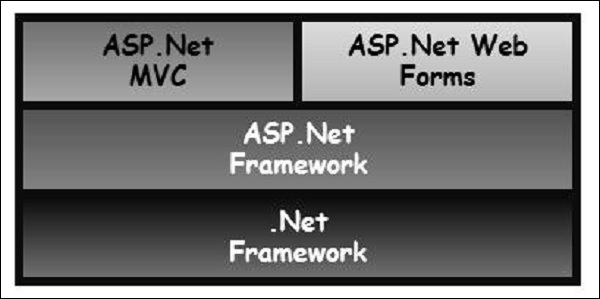
ASP.NET MVC is an open-source software from Microsoft. Its web development framework combines the features of MVC (Model-View-Controller) architecture, the most up-to-date ideas and techniques from Agile development and the best parts of the existing ASP.NET platform.

ASP.NET MVC is not something, which is built from ground zero. It is a complete alternative to traditional ASP.NET Web Forms. It is built on the top of ASP.NET, so developers enjoy almost all the ASP.NET features while building the MVC application.



History

ASP.NET 1.0 was released on January 5, 2002, as part of .Net Framework version 1.0. At that time, it was easy to think of ASP.NET and Web Forms as one and the same thing. ASP.NET has however always supported two layers of abstraction −

* **System.Web.UI** − The Web Forms layer, comprising server controls, ViewState, and so on.
* **System.Web** − It supplies the basic web stack, including modules, handlers, the HTTP stack, etc.

By the time ASP.NET MVC was announced in 2007, the MVC pattern was becoming one of the most popular ways of building web frameworks.

In April 2009, the ASP.NET MVC source code was released under the Microsoft Public License (MS-PL). "ASP.NET MVC framework is a lightweight, highly testable presentation framework that is integrated with the existing ASP.NET features.

Some of these integrated features are master pages and membership-based authentication. The MVC framework is defined in the System.Web.Mvc assembly.

In March 2012, Microsoft had released part of its web stack (including ASP.NET MVC, Razor and Web API) under an open source license (Apache License 2.0). ASP.NET Web Forms was not included in this initiative.

Why ASP.NET MVC?

Microsoft decided to create their own MVC framework for building web applications. The MVC framework simply builds on top of ASP.NET. When you are building a web application with ASP.NET MVC, there will be no illusions of state, there will not be such a thing as a page load and no page life cycle at all, etc.

Another design goal for ASP.NET MVC was to be extensible throughout all aspects of the framework. So when we talk about views, views have to be rendered by a particular type of view engine. The default view engine is still something that can take an ASPX file. But if you don't like using ASPX files, you can use something else and plug in your own view engine.

There is a component inside the MVC framework that will instantiate your controllers. You might not like the way that the MVC framework instantiates your controller, you might want to handle that job yourself. So, there are lots of places in MVC where you can inject your own custom logic to handle tasks.

The whole idea behind using the Model View Controller design pattern is that you maintain a separation of concerns. Your controller is no longer encumbered with a lot of ties to the ASP.NET runtime or ties to the ASPX page, which is very hard to test. You now just have a class with regular methods on it that you can invoke in unit tests to find out if that controller is going to behave correctly.

Benefits of ASP.NET MVC

Following are the benefits of using ASP.NET MVC −

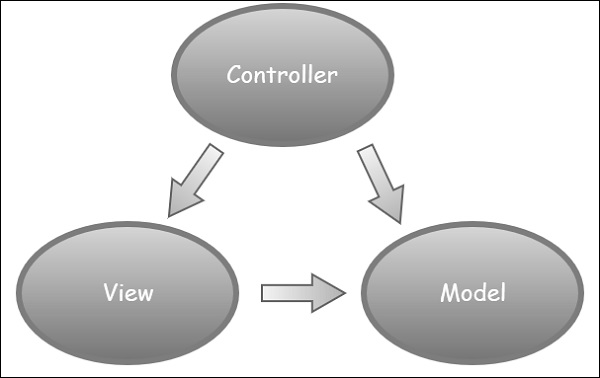
* Makes it easier to manage complexity by dividing an application into the model, the view, and the controller.
* Enables full control over the rendered HTML and provides a clean separation of concerns.
* Direct control over HTML also means better accessibility for implementing compliance with evolving Web standards.
* Facilitates adding more interactivity and responsiveness to existing apps.
* Provides better support for test-driven development (TDD).
* Works well for Web applications that are supported by large teams of developers and for Web designers who need a high degree of control over the application behavior.

The MVC (Model-View-Controller) design pattern has actually been around for a few decades, and it's been used across many different technologies. Everything from Smalltalk to C++ to Java, and now C Sharp and .NET use this design pattern to build a user interface.

Following are some salient features of the MVC pattern −

* Originally it was named Thing-Model-View-Editor in 1979, and then it was later simplified to Model- View-Controller.
* It is a powerful and elegant means of separating concerns within an application (for example, separating data access logic from display logic) and applies itself extremely well to web applications.
* Its explicit separation of concerns does add a small amount of extra complexity to an application’s design, but the extraordinary benefits outweigh the extra effort.

The MVC architectural pattern separates the user interface (UI) of an application into three main parts.



* **The Model** − A set of classes that describes the data you are working with as well as the business logic.
* **The View** − Defines how the application’s UI will be displayed. It is a pure HTML, which decides how the UI is going to look like.
* **The Controller** − A set of classes that handles communication from the user, overall application flow, and application-specific logic.

Idea Behind MVC

The idea is that you'll have a component called the view, which is solely responsible for rendering this user interface whether that be HTML or whether it actually be UI widgets on a desktop application.

The view talks to a model, and that model contains all of the data that the view needs to display. Views generally don't have much logic inside of them at all.

In a web application, the view might not have any code associated with it at all. It might just have HTML and then some expressions of where to take pieces of data from the model and plug them into the correct places inside the HTML template that you've built in the view.

The controller that organizes is everything. When an HTTP request arrives for an MVC application, that request gets routed to a controller, and then it's up to the controller to talk to either the database, the file system, or the model.

# **ASP.NET MVC Version History**

Microsoft had introduced ASP.NET MVC in .NET 3.5, since then lots of new features have been added.

The following table list brief history of ASP.NET MVC.

| MVC Version | Visual Studio | .NET Framework | Released Date | Features |
| --- | --- | --- | --- | --- |
| MVC 1.0 | VS2008 | .Net 3.5 | 13-Mar-2009 | * MVC architecture with webform   engine   * Routing * HTML Helpers * Ajax Helpers * Auto binding |
| MVC 2.0 | VS 2008, | .Net 3.5/4.0 | 10-Mar-2010 | * Area * Asynchronous controller * Html helper methods with lambda expression * DataAnnotations attributes * Client side validation * Custom template * Scaffolding |
| MVC 3.0 | VS 2010 | .Net 4.0 | 13-Jan-2011 | * Unobtrusive javascript validation * Razor view engine * Global filters * Remote validation * Dependency resolver for IoC * ViewBag |
| MVC 4.0 | VS 2010 SP1, VS 2012 | .NET 4.0/4.5 | 15-Aug-2012 | * Mobile project template * Bundling and minification * Support for Windows Azure SDK |
| MVC 5.0 | VS 2013 | .NET 4.5 | 17-oct-2013 | * Authentication filters * Bootstrap support * New scaffolding items * ASP.Net Identity |
| **MVC 5.2** - Current | VS 2013 | .NET 4.5 | 28-Aug-2014 | * Attribute based routing * bug fixes and minor features update |

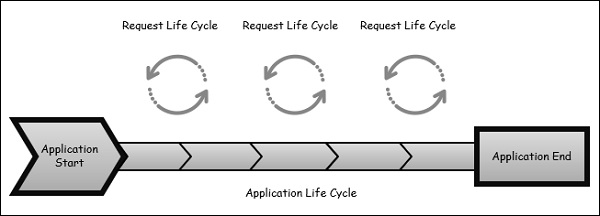
In this chapter, we will discuss the overall MVC pipeline and the life of an HTTP request as it travels through the MVC framework in ASP.NET.

