MVC 6

NEW CLASS NOTES

BY

MR.SUDHAKAR SHARMA SIR

NARESH TECHNOLOGIES

SRI RAGHAVENDRA XEROX

SOFTWARE LANGUAGES MATERIAL AVAILABLE

BESIDE BANGLORE IYYENGARS BAKERY,OPP.CDAC,AMEERPET,HYDERABAD

CELL:9951596199

MVC-Starting classes upto

Design Patterns:

Design patterns are solutions to software design problems you find again and again in real-world application development. Patterns are about reusable designs and interactions of objects.

-> A design patterns is not a finished design that can be transformed directly into code. It is a description (or) template for how to solve a problem that can be used in many different situations.

-> Design patterns can speedup the development process by providing tested, proven development paradigms. Effective Software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns help to prevent suitable issues that can cause major problems and improves code readability for coders and architects familiar with the patterns.

-> The 23 Gang of four (GoF) patterns are generally considered the foundation for all other patterns. They are categorized in three groups.

(1) Creational
(2) Structural
(3) Behavioral

Creational:

These design patterns are all about class instantiation This pattern can be further divided into class creation patterns and object creational patterns. While class creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.

(1) Abstract-factory: Creates an instance of several families of
(2) Builder: Separates object construction from its representation
(3) Factory Method: Creates an instance of several derived classes
(4) ProtoType: A fully initialized instance to be copied or cloned.
(5) SingleTon: A class of which only a single instance can be exit.

(2) Structural Patterns:

These design patterns concern class and object composition Concept of inheritance is used to compose: interfaces and define ways to compose objects to obtain new functionalities.

(i) Adapter: Match interfaces of different classes
(ii) Bridge: Separates an objects interface from its implementation
(iii) Composite: A tree structure of simple and composite objects
(iv) Decorator: Add responsibilities to objects dynamically
(v) facade: A single class that represents an entire Sub System.
(vi) FlyWeight: A fine-grained instance used for efficient sharing.
(vii) Proxy: An object representing and their object.

(3) Behavioral Patterns:

These design patterns are specifically concerned with communication between objects. By doing so, these patterns increase flexibility in carrying out this communication.

(i) chain of Response: A way of passing a request between a chain of object
(ii) Command: Encapsulated a command request as an object.
(iii) Interpreter: A way to include language elements in a program.
(iv) Iterator: Sequentially access the elements of a collection.
(v) Mediator: Defines Simplified communication between classes
(vi) Memento: Capture and restore an objects internal state.
(vii) Observer: A way of notifying change to a number of classes.
(viii) State: Alter an objects behaviour when its state changes
(ix) Strategy: Encapsulates an algorithm inside a class
(x) Template Method: Defer the exact steps of an algorithm to a Sub class
(xi) Visitor: Defines a new operation to a class without change.

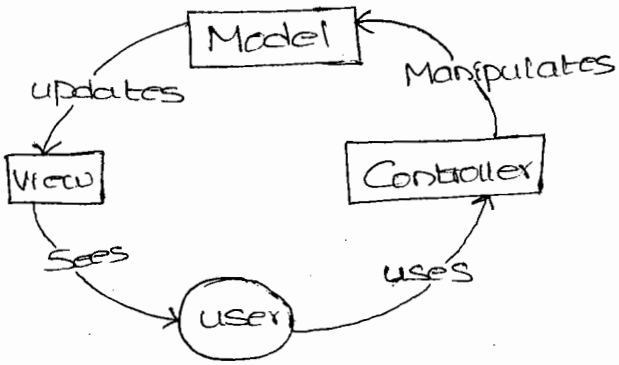
Architectural Patterns:

An architectural style, sometimes called an architectural patterns is a set of principles - a course grained pattern that provides an abstract frame work for a family of systems. An architectural style improves partitionary and promotes design reuse by providing solutions to frequently recurring problems.

MVC:

The Model-View-Controller (MVC) architectural patterns separates an application into three main components.

(1) Model
(2) View
(3) Controller



→ Model-View-Controller (MVC) is a software architecture pattern.
→ Originally formulated in the late 1970 by Trygve Reenskang as a part of small talk.
→ Code reusability and separation of concerns.
→ Originally developed for desktop, then adapted for internet app's.

Model:

* Set of classes that describes the data we are working with as well as the business.
* Rules for how the data can be changed and manipulated.
* May contain data validation rules
* Often encapsulate data stored in a database as well as code used to manipulate the data.
* Most likely a Data Access Layer of some kind.
* Apart from giving the data objects, it doesn't have significance in the framework.

View:

* Defines how the applications user interface (UI) will be displayed.
* May Support master views (layouts) and sub-views (partial views or controls)
* Web: Template to dynamically generate HTML.

Controller:

* The core MVC component
* The process the requests with the help of views and models.
* A set of classes that handles
  (1) Communication from the user
  (2) Overall application flow
  (3) Application-Sep Specific logic
* Every controller has one (or) more actions

MVC Frameworks:

PHP - Cake PHP, Code igniter

Java - Spring

Perl - Catalyst, Dancer

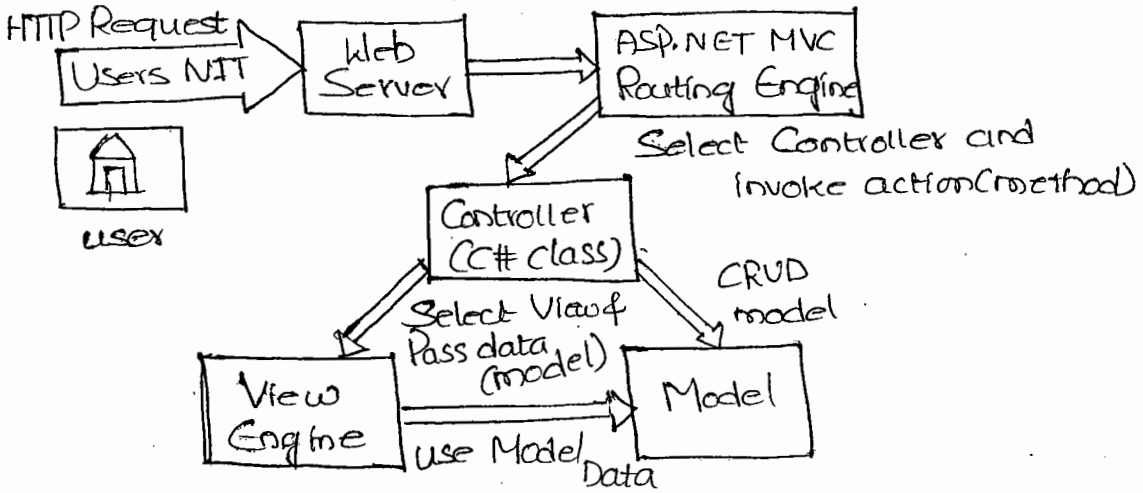
Python - Django, Flask, Grok

Ruby - Ruby on Rails, Camping, Nitro, Sinatra

JavaScript - Engular JS, JavaScript MVC, Spine

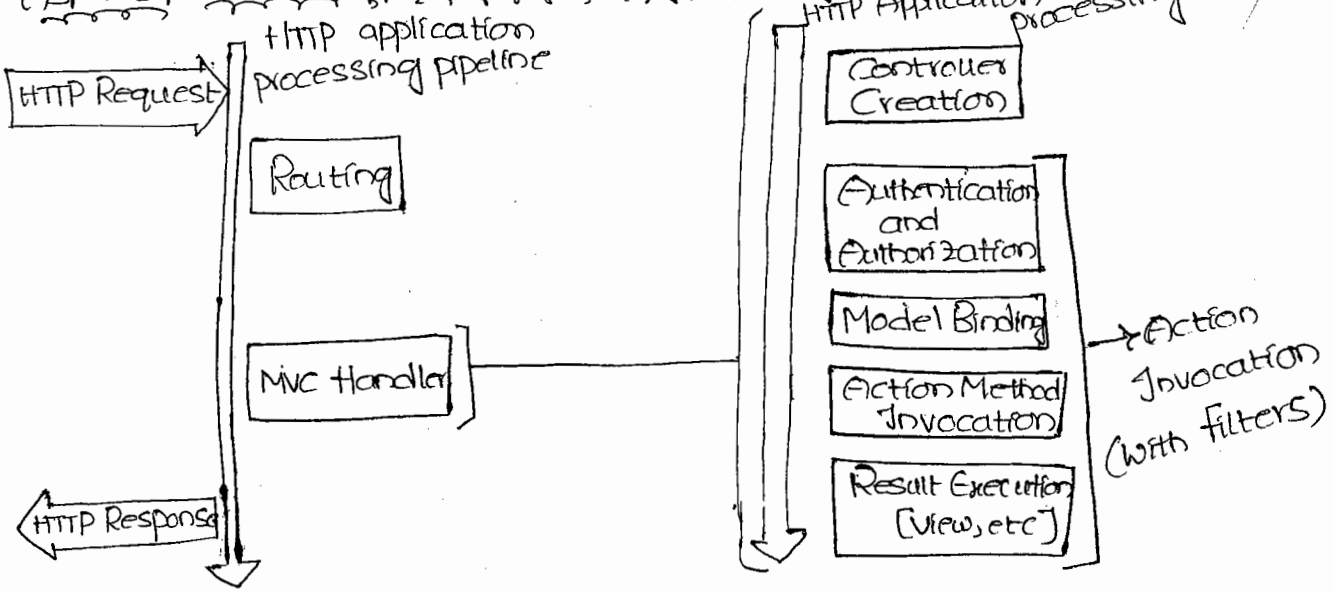
.NET Framework - ASP.NET MVC

The MVC Pattern for Web:



* Incoming request routed to Controller
  (1) for Web: HTTP Request
* Controller processes request and creates presentation model
  (1) Controller also Selects appropriate result (View)
* Model is passed to View.
* View transforms model into appropriate output format (HTML)
* Response is rendered (HTTP Response)

ASP.NET MVC5 Application Life Cycle - High level View:



Stage Details

(1) Recieve first Request for the application: In the global.asax file, Route objects are added to the Route table object.

(2) Perform Routing: The Url Routing module uses the first matching Route object in the Route table collection to create the Route object, which it then uses to create a request context (IHTTP Context) object.

(3) Create MVC request handler: The MVC RouteHandler object creates an instance of the MVC handler class and passes it the Request Context instance.

(4) Execute Controller: The MVC Handler instance calls the controller's Execute method.

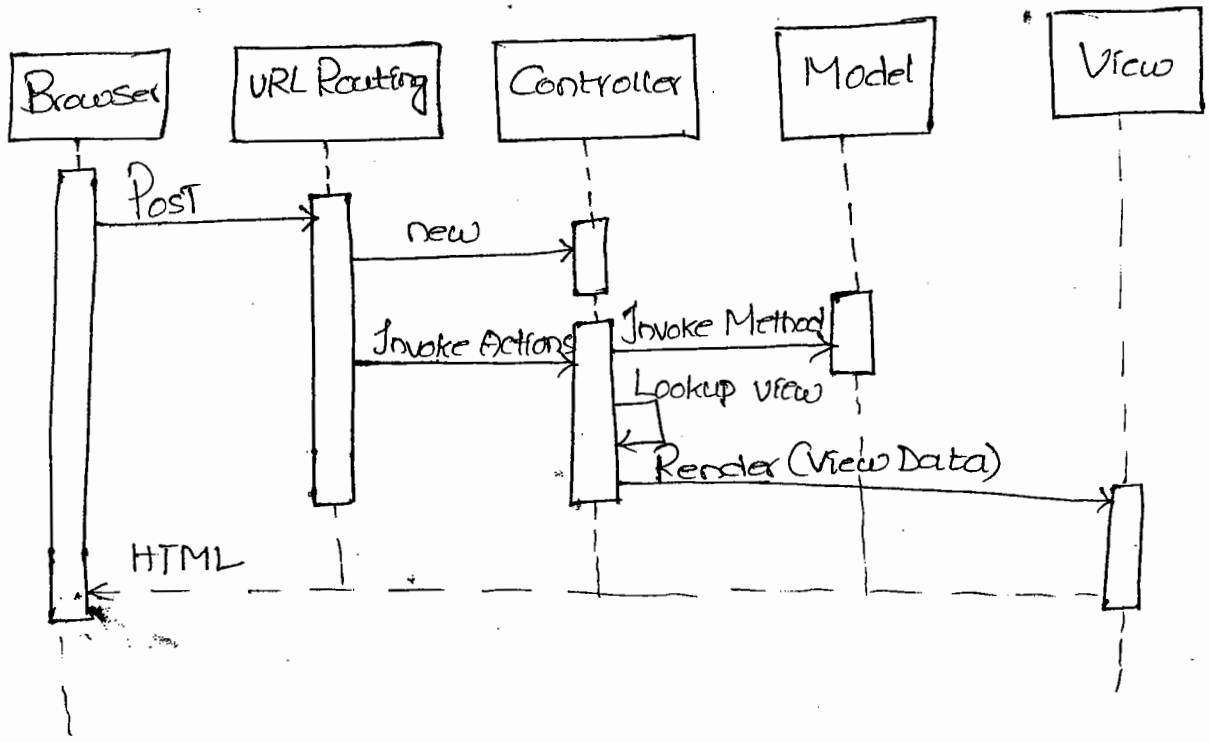
(5) Invoke Action:

Most controllers inherit from the controller base class for controllers that do so, the controller action invokes object that is associated with the controller determines which action method of the controller class to call and then calls that method.

(6) Execute Result: A typical action method might recieve user input, prepare the appropriate response data, and then the executed the result by returning a result type. The built in result types that can be executed include the following

(i) ViewResult (which renders a view and is most often used result Type), RedirectToRoute-Result, RedirectResult, ContentResult, JsonResult and EmptyResult.

