# Introduction

## What is ASP.Net MVC?

ASP.NET MVC is a web application framework that gives you a powerful, MVC architecture-based way to build dynamic web applications that enables a clean separation of concerns and that gives you full control over markup.

## Main Advantages:-

* Clean separation of concerns: - that means you can develop and test the model, view and controller independently without having each other.
* Faster performance: - In ASP.Net MVC server’s controls are eliminated.

ASP.NET divides into two parts:-

1. ASP.NET Web forms
2. ASP.NET MVC

## What is MVC?

MVC is an architectural pattern that dictates to write the application code as composition of three major parts:-

1. Model

* Data structure
* Business Logic

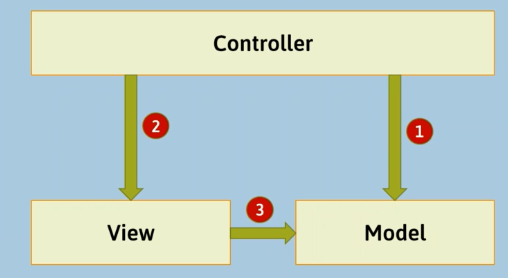
1. View

* Presentational Logic
* Reads data from Model

1. Controller

* Defines execution flow
* Execution starts from controller
* Fills data into model object
* Pass model objects to view

**Controller calls Model, Controller calls View, Views calls Model**

****

## Why ASP.Net MVC?

* Supports Clean separation of concerns.
* Supports Unit Testing.
* Supports Dependency Injection – Dependency Injection is a concept of loading business logic objects into controller dynamically rather than loading at compilation time.
* Supports faster performance than ASP.NET Web forms – ASP.NET web forms struggles with the slow performance because of heavy page life cycles, view state and post back. That means every small change the entire page need to be refresh.
* No Page Life Cycle, Controls, Postback and ViewState.

## ASP.NET MVC vs ASP.NET Web Forms

Both ASP.Net MVC and ASP.NET Web forms are the types of ASP.NET.

We can combine one project both with ASP.NET MVC and web forms, this is called hybrid application. This experiment is required particularly if you are migrating old ASP.NET Web forms into ASP.NET MVC.

1)

1. ASP.NET MVC supports clean separation of concerns. That means it allows you to develop Model, View and Controller parallel without depending on each other.
2. Asp.NET Webs forms on other hand, Presentation Logic (.aspx) and Event Logic (.aspx.cs) are tightly coupled.

2)

a) ASP.NET MVC Business logic layers, Controllers are unit testable.

b) ASP.NET Web forms Presentation and application logic is not unit testable.

3)

a) ASP.NET MVC supports dependency injection (we can inject the services into controller at run time)

b) ASP.NET Web forms does not support dependency injection

4)

a) ASP.NET MVC faster performance than ASP.NET Web Forms (No Page Life Cycle, Controls, Postback and ViewState)

b) ASP.NET Web forms are slower performance than ASP.NET MVC because of (Page Life Cycle, Server Controls, Postback and ViewState)

5)

a) ASP.NET MVC runs based on the principle of “Web is stateless”

b) ASP.NET Web forms hide the “Stateless nature of web” and try to mask the developers as they are in “Stateful environment”.

# Getting Started

## Creating First MVC App

* Creating new controller always name should be suffix with Controller like **HomeController.**
* Always include the namespace **System.Web.Mvc.**
* View must be present in the **dedicated Controller name folder like Home** under Views folder
* As along we are using razor view we have flow the below rules :

1. **View file name and the Action name should match** ( but if you give view name as different then in return View(“specify the ViewName”)
2. **Controller name and the folder name in views folder must be same**.

## Add Bootstrap to MVC Project

From NuGet Package Manager Console install the below package:

1. install-package jquery
2. install-package Popper.js
3. install-package Bootstrap

## Folder Structure of MVC App

* \**App\_Start**:- contains the files that need to be executed on the first request. Eg RouteConfig.cs, WebApiConfig.cs etc.
* **\App\_Data**:- contains SQL Server LocalDb Database files.
* \**Controllers**:- contains all controllers.
* \**Models**:- contains all models.
* \**Views**:- contains all views.
* \**Views\Web.config**:- contains configuration settings for all views.
* \**Global.asax**:- Contains application level and session level events
* \**Packages.config**:- Contains the list of NuGet packages currently installed in the project.
* \**Web.config**:- Contains web application configuration settings, that needs to be initialized on each request.

# Controllers and Action Methods

## Controllers

* Controller is a class that defines execution flow in MVC application.
* Controller receives request from browser, call the model, and call the view.
* Optionally, it is public class so that it can be call from Unit Test project.
* **Controller should be inherited from “System.Web.MVC.Controller” class**
* **Controller’s name should have suffix** “Controller”. Eg:- Home**Controller**

This is the process of ASP.NET MVC framework execution flow



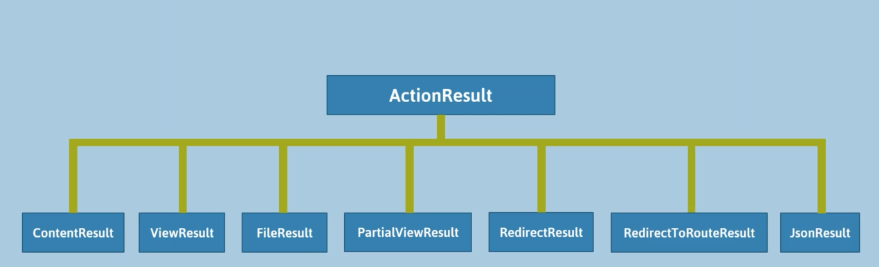
## Action Methods

* All the methods that are present in controller by default are treated as Action Method.
* Action Methods performs some operation on the database tables such as retrieving the data from database or insert, update or delete, and then return the appropriate result back to the browser
* **It is recommended to specify the return type of Action Method as Action Result**.

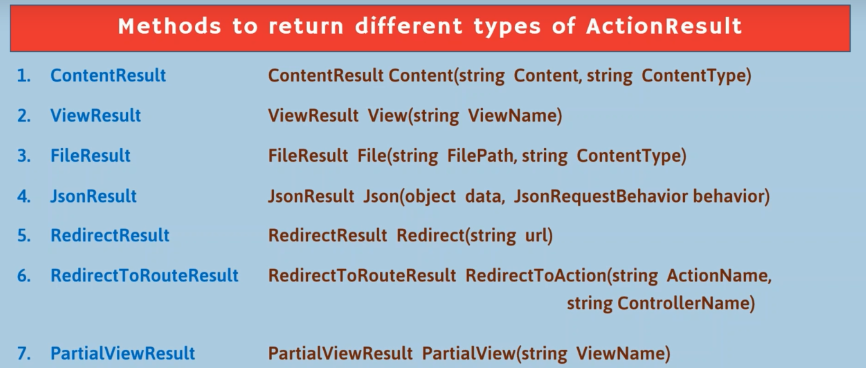
# ActionResult

## ActionResult

* ActionResult is a class that represents “result of action method”.
* Asp.Net MVC recommends specifying the return type of Action Method as “ActionResult”. It has two advantages :-
  + We can identify that it is an Action Method and it is able to receive the request from browser.
  + We can return any type of response like File, Content, RedirectToAction, JSON etc.
* ActionResult is an abstract class that has several child classes; you can return an object of any of the child classes. Below are the child classes :-



* **ContentResult**:- Represents any content with specific content-type
* **ViewResult**: - Represents result of a view. Use ViewResult class to send render HTML output of the view to the browser.
* **FileResult: -** Represents content of a file. Use FileResult class to send content of the class to the browser.
* **JsonResult: -** Represents json object/json array.
* **RedirectResult:** - Represents redirection to other websites (HTTP 302).
* **RedirectToRouteResult:** - Represents redirection to a specific action method (HTTP 302).
* **PartialViewResult**: - Represents the result of partial view. **This is possible when we are using child action method.**



## ContentResult

* Use ContentResult if we want some content along with content type, type can be text/plain, text/html, text/xml, text/css, application/pdf etc.

|  |
| --- |
| public ActionResult GetEmpName(int EmpId)  {  var employees = new[]  {  new {EmpId= 1,EmpName= "Jackie", Salary= "87772"},  new {EmpId= 2, EmpName= "Bala", Salary= "65662"},  new {EmpId= 3, EmpName= "Tammy", Salary= "112232"}  };  string matchEmpName = string.Empty;  foreach(var item in employees)  {  if(item.EmpId == EmpId)  {  matchEmpName = item.EmpName;  }  }  // return new ContentResult() { Content= matchEmpName, ContentType= "text/plain" };  // Below statement alternative is predefined method content();  return Content(matchEmpName, "text/plain");  } |

Here no need to create view for this action Methods.

Now pass the EmpId via query string in url

<http://localhost:60770/Home/GetEmpName?empId=2> . Output: - Bala

## FileResult

Use FileResult if we want a file response back to browser

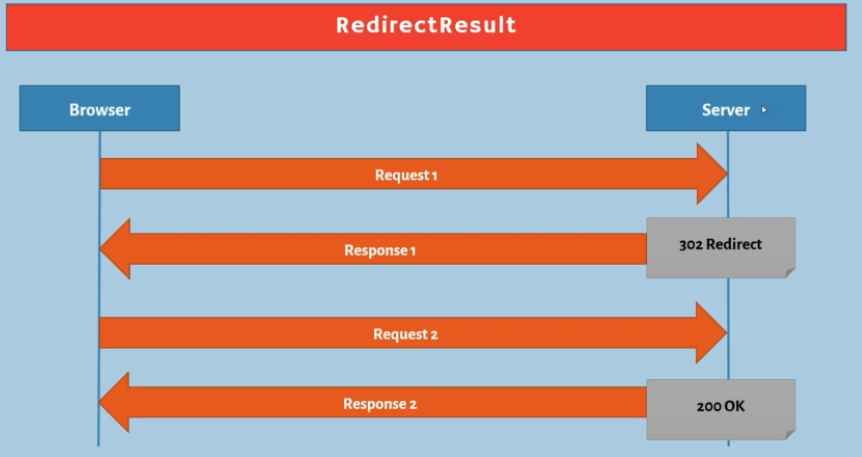
|  |
| --- |
| public ActionResult GetPaySlip(int EmpId)  {  // Here ~/ is used to refer to the root folder  string fileName = "~/PaySlip" + EmpId + ".pdf";  return File(fileName, "application/pdf");  } |

Now Pass the EmpId to get the particular PaySlip

<http://localhost:60770/Home/GetPaySlip?empId=2> Output: - The file PaySlip2.pdf will download in browser.

## RedirectResult

* This class’s object represents redirection from an action method to other website
* It sends HTTP 302 response to the browser; so the browser sends another request to the specific URL.
* The Redirect () method creates and returns an object of “RedirectResult” class.
* Syntax: return Redirect(“url”)



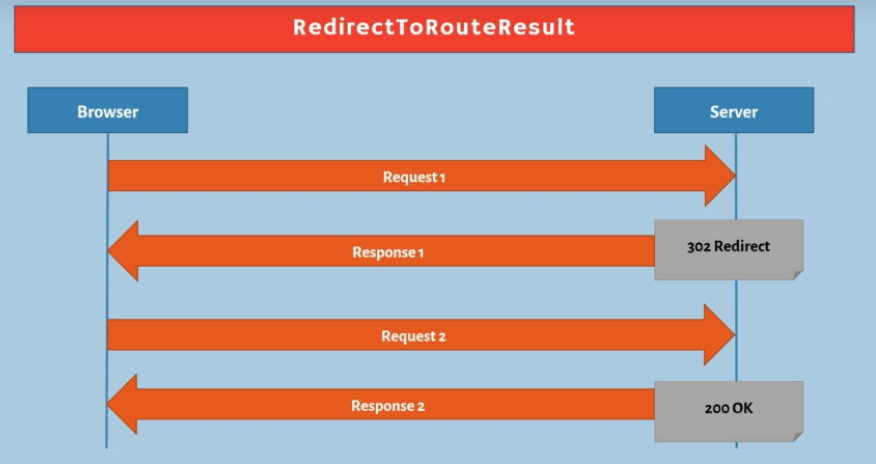
|  |
| --- |
| public ActionResult EmpFacebookPage(int EmpId)  {  // By using return type as ActionResult, we can return any type of ActionResult like (File,content etc)  // That's why it is suggestable to use return type as ActionResult  var employees = new[]  {  new {EmpId= 1,EmpName= "Jackie", Salary= "87772"},  new {EmpId= 2, EmpName= "Bala", Salary= "65662"},  new {EmpId= 3, EmpName= "Tammy", Salary= "112232"}  };  string fburl = null;  foreach(var item in employees)  {  if(item.EmpId == EmpId)  {  fburl = "http://www.facebook.com/emp" + EmpId;  }  }  if (fburl == null)  {  return Content("Invalid EmpId", "text/plain");  }  else  {  // Since facebook is otherwebsite we have to use Redirect  return Redirect(fburl);  }  } |

<http://localhost:60770/Home/GetPaySlip?empId=2> Output: - Page will redirect to Facebook website.

<http://localhost:60770/Home/GetPaySlip?empId=4> Output: - Invalid EmpId.

