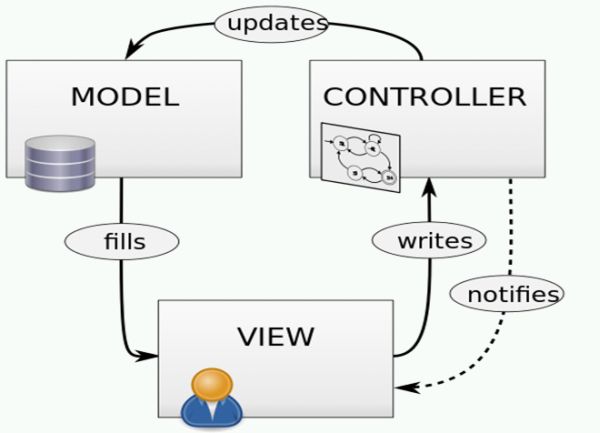
**In Details**

**What is MVC**  
MVC stands for Model View Controller. It is an architectural pattern to develop enterprise application. It is not a Microsoft technology, whether it is a common design pattern that is used to create web application. It is used to create structure user-oriented application. MVC is most similar to MVVM [Model View View-Model] but not completely.

* M – Model
* V – View
* C – Controller

The following is the detailed description:

1. **Model**  
   Model is entity class that encapsulates the properties and the behaviors of domain. It also contains the business logic. MVC components use Model to perform logic. Model is basically represented to data objects like it can be business logic, validation or data access logic. The view and the controller both can access the Model.
2. **View**  
   It’s a GUI. View is nothing but a User Interface. Basically user interacts with a View. View is used to display data, add data or modify data. View is rendered on browser. It renders the data in html form. So, we can say, a view is a result that a user can see while interacting with web application.
3. **Controller**  
   It handles the request coming from the users and as per data it connects with Model and perform some logic and display data on View. So, basically it validates the user request and process data as per requirement.  
     
   

**What is ASP.NET MVC?**  
ASP.NET MVC is a framework that supports MVC Design Pattern to develop different kinds of application. It is Microsoft Open Source Framework and also light weight to other framework. Now a days it is the most popular framework in web development because it supports and provides us many features such as Routing, Unit Test Capability, Bundling and Minification, Separation of the code, etc. Loose coupling is the main concern to use ASP.NET MVC.  
  
So, ASP.NET is A web application framework that is designed with separations of concerns and testability in mind.  
  
**Asp.Net MVC Advantages**It provides us numbers of benefits as compared to other programming approach. The following are the features which force us to use ASP.NET MVC.

* **Development**It supports Loose Coupling that means we can create a project with multiple components; these components will not be dependent to each other. If there is some complication in one component it will not be effect the development of other component. So, this feature makes development very easy for developers.
* **Testability**It is a Test Driven Development (TDD). As you know MVC is loosely coupled so we can test a particular component without affecting other component. If one component is going to be tested and it is dependent to other component then we can pass input as mock component and test the particular component. So, Mocking makes testing very easy.
* **Performance**ASP.NET MVC does not support normal web form features like ViewState, Postback etc. So, there is no need to maintain the state of control. It makes it faster in performance.
* **Full Html Control**It also does not support any server control and only supports pure html input control that makes it faster to normal web form.
* **Extensibility**It supports multiple view engines like aspx, razor. If you want to create your own engine, you can create.

**Model View Controller**

* MVC is a software architecture, which separates the logic from the user interface. This is done by separating the application into three parts Model, View and Controller. MVC is separation of concern. MVC is also a design pattern.