Request flow:



Difference Between ASP.NET Web forms and ASP.NET MVC:

ASP.NET Web forms	ASP.NET MVC
1. ASP.NET web forms fallow a traditional even driven development model.	1. ASP.NET MVC is a light weight and fallow MVC (model, view and controller) patterns based development model.
2. ASP.NET web form has Server Controls	2. ASP.NET MVC has HTML Helpers
3. ASP.NET webform Supports View state for state management at client Side	3. ASP.NET MVC does not support View state
4. ASP.NET web form has file based URIS means files name exist in the URIS must have its physically existence.	4. ASP.NET MVC has route-based URIS means URIS are divided into Controllers and actions and moreover it is based on controller not on physical file
5. ASP.NET web form fallows web forms Syntax.	5. ASP.NET MVC fallow customizable Syntax (Razor as default)
6. In ASP.NET web form, webforms (ASPX) i.e views are tightly Coupled to code behind (ASPX.CS) i.e logic	6. In ASP.NET MVC, has ~~layouts for consistent look and feels~~. Views and logic are kept Separately.
7. ASP.NET web form has master pages for Consistent look	7. ASP.NET MVC, has layouts for consistent look and feels.
8. It has user controls for code reusability.	8. It has partial views for code reusability.
9. It has built in data controls and best for rapid development with powerful data access.	9. It is light weight provide full control over markup and support many features that allow fast and agile development. Hence it is best for developing interactive web application with latest web standards.
10. ASP.NET web form is not an open Source.	10. ASP.NET web MVC is an open source

ASP.NET MVC features:

* Runs on top of ASP.NET
    (1) Not a replacement for web forms
    (2) Leverage the benefits of ASP.NET
* Embrace the web
    (1) User | SEO friendly URIS, HTML5, SPA
    (2) Adopt Rest concepts.
* Uses MVC pattern
    (1) Conventions and Guidance
    (2) Separation of Concerns
* Tight Control over markup
* Testable
* Loosely Coupled and extensible
* Convention over Configuration
* Razor View Engine
    (1) One of the greatest View Engines.
    (2) With intellisense, integrated in Visual Studio
* Resuse of current skills (C#, LINQ, HTML etc--)
* Application-based (not scripts like php)

Separation of Concerns:

* Each component has one responsibility.
    (1) SRP - Single Responsibility principle
    (2) DRY - Don't Repeat yourself
* More easily testable
    (1) TDD - Test-Driven Development
* Helps with Concurrent development
    (1) Performing tasks concurrently and one developer works on view and another work on controller.

Extensible:

* Replace any component of the system
    * Interface-based architecture
* Almost anything can be replaced (or) extended
    (i) Model binders (Request data to CLR objects)
    (ii) Action Result filters (eg: ON ActionExecuting)
    (iii) Custom action result Types
    (iv) View Engine (Razor, webforms, NHaml, Spark)
    (v) View helpers (HTML, Ajax, URL etc)
    (vi) Custom data providers (ADO.NET) etc

Clean URL's:

→ REST-Like
    (i) /Products/update
    (ii) /blog/posts/2014/11/28 (MVC-is-cool)

→ Friendlier to humans
    (i) /Products.aspx?catId=123 (or) post.php?id=123
    (ii) Becomes/products/chocolate/

→ Friendlier to web crawlers
    (i) Search engine optimization (SEO)

ASP.NET MVC6 Leaner, faster:

* MVC6 has no dependency on System.Web.dll. The result is a leaner framework, with faster startup time and lower memory consumption.
* VNext apps can use a cloud-optimized runtime and subset of the .NET framework. This subset of the framework is about 11 megabytes in size compared to 200 megabytes for the full framework, and is composed of a collection of NuGet packages.
* Because the cloud-optimized framework is a collection of NuGet packages, your app can include only the packages you actually needs. No unnecessary memory, diskspace, loading time etc.

* Microsoft can deliver updates to the framework on a faster cadence, because each part can be updated independently.

True Side by Side Deployment!

→ The reduced foot print of the cloud-optimized runtime makesit practical runtime makes it practical to deploy the framework with your app.

→ You can run apps side by side with different versions of the framework on the same server

→ Your can makes frameworks updates for each app on its own schedule.

→ No errors when you deploy to production resulting from a mismatch between the framework patch level on the development machine and the production server.

~~Access Deploy~~

New Development Experience:,

→ VNext uses the Roslyn compiler to compile code dynamically

→ You can edit a code file, refresh the browser and see the changes without rebuilding the project

→ Besides streamlining the development process, dynamic code compilation enables development senarios that were not possible before such as editing code on the server using visual studio online ("monaco")

→ You can choose your own editors and tools.

ASP.NET MVC Release History:

Date	Version
10th Dec 2007	ASP.NET MVC
13th March 2009	ASP.NET MVC1.0
16th Dec 2009	ASP.NET MVC2RC
4th Feb 2010	ASP.NET MVC2RC2
10th March 2010	ASP.NET MVC2
6th Oct 2010	ASP.NET MVC3 Beta
9 Nov 2010	ASP.NET MVC3RC
10th Dec 2010	ASP.NET MVC3RC2
13th Jan 2011	ASP.NET MVC3
20th Sep 2011	ASP.NET MVC4 Developer Preview
15th Feb 2012	ASP.NET MVC4 Beta
31 May 2012	ASP.NET MVC4RC
15th Aug 2012	ASP.NET MVC4
30th May 2013	ASP.NET MVC4.0.30506.0
26th June 2013	ASP.NET MVC5 Preview
23rd Aug 2013	ASP.NET MVC5RC1
17th Oct 2013	ASP.NET MVC5
10th Feb 2014	ASP.NET MVC5.1.1
17th Jan 2014	ASP.NET MVC5.1
4th April 2014	ASP.NET MVC5.1.2
22nd June 2014	ASP.NET MVC5.1.3
1st July 2014	ASP.NET MVC5.2.0
28th Aug 2014	ASP.NET MVC5.2.2

28/12/15

MVC

Design Patterns:

The design patterns are solutions for software design problem that your find in real world application development.

* Patterns are about reusable designs and interaction of objects.

The 23GoF(gang of 4) patterns are generally considered as the foundation for all others patterns

* They are categorized into 3 groups
  * Creational
  * Structural
  * Behavioral

Creational:

Deal with instantiation i.e creating of object. The popular creations pattern is Abstract factory, Builder, factory method, prototype, Singleton.

structural:

The structural pattern are about designing of class and describe various implementation mechanism. They are Adaptor, Bridge, Composite, Decorator, facade, flyweight, proxy.

Behavioral Pattern:

Define the scope of object and specifies how the consumes the resources. They are chain of Resp, command, interpreter, iterator, mediator, memento, observer, State, strategy, Template, method, visitor.

29/12/15

MVC

(Model View Controller)

* MVC is an software architecture pattern.
* Introduced by Trygve in 1970 and formulated with the language "small Talk".
* Code separation and code reusability of concerns.
* Originally designed for desktop application.
* Now being adopted by web application.

Technologies using MVC framework:

Technologies	MVC framework
PHP	cake PHP, code igniter
Java	Spring
Perl	Catalyst, Dancer
Python	Django, flask Grok
Ruby	Ruby on Rails
Java Script	spine, Angular js, Back Bone js
.NET	ASP.NET MVC

Evolution of microsoft Server Side Technologies:

History of ASP (18 years):

1996 - Active Server Pages (ASP)

2002 - ASP.Net

2008 - ASP.Net MVC

2010 - ASP.Net web Pages

2012 - ASP.Net web API, Signal R

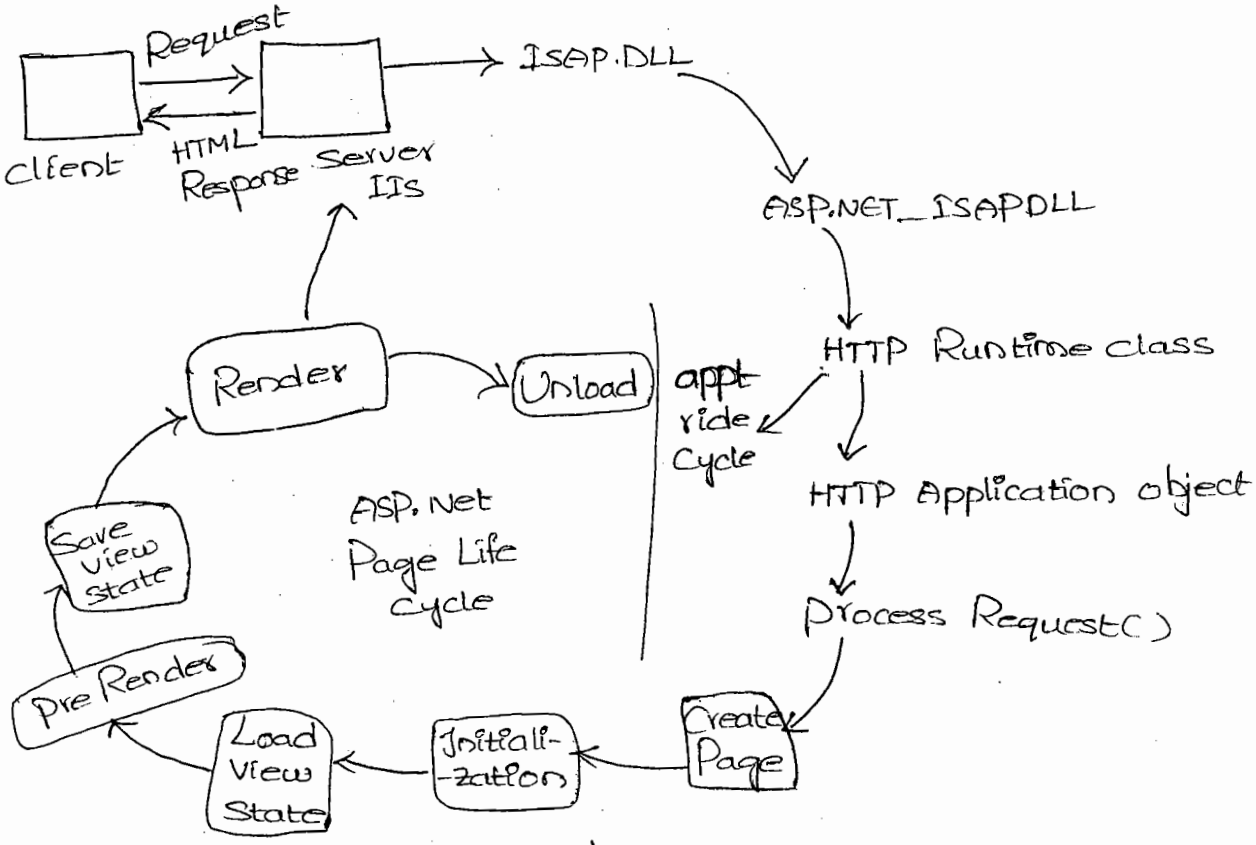
2014 - ASP.NET 5

Note:

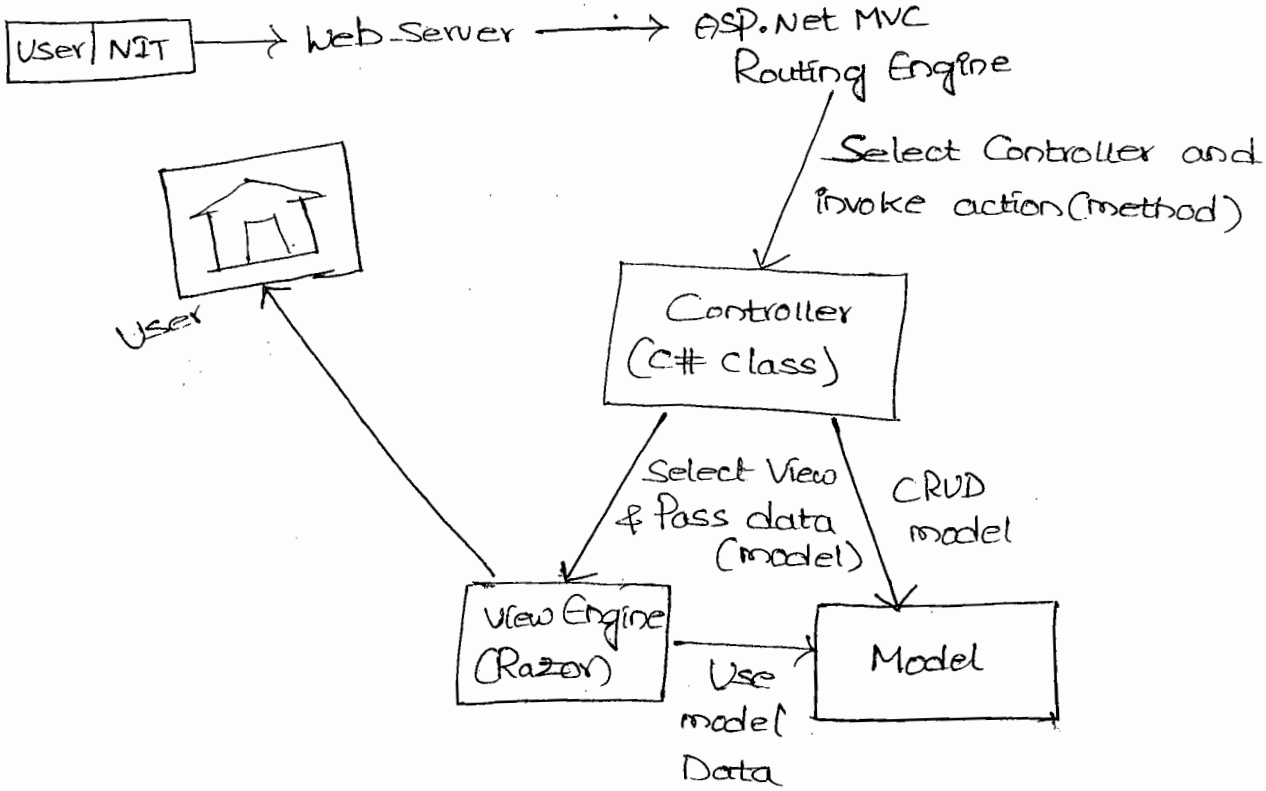
Asp.net MVC is just an alternative to asp.net webforms and not replacement for webforms.

ASP.NET Web-forms Vs ASP.NET MVC:

ASP.Net Webforms:



ASP.NET MVC (Pattern for web):



Model:

1. A set of classes that describe the data we are working with as well as the business
2. It contains routes that define how data can be changed and manipulated.
3. It may contain data validation routes.
4. It often encapsulate data stored in a database as well as code used to manipulate the data.
5. It is most likely a data access layer.
6. Apart from giving data objects it doesn't have any significance in the framework.

View:

1. It describes the application user interface.
2. It supports master views (layouts) and Sub views (partial views or User Controls)
3. In web view provides Template to dynamically generate HTML. The commonly used view Engines with MVC are:

   Razor → @ Name

   ASPX → <%= Name %>

   Spark → ${Name}

   Nhaml → @{% Name}

   Django → {% Name %}

Controller:

* It is the Core MVC Component
* It process the request with the help of views and Models.
* It is a set of classes that handles
  * Communication from the user
  * Overall application flow
  * Application-Specific logic
* A controller Contains methods that respond to various http request. Hence these methods are known as "Action Methods".

ASP.NET Web-form	ASP.NET MVC
1. Asp webform uses a page controller pattern, where every page will have its own controller.	1. MVC uses front-end Controller pattern, where all pages will use a common controller.
2. It uses an application and page life cycle to Send Response	2. No more page life cycle, only Request Cycle.
3. Provides a huge control library and requires lots of Server Side interactions and View State.	3. It is completly lightweight as it leverages the benefits of jquery and Ajax.
4. Tightly Coupled and hard to test	4. Loosely Coupled and Supports test driven development.
5. Will not Support Completly HTML.	5. Will Support Complete HTML
6. It is RAD (Rapid Appli. Develop)	6. Not RAD.