At a high level, a life cycle is simply a series of steps or events used to handle some type of request or to change an application state. You may already be familiar with various framework life cycles, the concept is not unique to MVC.

For example, the ASP.NET webforms platform features a complex page life cycle. Other .NET platforms, like Windows phone apps, have their own application life cycles. One thing that is true for all these platforms regardless of the technology is that understanding the processing pipeline can help you better leverage the features available and MVC is no different.

MVC has two life cycles −

* The application life cycle
* The request life cycle



The Application Life Cycle

The application life cycle refers to the time at which the application process actually begins running IIS until the time it stops. This is marked by the application start and end events in the startup file of your application.

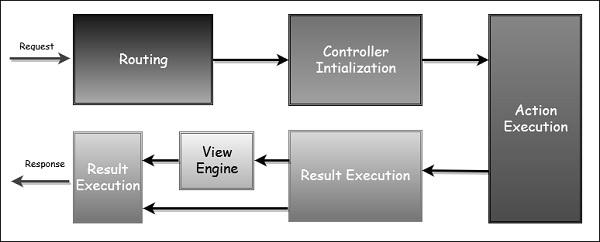
The Request Life Cycle

It is the sequence of events that happen every time an HTTP request is handled by our application.

The entry point for every MVC application begins with routing. After the ASP.NET platform has received a request, it figures out how it should be handled through the URL Routing Module.

Modules are .NET components that can hook into the application life cycle and add functionality. The routing module is responsible for matching the incoming URL to routes that we define in our application.

All routes have an associated route handler with them and this is the entry point to the MVC framework.



The MVC framework handles converting the route data into a concrete controller that can handle requests. After the controller has been created, the next major step is **Action Execution**. A component called the **action invoker** finds and selects an appropriate Action method to invoke the controller.

After our action result has been prepared, the next stage triggers, which is **Result Execution**. MVC separates declaring the result from executing the result. If the result is a view type, the View Engine will be called and it's responsible for finding and rending our view.

If the result is not a view, the action result will execute on its own. This Result Execution is what generates an actual response to the original HTTP request.

# **ASP.NET MVC - Routing**

Routing is the process of directing an HTTP request to a controller and the functionality of this processing is implemented in System.Web.Routing. This assembly is not part of ASP.NET MVC. It is actually part of the ASP.NET runtime, and it was officially released with the ASP.NET as a .NET 3.5 SP1.

**System.Web.Routing** is used by the MVC framework, but it's also used by ASP.NET Dynamic Data. The MVC framework leverages routing to direct a request to a controller. The Global.asax file is that part of your application, where you will define the route for your application.

This is the code from the application start event in Global.asax from the MVC App which we created in the previous chapter.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

using System.Web.Routing;

namespace MVCFirstApp {

public class MvcApplication : System.Web.HttpApplication {

protected void Application\_Start(){

AreaRegistration.RegisterAllAreas();

RouteConfig.RegisterRoutes(RouteTable.Routes);

}

}

}

Following is the implementation of RouteConfig class, which contains one method RegisterRoutes.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

using System.Web.Routing;

namespace MVCFirstApp {

public class RouteConfig

{

public static void RegisterRoutes(RouteCollection routes){

routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");

routes.MapRoute(

name: "Default",

url: "{controller}/{action}/{id}",

defaults: new{ controller = "Home", action = "Index", id = UrlParameter.Optional});

}

}

}

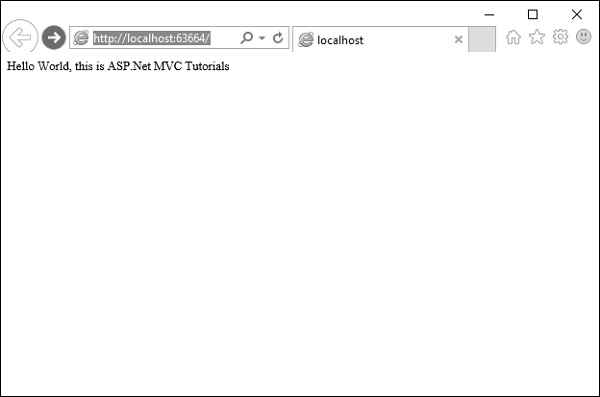
You will define the routes and those routes will map URLs to a specific controller action. An action is just a method on the controller. It can also pick parameters out of that URL and pass them as parameters into the method.

So this route that is defined in the application is the default route. As seen in the above code, when you see a URL arrive in the form of (something)/(something)/(something), then the first piece is the controller name, second piece is the action name, and the third piece is an ID parameter.

## Understanding Routes

MVC applications use the ASP.NET routing system, which decides how URLs map to controllers and actions.

When Visual Studio creates the MVC project, it adds some default routes to get us started. When you run your application, you will see that Visual Studio has directed the browser to port 63664. You will almost certainly see a different port number in the URL that your browser requests because Visual Studio allocates a random port when the project is created.



In the last example, we have added a HomeController, so you can also request any of the following URLs, and they will be directed to the Index action on the HomeController.

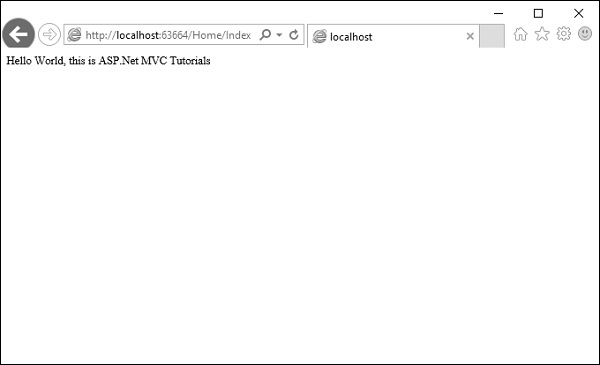
**http://localhost:63664/Home/**

**http://localhost:63664/Home/Index**

When a browser requests http://mysite/ or http://mysite/Home, it gets back the output from HomeController’s Index method.

You can try this as well by changing the URL in the browser. In this example, it is http://localhost:63664/, except that the port might be different.

If you append /Home or /Home/Index to the URL and press ‘Enter’ button, you will see the same result from the MVC application.



As you can see in this case, the convention is that we have a controller called HomeController and this HomeController will be the starting point for our MVC application.

The default routes that Visual Studio creates for a new project assumes that you will follow this convention. But if you want to follow your own convention then you would need to modify the routes.

## Custom Convention

You can certainly add your own routes. If you don't like these action names, if you have different ID parameters or if you just in general have a different URL structure for your site, then you can add your own route entries.

