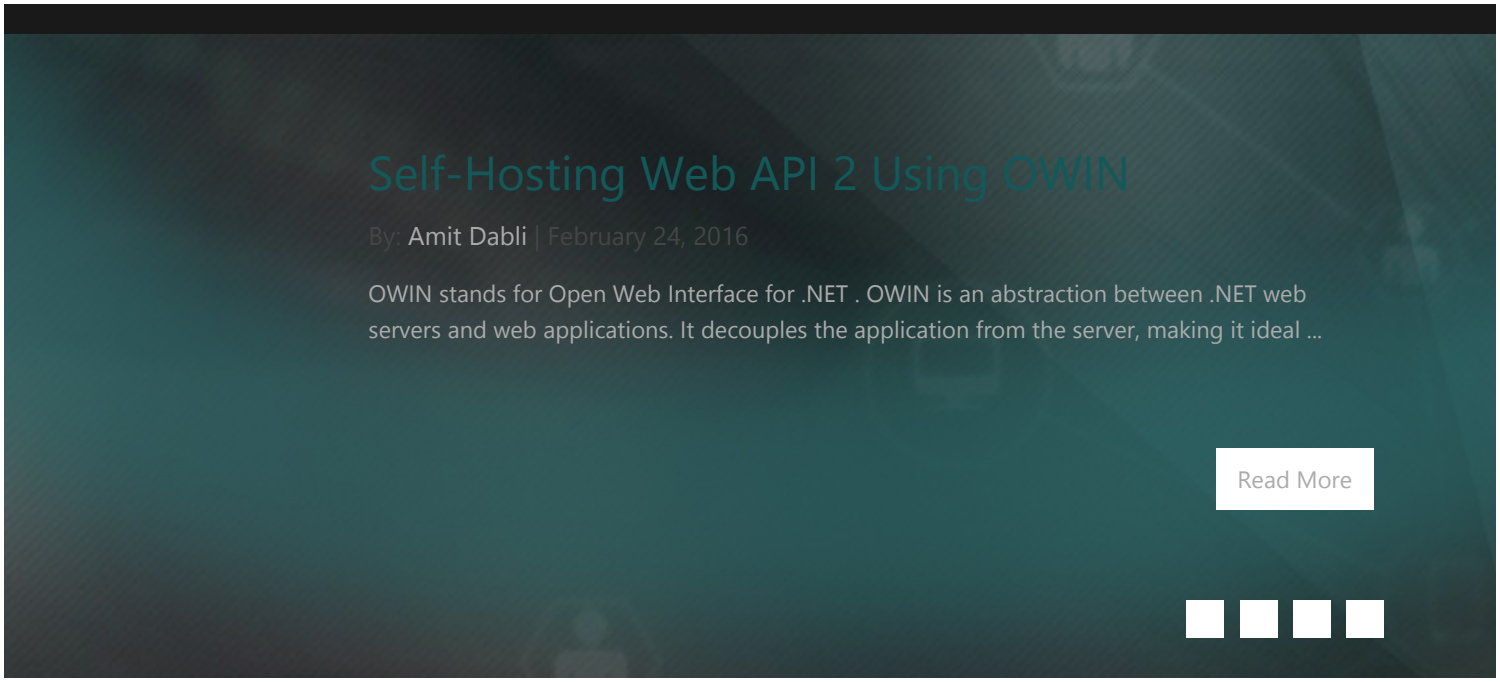
| BLOG



A Detailed Walkthrough of ASP.net MVC Request Life Cycle

By:
Chetan Vihite | June 30, 2015

MVC Request Life Cycle

Life cycle of MVC request is a series of steps involved in processing client request. Regardless of technology and platforms almost all the web frameworks have one or other type of Request life cycle and MVC is no different. Understanding the life cycle of any web framework helps better leverage the features for processing requests.

In this article I am going to explain what exactly happens in ASP.NET MVC request life cycle and what each step in the life cycle does and how we can leverage it further based on our needs. This article specifically targets Page life cycle which is different from Application life cycle. A typical Application life cycle contains Application start and Application End events, however http Life cycle is something which is repeated for every request. Since application events are also part of life cycle, we will see them as we move along.

The MVC Request Life Cycle

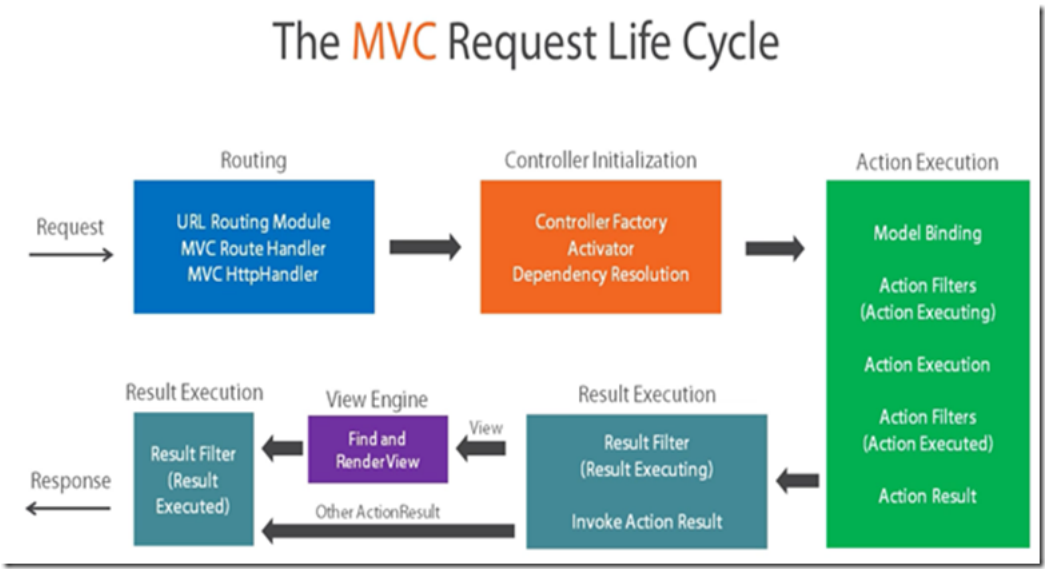
The entry point of MVC Request life cycle is URL Routing module, the incoming request from IIS pipeline is handed over to URL Routing module which analyses the request and looks up Routing table to figure out which controller the incoming request maps to. Routing Table is a static container of routes defined in MVC application with corresponding controller action mapping. If the route is found in the routing table ***MVCRouteHandler*** executes and brings the instance of ***MVCHttpHandler***. Together they act as a gateway into the MVC Framework.