What's new in MVC4? (ASP.NET Q.5)

* Bundling and Minification
* BootStrap, Web API, Signal R

What's new in MVC 5)

1. Filter Overriding
2. Attribute Routing
3. Unobtrusive jquery Validations With Remote Validation
4. Support for BootStrap in Editor Templates
5. WEB API2, Signal R2
6. Identity - Open ID
7. New Templates - Face book,
   Twitter,
   API

What's new in MVC6?

1. Webforms + API + MVC = ASP5

30/12/15 What's new in MVC4?

1. ASP.NET Web API
2. Refreshed and modernized default project templates.
3. New mobile project template.
4. Many new features to support mobile apps
5. Enhanced support for asynchronous methods
6. Bundling and Minification
7. Routing Improvements
8. Bootstrap
9. Signal R
10. SPA (Single Page Applications)

What's new in MVC5?

1. Attribute Routing Improvements
2. Bootstrap support for editor templates.
3. Enum Support in Views
4. Unobtrusive validation for MinLength | MaxLength Attributes
5. Supporting the "this" context in Unobtrusive Ajax.
6. filter Overriding
7. Web API2, Signal R2
8. Identity - Open ID

What's new in MVC6?

1. ASP webforms + API + MVC = MVC6 (ASP5 - vNext)
2. Modular: framework ships with application
3. faster Development Cycle
   - Same code runs on Development and Production
4. Open Source with Contributions
5. CrossPlatform
   - New framework for Linux, MAC

6. Agile (It is a method)
   - Uses Monaco, which is online visual studio
   - Azure
7. Cloud Ready
   - On Permises to Cloud
8. New Rosyln Compiler
9. True Side by Side Development
   - Side by Side Execution
10. Support for Multiple Servers.
11. Inbuilt - Support for Dependency Injection.
   - BOWER - Grunt
   - NPM.
   - NuGet
   - Github
   - Growl
12. Every feature ships like a package
13. OWIN Abstraction, Odata
14. Tag Helpers for MVC

Controller:

Controller Base → System.Web.MVC

1. Must be public
2. Can't be static
3. Must have a return type
4. It Can be parameterized or parameter less
5. Can't have ref and out params
6. Can't be generic types
7. Can't be extension methods
8. Can't override
9. Can overload
10. Can't be any method of Controller base
11. Can't be marked with obsolete or NonAction Methods
    ↓
    you Can't use No longer

3/12/15 Creating a new MVC Application:

1. Open Visual Studio 2013
2. Select "New Project".
3. Language as "Visual C#"
4. Goto "Web" Category and Select "ASP.NET Web Appli."
5. Specify a name and location for App
6. Then click OK.
7. This will prompt you to Select a template
8. Select "MVC" template then click Ok."
9. The basic file system of any MVC application comprises of following components.

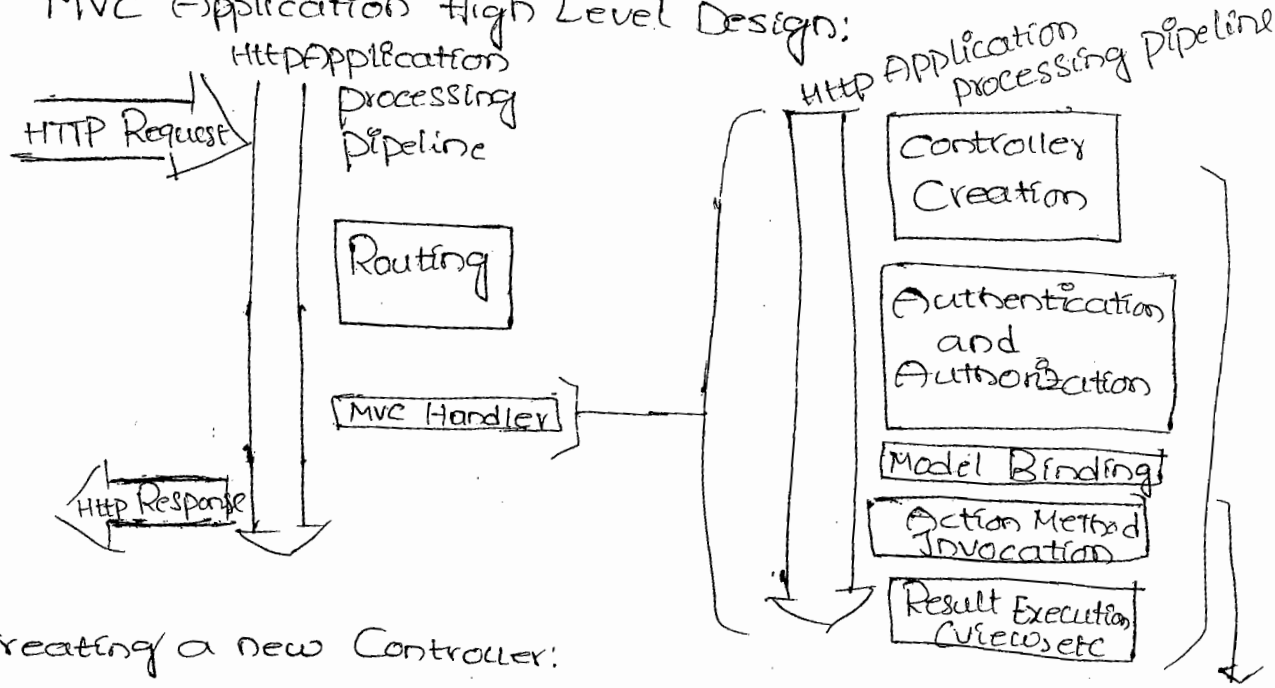
File/Folder	Description
App_Data	Contains local database files (.mdf, .Sdf)
App_Start	Contains classes that are intended to run on application startup - FilterConfig.cs - RouteConfig.cs - BundleConfig.cs - StartupAuth.cs
Content	Contains non dynamic files like css, images etc..
Controllers	Contains Controller classes
Fonts	Collection of fonts used by bootstrap
Models	Contains classes that are responsible for interaction with database
Views	Contains application UI - Views - PartialViews - Layouts

Scripts → contains all dynamic files

Global.asax → Global Application class file.

Web.Config → Application Configuration file

MVC Application High Level Design:



Creating a new Controller:

A controller is a class derived from the base Controller defined under "System.Web.MVC"

It Comprises of methods that respond to various Http verbs: GET, POST, PUT, DELETE etc

1. Right click on "Controllers" folder in MVC application
2. Select "Add → New → Controller"
3. Select controller type as "MVC5 Empty Controller"
4. Click "Add"
5. This will prompt you to define a name for Controller, and every controller name must use the Suffix "Controller" Ex: ProductsController
6. Then click OK.

Syntax:

```
public Class ProductsController : Controller
{
  - action Methods
}
```

Action Methods:

The actionMethods are controller Methods that return various values as a result. The following Conventions to be followed in order to create a controller method.

1. A controller action Method must be public.
2. It can't be static.
3. It must be defined with a return type, as every action method must return a value (can't be void).
4. Can be parameterized or parameterless.
5. Can't have ref or out parameters.
6. Can't be generic types.
7. Can overload but can't be override.
8. It can't be an extension method.
9. It can't be any method of base "Controller" class.
10. Can't be marked with "Obsolete | NonAction" Attributes.

Ex:

```
public class ProductsController: Controller
{
public string Details(int? id, string Name, double? Price)
 {
 return "Product ID:" + id + "<br>" + "Name:" + Name +
    "<br>" + "Price:" + Request.QueryString["Price"];
 }
}
```

Set your action in startup:

1. Goto "App_Start" folder
2. Open "RouteConfig.cs".

```
routes.MapRoute(name: "Default", url: "{controller}/
                                {action}/{id}",
defaults: new {controller = "Products", action = "Details",
);      id = UrlParameter.Optional}
```

Calling the Controller action:

http://localhost/demomvc/products/details/1?Name=TV&Price=45000

1st Parameter, 2nd Par, 3rd

Hosting MVC application on IIS:

1. Right click on Project name in Solution Explorer
2. Select "Properties"
3. Goto "Web" Category
4. Select Server as "Local IIS"
5. Click "Create Virtual Directory"

11/01/16 ActionResults in MVC:

A controller actionmethod can be defined with various return types. An ActionResult encapsulate the result of an actionmethod and used to perform a framework level operation on behalf of the actionmethod like returning a view, file, Json, JavaScript etc.

All actionresults in MVC are derived from the base "ActionResult" which is defined under System.Web.Mvc

ActionResult	Helper Method
ViewResult	View()
PartialViewResult	PartialView()
FileResult	File()
JsonResult	Json()
ContentResult	Content()
JavaScriptResult	JavaScript()
RedirectResult	Redirect()
RedirectToRouteResult	RedirectToAction()

ViewResult:

It represents a class that is used to render a view whenever the controller action is invoked the View Engine dynamically rendered HTML to client.

A controller action can return a view of any one of the following types.

- .aspx
- .ascx
- .cshtml
- .vbhtml

Ex: 1. Create a new action method in ProductsController

```
public ViewResult Details()
{
    return View();
}
```

2. Right click on Action name and Select "Add View"

View Name : Details

Template : Empty (without model)

Select the checkbox : Use Master Layout

Details.cshtml

```
<h1>Product Details </h1>
```

Note:

A controller action method can have more than one type of view. However the 1st priority is given to "aspx".

In order to access a specific view you have to mention the view name.

```
public ViewResult Details()
{
    return View("~/Views/Products/Details.cshtml");
}
```

Passing data from a controller to View:

MVC provides several dynamic expressions and properties that allow the UI to store values and transport them across multiple tiers.

System.Web.Mvc provides the following dynamic expressions.

1. ViewBag 2. ViewData

Ex:

1. Add a new action method into ProductsController.

```
public ViewResult Details()
{
List<string> users = new List<string>()
    {
     "John",
     "David",
     "Rahul"
    };
ViewBag.users = users;
  List<string> products = new List<string>()
       {
        "Mobile",
        "LCD TV",
        "Nike Shoes"
       };
    ViewData["Prods"] = products;
       return View();
    }
```

2. Add view for Details Action

Name: Details

Template: Empty(without model)

Details.cshtml

```
<h1>Users List</h1>
<ol>
@foreach(var item in ViewBag.users)
  {
   <li>@item</li>
  }
</ol>
<h1>Products List</h1>
<ul>
 @foreach(var item in (List<string>) ViewData["prods"])
  {
   <li>@item</li>
  }
</ul>
```

02/1/16 Creating strongly typed Model Views:

1. Go to Models folder in MVC application and add a new class file "Products.cs"

```
public class Product
  {
   public int ProductID {get; set;}
   public string Name {get; set;}
   public double Price {get; set;}
  }
```

2. Add another class file into models by name "ProductsData.cs"

```
public class ProductsData
 {
  List<Product> products=new List<Product>()
   {
```

```
        new Products{ProductID=1, Name="Mobile", Price=12000},
//      new Products{ProductID=2, Name="LED TV", Price=45000},
        new Products{ProductID=3, Name="Shoe", Price=7000},
        };
    public IEnumerable<Product> productlist
        {
            get
            {
                return products;
            }
        }
    }
```

3. Add a new controller by name "ProductsController"

4. Add following actions into controller

```
using MVCDemo.Models;
public class ProductsController : Controller
 {
   ProductsData db=new ProductsData();
    public ViewResult Index()
      {
        return View(db.productslist.ToList());
      }
    public ViewResult Details(int id)   // Renders a view to the response
    {
   return View(db.productslist.Single(x=>x.ProductID==
                                      id));
    }
 }
```

4. Add a View for Index action

Name: Index

Template: Empty

Model class: Product(Models)

Index.cshtml

```
@model IEnumerable<MVCApp.Models.Product>
<h2>Products Index</h2>
<table border="1">
    <th>ProductID</th>
    <th>Product Name</th>
    <th>Product Price</th>
    <th>Actions</th>
@foreach(var item in Model)
{
    <tr>
      <td>@item.ProductID</td>
      <td>@item.Name</td>
      <td>@item.Price</td>
    <td>@Html.ActionLink("Details", "Details",
    </tr>                    new{id=item.ProductID})</td>
}
</table>
```

HyperLink
ActionLink("LinkText", "ActionName", "ControllerName", new{ID=Item.ProductID})

5. Add a View for Details Action

Name: Details

Template: Empty

Model Class: Product(Models)

Details.cshtml

```
@model MVCApp.Models.Product
<h2>Product Details</h2>
<table border="1">
  <tr>
```

```
<td>ProductID</td>
<td>@Model.ProductID</td>
</tr>
<tr>
<td>Product Name</td>
<td>@Model.Name</td>
</tr>
<tr>
<td>Product Price</td>
<td>@Model.Price</td>
</tr>
</table>
<br/>
@Html.ActionLink("Goto Index", "Index")
```

@Html.Partial - uses roundtrip

II. PartialViewResult:

It represents a base class i.e is used to send partial view to the response. The partial views are reusable Prototypes that are accessible from any view. They are similar to web user control in asp webform

Ex:

1. Goto Products.cs and add a new field.

```
public string Photo {get; set;}
```

Koala.jpg
Lighthouse.jpg
Penguins.jpg
Tulips.jpg

2. Goto "ProductsData.cs" and photo for every product.

```
new Product {ProductID=1, Name="Mobile", Price=12000,
             photo="~/Photos/mobile.jpg"},
```

3. Add a new folder name import images into the folder

4. Goto ProductsController and add following actions

```
public PartialViewResult ProtoType()
    {
      return PartialView();
    }
public ViewResult ImageIndex()
   {
   return View(db.ProductsList.ToList());
    }
```

5. Add View for ProtoType

Name : ProtoType

Template : Empty

Model Class : Product (Models)

☑ Create as Partial View

ProtoType.cshtml

```
@model MvcApp.Models.Product
<table border="1" width="600">
    <tr>
   <td><img src="@Url.Content(Model.Photo)"
                width="200" height="200" /></td>
   <td>
   <table border="1">
      <tr>
      <td>Product ID</td>
      <td>@Model.ProductID</td>
       </tr>
       <tr>
       <td>Product Name</td>
       <td>@Model.Name</td>
       </tr>
```

```
<tr>
<td> Product Price </td>
<td>@Model.Price </td>
</tr>
<table>
<td>
</tr>
</table>
```

6. Add view for ImageIndex

Name : ImageIndex

Template : Empty

Model class : Product (Models)

ImageIndex.cshtml

```
@model IEnumerable<MvcApp.Models.Product>
<h2>Products Image Index</h2>
@foreach(var item in Model)
{
// @Html.Partial("Prototype", item)
@Html.RenderPartial("Prototype", item);
}
```

