Temp Data in Asp.Net MVC:

TempData in ASP.NET MVC can be used to store temporary data which can be used in the subsequent request. TempData will be cleared out after the completion of a subsequent request.

TempData is useful when you want to transfer data from one action method to another action method of the same or a different controller as well as redirects. It is dictionary type which is derived from [TempDataDictionary](https://msdn.microsoft.com/en-us/library/system.web.mvc.tempdatadictionary(v=vs.118).aspx" \t "_blank).

 TempData keeps the information for the time of an HTTP Request. This mean only from one page to another. It helps to maintain data when you move from one controller to other controller or from one action to other action. **It internally uses session variables.**

Example:-

Controller:-

using System.Web;

using System.Web.Mvc;

namespace TempDataExample.Controllers

{

public class HomeController : Controller

{

//

// GET: /Home/

public ActionResult Index ()

{

ViewBag.Address = "Shollinganallur";

TempData ["Name"] = "Ankit";

TempData ["Age"] = 26;

return RedirectToAction ("Details");

}

public ActionResult Details()

{

TempData. Keep (); // Use to maintain the Temp Data in the third request

return View ();

}

}

}

**View:-**

@{

ViewBag.Title = "Details";

}

<h2>Details</h2>

<h3>Temp Data Is:</h3>

<p>@TempData["Name"]</p>

<h3>View Bag</h3>

<p>@ViewBag.Address</p>

**Output:-**

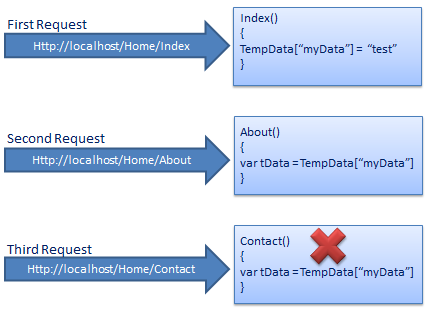
## Details

### Temp Data Is:

Ankit

View Bag

The following figure illustrates TempData.

TempData

TempData internally uses session to store the data. So the data must be serialized if you decide you to switch away from the default Session-State Mode, and use State Server Mode or SQL Server Mode.

As you can see in the above example, we add test data in TempData in the first request and in the second subsequent request we access test data from TempData which we stored in the first request. However, you can't get the same data in the third request because TempData will be cleared out after second request.

Call TempData. Keep () to retain TempData values in a third consecutive request.

## Key Difference between view and partial view in Asp.net mvc

1. Partial view generally does not have any Layout while view content Layout.
2. Partialview is reusable content which is render inside view (parent page).
3. View can basically contains a complete markup which may contain a master view (or master page) with all the design(s) etc. whereas Partial view is only a portion of page or a small markup which don't have master page. It is basically used as user control in mvc and it can be used at more than one views.

**Note:-Layout View is designed for the application that has different section for each block and to display same layout to entire web application (Like Master Page in ASP.Net Web Form)**

# Filters in MVC:

In ASP.NET MVC, a user request is routed to the appropriate controller and action method. However, there may be circumstances where you want to execute some logic before or after an action method executes. ASP.NET MVC provides filters for this purpose.

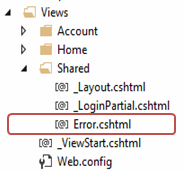
Filters can be applied to an action method or controller in a declarative or programmatic way. Declarative means by applying a filter attribute to an action method or controller class and programmatic means by implementing a corresponding interface.

MVC provides different types of filters. The following table list filter types, built-in filters for the type and interface which must be implemented to create a custom filter class.

| **Filter Type** | **Description** | **Built-in Filter** | **Interface** |
| --- | --- | --- | --- |
| Authorization filters | Performs authentication and authorizes before executing action method. | [Authorize], [RequireHttps] | IAuthorizationFilter |
| Action filters | Performs some operation before and after an action method executes. |  | IActionFilter |
| Result filters | Performs some operation before or after the execution of view result. | [OutputCache] | IResultFilter |
| Exception filters | Performs some operation if there is an unhandled exception thrown during the execution of the ASP.NET MVC pipeline. | [HandleError] | IExceptionFilter |

To understand the filter in detail, let's take an example of built-in Exception filter.

