12,210,083 members (46,072 online) Sign in

MVC Application Lifecycle

ashish__shukla, 7 Apr 2014 CPOL Rate this:

★★★★★ 4.81 (81 votes)

A look into the different components involved in an mvc application request processing.

Contents

- Introduction
- Background
- UrlRoutingModule
- RouteHandler
- MvcHandler
- ControllerFactory
- Controller
- ActionInvoker
- ActionResult
- ViewEngine

Introduction

In this article we will discuss about the MVC application lifecycle and how the request is processed as it passes from one component to the other. We will talk about the components in the order they occur in the application lifecycle .We will also look into the role of each of component and how they are related to the other component's in the pipeline.

Background

As developers we are aware about some of the components used by the MVC framework to process the request. Mostly we work with the controllers and the action methods.

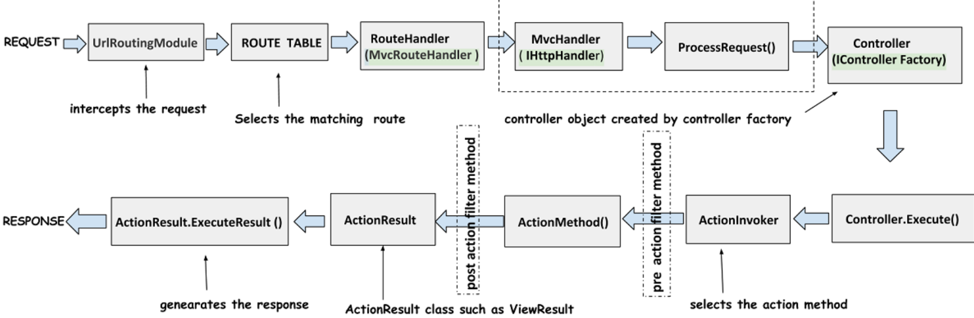
Also we work with the different ActionResult's and the Views . *But are we aware about the other important components involved in the request processing*?. *And how the request flows in the request pipeline*?

When I started learning MVC one of the things I could not understand was how the request flows from one component to the other.Also I was not clear about the role of HTTP module and the HTTP handler in the request processing.After all MVC is a web development framework so there has to be HTTP module and the HTTP handler involved somewhere in the pipeline.

We have more components involved in this request processing pipeline then we know of , the controller and the action methods that we work with, that have an equally important role in the request processing.

Though most of the time we can use the default functionality provided by the framework but if we understand what is the role of each of the component we can easily swap the components or provide our own custom implementations.

The main components and their role in the Request pipeline.

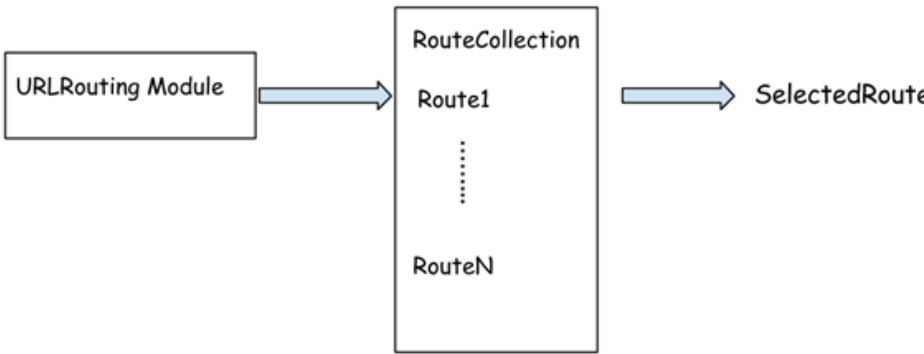


Let us see what happen's when the request if first made for a resource in an MVC application

UrlRoutingModule

The entry to the Mvc application

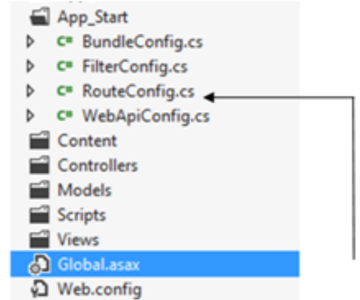
The request is first intercepted by the UrlRoutingModule which is a HTTP Module. *It is this module that decides whether the request would be handled by our MVC application*. UrlRouting Module selects the first matching route.



