

The module and handler are the entry points to the ASP.NET MVC framework. They perform the following actions:
<http://www.asp.net/mvc/tutorials/older-versions/overview/understanding-the-asp-net-mvc-execution-process>

- Select the appropriate controller in an MVC Web application.
- Obtain a specific controller instance.
- Call the controller's **Execute** method.

The following table lists the stages of execution for an MVC Web project. Create model objects by writing simple classes (These are also known as POCO classes, from "plain-old CLR objects.") You can then have the database created on the fly from your classes, which enables a very clean and rapid development workflow.

Stage	Details
Receive first request for the application	In the Global.asax file, Route objects are added to the RouteTable object.
Perform routing	The UrlRoutingModule module uses the first matching Route object in the RouteTable collection to create the RouteData object, which it then uses to create a RequestContext (IHttpContext) object.
Create MVC request handler	The MvcRouteHandler object creates an instance of the MvcHandler class and passes it the RequestContext instance.
Create controller	The MvcHandler object uses the RequestContext instance to identify the ApiControllerFactory object (typically an instance of the DefaultControllerFactory class) to create the controller instance with.
Execute controller	The MvcHandler instance calls the controller's Execute method.
Invoke action	Most controllers inherit from the Controller base class. For controllers that do so, the ControllerActionInvoker object that is associated with the controller determines which action method of the controller class to call, and then calls that method.
Execute result	A typical action method might receive user input, prepare the appropriate response data, and then execute the result by returning a result type. The built-in result types that can be executed include the following: ViewResult (which renders a view and is the most-often used result type), RedirectToRouteResult , RedirectResult , ContentResult , JsonResult , and EmptyResult .

Before ASP.NET MVC, URLs in web application mapped to physical files at a disk location. So for example if you had a URL 'http://www.testurl.com/products/Default.aspx' it simply meant there was a Default.aspx file in a 'products' folder at the root of the website. This URL had no other meaning. Any parameters it took was probably passed in the query string making it look like 'http://www.testurl.com/products/Default.aspx?productid=99'

Note: To avoid hyperlinks in this article, we have renamed http to http. Wherever you see http, please read it as http.

However if we built a MVC site for the same, the URL would probably look like 'http://www.testurl.com/product/' and 'http://www.testurl.com/product/details/99'. By convention (and default) the first URL maps to the ProductController class with an action named Index. The second URL maps to the ProductController class with an action named Details. As we can see, MVC Routing helps abstract URLs away from physical files which by default maps to Controller/Action method pairs by default.

We will now go into details of how this mapping happens and how to modify the defaults. But before that, some best practices!

[Subscribe to our Free Digital Magazines for .NET Professionals](#)

Routing and URL Etiquettes

The concept of clean URLs essentially came down from frameworks like Ruby. Some of the accepted conventions for clean URLs are:

1. **Keep URLs clean:** For example instead of 'http://www.testurl.com/products/Default.aspx?productid=99&view=details' have 'http://www.testurl.com/product/details/99'

2. **Keep URLs discoverable by end-users:** Having URL parameters baked into routes makes URLs easier to understand and encourages users to play around and discover available functionality. For example in the above URL 'http://www.testurl.com/product/details/100' would mean the product details for product id 100. But guessing numbers is no fun as we will see in the next practice.

3. **Avoid Database IDs in URL:** In the above examples we have used 100, which is a database id and has no meaning for the end user. But if it were to use 'http://www.testurl.com/product/details/whole-milk' our users would probably try out 'http://www.testurl.com/product/details/butter' on their own. This makes points 1 and 2 even more relevant now.

However if IDs are unavoidable consider adding extra information in the URL like 'http://www.testurl.com/product/details/dairy-product-99'

Clean URLs described above have the added advantage of being more 'search robot' friendly and thus is good for SEO.

With the - what (is routing) and why (we should have clean URLs) out of the way, let's looking deeper into how routing works in MVC.

How does the Route Handler work

Quite sometime back I saw a [Scott Hanselman presentation](#) on MVC2 where he stopped the execution of sample MVC app on the Controller Action and then walked through the stack trace to show the inner workings of the MVC pipeline. I tried the same for the route handler and ended up with a 1000+ pixels stack trace, part of which is reproduced below. As highlighted below, the first thing that happens in the pipeline during transition from System.Web to System.Web.Mvc is the execution of all registered handlers.