An exception filter executes when there is an unhandled exception occurs in your application. HandleErrorAttribute ([HandlerError]) class is a built-in exception filter class in MVC framework. This built-in HandleErrorAttribute class renders Error.cshtml included in the Shared folder by default, when an unhandled exception occurs.

[](http://www.tutorialsteacher.com/Content/images/mvc/errorpage.png" \t "_blank)

The following example demonstrates built-in exception filter HandErrorAttribute.

Authorization filter example:

[HandleError]

public class HomeController: Controller

{

public ActionResult Index ()

{

throw new Exception ("This is unhandled exception");

return View ();

}

public ActionResult About ()

{

return View ();

}

public ActionResult Contact ()

{

return View ();

}

}

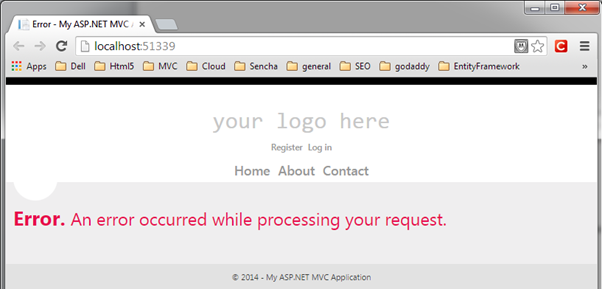
In the above example, we have applied [HandleError] attribute to HomeController. So now it will display Error page if any action method of HomeController would throw unhandled exception. Please note that unhandled exception is an exception which is not handled by the try-catch block.

Please make sure that CustomError mode is on in System.web section of web.config, in order for HandleErrorAttribute work properly.

CustomError in web.config:

<customErrors mode="On" />

Now, if you run the application. You would get following error page because we throw exception in Index action method for the demo purpose.



## Register Filters:

Filters can be applied at three levels.

### 1. Global Level:

You can apply filters at global level in the Application\_Start event of Global.asax.cs file by using default FilterConfig.RegisterGlobalFilters () mehtod. Global filters will be applied to all the controller and action methods of an application.

The [HandleError] filter is applied globaly in MVC Application by default in every MVC application created using Visual Studio as shown below.

Example: Register Global level Filters

// MvcApplication class contains in Global.asax.cs file

public class MvcApplication: System.Web.HttpApplication

{

protected void Application Start ()

{

FilterConfig.RegisterGlobalFilters (GlobalFilters.Filters);

}

}

// FilterConfig.cs located in App\_Start folder

public class FilterConfig

{

public static void RegisterGlobalFilters (GlobalFilterCollection filters)

{

filters.Add (new HandleErrorAttribute ());

}

}

2. Controller level:

Filters can also be applied to the controller class. So, filters will be applicable to all the action method of Controller class if it is applied to a controller class.

Example: Action Filters to Controller

[HandleError]

public class HomeController: Controller

{

public ActionResult Index ()

{

return View ();

}

}

3. Action method level:

You can apply filters to an individual action method also. So, filter will be applicable to that particular action method only.

Apply Filters to Action method:

public class HomeController: Controller

{

[HandleError]

public ActionResult Index ()

{

return View ();