**Model**

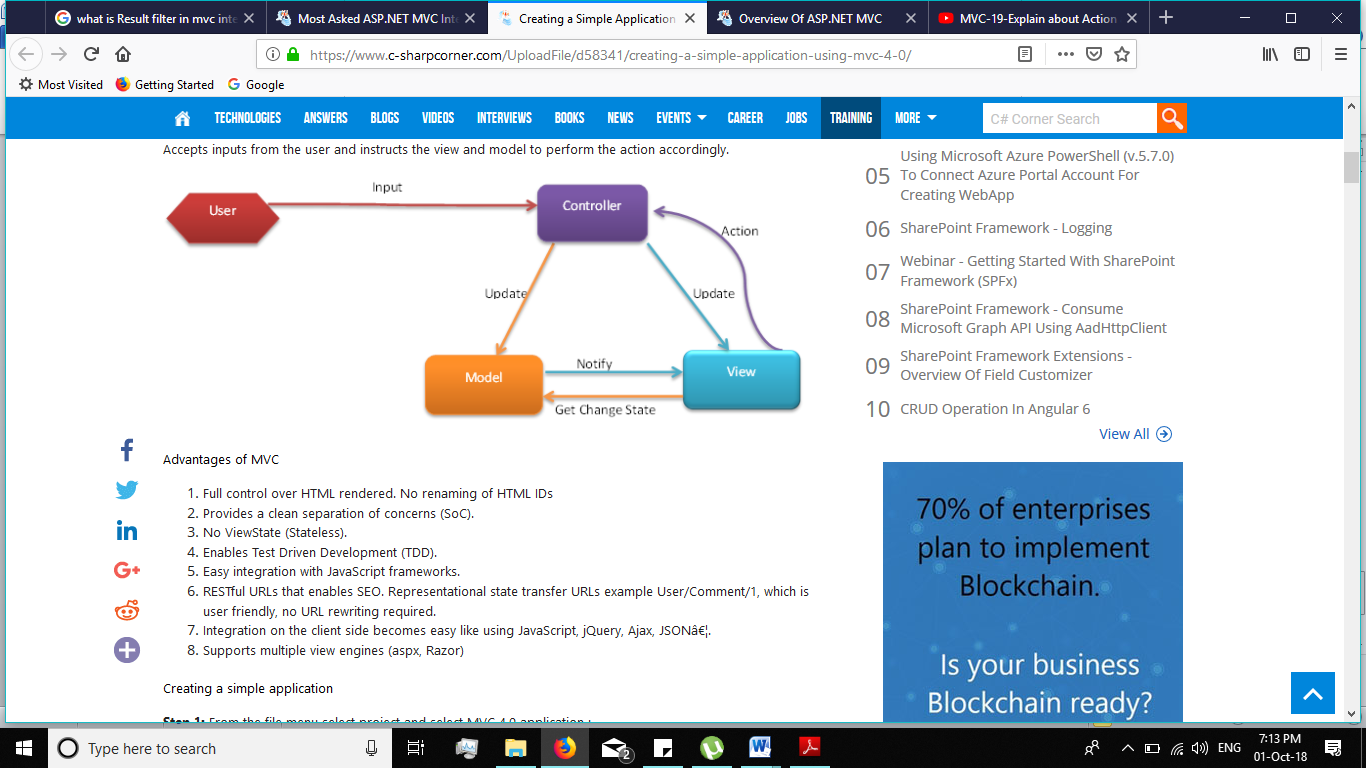
* Represents the logical behavior of data in the application. It represents applications business logic. Model notifies view and controller whenever there is change in state.

**View**

* Provides the user interface of the application. A view is the one which transforms the state of the model into readable HTML.

**Controller**

* Accepts inputs from the user and instructs the view and model to perform the action accordingly.



**Advantages of MVC**

1. Full control over HTML rendered. No renaming of HTML IDs
2. Provides a clean separation of concerns (SoC).
3. No ViewState (Stateless).
4. Enables Test Driven Development (TDD).
5. Easy integration with JavaScript frameworks.
6. RESTful URLs that enables SEO. Representational state transfer URLs example User/Comment/1, which is user friendly, no URL rewriting required.
7. Integration on the client side becomes easy like using JavaScript, jQuery, Ajax, JSONâ€¦.
8. Supports multiple view engines (aspx, Razor)

**What happens in a normal ASP.NET application?**

1. In an ASP.NET application each ASP.NET page inherits from System.Web.UI.Page that implements the IHTTPHandler interface.
2. This interface has an abstract method ProcessRequest() and hence is implemented in the Page class. This method is called when you request a page.
3. The ProcessRequest() method takes an instance of HttpContext and is responsible for processing the request and generating the response.