Let’s take a look at a simple example. Consider we have a page that contains the list of processes. Following is the code, which will route to the process page.

routes.MapRoute(

"Process",

"Process/{action}/{id}",

defaults: new{

controller = "Process", action = "List ", id = UrlParameter.Optional}

);

When someone comes in and looks for a URL with Process/Action/Id, they will go to the Process Controller. We can make the action a little bit different, the default action, we can make that a List instead of Index.

Now a request that arrives looks like localhosts/process. The routing engine will use this routing configuration to pass that along, so it's going to use a default action of List.

Following is the complete class implementation.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

using System.Web.Routing;

namespace MVCFirstApp{

public class RouteConfig{

public static void RegisterRoutes(RouteCollection routes){

routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");

routes.MapRoute(

"Process", "Process/{action}/{id}",

defaults: new{

controller = " Process", action = "List ", id =

UrlParameter.Optional});

routes.MapRoute(

name: "Default", url: "{controller}/{action}/{id}",

defaults: new{

controller = "Home", action = "Index", id =

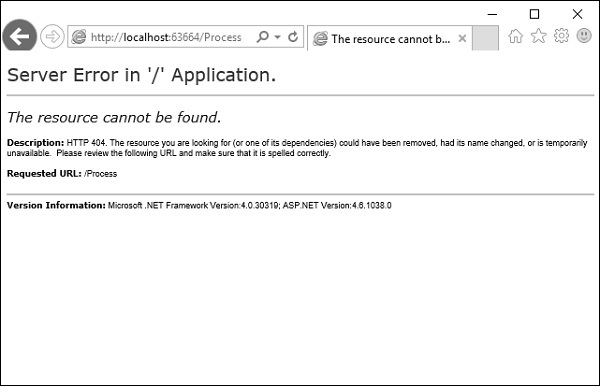
UrlParameter.Optional});

}

}

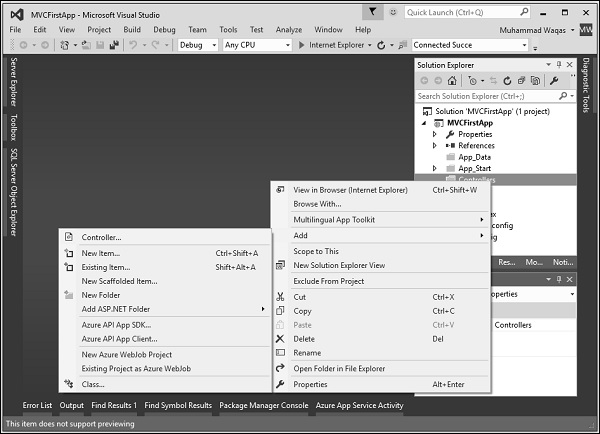
}

**Step 1** − Run this and request for a process page with the following URL **http://localhost:63664/Process**

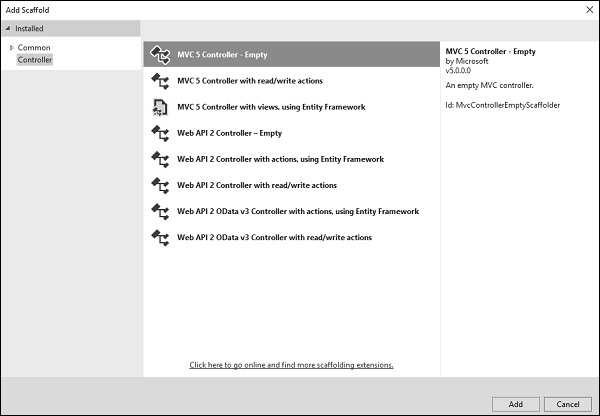


You will see an HTTP 404, because the routing engine is looking for ProcessController, which is not available.

**Step 2** − Create ProcessController by right-clicking on Controllers folder in the solution explorer and select Add → Controller.



It will display the Add Scaffold dialog.



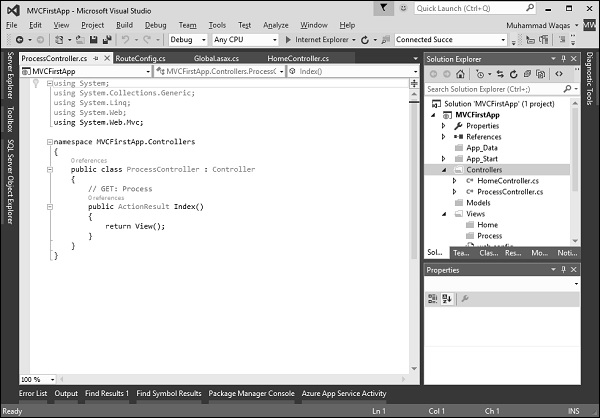
**Step 3** − Select the MVC 5 Controller – Empty option and click ‘Add’ button.

The Add Controller dialog will appear.



**Step 4** − Set the name to ProcessController and click ‘Add’ button.

Now you will see a new C# file ProcessController.cs in the Controllers folder, which is open for editing in Visual Studio as well.



Now our default action is going to be List, so we want to have a List action here instead of Index.

**Step 5** − Change the return type from ActionResult to string and also return some string from this action method using the following code.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace MVCFirstApp.Controllers{

public class ProcessController : Controller{

// GET: Process

public string List(){

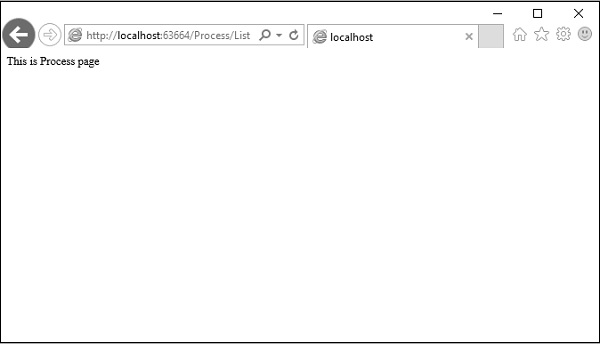
return "This is Process page";

}

}

}

**Step 6** − When you run this application, again you will see the result from the default route. When you specify the following URL, http://localhost:63664/Process/List, then you will see the result from the ProcessController.



# **ASP.NET MVC - Controllers**

Controllers are essentially the central unit of your ASP.NET MVC application. It is the 1st recipient, which interacts with incoming HTTP Request. So, the controller decides which model will be selected, and then it takes the data from the model and passes the same to the respective view, after that view is rendered. Actually, controllers are controlling the overall flow of the application taking the input and rendering the proper output.

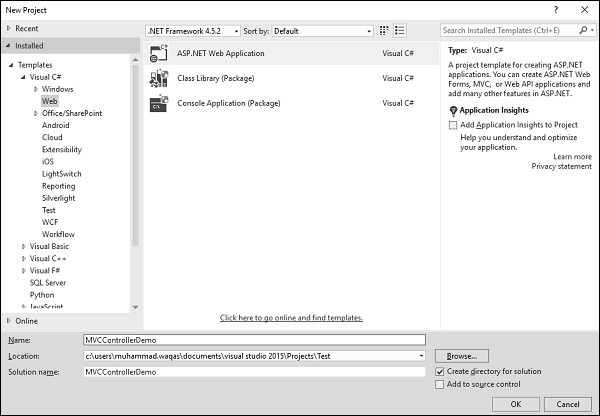
Controllers are C# classes inheriting from System.Web.Mvc.Controller, which is the builtin controller base class. Each public method in a controller is known as an action method, meaning you can invoke it from the Web via some URL to perform an action.