System.Web.Mvc.dll!System.Web.Mvc.Async.AsyncResultWrapper.TrappedAsyncResult~System.Web.Mvc.Async.Asy	
System.Web.Mvc.dll!System.Web.Mvc.MvcHandler.EndProcessRequest.AnonymousMethod_d() + 0x33 bytes	
System.Web.Mvc.dll!System.Web.Mvc.SecurityUtil.GetCallInAppTrustThunk.AnonymousMethod_0(System.Action f)	
System.Web.Mvc.dll!System.Web.Mvc.SecurityUtil.ProcessInApplicationTrust(System.Action action) + 0x17 bytes	
System.Web.Mvc.dll!System.Web.Mvc.MvcHandler.EndProcessRequest(System.IAsyncResult asyncResult) + 0x3d byte	
System.Web.Mvc.dll!System.Web.Mvc.MvcHandler.System.Web.IHttpAsyncHandler.EndProcessRequest(System.IAsyn	
System.Web.dll!System.Web.HttpApplication.CallHandlerExecutionStep.System.Web.HttpApplication.IExecutionStep.E	
System.Web.dll!System.Web.HttpApplication.ExecuteStep(System.Web.HttpApplication.IExecutionStep step, ref bool c	
System.Web.dll!System.Web.HttpApplication.PipelineStepManager.ResumeSteps(System.Exception error) + 0x42e byte	
System.Web.dll!System.Web.HttpApplication.BeginProcessRequestNotification(System.Web.HttpContext context, Syst	
System.Web.dll!System.Web.HttpRuntime.ProcessRequestNotificationPrivate(System.Web.Hosting.IIS7WorkerRequest	
System.Web.dll!System.Web.Hosting.PipelineRuntime.ProcessRequestNotificationHelper(System.IntPtr managedHttp	
System.Web.dll!System.Web.Hosting.PipelineRuntime.ProcessRequestNotification(System.IntPtr managedHttpContex	
[Native to Managed Transition]	
[Managed to Native Transition]	
System.Web.dll!System.Web.Hosting.PipelineRuntime.ProcessRequestNotificationHelper(System.IntPtr managedHttp	
System.Web.dll!System.Web.Hosting.PipelineRuntime.ProcessRequestNotification(System.IntPtr managedHttpContex	
[Appdomain Transition]	

© DotNetCurry.com

Fact is the RouteHandler is first to be executed. It follows these steps (not evident from the stack trace)

1. Check if route is static file on disk, if so the resource is served directly
2. If it's not a static route, check if there is a custom route handler, if so it hands off the request to the custom route handler
3. If there are no custom route handlers it hands over to the default MVC Routing handler. Now that we have reached the route handler let us see how it treats routes

To see the entire pipeline refer to Steve Sanderson's MVC Pipeline diagram [here](#).

Understanding the default Route declaration

In ASP.NET MVC, by default a couple of routes are defined for you. With the introduction of WebAPI, another additional route is declared for WebAPI controller actions. Let us look at these routes and see what they imply.

```

public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}"); 2

        routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional }
        ); 3

        routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new
            {
                controller = "Home",
                action = "Index",
                id = UrlParameter.Optional
            }
        ); 1
    }
}

```

The Route labeled (1) is named 'Default' and it has a url template of type {controller}/{action}/{id}. Note the third parameter, which is an anonymous object with three properties, each of which matches a section in the url template.

The Route labeled (2) is information for MVC Routing to ignore anything that ends with an axd and having additional parameters. The { *pathInfo } is a wildcard for all query params.

Finally the Route Labeled (3) is for WebApi based controllers that derive of ApiController instead of the regular MvcController. Note WebApi routes are adding using the MapHttpRoute method. Also the route adds an 'api/' for every controller present. So even though you may have the ValuesController derive from ApiController and in the same folder as the HomeController, it will still be mapped to hxxp://<root>/api/Values. Here 'api' is the static part of a route.

Next we will see how we can add our own custom routes for MvcControllers.

Creating Custom Routes

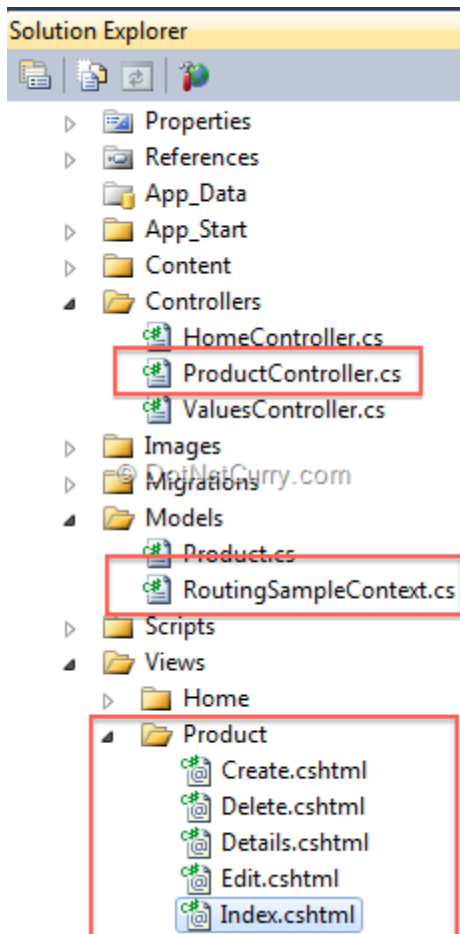
Now if a user visited 'hxxp://www.testurl.com/product/edit/1' it would match the 'Default' route defined above and MVC would look for a controller called ProductController with an action Edit that takes an input parameter called id.