## RedirectToRouteResult

* This class’s object represents redirection from one action method to another action method.
* It sends HTTP 302 response to the browser; so the browser sends another request to the specific action method
* The RedirectToAction() method creates and returns an object of “RedirectToRouteResult” class.
* Syntax: return RedirectToAction(“action name”,”controller name”);
* **Here Action1 will raise a request if not found, another request for Action2 will raise. So indirectly Action1 is dependent on Action2.**

****

**Below is the scenario will require to use RedirectToAction () method class**

|  |
| --- |
|  |

**In HomeController.cs**

|  |
| --- |
| public ActionResult Login(string Username, string Password)  {  if(Username == "admin" && Password=="manager")  // Here we are redirecting to ActionMethod of other controller (Admin)  return RedirectToAction("Dashboard", "Admin");  else  // Here we are redirecting to ActionMethod of same controller, so no Controller name  return RedirectToAction("InvalidLogin");  }  public ActionResult InvalidLogin()  {  return View();  } |

**In AdminController.cs**

|  |
| --- |
| public class AdminController : Controller  {  // GET: Admin  public ActionResult Dashboard()  {  return View();  }  } |

[**http://localhost:60770/Home/Login?Username=admin&Password=manager**](http://localhost:60770/Home/Login?Username=admin&Password=manager)

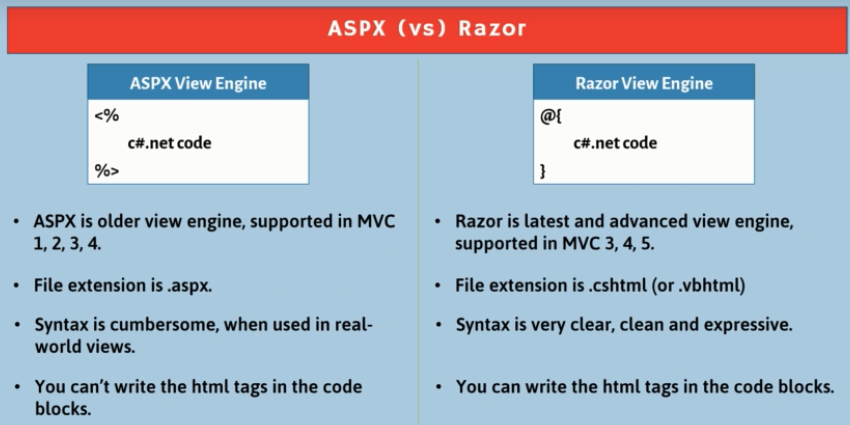
**In real-time scenario we are not using query, but will HTTP forms**

Output: - If the username and password match it will transfer to dashboard view, else Invalid login view.

# Razor View Engine

## View Engines

* In View we write both HTML Code and C# code together, then how the ASP.NET MVC knows which one is HTML Code and which code on is C# code. So View Engines helps on that.
* For Example in Razor View for differentiate C# code we use @{ C# code } , so if this format then it is C# code which needs to be execute in server.
* **View Engine provides a set of syntaxes to write C#.net code (Server side code) in View**
* View Engine is also responsible to render the view as HTML
* ASP.NET MVC supports two types of View Engines:-
  + ASPX View Engine
  + Razor View Engine



## Razor Expressions

In order to pass data from Controller to View, we can use ViewBag and it’s a dynamic object

In Controller:-

|  |
| --- |
| public ActionResult StudentDetails()  {  // These 3 details will be available in the particular View  ViewBag.StudentId = 101;  ViewBag.StudentName = "Jeet";  ViewBag.Marks = "87";  return View();  } |

In View

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta name="viewport" content="width=device-width" />  <title>StudentDetails</title>  </head>  <body>  <div>  <table class="table">  <tr>  <td>Student ID</td>  <td>@ViewBag.StudentId</td>  </tr>  <tr>  <td>Student Name</td>  <td>@ViewBag.StudentName</td>  </tr>  <tr>  <td>Student Marks</td>  <td>@ViewBag.Marks</td>  </tr>  </table>  </div>  </body>  </html> |

## Razor Code Blocks

Whenever we have bunch of C# code, then we should use Razor code blocks

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta name="viewport" content="width=device-width" />  <title>StudentDetails</title>  </head>  <body>  @{  string StudentResult = string.Empty;  if(ViewBag.Marks >= 35 )  {  StudentResult = "Pass";  }  else  {  StudentResult = "Fail";  }  }  <div>  <table class="table">  <tr>  <td>Student ID</td>  <td>@ViewBag.StudentId</td>  </tr>  <tr>  <td>Student Name</td>  <td>@ViewBag.StudentName</td>  </tr>  <tr>  <td>Student Marks</td>  <td>@ViewBag.Marks</td>  </tr>  <tr>  <td>Result</td>  <td>@StudentResult</td>  </tr>  </table>  </div>  </body>  </html> |

## Razor If

**We can write If block in Razor and inside if block we can write multiple C# and HTML code. Basically HTML Code is identify by < tag**

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta name="viewport" content="width=device-width" />  <title>StudentDetails</title>  </head>  <body>  @{  string StudentResult = string.Empty;  if(ViewBag.Marks >= 35 )  {  StudentResult = "Pass";  }  else  {  StudentResult = "Fail";  }  }  <div>  <table class="table">  <tr>  <td>Student ID</td>  <td>@ViewBag.StudentId</td>  </tr>  <tr>  <td>Student Name</td>  <td>@ViewBag.StudentName</td>  </tr>  <tr>  <td>Student Marks</td>  <td>@ViewBag.Marks</td>  </tr>  <tr>  <td>Result</td>  <td>@StudentResult</td>  </tr>  <tr>  <td colspan="2">  @if(StudentResult == "Pass")  {  <span class="text-success">Congrats!</span>  }  else  {  <span class="text-danger">Better luck next time!</span>  }  </td>  </tr>  </table>  </div>  </body>  </html> |

## Razor For

* If we want to iterate some value in View we have to use Razor For

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta name="viewport" content="width=device-width" />  <title>StudentDetails</title>  </head>  <body>  @{  string StudentResult = string.Empty;  if(ViewBag.Marks >= 35 )  {  StudentResult = "Pass";  }  else  {  StudentResult = "Fail";  }  }  <div>  <table class="table">  <tr>  <td>Student ID</td>  <td>@ViewBag.StudentId</td>  </tr>  <tr>  <td>Student Name</td>  <td>@ViewBag.StudentName</td>  </tr>  <tr>  <td>Student Marks</td>  <td>@ViewBag.Marks</td>  </tr>  <tr>  <td>Result</td>  <td>@StudentResult</td>  </tr>  <tr>  <td>Semesters</td>  <td>  @for(int i =1;i<=ViewBag.NoOfSemesters;i++)  {  <span>@i</span>  }  </td>  </tr>  </table>  </div>  </body>  </html> |

## Razor Foreach

* Use Razor Foreach if you want to read the data from array or collection. For this you need to import Sytem.Collections.generic in controller

In Controller we are declaring the collection

|  |
| --- |
| public ActionResult StudentDetails()  {  // These 3 details will be available in the particular View  ViewBag.StudentId = 101;  ViewBag.StudentName = "Jeet";  ViewBag.Marks = 87;  ViewBag.NoOfSemesters = 8;  ViewBag.Subjects = new List<string>() { "Maths", "Chemistry", "Physics" };  return View();  } |

In View Foreach as highlighted

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta name="viewport" content="width=device-width" />  <title>StudentDetails</title>  </head>  <body>  @{  string StudentResult = string.Empty;  if(ViewBag.Marks >= 35 )  {  StudentResult = "Pass";  }  else  {  StudentResult = "Fail";  }  }  <div>  <table class="table">  <tr>  <td>Student ID</td>  <td>@ViewBag.StudentId</td>  </tr>  <tr>  <td>Student Name</td>  <td>@ViewBag.StudentName</td>  </tr>  <tr>  <td>Student Marks</td>  <td>@ViewBag.Marks</td>  </tr>  <tr>  <td>Result</td>  <td>@StudentResult</td>  </tr>  <tr>  <td colspan="2">  @if(StudentResult == "Pass")  {  <span class="text-success">Congrats!</span>  }  else  {  <span class="text-danger">Better luck next time!</span>  }  </td>  </tr>  <tr>  <td>Semesters</td>  <td>  @for(int i =1;i<=ViewBag.NoOfSemesters;i++)  {  <span>@i</span>  }  </td>  </tr>  <tr>  <td>Subject</td>  <td>  @foreach(var item in ViewBag.Subjects)  {  <li>@item</li>  }  </td>  </tr>  </table>  </div>  </body>  </html> |

# HTTP

## HTTP

* HTTP is an application protocol that defines set of rules to send request from browser to server and send response from server to browser.

|  |
| --- |
|  |

## HTTP Request Message Format

|  |
| --- |
|  |

## HTTP Methods

* **GET**
  + Used to retrieve data from server: Ex: Search.
  + Parameters will be appended to the url as “query string”

Eg: - <http://localhost:portno?Parameter=value&Parameter=value&>...

* + Parameters will be displayed in the address bar of the browser.
  + Can pass only limited no. of characters only. Maximum no. of characters per URL: 2048 (depends on browser).
  + Can be cached by the browser / search engines.
  + Can pass only ASCII characters; can’t pass Unicode characters.
* **POST**
  + Used to perform insert / update / delete operations.
  + Parameters are not in URL, but will be sending in “request body”.
  + Parameters will not be displayed in the addresses bar of the browser, as it is sending in “request body”.
  + Can pass unlimited data as the parameters are passed in the request message body.
  + Can’t be cached by the browser / search engines.
  + Supports Unicode characters / binary data.

## HTTP Request Headers

* We have so many Request Headers, but below are the important ones.

|  |
| --- |
|  |

## HTTP Response Message Format

|  |
| --- |
|  |

## HTTP Response Headers

Below are the most commonly used response headers.

|  |
| --- |
|  |

## HTTP MIME Types

* HTTP MIME Types are used for the type of the content that is transferring from server to client or client to server.
* It is used in Content-Type header in request and response message.

Eg: - **text/plain, text/html, text/css, text/javascript, text/xml, application/json, image/png, image/jpeg, audio/mp3**

## HTTP Status code

* **100**: Continue (browser is instructed to send another message to the server).
* **200**: Ok (Process is completed successfully on the server).
* **302**: Redirection (Server is instructing to browser to send another request to another url ).
* **304**: Not Modified (Data is already cached in the browser memory).
* **400**: Bad Request (Invalid message request format).
* **401**: Unauthorized (Username or password or any other credentials is not valid)
* **404**: Not Found (URL is wrong).
* **500**: Internal server error.

## Request Object

* Request is a predefined object which contains request details sent from browser to server.
* It is the member of “System.web.Mvc.Controller” class.
* **Request object will be present in all controllers by default.**

## Request Properties

* **Request.Url**: Represents current URL. Ex: <http://localhost:1020/home/index>.
* **Request.PhysicalApplicationPath**: Represents current folder path. Ex: c:\mvc\MyFirstApp.
* **Request.Path**: Represents current virtual path. Ex: /home/index.
* **Request**.**Browser.Type:** Represents current browser name. Ex: Chrome89
* **Request.QueryString**: Represents current query string. Ex: Param1=Value1&Param2=Value2.
* **Request.Headers**: Represents request header that are pass from browser to server.
* **Request.HttpMethod**: Represents Http Method of the request (Get/Post).

|  |
| --- |
| public ActionResult RequestExample()  {  ViewBag.Url = Request.Url;  ViewBag.PhysicalApplicationPath = Request.PhysicalApplicationPath;  ViewBag.Path = Request.Path;  ViewBag.Type = Request.Browser.Type;  ViewBag.QueryString = Request.QueryString["n"];  ViewBag.Headers = Request.Headers["Accept"];  ViewBag.HttpMethod = Request.HttpMethod;  return View();  } |

## Response Object

* It is a predefined object which contains the response data that is need to be send from server to browser.
* It is the member of “System.web.Mvc.Controller” class, which is the common parent class for all the controllers.
* **Response object will be present in all controllers by default.**

## Response Properties

* **Response.Write()**: Sends given content to the browser. If we use multiple Response.Write() it will send together at once.
* **Response.ContentType**: Represents type of response content. Eg: text/html.
* **Response.Headers**: Represents response headers that can be sent from server to browser.
* **Response.StatusCode**: Represents Status of request. Eg: 200,302 etc.