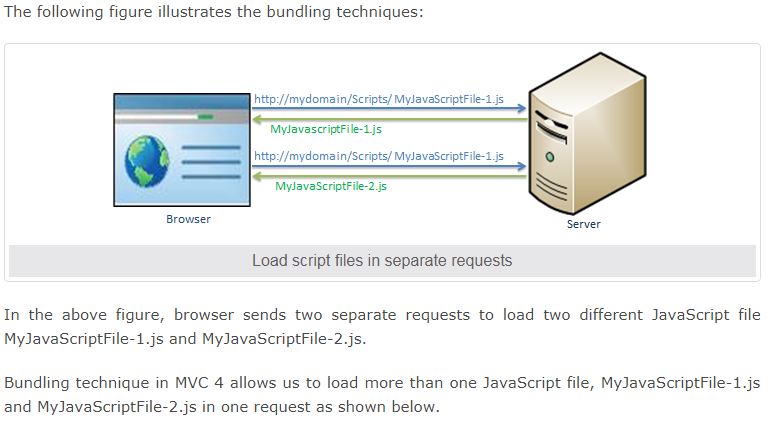
}

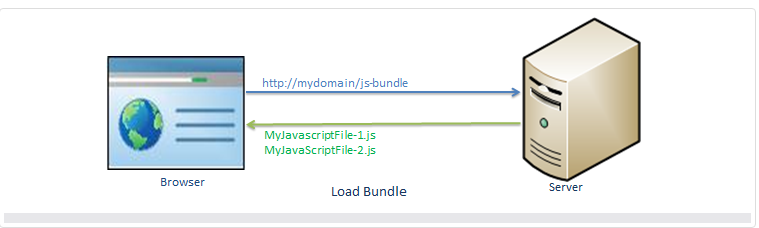
}

The same way, you can apply multiple built-in or custom filters globally or at controller or action method level for different purpose such as [Authorize], [RequireHttps], [ChildActionOnly], [OutputCache], [HandleError].

# Bundling:

Bundling and minification techniques were introduced in MVC 4 to improve request load time. Bundling allow us to load the bunch of static files from the server into one http request.





## Minification:

Minification technique optimizes script or css file size by removing unnecessary white space and comments and shortening variable names to one character.

For example, consider following JavaScript function.

Example: JavaScript

sayHello = function (name) {

//this is comment

var msg = "Hello" + name;

alert (msg);

}

The above JavaScript will be optimized and minimized into following script.

Example: Minified JavaScript

SayHello=function (n) {var t="Hello"+n; alert (t)}

As you can see above, it has removed unnecessary white space, comments and also shortening variable names to reduce the characters which in turn will reduce the size of JavaScript file.

Bundling and minification impacts on the loading of the page, it loads page faster by minimizing size of the file and number of requests.

## Bundle Types:

MVC 5 includes following bundle classes in System.web.Optimization namespace:

**ScriptBundle**: ScriptBundle is responsible for JavaScript minification of single or multiple script files.

**StyleBundle**: StyleBundle is responsible for CSS minification of single or multiple style sheet files.

**DynamicFolderBundle**: Represents a Bundle object that ASP.NET creates from a folder that contains files of the same type.

All the above bundle classes are included in *System.Web.Optimization.Bundle* namespace and derived from [Bundle class](http://msdn.microsoft.com/en-us/library/system.web.optimization.bundle(v=vs.110).aspx" \t "_blank).

# ScriptBundle in ASP.NET MVC:

ASP.NET MVC API includes [ScriptBundle](https://msdn.microsoft.com/en-us/library/system.web.optimization.scriptbundle(v=vs.110).aspx" \t "_blank) class that does JavaScript minification and bundling.

Open App\_Start\BundleConfig.cs file in the MVC folders. The BundleConfig.cs file is created by MVC framework by default. You should write your all bundling code in the BundleConfig.RegisterBundles () method. (you can create your own custom class instead of using BundleConfig class, but it is recommended to follow standard practice.) The following code shows a portion of the RegisterBundles method.

Example: BundleConfig.RegisterBundle()

using System.Web;

using System.Web.Optimization;

public class BundleConfig

{

public static void RegisterBundles (BundleCollection bundles)

{

// create an object of ScriptBundle and

// specify bundle name (as virtual path) as constructor parameter

ScriptBundle scriptBndl = new ScriptBundle ("~/bundles/bootstrap");

//use Include () method to add all the script files with their paths

scriptBndl.Include (

"~/Scripts/bootstrap.js",

"~/Scripts/respond.js"

);

//Add the bundle into BundleCollection

bundles.Add (scriptBndl);

BundleTable.EnableOptimizations = true;

}

}

In the above example, we have created a bundle of two JavaScript files, bootstrap.js and respond.js using ScriptBundle for demo purposes.

1. First of all create an instance of ScriptBundle class by specifing the bundle name as a constructor parameter. This bundle name is a virtual path starting with ~/. You can give anything in virtual path but it's recommended to give a path that will be easy to identify as a bundle. Here, we have given "~/bundles/bootstrap" path, so that we can easily identify that this bundle includes bootstrap related files.
2. Use Include method to add one or more JS files into a bundle with its relative path after root path using ~ sign.
3. Final, add the bundle into BundleCollection instance, which is specified as a parameter in RegisterBundle () method.
4. Last, BundleTable.EnableOptimizations = true enables bundling and minification in debug mode. If you set it to false then it will not do bundling and minification.

Thus, you can create a bundle of JavaScript files using ScriptBundle. MVC framework invokes BundleConfig.RegisterBundle () method from the Application\_Start event in Global.asax.cs file, so that it can add all the bundles into BundleCollection at the starting of an application.