MVC handler begins initializing and executing controller. The MVCHttpHandler also takes of converting route data into concrete controller that is capable of serving the request. MVC handler does all this with the help of MVC Controller factory and activator which are responsible for creating an instance of the controller. This is also the place where the Dependency Injection is performed if the application has been designed to invoke parameterized controller constructor and satisfy its dependencies.

After the controller instance is created the next major step is to find and execute the corresponding action. A component called ***ActionInvoker*** finds and executes the action defined in routing table. Before the action method is called model bindings takes place which maps data from http request to action method parameters. After the model binding, action filters are invoked which includes OnActionExecuting filter. This is followed by action execution and Action Executed filter execution and finally preparing Action Result.

Once the Action method has been finished executing the next step is Result execution. MVC separates the action declaration from Result execution. If the Result from action execution is view, then depending upon configuration, ASPX or Razor view engine will be called to find and render the html view as a response of http request. If the result was not view then it's passed as-is to http response.

Following is the conceptual view of MVC request life cycle.



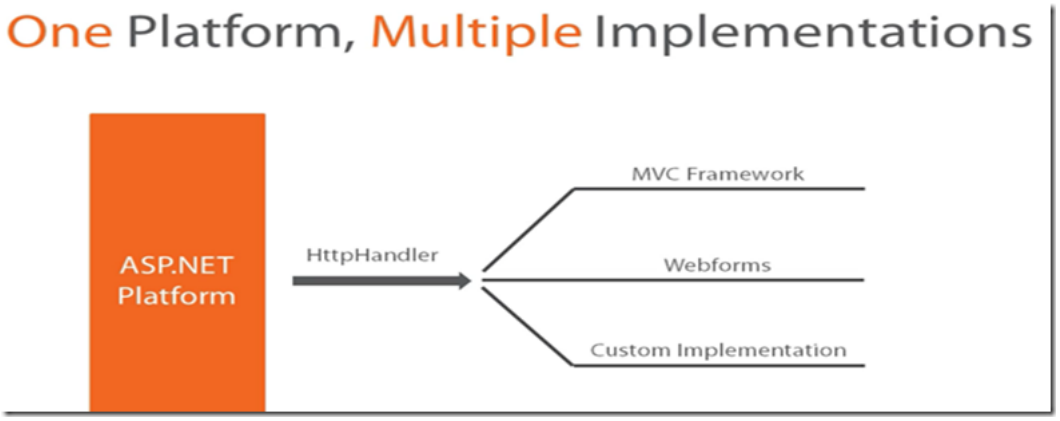
Webforms and MVC

If you have ever developed a web forms application in the past, then it's always a good idea to understand some of the differences between these two.

· Service to a request in Web forms life cycle corresponds to serving a physical file on the disk. However this is not the case in MVC. IN MVC, there is not real concept of serving files from the disk, rather Controller action are executed to render a view to the user.

· Despite the differences ***both*** the web forms and MVC requests implement through HttpHandler. In web forms each page is derived from IHttp-Handler interface and request is served more directly. MVC controllers are also derived from HttpHandlers.

From a higher level, MVC is just another way to manage request from ASP.net platform. One can create their own pipeline and build a framework like MVC or web forms on their own. Following image depicts this idea. also, throughout this article you will recognize the extension points to better leverage the framework.



Application Life Cycle

MVC application life cycle contains two application level events that are associated with start and end events of the application. Application start fires when the application is brought to life by a very first request to the application. Application end event is fired when application has been shut down.

It's important to understand application life cycle events to get a better understanding on how MVC life cycle starts. So far we have seen that URL Routing module is the starting point for MVC application that has a collection of predefined routes to map from. Now, the question here is how does URL routing handler gets this information from? The answer is simple, using Application start event. MVC applications provide these events in Global.asax file which contains all the application level events. All the prestart things are managed in the application start event.

MVC Application_Start event is:

· An event that fires when **first** request is received.

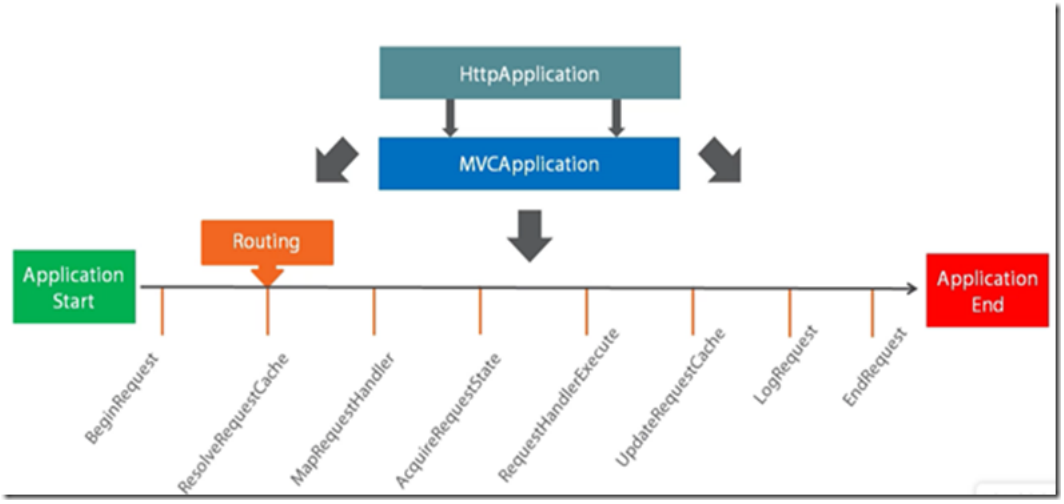
· Can be used to run initial configuration and settings code

· The event takes care of registering all areas of MVC application, installing global filters, adding routes and bundles.

Register routes

Since Application_start event is the first event that gets called when application receives its very first request, all the pre application tasks like routing takes place here.

As you see in the diagram below ResolveRequestCache needs to know the routes to choose from, and this needs to have static route collection already created.



PreApplicationStart:

PreApplicationStart is another option at the assembly level to register something before the application starts. It could be used to run some initial configuration code or register modules or any other code that needs to be executed before the application starts.

