

Creating your first asp.net mvc application - Part 3

Suggested Videos

[Part 1 - Installing asp.net mvc](#)

[Part 2 - Which asp.net mvc version is my mvc application using](#)

In this video we will discuss about

1. Creating an asp.net mvc application
2. Understand how mvc request is processed as apposed to webform request

Creating an mvc application:

1. Open visual studio
2. Click **File > New Project**
3. Select **"Web"** from **"Installed Templates"** section
4. Select **ASP.NET MVC 4 Web Application**
5. Set Name="**MVCDemo**"
6. Click OK
7. Select **"Empty"** template. Select **"Razor"** as the ViewEngine. There are 2 built in view engines - Razor and ASPX. Razor is preferred by most mvc developers. We will discuss about Razor view engine in detail in a later video session.
8. At this point you should have an mvc application created.

Notice that in the solution explorer, you have several folders - **Models, Views, Controllers** etc. As the names suggest these folders are going to contain Models, Views, and Controllers. We will discuss about Models, Views, and Controllers in a later video session.

Now let's add a controller to our project

1. Right Click on **"Controllers"** folder
2. Select **Add > Controller**
3. Set Controller Name = **HomeController**
4. Leave rest of the defaults and click **"Add"**

We should have HomeController.cs added to "Controllers" folder.

At this point run the application by pressing **CTRL+F5**. Notice that you get an error as shown below.

Server Error in '/' Application.

The view 'Index' or its master was not found or no view engine supports the searched locations. The following locations were searched:

*~/Views/Home/Index.aspx
~/Views/Home/Index.ascx
~/Views/Shared/Index.aspx
~/Views/Shared/Index.ascx
~/Views/Home/Index.cshtml
~/Views/Home/Index.vbhtml
~/Views/Shared/Index.cshtml
~/Views/Shared/Index.vbhtml*

To fix this error, we need to add a view with name, **"Index"**. We will discuss about views in detail in a later video session. Let's fix it another way. The following is the function that is automatically added to HomeController class

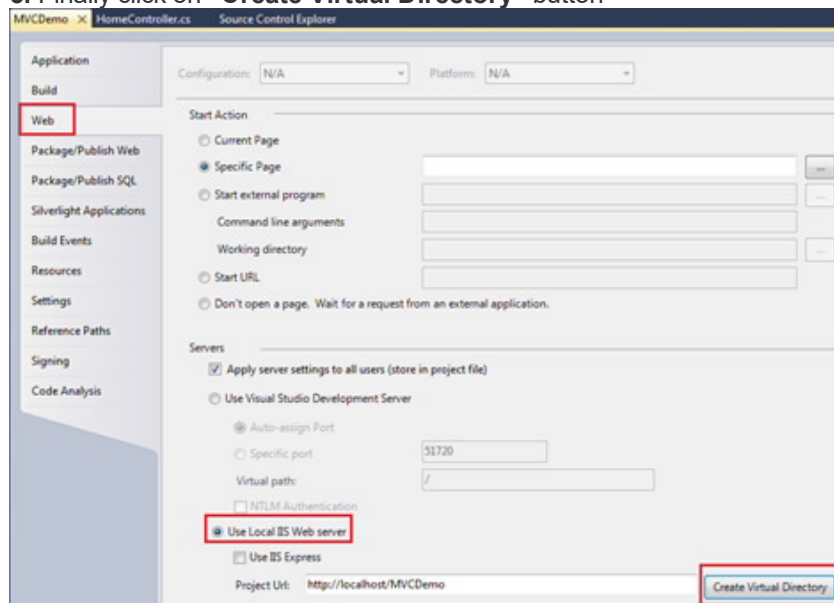
```
public ActionResult Index()
{
    return View();
}
```

Change the return type of Index() function from **"ActionResult"** to **"string"**, and return string **"Hello from MVC Application"** instead of View().

```
public string Index()
{
    return "Hello from MVC Application";
}
```

Run the application and notice that, the string is rendered on the screen. When you run the application, by default it is using **built-in asp.net development** server. Let's use IIS, to run the application instead of the built-in asp.net development server.

1. In the solution explorer, right click on the project and select **"Properties"**
2. Click on **"Web"** tab
3. Select **"Use Local IIS Web Server"** radio button
4. Notice that the Project Url is set to **http://localhost/MVCDemo** by default
5. Finally click on **"Create Virtual Directory"** button



Run the application, and notice that the URL is **"http://localhost/MVCDemo/"**

Now change the URL to **"http://localhost/MVCDemo/Home/index"**

In the URL **"Home"** is the **name of the controller** and **"Index"** is the **method** within HomeController class.

So the important point to understand is that the **URL** is mapped to a **controller action method**. Whereas in web forms application, the **URL** is mapped to a **physical file**. For example, in a web application, if we have to display the same message.

1. We add a webform and in the Page_load() event include Response.Write("Hello from ASP.NET Web Forms Application");
2. We then access WebForm1.aspx as shown below
http://localhost/WebFormsApplication/WebForm1.aspx
3. The Page load event gets executed and the message string is displayed.

Controllers in an mvc application - Part 4

Suggested Videos

[Part 1 - Installing asp.net mvc](#)

[Part 2 - Which asp.net mvc version is my mvc application using](#)

[Part 3 - Creating your first asp.net mvc application](#)

In this video we will discuss about **controllers**. Please watch [Part 3 of MVC tutorial](#) before proceeding. In Part 3, we discussed that, the URL - `http://localhost/MVCDemo/Home/Index` will invoke **Index()** function of **HomeController** class. So, the question is, where is this mapping defined. The mapping is defined in **Global.asax**. Notice that in **Global.asax** we have **RegisterRoutes()** method.

```
RouteConfig.RegisterRoutes(RouteTable.Routes);
```

Right click on this method, and select **"Go to Definition"**. Notice the implementation of **RegisterRoutes()** method in **RouteConfig** class. This method has got a default route.

```
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 }
    );
}
```

The following URL does not have id. This is not a problem because id is optional in the default route.
`http://localhost/MVCDemo/Home/Index`

Now pass **id** in the URL as shown below and press enter. Nothing happens.
`http://localhost/MVCDemo/Home/Index/10`

Change the **Index()** function in **HomeController** as shown below.

```
public string Index(string id)
{
    return "The value of Id = " + id;
}
```

Enter the following URL and press enter. We get the output as expected.
`http://localhost/MVCDemo/Home/Index/10`

In the following URL, 10 is the value for id parameter and we also have a query string **"name"**.
`http://localhost/MVCDemo/home/index/10?name=Pragim`

Change the **Index()** function in **HomeController** as shown below, to read both the parameter values.

```
public string Index(string id, string name)
{
    return "The value of Id = " + id + " and Name = " + name;
}
```

Just like web forms, you can also use **"Request.QueryString"**

```
public string Index(string id)
{
    return "The value of Id = " + id + " and Name = " + Request.QueryString["name"];
}
```

Part 5 - Views in an mvc application

Suggested Videos

[Part 2 - Which asp.net mvc version is my mvc application using](#)

[Part 3 - Creating your first asp.net mvc application](#)

[Part 4 - Controllers in an mvc application](#)

In this video we will discuss **views**. Please watch [Part 4 of MVC tutorial](#) before proceeding. Let's change the **Index()** method in HomeController to return a list of country names. Since country names are strings, return List<string>. Get rid of id and name parameters.

```
public List<string> Index()
{
    return new List<string>()
    {
        "India",
        "US",
        "UK",
        "Canada"
    };
}
```

Run the application and notice that the output is not as expected.

System.Collections.Generic.List`1[System.String]

To correct this problem, let's add a view to our project.

1. Right click on the **Index()** function
2. Click on **"Add View"**
3. Notice that, the view name in **"Add View"** dialog box matches the name of the controller action method
4. Select **"Razor"** as the view engine
5. Leave the rest of the defaults as is, and click **"Add"** button

Make the following modifications to the Index() function of the HomeController class, so that, the HomeController returns a view instead of List<string>.

// Change the return type from List<string> to ActionResult

```
public ActionResult Index()
{
    // Store the list of Countries in ViewBag.
    ViewBag.Countries = new List<string>()
    {
        "India",
        "US",
        "UK",
        "Canada"
    };

    // Finally return a view
    return View();
}
```

We will discuss **ViewBag & ViewData**, and the differences between them in our next video session. For now, understand that, ViewBag & ViewData is a mechanism to pass data from the controller to the view.

Please Note: To pass data from controller to a view, It's always a good practice to use strongly typed view models instead of using ViewBag & ViewData. We will discuss view models in a later video session.

Now, copy and paste the following code in "Index.cshtml" view

```
@{
    ViewBag.Title = "Countries List";
}

<h2>Countries List</h2>

<ul>

@foreach (string strCountry in ViewBag.Countries)
{
    <li>@strCountry</li>
}
```


Please Note: We use "@" symbol to switch between html and c# code. We will discuss razor views and their syntax in a later video session.

Part 6 - ViewData and ViewBag in mvc

Suggested Videos

[Part 3 - Creating your first asp.net mvc application](#)

[Part 4 - Controllers in an mvc application](#)

[Part 5 - Views in an mvc application](#)

In this video we will discuss

1. What is ViewData
2. What is ViewBag
3. Difference between ViewData and ViewBag

Both **ViewData** and **ViewBag** are used to pass data from a controller to a view. ViewData is a dictionary of objects that are stored and retrieved using strings as keys. The syntax of ViewData is very similar to that of ViewState, SessionState and ApplicationState.

// Storing data in ViewData

```
ViewData["YourData"] = "SomeData";
```

// Retrieving data from ViewData

```
string strData = ViewData["YourData"].ToString();
```

ViewData does not provide compile time error checking. For example, if you mis-spell the key names you wouldn't get any compile time error. You get to know about the error only at runtime.

ViewBag uses the dynamic feature that was introduced in to C# 4.0. It allows an object to have properties dynamically added to it. Using ViewBag the above code can be rewritten as below.

// Storing data in ViewBag

```
ViewBag.YourData = "SomeData";
```

// Retrieving data from ViewBag

```
string strData = ViewBag.YourData;
```

Just like ViewData, ViewBag does not provide compile time error checking. For example, if you mis-spell the property name, you wouldn't get any compile time error. You get to know about the error only at runtime.

Internally ViewBag properties are stored as name/value pairs in the ViewData dictionary.

Please Note: To pass data from controller to a view, It's always a good practice to use strongly typed view models instead of using ViewBag & ViewData. Strongly typed view models provide compile time error checking. We will discuss view models in a later video session.

Part 7 - Models in an mvc application

Suggested Videos

[Part 4 - Controllers in an mvc application](#)

[Part 5 - Views in an mvc application](#)

[Part 6 - ViewData and ViewBag in mvc](#)

In this video we will discuss **models** in an **mvc application**.

Let's understand **models** with an example. We want to retrieve an employee information from tblEmployee table and display it as shown below.

Employee Details

```
Employee ID: 101
Name:      John
Gender:    Male
City:      London
```

To encapsulate Employee information, add Employee model class to the Models folder. To do this

1. Right click on "**Models**" folder > Add > Class
2. Name the class as Employee.cs
3. Click "**Add**"

Copy and paste the following code in Employee.cs class file.

```
public class Employee
{
    public int EmployeeId { get; set; }
    public string Name { get; set; }
    public string Gender { get; set; }
    public string City { get; set; }
}
```

Now let's Add EmployeeController class to "Controllers" folder. To do this

1. Right click on "Controllers" folder > Add > Controller
2. Use EmployeeController as the name
3. Click "Add"

We want to use "**Employee**" model class in **EmployeeController**. So copy and paste the following "**using**" statement in "EmployeeController.cs"

```
using MVCDemo.Models;
```

By default an Index() Action method is created in EmployeeController. Change the name of the function to **Details()**. Create an instance of Employee class. For now we will hard code Employee data in this class. In a later video session, we will discuss about retrieving employee information from the database table tblEmployee. At this point EmployeeController should look as shown below.

```
public ActionResult Details()
{
    Employee employee = new Employee()
    {
        EmployeeId = 101,
        Name = "John",
        Gender = "Male",
        City = "London"
    };

    return View();
}
```

Now, we need to pass the **employee** model object that we constructed in EmployeeController to a view, so the view can generate the HTML and send it to the requested client. To do this we first need to add a view. To add a view

1. Right click on Details() action method and select "Add View" from the context menu
2. Set
 - a) View Name = Details
 - b) View Engine = Razor
 - c) Select "Create strongly typed view" check box
 - d) From the "Model class" dropdownlist, select "Employee (MVCDemo.Models)"

Note: If Employee class is not visible in the dropdownlist, please build your project and then try adding the view again.

3. Finally click "Add"

At this point, **Details.cshtml** should be added to **"Employee"** folder. Please note that "Employee" folder is automatically created and added to "Views" folder.

Copy and paste the following code in Details.cshtml file.

```
@model MVCDemo.Models.Employee
```

```
@{
    ViewBag.Title = "Employee Details";
}
```

```
<h2>Employee Details</h2>
```

```
<table style="font-family:Arial">
```

```
<tr>
```

```
<td>
```

```
Employee ID:
```

```
</td>
```

```
<td>
```

```
@Model.EmployeeId
```

```
</td>
```

```
</tr>
```

```
<tr>
```

```
<td>
```

```
Name:
```

```
</td>
```

```
<td>
```

```
@Model.Name
```

```
</td>
```

```
</tr>
```

```
<tr>
```

```
<td>
```

```
Gender:
```

```
</td>
```

```
<td>
```

```
@Model.Gender
```

```
</td>
```

```
</tr>
```

```
<tr>
```

```
<td>
```

```
City:
```

```
</td>
```

```
<td>
```

```
@Model.City
```

```
</td>
```

```
</tr>
```

```
</table>
```

At this point if you run the project, and if you navigate to the following URL, you get a runtime error stating - **Object reference not set to an instance of an object.**

<http://localhost/MVCDemo/Employee/Details>

To fix this error, pass **"Employee"** object to the view. The **"return"** statement in Details() action method need to be modified as shown below.

```
return View(employee);
```

That's it. Run the application and navigate to <http://localhost/MVCDemo/Employee/Details>. We should get the output as expected.

Part 8 - Data access in mvc using entity framework

Suggested Videos

[Part 5 - Views in an mvc application](#)

[Part 6 - ViewData and ViewBag in mvc](#)

[Part 7 - Models in MVC](#)

The controller responds to URL request, gets data from a model and hands it over to the view. The view then renders the data. Model can be entities or business objects.

In **part 7**, we have built **Employee** entity.

```
public class Employee
{
    public int EmployeeId { get; set; }
    public string Name { get; set; }
    public string Gender { get; set; }
    public string City { get; set; }
}
```

In this video, we will discuss, retrieving data from a database table **tblEmployee** using entity framework. In a later video, we will discuss using business objects as our model.

Step 1: Install entity framework, if you don't have it installed already on your computer. At the time of this recording the latest version is 5.0.0.0. Using nuget package manager, is the easiest way to install. A reference to EntityFramework.dll is automatically added. Open visual studio > Tools > Library Package Manager > Manage NuGet Packages for Solution

Step 2: Add **EmployeeContext.cs** class file to the Models folder. Add the following **"using"** declaration.

```
using System.Data.Entity;
```

Copy & paste the following code in EmployeeContext.cs

```
public class EmployeeContext : DbContext
{
    public DbSet<Employee> Employees {get; set;}
}
```

EmployeeContext class derives from **DbContext** class, and is responsible for establishing a connection to the database. So the next step, is to include connection string in web.config file.

Step 3: Add a connection string, to the web.config file, in the root directory.

```
<connectionStrings>
  <add name="EmployeeContext"
        connectionString="server=.; database=Sample; integrated security=SSPI"
        providerName="System.Data.SqlClient"/>
</connectionStrings>
```

Step 4: Map **"Employee"** model class to the database table, **tblEmployee** using **"Table"** attribute as shown below.

```
[Table("tblEmployee")]
public class Employee
{
    public int EmployeeId { get; set; }
    public string Name { get; set; }
    public string Gender { get; set; }
    public string City { get; set; }
}
```

Note: "Table" attribute is present in "System.ComponentModel.DataAnnotations.Schema" namespace.

Step 5: Make the changes to "Details()" action method in "EmployeeController" as shown below.

```
public ActionResult Details(int id)
{
    EmployeeContext employeeContext = new EmployeeContext();
    Employee employee = employeeContext.Employees.Single(x => x.EmployeeId == id);

    return View(employee);
}
```

Step 6: Finally, copy and paste the following code in **Application_Start()** function,

in **Global.asax** file. Database class is present "in **System.Data.Entity**" namespace. Existing databases do not need, database initializer so it can be turned off.

`Database.SetInitializer<MVCDemo.Models.EmployeeContext>(null);`

That's it, run the application and navigate to the following URL's and notice that the relevant employee details are displayed as expected

<http://localhost/MVCDemo/Employee/details/1>

<http://localhost/MVCDemo/Employee/details/2>

Part 9 - Generate hyperlinks using actionlink html helper

Suggested Videos

[Part 6 - ViewData and ViewBag in mvc](#)

[Part 7 - Models in MVC](#)

[Part 8 - Data access in MVC using entity framework](#)

In this video we will discuss, generating hyperlinks using actionlink html helper, for navigation between MVC pages.

[Please watch Part 8](#), before proceeding.

We want to display all the employees in a **bulleted list** as shown below. Notice that all the employee names are rendered as **hyperlinks**.

Employee List

- [Mark](#)
- [John](#)
- [Mary](#)
- [Mike](#)

When the hyperlink is clicked, the user will be redirected to employee details page, displaying the full details of the employee as shown below.

Employee Details

Employee Id : 1

Name: Mark

Gender: Male

City: London

[Back to List](#)

Copy and paste the following **Index()** action method in **EmployeeController** class. This method retrieves the list of employees, which is then passed on to the view for rendering.

```
public ActionResult Index()
{
    EmployeeContext employeeContext = new EmployeeContext();
    List<Employee> employees = employeeContext.Employees.ToList();

    return View(employees);
}
```

At the moment, we **don't have a view** that can display the list of employees. To add the view

1. Right click on the Index() action method
2. Set
View name = **Index**
View engine = **Razor**
Select, **Create a strongly-typed view checkbox**
Select "**Employee**" from "**Model class**" dropdownlist
3. Click Add

At this point, "**Index.cshtml**" view should be generated. Copy and paste the following code in "Index.cshtml".

```
@model IEnumerable<MVCDemo.Models.Employee>

@using MVCDemo.Models;

<div style="font-family:Arial">
@{
    ViewBag.Title = "Employee List";
}

<h2>Employee List</h2>
<ul>
@foreach (Employee employee in @Model)
{
    <li>@Html.ActionLink(employee.Name, "Details", new { id = employee.EmployeeId })</li>
}
</ul>
</div>
```

Points to Remember:

1. @model is set to `IEnumerable<MVCDemo.Models.Employee>`
2. We are using `Html.ActionLink` html helper to generate links

Copy and paste the following code in Details.cshtml

```
@Html.ActionLink("Back to List", "Index")
```

Part 10 - Working with multiple tables in mvc

Suggested Videos

[Part 7 - Models in MVC](#)

[Part 8 - Data access in MVC using entity framework](#)

[Part 9 - Generate hyperlinks using actionlink html helper](#)

Please [watch Part 9](#), before proceeding.

In this video we will discuss working with 2 related tables in MVC

1. tblDepartment
2. tblEmployee

Use the sql script to create and populate these 2 tables

Create table tblDepartment

```
(
    Id int primary key identity,
    Name nvarchar(50)
)
```

Insert into tblDepartment values('IT')

Insert into tblDepartment values('HR')

```
Insert into tblDepartment values('Payroll')
```

```
Create table tblEmployee
(
EmployeeId int Primary Key Identity(1,1),
Name nvarchar(50),
Gender nvarchar(10),
City nvarchar(50),
DepartmentId int
)
```

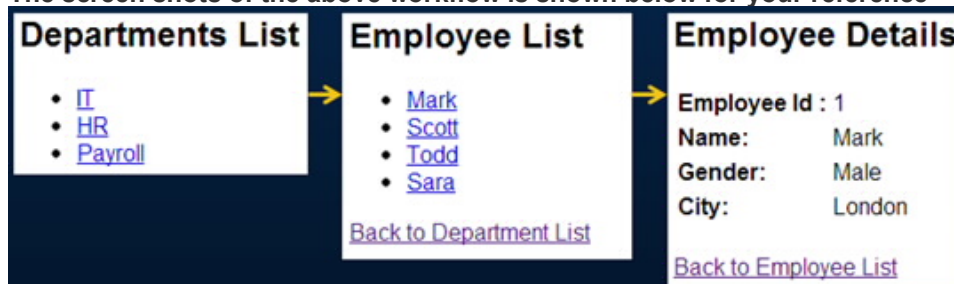
```
Alter table tblEmployee
add foreign key (DepartmentId)
references tblDepartment(Id)
```

```
Insert into tblEmployee values('Mark','Male','London',1)
Insert into tblEmployee values('John','Male','Chennai',3)
Insert into tblEmployee values('Mary','Female','New York',3)
Insert into tblEmployee values('Mike','Male','Sydney',2)
Insert into tblEmployee values('Scott','Male','London',1)
Insert into tblEmployee values('Pam','Female','Falls Church',2)
Insert into tblEmployee values('Todd','Male','Sydney',1)
Insert into tblEmployee values('Ben','Male','New Delhi',2)
Insert into tblEmployee values('Sara','Female','London',1)
```

This is what we want to achieve

1. Display all the departments from **tblDepartments** table. The Department names should be rendered as hyperlinks.
2. On clicking the **department name link**, all the employees in the department should be displayed. The employee names should be rendered as hyperlinks.
3. On clicking the **employee name link**, the full details of the employee should be displayed.
4. A link should also be provided on the employee full details page to navigate back to **Employee list page**. Along the same lines, a link should also be provided on the employee list page to navigate back to Departments list page.

The screen shots of the above workflow is shown below for your reference



Implementing Departments List:

Step 1: Right click on the **"Models"** folder and add a class file with name=**Department.cs**. Copy and paste the following code.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.ComponentModel.DataAnnotations.Schema;
```

```
namespace MVCDemo.Models
{
    [Table("tblDepartment")]
    public class Department
    {
        public int ID { get; set; }
        public string Name { get; set; }
        public List<Employee> Employees { get; set; }
    }
}
```

Step 2: Add **"Departments"** property to **"EmployeeContext"** class that is present

in "**EmployeeContext.cs**" file in "**Models**" folder.

```
public class EmployeeContext : DbContext
{
    public DbSet<Department> Departments { get; set; }
    public DbSet<Employee> Employees { get; set; }
}
```

Step 3: Right click on the "**Controllers**" folder and add a Controller, with name=**DepartmentController**. Copy and paste the following code.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using MVCDemo.Models;

namespace MVCDemo.Controllers
{
    public class DepartmentController : Controller
    {
        public ActionResult Index()
        {
            EmployeeContext employeeContext = new EmployeeContext();
            List<Department> departments = employeeContext.Departments.ToList();
            return View(departments);
        }
    }
}
```

Step 4: Right click on the **Index()** action method in **DepartmentController** class and select "**Add View**" from the context menu. Set

View name = **Index**

View engine = **Razor**

Select "**Create Strongly-typed view**" checkbox

Select **Department** class, from "**Model class**" dropdownlist

Click "**Add**" button

Copy and paste the following code in Index.cshtml view file in Department folder

```
@using MVCDemo.Models;

@model IEnumerable<Department>

<div style="font-family:Arial">
    @{
        ViewBag.Title = "Departments List";
    }

    <h2>Departments List</h2>
    <ul>
        @foreach (Department department in @Model)
        {
            <li>@Html.ActionLink(department.Name, "Index", "Employee",
                new { departmentId = department.ID }, null)</li>
        }
    </ul>
</div>
```

Changes to Employee List and Detail pages

Add "**DepartmentId**" property to "**Employee**" model class that is present in **Employee.cs** file in "**Models**" folder.

```
[Table("tblEmployee")]
public class Employee
{
    public int EmployeeId { get; set; }
    public string Name { get; set; }
    public string Gender { get; set; }
    public string City { get; set; }
    public int DepartmentId { get; set; }
}
```

Add "**departmentId**" parameter to **Index()** action method in "**EmployeeController**" class that is present in "**EmployeeController.cs**" file in "**Controllers**" folder. Use the "**departmentId**" parameter to filter the list of employees as shown below.

```
public ActionResult Index(int departmentId)
{
    EmployeeContext employeeContext = new EmployeeContext();
    List<Employee> employees = employeeContext.Employees.Where(emp => emp.DepartmentId
    == departmentId).ToList();

    return View(employees);
}
```

Copy and paste the following line in "**Index.cshtml**" that is present in "**Employee**" folder in "**Views**" folder. With this change we are able to generate an action link to redirect the user to a different controller action method.

```
@Html.ActionLink("Back to Department List", "Index", "Department")
```

Change the following line in "**Details.cshtml**" that is present in "**Employee**" folder in "**Views**" folder.

CHANGE THIS LINE `@Html.ActionLink("Back to List", "Index")`

TO

```
@Html.ActionLink("Back to Employee List", "Index", new { departmentId = @Model.DepartmentId })
```

Part 11 - Using business objects as model in mvc

Suggested Videos

[Part 8 - Data access in MVC using entity framework](#)

[Part 9 - Generate hyperlinks using actionlink html helper](#)

[Part 10 - Working with multiple tables in MVC](#)

In this video, we will discuss using business objects as model. Until now, we have been using entity framework and entities. Entities are mapped to database tables, and object relational mapping tools like Entity Framework, nHibernate, etc are used to retrieve and save data. Business objects contain both state(data) and behaviour, that is logic specific to the business.

In MVC there are several conventions that needs to be followed. For example, controllers need to have the word controller in them and should implement IController interface either directly or indirectly. Views should be placed in a specific location that MVC can find them.

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        ViewData["Countries"] = new List<string>()
        {
            "India",
            "US",
            "UK",
            "Canada"
        };

        return View();
    }
}
```

The following URL will invoke Index() action method with in the HomeController. Notice that our HomeController inherits from base Controller class which inturn inherits from ControllerBase class. ControllerBase inturn inherits from IController class.

http://localhost/MVCDemo/Home/Index

return View() statement with in the **HomeController** by default looks for a view with name = "Index" in "/Views/Home/" and "/Views/Shared/" folders. If a view with name = "Index" is not found, then, we get an error stating

The view 'Index' or its master was not found or no view engine supports the searched locations. The following locations were searched:

~/Views/Home/Index.aspx
 ~/Views/Home/Index.ascx
 ~/Views/Shared/Index.aspx
 ~/Views/Shared/Index.ascx
 ~/Views/Home/Index.cshtml
 ~/Views/Home/Index.vbhtml
 ~/Views/Shared/Index.cshtml
 ~/Views/Shared/Index.vbhtml

But with models, there are no strict rules. Infact **"Models"** folder is optional and they can live anywhere. They can even be present in a separate project.

Let's now turn our attention to using business objects as model. We will be using table **"tblEmployee"** for this demo. Use the sql script to create and populate this table.

Create table tblEmployee

```
(
  Id int Primary Key Identity(1,1),
  Name nvarchar(50),
  Gender nvarchar(10),
  City nvarchar(50),
  DateOfBirth DateTime
)
```

```
Insert into tblEmployee values('Mark','Male','London','01/05/1979')
Insert into tblEmployee values('John','Male','Chennai','03/07/1981')
Insert into tblEmployee values('Mary','Female','New York','02/04/1978')
Insert into tblEmployee values('Mike','Male','Sydeny','02/03/1974')
Insert into tblEmployee values('Scott','Male','London','04/06/1972')
```

Stored procedure to retrieve data

```
Create procedure spGetAllEmployees
as
Begin
Select Id, Name, Gender, City, DateOfBirth
from tblEmployee
End
```

Step 1: Create an ASP.NET MVC 4 Web application with name = **MVCDemo**

Step 2: Add a Class Library project with Name=**"BusinessLayer"**

Step 3: Right click on the **BusinessLayer** class library project, and add a class file with name = **Employee.cs**.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace BusinessLayer
{
    public class Employee
    {
        public int ID { get; set; }
        public string Name { get; set; }
        public string Gender { get; set; }
        public string City { get; set; }
        public DateTime DateOfBirth { get; set; }
    }
}
```

Step 4: Right click on the **"References"** folder of the **"BusinessLayer"** class library project, and

add a reference to **"System.Configuration"** assembly.

Step 5: Right click on the **BusinessLayer** class library project, and add a class file with name = **EmployeeBusinessLayer.cs**.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using System.Data.SqlClient;
using System.Configuration;

namespace BusinessLayer
{
    public class EmployeeBusinessLayer
    {
        public IEnumerable<Employee> Employees
        {
            get
            {
                string connectionString =
                    ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;

                List<Employee> employees = new List<Employee>();

                using (SqlConnection con = new SqlConnection(connectionString))
                {
                    SqlCommand cmd = new SqlCommand("spGetAllEmployees", con);
                    cmd.CommandType = CommandType.StoredProcedure;
                    con.Open();
                    SqlDataReader rdr = cmd.ExecuteReader();
                    while (rdr.Read())
                    {
                        Employee employee = new Employee();
                        employee.ID = Convert.ToInt32(rdr["Id"]);
                        employee.Name = rdr["Name"].ToString();
                        employee.Gender = rdr["Gender"].ToString();
                        employee.City = rdr["City"].ToString();
                        employee.DateOfBirth = Convert.ToDateTime(rdr["DateOfBirth"]);

                        employees.Add(employee);
                    }
                }

                return employees;
            }
        }
    }
}
```

Step 6: Right click on the **"References"** folder of the **"MVC Demo"** project, and add a reference to **"BusinessLayer"** project.

Step 7: Include a connection string with **name = "DBCS"** in Web.Config file

```
<add name="DBCS"
      connectionString="server=.; database=Sample; integrated security=SSPI"
      providerName="System.Data.SqlClient"/>
```

Step 8: Right click on the **"Controllers"** folder and add Controller with name = **"EmployeeController.cs"**.

```
public class EmployeeController : Controller
{
    public ActionResult Index()
    {
        EmployeeBusinessLayer employeeBusinessLayer =
            new EmployeeBusinessLayer();

        List<Employee> employees = employeeBusinessLayer.Employees.ToList();
        return View(employees);
    }
}
```



```
}
}
```

Step 9: Right click on the **Index()** action method in the "**EmployeeController**" class and select "**Add View**" from the context menu. Set
 View name = **Index**
 View engine = **Razor**
 Select "**Create a strongly-typed view**" checkbox
 Scaffold Template = **List**
 Click "**Add**" button

Run the application and navigate to <http://localhost/MVCDemo/Employee/Index>. The output should be as shown below.

Index

[Create New](#)

Name	Gender	City	DateOfBirth	Action
Mark	Male	London	05/01/1979 00:00:00	Edit Details Delete
John	Male	Chennai	07/03/1981 00:00:00	Edit Details Delete
Mary	Female	New York	04/02/1978 00:00:00	Edit Details Delete
Mike	Male	Sydeny	03/02/1974 00:00:00	Edit Details Delete
Scott	Male	London	06/04/1972 00:00:00	Edit Details Delete

Part 12 - Creating a view to insert data using mvc

Suggested Videos

[Part 9 - Generate hyperlinks using actionlink html helper](#)

[Part 10 - Working with multiple tables in MVC](#)

[Part 11 - Using business objects as model in mvc](#)

In this video we will discuss, creating a view to insert a new employee into the database table **tblEmployee**. Please [watch Part 11](#), before proceeding with this video.

We want to present the end user with a form as shown in the image below

Create

Employee

Name

Gender

Select Gender ▼

City

DateOfBirth

Create

[Back to List](#)

Copy and paste the following **"Create"** action method, in **EmployeeController** class.

```
[HttpGet]
public ActionResult Create()
{
    return View();
}
```

Please note that, the method is decorated with **"HttpGet"** attribute. This makes this action method to respond only to the **"GET"** request.

Now let's add a **"Create"** view. To do this, right click on the **"Create"** action method and select **"Add View"** from the context menu. Set

1. View name = **"Create"**
2. View engine = **"Razor"**
3. Select **"Create Strongly-typed view"** checkbox
4. Select **"Employee"** class, from **"Model class"** dropdownlist
5. Scaffold Template = **Create**
6. Click **"Add"** button

At this point **"Create.cshtml"** view will be added in **"Employee"** folder. If you have the following **"Scripts"** section at the bottom of the view, please delete it. We will discuss about sections and scripts in a later video session.

```
@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

Run the application and navigate to the following URL

"http://localhost/MVCDemo/Employee/Index"

Click on "Create New" link. You will be navigated to the following URL

"http://localhost/MVCDemo/Employee/Create"

A form with textboxes to add a new employee is rendered. For employee **"Gender"** it is ideal to have a dropdownlist instead of a text box. To achieve this, REPLACE THE FOLLOWING LINE

```
@Html.EditorFor(model => model.Gender)
```

WITH

```
@Html.DropDownList("Gender", new List<SelectListItem>
{
    new SelectListItem { Text = "Male", Value="Male" },
    new SelectListItem { Text = "Female", Value="Female" }
}, "Select Gender")
```

Run the application and notice that, a dropdownlist is now displayed for **"Gender"**.

If you click on **"Create"** button, you will get an error message stating - **The resource cannot be found**. This is because we don't have the **"Create"** controller action method that can handle **HTTPPost** request. We will discuss, fixing this in our next video.

Part 13 - FormCollection in mvc

Suggested Videos

[Part 10 - Working with multiple tables in MVC](#)

[Part 11 - Using business objects as model in mvc](#)

[Part 12 - Creating a view to insert data](#)

In this video we will discuss using **FormCollection** object in mvc and it's purpose. Please [watch Part 12](#), before proceeding with this video.

FormCollection class will automatically receive the posted form values in the controller action method, in **key/value** pairs. **Keys & values** can be accessed using key names or index.

We implemented the following **"Create"** view in [Part 12](#)

Create

Employee

Name
Venkat

Gender
Male

City
London

DateOfBirth
01/01/2010

Create

[Back to List](#)

We can use the **FormCollection** to loop thru each key and it's value that is posted to the server.