# Shared Views

## Shared Views

* Shared views are present in the “**Views\Shared**” folder.
* Shared views are the views that can be called from any controller of the entire project.
* The views that belong to all controllers are created as shared views.
* When we call a view, it checks the view in the “**Views\Controllername**” folder first; if it is not found, it will search in the “**Views\Shared**” folder.

|  |
| --- |
|  |

## Passing Data Dynamically to Shared View

* Scenario is wanted to see different values in Shared View, when call from different controller. In that case we need to pass dynamic values to the shared view by using viewbag

In Shared View

|  |
| --- |
| <div>  <h1>Hello I am from Shared View :) !..</h1>  <h2>TollFree Number: @ViewBag.TollFreeNumber</h2>  </div> |

# Layout Views

## Layout Views with Bootstrap

* Layout views contain “page template”, which contains common parts of the UI, such as logo, header, menubar, side bar etc.
* **@RenderBody()** method represents the reserved area for the actual content of view.
* **Execution Flow: - Controller -> View -> Layout View -> Rendered View Result -> Send response to browser.**
* **We can also create Multiple Layout Views in same project.**
* **If you mention in single View, multiple Layout Views then the Layout Views which is defined at last will be execute.**
* One View can have one Layout View.
* **@RenderBody() must present in Layout View, otherwise it will not consider as Layout View**.
* **It is recommended to place the Layout views in** “**Views\Shared**” folder.

|  |
| --- |
|  |

* Below is the highlighted code in View to access Layout view

|  |
| --- |
| @{  ViewBag.Title = "Index";  Layout = "~/Views/Shared/\_LayoutPage1.cshtml";  }  <h2>Index- Home Page Content</h2> |

* Below is the Sample Layout view with Bootstrap.
* In highlighted **@ViewBag.Title** in below Layout View, will be retrieve from the view as mentioned in above (ViewBag.Title = "Index";)
* In highlighted **@RenderBody** as below will retrieve the data from above as

<h2>Index- Home Page Content</h2>

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <link href="~/Content/bootstrap.css" rel="stylesheet" />  <script src="~/Scripts/umd/popper.js"></script>  <script src="~/Scripts/jquery-3.3.1.js"></script>  <script src="~/Scripts/bootstrap.js"></script>  <meta name="viewport" content="width=device-width" />  <title>@ViewBag.Title</title>  </head>  <body>  <nav class="navbar navbar-expand-sm bg-info navbar-dark">  <a class="navbar-brand" href="#">Company</a>  <div class="col-8">  <ul class="navbar-nav">  <li class="nav-item">  <a class="nav-link" href="/home/index">Home</a>  </li>  <li class="nav-item">  <a class="nav-link" href="/home/about">About</a>  </li>  <li class="nav-item">  <a class="nav-link" href="/home/contact">Contact</a>  </li>  </ul>  </div>  </nav>  <div class="container-fluid">  @RenderBody()  </div>  </body>  </html> |

## Sharing Data from View to Layout View

**We always know that before executing Layout View, normal view will always execute.**

**So if we want to pass data from View to Layout View, so for that from the View we have to assign the value in ViewBag and then have to access the same ViewBag object in Layout view.**

## Creating Sidebar in Layout View

We use bootstrap to create side bar in Layout View as highlighted below

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <link href="~/Content/bootstrap.css" rel="stylesheet" />  <script src="~/Scripts/umd/popper.js"></script>  <script src="~/Scripts/jquery-3.3.1.js"></script>  <script src="~/Scripts/bootstrap.js"></script>  <meta name="viewport" content="width=device-width" />  <title>@ViewBag.Title</title>  </head>  <body>  <nav class="navbar navbar-expand-sm bg-info navbar-dark">  <a class="navbar-brand" href="#">Company</a>  <div class="col-8">  <ul class="navbar-nav">  <li class="nav-item">  <a class="nav-link" href="/home/index">Home</a>  </li>  <li class="nav-item">  <a class="nav-link" href="/home/about">About</a>  </li>  <li class="nav-item">  <a class="nav-link" href="/home/contact">Contact</a>  </li>  </ul>  </div>  <div class="col-4 text-white">  @ViewBag.Message  </div>  </nav>  <div class="container-fluid">  <div class="col-2 text-white" style="background-color:#fa8405;min-height:1050px;padding:0px">  <div class="list-group">  <a href="#" class="list-group-item text-white" style="background-color:transparent">Favourites</a>  <a href="#" class="list-group-item text-white" style="background-color:transparent">Products</a>  <a href="#" class="list-group-item text-white" style="background-color:transparent">Your Orders</a>  </div>  </div>  <div class="col-10" style="min-height:610px">  @RenderBody()  </div>  </div>  </body>  </html> |

## Sections in Layout Views

* **Sections are used to display view-specific content in layout view.**
* Sections are defined in view and rendered in layout view.
* Can render different content for the same section in different views.
* **In Layout view can use multiple sections with different names, if don’t want to specify the content for one or more views for that, required:false need to be use as a second argument in render section method, which makes section as optional that means for every view no need to send the content for that particular section.**

|  |
| --- |
| @RenderSection("sideBarOptions",required:false) |

|  |
| --- |
|  |

**In Index View**

|  |
| --- |
| @section sideBarOptions  {  <a href="#" class="list-group-item" style="background-color:transparent">Admin Home</a>  <a href="#" class="list-group-item" style="background-color:transparent">Agent Home</a>  <a href="#" class="list-group-item" style="background-color:transparent">Customer Home</a>  } |

**In Layout View**

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <link href="~/Content/bootstrap.css" rel="stylesheet" />  <script src="~/Scripts/umd/popper.js"></script>  <script src="~/Scripts/jquery-3.3.1.js"></script>  <script src="~/Scripts/bootstrap.js"></script>  <meta name="viewport" content="width=device-width" />  <title>@ViewBag.Title</title>  </head>  <body>  <nav class="navbar navbar-expand-sm bg-info navbar-dark">  <a class="navbar-brand" href="#">Company</a>  <div class="col-8">  <ul class="navbar-nav">  <li class="nav-item">  <a class="nav-link" href="/home/index">Home</a>  </li>  <li class="nav-item">  <a class="nav-link" href="/home/about">About</a>  </li>  <li class="nav-item">  <a class="nav-link" href="/home/contact">Contact</a>  </li>  </ul>  </div>  <div class="col-4 text-white">  @ViewBag.Message  </div>  </nav>  <div class="container-fluid">  <div class="col-2 text-white" style="background-color:#fa8405;min-height:1050px;padding:0px">  <div class="list-group">  <a href="#" class="list-group-item text-white" style="background-color:transparent">Favourites</a>  <a href="#" class="list-group-item text-white" style="background-color:transparent">Products</a>  <a href="#" class="list-group-item text-white" style="background-color:transparent">Your Orders</a>  @RenderSection("sideBarOptions",required:false)  </div>  </div>  <div class="col-10" style="min-height:610px">  @RenderBody()  </div>  </div>  </body>  </html> |

## \_ViewStart.cshtml

* **It defines the default layout view for all the views of a folder.**
* **If it is present in “Views” folder, it defines the default layout view of all the views of entire project.**
* **If it is present in “Views\Controllername” folder, it defines the default layout view of all the views of same controller only.**
* **Flow of Execution: Controller -> \_ViewStart.cshtml of “Views” folder -> \_ViewStart.cshtml of “Controller1” folder -> View -> Layout View -> Generate View Result -> Response.**
* **The main advantage of \_ViewStart.cshtml is not to specify the path of Layout view in every view individually.**

|  |
| --- |
|  |

## Multiple Layout Views

* We have to create two or more Layout Views in Shared folder.
* Layout1 mapped via Views/\_ViewStart.cshtml and Layout2 mapped via Views/Controller1/\_ViewStart.cshtml.
* Now for the specify View present in controller1 will be mapped to Layout2 View.
* And other controller2... Will be mapped to Layout1 View or we can change the Layout View specifically in any Views of Controller1.

## Partial Views

* Partial View is a small view that contains content that can be shared among multiple views.
* **Partial View avoids the repetition of same presentational logic in multiple views**, eg: - **We want to display list of employee as a table and want to display the same table in different views. So for that instead of repeating the code in every view, we can create a partial view and invoke the same in any view anywhere.**
* Can be present in “View\Controlllername” folder or in “View\Shared” folder.

|  |
| --- |
|  |

|  |
| --- |
|  |

**In PartialView (ListPartialView.cshtml)**

|  |
| --- |
| <div>  <h3>@ViewBag.ListTitle</h3>  <ul>  @for(int i=0;i<ViewBag.items.Count;i++)  {  <li>@ViewBag.items[i]</li>  }  </ul>  </div> |

**In View**

|  |
| --- |
| @{  ViewBag.ListTitle = "Contact Details";  ViewBag.items = new List<string>() { "Business Team", "Support Team", "Technical Team" };  **Html.RenderPartial("ListPartialView");**  } |

# URL Routing

## URL Routing

* URL Routing is a pattern matching system that monitors the incoming request URL and figure out what to do.
* It allows you to create meaningful URLs, instead of mapping to physical files on the server.
* Advantages:
  + **Makes the URL not to map to physical files on the server**

Old URL e: /folder/subfolder/search.aspx

* + **URLs are user friendly**
  + **URLs are search-engine friendly (they can be cached by search engine like google etc.) so the website get more visitors.**

## Route

* Route is a URL pattern which includes literals / parameters.
* Literal is a fixed text that must be present in the URL.
* Parameter is a variable, which value can be entered by the user.
* All the routes are stored in “Routing table”, which will be stored in (RAM).

|  |
| --- |
|  |

|  |
| --- |
|  |

## Understanding Default Routing

|  |
| --- |
| public static void RegisterRoutes(RouteCollection routes)  {  routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");  routes.MapRoute(  name: "Default",  url: "{controller}/{action}",  defaults: new { controller = "Home", action = "Index"}  );  } |

routes.IgnoreRoute("{resource}.axd/{\*pathInfo}"); Is used to ignore trace file as we know trace file in ASP.NET MVC is saved as trace.axd.

name: "Default" you can give any name but the name should not be repeat for other routes

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

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

Eg

|  |  |
| --- | --- |
| url | Redirect View |
| http://localhost/ | Home/Index |
| <http://localhost/Home> | Home/Index |
| <http://localhost/Home/Index> | Home/Index |
| <http://localhost/Product> | Product/Index |
| <http://localhost/Product/Contact> | Product/Contact |

**If not pass value for controller and Action Methods – it will take the value from default which is**

**Home /Index as highlighted above.**

## ID Parameter

|  |
| --- |
| public static void RegisterRoutes(RouteCollection routes)  {  routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");  routes.MapRoute(  name: "Default",  url: "{controller}/{action}/{id}",  defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }  );  } |

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

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

**Here Id is optional as we mention in defaults, Id value can be integer or string**

**If the URL is** [**http://localhost/Product/Details/1**](http://localhost/Product/Details/1) **output will be Nokia**

|  |
| --- |
| public ActionResult Details(int? id)  {  var products = new[]  {  new {productId = 1, ProductName = "Nokia", cost = "78982"},  new {productId = 2, ProductName = "MI", cost = "24452"},  new {productId = 3, ProductName = "OnePlus", cost = "23413"}  };  if (id == null)  {  return Content("Please pass any product id");  }  else  {  string prodName = string.Empty;  foreach (var item in products)  {  if (item.productId == id)  {  prodName = item.ProductName;  }  }  return Content(prodName);  }  } |

**For accepting null in parameter we have to use int? as highlighted above and Null check is very important.**

## ID as String Parameter

Here every will be same as previous example of ID as Integer Parameter, **but NULL check is very important**

|  |
| --- |
| public ActionResult GetProductId(string id)  {  var products = new[]  {  new {productId = 1, ProductName = "Nokia", cost = "78982"},  new {productId = 2, ProductName = "MI", cost = "24452"},  new {productId = 3, ProductName = "OnePlus", cost = "23413"}  };    if (id == null)  {  return Content("Please pass any product name");  }  else  {  int prodId = 0;  foreach (var item in products)  {  if (item.ProductName.ToUpper() == id.ToUpper())  {  prodId = item.productId;  }  }  return Content(prodId.ToString());  }  } |

## Resolving Conflicts between Routes

In above we should use id as a parameter because in Route has defined as url: "{controller}/{action}/{id}" so it will accept only id as a name only in a parameter. So if we want different name in parameter for that we have to create one more route

|  |
| --- |
| public static void RegisterRoutes(RouteCollection routes)  {  routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");  routes.MapRoute(  name: "Products",  url: "products/GetProductId/{productName}",  defaults: new { controller = "products", action = "GetProductId" }  );  routes.MapRoute(  name: "Default",  url: "{controller}/{action}/{id}",  defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }  );  } |

**But it will create conflicts if we don’t change the order i.e default route should be at end and specifically mention the controller and action name specifically** url: "products/GetProductId/{productName}", **and in defaults we need to mention the controller and action name** defaults: new { controller = "products", action = "GetProductId"}

So for that how many times we need to write route for different parameter name. So for that in ASP.NET MVC5 **attribute route** is introduced.

## Route Constraints

In the previous section we saw how to avoid conflicts in Routes by using literal, or by using constraint also we can do as highlighted below:

|  |
| --- |
| public static void RegisterRoutes(RouteCollection routes)  {  routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");  routes.MapRoute(  name: "Products",  url: "{controller}/{action}/{productName}",  defaults: new {},  constraints: new { productName = @"^[A-Za-z]\*$"}  );  routes.MapRoute(  name: "Default",  url: "{controller}/{action}/{id}",  defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }  );  } |

# Attribute Routing

## Introduction to Attribute Routing

* Attribute Routing is introduced to overcome the problems in convention routing.

**Problems in Convention Routing**

* Very difficult to understand for the developers, which route is for which action methods?
* Very difficult to avoid conflicts among the routes [ sometimes , we can’t apply constraints]
* Overall, some routes for multiple action methods; some routes for specific action methods; overall, it looks cumbersome.

|  |
| --- |
|  |

## Route Prefixes

* Route Prefixes are nothing but the prefix for any route that we want to apply, all we need to do is to define the route prefix on a controller so that all the action methods inside it can follow the prefix.  
    