How does the UrlRoutingModule matches the request with the routes in the application?.

If you look into the RegisterRoutes method called from the global.asax you will notice that we add routes to the routes RouteCollection. This method is called from the application_start event handler of the global.asax

It is this RegisterRoutes method which registers all the routes in the application



Hide Copy Code

```
RouteConfig.RegisterRoutes(RouteTable.Routes);
 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 }
            );
        }
```

Now you may ask how the UrlRouting Module knows about these routes and how the routes are associated with the RouteHandler? UrlRouting Module knows about these routes using the *maproute method.*

I*f you look at the the maproute method you will notice that it is defined as an extension method.*

```
routes.MapRoute(
    nam
    url
    def
);
```

(extension) Route RouteCollection.MapRoute(string name, string url, object defaults) (+ 5 overload(s))
Maps the specified URL route and sets default route values.

Exceptions:
System.ArgumentNullException

Behind the scenes it associates the routeHandler with the routes.Internally MapRoute method is implemented as:

```
var Default = new Route(url, defaults , routeHandler);
```

So basically what this method does is attach a routeHandler with the route.

UrlRoutingModule is defined as

Hide Copy Code

```
public class UrlRoutingModule : IHttpModule
    {
public UrlRoutingModule();
public RouteCollection RouteCollection { get; set; }  //omitting the other details
```

So now we know that the UrlRoutingModule is aware of all the routes in the application and hence it can match the correct route for the request.*The main thing to note here is that the UrlRoutingModule selects the first matching route. As soon as a match is found in the routing table, scanning process stops.*

So let's say we have 10 routes in our application and the more specific route is defined after the more general route so in this case *the specific route that is added later will never be matched since the more general route will always be matched* .So we need to take care of this when adding routes to the route collection.

Here if the request is matched by any of the routes in route collection then the other routes added later in the collection will not be able to handle request.Please note that If the request is not matched by any of the routes in the UrlRoutingModule then it is not handled by the MvcApplication.

So following happens at this stage.

- *The URLRoutingModule attaches the route handler to the routes.*

RouteHandler

The generator of MvcHandler

As we have already seen that MvcRouteHandler instance gets attached with the route's using the MapRoute method.MvcRouteHandler implements the IRouteHandler interface.

This MvcRouteHandler object is used to obtain a reference to the MvcHandler object which is the HTTPHandler for our application.

When the MvcRouteHandler is created one of the things it do is to call the PostResolveRequestCache() method. PostResolveRequestCache() method is defined as

Hide Copy Code

```
public virtual void PostResolveRequestCache(HttpContextBase context) {
 RouteData routeData = this.RouteCollection.GetRouteData(context);
```

```
  if (routeData != null)      {
 IRouteHandler routeHandler = routeData.RouteHandler;

IHttpHandler httpHandler = routeHandler.GetHttpHandler(requestContext);
```

Following happens in the PostResolveRequestCache() method.

- *RouteCollection porperty has a GetRouteData() method.This GetRouteData() method is called and passed the HttpContext.*
- *GetRouteData() method returns the RouteData object*
- *routeData has a RouteHandler property which returns the IRouteHandler for the current request which is the MvcRouteHandler.*
- *This MvcRouteHandler has the GetHttpHandler() method which returns the reference to the MvcHandler*
- *It then delegates control to the new MvcHandler instance.*

MvcHandler

The Request Processor

MvcHandler is defined as

```
namespace System.Web.Mvc
{
    [...]public class MvcHandler : IHttpAsyncHandler, IHttpHandler, IRequiresSessionState
    {
        [...]public static readonly string MvcVersionHeaderName;

        [...]public MvcHandler(RequestContext requestContext);
```