[HttpPost]

```
public ActionResult Create(FormCollection formCollection)
{
    if (ModelState.IsValid)
    {
        foreach (string key in formCollection.AllKeys)
        {
            Response.Write("Key = " + key + " ");
            Response.Write("Value = " + formCollection[key]);
            Response.Write("<br/>");
        }
    }
    return View();
}
```

The output is as shown below

Key = Name Value = Venkat
 Key = Gender Value = Male
 Key = City Value = London
 Key = DateOfBirth Value = 01/01/2010

Create the following stored procedure to insert employee data into tblEmployee table

```
Create procedure spAddEmployee
@Name nvarchar(50),
@Gender nvarchar (10),
@City nvarchar (50),
@DateOfBirth DateTime
as
Begin
Insert into tblEmployee (Name, Gender, City, DateOfBirth)
Values (@Name, @Gender, @City, @DateOfBirth)
End
```

Add the following method to EmployeeBusinessLayer.cs file.

```
public void AddEmployee(Employee employee)
{
    string connectionString =
        ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;

    using (SqlConnection con = new SqlConnection(connectionString))
    {
        SqlCommand cmd = new SqlCommand("spAddEmployee", con);
        cmd.CommandType = CommandType.StoredProcedure;

        SqlParameter paramName = new SqlParameter();
        paramName.ParameterName = "@Name";
        paramName.Value = employee.Name;
        cmd.Parameters.Add(paramName);

        SqlParameter paramGender = new SqlParameter();
        paramGender.ParameterName = "@Gender";
        paramGender.Value = employee.Gender;
        cmd.Parameters.Add(paramGender);

        SqlParameter paramCity = new SqlParameter();
        paramCity.ParameterName = "@City";
        paramCity.Value = employee.City;
        cmd.Parameters.Add(paramCity);

        SqlParameter paramDateOfBirth = new SqlParameter();
        paramDateOfBirth.ParameterName = "@DateOfBirth";
        paramDateOfBirth.Value = employee.DateOfBirth;
        cmd.Parameters.Add(paramDateOfBirth);

        con.Open();
        cmd.ExecuteNonQuery();
    }
}
```

To save form data, to a database table, copy and paste the following code in **EmployeeController.cs** file.

```
[HttpPost]
public ActionResult Create(FormCollection formCollection)
{
    Employee employee = new Employee();
    // Retrieve form data using form collection
    employee.Name = formCollection["Name"];
    employee.Gender = formCollection["Gender"];
    employee.City = formCollection["City"];
    employee.DateOfBirth =
        Convert.ToDateTime(formCollection["DateOfBirth"]);

    EmployeeBusinessLayer employeeBusinessLayer =
```

```

        new EmployeeBusinessLayer();

        employeeBusinessLayer.AddEmployee(employee);
        return RedirectToAction("Index");
    }

```

Do we really have to write all the dirty code of retrieving data from **FormCollection** and assign it to the properties of **"employee"** object. The answer is no. This is the job of the **model binder** in MVC. We will discuss model binders in our next video.

Part 14 - Mapping asp.net request data to controller action simple parameter types

Suggested Videos

[Part 11 - Using business objects as model in mvc](#)

[Part 12 - Creating a view to insert data](#)

[Part 13 - FormCollection in mvc](#)

In this video we will discuss mapping asp.net request data to controller action simple parameter types. [Please watch Part 13](#), before proceeding with this video.

To save form data, to a database table, we used the following code in [Part 13](#). From [Part 13](#), it should be clear that, **FormCollection** class will automatically receive the posted form values in the controller action method.

```

[HttpPost]
public ActionResult Create(FormCollection formCollection)
{
    Employee employee = new Employee();
    // Retrieve form data using form collection
    employee.Name = formCollection["Name"];
    employee.Gender = formCollection["Gender"];
    employee.City = formCollection["City"];
    employee.DateOfBirth =
        Convert.ToDateTime(formCollection["DateOfBirth"]);

    EmployeeBusinessLayer employeeBusinessLayer =
        new EmployeeBusinessLayer();

    employeeBusinessLayer.AddEmployee(employee);
    return RedirectToAction("Index");
}

```

The above controller **"Create"** action method can be re-written using simple types as shown below. Notice that, the create action method has got parameter names **that match** with the names of the form controls. To see the names of the form controls, right click on the browser and view page source. Model binder in mvc maps the values of these control, to the respective parameters.

```

[HttpPost]
public ActionResult Create(string name, string gender, string city, DateTime dateOfBirth)
{
    Employee employee = new Employee();
    employee.Name = name;
    employee.Gender = gender;
    employee.City = city;
    employee.DateOfBirth = dateOfBirth;

    EmployeeBusinessLayer employeeBusinessLayer =
        new EmployeeBusinessLayer();

    employeeBusinessLayer.AddEmployee(employee);
    return RedirectToAction("Index");
}

```

```
}
```

But do we really to do these mappings manually. The answer is no. In a later video session we will see how to automatically map the request data, without having to do it manually.

Please note that, **the order of the parameters does not matter**. What matters is the name of the parameter. If the parameter name is different from the form control name, then the form data will not be mapped as expected.

Part 15 - UpdateModel function in mvc

Suggested Videos

[Part 12 - Creating a view to insert data](#)

[Part 13 - FormCollection in mvc](#)

[Part 14 - Mapping asp.net request data to controller action simple parameter types](#)

In this video we will discuss using UpdateModel() function. [Please watch Part 14](#), before proceeding.

Change the implementation of **"Create"** action method that is decorated with **[HttpPost]** attribute in **"EmployeeController"** as shown below.

```
[HttpPost]
public ActionResult Create(Employee employee)
{
    if (ModelState.IsValid)
    {
        EmployeeBusinessLayer employeeBusinessLayer =
            new EmployeeBusinessLayer();

        employeeBusinessLayer.AddEmployee(employee);
        return RedirectToAction("Index");
    }
    return View();
}
```

Please note:

1. Model state is being checked using **IsValid** boolean property of **ModelState** object. We will discuss **ModelState** in a later video session.
2. Instead of passing the individual properties of **"Employee"** object as parameters to the **"Create"** action method, we are now passing the **"Employee"** object itself.
3. The **"Employee"** object is then handed over to **AddEmployee()** method of **"EmployeeBusinessLayer"** class, which takes the responsibility of saving the **"Employee"** object to the database table.
4. Upon saving the employee, the user is then redirected to the **"Index"** action method.
5. If there are any "Model" validation errors, **ModelState.IsValid** returns false. In this case, we stay on the same create view, which gives the opportunity to correct the errors and resubmit the page.

The above method can be rewritten as shown below.

```
[HttpPost]
public ActionResult Create()
{
    if (ModelState.IsValid)
    {
        EmployeeBusinessLayer employeeBusinessLayer =
            new EmployeeBusinessLayer();

        Employee employee = new Employee();
        UpdateModel<Employee>(employee);

        employeeBusinessLayer.AddEmployee(employee);
        return RedirectToAction("Index");
    }
}
```

```

    }
    return View();
}

```

When you make this change, you get a compilation error stating - **Type 'MVCDemo.Controllers.EmployeeController' already defines a member called 'Create' with the same parameter types**. Our intention here is to overload the **"Create"** controller action method based on **"HttpGet"** and **"HttpPost"**. To fix this error, use **"ActionName"** attribute as shown below.

```

[HttpGet]
[ActionName("Create")]
public ActionResult Create_Get()
{
    return View();
}

[HttpPost]
[ActionName("Create")]
public ActionResult Create_Post()
{
    if (ModelState.IsValid)
    {
        EmployeeBusinessLayer employeeBusinessLayer =
            new EmployeeBusinessLayer();

        Employee employee = new Employee();
        UpdateModel<Employee>(employee);

        employeeBusinessLayer.AddEmployee(employee);
        return RedirectToAction("Index");
    }
    return View();
}

```

Please Note:

1. We have changed the names of **"Create"** action methods to **"Create_Get"** and **"Create_Post"** depending on the actions they respond to.
2. **"ActionName"** is specified as **"Create"** for both of these methods. So, if a **"GET"** request is made to the **"URL - http://localhost/MVCDemo/Employee/Create"** then **"Create_Get()"** controller action method is invoked. On the other hand if a **"POST"** request is made to the same URL, then **"Create_Post()"** controller action method is invoked.
3. Instead of passing **"Employee"** object as a parameter to **"Create_Post()"** action method, we are creating an instance of an **"Employee"** object with in the function, and updating it using **"UpdateModel()"** function. **"UpdateModel()"** function inspects all the **HttpRequest** inputs such as posted Form data, QueryString, Cookies and Server variables and populate the employee object.

When you run the application, you may get an intermittent error stating - **Adding the specified count to the semaphore would cause it to exceed its maximum count**. To fix this error, either

1. Restart IIS
- OR
2. Disable connection pooling in the connection string of your web.config file

Part 16 - Difference between updatemodel and tryupdatemodel

Suggested Videos

[Part 13 - FormCollection in mvc](#)

[Part 14 - Mapping asp.net request data to controller action simple parameter types](#)

[Part 15 - Updatemodel function in MVC](#)

In this video we will discuss the differences between **updatemodel** and **tryupdatemodel** functions. [Please watch Part 15](#), before proceeding.

Make changes to **"Create_Post()"** controller action method as shown below.

```

[HttpPost]
[ActionName("Create")]

```



```

public ActionResult Create_Post()
{
    EmployeeBusinessLayer employeeBusinessLayer =
        new EmployeeBusinessLayer();

    Employee employee = new Employee();
    UpdateModel(employee);
    if (ModelState.IsValid)
    {
        employeeBusinessLayer.AddEmmployee(employee);
        return RedirectToAction("Index");
    }
    else
    {
        return View();
    }
}

```

Please note that, **"AddEmmployee()"** method is now inside the **"IF"** condition that checks the validity of ModelState using **ModelState.IsValid** boolean property. Run the application and navigate to the following URL.

http://localhost/MVCDemo/Employee/Create

Submit the page without entering any data. You will get an error stating - **"The model of type 'BusinessLayer.Employee' could not be updated"**. This is because **"DateOfBirth"** property of **"Employee"** class is a **non-nullable** DateTime data type. DateTime is a value type, and needs to have value when we post the form. To make **"DateOfBirth"** optional change the data type to nullable DateTime as shown below.

```

public class Employee
{
    public int ID { get; set; }
    public string Name { get; set; }
    public string Gender { get; set; }
    public string City { get; set; }
    public DateTime? DateOfBirth { get; set; }
}

```

Run the application and navigate to the following URL.

http://localhost/MVCDemo/Employee/Create

Submit the page without entering any data. You will now get a different error stating - **Procedure or function 'spAddEmployee' expects parameter '@Name', which was not supplied.**

This is because, the following parameters of stored procedure **"spAddEmployee"** are all required.

```

@Name
@Gender
@City
@DateOfBirth

```

To make all these parameters optional, modify the stored procedure as shown below.

```

Alter procedure spAddEmployee
@Name nvarchar(50) = null,
@Gender nvarchar(10) = null,
@City nvarchar (50) = null,
@DateOfBirth DateTime = null
as
Begin
    Insert into tblEmployee (Name, Gender, City, DateOfBirth)
    Values (@Name, @Gender, @City, @DateOfBirth)
End

```

Run the application and navigate to the following URL.

http://localhost/MVCDemo/Employee/Create

Submit the page without entering any data. You will now get a different error stating - **Object cannot be cast from DBNull to other types.**

To fix this error, make changes to **"Employees"** property in **"EmployeeBusinessLayer.cs"** file as shown below. Notice that we are populating **"DateOfBirth"** property of **"Employee"** object only

```

if "DateOfBirth" column value is not "DBNull".
public IEnumerable<Employee> Employees
{
    get
    {
        string connectionString =
            ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;

        List<Employee> employees = new List<Employee>();

        using (SqlConnection con = new SqlConnection(connectionString))
        {
            SqlCommand cmd = new SqlCommand("spGetAllEmployees", con);
            cmd.CommandType = CommandType.StoredProcedure;
            con.Open();
            SqlDataReader rdr = cmd.ExecuteReader();
            while (rdr.Read())
            {
                Employee employee = new Employee();
                employee.ID = Convert.ToInt32(rdr["Id"]);
                employee.Name = rdr["Name"].ToString();
                employee.Gender = rdr["Gender"].ToString();
                employee.City = rdr["City"].ToString();
                if (!(rdr["DateOfBirth"] is DBNull))
                {
                    employee.DateOfBirth = Convert.ToDateTime(rdr["DateOfBirth"]);
                }

                employees.Add(employee);
            }
        }

        return employees;
    }
}

```

Run the application and navigate to the following URL

<http://localhost/MVCDemo/Employee/Create>

Submit the page without entering any data. Notice that a blank employee row is inserted into tblEmployee table.

Now let's make the following properties of "Employee" class required.

Name
City
DateOfBirth

To achieve this we can use **"Required"** attribute that is present in **System.ComponentModel.DataAnnotations** namespace. To use this namespace, BusinessLayer project need a reference to **"EntityFramework"** assembly. The changes to the "Employee" class are shown below.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ComponentModel.DataAnnotations;

namespace BusinessLayer
{
    public class Employee
    {
        public int ID { get; set; }
        [Required]
        public string Name { get; set; }
        public string Gender { get; set; }
        [Required]
        public string City { get; set; }
        [Required]
        public DateTime? DateOfBirth { get; set; }
    }
}

```

```
}
}
```

Run the application and navigate to the following URL.

<http://localhost/MVCDemo/Employee/Create>

Submit the page without entering any data. We now get an error stating - **The model of type 'BusinessLayer.Employee' could not be updated.** Notice that this error is thrown when UpdateModel() function is invoked.

Now let's use TryUpdateModel() instead of UpdateModel(). Make changes to "Create_Post()" controller action method in "EmployeeController" as shown below.

```
[HttpPost]
[ActionName("Create")]
public ActionResult Create_Post()
{
    EmployeeBusinessLayer employeeBusinessLayer =
        new EmployeeBusinessLayer();

    Employee employee = new Employee();
    TryUpdateModel(employee);
    if (ModelState.IsValid)
    {
        employeeBusinessLayer.AddEmmployee(employee);
        return RedirectToAction("Index");
    }
    else
    {
        return View();
    }
}
```

Run the application and navigate to the following URL

<http://localhost/MVCDemo/Employee/Create>

Submit the page without entering any data. Notice that, we don't get an exception now and the user remains on "Create" view and the validation errors are displayed to the user.

So, the difference is UpdateModel() throws an exception if validation fails, where as TryUpdateModel() will never throw an exception. The similarity is, both the functions are used to update the Model with the Form values and perform the validations.

Is it mandatory to use "UpdateModel()" or "TryUpdateModel()" function to update the Model?

The answer is NO.

The above method can be re-written as shown below and we get the same behaviour.

```
[HttpPost]
[ActionName("Create")]
public ActionResult Create_Post(Employee employee)
{
    EmployeeBusinessLayer employeeBusinessLayer =
        new EmployeeBusinessLayer();

    if (ModelState.IsValid)
    {
        employeeBusinessLayer.AddEmmployee(employee);
        return RedirectToAction("Index");
    }
    else
    {
        return View();
    }
}
```

So the next question is, Why do we need to explicitly invoke model binding?

If you want to limit on what can be bound, explicitly invoking model binding can be very useful. We will discuss more about this in a later video session.

Part 17 - Editing a model in mvc

Suggested Videos

[Part 14 - Mapping asp.net request data to controller action simple parameter types](#)

[Part 15 - Updatemodel function in MVC](#)

[Part 16 - Difference between UpdateModel and TryUpdateModel](#)

In this video we will discuss editing a model in mvc. Please [watch Part 16](#), before proceeding.

Step 1: Copy and paste the following **"Edit"** controller action method in "EmployeeController.cs" file.

```
[HttpGet]
public ActionResult Edit(int id)
{
    EmployeeBusinessLayer employeeBusinessLayer =
        new EmployeeBusinessLayer();
    Employee employee =
        employeeBusinessLayer.Employees.Single(emp => emp.ID == id);

    return View(employee);
}
```

Please note:

1. This method is decorated with [HttpGet] attribute. So this method only responds to HTTP get request when editing data.
2. The **"Edit"** action method also receives "id" of the employee that is being edited. This "id" is used to retrieve the employee details.
3. The employee object is passed to the view

Step 2: Add "Edit" view

a) Right click on the "Edit" controller action method, and select "Add view" from the context menu

b) Set
 View name = Edit
 View engine = Razor
 Select "Create a strongly-typed view" check box
 Model class = "Employee"
 Scaffold template = "Edit"
 Finally click "Add" button

c) This should add "Edit.cshtml" to "Employee" folder in "Views" folder

d) Delete the following scripts section that is present at the bottom of "Edit.cshtml" view

```
@section Scripts
{
    @Scripts.Render("~/bundles/jqueryval")
}
```

Run the application and navigate to <http://localhost/MVCDemo/Employee/Index>. This page should list all the employees. Click on **"Edit"** link. The "Edit" page should display the details of the **"Employee"** being edited. Notice that, by default **"textboxes"** are used for editing. It is ideal to have a dropdownlist for gender rather than a textbox. To achieve this. make the following changes to **"Edit.cshtml"**

REPLACE THE FOLLOWING CODE

```
@Html.EditorFor(model => model.Gender)
@Html.ValidationMessageFor(model => model.Gender)
```

WITH

```
@Html.DropDownList("Gender", new List<SelectListItem>
{
    new SelectListItem { Text = "Male", Value="Male" },
    new SelectListItem { Text = "Female", Value="Female" }
}, "Select Gender")
@Html.ValidationMessageFor(model => model.Gender)
```

Run the application. Edit an employee, and notice that a DropDownList is used for gender as

expected. Post the form by clicking on "Save" button. We will get an error stating - **The resource cannot be found**. We will discuss fixing this in our next video.

Part 18 - Updating data in mvc

Suggested Videos

[Part 15 - UpdateModel function in MVC](#)

[Part 16 - Difference between UpdateModel and TryUpdateModel](#)

[Part 17 - Editing a model in mvc](#)

In this video we will discuss updating data in mvc. Please [watch Part 17](#), before proceeding.

Step 1: Create a stored procedure to update employee data.

Create procedure spSaveEmployee

```
@Id int,
@Name nvarchar(50),
@Gender nvarchar (10),
@City nvarchar (50),
@DateOfBirth DateTime
as
Begin
    Update tblEmployee Set
    Name = @Name,
    Gender = @Gender,
    City = @City,
    DateOfBirth = @DateOfBirth
    Where Id = @Id
End
```

Step 2: Add the following **"SaveEmployee()"** method to **"EmployeeBusinessLayer"** class in **"BusinessLayer"** project. This method is used to save employee data to the database table.

```
public void SaveEmployee(Employee employee)
{
    string connectionString =
        ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;

    using (SqlConnection con = new SqlConnection(connectionString))
    {
        SqlCommand cmd = new SqlCommand("spSaveEmployee", con);
        cmd.CommandType = CommandType.StoredProcedure;

        SqlParameter paramId = new SqlParameter();
        paramId.ParameterName = "@Id";
        paramId.Value = employee.ID;
        cmd.Parameters.Add(paramId);

        SqlParameter paramName = new SqlParameter();
        paramName.ParameterName = "@Name";
        paramName.Value = employee.Name;
        cmd.Parameters.Add(paramName);

        SqlParameter paramGender = new SqlParameter();
        paramGender.ParameterName = "@Gender";
        paramGender.Value = employee.Gender;
        cmd.Parameters.Add(paramGender);

        SqlParameter paramCity = new SqlParameter();
        paramCity.ParameterName = "@City";
        paramCity.Value = employee.City;
        cmd.Parameters.Add(paramCity);
    }
}
```

```

SqlParameter paramDateOfBirth = new SqlParameter();
paramDateOfBirth.ParameterName = "@DateOfBirth";
paramDateOfBirth.Value = employee.DateOfBirth;
cmd.Parameters.Add(paramDateOfBirth);

con.Open();
cmd.ExecuteNonQuery();
}
}

```

Step 3: Copy and paste the following **"Edit"** controller action method in **"EmployeeController.cs"** file.

```

[HttpPost]
public ActionResult Edit(Employee employee)
{
    if (ModelState.IsValid)
    {
        EmployeeBusinessLayer employeeBusinessLayer =
            new EmployeeBusinessLayer();
        employeeBusinessLayer.SaveEmployee(employee);

        return RedirectToAction("Index");
    }
    return View(employee);
}

```

Please note:

1. This method is decorated with `[HttpPost]` attribute. So this method only responds to HTTP post request when updating data.
2. The **"Edit"** action method receives the modified **"Employee"** object. This object is then passed to **SaveEmployee()** method, which saves the employee details. After the employee details are saved, the user is redirected to **"Index"** action.
3. If there are model validation errors, none of the code in the **IF** block gets executed. In this case, the user stays on the **"Edit"** view. Since we are passing **"Employee"** object to the **"Edit"** view, the user gets to see the validation errors. This allows him to fix those errors and re-submit the view.

Part 19 - Unintended updates in mvc

Suggested Videos

[Part 16 - Difference between UpdateModel and TryUpdateModel](#)

[Part 17 - Editing a model in mvc](#)

[Part 18 - Updating data in mvc](#)

In this video we will discuss, how, unintended updates can happen in mvc. Please [watch Part 18](#), before proceeding. Let's understand this with an example.

At the moment, **"Employee Edit"** view can be used to change all of the following fields.

1. Name
2. Gender
3. City
4. DateOfBirth

Let's make **"Name"** non-editable. To achieve this
CHANGE THE FOLLOWING CODE IN EDIT.CSHTML

```
@Html.EditorFor(model => model.Name)
```

TO

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

```
@Html.HiddenFor(model => model.Name)
```

Run the application and edit an employee. Notice that, **Name** of the employee is no longer rendered using a textbox. At this point we may think, that it is impossible for the user to change the name of the employee using **"Edit"** view. That is not true. Because of the way we have written our code, tools like Fiddler can be used to very easily change any properties of the **"Employee"** object.

Fiddler can be downloaded from the following URL

<http://fiddler2.com/get-fiddler>

After you have downloaded and installed fiddler, run fiddler, and navigate to the following URL <http://localhost/MVCDemo/Employee/Edit/1>

In fiddler, in **web sessions** window, select the url. Under the **"Inspectors"** tab you can see Request headers and response. We will discuss more about fiddler in a later video session.

Now click on **"Save"** button on **"Edit"** view. Notice that, under **"Web Sessions"** in fiddler, another request is captured for the same URL - <http://localhost/MVCDemo/Employee/Edit/1>

Now, without using the browser, let' us see how to generate a post request using fiddler.

1. Click on **"Composer"** tab in fiddler
2. Drag and drop the following URL from **"Web Sessions"** window onto Composer window. <http://localhost/MVCDemo/Employee/Edit/1>
3. In **"Reques Body"** under **"Composer"** tab, change **"Name"** of the employee to **"XYZ"**
4. Finally click **"Execute"** button on **"Fiddler"**

Now either query the database table, or navigate to **"Index"** view and notice that the employee name is changed to **"XYZ"**.

In our next video, we will discuss preventing these type of un-intended updates.

Part 20 - Preventing unintended updates in mvc

Suggested Videos

[Part 17 - Editing a model](#)

[Part 18 - Updating data](#)

[Part 19 - Unintended updates](#)

In this video we will discuss, preventing unintended updates in mvc. Please [watch Part 19](#), before proceeding.

Modify **"Edit"** controller action method that is decorated with `[HttpPost]` attribute as shown below. This method is present in **"EmployeeController.cs"** file.

```
[HttpPost]
[ActionName("Edit")]
public ActionResult Edit_Post(int id)
{
    EmployeeBusinessLayer employeeBusinessLayer = new EmployeeBusinessLayer();

    Employee employee = employeeBusinessLayer.Employees.Single(x => x.ID == id);
    UpdateModel(employee, new string[] { "ID", "Gender", "City", "DateOfBirth" });

    if (ModelState.IsValid)
    {
        employeeBusinessLayer.SaveEmployee(employee);

        return RedirectToAction("Index");
    }

    return View(employee);
}
```


Please note:

1. The name of the method is changed from **"Edit"** to **"Edit_Post"**
2. The method is decorated with `[ActionName("Edit")]` and `[HttpPost]` attributes. This indicates that, this method is going to respond to **"Edit"** action, when the form is posted to the server.
3. The **"id"** of the employee that is being edited, is passed as a parameter to this method.
4. Using the **"id"** parameter we load the employee details(Id, Name, Gender, City & DateOfBirth) from the database.
`Employee employee = employeeBusinessLayer.Employees.Single(x => x.ID == id);`
5. We then call **UpdateModel()** function. This should automatically update **"Employee"** object with data from the posted form. We are also passing a string array as the second parameter. This parameter specifies the list of model properties to update. This is also called as **include list** or **white list**. Notice that, we did not include **"Name"** property in the list. This means, even if the posted form data contains value for **"Name"** property, it will not be used to update the **"Name"** property of the **"Employee"** object.
`UpdateModel(employee, new string[] { "ID", "Gender", "City", "DateOfBirth" });`

So, if we were to generate a post request using fiddler as we did in the previous session, **"Name"** property of the **"Employee"** object will not be updated.

Alternatively, to exclude properties from binding, we can specify the exclude list as shown below.

```
[HttpPost]
[ActionName("Edit")]
public ActionResult Edit_Post(int id)
{
    EmployeeBusinessLayer employeeBusinessLayer = new EmployeeBusinessLayer();

    Employee employee = employeeBusinessLayer.Employees.Single(x => x.ID == id);
    UpdateModel(employee, null, null, new string[] { "Name" });

    if (ModelState.IsValid)
    {
        employeeBusinessLayer.SaveEmployee(employee);

        return RedirectToAction("Index");
    }

    return View(employee);
}
```

Notice that we are using a different overloaded version of **UpdateModel()** function. We are passing **"NULL"** for **"prefix"** and **"includeProperties"** parameters.

`UpdateModel<TModel>(TModel model, string prefix, string[] includeProperties, string[] excludeProperties)`

Part 21 - Including and excluding properties from model binding using bind attribute

Suggested Videos

[Part 18 - Updating data](#)

[Part 19 - Unintended updates](#)

[Part 20 - Preventing unintended updates](#)

In this video we will discuss, **including and excluding properties from model binding** using **BIND** attribute. Please [watch Part 20](#), before proceeding.

In [part 20](#), we have seen how to include and exclude properties from model binding, by passing a string array to **UpdateModel()** method. There is another way to do the same thing using **"Bind"** attribute.

Modify **"Edit_Post()"** controller action method that is present in **"EmployeeController.cs"** file, as shown below.

```
[HttpPost]
```

```
[ActionName("Edit")]
public ActionResult Edit_Post([Bind(Include = "Id, Gender, City, DateOfBirth")] Employee employee)
{
    EmployeeBusinessLayer employeeBusinessLayer = new EmployeeBusinessLayer();
    employee.Name = employeeBusinessLayer.Employees.Single(x => x.ID == employee.ID).Name;

    if (ModelState.IsValid)
    {
        employeeBusinessLayer.SaveEmployee(employee);

        return RedirectToAction("Index");
    }

    return View(employee);
}
```

Notice that, we are using **"BIND"** attribute and specifying the properties that we want to include in model binding. Since, **"Name"** property is not specified in the **INCLUDE** list, it will be excluded from model binding.

```
public ActionResult Edit_Post([Bind(Include = "Id, Gender, City, DateOfBirth")] Employee employee)
```

At this point, run the application and navigate to ["http://localhost/MVCDemo/Employee/Edit/1"](http://localhost/MVCDemo/Employee/Edit/1). Click **"Save"** button, you will get a **"Model"** validation error stating - **"The Name field is required"**.

This is because, we marked **"Name"** property in **"Employee"** class with **"Required"** attribute. Remove the **"Required"** attribute from **"Name"** property.

```
public class Employee
{
    public int ID { get; set; }
    public string Name { get; set; }
    [Required]
    public string Gender { get; set; }
    [Required]
    public string City { get; set; }
    [Required]
    public DateTime? DateOfBirth { get; set; }
}
```

So, if we were to generate a post request using fiddler as we did in the previous session, **"Name"** property of the **"Employee"** object will not be updated.

Alternatively, to **exclude** properties from binding, we can specify the **exclude list** using **"Bind"** attribute as shown below.

```
[HttpPost]
[ActionName("Edit")]
public ActionResult Edit_Post([Bind(Exclude = "Name")] Employee employee)
{
    // Rest of the method implementation remains the same
}
```

Part 22 - Including and excluding properties from model binding using interfaces

Suggested Videos

[Part 19 - Unintended updates](#)

[Part 20 - Preventing unintended updates](#)

[Part 21 - Including and excluding properties from model binding using bind attribute](#)

In this video we will discuss, **including and excluding properties from model binding using interfaces**. Please watch [Part 21](#), before proceeding.

In [part 20](#), we have seen how to include and exclude properties from model binding, by passing a string array to **UpdateModel()** method, and in [part 21](#) we have seen achieving the same using **"BIND"** attribute.

To include and exclude properties from model binding using interfaces

Step 1: Create an interface "IEmployee" as shown below. Notice that this interface, has got only the properties that we want to include in model binding. "Name" property is not present. This means, "Name" property will be excluded from model binding. Copy and paste this code in "Employee.cs" class file in "BusinessLayer" project

```
public interface IEmployee
{
    int ID { get; set; }
    string Gender { get; set; }
    string City { get; set; }
    DateTime? DateOfBirth { get; set; }
}
```

Step 2: Make "Employee" class inherit from IEmployee interface

```
public class Employee : IEmployee
{
    public int ID { get; set; }
    public string Name { get; set; }
    [Required]
    public string Gender { get; set; }
    [Required]
    public string City { get; set; }
    [Required]
    public DateTime? DateOfBirth { get; set; }
}
```

Step 3: Modify "Edit_Post()" controller action method that is present in "EmployeeController.cs" file, as shown below.

```
[HttpPost]
[ActionName("Edit")]
public ActionResult Edit_Post(int id)
{
    EmployeeBusinessLayer employeeBusinessLayer = new EmployeeBusinessLayer();
    Employee employee = employeeBusinessLayer.Employees.Single(x => x.ID == id);
    UpdateModel<IEmployee>(employee);

    if (ModelState.IsValid)
    {
        employeeBusinessLayer.SaveEmployee(employee);
        return RedirectToAction("Index");
    }

    return View(employee);
}
```

Notice that we are explicitly calling the model binder, by calling **UpdateModel()** function passing in our interface **IEmployee**. The model binder will update only the properties that are present in the interface.

So, if we were to generate a post request using fiddler as we did in the previous session, "Name" property of the "Employee" object will not be updated.

So, in short, there are several ways to **include** and **exclude** properties from **Model Binding**. Depending on the architecture and requirements of your project, you may choose the approach that best fit your needs.

Part 23 - Why deleting database records using get request is bad

Suggested Videos

[Part 20 - Preventing unintended updates](#)

[Part 21 - Including and excluding properties from model binding using bind attribute](#)

[Part 22 - Including and excluding properties from model binding using interfaces](#)

In this video we will discuss, why deleting database records using **GET** request is bad. [Please watch Part 22](#), before proceeding.

First let's discuss, how to delete data in MVC using GET request and then we will discuss, why it is bad to do so.

Step 1: Create a stored procedure to delete employee data by "ID"

Create procedure spDeleteEmployee

```
@Id int
as
Begin
Delete from tblEmployee
where Id = @Id
End
```

Step 2: Add the following **DeleteEmployee()** method to "**EmployeeBusinessLayer.cs**" file in "**BusinessLayer**" project. This method calls the stored procedure "**spDeleteEmployee**" that we just created.

```
public void DeleteEmployee(int id)
{
    string connectionString =
        ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;

    using (SqlConnection con = new SqlConnection(connectionString))
    {
        SqlCommand cmd = new SqlCommand("spDeleteEmployee", con);
        cmd.CommandType = CommandType.StoredProcedure;

        SqlParameter paramId = new SqlParameter();
        paramId.ParameterName = "@Id";
        paramId.Value = id;
        cmd.Parameters.Add(paramId);

        con.Open();
        cmd.ExecuteNonQuery();
    }
}
```

Step 3: Add the following "**DELETE**" controller action method to "**EmployeeController**".

```
public ActionResult Delete(int id)
{
    EmployeeBusinessLayer employeeBusinessLayer =
        new EmployeeBusinessLayer();
    employeeBusinessLayer.DeleteEmployee(id);
    return RedirectToAction("Index");
}
```

Run the application and navigate to "**Index**" action. Click the "**Delete**" link. This issues "**GET**" request to the following URL, and deletes the record.

<http://localhost/MVCDemo/Employee/Delete/1>

Deleting database records using GET request opens a security hole and is not recommended by Microsoft. Just imagine what can happen if there is an image tag in a malicious email as shown below. The moment we open the email, the image tries to load and issues a GET request, which would delete the data.

Also, when search engines index your page, they issue a **GET** request which would delete the data. In general **GET** request should be free of any side-effects, meaning it should not change the state.

Deletes should always be performed using a **POST** request. We will discuss, implementing this in our next video.

Part 24 - Deleting database records using post request in mvc

Suggested Videos

[Part 21 - Including and excluding properties from model binding using bind attribute](#)

[Part 22 - Including and excluding properties from model binding using interfaces](#)

[Part 23 - Why deleting database records using get request is bad](#)

In this video we will discuss

1. Deleting database records using **POST** request
2. Showing the **client side javascript confirmation** dialog box before deleting

[Please watch Part 23](#), before proceeding.

Step 1: Mark **"Delete"** action method in **"Employee"** controller with [HttpPost] attribute. With this change, the **"Delete"** method will no longer respond to **"GET"** request. At this point, if we run the application and click on **"Delete"** link on the **"Index"** view, we get an error stating - **"The resource cannot be found"**

```
[HttpPost]
public ActionResult Delete(int id)
{
    EmployeeBusinessLayer employeeBusinessLayer =
        new EmployeeBusinessLayer();
    employeeBusinessLayer.DeleteEmployee(id);
    return RedirectToAction("Index");
}
```

Step 2: In "Index.cshtml"

REPLACE THE FOLLOWING CODE

```
@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Name)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Gender)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.City)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.DateOfBirth)
        </td>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.ID }) |
            @Html.ActionLink("Details", "Details", new { id=item.ID }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.ID })
        </td>
    </tr>
}
```

WITH

```
@foreach (var item in Model)
{
    using (Html.BeginForm("Delete", "Employee", new { id = item.ID }))
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Name)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Gender)
            </td>
```

```

        <td>
            @Html.DisplayFor(modelItem => item.City)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.DateOfBirth)
        </td>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id = item.ID }) |
            <input type="submit" value="Delete" />
        </td>
    </tr>
}
}

```

Notice that, we are using "**Html.BeginForm()**" html helper to generate a form tag.

Step 3: To include client-side confirmation, before the data can be deleted, add the "**onclick**" attribute to "**Delete**" button as shown below.

