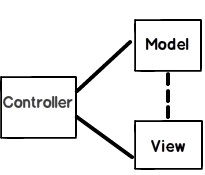
## What is MVC (Model View Controller)?

MVC is an architectural pattern which separates the representation and user interaction. It’s divided into three broader sections, Model, View, and Controller. Below is how each one of them handles the task. MVC framework is defined in System.Web.Mvc namespace.

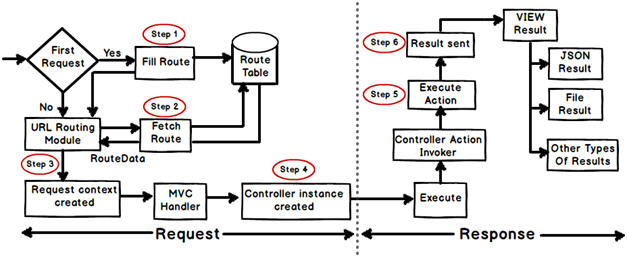
* The View is responsible for the look and feel.
* Model represents the real world object and provides data to the View.
* The Controller is responsible for taking the end user request and loading the appropriate Model and View.



***Figure: MVC (Model view controller)***

## Explain MVC application life cycle?

There are six broader events which occur in MVC application life cycle below diagrams summarize it.



Any web application has two main execution steps first understanding the request and depending on the type of the request sending out appropriate response. MVC application life cycle is not different it has two main phases first creating the request object and second sending our response to the browser.

**Creating the request object: -**The request object creation has four major steps. Below is the detail explanation of the same.

**Step 1 Fill route: -** MVC requests are mapped to route tables which in turn specify which controller and action to be invoked. So if the request is the first request the first thing is to fill the route table with routes collection. This filling of route table happens in the global.asax file.

**Step 2 Fetch route: -** Depending on the URL sent “UrlRoutingModule” searches the route table to create “RouteData” object which has the details of which controller and action to invoke.

**Step 3 Request context created: -** The “RouteData” object is used to create the “RequestContext” object.

**Step 4 Controller instance created: -** This request object is sent to “MvcHandler” instance to create the controller class instance. Once the controller class object is created it calls the “Execute” method of the controller class.

**Creating Response object: -** This phase has two steps executing the action and finally sending the response as a result to the view.

## Is MVC suitable for both Windows and Web applications?

The MVC architecture is suited for a web application than Windows. For Window applications, MVP, i.e., “Model View Presenter” is more applicable. If you are using WPF and Silverlight, MVVM is more suitable due to bindings.

## What are the benefits of using MVC?

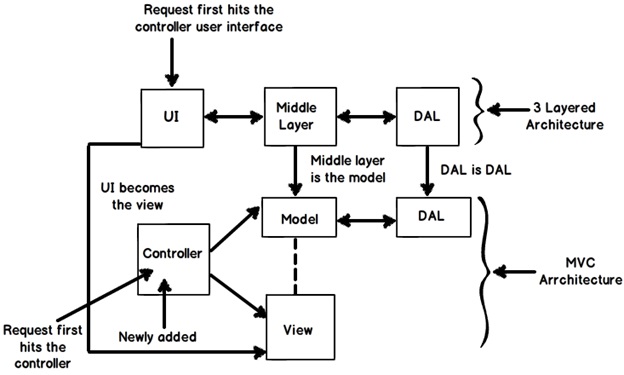
There are two big benefits of MVC:

* Separation of concerns is achieved as we are moving the code-behind to a separate class file. By moving the binding code to a separate class file we can reuse the code to a great extent.
* Automated UI testing is possible because now the behind code (UI interaction code) has moved to a simple .NET class. This gives us opportunity to write unit tests and automate manual testing.

## Is MVC different from a three layered architecture?

MVC is an evolution of a three layered traditional architecture. Many components of the three layered architecture are part of MVC. So below is how the mapping goes:

|  |  |  |
| --- | --- | --- |
| **Functionality** | **Three layered / tiered architecture** | **Model view controller architecture** |
| Look and Feel | User interface | View |
| UI logic | User interface | Controller |
| Business logic /validations | Middle layer | Model |
| Request is first sent to | User interface | Controller |
| Accessing data | Data access layer | Data Access Layer |



***Figure: Three layered architecture***

## What is the latest version of MVC?

MVC 6 is the latest version which is also termed as ASP VNEXT.

## What is the difference between each version of MVC 2, 3 , 4, 5 and 6?

|  |  |
| --- | --- |
| **MVC 5** | **MVC 6** |
| 1. One ASP.NET 2. Attribute based routing 3. Asp.Net Identity 4. Bootstrap in the MVC template 5. Authentication Filters 6. Filter overrides | 1. ASP.NET MVC and Web API has been merged in to one. 2. Dependency injection is inbuilt and part of MVC. 3. Side by side - deploy the runtime and framework with your application 4. Everything packaged with NuGet, Including the .NET runtime itself. 5. New JSON based project structure. 6. No need to recompile for every change. Just hit save and refresh the browser. 7. Compilation done with the new Roslyn real-time compiler. 8. vNext is Open Source via the .NET Foundation and is taking public contributions. 9. vNext (and Rosyln) also runs on Mono, on both Mac and Linux today. |

|  |  |  |
| --- | --- | --- |
| **MVC 2** | **MVC 3** | **MVC4** |
| 1. Client-Side Validation 2. Templated Helpers 3. Areas 4. Asynchronous Controllers 5. Html.ValidationSummary Helper Method 6. DefaultValueAttribute in Action-Method Parameters 7. Binding Binary Data with Model Binders 8. DataAnnotations Attributes 9. Model-Validator Providers 10. New RequireHttpsAttribute Action Filter 11. Templated Helpers 12. Display Model-Level Errors | 1. Razor 2. Readymade project templates 3. HTML 5 enabled templates 4. Support for Multiple View Engines 5. JavaScript and Ajax 6. Model Validation Improvements | 1. ASP.NET Web API 2. Refreshed and modernized default project templates 3. New mobile project template 4. Many new features to support mobile apps 5. Enhanced support for asynchronous methods |

## What are HTML helpers in MVC?

HTML helpers help you to render HTML controls in the view. For instance if you want to display a HTML textbox on the view , below is the HTML helper code.

<%= Html.TextBox("LastName") %>

For checkbox below is the HTML helper code. In this way we have HTML helper methods for every HTML control that exists.

<%= Html.CheckBox("Married") %>

## What is the difference between “HTML.TextBox” vs “HTML.TextBoxFor”?

Both of them provide the same HTML output, “HTML.TextBoxFor” is strongly typed while “HTML.TextBox” isn’t. Below is a simple HTML code which just creates a simple textbox with “CustomerCode” as name.

Html.TextBox("CustomerCode")

Below is “Html.TextBoxFor” code which creates HTML textbox using the property name ‘CustomerCode” from object “m”.

Html.TextBoxFor(m => m.CustomerCode)

In the same way we have for other HTML controls like for checkbox we have “Html.CheckBox” and “Html.CheckBoxFor”.

## What is routing in MVC?

Routing helps you to define a URL structure and map the URL with the controller.

For instance let’s say we want that when a user types “*http://localhost/View/ViewCustomer/*”, it goes to the “Customer” Controller and invokes the DisplayCustomer action. This is defined by adding an entry in to theroutes collection using the maproute function. Below is the underlined code which shows how the URL structure and mapping with controller and action is defined.

routes.MapRoute(

"View", // Route name

"View/ViewCustomer/{id}", // URL with parameters

new { controller = "Customer", action = "DisplayCustomer",

id = UrlParameter.Optional }); // Parameter defaults

## Where is the route mapping code written?

The route mapping code is written in "RouteConfig.cs" file and registered using "global.asax" application start event.

## Can we map multiple URL’s to the same action?

Yes, you can, you just need to make two entries with different key names and specify the same controller and action.

## Explain attribute based routing in MVC?

This is a feature introduced in MVC 5. By using the "Route" attribute we can define the URL structure. For example in the below code we have decorated the "GotoAbout" action with the route attribute. The route attribute says that the "GotoAbout" can be invoked using the URL structure "Users/about".

public class HomeController : Controller

{

[Route("Users/about")]

public ActionResult GotoAbout()

{

return View();

}

}

## What is the advantage of defining route structures in the code?

Most of the time developers code in the action methods. Developers can see the URL structure right upfront rather than going to the “routeconfig.cs” and see the lengthy codes. For instance in the below code the developer can see right upfront that the “GotoAbout” action can be invoked by four different URL structure.

This is much user friendly as compared to scrolling through the “routeconfig.cs” file and going through the length line of code to figure out which URL structure is mapped to which action.

public class HomeController : Controller

{

[Route("Users/about")]

[Route("Users/WhoareWe")]

[Route("Users/OurTeam")]

[Route("Users/aboutCompany")]

public ActionResult GotoAbout()

{

return View();

}

}

## How can we navigate from one view to another using a hyperlink?

By using the ActionLink method as shown in the below code. The below code will create a simple URL which helps to navigate to the “Home” controller and invoke the GotoHome action.

<%= Html.ActionLink("Home","Gotohome") %>

## How can we restrict MVC actions to be invoked only by GET or POST?

We can decorate the MVC action with the HttpGet or HttpPost attribute to restrict the type of HTTP calls. For instance you can see in the below code snippet the DisplayCustomer action can only be invoked by HttpGet. If we try to make HTTP POST on DisplayCustomer, it will throw an error.

[HttpGet]

public ViewResult DisplayCustomer(int id)

{

Customer objCustomer = Customers[id];

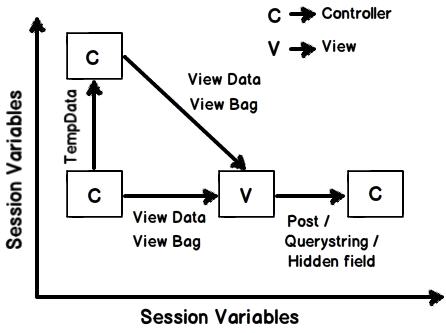
return View("DisplayCustomer",objCustomer);

}

## How can we maintain sessions in MVC?

Sessions can be maintained in MVC by three ways: tempdata, viewdata, and viewbag.

## What is the difference between tempdata, viewdata, and viewbag?

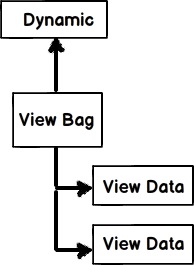


***Figure: Difference between tempdata, viewdata, and viewbag***

**Temp data** - Helps to maintain data when you move from one controller to another controller or from one action to another action. In other words when you redirect, tempdata helps to maintain data between those redirects. It internally uses session variables.

**View data** - Helps to maintain data when you move from controller to view.

**View Bag** - It’s a dynamic wrapper around view data. When you use Viewbag type, casting is not required. It uses the dynamic keyword internally.



***Figure: dynamic keyword***

**Session variables -** By using session variables we can maintain data from any entity to any entity.

**Hidden fields and HTML controls -** Helps to maintain data from UI to controller only. So you can send data from HTML controls or hidden fields to the controller using POST or GET HTTP methods.

Below is a summary table which shows the different mechanisms for persistence.

| **Maintains data between** | **ViewData/ViewBag** | **TempData** | **Hidden fields** | **Session** |
| --- | --- | --- | --- | --- |
| **Controller to Controller** | No | Yes | No | Yes |
| **Controller to View** | Yes | No | No | Yes |
| **View to Controller** | No | No | Yes | Yes |

## What is difference between TempData and ViewData ?

“TempData” maintains data for the complete request while “ViewData” maintains data only from Controller to the view.

## Does “TempData” preserve data in the next request also?

“TempData” is available through out for the current request and in the subsequent request it’s available depending on whether “TempData” is read or not.

So if “TempData” is once read it will not be available in the subsequent request.

## What is the use of Keep and Peek in “TempData”?

Once “TempData” is read in the current request it’s not available in the subsequent request. If we want “TempData” to be read and also available in the subsequent request then after reading we need to call “Keep” method as shown in the code below.

@TempData[&ldquo;MyData&rdquo;];

TempData.Keep(&ldquo;MyData&rdquo;);

The more shortcut way of achieving the same is by using “Peek”. This function helps to read as well advices MVC to maintain “TempData” for the subsequent request.

string str = TempData.Peek("Td").ToString();

If you want to read more in detail you can read from this detailed blog on [MVC Peek and Keep](http://www.codeproject.com/Articles/818493/MVC-Tempdata-Peek-and-Keep-confusion).

## What are partial views in MVC?

Partial view is a reusable view (like a user control) which can be embedded inside other view. For example let’s say all your pages of your site have a standard structure with left menu, header, and footer as shown in the image below.

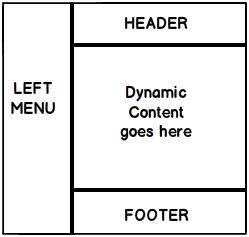
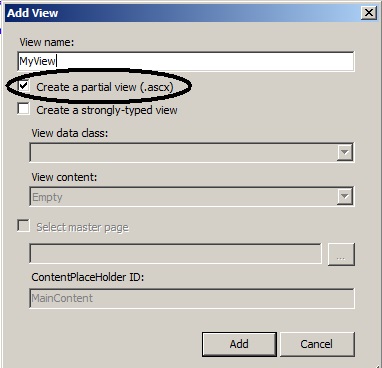


Figure: Partial views in MVC

For every page you would like to reuse the left menu, header, and footer controls. So you can go and create partial views for each of these items and then you call that partial view in the main view.