HttpHandlers

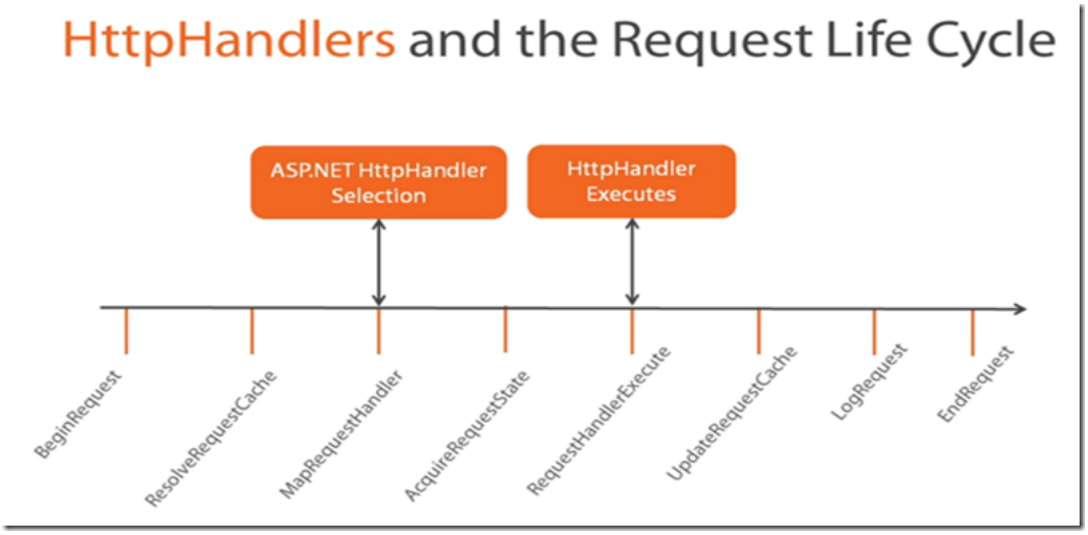
HttpHandlers are classes that implement IHttpHandler and generate a response to HttpRequest. There could be httpHandler re-assignment in a life cycle of a request but only one http handler executes and provides response.

There are two sets of events in the MVC request life cycle that concerns HttpHandlers, [MapRequestHandler and PostMapRequestHandler] and [RequestHandlerExecute and PostRequestHandlerExecute].

MapRequestHandler and PostMapRequestHandler are the events in the life cycle which determines the httpHandler responsible for executing the request. Only the selection happens during this time.

RequestHandlerExecute and PostRequestHandlerExecute are the life cycle events that actually executes the htttp handler determined in the earlier phases of request life cycle.

Refer to following diagram below (note post events have been omitted in the diagram).



Creating an HttpHandler:
Create a class that implements IHttpHandler interface

Register the HttpHandler through code or web.config

IHttpHandler exposes two members:

· IsReusable

· ProcessRequest()

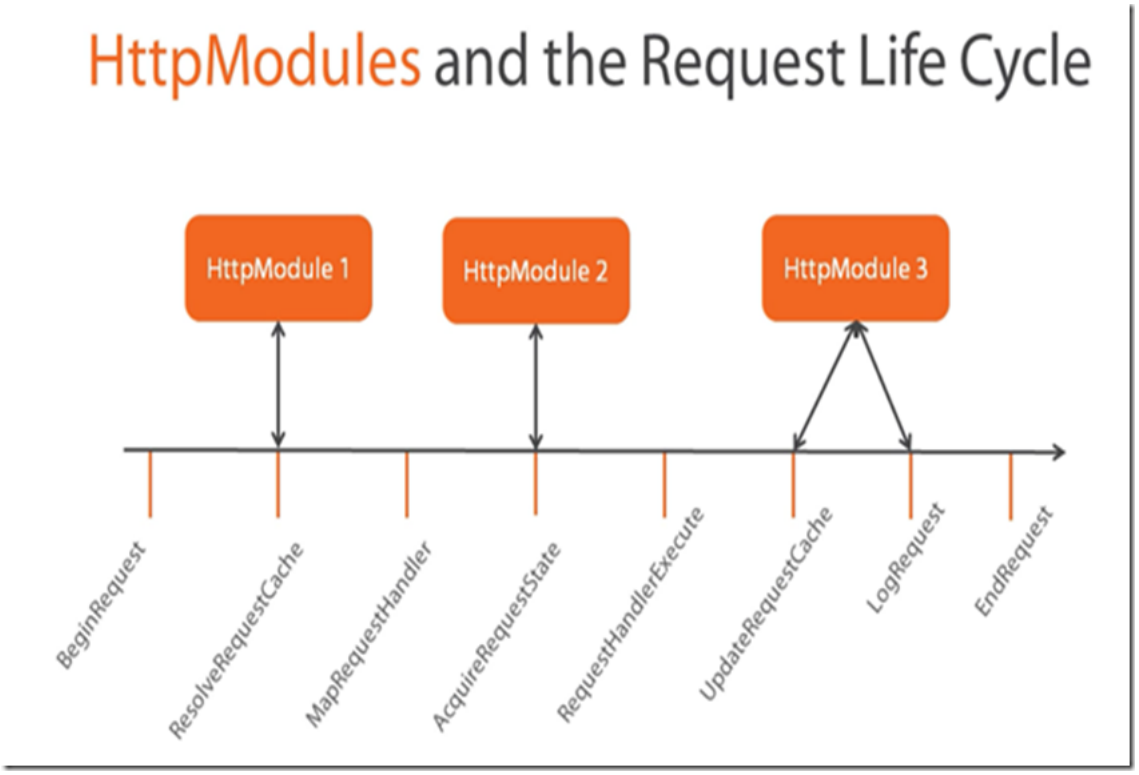
SampleHandler.cs

```
public class SampleHttpHandler:  IHttpHandler
    {
        public bool IsReusable { get { return false; } }

        public void ProcessRequest(HttpContext context)
        {
            context.Response.Write("<b>this is the response from HttpHandler</b>");
        }
    }
```

HttpModules

HttpModules are classes that implement IHttpModule interface and are designed to respond to Life cycle events. In a given Http life cycle, multiple http modules can respond to one single request and can also hook into multiple life cycle events. So they are not tied to any specific event, rather they can act at several places during the life cycle and expose multiple development possibilities.

