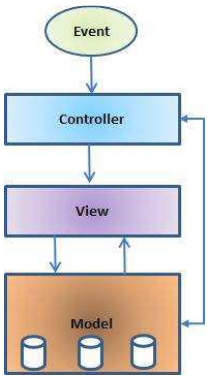
MVC Topics

What is MVC?

- Model View Controller or MVC as it is popularly called, is a software design pattern for developing web applications.
- A Model View Controller pattern is made up of the following three parts:

  Model - The lowest level of the pattern which is responsible for maintaining data.
  View - This is responsible for displaying all or a portion of the data to the user.
  Controller - Software Code that controls the interactions between the Model and View.

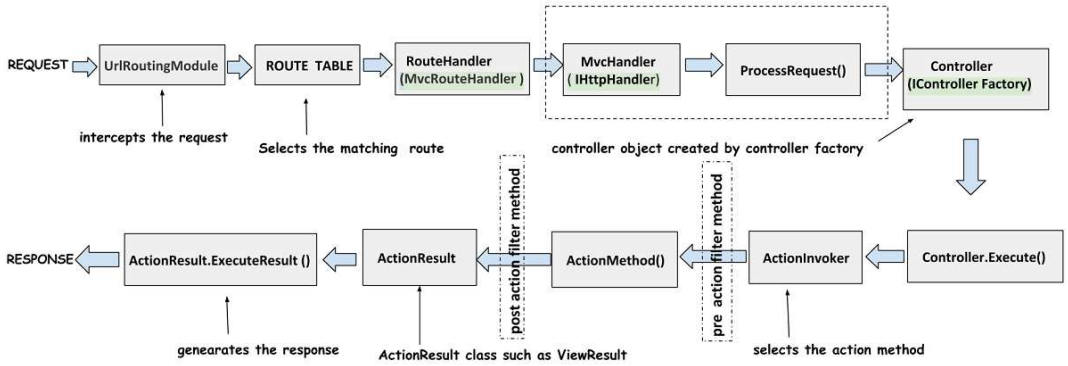


Advantage of ASP.Net MVC over ASP.Net

Below are the points which shows the advantage of MVC over ASP.Net;

- ASP.Net is a View based solution whereas MVC is an Action based solution. As in the case of View based solution, the request first hits view and there after invokes code behind code. The code behind code have a complex page life cycle after which the result is given back to View. But in case of MVC, the request first hits the controller/action which does the Execution and returns the result back to View. This process in MVC is simple and clear.
- ASP.Net code behind classes are tightly coupled with the respective View, thereby the code behind classes cannot be reused by any other views. But in case of MVC, the controller/action can be used across multiple Views making it a reusable component.
- Because of the tightly coupled architecture, the response type is fixed in the webform. It's by default HTML. If you wish to change then you need to play across the content type and response end methods which is quite tedious. On the other hand, MVC has the flexibility to send Json, Html, File, etc. from the action to view.
- Asp.Net code behind are partial classes which cannot be instantiated easily making it difficult to write unit test case against them. And also the webpages inherits from page class making it difficult to create object of the same as it has a lot of dependencies to do so. On the other hand in MVC all components i.e. Model, View & Controller all are inherited through Interface making it easy to mock the Interfaces to write unit test cases.

Architecture of ASP.Net MVC



- UrlRoutingModule is one type of HttpModule which identifies whether the request is a MVC type.
- Route table will have the entries of mapping between urls and the defaults (controller/action).
- RouteHandler is one type of HttpHandler which executes the request.
- ProceesRequest method is executed by RouteHandler to create an instance of IControllerFactory.
- ActionInvoker method selects the action/method that will be executed.
- The Action method will return an ActionResult object (which will return the response back to View).
- `RouteConfig.RegisterRoutes(RouteTable.Routes) ---- Global.asax class`

MVC version

MVC 1.0

- **The concepts of Model, View & Controller in ASP.Net**
- **The concept of routing.**
- **Html Helpers for rendering Html tags.**
- **Automatic binding of posted forms**

MVC 2.0

- **Model validation based on attributes, both client as well as server side.**
- **Areas was introduced.**
- **Html Helpers got improved by having automatic edit forms and display pages.**
- **Asynchronous controllers.**

MVC 3.0

- **Unobtrusive javascript validation.**
- **Razor view engine.**
- **Global filters.**
- **Partial page caching.**
- **ViewBag property is introduced.**
- **“HttpNotFoundResult” is new ActionResult type.**

MVC 4.0(It comes in VS 2010)