## How did you create a partial view and consume it?

When you add a view to your project you need to check the “Create partial view” check box.



Once the partial view is created you can then call the partial view in the main view using theHtml.RenderPartial method as shown in the below code snippet:

<body>

<div>

<% Html.RenderPartial("MyView"); %>

</div>

</body>

## How can we do validations in MVC?

One of the easiest ways of doing validation in MVC is by using data annotations. Data annotations are nothing but attributes which can be applied on model properties. For example, in the below code snippet we have a simple Customer class with a property customercode.

This CustomerCode property is tagged with a Required data annotation attribute. In other words if this model is not provided customer code, it will not accept it.

public class Customer

{

[Required(ErrorMessage="Customer code is required")]

public string CustomerCode

{

set;

get;

}

}

In order to display the validation error message we need to use the ValidateMessageFor method which belongs to he Html helper class.

<% using (Html.BeginForm("PostCustomer", "Home", FormMethod.Post))

{ %>

<%=Html.TextBoxFor(m => m.CustomerCode)%>

<%=Html.ValidationMessageFor(m => m.CustomerCode)%>

<input type="submit" value="Submit customer data" />

<%}%>

Later in the controller we can check if the model is proper or not by using the ModelState.IsValid property and accordingly we can take actions.

public ActionResult PostCustomer(Customer obj)

{

if (ModelState.IsValid)

{

obj.Save();

return View("Thanks");

}

else { return View("Customer"); }

}

Below is a simple view of how the error message is displayed on the view.

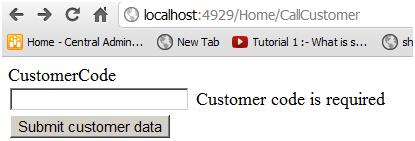


Figure: Validations in MVC

## Can we display all errors in one go?

Yes, we can; use the ValidationSummary method from the Html helper class.

<%= Html.ValidationSummary() %>

## What are the other data annotation attributes for validation in MVC?

If you want to check string length, you can use StringLength.

[StringLength(160)]

public string FirstName { get; set; }

In case you want to use a regular expression, you can use the RegularExpression attribute.

[RegularExpression(@"[A-Za-z0-9.\_%+-][+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}](mailto:+@[A-Za-z0-9.-]+\.%5bA-Za-z%5d%7b2,4%7d)")]

public string Email { get; set; }

If you want to check whether the numbers are in range, you can use the Range attribute.

[Range(10,25)]

public int Age { get; set; }

Sometimes you would like to compare the value of one field with another field, we can use the Compareattribute.

public string Password { get; set; }

[Compare("Password")]

public string ConfirmPass { get; set; }

In case you want to get a particular error message, you can use the Errors collection.

var ErrMessage = ModelState["Email"].Errors[0].ErrorMessage;

If you have created the model object yourself you can explicitly call TryUpdateModel in your controller to check if the object is valid or not.

TryUpdateModel(NewCustomer);

In case you want add errors in the controller you can use the AddModelError function.

ModelState.AddModelError("FirstName", "This is my server-side error.");

## How can we enable data annotation validation on client side?

It’s a two-step process: first reference the necessary jQuery files.

<script src="<%= Url.Content("~/Scripts/jquery-1.5.1.js") %>" type="text/javascript"></script>

The second step is to call the EnableClientValidation method.

<% Html.EnableClientValidation(); %>

## What is Razor in MVC?

It’s a light weight view engine. Till MVC we had only one view type, i.e., ASPX. Razor was introduced in MVC 3.

## Why Razor when we already have ASPX?

Razor is clean, lightweight, and syntaxes are easy as compared to ASPX. For example, in ASPX to display simple time, we need to write:

<%=DateTime.Now%>

In Razor, it’s just one line of code:

@DateTime.Now

## So which is a better fit, Razor or ASPX?

As per Microsoft, Razor is more preferred because it’s light weight and has simple syntaxes.

## How can you do authentication and authorization in MVC?

You can use Windows or Forms authentication for MVC.

## How to implement Windows authentication for MVC?

For Windows authentication you need to modify the web.config file and set the authentication mode to Windows.

<authentication mode="Windows"/>

<authorization>

<deny users="?"/>

</authorization>

Then in the controller or on the action, you can use the Authorize attribute which specifies which users have access to these controllers and actions. Below is the code snippet for that. Now only the user controller and action can access it.

[Authorize(Users= @"WIN-3LI600MWLQN\Administrator")]

public class StartController : Controller

{

//

// GET: /Start/

[Authorize(Users = @"WIN-3LI600MWLQN\Administrator")]

public ActionResult Index()

{

return View("MyView");

}

}

## How do you implement Forms authentication in MVC?

Forms authentication is implemented the same way as in ASP.NET. The first step is to set the authentication mode equal to Forms. The loginUrl points to a controller here rather than a page.

<authentication mode="Forms">

<forms loginUrl="~/Home/Login" timeout="2880"/>

</authentication>

We also need to create a controller where we will check if the user is proper or not. If the user is proper we will set the cookie value.

public ActionResult Login()

{

if ((Request.Form["txtUserName"] == "Shiv") &&

(Request.Form["txtPassword"] == "Shiv@123"))

{

FormsAuthentication.SetAuthCookie("Shiv",true);

return View("About");

}

else

{

return View("Index");

}

}

All the other actions need to be attributed with the Authorize attribute so that any unauthorized user making a call to these controllers will be redirected to the controller (in this case the controller is “Login”) which will do the authentication.

[Authorize]

PublicActionResult Default()

{

return View();

}

[Authorize]

publicActionResult About() { return View(); }

## How to implement AJAX in MVC?

You can implement AJAX in two ways in MVC:

* AJAX libraries
* jQuery

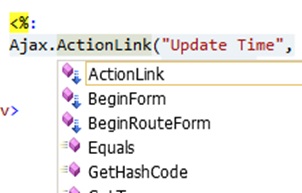
Below is a simple sample of how to implement AJAX by using the “AJAX” helper library. In the below code you can see we have a simple form which is created by using the Ajax.BeginForm syntax. This form calls a controller action called getCustomer. So now the submit action click will be an asynchronous AJAX call.

<script language="javascript">

function OnSuccess(data1) { // Do something here }

</script>

In case you want to make AJAX calls on hyperlink clicks, you can use the Ajax.ActionLink function as shown in the below code.



***Figure: Implement AJAX in MVC***

So if you want to create an AJAX asynchronous hyperlink by name GetDate which calls the GetDate function in the controller, below is the code for that. Once the controller responds, this data is displayed in the HTML DIVtag named DateDiv.

<span id="DateDiv" />

<%:

Ajax.ActionLink("Get Date","GetDate",

new AjaxOptions {UpdateTargetId = "DateDiv" })

%>

Below is the controller code. You can see how the GetDate function has a pause of 10 seconds.

public class Default1Controller : Controller

{

public string GetDate()

{

Thread.Sleep(10000);

return DateTime.Now.ToString();

}

}

The second way of making an AJAX call in MVC is by using jQuery. In the below code you can see we are making an AJAX POST call to a URL /MyAjax/getCustomer. This is done by using $.post. All this logic is put into a function called GetData and you can make a call to the GetData function on a button or a hyperlink click event as you want.

function GetData()

{

var url = "/MyAjax/getCustomer";

$.post(url, function (data)

{

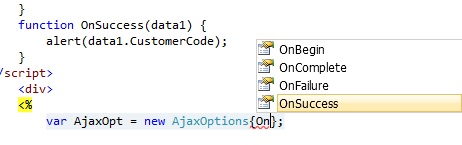
$("#txtCustomerCode").val(data.CustomerCode);

$("#txtCustomerName").val(data.CustomerName);

}

}

## What kind of events can be tracked in AJAX?



***Figure: Tracked in AJAX***

## What is the difference between ActionResult and ViewResult?

ActionResult is an abstract class while ViewResult derives from the ActionResult class.ActionResult has several derived classes like ViewResult, JsonResult, FileStreamResult, and so on.

ActionResult can be used to exploit polymorphism and dynamism. So if you are returning different types of views dynamically, ActionResult is the best thing. For example in the below code snippet, you can see we have a simple action called DynamicView. Depending on the flag (IsHtmlView) it will either return a ViewResult or JsonResult.

public ActionResult DynamicView()

{

if (IsHtmlView)

return View(); // returns simple ViewResult

else

return Json(); // returns JsonResult view

}

## What are the different types of results in MVC?

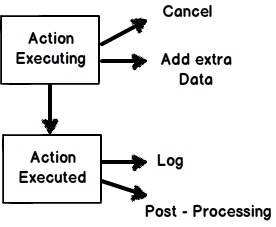
Note: It’s difficult to remember all the 12 types. But some important ones you can remember for the interview are ActionResult, ViewResult, and JsonResult. Below is a detailed list for your interest:

There 12 kinds of results in MVC, at the top is the ActionResult class which is a base class that can have 11 subtypes as listed below:

* **ViewResult** - Renders a specified view to the response stream
* **PartialViewResult** - Renders a specified partial view to the response stream
* **EmptyResult** - An empty response is returned
* **RedirectResult** - Performs an HTTP redirection to a specified URL
* **RedirectToRouteResult** - Performs an HTTP redirection to a URL that is determined by the routing engine, based on given route data
* **JsonResult** - Serializes a given ViewData object to JSON format
* **JavaScriptResult** - Returns a piece of JavaScript code that can be executed on the client
* **ContentResult** - Writes content to the response stream without requiring a view
* **FileContentResult** - Returns a file to the client
* **FileStreamResult** - Returns a file to the client, which is provided by a Stream
* **FilePathResult** - Returns a file to the client

## What are ActionFilters in MVC?

ActionFilters help you to perform logic while an MVC action is executing or after an MVC action has executed.



***Figure: ActionFilters in MVC***

Action filters are useful in the following scenarios:

* Implement post-processing logic before the action happens.
* Cancel a current execution.
* Inspect the returned value.
* Provide extra data to the action.

You can create action filters by two ways:

* Inline action filter.
* Creating an ActionFilter attribute.

To create an inline action attribute we need to implement the IActionFilter interface. The IActionFilterinterface has two methods: OnActionExecuted and OnActionExecuting. We can implement pre-processing logic or cancellation logic in these methods.

public class Default1Controller : Controller , IActionFilter

{

public ActionResult Index(Customer obj) { return View(obj); }

void IActionFilter.OnActionExecuted(ActionExecutedContext filterContext)

{

Trace.WriteLine("Action Executed");

}

void IActionFilter.OnActionExecuting(ActionExecutingContext filterContext)

{

Trace.WriteLine("Action is executing");

}

}

The problem with the inline action attribute is that it cannot be reused across controllers. So we can convert the inline action filter to an action filter attribute. To create an action filter attribute we need to inherit fromActionFilterAttribute and implement the IActionFilter interface as shown in the below code.

public class MyActionAttribute : ActionFilterAttribute , IActionFilter

{

void IActionFilter.OnActionExecuted(ActionExecutedContext filterContext)

{

Trace.WriteLine("Action Executed");

}

void IActionFilter.OnActionExecuting(ActionExecutingContext filterContext)

{

Trace.WriteLine("Action executing");

}

}

Later we can decorate the controllers on which we want the action attribute to execute. You can see in the below code I have decorated the Default1Controller with the MyActionAttribute class which was created in the previous code.

[MyActionAttribute]

public class Default1Controller : Controller

{

public ActionResult Index(Customer obj)

{

return View(obj);

}

}

## What are the different types of action filters?

* Authorization filters
* Action filters
* Result filters
* Exception filters

## If we have multiple filters, what’s the sequence for execution?

* Authorization filters
* Action filters
* Response filters
* Exception filters

## Can we create our custom view engine using MVC?

Yes, we can create our own custom view engine in MVC. To create our own custom view engine we need to follow three steps:

Let’ say we want to create a custom view engine where in the user can type a command like “<DateTime>” and it should display the current date and time.