One of the advantage HttpModules bring is the reusability, modules written once can be reused any several application across frameworks. Features such as logging and authentication are best examples of wrapping things up in a HttpModule. One can also do the all these things possible in Http-Module in a Global.asax file, but that won't achieve reusability and abstraction.



Creating a HttpModule

· Implement the IHttpModule interface

· Register the HttpModule through code or config

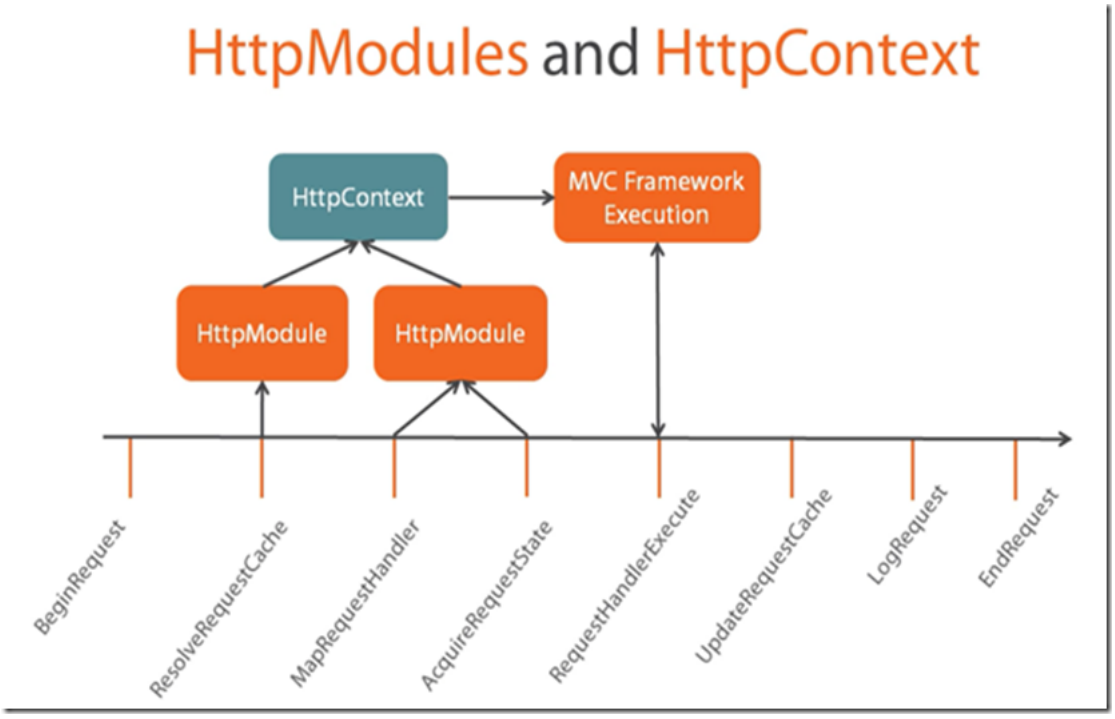
· IHttpModule exposes two members:

1. Init()

2. Dispose()

HttpModule and HttpContext

HttpModule and HttpContext work with each other in the life cycle of a request to serve and achieve the goals of a module. For instance, HttpModule can populate the user information and populate UserInfo object inside HttpContext for applications read its value and proceed accordingly. HttpModules takes advantage of hooking within the lifecycle events, even before the ASP.net MVC framework starts taking control of request. In fact, HttpModules can act early on to request in the very first event itself, "Begin Request", much before the MVC life cycle takes over. This gives HttpModules an edge over MVC requests because you could respond to a request early on and take appropriate actions well before. This can have some serious implications, there are certain things which HttpModules can do really better, while avoiding the MVC framework completely.



Comparing HttpHandlers and HttpModules

Many HttpModules can service one request, however only one HttpHandler can service a request.

HpptModules are primarily used to modify and support requests through services, however HttpHandlers are used to generate a response that is sent back to browser using an HttpHandler.

HpptModules are implemented through IHttpModule interface, and HttpHandlers are implemented through IHttpHandler interface

Both HttpModule and HttpHandler can be registered through code or config file.

HttpModules and deigned to integrate with any of the life cycle events, however HttpHandlers are generally concerned with events related to mapping and execution.

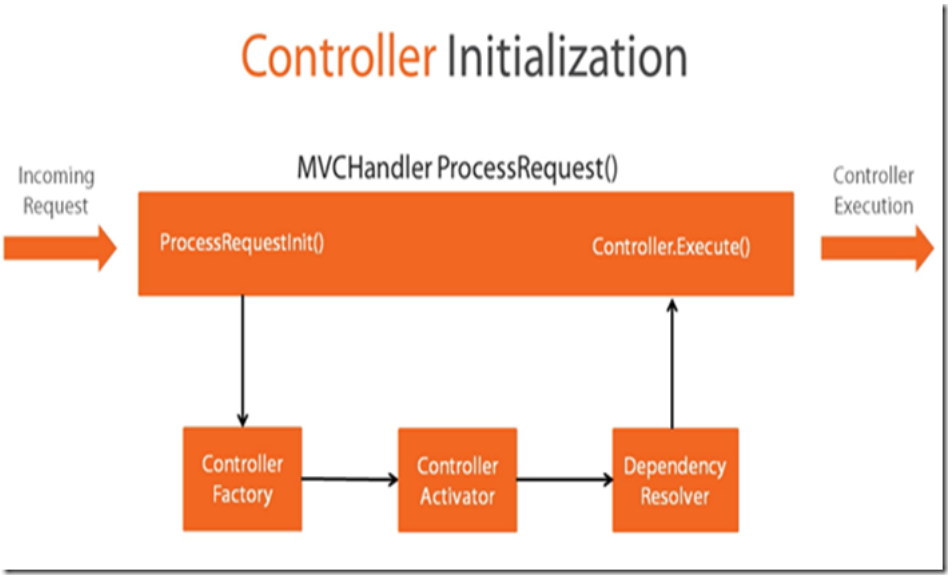
MVCRouteHandler and MVCHandler

UrlRouting module is a special module that contains PostResolveRequestCache event that actually matches the incoming request to routing table and executes routing handler for it. Routing handler is simply a class that implements IRouteHandler interface and exposes one method "**GetHttpHandler**" and return MvcHandler to execute further. MvcHandler is a standard HttpHandler which exposes two methods, IsReusable and ProcessRequest. The ProcessRequest inside the MvcHandler is the entrance into the MvcFramework.