So in an ASP.NET application it is so straight forward, you request a page with an URL like http://mysite/default.aspx. Then, ASP.NET searches for that page on the disk, executes the ProcessRequest() method, generates and renders the response. There is a one-to-one mapping between URL and physical page.  
  
**The ASP.NET MVC Process**  
In a MVC application, no physical page exists for a specific request. All the requests are routed to a special class called the Controller. The controller is responsible for generating the response and sending the content back to the browser. Also, there is a many-to-one mapping between URL and controller.  
  
When you request a MVC application, you are directly calling the action method of a controller.  
  
When you request http://mysite/Controller1/method1, you are actually calling Controller1's method1. We will see how our request is routing to an ActionMethod of a controller.  
  
**The procedure involved is:**

1. An instance of the RouteTable class is created on application start. This happens only once when the application is requested for the first time.
2. The UrlRoutingModule intercepts each request, finds a matching RouteData from a RouteTable and instantiates a MVCHandler (an HttpHandler).
3. The MVCHandler creates a DefaultControllerFactory (you can create your own controller factory also). It processes the RequestContext and gets a specific controller (from the controllers you have written). Creates a ControllerContext. es the controller a ControllerContext and executes the controller.
4. Gets the ActionMethod from the RouteData based on the URL. The Controller Class then builds a list of parameters (to to the ActionMethod) from the request.
5. The ActionMethod returns an instance of a class inherited from the ActionResult class and the View Engine renders a view as a web page.

**Now, let's understand in detail**

1. Every ASP.NET MVC application has a RouteTable class. This RouteTable is responsible for mapping the MVC requests to a specific controller's ActionMethod.   
     
   Whenever the application starts, the Application\_Start event will be fired and it will call the RegisterRoutes() with the collection of all the available routes of the MVC application as a parameter that will add the routes to the Routes property of the System.Web.Routing.RouteTable class. The Routes property is of type RouteCollection.

**Global.asax**

1. public class MvcApplication : System.Web.HttpApplication
2. {
3. protected void Application\_Start()
4. {
5. WebSecurity.InitializeDatabaseConnection("DefaultConnection", "UserProfile", "UserId", "UserName", autoCreateTables:true);
6. AreaRegistration.RegisterAllAreas();
7. WebApiConfig.Register(GlobalConfiguration.Configuration);
8. FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
9. RouteConfig.RegisterRoutes(RouteTable.Routes);
10. BundleConfig.RegisterBundles(BundleTable.Bundles);
11. }
12. }

 **RouteConfig.cs**

1. public class RouteConfig
2. {
3. public static void RegisterRoutes(RouteCollection routes)
4. {
5. routes.IgnoreRoute("{resource}.axd/{\*pathInfo}"); // Ignore route ending with axd
6. routes.MapRoute(
7. name: "Default", // route name
8. url: "{controller}/{action}/{id}", // Url pattern
9. defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional } // create a default route
10. );
11. }
12. }

The MapRoute() actually creates a default route. Adds all routes to the RouteTable and associates the RouteHandlers with the routes.  
  
Now, all the mapped routes are stored as a RouteCollection in the Routes property of the RouteTable class.   
  
**NOTE:** The RouteTable class has a Routes property that holds a collection of objects that derive from the [RouteBase](http://msdn.microsoft.com/en-us/library/system.web.routing.routebase(v=vs.110).aspx) class. The RouteCollection class is derived from Collection<RouteBase>. Hence RegisterRoutes() is taking an object of RouteCollection.  
  
When an ASP.NET MVC application handles a request, the application iterates through the collection of routes in the [Routes](http://msdn.microsoft.com/en-us/library/system.web.routing.routetable.routes(v=vs.110).aspx) property to find the route that matches the format of the URL requested. The application uses the first route that it finds in the collection that matches the URL. So the most specific route should be added first then the general ones.

 Whenever you request an ASP.NET MVC application, the request is intercepted by the UrlRoutingModule (an HTTP Module) and:

1. The UrlRoutingModule wraps up the current HttpContext (including the URL, form parameters, query string parameters and cookies associated with the current request) in an HttpContextWrapper object as below.  
     