**Step 1**: We need to create a class which implements the IView interface. In this class we should write the logic of how the view will be rendered in the render function. Below is a simple code snippet for that.

public class MyCustomView : IView

{

private string \_FolderPath; // Define where our views are stored

public string FolderPath

{

get { return \_FolderPath; }

set { \_FolderPath = value; }

}

public void Render(ViewContext viewContext, System.IO.TextWriter writer)

{

// Parsing logic <dateTime>

// read the view file

string strFileData = File.ReadAllText(\_FolderPath);

// we need to and replace <datetime> datetime.now value

string strFinal= strFileData.Replace("<DateTime>",DateTime.Now.ToString());

// this replaced data has to sent for display

writer.Write(strFinal);

}

}

**Step 2**: We need to create a class which inherits from VirtualPathProviderViewEngine and in this class we need to provide the folder path and the extension of the view name. For instance, for Razor the extension is “cshtml”; for aspx, the view extension is “.aspx”, so in the same way for our custom view, we need to provide an extension. Below is how the code looks like. You can see the ViewLocationFormats is set to the *Views* folder and the extension is “*.myview*”.

public class MyViewEngineProvider : VirtualPathProviderViewEngine

{

// We will create the object of Mycustome view

public MyViewEngineProvider() **// constructor**

{

// Define the location of the View file

this.ViewLocationFormats = new string[] { "~/Views/{1}/{0}.myview",

"~/Views/Shared/{0}.myview" }; //location and extension of our views

}

protected override IView CreateView(ControllerContext controllerContext,

string viewPath, string masterPath)

{

var physicalpath = controllerContext.HttpContext.Server.MapPath(viewPath);

MyCustomView obj = new MyCustomView(); **// Custom view engine class**

obj.FolderPath = physicalpath; **// set the path where the views**

**// will be stored**

**// returned this view paresing logic so that it can be registered in the**

**// view engine collection**

return obj;

}

protected override IView CreatePartialView(ControllerContext controllerContext,

string partialPath)

{

var physicalpath=controllerContext.HttpContext.Server.MapPath(partialPath);

MyCustomView obj = new MyCustomView(); **// Custom view engine class**

obj.FolderPath = physicalpath; **// set the path where the views**

**// will be stored**

**// returned this view paresing logic so that it can be registered in the**

**// view engine collection**

return obj;

}

**Step 3**: We need to register the view in the custom view collection. The best place to register the custom view engine in the ViewEngines collection is the *global.asax* file. Below is the code snippet for that.

protected void Application\_Start()

{

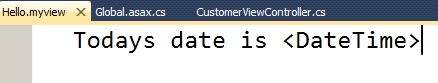
// Step3:- register this object in the view engine collection

ViewEngines.Engines.Add(new MyViewEngineProvider());

&hellip;..

}

Below is a simple output of the custom view written using the commands defined at the top.



**Figure: Custom view engine using MVC**

If you invoke this view, you should see the following output:



## How to send result back in JSON format in MVC

In MVC, we have the JsonResult class by which we can return back data in JSON format. Below is a simple sample code which returns back a Customer object in JSON format using JsonResult.

public JsonResult getCustomer()

{

Customer obj = new Customer();

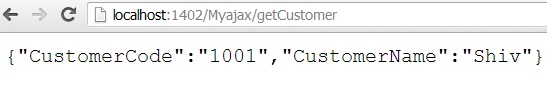
obj.CustomerCode = "1001";

obj.CustomerName = "Shiv";

return Json(obj,JsonRequestBehavior.AllowGet);

}

Below is the JSON output of the above code if you invoke the action via the browser.



## What is WebAPI?

HTTP is the most used protocol. For the past many years, browser was the most preferred client by which we consumed data exposed over HTTP. But as years passed by, client variety started spreading out. We had demand to consume data on HTTP from clients like mobile, JavaScript, Windows applications, etc.

For satisfying the broad range of clients REST was the proposed approach. You can read more about REST from the WCF chapter.

WebAPI is the technology by which you can expose data over HTTP following REST principles.

## But WCF SOAP also does the same thing, so how does WebAPI differ?

|  | SOAP | WEB API |
| --- | --- | --- |
| Size | Heavy weight because of complicated WSDL structure. | Light weight, only the necessary information is transferred. |
| Protocol | Independent of protocols. | Only for HTTP protocol |
| Formats | To parse SOAP message, the client needs to understand WSDL format. Writing custom code for parsing WSDL is a heavy duty task. If your client is smart enough to create proxy objects like how we have in .NET (add reference) then SOAP is easier to consume and call. | Output of WebAPI are simple string messages, JSON, simple XML format, etc. So writing parsing logic for that is very easy. |
| Principles | SOAP follows WS-\* specification. | WebAPI follows REST principles. (Please refer to REST in WCF chapter.) |

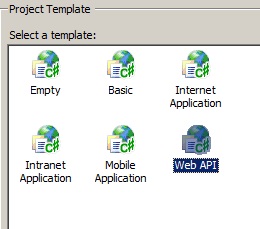
## With WCF you can implement REST, so why WebAPI?

WCF was brought into implement SOA, the intention was never to implement REST. WebAPI is built from scratch and the only goal is to create HTTP services using REST. Due to the one point focus for creating REST service, WebAPI is more preferred.

## How to implement WebAPI in MVC

Below are the steps to implement WebAPI:

**Step 1**: Create the project using the WebAPI template.



***Figure: Implement WebAPI in MVC***

**Step 2**: Once you have created the project you will notice that the controller now inherits from ApiControllerand you can now implement POST, GET, PUT, and DELETE methods of the HTTP protocol.

public class ValuesController : ApiController

{

// GET api/values

public IEnumerable<string> Get()

{

return new string[] { "value1", "value2" };

}

// GET api/values/5

public string Get(int id) { return "value"; }

// POST api/values

public void Post([FromBody]string value) { }

// PUT api/values/5

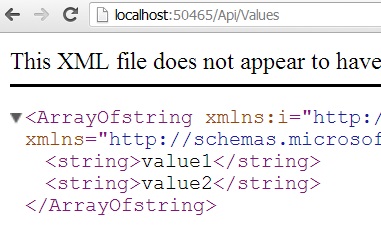
public void Put(int id, [FromBody]string value) { }

// DELETE api/values/5

public void Delete(int id) { }

}

**Step 3**: If you make an HTTP GET call you should get the below results:



***Figure: HTTP***

## How can we detect that an MVC controller is called by POST or GET?

To detect if the call on the controller is a POST action or a GET action we can use the Request.HttpMethodproperty as shown in the below code snippet.

public ActionResult SomeAction()

{

if (Request.HttpMethod == "POST")

{

return View("SomePage");

}

else

{

return View("SomeOtherPage");

}

}

## What is bundling and minification in MVC?

Bundling and minification helps us improve request load times of a page thus increasing performance.

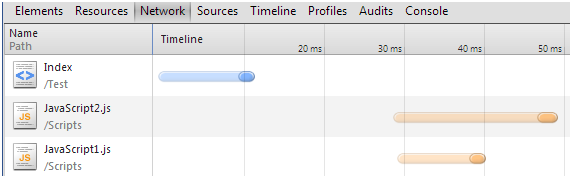
## How does bundling increase performance?

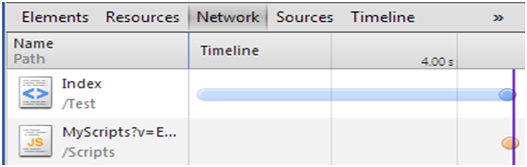
Web projects always need CSS and script files. Bundling helps us combine multiple JavaScript and CSS files in to a single entity thus minimizing multiple requests in to a single request.

For example consider the below web request to a page . This page consumes two JavaScript files Javascript1.jsand Javascript2.js. So when this is page is requested it makes three request calls:

* One for the Index page.
* Two requests for the other two JavaScript files: Javascript1.js and Javascript2.js.

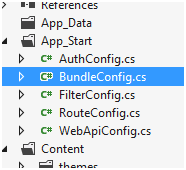
The below scenario can become worse if we have a lot of JavaScript files resulting in multiple requests, thus decreasing performance. If we can somehow combine all the JS files into a single bundle and request them as a single unit that would result in increased performance (see the next figure which has a single request).





## So how do we implement bundling in MVC?

Open BundleConfig.cs from the App\_Start folder.



In BundleConfig.cs, add the JS files you want bundle into a single entity in to the bundles collection. In the below code we are combining all the javascript JS files which exist in the Scripts folder as a single unit in to the bundle collection.

bundles.Add(new ScriptBundle("~/Scripts/MyScripts").Include(

"~/Scripts/\*.js"));

Below is how your *BundleConfig.cs* file will look like:

public class BundleConfig

{

public static void RegisterBundles(BundleCollection bundles)

{

bundles.Add(new ScriptBundle("~/Scripts/MyScripts").Include(

"~/Scripts/\*.js"));

BundleTable.EnableOptimizations = true;

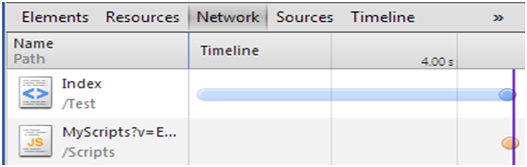
}

}

Once you have combined your scripts into one single unit we then to include all the JS files into the view using the below code. The below code needs to be put in the ASPX or Razor view.

<%= Scripts.Render("~/Scripts/MyScripts") %>

If you now see your page requests you would see that script request is combined into one request.



## How can you test bundling in debug mode?

If you are in a debug mode you need to set EnableOptimizations to true in the bundleconfig.cs file or else you will not see the bundling effect in the page requests.

BundleTable.EnableOptimizations = true;

## Explain minification and how to implement it

Minification reduces the size of script and CSS files by removing blank spaces , comments etc. For example below is a simple javascript code with comments.

// This is test

var x = 0;

x = x + 1;

x = x \* 2;

After implementing minification the JavaScript code looks like below. You can see how whitespaces and comments are removed to minimize file size, thus increasing performance.

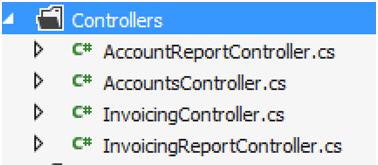
var x=0;x=x+1;x=x\*2;

## How do we implement minification?

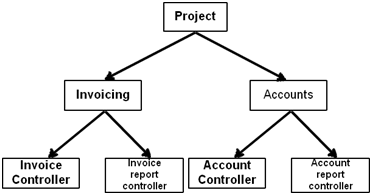
When you implement bundling, minification is implemented by itself. In other words the steps to implement bundling and minification are the same.

## Explain Areas in MVC?

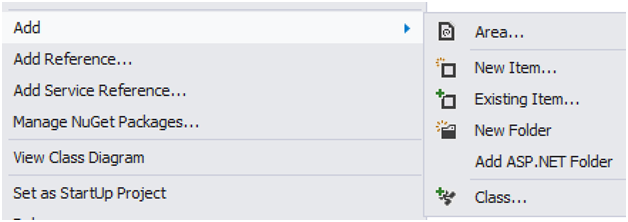
Areas help you to group functionalities in to independent modules thus making your project more organized. For example in the below MVC project we have four controller classes and as time passes by if more controller classes are added it will be difficult to manage. In bigger projects you will end up with 100’s of controller classes making life hell for maintenance.



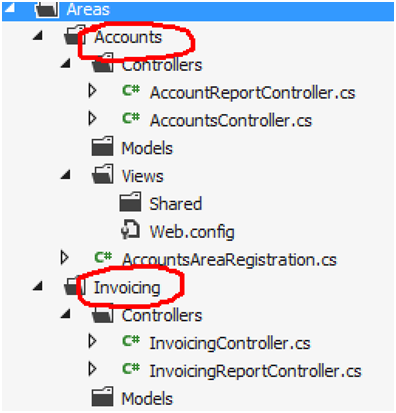
If we can group controller classes in to logical section like “Invoicing” and “Accounting” that would make life easier and that’s what “Area” are meant to.



You can add an area by right clicking on the MVC solution and clicking on “Area” menu as shown in the below figure.



In the below image we have two “Areas” created “Account” and “Invoicing” and in that I have put the respective controllers. You can see how the project is looking more organized as compared to the previous state.



## Explain the concept of View Model in MVC?

A view model is a simple class which represents data to be displayed on the view.

For example below is a simple customermodel object with “CustomerName” and “Amount” property.

CustomerViewModel obj = new CustomerViewModel();

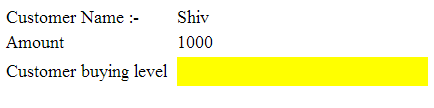
obj.Customer.CustomerName = "Shiv";

obj.Customer.Amount = 1000;

But when this “Customer” model object is displayed on the MVC view it looks something as shown in the below figure. It has “CustomerName” , “Amount” plus “Customer Buying Level” fields on the view / screen. “Customer buying Level” is a color indicationwhich indicates how aggressive the customer is buying.

“Customer buying level” color depends on the value of the “Amount property. If the amount is greater than 2000 then color is red , if amount is greater than 1500 then color is orange or else the color is yellow.

In other words “Customer buying level” is an extra property which is calculated on the basis of amount.



So the Customer viewmodel class has three properties

* “TxtCustomerName” textbox takes data from “CustomerName” property as it is.
* “TxtAmount” textbox takes data from “Amount” property of model as it is.
* “CustomerBuyingLevelColor” displays color value depending on the “Amount “ value.

|  |  |
| --- | --- |
| **Customer Model** | **Customer ViewModel** |
| CustomerName | TxtCustomerName |
| Amount | TxtAmount |
|  | CustomerBuyingLevelColor |

## What kind of logic view model class will have?

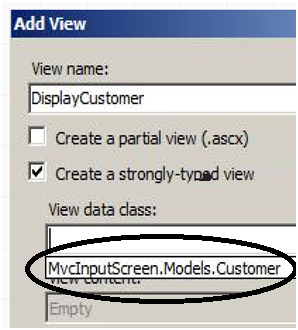
As the name says view model this class has the gel code or connection code which connects the view and the model.

So the view model class can have following kind of logics:-

* **Color transformation logic: -**For example you have a “Grade” property in model and you would like your UI to display “red” color for high level grade, “yellow” color for low level grade and “green” color of ok grade.
* **Data format transformation logic :-**Your model has a property “Status” with “Married” and “Unmarried” value. In the UI you would like to display it as a checkbox which is checked if “married” and unchecked if “unmarried”.
* **Aggregation logic: -**You have two differentCustomer and Address model classes and you have view which displays both “Customer” and “Address” data on one go.
* **Structure downsizing: -**You have “Customer” model with “customerCode” and “CustomerName” and you want to display just “CustomerName”. So you can create a wrapper around model and expose the necessary properties.