- **Bundling and minification**
- **Asynchronous task for action methods.**
- **Enabling logins from facebook & other sites using OAuth & OpenId.**
- **App_Start folder gets created with default files.**
- **Asp.Net Web API introduced in MVC 4.0**

MVC 5.0

- **Attribute routing.**
- **Optional Parameter**
- **Route Constraints**
- **RoutePrefix**
- **FilterOverride**
- **Bootstrap**

Model Binders

Model binder is a concept in MVC where the parameters are retrieved in the action method. There are different ways to do the same which are explained below;

- Automatic Model binders

- Form Collections
- Custom Model Binders

Model Binding

- Model binding is used to retrieve the values in the action.
- Model binding is automatic because the model property matches with field name property.

MVC Controls

- We can use plain html controls
- Also to use mvc controls we need htmlhelper class.

Custom Model Binders

- Create a class and inherit from IModelBinder interface and write the custom logic in BindModel method.

```
public class CustomRegistrationModel:System.Web.Mvc.IModelBinder
    {
        public object BindModel(ControllerContext controllerContext,
System.Web.Mvc.ModelBindingContext bindingContext)
        {
            HttpRequestBase request = controllerContext.HttpContext.Request;
            string userName = request.Form["UserName"];
            if(string.IsNullOrEmpty(userName))
            {
                bindingContext.ModelState.AddModelError("UserName", "Compulsory field");
            }
            return new RegistrationModel { FullName = userName }; // whenever you create
any custom features in mvc we need to do a registration in global.asax
        }

           }
```

- Register the custom model binder class in global.asax as shown below;

```
ModelBinders.Binders.Add(typeof(RegistrationModel), new CustomRegistrationModel())
```

- After registering in global.asax class, we need to decorate the custom model binder in the Action method as below;

```
public ActionResult RegisterUser([ModelBinder(typeof(CustomRegistrationModel))]
RegistrationModel model)
{
}
```

HTML Encoding in MVC

ASP.NET 3.5 and below: <%= Html.Encode (ViewBag.Message) %>

ASP.NET 4: <%: ViewBag.Message %>

Razor View: @ViewBag.Message. It does an automatic encoding.

Html.RenderAction/Html.Action

- These methods are used to call action from the view and output the results of action within the view.
- RenderAction will render the result directly to the response whereas Html.Action returns a string.
- Html.Partial will return an MVCHtmlString.

ChildActionOnlyAttribute

- If any action is decorated for ChildActionOnly attribute then the action cannot be called from URL.

HtmlHelpers

- The final HTML will be generated by these functions at the runtime i.e. we don't have to worry about the correctness of generated HTML.
- Following HTML helpers are built into the ASP.NET MVC framework:

- Html.BeginForm
- Html.EndForm
- Html.TextBox
- Html.TextArea
- Html.Password
- Html.Hidden
- Html.CheckBox
- Html.RadioButton
- Html.DropDownList
- Html.ListBox
- Html.ActionLink("Back","Home")

Creating Custom HTML Helpers

If we want to create our own HTML helpers than that can also be done very easily. There are quite a few ways of creating custom helpers. Let's look at all the possible methods.

1. Creating a static method
2. Writing an extension method
3. Using the @helper(razor only)

Creating a static method

In this approach we can simply create a static class with static method which will return the HTML string to be used at the time of rendering.

```
namespace HTMLHelpersDemo.Helpers
{
    public class MyHTMLHelpers
    {
```

```
        public static IHtmlString LabelWithMark(string content)
        {
            string htmlString = String.Format("<label><mark>{0}</mark></label>", content);
            return new HtmlString(htmlString);
        }
    }
}
```

```
<p>

@MyHTMLHelpers.LabelWithMark ("Using static method")

</p>
```

Writing an extension method

In this approach we will write a simple extension method for the built in html helper class. this will enable us to use the same Html object to use our custom helper method.

```
Public static class MyExtensionMethods
{
    Public static IHtmlString LabelWithMark (this HtmlHelper helper, string content)
    {
        string htmlString = String.Format("<label><mark>{0}</mark></label>", content);
        return new HtmlString(htmlString);
    }
}
```

```
<p>

@Html.LabelWithMark ("Using extension method")

</p>
```

Using the @helper (razor only)

This method is pretty specific to razor view engine. Let us see how this can be done. We need to write this method in the view itself.

```
@helper LabelWithMarkRazor (string content)

{

    <label><mark>@content</mark></label>

}

@LabelWithMarkRazor ("This is created using @helper methods")
```

Razor View Engine

- System.Web.Razor is the namespace for razor view engine.

