**Strongly typed views vs Dynamic typed views -**

**Model & ViewModel difference –**

* The only difference is Model is a domain model, whereas ViewModel is used only in rendering Views.
* So ViewModel is attached to Views instead of Models. For this we need to create a separate folder named ViewModels, and we place all viewmodels here.
* Use –
  + We can safeguard database from redundant data.

**Routing -**

* Routing defines a URL pattern that maps to a request handler.
* E.g.,

**URL –**

<http://localhost/home/index/123>

**Route –**

Public class RouteConfig(RouteCollection routes)

{

routes.ignoreRoutes(“{resources}.axd/{\*pathInfo}”);

routes.MapRoute(

name : “myRoute”,

url : “{controller}/{action}/{id}”,

default : new {controller=”Home”, action=”Index”, id=UrlParameter.Optional}

);

}

* **Route Constraints –**
  + The constraints that we can apply to a route OR route parameter.
  + ***Add constraint to URL –*** you can append some string after the host name. If you want to have URL like <http://www.myDomain.com/dhiraj/home/index>, then define **url: {dhiraj}/{controller}/{action}** in route.
  + ***Allow only integer parameter in controller –*** add **constraints: new {“id”, @“/d+”}** in the route.
  + ***Allow only specific controller or action –*** if we want to allow only controllers that starts with ‘H’ & only ‘Index’ or ‘contact’ action methods are allowed, then add **new {controller=”^H.\*”, action=”^Index$ | ^Contact$”}** in the route.
* Conventional routing vs Attribute routing.

**Model binding in MVC –**

* With model binding, MVC framework converts the http request values (from query string or form collection) to action method parameters. These parameters can be of primitive type or complex type.
* Model binding is a two step process.

First, it collects values from the incoming http request and

second, populates primitive type or complex type with these values.

* **Binding to Primitive type –** 
  + HTTP GET embeds data into query string. MVC will automatically convert query string parameters into action method parameters.
  + E.g.,

**Sample Requests -**

/Student/Edit?id=1

/Student/Edit/1

**Controller -**

Public actionResult Edit(int id)

{

Var student = studentList.Where(s=> s.studentId == id);

Return View(student);

}

In this example, the query string “id” from the GET request will automatically converted into “id” parameter of the Edit action method.

* + Query string values will be converted into parameters based on matching names.
  + The order of the parameters is not important, the important thing here is the parameter names.
* **Binding to Complex type –** 
  + MVC automatically converts form field data of HTTP POST request to the properties of complex type parameters of the action method.
  + E.g.,

**Model class –**

public class Student

{

public int StudentId { get; set; }

public string StudentName { get; set; }

public int Age { get; set; }

public Standard standard { get; set; }

}

public class Standard

{

public int StandardId { get; set; }

public string StandardName { get; set; }

}

**Action method –**

[HttpPost]

public ActionResult Edit(Student std)

{

var id = std.StudentId;

var name = std.StudentName;

var age = std.Age;

var standardName = std.standard.StandardName;

//update database here..

return RedirectToAction("Index");

}

So, MVC will automatically maps form collection values into Student model properties.

* + You can also include **FormCollection** type parameter in the action method instead of complex type, to retrieve all the values from view form fields as shown below.

e.g.,

**View –**

Student Name

Age

Submit

**Action method –**

[HttpPost]

public ActionResult Edit(FormCollection formValues)

{

var name = formValues[“StudentName”];

var age = formValues[“Age”];

//update database here..

return RedirectToAction("Index");

}

* + The **[Bind] attribute** will let you specify the exact properties a model binder should include or exclude in binding.

e.g.,

**Action method –**

[HttpPost]

public ActionResult Edit([Bind(Include = "StudentId, StudentName")] Student std)

{

var id = std.StudentId;

var name = std.StudentName;

//write code to update student

return RedirectToAction("Index");

}

[HttpPost]

public ActionResult Edit([Bind(Exclude = "Age")] Student std)

{

var name = std.StudentName;

//write code to update student

return RedirectToAction("Index");

}

The Bind attribute will improve the performance by only binding properties which you require.

