param2)
        {
            return new ComputeViewModel
            {
                Option = option,
```

```
                Param1 = param1,
                Param2 = param2,
                Result = param1*param2
            }
        }

        public ComputeViewModel Compute(ComputeViewModel model)
        {
            model.Result = model.Param1 * model.Param2;
            return model;
        }
    }
```

Model

```
    {
        public string Option { get; set; }
        public double Param1 { get; set; }
        public double Param2 { get; set; }
        public double Result { get; set; }
    }
```

Output View

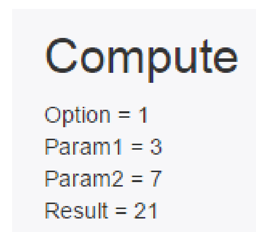

Figure 2- Output View

Output View Code

```
@model ASPNetPlayground.ViewModels.ComputeViewModel

@{
    ViewBag.Title = "Compute";
}

<h2>Compute</h2>
Option = @Model.Option<br/>
Param1 = @Model.Param1<br/>
Param2 = @Model.Param2<br/>
Result = @Model.Result<br />
```

Controller

The Controller is the target for posting the data after the user fills out the Data Input Form and clicks the submit button, In this case, the submit button is labeled “Update Template.”

There are four common ways to construct controller methods to handle data input from the Data Input Form. Each of these four ways is illustrated below. In each case, the input data is extracted and sent to the Mapper for processing.

Passing by Typed Arguments

A raw Controller method strongly types all of the data being posted from the Application's Data Input Form. In this case the option (string), Param1 (double data type) and param2 (double data type).

Passing by Typed Arguments Code.

```
[HttpPost]
public ActionResult Compute(string option, double param1, double param2)
{
    var model = _computerMapper.Compute(option, param1, param2);
    return View(model);
}
```

Advantages

- Input data is obvious
- Input data is typed and does not require conversion (casting)

Disadvantages

- Change in data input parameters, including either name or data type, requires modification of the method
- Does not scale well to large forms

Request Object

Using the Request object can also be used to handle posted data from the Application's input form. Using this method, the Request object that is created from posting the form contains all of the input data from the form. However, each Request element is mapped by element name, and is typed as a string. This means that every input parameter must be decoded and all non-string parameters must be converted to the appropriate type (see example below).

Request Object Code

```
[HttpPost]
  public ActionResult Compute()
  {
      var option = Request["Option"];
      var param1 = Convert.ToDouble(Request["Param1"]);
      var param2 = Convert.ToDouble(Request["Param2"]);
      var model = _computerMapper.Compute(option, param1, param2);
      return View(model);
  }
```

Advantages

- Uses automatically created Request object

Disadvantages

- Can easily cause errors by referencing objects in the Request object, not on the form
- Change in data input parameters, including either name or data type, requires modification of the method.

- All non-string input must be converted to the appropriate type
- Does not scale well to large forms

Form Collection Object

Using the Form collection object can also be used to handle posted data from the Application's input form. The collection object method handles data similar to the example using the Request object above; however, the Form collection object must be used as the input to the controller method.

Processing of data is similar to using the Request object as data; it must also be extracted from the element and converted to the correct data type if not a string.

Form Collection Object Code

```
[HttpPost]
public ActionResult Compute(FormCollection collection)
{
    var option = collection["Option"];
    var param1 = Convert.ToDouble(collection["Param1"]);
    var param2 = Convert.ToDouble(collection["Param2"]);
    var model = _computerMapper.Compute(option, param1, param2);
    return View(model);
}
```

Advantages

- Only contains the input data from the form

Disadvantages

- Change in data input parameters, including either name or data type, requires modification of the method
- All non-string input must be converted to the appropriate type
- Does not scale well to large forms

Data Binding

Data binding is used to allow input data to be mapped directly to a class, and specification of the class as an input to the controller method. The data binder automatically maps input data directly into class member variables. The sample below shows the data in the model extracted by parameter and sent to the Mapper. In real-world scenarios, it would be more appropriate to pass the model directly to the Mapper.

Data Binding Code

```
[HttpPost]
public ActionResult Compute(ComputeViewModel inputModel)
{
    var model = _computerMapper.Compute(inputModel);
```

```
            return View(model);
        }
```

Advantages

- Can be used to model all or some of the data from the form
- Can scale well to complex data input forms
- Supports separation of concerns best

Disadvantages

- Change in data input parameters, including either name or data type, requires modification of the model and the mapper method

Summary

While there are four common methods to pass data from a data input form to an MVC application's controller method, understanding the methods available to handle data contributes to application scalability and robustness.