Single statement block and inline expression

- Each code block will start and end by opening and closing curly brackets {..}, respectively
- A statement is defined in a code block, in other words between opening and closing curly brackets, and end with a semicolon (";")

Example: @{ var message = "Hello,Razor view engine";}

Multi statement block

- In a multiline statement block, we can define multiple code statements and can process data.
- A multiline block will exist between opening and closing curly braces but the opening brace will have the "@" character in the same line.

Example: @{

```
    var priciple = 100;

    var rate = 12;

    var time = 2;

    var interest = (priciple * rate * time) / 100;

}
```

Conditional statements

```
@{
    var isValid = true;
    if(isValid)
    {
        It is an if statement in code block
    }
    else
    {
        It is an else statement in code block
    }
}
```

Looping

- **All loops work the same as in other programming languages, we can define looping inside a code or outside a code block.**
- **We can define a for, do while, or while loop in a code block and use the same syntax for initialization, increment/decrement, and to check a condition.**

```
@{
    for(var count = 1;count,=3;count++)
    {
        @Count is : @count
    }
    string [] nameArray = {"Sandeep","Mandeep","Kuldeep","Pradeep"};
    foreach(var name in nameArray)
    {
```

```
            Your Name is : @name
        }
}
```

Comments

- @* ..*@ for multi line comments.
- // for single line comments.

Creating a User Registration Application using Razor

What are the different types of results in MVC?

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:

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

Can we create our custom view engine using MVC?

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
        return obj; // returned this view paresing
        // logic so that it can be registered in the view engine collection
    }
    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
        return obj;
        // returned this view paresing logic
        // so that it can be registered in the view engine collection
    }
}
```

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.

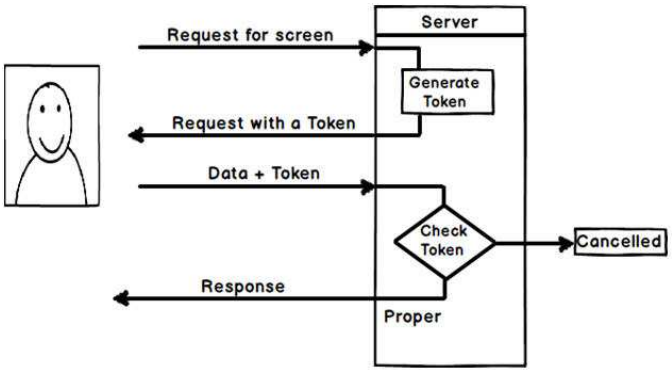
```
protected void Application_Start()
{
    // Step3 :-  register this object in the view engine collection
    ViewEngines.Engines.Add(new MyViewEngineProvider());
    &hellip;..
}
```

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

- Add 2 models in a new model and pass it to the view
- ViewModel concepts.

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

- CSRF stands for Cross site request forgery.



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"]);
}
```

```
<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>
```

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

What is ViewData, ViewBag and TempData

ASP.NET MVC offers us three options - ViewData, ViewBag and TempData for passing data from controller to view and in next request. ViewData and ViewBag are almost similar and TempData performs additional responsibility. Let's discuss or get key points on those three objects.

Similarities between ViewBag & ViewData:

1. **Helps to maintain data when you move from controller to view.**
2. **Used to pass data from controller to corresponding view.**
3. **Short life means value becomes null when redirection occurs. This is because their goal is to provide a way to communicate between controllers and views. It's a communication mechanism within the server call.**

Difference between ViewBag & ViewData:

1. **ViewData is a dictionary of objects that is derived from ViewDataDictionary class and is accessible using strings as keys.**
2. **ViewBag is a dynamic property that takes advantage of the new dynamic features in C# 4.0.**
3. **ViewData requires typecasting for complex data type and check for null values to avoid error.**
4. **ViewBag doesn't require typecasting for complex data type.**

```
public ActionResult Index()
{
   ViewBag.Name = "Hello World";
   return View();
}
public ActionResult Index()
{
   ViewData["Name"] = " Hello World ";
   return View();
}
```

In View:

```
@ViewBag.Name
@ViewData["Name"]
```

TempData:

- TempData is also a dictionary derived from TempDataDictionary class and stored in short lives session and it is a string key and object value.
- TempData keeps the information for the time of an HTTP Request.
- This also works with a 302/303 redirection because it's in the same HTTP Request.
- It requires typecasting for complex data type and check for null values to avoid error.
- It is generally used to store only one time messages like error messages, validation messages.

```
public ActionResult Index()
{
  var model = new Review()
        {
           Body = "Start",
           Rating=5
        };
    TempData["ModelName"] = model;
    return RedirectToAction("About");
}

public ActionResult About()
{
    var model= TempData["ModelName"];
    return View(model);
    return 302();
}
Public ActionResult About1()
{
  var model= TempData["ModelName"];
}
```

Persisting Data with TempData

```
@model MyProject.Models.EmpModel;
@{
 Layout = "~/Views/Shared/_Layout.cshtml";
 ViewBag.Title = "About";
 var tempDataEmployeet = TempData["emp"] as Employee; //need typcasting
 TempData.Keep(); // retains all strings values
}


@model MyProject.Models.EmpModel;
@{
 Layout = "~/Views/Shared/_Layout.cshtml";
 ViewBag.Title = "About";
 var tempDataEmployeet = TempData["emp"] as Employee; //need typcasting
 TempData.Keep("emp"); // retains only "emp" string values
}
```

- http://www.codeproject.com/Articles/826417/Advantages-of-ViewModel-in-MVC-Model-View-Controll

- http://www.dotnet-tricks.com/Tutorial/mvc/9KHW190712-ViewData-vs-ViewBag-vs-TempData-vs-Session.html

- **Session["name"] = "Hello world"**

Creating Simple Grid in MVC Using Grid.MVC

- **http://www.c-sharpcorner.com/UploadFile/4d9083/creating-simple-grid-in-mvc-using-grid-mvc**
- **/**https://gridmvc.codeplex.com/wikipage?title=Custom%20columns.
- `using GridMvc which is installed through Nuggets package.`
- `Using using System.Web.Helpers which is @grid.GetHtml();`
- `Difference between Grid.GetHtml & Html.Grid.(Sorting, Pagination & Filtering)`

MVC 6

- ASP.NET MVC and Web API has been merged in to one.
- Dependency injection is inbuilt and part of MVC.
- Side by side - deploy the runtime and framework with your application
- Everything packaged with NuGet, Including the .NET runtime itself.
- New JSON based project structure.
- No need to recompile for every change. Just hit save and refresh the browser.
- Compilation done with the new Roslyn real-time compiler.
- vNext is Open Source via the .NET Foundation and is taking public contributions.
- vNext (and Rosyln) also runs on Mono, on both Mac and Linux today.

MVC 5

- One ASP.NET
- Attribute based routing
- Asp.Net Identity
- Bootstrap in the MVC template
- Authentication Filters
- Filter overrides

MVC 4

- ASP.NET Web API
- Refreshed and modernized default project templates
- New mobile project template
- Many new features to support mobile apps
- Enhanced support for asynchronous methods

MVC 3

- Razor
- Readymade project templates
- HTML 5 enabled templates
- Support for Multiple View Engines
- JavaScript and Ajax
- Model Validation Improvements

MVC 2

- Client-Side Validation
- Templated Helpers
- Areas
- Asynchronous Controllers
- Html.ValidationSummary Helper Method
- DefaultValueAttribute in Action-Method Parameters
- Binding Binary Data with Model Binders
- DataAnnotations Attributes
- Model-Validator Providers
- New RequireHttpsAttribute Action Filter
- Templated Helpers
- Display Model-Level Errors

Controllers

Async Contollers

- The asynchronous controller enables you to write asynchronous action methods
- It allows you to perform long running operation(s) without making the running thread idle
- During asynchronous call, the server is not blocked from responding to the other requests.
- The difference is that the asynchronous approach will just not keep the action method blocked for the time period it is waiting for the call to the external resource like a network call or a database call to complete.
- **Asynchronous action** methods are useful when an action must perform several independent long running operations.
- Suppose we have three operations which takes **500, 600 and 700** milliseconds. With the synchronous call, total response time would be slightly more than 1800 milliseconds. However, if the calls are made asynchronously (in parallel), total response time would be slightly more than 700 milliseconds.
- A typical use for the AsyncController class is long-running Web service calls.
- Async, Task and await for Asynchronous method.
- Public **async Task**<Employee> GetEmployee(int id) // 1 thread
- {
- Var responseTask = await repo.GetEmp(int id); /// 1 thread
- Var response1 Task= await repo.GetEmp1(int id); /// 2 thread
- If(response!=null && response1!=null)
  {

  }
- `await Task.WhenAll(responseTask, response1Task) // New Task is created when other 2 task is completed.`
- `await Task.WhenAny(responseTask, response1Task) // New Task is created when other 2 task is completed`
- `TaskStatus provides the Task status.`
- `Created, WaitingForActivation, WaitingToRun, Running, WaitingForChildrenToComplete, RanToCompletion.`
- `WaitAll & WhenAll , WaitAny & WhenAny`

```
Public async Task<Employee> GetEmp(int id)
{


}
```

If you observe there are only three differences in the asynchronous and synchronous action methods.