   **NOTE:** The **HttpContext** class has no base class and isn't virtual and hence is unusable for testing (it cannot be mocked). The **HttpContextBase** class is a (from C# 3.5) replacement to HttpContext. Since it is abstract, it is mockable. It is concretely implemented by **HttpContextWrapper**. To create an instance of HttpContextBase in a normal web page, we use:  
     
   **New HttpContextWrapper(HttpContext.Current).** 
   1. private void OnApplicationPostResolveRequestCache(object sender, EventArgs e)
   2. {
   3. HttpApplication application = (HttpApplication) sender;
   4. HttpContextBase context = new HttpContextWrapper(application.Context);
   5. this.PostResolveRequestCache(context);
   6. }

Now, the module sends this HTTPContextBase object to the PostResolveRequestCache() method.

1. Based on the HttpBaseContext object, the postResolveRequestCache() will return the correct RouteData from the RouteTable (that was created in the previous Step #1).
2. If the UrlRoutingModule successfully retrieves a RouteData object then it creates a RequestContext object that represents the current HttpContext and RouteData.
3. The UrlRoutingModule then instantiates a new HttpHandler based on the RouteTable and es the RequestContext (created in Step c) to the new handler's constructor.
4. For an ASP.NET MVC application, the handler returned from the RouteTable will always be an MvcHandler. This MVCHandler implements an IHTTPHandler interface and hence the ProcessRequest() method.
5. Finally, it will call the RemapHandler() method that will set the MVCHandler just obtained to be the Current HTTP Handler.
   1. public virtual void PostResolveRequestCache(HttpContextBase context)
   2. {
   3. RouteData routeData = this.RouteCollection.GetRouteData(context);
   4. if (routeData != null)
   5. {
   6. IRouteHandler routeHandler = routeData.RouteHandler;
   7. if (routeHandler == null)
   8. {
   9. throw new InvalidOperationException(string.Format(CultureInfo.CurrentCulture,
   10. SR.GetString("UrlRoutingModule\_NoRouteHandler"), new object[0]));
   11. }
   12. if (!(routeHandler == StopRoutingHandler))
   13. {
   14. RequestContext requestContext = new RequestContext(context, routeData);
   15. context.Request.RequestContext = requestContext;
   16. IHttpHandler httpHandler = routeHandler.GetHttpHandler(requestContext);
   18. if (httpHandler == null)
   19. {
   20. throw new InvalidOperationException(
   21. string.Format(CultureInfo.CurrentUICulture,
   22. SR.GetString("UrlRoutingModule\_NoHttpHandler"),
   23. new object[] { routeHandler.GetType() }));
   24. }
   26. if (httpHandler == UrlAuthFailureHandler)
   27. {
   28. if (!FormsAuthenticationModule.FormsAuthRequired)
   29. {
   30. throw new HttpException(0x191,
   31. SR.GetString("Assess\_Denied\_Description3"));
   32. }
   33. UrlAuthorizationModule.ReportUrlAuthorizationFailure(HttpContext.Current,this);
   34. }
   35. else
   36. {
   37. context.RemapHandler(httpHandler);
   38. }
   39. }
   40. }
   41. }

 MVCHandler is also inherited from the IHTTPAsyncHandler hence implements the ProcessRequest() method. When MVCHandler executes, it calls the ProcessRequest() method that in turn calls the ProcessRequestInit() method.  
  
The ProcessRequestInit() method creates a ControllerFactory and a Controller. The Controller is created from a ControllerFactory. There is a ControllerBuilder class that will set the ControllerFactory.   
  
**NOTE:** By default it will be DefaultControllerFactory. But you can create your own ControllerFactory as well.   
  
By implementing the IControllerFactory interface and then adding the following code to the Application\_Start event in the gloabal.asax.  
  
**ControllerBuilder.Current.SetControllerFactory(typeof(NewFactory))**  
Now, the NewFactory will be used instead of the DefaultControllerFactory .  
  
The RequestContext and the name of the Contoller (from the URL) will be ed to CreateController() method to get the specific Contoller (that you have written).  
  