```

<input type="submit" value="Delete" onclick="return confirm('Are you sure you want to delete record with ID = @item.ID');" />

```

art 25 - Insert update delete in mvc using entity framework

Suggested Videos

[Part 22 - Including and excluding properties from model binding using interfaces](#)

[Part 23 - Why deleting database records using get request is bad](#)

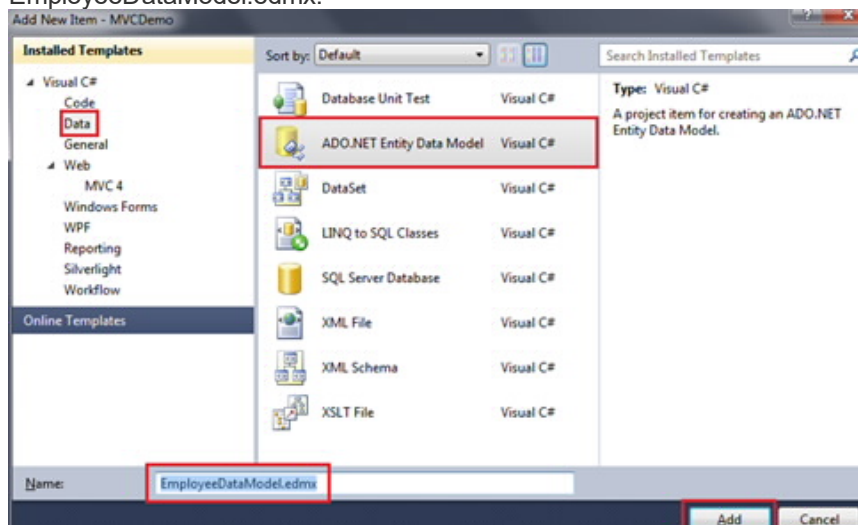
[Part 24 - Deleting database records using post request in mvc](#)

In this video we will discuss, **selecting, inserting, updating and deleting data** in mvc using entity framework. [Please watch Part 24](#), before proceeding.

We will be using tables **tblDepartment** and **tblEmployee** for this demo. You can get the sql script to create and populate these tables from [Part 10](#) of this video series.

Step 1: Create a new asp.net mvc 4 web application.

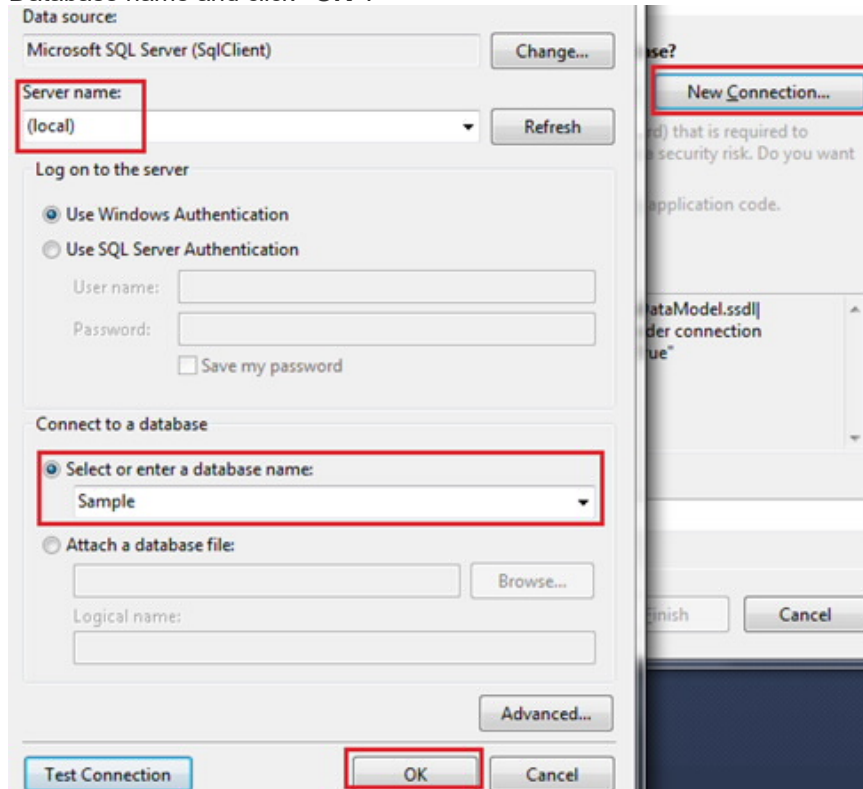
Step 2: Right click on the "**Models**" folder and add "**ADO.NET Entity Data Model**". Set Name = EmployeeDataModel.edmx.



On the subsequent screen, select "**Generate from database**" option and click "**Next**".

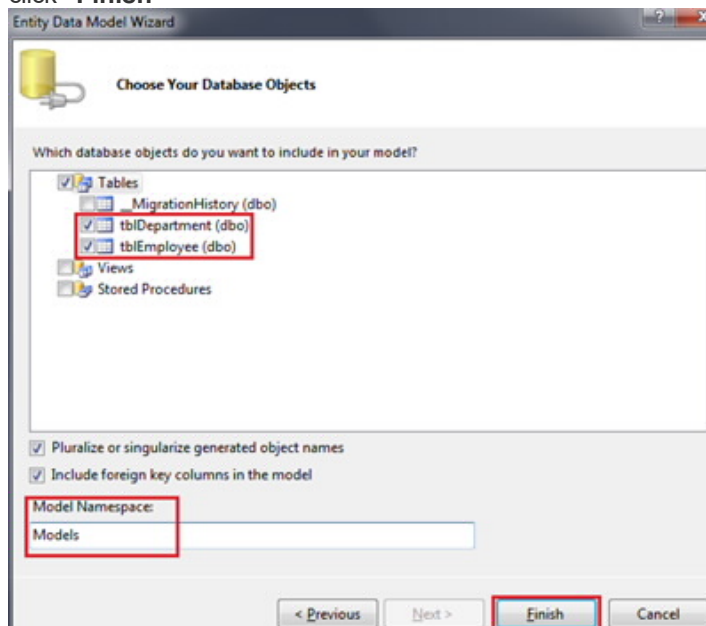
On "**Choose your data connection screen**", click on "**New Connection**" button.

Specify the sql server name. In my case, I have sql server installed on my local machine. So I have set "**Server Name=(local)**". From "**Select or enter a database name**"dropdownlist, select the Database name and click "**OK**".



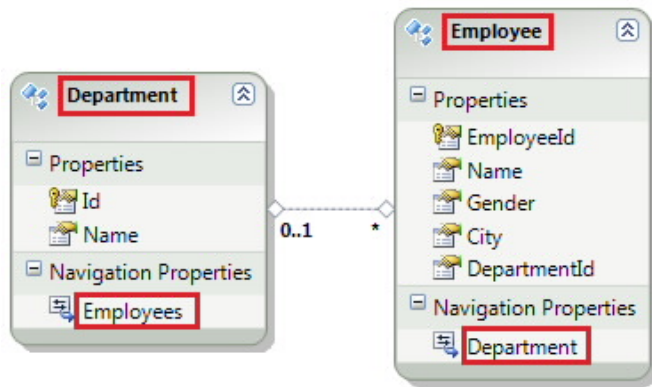
Click "**Next**".

On "**Choose your database objects**" screen, expand "**Tables**" and select "**tblDepartment**" and "**tblEmployee**" tables. Set "**Model Namespace=Models**" and click "**Finish**".



At this point we should have **tblDepartment** and **tblEmployee** entities generated.

- Change **tblDepartment** to **Department**
- Change **tblEmployee** to **Employee**
- Change **tblEmployees** navigation property to **Employees**
- Change **tblDepartment** navigation property to **Department**



Build the solution.

Step 3: Right click on the **"Controllers"** folder and select Add - Controller. Set
 Name = EmployeeController
 Template = MVC controller with read/write actions and views, using Entity Framework
 Model class = Employee(MVCDemo.Models)
 Data Context Class = EmployeeContext(MVCDemo.Models)
 Views = Razor

Finally click **"Add"**.

At this point you should have the following files automatically added.

1. EmployeeController.cs file in "Controllers" folder
2. Index, Create, Edit, Detail and Delete views in "Employee" folder.

On Create and Edit views, please delete the following scripts section. We will discuss these in a later video session.

```

@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
  
```

At this point, if you run the application by pressing CTRL + F5, you will get an error stating - **The resource cannot be found**. This is because, by default, the application goes to "HOME" controller and "Index" action.

To fix this,

1. Open **"RouteConfig.cs"** file from **"App_Start"** folder
2. Set Controller = **"Employee"**

Run the application again. Notice that, all the employees are listed on the index view. We can also create a new employee, edit an employee, view their full details and delete an employee as well. However, there are few issues, with each of the views, which we will address in our upcoming videos.

Part 26 - Customizing the autogenerated index view

Suggested Videos

- [Part 23 - Why deleting database records using get request is bad](#)
- [Part 24 - Deleting database records using post request in mvc](#)
- [Part 25 - Insert update delete in mvc using entity framework](#)

In this video we will discuss, customizing the **auto-generated** index view. Please [watch Part 25](#), before proceeding.

At the moment, the **"Index"** view is using **"Name"** as the column header for both employee name and department name. This is because **"Name"** column is used in both the database tables (tblEmployee & tblDepartment) and entity framework has used these column names to

generate **"Name"** property in Employee and Department classes that are auto-generated.

I want to change the **depratment** column header to **"Department Name"** instead of just **"Name"**. To achieve this, add a class file with **"name=Department.cs"** to **"Models"** folder.

Copy and paste the following code in "Department.cs" file

```
[MetadataType(typeof(DepartmentMetaData))]
public partial class Department
{
}

public class DepartmentMetaData
{
    [Display(Name="Department Name")]
    public string Name { get; set; }
}
```

With these changes run the application and notice the column name is displayed as **Department Name**. This is achieved by using **"Display"** attribute that is present in **"System.ComponentModel.DataAnnotations"** namespace.

If you are wondering why can't we apply **"Display"** attribute directly to the auto-generated **"Department"** class instead of creating another partial **"Department"** and **DepartmentMetaData** class. We can do it. There is nothing stopping us from doing it, but every time the Department class is auto-generated, our custom changes will be lost. This is the reason for creating another partial class, and applying our changes.

Part 27 - Customizing the autogenerated create view

Suggested Videos

[Part 24 - Deleting database records using post request in mvc](#)

[Part 25 - Insert update delete in mvc using entity framework](#)

[Part 26 - Customizing the autogenerated index view](#)

In this video we will discuss, customizing the auto-generated create view. Please [watch Part 26](#), before proceeding.

At the moment, none of the fields on **"Create"** view are required. This means, when you click on the **"Create"** button without filling any data, **NULL values** are stored in all the columns of **tblEmployee** table.

So, how to make these fields on the "Create" view required?

Add [\[Required\]](#) attribute to the **"Employee"** class. The **"Employee"** class that is present in **"EmployeeDataModel.Designer.cs"** is auto-generated by the **entity framework**. There is no point in adding the [\[Required\]](#) attribute to this class, as we will lose the changes if the class is auto-generated again.

To achieve this, add a class file with **"name=Employee.cs"** to **"Models"** folder.

Copy and paste the following code in "Employee.cs" file

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.ComponentModel.DataAnnotations;

namespace MVCDemo.Models
```

```

{
    [MetadataType(typeof(EmployeeMetaData))]
    public partial class Employee
    {
    }

    public class EmployeeMetaData
    {
        [Required]
        public string Name { get; set; }

        [Required]
        public string Gender { get; set; }

        [Required]
        public string City { get; set; }

        [Required]
        [Display(Name="Department")]
        public int DepartmentId { get; set; }
    }
}

```

At this point, run the application and click on the **"Create"** button without filling any data. Notice that we get validation error messages as expected. In a later video session, we will discuss changing the colour of the validation messages.

If you want **"Select Department"** as the first item in the **"Department"** dropdownlist on **"Create"** view, then,

REPLACE THE FOLLOWING LINE

```
@Html.DropDownList("DepartmentId", String.Empty)
```

WITH

```
@Html.DropDownList("DepartmentId", "Select Department")
```

Notice that, a textbox is used for gender. It is ideal to have a dropdownlist for gender rather than a textbox. To achieve this, make the following changes to **"Create.cshtml"** view.

REPLACE THE FOLLOWING CODE

```
@Html.EditorFor(model => model.Gender)
```

WITH

```

@Html.DropDownList("Gender", new List<SelectListItem>
{
    new SelectListItem { Text = "Male", Value="Male" },
    new SelectListItem { Text = "Female", Value="Female" }
}, "Select Gender")

```

Part 28 - Customizing the autogenerated edit view

Suggested Videos

[Part 25 - Insert update delete in mvc using entity framework](#)

[Part 26 - Customizing the autogenerated index view](#)

[Part 27 - Customizing the autogenerated create view](#)

In this video we will discuss, customizing the auto-generated edit view. Please [watch Part 27](#), before proceeding.

If you want **"Select Department"** as the first item in the **"Department"** dropdownlist on **"Edit"** view, then,

REPLACE THE FOLLOWING LINE

```
@Html.DropDownList("DepartmentId", String.Empty)
```

WITH

```
@Html.DropDownList("DepartmentId", "Select Department")
```

Notice that, a textbox is used for gender. It is ideal to have a dropdownlist for gender rather than a textbox. To achieve this, make the following changes to **"Edit.cshtml"** view.

REPLACE THE FOLLOWING CODE

```
@Html.EditorFor(model => model.Gender)
```

WITH

```
@Html.DropDownList("Gender", new List<SelectListItem>
{
    new SelectListItem { Text = "Male", Value="Male" },
    new SelectListItem { Text = "Female", Value="Female" }
}, "Select Gender")
```

Let's make **"Name"** non-editable. To achieve this

CHANGE THE FOLLOWING CODE IN EDIT.CSHTML

```
@Html.EditorFor(model => model.Name)
```

TO

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

At this point, we will still be able to change **"Name"** property of the employee, using tools like fiddler. We discussed this in [Part 19](#) of this [video series](#). There are several ways to prevent **"Name"** property from being updated.

1. Use **UpdateModel()** function and pass include and exclude list as a parameter - Discussed in [Part 20](#)
2. Use Bind attribute - Discussed in [Part 21](#)
3. Use interfaces - Discussed in [Part 22](#)

Now, let's discuss using **BIND** attribute to prevent updating **"Name"** property using tools like fiddler. Along the way, I will demonstrate adding model validation errors dynamically.

Change the implementation of **"Edit"** controller action method, that responds to **[HttpPost]** request as shown below

```
[HttpPost]
public ActionResult Edit([Bind(Exclude="Name")] Employee employee)
{
    Employee employeeFromDB = db.Employees.Single(x => x.EmployeeId ==
employee.EmployeeId);

    employeeFromDB.EmployeeId = employee.EmployeeId;
    employeeFromDB.Gender = employee.Gender;
    employeeFromDB.City = employee.City;
    employeeFromDB.DepartmentId = employee.DepartmentId;
    employee.Name = employeeFromDB.Name;

    if (ModelState.IsValid)
    {
        db.ObjectStateManager.ChangeObjectState(employeeFromDB, EntityState.Modified);
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    ViewBag.DepartmentId = new SelectList(db.Departments, "Id", "Name",
employee.DepartmentId);
    return View(employee);
}
```

Please note that, we have excluded **"Name"** property from model binding using **"Bind"** attribute. Even without **BIND** attribute, users will not be able to change the **"NAME"** of the employee, as we are copying only the required properties(Excluding NAME property) from **"employee"** object to **"employeeFromDB"** which in turn is persisted to the database. Since, I want to demonstrate adding model validation errors dynamically, let the attribute be there.

At this point if we run the application and click on "Save" button on "Edit" view, we get a validation

error stating - The Name field is required. This is because "Name" property is decorated with [Required] attribute in "Employee.cs" file. To prevent the validation error, remove the [Required] attribute.

The problem with this change is that, **"Name"** field on **"Create"** view is no longer mandatory. This means we will now be able to create a new employee, without **NAME**. To fix the **"Create"** view, let's add model validation errors dynamically. Change the implementation of **"Create"** controller action method that responds to [HttpPost] request as shown below.

[HttpPost]

```
public ActionResult Create(Employee employee)
{
    if (string.IsNullOrEmpty(employee.Name))
    {
        ModelState.AddModelError("Name", "The Name field is required.");
    }

    if (ModelState.IsValid)
    {
        db.Employees.AddObject(employee);
        db.SaveChanges();
        return RedirectToAction("Index");
    }

    ViewBag.DepartmentId = new SelectList(db.Departments, "Id", "Name",
employee.DepartmentId);
    return View(employee);
}
```

Part 29 - Using data transfer object as the model in mvc

Suggested Videos

[Part 26 - Customizing the auto-generated index view](#)

[Part 27 - Customizing the auto-generated create view](#)

[Part 28 - Customizing the auto-generated edit view](#)

In this video we will discuss, using data transfer object as the model in mvc. Please [watch Part 28](#), before proceeding.

Let's say the business requirement is such that, we want to display **total number of employees by department** as shown below. At the moment, either the **Employee** or **Department** class does not have Total property. This is one example, where a **Data Transfer Object** can be used as a model.

Employees By Department

[Create New](#)

Name	Total
HR	4
IT	3
Payroll	2

Right click on the **"Models"** folder and add a class with name=**"DepartmentTotals.cs"**. Copy and paste the following code.

```
public class DepartmentTotals
{
    public string Name { get; set; }
    public int Total { get; set; }
}
```

Now add the following **"EmployeesByDepartment"** controller action method to **EmployeeController** class.

```
public ActionResult EmployeesByDepartment()
{
    var departmentTotals = db.Employees.Include("Department")
        .GroupBy(x => x.Department.Name)
        .Select(y => new DepartmentTotals
        {
            Name = y.Key, Total = y.Count()
        }).ToList();
    return View(departmentTotals);
}
```

At this point, build the solution, so that the newly added **DepartmentTotals** class is compiled.

Now right click on **"EmployeesByDepartment"** action method in **"EmployeeController"** and select **"Add View"** from the context menu.

View name = EmployeesByDepartment

View engine = Razor

Select **"Create a strongly-typed view"** checkbox

Model class = DepartmentTotals

Model class = DepartmentTotals

To list the employees in **ascending** order of total employee, use **OrderBy()** LINQ method as shown below.

```
var departmentTotals = db.Employees.Include("Department")
    .GroupBy(x => x.Department.Name)
    .Select(y => new DepartmentTotals
    {
        Name = y.Key, Total = y.Count()
    }).ToList().OrderBy(y => y.Total);
```

To sort the list in **descending** order use, **OrderByDescending()** LINQ method.

```
var departmentTotals = db.Employees.Include("Department")
    .GroupBy(x => x.Department.Name)
    .Select(y => new DepartmentTotals
    {
        Name = y.Key, Total = y.Count()
    }).ToList().OrderByDescending(y => y.Total);
return View(departmentTotals);
```

Part 30 - View engines in asp.net mvc

Suggested Videos

[Part 27 - Customizing the auto-generated create view](#)

[Part 28 - Customizing the auto-generated edit view](#)

[Part 29 - Using data transfer object as the model in mvc](#)

In this video we will discuss, different view engines that are available in asp.net mvc. [Please watch Part 29](#), before proceeding.

Out of the box asp.net offers the following 2 view engines.

1. ASPX
2. Razor

There are **3 possible interview questions** here, to test your understanding of view engines.

1. What is the difference between RAZOR and ASPX view engines?

It mostly, boils down to the syntax. Otherwise there are no major differences between the two. In ASPX view engine, the server side script is wrapped between `<% %>`, where as in RAZOR we use `@`. Personally, I prefer using RAZOR views, as it is very easy to switch between HTML and Code.

Depending on the programming language you have chosen, RAZOR views have the extension of **.CSHTML** or **.VBHTML**, where as ASPX views has the extension of **.ASPX**

2. Is it possible, to have both RAZOR and ASPX views in one application?

Yes, when you right click on any controller action method, and select **"Add View"** from the context menu, you will have the option to choose the view engine of your choice from the "Add View" dialog box.

3. Is it possible, to use a third party view engine with asp.net mvc?

ASP.NET MVC is designed with **extensibility** in mind. So, it's very easy to include third party view engine as well. We will discuss this in detail in our next video.

Part 31 - Using custom view engines with asp.net mvc

Suggested Videos

[Part 28 - Customizing the auto-generated edit view](#)

[Part 29 - Using data transfer object as the model](#)

[Part 30 - View engines](#)

Out of the box ASP.NET MVC supports the following 2 view engines

1. ASPX
2. Razor

In addition to the above 2 view engines, there are several **custom view engines** that can be used with asp.net mvc. The following are a few of these custom view engines.

1. Spark
2. NHaml
3. SharpDOM
4. Brail etc....

For example, if you want to use Spark as the view engine for your asp.net mvc 4 project, then install **Spark.Web.Mvc4** using **NuGet Package Manager**.

At this point, right click on any controller action method and select **"Add View"** from the context menu. Notice that, **"View Engine"** dropdownlist in **"Add View"** dialog box only shows - Razor and ASPX view engines. If you want the **"Spark"** view engine to be listed, then, we need to register **"Spark View Engine"** templates.

Step 1: Navigate to the following path and create a folder with name=**"Spark"**. Spark specific templates must be copied into "Spark" folder.

C:\Program Files (x86)\Microsoft Visual Studio

10.0\Common7\IDE\ItemTemplates\CSharp\Web\MVC 4\CodeTemplates\AddView

Step 2: Create an xml file with in **"Spark"** folder. The name of the XML file must be **"ViewEngine.xml"** and must contain the following xml content.

```
<?xml version="1.0" encoding="utf-8" ?>
<ViewEngine DisplayName="Spark"
  ViewFileExtension=".spark"
  DefaultLayoutPage="~/Views/Shared/_Layout.spark"
  PartialViewFileExtension=".spark" />
```

Now, right click on any controller action method and select "Add View" from the context menu. Notice that, "Spark" view engine is also listed.

Part 32 - How does a controller find a view in mvc

Suggested Videos

[Part 29 - Using data transfer object as the model](#)

[Part 30 - View engines](#)

[Part 31 - Using custom view engines](#)

In this video, we will discuss the convention used by mvc to find views. Let's understand this with an example.

Notice that, the **Index()** action method does not specify the name of the view. So,

1. How does asp.net mvc know, which view to use

2. What locations does asp.net mvc search

```
public class EmployeeController : Controller
{
    private EmployeeContext db = new EmployeeContext();

    public ActionResult Index()
    {
        var employees = db.Employees.Include("Department");
        return View(employees.ToList());
    }

    protected override void Dispose(bool disposing)
    {
        db.Dispose();
        base.Dispose(disposing);
    }
}
```

To find the answer for the above 2 questions, delete the **"Index.cshtml"** view from **"Views/Employee"** folder. Run the application, and notice that we get an error message stating

The view 'Index' or its master was not found or no view engine supports the searched locations. The following locations were searched:

```
~/Views/Employee/Index.aspx
~/Views/Employee/Index.ascx
~/Views/Shared/Index.aspx
~/Views/Shared/Index.ascx
~/Views/Employee/Index.cshtml
~/Views/Employee/Index.vbhtml
~/Views/Shared/Index.cshtml
~/Views/Shared/Index.vbhtml
```

So, from the error message, it should be clear that, **MVC looks for a view with the same name as that of the controller action method** in the following locations

1. Views/Shared
2. Views/FolderNameMatchingControllerName

Please note that, the view extension can be any of the following

- a).cshtml
- b).vbhtml
- c).aspx
- d).ascx

If I have all of the following files in "Views/Employee" folder, then MVC picks up "Index.aspx"

- a) Index.aspx
- b) Index.cshtml
- c) Index.vbhtml

d) Index.ascx

If you want to use "Index.cshtml" instead, then specify the full path as shown below.

```
public ActionResult Index()
{
    var employees = db.Employees.Include("Department");
    return View("~/Views/Employee/Index.cshtml", employees.ToList());
}
```

If you specify only the name of the view along with it's extension as shown below, you will get an error.

```
return View("Index.cshtml", employees.ToList());
```

If you want to use a view name which is not inside the views folder of the current controller, then specify the full path as shown below.

```
public ActionResult Index()
{
    var employees = db.Employees.Include("Department");
    return View("~/Views/Home/Index.aspx", employees.ToList());
}
```

Please note that asp.net mvc

1. Is all about convention over configuration
2. Views folder contains one folder for each controller and a "Shared" folder
3. Shared folder is used to store views shared between controllers. Master and Layout pages are stored in "Shared" folder.

Part 32 - How does a controller find a view in mvc

Suggested Videos

[Part 29 - Using data transfer object as the model](#)

[Part 30 - View engines](#)

[Part 31 - Using custom view engines](#)

In this video, we will discuss the convention used by mvc to find views. Let's understand this with an example.

Notice that, the **Index()** action method does not specify the name of the view. So,

1. How does asp.net mvc know, which view to use
2. What locations does asp.net mvc search

```
public class EmployeeController : Controller
{
    private EmployeeContext db = new EmployeeContext();

    public ActionResult Index()
    {
        var employees = db.Employees.Include("Department");
        return View(employees.ToList());
    }

    protected override void Dispose(bool disposing)
    {
        db.Dispose();
        base.Dispose(disposing);
    }
}
```

To find the answer for the above 2 questions, delete the "Index.cshtml" view from "Views/Employee" folder. Run the application, and notice that we get an error message stating

The view 'Index' or its master was not found or no view engine supports the searched locations. The

following locations were searched:

~/Views/Employee/Index.aspx
 ~/Views/Employee/Index.ascx
 ~/Views/Shared/Index.aspx
 ~/Views/Shared/Index.ascx
 ~/Views/Employee/Index.cshtml
 ~/Views/Employee/Index.vbhtml
 ~/Views/Shared/Index.cshtml
 ~/Views/Shared/Index.vbhtml

So, from the error message, it should be clear that, **MVC looks for a view with the same name as that of the controller action method** in the following locations

1. Views/Shared
2. Views/FolderNameMatchingControllerName

Please note that, the view extension can be any of the following

- a).cshtml
- b).vbhtml
- c).aspx
- d).ascx

If I have all of the following files in "Views/Employee" folder, then MVC picks up "Index.aspx"

- a) Index.aspx
- b) Index.cshtml
- c) Index.vbhtml
- d) Index.ascx

If you want to use "Index.cshtml" instead, then specify the full path as shown below.

```
public ActionResult Index()
{
    var employees = db.Employees.Include("Department");
    return View("~/Views/Employee/Index.cshtml", employees.ToList());
}
```

If you specify only the name of the view along with it's extension as shown below, you will get an error.

```
return View("Index.cshtml", employees.ToList());
```

If you want to use a view name which is not inside the views folder of the current controller, then specify the full path as shown below.

```
public ActionResult Index()
{
    var employees = db.Employees.Include("Department");
    return View("~/Views/Home/Index.aspx", employees.ToList());
}
```

Please note that asp.net mvc

1. Is all about convention over configuration
2. Views folder contains one folder for each controller and a **"Shared"** folder
3. Shared folder is used to store views shared between controllers. Master and Layout pages are stored in "Shared" folder.

Part 33 - Html helpers in mvc

Suggested Videos

[Part 30 - View engines](#)

[Part 31 - Using custom view engines](#)

[Part 32 - How does a controller find a view](#)

In this video, we will discuss

1. The purpose of html helpers
2. Some of the standard html helpers

What is an HTML helper?

An HTML helper is a method that is used to render html content in a view. **HTML helpers** are implemented as **extension methods**.

For example, to produce the HTML for a textbox with id="firstname" and name="firstname", we can type all the html in the view as shown below

```
<input type="text" name="firstname" id="firstname" />
```

OR

We can use the "TextBox" html helper.

```
@Html.TextBox("firstname")
```

There are several overloaded versions. To set a value, along with the name, use the following overloaded version.

```
@Html.TextBox("firstname", "John")
```

The above html helper, generates the following HTML

```
<input id="firstname" name="firstname" type="text" value="John" />
```

To set **HTML attributes**, use the following overloaded version. Notice that, we are passing HTML attributes (**style & title**) as an **anonymous type**.

```
@Html.TextBox("firstname", "John", new { style = "background-color:Red; color:White; font-weight:bold", title="Please enter your first name" })
```

Some of the **html attributes**, are reserved keywords. Examples include **class**, **readonly** etc. To use these attributes, use **"@"** symbol as shown below.

```
@Html.TextBox("firstname", "John", new { @class = "redtextbox", @readonly="true" })
```

To generate a label for "First Name"

```
@Html.Label("firstname", "First Name")
```

To generate a textbox to enter password, so that the input is masked

```
@Html.Password("Password")
```

To generate a multi-line textbox with 5 rows and 20 columns

```
@Html.TextArea("Comments", "", 5, 20, null)
```

To generate a hidden textbox

```
@Html.Hidden("id")
```

Hidden textbox is used to store id values. Id values are not displayed on the page to the end user, but we need them to update data when the form is posted to the server.

Is it possible to create our own custom html helpers?

Yes, we will discuss this in a later video session.

Is it mandatory to use HTML helpers?

No, you can type the required HTML, but using HTML helpers will greatly reduce the amount of HTML that we have to write in a view. Views should be as simple as possible. All the complicated logic to generate a control can be encapsulated into the helper, to keep views simple.

Part 34 - Generating a dropdownlist control in mvc using HTML helpers

Suggested Videos

[Part 31 - Using custom view engines](#)

[Part 32 - How does a controller find a view](#)

[Part 33 - Html helpers](#)

To generate a dropdownlist, use **DropDownList** html helper. A **dropdownlist in MVC** is a collection of **SelectListItem** objects. Depending on your project requirement you may either **hard code the**

values in code or **retrieve them from a database table**. In this video, we will discuss both the approaches.

Generating a dropdownlist using hard coded values: We will use the following overloaded version of "DropDownList" html helper.

DropDownList([string](#) name, [IEnumerable<SelectListItem>](#) selectList, [string](#) optionLabel)

The following code will generate a departments dropdown list. The first item in the list will be **"Select Department"**.

```
@Html.DropDownList("Departments", new List<SelectListItem>
{
    new SelectListItem { Text = "IT", Value = "1", Selected=true},
    new SelectListItem { Text = "HR", Value = "2"},
    new SelectListItem { Text = "Payroll", Value = "3"}
}, "Select Department")
```

The downside of hard coding dropdownlist values with-in code is that, if we have to add or remove departments from the dropdownlist, the code needs to be modified.

In most cases, we get values from the database table. For this example, let's use entity framework to retrieve data. Add **ADO.NET** entity data model. We discussed working with entity framework in [Part 8](#) & [Part 25](#).

To pass list of Departments from the controller, store them in "ViewBag"

```
public ActionResult Index()
{
    // Connect to the database
    SampleDBContext db = new SampleDBContext();
    // Retrieve departments, and build SelectList
    ViewBag.Departments = new SelectList(db.Departments, "Id", "Name");

    return View();
}
```

Now in the "Index" view, access Departments list from "ViewBag"

```
@Html.DropDownList("Departments", "Select Department")
```

Part 35 - Setting an item selected when an asp.net mvc dropdownlist is loaded

Suggested Videos

[Part 32 - How does a controller find a view](#)

[Part 33 - Html helpers](#)

[Part 34 - Generating a dropdownlist control in mvc using HTML helpers](#)

In this video we will discuss, **how to set an item selected when an asp.net mvc dropdownlist options are loaded from a database table**. [Please watch Part 34](#) before proceeding.

To have the **"IT"** department selected, when the departments are loaded from **tblDepartment** table, use the following overloaded constructor of **"SelectList"** class. Notice that we are passing a value of **"1"** for "selectedValue" parameter.

```
ViewBag.Departments = new SelectList(db.Departments, "Id", "Name", "1");
```

If you run the application at this point, **"IT"** department will be selected, when the dropdownlist is rendered. The downside of hard-coding the **"selectedValue"** in code is that, application code needs to be modified, if we want **"HR"** department to be selected instead of **"IT"**. **So every time there is a change in requirement, we need to change the application code.**

Let's now discuss, the steps required to drive the selection of an item in the dropdownlist using a column in **tblDepartment** table.

Step 1: Add **"IsSelected"** bit column to tblDepartment table

ALTER TABLE tblDepartment

ADD IsSelected BIT

Step 2: At this point, this column will be null for all the rows in **tblDepartment** table. If we want **IT** department to be selected by default when the dropdownlist is loaded, set **"IsSelected=1"** for the **"IT"** department row.

Update tblDepartment **Set IsSelected = 1 Where** Id = 2

Step 3: Refresh ADO.NET Entity Data Model

Step 4: Finally, make the following changes to the **"Index()"** action method in **"HomeController"** class.

```
public ActionResult Index()
{
    SampleDbContext db = new SampleDbContext();
    List<SelectListItem> selectListItems = new List<SelectListItem>();

    foreach (Department department in db.Departments)
    {
        SelectListItem selectListItem = new SelectListItem
        {
            Text = department.Name,
            Value = department.Id.ToString(),
            Selected = department.IsSelected.HasValue ? department.IsSelected.Value : false
        };
        selectListItems.Add(selectListItem);
    }

    ViewBag.Departments = selectListItems;
    return View();
}
```

Run the application and notice that, **"IT"** department is selected, when the dropdownlist is loaded.

If you now want **"HR"** department to be selected, instead of **"IT"**, set **"IsSelected=1"** for **"HR"** department and **"IsSelected=0"** for **"IT"** department.

Update tblDepartment **Set IsSelected = 1 Where** Id = 2

Update tblDepartment **Set IsSelected = 0 Where** Id = 1

Part 36 - Difference between Html.TextBox and Html.TextBoxFor

Suggested Videos

[Part 33 - Html helpers](#)

[Part 34 - Generating a dropdownlist control in mvc using HTML helpers](#)

[Part 35 - How to set an item selected when dropdownlist is loaded](#)

Let's understand the difference

between **TextBox** and **TextBoxFor** & **DropDownList** and **DropDownListFor** HTML helpers with an example. [Please watch Part 35](#), before proceeding.

Right click on the "Models" folder and add a class file with "name=Company.cs". Copy and paste the following code.

```
public class Company
{
    private string _name;
    public Company(string name)
    {
        this._name = name;
    }
}
```

```

    }

    public List<Department> Departments
    {
        get
        {
            SampleDBContext db = new SampleDBContext();
            return db.Departments.ToList();
        }
    }

    public string CompanyName
    {
        get
        {
            return _name;
        }
        set
        {
            _name = value;
        }
    }
}

```

Copy and paste the following code in **HomeController** class. Notice that we are storing the **"Departments"** and **"CompanyName"** in the **ViewBag** object.

```

public ActionResult Index()
{
    Company company = new Company("Pragim");

    ViewBag.Departments = new SelectList(company.Departments, "Id", "Name");
    ViewBag.CompanyName = company.CompanyName;

    return View();
}

```

Right click on the **"Index"** action method in **"HomeController"** and add a view with **"name=Index"**. Copy and paste the following code. Notice that, here **the view is not strongly typed**, and we are hard-coding the name for **TextBox** and **DropDownListHTML** helpers.

```

@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>
@Html.TextBox("CompanyName", (string)ViewBag.CompanyName)
<br />
@Html.DropDownList("Departments", "Select Department")

```

Add the following **"Index1"** action method to **"HomeController"**. Notice that we are passing **"Company"** object to the View, and hence the **view is strongly typed**. Since the view is strongly typed, we can use **TextBoxFor** and **DropDownListFor** HTML helpers.

```

public ActionResult Index1()
{
    Company company = new Company("Pragim");
    return View(company);
}

```

Right click on the **"Index1"** action method in **"HomeController"** and add a view with **"name=Index1"**. Copy and paste the following code.

```

@model MVCDemo.Models.Company

@{
    ViewBag.Title = "Index1";
}

<h2>Index1</h2>
@Html.TextBoxFor(m => m.CompanyName)

```

```
<br />
@Html.DropDownListFor(m =>
m.Departments, new SelectList(Model.Departments, "Id", "Name"), "Select Department")
```

At this point, run the application and navigate to "<http://localhost/MVCDemo/home/index>". A textbox and a dropdownlist will be rendered. Right click on the page and view it's source. The generated HTML is as shown below.

```
<h2>Index</h2>
<input id="CompanyName" name="CompanyName" type="text" value="Test" />
<br />
<select id="Departments" name="Departments"><option value="">Select Department</option>
<option value="1">IT</option>
<option value="2">HR</option>
<option value="3">Payroll</option>
</select>
```

Now navigate to "<http://localhost/MVCDemo/home/index1>" and view page source. The HTML will be exactly the same as above.

So, in short, here are the differences

Html.TextBox amd Html.DropDownList are not strongly typed and hence they doesn't require a strongly typed view. This means that we can hardcode whatever name we want. On the other hand, Html.TextBoxFor and Html.DropDownListFor are strongly typed and requires a strongly typed view, and the name is inferred from the lambda expression.

Strongly typed HTML helpers also provide compile time checking.

Since, in real time, we mostly use strongly typed views, prefer to use Html.TextBoxFor and Html.DropDownListFor over their counterparts.