I have created a small application with a single Entity called Product that has the following properties

```
public class Product
{
    [Key()]
    public int Id { get; set; }
    public string Category { get; set; }
    [DisplayName("Name")]
    public string ProductName { get; set; }
    public string Description { get; set; }
    [DisplayName("Available Date")]
    public DateTime AvailableDate { get; set; }
}
```

© DotNetCurry.com

This entity is serialized to the database using the EntityFramework and I built a Controller using the Visual Studio 'Add Controller' wizard. The files added are as highlighted below:



Now if we look in the ProductController.cs, we will find Action methods for Get and Post Http actions for each of the above views. This makes the following default routes available

Path	Route Parameters	Action Method invoked
http://localhost:11618/Product	{controller} = Product {action} = <Index>	ProductController.Index()
http://localhost:11618/Product/Index	{controller} = Product {action} = Index	ProductController.Index()
http://localhost:11618/Product/Details/5	{controller} = Product {action} = Details {id} = 5	ProductController.Details(5)
http://localhost:11618/Product/Edit/5	{controller}=Product {action}=Edit {id}=5	ProductController.Edit(5)
http://localhost:11618/Product/Create	{controller}=Product {action}=Create	ProductController.Create()
http://localhost:11618/Product/Delete/5	{controller}=Product {action}=Delete {id}=5	ProductController.Delete(5)

© DotNetCurry.com

Now let us see some Sample data

Index

[Create New](#)

Category	Name	Description	Available Date	
Electronics	Sony	46" TV	1/1/2012 12:00:00 AM	Edit Details Delete
Electronics	Sony	32" LCD TV	1/1/2010 12:00:00 AM	Edit Details Delete
Laptops	Asus	Zenbook 13"	2/1/2012 12:00:00 AM	Edit Details Delete
Electronics	Samsung	46" LED TV	1/6/2011 12:00:00 AM	Edit Details Delete
Laptops	Apple	MBP 13"	6/1/2012 12:00:00 AM	Edit Details Delete

As we can see above, we have two categories of products and 3 different names. Now each are identified by their IDs and you can use the Edit/Details/Delete action links to act upon them. However, if this list were to get much bigger, it would be really nice to have the ability to look at all items under the category 'Electronics' or 'Electronics' from 'Sony'. Currently there is no such 'filter' available.

Let us implement this. First we add a parameter category to the Index action method, and filter the data we are getting from the Database and return it to the view.

```
public ActionResult Index(string category = "")
{
    if (string.IsNullOrEmpty(category))
    {
        return View(db.Products.ToList());
    }
    else
    {
        return View((from product in db.Products
                      where product.Category == category
                      select product).ToList<Product>());
    }
}
```

Now if we run the application and provide the URL

<http://localhost:11618/Product/Index/Electronics>

We will see the Index page but nothing has changed. It is still showing all Products.



Index

[Create New](#)

Category	Name	Description	Available Date			
Electronics	Sony	46" TV	1/1/2012 12:00:00 AM	Edit	Details	Delete
Electronics	Sony	32" LCD TV	1/1/2010 12:00:00 AM	Edit	Details	Delete
Laptops	Asus	Zenbook 13"	2/1/2012 12:00:00 AM	Edit	Details	Delete
Electronics	Samsung	46" LED TV	1/6/2011 12:00:00 AM	Edit	Details	Delete
Laptops	Apple	MBP 13"	6/1/2012 12:00:00 AM	Edit	Details	Delete

This is because the default route has specified a parameter called {id} but no {id} parameter was available in the Controller method. If we put a breakpoint in the Controller method, we will see that the 'category' parameter is coming in as null.

```
public ActionResult Index(string category)
{
    if (string.IsNullOrEmpty(category))
    {
        return View(db.Products.ToList());
    }
    else
    {
        return View((from product in db.Products
                      where product.Category == category
                      select product).ToList<Product>());
    }
}
```

How can we fix this?

..By defining a new Route that tells the routing handler how to navigate to an action method, when a 'category' parameter is specified for the Index method. The route is follows

```
routes.MapRoute(
    name: "IndexByCategory",
    url: "{controller}/{action}/{category}",
    defaults: new
    {
        controller = "Product",
        action = "Index",
        category = UrlParameter.Optional
    }
);
```

As we can see, we have defined a new route that expects a category parameter. Now when we provide Electronics as a parameter, we get a nicely filtered list



Index

[Create New](#)

© DotNetCurry.com

Category	Name	Description	Available Date	
Electronics	Sony	46" TV	1/1/2012 12:00:00 AM	Edit Details Delete
Electronics	Sony	32" LCD TV	1/1/2010 12:00:00 AM	Edit Details Delete
Electronics	Samsung	46" LED TV	1/6/2011 12:00:00 AM	Edit Details Delete

Now we could add this URL to the Category column such that clicking on any category would filter the list.

Next, to continue making our URL more discoverable, we see we can add a Name filter too. Let us see what the Route and code looks like


```
routes.MapRoute(
    name: "IndexByCategory",
    url: "{controller}/{action}/{category}/{name}",
    defaults: new
    {
        controller = "Product",
        action = "Index",
        category = UrlParameter.Optional,
        name = UrlParameter.Optional
    }
);
```

© DotNetCurry.com

The change is simple, we have added {name} in the URL template and then specified it as a parameter in the anonymous object.