The MVC convention is to put controllers in the Controllers folder that Visual Studio created when the project was set up.

Let’s take a look at a simple example of Controller by creating a new ASP.Net MVC project.

**Step 1** − Open the Visual Studio and click on File → New → Project menu option.

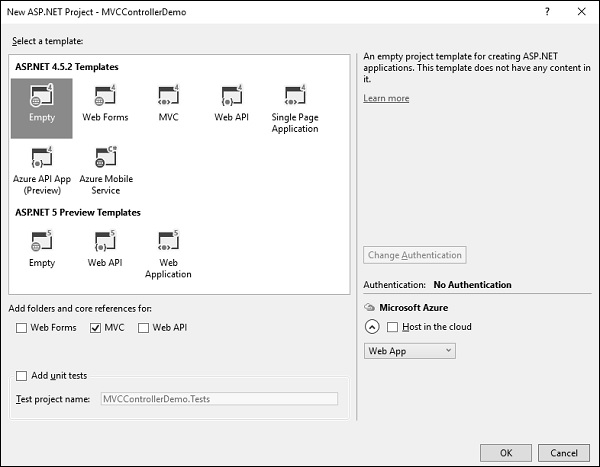
A new Project dialog opens.



**Step 2** − From the left pane, select Templates → Visual C# → Web.

**Step 3** − In the middle pane, select ASP.NET Web Application.

**Step 4** − Enter the project name ‘MVCControllerDemo’ in the Name field and click ok to continue. You will see the following dialog, which asks you to set the initial content for the ASP.NET project.



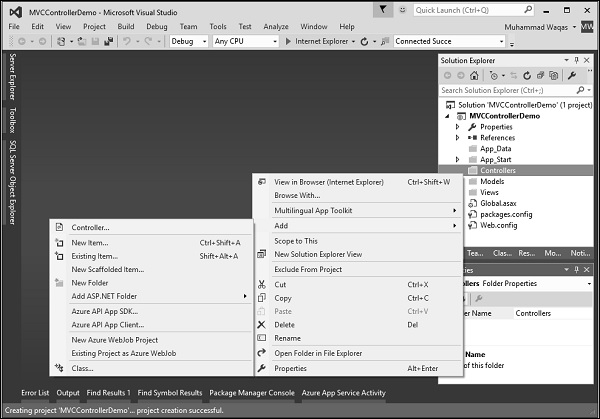
**Step 5** − To keep things simple, select the Empty option and check the MVC checkbox in the ‘Add folders and core references for’ section and click Ok.

It will create a basic MVC project with minimal predefined content.

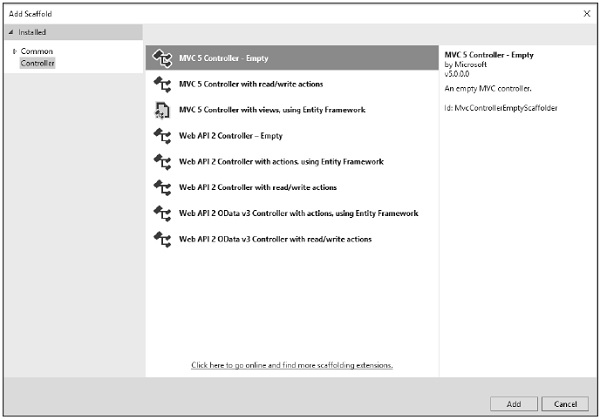
Once the project is created by Visual Studio you will see a number of files and folders displayed in the Solution Explorer window.

Since we have created ASP.Net MVC project from an empty project template, so at the moment, the application does not contain anything to run.

**Step 6** − Add EmployeeController by right-clicking on Controllers folder in the solution explorer. Select Add → Controller.



It will display the Add Scaffold dialog.



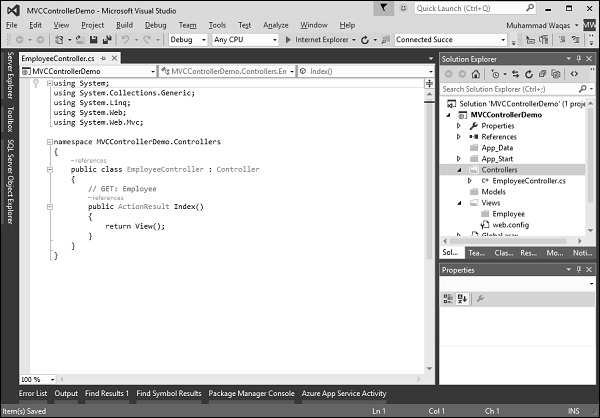
**Step 7** − Select the MVC 5 Controller – Empty option and click ‘Add’ button.

The Add Controller dialog will appear.



**Step 8** − Set the name to EmployeeController and click ‘Add’ button.

You will see a new C# file EmployeeController.cs in the Controllers folder, which is open for editing in Visual Studio as well.



Now, in this application we will add a custom route for Employee controller with the default Route.

**Step 1** − Go to “RouteConfig.cs” file under “App\_Start” folder and add the following route.

routes.MapRoute(

"Employee", "Employee/{name}", new{

controller = "Employee", action = "Search", name =

UrlParameter.Optional });

Following is the complete implementation of RouteConfig.cs file.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

using System.Web.Routing;

namespace MVCControllerDemo {

public class RouteConfig {

public static void RegisterRoutes(RouteCollection routes){

routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");

routes.MapRoute(

"Employee", "Employee/{name}", new{

controller = "Employee", action = "Search", name = UrlParameter.Optional });

routes.MapRoute(

name: "Default", url: "{controller}/{action}/{id}", defaults: new{

controller = "Home", action = "Index", id = UrlParameter.Optional });

}

}

}

Consider a scenario wherein any user comes and searches for an employee, specifying the URL “Employee/Mark”. In this case, Mark will be treated as a parameter name not like Action method. So in this kind of scenario our default route won’t work significantly.

To fetch the incoming value from the browser when the parameter is getting passed, MVC framework provides a simple way to address this problem. It is by using the parameter inside the Action method.

**Step 2** − Change the EmployeeController class using the following code.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace MVCControllerDemo.Controllers {

public class EmployeeController : Controller {

// GET: Employee

public ActionResult Search(string name){

var input = Server.HtmlEncode(name);

return Content(input);

}

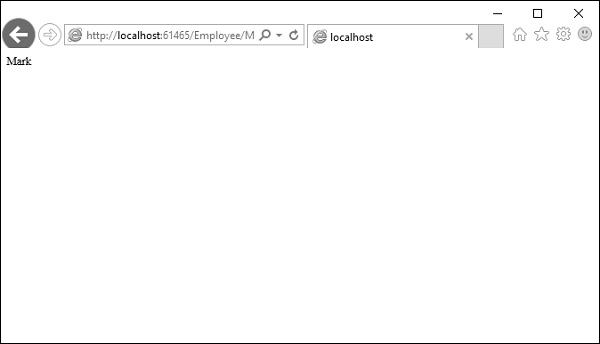
}

}

If you add a parameter to an action method, then the MVC framework will look for the value that matches the parameter name. It will apply all the possible combination to find out the parameter value. It will search in the Route data, query string, etc.

Hence, if you request for /Employee/Mark”, then the MVC framework will decide that I need a parameter with “UserInput”, and then Mark will get picked from the URL and that will get automatically passed.

Server.HtmlEncode will simply convert any kind of malicious script in plain text. When the above code is compiled and executed and requests the following URL **http://localhost:61465/Employee/Mark**, you will get the following output.