Whether, we use Html.TextBox & Html.DropDownList OR Html.TextBoxFor & Html.DropDownListFor, the end result is the same, that is they produce the same HTML.

Strongly typed HTML helpers are added in MVC2.

Part 37 - Generating a radiobuttonlist control in mvc using HTML helpers

Suggested Videos

[Part 34 - Generating a dropdownlist control in mvc using HTML helpers](#)

[Part 35 - How to set an item selected when dropdownlist is loaded](#)

[Part 36 - Difference between Html.TextBox and Html.TextBoxFor](#)

In this video, we will discuss, **generating a radiobuttonlist in mvc** using Html.RadioButtonFor helper. [Please watch Part 36](#) before proceeding.

Right click on the "Models" folder and add a class file with "name=Company.cs". Copy and paste the following code.

```
public class Company
{
    public string SelectedDepartment { get; set; }
    public List<Department> Departments
    {
        get
        {
            SampleDBContext db = new SampleDBContext();
            return db.Departments.ToList();
        }
    }
}
```

Copy and paste the following 2 "**Index**" action methods in **HomeController** class.

[HttpGet]

```

public ActionResult Index()
{
    Company company = new Company();
    return View(company);
}

[HttpPost]
public string Index(Company company)
{
    if (string.IsNullOrEmpty(company.SelectedDepartment))
    {
        return "You did not select any department";
    }
    else
    {
        return "You selected department with ID = " + company.SelectedDepartment;
    }
}

```

Right click on the **"Index"** action method in **"HomeController"** and add a view with **"name=Index"**. Copy and paste the following code.

```

@model MVCDemo.Models.Company
@{
    ViewBag.Title = "Index";
}

```

```
<h2>Index</h2>
```

```

@using (Html.BeginForm())
{
    foreach (var department in Model.Departments)
    {
        @Html.RadioButtonFor(m => m.SelectedDepartment, department.Id) @department.Name
    }
    <br />
    <br />
    <input type="submit" value="Submit" />
}

```

Run the application and click on **"Submit"** without selecting any department. Notice that, you get a message stating you have not selected any department. On the other hand, select a department and click **"Submit"**. The selected department ID must be displayed.

Part 38 - CheckBoxList in asp.net mvc

Suggested Videos

[Part 35 - How to set an item selected when dropdownlist is loaded](#)

[Part 36 - Difference between Html.TextBox and Html.TextBoxFor](#)

[Part 37 - Generating a radiobuttonlist control in mvc using HTML helpers](#)

In this video we will discuss implementing a checkbox list in asp.net mvc. We will be using table **"tblCity"** for this demo.

ID	Name	IsSelected
1	London	0
2	New York	0
3	Sydney	1
4	Mumbai	0
5	Cambridge	0

We should be **generating a checkbox** for each city from the table **tblCity**.

Index

☐ London
 ☒ New York
 ☒ Sydney
 ☐ Mumbai
 ☐ Cambridge

Submit

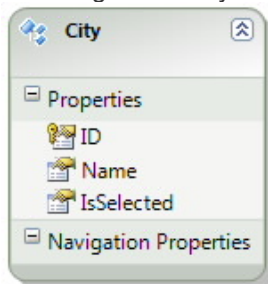
Sql script to create table tblCity

```
CREATE TABLE tblCity
(
  ID INT IDENTITY PRIMARY KEY,
  Name NVARCHAR(100) NOT NULL,
  IsSelected BIT NOT NULL
)
```

```
Insert into tblCity values ('London', 0)
Insert into tblCity values ('New York', 0)
Insert into tblCity values ('Sydney', 1)
Insert into tblCity values ('Mumbai', 0)
Insert into tblCity values ('Cambridge', 0)
```

Let's add ADO.NET data model, to retrieve data from database

1. Right click on the "Models" folder > Add > Add New Item
2. From "Add New Item" dialog box, select "ADO.NET Entity Data Model"
3. Set Name=SampleDataModel.edmx and click "Add"
4. On "Entity Data Model" wizard, select "Generate from database" and click "Next"
5. Check "Save entity connection settings in Web.Config as:" checkbox
6. Type "SampleDBContext" as the connection string name and click "Next"
7. On "Choose Your Database Objects" screen, expand "Tables" and select "tblCity" table
8. Type "Models" in "Model Namespace" textbox and click "Finish"
9. Change the entity name from "tblCity" to "City"



Right click on the "Controllers" folder, and add a "HomeController". Include the following 2 namespaces in "HomeController"

```
using MVCDemo.Models;
using System.Text;
```

Copy and paste the following code.

```
[HttpGet]
public ActionResult Index()
{
    SampleDBContext db = new SampleDBContext();
    return View(db.Cities);
}

[HttpPost]
public string Index(IEnumerable<City> cities)
{
    if (cities.Count(x => x.IsSelected) == 0)
    {
        return "You have not selected any City";
    }
    else
    {
        StringBuilder sb = new StringBuilder();
        sb.Append("You selected - ");
    }
}
```

```

        foreach (City city in cities)
        {
            if (city.IsSelected)
            {
                sb.Append(city.Name + ", ");
            }
        }
        sb.Remove(sb.ToString().LastIndexOf(","), 1);
        return sb.ToString();
    }
}

```

Right click on the **"Views"** folder, and a **"Home"** folder. Right click on the **"Home"** folder and **"EditorTemplates"** folder.

Right click on **"EditorTemplates"** folder > **Add > View**. In the **"Add View"** dialog box, set View Name = City
View Engine = Razor
and click "Add".

Copy and paste the following code in "City.cshtml"

```
@model MVCDemo.Models.City
```

```
@{
    ViewBag.Title = "City";
}
```

```
@Html.HiddenFor(x => x.ID)
@Html.HiddenFor(x => x.Name)
```

```
@Html.CheckBoxFor(x => x.IsSelected)
```

```
@Html.DisplayFor(x => x.Name)
```

Please Note: Put the templates in **"Shared"** folder, if you want the **"Templates"**, to be available for all the views.

Right click on the **"Index"** action method in **"HomeController"**, and select **"Add View"** from the context menu. Set
View Name = Index
View Engine = Razor and click "Add"

Copy and paste the following code in "Index.cshtml"

```
@model IEnumerable<MVCDemo.Models.City>
```

```
@{
    ViewBag.Title = "Index";
}
```

```
<div style="font-family:Arial">
<h2>Index</h2>
```

```
@using (Html.BeginForm())
{
    @Html.EditorForModel()
    <br />
    <input type="submit" value="Submit" />
}
</div>
```

Part 39 - ListBox in asp.net mvc

Suggested Videos

[Part 36 - Difference between Html.TextBox and Html.TextBoxFor](#)

[Part 37 - Generating a radiobuttonlist control in mvc using HTML helpers](#)

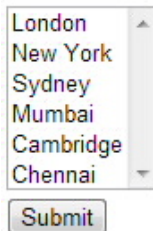
[Part 38 - CheckBoxList in asp.net mvc](#)

In this video we will discuss **implementing ListBox in asp.net mvc**. We will be using table "**tblCity**" for this demo.

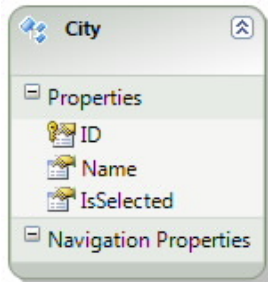
ID	Name	IsSelected
1	London	0
2	New York	0
3	Sydney	1
4	Mumbai	0
5	Cambridge	0

Please refer to [Part 38](#), if you need the sql script to create and populate this table. Please watch [Part 38](#), before proceeding.

The generated listbox should be as shown below. Notice that for each city in table "tblCity", there is an entry in the listbox.



For this demo, we will be using the "**City**" entity that we generated using **ADO.NET entity framework**, in [Part 38](#).



The first step is to create a **ViewModel** class. In asp.net mvc, view model's are used as techniques to shuttle data between the controller and the view. Right click on the "Models" folder, and add a class file with **name=CitiesViewModel.cs**. Copy and paste the following code. This class is going to be the Model for the view.

```
public class CitiesViewModel
{
    public IEnumerable<string> SelectedCities { get; set; }
    public IEnumerable<SelectListItem> Cities { get; set; }
}
```

Right click on the "**Controllers**" folder, and add a "**HomeController**". Include the following 2 namespaces in "**HomeController**"

```
using MVCDemo.Models;
using System.Text;
```

Copy and paste the following code.

```
[HttpGet]
public ActionResult Index()
{
    SampleDBContext db = new SampleDBContext();
    List<SelectListItem> listSelectListItems = new List<SelectListItem>();

    foreach (City city in db.Cities)
    {
```

```

        SelectListItem selectList = new SelectListItem()
        {
            Text = city.Name,
            Value = city.ID.ToString(),
            Selected = city.IsSelected
        };
        listSelectListItems.Add(selectList);
    }

    CitiesViewModel citiesViewModel = new CitiesViewModel()
    {
        Cities = listSelectListItems
    };

    return View(citiesViewModel);
}

[HttpPost]
public string Index(IEnumerable<string> selectedCities)
{
    if (selectedCities == null)
    {
        return "No cities selected";
    }
    else
    {
        StringBuilder sb = new StringBuilder();
        sb.Append("You selected - " + string.Join(", ", selectedCities));
        return sb.ToString();
    }
}

```

Right click on the **"Index"** action method in **"HomeController"** and select **"Add View"** from the context menu. Set
View Name = Index
View Engine = Razor
and click "Add".

Copy and paste the following code in "Index.cshtml"

```
@model MVCDemo.Models.CitiesViewModel
```

```

@{
    ViewBag.Title = "Index";
}

<div style="font-family:Arial">
<h2>Index</h2>
@using (Html.BeginForm())
{
    @Html.ListBoxFor(m => m.SelectedCities, Model.Cities, new { size = 4 })
    <br />
    <input type="submit" value="Submit" />
}
</div>

```

Part 40 - Using displayname, displayformat, scaffoldcolumn attributes in asp.net mvc application

Suggested Videos

[Part 37 - Generating a radiobuttonlist control in mvc using HTML helpers](#)

[Part 38 - CheckBoxList in asp.net mvc](#)

[Part 39 - ListBox in asp.net mvc](#)

In this video, we will discuss using the following attributes, with examples.

1. Display
2. DisplayName
3. DisplayFormat
4. ScaffoldColumn

We will be using table tblEmployee for this demo.

```
Create table tblEmployee
(
    Id int primary key identity,
    FullName nvarchar(100),
    Gender nvarchar(10),
    Age int,
    HireDate DateTime,
    EmailAddress nvarchar(100),
    Salary int,
    PersonalWebSite nvarchar(100)
)
```

```
Insert into tblEmployee values
('John Smith', 'Male', 35, '2007-01-02 17:53:46.833', 'JohnSmith@pragimtech.com',
45000, 'http://www.pragimtech.com')
Insert into tblEmployee values
('Mary Jane', NULL, 30, '2009-05-02 19:43:25.965', 'MaryJane@pragimtech.com',
35000, 'http://www.pragimtech.com')
```

Generate ADO.NET entity data model for table **tblEmployee**. Change the entity name from **tblEmployee** to **Employee**. Save and build the project.

Right click on the **"Controllers"** folder and add **"HomeController"**. Include the following **"USING"** statement.

```
using MVCDemo.Models;
```

Copy and paste the following code.

```
public class HomeController : Controller
{
    public ActionResult Details(int id)
    {
        SampleDBContext db = new SampleDBContext();
        Employee employee = db.Employees.Single(x => x.Id == id);
        return View(employee);
    }
}
```

Right click on the **"Details"** action method, and add **"Details"** view. Make sure you are creating a strongly typed view against **"Employee"** class. Select **"Details"** as the **"Scaffold Template"**. Run the application and notice that, the output is not that pretty.

We can control the display of data in a view using display attributes that are found in **System.ComponentModel.DataAnnotations** namespace. It is not a good idea, to add display attributes to the properties of auto-generated "Employee" class, as our changes will be lost, if the class is auto-generated again.

So, let's create another **partial "Employee" class**, and decorate that class with the display attributes. Right click on the "Models" folder and add Employee.cs class file. Copy and paste the following code.

```
namespace MVCDemo.Models
{
    [MetadataType(typeof(EmployeeMetaData))]
    public partial class Employee
    {
    }

    public class EmployeeMetaData
    {
    }
}
```

```

//If you want "FullName" to be displayed as "Full Name",
//use DisplayAttribute or DisplayName attribute.
//DisplayName attribute is in System.ComponentModel namespace.
//[DisplayAttribute(Name="Full Name")]
//[Display(Name = "Full Name")]
[DisplayName("Full Name")]
public string FullName { get; set; }

//To get only the date part in a datetime data type
//[DisplayFormat(DataFormatString = "{0:d}")]
//[DisplayFormatAttribute(DataFormatString="{0:d}")]

//To get time in 24 hour notation
//[DisplayFormat(DataFormatString = "{0:dd/MM/yyyy HH:mm:ss}")]

//To get time in 12 hour notation with AM PM
[DisplayFormat(DataFormatString = "{0:dd/MM/yyyy hh:mm:ss tt}")]
public DateTime? HireDate { get; set; }

// If gender is NULL, "Gender not specified" text will be displayed.
[DisplayFormat(NullDisplayText = "Gender not specified")]
public string Gender { get; set; }

//If you don't want to display a column use ScaffoldColumn attribute.
//This only works when you use @Html.DisplayForModel() helper
[ScaffoldColumn(false)]
public int? Salary { get; set; }
}
}

```

Make sure to include the following using statements:

```

using System.ComponentModel.DataAnnotations;
using System.ComponentModel;

```

We will discuss the following attributes in our next video session.

DataTypeAttribute,
DisplayColumnAttribute

Part 41 - Using datatype and displaycolumn attributes in asp.net mvc application

Suggested Videos

[Part 38 - CheckBoxList in asp.net mvc](#)

[Part 39 - ListBox in asp.net mvc](#)

[Part 40 - Using displayname, displayformat, & scaffoldcolumn attributes](#)

In this video, we will discuss using datatype and displaycolumn attributes. [Please watch Part 40](#), before proceeding.

To understand **DataType** attribute, we will be using the same example, we used in [Part 40](#). Copy and paste the following code in "**Employee**" class, and navigate to localhost/MVCDemo/Home/Details/1.

```

[MetadataType(typeof(EmployeeMetaData))]
public partial class Employee
{
}

```

```

public class EmployeeMetaData
{
    // Display mailto hyperlink
    [DataType(DataType.EmailAddress)]
    public string EmailAddress { get; set; }
}

```

// Display currency symbol. For country specific currency, set

```
// culture using globalization element in web.config.
// For Great Britain Pound symbol
// <globalization culture="en-gb"/>
[DataType(DataType.Currency)]
public int? Salary { get; set; }

// Generate a hyperlink
[DataType(DataType.Url)]
public string PersonalWebSite { get; set; }

// Display only Time Part
// [DataType(DataType.Time)]
// Display only Date Part
[DataType(DataType.Date)]
public DateTime? HireDate { get; set; }
}
```

DisplayColumn attribute is useful, when a class has a property of complex type, and you want to pick one property of this complex object for display purpose. Let's understand this with an example.

Right click on the **"Models"** folder and add **Company.cs** class file. Copy and paste the following code.

```
public class Company
{
    public Employee CompanyDirector
    {
        get
        {
            SampleDbContext db = new SampleDbContext();
            return db.Employees.Single(x => x.Id == 1);
        }
    }
}
```

Notice that, this class has **CompanyDirector** property which returns an **Employee** object. Employee is a complex type. Employee object has got several properties. If you want FullName to be used for display purpose, then make the following changes.

Decorate **"Employee"** partial class in **"Models"** folder, with **DisplayColumn** attribute.

```
[MetadataType(typeof(EmployeeMetaData))]
[DisplayColumn("FullName")]
public partial class Employee
{
}
```

Change "Details" action method in "Home" controller as shown below.

```
public ActionResult Details(int id)
{
    Company company = new Company();
    return View(company);
}
```

Copy and paste the following code in Details.cshtml view

```
@model MVCDemo.Models.Company
```

```
@{
    ViewBag.Title = "Details";
}
```

```
@Html.DisplayTextFor(x => x.CompanyDirector)
```

Navigate to localhost/MVCDemo/Home/Details/1 and you should see the FullName of the employee.

Part 42 - Opening a page in new browser window in asp.net mvc application

Suggested Videos

[Part 39 - ListBox in asp.net mvc](#)

[Part 40 - Using displayname, displayformat, & scaffoldcolumn attributes](#)

[Part 41 - Using datatype and displaycolumn attributes](#)

In this video, we will discuss opening URL's in a new window. Along the way, we also discuss using **UIHint** attribute. Please watch [Part 41](#), before proceeding. We will be using the same example that we started in [Part 40](#) of this video series.

Changes to Employee.cs class file in Models folder

```
[MetadataType(typeof(EmployeeMetaData))]
```

```
public partial class Employee
```

```
{
}
```

```
public class EmployeeMetaData
```

```
{
```

```
    [DataType(DataType.Url)]
```

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

```
}
```

Details action method in HomeController

```
public ActionResult Details(int id)
```

```
{
```

```
    SampleDbContext db = new SampleDbContext();
```

```
    Employee employee = db.Employees.Single(x => x.Id == id);
```

```
    return View(employee);
```

```
}
```

Code in Details.cshtml view

```
@model MVCDemo.Models.Employee
```

```
@{
```

```
    ViewBag.Title = "Details";
```

```
}
```

```
@Html.DisplayForModel()
```

At this point, build the application and navigate to **localhost/MVCDemo/Home/Details/1**. When you click on the personal website link, the target page will open in the same window.

If you want the page to open in a new window,

1. Right click on **"Views"** folder, and add **"Shared"** folder.
2. Right click on **"Shared"** folder, and add **"DisplayTemplates"** folder.
3. Right click on **DisplayTemplates** folder, and add a view. Set **"Url"** as the name and use Razor view engine.
4. Copy and paste the following code in **Url.cshtml** view

```
<a href="@ViewData.Model" target="_blank">@ViewData.Model</a>
```

That's it. Build the application and click on the link. Notice that, the page, now opens in a new window. The downside of this approach is that, from now on all the links, will open in a new window. To overcome this, follow these steps.

1. Rename **Url.cshtml** to **OpenInNewWindow.cshtml**
2. Decorate **"PersonalWebSite"** property in **EmployeeMetaData** class with **UIHint** attribute and specify the name of the template to use. In our case, the name of the template is **"OpenInNewWindow"**.

```
public class EmployeeMetaData
```

```
{
```

```
    [DataType(DataType.Url)]
```

```
    [UIHint("OpenInNewWindow")]
```

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

```
}
```

So, **UIHint** attribute is used to specify the name of the template to use to display the data field.

Part 43 - Hiddeninput and readonly attributes in mvc

Suggested Videos

[Part 40 - Using displayname, displayformat, & scaffoldcolumn attributes](#)

[Part 41 - Using datatype and displaycolumn attributes](#)

[Part 42 - Opening a page in new browser window in mvc](#)

HiddenInput attribute is useful when you want to render a property using **input type=hidden**. This attribute is extremely useful, when you don't want the user to see or edit the property, but you need to post the property value to the server when the form is submitted, so the correct record can be updated. **HiddenInput** is present in **System.Web.Mvc** namespace.

ReadOnly attribute is present in **System.ComponentModel** namespace. As the name suggests, this attribute is used to make a property readonly. Please note that, we will still be able to change the property value on the view, but once we post the form the model binder will respect the readonly attribute and will not move the value to the property. You can also, make property of a class **readonly** simply, by removing the SET accessor.

Changes to Employee.cs file used in the demo. Notice that Id property is decorated with HiddenInput attribute, and EmailAddress is decorated with ReadOnly attribute.

```
public class EmployeeMetadata
{
    // Id property is hidden and cannot be changed
    [HiddenInput(DisplayValue=false)]
    public int Id { get; set; }

    // EmailAddress is read only
    [ReadOnly(true)]
    [DataType(DataType.EmailAddress)]
    public string EmailAddress { get; set; }

    [ScaffoldColumn(true)]
    [DataType(DataType.Currency)]
    public int? Salary { get; set; }

    [DataType(DataType.Url)]
    [UIHint("OpenInNewWindow")]
    public string PersonalWebSite { get; set; }

    [DisplayAttribute(Name= "Full Name")]
    public string FullName { get; set; }

    [DisplayFormat(DataFormatString="{0:d}")]
    public DateTime? HireDate { get; set; }

    [DisplayFormat(NullDisplayText="Gender not specified")]
    public string Gender { get; set; }
}
```

Changes to HomeController.cs file

```
public ActionResult Edit(int id)
{
    SampleDbContext db = new SampleDbContext();
    Employee employee = db.Employees.Single(x => x.Id == id);

    return View(employee);
}
```

```
[HttpPost]
public ActionResult Edit(Employee employee)
{
    if (ModelState.IsValid)
    {
        SampleDbContext db = new SampleDbContext();
        Employee employeeFromDB = db.Employees.Single(x => x.Id == employee.Id);

        // Populate all the properties except EmailAddresses
        employeeFromDB.FullName = employee.FullName;
        employeeFromDB.Gender = employee.Gender;
        employeeFromDB.Age = employee.Age;
        employeeFromDB.HireDate = employee.HireDate;
        employeeFromDB.Salary = employee.Salary;
        employeeFromDB.PersonalWebSite = employee.PersonalWebSite;

        db.ObjectStateManager.ChangeObjectState(employeeFromDB,
        System.Data.EntityState.Modified);
        db.SaveChanges();
        return RedirectToAction("Details", new { id = employee.Id });
    }
    return View(employee);
}
```

Edit.cshtml view

```
@model MVCDemo.Models.Employee
@{
    ViewBag.Title = "Edit";
}

<div style="font-family:Arial">

    @using (Html.BeginForm())
    {
        @Html.EditorForModel()
        <br />
        <br />
        <input type="submit" value="Save" />
    }
</div>
```

Part 44 - Display and edit templated helpers in asp.net mvc

Suggested Videos

[Part 41 - Using datatype and displaycolumn attributes](#)

[Part 42 - Opening a page in new browser window in mvc](#)

[Part 43 - Hiddeninput and readonly attributes](#)

Templated helpers are introduced in mvc 2. These built in templated helpers can be broadly classified into 2 categories.

1. Display Templates
2. Editor Templates

There are 3 DISPLAY templated helpers

@Html.Display("EmployeeData") - Used with a view that is not strongly typed. For example, if you have stored data in ViewData, then we can use this templated helper using the key that was used to store data in ViewData.

@Html.DisplayFor(model => model) - Used with strongly typed views. If your model has properties that return complex objects, then this templated helper is very useful.

@Html.DisplayForModel() - Used with strongly typed views. Walks thru each property, in the model to display the object.

Along the same lines, there are 3 EDIT templated helpers

```
@Html.Editor("EmployeeData")
@Html.EditorFor(model => model)
@Html.EditorForModel()
```

To associate metadata with model class properties, we use attributes. In the previous sessions of this video series, we have discussed about using various data annotations attributes. These templated helpers use metadata associated with the model to render the user interface.

The built-in display and edit templated helpers can be very easily customised. We will discuss this in a later video session.

We will use the following Employee class that we have been working with in the previous sessions.

```
[MetadataType(typeof(EmployeeMetadata))]
public partial class Employee
{
}
public class EmployeeMetadata
{
    [HiddenInput(DisplayValue = false)]
    public int Id { get; set; }

    [ReadOnly(true)]
    [DataType(DataType.EmailAddress)]
    public string EmailAddress { get; set; }

    [ScaffoldColumn(true)]
    [DataType(DataType.Currency)]
    public int? Salary { get; set; }

    [DataType(DataType.Url)]
    [UIHint("OpenInNewWindow")]
    public string PersonalWebSite { get; set; }

    [DisplayAttribute(Name = "Full Name")]
    public string FullName { get; set; }

    [DisplayFormat(DataFormatString = "{0:d}")]
    public DateTime? HireDate { get; set; }

    [DisplayFormat(NullDisplayText = "Gender not specified")]
    public string Gender { get; set; }
}
```

Copy and paste the following **Details** action method in **HomeController**. Notice that, the employee object is stored in ViewData using **"EmployeeData"** key.

```
public ActionResult Details(int id)
{
    SampleDbContext db = new SampleDbContext();
    Employee employee = db.Employees.Single(x => x.Id == id);
    ViewData["EmployeeData"] = employee;
    return View();
}
```

Copy and paste the following code in **Details.cshtml** view. Since our employee object is in ViewData, we are using **@Html.Display("EmployeeData")** templated helper. At the moment **"Details.cshtml"** view does not have a Model associated with it. So it is not a strongly typed view.

```
@{
    ViewBag.Title = "Details";
}
```

```
<h2>Details</h2>
```

```
<fieldset>
```

```

<legend>Employee</legend>
@Html.Display("EmployeeData")
</fieldset>

```

At this point, if you run the application, you should be able to view Employee details, as expected.

Now, change the implementation of **"Details"** action method with in home controller as shown below. Notice that, instead of storing the **"Employee"** object in ViewData, we are passing it to the View.

```

public ActionResult Details(int id)
{
    SampleDbContext db = new SampleDbContext();
    Employee employee = db.Employees.Single(x => x.Id == id);
    return View(employee);
}

```

Change **Details.cshtml** view as below. We have specified **"Employee"** as the model object. So, here we are working with a strongly typed view, and hence we are using `@Html.DisplayFor(model => model)` templated helper. Since, none of the properties of Employee class return a complex object, the ideal choice here would be, to use `@Html.DisplayForModel()` templated helper. In either cases, in this scenario you will get the same output.

```

@model MVCDemo.Models.Employee
@{
    ViewBag.Title = "Details";
}

```

```

<h2>Details</h2>

<fieldset>
    <legend>Employee</legend>
    @Html.DisplayFor(model => model)
</fieldset>

```

You work with **Editor templates** in the same way. In HomeController, implement **Edit** action method as shown below.

```

public ActionResult Edit(int id)
{
    SampleDbContext db = new SampleDbContext();
    Employee employee = db.Employees.Single(x => x.Id == id);
    return View(employee);
}

```

and in Edit.cshtml view

```

@model MVCDemo.Models.Employee
@{
    ViewBag.Title = "Edit";
}

```

```

<h2>Edit</h2>

@using (@Html.BeginForm())
{
    @Html.EditorForModel()
}

```

Part 45 - Customize display and edit templates in asp.net mvc

Suggested Videos

[Part 42 - Opening a page in new browser window in mvc](#)

[Part 43 - Hiddeninput and readonly attributes](#)

Part 44 - Display and edit templated helpers

In this video, we will discuss customizing datetime editor template. Please watch [Part 44](#), before proceeding. We will be using the same example, that we worked with in [Part 44](#).

At the moment, when you navigate to **localhost/MVCDemo/Home/Edit/1**, the output is as shown below.

Full Name

Gender

Age

HireDate

EmailAddress

Salary

PersonalWebSite

Notice that, for **HireDate**, users have to type in the date. Dates have got different formats. For example, **MM/DD/YYYY** or **DD/MM/YY** etc. So, different users may type it differently. Also, from a user experience, it is better to display a **DateTime** picker from which the user can simply select the date.

The **built-in DateTime editor** template used by MVC, simply displays a textbox for editing Dates. So, let's customize the DateTime editor template, to use jQuery calendar. We want the output as shown below.

Full Name

Gender

Age

HireDate

May 2012						
Su	Mo	Tu	We	Th	Fr	Sa
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

The following is the convention used by MVC to find the customized templates

1. The customized display templates must be in a sub-folder that is named **-DisplayTemplates**. Editor templates must be in a sub-folder that is named **-EditorTemplates**.
2. These sub-folders can live in **"Shared"** folder, or a specific views folder. If these folders are present in the Shared folder, then the templates are available for all the views. If they are in a specific views folder, then, they are available only for that set of views.
3. The **name of the template** must match the **name of the type**. For example, as we are customizing DateTime template, the name of the template in this case has to be DateTime.ascx or DateTime.cshtml.

Adding a Custom DateTime Editor template

Step 1: If **"Shared"** folder does not already exists in your project, right click on the project in solution explorer and add it.

Step 2: Right click on the **"Shared"** folder, and **"EditorTemplates"** folder.

Step 3: Right click on **"EditorTemplates"** folder and add a view with **name = DateTime**

Step 4: Copy and paste the following code in **DateTime.cshtml** partial view

```
@model DateTime?
@Html.TextBox("", (Model.HasValue ? Model.Value.ToString("dd/MM/yyyy") : string.Empty), new {
    @class = "date" })
```

Note: Please refer to the following MSDN article for all the DateTime format strings

<http://msdn.microsoft.com/en-us/library/8kb3ddd4.aspx>

Step 5: Copy and paste the following code in **Edit.cshtml** view

```
@model MVCDemo.Models.Employee
```

```
@{
    ViewBag.Title = "Edit";
}
```

```
<h2>Edit</h2>
```

```
<script src="~/Scripts/jquery-1.7.1.min.js" type="text/javascript"></script>
```

```
<script src="~/Scripts/jquery-ui-1.8.20.min.js" type="text/javascript"></script>
```

```
<link href="~/Content/Site.css" rel="stylesheet" type="text/css" />
```

```
<link href="~/Content/themes/base/jquery.ui.all.css" rel="stylesheet" type="text/css" />
```

```
<script type="text/javascript">
```

```
$(function()
{
    $("input:text.date").datepicker(
    {
        dateFormat: "dd/mm/yy"
    });
});
```

```
</script>
```

```
@using (@Html.BeginForm())
```

```
{
    @Html.EditorForModel()
    <br />
    <input type="submit" value="Save" />
}
```

Note: Please refer to the following jQuery link for DateTime format strings

<http://jqueryui.com/resources/demos/datepicker/date-formats.html>

The following **jQuery scripts** and **css** files are required for **jQuery DateTime picker**

control. However, these files may change depending on the version of jQuery you are working with.

Scripts/jquery-1.7.1.min.js

Scripts/jquery-ui-1.8.20.min.js

Content/Site.css

Content/themes/base/jquery.ui.all.css

Part 46 - Accessing model metadata from custom templated helpers

Suggested Videos

[Part 43 - Hiddeninput and readonly attributes](#)

[Part 44 - Display and edit templated helpers](#)

[Part 45 - Customize display and edit templates](#)

In this video, we will discuss, **accessing model metadata from with in customized display and edit templated helpers**. We will be using the same example, that we worked with in [Part 45](#).

In [Part 45](#), we have customized DateTime editor template to use jQuery calendar. The problem with this template is that, we have hard-coded the date format string to **dd/MM/yyyy**. So, from this point, any DateTime property that uses this template will format the date using the hard-coded date format string.

We want our DateTime template to be generic. We don't want any hard-coded logic, with-in the DateTime editor template. The DisplayFormat attribute on the HireDate property of the Employee class must be used to determine the display format of the date.

Let's remove the hard-coded date format string (**dd/MM/yyyy**) from DateTime.cshtml.

```
@Html.TextBox("", Model.HasValue ? Model.Value.ToString("dd/MM/yyyy") : "", new { @class = "date" })
```

Decorate **HireDate** property in Employee class, with **DisplayFormat** attribute as shown below.

Also, make sure **ApplyFormatInEditMode** parameter is set to true, otherwise the formatting will not be applied in Edit mode.

```
[DisplayFormat(DataFormatString = "{0:dd/MM/yyyy}", ApplyFormatInEditMode=true)]
```

```
public DateTime? HireDate { get; set; }
```

Now, change the code in DateTime.cshtml to use the formatted datetime value as show below.

Notice that, we are using ViewData.TemplateInfo.FormattedModelValue.

```
@Html.TextBox("", Model.HasValue ? ViewData.TemplateInfo.FormattedModelValue : "", new { @class = "date" })
```

To access model metadata in templates, use `@ViewData.ModelMetadata`. For, example to access the DisplayFormatString, use `@ViewData.ModelMetadata.DisplayFormatString`. Along the same lines, if you want to know, the name of the containing class(i.e the class that contains the HireDate property) then use `@ViewData.ModelMetadata.ContainerType.ToString()`.

Part 47 - Displaying images in asp.net mvc

Suggested Videos

[Part 44 - Display and edit templated helpers](#)

[Part 45 - Customize display and edit templates](#)

[Part 46 - Accessing model metadata from custom templated helpers](#)

In this video, we will discuss **displaying images in MVC application**. In Part 48, we will create a custom html helper to display images. We will be using the example, that we worked with in [Part 46](#). We want to display **Employee** photo, along with the personal details as you can see in the image below.

Employee
Full Name
John Smith
Gender
Male
Age
35
HireDate
30/11/2007
EmailAddress
JohnSmith@pragimtech.com
Salary
£45,000.00
PersonalWebSite
http://www.pragimtech.com
Photo


Alter table tblEmployee to add Photo, and AlternateText columns.

Alter table tblEmployee Add Photo **nvarchar**(100), AlternateText **nvarchar**(100)

Update Photo and AlternateText columns

Update tblEmployee set Photo='~/Photos/JohnSmith.png',
AlternateText = 'John Smith Photo' where Id = 1

Right click on the solution explorer and add **"Photos"** folder. Download the following image and paste in **"Photos"** folder.



Now, in MVCDemo project, update SampleDataModel.edmx.

1. Right click on **"Employee"** table and select **"Update Model from database"** option
2. On **"Update Wizard"** window, click on **"Refresh"** tab
3. Expand tables node, and select **"tblEmployee"** table
4. Finally click **"Finish"**

At this point, **"Photo"** and **"AlternateText"** properties must be added to the auto-generated **"Employee"** class.

Generate Details view

1. Delete **"Details.cshtml"** view, if it already exists.
2. Right click on **"Details"** action method and select **"Add View"**
3. In **"Add View"** window, set
View Name = Details

View engine = Razor
 Create a strongly typed view = Select the checkbox
 Model class = Employee(MVCDemo.Models)
 Scaffold template = Details

Notice that for **Photo** and **AlternateText** properties, the following HTML is generated. At this point, if you run the application, instead of rendering the photo, the PhotoPath and AlternateText property values are displayed.

```
<div class="display-label">
    @Html.DisplayNameFor(model => model.Photo)
</div>
<div class="display-field">
    @Html.DisplayFor(model => model.Photo)
</div>

<div class="display-label">
    @Html.DisplayNameFor(model => model.AlternateText)
</div>
<div class="display-field">
    @Html.DisplayFor(model => model.AlternateText)
</div>
```

Replace the above code with the following. Notice that, we are using **Url.Content()** html helper method. This method resolves a url for a resource when we pass it the relative path.

```
<div class="display-label">
    @Html.DisplayNameFor(model => model.Photo)
</div>
<div class="display-field">
    
</div>
```

Run the application, and notice that, the image is rendered as expected. In our next video, we will discuss creating a custom html image helper.

Part 48 - Custom html helpers in mvc

Suggested Videos

[Part 45 - Customize display and edit templates](#)

[Part 46 - Accessing model metadata from custom templated helpers](#)

[Part 47 - Displaying images in MVC](#)

In this video, we will discuss, **creating custom html helpers**. We will be using the example, that we worked with in [Part 47](#). Please watch [Part 47](#), before proceeding.