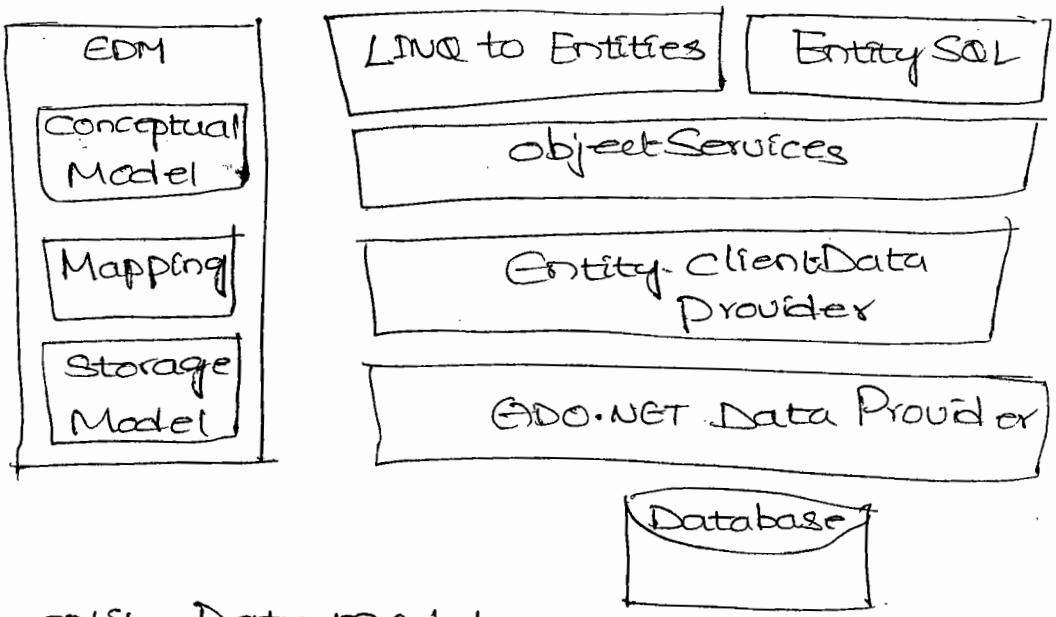
4/01/16

4/1/16 Accessing Data from Database using ADO.NET Entity framework:

ADO.NET EF

1. Microsoft introduced entity framework from .NET framework 3.5
2. Entity framework provides new multiple modeling techniques like code first, Database first, Model first.
3. Entity framework supports full provider model that can communicate with SQL Server, Oracle, MySQL, DB2--etc. (MongoDb)

Entity framework Architecture



1. The entity Data model comprises of three major components.
   a. Conceptual Model
   b. Storage Model
   c. Mapping
2. Conceptual model represents the data you are working with as well as the business.
3. Storage Model Contains the data.
4. Mapping contains information that specify how Conceptual model will communicate with storage model.

5. LINQ to Entities & Entity SQL are query approaches.
6. These query approaches are translated into database native queries by using Entity Client data Provider.
7. ObjectServices is a main entry point for accessing the data from database and return it back, it is responsible for meterilization which is process of converting data returns from Entity framework Library: entity client data provider to an "System.Data.Entity" entity object structure.

class	Description
DbContext	A Dbcontext instance represents a combination of repository patterns, which are used to query from a database and group together the changes, and return back to the data store as a unit.
Dbset	A Dbset represents the collection of Entities in the context. It is reponsible for quering the data from a specific data table
Database	It defines the strategy like Create a new database or to use an existing database.

6/11/16 Ex: Accessing Data from DB using ADO.NET EF:

step-1: Design Database

- Create a new DB by name "MVCProductsDB"
- Add a new table by name "tblProducts"

productID (PK - Identity)	int
Name	varchar
Price	money

Step-2: Design MVC Application

- Create a new MVC Application
- Enable Entity framework (optional in 2013)
  - Right click on References
  - Manage NuGetPackages
  - Select Entity framework
  - Install
- Goto web.config and write the connection string

```
<ConnectionStrings>
<add name="ProductsConnection" providerName="System.Data.SqlClient" connectionString="Data Source=.; Initial Catalog=MVCProductsDb; Integrated Security=SSPI; user Id=sa; password=1234" />
```

- Goto Models folder and add a new class file by name "Products.cs"

```
using System.ComponentModel;
using System.ComponentModel.DataAnnotations.Schema;
[Table("tblProducts")]
public class Product
{
  [Key]
  public int ProductID {get; set}
  public string Name {get; set}
  public decimal? Price {get; Set}
}
```

- Add Another class file into models by name "ProductsContext.cs"

```
using System.Data.Entity;
public class ProductContext: DbContext
{
public ProductContext() :base("name=ProductsConne
                                -ction")
{
}
public DbSet<Product> productslist {get; set}
}
```

- Add a new controller by name "ProductsController"

```
using EntityMvc.Models;
public class ProductsController: Controller
{
ProductContext db=new ProductContext();
public ViewResult Index()
{
return View(db.productslist.ToList());
}
public ViewResult Details(int id)
{
return View(db.productslist.Single(x => x.ProductID
                                == id));
}
}
```

- Add View for Index Action

Name: Index

Template: Empty

Model Class: Product(Models)

Index.cshtml

code for index.cshtml:-

```
@model IEnumerable<EntityMVC.Models.Product>
<h2> Products Index </h2>
<table border="1">
    <th> Product ID </th>
    <th> Product Name </th>
    <th> Product Price </th>
    <th> Actions </th>
@foreach (var item in Model)
    {
      <tr>
       <td> @item.ProductID </td>
       <td> @item.Name </td>
       <td> @item.Price </td>
       <td> @Html.ActionLink("Details", "Details",
                              new{id=item.ProductID}) </td>
    </tr>
}
</table>
```

- Add a view for Details Action

  Name: Details

  Template: Empty

  Model Class: Product (Models)

  Details.cshtml.

code:

```
@model EntityMVC.Models.Product
<h2> Products Details </h2>
<table border="1">
  <tr>
   <td> Product ID </td>
   <td> @Model.ProductID </td>
</tr>
```

```
<tr>
  <td> Product Name </td>
  <td> @Model.Name </td>
</tr>
<tr>
  <td> Product Price </td>
  <td> @Model.Price </td>
</tr>
</table>
<br>
@Html.ActionLink("Goto Index", "Index")
```

→ Click on F5

Note: Set database Initializer to define the strategy.

- Go to Global.asax
- Write the following in Application_Start()

```
Database.SetInitializer<ProductsController>(null);
```

6/1/16 FileResult:

It represents a base class that is used to send binary file content to the response if browser is supported with MIME Types then it can read or download the file.

Syntax:

```
public FileResult Action()
{
return File("Path", "MIME Type");
}
```

Ex:

```
public FileResult Action()
{
return file("~/Content/asp.pdf", "application/pdf");
}
```

JsonResult:

It represents a base class that is used to send Json formatted Content to response. It will Serialize the content and make it available offline.

Syntax:

```
public JsonResult() Action()
  {
   return Json(Object, JsonRequestBehaviour.AllowGet);
  }
```

Ex:

```
public JsonResult Users()
  {
   List<string> users = new List<string>()
     {
      "Rahul",
      "David",
      "John"
     };
  return Json(users, JsonRequestBehaviour.AllowGet);
  }
```

RedirectResult:

It controls the processing of applications actions by redirecting to specified Url. It can be used within or blw applications.

Syntax:

```
public RedirectResult Action()
  {
   return Redirect("Url");
  }
```

Ex: public RedirectResult AspNet()
{
return Redirect("http://asp.net/vnext");
}

RedirectToRouteResult:

It represents a result that performs a redirection by using the specified Route values. It can be used to redirect to any controller action within the application

Syntax:

public RedirectToRouteResult Action()
{
return RedirectToAction("ActionName", "ControllerName");
}

Ex:

public RedirectToRouteResult ProductsList()
{
return RedirectToAction("Index", "Products");
}

ContentResult:

It represents a user defined content type that is sent as response to the client. It also can define the content type and its encoding.

syntax:

public ContentResult Action()
{
return Content("String", "MIME Type, encoding");
}

Ex: public ContentResult Sample()
{
return Content("Products", application/xml, utf-8);
}

JavaScriptResult:

It represents a base class that sents JavaScript content to response. You can call any JavaScript function or use a JavaScript object to send as a response.

Syntax:

```
public JavaScriptResult Action()
{
 return JavaScript(function);
}
```

EmptyResult:

It represents a result that return nothing. So it will not have any return value. You can use such results to access from any another action.

Note:

All action methods in a controller can be invoked directly by url request. Inorder to mark it or restrict its accessbility you have to use the attribute "NonAction"

Ex:

```
[NonAction]
public string hello()
{
 return "Hello word";
}
public ViewResult Index()
{
 ViewBag.msg = hello();
  return View();
}
```

index.cshtml

```
<h1>@ViewBag.msg</h1>
```

7/01/16
8/01/16 CRUD operations using scaffold templates in MVC and 3 Layer architecture.

Step1: Design your database

- Create a new database in SQL server by name "ProductsDb"
- Add a new table by name "tblProducts"

ProductID (PK - Identity)	int
Name	varchar
Price	Money

- Create stored Procedures

Procedure	Action
SPGetProducts SP1	Retain all products
SPAddProducts SP2	Insert a new product
SPUpdateProducts SP3	Update product by its id
SPDeleteProduct SP4	Delete product by its id

Step-2: ~~Design application~~

Data Access Layer

- Go to file menu add → New project
- Select "class Library project"
- Name it as "DataAccessLayer"
- Import the Reference

"System.Configuration"

- Add a new class file by name "Product.cs"

```
public class Product
{
public int ProductID {get; set;}
public string Name {get; set;}
public decimal? Price {get; set;}
}
```

6. Add another class file by name "ProductCRUD.cs"

```
using System.Data;
using System.Data.SqlClient;
using System.Configuration;

namespace DataAccessLayer
{
public class ProductCRUD
 { string strcon=ConfigurationManager.ConnectionStrings
          ["ProductsConnection"].ToString();
  SqlConnection con;
  SqlCommand cmd;
  //Read Operation
 public IEnumerable<Product> productsList
 {get
  {
   List<Product> products=new List<Product>();
   con=new SqlConnection(strcon);
    con.Open();
   cmd=new SqlCommand("SPGetProducts", con);
  SqlDataReader dr=cmd.ExecuteReader();
   while(dr.Read())
   {
  Product product=new Product();
   product.ProductID=Convert.ToInt16(dr["ProductID"]);
  product.Name=dr["Name"].ToString();
  product.Price=Convert.ToDecimal(dr["Price"]);
   products.Add(product);
  }
  dr.close();
  con.close();
 } return products;
}
```

```
// Create Operation
public void AddProduct(Product product)
 {
   con = new SqlConnection(strCon);
   con.Open();
   cmd = new SqlCommand("spAddProducts", con);
   cmd.CommandType = CommandType.StoredProcedure;
SqlParameter paramProductID = new SqlParameter("@ProductID", product.Prod
   cmd.Parameters.Add(paramProductID);
step① ** SqlParameter paramName = new SqlParameter();
       paramName.ParameterName = "@Name";
       paramName.Value = product.Name;
       cmd.Parameters.Add(paramName);
     SqlParameter paramPrice = new SqlParameter("@Price",
                                          product.Price);
     cmd.Parameters.Add(paramPrice);
       cmd.ExecuteNonQuery();
       con.Close();
     }
// Update Operation
public void UpdateProduct(Product product)
   {
    con = new SqlConnection(strcon);
      con.Open();
    cmd = new SqlCommand("spUpdateProduct", con);
    cmd.CommandType = CommandType.StoredProcedure;
      // Pass the parameters ProductID, Name and Price
SqlParameter paramProductID = new SqlParameter("@ProductID",
      cmd.ExecuteNonQuery();   product.ProductID);
      con.Close();             cmd.Parameters.Add
     }                              (paramProductID);
                                    ↓
                                Write step-① here
  // Delete Operation
public void DeleteProduct(int Id)
   {
```

```
        con = new SqlConnection(strcon);
         con.Open();
        cmd=new SqlCommand("spDeleteProduct", con);
         cmd.CommandType = CommandType.StoredProcedure;
    cmd.Parameters.AddWithValue("@ProductID", id);
        cmd.ExecuteNonQuery();
          con.Close();
        }
    }
```

9/1/16 step-3: Application Logic

1. Add a new MVC application to solution
2. Import Reference for DataAccessLayer
3. Create a new controller by name "ProductsController"
4. Write ConnectionString in Web.config File:

```
using DataAccessLayer;
public class ProductsController: Controller
  {
   ProductsCRUD db=new ProductsCRUD();
     public ActionResult Index()
       {
        List<Product> prods=db.productsList.ToList();
        return View(prods);
       }
     public ActionResult Details(int id)
       {
Product prod = db.productsList.Single(x=> x.ProductID
                                          == id);
        return View(prod);
       }
[AcceptVerbs(HttpVerbs.Get)]
 [ActionName("Create")]
public ActionResult CreateOnGet()
    {
      return View();
    }
```

```
[HttpPost]
[ActionName("Create")]
public ActionResult CreateOnPost()
    {
    if (ModelState.IsValid)
    {
     Product product=new Product();
     TryUpdateModel(product);
     db.AddProduct(product);
    }
return RedirectToAction("Index");
 }
[HttpGet]
public ActionResult Edit(int id)
  {
    return View(db.productsList.Single(x=>x.ProductID==id));
   }
[HttpPost]
public ActionResult Edit()
  {
   if (ModelState.IsValid)
   {
    Product product=new Product();
    TryUpdateModel(product);
   db.UpdateProduct(product);
   }
return RedirectToAction("Index");
  }
[HttpGet]
[ActionName("Delete")]
public ActionResult DeleteOnGet(int id)
{
return View(db.productsList.Single(x=>x.ProductID==id));
}
```

```
[HttpPost]
[ActionName("Delete")]
[ValidateAntiForgeryToken]
public ActionResult DeleteOnPost(int id)
    {
    db.DeleteProduct(id);
    return RedirectToAction("Index");
    }
}
```

Step-4: Add Views for your actions

1. View for Index action
   - Name: Index
   - Template: List
   - Model class: Product(DataAccessLayer)
2. View for Details action
   - Name: Details
   - Template: Details
   - Model class: Product(DataAccessLayer)
3. View for Edit Action
   - Name: Edit
   - Template: Edit
   - Model class: Product(DataAccessLayer)
4. View for Delete Action
   - Name: Delete
   - Template: Delete
   - Model class: Product(DataAccessLayer)

Various methods of Binding form Data to a Controller method:

1. Using model field names as params (control "id" is the model field name)

Ex: [HttpPost]

```
public ActionResult Create(string Name, decimal Price)
{
Product product = new Product();
product.Name = Name;
product.Price = Price;
db.AddProduct(product);
return RedirectToAction("Index");
}
```

2. Use a model object to bind the values.

Ex:

```
[HttpPost]
public ActionResult Create(Product product)
{db.AddProduct(product);
return RedirectToAction("Index");
}
```

3. Use a form collection which is dictionary with field names as keys and their values as "Values"

Ex:

```
[HttpPost]
public ActionResult Create(FormCollection fromCollection)
{
Product product = new Product();
product.Name = fromCollection["Name"].ToString();
product.Price = Convert.ToDecimal(fromCollection["Price"]);
db.AddProduct(product);
return RedirectToAction("Index");
}
```

4. Update Model method

Ex: [HttpPost]

```
public ActionResult Create()
```

```
{
  Product product=new Product();
  UpdateModel(product);
  db.AddProduct(product);
  return RedirectToAction("Index");
}
```

5. Try Update Model method, which update the model when model state is valid.

```
Ex: [HttpPost]
public ActionResult Create()
{
  if(ModelState.IsValid)
  {
    Product product=new Product();
    TryUpdateModel(product);
    db.AddProduct(product);
  }
  return RedirectToAction("Index");
}
```

11/1/16

Data Annotations in MVC:-

Data annotations are attributes used with the model fields to control their behaviour and functionality. All attributes in .NET framework are derived from the library System.Attribute. However the data annotations are configured under the library System.ComponentModel.DataAnnotations.