As you can see that its a normal Http handler.Being an Http handler *it implements the ProcessRequest() method.* The ProcessRequest() method is defined as:

Hide Copy Code

```
 // Copyright (c) Microsoft Open Technologies, Inc.<pre>// All rights reserved. See License.txt
in the project root for license information.
void IHttpHandler.ProcessRequest(HttpContext httpContext)
{
    ProcessRequest(httpContext);
}
protected virtual void ProcessRequest(HttpContext httpContext)
{
    HttpContextBase iHttpContext = new HttpContextWrapper(httpContext);
                ProcessRequest(iHttpContext);
}
protected internal virtual void ProcessRequest(HttpContextBase httpContext) {
    SecurityUtil.ProcessInApplicationTrust(() => {
        IController controller;
        IControllerFactory factory;
        ProcessRequestInit(httpContext, out controller, out factory);
    try
    {
        controller.Execute(RequestContext);
    }
    finally
    {
        factory.ReleaseController(controller);
    }
    });
}
```

As you can see above the ProcessRequest() method calls the ProcessRequestInit() method which is defined as:

Hide Copy Code

```
 private void ProcessRequestInit(HttpContextBase httpContext,
             out IController controller, out IControllerFactory factory) {
    // If request validation has already been enabled, make it lazy.
    // This allows attributes like [HttpPost] (which looks
    // at Request.Form) to work correctly without triggering full validation.
    bool? isRequestValidationEnabled =
       ValidationUtility.IsValidationEnabled(HttpContext.Current);
    if (isRequestValidationEnabled == true) {
        ValidationUtility.EnableDynamicValidation(HttpContext.Current);
    }
```

```
    AddVersionHeader(httpContext);
    RemoveOptionalRoutingParameters();
    // Get the controller type
    string controllerName = RequestContext.RouteData.GetRequiredString("controller");
    // Instantiate the controller and call Execute
    factory = ControllerBuilder.GetControllerFactory();
    controller = factory.CreateController(RequestContext, controllerName);
    if (controller == null) {
        throw new InvalidOperationException(
        String.Format(
            CultureInfo.CurrentCulture,
            MvcResources.ControllerBuilder_FactoryReturnedNull,
            factory.GetType(),
            controllerName));
    }
}
```

In the ProcessRequest() method following happens:

- *ProcessRequestInit() method is called which creates the ControllerFactory.*
- *This ControllerFactory creates the Controller.*
- *Controller's Execute() method is called*

ControllerFactory

The generator of Controller

As you can see above one of the things that happens inside the ProcessRequest() method is that ControllerFactory is obtained that is used to create the Controller object. Controller factory implements the interface IControllerFactory.

By default the framework creates the DefaultControllerFactory type when the ControllerFactory is created using the ControllerBuilder.

*The ControllerBuilder is a singleton class and is used for creating the ControllerFactory.*Following line in the ProcessRequestInit() method creates the ControllerFactory.

Hide Copy Code

```
factory = ControllerBuilder.GetControllerFactory();
```

So the GetControllerFactory() method returns the ControllerFactory object. So now we have the ControllerFactory object.

ControllerFactory uses the CreateController method to create the controller. CreateController is defined as:

Hide Copy Code

```
IController CreateController(
                    RequestContext requestContext,
                    string controllerName )
```

The ControllerBase object is created using the default ControllerFactory implementation .

If required we can extend this factory by implementing the IControllerFactory interface and then declaring the following in the Application_Start event in the gloabal.asax.

Hide Copy Code

```
ControllerBuilder.Current.SetDefaultControllerFactory(typeof(NewFactory))
```

The SetControllerFactory() method is used to set the custom controller factory instead of the Default Controller Factory that is used by the framework .

