ASP.NET MVC Concepts

----------------------------------

Modell

* ======Model represents the real world object and provides data to the View.

View

* =======The View is responsible for the look and feel.

Controller

* The Controller is responsible for taking the end user request and loading the appropriate Model and View.

Viewbag

1. ViewBag transfers data from the controller to the view, ideally temporary data which in not included in a model.
2. ViewBag is a dynamic property that takes advantage of the new dynamic features in C# 4.0
3. You can assign any number of properties and values to ViewBag
4. The ViewBag's life only lasts during the current http request. ViewBag values will be null if redirection occurs.
5. ViewBag is actually a wrapper around ViewData.

Viewresult

|  |  |
| --- | --- |
|  | ActionResult is an abstract class that can have several subtypes. **ActionResult Subtypes**  * **ViewResult** - Renders a specifed view to the response stream * **PartialViewResult** - Renders a specifed partial view to the response stream * **EmptyResult** - An empty response is returned * **RedirectResult** - Performs an HTTP redirection to a specifed 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 |

Scaffolding.

ASP.NET **Scaffolding** is a code generation framework for ASP.NET Web applications. Visual Studio 2013 includes pre-installed code generators for **MVC** and Web API projects. You add **scaffolding** to your project when you want to quickly add code that interacts with data models.

**Htmlhelpers**

| **HtmlHelper** | **Strogly Typed HtmlHelpers** | **Html Control** |
| --- | --- | --- |
| Html.ActionLink |  | Anchor link |
| Html.TextBox | Html.TextBoxFor | Textbox |
| Html.TextArea | Html.TextAreaFor | TextArea |
| Html.CheckBox | Html.CheckBoxFor | Checkbox |
| Html.RadioButton | Html.RadioButtonFor | Radio button |
| Html.DropDownList | Html.DropDownListFor | Dropdown, combobox |
| Html.ListBox | Html.ListBoxFor | multi-select list box |
| Html.Hidden | Html.HiddenFor | Hidden field |
| Password | Html.PasswordFor | Password textbox |
| Html.Display | Html.DisplayFor | Html text |
| Html.Label | Html.LabelFor | Label |
| Html.Editor | Html.EditorFor | Generates Html controls based on data type of specified model property e.g. textbox for string property, numeric field for int, double or other numeric type. |

**Tempdata**

TempData in ASP.NET MVC is basically a dictionary object derived from TempDataDictionary. TempData stays for a subsequent HTTP Request as opposed to other options (ViewBag and ViewData) those stay only for current request. So, TempdData can be used to maintain data between controller actions as well as redirects.

public ActionResult TempDataExample()

{

TempData["Message"] = "Temp data Hello World!";

return View();

}

**Filters and Attributes**

### **Type of filters**

Now taking this discussion further, Let us first discuss the various types of filters that can be implemented to inject custom processing logic.

* Authorization filter
* Action filter
* Result filter
* Exception filter

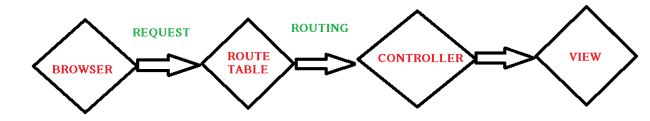
### **Points to Remember :**

1. MVC Filters are used to execute custom logic before or after executing action method.
2. MVC Filter types:
   1. Authorization filters
   2. Action filters
   3. Result filters
   4. Exception filters
3. Filters can be applied globally in FilterConfig class, at controller level or action method level.
4. Custom filter class can be created by implementing FilterAttribute class and corresponding interface.