Example: Invoke RegisterBundle () in Application\_Start event

public class MvcApplication : System.Web.HttpApplication

{

protected void Application\_Start ()

{

BundleConfig.RegisterBundles (BundleTable.Bundles);

}

}

## Include ScriptBundle in Razor View:

We have created a script bundle above. Now, we will learn how to include bundle into razor view.

The script bundles can be included using static [Scripts](http://msdn.microsoft.com/en-us/library/system.web.optimization.scripts(v=vs.110).aspx" \t "_blank) class. Use Scripts.Render () method to include specified script bundle at runtime.

Example: Scripts. Render ()

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>@ViewBag.Title</title>

**@Scripts.Render ("~/bundles/bootstrap")**

</head>

<body>

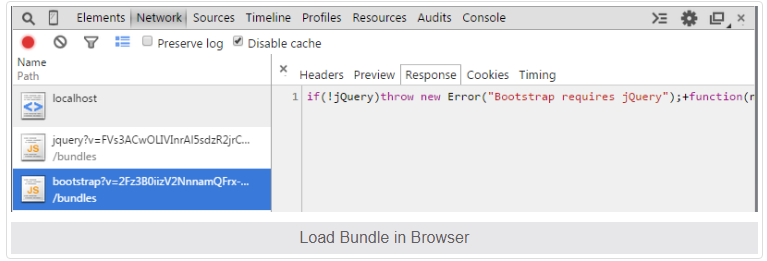
@\*html code removed for clarity \*@

</body>

</html>

Now, if you run the above example then you will find two script files is combined, minified and loaded in a single request. Please make sure that you have set debug = false in web.config.

<compilation debug="false" targetFramework="4.5"/>



# StyleBundle:

ASP.NET MVC API includes [StyleBundle](https://msdn.microsoft.com/en-us/library/system.web.optimization.stylebundle(v=vs.110).aspx" \t "_blank) class that does CSS minification and bundling. StyleBundle is also derived from a Bundle class so it supports same methods as ScriptBundle.

You should create bundles of script and css files in the RegisterBundles () method of BundleConfig class contained in App\_Start -> BundleConfig.cs file.

Use ScriptsInclude or IncludeDerictory method to add css files into bundle as shown below:

Example: StyleBundle

public class BundleConfig

{

public static void RegisterBundles (BundleCollection bundles)

{

bundles.Add (new StyleBundle ("~/bundles/css").Include (

"~/Content/bootstrap.css",

"~/Content/site.css"

));

// add ScriptBundle here..

}

}

## Include Style Bundle in Razor View:

You can use StyleBundle in a layout view and render bunch of css files in a single request using static [Styles](http://msdn.microsoft.com/en-us/library/system.web.optimization.styles(v=vs.110).aspx" \t "_blank) class. Styles is a helper class to render css bundles.

Example: Include Style Bundle in View

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>@ViewBag.Title - My ASP.NET Application</title>

**@Styles.Render("~/bundles/css")**

</head>

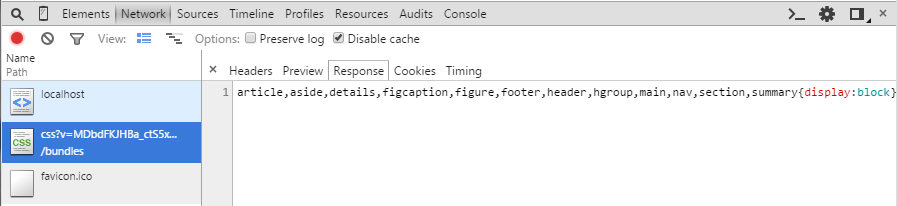
<body>

@\*html code removed for clarity \*@

</body>

</html>

As shown in the above example, use Styles.Render() method to include specified css bundle at runtime. Open developer tool of the browser and check that it has minified and loaded css files as shown below.

Load Bundle in Browser

# Action Filters (Interview Qus.):

Action filter executes before and after an action method executes. Action filter attributes can be applied to an individual action method or to a controller. When action filter applied to controller then it will be applied to all the action methods in that controller.