Controller

The container for the user defined logic

So we have seen that the ControllerFactory creates the Controller object in the `ProcessRequest()` *method of the MvcHandler.*

As we know a controller contains the action methods.An action method gets called when we request a URL in the browser.Instead of explicitly implementing the IController interface we create our controllers using the *Controller* class which provides many features for us.

Now this Controller class inherits from another Controller class called the "ControllerBase" which is defined as:

Hide Copy Code

```
public abstract class ControllerBase : IController
{
    protected virtual void Execute(RequestContext requestContext)
    {
        if (requestContext == null)
        {
            throw new ArgumentNullException("requestContext");
        }
        if (requestContext.HttpContext == null)
        {
            throw new ArgumentException(
              MvcResources.ControllerBase_CannotExecuteWithNullHttpContext,
              "requestContext");
        }
        VerifyExecuteCalledOnce();
        Initialize(requestContext);
        using (ScopeStorage.CreateTransientScope())
        {
            ExecuteCore();
        }
    }
    protected abstract void ExecuteCore();
 // .......
```

The controller object uses the ActionInvoker to call the action methods in the controller which we will look into later.

After the controller object is created using the controller factory following happens :

- *The Execute() method of the controllerbase is called*
- *This Execute() method calls the ExecuteCore() method which is declared abstract and is defined by Controller class.*
- *The Controller class's implementation of the ExecuteCore() method retrieves the action name from the RouteData*
- *ExecuteCore() method calls ActionInvoker's InvokeAction() method.*

ActionInvoker

The Action Selector

The ActionInvoker class has some of the most important responsibilities of finding the action method in the controller and then invoking the action method.

The ActionInvoker is an object of a type that implements the IActionInvoker interface. The IActionInvoker interface has a single method defined as:

Hide Copy Code

```
  bool InvokeAction(
              ControllerContext controllerContext,
              string actionName
)
```

The controller class provides the default implementation of IActionInvoker , which is ControllerActionInvoker

The controller class exposes a property named *ActionInvoker* which returns the ControllerActionInvoker .It uses the CreateActionInvoker() method to create the ControllerActionInvoker .If you see the method it is defined as virtual so we can override it and provide our own implementation to return a custom ActionInvoker.

Hide Copy Code

```
    public IActionInvoker ActionInvoker {
    get {
        if (_actionInvoker == null) {
            _actionInvoker = CreateActionInvoker();
        }
        return _actionInvoker;
    }
    set {
       _actionInvoker = value;
    }
}
   protected virtual IActionInvoker CreateActionInvoker() {
    return new ControllerActionInvoker();
}
```

The ActionInvoker class needs to fetch the details of the action method and the controller to execute.These details are provided by the ControllerDescriptor .ControllerDescriptor and ActionDescriptor have an important role to play in the ActionInvoker.

ControllerDescriptor is defined as *"Encapsulates information that describes a controller, such as its name, type, and actions".*

ActionDescriptor is defined as "*Provides information about an action method, such as its name, controller, parameters, attributes, and filters*".

One important method of the ActionDescriptor is "FindAction()".This method returns an ActionDescriptor object representing the action to be executed. So ActionInvoker knows which action to call.

As we have seen above the ActionInvoker's InvokeAction() method is called in the ExecuteCore() method.

Following happens when the ActionInvoker's InvokeAction() method is called

- *The ActionInvoker has to get the information about the controller and the action to perform. This information is provided by descriptor object's .The action and controller descriptor class's provides the name of the controller and the action.*
- *ActionMethod is Invoked*

ActionResult

The command object

So we have seen till now that the ActionMethod is called by the ActionInvoker.

One of the characteristics of the action method is that instead of returning different data types it always returns the ActionResult type. ActionResult is an abstract class which is defined as:

Hide Copy Code

```
public abstract class ActionResult
  {
      public abstract void ExecuteResult(ControllerContext context);
  }
```

As the ExecuteResult() is an abstract method so different sub class's provides different implementations of the ExecuteResult() method.