Controllers

Controllers and actions are the two very important components in MVC that every developer work with. Controllers takes care of orchestrating relationships between views and models. Controllers are nothing complex but a class that implements IController interface and exposes one method "**Execute** ()". This execute method actually kicks the execution of controllers in the request life cycle. At a high level, the job of a MvcHandler is to generate a response by executing a controller. But a lot goes under the curtain so let's try to understand how the controller is selected, instantiated and executed to serve the request.

MvcHandler doesn't create an instance of controller right away, rather takes help from several other important classes to perform this. MvcHandler first calls a method ProcessRequestInit() which calls a controller factory to select a controller using the route data provided. Once the selection is performed by controller factory, Controller Activator creates an instance of requested controller using Dependency Resolver. If no dependency resolved was found then Controller Activator manually creates an instance of controller and returns it for execution. Once the controller has been initialized MvcHandler calls controller.Execute() method to begin processing execution. Following diagram depicts the control flow.



Controller Factory

Controller factory is used directly by MvcHandler to get the Controller to execute. Its primary responsibility to find the appropriate type of controller to serve the request. At the heart of ControllerFactory is the interface IControllerFactory that a default controller factory implements.

You could add your own custom factory by implementing IControllerFactory interface. The interface exposes three main methods:

· CreateController for creating controller, it should return a type that implements IController.

· GetControllerSessionStateBehavior determines how session is handled for a given controller.

· ReleaseController for releasing any resources factory is holding onto.

DefaultControllerFactory

MVC framework provides a robust controller factory out-of-the box, called as "**DefaultControllerFactory**". This default factory handles a lot of low level work for getting and creating an instance of desired controller. For instance, if the incoming request is myAPP/Order/22, then controller factory would create an instance of OrderController or whatever is being specified in routing data.

The default controller also has access to "Dependency Resolver", which helps in resolving dependencies of controller. This is the place where custom dependency resolvers fits in to give way for IoC container to come in and resolve dependencies like parameterized contractors etc.

Creating a Custom controller factory

In most of cases you would attach a controller factory component into your application and let the external components handle creating objects for you. Of course, there are some solid reasons for employing an external component to do this, since controller creation and dependency resolution could be some complex. There are several IoC containers available like CastleWindsor, Unity, Ninject etc. that works out of the box and are available on nuget for download. However I still want to pursue creating a custom controller factory for demonstration purposes anyway.

To create a custom controller factory, first create take a controller, that implements IController as below.

```
namespace MVCPageLifeCycle.Controllers
{
    public class ContactController : IController
    {
        public ContactController(ILoggingService logging)
        {

        }

        public void Execute(System.Web.Routing.RequestContext requestContext)
        {
            HttpContext.Current.Response.Write("This was genrated by custom controller");
        }
    }
}
```

Next, create a class which implements IControllerFactory. IControllerFactory exposes three methods, CreateController, GetSessionStateBehavior and ReleaseController.

```
namespace MVCPageLifeCycle.Controllers
{
    public class Logger : ILoggingService
    {
    }
    public class CustomControllerFactory : IControllerFactory
    {
        public IController CreateController(System.Web.Routing.RequestContext requestContext, string controllerName)
        {
            var logger = new Logger();

            if (controllerName == "contact")
            {
                return new ContactController(logger);
            }
            // returning HomeController for demo purposes, in other cases you would
            //have to add logic to instantiate other controllers depending upon context and contollerName
            return new HomeController();
        }
        public System.Web.SessionState.SessionStateBehavior GetControllerSessionBehavior(System.Web.Routing.RequestContext requestContext,
            string controllerName)
        {
            return SessionStateBehavior.Default; // use the default session state for demo purpose
        }
        public void ReleaseController(IController controller)
        {
            // do nothing, as we dont have any resources to free up
        }
    }
}
```

Finally, just set the custom controller factory in Global.asax file. And that's it.

Now when you browse your /contact page, you should see the response that you set in the controller.

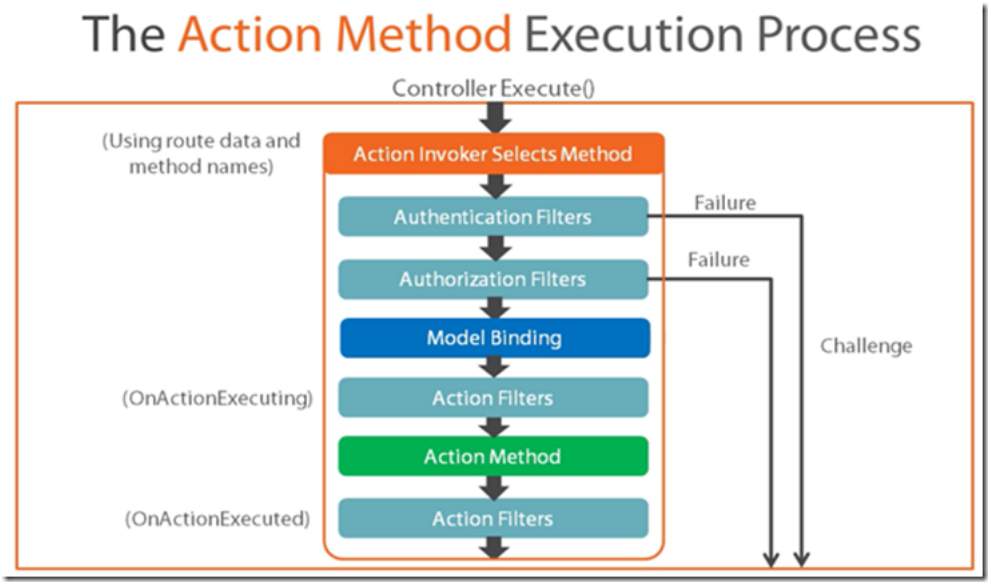
```
protected void Application_Start()
{
    ControllerBuilder.Current.SetControllerFactory(new CustomControllerFactory());

    AreaRegistration.RegisterAllAreas();
    FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
    RouteConfig.RegisterRoutes(RouteTable.Routes);
    BundleConfig.RegisterBundles(BundleTable.Bundles);
}
```

Action Method Execution

Once the Controller has been chosen and initialized execution start, this includes choosing an Action using ActionInvoker's Select method. Action invoker selects appropriate method using route data and method names to choose best fit method. Once the selection is done, Authentication filters fires to ensure that current user is authenticated. If the authentication fails then a challenge is sent back to the use, which could involve redirect user to a login page. If the authentication passes, request moves ahead with authorization. Once again, the same process happens, if the user is not authorized, then a challenge is sent back to browser, otherwise request pipeline moves. Once the authentication and authorization are passed, request moves forward with identifying parameters required to pass in to selected action. This process is called as Model Binding. Model binding collects data from query string, route data and request to generate objects required by Action method. The process correctly maps the parameters to objects and also takes care of converting incoming types to correct types required by Action method.