Next we updated the action method in the controller as follows

```
public ActionResult Index(string category, string name)
{
    int cat = String.IsNullOrEmpty(category) ? 0 : 2;
    int nam = String.IsNullOrEmpty(name) ? 0 : 1;
    int val = cat | nam;
    switch (val)
    {
        case 0:
            // This is for the base route .../Product/Index/ or .../Produc
            return View(db.Products.ToList());
        case 2:
            // This is for only category .../Product/Index/Electronics
            return View((from product in db.Products
                where product.Category == category
                select product).ToList<Product>());
        case 1:
            // This case is reachable only via .../Product/Index?name=Asus
            return View((from product in db.Products
                where product.ProductName == name
                select product).ToList<Product>());
        case 3:
            // This case is reachable .../Product/Electronics/Sony
            return View((from product in db.Products
                where product.Category == category &&
                product.ProductName == name
                select product).ToList<Product>());
        default:
            return View();
    }
}
```

We basically do a bit-wise operation to determine which of the parameters have been passed. 00 – No parameters, 01 – Name passed, 10 – Category Passed, 11 – Both name and category passed.

Of the four cases case 1: is interesting because as we see, we can mix clean urls with urls using named query strings. This is because the sequence of parameters is important. So we cannot mix up the sequence of parameters in URL and expect MVC to understand it. If there are optional parameters then parameters coming **after** the optional param must be named in the Url for the routing to work correctly.

So we have now seen what the default ASP.NET route means and how we can leverage routes in MVC to make discoverable URLs that respond to changes in the URL by presenting context aware data.

To round off, we see how we can use the Routing mechanism to generate URLs for us as I mentioned above.

Using MVC Routing to Generate URLs

Currently our Index page only shows links for Edit, Details and Delete. However we could convert the Category and Name column to have links that route MVC to pull up the index page with the selected category and name. In the Index.cshtml, we change the Razor markup as shown below to generate the link

```
@foreach (var item in Model)
{
    <tr>
        <td>
            @Html.ActionLink(item.Category, "Index", new { category = item.Category })
        </td>
        <td>
            @Html.ActionLink(item.ProductName, "Index", new { name = item.ProductName })
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Description)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.AvailableDate)
        </td>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id = item.Id }) |
            @Html.ActionLink("Details", "Details", new { id = item.Id }) |
            @Html.ActionLink("Delete", "Delete", new { id = item.Id })
        </td>
    </tr>
}
```

ActionLink and RouteLink

ActionLinks use the context of the current page while generating the target link. This results in the route-mapping happening based on the sequence in which the routes have been added. However if want to use a specific route to generate the URL we can use the RouteLink HTML helper. An example is shown below

```
@model IEnumerable<RoutingSample.Models.Product>
@{
    ViewBag.Title = "Index";
}
<h2>
    @Html.RouteLink("Index", "Default")</h2>
<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
    <tr>...</tr>
```

Here we are trying to setup the 'Index' heading into a link that navigates to the Product/Index page without any parameters. If we use Action Link, it will use the current page context and automatically add the available parameters, which will result in generated link always pointing to the current page. Instead if we use RouteLink and ask the routing framework to use the 'Default' route, then the link that is generated will point to Product/Index always. Thus the RouteLink html helper comes in really handy when we have to pick and choose which of the available routes we want to use to generate URLs.

DatePicker Lab – “Code Flow”:

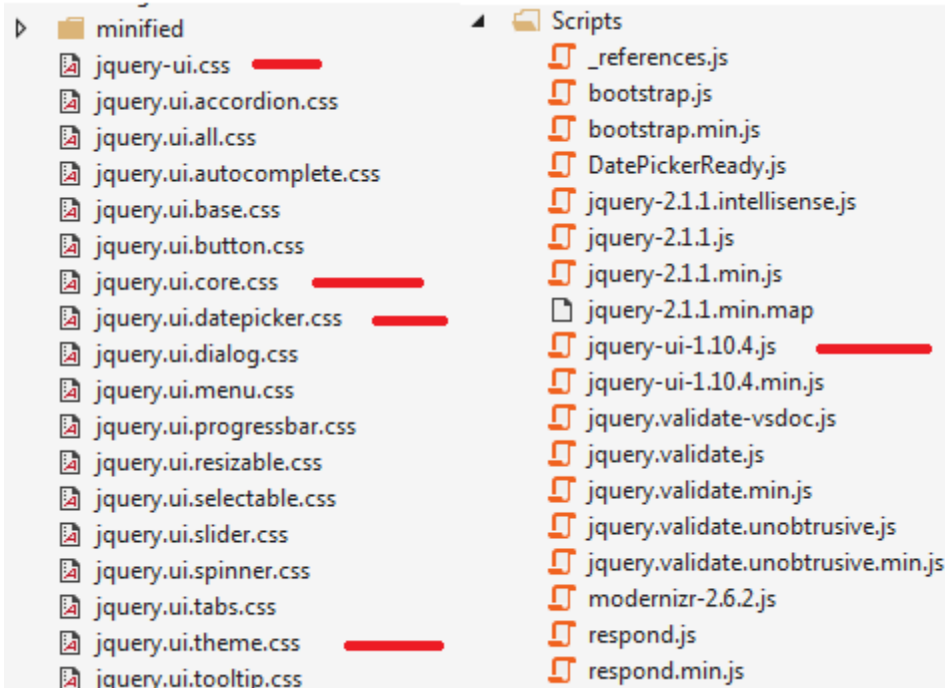
Add the following code to the DatePickerReady.js file:

```
$(function () {
    $(".datefield").datepicker();
});
```