The annotations are in MVC are classified into 3 types.

1. Display Annotations
2. Edit Annotations
3. Validation Annotations

Display Annotations:

The display annotations are used to specify the functionality of model fields in Display Templates like List, Details, Delete.

These annotations will effect the display templated Helpers like

a) Html.Display()

b) Html.DisplayFor()

c) Html.DisplayForModel()

The display annotations are

- DisplayName
- DisplayFormat
- ScaffoldColumn

DisplayName:

It is used to specify a friendly Display name for the model fields.

ex:

```
[DisplayName("ProductID")]
public int ProductID{get; set}
```

(or)

```
[Display(Name="Product Price")]
public double Price {get; set}
```

DisplayFormat:

It represents an attribute that specify's how data fields are displayed and formatted by ASP.NET dynamic data. It is used to set various data format strings for a model field.

Data formats

{0:c} - Currency
{0:d} - Short Date
{0:D} - Long Date
{0:t} - short time
{0:T} - Long Time

Ex:

```
[DisplayFormat(DataFormatString = "{0:c}")]
public double Price {get; set;}
```

ScaffoldColumn:

It represents an attribute that specify's whether the model field will use the scaffold template, which uses the templated helpers like

a) @Html.DisplayForModel()

b) @Html.EditorForModel()

The scaffold templates that are using these templated helpers will not display the model field if it is set to boolean "false".

Ex:

```
[ScaffoldColumn(false)]
public int ProductID {get; set;}
```

Edit Annotations:

The edit annotations are attributes that can control model fields and restrict them to a specific data type are input value so that they cannot set the value anything other than specified.

Ex:

```
[DataType(DataType.Password)]
public string Password {get; set;}

[DataType(DataType.Date)]
public DateTime Manufactured {get; set;}
```

ReadOnly:

It specify's the whether the property is bound to ReadOnly or Read-Write. It will not allow to set a value when specify to boolean "true".

Ex:

```
[ReadOnly(true)]
public int ProductID {get; set;}
```

18/1/16 Validations in MVC:

Validations are required to ensure that contradictionary (or) unauthorized data is not get stored into the "database"

ASP.NET 4.5 introduced "unobstrusive validation" which required jquery script resources mapping.

MVC introduces annotations for validations which are defined under "System.Web.Mvc" and "System.ComponentModel"

→ Validation annotations are attributes defined for model fields and validation messages are displayed by using "Html helpers" (controls)

"@HTML.ValidationMessageFor()"

→ To enable unabstrusive validation set the fallowing in Web.Config file.

```
<appSettings>
  <add key="unobstrusiveJavaScriptEnabled" value="true" />
</appSettings>
```

Validation Annotations in MVC:

1. Required
2. StringLength
3. Compare
4. Range
5. Regular Expression
6. Remote

1. Required:

It represents an attribute that specifies a datafield can't be "null" i.e the field must be protected with value

Ex: [Required (error message = "userid required")]
public string userid {get; set;}

2. StringLength:

It specifies the minimum and maximum length of characters allowed in the data field.

Ex:
```
[StringLength(10, minimumlength=4, ErrorMessage
                = "userid 4 to 10 characters")]
public string userid{get;set;}
```

3. Compare:

It represents an attribute that is used to compare the value in a data field with the value in another values. It returns boolean "false" if both are not equal

Syntax:
```
[Compare("Password", ErrorMessage="Password mismatch")]
public string ConfirmPassword{get;set;}
```

4. Range:

It specifies the numeric range constants for the value in data field and ensures that input value false within specified range.

Syntax:
```
[Range(15,25,ErroMessage="Age is 15 to 25 only")]
public int? Age{get;Set;}
```

5. Regular Expression:

It Specifies that the data field values in ASP.NET dynamic classes must match with the specified the regular-expression.

Ex:
```
[RegularExpression(@"(?=.*[A-Z])\w{4,15}", ErraMessage
                = "Password 4 to 15 chars with atleast one
                                                uppercase")]
public string Password{get;set;}
```

6. Remote:

MVC5 introduces new remote validations that provides an attribute using Jquery validation plugin in order to validate the value present in datafield. It is similar to a custom validator which uses a server side validation function.

Ex:

1. Create a new database Table with fields and also values

userid[PK]	varchar
usaName	varchar

2. By using entity data model bind the table with your application "usersDataModel.edmx"

- tblusers.cs
- MvcProductsDbEntities.cs[Context]

3. Create a new Controller by name "usersController"

```
using MvcProducts.Model;
public class usersController : Controller
{
MvcProductsDbEntities db=new MvcProductsDbEntities();
public JsonResult IsUseridTaken(string userid)
{
return Json(!db.tblusers.Any(x=>x.userid==userid),
JsonRequestBehaviour.AllowGet);
}
public ActionResult Index()
{
return View();
}
}
```

Create View for Index
Name: Index
Template: Create
Model: tbluser(Models)

4. Go to "tbluser.cs"

```
using System.Web.Mvc;
public partial class tblusers
{
[Remote("IsUseridTaken", "users", ErrorMessage="Userid
Taken - Please Try Another")]
public string Userid{get;set;}
}
```

Action Selector in MVC:

Action selectors are attributes defined for controller action method in order to control their behaviour. These are defined by System.Web.Mvc

→ The Commonly used action selectors are

1. [ActionName]
2. [AcceptVerbs]
3. [NonAction]

1. [ActionName]:

It represents an attributes that is used to specify an alias name for controller action.

Ex: [ActionName("Create")]
public ActionResult CreateOnGet()
{
return View();
}

2. [AcceptVerbs]:

It represents an attribute that specifies which HttpVerbs an action method will respond to i.e Get, Post, Put, delete etc

Ex: [AcceptVerbs(HttpVerbs.Post)]
public ActionResult Create()
{
return RedirectToAction("Index");
}

3. [NonAction]:

It represents an attribute that is used to indicate that controller methods are not an action method can't invoked by Url request.

Ex: [NonAction]
public string username(string uname)
{
return "Hello!" + uname;
}

19/1/16 Action filters in MVC:

Action filters are attributes defined for a controller or its action method. All action filters are used to control the functionality of an action method or Controller class, which includes authentication, authorization, ValidateInput etc.

All action filters in MVC are derived from the base "ActionFilterAttribute" it is the base class defined under "System.Web.Mvc". Some of the commonly used action filters are

1. ValidateInput
2. ChildActionOnly
3. OutputCache
4. HandleError
5. RequireHttps
6. Authorize
7. AllowAnonymous
8. ValidateAntiForgeryToken etc...

ValidateInput:

It represents an attribute that is used to mark action methods whose input must be validated. It Controls cross site scripting attack and will not allow to post queries from controls.

You can allow that by setting that boolean "false"

Ex:

1. Create a new controller by name "DemoController"
2. Add following methods

```
[HttpGet]
public ViewResult Comments()
{
```

```
        return View();
    }
    [HttpPost]
    [ValidateInput(false)]
public string Comments(string Comments)
    {
    return "Your Comments: " + Comments;
    }
```

for binding to user control (#Eval & #Bind) ↓ Read ↓ Read+Write

uses 302 method

Html.Partial - Uses Round Trip
Html.RenderPartial - It will not use RoundTrip

3. Add View for comments action

Name: Comments

Template: Empty (Without model)

Comments.cshtml

```
<h2> Your Comments </h2>
<form method="post">
<textarea name="comments" rows="10" cols="40">
</textarea>
<br/>
<input type="submit" value="Post Comments" />
</form>
```

ChildActionOnly:

It represents an attribute i.e used to indicate that an action method should be called only as child Request. and can't be invoked directly by URL request.

Ex:

1. Create a new database table. "tblProducts"

ProductID	int
Name	varchar(20)
Price	double
Photo	varchar

2. Add records with Photo field using the Image Path.
   ~/Photos/mobile.jpg
3. Goto MVC application
4. Add a new folder by name "photos"
5. Add product images into the folder.
6. Goto Models and add "ADO.NET Entity Data Model" for your table ProductsDataModel.edmx
   Model class : tblProduct
   Context class : ProductsContext
   DbSet : tblProducts
7. Add a new controller by name "ProductsController"

```
using MvcDemo.Models;
public class ProductsController : Controller
  {
   ProductsContext db=new ProductsContext();
   public ActionResult Index()
   {
   return View(db.tblProducts.ToList());
    }
 [ChildActionOnly]
public PartialViewResult ProtoType()
    {
    return PartialView();
     }
}
```

8. Add View for ProtoType Action
   Name: ProtoType
   Template: Empty
   Model class: tblProduct(Models)
   ☑ Create as Partial

ProtoType.cshtml

```
@model IEnumerable<MvcDemo.Models.tblProduct>
<table border="1">
    <tr>
     <td>
<img src="@Url.Content(Model.Photo)" width="100"
                                        height="100" />
    </td>
   <td>
    <table border="1">
      <tr>
      <td>Product ID </td>
      <td> @Model.ProductID </td>
      </tr>
      <tr>
      <td>Product Name </td>
      <td>@Model.Name </td>
      </tr>
      <tr>
      <td>Product Price </td>
      <td>@Model.Price</td>
      </tr>
     </table>
    </td>
   </tr>
  </table>
```

9. Add view for Index Action

Name : Index

Template : Empty

Model Class : tblProduct (Models)

Index.cshtml

```
@model IEnumerable<MvcDemo.Models.tblProduct>
<h2> Products Index </h2>
@foreach (var item in Model)
{
|| @Html.Partial ("ProtoType", item)
   {
     Html.RenderPartial ("ProtoType", item);
   }
}
```

Note: Html.Partial uses a round trip and returns a string so that its value can be stored in a variable and accessed from any method.

Html.RenderPartial returns void (nothing) and directly write the output to response. So that its faster when compared to Html.Partial.

20/1/16 OutputCache:

It represents an attribute that is used to mark an action method whose output will be cache. That is it uses the cache profiles and makes the data available in the cache memory so that if it is frequently requested then it will access from the cache instead of communicating with server.

Ex:

1. Go to Web.config and create cache profiles.

```
<system.web>
<caching>
 <outputCache enableOutputCache="true" />
<|outputCache>
```

```
<outputCacheSettings>
 <outputCacheProfiles>
<add name="30sec" duration="30" varyByParam="None" />
<outputCacheSettings>
</outputCacheProfiles>
</system.web>
```

2. Goto "ProductsController" and set cache profile for any action

```
[OutputCache(CacheProfile="30Sec")]
public ActionResult Index()
  {
   return View(db.tblProducts.ToList());
  }
```

3. Add View for Index Action.

Name: Index

Template: Empty (without model)

index.cshtml

```
@DateTime.Now.ToString()
```

HandleError:

It represents an attribute i.e used to handle an Exception i.e thrown by an action method, it requires custom error mode set to "On" so that it displays user friendly error pages instead of yellow death screens.

1. Yellow death
2. Blue death
3. Kiss of death (Uses in HTML5)

} death screens

Ex:

1. Create a new controller called "ErrorController"

```
public class ErrorController : Controller
 {
 public ActionResult Notfound()
  {
   return View();
  }
}
```

2. Add View for "Notfound" Action

Notfound.cshtml

```
<h1>Action You Requested - Not Found</h1>
```

3. Goto "Shared" folder in Views and add a new View by name "Error.cshtml"

```
<h1> Something Went Wrong - Please Contact Your admin
</h1>
```

4. Goto Web.Config and set Custom Errors

```
<System.web>
 <customErrors mode="On" defaultRedirect="
                    ~/Views/Shared/Error.cshtml">
 <error statusCode="404" redirect="Error/NotFound"
 />
 </customErrors>
 </system.web>
```

5. Goto "DemoController and enable HandleError

```
[HandleError]
public class DemoController : Controller
  {
  }
```

21/1/16 Authorize:

It represents an attribute that is used to restrict access for a controller or action method. It sets authorization for specific roles and users.

Ex1: Windows Authentication

1. Create Users and Groups on your windows
   - Right click on "PC" icon
   - Select Manage
   - Goto Users and Roles
   - Right click on Users and Create new user.
   - Set User Name and password
   - Goto Groups and Create new group by name "NareshAdmin"
   - In Group Properties click "Add" button and add users into group.
     "RahulAdmin"
2. Create a new MVC application with authentication type as "Windows Authentication"
   (Click "Change Authentication" button while creating a new MVC application)
3. Host the application on IIS and Enable Windows authentication on IIS.
   (On IIS goto Authorize category)
4. Create a new Controller by name "AdminController"

```
[Authorize(Roles = "NareshAdmin")]
public class AdminController: Controller
{
    public ActionResult Index()
    {
        return Content("Admin Home");
    }
}
```

Ex-2: Form Authentication using Membership

1. Create a new MVC application with authentication type as "Individual User Accounts".
2. Run application
3. Goto Home-Index
4. Click on "Register" and Register users
   - John
   - David
5. ASP.NET Profile provider will create a local database by name "Aspnet.mdf" (App_Data)
6. Goto Server Explorer and Manage the following tables in Aspnet.mdf

Users

UserId	UName
1	John
2	David

UsersRoles

UserID	RoleID
1	2
2	1

Roles

RoleID	Name
1	HR
2	Admin

7. Create a new controller by name "AdminController"

```
[Authorize(Roles = "Admin")]
public class AdminController : Controller
{
    public ActionResult Index()
    {
        return View();
    }
}
```

AllowAnonymous:

It represents an attribute that marks the controller action to skip authentication during the authorization process.

Ex:

```
[AllowAnonymous]
public ActionResult Register()
{
return View();
}
```

for MIMETYPES goto → Inetmgr → find MIME Type

RequireHttps:

It represents an attribute that forces an unsecured Http request to be resent over Https request. This requires access to domains configured under Https protocol.