Once the model binding has been done, Action filters kick in. before an action method is being called, OnActionExecuting filter is executed and any code placed inside this filter gets executed. Once the OnActionExecuting has been executed, Action method execution starts. Once the Action method has been executed and a response has been generated, another filter OnActionExecuted gets executed.

All the steps involved in Action execution are very much extensible and that makes MVC a great framework to work with. Following figure depicts the Action method execution flow.



Action Invoker

Controller's execution is really empowered by ActionInvoker's select method in the sense that it can select appropriate action in variety of ways. Like many other MVC framework components Action Invoker also implements a simple interface "IActionInvoker" that exposes one method InvokeAction.

```
public interface IActionInvoker()
{
        bool InvokeAction(ControllerContext controllerContext, string actionName);
}
```

The method takes two arguments, controllerContext and actionName that are used to select and invoke appropriate action. In almost all the cases you don't need to implement this interface because MVC by default provides a powerful implementation, ControllerActionInvoker. Further, default behaviour of controller action invoker can be customized using extension points.

Action Selectors

So how does Action Selector work? In a very straight forward way, Action Selectors select an action based on Route Data and matches the method name. Sounds simple, but there is more to it. If you have worked with Http methods before, you might recognize that MVC provides Action methods with http attributes. One can decorate action methods with these http attributes to specify how this method should be invoked. For example, it ensure that a http GET request can't invoke http POST method, just because the name matches.

Some common Action selectors that MVC provides are HttpGet, HttpPost, AcceptVerbs, ActionName, NonAction and Custom.

Selecting an action is a multi-step process. First off, MVC determines which methods are eligible, only public non-static, non-special methods (e.g.

constructors, ToString() are excluded) are considered in the Action method selection. Then, the framework matches signature of the methods with incoming request.

· If any valid method, with signatures matches to only one method then that method is returned otherwise No Match error is thrown.

· If there are multiple methods found with same signature then only one where action selectors matches is returned otherwise No Match error is thrown.

· If there are multiple methods with same action selectors then Ambiguous Match error is thrown.

Refer to code fragment below, which depicts a common scenario where we have one method to show form to register and another one to handle posting of same form. In such cases, it's always best to have HttpGet or HttpPost selectors specified so that appropriate method is called to serve the request.

```
[HttpGet]
public ActionResult Register()
{
        return View();
}
[HttpPost]
public ActionResult Register(RegistrationInfo personalInfo)
{
        return View(personalInfo);
}
```

Just like other components of MVC, Action Selectors are also extensible. You could create your own custom action selector by deriving ActionMethodSelectorAttribute class. This class exposes one method to determine if the current method is valid for a request or not using "IsValidForRequest" method.

```
public class CustomSelector : ActionMethodSelectorAttribute
{
       public override bool IsValidForRequest(ControllerContext controllerContext,
       MethodInfo methodInfo)
}
```

Model Binding

Model binding is the process that maps data from request and supply to Action methods parameters. Model binding takes places after the authorization filters have executed and model is ready to bind. Model binder works with Value provides to get and map data. By default there are four value provides that MVC framework uses, Form Data, Route Data, Query string and posted Files. You can create your own custom value provider to choose data from, such as Cookie Value provider as custom value provider.

Model Binder is implemented from IModelBinder interface. By default, MVC provides a very powerful model binder but you can create your own custom model binder for your specific project needs.

```
public interface IModelBinder
{
        public object BindModel(ControllerContext controllerContext,
        ModelBindingContext bindingContext)}
}
```

Filters

Filters are interesting and somewhat unique part of MVC. Filters are designed to inject logic in between MVC request life cycle. Filters provide users with powerful ways to inspect, analyze, capture and instruments several things going around within MVC projects. A very interesting thing that how filters run at multiple points in the request life cycle compared to other components. As of MVC5, there are 5 types of filters.

Filter type	Implemented by
Authentication Filters	IAuthenticationFilter
Authorization Filters	IAuthorizationFilter
Action Filters	IActionFilter
Result Filters	IResultFilter
Exception Filters	IExceptionFilter

Authentication filters are new addition from MVC 5. These filters kick in first in the request life cycle and perform the authentication logic.

Authorization filters are executed after the Authentication filters successfully executed and authorizes users roles to ensure current user has access to request resource.

Action filters are executed next in the request life cycle and execute any custom logic you may want to perform before and after action execution.

Result filters are executed before and after result execution.

Finally, the exception filters are executed when there is an exception during processing of request.

Scope of filters

Filters are designed keeping scope in mind and nature and need of your projects. Filters can be applied to specific actions and it will run for only those specified actions. Next, filters could also be decorated at the controller level, so it run for all the action of that controller. Finally, we could also specify filters to run at a global level for all the controllers and actions.