- Brief explanation of what this does: the first line is the "jQuery ready" function, which is called when all the DOM elements in a page have loaded.
 - The second line selects all DOM elements that have the class name datefield, then invokes the datepicker function for each of them.
 - Remember added the datefield class to the Views\Shared\EditorTemplates\Date.cshtml .
1. Create a template for editing dates that will be applied when ASP.NET MVC displays UI for editing model properties that are marked with the Date enumeration of the DataType attribute.
 2. The template will render only the date; time will not be displayed.
 3. Use the jQuery UI Datepicker popup calendar to provide a way to edit dates.
 - a. [DataType(DataType.Date)]
 - b. public DateTime ReleaseDate { get; set; }
 4. causes the **ReleaseDate** field to be displayed without the time in both display templates and edit templates
 5. Application contains a *date.cshtml* template in the *Views\Shared\EditorTemplates* folder or in the *Views\Movies\EditorTemplates* folder, that template will be used to render any **Date** property while editing.
 6. Include files: <http://stackoverflow.com/questions/20081328/how-to-add-jqueryui-library-in-mvc-5-project>

Links from JQueryUI:

- <link rel="stylesheet" href="//code.jquery.com/ui/1.10.4/themes/smoothness/jquery-ui.css">
- <script src="//code.jquery.com/jquery-1.10.2.js"></script>
- <script src="//code.jquery.com/ui/1.10.4/jquery-ui.js"></script>
- <link rel="stylesheet" href="/resources/demos/style.css">



- *Content/themes/base/jquery.ui.core.css* --- (these are the lab datepicker includes)
- *Content/themes/base/jquery.ui.datepicker.css* ---
- *Content/themes/base/jquery.ui.theme.css* ---
- *jquery.ui.core.min.js*
- *jquery.ui.datepicker.min.js*
- *DatePickerReady.js* – we create...

The above files may have changed version, bundle, build etc – what are current needs?

Discussion:

The code you see rendering css and scripts on your `_Layout.cshtml` page (i.e. `@Scripts.Render("~/bundles/modernizr")`) is called **bundling**. Check out some info here: <http://www.asp.net/mvc/tutorials/mvc-4/bundling-and-minification>

So, to add jQueryUI you would do the following:

In your `Global.asax.cs` file you should see a number of registrations:

```
BundleConfig.RegisterBundles(BundleTable.Bundles);
```

This goes to the `BundleConfig` class which registers any bundles. For jQueryUI you could do the following:

```
bundles.Add(new ScriptBundle("~/bundles/jqueryui").Include(
    "~/Scripts/jquery-ui-{version}.js"));
```

This is creating a new script bundle called `~/bundles/jqueryui`.

Then it can be added to your layout page by doing this:

```
@Scripts.Render("~/bundles/jqueryui")
```

Then you'll do the same for css:

```
bundles.Add(new StyleBundle("~/Content/themes/base/css").Include(
    "~/Content/themes/base/jquery.ui.core.css",
    "~/Content/themes/base/jquery.ui.resizable.css",
    "~/Content/themes/base/jquery.ui.selectable.css",
    "~/Content/themes/base/jquery.ui.accordion.css",
    "~/Content/themes/base/jquery.ui.autocomplete.css",
    "~/Content/themes/base/jquery.ui.button.css",
    "~/Content/themes/base/jquery.ui.dialog.css",
    "~/Content/themes/base/jquery.ui.slider.css",
    "~/Content/themes/base/jquery.ui.tabs.css",
    "~/Content/themes/base/jquery.ui.datepicker.css",
    "~/Content/themes/base/jquery.ui.progressbar.css",
    "~/Content/themes/base/jquery.ui.theme.css"));
```

and add it with

```
@Styles.Render("~/Content/themes/base/css")
```

Note:

- In MVC4, a non-empty project already has jQuery set up. For an empty project you would have to add it yourself. Not 100% sure about the new MVC 5.
- You can install jQueryUi from nuget, but the official package doesn't add this bundling stuff.

You could just do the old fashioned referencing of you css and js files (e.g. <script language="JavaScript" src="../../Scripts/jquery.ui.1.8.2.js" />

Discussion 2

A bundle in MVC4 is a collection of scripts, styles or other files bundled together into a single bundle.