Ex:

```
[RequireHttps]
public ActionResult NareshITWebsite()
{
return Redirect("https://localhost/Init");
}
```

ValidateAntiforgeryToken:

It represents an attribute that is used to prevent the forgery of a request that is it will not allow any 3rd party application to post data into in your applications

Ex:

```
[HttpPost]
[ValidateAntiforgeryToken]
public ActionResult Edit(Product product)
{
db.UpdateProduct(product);
return RedirectToAction("Index");
}
```

22/1/16 Custom Filters in MVC (AppendAllText - for CRUD & also close the file)

Action filters in MVC are implemented from action filter attribute, which provides methods that can be overriden and allows to create custom filters. The methods are

* OnActionExecuted()
* OnActionExecuting()
* OnResultExecuted()
* OnResultExecuting()

Ex:

1. Add a new folder by name "Custom Filters"
2. Add a new class file into CustomFilter folder
   "TrackControllerActions.cs"

```
using System.Web.Mvc;
using System.IO;
public class TrackControllerActions : ActionFilterAttribute
{
  public static void CreateTraceFile(string str)
  {
    File.AppendAllText(HttpContext.Current.Server.
        MapPath("~/Content/trace.txt"), str);
  }
public override void OnActionExecuting(ActionExecutingContext
                                       filterContext)
  {
    string str="\n"+filterContext.ActionDescriptor.
  ControllerDescriptor.ControllerName+"-->"+
  filterContext.ActionDescriptor.ActionName+"-->"+
    "Executed on:"+DateTime.Now.ToString()+
                    "\n";
    CreateTraceFile(str);
  }
}
```

3. Goto "ProductsController" and apply your custom filter

```
using MvcProject.CustomFilters;
[TrackControllerActions]
public class ProductsController : Controller
{
    --actions--
    FileResult ImageIndex()
    {
        return File("~/Content/trace.txt");
    }
}
```

Minification and Bundling:-

Minification is the process of reducing the size of JavaScript and css files in order to improve the load time. ASP.NET 4.5 provides several minification plugins which allow various code optimization to scripts or css they remove unnecessary white space, line break and sortening variable names into one character.

Ex:

1. Add a new style sheet into /Content folder by name "Headings.css"

```
.headings
{
    background-color: red;
    color: white;
    text-align:center
}
```

2. Install minification Tools or Plugin
   - ToolsMenu
   - Update and Extensions
   - Search online for "WebEssentials"
   - Download and install
3. Right click on "Headings.css" and select "Minify"

   Headings.min.css

Bundling:

It is the process of creating a bundle of css and JavaScript file in order to reduce the no.of requests. Fewer files means Fewer Http request and that can improve page load performance

Ex:

1. Create following style sheets and add onto content folder.
   - headings.css
   - paragraphs.css
2. goto App_Start → BundleConfig.css

```
using System.Web.Optimization;
public class BundleConfig
 {
public static void RegisterBundles(BundleCollection bundles)
  {
  bundles.Add(new StyleBundle("~/Content/Demo").
    Include("~/Content/headings.css",
          "~/Content/paragraphs.css"));
  }
}
```

3. You can access the bundle from any View

index.cshtml

```
@Styles.Render("~/Content/Demo")
```

If it is Scripts then

```
@Scripts.Render("~/Content/Demo")
```

23/1/16

Bootstrap:

* It is one of the largest repository of css and JavaScript
* It provides templates that can implement in your app's.
* ASP.NET 4.5 onwards bootstrap is ~~installed~~ Integrated into web applications.
  a) Web Form
  b) Mvc app
  c) SPA

Ex:

1. Visit GetBootStrap official web site
2. Goto CSS or Javascript category.
3. Select any example like buttons, menus, forms...
4. This will provide you ess and HTML code.
5. Copy the css code into a css file in your website and HTML code into your views.
6. Import the following files into your cshtml View
   - bootstrap.min.css
   - bootstrap.min.js

View start.cshtml is default master page

Master Layouts:

1. A master layout is nothing but a master page used in web applications to give a uniform appearence and functionality for all pages
2. A master layout is also a view with extension ".cshtml"
3. The key component of master layout is "RenderBody()", which specifies the location that indicates where the child page content will be rendered.
4. You can assign master layout to any existing view by using the attribute "Layout"

Index.cshtml

```
@{
    Layout = "~/Views/Shared/SiteMaster.cshtml";
}
```

5. If you want to set the master layout as the default master layout for the application then configure the Layout attribute in "_ViewStart.cshtml" (Views → Shared → _ViewStart.cshtml)

Ex:

1. Go to Views → Shared folder
2. Right click and Select Add new View
   Name: SiteMaster
   Template: Empty (without model)

☐ Use Master Layout uncheck

SiteMaster.cshtml

```
<!DOCTYPE html>
<html>
<head>
... Your styles and Scripts
</head>
<body>
<table width="800">
<tr style="background-color: red; color: white;
                text-align: center">
<td> @ Html.ActionLink ("Home", "Home") </td>
<td> @ Html.ActionLink ("About", "About") </td>
<td> @ Html.ActionLink ("Contact", "Contact") </td>
</tr>
<tr height="400">
<td colspan="3"> @ RenderBody() </td>
</tr>
<tr style="background-color: red; color: white;
                text-align: center" />
<td colspan="3"> &copy; Copy Right 2016 </td>
</tr>
</table>
</body>
</html>
```

25/11/16 MVC Helpers and Ajax Helpers:

Mvc Ajax Helpers:

Helpers are methods that dynamically renders Html controls. MVC Supports Html helpers and Ajax Helpers.

Ajax Helpers:

10

Ajax Helpers:

Helper	Description
Ajax.ActionLink()	Creates an hyperlink to an controller action that fires Ajax request when clicked.
Ajax.RouteLink()	Similar to action link but generates a link to particular route instead of controller action
Ajax.BeginForm()	Creates a form element that submits the data to a particular Controller using Ajax.
Ajax.BeginRouteForm()	Similar to BeginForm but submits the data to a particular route.

Ajax Options:

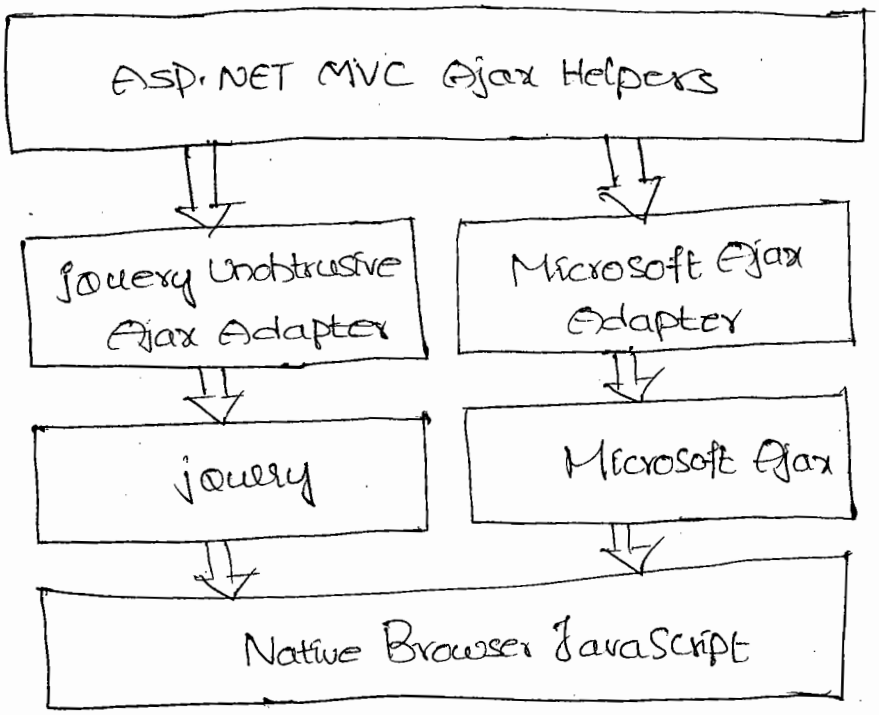
Option	Description
HttpMethod	Specifies the HTTP method GET or POST
UpdateTargetId	Specifies the element into which the resulting markup must be rendered.
LoadingElementId	Specifies the element that displays Ajax progress.
InsertionMode	Sets the insertion mode, which specifies where the resulting markup must be rendered i.e Before Existing, After Existing or Replace Content.

OnBegin
OnComplete
OnFailure
OnSuccess

} JavaScript functions → Specifies the javascript function to be called on various events

LoadingElementDuration — Specifies how long the progress animation to display.

Ajax Architecture for MVC:



Ex:

1. Create a new Database Table by "tblItems"
   - ItemID
   - IName
   - Branch
   - Sales
2. Create a new MVC Application
3. Right click on References and select "Manage NuGet Packages"
4. Search Online for "Ajax" Packages.

5. Install the following

- Ajax Unobtrusive Jquery
- Ajax Helpers

6. Goto Models and Add "ADO.NET Entity Data Model"

Name : ItemsSalesModel.edmx

ContextClass : ProductDbEntities

DbSet : tblItems

Model class : tblItem

7. Goto Shared folder in Views and Add a new "Partial View"

Name : Sales

Template : Empty

Model Class : tblItem

Check the option Create as Partial.

_Sales.cshtml

```
@model IEnumerable<MvcHelpers.Models.tblItem>
    <table border="1">
     <th>Item ID</th>
     <th>Item Name</th>
     <th> Item Branch</th>
     <th> Item  Sales</th>
    @foreach(var item in Model)
     { <tr>
<td> @item.ItemID</td>
<td> @item.Name</td>
<td> @item.Branch</td>
<td> @item.Sales</td>
   </tr>
}
</table>
```

8. Add a new Controller by name

ItemsSalesController.cs

```
using MvcHelper.Models;
public class ItemsSalesController: Controller
    {
     ProductsDbEntities db=new ProductsDbEntities();
    public ViewResult Index()
            {
              return View();
            }
        public PartialViewResult ShowAll()
         {
         System.Threading.Thread.Sleep(3000);
       List<tblItem> sales= db.tblItems.ToList();
      return PartialView("_Sales", sales);
        }
      public PartialViewResult Top2()
        {
      System.Threading.Thread.Sleep(3000);
      List<tblItem> sales = db.tblItems.OrderByDescending.
                                        (x => x.Sales).Take(2).ToList();
      return PartialView("_Sales", sales);
      }
  public PartialViewResult Bottom2()
    {
    System.Threading.Thread.Sleep(3000);
  List<tblItem> sales= db.tblItems.OrderBy(x => x.Sales).
                                        Take(2).ToList();
    return PartialView("_Sales", sales);
  }
```

9. Add View for Index Action

Name : Index

Template : Empty (without model)

Index.cshtml

```
<script src="jquery-1.10.2.min.js"></script>
<script src="jquery.unobtrusive.ajax.min.js"></script>
<h2>Items Sales Info</h2>
@Ajax.ActionLink("Show All", "ShowAll", new AjaxOptions()
    {
     HttpMethod = "GET",
     UpdateTargetId = "dataList",
     LoadingElementId = "loading",
     InsertionMode = InsertionMode.Replace,
    }
)
<span>|</span>
--- similarly create action links for Top2 and Bottom2 ---
<hr noshade />
<div id = "loading" style = "display: none">
 <img src = "~/Content/spinner.gif" />
 Loading Please Wait...
</div>
<div id = "dataList"></div>
```

26/1/16 HTML Helpers:

Templated Helpers in MVC:

The scaffold templates in MVC render model fields and their values by using two types of templated helpers.

1. Display Templated Helpers
2. Editor Templated Helpers

Display Templated Helpers:

These are HTML methods which render a label control for specified values. MVC provides the following Display Helpers.

Helper	Description
Html.Display()	To display literal text
Html.DisplayFor()	To Bind any specified model field
Html.DisplayForModel()	Renders a label for every field in the model.

Syntax:

@ Html.Display("string expression");

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

@ Html.DisplayForModel()

Editor Templated Helpers:

These are HTML methods that render a TextBox for specified model fields. They are similar to Display Helpers in attributes and Syntax

a) Html.Editor()

b) Html.EditorFor()

c) Html.EditorForModel()

HTML Helpers:

MVC provides a small set of helpers, which render HTML controls. However it provides options to create custom helpers as every helper is derived from "HTTPHtmlHelper" base

The commonly available helpers in MVC are

@Html.BeginForm()

@Html.EndForm()

@Html.ActionLink()

@Html.TextBox()

@Html.TextArea()

@Html.CheckBox()

@Html.RadioButton()

@Html.DropDownList()

@Html.ListBox()

@Html.ValidationMessage()

@Html.AntiForgeryToken() etc --

Creating HTML forms:

```
@HTML.BeginForm(); //<form>
    ---form elements---
@Html.EndForm(); // renders a </form>
```

(or)

```
@using (Html.BeginForm())
{
    ----form elements
}
```

Syntax: @using(Html.BeginForm("ActionName", "ControllerName", formMethod.GET/POST, new{attributes}))

{

}

--- this will render the following ---

<form action="Controller/Action" method="POST|GET">

</form>

Html.TextBox():

It renders input type text. And uses all attributes of HTML5 textBox.

Syntax:

@ Html.TextBox("name", "value", new {attributes})

Note:

If any attribute is a keyword of .NET framework then use "@".

Ex:

@Html.TextBox("username", "", new {placeholder="Name max 10 chars", style="background-color:red", @readonly})

27/01/16 Transporting values from one action to another:

1. Add a new controller by name "DemoController"

2. Add following action methods

```
[HttpGet]
public ActionResult Index()
{
  return View();
}
[HttpPost]
public ActionResult Index(string txtname)
{
return RedirectToAction("Result", new {txtname});
}
public ActionResult Result(string txtname)
{
  ViewBag.msg=txtname;
  return View();
}
```

3. Add View for Index Action.

Index.cshtml

```
@using (Html.BeginForm())
{
  <div> Enter Name:
      @Html.TextBox("txtname", " ")
  <input type="submit" value="Submit" />
  </div>
}
```

4. Add View for Result Action.

Result.cshtml

```
<h2> Hello! @ViewBag.msg </h2>
```

@Html.DropdownList and @Html.ListBox():

These are list controls in MVC that render <select> in HTML. Every item in the list is of type "SelectListItem". Each item comprises of following attributes

a) Text
b) Value
c) Selected

Ex1: Create a DropDownList in View with hard coded values.

```
<div> @using (Html.BeginForm())
      {
       <div>
       Select a City: @Html.DropDownList ("lstCities",
       new List<SelectListItem>()
            {
            new SelectListItem{Text="Delhi", Value="Delhi"},
            new SelectListItem{Text="Hyd", Value="Hyd"},
           }, "Select Your City")
```

28/1/16 Ex2: Populating dropdownlist Items with a collection accessed from ViewBag.

```
public ActionResult Comments()
  {
   ViewBag.cities=new List<SelectListItem>()
   {
    new SelectListItem {Text="Goa", Value="Goa"},
    new SelectListItem {Text="Mumbai", Value="Mumbai"}
    };
  }
```

Comments.cshtml

```
@Html.DropDownList("lstCities", (List<SelectListItem>)
        ViewBag.cities,"Select Your City")
```

Ex 3: DropDownlist Items accessed from Model Field.