One important thing to note is that an action result represents a command that the framework performs on behalf of the action method.As we know ActionMethods contains the logic that is executed and the result is returned to the client.Action methods themselves just return the ActionResult but don't execute it.

This ActionResult is executed and the response is returned back to the client .*So ActionResult object represents the result that can be passed across the methods.*

Thus it separates the specification from the implementation as it represents the command object.For understanding commands in .NET please refer to the commands.

There are specific ActionResult classes depending on the type of result we want to return like the Json or Redirection to another method.

The *"Controller"* class that we use to inherit our controller class's provides many useful features that we can use out of the box.

One such feature it provides is the methods which return's the specific types of ActionResult's . So instead of specifically creating the ActionResult object we can just call these methods.

```
public ActionResult Create()
{
    return View();
}
```

ViewResult Controller.View() (+ 7 overload(s))
Creates a System.Web.Mvc.ViewResult object that renders a view to the response.

Below are some of these methods and the type of ActionResult they return

ActionResult Class	Helper Method	ReturnType
ViewResult	View	web page
JsonResult	Json	Retuns a serialized JSON object
RedirectResult	Redirect	Redirects to another action method
ContentResult	Content	Returns a user-defined content type

So we have seen till now that the ActionMethod is called by the ActionInvoker

Following happens after the action method is invoked.

- *The OnActionExecuting methods of the ActionFilters are invoked.*
- *After this the action method itself is invoked.*
- *After the action method is invoked the OnActionExecuted methods of ActionFilters are invoked.*
- *The ActionResult is returned back from the ActionMethod*
- *The ExecuteResult() method of the ActionResult is called.*

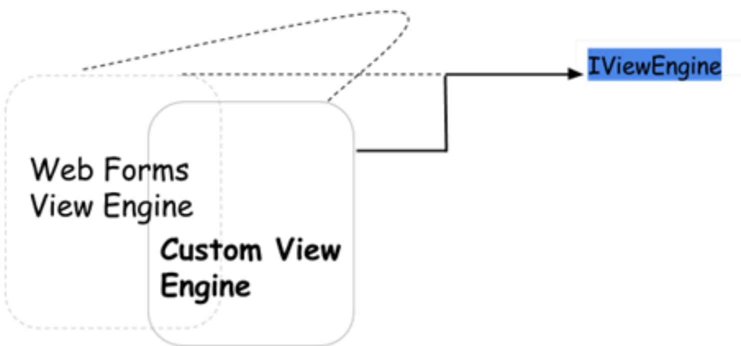
ViewEngine

The Renderer of the View

ViewResult is one of the most common return type used in almost all of the applications.It is used to render a view to the client using the ViewEngine. The view engine is responsible for generating the HTML from view

When ViewResult is called by the action invoker it renders the view to the response by overriding the ExecuteResult method.

The View engines provided by the framework are the Razor View Engine and Web Form View Engine. But if you need some custom view engine for some custom functionality *you can create a new view engine by implementing the IViewEngine interface which all the View Engine's implement.*



The IViewEngine has the following methods:

- FindPartialView The FindPartialView method is called when the Controller is looking to return a Partial View with the given Name.
- FindView The FindView method is called when Controller is looking for a View with a given Name.
- ReleaseView method is used for releasing the resources held by the ViewEngine.

But instead of implementing these methods an *easier way to create a view engine is to derive a new class from the abstract "VirtualPathProviderViewEngine" class.This class handles the low level details like finding the view's.*

Article

So we have seen above the of the ActionResult is called. Since ViewResult is the most common type of ActionResult we will look into what happens if the the ExecuteResult() method of the ViewResult is called.