As you can see in the above screenshot, Mark is picked from the URL.

# **ASP.NET MVC - Actions**

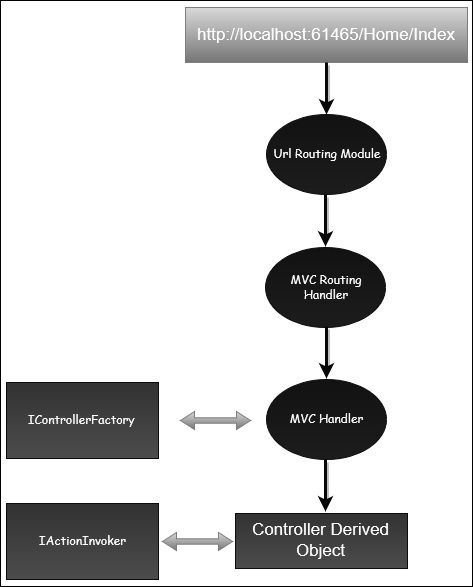
ASP.NET MVC Action Methods are responsible to execute requests and generate responses to it. By default, it generates a response in the form of ActionResult. Actions typically have a one-to-one mapping with user interactions.

For example, enter a URL into the browser, click on any particular link, and submit a form, etc. Each of these user interactions causes a request to be sent to the server. In each case, the URL of the request includes information that the MVC framework uses to invoke an action method. The one restriction on action method is that they have to be instance method, so they cannot be static methods. Also there is no return value restrictions. So you can return the string, integer, etc.

## Request Processing

Actions are the ultimate request destination in an MVC application and it uses the controller base class. Let's take a look at the request processing.

* When a URL arrives, like /Home/index, it is the UrlRoutingModule that inspects and understands that something configured within the routing table knows how to handle that URL.



* The UrlRoutingModule puts together the information we've configured in the routing table and hands over control to the MVC route handler.
* The MVC route handler passes the controller over to the MvcHandler which is an HTTP handler.
* MvcHandler uses a controller factory to instantiate the controller and it knows what controller to instantiate because it looks in the RouteData for that controller value.
* Once the MvcHandler has a controller, the only thing that MvcHandler knows about is IController Interface, so it simply tells the controller to execute.
* When it tells the controller to execute, that's been derived from the MVC's controller base class. The Execute method creates an action invoker and tells that action invoker to go and find a method to invoke, find an action to invoke.
* The action invoker, again, looks in the RouteData and finds that action parameter that's been passed along from the routing engine.

## Types of Action

Actions basically return different types of action results. The ActionResult class is the base for all action results. Following is the list of different kind of action results and its behavior.

|  |  |
| --- | --- |
| **Sr.No.** | **Name and Behavior** |
| 1 | **ContentResult**  Returns a string |
| 2 | **FileContentResult**  Returns file content |
| 3 | **FilePathResult**  Returns file content |
| 4 | **FileStreamResult**  Returns file content |
| 5 | **EmptyResult**  Returns nothing |
| 6 | **JavaScriptResult**  Returns script for execution |
| 7 | **JsonResult**  Returns JSON formatted data |
| 8 | **RedirectToResult**  Redirects to the specified URL |
| 9 | **HttpUnauthorizedResult**  Returns 403 HTTP Status code |
| 10 | **RedirectToRouteResult**  Redirects to different action/different controller action |
| 11 | **ViewResult**  Received as a response for view engine |
| 12 | **PartialViewResult**  Received as a response for view engine |

Let’s have a look at a simple example from the previous chapter in which we have created an EmployeeController.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace MVCControllerDemo.Controllers {

public class EmployeeController : Controller{

// GET: Employee

public ActionResult Search(string name){

var input = Server.HtmlEncode(name);

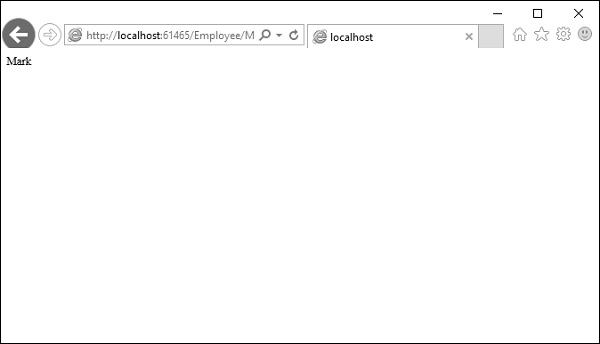
return Content(input);

}

}

}

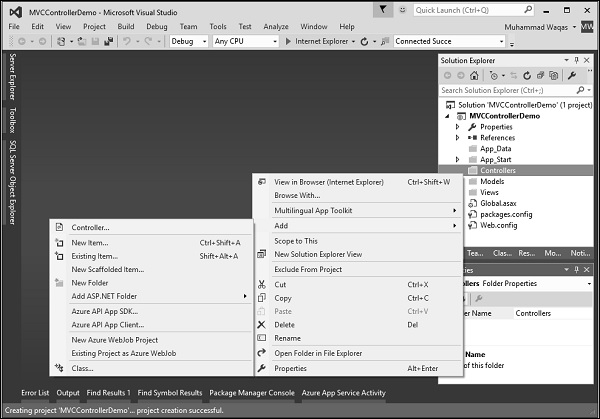
When you request the following URL **http://localhost:61465/Employee/Mark**, then you will receive the following output as an action.



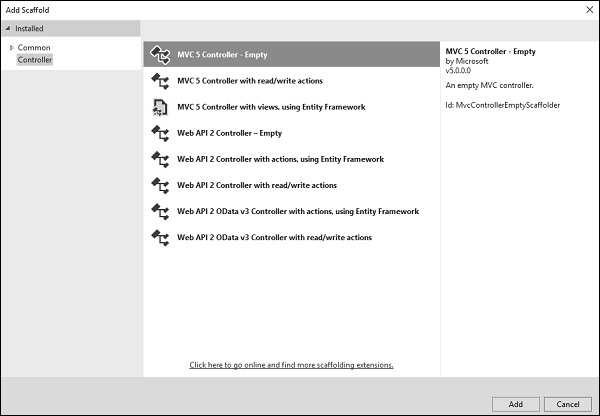
## Add Controller

Let us add one another controller.

**Step 1** − Right-click on Controllers folder and select Add → Controller.



It will display the Add Scaffold dialog.



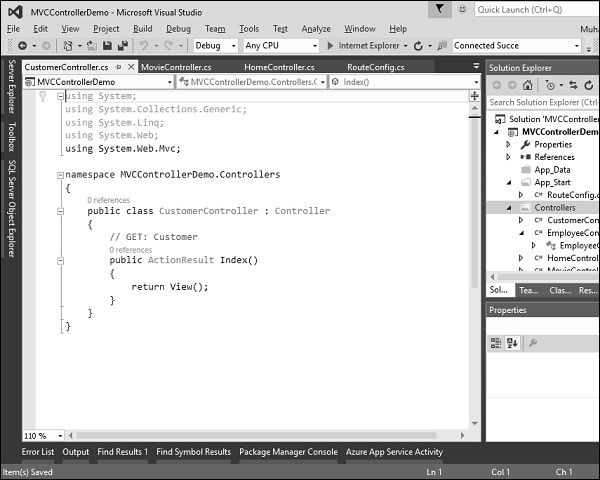
**Step 2** − Select the MVC 5 Controller – Empty option and click ‘Add’ button.