**MVC provides following action filters:**

**1-Output Cache**

**2-Handle Error**

**3-Authorize**

**1-Output Cache:-**

The main purpose of using Output Caching is to dramatically improve the performance of an ASP.NET MVC Application. It enables us to cache the content returned by any controller method so that the same content does not need to be generated each time the same controller method is invoked. Output Caching has huge advantages, such as it reduces server round trips, reduces database server round trips, reduces network traffic etc.

Let's take an example. My MVC application displays a list of database records on the view page so by default each time the user invokes the controller method to see records, the application loops through the entire process and executes the database query. And this can actually decrease the application performance. So, we can advantage of the "Output Caching" that avoids executing database queries each time the user invokes the controller method. Here the view page is retrieved from the cache instead of invoking the controller method and doing redundant work.

**Example** − Specifies the return value to be cached for 10 seconds.

public class ActionFilterDemoController : Controller {

[HttpGet]

OutputCache(Duration = 10)]

public string Index() {

return DateTime.Now.ToString("T");

}

}

**HandleError:-**

ASP.Net MVC has an attribute called "HandleError" that provides built-in exception filters. The HandleError attribute in ASP.NET MVC can be applied over the action method as well as Controller or at the global level. The HandleError attribute is the default implementation of IExceptionFilter. When we create a MVC application, the HandleError attribute is added within the Global.asax.cs file and registered in the Application\_Start event.

public static void RegisterGlobalFilters(GlobalFilterCollection filters)  
{  
    filters.Add(new HandleErrorAttribute());  
}

protected void Application\_Start()  
{  
    AreaRegistration.RegisterAllAreas();  
    RegisterGlobalFilters(GlobalFilters.Filters);  
    RegisterRoutes(RouteTable.Routes);  
}

**Example**

[HandleError(View = "Error")]  
public class HomeController : Controller  
{  
    public ActionResult Index()  
    {  
        ViewBag.Message = "Welcome to ASP.NET MVC!";  
        int u = Convert.ToInt32("");// Error line  
        return View();  
    }  
}

**Defining HandleError Attribute at Global Level**  
We can also apply the HandleError Attribute at the global level by registering it in the Global.asax in Application\_Start event.  
  
**Example**

public static void RegisterGlobalFilters(GlobalFilterCollection filters)  
{  
    filters.Add(new HandleErrorAttribute(), 2); //by default added

    filters.Add(new HandleErrorAttribute  
    {  
        View = "Error"  
    }, 1);  
}

protected void Application\_Start()  
{  
    AreaRegistration.RegisterAllAreas();  
    RegisterGlobalFilters(GlobalFilters.Filters);  
    RegisterRoutes(RouteTable.Routes);  
}

**Error Page**

@model System.Web.Mvc.HandleErrorInfo  
@{  
    ViewBag.Title = "Error";  
}  
<h2>  
    Sorry, an error occurred while processing your request.  
    <br />  
    Error :  
</h2>  
<p>  
     @Model.Exception  
</p>

Authorize:-

**Example** − Allowing only authorized users to log in the application.

public class ActionFilterDemoController: Controller {

[Authorize]

public ActionResult Index() {

ViewBag.Message = "This can be viewed only by authenticated users only";

return View();

}

[Authorize(Roles="admin")]

public ActionResult AdminIndex() {

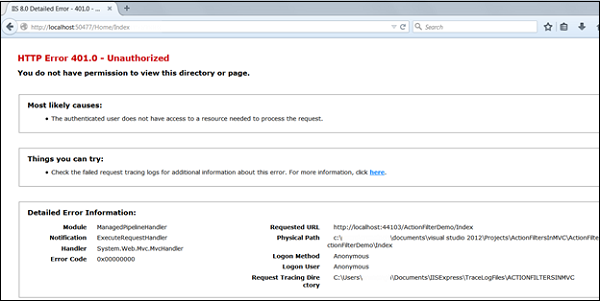
ViewBag.Message = "This can be viewed only by users in Admin role only";

return View();

}

}

With the above code, if you would try to access the application without logging in, it will throw an error similar to the one shown in the following screenshot.



using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace WebApplication2.Controllers

{

[Authorize]

public class HomeController : Controller

{

//

[AllowAnonymous]

public ActionResult Index ()

{

// throw new DivideByZeroException();

return View ();

}

}

}

Using [AllowAnonymous] we can skip the [Authorize] attribute.

What is the difference between TempData keep() and peek() function?