**Action Selectors –**

* Action selector is the attribute that can be applied to an action method
* It helps routing engine to identify the correct action method to handle
* MVC 5 has following action selectors –
* **ActionName –** 
  + - This attribute allows to specify different action name other than method name
    - E.g.,

*[ActionName(“GetEmployee”)]*

*[HTTPGET]*

*Public ActionResult GetEmployeeById (int id)*

*{*

*// code to retrieve employee details by ID*

*}*

So now, the request URL can be - **/GetEmployee/1**

* **NonAction –**
  + This attribute specifies that a public method of a controller is not an action method.
  + Use – When you want a public method in a controller but don’t want it to be action method.
  + E.g.,

*[NonAction]*

*Public ActionResult GetEmployeeById (int id)*

*{*

*// code to retrieve employee details by ID*

*}*

* **ActionVerbs –**
  + MVC framework supports different ActionVerbs, such as HttpGet, HttpPost, HttpPut, HttpDelete, HttpOptions & HttpPatch.
  + You can apply these attributes to action method to indicate the kind of Http request the action method supports.
  + If you do not apply any attribute then it considers it a GET request by default.
  + We can treat a single action method for both HTTP GET & POST by using **AcceptVerbs** attribute.

e.g.,

*[AcceptVerbs(HttpVerbs.Post | HttpVerbs.Get)]*

*Public ActionResult AcceptBoth()*

*{*

*// code goes here*

*}*

**Partial View –**

* Partial view is a reusable view, which can be used as a child view in multiple other views.
* You can use the partial view in the layout view, as well as other content views.
* If a partial view will be shared with multiple views of different controller folder then create it in the Shared folder, otherwise you can create the partial view in the same folder where it is going to be used.
* We can render partial view using html helper methods – Partial(), RenderPartial(), RenderAction()
* **Html.Partial() –** 
  + It takes partial view name as input and returns MVCHtmlString.
  + As it returns HTML string, you can modify it before rendering.
  + E.g., @Html.Partial("\_HeaderNavBar")
* **Html.RenderPartial() –** 
  + It returns void & directly writes the resulted html output to the http response stream.
  + A semicolon is required at the end of statement & it must be enclosed in braces.
  + E.g., @{

Html.RenderPartial("\_HeaderNavBar");

}

* **Html.RenderAction() –** 
  + It invokes a specific Controller & Action and renders result as a partial view.
  + The specified action method should return PartialViewResult using Partial() method.
  + E.g., @Html.RenderAction(“yourActionName”);

**ViewBag–**

* Viewbag can be used to transfer small data from Controller to View.
* It is a dynamic property that takes advantage of dynamic feature in C# 4.0
* You can assign any number of properties & values to ViewBag
* Value persists only during current http request. Its value becomes null if redirection occurs.
* ViewBag is a wrapper around ViewData.
* Type casting is not required before using as it is a dynamic property.
* E.g.,

**Action method –**

ViewBag.Name = “My ViewBag Value”;

**View –**

<h1> @ViewBag.Name </h1>

**ViewData–**

* It is used to transfer small data from Controller to View.
* It is a dictionary object derived from ViewDataDictionary.
* We can add key-value pair to it.
* Value persists only during current http request. So, value is lost if redirection occurs.
* Type casting is required before using it.
* We should check for null to avoid runtime error.
* E.g.,

**Action method –**

public ActionResult Index()

{

IList<Student> studentList = new List<Student>();

studentList.Add(new Student(){ StudentName = "Bill" });

studentList.Add(new Student(){ StudentName = "Steve" });

studentList.Add(new Student(){ StudentName = "Ram" });

ViewData["students"] = studentList;

return View();

}

**View –**

<ul>

@foreach (var std in ViewData["students"] as IList<Student>)

{

<li>

@std.StudentName

</li>

}

</ul>

* ViewData & ViewBag both uses same dictionary internally. So, the key of ViewData & property of ViewBag **MUST not** match, otherwise you will get a runtime error.

**TempData–**

* TempData is used to transfer non-sensitive data from one action method to another action method of same or different controller.
* It is a dictionary type derived from TempDataDictionary.
* We can add key-value pair to it.
* It uses session to store the data, so we can say it is a short-lived session.
* Type casting is required before using it.
* We should check for null to avoid runtime error.
* Value persists during subsequent requests, i.e. value persists between two consecutive requests. So, value is preserved during redirection.
* If you want to access the value for 3rd request, then use **TempData.Keep()**.
* E.g.,

public class HomeController : Controller

{

public HomeController()

{

}

public ActionResult Index()

{

TempData["myData"] = "Test data";

return View();

}

public ActionResult About()