Next, a ControllerContext object is constructed from the RequestContext and the controller using the method GetContollerInstance().

1. private void ProcessRequestInit(HttpContextBase httpContext, out IController controller, out IControllerFactory factory)
2. {
3. bool? isRequestValidationEnabled = ValidationUtility.IsValidationEnabled(HttpContext.Current);
4. if (isRequestValidationEnabled == true)
5. {
6. ValidationUtility.EnableDynamicValidation(HttpContext.Current);
7. }
8. AddVersionHeader(httpContext);
9. RemoveOptionalRoutingParameters();
10. // Get the controller type
11. string controllerName = RequestContext.RouteData.GetRequiredString("controller");
12. // Instantiate the controller and call Execute
13. factory = ControllerBuilder.GetControllerFactory();
14. controller = factory.CreateController(RequestContext, controllerName);
15. if (controller == null)
16. {
17. throw new InvalidOperationException(
18. String.Format(CultureInfo.CurrentCulture, MvcResources.ControllerBuilder\_FactoryReturnedNull,factory.GetType(),controllerName));
19. }
20. }
21. public virtual IController CreateController(RequestContext requestContext, string controllerName)
22. {
23. if (requestContext == null)
24. {
25. throw new ArgumentNullException("requestContext");
26. }
27. if (string.IsNullOrEmpty(controllerName))
28. {
29. throw new ArgumentException(MvcResources.Common\_NullOrEmpty, "controllerName");
30. }
31. Type controllerType = this.GetControllerType(requestContext, controllerName);
32. return this.GetControllerinstance(requestContext, controllerType);
33. }

 Our Controller inherits from the Controller class that inherits from ControllerBase that implements the Icontroller interface. The Icontroller interface has an Execute() abstract method that is implemented in the ControllerBase class.

1. public abstract class ControllerBase : Icontroller
2. {
3. protected virtual void Execute(RequestContext requestContext)
4. {
5. if (requestContext == null)
6. {
7. throw new ArgumentNullException("requestContext");
8. }
9. if (requestContext.HttpContext == null)
10. {
11. throw new ArgumentException(              MvcResources.ControllerBase\_CannotExecuteWithNullHttpContext,
12. "requestContext");
13. }
14. VerifyExecuteCalledOnce();
15. Initialize(requestContext);
16. using (ScopeStorage.CreateTransientScope())
17. {
18. ExecuteCore();
19. }
20. }
21. protected abstract void ExecuteCore();
22. //Other stuffs here
23. }
24. The ViewBag, ViewData, TempData and so on properties of the ControllerBase class is initialized. These properties are used for ing data from the View to the Controller or vice-versa or among action methods.
25. The Execute() method of the ControllerBase class is executed that calls the ExecuteCore() abstract method. ExecuteCore() is implemented in the Controller class.
    1. protected override void ExecuteCore()
    2. {
    3. PossiblyLoadTempData();
    4. try
    5. {
    6. string actionName = RouteData.GetRequiredString("action");
    7. if (!ActionInvoker.InvokeAction(ControllerContext, actionName))
    8. {
    9. HandleUnknownAction(actionName);
    10. }
    11. }
    12. finally
    13. {
    14. PossiblySaveTempData();
    15. }
    16. }
26. The ExecuteCore() method gets the Action name from the RouteData based on the URL.
27. The ExecuteCore() method then calls the InvokeAction() method of the ActionInvoker class. This builds a list of parameters from the request. This list of parameters are ed as method parameters to the ActionMethod that is executed. Here the Descriptor objects viz.ControllerDescriptor and ActionDescriptor, that provide information on the controller (like name, type, actions) and Action (name, parameter and controller) respectively play a major role. Now you have your Controller name and Action name.

This controller class is something that you wrote. So one of the methods that you wrote for your controller class is invoked.   
  
**NOTE:** controller methods that are decorated with the [NonAction] attribute will never be executed.   
  
Finally It will call the InvokeAction method to execute the Action.