The query is so simple. Why to make its answer complex. Lemme try to explain in simple words. TempData Keep() & Peek(): Most of the preliminary developers know the basic thing about this is: "TempData is used to preserve the data for next request".... i.e. Once the data is read by the view its lost, and not available for next request...... This is true but its half truth. The other half is: The data can be retained & preserved for next request, and here comes Keep() & Peek() Both Keep() & Peek() is used to tell the server to keep this data for next request even if its read by view. This was the basic knowledge.You can read further if wanna know in detail. In detail: There are mainly 4 conditions when data is/can be preserved: 1. Is the data NOT read by view 2. Is the data READ by view 3. data read and its marked as Keep() 4. data marked as Peek() and then its readThe above 4 conditions are quite clear. 1. Is the data NOT read by view: Data will be available for next request if its not read. 2. Is the data READ by view: Data will not be available for next request if its read by view. 3. data read and its marked as Keep(): data will be available for next request if its marked as Keep() after reading the data. 4. data marked as Peek() and then its read: Data will be available for next request if ts marked as Peek() and then read.Note: something we noted here is that, Keep is used after reading the data where as Peek() is used before reading data. e.g.: @TempData["userName "]; <---- read the data TempData.Keep("userName "); <--- marked with Keep() --------- var userName = TempData.Peek("userName ").ToString();Thanks & Enjoy :-)

//second request, read value and is marked for deletion

object value = TempData["value"];

//third request, value is not there as it was deleted at the end of the second request

TempData["value"] == null

The Peek and Keep methods allow you to read the value without marking it for deletion. Say we get back to the first request where the value was saved to TempData.

With Peek you get the value without marking it for deletion with a single call, see [msdn](http://msdn.microsoft.com/en-us/library/system.web.mvc.tempdatadictionary.peek(v=vs.118).aspx):

//second request, PEEK value so it is not deleted at the end of the request

object value = TempData.Peek("value");

//third request, read value and mark it for deletion

object value = TempData["value"];

With Keep you specify a key that was marked for deletion that you want to keep. Retrieving the object and later on saving it from deletion are 2 different calls. See [msdn](http://msdn.microsoft.com/en-us/library/ee703497(v=vs.118).aspx)

//second request, get value marking it from deletion

object value = TempData["value"];

//later on decide to keep it

TempData.Keep("value");

//third request, read value and mark it for deletion

object value = TempData["value"];

You can use Peek when you always want to retain the value for another request. Use Keep when retaining the value depends on additional logic.

**HTTPGet and HTTP Post Method in MVC5:-**

**HTTPGET**

HttpGet method sends data using query string. The data is attached with URL and it is visible to all the users. However, it is not secure but it is fast and quick. It is mostly used when you are not posting any sensitive data to server like username, password, credit card info etc.

**Some Fact about HttpGet Method**

1. **I**t is fast and quick but not secure.
2. **I**t is default method.
3. **B**ecause it attached form data in query string, so the data is visible to other users.
4. **I**t uses stack method for passing form variable.
5. **D**ata is limited to max length of query string.
6. **I**t is very useful when data is not sensitive.
7. **I**t creates URL that is easily readable.
8. **I**t can carry only text data.

**Example**

**StudentModel.cs**

namespace MvcForms.Models

{

    public class StudentModel

    {

        public int Id { get; set; }

        public string Name { get; set; }

    }

}

**Index.cshtml**

<h3><b>Forms - HTTPGET Method</b></h3>

@using (Html.BeginForm("Submit", "Home", FormMethod.Get))

{

    <table>

        <tr>

            <td>Enter ID: </td>

            <td>@Html.TextBox("Id")</td>

        </tr>

        <tr>

            <td>Enter Name: </td>

            <td>@Html.TextBox("Name")</td>

        </tr>

        <tr>

            <td colspan="2"><input type="submit" value="Submit"></td>

        </tr>

    </table>

}

<h4 style="color:purple">

    ID: @ViewBag.Id<br />

    Name: @ViewBag.Name<br />

</h4>

**HomeController**

[HttpGet]

        public ActionResult Submit(int id, string name)