{

string data;

if(TempData["myData"] != null)

data = TempData["myData"] as string;

**TempData.Keep();**

return View();

}

public ActionResult Contact()

{

string data;

if(TempData["myData"] != null)

data = TempData["myData"] as string;

return View();

}

}

**Filters –**

* + Filters are used to execute some custom logic before or after executing an action method.
  + Types & order of execution of filters –

1. Authentication filters
2. Authorization filters
3. Action filters
4. Result filters
5. Exception filters
   * Filters can be applied Globally (in application\_start method of Global.asax) OR at Controller level OR at Action method level.
   * Custom filters can be created by implementing *FilterAttribute* class and corresponding interface (like, IExceptionFilter)
   * Built-in filters - [Authorize], [RequireHttps],[ChildActionOnly],[OutputCache],[HandleError].
   * **[HandleError] attribute –** 
     + Exceptions which are not handled by try/catch block are called unhandled exceptions, these unhandled exceptions can be handled by a built-in exception filter, i.e., HandleErrorAttribute.
     + It renders Error.cshtml page from Shared folder to display the error page.
     + It also needs to have ***<customErrors mode="On" />*** in web.config.

**Bundling & Minification –**

* Bundling allows us to load bunch of static files from server in single http request
* Introduced in MVC 4 to improve request load time
* Minification optimizes the script or css files by removing extra whitespaces, comments and shortening variable names to single characters.
* Types of Bundles – ScriptBundle, StyleBundle, DynamicFolderBundle
* ScriptBundle does minification of Js files.
* StyleBundle does minification of css files.
* Use **{version}** wildcard to render available version files at runtime.
* Use Scripts.Render("bundle name") method to include script bundle in a razor view.
* Use Styles.Render("bundle name") method to include style bundles in a razor view.

**Area –**

* ASP.NET MVC 2 introduced Area. Area allows us to partition large application into smaller units where each unit contains separate MVC folder structure.

**Security Features in MVC –**

* **Cross Site Scripting (XSS) –** 
  + XSS attack means, an attacker can inject some script in the textbox/input fields of a form and try to execute some malicious code.
  + By default, MVC handles this attack. So, when we try to enter some script in the input field, MVC will throw error saying –

*“A potentially dangerous request.form value was detected from the client…”*

* + What if we want to insert some data like styling the HTML from the textbox, i.e., we want to insert some script data. Then we can add ***[ValidateInput(False)]*** attribute to the action method. This will allow the script tags to be inserted.

***[ValidateInput(false)]***

***public ActionResult Savedata(Customer c)***

***{***

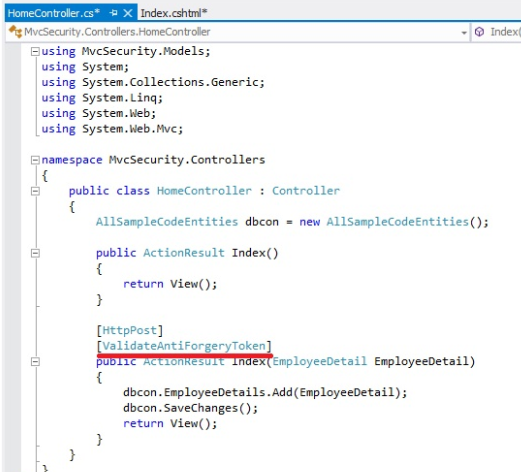
***return View("SavedataAspx",c);***

***}***

* **Cross Site Request Forgery (CSRF) –** 
  + You may have seen an ad that appears suddenly on some of the websites. Once we click on that popup ad, it redirects to some URL and moves back to our own website. We think that it is just an ad popup. But you have a wrong impression. These redirects try to hack our data or insert some invalid data in our website. This is what we call Cross Site Request Forgery where we are redirect to other websites without our knowledge and hacked our data.
  + To Prevent the attack,
    - Add ***@Html.AntiForgeryToken()*** in the form tag on the View.



* + - Add ***[ValidateAntiForgeryToken]*** attribute to the action method.



* + When we specify @Html.AntiForgeryToken(), a token is created & saved in a hidden field. So, when the form is posted MVC will check for this token, if it didn’t find any then it will not execute the action.
* Authentication
* Authorization

**Dependency Injection in MVC –**

Read it from Microsoft docs - <https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/hands-on-labs/aspnet-mvc-4-dependency-injection>