In [Part 47](#), to render the image, we used the following code. We are building the image tag, by passing values for "src" and "alt" attributes.

```

```

Though the above code is not very complex, it still makes sense to move this logic into its own helper method. We don't want any complicated logic in our views. Views should be as simple as possible. Don't you think, it would be very nice, if we can render the image, using Image() html helper method as shown below. Unfortunately, MVC does not have built-in Image() html helper. So, let's build our own custom image html helper method.

```
@Html.Image(Model.Photo, Model.AlternateText)
```

Let's take a step back and understand html helper methods. The HTML helper method simply returns a string. To generate a textbox, we use the following code in our view.

```
@Html.TextBox("TextBox Name")
```

So, here **TextBox()** is an extension method defined in **HtmlHelper** class. In the above code, **Html** is the property of the View, which returns an instance of the **HtmlHelper** class.

Let's now add, Image() extension method, to HtmlHelper class.

1. Right click on **MVCDemo** project and add "**CustomHtmlHelpers**" folder.
2. Right click on "**CustomHtmlHelpers**" folder and add "**CustomHtmlHelpers.cs**" class file.
3. Copy and paste the following code. The code is commented and self-explanatory. **TagBuilder** class is in **System.Web.Mvc** namespace.

```
namespace MVCDemo.CustomHtmlHelpers
{
    public static class CustomHtmlHelpers
    {
        public static IHtmlString Image(this HtmlHelper helper, string src, string alt)
        {
            // Build <img> tag
            TagBuilder tb = new TagBuilder("img");
            // Add "src" attribute
            tb.Attributes.Add("src", VirtualPathUtility.ToAbsolute(src));
            // Add "alt" attribute
            tb.Attributes.Add("alt", alt);
            // return MvcHtmlString. This class implements IHtmlString
            // interface. IHtmlStrings will not be html encoded.
            return new MvcHtmlString(tb.ToString(TagRenderMode.SelfClosing));
        }
    }
}
```

To use the custom Image() html helper in **Details.cshtml** view, please include the following using statement in Details.cshtml

```
@using MVCDemo.CustomHtmlHelpers;
```

As we intend to use this **Image()** html helper, in all our views, let's include "**MVCDemo.CustomHtmlHelpers**" namespace in **web.config**. This eliminates the need to include the namespace, in every view.

```
<system.web.webPages.razor>
  <pages pageBaseType="System.Web.Mvc.WebViewPage">
    <namespaces>
      <add namespace="System.Web.Mvc" />
      <add namespace="System.Web.Mvc.Ajax" />
      <add namespace="System.Web.Mvc.Html" />
      <add namespace="System.Web.Routing" />
      <add namespace="MVCDemo.CustomHtmlHelpers" />
    </namespaces>
  </pages>
</host ...../>
</system.web.webPages.razor>
```

If you intend to use the Image() custom html helper, only with a set of views, then, include a web.config file in the specific views folder, and then specify the namespace in it.

Part 49 - Html encoding in asp.net mvc

Suggested Videos

[Part 46 - Accessing model metadata from custom templated helpers](#)

[Part 47 - Displaying images in MVC](#)

[Part 48 - Custom html helpers in mvc](#)

In this video, we will discuss

1. What is HTML encoding
2. Why would you html encode
3. How to avoid html encoding in aspx and razor views

What is HTML encoding?

HTML encoding is the process of replacing ASCII characters with their 'HTML Entity' equivalents. For example replacing

ASCII characters	HTML Encoded
<	<
>	>
&	&
Single Quote (')	'
Double Quote (")	"

Why would you html encode?

To avoid cross site scripting attacks, all output is automatically html encoded in mvc. We will discuss cross-site scripting attack in a later video session.

Avoiding html encoding in razor views:

Sometimes, we have to avoid HTML encoding. There are 2 ways to disable html encoding

1. `@Html.Raw("YourHTMLString")`
2. Strings of type `IHtmlString` are not encoded

Consider the following custom Image() html helper.

```
public static class CustomHtmlHelpers
{
    public static IHtmlString Image(this HtmlHelper helper, string src, string alt)
    {
        TagBuilder tb = new TagBuilder("img");
        tb.Attributes.Add("src", VirtualPathUtility.ToAbsolute(src));
        tb.Attributes.Add("alt", alt);
        return new MvcHtmlString(tb.ToString(TagRenderMode.SelfClosing));
    }
}
```

Notice that, this custom **Image()** HTML helper method returns string of type, **IHtmlString**. Strings of type `IHtmlString` are excluded from html encoding. So, when we invoke **Image()** helper method from a razor view as shown below, the image is rendered as expected.

```
@Html.Image(@Model.Photo, @Model.AlternateText)
```

However, if you modify the **Image()** method to return string of type **System.String**, the HTML is encoded and that's what is shown on the view, instead of actually rendering the image.

```

```

@Html.Raw() method can also be used to avoid automatic html encoding. Notice that, the string that is returned by **Image()** method is passed as the input for **Raw()** method, which renders the image as expected.

```
@Html.Raw(Html.Image(@Model.Photo, @Model.AlternateText))
```

Avoiding html encoding in ASPX views:

`<%= %>` syntax will automatically encode html in aspx views. So, the following will encode and display the html, instead of rendering the image. At the moment, the custom **Image()** html helper method is returning string of type `system.string`. If you make this method return `IHtmlString`, then the following code will render the image instead of html encoding it.

```
<%= Html.Image(Model.Photo, Model.AlternateText) %>
```

To avoid automatic html encoding, you can use

1. `<%= %>`
2. `Html.Raw()`
3. Strings of type `IHtmlString` are not encoded

Both the following code blocks will render the image

```
<%= Html.Image(Model.Photo, Model.AlternateText) %>
```

OR

```
<%= Html.Raw(Html.Image(Model.Photo, Model.AlternateText)) %>
```

Different techniques to avoid automatic html encoding in MVC

Technique	Razor Views	ASPX Views
IHTML Strings	✓	✓
HTML.Raw()	✓	✓
<% = %>	✗	✓

Part 50 - Detect errors in views at compile time

Suggested Videos

[Part 47 - Displaying images in MVC](#)

[Part 48 - Custom html helpers in mvc](#)

[Part 49 - Html encoding in asp.net mvc](#)

In this video, we will discuss, **detecting errors in views at compile-time rather than at run-time.**

The following code will display employee's **FullName** and **Gender**. Here we are working with a strongly typed view. **Employee** is the model class for this view. This class has got **"FullName"** and **"Gender"** properties.

```
@model MVCDemo.Models.Employee
<fieldset>
    <legend>Employee</legend>

    <div class="display-label">
        @Html.DisplayNameFor(model => model.FullName)
    </div>
    <div class="display-field">
        @Html.DisplayFor(model => model.FullName)
    </div>

    <div class="display-label">
        @Html.DisplayNameFor(model => model.Gender)
    </div>
    <div class="display-field">
        @Html.DisplayFor(model => model.Gender)
    </div>
</fieldset>
```

For example, if you **mis-spell FullName property** as shown below, and when you compile the project, you wouldn't get any compile time errors.

```
@Html.DisplayNameFor(model => model.FullName1)
```

You will only come to know, about the error when the page crashes at run-time. If you want to enable compile time error checking for views in MVC

1. Open MVC project file using a notepad. Project files have the extension of **.csproj** or **.vbproj**
2. Search for **MvcBuildViews** under **PropertyGroup**. **MvcBuildViews** is **false** by default. Turn this to **true** as shown below.

```
<MvcBuildViews>true</MvcBuildViews>
```

3. Save the changes.

If you now build the project, you should get compile time error.

Please Note: Pre-compiling views is different from compile-time error checking. We will discuss pre-compiling views in a later video session.

Part 51 - Advantages of using strongly typed views

Suggested Videos

[Part 48 - Custom html helpers in mvc](#)

[Part 49 - Html encoding in asp.net mvc](#)

[Part 50 - Detect errors in views at compile time](#)

There are several ways available to pass data **from a controller to a view** in an mvc application.

1. ViewBag or ViewData
2. Dynamic type
3. Strongly typed view

The following are the advantages of using strongly typed views. We get

1. Intellisense and
2. Compile-time error checking

With ViewBag and Dynamic type, we don't have these advantages.

Using ViewBag:

Notice that, the employee object is stored in ViewBag.

In HomeController.cs

```
public ActionResult Details(int id)
{
    SampleDBContext db = new SampleDBContext();
    Employee employee = db.Employees.Single(x => x.Id == id);
    ViewBag.EmployeeData = employee;
    return View();
}
```

We want to display **employee FullName** and **Gender**. Notice that, as we are typing **FullName** and **Gender** properties, we don't get **intellisense**. Also, if we misspell **FullName** or **Gender** properties, we will not get any compilation errors. We will come to know about these errors only at runtime.

In Details.cshtml View

```
<div class="display-label">
    @Html.DisplayName("FullName")
</div>
<div class="display-field">
    @ViewBag.EmployeeData.FullName
</div>
<div class="display-label">
    @Html.DisplayName("Gender")
</div>
<div class="display-field">
    @ViewBag.EmployeeData.Gender
</div>
```

Using Dynamic Type:

In HomeController.cs

```
public ActionResult Details(int id)
{
    SampleDBContext db = new SampleDBContext();
    Employee employee = db.Employees.Single(x => x.Id == id);
    return View(employee);
}
```

In Details.cshtml View

```
@model dynamic
<div class="display-label">
    @Html.DisplayName("FullName")
</div>
<div class="display-field">
    @Model.FullName
</div>
```



```

<div class="display-label">
    @Html.DisplayName("Gender")
</div>
<div class="display-field">
    @Model.Gender
</div>

```

With dynamic type also, we don't get intellisense and compile-time error checking.

Using Strongly Typed View: No change is required in the controller action method. Make the following change to **Details.cshtml** view. Notice that the view is strongly typed against Employee model class. We get **intellisense** and if we **mis-spell** a property name, we get to know about it at compile time.

```

@model MVCDemo.Models.Employee
<div class="display-label">
    @Html.DisplayName("FullName")
</div>
<div class="display-field">
    @Model.FullName
</div>
<div class="display-label">
    @Html.DisplayName("Gender")
</div>
<div class="display-field">
    @Model.Gender
</div>

```

Please Note: We discussed enabling compile time error checking in views in [Part 50](#) of the [MVC tutorial](#).

art 52 - Partial views in mvc

Suggested Videos

[Part 49 - Html encoding in asp.net mvc](#)

[Part 50 - Detect errors in views at compile time](#)

[Part 51 - Advantages of using strongly typed views](#)

In this video we will discuss partial views in mvc.

If you are an asp.net web-forms developer, then you will realize that **partial views in mvc are similar to user controls in asp.net webforms**.

Partial views are used to **encapsulate re-usable view logic** and are a great means to simplify the complexity of views. These partial views can then be used on multiple views, where we need similar view logic.

If you are using web forms view engine, then the partial views have the extension of **.ascx**. If the view engine is razor and programming language is c#, then partial views have the extension of **.cshtml**. On the other hand if the programming language is visual basic, then the extension is **.vbhtml**.

Language	Razor Views	ASPX Views
C#	.cshtml	.ascx
Visual Basic	.vbhtml	.ascx

Let us understand **partial views with an example**. We want to display, employee photo and his details as shown in the image below.

	Age: 35 Gender: Male Salary: 45000
	Age: 30 Gender: Female Salary: 35000
	Age: 28 Gender: Male Salary: 6200

Index Action() method in HomeController retrurns the list of employees.

```
public ActionResult Index()
{
    SampleDbContext db = new SampleDbContext();
    return View(db.Employees.ToList());
}
```

We will have the following code in Index.cshtml. This view is a bit messy and complex to understand.

```
@model IEnumerable<MVCDemo.Models.Employee>
@foreach (var item in Model)
{
    <table style="font-family:Arial; border:1px solid black; width: 300px">
    <tr>
    <td>
    
    </td>
    <td>
    <table>
    <tr>
    <td><b>Age:</b></td>
    <td>@item.Age</td>
    </tr>
    <tr>
    <td><b>Gender:</b></td>
    <td>@item.Gender</td>
    </tr>
    <tr>
    <td><b>Salary:</b></td>
    <td>@item.Salary</td>
    </tr>
    </table>
    </td>
    </tr>
    </table>
}
```

To **simplify this view**, let's encapsulate the HTML and code that produces the employee table in a partial view.

Right click on the **"Shared"** folder and add a view. Set

View name = `_Employee`

View engine = Razor

Create a strongly typed view = Checked

Model class = Employee (MVCDemo.Models)

Scaffold template = Empty

Create as a partial view = Checked

This should add **"_Employee.cshtml"** partial view to the **"Shared"** folder.

Please note that, partial views can be added to **"Shared"** folder or to a **specific views folder**. Partial views that are in the "Shared" folder are available for all the views in the entire project, where as partial views in a specific folder are available only for the views with-in that folder.

Copy and paste the following code in **"_Employee.cshtml"** partial view

```
@model MVCDemo.Models.Employee
<table style="font-family:Arial; border:1px solid black; width: 300px">
  <tr>
    <td>
      
    </td>
    <td>
      <table>
        <tr>
          <td><b>Age:</b></td>
          <td>@Model.Age</td>
        </tr>
        <tr>
          <td><b>Gender:</b></td>
          <td>@Model.Gender</td>
        </tr>
        <tr>
          <td><b>Salary:</b></td>
          <td>@Model.Salary</td>
        </tr>
      </table>
    </td>
  </tr>
</table>
```

Now, make the following changes to **Index.cshtml** view. Notice that the view is much simplified now. To render the partial view, we are using **Partial()** html helper method. There are several overloaded versions of this method. We are using a version that expects **2 parameters**, i.e the name of the partial view and the model object.

```
@model IEnumerable<MVCDemo.Models.Employee>
@foreach (var item in Model)
{
    @Html.Partial("_Employee", item)
}
```

art 53 - Difference between html.partial and html.renderpartial

Suggested Videos

[Part 50 - Detect errors in views at compile time](#)

[Part 51 - Advantages of using strongly typed views](#)

[Part 52 - Partial views in mvc](#)

In this video, we will discuss the difference between **Partial()** and **RenderPartial()** html helper methods. Both of these helper methods are used for rendering partial views.

Differences:

1. The return type of **"RenderPartial"** is **void**, where as **"Partial"** returns **"MvcHtmlString"**

2. Syntax for invoking **Partial()** and **RenderPartial()** methods in Razor views

```
@Html.Partial("PartialViewName")
{ Html.RenderPartial("PartialViewName"); }
```

3. Syntax for invoking **Partial()** and **RenderPartial()** methods in webform views

```
<%= Html.Partial("PartialViewName") %>
```

```
<% Html.RenderPartial("PartialViewName"); %>
```

The following are the 2 common interview questions related to **Partial()** and **RenderPartial()**

When would you use **Partial()** over **RenderPartial()** and vice versa?

The main difference is that "**RenderPartial()**" returns void and the output will be written directly to the output stream, where as the "**Partial()**" method returns **MvcHtmlString**, which can be assigned to a variable and manipulate it if required. So, when there is a need to assign the output to a variable for manipulating it, then use **Partial()**, else use **RenderPartial()**.

Which one is better for performance?

From a performance perspective, rendering directly to the output stream is better. **RenderPartial()** does exactly the same thing and is **better for performance** over **Partial()**.

Part 54 - T4 templates in asp.net mvc

Suggested Videos

[Part 51 - Advantages of using strongly typed views](#)

[Part 52 - Partial views in mvc](#)

[Part 53 - Difference between html.partial and html.renderpartial](#)

In this video, we will discuss

1. What are T4 templates and their purpose
2. Customizing T4 templates

What are T4 templates and their purpose?

T4 stands for **Text Template Transformation Toolkit** and are used by visual studio to generate code when you add a view or a controller.

Where does these T4 templates live?

C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\ItemTemplates\[CSharp | FSharp | VisualBasic]\Web\[MVC 2 | MVC 3 | MVC 4]\CodeTemplates

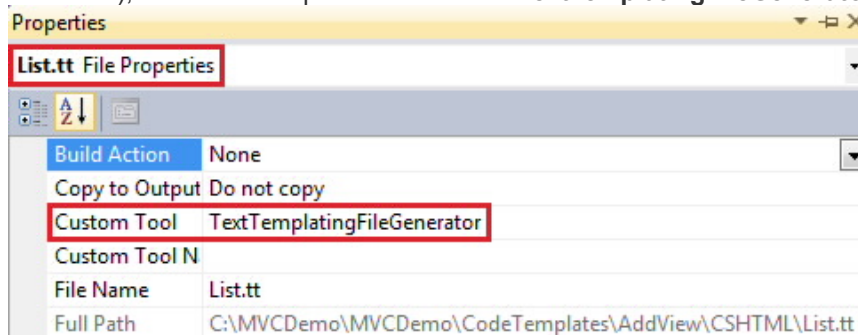
What is the file extension for T4 templates?

.tt

Is it possible to customize T4 templates?

Yes, T4 templates can be customized in place, if you want the customized T4 templates available for all MVC projects on that machine.

If you want to customize T4 templates, only for a specific project, then copy "**CodeTemplates**" folder and paste it in the root directory of your MVC project. Right click on the template files (with .tt extension), and select Properties and delete "**TextTemplatingFileGenerator**".



By deleting this from **CustomTool** property, we are telling visual studio not to run them during the build. They will be manually called when we add a view or a controller using "**Add View**" and "**Add Controller**" dialog box.

Is it possible to add your own T4 template to the existing list of templates?

Absolutely, simply create a file with ".tt" file extension in "AddController" folder in "CodeTemplates". If it is for adding a view, then put it in "AspxCSharp"(if view engine is aspx) or "CSHTML"(if view engine is razor) folder.

Part 55 - What is cross site scripting attack

Suggested Videos

[Part 52 - Partial views in mvc](#)

[Part 53 - Difference between html.partial and html.renderpartial](#)

[Part 54 - T4 templates in asp.net mvc](#)

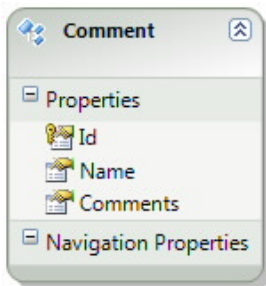
Cross-site scripting attack also called as **XSS attack**, is a security vulnerability found in Web applications. **XSS** allows hackers to inject client-side script into Web pages, and later, if that web page is viewed by others, the stored script gets executed. **The consequences of XSS may range from a petty nuisance like displaying an alert() box to a significant security risk, like stealing session cookies.**

Let's understand cross site scripting attack with an example. We will be using table tblComments for this demo.

CREATE TABLE tblComments

```
(
  Id INT IDENTITY PRIMARY KEY,
  Name NVARCHAR(50),
  Comments NVARCHAR(500)
)
```

Create an asp.net mvc 4.0 application. Right click on "Models" folder, and add **ADO.NET entity data model**, based on table **tblComments**. Change the name of the entity from **tblComment** to **Comment**.



Build the solution, to make sure "**Comment**" model is compiled.

Now right click on Controllers folder, and add a "new controller". Set

1. Controller name = HomeController
2. Template = MVC controller with read/write actions and views, using Entity Framework
3. Model class = Comment
4. Data context class = SampleDbContext
5. Views = Razor (CSHTML)

Click "**Add**". This generates the **HomeController** and the required views.

In **Create.cshtml** view, change

@Html.EditorFor(model => model.Comments)

TO

@Html.TextAreaFor(model => model.Comments)

The above change is to display a multi-line textbox for entering comments.

The screenshot shows a web form titled 'Comment'. It contains two input fields: 'Name' (a single-line text box) and 'Comments' (a multi-line text box). Below the 'Comments' field is a 'Create' button.

Now let's inject javascript using the "Create" view. Navigate to "localhost/MVCDemo/Home/Create" view. Type the following client-side script, into "Comments" textbox, and click "Create". We get an error - **A potentially dangerous Request.Form value was detected from the client (Comments="<script>")**. This is one of the security measures in place, to prevent script injection, to avoid XSS attack.

```
<script type="text/javascript">
alert('Your site is hacked');
</script>
```

Depending on the project requirement, there may be legitimate reasons, to submit html. For example, let's say we want to bold and underline a word in the comment that we type. To bold and underline the word "very good" in the following statement, we would type a statement as shown below and submit.

This website is <u>very good</u>

To allow this HTML to be submitted. We need to disable input validation. When we click the "Create" button on "Create" view, the "Create()" controller action method that is decorated with [HttpPost] attribute in the "HomeController". To disable input validation, decorate Create() action method with "ValidateInput" attribute as shown below.

```
[HttpPost]
[ValidateInput(false)]
public ActionResult Create(Comment comment)
{
    if (ModelState.IsValid)
    {
        db.Comments.AddObject(comment);
        db.SaveChanges();
        return RedirectToAction("Index");
    }

    return View(comment);
}
```

Build the solution. Navigate to "Create" view. Enter

Name = Tom

Comments = This website is <u>very good</u>

The screenshot shows the 'Create' view after data entry. The 'Name' field contains 'Tom'. The 'Comments' field contains the text: 'This website is <u>very good</u>'. The 'Create' button is still visible at the bottom.

Now click "Create". Data gets saved as expected, and the user is redirected to Index action. On

the "Index" view, instead of rendering the word "**very good**" with an under-line and in bold, the encoded html is displayed as shown below.

Name	Comments
Tom	This website is <u>very good</u> Edit Details Delete

By default asp.net mvc encodes all html. This is another security measure in place, to prevent XSS attack. To disable html encoding, make the following change on Index.cshtml view.

CHANGE

@Html.DisplayFor(modelItem => item.Comments)

To

@Html.Raw(item.Comments)

Navigate to Index view, and notice that, the word "**Very Good**" is rendered with an underline and in bold.

Name	Comments
Tom	This website is <u>very good</u> Edit Details Delete

The following 2 changes that we have done has opened the doors for XSS.

1. By allowing HTML to be submitted on Create.cshtml view
2. By disabling HTML encoding on Index.cshtml

To initiate a cross site scripting attack

1. Navigate to "Create" view
2. Type the following javascript code in "Comments" textbox

```
<script type="text/javascript">
alert('Your site is hacked');
</script>
```
3. Click "Create"

Refresh Index view and notice that javascript alert() box is displayed, stating that the site is hacked. By injecting script, it is also very easy to steal session cookies. These stolen session cookies can then be used to log into the site and do destructive things.

So, in short, by allowing HTML to be submitted and disabling HTML encoding we are opening doors for XSS attack.

In our next video, we will discuss preventing XSS while allowing only the HTML that we want to accept

art 56 - How to prevent cross site scripting attack

Suggested Videos

[Part 53 - Difference between html.partial and html.renderpartial](#)

[Part 54 - T4 templates in asp.net mvc](#)

[Part 55 - What is cross site scripting attack](#)

In this video, we will discuss **preventing XSS while allowing only the HTML that we want to accept**. For example, we only want to accept **** and **<u>** tags.

To achieve this let's filter the user input, and accept only **** and **<u></u>** tags. The following code,

1. Disables input validation
2. Encodes all the input that is coming from the user
3. Finally we selectively replace, the encoded html with the HTML elements that we want to allow.

[HttpPost]

// Input validation is disabled, so the users can submit HTML

```
[ValidateInput(false)]
public ActionResult Create(Comment comment)
{
    StringBuilder sbComments = new StringBuilder();

    // Encode the text that is coming from comments textbox
    sbComments.Append(HttpUtility.HtmlEncode(comment.Comments));

    // Only decode bold and underline tags
    sbComments.Replace("&lt;b&gt;", "<b>");
    sbComments.Replace("&lt;/b&gt;", "</b>");
    sbComments.Replace("&lt;u&gt;", "<u>");
    sbComments.Replace("&lt;/u&gt;", "</u>");
    comment.Comments = sbComments.ToString();

    // HTML encode the text that is coming from name textbox
    string strEncodedName = HttpUtility.HtmlEncode(comment.Name);
    comment.Name = strEncodedName;

    if (ModelState.IsValid)
    {
        db.Comments.AddObject(comment);
        db.SaveChanges();
        return RedirectToAction("Index");
    }

    return View(comment);
}
```

Warning: Relying on just filtering the user input, cannot guarantee XSS elimination. XSS can happen in different ways and forms. This is just one example. Please read MSDN documentation on XSS and it's counter measures.

Part 57 - Razor views in mvc

Suggested Videos

[Part 54 - T4 templates in asp.net mvc](#)

[Part 55 - What is cross site scripting attack](#)

[Part 56 - How to prevent cross site scripting attack](#)

In this video, we will discuss razor view syntax.

Use @ symbol to switch between c# code and html.

```
@for (int i = 1; i <= 10; i++)
{
    <b>@i</b>
}
```

Output:

1 2 3 4 5 6 7 8 9 10

Use @{ } to define a code block. If we want to define some variables and perform calculations, then use code block. The following code block defines 2 variables and computes the sum of first 10 even and odd numbers.

```
@{
    int SumOfEvenNumbers = 0;
    int SumOfOddNumbers = 0;

    for(int i = 1; i <= 10; i++)
    {
```

```

        if(i %2 == 0)
        {
            SumOfEvenNumbers = SumOfEvenNumbers + i;
        }
        else
        {
            SumOfOddNumbers = SumOfOddNumbers + i;
        }
    }
}

```

<h3>Sum of Even Numbers = @SumOfEvenNumbers</h3>

<h3>Sum of Odd Numbers = @SumOfOddNumbers</h3>

Output:

Sum of Even Numbers = 30

Sum of Odd Numbers = 25

Use <text> element or @: to switch between c# code and literal text

@for (int i = 1; i <= 10; i++)

```

{
    <b>@i</b>
    if (i % 2 == 0)
    {
        <text> - Even </text>
    }
    else
    {
        <text> - Odd </text>
    }
    <br />
}

```

The above program can be re-written using @: as shown below.

@for (int i = 1; i <= 10; i++)

```

{
    <b>@i</b>
    if (i % 2 == 0)
    {
        @: - Even
    }
    else
    {
        @: - Odd
    }
    <br />
}

```

Output:

1 - Odd
 2 - Even
 3 - Odd
 4 - Even
 5 - Odd
 6 - Even
 7 - Odd
 8 - Even
 9 - Odd
 10 - Even

art 58 - Razor views in mvc continued

Suggested Videos

[Part 55 - What is cross site scripting attack](#)

[Part 56 - How to prevent cross site scripting attack](#)

[Part 57 - Razor views in mvc](#)

In this video, we will discuss razor view syntax. This is continuation to [Part 57](#), so please watch [Part 57](#), before proceeding.

Use @* *@ to comment in razor views

@*This is a comment
in razor views*@

Transition between c# expressions and literal text

```
@{
    int day = 31;
    int month = 12;
    int year = 2013;
}
```

Date is @day-@month-@year

Output:

Date is 31-12-2013

Using explicit code nugget

```
@for (int i = 1; i <= 5; i++)
{
    
}
```

The above code generates the following HTML

```





```

Output:



@ symbol is used as code delimiter in razor views. However, razor is smart enough to recognize the format of internet email address and not to treat the @ symbol as a code delimiter.

This is my email address

kudvenkat@gmail.com

Use @ symbol to escape @

I will meet you @@ office

Output:

I will meet you @ office

Part 59 - Layout view in mvc

Suggested Videos

[Part 56 - How to prevent cross site scripting attack](#)

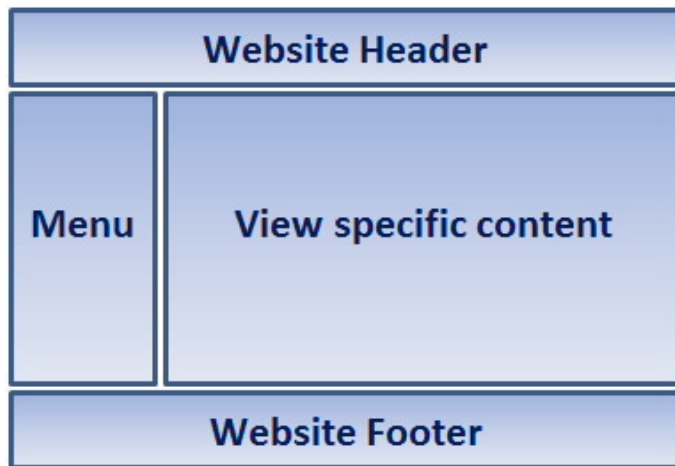
[Part 57 - Razor views in mvc](#)

[Part 58 - Razor views in mvc continued](#)

What is the advantage of using _Layout.cshtml view?

Layout views provide the advantage of maintaining consistent look and feel across all the views in an mvc application. A typical layout view consists of

1. Header
2. Footer
3. Navigation menu
4. View specific content



Rather than having all of these sections, in each and every view, we can define them in a layout view and then inherit that look and feel in all the views. With layout views, maintaining the consistent look and feel across all the views becomes much easier, as we have only one layout file to modify, should there be any change. The change will then be immediately reflected across all the views in entire application.

Let us now create a layout view with

1. Header
2. Footer
3. Navigation menu and
4. a place to plugin view specific content

Step 1: Create an empty asp.net mvc4 application.

Step 2: Right click on "Views" folder and add "Shared" folder.

Step 3: Right click on "Shared" folder and add "**_Layout.cshtml**" view. This is our layout view, where we will define the site wide look and feel. The layout file can have any name, and will have .cshtml file extension. Copy and paste the following html

```

<html>
<head>
  <title>@ViewBag.Title</title>
  @*All the javascript and css files that are required by the
  application can be referenced here so that there is no
  need to reference them in each and every view*@
</head>
<body>
  <table border="1" style="width:800px; font-family:Arial">
    <tr>
      <td colspan="2" style="text-align:center">
        <h3>Website Header</h3>
      </td>
    </tr>
    <tr>
      <td style="width:200px">
        <h3>Menu</h3>
      </td>
      <td style="width:600px">
        @RenderBody()
      </td>
    </tr>
  </table>
  <tr>
    <td colspan="2">
      <h3>Website Footer</h3>
    </td>
  </tr>
  </tr>
  </table>
  
```

```

        <td colspan="2" style="text-align:center; font-size:x-small">
            <h3>Website Footer</h3>
        </td>
    </tr>
</table>
</body>
</html>

```

Points to note:

1. View specific title is retrieved using @ViewBag.Title.
2. View specific content will be plugged-in at the location, where RenderBody() function is called.

Step 4: Let us use the following table tblEmployee, and generate a few views that can be used with our layout view.

Create table tblEmployee

```

(
    Id int identity primary key,
    FirstName nvarchar(50),
    LastName nvarchar(50),
    Salary int
)

```

Insert into tblEmployee values('Tom', 'S', 5000)

Insert into tblEmployee values('Mary', 'P', 8000)

Insert into tblEmployee values('Ben', 'K', 3000)

Step 5: Add ADO.NET entity data model based on the above table. Build the solution, so that Employee model class is compiled.

Step 6: Add a HomeController, with the following settings, so that Index, Details, Create, Edit and Delete views are auto-generated.

1. Controller name - HomeController
2. Template - MVC controller with read/write actions and views, using Entity Framework
3. Model class - Employee
4. Data context class - SampleDbContext
5. Views - Razor

Step 7: Now, we need to make modifications to Index.cshtml view, to use _Layout.cshtml layout view. Copy and paste the following code just below, @model declaration. Notice that, we are storing title in ViewBag object. The layout view is going to retrieve it from viewbag and use it as the title. The next statement, specifies the layout file to use.

```

@{
    ViewBag.Title = "Employee List Page";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

```

At this point navigate to Index view and notice that, it uses the layout file that we have defined.

Step 8: Make the following modifications to layout view. Replace

1. Website Header with Employee Portal
2. <h3>Website Footer</h3> with © 2013 Pragim Technologies
3. <h3>Menu</h3> with @Html.ActionLink("Employee List", "Index")

Save changes and navigate to Index view. Now click on "Edit" link. The page crashes. To fix it, delete "Scripts" section that is at the bottom of the Edit view. Refresh the page. Notice that, we don't have error now, but this view is not using the layout view. To fix it, we need to include the following code, as we did on index view.

```

@{
    ViewBag.Title = "Employee Edit Page";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

```

In our next video, we will discuss - **How to specify layout view setting for all the views at one place**

Part 60 - ViewStart in asp.net mvc

Suggested Videos

[Part 57 - Razor views in mvc](#)

[Part 58 - Razor views in mvc continued](#)

[Part 59 - Layout view in mvc](#)

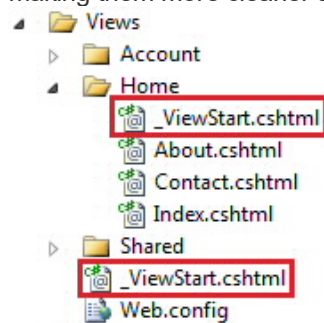
In this video, we will discuss the advantage of using **_ViewStart.cshtml**. Please watch [Part 59](#), before proceeding.

In [Part 59](#), we discussed that, to associate a view with a layout file, we have to set Layout property on each and every view. This violates **DRY** (Don't Repeat Yourself) principle, and has the following disadvantages

1. Redundant code.
2. Maintenance overhead. To use a different layout file, all the views need to be updated.

What is _ViewStart.cshtml file?

ASP.NET MVC 3, has introduced **_ViewStart.cshtml**. Specify the Layout property in this file and place it in the Views folder. All the views will then use the layout file that is specified in **_ViewStart.cshtml**. This eliminates the need to specify Layout property on each and every view, and making them more cleaner and maintainable.



If you want a set of views in a specific folder, to use a different layout file, then you can include another **_ViewStart.cshtml** file in that specific folder.

When I use _ViewStart.cshtml file, can I still set Layout property on individual views?

Yes, if you want to use a layout file that is different from what is specified in **_ViewStart.cshtml**

Where else can I specify a layout file?

Layout file can also be specified in a controller action method or in an action filter.

In Controller Action Method:

Specify which layout to use when returning a view inside a controller action

```
public ActionResult Create()
{
    return View("Create", "_Layout");
}
```

We will discuss action filters in a later video session.

Can we write some logic in "_ViewStart.cshtml" to dynamically specify which layout file to use?

Yes, the following code will change the layout file to use based on the browser type.

If the browser is google chrome,
then **"_Layout.cshtml"** layout file is used
Else
"_DifferentLayout.cshtml" layout file is used

Code in "_ViewStart.cshtml" file

```
@{
```

```
Layout = Request.Browser.IsBrowser("Chrome")
? "~/Views/Shared/_Layout.cshtml" : "~/Views/Shared/_DifferentLayout.cshtml" ;
}
```

All partial views in my application are now using the layout file specified in "_ViewStart.cshtml". How do I prevent these partial views from using a layout file?

Details action method below, returns "_Employee" partial view, and is using the layout file specified in "_ViewStart.cshtml"

```
public ActionResult Details(int id)
{
    Employee employee = db.Employees.Single(e => e.Id == id);
    return View("_Employee", employee);
}
```

To prevent this partial view from using the layout file, specified in "_ViewStart.cshtml", return **"PartialViewResult"** from the controller action method as shown below.

```
public PartialViewResult Details(int id)
{
    Employee employee = db.Employees.Single(e => e.Id == id);
    return PartialView("_Employee", employee);
}
```

What will be the layout file extension, if VB.NET is my programming language?

.vbhtml

In our next video, we will discuss using **named sections in a layout file**.

art 61 - Named sections in layout files in mvc

Suggested Videos

[Part 58 - Razor views in mvc continued](#)

[Part 59 - Layout view in mvc](#)

[Part 60 - ViewStart in asp.net mvc](#)

Let us understand sections in a layout file with an example. Please watch [Parts 59](#) and [60](#) from [mvc tutorial](#) before proceeding.

At the moment on all the views(Index, Create, Edit, Details & Delete), we see the same navigation menu as shown below.