## How can we use two (multiple) models with a single view?

Let us first try to understand what the interviewer is asking. When we bind a model with a view we use the model dropdown as shown in the below figure. In the below figure we can only select one model.



But what if we want to bind “Customer” as well as “Order” class to the view.

For that we need to create a view model which aggregates both the classes as shown in the below code. And then bind that view model with the view.

public class CustOrderVM

{

public Customer cust = new Customer();

public Order Ord = new Order();

}

In the view we can refer both the model using the view model as shown in the below code.

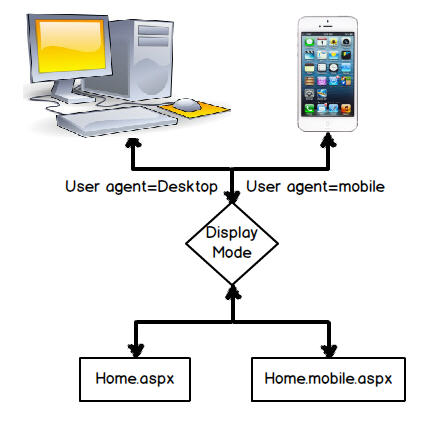
<%= model.cust.Name %>

<%= model.Ord.Number %>

## Explain the need of display mode in MVC?

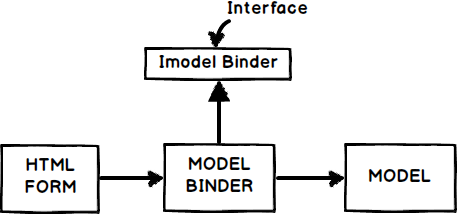
Display mode displays views depending on the device the user has logged in with. So we can create different views for different devices anddisplay mode will handle the rest.

For example we can create a view “Home.aspx” which will render for the desktop computers and[Home.Mobile.aspx](http://www.codeproject.com/Articles/556995/Home.Mobile.aspx) for mobile devices. Now when an end user sends a request to the MVC application, display mode checks the “user agent” headers and renders the appropriate view to the device accordingly.



## Explain MVC model binders?

Model binder maps HTML form elements to the model. It acts like a bridge between HTML UI and MVC model. Many times HTML UI names are different than the model property names. So in the binder we can write the mapping logic between the UI and the model.

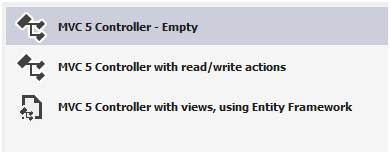


## Explain the concept of MVC Scaffolding?

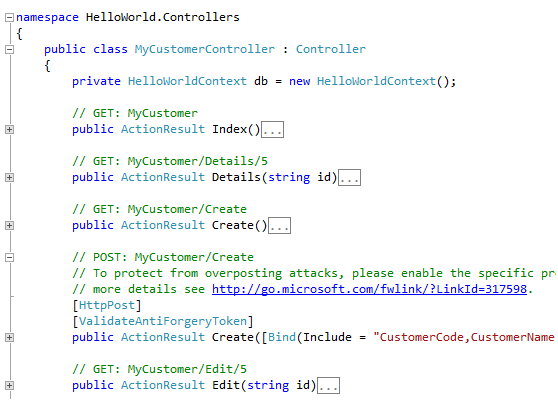
Note :- Do not get scared with the word. Its actually a very simple thing.

Scaffolding is a technique in which the MVC template helps to auto-generate CRUD code. CRUD stands for create, read, update and delete.

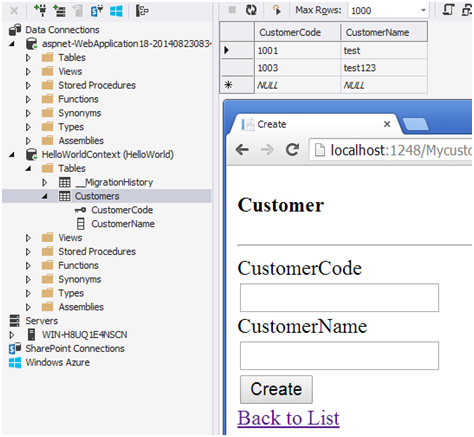
So to generate code using scaffolding technique we need to select one of the types of templates (leave the empty one).



For instance if you choose “using Entity framework” template the following code is generated.



It creates controller code, view and also table structure as shown in the below figure.



## What does scaffolding use internally to connect to database?

It uses Entity framework internally.

## How can we do exception handling in MVC?

In the controller you can override the “OnException” event and set the “Result” to the view name which you want to invoke when error occurs. In the below code you can see we have set the “Result” to a view named as “Error”.

We have also set the exception so that it can be displayed inside the view.

public class HomeController : Controller

{

protected override void OnException(ExceptionContext filterContext)

{

Exception ex = filterContext.Exception;

filterContext.ExceptionHandled = true;

var model = new HandleErrorInfo(filterContext.Exception,

"Controller","Action");

filterContext.Result = new ViewResult()

{

ViewName = "Error",

ViewData = new ViewDataDictionary(model)

};

}

}

To display the above error in view we can use the below code

@Model.Exception;

## How can you handle multiple Submit buttons pointing to multiple actions in a single MVC view?

Let us elaborate on what the interviewer wants to ask because the above question is just a single liner and is not clear about what the interviewer wants.

Take a scenario where you have a view with two submit buttons as shown in the below code.

<form action="Action1" method=post>

<input type=&rdquo;submit&rdquo; name=&rdquo;Submit1&rdquo;/>

<input type=&rdquo;submit&rdquo; name=&rdquo;Submit2&rdquo;>

</form>

In the above code when the end user clicks on any of the submit buttons it will make a HTTP POST to “Action1”.

The question from the interviewer is:-

*“What if we have want that on “Submit1” button click it should invoke “Action1” and on the “Submit2” button click it should invoke “Action2”.”*

Now that we have understood the question let us answer the question in a detailed manner. There are two approaches to solve the above problem one is the normal HTML way and the other is the “Ajax” way.

In the HTML way we need to create two forms and place the “Submit” button inside each of the forms. And every form’s action will point to different / respective actions. You can see the below code the first form is posting to “Action1” and the second form will post to “Action2” depending on which “Submit” button is clicked.

<form action="Action1" method=post>

<input type=&rdquo;submit&rdquo; name=&rdquo;Submit1&rdquo;/>

</form>

In case the interviewer complains that the above approach is not AJAX this is where the second approach comes in. In the Ajax way we can create two different functions “Fun1” and “Fun1” , see the below code. These function will make Ajax calls by using JQUERY or any other framework. Each of these functions are binded with the “Submit” button’s “OnClick” events.

<Script language="javascript">

function Fun1()

{

$.post(&ldquo;/Action1&rdquo;,null,CallBack1);

}

function Fun2()

{

$.post(&ldquo;/Action2&rdquo;,null,CallBack2);

}

</Script>

## What is CSRF attack and how can we prevent the same in MVC?

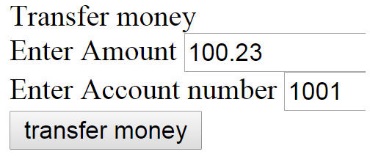
CSRF stands for Cross site request forgery. So if you see the dictonary meaning of forgery: -

“It’s an act of copying or imitating things like signature on a cheque, official documents to deceive the authority source for financial gains.”

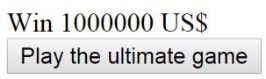
So when it comes to website this forgery is termed as CSRF (Cross Site Request Forgery).

CSRF is a method of attacking a website where the attacker imitates a.k.a forges as a trusted source and sends data to the site. Genuine site processes the information innocently thinking that data is coming from a trusted source.

For example conside the below screen of a online bank. End user’s uses this screen to transfer money.



Below is a forged site created by an attacker which looks a game site from outside, but internally it hits the bank site for money transfer.



The internal HTML of the forged site has those hidden fields which have the account number and amount to do money transfer.

<div>

Win 1000000 US$

<form action="http://localhost:23936/Genuine/Transfer" method=post>

<input type=hidden name="amount" value="10000" />

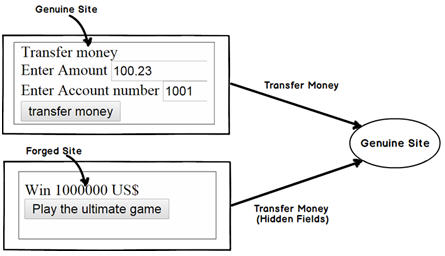
<input type=hidden name="account" value="3002" />

<input type=submit value="Play the ultimate game" />

</form>

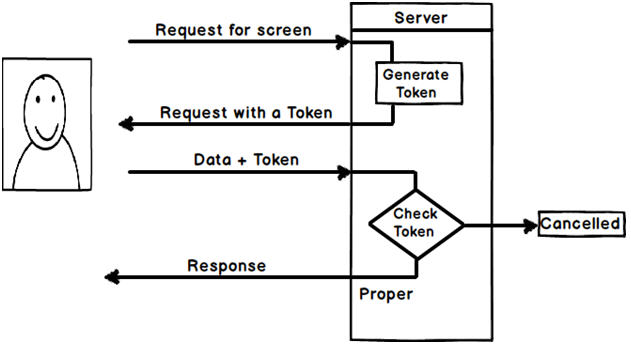
</div>

Now let’s say the user has logged in to the genuine bank site and the attacker sent this forged game link to his email. The end user thinking that it’s a game site clicks on the “Play the Ultimate Game” button and internally the malicious code does the money transfer process.



So a proper solution to this issue can be solved by using tokens: -

* End user browses to the screen of the money transfer. Before the screen is served server injects a secret token inside the HTML screen in form a hidden field.
* Now hence forth when the end user sends request back he has to always send the secret token. This token is validated on the server.



Implementing token is a two-step process in MVC: -

First apply “ValidateAntiForgeryToken” attribute on the action.

[ValidateAntiForgeryToken]

public ActionResult Transfer()

{

// password sending logic will be here

return Content(Request.Form["amount"] + " has been transferred to account "

+ Request.Form["account"]);

}

Second in the HTML UI screen call “@Html.AntiForgeryToken()” to generate the token.

<div>

Transfer money

<form action="Transfer" method=post>

Enter Amount

<input type="text" name="amount" value="" />

Enter Account number

@Html.AntiForgeryToken()

<input type=submit value="transfer money" />

</form>

</div>

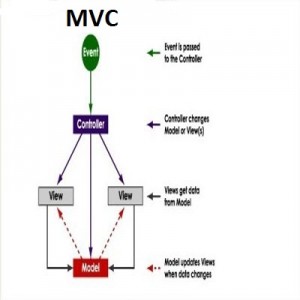
So now henceforth when any untrusted source send a request to the server it would give the below forgery error.

If you do a view source of the HTML you would find the below verification token hidden field with the secret key.

<input name="\_\_RequestVerificationToken" type="hidden" value="7iUdhsDNpEwiZFTYrH5kp/q7jL0sZz+CSBh8mb2ebwvxMJ3eYmUZXp+uofko6eiPD0fmC7Q0o4SXeGgRpxFp0i+Hx3fgVlVybgCYpyhFw5IRyYhNqi9KyH0se0hBPRu/9kYwEXXnVGB9ggdXCVPcIud/gUzjWVCvU1QxGA9dKPA=" />

## Mention what is the difference between adding routes, to a webform application and an MVC application?

To add routes to a webform application, we can use MapPageRoute() method of the RouteCollection class, where adding routes to an MVC application, you can use MapRoute() method.

[](http://career.guru99.com/wp-content/uploads/2014/12/mvc-design-diagram1.jpg)

## Mention what are the two ways to add constraints to a route?

The two methods to add constraints to a route is

* Use regular expressions
* Use an object that implements IRouteConstraint Interface

## Mention what “beforFilter()”,“beforeRender” and “afterFilter” functions do in Controller?

**beforeFilter():** This function is run before every action in the controller. It’s the right place to check for an active session or inspect user permissions.

**beforeRender():** This function is called after controller action logic, but before the view is rendered. This function is not often used, but may be required If you are calling render() manually before the end of a given action

**afterFilter():** This function is called after every controller action, and after rendering is done. It is the last controller method to run

## Explain what are the steps for the execution of an MVC project?

The steps for the execution of an MVC project includes

* Receive first request for the application
* Performs routing
* Creates MVC request handler
* Create Controller
* Execute Controller
* Invoke action
* Execute Result

## Explain what is the difference between View and Partial View?

|  |  |
| --- | --- |
| **View** | **Partial View** |
| 1. It contains the layout page 2. Before any view is rendered, viewstart page is rendered 3. View might have markup tags like body, html, head, title, meta etc. 4. View is not lightweight as compare to Partial View | 1. It does not contain the layout page 2. Partial view does not verify for a viewstart.cshtml. We cannot put common code for a partial view within the viewStart.cshtml.page 3. Partial view is designed specially to render within the view and just because of that it does not consist any mark up 4. We can pass a regular view to the RenderPartial method |

## Mention what is the importance of NonActionAttribute?

All public methods of a controller class are treated as the action method if you want to prevent this default method then you have to assign the public method with NonActionAttribute.