1. Create a new model class City.cs

```
public class City
{
  public int CityId {get;set;}
  public string Name {get;set;}
 }
```

2. Add another class with Cities List CitiesData.cs

```
public class CitiesData
  {
   List<City> cities=new List<City>()
    {
     new City {CityId=1, Name="Delhi"},
     new City {CityId=2, Name="Hyd"},
    };
public IEnumerable<City> citiesList
   {get { return cities;}
} } }
```

3. Goto Controller and add a new ActionMethod

```
using HtmlHelpers.Models;
public class DemoController : Controller
{
  CitiesData db=new CitiesData();
  [HttpGet]
  public ActionResult Index()
  {
  ViewBag.Cities = new SelectList(db.CitiesList,
                                  "CityId", "Name");
    return View();
  }
}
```

ctrl+k+d ↓ for organize the data in a proper way

4. Index.cshtml

Select a City : @ Html.DropDownList("Cities", "Select Your City")

Implementing Search in a List:

1. Create a new database Table with fields
   - ProductId
   - Name
   - Price
   - Category
2. Create a new MVC application
3. Add ADO.NET Entity Data Model for Products table
   - tblProducts.cs (Model)
   - ProductsDbEntities (Context)
4. Add a New Controller
   - MVC5 Controller with Views using Entity Framework

   Name: Products Controller

   Model Class: tblProduct

   Context Class: ProductsDbEntities

5. Goto ProductsController and modify Index action.

```
public ActionResult Index(string searchBy, string search)
{
 if(searchBy=="Name")
  {
  return View(db.tblProducts.Where(x=>x.Name==
                                   search || search==null).
                                   ToList());
  }
 else
  {
 return View(db.tblProducts.Where(x=>x.Category==
                                   search || search==null).ToList());
  }
}
```

6. Goto Index.cshtml

```
<h2>Index</h2>
<div>
@using(Html.BeginForm("Index","Products",FormMethod.
                                          Get))
  {
   <div>
  Search By: @Html.RadioButton("searchBy","Name",true)
      Name @Html.RadioButton("SearchBy","Category")
       category
   <br/>
 @Html.TextBox("search")
 <input type="Submit" value="Search" />
  </div>
 }
</div>
```

---- below the <th> in the table add the following ----

```
<table>
 @if(Model.Count()==0)
  {
```

```
<tr>
<td colspan="3" align="center">
No matching Records found </td>
</tr>
}
else
{
foreach(var item in Model)
{
---- <tr> - —
}
}
```

Package provider
Telerik - MVC ASP.NET
DevExpress - MVC

29/01/16 Paging Control in MVC Views:

1. Goto "references" → "Manage NuGet Packages"
2. Search online for "PagedList" and install the following
   - PagedList
   - PagedList.MVC
3. Goto ProductsController

```
using PagedList.MVC;
using PagedList;
public class ProductsController: Controller
{
private ProductsDbEntities db=new ProductsDbEntities();
public ActionResult Index(string searchBy,string search,
                          int? page)
{
if(searchBy=="Name")
{
return View(db.tblProducts.Where(x=>x.Name==
    search || search==null).ToList().ToPagedList(
                                    page ?? 1,2));
}
else
```

```
        {
        return ViewCdb.tblProducts.Where(x=>x.Category==
        search || search==null).ToList().ToPagedList(page ?? 1,2));
        }
    }
}
```

4. Goto Index.cshtml

```
@using PagedList;
@using PagedList.MVC;
@model IPagedList<MvcSearch.Model.tblProducts>
--- make changes in table header Row ---
<th>
@Html.DisplayNameFor(model=>model.First().Name)
</th>
// Similar for other fields like Category.etc --
--- Add PagedList control at the bottom of page ---
@Html.PagedListPager(Model,page=>Url.Action("Index",
new{page, searchBy=Request.QueryString["searchBy"],
search=Request.QueryString["search"]}),new PagedList-
-RenderOptions(){DisplayPageCountAndCurrentLocation=true,
DisplayItemSliceAndTotal=true})
```

30/11/16 Accessing Data From multiple tables:

* Create following tables in your SQL DB

tblCategory

(PK)	CategoryId	int
	CategoryName	varchar(30)

tblProducts

(PK)	ProductID	int
	Name	varchar
	Price	money
(FK)	CategoryId	

2. Records in the table "tblCategory"

CategoryID	CategoryName
1	Electronics
2	Shoes

3. Records in the table "tblProducts"

ProductID	Name	Price	CategoryId
1	mobile	15000	1
2	Shoe	8000	2
3	Nike	7000	1

4. Create a new Mvc application and write the connection string in Web.Config to connect with Products database

Name: "ProductsContext"

5. Goto Models folder and add the following classes

Category.cs

```
using System.ComponentModel.DataAnnotations.Schema;
[Table("tblCategories")]
public class Category
{
 [Key]
 public int CategoryId {get; set;}
 public string CategoryName {get;set;}
List<Product> products {get; set;}
}
```

Product.cs

```
[Table("tblProducts")]
public class Product
{
 [Key]
 public int ProductID {get;set;}
 public string Name{get;set;}
 public decimal Price{get; set;}
```

```
    public int CategoryId {get; set;}
}
```

ProductsContext.cs

```
public class ProductsContext : DbContext
{
public DbSet<Product> productsList {get; set;}
public Dbset<Category> CategoryList {get; set;}
}
```

6. Add a new Controller by name "CategoriesController"

```
public class CategoriesController : Controller
  {
   ProductsContext db=new ProductsContext();
      public ActionResult Index()
      {
      return View(db.categoriesList.ToList());
       }
  }
```

7. Add a View for Index action

Index.cshtml

```
@model IEnumerable<MultipleTablesMvc.Models.Category>
    <h2> Categories List </h2>
    <ol>
    @foreach(var item in Model)
    {
      <li>@Html.ActionLink("@(item.CategoryName, "Index",
    "Products", new{CategoryId=item.CategoryId}, null) </li>
    }
  </ol>
```

8. Add another controller by name "ProductsController"

```
public class ProductsController: Controller
{
    ProductsContext db=new ProductsContext();
    public ActionResult Index(int categoryId)
    {
    return View(db.productsList.Where(x=> x.CategoryId==
                                        CategoryId).ToList());
    }
public ActionResult Details(int id)
    {
    return View(db.productsList.Single(x=> x.ProductID
                                        ==id));
    }
}
```

9. Add View for Index Action

Index.cshtml

```
@model IEnumerable<MultipleTablesMvc.Models.Product>
<h2>Products List</h2>
<ol>
@foreach(var item in Model)
  {
<li> @Html.ActionLink(item.Name,"Details",new{ id=
                                        item.ProductID}) </li>
  }
</ol>
```

10. Add View for Details Action

Name: Details

Template: Details

Model class: Product(Models)

Handling Multiple Button Clicks in a View: (Performing Various Actions on Various Button Click Events)

1. Add a new Controller by name DemoController

```
public class DemoController : Controller
{
public ActionResult Index(string submitform)
  {
  if(submitform == "Insert")
    {
    return Content("Record Inserted");
    }
  else if(submitform == "Delete")
    {
    return Content("Record Deleted Successfully");
    }
  return View();
  }
```

→ // On Post Request it will get and return the values.

// On Get Request it will return a View

2. Add View for Index action

Index.cshtml

```
<h2> Index </h2>
@using (Html.BeginForm("Index", "Demo", FormMethod.Post,
                new { id = "submitform" }))
{
<input type="submit" name="submitform" value="Insert" />
<input type="submit" name="submitform" value="Delete" />
}
```

→ to maintain same name for both buttons.

Multiple Tables Example with DropDownList to Select Categories

1. Goto CategoriesController

```
public ActionResult Index()
{
ViewBag.categories=new SelectList(db.categoriesLists
            "CategoryId","CategoryName");
             ↓value field      ↓Text field
 return View(db.categoriesList.ToList());
 }
```

2. Goto Index.cshtml

```
<div>
@using(Html.BeginForm("Index","Products",FormMethod.Post,
                                    new {id="categories"}))
 {
  <div>
Select a category: @Html.DropDownList("categories",
                                "Select a Category")

<input type="submit" value="Show Products" />

  </div>
 }
</div>
```

→ Create a table like

tblGender

GendID	Name
1	Male
2	Female

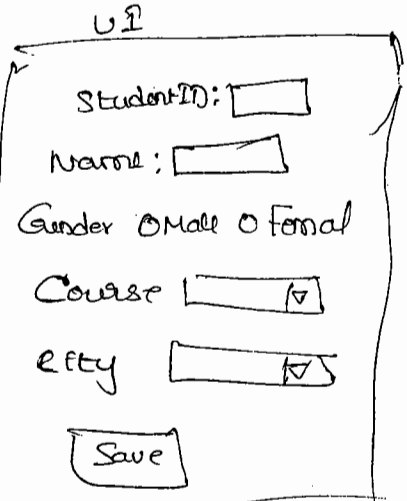
tblCourse

CourseID	Name
1	.NET
2	Java

tblStudents

StudID	int
Name	varchar
Gender	GendID
Course	CourseID
City	varchar

UI



Index

Male

.NET

11/2/16 Consuming WCF Services:

Step1: Create a new WCF service

* Create a new Project
* Select "Visual C#" → WCF → WCF Service application
* Goto "Service1.svc.cs" and add the following method

```
public string UserName(string uname)
{
 return "Hello " + uname + "Welcome to WCF Services";
}
```

* Goto "IService1.cs" and register the method as "OperationContract"

```
[OperationContract]
string UserName(string uname);
```

* Run the service and copy service URL

http://localhost:60806/Service1.svc

Step2: Consume the service in MVC application

* Create a new MVC application
* Right click on "References" and select "Add Service Reference()"
* Type the service URL and click "Go"
* After discovering the service click "Advanced" button
* ☐ "Reuse Types" must be unchecked.
* Click "ok"
* Add a new controller by name "DemoController"

```
public ActionResult Index(String id)
{
ServiceReference1.Service1Client db= new
                ServiceReference1.Service1Client();
 ViewBag.user = db.UserName(id);
} return View();
```

* Add View for Index action

"Index.cshtml"

```
<h2> Demo-Index </h2>
@ViewBag.user
```

Creating Unit Tests for MVC application:

1. Create a new project
2. Select "ASP.NET web application"
3. Select "MVC" template and also choose the option as "Create Unit Test"
4. This will add "MvcTestDemo.Tests" to your solution

MVC Project: MVCTestDemo

Test Project: MVCTestDemo.Tests

5. Goto MVCTestDemo and add a new controller by name "ProductsController"

```
public class ProductsController: Controller
{
    public ViewResult Index()
    {
        return View();
    }
    public ActionResult Details()
    {
        throw new NotImplementedException();
    }
}
```

6. Add View for Index action

"Index.cshtml"

7. Goto MVCTestDemo.Tests project
8. Right click on Controllers folder and add a new class by name "ProductsControllerTest.cs"

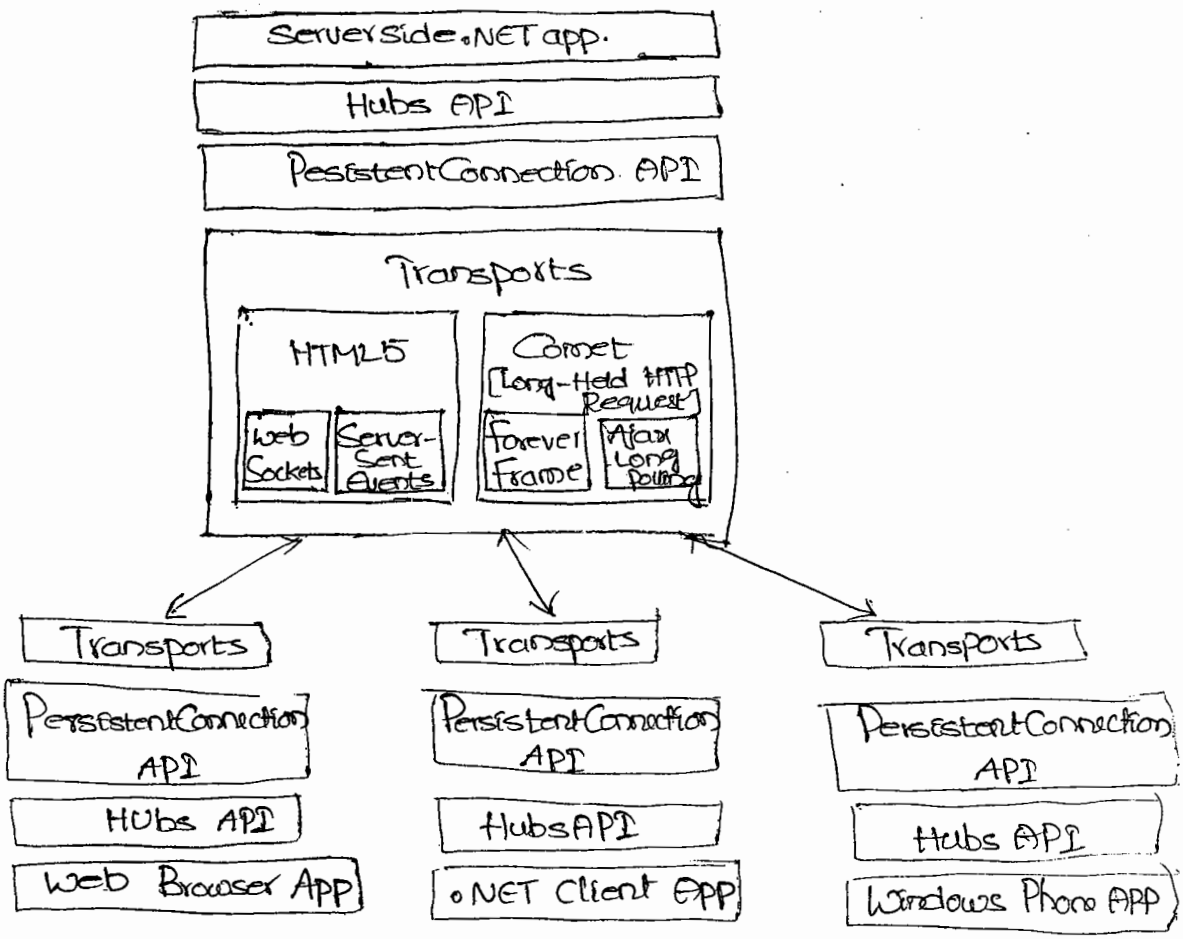
```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using MvcTestDemo.Controllers;
using System.Web.Mvc;

[TestClass]
public class ProductsControllerTest
    {
      [TestMethod]
      public void Index()
      {
        // Arrange (Which type of controller)
    ProductsController controller=new ProductsController();
        // Act
    ViewResult result=controller.Index() as ViewResult;
        // Assert
      Assert.IsNotNull(result);
      }
    [TestMethod]
      public void Details()
      {
        // Arrange
      ProductsController controller=new ProductsController();
        // Act
      ViewResult result=Controller.Details as ViewResult;
        // Assert
      Assert.IsNotNull(result);
    }
}
```

9. Goto "Test" Menu and Select Run → "All tests".

2/02/16 SignalR:

It is a service that uses hubs and web sockets to transport information to all clients or to a specific client. Server will push the updates to all connected clients.