  For example,

1. [RoutePrefix("products")]
2. **public** **class** ProductController : Controller
3. {
4. //This will be translated to /products
6. [Route]
7. **public** ActionResult List()
8. {
9. **return** View();
10. }
12. //This will be translated to /products/2
14. [Route("{id?}")]
15. **public** ActionResult Details(**string** id)
16. {
17. **if** (**string**.IsNullOrEmpty(id))
18. {
19. **return** View("List");
20. }
22. **return** View("Details");
23. }

## Attribute Routing Example

* [Route("Products/Details/{id:int?}")] = http//localhost/Products/details/1 and id:int? **Means it is optional and accepts only integer value.**
* [Route("Products/GetProductId/{productName}")] = = http//localhost/Products/GetProductId/Nokia and productName? **Means it is optional (for string by default it is optional, so need not to use datatype and ?) and accepts only string value**
* Default Route in Attribute Routing is used by using empty Route(“”) as highlighted.

|  |
| --- |
| [Route("Home/Index")]  [Route("")]  public ActionResult Index()  {  return View();  } |

|  |
| --- |
|  |

# MODELS

* Model is a class that defines structure of the data that you want to store/display.
* Also contains business logic.
* Model will be called by Controller and View.

**Types of Model:-**

|  |
| --- |
|  |

## Creating List of Products

**Step 1:**

Create a class File in Model Folder and name as Products.cs**. And here we have use both View Model and Domain Model in same class; In Entity Framework section will separate both the classes.**

|  |
| --- |
| namespace ModelExample.Models  {  public class Product  {  public int ProductId { get; set; }  public string ProductName { get; set; }  public decimal Rate { get; set; }  }  } |

**Step 2:**

**Add the namespace of model in Controller class to use the class Product**

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.Web.Mvc;  using ModelExample.Models;  namespace ModelExample.Controllers  {  public class ProductsController : Controller  {  // GET: Products  public ActionResult Index()  {  List<Product> products = new List<Product>()  {  new Product() {ProductId = 101, ProductName= "Nokia", Rate = 2938},  new Product() {ProductId = 102, ProductName = "MI", Rate= 8746},  new Product() {ProductId = 103, ProductName = "OnePlus", Rate = 9876}  };  ViewBag.products = products;  return View();  }  }  } |

**Step 3:**

**Create the view and retrieve the products value via ViewBag object**

|  |
| --- |
| @{  ViewBag.Title = "Index";  }  <h2>Products</h2>  <table class="table">  <thead>  <tr>  <th>ProductId</th>  <th>ProductName</th>  <th>Rate</th>  </tr>  </thead>  <tbody>  @foreach(var item in ViewBag.Products)  {  <tr>  <td>@item.ProductId</td>  <td>@item.ProductName</td>  <td>@item.Rate</td>  </tr>  }  </tbody>  </table> |

## Creating Products Details

**In Controller**

**Here id is passed from Product view**

|  |
| --- |
| public ActionResult Details( int id)  {  List<Product> products = new List<Product>()  {  new Product() {ProductId = 101, ProductName= "Nokia", Rate = 2938},  new Product() {ProductId = 102, ProductName = "MI", Rate= 8746},  new Product() {ProductId = 103, ProductName = "OnePlus", Rate = 9876}  };  Product matchingProduct = null;  foreach(var item in products)  {  if(item.ProductId == id)  {  matchingProduct = item;  }  }  ViewBag.matchingProduct = matchingProduct;  return View();  } |

In details View

|  |
| --- |
| @{  ViewBag.Title = "Details";  }  <h2>Product Details</h2>  <table class="table">  <tr>  <th>Product Id</th>  <td>@ViewBag.matchingProduct.ProductId</td>  </tr>  <tr>  <th>Product Name</th>  <td>@ViewBag.matchingProduct.ProductName</td>  </tr>  <tr>  <th>Rate</th>  <td>@ViewBag.matchingProduct.Rate</td>  </tr>  </table> |

## Strongly Typed View

In the previous example in Details view Property name was not showing in intelligent, so manually we have to type the property name. **To overcome this problem strongly typed view has been introduced.**

**Strongly Typed View**

* View that is associated to a specific model class is called as “Strongly Typed View”.
* Strongly typed views have to specify the model class name with @model directive at the top of the view.
* Strongly Typed Views can receive model objects from the controller.

|  |
| --- |
|  |

**In Controller instead of passing data in ViewBag, we can pass the model object directly as highlighted View name and model object**

|  |
| --- |
| public ActionResult Details( int id)  {  List<Product> products = new List<Product>()  {  new Product() {ProductId = 101, ProductName= "Nokia", Rate = 2938},  new Product() {ProductId = 102, ProductName = "MI", Rate= 8746},  new Product() {ProductId = 103, ProductName = "OnePlus", Rate = 9876}  };  Product matchingProduct = null;  foreach(var item in products)  {  if(item.ProductId == id)  {  matchingProduct = item;  }  }  return View("Details",matchingProduct);  } |

**In Details View make the view as strongly type by @model.namespace**

**(**@model ModelExample.Models.Product**) and will retrieving the value use** Model.Propertyname **as highlighted.**

**Also if we pass as collection we need to mention as**

@model List<ModelExample.Models.Product>

|  |
| --- |
| @model ModelExample.Models.Product  @{  ViewBag.Title = "Details";  }  <h2>Product Details</h2>  <table class="table">  <tr>  <th>Product Id</th>  <td>@Model.ProductId</td>  </tr>  <tr>  <th>Product Name</th>  <td>@Model.ProductName</td>  </tr>  <tr>  <th>Rate</th>  <td>@Model.Rate</td>  </tr>  </table> |

**And also if we pass as collection, we need declare model as collection and loop we need to Model only as highlighted below:**

|  |
| --- |
| @model List<ModelExample.Models.Product>  @{  ViewBag.Title = "Index";  }  <h2>Products</h2>  <table class="table">  <thead>  <tr>  <th>ProductId</th>  <th>ProductName</th>  <th>Rate</th>  </tr>  </thead>  <tbody>  @foreach(var item in Model)  {  <tr>  <td>@item.ProductId</td>  <td><a href="/Products/details/@item.ProductId">@item.ProductName</a></td>  <td>@item.Rate</td>  </tr>  }  </tbody>  </table> |

# Model Binding

## Model Binding

* Process of “receiving values from different sources of the request and passing them as arguments to action method”.
* Assign values of different parameters of the action method automatically.

|  |
| --- |
|  |

**Model Binding with Complex types**

* Model Binding can work with complex types.
* Model Binding can automatically convert form field data or query string values to the properties of a complex type parameter of an action method.
* Default values are null or zero or false.

|  |
| --- |
|  |

**Sources of Model Binding**

* Query String
* Form Data
* Route Data (the data that is passed from other action method while redirecting)
* JSON request body (in case of AJAX)

## Working with Forms

* In View of Create Product with Forms

|  |
| --- |
| @{  ViewBag.Title = "Create";  }  <h2>Create Products</h2>  <form action="/Products/Create" method="post">  <div class="form-group">  <label>Product Id</label>  <input type="text" class="form-control" id="txtProductId" placeholder="Product Id" name="ProductId"/>  </div>  <div class="form-group">  <label>Product Name</label>  <input type="text" class="form-control" id="txtProductName" placeholder="Product Name" name="ProductName"/>  </div>  <div class="form-group">  <label>Rate</label>  <input type="text" class="form-control" id="txtRate" placeholder="Rate" name="Rate"/>  </div>  <button>Submit</button>  </form> |

* In Controller we need to action one for GET Request and other for POST Request
* **And in POST action method we can get the form data value with two ways** :-
  + Pass all the properties one by one, **but will leads problem for large number of properties.**

|  |
| --- |
| [HttpPost]  public ActionResult Create(int ProductId, string ProductName, decimal Rate)  {  return View();  } |

* + **Pass the argument of type class (Product), but here the rule is model class property name must be matching with the name of each textbox in Form. And in future we can add code to insert data in database, after clicking on submit button.**

|  |
| --- |
| [HttpPost]  public ActionResult Create(Product p)  {  return View();  } |

## Bind Attribute

* The [Bind] Attribute allows you to specify the list of properties that you want to bind into the model object that is receiving using “Model Binding”.
* It allows you to specify “include” and “exclude” comma separate list of properties.
* **Include**

|  |
| --- |
|  |

* **Exclude**

|  |
| --- |
|  |

## Custom Model Binders

* Whenever the list of fields of the view and list of fields of model are different, you have to use “Custom Model Binders”, to map exactly, which value of view should be stored in which property of model.

|  |
| --- |
|  |

* Add Custom Model Binders to “Binders” collection in Application\_Start() method.

|  |
| --- |
|  |

**In Model: - Student.cs**

|  |
| --- |
| public class Student  {  public int StudentId { get; set; }  public string StudentName { get; set; }  public string Address { get; set; }  } |

**In View: - Last four parameters will be merging to one as address**

|  |
| --- |
| @{  ViewBag.Title = "Create";  }  <h2>Create Student</h2>  <form action="/Student/Create" method="post">  <div class="form-group">  <label>Student Id</label>  <input type="text" class="form-control" id="txtStudentId" placeholder="Student Id" name="StudentId" />  </div>  <div class="form-control">  <label>Student Name</label>  <input type="text" class="form-control" id="txtStudentName" placeholder="Student Name" name="StudentName" />  </div>  <div class="form-control">  <label>D.No</label>  <input type="text" class="form-control" id="txtDNo" placeholder="D.No" name="DNo" />  </div>  <div class="form-control">  <label>Street</label>  <input type="text" class="form-control" id="txtStreet" placeholder="Street" name="Street" />  </div>  <div class="form-control">  <label>LandMark</label>  <input type="text" class="form-control" id="txtLandMark" placeholder="Land Mark" name="LandMark" />  </div>  <div class="form-control">  <label>City</label>  <input type="text" class="form-control" id="txtCity" placeholder="City" name="City" />  </div>  <button class="btn btn-success">Submit</button>  </form> |

**Create a model binder class to bind the (D.No , Street, LandMark, City) to address.**

|  |
| --- |
| public class CustomBinder: IModelBinder  {  public object BindModel(ControllerContext controllerContext, ModelBindingContext bindingContext)  {  int studentId = Convert.ToInt32(controllerContext.HttpContext.Request.Form["StudentId"]);  string studentName = controllerContext.HttpContext.Request.Form["StudentName"];  string DNo = controllerContext.HttpContext.Request.Form["DNo"];  string Street = controllerContext.HttpContext.Request.Form["Street"];  string LandMark = controllerContext.HttpContext.Request.Form["LandMark"];  string City = controllerContext.HttpContext.Request.Form["City"];  return new Student() { StudentId =studentId, StudentName = studentName, Address = DNo + "," + Street + "," + LandMark + "," + City}  }  } |

**In Controller POST action method pass the argument CustomBinder**

|  |
| --- |
| [HttpPost]  public ActionResult Create([ModelBinder(typeof (CustomBinder))] Student stu)  {  return View();  } |

**At last in Application start add the code , so the ASP.NET MVC framework can know about the custom Binder**

|  |
| --- |
| protected void Application\_Start()  {  AreaRegistration.RegisterAllAreas();  RouteConfig.RegisterRoutes(RouteTable.Routes);  ModelBinders.Binders.Add(typeof(Student), new CustomBinder());  } |

# Entity Framework

## Introduction

* “Entity Framework” (EF) is a database technology, which is built based on ADO.NET, that supports ORM (object Relational Mapping) pattern.
* It bridges between “Object Oriented Programming” and “Relational Databases”, using Model classes.

|  |
| --- |
|  |

|  |
| --- |
|  |

## Features of EF:-

* Modeling
* Querying: - EF allows us to use LINQ query to retrieve data from underline database.
* Change Tracking
* Saving
* Concurrency
* Transactions
* Caching

## Entity Framework – Work Flow

|  |
| --- |
|  |

## Entity Framework – Architecture

|  |
| --- |
|  |

|  |
| --- |
|  |

## DbContext and DbSet

* DbContext is a class, based on which you can write LINQ queries to perform CRUD operations on table.
* DbContext is the collection of DbSets. ( For one database needs to create one DbContext)
* DbSet object that represents a table. ( For one table one DbSet needs to be created)

|  |
| --- |
|  |

## Retrieving All Rows from Database

|  |
| --- |
|  |

* Here db is reference variable of DbContext type ( we can give any name for it)
* ToList() – invoke the EF to start the query process converting the LINQ statement to equivalent SQL statement, execute the query against the database and also converts the retrieve rows as object of model class and return them as collection finally.

As we Database First approach, so first let us create necessary database

**Step 1:- (**Create a Model class)

* As here are using Database First approach Model class will automatically create

**Models -> add new item -> Data -> ADO.NET Entity Data Model -> EF Designer From database -> Choose Data Source (Mircrosoft SQL Server) -> Select Server Name and database name -> Entity Framework 6.x -> Select the tables -> Finish**

**After that all the model class, context class will create automatically.**

|  |
| --- |
|  |

And here in

* Model1.csdl represents the conceptual Model. (Model Class).
* Model1.ssdl represents the Storage Model (store list of tables/columns).
* Model1.msl represents the Mapping (Mapping from conceptual model to storage model).