        {

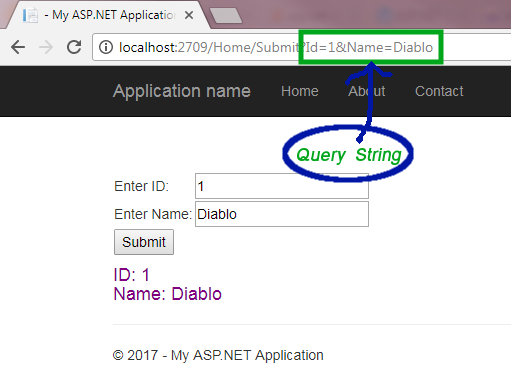
            ViewBag.Id = id;

            ViewBag.Name = name;

            return View("Index");

        }

**Output Image**



**HTTPPOST**

**HTTPPost** method hides information from url and does not bind data to url. It is more secure than HttpGet method but it is slower than HttpGet. It is only useful when you are passing sensitive information to server.

**Some Fact about HttpPost Method**

1. **D**ata is sent via HttpPost method, is not visible to user.
2. **I**t is more secured but slower than HttpGet.
3. **I**t uses Heap method for passing form variable.
4. **I**t has no restriction of passing data and can post unlimited form variables.
5. **I**t is used when sending critical data.
6. **I**t can carry both text and binary data.

**Example**

**StudentModel.cs**

namespace MvcForms.Models

{

    public class StudentModel

    {

        public int Id { get; set; }

        public string Name { get; set; }

    }

}

**Index.cshtml** Form

<h3><b>HTTPPost Method</b></h3>

@using (Html.BeginForm("Submit", "Home", FormMethod.Post))

{

    <table>

        <tr>

            <td>Enter ID: </td>

            <td>@Html.TextBox("Id")</td>

        </tr>

        <tr>

            <td>Enter Name: </td>

            <td>@Html.TextBox("Name")</td>

        </tr>

        <tr>

            <td colspan="2"><input type="submit" value="Submit"></td>

        </tr>

    </table>

}

<h4 style="color:purple">

    ID: @ViewBag.Id<br />

    Name: @ViewBag.Name<br />

</h4>

**HomeController.cs**

[HttpPost]

        public ActionResult Submit(FormCollection fc)

        {

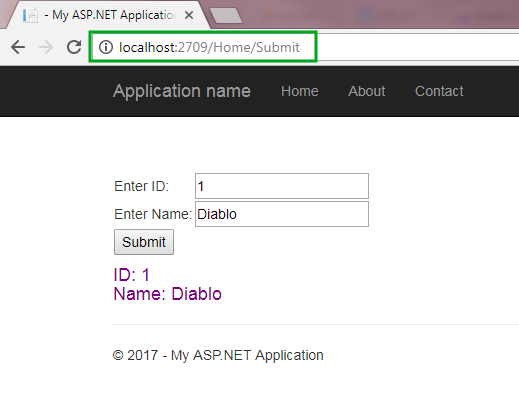
            ViewBag.Id = fc["Id"];

            ViewBag.Name = fc["Name"];

            return View("Index");

        }

**Output**



**Difference between HttpGet and HttpPost Method**

**1.** HTTPGet method is default whereas you need to specify HTTPPost attribute if you are posting data using HTTPPost method.  
**2.** HTTPGet method creates a query string of the name-value pair whereas HTTPPost method passes the name and value pairs in the body of the HTTP request.  
**3.** HTTPGet request has limited length and mostly it is limited to 255 characters long whereas HTTPPost request has no maximum limit.  
**4.** HTTPGet is comparatively faster than HTTPPost. HTTPPost takes extra time in encapsulating the data.  
**5.** HTTPGet can carry only string data whereas HTTPPost can carry both string and binary data.  
**6.** HTTPGet method creates readable url so it can be cached and bookmarked whereas such facility is not available in HTTPPost method.