## Mention what is the use of the default route {resource}.axd/{\*pathinfo} ?

This default route prevents request for a web resource file such as Webresource.axd or ScriptResource.axd from being passed to the controller.

## Mention what are the file extensions for razor views?

For razor views the file extensions are

* .cshtml: If C# is the programming language
* .vbhtml: If VB is the programming language

## Mention two instances where routing is not implemented or required?

Two instance where routing is not required are

* When a physical file is found that matches the URL pattern
* When routing is disabled for a URL pattern

## What are Action Filters in MVC?

Action Filters are additional attributes that can be applied to either a controller section or the entire controller to modify the way in which action is executed. These attributes are special .NET classes derived from System.Attribute which can be attached to classes, methods, properties and fields.

*ASP.NET MVC provides the following action filters:*

* Output Cache: This action filter caches the output of a controller action for a specified amount of time.
* Handle Error: This action filter handles errors raised when a controller action executes.
* Authorize: This action filter enables you to restrict access to a particular user or role.

Now we will see the code example to apply these filters on an example controller ActionFilterDemoController. (ActionFilterDemoController is just used as an example. You can use these filters on any of your controllers.)  
  
**Output Cache  
  
E.g.:** Specifies the return value to be cached for 10 seconds.

publicclassActionFilterDemoController: Controller

{

    [HttpGet]

    OutputCache(Duration = 10)]

publicstringIndex()

{

     returnDateTime.Now.ToString("T");

}

}

## What is TempData in MVC?

TempData is a dictionary object to store data temporarily. It is a TempDataDictionary class type and instance property of the Controller base class.

TempData is able to keep data for the duration of a HTP request, in other words it can keep live data between two consecutive HTTP requests. It will help us to pass the state between action methods. TempData only works with the current and subsequent request. TempData uses a session variable to store the data. TempData Requires type casting when used to retrieve data.

TempDataDictionary is inherited from the IDictionary<string, object>, ICollection<KeyValuePair<string, object>>, IEnumerable<KeyValuePair<string, object>> and IEnumerable interfaces.  
Example

public ActionResult FirstRequest()

{

    List < string > TempDataTest = new List < string > ();

    TempDataTest.Add("Tejas");

    TempDataTest.Add("Jignesh");

    TempDataTest.Add("Rakesh");

    TempData["EmpName"] = TempDataTest;

    return View();

}

public ActionResult ConsecutiveRequest()

{

    List < string > modelData = TempData["EmpName"] as List < string > ;

    TempData.Keep();

    return View(modelData);

}

## Differences between Razor and ASPX View Engine in MVC?

Razor View Engine VS ASPX View Engine

|  |  |
| --- | --- |
| Razor View Engine | ASPX View Engine (Web form view engine) |
| The namespace used by the Razor View Engine is System.Web.Razor | The namespace used by the ASPX View Engine is System.Web.Mvc.WebFormViewEngine |
| The file extensions used by the Razor View Engine are different from a web form view engine. It uses cshtml with C# and vbhtml with vb for views, partial view, templates and layout pages. | The file extensions used by the Web Form View Engines are like ASP.Net web forms. It uses the ASPX extension to view the aspc extension for partial views or User Controls or templates and master extensions for layout/master pages. |
| The Razor View Engine is an advanced view engine that was introduced with MVC 3.0. This is not a new language but it is markup. | A web form view engine is the default view engine and available from the beginning of MVC |
| Razor has a syntax that is very compact and helps us to reduce typing. | The web form view engine has syntax that is the same as an ASP.Net forms application. |
| The Razor View Engine uses @ to render server-side content. | The ASPX/web form view engine uses "<%= %>" or "<%: %>" to render server-side content. |
| By default all text from an @ expression is HTML encoded. | There is a different syntax ("<%: %>") to make text HTML encoded. |
| Razor does not require the code block to be closed, the Razor View Engine parses itself and it is able to decide at runtime which is a content element and which is a code element. | A web form view engine requires the code block to be closed properly otherwise it throws a runtime exception. |
| The Razor View Engine prevents Cross Site Scripting (XSS) attacks by encoding the script or HTML tags before rendering to the view. | A web form View engine does not prevent Cross Site Scripting (XSS) attack. |
| The Razor Engine supports Test Driven Development (TDD). | Web Form view engine does not support Test Driven Development (TDD) because it depends on the System.Web.UI.Page class to make the testing complex. |
| Razor uses "@\* â€¦ \*@" for multiline comments. | The ASPX View Engine uses "<!--...-->" for markup and "/\* â€¦ \*/" for C# code. |
| There is only three transition characters with the Razor View Engine. | There are only three transition characters with the Razor View Engine. |

The Razor View Engine is a bit slower than the ASPX View Engine.  
  
**Conclusion**Razor provides a new view engine with streamlined code for focused templating. Razor's syntax is very compact and improves readability of the markup and code. By default MVC supports ASPX (web forms) and Razor View Engine. MVC also supports third-party view engines like Spark, Nhaml, NDjango, SharpDOM and so on. ASP.NET MVC is open source.

## What is Output Caching in MVC?

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

Keep the following in mind.

* Avoid caching contents that are accessed rarely.
* Use caching for contents that are accessed frequently.

Let's take an example. My MVC application displays a list of database records on the view page so by default each time the user invokes the controller method to see records, the application loops through the entire process and executes the database query. And this can actually decrease the application performance. So, we can advantage of the "Output Caching" that avoids executing database queries each time the user invokes the controller method. Here the view page is retrieved from the cache instead of invoking the controller method and doing redundant work.  
  
**Cached Content Locations**  
In the above paragraph I said, in Output Caching the view page is retrieved from the cache, so where is the content cached/stored?  
  
Please note, there is no guarantee that content will be cached for the amount of time that we specify. When memory resources become low, the cache starts evicting content automatically.  
  
OutputCache label has a "Location" attribute and it is fully controllable. Its default value is "Any", however there are the [following locations](http://msdn.microsoft.com/en-us/library/system.web.ui.outputcachelocation(v=vs.100).aspx) available; as of now, we can use any one.

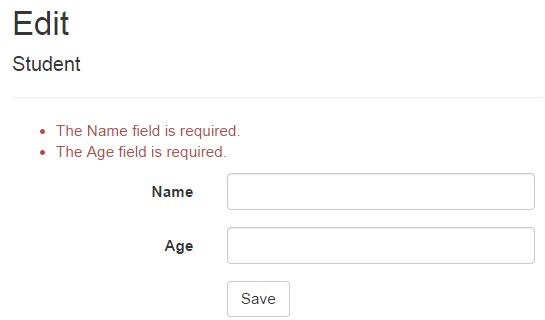
* Any
* Client
* Downstream
* Server
* None
* ServerAndClient

With "Any", the output cache is stored on the server where the request was processed. The recommended store cache is always on the server very carefully. You will learn about some security related tips in the following "Don't use Output Cache".

## What is Validation Summary in MVC?

The ValidationSummary helper method generates an unordered list (ul element) of validation messages that are in the ModelStateDictionary object.

The ValidationSummary can be used to display all the error messages for all the fields. It can also be used to display custom error messages. The following figure shows how ValidationSummary displays the error messages.



**ValidationSummary() Signature:**

MvcHtmlStringValidateMessage(bool excludePropertyErrors, string message, object htmlAttributes)

**Display field level error messages using ValidationSummary:**

By default, ValidationSummary filters out field level error messages. If you want to display field level error messages as a summary then specify excludePropertyErrors = false.

Example: ValidationSummary to display field errors:

@Html.ValidationSummary(false, "", new { @class = "text-danger" })

So now, the following Edit view will display error messages as a summary at the top. Please make sure that you don't have a ValidationMessageFor method for each of the fields.



## What is ViewStart?

Razor View Engine introduced a new layout named \_ViewStart which is applied on all view automatically. Razor View Engine firstly executes the \_ViewStart and then start rendering the other view and merges them.   
Example of Viewstart:

@ {

    Layout = "~/Views/Shared/\_v1.cshtml";

}

< !DOCTYPE html >

    < html >

    < head >

    < meta name = "viewport" content = "width=device-width" / >

    < title > ViewStart < /title> < /head> < body >

    < /body> < /html>

## What are the Difference between ViewBag&ViewData?

ViewData is a dictionary of objects that is derived from ViewDataDictionary class and accessible using strings as keys.

ViewBag is a dynamic property that takes advantage of the new dynamic features in C# 4.0.

ViewData requires typecasting for complex data type and check for null values to avoid error.

ViewBag doesn't require typecasting for complex data type.

**Calling of ViewBag is:** ViewBag.Name = "Yogesh";  
**Calling of ViewDatais :** ViewData["Name"] = "yogesh";

## How can we done Custom Error Page in MVC?

The HandleErrorAttribute allows you to use a custom page for this error. First you need to update your web.config file to allow your application to handle custom errors.

<system.web>

    <customErrors mode="On">

</system.web>

Then, your action method needs to be marked with the attribute.

[HandleError]

public class HomeController: Controller

{

    [HandleError]

    publicActionResultThrowException()

    {

        throw new ApplicationException();

    }

}

By calling the ThrowException action, this would then redirect the user to the default error page. In our case though, we want to use a custom error page and redirect the user there instead.So, let's create our new custom view page.  
  
  
  
Next, we simply need to update the HandleErrorAttribute on the action method.

[HandleError]

public class HomeController: Controller

{

    [HandleError(View = "CustomErrorView")]

    publicActionResultThrowException()

    {

        throw new ApplicationException();

    }

}

## What is Server Side Validation in MVC?

The ASP.NET MVC Framework validates any data passed to the controller action that is executing, It populates a ModelState object with any validation failures that it finds and passes that object to the controller. Then the controller actions can query the ModelState to discover whether the request is valid and react accordingly.  
  
I will use two approaches in this article to validate a model data. One is to manually add an error to the ModelState object and another uses the Data Annotation API to validate the model data.  
  
**Approach 1:** *Manually Add Error to ModelState object*I create a User class under the Models folder. The User class has two properties "Name" and "Email". The "Name" field has required field validations while the "Email" field has Email validation. So let's see the procedure to implement the validation. Create the User Model as in the following:

namespace ServerValidation.Models

{

    public class User

    {

        public string Name

       {

            get;

            set;

        }

        public string Email

        {

            get;

            set;

        }

    }

}

After that I create a controller action in User Controller (UserController.cs under Controllers folder). That action method has logic for the required validation for Name and Email validation on the Email field. I add an error message on ModelState with a key and that message will be shown on the view whenever the data is not to be validated in the model.

using System.Text.RegularExpressions;

using System.Web.Mvc;

namespace ServerValidation.Controllers

{

    public class UserController: Controller

    {

        public ActionResult Index()

        {

                return View();

            }

            [HttpPost]

        public ActionResult Index(ServerValidation.Models.User model)

        {

            if (string.IsNullOrEmpty(model.Name))

            {

                ModelState.AddModelError("Name", "Name is required");

            }

            if (!string.IsNullOrEmpty(model.Email))

            {

                string emailRegex = @ "^([a-zA-Z0-9\_\-\.]+)@((\[[0-9]{1,3}" +

                    @ "\.[0-9]{1,3}\.[0-9]{1,3}\.)|(([a-zA-Z0-9\-]+\" +

                @ ".)+))([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$";

                Regex re = new Regex(emailRegex);

                if (!re.IsMatch(model.Email))

                {

                    ModelState.AddModelError("Email", "Email is not valid");

                }

            } else {

                ModelState.AddModelError("Email", "Email is required");

            }

            if (ModelState.IsValid)

            {

                ViewBag.Name = model.Name;

                ViewBag.Email = model.Email;

            }

            return View(model);

        }

    }

}

Thereafter I create a view (Index.cshtml) for the user input under the User folder.

@model ServerValidation.Models.User

@ {

    ViewBag.Title = "Index";

}

@using(Html.BeginForm())

{

    if (@ViewData.ModelState.IsValid)

    {

        if (@ViewBag.Name != null)

     {  <b>

                Name: @ViewBag.Name <br/>

                Email: @ViewBag.Email

</b>

        }

    }

<fieldset>

        <legend> User </legend>

<div class = "editor-label">

@Html.LabelFor(model => model.Name)   
 </div>

<div class = "editor-field">

         @Html.EditorFor(model => model.Name)

     @if(!ViewData.ModelState.IsValid)

{

<span class = "field-validation-error">

@ViewData.ModelState["Name"].Errors[0].ErrorMessage

</span>

}

</div>

<div class = "editor-label">

        @Html.LabelFor(model => model.Email)

</div>

<div class = "editor-field">

        @Html.EditorFor(model => model.Email)

  @if(!ViewData.ModelState.IsValid)

  {

<span class = "field-validation-error">

@ViewData.ModelState["Email"].Errors[0].ErrorMessage

</span>

  }

  </div>

<p> <input type = "submit" value = "Create"/> </p>

< /fieldset>

}

## What is the use of remote validation in MVC?

Remote validation is the process where we validate specific data posting data to a server without posting the entire form data to the server. Let's see an actual scenario, in one of my projects I had a requirement to validate an email address, whether it already exists in the database. Remote validation was useful for that; without posting all the data we can validate only the email address supplied by the user.  
  