**MVC Pipeline**

**Pipeline In MVC**

* **ROUTING:** This is the first step in the MVC pipeline, it is a pattern to match the incoming URL request to the application. These URLs are matched to the URLs registered to the Route Table. The route table has its set up done in the Global.asax. The routes are registered to the RouteConfig class that has the route URLs to be mapped and ignored in the application. Let's have a look at it:  
  1. **protected** **void** Application\_Start()
  2. {
  3. RouteConfig.RegisterRoutes(RouteTables.Routes);
  4. }
  5. **public** **class** RouteConfig
  6. {
  7. **public** **static** **void** RegisterRoutes(RouteCollection routes)
  8. {
  9. routes.Ignore(“{resource}.axd/{\*pathinfo}”);
  10. routes.MapRoute(
  11. name:"",
  12. url:"{controller}/{action}/{id}",
  13. defaults: **new** {controller = "Home",action="Index",id=UrlParameter.optional}
  14. );
  15. }
  16. }

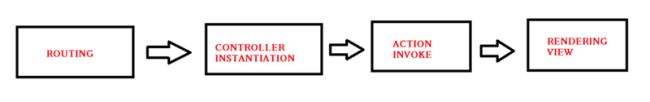
Then the UrlRoutingModule discussed previously finds a matching route for the incoming URL from the Route Table. When it matches, it retrieves the IRouteHandler instance for that route (MvcRoutehandler). Then the IRouteHandler's **GetHttpHandler()** is called and finally the **ProcessRequest()** method is invoked. Then MvcRoutehandler instantiates and executes controller.   
  
Here as you can see, the URL has the **{controller}/{action}/{id}**, the id is optional that is defined in the defaults. The URL may also be only **{controller}/{action}**. Since the routes will be specified or registered in the Route Table, so will be the navigation.   
  


* 1. **public** **class** TenantController : Controller //Derived from Base controller
  2. {
  3. **public** ActionResult Index() //Default Action method-URL: /Home/Index
  4. {
  5. **return** View();
  6. }
  7. **public** ActionResult Index(**int** id) //Action Method with an Id as parameter     // -URL: /Home/Index/3
  8. {
  9. **return** View();
  10. }
  11. }

This is how the routing works on an incoming request from the browser. Similarly we can use custom routes also since we do not always use Home/Index .. ;)

* 1. **public** **class** RouteConfig
  2. {
  3. **public** **static** **void** RegisterRoutes(RouteCollection routes)
  4. {
  5. routes.Ignore(“{resource}.axd/{\*pathinfo}”);
  6. routes.MapRoute(
  7. name:"Employee",
  8. url:"{controller}/{action}/{id}",
  9. defaults: **new**{controller = "Employee",action="Add",id=UrlParameter.optional}//Adds a new employee action
  10. );
  11. }
  12. }

If the URL matches any of the routings registered in the Table, the user lands on the View else 404 error not found page.   
  
                              

* **Initialization of Controller:** The real processing starts by the use of ProcessRequest(), that uses the IcontrollerFactory instance to create the corresponding controller for the URL request. The IcontrollerFactory returns the appropriate controller for the request and instantiates that. Then the Execute() method mentioned above is invoked.
* **Invoking Controller Action:** The Execute() method refers to the RouteData to get the appropriate action for the controller in the URL request. The controller then invokes its own InvokeAction() method as in the following:  
    
  public virtual bool InvokeAction(ControllerContext controllerContext, string actionName)  
    
  After getting the appropriate action, then comes the work of the model binders, the default being System.Web.Mvc.DefaultModelBinder, receives the Http request, validates and does the conversions required and also maps the input values from the user.
* **Render View:** The user will now be expecting the beauty of the application to be shown. The View follows the same pattern as the Route. That is, the controller factory has a property known as **ViewFactory()**, that is a type of IviewFactory. This **IviewFactory** defines a method called **CreateView()**, that gets a name and instantiates and returns an **IView** that has a property/method called RenderView() that has the necessary context information from the HttpResponse and returns the HTML data as the response.  
    