1) In the asynchronous method we are wrapping the return type in a Task while in a normal synchronous method we just return an ActionResult type.

2) The asynchronous method uses the async keyword in its declaration.

3) The asynchronous method uses the await keyword while making a method call that is expected to take a while to complete.

4) [AsyncTimeout(200)]

5) Task.WhenAny and Task.WhenAll methods

Areas(MVC 2.0)

- Areas are logical grouping of Controller, Models and Views and other related folders for a module in MVC applications.
- Areas help us to organize the code.
- Area provides a better way to separate the code for modules in a much elegant way.
- We can keep all the specific resources needed for a module in its Area.
- The reusable and shared resource can be put inside corresponding top level folders in the ASP.NET MVC application.

Action Selectors:

- Action selector is the attribute that can be applied to the action methods.
- It helps routing engine to select the correct action method to handle a particular request. MVC 5 includes the following action selector attributes.
- MVC framework routing engine uses Action Selectors attributes to determine which action method to invoke.
- Three action selectors attributes are available in MVC 5
  - ActionName
  - NonAction
  - ActionVerbs
- ActionName attribute is used to specify different name of action than method name.
- NonAction attribute marks the public method of controller class as non-action method. It cannot be invoked.

Example

```
public class StudentController : Controller
{

    public StudentController()
    {

    }

    [ActionName("find")]
    public ActionResult GetById(int id)
    {
        // get student from the database
        return View();
    }
}
```

http://localhost/student/find/1

```
public class StudentController : Controller
{
    public StudentController()
    {

    }

    [NonAction]
    public Student GetStudnet(int id)
    {
        return studentList.Where(s => s.StudentId == id).FirstOrDefault();
    }
}
```

- ActionVerbs includes HttpGet, HttpPost, HttpPut.

Filters in MVC

- In an ASP.NET MVC application the request from the user first lands at the `UrlRoutingModule.`
- This module parses the requested URL and then invokes the corresponding controller and action
- The controller will then render the appropriate view and the response will be sent to the user.
- Now what if we want to inject some extra processing logic in this request-response life cycle. Some extra logic that is written once and can be reused across multiple controllers and/or action.
- This can be done by using custom filters; There are 5 different types of filter;
  1) Authentication Filter (New in MVC 5.0)
  2) Authorization Filter
  3) Action Filter
  4) Result Filter
  5) Exception Filter

Authorization filter

- This filter provides authorization logic.
- It will be executed before the action gets executed.
- To implement this action the interface `IAuthorizationFilter` should be implemented by the custom attribute class.

```
public class CustomAuthorizationAttribute : FilterAttribute, IAuthorizationFilter
 {
```

```
        void IAuthorizationFilter.OnAuthorization(AuthorizationContext filterContext)
        {
            filterContext.Controller.ViewBag.OnAuthorization =
          "IAuthorizationFilter.OnAuthorization filter called";
        }
    }
```

```
Public Class HomeController: Controller
{
    [CustomAuthorization("Admin")]
    Public ActionResult Index()
    {
    }
}
```

Action filter

- This filter will be called before and after the action starts executing and after the action has executed.
- We can put our custom pre-processing and post-processing logic in this filter.
-

```
public class CustomActionAttribute : FilterAttribute, IActionFilter
{
   void IActionFilter.OnActionExecuted(ActionExecutedContext filterContext)
   {
      filterContext.Controller.ViewBag.OnActionExecuted =
"IActionFilter.OnActionExecuted filter called";
   }
```

```
        void IActionFilter.OnActionExecuting(ActionExecutingContext filterContext)
        {
            filterContext.Controller.ViewBag.OnActionExecuting =
"IActionFilter.OnActionExecuting filter called";
        }
    }
```

Result filter

- This filter will execute before and after the result of the action method has been executed.
- We can use this filter if we want some modification to be done in the action's result.

```
    public class CustomResultAttribute : FilterAttribute, IResultFilter
    {
        void IResultFilter.OnResultExecuted(ResultExecutedContext filterContext)
        {
            filterContext.Controller.ViewBag.OnResultExecuted =
    "IResultFilter.OnResultExecuted filter called";
        }

    void IResultFilter.OnResultExecuting(ResultExecutingContext filterContext)
        {
            filterContext.Controller.ViewBag.OnResultExecuting =
    "IResultFilter.OnResultExecuting filter called";
        }
    }
```

Exception filter

- This filter will be invoked whenever a controller or action of the controller throws an exception.
- This is particularly useful when we need custom error logging module.