There are two important classes ViewResultBase and ViewResult.ViewResultBase contains the following code that calls the FindViewMethod in ViewResult

Hide Copy Code

```
if (View == null)
            {
                result = FindView(context); //calls the ViewResult's FindView() method
                View = result.View;
            }

            ViewContext viewContext = new ViewContext(context, View, ViewData, TempData);
            View.Render(viewContext, context.HttpContext.Response.Output);

protected abstract ViewEngineResult FindView(ControllerContext context); //this is implemented by
                                                                           //the ViewResult
```

Hide Copy Code

```
protected override ViewEngineResult FindView(ControllerContext context)
    {
        ViewEngineResult result = ViewEngineCollection.FindView(context, ViewName, MasterName);
        if (result.View != null)
        {
            return result;
        }
       //rest of the code omitted
    }
```

Following happens after the ExecuteResult() method of ViewResult is called.

- *ExecuteResult of ViewResultBase is Invoked*
- *ViewResultBase calls the FindView of the ViewResult*
- *ViewResult returns the ViewEngineResult*
- *The Render() method of the ViewEngineResult is called to Render the view using the ViewEngine.*
- *The response is returned to the client.*

Summary

If we have understanding of what is happening under the hood we are better able to understand the role of each component and how the different components are connected to each other.We have looked into some of the main Interfaces and classes used by the framework to handle the response. I believe this article will be helpful to you in understanding the internal details of the MVC application.

Points of interest

One of the nice things about MVC is that all these components are loosely coupled.We can replace any of the component's in the pipeline with another one.This gives much freedom to the developer. This means that each stage in the request pipeline we have the choice of the appropriate component to handle the request.

We are free to provide our own implementation as well. This makes the applications more maintainable.

It is because of this loosely coupled architecture that makes MVC applications are suitable for testing.We can easily provide the mock objects in place of the actual objects since there is no concrete class dependency.

History

4 April 2014 Article of the day

License

This article, along with any associated source code and files, is licensed under The Code Project Open License (CPOL)

Share

EMAIL TWITTER

About the Author

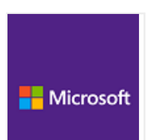
ashish__shukla

India

I have a keen interest in technology and software development.I have worked in C#,ASP.NET WebForms,MVC,HTML5 and SQL Server.I try to keep myself updated and learn and implement new technologies.
Please vote if you like my article ,also I would appreciate your suggestions.For more please visit Code Compiled

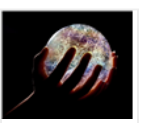
You may also be interested in...