1. public virtual bool InvokeAction(ControllerContext controllerContext, string actionName)
2. {
3. if (controllerContext == null)
4. {
5. throw new ArgumentNullException("controllerContext");
6. }
7. if (string.IsNullOrEmpty(actionName))
8. {
9. throw new ArgumentException(MvcResources.Common\_NullOrEmpty, "actionName");
10. }
11. ControllerDescriptor controllerDescriptor = this. GetControllerDescriptor(controllerContext);
12. ActionDescriptor actionDescriptor = this.FindAction(controllerContext, controllerDescriptor, actionName);
13. if (actionDescriptor == null)
14. {
15. return false;
16. }
17. FilterInfo filters = this.GetFilters(controllerContext, actionDescriptor);
19. try
20. {
21. AuthorizationContext context = this.InvokeAuthorizationFilters(controllerContext, filters.AuthorizationFilters, actionDescriptor);
22. if (context.Result != null)
23. {
24. this.InvokeActionResult(controllerContext, context.Result);
25. }
26. else
27. {
28. if (controllerContext.Controller.ValidateRequest)
29. {
30. ValidateRequest(controllerContext);
31. }
32. IDictionary<string, object> parameterValues = this.GetParameterValues(controllerContext, actionDescriptor);
33. ActionExecutedContext context2 = this.InvokeActionMethodWithFilters(controllerContext, filters.ActionFilters, actionDescriptor, parameterValues);
34. this.InvokeActionResultWithFilters(controllerContext, filters.ResultFilters, context2.Result);
35. }
36. }
37. catch (ThreadAbortException)
38. {
39. throw;
40. }
41. catch (Exception exception)
42. {
43. ExceptionContext context3 = this.InvokeExceptionFilters(controllerContext, filters.ExceptionFilters, exception);
44. if (!context3.ExceptionHandled)
45. {
46. throw;
47. }
48. this.InvokeActionResult(controllerContext, context3.Result);
49. }
50. return true;
51. }

 The Controller returns an instance of ActionResult. The Controller typically executes one of the helper methods (mostly View() that returns an instance of the ViewResult class, that is derived from the ActionResult class). Here's the list of classes that extend from the ActionResult class. You just need to call a specific Helper method to return the respective ActionResult.

|  |  |  |
| --- | --- | --- |
| **Action Result** | **Helper Method** | **Description** |
| [ViewResult](http://msdn.microsoft.com/en-in/library/system.web.mvc.viewresult(v=vs.100).aspx) | [View](http://msdn.microsoft.com/en-in/library/system.web.mvc.controller.view(v=vs.100).aspx) | Renders a view as a Web page. |
| [PartialViewResult](http://msdn.microsoft.com/en-in/library/system.web.mvc.partialviewresult(v=vs.100).aspx) | [PartialView](http://msdn.microsoft.com/en-in/library/system.web.mvc.controller.partialview(v=vs.100).aspx) | Renders a partial view, that defines a section of a view that can be rendered inside another view. |
| [RedirectResult](http://msdn.microsoft.com/en-in/library/system.web.mvc.redirectresult(v=vs.100).aspx) | [Redirect](http://msdn.microsoft.com/en-in/library/system.web.mvc.controller.redirect(v=vs.100).aspx) | Redirects to another action method by using its URL. |
| [RedirectToRouteResult](http://msdn.microsoft.com/en-in/library/system.web.mvc.redirecttorouteresult(v=vs.100).aspx) | [RedirectToAction RedirectToRoute](http://msdn.microsoft.com/en-in/library/system.web.mvc.controller.redirecttoaction(v=vs.100).aspx) | Redirects to another action method. |
| [ContentResult](http://msdn.microsoft.com/en-in/library/system.web.mvc.contentresult(v=vs.100).aspx) | [Content](http://msdn.microsoft.com/en-in/library/system.web.mvc.controller.content(v=vs.100).aspx) | Returns a user-defined content type. |
| [JsonResult](http://msdn.microsoft.com/en-in/library/system.web.mvc.jsonresult(v=vs.100).aspx) | [Json](http://msdn.microsoft.com/en-in/library/system.web.mvc.controller.json(v=vs.100).aspx) | Returns a serialized JSON object. |
| [JavaScriptResult](http://msdn.microsoft.com/en-in/library/system.web.mvc.javascriptresult(v=vs.100).aspx) | [JavaScript](http://msdn.microsoft.com/en-in/library/system.web.mvc.controller.javascript(v=vs.100).aspx) | Returns a script that can be executed on the client. |
| [FileResult](http://msdn.microsoft.com/en-in/library/system.web.mvc.fileresult(v=vs.100).aspx) | [File](http://msdn.microsoft.com/en-in/library/system.web.mvc.controller.file(v=vs.100).aspx) | Returns binary output to write to the response. |
| [EmptyResult](http://msdn.microsoft.com/en-in/library/system.web.mvc.emptyresult(v=vs.100).aspx) | (None) | Represents a return value that is used if the action method must return a null result (void). |