  



Layouts in MVC

Controller and Action in MVC

Partial views

@html.partial(“webpage name”);

Data annotations

* **Required** – Indicates that the property is a required field
* **DisplayName** – Defines the text we want used on form fields and validation messages
* **StringLength** – Defines a maximum length for a string field
* **Range** – Gives a maximum and minimum value for a numeric field
* **Bind** – Lists fields to exclude or include when binding parameter or form values to model properties
* **ScaffoldColumn** – Allows hiding fields from editor forms

**Sessions**

//adding data to session

//assuming the method below will return list of Products

var products=Db.GetProducts();

//Store the products to a session

Session["products"]=products;

//To get what you have stored to a session

var products=Session["products"] as List<Product>;

//to clear the session value

Session["products"]=null;

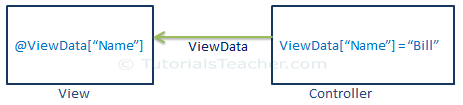
Viewdata

# **ViewData:**

ViewData is similar to ViewBag. It is useful in transferring data from Controller to View.

ViewData is a dictionary which can contain key-value pairs where each key must be string.

The following figure illustrates the ViewData.

[](http://www.tutorialsteacher.com/Content/images/mvc/viewdata.png)

Viewmodel

A view model represents the data that you want to display on your view/page, whether it be used for static text or for input values (like textboxes and dropdown lists) that can be added to the database (or edited). It is something different than your domain model. It is a model for the view.

Attribute Routing

Routing

public static void RegisterRoutes(RouteCollection routes)

{

routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");

routes.MapRoute(

name: "Student",

url: "students/{id}",

defaults: new { controller = "Student", action = "Index"}

);

routes.MapRoute(

name: "Default",

url: "{controller}/{action}/{id}",

defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }

);

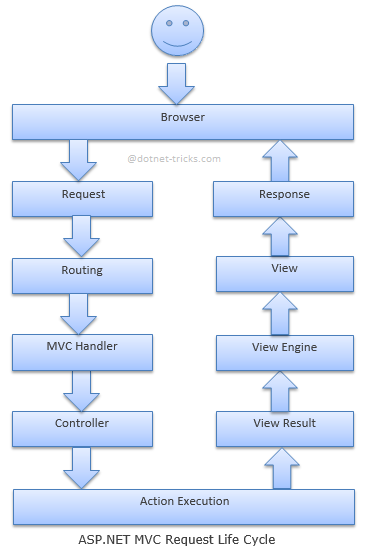
Client side validations in MVC

You can enable client side validation for specific view only by adding Html.EnableClientValidation(true) at the top in view page.

MVC Razor

Using @ in the views

Request lifecycle



Server side validations

**We write the validation script in controller or view**

Redirect Action

return RedirectToAction("Index","Customers");

Redirect to route

return RedirectToRoute("MyCustomRoute", new { Id = 5 });

Return View

return View("NameOfView", Model);

Render Partial

Html.Partial returns a String. Html.RenderPartial calls Write internally and returns void

The main difference is that "RenderPartial" returns void and output will be written directly to the output sream, where as the "Partial" returns MvcHtmlString which can be assigned to a variable and manipulate it if required. So, when there is no need to assign the output to a variable for manipulating it, then use Partial, else use RenderPartial. Renderpartial does exactly the same thing and is better for performance over partial().

Render Action

Rendering the action in the views

Caching in asp.net mvc

Master pages

Render Body

The RenderBody method resides in the master page, or in Razor this is commonly referred to as the *Layout* page. There can only be one RenderBody method per Layout page.

Render section

@RenderSection("scripts", required: false)

Ajax Helpers

MVC Filters types

Authentication

Authorization

Webgrid

Http Get

JSON Results

Razor Engine

**Razor View Engine**  
1. Razor Engine is an advanced view engine that was introduced with MVC3. This is not a new language but it is a new markup syntax.  
  
2. The file extensions used with Razor Engine are different from Web Form Engine. It has .cshtml (Razor with C#) or .vbhtml (Razor with VB) extension for views, partial views, editor templates and for layout pages.  
  
3. Razor has new and advance syntax that are compact, expressive and reduces typing.  
  
4. Razor syntax are easy to learn and much clean than Web Form syntax. Razor uses @ symbol to make the code like as:  
  
@Html.ActionLink("SignUp", "SignUp")  
  
5. Razor Engine support TDD (Test Driven Development) since it is not depend on System.Web.UI.Page class.

ASP.NET Engine

Exception Error Handling and Login