SignalR hub service provides properties and methods that are responsible to push updates to specific caller or group



Ex: Chat Service in MVC App

1. Create a new MVC application
2. Install SignalR service (References)
3. Right click on Project name and select "Add → New Items"
4. Select Goto SignalR category and select "SignalR Hub class"
5. Name it as "chatHub.cs"

```
using Microsoft.AspNet.SignalR;
public class chatHub : Hub
{
```

```
    public void Send(string name, string message)
    {
      Clients.All.addNewMessageToPage(name, message);
    }
}
```

6. Add a new Controller by name "HomeController"

```
public ActionResult Chat()
{
  return View();
}
```

7. Add view for chat action

Chat.cshtml

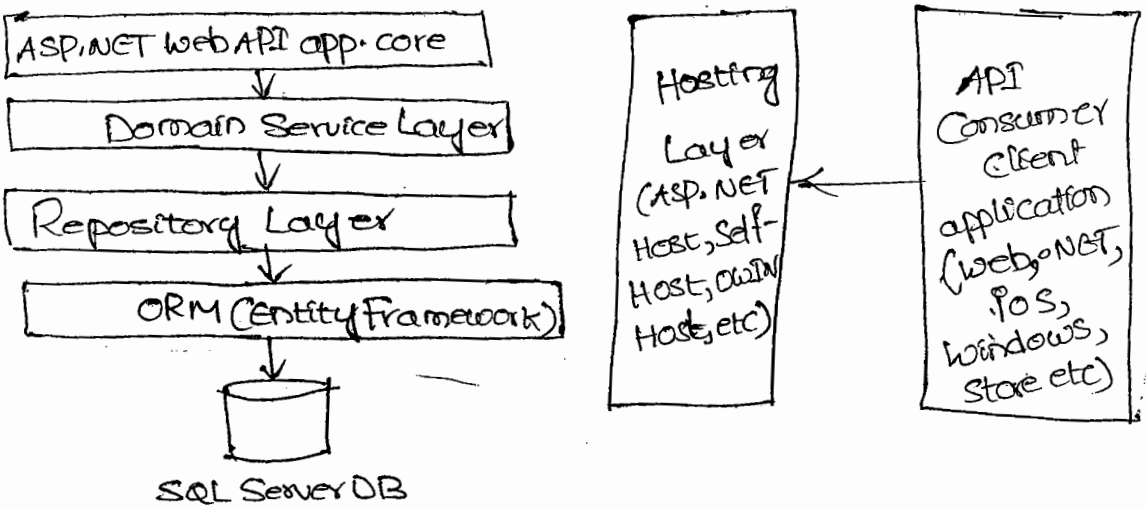
```
<h2>Chat</h2>
<div class="container">
  <input type="text" id="message" />
  <input type="button" id="sendmessage" value="Send" />
  <input type="hidden" id="displayname" />
  <ul id="discussion"></ul>
</div>
<script src="~/Scripts/jquery.signalR-2.0.3.min.js"></script>
<script src="~/Scripts/jquery-1.10.2.min.cs"></script>
<script>
    $(function(){
    var chat = $.connection.chatHub;
    chat.client.addNewMessageToPage = function(name, message){
      $('#discussion').append('<li><strong>' +
    htmlEncode(name) + '</strong>: ' + htmlEncode(message) + '</li>');
    };
$('#displayname').val(prompt('Enter your Name:', ''));
      $('#message').focus();
$.connection.hub.start().done(function(){
```

```
$('#sendmessage').click(function () {
    chat.server.send($('#displayname').val(),
    $('#message').val());
    $('#message').val('').focus();
    });
});
});
```

3/02/16 Web API:

Web Api provides service that can be consumed into various applications with different platforms like windows, web, Mobile etc.

API provides a service that can be configured in JSON or XML so that you can serialize and de-serialize the JSON or XML content



1. Creating a Web API:
   1. Create a new project
   2. Select "ASP.NET web app"
   3. Select the template as "Web API"
   4. Select the reference for API
   5. Goto Models and add model classes
      a) Conceptual
      b) Context (for communicating with DB)

6. Goto Controllers and add new Controller

7. Select Controller type as "API Controller with Read/Write Actions using Entity Framework"

Controller Name: ProductsController

Model class: Conceptual

Context class: Context

4/02/16 Ex: 8. Goto models and add a new class (Model) Product.cs

```
public class Product
{
 public int id {get;set;}
 public string Name {get;set;}
 public string Category {get;set;}
 public decimal Price {get;set;}
}
```

2. Goto controller and add → New → Controller

3. Select "Web API Controller Empty"

4. Name it as "ProductsController"

```
public class ProductsController : ApiController
{
 Product[] products = new Product[]
 {
  new Product {id=1, Name="Mobile", Category="Electronics",
               Price=1000};
  new Product {id=2, Name="TV", Category="Electronics",
               Price=43000M};
 };
public IEnumerable<Product> GetAllProducts()
    {
     return products;
    }
public IHttpActionResult GetProduct(int id)
    {
```

```
var product = products.FirstOrDefault((p) => p.Id == id);
        if (product == null)
        {
          return NotFound();
        }
        return Ok(product);
      }
  }
```

5. Add a new Action in [ProductsController] a normal controller "HomeController" (Not Api)

```
public ActionResult Index()
{
  return View();
}
```

6. Add View for Index action

Index.cshtml

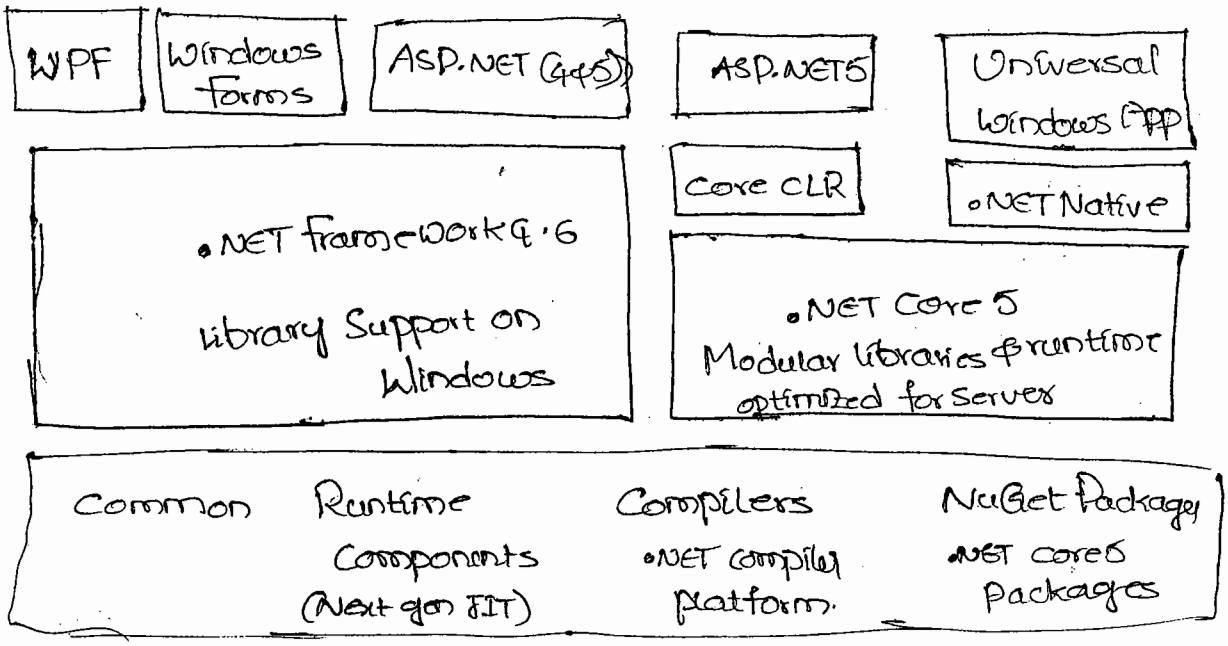
```
<!DOCTYPE html>
<html>
<head>
</head>
<body>
  <div>
    <h2>All Products</h2>
    <ul id="products" />
  </div>
  <div>
    <h2>Search by ID</h2>
    <input type="text" id="prodId" size="5" />
    <input type="button" value="Search" onclick="find()" />
    <p id="product" />
  </div>
```

```
<script src="http://ajax.aspnetcdn.com/ajax/jquery/jquery-2.0.3.
</script>                                                    min.js">
<script>
  var uri= 'api/products';
  $(document).ready(function (){
    //Send an AJAX request
    $.getJSON(uri).done(function(data) {
  //On success, 'data' contains a list of products.
  $.each(data,function(key,item) {
    // Add a list item for the product
  $('<li>', {text : formatItem(item)}).appendTo($('#products'));
    });
  });
});
function formatItem(item) { return item.Name+ ':$' +item.Price;
}
function find() { var id = $('#prodId').val();
      $.getJSON(uri+'/'+id).done(function (data) {
  $('#product').text(formatItem(data));
  })
  .fail(function (jqXHR,textStatus, err){
    $('#product').text('Error:'+err);
    });
  }
</script>
</body>
</html>
```

5/02/16

MVC 6

.NET 4.5 Architecture:



What's Open Source in 4.6?

1. ASP.NET 4 & 5
2. ASP 5
3. .NET core 5
4. Runtime Components
5. Compilers
6. Libraries (.NET core 5 Library)

What is new in ASP 5?

1. ASP 5 = Web forms + Web API + MVC
2. Modular
   - framework ships with application
3. Faster Development Life cycle
   - Same code runs on both development and Production
   - New Rosyln JIT compiler
4. Cloud Ready
   - On Permises to cloud
   - Ready to host on cloud servers

5. Cross Platform
   - New framework for MAC, LINUX
6. True side by side execution
   - Supports multiple servers like Apache, JBoss, Light PGD
   - No more DLL Hell Issues
7. Uses Agile Methodology with Azure to develop applications
8. Inbuilt support for dependency injection

Creating first MVC6 (ASP5) application:

1. Open VS2015
2. New Project → Visual C# → Web
3. ASP.NET Web Application
4. Select Template "ASP5"

New File system of MVC6 application:

Resource	Description
Solution Items	Contains global.json for application startup configuration - framework - Runtime - architecture...
Src	→ Contains projects in your solution
wwwroot	→ It is the default website application folder, where applications are hosted. It is mapped to "c:\inetput\wwwroot" It contains other resources like css, images, Js, Lib...
Dependencies	→ Collection of dependencies that Support Css and Js plug-in like NPM, Bower etc
Controllers	→ Contains controller classes

Migrations → Contains Migration history for various frameworks used in application, like WCF services, EF

Models → Contains model classes

Services → Shows the services consumed by your application. Allows to manage services.

Views → Contains Application UI

Startup.cs → Contains startup configurations like Routes, Bundles, Authentication etc.

Adding a new Controller:

1. Right click on Controllers
2. Select Add → New Item
3. Goto Server category in left side corner
4. Select "MVC controller class"
5. Name it as "ProductsController.cs"

```
using Microsoft.AspNet.Mvc;
public class ProductsController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
}
```

6. Add view for Index Action:
   1. Goto Views folder
   2. Right click on that → Add → New folder
   3. Name it as "Products"
   4. Right click on "Products" folder
   5. Select "Add New Item"

6. Goto "Server" category and select "MVC View Page"

7. Name it as "Index.cshtml"

Define Route for application: (The below syntax belong to Parallel Programming)

1. Goto Startup.cs file
2. Configure the route

```
app.UseMvc(routes =>
    {
     routes.MapRoute(
      name:"default",
template:"{controller=Products}/{action=Index}/{id?}");}
```

8/2/16

Routing:

1. Routing introduced from ASP.NET 4.5
2. It is the process of creating user friendly and SEO friendly urls.
3. MVC 5 introduces attribute

9/02/16 Areas in MVC:

Areas allow to organize the controllers, models and Views into multiple categories. So that every area will be independent and individual from other controllers and views in the appli:

This requires route configuration to be defined for the areas inside the application and registering the areas

Ex:

1. Right click on Project name and Select "Add Area" name it as "Manager"
2. This will add individual controllers, action and views for Manager area.
3. Add a new controller by name "HomeController"
4. Add index action with a View

Index.cshtml

<h1> Manager Home </h1>

5. You can access the controller action by using a HyperLink

@Html.ActionLink("Manager Home", "Index", "Home", new {area = "Manager"}, null)

6. You can set the area in the startup by specifying the route

```
routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}",
    defaults: new { controller = "Home", action = "Index",
        id = UrlParameter.Optional },
    namespaces: new [] {"MvcAreasDemo.Areas.Manager.
                        Controllers"});
```

angular JS in MVC:

-Features:

1. Open source Javascript MVC framework
2. Supports separation of concerns by using MVC design pattern
3. Built-in attributes (directives) which makes HTML dynamic
4. Easy to customize and extend
5. Uses Dependency Injection
6. Easy to unit test
7. REST friendly

Enabling Angular in ASP.NET MVC:

1. Manage NuGet Packages
2. Search for "Angular Core" and install
3. scripts/Angular.min.js

Angular Directives:

1. ng-app : Angular application
2. ng-init : Initialize angular variable
3. ng-model : Binds the control's value property
4. ng-controller : Attaches a controller to view
5. ng-bind : Binds the control values to specific expressions
6. ng-repeat : repeats HTML template
7. ng-show : shows the specified expression
8. ng-readonly : marks as readonly
9. ng-disabled : disables the control
10. ng-click : Performs actions on click.

Ex: a simple validation with angular

```
<body ng-app>
<form name="studentform" novalidate>
<label for="firstName"> First Name: </label> <br/>
<input type="text" name="firstName" ng-model="studentfirstname
```

```
    ng-required="true" />
<span ng-show="studentform.firstName.$touched &&
        studentform.firstName.$error.required">
First Name is required </span> <br/> <br/>
<label for="lastName">Last Name </label> <br/>
  <input type="text" name="lastName" ng-minLength="3"
        ng-maxLength="10" ng-model="student.lastName" />
 <br/>
<span ng-show="studentform.lastName.$error.minlength">
 min 3 chars </span>
<span ng-show="studentform.lastName.$error.maxlength">
 max 10 chars </span> <br/>
  <input type="submit" value="Save" />
  </form>
  </body>
```