The Add Controller dialog will appear.



**Step 3** − Set the name to CustomerController and click ‘Add’ button.

Now you will see a new C# file ‘CustomerController.cs’ in the Controllers folder, which is open for editing in Visual Studio as well.



Similarly, add one more controller with name HomeController. Following is the HomeController.cs class implementation.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace MVCControllerDemo.Controllers {

public class HomeController : Controller{

// GET: Home

public string Index(){

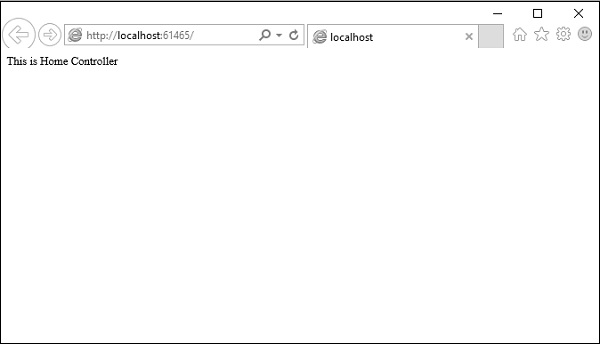
return "This is Home Controller";

}

}

}

**Step 4** − Run this application and you will receive the following output.



**Step 5** − Add the following code in Customer controller, which we have created above.

public string GetAllCustomers(){

return @"<ul>

<li>Ali Raza</li>

<li>Mark Upston</li>

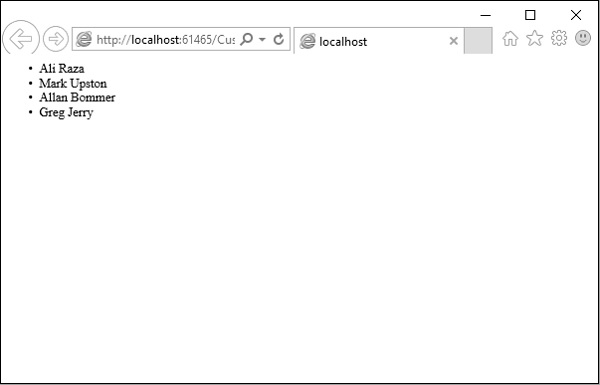
<li>Allan Bommer</li>

<li>Greg Jerry</li>

</ul>";

}

**Step 6** − Run this application and request for **http://localhost:61465/Customer/GetAllCustomers**. You will see the following output.



You can also redirect to actions for the same controller or even for a different controller.

Following is a simple example in which we will redirect from HomeController to Customer Controller by changing the code in HomeController using the following code.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace MVCControllerDemo.Controllers{

public class HomeController : Controller{

// GET: Home

public ActionResult Index(){

return RedirectToAction("GetAllCustomers","Customer");

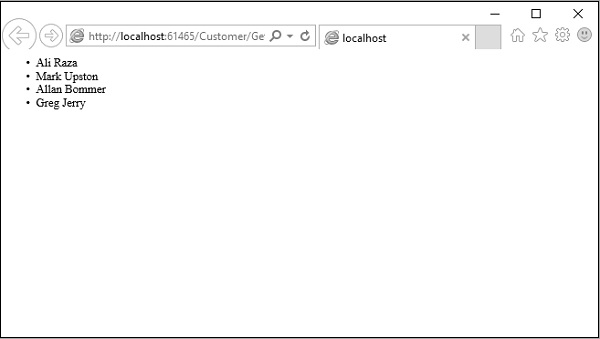
}

}

}

As you can see, we have used the RedirectToAction() method ActionResult, which takes two parameters, action name and controller name.

When you run this application, you will see the default route will redirect it to /Customer/GetAllCustomers



# **ASP.NET MVC - Filters**

Advertisements

[Previous Page](https://www.tutorialspoint.com/asp.net_mvc/asp.net_mvc_actions.htm)

[Next Page](https://www.tutorialspoint.com/asp.net_mvc/asp.net_mvc_selectors.htm)

In ASP.NET MVC, controllers define action methods that usually have a one-to-one relationship with possible user interactions, but sometimes you want to perform logic either before an action method is called or after an action method runs.

To support this, ASP.NET MVC provides filters. **Filters** are custom classes that provide both a declarative and programmatic means to add pre-action and post-action behavior to controller action methods.

## Action Filters

An action filter is an attribute that you can apply to a controller action or an entire controller that modifies the way in which the action is executed. The ASP.NET MVC framework includes several action filters −

* **OutputCache** − Caches the output of a controller action for a specified amount of time.
* **HandleError** − Handles errors raised when a controller action is executed.
* **Authorize** − Enables you to restrict access to a particular user or role.

## Types of Filters

The ASP.NET MVC framework supports four different types of filters −

* **Authorization Filters** − Implements the IAuthorizationFilter attribute.
* **Action Filters** − Implements the IActionFilter attribute.
* **Result Filters** − Implements the IResultFilter attribute.
* **Exception Filters** − Implements the IExceptionFilter attribute.

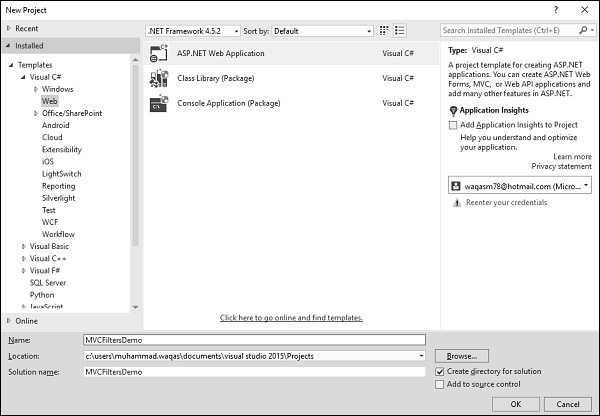
Filters are executed in the order listed above. For example, authorization filters are always executed before action filters and exception filters are always executed after every other type of filter.

Authorization filters are used to implement authentication and authorization for controller actions. For example, the Authorize filter is an example of an Authorization filter.

Let’s take a look at a simple example by creating a new ASP.Net MVC project.

**Step 1** − Open the Visual Studio and click File → New → Project menu option.

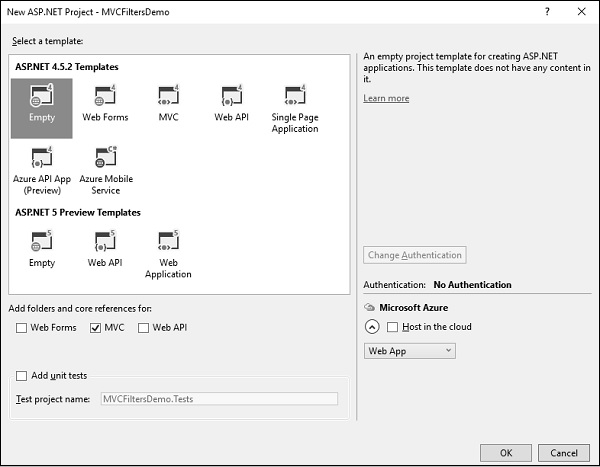
A new Project dialog opens.



**Step 2** − From the left pane, select Templates → Visual C# → Web.

**Step 3** − In the middle pane, select ASP.NET Web Application.

**Step 4** − Enter project name MVCFiltersDemo in the Name field and click ok to continue and you will see the following dialog which asks you to set the initial content for the ASP.NET project.