Employee Portal

Employee List

Index

[Create New](#)

FirstName	LastName	Salary	
Tom	S	5000	Edit Details Delete
Mary	P	8000	Edit Details Delete
Ben	K	3000	Edit Details Delete

© 2013 Pragim Technologies

Let us say we want to change the navigation menu dynamically. For example, if I am on the Edit view, then I want the navigation menu to contain links for **List, Details and Delete** views as shown below.

Employee Portal	
List Details Delete	<h2>Edit</h2> <div> Employee <div> <div> <div>FirstName</div> <input type="text" value="Tom"/> </div> <div> <div>LastName</div> <input type="text" value="S"/> </div> <div> <div>Salary</div> <input type="text" value="5000"/> </div> </div> <div>Save</div> </div> <p>Back to List</p>
© 2013 Pragim Technologies	

Here are the steps to achieve this using sections in layout file

Step 1: Define "Menu" section in Edit view. To define a section, use @section followed by the name of the section. The menu section, is going to display List, Details and Delete links.

```
@section Menu
```

```
{
    @Html.ActionLink("List", "Index") <br />
    @Html.ActionLink("Details", "Details", new { id = Model.Id }) <br />
    @Html.ActionLink("Delete", "Delete", new { id = Model.Id })
}
```

Step 2: Specify a location in layout file, where we want the "Menu" section to be rendered.

```
<html>
```

```
<head>...</head>
```

```
<body>
```

```
    <table border="1" style="width:800px; font-family:Arial">
```

```
        <tr>...</tr>
```

```
        <tr>
```

```
            <td style="width:200px">
```

```
                @if(@IsSectionDefined("Menu"))
```

```
                {
```

```
                    @RenderSection("Menu", false)
```

```
                }
```

```
            else
```

```
            {
```

```
                @Html.ActionLink("Employee List", "Index")
```

```
            }
```

```
        </td>
```

```
        <td style="width:600px">
```

```
            @RenderBody()
```

```
        </td>
```

```
    </tr>
```

```
    <tr>...</tr>
```

```
</table>
```

```
</body>
```

```
</html>
```

The above code that is marked in red, is very simple to understand. If you navigate to a view, and if there is a "Menu" section defined in that view, then that content will be injected, else, the default content that is specified in the layout file is used.

For example, Navigate to Edit view. Since "Edit" view has got "Menu" section defined, the content from that section (i.e List, Details and Delete links) will be displayed.

Now navigate to "Delete" view. "Menu" section is not defined in this view, so default content from the layout file (i.e Index action link) will be displayed.

```

else
{
    return View(db.Employees.Where(x => x.Name.StartsWith(search) || search ==null).ToList());
}
}

```

Step 5: Copy and paste the following code in Index.cshtml view.

```

@model IEnumerable<MVCDemo.Models.Employee>

@{
    ViewBag.Title = "Index";
}
<div style="font-family:Arial">
<h2>Employee List</h2>
<p>
    @using (@Html.BeginForm("Index", "Home", FormMethod.Get))
    {
        <b>Search By:</b>
        @Html.RadioButton("searchBy", "Name", true) <text>Name</text>
        @Html.RadioButton("searchBy", "Gender") <text>Gender</text><br />
        @Html.TextBox("search") <input type="submit" value="search" />
    }
</p>
<table border="1">
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Name)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Gender)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Email)
        </th>
        <th>Action</th>
    </tr>
    @if (Model.Count() == 0)
    {
        <tr>
            <td colspan="4">
                No records match search criteria
            </td>
        </tr>
    }
    else
    {
        foreach (var item in Model)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Name)
                </td>

```

```

<td>
    @Html.DisplayFor(modelItem => item.Gender)
</td>
<td>
    @Html.DisplayFor(modelItem => item.Email)
</td>
<td>
    @Html.ActionLink("Edit", "Edit", new { id = item.ID }) |
    @Html.ActionLink("Details", "Details", new { id = item.ID }) |
    @Html.ActionLink("Delete", "Delete", new { id = item.ID })
</td>
</tr>
}
}
</table>
</div>

```

art 63 - Implement paging in asp.net mvc

Suggested Videos

[Part 60 - ViewStart in asp.net mvc](#)

[Part 61 - Named sections in layout files in mvc](#)

[Part 62 - Implementing search functionality in asp.net mvc](#)


In this video we will discuss, **implementing pagination in an asp.net mvc application**. Please watch [Part 62](#), before proceeding.

By the end of this video, the index page should support both **search functionality** and **pagination** as shown below.


Search By: ☒ Name ☐ Gender

Name	Gender	Email	Action
Paul Sensit	Male	Paul.Sensit@test.com	Edit Details Delete
Mike Tan	Male	Mike.Tan@pragimtech.com	Edit Details Delete
Mark Waugh	Male	Mark.Waugh@pragimtech.com	Edit Details Delete

Step 1: Install **PagedList.Mvc** using NuGet package manager. PagedList.Mvc is dependent on PagedList. Installing PagedList.Mvc will automatically install PagedList package as well.



PagedList
PagedList makes it easier for .Net developers to write paging code. It allows you to take any IEnumerable(T) and by specifying the page size a...



PagedList.Mvc
Asp.Net MVC HtmlHelper method for generating paging control for use with PagedList library.

Step 2: Include the following using statements in HomeController.cs file
using PagedList.Mvc;


```
using PagedList;
```

Modify the **Index()** action method as shown below. Notice that we are passing **page** parameter to this function. This parameter is used for specifying the page number. This parameter can be null, and that's the reason we have chosen a **nullable integer**. We convert the list, to a paged list, using **ToPagedList()**. Also, notice that, we are using null-coalescing operator. If the "page" parameter is null, then 1 is passed as the page number, else, the value contained in the "page" parameter is used as the page number.

```
public ActionResult Index(string searchBy, string search, int? page)
{
    if (searchBy == "Gender")
    {
        return View(db.Employees.Where(x => x.Gender == search || search
==null).ToList().ToPagedList(page ?? 1, 3));
    }
    else
    {
        return View(db.Employees.Where(x => x.Name.StartsWith(search) || search
==null).ToList().ToPagedList(page ?? 1, 3));
    }
}
```

Step 3: Make the following modifications to Index.cshtml view

a) Include the following 2 using statements on the view.

```
@using PagedList.Mvc;
```

```
@using PagedList;
```

b) The model for the view should be **IPagedList<Employee>**.

```
@model IPagedList<MVCDemo.Models.Employee>
```

c) Since, we have changed the model of the view, from **IEnumerable<MVCDemo.Models.Employee>** to **IPagedList<MVCDemo.Models.Employee>**, change the section that displays table headings as shown below.

```
<tr>
    <th>
        @Html.DisplayNameFor(model => model.First().Name)
    </th>
    <th>
        @Html.DisplayNameFor(model => model.First().Gender)
    </th>
    <th>
        @Html.DisplayNameFor(model => model.First().Email)
    </th>
    <th>Action</th>
</tr>
```

d) Finally to display page numbers for paging

```
@Html.PagedListPager(Model, page => Url.Action("Index", new { page, searchBy =
Request.QueryString["searchBy"], search = Request.QueryString["search"] })))
```

e) If you want to display the pager, only if there are more than 1 page

```
@Html.PagedListPager(Model, page => Url.Action("Index", new { page, searchBy =
Request.QueryString["searchBy"], search = Request.QueryString["search"]
}), new PagedListRenderOptions() { Display = PagedListDisplayMode.IfNeeded })
```

f) If you want to display, the current active page and the total number of pages

```
@Html.PagedListPager(Model, page => Url.Action("Index", new { page, searchBy =
Request.QueryString["searchBy"], search = Request.QueryString["search"]
}), new PagedListRenderOptions() { Display = PagedListDisplayMode.IfNeeded,
DisplayPageCountAndCurrentLocation = true })
```

g) If you want to display the number of rows displayed, of the total number of rows available.

```
@Html.PagedListPager(Model, page => Url.Action("Index", new { page, searchBy =
Request.QueryString["searchBy"], search = Request.QueryString["search"]
}), new PagedListRenderOptions() { Display = PagedListDisplayMode.IfNeeded,
DisplayItemSliceAndTotal = true })
```

Part 64 - Implement sorting in asp.net mvc

Suggested Videos

[Part 61 - Named sections in layout files in mvc](#)

[Part 62 - Implementing search functionality in asp.net mvc](#)

[Part 63 - Implement paging in asp.net mvc](#)

In this video, we will discuss, **implementing sort functionality in an asp.net mvc application**. We will be working with the example that we started in [Part 62](#). So, please watch [Parts 62](#) and [63](#) before proceeding with this video.

We want to support bi-directional sorting by Name and Gender columns. Here's the requirement

1. **Name & Gender** columns must be **click-able hyperlinks**
2. Clicking on the column headers should sort the data. If the data is not already sorted by the column on which you have clicked, the data should be sorted in ascending order. Clicking again on the same column should sort the data in descending order.
3. By default, the data should be sorted by **"Name"** in ascending order.

By the end of this video, the output should be as shown below. Notice that **"Name"** and **"Gender"** columns are rendered as hyperlinks, which the user can click to sort data.

Search By: ☒ Name ☐ Gender

Name	Gender	Email	Action
James Histo	Male	James.Histo@test.com	Edit Details Delete
Mark Waugh	Male	Mark.Waugh@pragimtech.com	Edit Details Delete
Mary Jane	Female	Mary.Jane@test.com	Edit Details Delete

Page 1 of 3. [1](#) [2](#) [3](#) [»](#)

Step 1: Modify the **"Index()"** action method in **HomeController** as shown below.

```
public ActionResult Index(string searchBy, string search, int? page, string sortBy)
{
    ViewBag.NameSort = String.IsNullOrEmpty(sortBy) ? "Name desc" : "";
    ViewBag.GenderSort = sortBy == "Gender" ? "Gender desc" : "Gender";

    var employees = db.Employees.AsQueryable();

    if (searchBy == "Gender")
    {
        employees = employees.Where(x => x.Gender == search || search == null);
    }
    else
    {
        employees = employees.Where(x => x.Name.StartsWith(search) || search == null);
    }

    switch (sortBy)
    {
        case "Name desc":
            employees = employees.OrderByDescending(x => x.Name);
            break;
        case "Gender desc":
            employees = employees.OrderByDescending(x => x.Gender);
            break;
    }
}
```

```

        case "Gender":
            employees = employees.OrderBy(x => x.Gender);
            break;
        default:
            employees = employees.OrderBy(x => x.Name);
            break;
    }

    return View(employees.ToPagedList(page ?? 1, 3));
}

```

Step 2: Modify the code in **Index.cshtml** view as shown below. Please pay attention to the code highlighted with **Grey** colour.

```

@using PagedList;
@using PagedList.Mvc;

@model PagedList.IPagedList<MVCDemo.Models.Employee>

@{
    ViewBag.Title = "Index";
}
<link href="~/Content/PagedList.css" rel="stylesheet" type="text/css" />
<div style="font-family:Arial">
<h2>Employee List</h2>
<p>
    @using (@Html.BeginForm("Index", "Home", FormMethod.Get))
    {
        <b>Search By:</b>
        @Html.RadioButton("searchBy", "Name", true) <text>Name</text>
        @Html.RadioButton("searchBy", "Gender") <text>Gender</text><br />
        @Html.TextBox("search") <input type="submit" value="search" />
    }
</p>
<table border="1">
    <tr>
        <th>
            @Html.ActionLink("Name", "Index", new { sortBy = ViewBag.NameSort, searchBy =
Request["searchBy"], search = Request["search"] })
        </th>
        <th>
            @Html.ActionLink("Gender", "Index", new { sortBy = ViewBag.GenderSort, searchBy =
Request["searchBy"], search = Request["search"] })
        </th>
        <th>
            @Html.DisplayNameFor(model => model.First().Email)
        </th>
        <th>Action</th>
    </tr>
    @if (Model.Count() == 0)
    {
        <tr>
            <td colspan="4">
                No records match search criteria
            </td>
        </tr>
    }
    else
    {
        foreach (var item in Model)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Name)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Gender)
                </td>
                <td>

```

```

        @Html.DisplayFor(modelItem => item.Email)
    </td>
    <td>
        @Html.ActionLink("Edit", "Edit", new { id = item.ID }) |
        @Html.ActionLink("Details", "Details", new { id = item.ID }) |
        @Html.ActionLink("Delete", "Delete", new { id = item.ID })
    </td>
    </tr>
}
}
</table>
@Html.PagedListPager(Model, page => Url.Action("Index", new { page, searchBy =
Request.QueryString["searchBy"], search = Request.QueryString["search"], sortBy =
Request.QueryString["sortBy"] }, new PagedListRenderOptions() { Display = PagedListDisplayMode.IfNeeded })
</div>

```

Part 65 - Deleting multiple rows in mvc

Suggested Videos

[Part 62 - Implementing search functionality](#)

[Part 63 - Implement paging](#)

[Part 64 - Implement sorting](#)

In this video, we will discuss deleting multiple rows in an asp.net mvc application.

We will be using table **tblEmployee** for this demo. Please refer to [Part 62](#), if you need SQL script to create and populate this table.

We want to provide a checkbox next to every row, to enable users to select multiple rows for deletion as shown below.

Select	Name	Gender	Email
<input type="checkbox"/>	Mary Jane	Female	Mary.Jane@test.com
<input type="checkbox"/>	Paul Sensit	Male	Paul.Sensit@test.com
<input type="checkbox"/>	Mark Waugh	Male	Mark.Waugh@pragimtech.com
<input type="checkbox"/>	Nancy Neird	Female	Nancy.Neird@pragimtech.com

Delete selected employees

Step 1: Create an empty asp.net mvc 4 application.

Step 2: Generate ADO.NET entity data model from database using table tblEmployee. Change the entity name from tblEmployee to Employee. Save changes and build the application.

Step 3: Add HomeController with the following settings.

a) Controller name = HomeController

b) Template = Empty MVC controller

Step 4: Add "Shared" folder under "Views", if it is not already present.

Add "EditorTemplates" folder, under "Shared" folder. Right click on "EditorTemplates" folder and "Employee.cshtml" view with the following settings

View name = Employee

View engine = Razor

Create a strongly-typed view = checked

Model class = Employee (MVCDemo.Models)

Scaffold Template = Empty

and finally click "Add"

Step 5: Copy and paste the following code in Employee.cshtml view

```
@model MVCDemo.Models.Employee
```

```
<tr>
    <td>
```

```

<input type="checkbox" name="employeeIdsToDelete" id="employeeIdsToDelete" value="@Model.ID"
/>
</td>
<td>
    @Model.Name
</td>
<td>
    @Model.Gender
</td>
<td>
    @Model.Email
</td>
</tr>

```

Step 6: Copy and paste the following code in HomeController.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using MVCDemo.Models;

namespace MVCTest.Controllers
{
    public class HomeController : Controller
    {
        private SampleDBContext db = new SampleDBContext();

        public ActionResult Index()
        {
            return View(db.Employees.ToList());
        }

        [HttpPost]
        public ActionResult Delete(IEnumerable<int> employeeIdsToDelete)
        {
            db.Employees.Where(x =>
employeeIdsToDelete.Contains(x.ID)).ToList().ForEach(db.Employees.DeleteObject);
            db.SaveChanges();
            return RedirectToAction("Index");
        }
    }
}

```

Setp 7: Right click on the Index() action and add "Index" view with the following settings.

View name = Index

View engine = Razor

Create a strongly-typed view = checked

Model class = Employee (MVCDemo.Models)

Scaffold Template = Empty

and finally click **"Add"**

Setp 8: Copy and paste the following code in Index.cshtml view

```

@model IEnumerable<MVCDemo.Models.Employee>

<div style="font-family:Arial">
<h2>Employee List</h2>

@using (Html.BeginForm("Delete", "Home", FormMethod.Post))
{
    <table border="1">
        <thead>
            <tr>
                <th>
                    Select
                </th>
                <th>
                    Name

```

```

</th>
<th>
    Gender
</th>
<th>
    Email
</th>
</tr>
</thead>
<tbody>
    @Html.EditorForModel()
</tbody>
</table>
<input type="submit" value="Delete selected employees" />
}
</div>

```

In our next video, we will discuss providing a **"SELECT ALL"** checkbox to select and de-select all rows.

Part 66 - Check uncheck all checkboxes with another single checkbox using jquery

Suggested Videos

[Part 63 - Implement paging](#)

[Part 64 - Implement sorting](#)

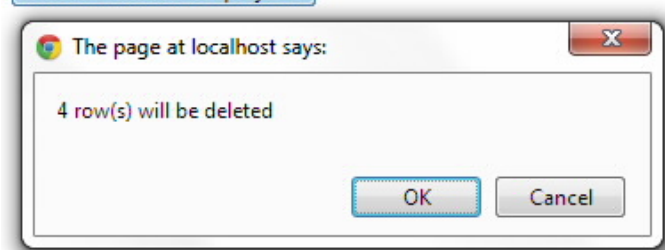
[Part 65 - Deleting multiple rows](#)

In this video, we will discuss, how to check/uncheck all checkboxes with another single checkbox using jquery in an asp.net mvc application. Please watch [Part 65](#), before proceeding.

We want the output as shown below

<input checked="" type="checkbox"/>	Name	Gender	Email
<input checked="" type="checkbox"/>	Sara Nani	Female	Sara.Nani@test.com
<input checked="" type="checkbox"/>	James Histo	Male	James.Histo@test.com
<input checked="" type="checkbox"/>	Mary Jane	Female	Mary.Jane@test.com
<input checked="" type="checkbox"/>	Paul Sensit	Male	Paul.Sensit@test.com

Delete selected employees



Here is the requirement

1. When the checkbox in the header is selected, all the checkboxes in the respective rows should be selected. If we deselect any of the checkbox, then the header checkbox should be automatically deselected.
2. When the checkbox in the header is deselected, all the checkboxes in the respective rows should be deselected as well. If we start to select each checkbox in the datarows, and upon selecting checkboxes in all the datarows, the checkbox in the header should be selected.
3. We also need a client side confirmation, on the number of rows to be deleted. The selected

rows should be deleted from the database table, only when OK button is clicked. If cancel button is clicked, the form should not be posted to the server, and no rows should be deleted.

Here's the required jQuery script. Please paste this in the "Index.cshtml" view.

```
<script src="../../Scripts/jquery-1.7.1.min.js" type="text/javascript"></script>
<script type="text/javascript" language="javascript">
    $(function () {
        $("#checkAll").click(function () {
            $("input[name='employeeIdsToDelete']").attr("checked", this.checked);

            $("input[name='employeeIdsToDelete']").click(function () {
                if ($("input[name='employeeIdsToDelete']").length ==
                    $("input[name='employeeIdsToDelete']:checked").length) {
                    $("#checkAll").attr("checked", "checked");
                }
                else {
                    $("#checkAll").removeAttr("checked");
                }
            });
        });

        $("#btnSubmit").click(function () {
            var count = $("input[name='employeeIdsToDelete']:checked").length;
            if (count == 0) {
                alert("No rows selected to delete");
                return false;
            }
            else {
                return confirm(count + " row(s) will be deleted");
            }
        });
    });
</script>
```

jQuery file can also be referenced from the following website

jQuery - <http://code.jquery.com/jquery-latest.min.js>

Google - <http://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js>

Microsoft - <http://ajax.microsoft.com/ajax/jQuery/jquery-1.9.1.min.js>

rt 67 - Action selectors in mvc

Suggested Videos

[Part 64 - Implement sorting](#)

[Part 65 - Deleting multiple rows](#)

[Part 66 - Check uncheck all checkboxes using jquery](#)

In this video, we will discuss **action selectors** in asp.net mvc

Actions are public methods in an mvc controller that responds to an URL request. You can **control or influence which action method gets invoked using action selectors** in mvc. Action selectors are attributes that can be applied to an action method in a controller.

ActionName selector: This action selector is used when you want to invoke an action method with a different name, than what is already given to the action method.

For example, the following URL request would invoke Index() action method in HomeController
/Home/Index

public class HomeController : Controller


```
{
    public string Index()
    {
        return "Index action method invoked";
    }
}
```

If you want to invoke **Index()** action method, with the following URL
/Home/List

Then decorate the action method with **ActionName** attribute as shown below.

```
public class HomeController : Controller
{
    [ActionName("List")]
    public string Index()
    {
        return "Index action method invoked";
    }
}
```

Now, if you navigate to **/Home/Index**, you will get an error - **The resource cannot be found.**

At the moment, the **Index()** action method is returning a string, but if it returns a view, should the view be named - Index or List.?

```
[ActionName("List")]
public ActionResult Index()
{
    return View();
}
```

List should be the view name. If for some reason, you want to use **"Index"** as the view name, then modify the controller action method as shown below.

```
[ActionName("List")]
public ActionResult Index()
{
    return View("Index");
}
```

AcceptVerbs selector: Use this selector, when you want to control, the invocation of an action method based on the request type. In the example below, the "Edit" method that is decorated with GET acceptverb responds to the GET request, where as the other "Edit" method responds to POST request. The default is GET. So, if you don't decorate an action method with any accept verb, then, by default, the method responds to GET request.

```
public class HomeController : Controller
{
    [AcceptVerbs(HttpVerbs.Get)]
    public ActionResult Edit(int id)
    {
        Employee employee = GetEmployeeFromDB(id);
        return View(employee);
    }

    [AcceptVerbs(HttpVerbs.Post)]
    public ActionResult Save(Employee employee)
    {
        if (ModelState.IsValid)
        {
            // Save employee to the database
            return RedirectToAction("Index");
        }
        return View(employee);
    }
}
```

HttpGet and **HttpPost** attributes can be used as shown below. This is an alternative to using **AcceptVerbs** attribute.

```
public class HomeController : Controller
{
    [HttpGet]
```



```

public ActionResult Edit(int id)
{
    Employee employee = GetEmployeeFromDB(id);
    return View(employee);
}

[HttpPost]
public ActionResult Save(Employee employee)
{
    if (ModelState.IsValid)
    {
        // Save employee to the database
        return RedirectToAction("Index");
    }
    return View(employee);
}
}

```

Part 68 - What is the use of NonAction attribute in mvc

Suggested Videos

[Part 65 - Deleting multiple rows](#)

[Part 66 - Check uncheck all checkboxes using jquery](#)

[Part 67 - Action selectors](#)

In this video, we will discuss the use of **NonAction** attribute in asp.net mvc

The following questions could be asked in an interview

What is the use of NonAction attribute in MVC?

OR

How do you restrict access to public methods in a controller?

An action method is a public method in a controller that can be invoked using a URL. So, by default, if you have any public method in a controller then it can be invoked using a URL request. To restrict access to public methods in a controller, NonAction attribute can be used. Let's understand this with an example.

We have 2 public methods in HomeController, Method1() and Method2().

Method1 can be invoked using URL **/Home/Method1**

Method2 can be invoked using URL **/Home/Method2**

```

public class HomeController : Controller
{
    public string Method1()
    {
        return "<h1>Method 1 Invoked</h1>";
    }

    public string Method2()
    {
        return "<h1>Method 2 Invoked</h1>";
    }
}

```

Let's say **Method2()** is for doing some internal work, and we don't want it to be invoked using a URL request. To achieve this, decorate **Method2()** with **NonAction** attribute.

```

[NonAction]
public string Method2()
{
    return "<h1>Method 2 Invoked</h1>";
}

```

Now, if you navigate to URL [/Home/Method2](#), you will get an error - **The resource cannot be found.**

Another way to restrict access to methods in a controller, is by making them private.

```
private string Method2()
{
    return "<h1>Method 2 Invoked</h1>";
}
```

In general, it's a bad design to have a public method in a controller that is not an action method. If you have any such method for performing business calculations, it should be somewhere in the model and not in the controller.

However, if for some reason, if you want to have public methods in a controller and you don't want to treat them as actions, then use `NonAction` attribute.

Part 69 - Action filters in mvc

Suggested Videos

[Part 66 - Check uncheck all checkboxes using jquery](#)

[Part 67 - Action selectors](#)

[Part 68 - What is the use of NonAction attribute](#)

What are action filters in asp.net mvc?

Action filters are attributes that can be applied either on a controller action method or on a controller. When applied at the controller level, they are applicable for all actions within that controller. Action filters allow us to add, pre and post processing logic to an action method. This means, they allow us to modify the way in which an action is executed.

Name a few action filters in mvc?

Authorize
ChildActionOnly
HandleError
OutputCache
RequireHttps
ValidateInput
ValidateAntiForgeryToken

We will discuss each of these action filters in detail in our upcoming videos.

Can you create a custom action filter in mvc?

Yes, we will discuss this in a later video session.

Part 70 - Authorize and AllowAnonymous action filters in mvc

Suggested Videos

[Part 67 - Action selectors](#)

[Part 68 - What is the use of NonAction attribute](#)

[Part 69 - Action filters](#)

In this video, we will discuss **Authorize** and **AllowAnonymous** action filters in mvc.

In ASP.NET MVC, by default, all the controller action methods are accessible to both **anonymous** and **authenticated** users. If you want action methods, to be available only for authenticated and authorised users, then use `Authorize` attribute. Let us understand "**Authorize**" and "**AllowAnonymous**" action filters with an example.

1. Create a blank asp.net mvc4 application. Name your application MVCDemo.
2. Right click on the **"Controllers"** folder and add HomeController. Copy and paste the following code.

```
public class HomeController : Controller
{
    public ActionResult NonSecureMethod()
    {
        return View();
    }

    public ActionResult SecureMethod()
    {
        return View();
    }
}
```
3. Right click on **NonSecureMethod()** and add a view with name = **NonSecureMethod**. Similarly add a view with name = **SecureMethod**.

4. Associate MVCDemo project with IIS.

a) Right click on the project name in **"solution explorer"** and select **"Properties"**

b) Click on **"Web"** tab

c) Select **"Use Local IIS Web Server"**. In the **"Project Url"** textbox, type - <http://localhost/MVCDemo>

d) Click **"Create Virtual Directory"** button

5. Open IIS. Expand **"Sites"** and then **"Default Web Site"** and select **"MVCDemo"**. Double click on **"Authentication"** icon. Enable **"Anonymous Authentication"** and **"Windows Authentication"**, if they are not already enabled.

6. At this point, you will be able to access, both **"SecureMethod"** and **"NonSecureMethod"**, by visiting the following URLs.

<http://localhost/MVCDemo/Home/SecureMethod>

<http://localhost/MVCDemo/Home/NonSecureMethod>

7. If you want **"SecureMethod"** to be available only for authenticated users, then decorate it with **"Authorize"** attribute.

[Authorize]

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

8. Now, if you navigate to **"http://localhost/MVCDemo/Home/SecureMethod"**, then you will be prompted for your windows credentials. If you don't provide valid windows credentials or if you click cancel, you will get an error - **401 - Unauthorized: Access is denied due to invalid credentials. You do not have permission to view this directory or page using the credentials that you supplied.** You should be able to access **"NonSecureMethod"**

9. Now remove the [Authorize] attribute from SecureMethod(), and apply it on the HomeController.

[Authorize]

```
public class HomeController : Controller
{
    public ActionResult NonSecureMethod()
    {
        return View();
    }

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

At this point, **"Authorize"** attribute is applicable for all action methods in the HomeController. So,

only authenticated users will be able to access SecureMethod() and NonSecureMethod().

10. To allow anonymous access to NonSecureMethod(), apply [AllowAnonymous] attribute. AllowAnonymous attribute is used to skip authorization enforced by Authorize attribute.

[AllowAnonymous]

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

Part 71 - childactiononly attribute in mvc

Suggested Videos

[Part 68 - What is the use of NonAction attribute](#)

[Part 69 - Action filters](#)

[Part 70 - Authorize and AllowAnonymous action filters](#)

In this video, we will discuss **childactiononly** attribute in asp.net mvc. Let us understand this with an example.

Step 1: Create a blank asp.net mvc 4 application

Step 2: Add HomeController. Copy and paste the following code.

```
public class HomeController : Controller
{
    // Public action method that can be invoked using a URL request
    public ActionResult Index()
    {
        return View();
    }

    // This method is accessible only by a child request. A runtime
    // exception will be thrown if a URL request is made to this method
    [ChildActionOnly]
    public ActionResult Countries(List<String> countryData)
    {
        return View(countryData);
    }
}
```

Step 3: Right click on the "**Countries()**" action method and add "**Countries**" view. This view will render the given list of strings as an un-ordered list.

```
@model List<string>
@foreach (string country in Model)
{
    <ul>
        <li>
            <b>
                @country
            </b>
        </li>
    </ul>
}
```

Step 4: Right click on the **"Index()"** action method and add **"Index"** view. Copy and paste the following code. Notice that, to invoke childaction, we are using Action() HTML Helper.

```
<h2>Countries List</h2>
@Html.Action("Countries", new { countryData = new List<string>() { "US", "UK", "India" } })
```

Please Note: Child actions can also be invoked using **"RenderAction()"** HTML helper as shown below.

```
@{
    Html.RenderAction("Countries", new { countryData = new List<string>() { "US", "UK", "India" } });
}
```

Points to remember about "ChildActionOnly" attribute

1. Any action method that is decorated with [ChildActionOnly] attribute is a child action method.
2. Child action methods will not respond to URL requests. If an attempt is made, a runtime error will be thrown stating - **Child action is accessible only by a child request.**
3. Child action methods can be invoked by making child request from a view using **"Action()"** and **"RenderAction()"** html helpers.
4. An action method doesn't need to have [ChildActionOnly] attribute to be used as a child action, but use this attribute to prevent if you want to prevent the action method from being invoked as a result of a user request.
5. Child actions are typically associated with partial views, although this is not compulsory.
6. Child action methods are different from NonAction methods, in that NonAction methods cannot be invoked using Action() or RenderAction() helpers. We discussed NonAction methods in [Part 70 of ASP.NET MVC tutorial](#) series.
7. Using child action methods, it is possible to cache portions of a view. This is the main advantage of child action methods. We will cover this when we discuss [OutputCache] attribute.

Part 72 - HandleError attribute in mvc

Suggested Videos

[Part 69 - Action filters](#)

[Part 70 - Authorize and AllowAnonymous action filters](#)

[Part 71 - ChildActionOnly attribute](#)

In this video, we will discuss **HandleError** attribute in asp.net mvc. HandleErrorAttribute is used to display friendly error pages to end user when there is an unhandled exception. Let us understand this with an example.

Step 1: Create a blank asp.net mvc 4 application.

Step 2: Add a HomeController. Copy and paste the following code.

```
public ActionResult Index()
{
    throw new Exception("Something went wrong");
}
```

Notice that, the **Index()** action method throws an exception. As this exception is not handled, when you run the application, you will get the default **"yellow screen of death"** which does not make sense to the end user.

Server Error in '/MVCDemo' Application.

Something went wrong

Description: An unhandled exception occurred during the execution of the current web request. Please refer to the error details for more information.

Exception Details: System.Exception: Something went wrong

Source Error:

```

Line 12:         public ActionResult Index()
Line 13:         {
Line 14:             throw new Exception("Something went wrong");
Line 15:         }
Line 16:     }

```

Now, let us understand replacing this yellow screen of death, with a friendly error page.

Step 3: Enable custom errors in web.config file, that is present in the root directory of your mvc application. "customErrors" element must be nested under "<system.web>". For detailed explanation on MODE attribute, please watch [Part 71 of ASP.NET Tutorial](#).

```
<customErrors mode="On">
</customErrors>
```

Step 4: Add "Shared" folder under "Views" folder. Add **Error.cshtml** view inside this folder. Paste the following HTML in Error.cshtml view.

```
<h2>An unknown problem has occurred, please contact Admin</h2>
```

Run the application, and notice that, you are redirected to the friendly "Error" view, instead of the generic "Yellow screen of death".

We did not apply HandleError attribute anywhere. So how did all this work?

HandleErrorAttribute is added to the GlobalFilters collection in global.asax. When a filter is added to the GlobalFilters collection, then it is applicable for all controllers and their action methods in the entire application.

Right click on "RegisterGlobalFilters()" method in Global.asax, and select "Go To Definition" and you can find the code that adds "HandleErrorAttribute" to GlobalFilterCollection.

```

public static void RegisterGlobalFilters(GlobalFilterCollection filters)
{
    filters.Add(new HandleErrorAttribute());
}

```

Is the friendly error page displayed for HTTP status code 404?

No, but there is a way to display the friendly error page.

In the HomeController, we do not have List() action method. So, if a user navigates to **/Home/List**, we get an error - **The resource cannot be found. HTTP 404.**

To display a friendly error page in this case

Step 1: Add "ErrorController" to controllers folder. Copy and paste the following code.

```

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

```

Step 2: Right click on "Shared" folder and add "NotFound.cshtml" view. Copy and paste the following code.

```
<h2>Please check the URL. The page you are looking for cannot be found</h2>
```

Step 3: Change "customErrors" element in web.config as shown below.

```
<customErrors mode="On">
```

```
<error statusCode="404" redirect="~/Error/NotFound"/>
</customErrors>
```

Part 73 - OutputCache attribute in mvc

Suggested Videos

[Part 70 - Authorize and AllowAnonymous action filters](#)

[Part 71 - ChildActionOnly attribute](#)

[Part 72 - HandleError attribute](#)

In this video, we will discuss **OutputCache** attribute and partial caching in asp.net mvc. **OutputCacheAttribute** is used to cache the content returned by a controller action method, so that, the same content does not need to be generated each and every time the same controller action is invoked. Let us understand this with an example.

We will be using table **tblEmployee** for this demo. Please refer to [Part 62](#) of [MVC Tutorial](#), if you need SQL script to create and populate this table.

Step 1: Add ado.net entity data model based on table **tblEmployee**. We have discussed doing this several times, in previous sessions of this video series.

Step 2: Add HomeController with the following settings.

- a) Controller name = HomeController
- b) Template = MVC controller with read/write actions and views, using Entity Framework
- c) Model class = Employee
- d) Data context class = SampleDbContext
- e) Views = Razor

Step 3: Modify the **Index()** action method in **HomeController** as shown below. Notice that, we are using **OutputCache** attribute to cache the content returned by **Index()** action method for 10 seconds.

```
[OutputCache(Duration = 10)]
public ActionResult Index()
{
    System.Threading.Thread.Sleep(3000);
    return View(db.Employees.ToList());
}
```

Step 4: Modify code in **"Index.cshtml"** view as shown below. The modified code is highlighted in Yellow.

```
@model IEnumerable<MVCDemo.Models.Employee>
<div style="font-family:Arial">
<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table border="1">
<tr>
    <td colspan="4">
        <b>Employee List retrieved @@ @DateTime.Now.ToString()</b>
    </td>
</tr>
<tr>
    <th>
        @Html.DisplayNameFor(model => model.Name)
    </th>
    <th>
        @Html.DisplayNameFor(model => model.Gender)
    </th>
    <th>
        @Html.DisplayNameFor(model => model.Email)
    </th>
```

```

        <th></th>
    </tr>

    @foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Name)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Gender)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Email)
            </td>
            <td>
                @Html.ActionLink("Edit", "Edit", new { id=item.ID }) |
                @Html.ActionLink("Details", "Details", new { id=item.ID }) |
                @Html.ActionLink("Delete", "Delete", new { id=item.ID })
            </td>
        </tr>
    }
</table>
</div>

```

When you navigate to **/Home/Index**, the view output is cached for 10 seconds. If you refresh the view, within 10 seconds you will get a cached response. After 10 seconds, the cache expires, code is executed again and the output is cached for another 10 seconds.