You will have a BundleConfig.cs file in the App_Start folder, which contains the settings of which file is added to which bundle.

```
bundles.Add(new ScriptBundle("~/bundles/jqueryval").Include(
    "~/Scripts/jquery.unobtrusive*",
    "~/Scripts/jquery.validate*"));
```

As you can see above ~/bundles/jqueryval is the virtual path of the bundle which combines the files specified in it. So later on when you see this:

```
@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

The above will include the scripts bundled under that reference.

Should I keep it? Should I reference all of these JS files that were initially defined in BundleConfig.cs?

In the case of the jqueryval bundle you might find that the unobtrusive and validation scripts included are very useful.

They are the scripts which will take care of managing unobtrusive validation, keeping your DOM nice and clean.

You can remove the bundle off course if you don't need or want to use unobtrusive validation. If you do that then I believe you will also need to update your web.config, setting the required fields to false to ensure your project will not be looking for the files, similar to this:

```
<add key="ClientValidationEnabled" value="false" />
<add key="UnobtrusiveJavaScriptEnabled" value="false" />
```

The benefit and exact difference between using obtrusive and unobtrusive validation is explained very well in this article: [Brad Wilson: Unobtrusive Client Validation in ASP.NET MVC 3](#)

In general, I would assume it is good to only include what you need. If you don't need all the files specified in a bundle, remove those files, exclude the bundle all together or create your own custom bundles.

Trial and error. If you remove them and find random exceptions in your browser debugger console, try adding some of the files/bundles back in.

In general, bundling also works with style-sheets:

```
bundles.Add(new StyleBundle("~/Content/themes/base/css").Include(
    "~/Content/themes/base/jquery.ui.core.css",
    "~/Content/themes/base/jquery.ui.resizable.css",
    "~/Content/themes/base/jquery.ui.selectable.css",
    "~/Content/themes/base/jquery.ui.accordion.css",
    "~/Content/themes/base/jquery.ui.autocomplete.css",
    "~/Content/themes/base/jquery.ui.button.css",
    "~/Content/themes/base/jquery.ui.dialog.css",
    "~/Content/themes/base/jquery.ui.slider.css",
    "~/Content/themes/base/jquery.ui.tabs.css",
    "~/Content/themes/base/jquery.ui.datepicker.css",
    "~/Content/themes/base/jquery.ui.progressbar.css",
    "~/Content/themes/base/jquery.ui.theme.css"));
```

The benefit to the developer is only having to reference an individual bundle instead of several files.

The benefit to the client is how many individual loads the browser has to do to get the scripts/css files.

If you for example have 5 files references in your view the client browser will download all 5 separately and there is a limit in each browser how many files can be downloaded simultaneously. This means that if a client has a slower connection they could wait a few seconds before the files are loaded.

However, if you have all 5 files configured to be in a single bundle, the browser only downloads 1 file, the bundled one.

In addition the bundles are minified (or the files in the bundles) so you are not only saving on time it takes to download the scripts but you also save on download size.

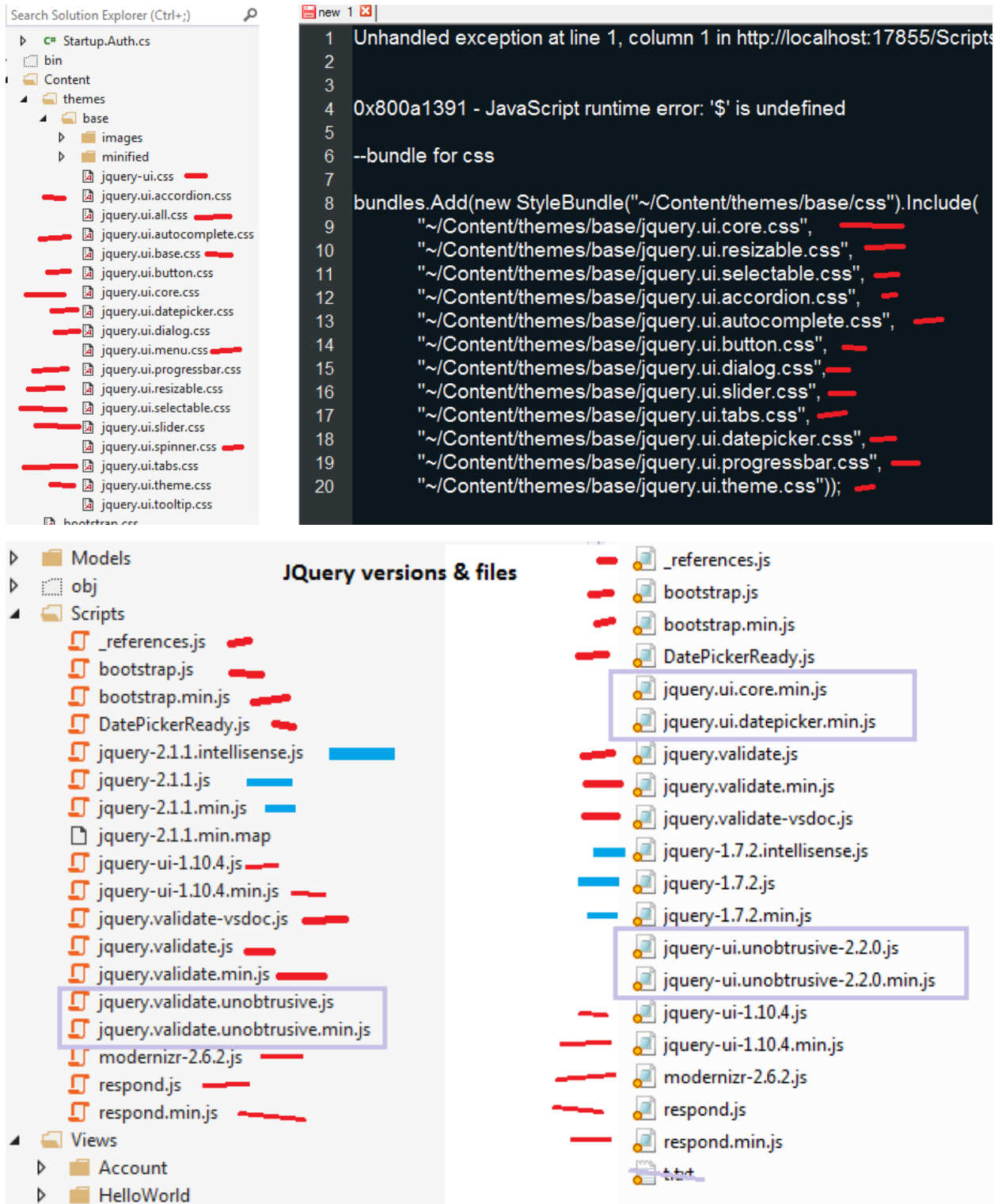
When you test this, note in debug mode is no difference, you need to be in release mode or enable optimization of the bundle table in the BundleConfig.cs file at the bottom of the RegisterBundles method.

BundleTable.EnableOptimizations = true;

You don't have to use the bundles, you still can freely reference individual scripts/css files. It does makes things easier though when you need it.

List from MVC 5 app with JQuery "All" package

<pre> bundles.Add(new StyleBundle("~/Content/themes/base/css").Include("~/Content/themes/base/jquery.ui.core.css", "~/Content/themes/base/jquery.ui.resizable.css", "~/Content/themes/base/jquery.ui.selectable.css", "~/Content/themes/base/jquery.ui.accordion.css", "~/Content/themes/base/jquery.ui.autocomplete.css", "~/Content/themes/base/jquery.ui.button.css", "~/Content/themes/base/jquery.ui.dialog.css", "~/Content/themes/base/jquery.ui.slider.css", "~/Content/themes/base/jquery.ui.tabs.css", "~/Content/themes/base/jquery.ui.datepicker.css", "~/Content/themes/base/jquery.ui.progressbar.css", "~/Content/themes/base/jquery.ui.theme.css", "~/Content/themes/base/jquery.ui.css" "~/Content/themes/base/jquery.ui.all.css", "~/Content/themes/base/jquery.ui.base.css", "~/Content/themes/base/jquery.ui.menu.css", "~/Content/themes/base/jquery.ui.spinner.css", "~/Content/themes/base/jquery.ui.tooltip.css")); </pre>	<p>Add to _layout:</p> <pre> @Styles.Render("~/Content/themes/base/css") </pre>
--	---



The screenshot displays three panels from a Visual Studio IDE:

- Solution Explorer (Top Left):** Shows the project structure. The **Content** folder is expanded, showing a **themes** folder with a **base** subfolder. The **base** folder contains a **css** subfolder with various jQuery UI CSS files (e.g., `jquery-ui.all.css`, `jquery-ui.accordion.css`, etc.).
- Console Window (Top Right):** Shows an error message: "Unhandled exception at line 1, column 1 in http://localhost:17855/Scripts/0x800a1391 - JavaScript runtime error: '\$' is undefined". Below the error, the code for `--bundle for css` is visible, showing a `bundles.Add` call that includes several CSS files from the `~/Content/themes/base/css` directory.
- File Explorer (Bottom):** Titled "jQuery versions & files", it lists various JavaScript files. The files are organized into two columns. The left column lists files like `_references.js`, `bootstrap.js`, `bootstrap.min.js`, `DatePickerReady.js`, `jquery-2.1.1.intellisense.js`, `jquery-2.1.1.js`, `jquery-2.1.1.min.js`, `jquery-2.1.1.min.map`, `jquery-ui-1.10.4.js`, `jquery-ui-1.10.4.min.js`, `jquery.validate-vsdoc.js`, `jquery.validate.js`, `jquery.validate.min.js`, `jquery.validate.unobtrusive.js`, `jquery.validate.unobtrusive.min.js`, `modernizr-2.6.2.js`, `respond.js`, and `respond.min.js`. The right column lists files like `_references.js`, `bootstrap.js`, `bootstrap.min.js`, `DatePickerReady.js`, `jquery.ui.core.min.js`, `jquery.ui.datepicker.min.js`, `jquery.validate.js`, `jquery.validate.min.js`, `jquery.validate-vsdoc.js`, `jquery-1.7.2.intellisense.js`, `jquery-1.7.2.js`, `jquery-1.7.2.min.js`, `jquery-ui.unobtrusive-2.2.0.js`, `jquery-ui.unobtrusive-2.2.0.min.js`, `jquery-ui-1.10.4.js`, `jquery-ui-1.10.4.min.js`, `modernizr-2.6.2.js`, `respond.js`, and `respond.min.js`. Several files are highlighted with red boxes, indicating they are the focus of the note.

if you're using any plugins with jquery, make sure you use the following order of setting reference to those files.

1. reference to the jquery library
2. reference to the other subsequent plug-in (dependant) libraries and so on...

e.g.:

- 1."script src="js/jquery-1.3.2.min.js" type="text/javascript"...
- 2."script src="js/jqDnR.min.js" type="text/javascript"...
- 3."script src="js/jquery.jqpopup.min.js" type="text/javascript"...
- 4."script src="js/jquery.bgiframe.min.js" type="text/javascript"...

Always make sure you must put the jquery reference to first and then the subsequent libraries.

<http://stackoverflow.com/questions/339314/jquery-is-undefined>