Action Result Execution

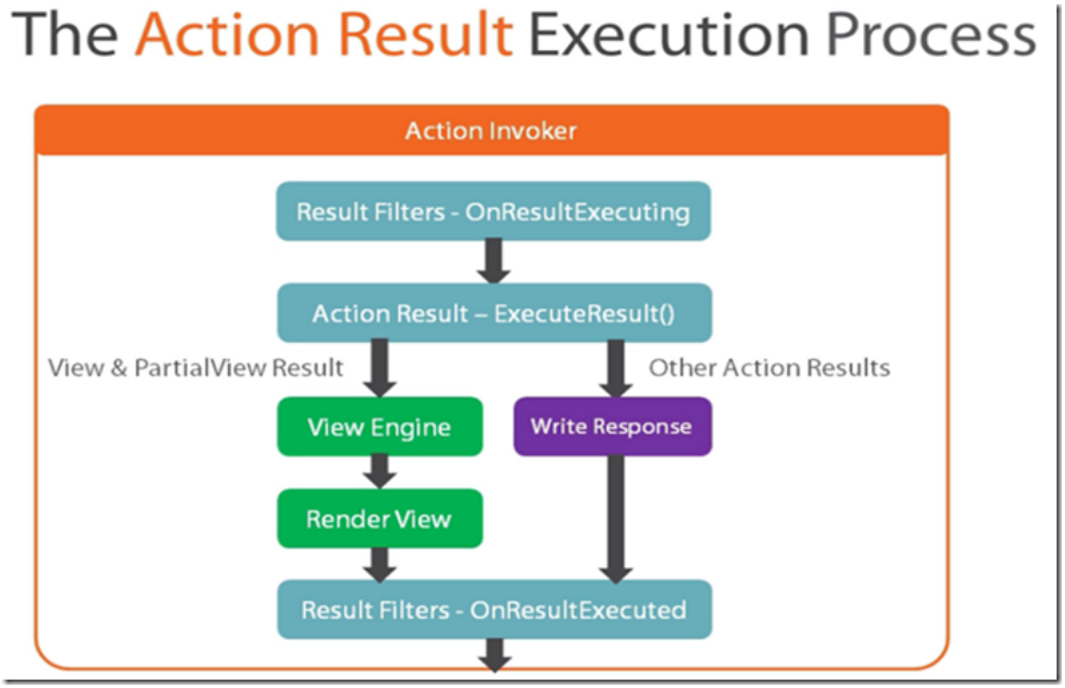
The journey of MVC life cycle ends with Action result execution. It's the last and final step in MVC request life cycle. It's also the most important one, because this is where result is sent back to browser.

Action result execution starts after action invoker are identified an action to execute. At this point, first the Result Filters are executed before and after actual result execution. OnResultExecuting filter executes first in the sequence, followed by Action Result – ExecuteResult(), which takes care of executing result.

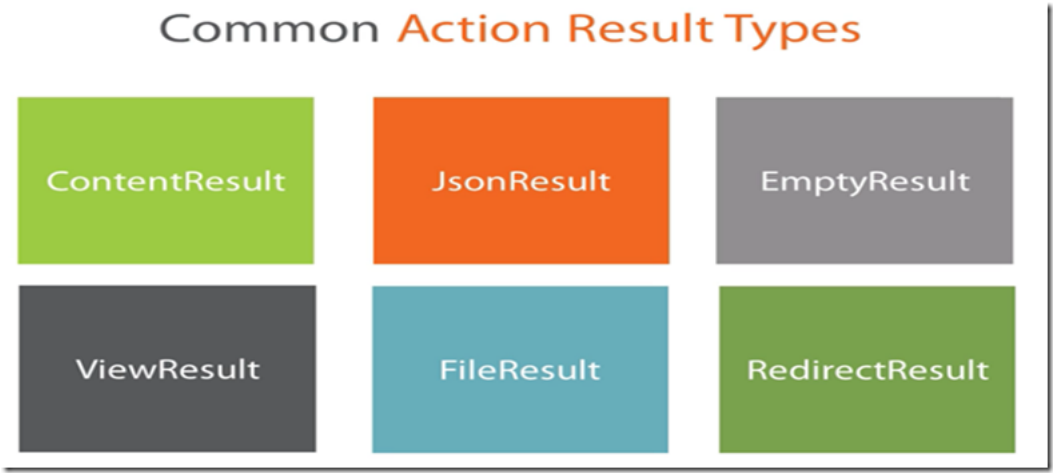
At this point the further execution could branches in to two separate routes. If the result type is View Engine then appropriate view engine Razor or ASPX is brought in to render partial view or view. At the same time, if the Response is any other type, then response if directly written without bringing view engine altogether.

Finally, ResultFilter "OnResultExecuted" is executed which could perform any custom logic written in this filter. Generally speaking, this is the last point in the MVC request life cycle to inject your custom code into the life cycle.

Following the diagram that depicts the Action Result execution flow.



Some common Action Result Types



Custom Action Result Types

Adding a new custom Action result type is very and requires very little effort to plug n new action result types.

First off, create a new class derived from ActionResult class, and then override ExecuteResult method to provide your own implementation.

Lets say, we want to change the default json result type to something different, like use Json.net to covert to json instead of default json conversion.

We will have to start with a class that is derived from ActionResult class and overrideExecuteResult method to use our own custom logic.

See the class below,

```
public class JsonNETResult: ActionResult
{
    public object Data { get; set; }

    public override void ExecuteResult(ControllerContext context)
    {
        var response = context.HttpContext.Response;

        response.ContentType = "application/json";

        response.Write(JsonConvert.SerializeObject(Data));
    }
}
```

Once this is in place, on any action method, we can use this as return type, see the code fragment below –

```
[HttpGet]
public JsonNETResult Json()
{
    var person = new Person { Name = "Chetan", Email = "cvihite@gmail.com"};

    return new JsonNETResult() { Data = person };
}
```

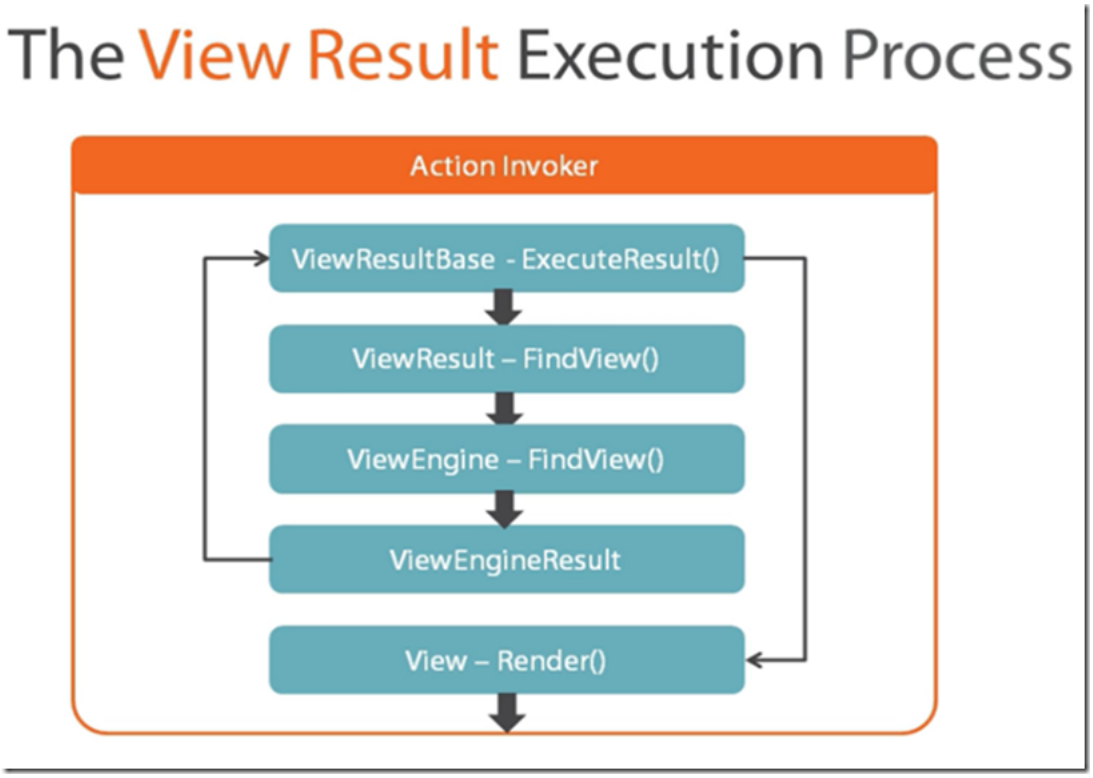
Yes, it is that simple to create a custom Action result type.