**Practical Explanation**Let's create a MVC project and name it accordingly, for me its “TestingRemoteValidation”. Once the project is created let's create a model named UserModel that will look like:

public class UserModel

{

    [Required]

    public string UserName

    {

        get;

        set;

    }

    [Remote("CheckExistingEmail", "Home", ErrorMessage = "Email already exists!")]

    public string UserEmailAddress

    {

        get;

        set;

    }

}

Let's get some understanding of the remote attribute used, so the very first parameter “CheckExistingEmail” is the the name of the action. The second parameter “Home” is referred to as controller so to validate the input for the UserEmailAddress the “CheckExistingEmail” action of the “Home” controller is called and the third parameter is the error message. Let's implement the “CheckExistingEmail” action result in our home controller.

public ActionResult CheckExistingEmail(string UserEmailAddress)

{

    bool ifEmailExist = false;

    try

    {

        ifEmailExist= UserEmailAddress.Equals("mukeshknayak@gmail.com")?true:false;

        return Json(!ifEmailExist, JsonRequestBehavior.AllowGet);

    }

catch (Exception ex)  {  return Json(false, JsonRequestBehavior.AllowGet);  }

}

## Explain RenderSection in MVC?

RenderSection() is a method of the WebPageBase class. Scott wrote at one point, The first parameter to the "RenderSection()" helper method specifies the name of the section we want to render at that location in the layout template. The second parameter is optional, and allows us to define whether the section we are rendering is required or not. If a section is "required", then Razor will throw an error at runtime if that section is not implemented within a view template that is based on the layout file (that can make it easier to track down content errors). It returns the HTML content to render.

<div id="body">

    @RenderSection("featured", required: false)

    <section class="content-wrapper main-content clear-fix">

        @RenderBody()

    </section>

</div>

## What is GET and POST Actions Types?

GET is used to request data from a specified resource. With all the GET request we pass the URL which is compulsory, however it can take the following overloads.  
.get(url [, data ] [, success(data, textStatus, jqXHR) ] [, dataType ] ).done/.fail  
  
POST is used to submit data to be processed to a specified resource. With all the POST requests we pass the URL which is compulsory and the data, however it can take the following overloads.   
.post(url [, data ] [, success(data, textStatus, jqXHR) ] [, dataType ] )

## What is the difference between Html.Partial and Html.RenderPartial in MVC?

**Html**.**Partial** returns a String. **Html**.**RenderPartial** calls Write internally and returns void . In the snippet above, both calls will yield the same result. While one can store the output of **Html**.**Partial** in a variable or return it from a method, one cannot do this with **Html**.**RenderPartial**. The result will be written to the Response stream during execution/evaluation.

// Razor syntax

@Html.Partial("ViewName")

@{ Html.RenderPartial("ViewName"); }

// WebView syntax

<%: Html.Partial("ViewName") %>

<% Html.RenderPartial("ViewName"); %>

## What is the difference between Html.Action and Html.RenderAction in MVC?

## Explain Parse() and TryParse() in MVC?

To answer this in a sort way, the int.**Parse**() **and** int.TryPrase() methods is used to convert a string representation of number to an integer. In case of the string can't be converted the int.**Parse**() throws an exceptions where as int.**TryParse**() return a bool value, false.

## How Convert.ToInt32 is different from Parse() and TryParse() functions?

Both **int**.**Parse** and **Convert**.**ToInt32** are used to **convert** string into the integer but Only **difference between** them is to **Convert**.**ToInt32** handle null and returns '0' as output and **int**.**parse** is not going to handle NULL and will give a Argument Null Exception.

Simply, int.Parse (string s) method converts the string to integer. If string s is null, then it will throw **ArgumentNullException**. If string s is other than integer value, then it will throw **FormatException**. If string s represents out of integer ranges, then it will throw **OverflowException**.

Simply, Convert.ToInt32(string s) method converts the string to integer. If string s is null, then it will return **0** rather than throw **ArgumentNullException**. If string s is other than integer value, then it will throw **FormatException**. If string s represents out of integer ranges, then it will throw **OverflowException**.

Simply int.TryParse(string s,out int) method converts the string to integer out variable and returns true if successfully parsed, otherwise false. If string s is null, then the out variable has 0 rather than throw **ArgumentNullException**. If string s is other than integer value, then the out variable will have 0rather than **FormatException**. If string s represents out of integer ranges, then the out variable will have 0rather than throw **OverflowException**.

## Does Asp .Net MVC has Page life cycle?

In my initial days of learning MVC, I was curious about it's life cycle. But I found it a bit confusing. So I have documented my understandings. I hope this may help some one. This is a pure conceptual thing. To understand this, one needs to have a solid understanding of OOP concepts (especially about classes, interfaces, abstraction, inheritances and so on).   
  
**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.

1. 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. }

1. 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. }
2. 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. }

1. 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). |

1. [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.

## If you want to render two models in your view, what approach you would take to achieve it?

Viemodels i.e. MVVM

## What is calling asynchronously means in asp .net MVC? How can you call a method in mvc asynchronously? Suppose you have triggered some event in your application and started working on some other tasks within in application. Now you want to be notified once the triggered event completes. How can you achieve that?

Let's first consider a standard synchronous action:

public ActionResult Index()

{

// some processing

return View();

}

When a request is made to this action a thread is drawn from the thread pool and the body of this action is executed on this thread. So if the processing inside this action is slow you are blocking this thread for the entire processing, so this thread cannot be reused to process other requests. At the end of the request execution, the thread is returned to the thread pool.

Now let's take an example of the asynchronous pattern:

public void IndexAsync()

{

// perform some processing

}

public ActionResult IndexCompleted(object result)

{

return View();

}

When a request is sent to the Index action, a thread is drawn from the thread pool and the body of the IndexAsync method is executed. Once the body of this method finishes executing, the thread is returned to the thread pool. Then, using the standard AsyncManager.OutstandingOperations, once you signal the completion of the async operation, another thread is drawn from the thread pool and the body of the IndexCompleted action is executed on it and the result rendered to the client.

So what we can see in this pattern is that a single client HTTP request could be executed by two different threads.

Now the interesting part happens inside the IndexAsync method. If you have a blocking operation inside it, you are totally wasting the whole purpose of the asynchronous controllers because you are blocking the worker thread (remember that the body of this action is executed on a thread drawn from the thread pool).

So when can we take real advantage of asynchronous controllers you might ask?

IMHO we can gain most when we have I/O intensive operations (such as database and network calls to remote services). If you have a CPU intensive operation, asynchronous actions won't bring you much benefit.

So why can we gain benefit from I/O intensive operations? Because we could use [I/O Completion Ports](http://msdn.microsoft.com/en-us/library/windows/desktop/aa365198%28v=vs.85%29.aspx). IOCP are extremely powerful because you do not consume any threads or resources on the server during the execution of the entire operation.

How do they work?

Suppose that we want to download the contents of a remote web page using the [WebClient.DownloadStringAsync](http://msdn.microsoft.com/en-us/library/system.net.webclient.downloadstringasync.aspx) method. You call this method which will register an IOCP within the operating system and return immediately. During the processing of the entire request, no threads are consumed on your server. Everything happens on the remote server. This could take lots of time but you don't care as you are not jeopardizing your worker threads. Once a response is received the IOCP is signaled, a thread is drawn from the thread pool and the callback is executed on this thread. But as you can see, during the entire process, we have not monopolized any threads.

The same stands true with methods such as FileStream.BeginRead, SqlCommand.BeginExecute, ...

What about parallelizing multiple database calls? Suppose that you had a synchronous controller action in which you performed 4 blocking database calls in sequence. It's easy to calculate that if each database call takes 200ms, your controller action will take roughly 800ms to execute.

If you don't need to run those calls sequentially, would parallelizing them improve performance?

That's the big question, which is not easy to answer. Maybe yes, maybe no. It will entirely depend on how you implement those database calls. If you use async controllers and I/O Completion Ports as discussed previously you will boost the performance of this controller action and of other actions as well, as you won't be monopolizing worker threads.

On the other hand if you implement them poorly (with a blocking database call performed on a thread from the thread pool), you will basically lower the total time of execution of this action to roughly 200ms but you would have consumed 4 worker threads so you might have degraded the performance of other requests which might become starving because of missing threads in the pool to process them.

So it is very difficult and if you don't feel ready to perform extensive tests on your application, do not implement asynchronous controllers, as chances are that you will do more damage than benefit. Implement them only if you have a reason to do so: for example you have identified that standard synchronous controller actions are a bottleneck to your application (after performing extensive load tests and measurements of course).

Now let's consider your example:

public ViewResult Index() {

Task.Factory.StartNew(() => {

//Do an advanced looging here which takes a while

});

return View();

}

When a request is received for the Index action a thread is drawn from the thread pool to execute its body, but its body only schedules a new task using [TPL](https://msdn.microsoft.com/en-us/library/dd537609(v=vs.110).aspx). So the action execution ends and the thread is returned to the thread pool. Except that, [TPL](https://msdn.microsoft.com/en-us/library/dd537609(v=vs.110).aspx) uses threads from the thread pool to perform their processing. So even if the original thread was returned to the thread pool, you have drawn another thread from this pool to execute the body of the task. So you have jeopardized 2 threads from your precious pool.

Now let's consider the following:

public ViewResult Index() {

new Thread(() => {

//Do an advanced looging here which takes a while

}).Start();

return View();

}

In this case we are manually spawning a thread. In this case the execution of the body of the Index action might take slightly longer (because spawning a new thread is more expensive than drawing one from an existing pool). But the execution of the advanced logging operation will be done on a thread which is not part of the pool. So we are not jeopardizing threads from the pool which remain free for serving another requests.

Yes - all threads come from the thread-pool. Your MVC app is already multi-threaded, when a request comes in a new thread will be taken from the pool and used to service the request. That thread will be 'locked' (from other requests) until the request is fully serviced and completed. If there is no thread available in the pool the request will have to wait until one is available.

If you have async controllers they still get a thread from the pool but while servicing the request they can give up the thread, while waiting for something to happen (and that thread can be given to another request) and when the original request needs a thread again it gets one from the pool.

The difference is that if you have a lot of long-running requests (where the thread is waiting for a response from something) you might run out of threads from the the pool to service even basic requests. If you have async controllers, you don't have any more threads but those threads that are waiting are returned to the pool and can service other requests.

A nearly real life example... Think of it like getting on a bus, there's five people waiting to get on, the first gets on, pays and sits down (the driver serviced their request), you get on (the driver is servicing your request) but you can't find your money; as you fumble in your pockets the driver gives up on you and gets the next two people on (servicing their requests), when you find your money the driver starts dealing with you again (completing your request) - the fifth person has to wait until you are done but the third and fourth people got served while you were half way through getting served. This means that the driver is the one and only thread from the pool and the passengers are the requests. It was too complicated to write how it would work if there was two drivers but you can imagine...

Without an async controller, the passengers behind you would have to wait ages while you looked for your money, meanwhile the bus driver would be doing no work.

So the conclusion is, if lots of people don't know where their money is (i.e. require a long time to respond to something the driver has asked) async controllers could well help throughput of requests, speeding up the process from some. Without an aysnc controller everyone waits until the person in front has been completely dealt with. BUT don't forget that in MVC you have a lot of bus drivers on a single bus so async is not an automatic choice.

## How can you achieve role based login authentication in asp .net mvc?

## On selecting value I country dropdown, state dropdown should get populated and similarly cities. What are the different ways to achieve that?

Controller

The Controller consists of three Action methods.

Action method for handling GET operation

Inside this Action method, the Countries DropDownList is populated using the PopulateDropDown method which executes the SQL query passed as parameter and returns a Generic List of SelectListItem class.

The list of Countries is assigned to the Countries property of the Model class which is ultimately returned to the View.

Action method for handling AJAX operation

This Action method is executed when the jQuery AJAX call is made from View on changed event of the Country and State DropDownLists.

It accepts type and value parameter and based on the type i.e. Country or State, the list of States or Cities respectively are fetched from the database and returned to View as JSON.

Action method for handling POST operation

This Action method handles the call made from the POST function from the View and is executed when the Submit button is clicked.

When the Form is submitted, the posted values are captured in three variables one for each i.e. Country, State and City.

Using these values, the Countries, States and Cities are again populated in the Model class object which is then returned back to the View.

One option is, you can load all of the countries, state/province and cities in page (memory) and filter them using JQuery/JavaScript. You may need to use very complex query to retrieve data for dependent dropdown list. Or you may have to retrieve huge amount of data. For those type of scenarios, loading all data in memory is not a good solution.

Using MVC’s Action and JQuery’s Load methods, you can retrieve data for dependent dropdown list (State/Province) as user selects the parent (Country). In this walk through, you will learn how to use the Action and Load methods to re-load a part of your page asynchronously.

## What are the different types of authentication in MVC?

**1.Windows authentication**   
2.**Forms authentication**   
3.**Passport authentication**   
4.**Anonymous access**   
  
1.**Windows authentication**   
If your application is targeted for use inside an organization, and users accessing the application have existing user accounts within the local user database of the Web server or Active Directory, you should authenticate users with Windows authentication.   
  
2.**Forms authentication**   
By default, Form authentication is used.   
Form-based authentication presents the user with an HTML-based Web page that prompts the user for credentials.   
  
3.**Passport authentication**   
You can also authenticate users using a service from Microsoft called Passport. Passport   
is a centralized directory of user information that Web sites can use, in exchange for a fee, to authenticate users. Users can choose to allow the Web site access to personal   
information stored on Passport, such as the users' addresses, ages, and interests.   
  