The ASP.NET Page Lifecycle – A Basic Approach



Your Guide to Modern Dev/Test

Lifecycle Profile Settings

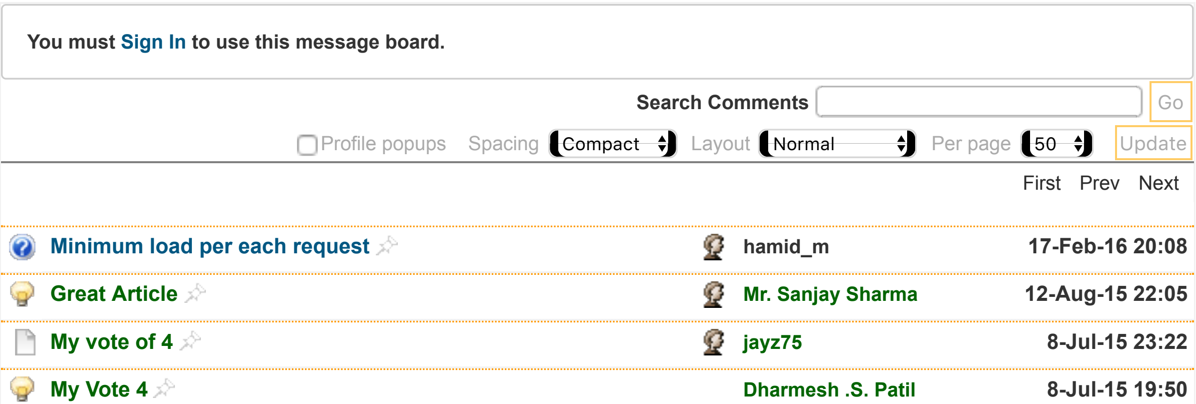


The Past, Present, and Future of IoT

BDD using SpecFlow on ASP.NET MVC Application

Java for Bluetooth LE applications

Comments and Discussions



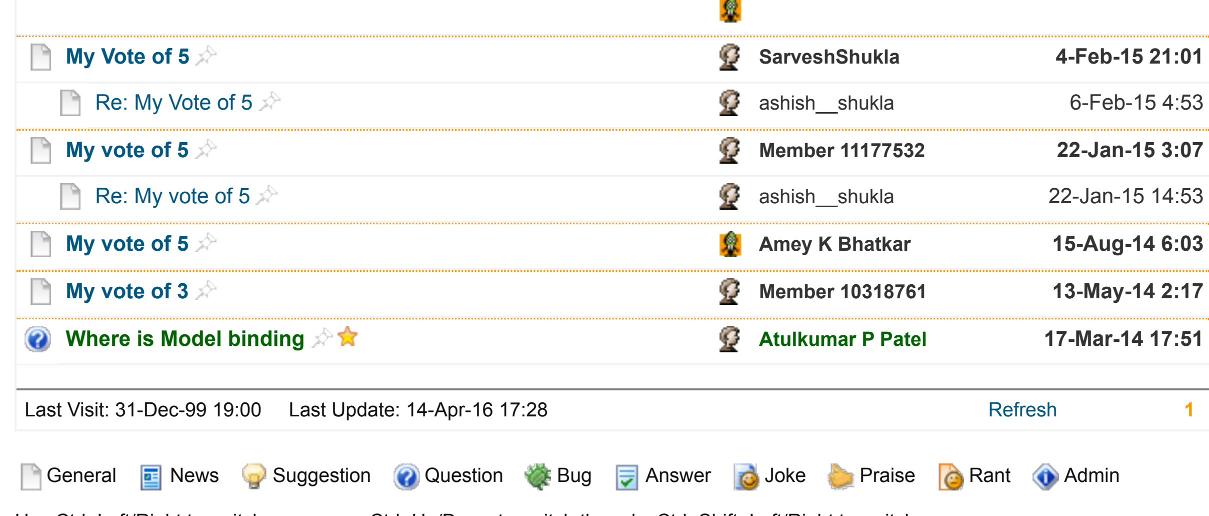
You must Sign In to use this message board.

Search Comments Go

Profile popups Spacing Compact Layout Normal Per page 50 Update

First Prev Next

Minimum load per each request	hamid_m	17-Feb-16 20:08
Great Article	Mr. Sanjay Sharma	12-Aug-15 22:05
My vote of 4	jayz75	8-Jul-15 23:22
My Vote 4	Dharmesh .S. Patil	8-Jul-15 19:50



Subject	Author	Date
My Vote of 5	**SarveshShukla**	**4-Feb-15 21:01**
Re: My Vote of 5	ashish__shukla	6-Feb-15 4:53
My vote of 5	**Member 11177532**	**22-Jan-15 3:07**
Re: My vote of 5	ashish__shukla	22-Jan-15 14:53
My vote of 5	**Amey K Bhatkar**	**15-Aug-14 6:03**
My vote of 3	**Member 10318761**	**13-May-14 2:17**
Where is Model binding	**Atulkumar P Patel**	**17-Mar-14 17:51**

Last Visit: 31-Dec-99 19:00 Last Update: 14-Apr-16 17:28 Refresh 1

General News Suggestion Question Bug Answer Joke Praise Rant Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.

Permalink | Advertise | Privacy | Terms of Use | Mobile
Web02 | 2.8.160413.1 | Last Updated 7 Apr 2014

Select Language ▼

Layout: fixed | fluid

Article Copyright 2014 by ashish__shukla
Everything else Copyright © CodeProject, 1999-2016