View Results and View Engine

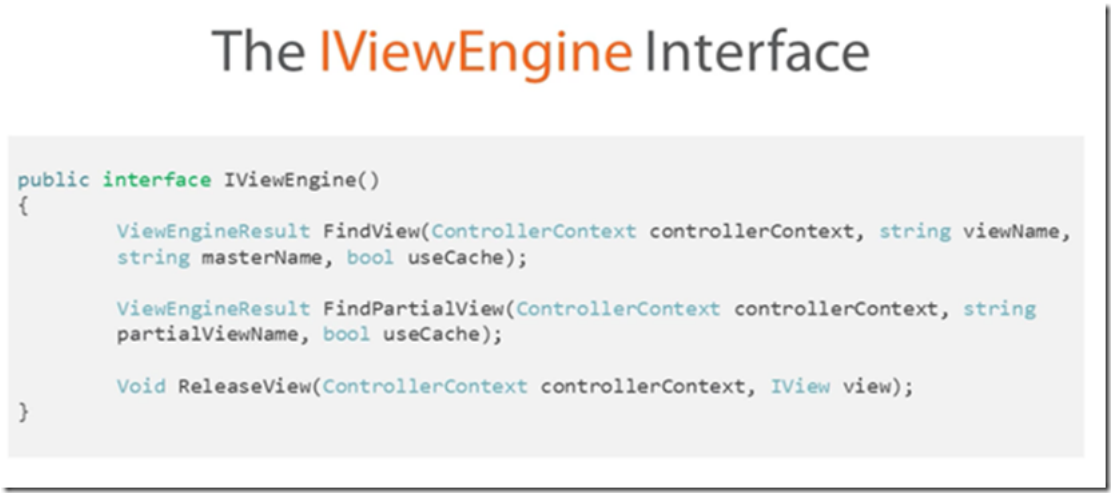
View results are the most common results type that as a MVC developer you might be using. On a high level, View result is responsible for returning HTML back to your browser. View Result employs View Engine to do this task and ensures the correct views are being rendered.

There are two view engines that MVC provides by default, Razor View Engine and Legacy View Engine. Razor view engine is more sophisticated and fast compared to Legacy engine. Legacy view engine understands ASPX format code and emits raw HTML.

View rendering process starts when the View Result calls the Execute Result method from its base class ViewResultBase. The ExecuteResult from viewResultBase calls the abstract method FindView. The child view result overrides this method and calls FindView from view engine, which either returns a view or list of locations searched to locate a view. Finally ViewEngineResult returns a view which has single method “Render” which actually writes response to the request. Following is the diagram that depicts this flow.

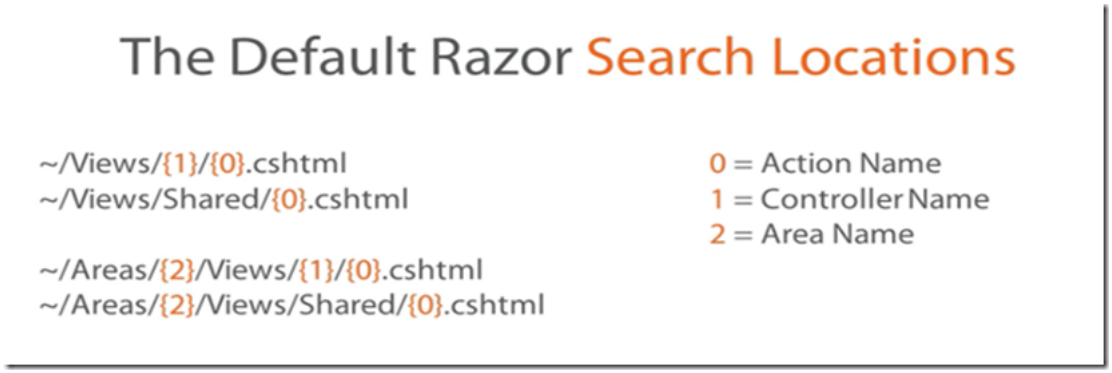


If we look at ViewEngine, which actually implements IViewEngine interface and exposes three methods, FindView, FIndPartialView and ReleaseView. First two methods which deals with finding view are important to search and locate view and return it for processing. ReleaseView method simply releases any resources holding by viewengine like FileStream



Default Razor Search Locations

By default view engine looks up at location, shown in the diagram below. If for some reason it can't find a view then it throws an exception. Once the view has been found viewengine calls its render method to emit html code for browser to display on the screen.



This concludes the MVC request life cycle.

I hope I have touched much more points than a regular article and presented a lot many ways to leverage MVC. Feel free to reach me on chetan.vihite@gmail.com should you have any questions.

This post has been viewed 15,667 times

3 thoughts on "A Detailed Walkthrough of ASP.net MVC Request Life Cycle"

Pingback: Kool Open Source Web 2.0 Apps in .NET | Insight's Delight

Navaseelan
August 28, 2015 at 3:11 am

Very nice...keep up the work !

Ganesh
October 6, 2015 at 12:47 pm

Good explanation, First time i got complete life cycle of MVC in one page