Caching specific portion of a view using **ChildActionOnly** attribute:

Step 1: Remove **OutputCache** attribute and the line which calls **Thread.Sleep()**, from the **Index()** action method in **HomeController**. After the changes, the **Index()** action method should be as shown below.

```

public ActionResult Index()
{
    return View(db.Employees.ToList());
}

```

Step 2: Add **GetEmployeeCount()** action method to **HomeController**. Notice that, this method is decorated with **OutputCache** and **ChildActionOnly** attributes.

```

// Child actions can be used to implement partial caching,
// although not necessary. In this case, even if the ChildActionOnly
// attribute is removed, a portion of the view will be cached as expected
[ChildActionOnly]
[OutputCache(Duration = 10)]
public string GetEmployeeCount()
{
    return "Employee Count = " + db.Employees.Count().ToString() + "@"
    + DateTime.Now.ToString();
}

```

Step 3: Copy and paste the following code, just below the closing table tag in **Index.cshtml** view.

```

<br /><br />
<b> @Html.Action("GetEmployeeCount") </b>

```

Navigate to **/Home/Index**. Notice that, everytime you refresh the page, the time in the section of the page that displays employee list changes, but not the time, that displays the employee count. This proves that, only a portion of the view, is cached.

art 74 - CacheProfiles in mvc

Suggested Videos

[Part 71 - ChildActionOnly attribute](#)

[Part 72 - HandleError attribute](#)

Part 73 - OutputCache attribute

In this video, we will discuss creating **CacheProfiles**. We will be using the example that we started in [Part 73](#). Please watch [Part 73](#) before proceeding.

To cache the data returned by **Index()** action method, for 60 seconds, we would use **[OutputCache]** attribute as shown below.

```
[OutputCache(Duration=60)]
public ActionResult Index()
{
    return View(db.Employees.ToList());
}
```

In the example above, the **OutputCache** settings are specified in code. The disadvantage of this approach is that

1. If you have to apply the same cache settings, to several methods, then the code needs to be duplicated.
2. Later, if we have to change these cache settings, then we need to change them at several places. Maintaining the code becomes complicated. Also, changing the application code requires build and re-deployment.

To overcome these disadvantages, cache settings can be specified in web.config file using cache profiles.

Step 1: Specify cache settings in web.config using cache profiles

```
<system.web>
  <caching>
    <outputCacheSettings>
      <outputCacheProfiles>
        <clear/>
        <add name="1MinuteCache" duration="60" varyByParam="none"/>
      </outputCacheProfiles>
    </outputCacheSettings>
  </caching>
</system.web>
```

Step 2: Reference the cache profile in application code

```
[OutputCache(CacheProfile = "1MinuteCache")]
public ActionResult Index()
{
    return View(db.Employees.ToList());
}
```

The cache settings are now read from one central location i.e from the web.config file. The advantage of using cache profiles is that

1. You have one place to change the cache settings. Maintainability is much easier.
2. Since the changes are done in web.config, we need not build and redeploy the application.

Using Cache profiles with child action methods

```
[ChildActionOnly]
[OutputCache(CacheProfile = "1MinuteCache")]
public string GetEmployeeCount()
{
    return "Employee Count = " + db.Employees.Count().ToString()
        + "@ " + DateTime.Now.ToString();
}
```

When Cache profiles are used with child action methods, you will get an error - **Duration must be a positive number**.

There could be several ways to make cache profiles work with child action methods. The following is the approach, that I am aware of. Please feel free to leave a comment, if you know of a better way of doing this.

Create a **custom OutputCache attribute**, that loads the cache settings from the cache profile in web.config.

Step 1: Right click on the project name in solution explorer, and add a folder with name = Common

Setp 2: Right click on "**Common**" folder and add a class file, with name = PartialCacheAttribute.cs

Step 3: Copy and paste the following code. Notice that, I have named the custom **OutputCache** attribute as **PartialCacheAttribute**.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Configuration;

namespace MVCDemo.Common
{
    public class PartialCacheAttribute : OutputCacheAttribute
    {
        public PartialCacheAttribute(string cacheProfileName)
        {
            OutputCacheSettingsSection cacheSettings =
            (OutputCacheSettingsSection)WebConfigurationManager.GetSection("system.web/caching/outputCacheSettings");
            OutputCacheProfile cacheProfile = cacheSettings.OutputCacheProfiles[cacheProfileName];
            Duration = cacheProfile.Duration;
            VaryByParam = cacheProfile.VaryByParam;
            VaryByCustom = cacheProfile.VaryByCustom;
        }
    }
}
```

Step 4: Use PartialCacheAttribute on the child action method, and pass it the name of the cache profile in web.config. Please note that, PartialCacheAttribute is in MVCDemo.Common namespace.

```
[ChildActionOnly]
[PartialCache("1MinuteCache")]
public string GetEmployeeCount()
{
    return "Employee Count = " + db.Employees.Count().ToString()
        + "@" + DateTime.Now.ToString();
}
```

Part 75 - RequireHttps attribute in mvc

Suggested Videos

[Part 72 - HandleError attribute](#)

[Part 73 - OutputCache attribute](#)

[Part 74 - CacheProfiles](#)

In this video, we will discuss **RequireHttps** attribute.

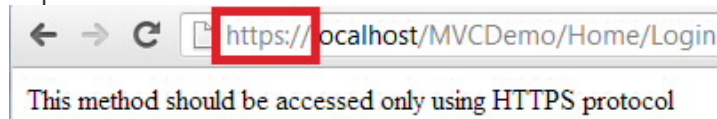
[RequireHttps] attribute forces an unsecured HTTP request to be re-sent over HTTPS. Let's understand [RequireHttps] attribute with an example.

Step 1: Create an asp.net mvc4 application using "Empty" template

Step 2: Add a HomeController. Copy and paste the Login() action method in the HomeController.

```
[RequireHttps]
public string Login()
{
    return "This method should be accessed only using HTTPS protocol";
}
```

Step 3: Try to navigate to <http://localhost/MVCDemo/Home/Login>. Notice that you are automatically redirected to <https://localhost/MVCDemo/Home/Login>. So, [RequireHttps] attribute, forces an HTTP request to be re-sent over HTTPS.



RequireHttps attribute can be applied on a controller as well. In this case, it is applicable for all action methods with in that controller.

Sensitive data such as login credentials, credit card information etc, must always be transmitted using HTTPS. Information transmitted over https is encrypted.

Part 76 - ValidateInput attribute in mvc

Suggested Videos

[Part 73 - OutputCache attribute](#)

[Part 74 - CacheProfiles](#)

[Part 75 - RequireHttps attribute](#)

In this video, we will discuss **ValidateInput** attribute. This attribute is used to enable or disable request validation. By default, request validation is enabled in asp.net mvc. Let's understand this with an example.

Step 1: Create an asp.net mvc4 application using Empty template.

Step 2: Add a HomeController. Copy and paste the following code.

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View();
    }

    [HttpPost]
    public string Index(string comments)
    {
        return "Your Comments: " + comments;
    }
}
```

Step 3: Add Index.cshtml view. Copy and paste the following code.

```
<div style="font-family:Arial">
@using (Html.BeginForm())
{
    <b>Comments:</b>
    <br />
    @Html.TextArea("comments")
    <br />
    <br />
    <input type="submit" value="Submit" />
}
</div>
```

Step 4: Navigate to **/Home/Index**. Type the following text in the **"Comments"** textbox and click **"Submit"**.

```
<h1>Hello</h1>
```

Notice that, you get an error - **A potentially dangerous Request.Form value was detected from the client (comments="<h1>Hello</h1>")**. This is because, by default, request validation is turned on in asp.net mvc and does not allow you to submit any HTML, to prevent XSS (Cross site scripting attacks). We discussed XSS in [Part 55](#) & [Part 56](#) of [asp.net mvc tutorial](#).

However, in some cases, you want the user to be able to submit HTML tags like ****, **<u>** etc. For this to happen, we need to turn off request validation, by decorating the action method with **ValidateInput** attribute as shown below.

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

At this point, you should be able to submit comments, with HTML tags in it.

Part 77 - Custom action filters in asp.net mvc

Suggested Videos

[Part 74 - CacheProfiles](#)

[Part 75 - RequireHttps attribute](#)

[Part 76 - ValidateInput attribute](#)

In this video, we will discuss creating custom action filters in asp.net mvc.

Actions are public methods in a controller. Action filters are attributes, that can be applied either on a controller or on a controller action method, which allow us to add pre and post processing logic to the action methods.

So, in simple terms an action filter allow us to execute some custom code, either, just before an action method is executed or immediately after an action method completes execution. We have discussed some of the built-in action filters in the previous sessions of this video series.

[Part 70 - Authorize attribute](#)

[Part 72 - HandleError attribute](#)

[Part 73 - OutputCache attribute](#)

[Part 75 - RequireHttps attribute](#)

[Part 76 - ValidateInput attribute](#)

Now let's discuss, creating a custom action filter. The custom action filter that we are going to build, should log the following information to a text file.

1. The name of the controller
2. The name of the action method
3. Execution time
4. If there is an exception, log the exception message and the time of the exception.

The output of the text file should be as shown below.

```

Home -> Welcome -> OnActionExecuting - 14/08/2013 16:31:36
Home -> Welcome -> OnActionExecuted - 14/08/2013 16:31:36
Home -> Welcome -> Exception ouured - 14/08/2013 16:31:36
-----

```

```

Home -> Index -> OnActionExecuting - 14/08/2013 16:31:39
Home -> Index -> OnActionExecuted - 14/08/2013 16:31:39
Home -> Index -> OnResultExecuting - 14/08/2013 16:31:39
Home -> Index -> OnResultExecuted - 14/08/2013 16:31:39
-----

```

There are 4 types of filters in asp.net mvc.

- 1. Authorization filters** - Implements `IAuthorizationFilter`. Examples include `AuthorizeAttribute` and `RequireHttpsAttribute`. These filters run before any other filter.
- 2. Action filters** - Implement `IActionFilter`
- 3. Result filters** - Implement `IResultFilter`. Examples include `OutputCacheAttribute`.
- 4. Exception filters** - Implement `IExceptionHandler`. Examples include `HandleErrorAttribute`.

For detailed explanation of these attributes, please refer to the following MSDN link

[http://msdn.microsoft.com/en-us/library/gg416513\(v=vs.98\).aspx](http://msdn.microsoft.com/en-us/library/gg416513(v=vs.98).aspx)

Step 1: Create an asp.net mvc 4 application using "Empty" template

Step 2: Right click on the project name in solution explorer and add "Data" folder. Add a text file to this folder and name it Data.txt

Step 3: Right click on the project name in solution explorer and add "Common" folder. Add a class file to this folder and name it "TrackExecutionTime.cs". Copy and paste the following code. Notice that our custom filter "TrackExecutionTime" inherits from `ActionFilterAttribute` and `IExceptionHandler`. `ActionFilterAttribute` class implements `IActionFilter` and `IResultFilter`.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.IO;

namespace MVCDemo.Common
{
    public class TrackExecutionTime : ActionFilterAttribute, IExceptionHandler
    {
        public override void OnActionExecuting(ActionExecutingContext filterContext)
        {
            string message = "\n" + filterContext.ActionDescriptor.ControllerDescriptor.ControllerName
+
            " -> " + filterContext.ActionDescriptor.ActionName + " -> OnActionExecuting \t- " +
            DateTime.Now.ToString() + "\n";
            LogExecutionTime(message);
        }

        public override void OnActionExecuted(ActionExecutedContext filterContext)
        {
            string message = "\n" + filterContext.ActionDescriptor.ControllerDescriptor.ControllerName
+
            " -> " + filterContext.ActionDescriptor.ActionName + " -> OnActionExecuted \t- " +
            DateTime.Now.ToString() + "\n";
            LogExecutionTime(message);
        }

        public override void OnResultExecuting(ResultExecutingContext filterContext)
        {
            string message = filterContext.RouteData.Values["controller"].ToString() +
            " -> " + filterContext.RouteData.Values["action"].ToString() +
            " -> OnResultExecuting \t- " + DateTime.Now.ToString() + "\n";
            LogExecutionTime(message);
        }
    }
}

```

```

public override void OnResultExecuted(ResultExecutedContext filterContext)
{
    string message = filterContext.RouteData.Values["controller"].ToString() +
        " -> " + filterContext.RouteData.Values["action"].ToString() +
        " -> OnResultExecuted \t- " + DateTime.Now.ToString() + "\n";
    LogExecutionTime(message);
    LogExecutionTime("-----\n");
}

public void OnException(ExceptionContext filterContext)
{
    string message = filterContext.RouteData.Values["controller"].ToString() + " -> " +
        filterContext.RouteData.Values["action"].ToString() + " -> " +
        filterContext.Exception.Message + " \t- " + DateTime.Now.ToString() + "\n";
    LogExecutionTime(message);
    LogExecutionTime("-----\n");
}

private void LogExecutionTime(string message)
{
    File.AppendAllText(HttpContext.Current.Server.MapPath("~/Data/Data.txt"), message);
}
}
}

```

Step 4: Add a HomeController. Copy and paste the following code.

```

public class HomeController : Controller
{
    [TrackExecutionTime]
    public string Index()
    {
        return "Index Action Invoked";
    }

    [TrackExecutionTime]
    public string Welcome()
    {
        throw new Exception("Exception occured");
    }
}

```

Please Note: TrackExecutionTime class is present in MVC Demo.Common namespace.

Build the application and navigate to **/Home/Index** and then to **/Home/Welcome**. Notice that the execution times and the exception details are logged in the text file.

Part 78 - Different types of ActionResult in asp.net mvc

Suggested Videos

[Part 75 - RequireHttps attribute](#)

[Part 76 - ValidateInput attribute](#)

[Part 77 - Custom action filters](#)

In this video, we will discuss different types of **ActionResult** objects that can be returned by an action method.

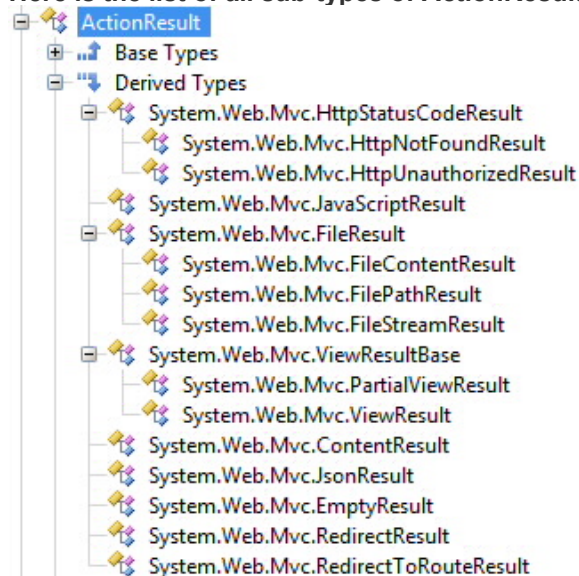
The following is the signature of a typical action method in a controller. Notice that, the return type is **ActionResult**. ActionResult is an abstract class and has several sub types.

```

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

```

Here is the list of all sub-types of ActionResult.



I have used a tool called **ILSpy**, to list all the sub-types of ActionResult. To use the tool yourself, here are the steps

1. Navigate to <http://ilspy.net>
2. Click on "**Download Binaries**" button, and extract them to a folder.
3. Run ILSpy.exe which can be found in the folder, where you have extracted the binaries.
4. Click on File - Open From GAC
5. Type "System.Web.Mvc" in the search box. Select the Assembly and click Open
6. At this point System.Web.Mvc assembly should be loaded into ILSpy. Expand System.Web.Mvc, then expand ActionResult and then expand "Derived Types". You should now be able to see all the derived types.

Why do we have so many sub-types?

An action method in a controller, can return a wide range of objects. For example, an action method can return

1. ViewResult
2. PartialViewResult
3. JsonResult
4. RedirectResult etc..

What should be the return type of an action method - ActionResult or specific derived type?

It's a good practise to return specific sub-types, but, if different paths of the action method returns different subtypes, then I would return an ActionResult object. An example is shown below.

```

public ActionResult Index()
{
    if (Your_Condition)
        return View(); // returns ViewResult object
    else
        return Json("Data"); // returns JsonResult object
}
  
```

The following table lists

1. Action Result Sub Types
2. The purpose of each sub-type
3. The helper methods used to retrun the specific sub-type

Action Result Sub Type	Description	Helper Method
HttpNotFoundResult	Returns an object to indicate that the requested resource cannot be found.	HttpNotFound
HttpUnauthorizedResult	Represents the result of an unauthorized HTTP request	None. Return new HttpUnauthorizedResult()
JavaScriptResult	Returns a piece of JavaScript code that can be executed on the client	JavaScript
FileContentResult	Returns a file to the client	File
FilePathResult	Returns a file to the client, which is provided by the given path	File
FileStreamResult	Returns a file to the client, which is provided by a Stream	File
PartialViewResult	Returns a specified partial view	PartialView
ViewResult	Returns a specified view	View
ContentResult	Writes content to the response stream without requiring a view	Content
JsonResult	Returns a JsonResult which serializes an object in JSON format.	Json
EmptyResult	An empty response is returned. Used when the action method returns void.	None.
RedirectResult	Performs an HTTP redirection to a specified new URL	Redirect
RedirectToRouteResult	Performs an HTTP redirection to another action method that is determined by the routing engine, based on given route data	RedirectToAction, RedirectToRoute, RedirectToActionPermanent, RedirectToRoutePermanent

Part 79 - Areas in asp.net mvc

Suggested Videos

[Part 76 - ValidateInput attribute](#)

[Part 77 - Custom action filters](#)

[Part 78 - Different types of ActionResult in asp.net mvc](#)

In this video, we will discuss **Areas**.

When you create a new asp.net mvc application, the following folders are created with in the root directory.

1. Models
2. Views
3. Controllers

This structure is fine for simple applications, but when your application gets big and complex, maintaining your Models, Views and Controllers can get complicated.

The structure of a complex asp.net mvc application can be very easily maintained using areas. So, in short areas are introduced in asp.net mvc 2, that allow us to breakdown a large complex application into a several small sections called areas. These areas can have their own set of

1. Models
2. Views
3. Controllers
4. Routes

Let's understand this with an example. Let's say, we are building a Job Portal. It's common for a typical job portal to have the following functional areas.

Employee Area - This functional area allows a job seeker to create their profile, upload resume, and perform job search.

Employer Area - This functional area allows a job provider to create employer profile, upload jobs, and search for suitable candidates.

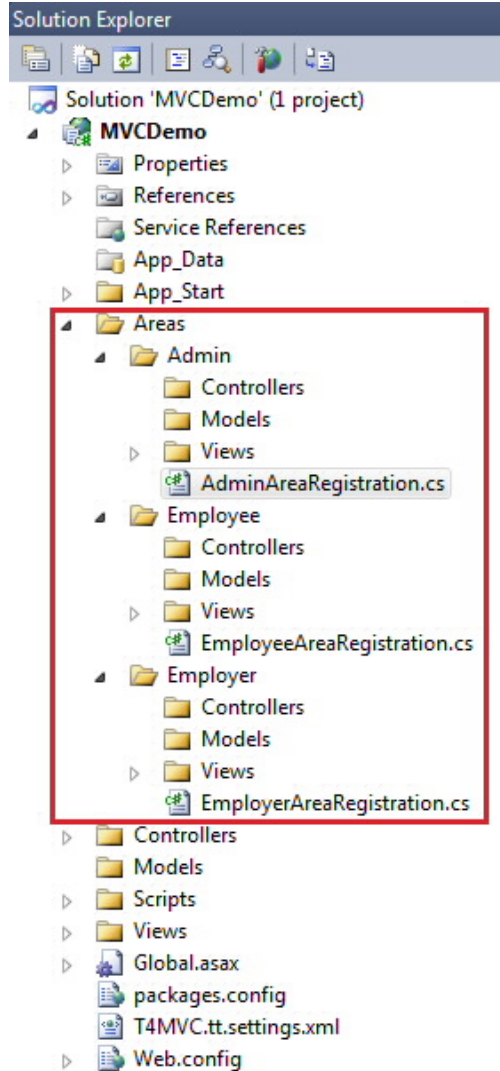
Administrator Area - This functional area allows an administrator to configure the site and maintain.

To create an area in an MVC application

1. Right click on the project name in Solution Explorer and select Add => Area
2. Provide a meaningful name. For example "Employee" and click Add

At this point, "Areas" folder will be created, and with in this, a folder for Employee area will be added. You can add as many areas as you want.

In a similar fashion, add areas for Employer and Admin. At this point, your solution explorer should look as shown below. Notice the Areas folder.



Notice that in each area folder (Employee, Employer and Admin), you have a set of Models, Views and Controllers folders. Also, there is "**AreaRegistration.cs**" file, which contains the code to register a route for the area.

Now navigate to **Global.asax.cs** file and notice **Application_Start()**. With in this method there is code to register all areas in your application.

```
protected void Application_Start()
{
    AreaRegistration.RegisterAllAreas();
    WebApiConfig.Register(GlobalConfiguration.Configuration);
    FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
    RouteConfig.RegisterRoutes(RouteTable.Routes);
}
```

At this point Add "**HomeController**" to the following areas. Notice that we can have a HomeController(Controller with the same) in Employee, Employer, Admin and MainArea.

1. Employee
2. Employer
3. Admin

4. Main Area

At this point, we have Index() action method in all of the HomeController's.

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

Now Add Index view to all the areas. Copy and paste the following HTML in respective Index views.

Main Area: <h1>Main Area Index View</h1>

Employee Area: <h1>Employee Area Index View</h1>

Employer Area: <h1>Employer Area Index View</h1>

Admin Area: <h1>Admin Area Index View</h1>

At this point, build the application, and navigate to <http://localhost/MVCDemo>. You will get an error **Multiple types were found that match the controller named 'Home'. This can happen if the route that services this request ('{controller}/{action}/{id}') does not specify namespaces to search for a controller that matches the request. If this is the case, register this route by calling an overload of the 'MapRoute' method that takes a 'namespaces' parameter.**

The request for 'Home' has found the following matching controllers:

MVCDemo.Controllers.HomeController

MVCDemo.Areas.Admin.Controllers.HomeController

MVCDemo.Areas.Employee.Controllers.HomeController

MVCDemo.Areas.Employer.Controllers.HomeController

To fix this change **RegisterRoutes()** method in **RouteConfig.cs** file in **App_start** folder. Notice that we are passing the namespace of the HomeController in the Main area using namespace 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 },
        namespaces: new [] { "MVCDemo.Controllers" }
    );
}
```

Now, if you navigate to <http://localhost/MVCDemo/Employee>, you will get an error - **The resource cannot be found.**

To fix this, change **RegisterArea()** area method in **EmployeeAreaRegistration.cs** file in Employee folder as shown below. Notice that we are setting **HomeController** as the default controller

```
public override void RegisterArea(AreaRegistrationContext context)
{
    context.MapRoute(
        "Employee_default",
        "Employee/{controller}/{action}/{id}",
        new { controller = "Home", action = "Index", id = UrlParameter.Optional }
    );
}
```

Navigating to <http://localhost/MVCDemo/Employee> may throw a compilation error related to System.Web.Optimization. If you get this error follow the steps below.

1. In Visual Studio, click Tools - Library Package Manager - Package Manager Console
2. In the Package Manager Console window, type the following command and press enter
Install-Package Microsoft.Web.Optimization -Pre

When we are building links using **ActionLink()** html helper to navigate between areas, we need to specify **area name** as shown below. Copy and paste the following code in all the Index views in all the areas and you will be able to navigate between areas when you click on the respective links.

```
<table border="1">
<tr>
<td>
<b>Links</b>
</td>
```

```

</tr>
<tr>
<td>
@Html.ActionLink("Main Area Home Page", "Index", "Home", new { area = "" },null)
</td>
</tr>
<tr>
<td>
@Html.ActionLink("Employee Area Home Page", "Index", "Home", new { area
="Employee" }, null)
</td>
</tr>
<tr>
<td>
@Html.ActionLink("Employer Area Home Page", "Index", "Home", new { area
="Employer" }, null)
</td>
</tr>
<tr>
<td>
@Html.ActionLink("Admin Area Home Page", "Index", "Home", new { area ="Admin" }, null)
</td>
</tr>
</table>

```

Part 80 - StringLength attribute in asp.net mvc

Suggested Videos

[Part 77 - Custom action filters](#)

[Part 78 - Different types of ActionResult in asp.net mvc](#)

[Part 79 - Areas in asp.net mvc](#)

In this video, we will discuss using **StringLength** attribute. This attribute is present in **System.ComponentModel.DataAnnotations** namespace and is used to enforce minimum and maximum length of characters that are allowed in a data field. Let's understand this with an example.

We will be using table tblEmployee for this demo.

Create table tblEmployee

```

(
Id int primary key identity(1,1),
Name nvarchar(50),
Email nvarchar(50),
Age int,
Gender nvarchar(50)
)

```

```

Insert into tblEmployee values('Sara Nan', 'Sara.Nani@test.com', 30, 'Female')
Insert into tblEmployee values('James Histo', 'James.Histo@test.com', 33, 'Male' )
Insert into tblEmployee values('Mary Jane', 'Mary.Jane@test.com', 28, 'Female' )
Insert into tblEmployee values('Paul Sensit', 'Paul.Sensit@test.com', 29, 'Male' )

```

Generate **ADO.NET entity data model** for table **tblEmployee**. Change the entity name from **tblEmployee** to **Employee**. Save and build the project.

Right click on the "Controllers" folder and select Add - Controller. Set

Name = HomeController

Template = MVC controller with read/write actions and views, using Entity Framework
 Model class = Employee(MVCDemo.Models)
 Data Context Class = EmployeeContext(MVCDemo.Models)
 Views = Razor

To validate data, use validation attributes that are found in System.ComponentModel.DataAnnotations namespace. It is not a good idea, to add these validation attributes to the properties of auto-generated **"Employee"** class, as our changes will be lost, if the class is auto-generated again.

So, let's create another partial "Employee" class, and decorate that class with the validation attributes. Right click on the "Models" folder and add Employee.cs class file. Copy and paste the following code.

```
using System.ComponentModel.DataAnnotations;
namespace MVCDemo.Models
{
    [MetadataType(typeof(EmployeeMetaData))]
    public partial class Employee
    {
    }

    public class EmployeeMetaData
    {
        [StringLength(10, MinimumLength = 5)]
        [Required]
        public string Name { get; set; }
    }
}
```

Notice that, we have decorated **"Name"** property with **"StringLength"** attribute and specified Minimum and Maximum length properties. We also used **[Required]** attribute. So, at this point Name property is required and should be between 5 and 10 characters.

Points to remember:

1. **[StringLength]** attribute is present in System.ComponentModel.DataAnnotations namespace.
2. **[StringLength]** attribute verifies that a string is of certain length, but does not enforce that the property is **REQUIRED**. If you want to enforce that the property is required use **[Required]** attribute.

We will discuss the following attributes in our upcoming video sessions.

RegularExpression
 Range

Part 81 - Range attribute in asp.net mvc

Suggested Videos

[Part 78 - Different types of ActionResult in asp.net mvc](#)

[Part 79 - Areas in asp.net mvc](#)

[Part 80 - StringLength attribute](#)

RangeAttribute checks if the value of a data field is within a specified range of values. We will be working with the example, that we started in [Part 80](#). Please watch [Part 80](#), before proceeding.

When you navigate to **/Home/Edit/1**, notice that we don't have validation on **Age field**. If you enter **5000** as the age and click Save, the date gets saved. Obviously an employee having 5000 years as the age is not practical. So, let's validate Age field, and enforce users to enter a value

between 1 and 100. To achieve this **RangeAttribute** can be used.

Make the following change to the **Employee** class in **Employee.cs** file in Models folder. Notice that, we are using **RangeAttribute**, and have set minimum and maximum as 1 and 100 respectively.

```
public class EmployeeMetaData
{
    [StringLength(10, MinimumLength = 5)]
    [Required]
    public string Name { get; set; }

    [Range(1, 100)]
    public int Age { get; set; }
}
```

At this point, we should not be able to enter any values outside the range of 1 and 100 for Age field.

Range attribute can also be used to validate **DateTime** fields. Let's now discuss using Range attribute with DateTime fields.

At the moment our Employee class does not have any **DateTime** field. Let's add **HireDate** column to table **tblEmployee**. Use the sql script below to alter the table.

```
Alter table tblEmployee
Add HireDate Date
```

SQL script to update the existing employee records:

```
Update tblEmployee Set HireDate='2009-08-20' where ID=1
Update tblEmployee Set HireDate='2008-07-13' where ID=2
Update tblEmployee Set HireDate='2005-11-11' where ID=3
Update tblEmployee Set HireDate='2007-10-23' where ID=4
```

Update the ADO.NET data model.

1. In the Solution Explorer, double click on SampleDataModel.edmx file in Models folder.
2. Right click on "Employee" model and select "Update Model from database" option
3. Click on "Refresh" tab on "Update Wizard"
4. Expand "Tables" and select "tblEmployee" table and click "Finish."
5. These steps should add HireDate property to the autogenerated Employee entity class

Build the solution to compile Employee entity class.

Copy and paste the following 2 DIV tags in Edit.cshtml view, just above the "Save" button.

```
<div class="editor-label">
    @Html.LabelFor(model => model.HireDate)
</div>
<div class="editor-field">
    @Html.EditorFor(model => model.HireDate)
    @Html.ValidationMessageFor(model => model.HireDate)
</div>
```

Make the following change to the Employee class in **Employee.cs** file in Models folder. Notice that, we are passing DateTime as the type and specifying the **minimum** and **maximum** values for HireDate. We are also using **DisplayFormat** attribute, so that only date part of **DateTime** is displayed in the Edit view.

```
public class EmployeeMetaData
{
    [StringLength(10, MinimumLength = 5)]
    [Required]
    public string Name { get; set; }

    [Range(1, 100)]
    public int Age { get; set; }

    [Range(typeof(DateTime), "01/01/2000", "01/01/2010")]
    [DisplayFormat(DataFormatString = "{0:d}", ApplyFormatInEditMode = true)]
    public DateTime HireDate { get; set; }
}
```

At this point, we should not be able to enter any values outside the range of "01/01/2000" and "01/01/2010" for HireDate field.

However, when the Range attribute is used with DateTime fields, the client side validation does not work as expected. We will discuss this in a later video session.

Part 82 - Creating custom validation attribute in asp.net mvc

Suggested Videos

[Part 79 - Areas in asp.net mvc](#)

[Part 80 - StringLength attribute](#)

[Part 81 - Range attribute](#)

In this video, we will discuss, **creating custom validation attribute** in asp.net mvc. We will be working with the example, that we started in [Part 80](#). Please watch [Part 80](#), and [Part 81](#) before proceeding.

At the moment, any value outside the range of "01/01/2000" and "01/01/2010" for HireDate filed, will raise a validation error.

```
[Range(typeof(DateTime), "01/01/2000", "01/01/2010")]
[DisplayFormat(DataFormatString = "{0:d}", ApplyFormatInEditMode = true)]
public DateTime HireDate { get; set; }
```

But, let's say, we want the end date to be today's date instead of the hardcoded "01/01/2010" value. To achieve this we would be tempted to use **DateTime.Now.ToShortDateString()** as shown below.

```
[Range(typeof(DateTime), "01/01/2000", DateTime.Now.ToShortDateString())]
[DisplayFormat(DataFormatString = "{0:d}", ApplyFormatInEditMode = true)]
public DateTime HireDate { get; set; }
```

At this point, if you compile, you will get an error - **An attribute argument must be a constant expression, typeof expression or array creation expression of an attribute parameter type.**

To fix this, we can create a custom DateRangeAttribute. Here are the steps

1. Right click on the project name in solution explorer, and add **"Common"** folder.
2. Right click on the **"Common"** folder and add a class file with name = **DateRangeAttribute.cs**
3. Copy and paste the following code in DateRangeAttribute.cs class file.

```
using System;
using System.ComponentModel.DataAnnotations;

namespace MVCDemo.Common
{
    public class DateRangeAttribute : RangeAttribute
    {
        public DateRangeAttribute(string minimumValue)
            : base(typeof(DateTime), minimumValue, DateTime.Now.ToShortDateString())
        {
        }
    }
}
```

4. Finally decorate **"HireDate"** property with our custom **DateRangeAttribute** as shown below. Notice that, we are only passing the minimum date value. Maximum date value will be today's date. Please note, DateRangeAttribute is present in MVCDemo.Common namespace.

```
[DateRange("01/01/2000")]
[DisplayFormat(DataFormatString = "{0:d}", ApplyFormatInEditMode = true)]
public DateTime HireDate { get; set; }
```

Let's now look at another example of creating a custom validation attribute. Let's say our business rules have changed, and the HireDate property should allow any valid date that is <= Today's Date. This means, there is no minimum value restriction and the maximum value should be less than or equal to Today's date. To achieve this, let's add another custom validation attribute. Here are the steps

1. Right click on the "Common" folder and add a class file with name = **CurrentDateAttribute.cs**
2. Copy and paste the following code in CurrentDateAttribute.cs class file.

```

using System;
using System.ComponentModel.DataAnnotations;

namespace MVCDemo.Common
{
    public class CurrentDateAttribute : ValidationAttribute
    {
        public override bool IsValid(object value)
        {
            DateTime dateTime = Convert.ToDateTime(value);
            return dateTime <= DateTime.Now;
        }
    }
}

```

3. Decorate "**HireDate**" property with our custom CurrentDateAttribute as shown below.

```

[CurrentDate]
[DisplayFormat(DataFormatString = "{0:d}", ApplyFormatInEditMode = true)]
public DateTime HireDate { get; set; }

```

Please note that the validation error message can be customised using named parameter "**ErrorMessage**" as shown below.

```

[CurrentDate(ErrorMessage = "Hire Date must be less than or equal to Today's Date")]
[DisplayFormat(DataFormatString = "{0:d}", ApplyFormatInEditMode = true)]
public DateTime HireDate { get; set; }

```

Part 83 - RegularExpression attribute in asp.net mvc

Suggested Videos

[Part 80 - StringLength attribute](#)

[Part 81 - Range attribute](#)

[Part 82 - Creating custom validation attribute](#)

Regular expression attribute is great for pattern matching validation. Let's understand using Regular expression attribute with an example. We will be continuing with the example that we started in [Part 80](#) of the [asp.net mvc tutorial](#).

Here is the requirement for validating Name property

1. Name can contain first and last name with a single space.
2. Last name is optional. If last name is not present, then there shouldn't be any space after the first name.
3. Only upper and lower case alphabets are allowed.

This requirement can be very easily met using **RegularExpression** attribute. In Employee.cs class file, decorate Name property with RegularExpression attribute.

```

[RegularExpression(@"^([A-Za-z]+\s{1}[A-Za-z]+)|([A-Za-z]+)$")]
public string Name { get; set; }

```

Notice that, we are passing regular expression string to the attribute constructor. Regular expressions are great for pattern matching and ensures that, the value for name property is in the format that we want. Also, notice that we are using a verbatim literal(@ symbol) string, as we don't want escape sequences to be processed. We discussed verbatim literal in [Part 4](#) of [C# tutorial](#).

Understanding and writing regular expressions is beyond the scope of this video. If you are interested in learning to write regular expressions, here is a link from MSDN

<http://msdn.microsoft.com/en-us/library/az24scfc.aspx>

The following website is very helpful, for writing and testing regular expressions. This website also contains commonly used regular expressions. Infact, I have picked up the regular expression for validating Name property from here.