```
@Html.ActionLink("Edit", "Edit", new { id=item.ID })
@Html.ActionLink("Link Text", "ActionName", new { id=item.ID })
```

```
<td>
@Html.ActionLink("Edit Me", "Edit", new { id=item.ID }) |
@Html. (extension) MvcHtmlString HtmlHelper.ActionLink(string linkText, string actionName, object routeValues)
@Html. Returns an anchor element (a element) that contains the virtual path of the specified action.

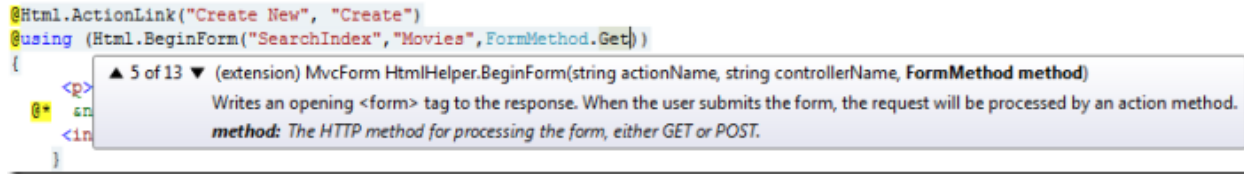
Exceptions:
    System.ArgumentException
```

The ASP.NET MVC model binder takes the posted form values and creates a Movie object that's passed as the movie parameter. The ModelState.IsValid method verifies that the data submitted in the form can be used to modify (edit or update) a Movie object. If the data is valid, the movie data is saved to the Movies collection of the db(MovieDbContext instance). The new movie data is saved to the database by calling the SaveChanges method of MovieDbContext. After saving the data, the code redirects the user to the Index action method of the MoviesController class, which displays the movie collection, including the changes just made.

Imagine that you want to bookmark a particular search or you want to send a link to friends that they can click in order to see the same filtered list of movies. Notice that the URL for the HTTP POST request is the same as the URL for the GET request (localhost:xxxxx/Movies/Index) -- there's no search information in the URL itself. Right now, the search string information is sent to the server as a form field value. This means you can't capture that search information to bookmark or send to friends in a URL.

The solution is to use an overload of [BeginForm](#) that specifies that the POST request should add the search information to the URL and that it should be routed to the **HttpGet** version of the **Index** method. Replace the existing parameterless **BeginForm** method with the following markup:

```
@using (Html.BeginForm("Index", "Movies", FormMethod.Get))
```



Now when you submit a search, the URL contains a search query string. Searching will also go to the **HttpGet Index** action method, even if you have a **HttpPost Index** method.

The date type is an HTML5 input type that enables HTML5-aware browsers to render a HTML5 calendar control. Later on you'll add some JavaScript to hook up the jQuery datepicker to the `Html.TextBox` element using the `datefield` class.

To actually use the jQuery date picker, you need to create a jQuery script that will hook up the calendar widget to the edit template. In Solution Explorer, right-click the Scripts folder and select Add, then New Item, and then JScript File. Name the file `DatePickerReady.js`.

Add the following code to the `DatePickerReady.js` file:

```

$(function () {
    $(".datefield").datepicker();
});

```

If you're not familiar with jQuery, here's a brief explanation of what this does: the first line is the "jQuery ready" function, which is called when all the DOM elements in a page have loaded. The second line selects all DOM elements that have the class name `datefield`, then invokes the `datepicker` function for each of them. (Remember that you added the `datefield` class to the `Views\Shared\EditorTemplates\Date.cshtml` template earlier in the tutorial.)

Next, open the `Views\Shared_Layout.cshtml` file. You need to add references to the following files, which are all required so that you can use the date picker:

- `Content/themes/base/jquery.ui.core.css`
- `Content/themes/base/jquery.ui.datepicker.css`
- `Content/themes/base/jquery.ui.theme.css`
- `jquery.ui.core.min.js`
- `jquery.ui.datepicker.min.js`
- `DatePickerReady.js`

The following example shows the actual code that you should add at the bottom of the `head` element in the `Views\Shared_Layout.cshtml` file.

```

<link href="@Url.Content("~/Content/themes/base/jquery.ui.core.css")"
      rel="stylesheet" type="text/css" />
<link
href="@Url.Content("~/Content/themes/base/jquery.ui.datepicker.css")"
rel="stylesheet" type="text/css" />
<link href="@Url.Content("~/Content/themes/base/jquery.ui.theme.css")"

```

```
rel="stylesheet" type="text/css" />

<script src="@Url.Content("~/Scripts/jquery.ui.core.min.js")"
    type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/jquery.ui.datepicker.min.js")"
    type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/DatePickerReady.js")"
    type="text/javascript"></script>
```