**Step 2:- (Create Categories controller and in the view we will pull data from Category using LINQ)**

* In Categories import the name of model - using EFDbFirstApproachExample.Models;
* Create the object for context class –

EFDBFirstDatabaseEntities db = new EFDBFirstDatabaseEntities();

* Now retrieve the value using toList() which convert the Linq query to SQL statement (select \* from table)

db.Categories.ToList(); *here categories is the Dbset*

* Assign it to variable of List<Category> type – here Category is model class

List<Category> categories = db.Categories.ToList();

* Now pass categories data to the view as model data and the particular view must be strongly typed view.

return View(categories);

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.Web.Mvc;  using EFDbFirstApproachExample.Models;  namespace EFDbFirstApproachExample.Controllers  {  public class CategoriesController : Controller  {  // GET: Categories  public ActionResult Index()  {  EFDBFirstDatabaseEntities db = new EFDBFirstDatabaseEntities();  List<Category> categories = db.Categories.ToList();  return View(categories);  }  }  } |

**Step 3:- (Create the view and retrieving the value in a table)**

* Make the view as strongly typed view

@model List<EFDbFirstApproachExample.Models.Category>

* Complete collection data that we passed from controller, will be retrieve as property called Model in the view

In View

|  |
| --- |
| @model List<EFDbFirstApproachExample.Models.Category>  @{  ViewBag.Title = "Index";  Layout = "~/Views/Shared/\_LayoutPage1.cshtml";  }  <h2>Categories</h2>  <table class="table">  <tr>  <th>Category Id</th>  <th>Category Name</th>  </tr>  @foreach (var item in Model)  {  <tr>  <td>@item.CategoryID</td>  <td>@item.CategoryName</td>  </tr>  }  </table> |

## Retrieving Multiple Rows Based on Condition

|  |
| --- |
|  |

* Suppose from the Product table, we want to retrieve for categoryId 1

List<Product> products = db.Products.Where(temp => temp.CategoryID ==1).ToList(); - Here temp is the temporary variable.

* Add one more condition based on Price of Product

List<Product> products = db.Products.Where(temp => temp.CategoryID ==1

&& temp.Price >= 50000).ToList();

|  |
| --- |
| public class ProductsController : Controller  {  // GET: Products  public ActionResult Index()  {  EFDBFirstDatabaseEntities db = new EFDBFirstDatabaseEntities();  List<Product> products = db.Products.Where(temp => temp.CategoryID ==1 && temp.Price >= 50000).ToList();  return View(products);  }  } |

## Searching Data

How to create search box in entity framework

|  |
| --- |
|  |

In Product View -**add the below inline form with input type as search**

|  |
| --- |
| <form class="form-inline" action="/products/index">  <label for="search">Search</label>  <input type="search" class="form-control mr-1" id="search" name="search" placeholder="Search"/>  <button class="btn btn-primary">Go</button>  </form> |

Whenever the user clicks on Go button, it sends request to **/Products/index action method**. So value of search box will automatically submit to the action method and inside the action method we can receive it.

**Now in action method pass the parameter as search and the parameter name should be same as name of the textbox must be same**

|  |
| --- |
| public ActionResult Index(string search = "")  {  EFDBFirstDatabaseEntities db = new EFDBFirstDatabaseEntities();  List<Product> products = db.Products.Where(temp => temp.ProductName.Contains(search)).ToList();  return View(products);  } |

**In the parameter we assign as “”, so that on page load all the products should be visible.**

**And Here** temp => temp.ProductName.Contains(search) contains is used to do the like functionality.

And one problem is the search texbox is getting empty after retrieving the searched value. So searched value need to pass back to view using viewbag.

|  |
| --- |
| public ActionResult Index(string search = "")  {  EFDBFirstDatabaseEntities db = new EFDBFirstDatabaseEntities();  ViewBag.Search = search;  List<Product> products = db.Products.Where(temp => temp.ProductName.Contains(search)).ToList();  return View(products);  } |

**In view of search textbox assign the value attribute to @ViewBag.Search.**

|  |
| --- |
| <form class="form-inline" action="/products/index">  <label for="search">Search</label>  <input type="search" class="form-control mr-1" id="search" name="search" placeholder="Search" value="@ViewBag.Search"/>  <button class="btn btn-primary">Go</button>  </form> |

## Retrieving Single Row Based on Condition

|  |
| --- |
|  |

Here in the Products/Index we will create hyperlink with the particular row to select the product.

**In Products Controller**

Create a new details action method with productID as argument.

|  |
| --- |
| public ActionResult Details(long id)  {  EFDBFirstDatabaseEntities db = new EFDBFirstDatabaseEntities();  Product p = db.Products.Where(temp => temp.ProductID == id).FirstOrDefault();  return View(p);  } |

Create a details View

|  |
| --- |
| model EFDbFirstApproachExample.Models.Product;  @{  ViewBag.Title = "Details";  Layout = "~/Views/Shared/\_LayoutPage1.cshtml";  }  <h2>Details</h2>  <table class="table">  <tr>  <th>Product ID</th>  <th>@Model.ProductID</th>  </tr>  <tr>  <th>Product Name</th>  <th>@Model.ProductName</th>  </tr>  </table>  <a href="/Products/Index">Back to Products</a> |

In Products add the below hyperlink to redirect to details view with the particular productId

|  |
| --- |
| <td>  <a href="/Products/Details/@item.ProductID">Details</a>  </td> |

Product View

|  |
| --- |
|  |

Details View

|  |
| --- |
|  |

## Inserting Row

|  |
| --- |
|  |

* Create Action method called create with get request ( if you not mentioned anything bydefault it is taking as httpGet request.

|  |
| --- |
| public ActionResult Create()  {  return View();  } |

* Create the view where it will enter all the details of products to insert in database.

|  |
| --- |
| @{  ViewBag.Title = "Create";  Layout = "~/Views/Shared/\_LayoutPage1.cshtml";  }  <h2>Create</h2>  <form action="/Products/Create" method="post">  <div class="form-group col-md-6">  <label for="productName">Product Name</label>  <input type="text" class="form-control" id="productName" name="productName" placeholder="product Name" />  </div>  <div class="form-group col-md-6">  <label for="price">Price</label>  <input type="number" class="form-control" id="price" name="price" placeholder="Price" />  </div>  <div class="form-group col-md-6">  <label for="DateOfPurchase">Date Of Purchase</label>  <input type="date" class="form-control" id="DateOfPurchase" name="DateOfPurchase" placeholder="Date Of Purchase" />  </div>  <div class="form-group col-md-6">  <label for="AvailabilityStatus">Availability Status</label>  <input type="text" class="form-control" id="AvailabilityStatus" name="AvailabilityStatus" placeholder="Availability Status" />  </div>  <div class="form-group col-md-6">  <label for="CategoryID">Category ID</label>  <input type="date" class="form-control" id="CategoryID" name="CategoryID" placeholder="Category ID" />  </div>  <div class="form-group col-md-6">  <label for="BrandID">Brand ID</label>  <input type="date" class="form-control" id="BrandID" name="BrandID" placeholder="Brand ID" />  </div>  <div class="form-group">  <input class="form-check-input" type="checkbox" id="Active" />  <label class="form-control-label" for="Active">  Active  </label>  </div>  <button type="submit" class="btn btn-success">Submit</button>  <a class="btn btn-danger" href="/Products/index">Cancel</a>  </form> |

* In Product view create a hyperlink for create action method

|  |
| --- |
| <a href="/Products/Create">Create</a> |

* Now after click on submit button form request post request to Create post method. And called the Add() and SaveChanges() method . Finally after inserting redirect to index action method

|  |
| --- |
| [HttpPost]  public ActionResult Create(Product p)  {  EFDBFirstDatabaseEntities db = new EFDBFirstDatabaseEntities();  db.Products.Add(p);  db.SaveChanges();  return RedirectToAction("Index");  } |

Create View

|  |
| --- |
|  |

## Updating Row

|  |
| --- |
|  |

* Create a hyperlink of Edit in Product page

|  |
| --- |
| <a href="/Products/Edit/@item.ProductID">Edit</a> |

* Create Edit Action method in Products Controller where we will receive productId as Id in parameter from Product Page.

|  |
| --- |
| public ActionResult Edit(long id)  {  EFDBFirstDatabaseEntities db = new EFDBFirstDatabaseEntities();  Product product = db.Products.Where(temp => temp.ProductID == id).FirstOrDefault();  return View(product);  } |

* Create the Edit View – Here we will form for edit and

A hidden field is declared for ProductID , this input type of hidden will automatically submitted along with the other textbox values, this hidden field will not be visible on the browser.

**And Datatime format should be (“yyyy-MM-dd”) only**

**And for checkbox value should assign to checked properties and value = true**

|  |
| --- |
| @model EFDbFirstApproachExample.Models.Product  @{  ViewBag.Title = "Edit";  Layout = "~/Views/Shared/\_LayoutPage1.cshtml";  }  <h2>Edit</h2>  <form action="/Products/Edit/@Model.ProductID" method="post">  <div class="form-row">  <div class="form-group col-6">  <label for="productName">Product Name</label>  <input type="hidden" name="ProductId" value="@Model.ProductID"/>  <input type="text" class="form-control" id="productName" name="productName" placeholder="product Name" value="@Model.ProductName"/>  </div>  </div>  <div class="form-row">  <div class="form-group col-6">  <label for="price">Price</label>  <input type="number" class="form-control" id="price" name="price" placeholder="Price" value="@Model.Price"/>  </div>  </div>  <div class="form-row">  <div class="form-group col-6">  <label for="DateOfPurchase">Date Of Purchase</label>  <input type="date" class="form-control" id="DateOfPurchase" name="DateOfPurchase" placeholder="Date Of Purchase" value="@Model.DateOfPurchase.Value.ToString("yyyy-MM-dd")"/>  </div>  </div>  <div class="form-row">  <div class="form-group col-6">  <label for="AvailabilityStatus">Availability Status</label>  <input type="text" class="form-control" id="AvailabilityStatus" name="AvailabilityStatus" placeholder="Availability Status" value="@Model.Price"/>  </div>  </div>  <div class="form-row">  <div class="form-group col-md-6">  <label for="CategoryID">Category ID</label>  <input type="text" class="form-control" id="CategoryID" name="CategoryID" placeholder="Category ID" value="@Model.CategoryID"/>  </div>  </div>  <div class="form-row">  <div class="form-group col-md-6">  <label for="BrandID">Brand ID</label>  <input type="text" class="form-control" id="BrandID" name="BrandID" placeholder="Brand ID" value="@Model.BrandID"/>  </div>  </div>  <div class="form-group">  <input class="form-check-input" type="checkbox" id="Active" name="Active" checked="@Model.Active" value “true”/>  <label class="form-control-label" for="Active">  Active  </label>  </div>  <button type="submit" class="btn btn-success">Save</button>  <a class="btn btn-danger" href="/Products/index">Cancel</a>  </form> |

* For Post request create another action method in Products controller and called the database to update the fields.

|  |
| --- |
| [HttpPost]  public ActionResult Edit(Product p)  {  EFDBFirstDatabaseEntities db = new EFDBFirstDatabaseEntities();  Product product = db.Products.Where(temp => temp.ProductID == p.ProductID).FirstOrDefault();  product.ProductName = p.ProductName;  product.Price = p.Price;  product.DateOfPurchase = p.DateOfPurchase;  product.BrandID = p.BrandID;  product.CategoryID = p.CategoryID;  product.AvailabilityStatus = p.AvailabilityStatus;  product.Active = p.Active;  db.SaveChanges();  return RedirectToAction("Index", "Products");  } |

In Product View

|  |
| --- |
|  |

In Edit Page

|  |
| --- |
|  |

## Delete Row

|  |
| --- |
|  |

* Create delete hyperlink in Product (Index page)

|  |
| --- |
| <a href="/Products/Delete/@item.ProductID">Delete</a> |

* Create Delete Action method in Products Controller where we will receive productId as Id in parameter from Product Page.

|  |
| --- |
| public ActionResult Delete(long id)  {  EFDBFirstDatabaseEntities db = new EFDBFirstDatabaseEntities();  Product existingProduct = db.Products.Where(temp => temp.ProductID == id).FirstOrDefault();  return View(existingProduct);  } |

* Create the Delete View – Here we will create a form to submit a post request to delete action method

|  |
| --- |
| <form action="/Products/Delete/@Model.ProductID" method="post">  <div class="form-row">  <div class="form-group col-6">  <label for="productName">Product Name</label>  <input type="hidden" name="ProductId" value="@Model.ProductID" />  @Model.ProductName  </div>  </div>  <div class="form-row">  <div class="form-group col-6">  <label for="price">Price</label>  @Model.Price  </div>  </div>  <button type="submit" class="btn btn-success">Delete</button>  <a class="btn btn-danger" href="/Products/index">Cancel</a>  </form> |

* Create the action method of post request for delete

Here p parameter is not use, just to differentiate with Get Delete action method.

|  |
| --- |
| [HttpPost]  public ActionResult Delete(long id, Product p)  {  EFDBFirstDatabaseEntities db = new EFDBFirstDatabaseEntities();  Product existingProduct = db.Products.Where(temp => temp.ProductID == id).FirstOrDefault();  db.Products.Remove(existingProduct);  db.SaveChanges();  return RedirectToAction("Index", "Products");  } |

## Navigation Properties

* Navigation Property is a complex-type property, with “**Virtual**” Keyword.
* Navigation Property stores reference of parent model class’s object, or vice versa.
* Used to access parent tables data from child table’s data, or vice versa.

|  |
| --- |
|  |

In Product page we shown categoryID as 1 or 2 instead of that we want to display as category name from Category table.