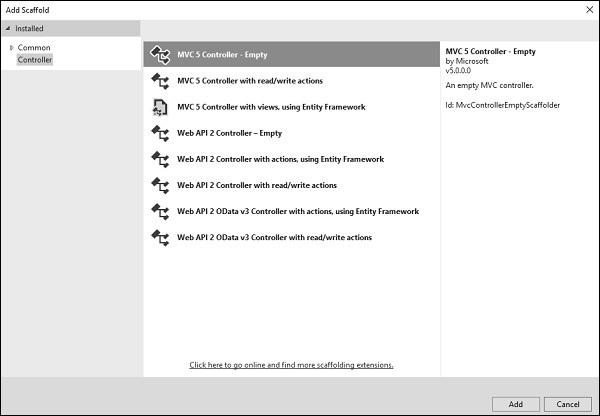


**Step 5** − To keep things simple, select the Empty option and check the MVC checkbox in the ‘Add folders and core references for’ section and click Ok.

It will create a basic MVC project with minimal predefined content.

**Step 6** − To add a controller, right-click on the controller folder in the solution explorer and select Add → Controller.

It will display the Add Scaffold dialog.



**Step 7** − Select the MVC 5 Controller – Empty option and click ‘Add’ button.

The Add Controller dialog will appear.



**Step 8** − Set the name to HomeController and click ‘Add’ button.

You will see a new C# file ‘HomeController.cs’ in the Controllers folder, which is open for editing in Visual Studio as well.

## Apply Action Filter

An action filter can be applied to either an individual controller action or an entire controller. For example, an action filter **OutputCache** is applied to an action named Index() that returns the string. This filter causes the value returned by the action to be cached for 15 seconds.

To make this a working example, let’s modify the controller class by changing the action method called **Index** using the following code.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace MVCFiltersDemo.Controllers {

public class HomeController : Controller{

// GET: Home

[OutputCache(Duration = 15)]

public string Index(){

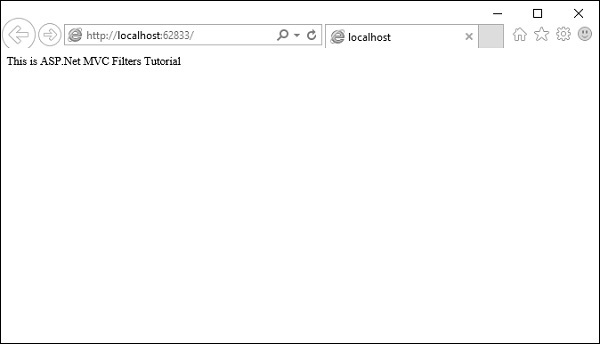
return "This is ASP.Net MVC Filters Tutorial";

}

}

}

When you run this application, you will see that the browser is displaying the result of the Index action method.



Let’s add another action method, which will display the current time.

namespace MVCFiltersDemo.Controllers{

public class HomeController : Controller{

// GET: Home

[OutputCache(Duration = 15)]

public string Index(){

return "This is ASP.Net MVC Filters Tutorial";

}

[OutputCache(Duration = 20)]

public string GetCurrentTime(){

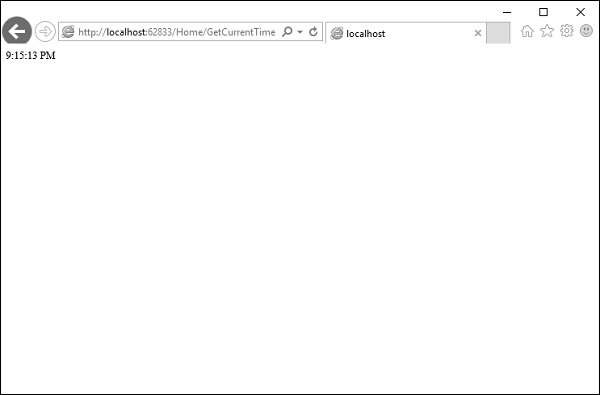
return DateTime.Now.ToString("T");

}

}

}

Request for the following URL, **http://localhost:62833/Home/GetCurrentTime**, and you will receive the following output.

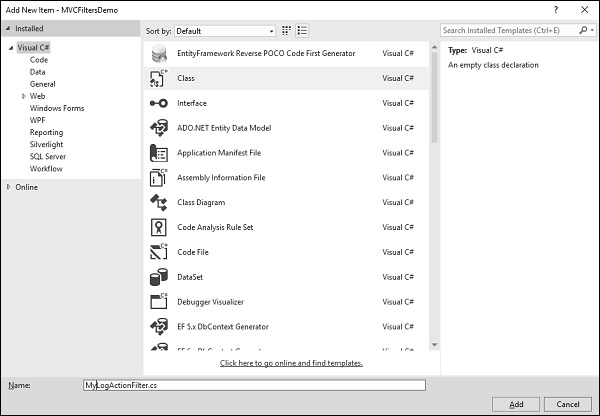


If you refresh the browser, you will see the same time because the action is cached for 20 seconds. It will be updated when you refresh it after 20 seconds.

## Custom Filters

To create your own custom filter, ASP.NET MVC framework provides a base class which is known as ActionFilterAttribute. This class implements both IActionFilter and IResultFilter interfaces and both are derived from the Filter class.

Let’s take a look at a simple example of custom filter by creating a new folder in your project with ActionFilters. Add one class for which right-click on ActionFilters folder and select Add → Class.



Enter ‘MyLogActionFilter’ in the name field and click ‘Add’ button.

This class will be derived from the **ActionFilterAttribute**, which is a base class and overrides the following method. Following is the complete implementation of MyLogActionFilter.

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Linq;

using System.Web;

using System.Web.Mvc;

using System.Web.Routing;

namespace MVCFiltersDemo.ActionFilters {

public class MyLogActionFilter : ActionFilterAttribute{

public override void OnActionExecuting(ActionExecutingContext filterContext){

Log("OnActionExecuting", filterContext.RouteData);

}

public override void OnActionExecuted(ActionExecutedContext filterContext){

Log("OnActionExecuted", filterContext.RouteData);

}

public override void OnResultExecuting(ResultExecutingContext filterContext){

Log("OnResultExecuting", filterContext.RouteData);

}

public override void OnResultExecuted(ResultExecutedContext filterContext){

Log("OnResultExecuted", filterContext.RouteData);

}

private void Log(string methodName, RouteData routeData){

var controllerName = routeData.Values["controller"];

var actionName = routeData.Values["action"];

var message = String.Format(

"{0} controller:{1} action:{2}", methodName, controllerName, actionName);

Debug.WriteLine(message, "Action Filter Log");

}

}

}

Let us now apply the log filter to the HomeController using the following code.

using MVCFiltersDemo.ActionFilters;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace MVCFiltersDemo.Controllers {

[MyLogActionFilter]

public class HomeController : Controller{

// GET: Home

[OutputCache(Duration = 10)]

public string Index(){

return "This is ASP.Net MVC Filters Tutorial";

}

[OutputCache(Duration = 10)]

public string GetCurrentTime(){

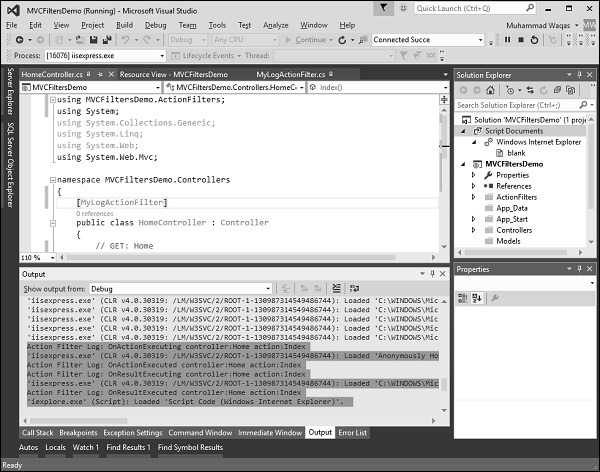
return DateTime.Now.ToString("T");

}

}

}

Run the application and then observe the output window.