[Courtesy: [Controllers and Action Methods in ASP.NET MVC Applications](http://msdn.microsoft.com/en-in/library/dd410269(v=vs.100).aspx) ]

1. public abstract class ActionResult
2. {
3. public abstract void ExecuteResult(ControllerContext context);
4. }

ExecuteResult() is implemented differently in various sub-classes of ActionResult. ViewResult is the most commonly used ActionResult. So let's discuss this.  
  
The following happens after the ExecuteResult() method of ViewResult is called.

1. ViewResultBase calls the FindView() method of the ViewResult class.
2. The FindView() method of the ViewResult class returns an instance of the ViewEngineResult class.
3. The Render() method of the ViewEngineResult class is called to Render the view using the ViewEngine.
4. The Render() method internally calls the RenderViewPage() method that sets the master page location and ViewData.
5. The response is rendered on client browser.
   1. public virtual ViewEngineResult FindView( ControllerContext controllerContext, string viewName, string masterName)
   2. {
   3. if (controllerContext == null)
   4. {
   5. throw new ArgumentNullException("ControllerContext");
   6. }
   7. if (string.IsNullOrEmpty(viewName))
   8. {
   9. throw new ArgumentException(MvcResources.Common\_NullOrEmpty, "viewName");
   10. }
   11. Func<IViewEngine, ViewEngineResult> cacheLocator = e => e.FindView(controllerContext,      viewName, masterName, true);
   12. Func<IViewEngine, ViewEngineResult> locator = e => e.FindView(controllerContext, viewName,     masterName, false);
   13. return Find(cacheLocator, locator);
   14. }
   16. public virtual void Render(ViewContext viewContext, TextWriter writer)
   17. {
   18. if (viewContext == null)
   19. {
   20. throw new ArgumentNullException("viewContext");
   21. }
   23. object obj2 = this.BuildManager.CreateInstanceFromVirtualPath(this.Viewpath,typeof(object));
   24. if (obj2 == null)
   25. {
   26. throw new InvalidOperationException(string.Format(CultureInfo.CurrentUICulture,                                                      MvcResources.WebFormViewEngine\_ViewCouldNotBeCreated,
   27. new object[] { this.ViewPath }));
   28. }
   29. ViewPage page = (ViewPage) obj2;
   30. if (page != null)
   31. {
   32. this.RenderViewPage(viewContext, page);
   33. }
   34. else
   35. {
   36. ViewUserControl control = (ViewUserControl) obj2;
   37. if (control == null)
   38. {
   39. throw new InvalidOperationException(string.Format(CultureInfo.CurrentUICulture,                                                              MvcResources.WebFormViewEngine\_WrongViewBase,                                                                           new object[] { this. ViewPath }));
   40. }
   41. this.RenderViewUserControl(viewContext, control);
   42. }
   43. }
   45. private void RenderViewPage(ViewContext context, ViewPage page)
   46. {
   47. if (!string.IsNullOrEmpty(this.MasterPath))
   48. {
   49. page.MasterLocation = this.MasterPath;
   50. }
   51. page.ViewData = context.ViewData;
   52. page.RenderView(context);
   53. }

**NOTE:** The ViewPage class is derived from System.Web.UI.Page class. This is the same base class from which classic ASP.NET pages are derived.  
  
The RenderView() method finally calls the ProcessRequest() method of the Page class that renders the view in the client browser.