**Fortunately Visual Studio automatically creates the navigation when we create edmx file in EF (DB First approach). But in code first approach we have to create these properties manually.**

|  |
| --- |
|  |

Now in index instead of retrieving categoryID retrieve category using the object Category

|  |
| --- |
| <td>@item.Category.CategoryName</td>  <td>@item.Brand.BrandName</td> |

## Creating Static Dropdown List

Create a static downdownList for the availability status field, which has two properties “InStock” and “OutOfStock” in create product page.

|  |
| --- |
| <div class="form-row">  <div class="form-group col-6">  <label for="AvailabilityStatus">Availability Status</label>  <select class="form-control" id="AvailabilityStatus" name="AvailabilityStatus" >  <option value="">Please Select</option>  <option value="InStock">In Stock</option>  <option value="OutOfStock">Out Of Stock</option>  </select>  </div> |

## Creating Dynamic Dropdownlist

In order to create dynamic Dropdownlist we have to pass data from controller to View using ViewBag.

* Before executing the create view fetch the Dropdownlist and assign to ViewBag

|  |
| --- |
| public ActionResult Create()  {  EFDBFirstDatabaseEntities db = new EFDBFirstDatabaseEntities();  ViewBag.Categories = db.Categories.ToList();  ViewBag.Brands = db.Brands.ToList();  return View();  } |

* In View using foreach loop retrieve the values from ViewBag and display in screen.

|  |
| --- |
| <div class="form-row">  <div class="form-group col-md-6">  <label for="CategoryID">Category ID</label>  <select class="form-control" id="CategoryID" name="CategoryID">  <option value="">Please Select</option>  @foreach(var cat in ViewBag.Categories)  {  <option value="cat.categoryID">@cat.CategoryName</option>  }  </select>  </div>  </div>  <div class="form-row">  <div class="form-group col-md-6">  <label for="BrandID">Brand ID</label>  <select class="form-control" id="BrandID" name="BrandID" >  <option value="">Please Select</option>  @foreach(var brand in ViewBag.Brands)  {  <option value="brand.BrandID">@brand.BrandName</option>  }  </select>  </div>  </div> |

In Edit page we want to do everything will same as create view, only for the selected value we need to put a condition as highlighted below:

|  |
| --- |
| <div class="form-row">  <div class="form-group col-md-6">  <label for="CategoryID">Category ID</label>  <select class="form-control" id="CategoryID" name="CategoryID" value="@Model.CategoryID">  <option value="">Please select</option>  @foreach (var cat in ViewBag.Categories)  {  if (cat.CategoryID == Model.CategoryID)  {  <option value="cat.CategoryID" selected="selected">@cat.CategoryName</option>  }  else  {  <option value="cat.CategoryID">@cat.CategoryName</option>  }  }  </select>  </div>  </div>  <div class="form-row">  <div class="form-group col-md-6">  <label for="BrandID">Brand</label>  <select class="form-control" id="BrandID" name="BrandID" value="@Model.BrandID">  <option value="">Please select</option>  @foreach (var brand in ViewBag.Brands)  {  if (brand.BrandID == Model.BrandID)  {  <option value="brand.BrandID" selected="selected">@brand.BrandName</option>  }  else  {  <option value="brand.BrandID">@brand.BrandName</option>  }    }  </select>  </div>  </div> |

## Sorting

|  |
| --- |
|  |

## Paging

|  |
| --- |
|  |

## Storing Images in Databases

There are several ways to perform these activities:-

1. Copy the image files into the server into a specific folder and store the image file path in the database. (in this case the size of the database will be less).
2. Other way is to convert the image file into base64 string and we can store that information in the database. (In this case the size of the database will be more but it will be more secure).

**Let see the second approach how to do.**

**Step 1:- Add a column for Photo to the particular table**

|  |
| --- |
| ALTER TABLE Products ADD Photo VARCHAR(max) |

**Step 2:-** **Add an attribute** enctype **in <from></form> and assign to** multipart/form-data. This encryption type is required in order to submit the files to the server.

|  |
| --- |
| <form action="/Products/Create" method="post" enctype="multipart/form-data">  </form> |

**Step 3:- Add the Input type for file inside the form**

|  |
| --- |
| <div class="form-group">  <label for="Image">Image</label>  <input type="file" class="form-control-file" id="Image" name="photo"/>  </div> |

**Step 4:- Now in Post controller call the request.files** is a collection type which contains all the files that are submitted from the browser through input type = file. But this will work when encryption type in form is enctype="multipart/form-data" which is mentioned in step 2.

|  |
| --- |
| [HttpPost]  public ActionResult Create(Product p)  {  EFDBFirstDatabaseEntities db = new EFDBFirstDatabaseEntities();  if (Request.Files.Count >= 1)  {  var file = Request.Files[0];  //Reading all the bytes from the file into the imgBytes  //storing base64String in photo property of Model class  var imgBytes = new Byte[file.ContentLength + 1];  file.InputStream.Read(imgBytes, 0, file.ContentLength);  var base64String = Convert.ToBase64String(imgBytes, 0, imgBytes.Length);  p.Photo = base64String;  }  db.Products.Add(p);  db.SaveChanges();  return RedirectToAction("Index");  } |

**Step 5: While retrieving the value from database and display in screen we have to mentioned image scr as** “data;image;base64”

|  |
| --- |
| <td><img scr="data:image;base64,@item.Photo" width="100px"/></td> |

# EF Code First Approach

## Database-First (vs) Code-First (vs) Model-First Approach

* **Database-First Approach**

Developer has to create database first; model class will be auto-generated next.

|  |
| --- |
|  |

* **Code-First Approach**

Developer has to create model class first; database will be auto-generated next.

|  |
| --- |
|  |

Not good if you are using stored procedures.

* **Model-First Approach**

Developer has to design the model class first by using ORM designer in VS, database will be auto-generated next.

|  |
| --- |
|  |

## Code-First Approach

Create the Model Class one by one Brand.cs etc.

**Step 1:- Create the Model Classes**

* Import the namespace **System.ComponentModel.DataAnnotations** and write the property name.
* As BrandID is a primary we need to specific the attribute **[key]** at top.

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.ComponentModel.DataAnnotations;  namespace EFDbFirstApproachExample.Models  {  public class Brand  { [Key]  public long BrandID { get; set; }  public string BrandName { get; set; }  }  } |

* For Foreign key we need to use **Virtual** keyword

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.ComponentModel.DataAnnotations;  namespace EFDbFirstApproachExample.Models  {  public class Product  {  [Key]  public long ProductID { get; set; }  public string ProductName { get; set; }  public Nullable<decimal> Price { get; set; }  public Nullable<System.DateTime> DateOfPurchase { get; set; }  public string AvailabilityStatus { get; set; }  public Nullable<long> CategoryID { get; set; }  public Nullable<long> BrandID { get; set; }  public Nullable<bool> Active { get; set; }  public string Photo { get; set; }  public virtual Brand Brand { get; set; }  public virtual Category Category { get; set; }  }  } |

**Step 2:- Create the DBContext class**

* Import the namespace **System.Data.Entity**
* **Inherit** the class from **DbContext** parent class.
* **Create the DbSet one by one.**

|  |
| --- |
| using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.Data.Entity;  namespace EFDbFirstApproachExample.Models  {  public class CompanyDbContext:DbContext  {  public DbSet<Brand> Brands { get; set; }  public DbSet<Category> Categories { get; set; }  public DbSet<Product> Products { get; set; }  }  } |

**Step 3:- Create the connection string in Web.config file.**

* Mark the intialCatelog as Database name, you want to create

|  |
| --- |
| <connectionStrings>  <add  name="MyConnectionString"  connectionString="Data Source=LH7U05CG5152DH4\SQLEXPRESS2012;InitialCatalog=Company;Integrated Security=True" providerName="System.Data.SqlClient"  />  </connectionStrings> |

**Step 4:- In DbContext class of step 3, create a constructor and call the parent class constructor by using the keyword base.**

* Specify the name of base() as connectionstring name from step 3 **MyConnectionString**

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.Data.Entity;  namespace EFDbFirstApproachExample.Models  {  public class CompanyDbContext : DbContext  {  public CompanyDbContext() : base("MyConnectionString")  {  }  public DbSet<Brand> Brands { get; set; }  public DbSet<Category> Categories { get; set; }  public DbSet<Product> Products { get; set; }  }  } |

**Step 5: Now if we run the application database we will create automatically**

## Re-Creating the Database

Let us see how to modify the columns in code-first approach. So for that we will add a property to the particular model class. But while running the application it will give error as the database table is not matching with the model class.

Here Nullable is used so the table column can accept null values.

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.ComponentModel.DataAnnotations;  namespace EFDbFirstApproachExample.Models  {  public class Product  {  [Key]  public long ProductID { get; set; }  public string ProductName { get; set; }  public Nullable<decimal> Price { get; set; }  public Nullable<System.DateTime> DateOfPurchase { get; set; }  public string AvailabilityStatus { get; set; }  public Nullable<long> CategoryID { get; set; }  public Nullable<long> BrandID { get; set; }  public Nullable<bool> Active { get; set; }  public string Photo { get; set; }  public Nullable<decimal> Quantity { get; set; }  public virtual Brand Brand { get; set; }  public virtual Category Category { get; set; }  }  } |

**So for that one option is to delete the database and run the application once again, so that it will create new. But problem is all the rows will be delete , in early stage is ok for deleting database but in middle of development it is not ok if the data is lost for that we need perform Code-First migration.**

## Automatic Code-First Migrations

|  |
| --- |
|  |

**Automated Migration in Entity Framework**

* **Updates the model changes to database automatically.**
* **Process Flow:-**
  + **Enable Migrations in Package Manager Console.**
  + **Set Database Initializer in DbContext.**
* **Step1:- Enable Migrations in Package Manager Console.**

|  |
| --- |
|  |

|  |
| --- |
| Enable-Migrations -EnableAutomaticMigrations:$true |

**This will create a folder called migration with a default file called configuration.cs as shown below**

**Configuration.cs**

|  |
| --- |
| namespace EFDbFirstApproachExample.Migrations  {  using System;  using System.Data.Entity;  using System.Data.Entity.Migrations;  using System.Linq;  internal sealed class Configuration : DbMigrationsConfiguration<EFDbFirstApproachExample.Models.CompanyDbContext>  {  public Configuration()  {  AutomaticMigrationsEnabled = true;  ContextKey = "EFDbFirstApproachExample.Models.CompanyDbContext";  }  protected override void Seed(EFDbFirstApproachExample.Models.CompanyDbContext context)  {  // This method will be called after migrating to the latest version.  // You can use the DbSet<T>.AddOrUpdate() helper extension method  // to avoid creating duplicate seed data.  }  }  } |

**Now in Seed method we can add the records we want to insert while creating the database.**

|  |
| --- |
| namespace EFDbFirstApproachExample.Migrations  {  using System;  using System.Data.Entity;  using System.Data.Entity.Migrations;  using System.Linq;  internal sealed class Configuration : DbMigrationsConfiguration<EFDbFirstApproachExample.Models.CompanyDbContext>  {  public Configuration()  {  AutomaticMigrationsEnabled = true;  ContextKey = "EFDbFirstApproachExample.Models.CompanyDbContext";  }  protected override void Seed(EFDbFirstApproachExample.Models.CompanyDbContext context)  {  // This method will be called after migrating to the latest version.  // You can use the DbSet<T>.AddOrUpdate() helper extension method  // to avoid creating duplicate seed data.  context.Brands.AddOrUpdate(new Models.Brand() { BrandID = 1, BrandName = "HP" }, new Models.Brand() { BrandID = 2, BrandName = "Lenovo" });  }  }  } |

* **Step 2: Set Database Initializer in DbContext.**

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.Data.Entity;  using EFDbFirstApproachExample.Migrations;  namespace EFDbFirstApproachExample.Models  {  public class CompanyDbContext : DbContext  {  public CompanyDbContext() : base("MyConnectionString")  {  Database.SetInitializer(new MigrateDatabaseToLatestVersion<CompanyDbContext, Configuration>());  }  public DbSet<Brand> Brands { get; set; }  public DbSet<Category> Categories { get; set; }  public DbSet<Product> Products { get; set; }  }  } |

## Performing Code-First Migrations Manually

|  |
| --- |
|  |

## Default Conventions in Code-First Approach

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.ComponentModel.DataAnnotations;  using System.ComponentModel.DataAnnotations.Schema;  namespace EFDbFirstApproachExample.Models  {  [Table("Products",Schema = "dbo")]  public class Product  {  [Key]  public long ProductID { get; set; }  public string ProductName { get; set; }  public Nullable<decimal> Price { get; set; }  [Column("DateOfPurchase",TypeName ="datetime")]  public Nullable<System.DateTime> DOP { get; set; }  public string AvailabilityStatus { get; set; }  public Nullable<long> CategoryID { get; set; }  public Nullable<long> BrandID { get; set; }  public Nullable<bool> Active { get; set; }  public string Photo { get; set; }  public Nullable<decimal> Quantity { get; set; }  public virtual Brand Brand { get; set; }  public virtual Category Category { get; set; }  }  } |

Benefits of Data Annotation are that we can any name for table instead of default name, column name also.

# HTML Helpers

## HTML Helpers

HTML Helpers generates HTML elements using model class objects in razor view, it binds the model object to HTML elements to display the value of model properties into HTML elements and also assigns the value HTML elements into Model properties while submitting the form.