Public class CustomExceptionAttribute : FilterAttribute, IExceptionFilter

{

void IExceptionFilter.OnException(ExceptionContext filterContext)

{

filterContext.Controller.ViewBag.OnException = "IExceptionFilter.OnException filter called";

}

}

Below are the order of execution for the filters;

1. `IAuthorizationFilter.OnAuthorization`
2. `IActionFilter.OnActionExecuting`
3. `IActionFilter.OnActionExecuted`
4. `IResultFilter.OnResultExecuting`
5. `IResultFilter.OnResultExecuted`
6. `iExceptionFilter.OnException`

Global Filters

- Global filters are used when you want to apply your filters across all action methods.
- Create a custom filter and add the filter in the global.asax class under the **`RegisterGlobalFilters section.`**

How to not apply global filters for one of my action?

- Create a custom attribute class

```
public class SkipMyGlobalActionFilterAttribute : Attribute
{
  // Empty markup
}
```

- In the global action filter check if the action is decorated with SkipMyGlobalAction attribute then return empty rather executing it.

```
public class MyGlobalActionFilter : ActionFilterAttribute
{
    public override void OnActionExecuting(ActionExecutingContext
filterContext)
    {
        if (
```

```
filterContext.ActionDescriptor.GetCustomAttributes(typeof(SkipMyGlobalActionFi
lterAttribute), false).Any())
        {
            return;
        }

        // here do whatever you were intending to do
    }
}
```

- Then decorate the Action with the SkipMyGlobalActionFilter attribute, which will make sure the global action wont execute for it.

```
[SkipMyGlobalActionFilter]
public ActionResult Index()
{
    return View();
}
```

- MVC 5.0 has a new feature called Override action which will make sure I can skip my action to not run any global filters or any controller level filters to be excuted.

```
[OverrideMyGlobalAction]
public ActionResult Index()
{
    return View();
}
```

Layout in View

`@RenderSection & @RenderBody()`

`Difference between @RenderPage & @Html.Partial`

Bundling & Minification in MVC 4.0

Dependency Injection (http://www.codeproject.com/Articles/615139/An-Absolute-Beginners-Tutorial-on-Dependency-Inver)

- I have an object which is dependent on another object.
- Unity, StructureMap (Go through this DI tools).

Dependency Inversion Principle

- Dependency inversion principle is a software design principle which provides us the guidelines to write loosely coupled classes
- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend upon details. Details should depend upon abstractions.

```
class EventLogWriter
{
   public void Write(string message)
   {
      //Write to event log here
   }
}

class AppPoolWatcher
{
   // Handle to EventLog writer to write to the logs
   EventLogWriter writer = null;

   // This function will be called when the app pool has problem
   public void Notify(string message)
   {
      if (writer == null)
      {
         writer = new EventLogWriter();
      }
      writer.Write(message);
   }
}
```

- Now this design becomes problematic when new requirement are added;

```
class EmailSend
{
   public void Write(string message)
   {
      //Write to event log here
   }
}
```

Dependency Injection Container

IOC

Public Interface INotiifcation

{

public void DoOperation();

}

Public Void EventWRitre: INotifcation

{

}

Public Void EmailSend: INotification

{

}

1)Constructor

2)Methods

3)Properties

Public Void AppPoolWatcher

{

private Interface _iNotification;

public AppPoolWatcher(INotification iNotifiction)

{

```
  _iNotification =  iNotifiction)
}

public void AssignDI(INotification iNotifiction)
{
 _iNotification =  iNotifiction;
}
_iNotification.DoOperation();

public INotitifiation AssignNotification
{
get
{
  return _iNotification;
}
set
{
  _iNotification = value;
}
}

}



public void main()
{
  INotification not = new EventWrite();
  INotification notEmail = new EmailSend();
  AppPoolWatcher = new ApppoolWatcher(not);
}
```

Dependency Injection Container

unity, structuremap

{

INotification, EventWire;

INotification, EmailSend;

}