As seen in the above screenshot, the stages of processing the action are logged to the Visual Studio output window.

# **ASP.NET MVC - Selectors**

Advertisements

[Previous Page](https://www.tutorialspoint.com/asp.net_mvc/asp.net_mvc_filters.htm)

[Next Page](https://www.tutorialspoint.com/asp.net_mvc/asp.net_mvc_views.htm)

Action selectors are attributes that can be applied to action methods and are used to influence which action method gets invoked in response to a request. It helps the routing engine to select the correct action method to handle a particular request.

It plays a very crucial role when you are writing your action methods. These selectors will decide the behavior of the method invocation based on the modified name given in front of the action method. It is usually used to alias the name of the action method.

There are three types of action selector attributes −

* ActionName
* NonAction
* ActionVerbs

## ActionName

This class represents an attribute that is used for the name of an action. It also allows developers to use a different action name than the method name.

Let’s take a look at a simple example from the last chapter in which we have HomeController containing two action methods.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace MVCFiltersDemo.Controllers {

public class HomeController : Controller{

// GET: Home

public string Index(){

return "This is ASP.Net MVC Filters Tutorial";

}

public string GetCurrentTime(){

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

}

}

}

Let’s apply the the ActionName selector for GetCurrentTime by writing [ActionName("CurrentTime")] above the GetCurrentTime() as shown in the following code.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace MVCFiltersDemo.Controllers {

public class HomeController : Controller{

// GET: Home

public string Index(){

return "This is ASP.Net MVC Filters Tutorial";

}

[ActionName("CurrentTime")]

public string GetCurrentTime(){

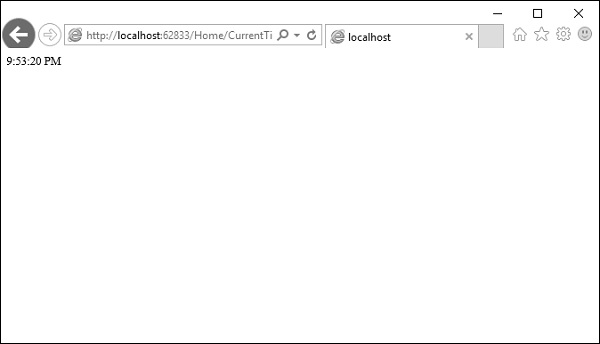
return DateTime.Now.ToString("T");

}

}

}

Now run this application and enter the following URL in the browser **http://localhost:62833/Home/CurrentTime**, you will receive the following output.



You can see that we have used the CurrentTime instead of the original action name, which is GetCurrentTime in the above URL.

## NonAction

NonAction is another built-in attribute, which indicates that a public method of a Controller is not an action method. It is used when you want that a method shouldn’t be treated as an action method.

Let’s take a look at a simple example by adding another method in HomeController and also apply the NonAction attribute using the following code.

using MVCFiltersDemo.ActionFilters;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace MVCFiltersDemo.Controllers {

public class HomeController : Controller{

// GET: Home

public string Index(){

return "This is ASP.Net MVC Filters Tutorial";

}

[ActionName("CurrentTime")]

public string GetCurrentTime(){

return TimeString();

}

[NonAction]

public string TimeString(){

return "Time is " + DateTime.Now.ToString("T");

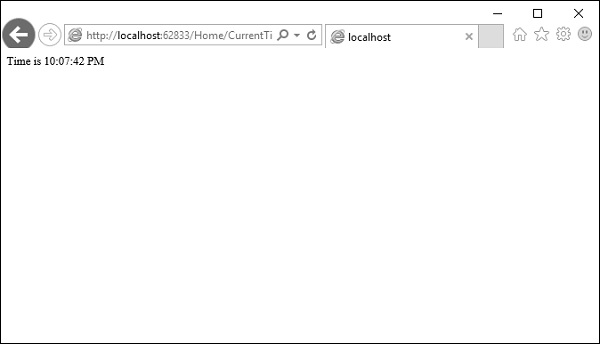
}

}

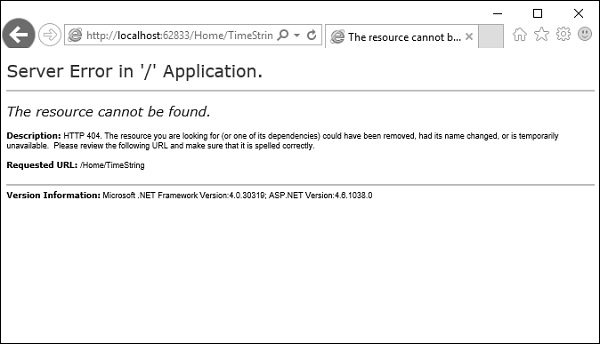
}

The new method TimeString is called from the GetCurrentTime() but you can’t use it as action in URL.

Let’s run this application and specify the following URL **http://localhost:62833/Home/CurrentTime** in the browser. You will receive the following output.



Let us now check the **/TimeString** as action in the URL and see what happens.



You can see that it gives ‘404—Not Found’ error.

## ActionVerbs

Another selector filter that you can apply is the ActionVerbs attributes. So this restricts the indication of a specific action to specific HttpVerbs. You can define two different action methods with the same name but one action method responds to an HTTP Get request and another action method responds to an HTTP Post request.

MVC framework supports the following ActionVerbs.

* HttpGet
* HttpPost
* HttpPut
* HttpDelete
* HttpOptions
* HttpPatch

Let’s take a look at a simple example in which we will create EmployeeController.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace MVCControllerDemo.Controllers {

public class EmployeeController : Controller{

// GET: Employee

public ActionResult Search(string name = “No name Entered”){

var input = Server.HtmlEncode(name);

return Content(input);

}

}

}

Now let’s add another action method with the same name using the following code.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace MVCControllerDemo.Controllers {

public class EmployeeController : Controller{

// GET: Employee

//public ActionResult Index()

//{

// return View();

//}

public ActionResult Search(string name){

var input = Server.HtmlEncode(name);

return Content(input);

}

public ActionResult Search(){

var input = "Another Search action";

return Content(input);

}

}

}

When you run this application, it will give an error because the MVC framework is unable to figure out which action method should be picked up for the request.

Let us specify the HttpGet ActionVerb with the action you want as response using the following code.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace MVCControllerDemo.Controllers {

public class EmployeeController : Controller{

// GET: Employee

//public ActionResult Index()

//{

// return View();

//}

public ActionResult Search(string name){

var input = Server.HtmlEncode(name);

return Content(input);

}

[HttpGet]

public ActionResult Search(){

var input = "Another Search action";

return Content(input);

}

}

}

When you run this application, you will receive the following output.