It always recommended using HTML Helpers in razor view instead of HTML tags manually.

|  |
| --- |
|  |

|  |
| --- |
|  |

**Html.TextBoxFor – whenever we want to create a property without model properties then don’t use For.**

**Html.TextFor- Whenever we want to create a property with model properties then use For.**

* **Label**

|  |
| --- |
| @Html.Label("Search", new { @class = "mr-1" })  <label for="search">Search</label> |

* **TextBox**

|  |
| --- |
| @Html.TextBox("Search",ViewBag.search as string, new { @class="form-control mr-1",placeholder="Search"})  <input type="search" class="form-control mr-1" id="search" name="search" placeholder="Search" value="@ViewBag.Search"/> |

* **DisplayFor**

|  |
| --- |
| <td>@Html.DisplayFor(temp => item.ProductID)</td>  <td>@item.ProductID</td> |

Here temp is temporary variable for model properties.

* **ActionLink**

|  |
| --- |
| @Html.ActionLink("Details","Details",new { id = item.ProductID,controller="Products"})  <!--Here First arguments is link text , second is action name, third is route variable  (additional parameter that are passing as part of hyperlink-->  <a href="/Products/Details/@item.ProductID">Details</a> |

* **Form**

|  |
| --- |
| @using (Html.BeginForm("Create", "Products", FormMethod.Post, new { enctype = "multipart/form-data" }))  {  }  <form action="/Products/Create" method="post" enctype="multipart/form-data">  </form> |
|  |

* **LabelFor and TextBoxFor**

|  |
| --- |
| @Html.LabelFor(temp => temp.Price)  @Html.TextBoxFor(temp => temp.Price, new { @class= "form-control", placeholder= "Price" })  <label for="price">Price</label>  <input type="number" class="form-control" id="price" name="price" placeholder="Price" /> |

* **DropDownListFor – we use SelectListItem**

|  |
| --- |
| @Html.DropDownListFor(temp => temp.AvailabilityStatus,new List<**SelectListItem**>() {  new SelectListItem() { Text= "In Stock", Value="InStock"},  new SelectListItem() { Text= "Out of Stock", Value="OutOfStock"}  },"Please Select", new { @class= "form-control" })  <select class="form-control" id="AvailabilityStatus" name="AvailabilityStatus">  <option value="">Please Select</option>  <option value="InStock">In Stock</option>  <option value="OutOfStock">Out Of Stock</option>  </select> |

* **Dynamic DropDownList – We use SelectList**

|  |
| --- |
| @Html.DropDownListFor(temp => temp.CategoryID,  new **SelectList**(ViewBag.Categories, "CategoryID", "CategoryName"),"Please Select",new { class="form-control"})  <select class="form-control" id="CategoryID" name="CategoryID">  <option value="">Please Select</option>  @foreach (var cat in ViewBag.Categories)  {  <option value="@cat.CategoryID">@cat.CategoryName</option>  } |

* **CheckBoxFor**

|  |
| --- |
| @Html.CheckBoxFor(temp => temp.Active.**Value**,new { @class= "form-check-input" })  <input class="form-check-input" type="checkbox" id="Active" name="Active" checked="checked" /> |

## Customizing Label in HTML Helpers

For customizing name of Label, we need to use data annotation called **[display(Name=”Product Name”]** in model class.

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.ComponentModel.DataAnnotations;  using System.ComponentModel.DataAnnotations.Schema;  namespace EFDbFirstApproachExample.Models  {  [Table("Products",Schema = "dbo")]  public class Product  {  [Key]  [Display(Name ="Product ID")]  public long ProductID { get; set; }  [Display(Name = "Product Name")]  public string ProductName { get; set; }  public Nullable<decimal> Price { get; set; }  [Display(Name = "Date Of Purchase")]  [Column("DateOfPurchase",TypeName ="datetime")]  public Nullable<System.DateTime> DOP { get; set; }  [Display(Name = "Availability Status")]  public string AvailabilityStatus { get; set; }  [Display(Name = "Category ID")]  public Nullable<long> CategoryID { get; set; }  [Display(Name = "Brand ID")]  public Nullable<long> BrandID { get; set; }  public Nullable<bool> Active { get; set; }  public string Photo { get; set; }  public Nullable<decimal> Quantity { get; set; }  public virtual Brand Brand { get; set; }  public virtual Category Category { get; set; }  }  } |

## Custom HTML Helpers

|  |
| --- |
|  |

Eg:- Suppose we need a file browse button which is reuse in multiple views, so for that we will create one custom HTML helpers (static method) and use them in all the views.

New HTMLHelpers method

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.Web.Mvc;  namespace EFDbFirstApproachExample.HTMLHelpers  {  public static class FileHelper  {  public static MvcHtmlString File(**this** HtmlHelper htmlHelper, string CssClassname)  {  TagBuilder tag = new TagBuilder("input");  tag.MergeAttribute("type", "file");  tag.MergeAttribute("id", "Image");  tag.MergeAttribute("name", "Photo");  tag.MergeAttribute("class", "CssClassname");  return MvcHtmlString.Create(tag.ToString(TagRenderMode.SelfClosing));  }  }  } |

Now call the HTMLHelpers method inside the View, first call the namespace of above of HTMLHelpers method and then call the method File(which we created above).

|  |
| --- |
| @using EFDbFirstApproachExample.HTMLHelpers  @Html.File("form-control-file");  <input type="file" class="form-control-file" id="Image" name="photo" /> |

# Validations

## Validations

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

For Server Side error:-

**Step 1:- Add Data Annotation in Model class**

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.ComponentModel.DataAnnotations;  using System.ComponentModel.DataAnnotations.Schema;  namespace EFDbFirstApproachExample.Models  {  [Table("Products",Schema = "dbo")]  public class Product  {  [Key]  [Display(Name ="Product ID")]  public long ProductID { get; set; }  [Display(Name = "Product Name")]  [Required]  public string ProductName { get; set; }  [Required]  public Nullable<decimal> Price { get; set; }  [Display(Name = "Date Of Purchase")]  [Column("DateOfPurchase",TypeName ="datetime")]  public Nullable<System.DateTime> DOP { get; set; }  [Display(Name = "Availability Status")]  public string AvailabilityStatus { get; set; }  [Display(Name = "Category ID")]  [Required]  public Nullable<long> CategoryID { get; set; }  [Display(Name = "Brand ID")]  [Required]  public Nullable<long> BrandID { get; set; }  public Nullable<bool> Active { get; set; }  public string Photo { get; set; }  public Nullable<decimal> Quantity { get; set; }  public virtual Brand Brand { get; set; }  public virtual Category Category { get; set; }  }  } |

Step 2:- Check for ModelState in Action Method while submitting, if ModelState.IsValid == true (validation succeed)

|  |
| --- |
| [HttpPost]  public ActionResult Create(Product p)  {  CompanyDbContext db = new CompanyDbContext();  if(ModelState.IsValid == true)  {  if (Request.Files.Count >= 1)  {  var file = Request.Files[0];  //Reading all the bytes from the file into the imgBytes  //storing base64String in photo property of Model class  var imgBytes = new Byte[file.ContentLength + 1];  file.InputStream.Read(imgBytes, 0, file.ContentLength);  var base64String = Convert.ToBase64String(imgBytes, 0, imgBytes.Length);  p.Photo = base64String;  }  db.Products.Add(p);  db.SaveChanges();  return RedirectToAction("Index");  }  else  {  return View();  }  } |

## Client Side Validations using jQuery

Step 1:- First need to download the package **jQuery, jQuery.Validation, Microsoft.jQuery.Unobtrusive.Validation**

**Step 2:- Need to import the below scripts in LayOut Page.**

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta name="viewport" content="width=device-width" />  <link href="~/Content/bootstrap.css" rel="stylesheet" />  <script src="~/Scripts/jquery-3.3.1.js"></script>  **<!-- This is use to perform DOM manipulation internally -->**  <script src="~/Scripts/jquery.validate.js"></script>  **<!-- This is use to define the validation rules-->**  <script src="~/Scripts/jquery.validate.unobtrusive.js"></script>  **<!-- This is use for showing or hiding the error message dynamically-->**  <script src="~/Scripts/umd/popper.js"></script>  <script src="~/Scripts/bootstrap.js"></script>  <title>@ViewBag.Title</title>  </head> |

**Step 3:- In the particular add the @Html.ValidationMessageFor() in the particular field.**

|  |
| --- |
| @model EFDbFirstApproachExample.Models.Product  @using EFDbFirstApproachExample.HTMLHelpers  @{  ViewBag.Title = "Create";  Layout = "~/Views/Shared/\_LayoutPage1.cshtml";  }  <h2>Create</h2>  @using (Html.BeginForm("Create", "Products", FormMethod.Post, new { enctype = "multipart/form-data" }))  {  <div class="form-row">  <div class="form-group col-6">  @Html.LabelFor(temp => temp.ProductName)  @Html.TextBoxFor(temp => temp.ProductName, new { @class= "form-control", placeholder= "product Name" })  @Html.ValidationMessageFor(temp => temp.ProductName)  </div>  </div>  <div class="form-row">  <div class="form-group col-6">  @Html.LabelFor(temp => temp.Price)  @Html.TextBoxFor(temp => temp.Price, new { @class = "form-control", placeholder = "Price" })  @Html.ValidationMessageFor(temp => temp.Price)  </div>  </div>  <div class="form-row">  <div class="form-group col-6">  @Html.LabelFor(temp => temp.DOP)  @Html.TextBoxFor(temp => temp.DOP, new { @class = "form-control", placeholder = "Date Of Purchase" })  @Html.ValidationMessageFor(temp => temp.DOP)  </div>  </div>  <div class="form-row">  <div class="form-group col-6">  @Html.LabelFor(temp => temp.AvailabilityStatus)  @Html.DropDownListFor(temp => temp.AvailabilityStatus,new List<SelectListItem>() {  new SelectListItem() { Text= "In Stock", Value="InStock"},  new SelectListItem() { Text= "Out of Stock", Value="OutOfStock"}  },"Please Select", new { @class= "form-control" })  @Html.ValidationMessageFor(temp => temp.AvailabilityStatus)  </div>  </div>  <div class="form-row">  <div class="form-group col-md-6">  @Html.LabelFor(temp => temp.CategoryID)  @Html.DropDownListFor(temp => temp.CategoryID,  new SelectList(ViewBag.Categories, "CategoryID", "CategoryName"),"Please Select",new { @class="form-control"})  @Html.ValidationMessageFor(temp => temp.CategoryID)  </div>  </div>  <div class="form-row">  <div class="form-group col-md-6">  @Html.LabelFor(temp => temp.BrandID)  @Html.DropDownListFor(temp => temp.BrandID,  new SelectList(ViewBag.Brands, "BrandID", "BrandName"),"Please Select", new { @class="form-control"})  @Html.ValidationMessageFor(temp => temp.BrandID)  </div>  </div>  <div class="form-group">  <label for="Image">Image</label>  @Html.File("form-control-file")  </div>  <div class="form-group">  @Html.CheckBoxFor(temp => temp.Active.Value, new { @class = "form-check-input" })  @Html.LabelFor(temp => temp.Active, new { @class = "form-control-label" })  @Html.ValidationMessageFor(temp => temp.Active)  </div>  <button type="submit" class="btn btn-success">Submit</button>  <a class="btn btn-danger" href="/Products/index">Cancel</a>  } |

**Step 4:- For changing the color of validation error message , change the color of cssClassname in Layout view.**

**Here field-validation-error is a predefined**

|  |
| --- |
| <style>  .field-validation-error {  color:red;  }  </style> |

## Customizing Error Messages

For customizing error messages in model class where the data annotation is used pass the argument as error message.

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.ComponentModel.DataAnnotations;  using System.ComponentModel.DataAnnotations.Schema;  namespace EFDbFirstApproachExample.Models  {  [Table("Products",Schema = "dbo")]  public class Product  {  [Key]  [Display(Name ="Product ID")]  public long ProductID { get; set; }  [Display(Name = "Product Name")]  [Required(ErrorMessage ="Product Name cann't be blank")]  public string ProductName { get; set; }  [Required(ErrorMessage = "Price cann't be blank")]  public Nullable<decimal> Price { get; set; }  [Display(Name = "Date Of Purchase")]  [Column("DateOfPurchase",TypeName ="datetime")]  public Nullable<System.DateTime> DOP { get; set; }  [Display(Name = "Availability Status")]  public string AvailabilityStatus { get; set; }  [Display(Name = "Category ID")]  [Required(ErrorMessage = "Category ID cann't be blank")]  public Nullable<long> CategoryID { get; set; }  [Display(Name = "Brand ID")]  [Required(ErrorMessage = "Brand ID cann't be blank")]  public Nullable<long> BrandID { get; set; }  public Nullable<bool> Active { get; set; }  public string Photo { get; set; }  public Nullable<decimal> Quantity { get; set; }  public virtual Brand Brand { get; set; }  public virtual Category Category { get; set; }  }  } |

## Displaying Validation Summary

Validation Summary is basically showing all the validation errors at the top of the form or at the bottom of the form.