4.**Anonymous access**   
You can explicitly disable authentication for your application if you know that it will be used only by anonymous users.   
<configuration>   
<system.web>   
<authentication mode="None" />   
</system.web>   
</configuration>

[Windows Authentication](https://www.codeproject.com/Articles/98950/ASP-NET-authentication-and-authorization#Windows Authentication)

[5 steps to enable authentication and authorization using Windows](https://www.codeproject.com/Articles/98950/ASP-NET-authentication-and-authorization#5 steps to enable authentication and authorization using Windows)

[Different methods of collecting username and password](https://www.codeproject.com/Articles/98950/ASP-NET-authentication-and-authorization#Different methods of collecting username and password)

[Basic Authentication](https://www.codeproject.com/Articles/98950/ASP-NET-authentication-and-authorization#Basic Authentication)

[Base64 is an encoding mechanism and not encryption](https://www.codeproject.com/Articles/98950/ASP-NET-authentication-and-authorization#Base64 is an encoding mechanism and not encryption)

[Digest Authentication](https://www.codeproject.com/Articles/98950/ASP-NET-authentication-and-authorization#Digest Authentication)

[Integrated Authentication](https://www.codeproject.com/Articles/98950/ASP-NET-authentication-and-authorization#Integrated Authentication)

[Order of Precedence](https://www.codeproject.com/Articles/98950/ASP-NET-authentication-and-authorization#Order of Precedence)

[Comparison of Basic, digest and windows authentication](https://www.codeproject.com/Articles/98950/ASP-NET-authentication-and-authorization#Comparison of Basic, digest and windows authentication)

[Forms Authentication](https://www.codeproject.com/Articles/98950/ASP-NET-authentication-and-authorization#Forms Authentication)

[Forms authentication using ‘web.config’ as a data store](https://www.codeproject.com/Articles/98950/ASP-NET-authentication-and-authorization#Forms authentication using %E2%80%98web.config%E2%80%99 as a data store)

[Forms Authentication using SQL server as a data store](https://www.codeproject.com/Articles/98950/ASP-NET-authentication-and-authorization#Forms Authentication using SQL server as a data store)

[Forms authentication using ASP.NET Membership and role](https://www.codeproject.com/Articles/98950/ASP-NET-authentication-and-authorization#Forms authentication using ASP.NET Membership and role)

[The dual combination](https://www.codeproject.com/Articles/98950/ASP-NET-authentication-and-authorization#The dual combination)

[Forms Authentication using Single Sign on](https://www.codeproject.com/Articles/98950/ASP-NET-authentication-and-authorization#Forms Authentication using Single Sign on)

[Passport Authentication](https://www.codeproject.com/Articles/98950/ASP-NET-authentication-and-authorization#Passport Authentication)

## What are asynchronous pages in asp .Net?

ASP.NET 2.0 is replete with new features ranging from declarative data binding and Master Pages to membership and role management services. But my vote for the coolest new feature goes to asynchronous pages, and here's why.

When ASP.NET receives a request for a page, it grabs a thread from a thread pool and assigns that request to the thread. A normal, or synchronous, page holds onto the thread for the duration of the request, preventing the thread from being used to process other requests. If a synchronous request becomes I/O bound—for example, if it calls out to a remote Web service or queries a remote database and waits for the call to come back—then the thread assigned to the request is stuck doing nothing until the call returns. That impedes scalability because the thread pool has a finite number of threads available. If all request-processing threads are blocked waiting for I/O operations to complete, additional requests get queued up waiting for threads to be free. At best, throughput decreases because requests wait longer to be processed. At worst, the queue fills up and ASP.NET fails subsequent requests with 503 "Server Unavailable" errors.

Asynchronous pages offer a neat solution to the problems caused by I/O-bound requests. Page processing begins on a thread-pool thread, but that thread is returned to the thread pool once an asynchronous I/O operation begins in response to a signal from ASP.NET. When the operation completes, ASP.NET grabs another thread from the thread pool and finishes processing the request. Scalability increases because thread-pool threads are used more efficiently. Threads that would otherwise be stuck waiting for I/O to complete can now be used to service other requests. The direct beneficiaries are requests that don't perform lengthy I/O operations and can therefore get in and out of the pipeline quickly. Long waits to get into the pipeline have a disproportionately negative impact on the performance of such requests.

The ASP.NET 2.0 Beta 2 async page infrastructure suffers from scant documentation. Let's fix that by surveying the landscape of async pages. Keep in mind that this column was developed with beta releases of ASP.NET 2.0 and the .NET Framework 2.0.

**Asynchronous Pages in ASP.NET 1.x**

ASP.NET 1.x doesn't support asynchronous pages per se, but it's possible to build them with a pinch of tenacity and a dash of ingenuity. For an excellent overview, see Fritz Onion's article entitled "[Use Threads and Build Asynchronous Handlers in Your Server-Side Web Code](https://web.archive.org/web/20150515023107/http:/msdn.microsoft.com/msdnmag/issues/03/06/Threading/)" in the June 2003 issue of MSDN®Magazine.

The trick here is to implement IHttpAsyncHandler in a page's codebehind class, prompting ASP.NET to process requests not by calling the page's IHttpHandler.ProcessRequest method, but by calling IHttpAsyncHandler.BeginProcessRequest instead. Your BeginProcessRequest implementation can then launch another thread. That thread calls base.ProcessRequest, causing the page to undergo its normal request-processing lifecycle (complete with events such as Load and Render) but on a non-threadpool thread. Meanwhile, BeginProcessRequest returns immediately after launching the new thread, allowing the thread that's executing BeginProcessRequest to return to the thread pool.

That's the basic idea, but the devil's in the details. Among other things, you need to implement IAsyncResult and return it from BeginProcessRequest. That typically means creating a ManualResetEvent object and signaling it when ProcessRequest returns in the background thread. In addition, you have to provide the thread that calls base.ProcessRequest. Unfortunately, most of the conventional techniques for moving work to background threads, including Thread.Start, ThreadPool.QueueUserWorkItem, and asynchronous delegates, are counterproductive in ASP.NET applications because they either steal threads from the thread pool or risk unconstrained thread growth. A proper asynchronous page implementation uses a custom thread pool, and custom thread pool classes are not trivial to write (for more information, see the [.NET Matters](https://web.archive.org/web/20150515023107/http:/msdn.microsoft.com/msdnmag/issues/05/02/NETMatters/)column in the February 2005 issue of MSDN Magazine).

The bottom line is that building async pages in ASP.NET 1.x isn't impossible, but it is tedious. And after doing it once or twice, you can't help but think that there has to be a better way. Today there is—ASP.NET 2.0.

**Asynchronous Pages in ASP.NET 2.0**

ASP.NET 2.0 vastly simplifies the way you build asynchronous pages. You begin by including an Async="true" attribute in the page's @ Page directive, like so:

<%@ Page Async="true" ... %>

Under the hood, this tells ASP.NET to implement IHttpAsyncHandler in the page. Next, you call the new Page.AddOnPreRenderCompleteAsync method early in the page's lifetime (for example, in Page\_Load) to register a Begin method and an End method, as shown in the following code:

AddOnPreRenderCompleteAsync (

new BeginEventHandler(MyBeginMethod),

new EndEventHandler (MyEndMethod)

);

What happens next is the interesting part. The page undergoes its normal processing lifecycle until shortly after the PreRender event fires. Then ASP.NET calls the Begin method that you registered using AddOnPreRenderCompleteAsync. The job of the Begin method is to launch an asynchronous operation such as a database query or Web service call and return immediately. At that point, the thread assigned to the request goes back to the thread pool. Furthermore, the Begin method returns an IAsyncResult that lets ASP.NET determine when the asynchronous operation has completed, at which point ASP.NET extracts a thread from the thread pool and calls your End method. After End returns, ASP.NET executes the remaining portion of the page's lifecycle, which includes the rendering phase. Between the time Begin returns and End gets called, the request-processing thread is free to service other requests, and until End is called, rendering is delayed. And because version 2.0 of the .NET Framework offers a variety of ways to perform asynchronous operations, you frequently don't even have to implement IAsyncResult. Instead, the Framework implements it for you.

The codebehind class in **Figure 1** provides an example. The corresponding page contains a Label control whose ID is "Output". The page uses the System.Net.HttpWebRequest class to fetch the contents of http://msdn.microsoft.com. Then it parses the returned HTML and writes out to the Label control a list of all the HREF targets it finds.

**Figure 1 AsyncPage.aspx.cs**

using System;

using System.Web;

using System.Web.UI;

using System.Web.UI.WebControls;

using System.Net;

using System.IO;

using System.Text;

using System.Text.RegularExpressions;

public partial class AsyncPage : System.Web.UI.Page

{

private WebRequest \_request;

void Page\_Load (object sender, EventArgs e)

{

AddOnPreRenderCompleteAsync (

new BeginEventHandler(BeginAsyncOperation),

new EndEventHandler (EndAsyncOperation)

);

}

IAsyncResult BeginAsyncOperation (object sender, EventArgs e,

AsyncCallback cb, object state)

{

\_request = WebRequest.Create("http://msdn.microsoft.com");

return \_request.BeginGetResponse (cb, state);

}

void EndAsyncOperation (IAsyncResult ar)

{

string text;

using (WebResponse response = \_request.EndGetResponse(ar))

{

using (StreamReader reader =

new StreamReader(response.GetResponseStream()))

{

text = reader.ReadToEnd();

}

}

Regex regex = new Regex ("href\\s\*=\\s\*\"([^\"]\*)\"",

RegexOptions.IgnoreCase);

MatchCollection matches = regex.Matches(text);

StringBuilder builder = new StringBuilder(1024);

foreach (Match match in matches)

{

builder.Append (match.Groups[1]);

builder.Append("<br/>");

}

Output.Text = builder.ToString ();

}

}

Since an HTTP request can take a long time to return, AsyncPage.aspx.cs performs its processing asynchronously. It registers Begin and End methods in Page\_Load, and in the Begin method, it calls HttpWebRequest.BeginGetResponse to launch an asynchronous HTTP request. BeginAsyncOperation returns to ASP.NET the IAsyncResult returned by BeginGetResponse, resulting in ASP.NET calling EndAsyncOperation when the HTTP request completes. EndAsyncOperation, in turn, parses the content and writes the results to the Label control, after which rendering occurs and an HTTP response goes back to the browser.

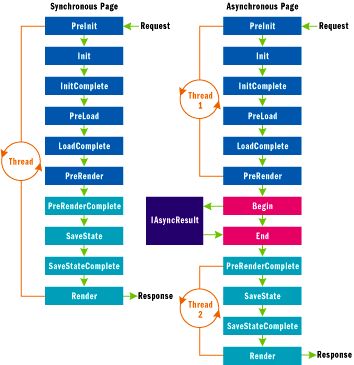


Figure 2**Synchronous vs. Asynchronous Page Processing**

**Figure 2** illustrates the difference between a synchronous page and an asynchronous page in ASP.NET 2.0. When a synchronous page is requested, ASP.NET assigns the request a thread from the thread pool and executes the page on that thread. If the request pauses to perform an I/O operation, the thread is tied up until the operation completes and the page lifecycle can be completed. An asychronous page, by contrast, executes as normal through the PreRender event. Then the Begin method that's registered using AddOnPreRenderCompleteAsync is called, after which the request-processing thread goes back to the thread pool. Begin launches an asynchronous I/O operation, and when the operation completes, ASP.NET grabs another thread from the thread pool and calls the End method and executes the remainder of the page's lifecycle on that thread.

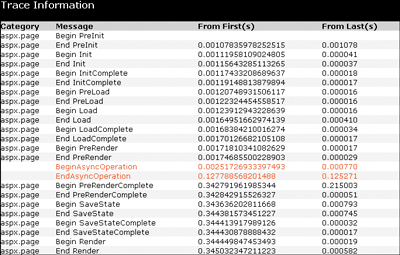


Figure 3**Trace Output Shows Async Page's Async Point**

The call to Begin marks the page's "async point." The trace in **Figure 3** shows exactly where the async point occurs. If called, AddOnPreRenderCompleteAsync must be called before the async point—that is, no later than the page's PreRender event.

**Asynchronous Data Binding**

It's not all that common for ASP.NET pages to use HttpWebRequest directly to request other pages, but it is common for them to query databases and data bind the results. So how would you use asynchronous pages to perform asynchronous data binding? The codebehind class in **Figure 4** shows one way to go about it.