## Multiple Action Verbs in Asp.Net MVC

We can also apply multiple http verbs using AcceptVerbs attribute. GetAndPostAction method supports both, GET and POST ActionVerbs in the following example:

Example: AcceptVerbs

[AcceptVerbs (HttpVerbs.Post | HttpVerbs.Get)]

public ActionResult GetAndPostAction ()

{

return RedirectToAction ("Index");

}

## ****Method 1: AcceptVerbs Attribute****

We can apply multiple http verbs using AcceptVerbs attribute by using OR operator.

public class dotnethelpersController : Controller  
{  
**[AcceptVerbs(HttpVerbs.Get | HttpVerbs.Post)]**  
public ActionResult EditEmpDetails(int id)  
{  
// retrieve the Details  
}

## ****Method 2: shortcut attributes****

public class dotnethelpersController : Controller

{  
**[HttpGet]**  
public ActionResult EditEmpDetails (int id)  
{  
}

**[HttpPost]**  
public ActionResult EditEmpDetails (string empCode)  
{  
}



# Handling Multiple Submit Buttons on the Same Form in ASP.Net MVC



In the preceding figure we have the three buttons Login, Register and Cancel. Here each button has different functionality. In this way each submit button will post a form to the server but will provide different values of each button.

**Procedure**

1. Create a controller with one action method that accepts two parameters, one is for the model and the other is for determining the status of the button click.
   1. [HttpPost]
   2. **public** ActionResult Index(Login model, **string** command)
   3. {
   4. **if** (command=="Login")
   5. {
   6. // do stuff
   7. **return** RedirectToAction("Home");
   8. }
   9. **else** **if** (command=="Register")
   10. {
   11. // do stuff
   12. ViewBag.msg = "You have Clicked Register button";
   13. **return** View();
   14. }
   16. **else** **if** (command=="Cancel")
   17. {
   18. // do stuff
   19. ViewBag.msg = "You have Clicked Cancel Button";
   20. **return** View();
   21. }
   22. **else**
   23. {
   24. **return** View();
   25. }
   26. }

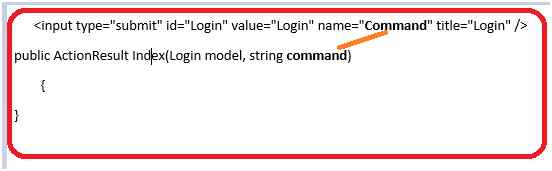
In the preceding code snippet, assume you clicked on the Login button, then the command parameter will have the values Login, null, null respectively.

2.Create a View for the preceding controller.

1. @model MvcMultipleSubmitButtons.Models.Login
2. @{
3. ViewBag.Title = "Index";
4. }
5. <h2>
6. Handling multiple submit buttons **in** MVC </h2>
7. <h5 style="color: Red">@ViewBag.msg</h5>
8. <form action="Home/Index" id="myform" method="post" >
9. //here action name is Index, controller name is Home. So the action path is Home/Index
10. <table>
11. <tr>
12. <td>
13. UserName
14. </td>
15. <td>
16. :
17. </td>
18. <td>@Html.TextBoxFor(m => m.userName)
19. </td>
20. <td>
21. @Html.ValidationMessageFor(m => m.userName)
22. </td>
23. </tr>
24. <tr>
25. <td>
26. Password
27. </td>
28. <td>
29. :
30. </td>
31. <td>@Html.TextBoxFor(m => m.password)
32. </td>
33. <td>
34. @Html.ValidationMessageFor(m => m.password)
35. </td>
36. </tr>
37. </table>
38. <br/>
40. <div style="padding-left: 80px;">
41. <input type="submit" id="Login" value="Login" name="Command" title="Login" />
42. <input type="submit" id="Register" value="Register" name="Command" title="Register" />
43. <input type="submit" value="Cancel" name="Command" title="Cancel" />
45. </div>
46. </form>

You can declare the form tag in another way as in the following:

1. @**using**(Html.BeginForm("Index","Home",FormMethod.Post))
2. {
3. //here action name is Index, controller name is Home and form method is post.
4. }

**Note:**there is a relation between button name and action method parameter. For example, the button name is “Command”, the action parameter name should be “command”.  
  
  
**Figure 2**

You can have different names for each button. So in that case you need to handle it as in the following:

1. <input type="submit" id="Login" value="Login" name="Command1" title="Login" />
2. <input type="submit" id="Register" value="Register" name="Command2" title="Register" />
3. <input type="submit" value="Cancel" name="Command3" title="Cancel" />

**Controller**

1. **public** ActionResult Index(Login model, **string** command1, **string** command2, **string** command3)
2. {
3. // here command1 is for Login, command2 is for Register and command3 is for cancel
4. }

Create a Model class with the name Login.

1. **public** **class** Login
2. {
3. **public** **string** userName { **get**; **set**; }
4. **public** **string** password { **get**; **set**; }
5. }

**Custom Action Filter:**