**Here I have added at the bottom of the page.**

|  |
| --- |
| @model EFDbFirstApproachExample.Models.Product  @using EFDbFirstApproachExample.HTMLHelpers  @{  ViewBag.Title = "Create";  Layout = "~/Views/Shared/\_LayoutPage1.cshtml";  }  <h2>Create</h2>  @using (Html.BeginForm("Create", "Products", FormMethod.Post, new { enctype = "multipart/form-data" }))  {  <div class="form-row">  <div class="form-group col-6">  @Html.LabelFor(temp => temp.ProductName)  @Html.TextBoxFor(temp => temp.ProductName, new { @class= "form-control", placeholder= "product Name" })  @Html.ValidationMessageFor(temp => temp.ProductName)  </div>  </div>  <div class="form-row">  <div class="form-group col-6">  @Html.LabelFor(temp => temp.Price)  @Html.TextBoxFor(temp => temp.Price, new { @class = "form-control", placeholder = "Price" })  @Html.ValidationMessageFor(temp => temp.Price)  </div>  </div>  <div class="form-row">  <div class="form-group col-6">  @Html.LabelFor(temp => temp.DOP)  @Html.TextBoxFor(temp => temp.DOP, new { @class = "form-control", placeholder = "Date Of Purchase" })  @Html.ValidationMessageFor(temp => temp.DOP)  </div>  </div>  <div class="form-row">  <div class="form-group col-6">  @Html.LabelFor(temp => temp.AvailabilityStatus)  @Html.DropDownListFor(temp => temp.AvailabilityStatus,new List<SelectListItem>() {  new SelectListItem() { Text= "In Stock", Value="InStock"},  new SelectListItem() { Text= "Out of Stock", Value="OutOfStock"}  },"Please Select", new { @class= "form-control" })  @Html.ValidationMessageFor(temp => temp.AvailabilityStatus)  </div>  </div>  <div class="form-row">  <div class="form-group col-md-6">  @Html.LabelFor(temp => temp.CategoryID)  @Html.DropDownListFor(temp => temp.CategoryID,  new SelectList(ViewBag.Categories, "CategoryID", "CategoryName"),"Please Select",new { @class="form-control"})  @Html.ValidationMessageFor(temp => temp.CategoryID)  </div>  </div>  <div class="form-row">  <div class="form-group col-md-6">  @Html.LabelFor(temp => temp.BrandID)  @Html.DropDownListFor(temp => temp.BrandID,  new SelectList(ViewBag.Brands, "BrandID", "BrandName"),"Please Select", new { @class="form-control"})  @Html.ValidationMessageFor(temp => temp.BrandID)  </div>  </div>  <div class="form-group">  <label for="Image">Image</label>  @Html.File("form-control-file")  </div>  <div class="form-group">  @Html.CheckBoxFor(temp => temp.Active.Value, new { @class = "form-check-input" })  @Html.LabelFor(temp => temp.Active, new { @class = "form-control-label" })  @Html.ValidationMessageFor(temp => temp.Active)  </div>  @Html.ValidationSummary()  <button type="submit" class="btn btn-success">Submit</button>  <a class="btn btn-danger" href="/Products/index">Cancel</a>  } |

**For changing the color of validation error message, change the color of cssClassname in Layout view.**

**Here validation-summary-errors is a predefined and by default it is showing black color.**

|  |
| --- |
| <style>  .field-validation-error, .validation-summary-errors  {  color: red;  }  </style> |

## All Types of Validation

For all other types of validation like Alphabets only, minlength, maxlength, range etc. Add the particular data annotation in Model class.

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.ComponentModel.DataAnnotations;  using System.ComponentModel.DataAnnotations.Schema;  namespace EFDbFirstApproachExample.Models  {  [Table("Products",Schema = "dbo")]  public class Product  {  [Key]  [Display(Name ="Product ID")]  public long ProductID { get; set; }  [Display(Name = "Product Name")]  [Required(ErrorMessage ="Product Name cann't be blank")]  [RegularExpression(@"^[A-Za-z]\*$", ErrorMessage ="Alphabets only")]  [MinLength(6,ErrorMessage ="Product name should contain atleast 6 characters")]  [MaxLength(20,ErrorMessage ="Product name can be maximum 20 charaters long")]  public string ProductName { get; set; }  [Required(ErrorMessage = "Price cann't be blank")]  [Range(0,10000,ErrorMessage ="Price should be in between 0 to 10000")]  public Nullable<decimal> Price { get; set; }  [Display(Name = "Date Of Purchase")]  [Column("DateOfPurchase",TypeName ="datetime")]  public Nullable<System.DateTime> DOP { get; set; }  [Display(Name = "Availability Status")]  public string AvailabilityStatus { get; set; }  [Display(Name = "Category ID")]  [Required(ErrorMessage = "Category ID cann't be blank")]  public Nullable<long> CategoryID { get; set; }  [Display(Name = "Brand ID")]  [Required(ErrorMessage = "Brand ID cann't be blank")]  public Nullable<long> BrandID { get; set; }  public Nullable<bool> Active { get; set; }  public string Photo { get; set; }  public Nullable<decimal> Quantity { get; set; }  public virtual Brand Brand { get; set; }  public virtual Category Category { get; set; }  }  } |

## Custom Validations

Custom validations is required if we want to validate something from database.

|  |
| --- |
|  |

**Step 1:-**

In Project add new folder (CustomValidations) and add a class to that folder give name particular to that functionality like for this (DivisibeBy10**Attribute.cs**) **it must suffix with name Attribute which we will use as data annotation in model class.**

**Step 2:-**

**Create the custom validation method in the new class as mentioned in step 1**

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.ComponentModel.DataAnnotations;  namespace EFDbFirstApproachExample.CustomValidations  {  public class DivisibleBy10Attribute: ValidationAttribute  {  protected override ValidationResult IsValid(object value,ValidationContext validationContext)  {  double price = Convert.ToDouble(value);  if (price % 10 == 0)  {  return ValidationResult.Success;  }  else  {  return new ValidationResult(this.ErrorMessage);  }  }  }  } |

**Step 3:- Apply the same to the specify model property. Now model class import the above namespace and apply the data annotation.**

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.ComponentModel.DataAnnotations;  using System.ComponentModel.DataAnnotations.Schema;  using EFDbFirstApproachExample.CustomValidations;  namespace EFDbFirstApproachExample.Models  {  [Table("Products",Schema = "dbo")]  public class Product  {  [Key]  [Display(Name ="Product ID")]  public long ProductID { get; set; }  [Display(Name = "Product Name")]  [Required(ErrorMessage ="Product Name cann't be blank")]  [RegularExpression(@"^[A-Za-z]\*$", ErrorMessage ="Alphabets only")]  [MinLength(6,ErrorMessage ="Product name should contain atleast 6 characters")]  [MaxLength(20,ErrorMessage ="Product name can be maximum 20 charaters long")]  public string ProductName { get; set; }  [Required(ErrorMessage = "Price cann't be blank")]  [Range(0,10000,ErrorMessage ="Price should be in between 0 to 10000")]  [DivisibleBy10(ErrorMessage ="Price should be in multiple of 10")]  public Nullable<decimal> Price { get; set; }  [Display(Name = "Date Of Purchase")]  [Column("DateOfPurchase",TypeName ="datetime")]  public Nullable<System.DateTime> DOP { get; set; }  [Display(Name = "Availability Status")]  public string AvailabilityStatus { get; set; }  [Display(Name = "Category ID")]  [Required(ErrorMessage = "Category ID cann't be blank")]  public Nullable<long> CategoryID { get; set; }  [Display(Name = "Brand ID")]  [Required(ErrorMessage = "Brand ID cann't be blank")]  public Nullable<long> BrandID { get; set; }  public Nullable<bool> Active { get; set; }  public string Photo { get; set; }  public Nullable<decimal> Quantity { get; set; }  public virtual Brand Brand { get; set; }  public virtual Category Category { get; set; }  }  } |

**Now disadvantages of Custom Validation is it will invoke the validation after submitting to the server, so it is not a client side validation**

# ASP.NET Identity

## Introduction to Asp.Net Identity

|  |
| --- |
|  |

In order to work with ASP.NET identity, we need to install the below packages.

# Filters

Filters execute at a specific situation, while executing the action method.

* **In ASP.NET MVC a user request is routed to an appropriate controller and action method.**
* **However there may be some circumstances where we want to execute some logic before or after the action method executes ASP.NET MVC provides filters for this purpose.**
* **Filters can be applied to an action method or controller.**
* **If applied to action method filter execute before or after that action method and**
* **If it applied to controller filters execute automatically before or after the any action method of that controller.**
* **And also can be applied globally which affects the entire controller.**

|  |
| --- |
|  |

|  |
| --- |
|  |

# Web Api

## What is ASP.NET Web API

ASP.Net Web Api is a framework for building Web API’s, i. e. HTTP based services on top of the .NET framework.

These services can be consumed by the broad ranges of clients :-

1. Browsers.
2. Mobile Applications.
3. Desktop Applications.
4. IOT’s

## What are RESTful Services?

REST stands for Representational State Transfer. REST is an architectural pattern for creating services.

REST architectural pattern specifies a set of constraints that a system should adhere to.

## REST Constraints

* Client Server
* Stateless (which means we should not store anything on the server with respect to client, client should give the necessary information every time.)
* Cacheable. (Some data like list of employee, customer we can cache in the client side, so that it should hit the server again and again).
* Uniform Interface (GET, PUT, POST, DELETE).
* Layered System
* Code on Demand (is optional).

## Difference between WCF and Web API?

Both can create RESTful services. But the problem with WCF is it needs lots of configuration to create

## When to use WCF over ASP.NET Web API?

* Creating services that are transport/protocol independent. Single service with multiple end points. Eg suppose we need service which communicate with Java Client which accepts (XML format and TCP protocol) and other Client .NET clients which accepts (JSON format and http protocol) for this kind of scenario WCF is right choose, we can create two different end points for this to communicate differently.
* You have an existing SOAP service you must support but want to add REST to reach more clients.
* .NET 3.5 limitations.

## Web API controller always inherit from ApiController

## Routing in Web API

|  |
| --- |
|  |

url : …api/ControllerName – GET method without id parameter.

url: ..api/ControlName/1 – GET method with id parameter will call. Remember id is optional

## HTTP – Terms and Concepts

* **Request Verbs** (GET, POST, PUT, DELETE).
* **Request Headers**: Contains additional information about the request. Example – What type of response is required? (Like response from server is required XML, JSON etc.)
* **Request Body**: Contains the data to send to the server. ( Data formats may be XML or JSON)
* **Response Body**: Contains the data send as a response from the server.
* **Response Status Code**: Provide the client, the status of the request

## Implementing Get functionally

|  |
| --- |
|  |

Modifying the get Method with necessary Status code

|  |
| --- |
|  |

## Content Negotiation

* Using **Accept header** the client can specify the format of the response they want from the server.
* Accept: application/xml
* Accept: application/json
* Multiple values can also be specified for the Accept header.
* Accept: application/xml,application/json

Here the response will return as xml which is defined at first.

* We can also specify quality factor
* application/xml;q=0.8,application/json;q=0.5

Here the server will see which one has higher value based on that it will return the type, her it will be xml.

**If Accept header is not specified, by default web API returns JSON data.**

**The formatters are used by the server for both request and response messages. These formatters are called Media Type Formatters.**

|  |
| --- |
|  |

**The above code is implemented in WebApiConfig.cs**

## MediaTypeFormatter

* **What is MediaTypeFormatter?**
* MediaTypeFormatter is an abstract class from which JSONMediaTypeFormatter and XMLMediaTypeFormatter classes inherit from.
* How to return only JSON from ASP.Net Web API service, if client is asking for some other format?

|  |
| --- |
|  |

* How to return only XML from ASP.Net Web API service, if client is asking for some other format?

|  |
| --- |
|  |

All the code is change in WebApiConfig.cs

* How to return JSON instead of XML from ASP.NET Web API service when a request is made from the browser?
* **Approach 1:-**

|  |
| --- |
|  |

* **Approach 2:-**

|  |
| --- |
|  |

* Creating Custom Media Type Formatter

|  |
| --- |
|  |

## Implementing POST Method

|  |
| --- |
|  |

* If a method return type is void, by default status code **204 No Content** is returned.
* When a new item is created, we should be returning **status code 201** item created.
* With **201 Status code we should also return the location i.e. URI of the newly created item**.
* When an item is not found, instead of returning NULL and **status code 200 OK**, return **404 NOT Found** along with a meaningful message such as “Employee with ID =101 NOT Found”

## Implementing DELETE Method.

|  |
| --- |
|  |

## Implementing PUT method

|  |
| --- |
|  |

## Custom Method names in Web API

|  |
| --- |
|  |

## Web API Query String Parameters

|  |
| --- |
|  |

|  |
| --- |
|  |

## FromBody and FromUri Attributes

|  |
| --- |
|  |

## Call Asp.NET Web API from Jquery

|  |
| --- |
|  |

## ASP.Net Web API – Cross Domain Ajax

|  |
| --- |
|  |

## Cross-Origin resource sharing in Web API

|  |
| --- |
|  |

## Enable SSL in Visual Studio

|  |
| --- |
|  |

# Important Points

* In a View if we <a href="/Products/Create">Create Product</a> this will not if the application is deployed in IIS. So it is always recommended to use <a href="@Url.Action("Create","Products")">Create Product</a> where **Create** is a Action name and **Products** is Controller name
* For Security purpose bind attribute is very helpful.
* If we changing anything in Model class, need to update the database otherwise it will give error.

1. add-migration MyValidation
2. update-database