**Figure 4 AsyncDataBind.aspx.cs**

using System;

using System.Data;

using System.Data.SqlClient;

using System.Web;

using System.Web.UI;

using System.Web.UI.WebControls;

using System.Web.Configuration;

public partial class AsyncDataBind : System.Web.UI.Page

{

private SqlConnection \_connection;

private SqlCommand \_command;

private SqlDataReader \_reader;

protected void Page\_Load(object sender, EventArgs e)

{

if (!IsPostBack)

{

// Hook PreRenderComplete event for data binding

this.PreRenderComplete +=

new EventHandler(Page\_PreRenderComplete);

// Register async methods

AddOnPreRenderCompleteAsync(

new BeginEventHandler(BeginAsyncOperation),

new EndEventHandler(EndAsyncOperation)

);

}

}

IAsyncResult BeginAsyncOperation (object sender, EventArgs e,

AsyncCallback cb, object state)

{

string connect = WebConfigurationManager.ConnectionStrings

["PubsConnectionString"].ConnectionString;

\_connection = new SqlConnection(connect);

\_connection.Open();

\_command = new SqlCommand(

"SELECT title\_id, title, price FROM titles", \_connection);

return \_command.BeginExecuteReader (cb, state);

}

void EndAsyncOperation(IAsyncResult ar)

{

\_reader = \_command.EndExecuteReader(ar);

}

protected void Page\_PreRenderComplete(object sender, EventArgs e)

{

Output.DataSource = \_reader;

Output.DataBind();

}

public override void Dispose()

{

if (\_connection != null) \_connection.Close();

base.Dispose();

}

}

AsyncDataBind.aspx.cs uses the same AddOnPreRenderCompleteAsync pattern that AsyncPage.aspx.cs uses. But rather than call HttpWebRequest.BeginGetResponse, its BeginAsyncOperation method calls SqlCommand.BeginExecuteReader (new in ADO.NET 2.0), to perform an asynchronous database query. When the call completes, EndAsyncOperation calls SqlCommand.EndExecuteReader to get a SqlDataReader, which it then stores in a private field. In an event handler for the PreRenderComplete event, which fires after the asynchronous operation completes but before the page is rendered, it then binds the SqlDataReader to the Output GridView control. On the outside, the page looks like a normal (synchronous) page that uses a GridView to render the results of a database query. But on the inside, this page is much more scalable because it doesn't tie up a thread-pool thread waiting for the query to return.

**Calling Web Services Asynchronously**

Another I/O-related task commonly performed by ASP.NET Web pages is callouts to Web services. Since Web service calls can take a long time to return, pages that execute them are ideal candidates for asynchronous processing.

**Figure 5** shows one way to build an asynchronous page that calls out to a Web service. It uses the same AddOnPreRenderCompleteAsync mechanism featured in **Figure 1** and **Figure 4**. The page's Begin method launches an asynchronous Web service call by calling the Web service proxy's asynchronous Begin method. The page's End method caches in a private field a reference to the DataSet returned by the Web method, and the PreRenderComplete handler binds the DataSet to a GridView. For reference, the Web method targeted by the call is shown in the following code:

[WebMethod]

public DataSet GetTitles ()

{

string connect = WebConfigurationManager.ConnectionStrings

["PubsConnectionString"].ConnectionString;

SqlDataAdapter adapter = new SqlDataAdapter

("SELECT title\_id, title, price FROM titles", connect);

DataSet ds = new DataSet();

adapter.Fill(ds);

return ds;

}

**Figure 5 AsyncWSInvoke1.aspx.cs**

using System;

using System.Data;

using System.Configuration;

using System.Web;

using System.Web.UI;

using System.Web.UI.WebControls;

public partial class AsyncWSInvoke1 : System.Web.UI.Page

{

private WS.PubsWebService \_ws;

private DataSet \_ds;

protected void Page\_Load(object sender, EventArgs e)

{

if (!IsPostBack)

{

// Hook PreRenderComplete event for data binding

this.PreRenderComplete +=

new EventHandler(Page\_PreRenderComplete);

// Register async methods

AddOnPreRenderCompleteAsync(

new BeginEventHandler(BeginAsyncOperation),

new EndEventHandler(EndAsyncOperation)

);

}

}

IAsyncResult BeginAsyncOperation (object sender, EventArgs e,

AsyncCallback cb, object state)

{

\_ws = new WS.PubsWebService();

// Fix up URL for call to local VWD-hosted Web service

\_ws.Url = new Uri(Request.Url, "Pubs.asmx").ToString();

\_ws.UseDefaultCredentials = true;

return \_ws.BeginGetTitles (cb, state);

}

void EndAsyncOperation(IAsyncResult ar)

{

\_ds = \_ws.EndGetTitles(ar);

}

protected void Page\_PreRenderComplete(object sender, EventArgs e)

{

Output.DataSource = \_ds;

Output.DataBind();

}

public override void Dispose()

{

if (\_ws != null) \_ws.Dispose();

base.Dispose();

}

}

That's one way to do it, but it's not the only way. The .NET Framework 2.0 Web service proxies support two mechanisms for placing asynchronous calls to Web services. One is the per-method Begin and End methods featured in .NET Framework 1.x. and 2.0 Web service proxies. The other is the new MethodAsync methods and MethodCompleted events found only in the Web service proxies of the .NET Framework 2.0.

If a Web service has a method named Foo, then in addition to having methods named Foo, BeginFoo, and EndFoo, a .NET Framework version 2.0 Web service proxy includes a method named FooAsync and an event named FooCompleted. You can call Foo asynchronously by registering a handler for FooCompleted events and calling FooAsync, like this:

proxy.FooCompleted += new FooCompletedEventHandler (OnFooCompleted);

proxy.FooAsync (...);

...

void OnFooCompleted (Object source, FooCompletedEventArgs e)

{

// Called when Foo completes

}

When the asynchronous call begun by FooAsync completes, a FooCompleted event fires, causing your FooCompleted event handler to be called. Both the delegate wrapping the event handler (FooCompletedEventHandler) and the second parameter passed to it (FooCompletedEventArgs) are generated along with the Web service proxy. You can access Foo's return value through FooCompletedEventArgs.Result.

**Figure 6** presents a codebehind class that calls a Web service's GetTitles method asynchronously using the MethodAsync pattern. Functionally, this page is identical to the one in **Figure 5**. Internally, it's quite different. AsyncWSInvoke2.aspx includes an @ Page Async="true" directive, just like AsyncWSInvoke1.aspx. But AsyncWSInvoke2.aspx.cs doesn't call AddOnPreRenderCompleteAsync; it registers a handler for GetTitlesCompleted events and calls GetTitlesAsync on the Web service proxy. ASP.NET still delays rendering the page until GetTitlesAsync completes. Under the hood, it uses an instance of System.Threading.SynchronizationContext, another new class in 2.0, to receive notifications when the asynchronous call begins and when it completes.

**Figure 6 AsyncWSInvoke2.aspx.cs**

using System;

using System.Data;

using System.Configuration;

using System.Web;

using System.Web.UI;

using System.Web.UI.WebControls;

public partial class AsyncWSInvoke2 : System.Web.UI.Page

{

private WS.PubsWebService \_ws;

private DataSet \_ds;

protected void Page\_Load(object sender, EventArgs e)

{

if (!IsPostBack)

{

// Hook PreRenderComplete event for data binding

this.PreRenderComplete +=

new EventHandler(Page\_PreRenderComplete);

// Call the Web service asynchronously

\_ws = new WS.PubsWebService();

\_ws.GetTitlesCompleted += new

WS.GetTitlesCompletedEventHandler(GetTitlesCompleted);

\_ws.Url = new Uri(Request.Url, "Pubs.asmx").ToString();

\_ws.UseDefaultCredentials = true;

\_ws.GetTitlesAsync();

}

}

void GetTitlesCompleted(Object source,

WS.GetTitlesCompletedEventArgs e)

{

\_ds = e.Result;

}

protected void Page\_PreRenderComplete(object sender, EventArgs e)

{

Output.DataSource = \_ds;

Output.DataBind();

}

public override void Dispose()

{

if (\_ws != null) \_ws.Dispose();

base.Dispose();

}

}

There are two advantages to using MethodAsync rather than AddOnPreRenderCompleteAsync to implement asynchronous pages. First, MethodAsync flows impersonation, culture, and HttpContext.Current to the MethodCompleted event handler. AddOnPreRenderCompleteAsync does not. Second, if the page makes multiple asynchronous calls and must delay rendering until all the calls have been completed, using AddOnPreRenderCompleteAsync requires you to compose an IAsyncResult that remains unsignaled until all the calls have completed. With MethodAsync, no such hijinks are necessary; you simply place the calls, as many of them as you like, and the ASP.NET engine delays the rendering phase until the final call returns.

**Asynchronous Tasks**

MethodAsync is a convenient way to make multiple asynchronous Web service calls from an asynchronous page and delay the rendering phase until all the calls complete. But what if you want to perform several asynchronous I/O operations in an asynchronous page and those operations don't involve Web services? Does that mean you're back to composing an IAsyncResult that you can return to ASP.NET to let it know when the last call has completed? Fortunately, no.

In ASP.NET 2.0, the System.Web.UI.Page class introduces another method to facilitate asynchronous operations: RegisterAsyncTask. RegisterAsyncTask has four advantages over AddOnPreRenderCompleteAsync. First, in addition to Begin and End methods, RegisterAsyncTask lets you register a timeout method that's called if an asynchronous operation takes too long to complete. You can set the timeout declaratively by including an AsyncTimeout attribute in the page's @ Page directive. AsyncTimeout="5" sets the timeout to 5 seconds. The second advantage is that you can call RegisterAsyncTask several times in one request to register several async operations. As with MethodAsync, ASP.NET delays rendering the page until all the operations have completed. Third, you can use RegisterAsyncTask's fourth parameter to pass state to your Begin methods. Finally, RegisterAsyncTask flows impersonation, culture, and HttpContext.Current to the End and Timeout methods. As mentioned earlier in this discussion, the same is not true of an End method registered with AddOnPreRenderCompleteAsync.

In other respects, an asynchronous page that relies on RegisterAsyncTask is similar to one that relies on AddOnPreRenderCompleteAsync. It still requires an Async="true" attribute in the @ Page directive (or the programmatic equivalent, which is to set the page's AsyncMode property to true), and it still executes as normal through the PreRender event, at which time the Begin methods registered using RegisterAsyncTask are called and further request processing is put on hold until the last operation completes.To demonstrate, the codebehind class in **Figure 7** is functionally equivalent to the one in **Figure 1**, but it uses RegisterTaskAsync instead of AddOnPreRenderCompleteAsync. Note the timeout handler named TimeoutAsyncOperation, which is called if HttpWebRequest.BeginGetRequest takes too long to complete. The corresponding .aspx file includes an AsyncTimeout attribute that sets the timeout interval to 5 seconds. Also note the null passed in RegisterAsyncTask's fourth parameter, which could have been used to pass data to the Begin method.

**Figure 7 AsyncPageTask.aspx.cs**

using System;

using System.Web;

using System.Web.UI;

using System.Web.UI.WebControls;

using System.Net;

using System.IO;

using System.Text;

using System.Text.RegularExpressions;

public partial class AsyncPageTask : System.Web.UI.Page

{

private WebRequest \_request;

protected void Page\_Load(object sender, EventArgs e)

{

PageAsyncTask task = new PageAsyncTask(

new BeginEventHandler(BeginAsyncOperation),

new EndEventHandler(EndAsyncOperation),

new EndEventHandler(TimeoutAsyncOperation),

null

);

RegisterAsyncTask(task);

}

IAsyncResult BeginAsyncOperation(object sender, EventArgs e,

AsyncCallback cb, object state)

{

\_request = WebRequest.Create("http://msdn.microsoft.com");

return \_request.BeginGetResponse(cb, state);

}

void EndAsyncOperation(IAsyncResult ar)

{

string text;

using (WebResponse response = \_request.EndGetResponse(ar))

{

using (StreamReader reader =

new StreamReader(response.GetResponseStream()))

{

text = reader.ReadToEnd();

}

}

Regex regex = new Regex("href\\s\*=\\s\*\"([^\"]\*)\"",

RegexOptions.IgnoreCase);

MatchCollection matches = regex.Matches(text);

StringBuilder builder = new StringBuilder(1024);

foreach (Match match in matches)

{

builder.Append(match.Groups[1]);

builder.Append("<br/>");

}

Output.Text = builder.ToString();

}

void TimeoutAsyncOperation(IAsyncResult ar)

{

Output.Text = "Data temporarily unavailable";

}

}

The primary advantage of RegisterAsyncTask is that it allows asynchronous pages to fire off multiple asynchronous calls and delay rendering until all the calls have completed. It works perfectly well for one asynchronous call, too, and it offers a timeout option that AddOnPreRenderCompleteAsync doesn't. If you build an asynchronous page that makes just one async call, you can use AddOnPreRenderCompleteAsync or RegisterAsyncTask. But for asynchronous pages that place two or more async calls, RegisterAsyncTask simplifies your life considerably.

Since the timeout value is a per-page rather than per-call setting, you may be wondering whether it's possible to vary the timeout value for individual calls. The short answer is no. You can vary the timeout from one request to the next by programmatically modifying the page's AsyncTimeout property, but you can't assign different timeouts to different calls initiated from the same request.

**Wrapping It Up**

So there you have it—the skinny on asynchronous pages in ASP.NET 2.0. They're significantly easier to implement in this upcoming version of ASP.NET, and the architecture is such that you can batch multiple async I/O operations in one request and delay the rendering of the page until all the operations have completed. Combined with async ADO.NET and other new asynchronous features in the .NET Framework, async ASP.NET pages offer a powerful and convenient solution to the problem of I/O-bound requests that inhibit scalability by saturating the thread pool.

A final point to keep in mind as you build asynchronous pages is that you should not launch asynchronous operations that borrow from the same thread pool that ASP.NET uses. For example, calling ThreadPool.QueueUserWorkItem at a page's asynchronous point is counterproductive because that method draws from the thread pool, resulting in a net gain of zero threads for processing requests. By contrast, calling asynchronous methods built into the Framework, methods such as HttpWebRequest.BeginGetResponse and SqlCommand.BeginExecuteReader, is generally considered to be safe because those methods tend to use completion ports to implement asynchronous behavior.