<http://gskinner.com/RegExr/>

Let's discuss another example of using validation attribute. A valid internet email address should have an @ and a DOT symbol in it. To match this pattern, use the following regular expression.
`^[w-\._+%]+@(?:[w-]+\.)+[w]{2,6}$`

In **Employee.cs** class file, decorate Email property with **RegularExpression** attribute as shown below.

```
[RegularExpression(@"^[w-\._+%]+@(?:[w-]+\.)+[w]{2,6}$", ErrorMessage = "Please enter a valid email address")]
public string Email { get; set; }
```

Part 84 - Compare attribute in asp.net mvc

Suggested Videos

[Part 81 - Range attribute](#)

[Part 82 - Creating custom validation attribute](#)

[Part 83 - RegularExpression attribute](#)

Compare attribute is used to compare 2 properties of a model. Comparing email addresses and passwords is the common use case of Compare attribute. Let's understand using Compare attribute with an example. We will be continuing with the example, that we discussed in [Part 83](#).

To ensure that the user has typed the correct email, let's include **"Confirm Email"** field on the **"Edit"** view. To achieve this, add **"ConfirmEmail"** property in **"Employee"** class in Employee.cs class file that is present in **"Models"** folder.

```
public partial class Employee
{
    public string ConfirmEmail { get; set; }
}
```

At this point you may get this question. Why are we not adding this property to EmployeeMetaData class. This is because **EmployeeMetaData** class, is used to associate metadata for the properties that are already present in the auto-generated Employee class. The auto-generated Employee class is present in **SampleDataModel.Designer.cs** file in Models folder. **ConfirmEmail** property does not exist in auto-generated Employee class. It is a new property that we want to add to Employee model class. ConfirmEmail property is going to be used only for validation. We will not be saving the value of this property to the database table. We will be storing Email property value to the database.

Build the solution, so that the Employee class is compiled.

Copy and paste the following 2 div tags, to add ConfirmEmail field to the Edit View.

```
<div class="editor-label">
    @Html.LabelFor(model => model.ConfirmEmail)
</div>
<div class="editor-field">
    @Html.EditorFor(model => model.ConfirmEmail)
    @Html.ValidationMessageFor(model => model.ConfirmEmail)
</div>
```

Finally, decorate **ConfirmEmail** property in **Employee** class with **Compare** attribute. Most of the validation attributes are present in System.ComponentModel.DataAnnotations namespace, but Compare attribute is present in System.Web.Mvc namespace.

```
public partial class Employee
{
    [Compare("Email")]
    public string ConfirmEmail { get; set; }
}
```


At this point, this confirm attribute will ensure **Email** and **ConfirmEmail** properties have the same values. If they don't, then a validation error message is displayed.

Part 85 - Enable client side validation in asp.net mvc

Suggested Videos

[Part 82 - Creating custom validation attribute](#)

[Part 83 - RegularExpression attribute](#)

[Part 84 - Compare attribute](#)

Validation attributes in asp.net mvc provides both client and server side validation. There are 2 simple steps to enable client side validation in asp.net mvc.

Step 1: Enable **ClientValidation** and **UnobtrusiveJavaScript** in web.config file.

```
<appSettings>
  <add key="ClientValidationEnabled" value="true" />
  <add key="UnobtrusiveJavaScriptEnabled" value="true" />
</appSettings>
```

Step 2: Include a reference to the following **javascript files**. In real time, the references to the javascript files are included in the master page. This avoids the need to reference them on each and every view where we need validation. **The order in which the script files are referenced is also important.** jquery.validate is dependant on jquery and /jquery.validate.unobtrusive is dependant on jquery.validate, so they should be referenced in the following order. Otherwise, client side validation will not work. In short, JavaScript is parsed "top-down", so all dependencies need to be referenced before the dependant reference.

```
<script src="~/Scripts/jquery-1.7.1.min.js" type="text/javascript"></script>
<script src="~/Scripts/jquery.validate.min.js" type="text/javascript"></script>
<script src="~/Scripts/jquery.validate.unobtrusive.min.js" type="text/javascript"></script>
```

With these 2 changes, validation should now happen on the client without a round trip to the server. If the client disables javascript in the browser, then client side validation does not work, but server side validation will continue to work as normal.

Part 86 - ValidationSummary in asp.net mvc

Suggested Videos

[Part 83 - RegularExpression attribute](#)

[Part 84 - Compare attribute](#)

[Part 85 - Enable client side validation](#)

In this video, we will discuss **displaying all validation errors at one place using validation summary html helper**. A red asterisk (start) should be displayed next to every field that has failed the validation. The output should be as shown below.

Name

Sara Nani234234 *

Email

Test@te *

ConfirmEmail

Test@test.com *

Age

450 *

Gender

Female

HireDate

01/01/2009

Please fix the following errors and then submit the form

- Please enter upper and lower case alphabets only
- Please enter a valid email
- Email and ConfirmEmail must match
- The field Age must be between 1 and 100.

Save

We will be working with the example that we discussed in [Part 85](#). Please watch [Part 85](#), before proceeding.

First of all, the validation error messages should be displayed in red color. On the "Edit" view, include a reference to **Site.css** style sheet.

```
<link href="~/Content/Site.css" rel="stylesheet" type="text/css" />
```

In **site.css** there are 3 CSS classes that control the display of validation error messages. Adjust the styles of these classes according to your needs.

field-validation-error

input.input-validation-error

validation-summary-errors

To display all errors at one place, use ValidationSummary() HTML helper.

```
@Html.ValidationSummary(false, "Please fix the following errors and then submit the form")
```

To display an asterisk, next to the field that has failed validation, modify ValidationMessageFor() html helper as shown below.

```
@Html.ValidationMessageFor(model => model.Name, "**")
```

At this point, next to the field that has failed the validation message, a star symbol will be displayed instead of the detailed error message. All the detailed error messages will be displayed using validation summary.

Part 86 - ValidationSummary in asp.net mvc

Suggested Videos

[Part 83 - RegularExpression attribute](#)

[Part 84 - Compare attribute](#)

[Part 85 - Enable client side validation](#)

In this video, we will discuss **displaying all validation errors at one place using validation summary html helper**. A red asterisk (star) should be displayed next to every field that has failed the validation. The output should be as shown below.

Name

Sara Nani234234 *

Email

Test@te *

ConfirmEmail

Test@test.com *

Age

450 *

Gender

Female

HireDate

01/01/2009

Please fix the following errors and then submit the form

- Please enter upper and lower case alphabets only
- Please enter a valid email
- Email and ConfirmEmail must match
- The field Age must be between 1 and 100.

Save

We will be working with the example that we discussed in [Part 85](#). Please watch [Part 85](#), before proceeding.

First of all, the validation error messages should be displayed in red color. On the "Edit" view, include a reference to **Site.css** style sheet.

```
<link href="~/Content/Site.css" rel="stylesheet" type="text/css" />
```

In **site.css** there are 3 CSS classes that control the display of validation error messages. Adjust the styles of these classes according to your needs.

field-validation-error

input.input-validation-error

validation-summary-errors

To display all errors at one place, use ValidationSummary() HTML helper.

```
@Html.ValidationSummary(false, "Please fix the following errors and then submit the form")
```

To display an asterisk, next to the field that has failed validation, modify ValidationMessageFor() html helper as shown below.

```
@Html.ValidationMessageFor(model => model.Name, "**")
```

At this point, next to the field that has failed the validation message, a star symbol will be displayed instead of the detailed error message. All the detailed error messages will be displayed using validation summary.

Part 87 - What is Unobtrusive JavaScript

Suggested Videos

[Part 84 - Compare attribute](#)

[Part 85 - Enable client side validation](#)

[Part 86 - ValidationSummary in asp.net mvc](#)

What is Unobtrusive JavaScript?

Unobtrusive JavaScript, is a JavaScript that is separated from the web site's html markup. There are several benefits of using Unobtrusive JavaScript. Separation of concerns i.e the HTML markup is now clean without any traces of javascript. Page load time is better. It is also easy to update the code as all the Javascript logic is present in a separate file. We also get, better cache support, as all our JavaScript is now present in a separate file, it can be cached and accessed much faster.

Example:

We want to change the backgroundColor of **"Save"** button on **"Edit"** view to **"Red"** on MouseOver and to **"Grey"** on MouseOut.

First let's look at achieving this using obtrusive javascript.

Step 1: Implement MouseOver() and MouseOut() functions

```
<script type="text/javascript" language="javascript">
    function MouseOver(controlId) {
        var control = document.getElementById(controlId);
        control.style.backgroundColor = 'red'
    }

    function MouseOut(controlId) {
        var control = document.getElementById(controlId);
        control.style.backgroundColor = '#d3dce0'
    }
</script>
```

Step 2: Associate the javascript functions with the respective events.

```
<input id="btnSubmit" type="submit" value="Save"
    onmouseover="MouseOver('btnSubmit')" onmouseout="MouseOut('btnSubmit')" />
```

Now let's look at making this javascript unobtrusive, using jQuery

Step 1: Right click on the "Scripts" folder in "Solution Explorer", and add a jsScript file with name = "CustomJavascript.js"

Step 2: Copy and paste the following code in CustomJavascript.js file.

```
$(function () {
    $("#btnSubmit").mouseover(function () {
        $("#btnSubmit").css("background-color", "red");
    });

    $("#btnSubmit").mouseout(function () {
        $("#btnSubmit").css("background-color", "#d3dce0");
    });
});
```

Step 3: Add a reference to CustomJavascript.js file in Edit view.

```
<script src="~/Scripts/CustomJavascript.js" type="text/javascript"></script>
```

Step 4: Remove the following obtrusive Javascript from "Edit" view

```
<script type="text/javascript" language="javascript">
    function MouseOver(controlId) {
        var control = document.getElementById(controlId);
```

```

        control.style.backgroundColor = 'red'
    }

    function MouseOut(controlId) {
        var control = document.getElementById(controlId);
        control.style.backgroundColor = '#d3dce0'
    }
</script>

```

Also, remove "onmouseover" and "onmouseout" events from the button control.

```

<input id="btnSubmit" type="submit" value="Save"
    onmouseover="MouseOver('btnSubmit')" onmouseout="MouseOut('btnSubmit')" />

```

Part 88 - Unobtrusive validation in asp.net mvc

Suggested Videos

[Part 85 - Enable client side validation](#)

[Part 86 - ValidationSummary in asp.net mvc](#)

[Part 87 - What is Unobtrusive JavaScript](#)

Client side validation in asp.net mvc is unobtrusive. To turn on client side validation and unobtrusive JavaScript, make sure the following 2 keys under appSettings element within web.config file are turned on. This will turn on client side validation and unobtrusive JavaScript for the entire application.

```

<appSettings>
  <add key="ClientValidationEnabled" value="true" />
  <add key="UnobtrusiveJavaScriptEnabled" value="true" />
</appSettings>

```

Is it possible to turn these features on/off using code?

Yes, the features can be enabled or disabled in Application_Start() event handler in Global.asax as shown below. This will turn on client side validation and unobtrusive JavaScript for the entire application.

```

protected void Application_Start()
{
    HtmlHelper.UnobtrusiveJavaScriptEnabled = true;
    HtmlHelper.ClientValidationEnabled = true;
}

```

Is it possible to turn these features on/off for a specific view?

Yes, include the following code, on a view where you want to enable/disable these features.

```

@{
    Html.EnableClientValidation(true);
    Html.EnableUnobtrusiveJavaScript(true);
}

```

How is Unobtrusive validation implemented in asp.net mvc?

Using "data" attributes. For example, if we use "Required" attribute, on Name property and if we enable client side validation and unobtrusive JavaScript, then the generated HTML for "Name" field is as shown below.

```

<input class="text-box single-line"
    data-val="true"
    data-val-required="The Name field is required."
    id="Name"
    name="Name"
    type="text"
    value="Sara Nan" />

```

data-val="true", indicates that the unobtrusive validation is turned on for this element.

data-val-required="The Name field is required.", indicates that the "Name" field is required and the associated error message will be displayed if the validation fails. These data attributes are used by

jQuery validation plugin for client side validation.

Part 89 - Remote validation in asp.net mvc

Suggested Videos

[Part 86 - ValidationSummary in asp.net mvc](#)

[Part 87 - What is Unobtrusive JavaScript](#)

[Part 88 - Unobtrusive validation](#)

Sometimes, to check if a field value is valid, we may need to make a database call. A classic example of this is the user registration page. To register a user, we need a unique username. So, to check, if the username is not taken already, we have to make a call to the server and check the database table. RemoteAttribute is useful in situations like this.

Example: When a user provides a username that already exists, the associated validation error message should be displayed immediately as shown below.

User Registration

FullName

UserName

UserName already in use.

Password

Step 1: Create tblUsers table

Create table tblUsers

```
(
[Id] int primary key identity,
[FullName] nvarchar(50),
[UserName] nvarchar(50),
[Password] nvarchar(50)
)
```

Step 2: Create an ado.net entity data model using table **tblUsers**. Upon creating the entity model, change the name of the entity from to User. Save changes and build the solution.

Step 3: Add HomeController with the following settings

1. controller Name = HomeController
2. Template = MVC controller with read/write actions and views, using Entity Framework
3. Model Class = User (MVCDemo.Models)
4. Data context class = SampleDbContext (MVCDemo.Models)
5. Views = Razor

Step 4: Copy and paste the following function in HomeController. This is the method which gets called to perform the remote validation. An AJAX request is issued to this method. If this method returns true, validation succeeds, else validation fails and the form is prevented from being submitted. The parameter name (UserName) must match the field name on the view. If they don't match, model binder will not be able bind the value with the parameter and validation may not work as expected.

```
public JsonResult IsUserNameAvailable(string UserName)
{
    return Json(!db.Users.Any(x => x.UserName == UserName),
```

```
JsonRequestBehavior.AllowGet);
```

```
}
```

Step 5: Right click on the Models folder and a class file with name = User.cs. Copy and paste the following code. Notice that the name of the method (IsUserNameAvailable) and the controller name (Home) and error message are passed as arguments to Remote Attribute

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.ComponentModel.DataAnnotations;
using System.Web.Mvc;

namespace MVCDemo.Models
{
    [MetadataType(typeof(UserMetaData))]
    public partial class User
    {
    }

    public class UserMetaData
    {
        [Remote("IsUserNameAvailable", "Home", ErrorMessage="UserName already in use.")]
        public string UserName { get; set; }
    }
}
```

Step 6: Include references to the following css and script files in Create.cshtml
view.jQuery, jquery.validate and jquery.validate.unobtrusive script files are required for remote validation to work.

```
<link href="/Content/Site.css" rel="stylesheet" type="text/css" />
<script src="/Scripts/jquery-1.7.1.min.js" type="text/javascript"></script>
<script src="/Scripts/jquery.validate.min.js" type="text/javascript"></script>
<script src="/Scripts/jquery.validate.unobtrusive.js" type="text/javascript"></script>
```

Step 7: Make sure **ClientValidation** and **UnobtrusiveJavaScript** are enabled in web.config

```
<add key="ClientValidationEnabled" value="true" />
<add key="UnobtrusiveJavaScriptEnabled" value="true" />
```

Part 90 - Remote validation in mvc when javascript is disabled

Suggested Videos

[Part 87 - What is Unobtrusive JavaScript](#)

[Part 88 - Unobtrusive validation](#)

[Part 89 - Remote validation in asp.net mvc](#)

Please watch [Part 89](#) from [asp.net mvc tutorial](#), before proceeding.

Out of the box, **Remote** attribute only works when JavaScript is enabled. If the end user, disables JavaScript on his/her machine then the validation does not work. This is because RemoteAttribute requires JavaScript to make an asynchronous AJAX call to the server side validation method. As a result, the user will be able to submit the form, bypassing the validation in place. This why it is always important to have server side validation.

To make server side validation work, when JavaScript is disabled, there are 2 ways

1. Add model validation error dynamically in the controller action method
2. Create a custom remote attribute and override IsValid() method

In this video, we will discuss, **adding model validation error dynamically in the controller action method**. We will continue with the example, that we worked with in Part 89. Modify the Create action method that is decorated with [HttpPost] attribute as shown below.


```
[HttpPost]
public ActionResult Create(User user)
{
    // Check if the UserName already exists, and if it does, add Model validation error
    if (db.Users.Any(x => x.UserName == user.UserName))
    {
        ModelState.AddModelError("UserName", "UserName already in use");
    }
    if (ModelState.IsValid)
    {
        db.Users.AddObject(user);
        db.SaveChanges();
        return RedirectToAction("Index");
    }

    return View(user);
}
```

At this point, disable JavaScript in the browser, and test your application. Notice that, we don't get client side validation, but when you submit the form, server side validation still prevents the user from submitting the form, if there are validation errors.

However, delegating the responsibility of performing validation, to a controller action method violates separation of concerns within MVC. Ideally all validation logic should be in the Model. Using validation attributes in mvc models, should be the preferred method for validation. In our next video, we will discuss, creating a custom remote attribute and overriding **IsValid()** method.

art 91 - Create a custom remote attribute and override IsValid() method

Suggested Videos

[Part 88 - Unobtrusive validation](#)

[Part 89 - Remote validation in asp.net mvc](#)

[Part 90 - Remote validation in mvc when javascript is disabled](#)

Please watch [Part 90](#) from [asp.net mvc tutorial](#), before proceeding.

Out of the box, Remote attribute only works when JavaScript is enabled. If the end user, disables JavaScript on his/her machine then the validation does not work. This is because **RemoteAttribute** requires JavaScript to make an asynchronous AJAX call to the server side validation method. As a result, the user will be able to submit the form, bypassing the validation in place. This why it is always important to have server side validation.

To make server side validation work, when JavaScript is disabled, there are 2 ways

1. Add model validation error dynamically in the controller action method - Discussed in [Part 90](#)
2. Create a custom remote attribute and override IsValid() method

In this video, we will discuss, creating a custom remote attribute

Step 1: Right click on the project name in solution explorer and a folder with name = "Common"

Step 2: Right click on the "Common" folder, you have just added and add a class file with name = RemoteClientServer.cs

Step 3: Copy and paste the following code.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.ComponentModel.DataAnnotations;
```



```

using System.Reflection;

namespace MVCDemo.Common
{
    public class RemoteClientServerAttribute : RemoteAttribute
    {
        protected override ValidationResult IsValid(object value, ValidationContext validationContext)
        {
            // Get the controller using reflection
            Type controller = Assembly.GetExecutingAssembly().GetTypes()
                .FirstOrDefault(type => type.Name.ToLower() == string.Format("{0}Controller",
                    this.RouteData["controller"].ToString()).ToLower());
            if (controller != null)
            {
                // Get the action method that has validation logic
                MethodInfo action = controller.GetMethods()
                    .FirstOrDefault(method => method.Name.ToLower() ==
                        this.RouteData["action"].ToString().ToLower());
                if (action != null)
                {
                    // Create an instance of the controller class
                    object instance = Activator.CreateInstance(controller);
                    // Invoke the action method that has validation logic
                    object response = action.Invoke(instance, new object[] { value });
                    if (response is JsonResult)
                    {
                        object jsonData = ((JsonResult)response).Data;
                        if (jsonData is bool)
                        {
                            return (bool)jsonData ? ValidationResult.Success :
                                new ValidationResult(this.ErrorMessage);
                        }
                    }
                }
            }

            return ValidationResult.Success;
            // If you want the validation to fail, create an instance of ValidationResult
            // return new ValidationResult(base.ErrorMessageString);
        }

        public RemoteClientServerAttribute(string routeName)
            : base(routeName)
        {
        }

        public RemoteClientServerAttribute(string action, string controller)
            : base(action, controller)
        {
        }

        public RemoteClientServerAttribute(string action, string controller,
            string areaName) : base(action, controller, areaName)
        {
        }
    }
}

```

Step 4: Open "**User.cs**" file, that is present in "**Models**" folder. Decorate "**UserName**" property with RemoteClientServerAttribute.

RemoteClientServerAttribute is in MVCDemo.Common namespace, so please make sure you have a using statement for this namespace.

```

public class UserMetadata
{
    [RemoteClientServer("IsUserNameAvailable", "Home",
        ErrorMessage="UserName already in use")]
    public string UserName { get; set; }
}

```

Disable JavaScript in the browser, and test your application. Notice that, we don't get client side validation, but when you submit the form, server side validation still prevents the user from submitting the form, if there are validation errors.

art 92 - Ajax with asp.net mvc

Suggested Videos

[Part 89 - Remote validation in asp.net mvc](#)

[Part 90 - Remote validation in mvc when javascript is disabled](#)

[Part 91 - Create a custom remote attribute and override IsValid\(\) method](#)

ASP.NET AJAX enables a Web application to retrieve data from the server asynchronously and to update portions of the existing page. So these, partial page updates make web application more responsive and hence improves user experience.

In this video, let's discuss using **Ajax.ActionLink** helper. By the end of this video, we should be able to load either

1. All students or
 2. Top 3 students or
 3. Bottom 3 students
- depending on the link we have clicked.

Students

[All](#) | [Top 3](#) | [Bottom 3](#)

Name	TotalMarks
Ram	990
Steve	983
John	980

Here are the steps to achieve this

Step 1: Sql script to create and populate table tblStudents

Create table tblStudents

```
(
  Id int identity primary key,
  Name nvarchar(50),
  TotalMarks int
)
```

```
Insert into tblStudents values ('Mark', 900)
Insert into tblStudents values ('Pam', 760)
Insert into tblStudents values ('John', 980)
Insert into tblStudents values ('Ram', 990)
Insert into tblStudents values ('Ron', 440)
Insert into tblStudents values ('Able', 320)
Insert into tblStudents values ('Steve', 983)
Insert into tblStudents values ('James', 720)
Insert into tblStudents values ('Mary', 870)
Insert into tblStudents values ('Nick', 680)
```

Step 2: Create an ado.net entity **data model** using table **tblStudents**. Upon creating the entity model, change the name of the entity to Student. Save changes and build the solution, so that the Student entity class gets compiled.

Step 3: Add **"Shared"** folder (if it doesn't already exists) in **"Views"** folder. Right click on **"Shared"** folder and add a partial view, with name = **_Student.cshtml**.

```
@model IEnumerable<MVCDemo.Models.Student>

<table style="border:1px solid black; background-color:Silver">
<tr>
<th>
    @Html.DisplayNameFor(model => model.Name)
</th>
<th>
    @Html.DisplayNameFor(model => model.TotalMarks)
</th>
</tr>

@foreach (var item in Model)
{
<tr>
<td>
    @Html.DisplayFor(modelItem => item.Name)
</td>
<td>
    @Html.DisplayFor(modelItem => item.TotalMarks)
</td>
</tr>
}
</table>
```

Step 4: Right click on the **"Controllers"** folder and add a controller with the following settings

1. Controller Name = HomeController
2. Template = Empty MVC controller

Copy and paste the following code. Please make sure to include MVCDemo.Models namespace.

```
public class HomeController : Controller
{
    // Create an instance of DbContext class
    DbContext db = new DbContext();

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

    // Return all students
    public PartialViewResult All()
    {
        List<Student> model = db.Students.ToList();
        return PartialView("_Student", model);
    }

    // Return Top3 students
    public PartialViewResult Top3()
    {
        List<Student> model = db.Students.OrderByDescending(x => x.TotalMarks).Take(3).ToList();
        return PartialView("_Student", model);
    }

    // Return Bottom3 students
    public PartialViewResult Bottom3()
    {
        List<Student> model = db.Students.OrderBy(x => x.TotalMarks).Take(3).ToList();
        return PartialView("_Student", model);
    }
}
```

Step 5: Right click on the **"Views"** folder and add a folder with **name = "Home"**. Right click on the **"Home"** folder and add a view with Name = **"Index"**.

Points to remember:

- a) For AJAX to work, jquery and jquery.unobtrusive-ajax javascript files need to be referenced. Reference to jquery.unobtrusive-ajax.js file should be after jquery.js reference.

b) Ajax.ActionLink() html helper has several overloaded versions. We have used the following version
 ActionLink(string linkText, string actionName, AjaxOptions ajaxOptions);
First parameter : is the link text that we see on the user interface
Second parameter : is the name of the action method that needs to be invoked, when the link is clicked.
Third parameter : is the ajaxOptions object.

```
<div style="font-family:Arial">
```

```
<script src=~/Scripts/jquery-1.7.1.min.js" type="text/javascript"></script>
```

```
<script src=~/Scripts/jquery.unobtrusive-ajax.min.js" type="text/javascript"></script>
```

```
<h2>Students</h2>
```

```
@Ajax.ActionLink("All", "All",
    new AjaxOptions
    {
        HttpMethod = "GET", // HttpMethod to use, GET or POST
        UpdateTargetId = "divStudents", // ID of the HTML element to update
        InsertionMode = InsertionMode.Replace // Replace the existing contents
    })
```

```
<span style="color:Blue">|</span>
```

```
@Ajax.ActionLink("Top 3", "Top3",
    new AjaxOptions
    {
        HttpMethod = "GET", // HttpMethod to use, GET or POST
        UpdateTargetId = "divStudents", // ID of the HTML element to update
        InsertionMode = InsertionMode.Replace // Replace the existing contents
    })
```

```
<span style="color:Blue">|</span>
```

```
@Ajax.ActionLink("Bottom 3", "Bottom3",
    new AjaxOptions
    {
        HttpMethod = "GET", // HttpMethod to use, GET or POST
        UpdateTargetId = "divStudents", // ID of the HTML element to update
        InsertionMode = InsertionMode.Replace // Replace the existing contents
    })
```

```
<br /><br />
```

```
<div id="divStudents">
```

```
</div>
```

```
</div>
```

Part 93 - What is Ajax and why should we use it

Suggested Videos

[Part 90 - Remote validation in mvc when javascript is disabled](#)

[Part 91 - Create a custom remote attribute and override IsValid\(\) method](#)

[Part 92 - Ajax with asp.net mvc](#)

In this video, we will discuss

1. What is AJAX
2. Advantages and disadvantages of using AJAX
3. Applications that are currently using AJAX.
4. When to use AJAX

What is AJAX

AJAX stands for **A**ynchronous **J**avaScript **A**nd **X**ML. AJAX enable web applications to retrieve data from the server asynchronously. Web application using AJAX enables partial page updates, ie. only the related section of the page is updated, without reloading the entire page.

Advantages of AJAX

1. AJAX applications are non blocking. As AJAX requests are asynchronous, the user doesn't have to wait for the request processing to complete. Even while the request is still being processed by the server, the application remains responsive and the user can interact with the application. When the request processing is complete, the user interface is automatically updated. This is not the case with, synchronous requests. The user interface is blocked and the user cannot do anything else until the request has completed processing.

2. Better performance and reduced network traffic. AJAX enables an application to send and receive only the data that is required. As a result, there is reduced traffic between the client and the server and better performance.

3. No screen flicker. An AJAX response consists of only the data that is relevant to the request. As a result, only a portion of the page is updated avoiding full page refresh and screen flickers.

Disadvantages of AJAX:

1. AJAX requests cannot be bookmarked easily
2. AJAX relies on JavaScript. If JavaScript is disabled, AJAX application won't work.
3. Harder to debug
4. Search Engine like Google, Bing, Yahoo etc cannot index AJAX pages.

Applications using AJAX

1. Many web sites like Google, bing, youtube, yahoo use AJAX to implement AutoComplete feature.
2. Gmail use AJAX to implement AutoSave feature
3. Gmail use AJAX to implement RemoteValidation i.e to validate if a user name is already in use, when creating a gmail account. We discussed remote validation in Parts [89](#), [90](#) & [91](#).
4. Facebook use AJAX, to load data as you keep scrolling down.

AJAX is generally used to implement features like

1. AutoComplete
2. AutoSave
3. Remote validation etc

Part 94 - Providing visual feedback using LoadingElementId AjaxOption

Suggested Videos

[Part 91 - Create a custom remote attribute and override IsValid\(\) method](#)

[Part 92 - Ajax with asp.net mvc](#)

[Part 93 - What is Ajax and why should we use it](#)

This is continuation to [Part 93](#), Please watch [Part 93](#) from the [ASP.NET MVC tutorial](#) before proceeding with this video.

One problem with partial page updates is that, they donot provide any visual feedback if the request is taking a long time to process. With full page postbacks, we don't have this problem, because the entire page is posted to the server and hence the user knows something is happening.

For example, let's say a request in an ajax enabled application takes 10 seconds to complete. When the user clicks the button and if we don't provide him with any visual feedback that the request is being processed, the user may think the website is not working and he may click the button again. So, it is very important that we provide some visual feedback. To achieve this here are the steps.

Step 1: Right click on the project name in solution explorer, and a folder with name = Images. Download and paste the following animated GIF image, in the Images folder.



Please note: You can create animated GIF images using the following website.

<http://spiffygif.com>

Step 2: Place the following DIV tag on the view, where you want the image to show up when request is still being processed.

```
<div id="divloading" style="display:none;">
  
</div>
```

Step 3: Modify the Ajax.ActionLink() html helper method on the view as shown below. Notice that, we have set **LoadingElementId = "divloading"**

```
@Ajax.ActionLink("All", "All",
    new AjaxOptions
    {
        HttpMethod = "GET",
        UpdateTargetId = "divStudents",
        InsertionMode = InsertionMode.Replace,
        LoadingElementId = "divloading"
    })
```

Step 4: The controller action method in our example, completes in no time. So the image never shows up. To introduce some latency include, `System.Threading.Thread.Sleep(1000);` statement in the controller action method. The duration for the sleep method is in milliseconds. We have specified 1000 mill-seconds, so the thread is going to sleep for 1 second.

```
public PartialViewResult All()
{
    System.Threading.Thread.Sleep(1000);
    List<Student> model = db.Students.ToList();
    return PartialView("_Student", model);
}
```

Now run the application and notice that image shows up, while the request is still being processed. When request processing completes and when the response is received, the loading image disappears and the UI is automatically updated, with the data received.

Part 95 - OnBegin, OnComplete, OnSuccess and OnFailure properties of AjaxOptions class

Suggested Videos

[Part 92 - Ajax with asp.net mvc](#)

[Part 93 - What is Ajax and why should we use it](#)

[Part 94 - Providing visual feedback using LoadingElementId AjaxOption](#)

This is continuation to [Part 94](#), Please watch [Part 94](#) from the [ASP.NET MVC tutorial](#) before proceeding with this video.

Using the following 4 properties of the AjaxOptions class, we can associate JavaScript functions that get called on the client at different stages of an AJAX request/response cycle.

1. **OnBegin** - The associated JavaScript function is called before the action method is invoked
2. **OnComplete** - The associated JavaScript function is invoked after the response data has been instantiated but before the page is updated.
3. **OnSuccess** - The associated JavaScript function is invoked after the page is updated.
4. **OnFailure** - The associated JavaScript function is invoked if the page update fails.

We will continue with the example that we discussed in [Part 94](#).

1. When you click on "All" link, all the students will be loaded into the <div> element that has got **id=divStudents**.
2. At the moment, when you click on "Top 3" link, the data from the previous request is still present

in **"divStudents"**. Only when "Top 3" students become available, **"divStudents"** is updated with new data.

3. Our application requirement is to clear, **"divStudents"** element content, as soon as we click on any AJAX link and before the associated action method is invoked.

To achieve this, let's use "OnBegin" property of AjaxOptions class.

Step 1: The following JavaScript function, finds the "divStudents" and clear it's contents

```
function ClearResults()
{
    $("#divStudents").empty();
}
```

Step 2: Associate ClearResults() function with "OnBegin" property of AjaxOptions class.

```
@Ajax.ActionLink("All", "All",
    new AjaxOptions
    {
        HttpMethod = "GET",
        UpdateTargetId = "divStudents",
        InsertionMode = InsertionMode.Replace,
        LoadingElementId = "divloading",
        OnBegin = "ClearResults"
    })
```

Please Note:

OnBegin property can also be used to cancel the invocation of the action method. The JavaScript function that is associated with **"OnBegin"** property is invoked before the action method is invoked. So if that associated JavaScript function returns false, then the action method will not be invoked. Your JavaScript function may look something like this.

```
function Validate()
{
    if (condition)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

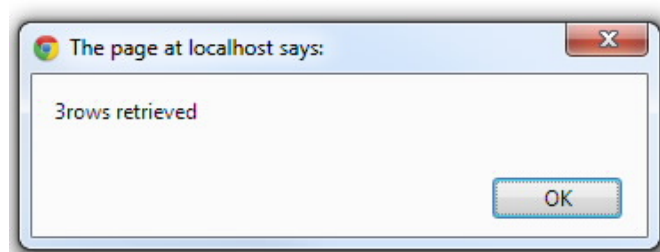
Let's now discuss an example of using **"OnSuccess"** property. Immediately after the page is updated with students data, we want to display the total number of rows retrieved using JavaScript alert.

Students

[All](#) | [Top 3](#) | [Bottom 3](#)



Name	TotalMarks
Ram	990
Steve	983
John	980



To achieve this, let's use "OnSuccess" property of AjaxOptions class.

Step 1: Set id attribute for the table element in **"_Student.cshtml"** partial view


```
<table id="tableStudents" style="border:1px solid black; background-color:Silver">
```

Step 2: The following JavaScript function counts the total number of rows.

```
function CountRows()
{
    alert($("#tableStudents tr").length - 1) + 'rows retrieved';
}
```

Step 3: Associate **CountRows()** function with **"OnSuccess"** property of AjaxOptions class.

```
@Ajax.ActionLink("All", "All",
    new AjaxOptions
    {
        HttpMethod = "GET",
        UpdateTargetId = "divStudents",
        InsertionMode = InsertionMode.Replace,
        LoadingElementId = "divloading",
        OnSuccess = "CountRows"
    })
```

Part 96 - LoadingElementDuration property of AjaxOptions class

Suggested Videos

[Part 93 - What is Ajax and why should we use it](#)

[Part 94 - Providing visual feedback using LoadingElementId AjaxOption](#)

[Part 95 - OnBegin, OnComplete, OnSuccess and OnFailure properties of AjaxOptions class](#)

This is continuation to [Part 95](#), Please watch [Part 95](#) from the [ASP.NET MVC tutorial](#) before proceeding with this video.

LoadingElementDuration property is used to control the animation duration of LoadingElement. The value for LoadingElementDuration property must be specified in milliseconds. By default, the LoadingElement fades in and fades out.

But I noticed that, irrespective of whatever duration you set, the **LoadingElement** was fading in and out at the same speed. I have done some research on google and came across the following stackoverflow article which explained the fix for the issue.

<http://stackoverflow.com/questions/8928671/asp-net-mvc-3-loadingelementduration-not-working>

To fix the issue, we need to parse the duration to integer. Open **"jquery.unobtrusive-ajax.js"** and change the following line

FROM

```
duration = element.getAttribute("data-ajax-loading-duration") || 0;
```

TO

```
duration = parseInt(element.getAttribute("data-ajax-loading-duration")) || 0;
```

After you have made the change, reference "jquery.unobtrusive-ajax.js" file in the **"Index"** view

```
<script src="~/Scripts/jquery.unobtrusive-ajax.js" type="text/javascript"></script>
```

Finally set, **LoadingElementDuration** and test your application

```
@Ajax.ActionLink("All", "All",
    new AjaxOptions
    {
        HttpMethod = "GET",
        UpdateTargetId = "divStudents",
        InsertionMode = InsertionMode.Replace,
        LoadingElementId = "divloading",
        OnBegin = "ClearResults",
```



```

    OnSuccess = "CountRows",
    LoadingElementDuration = 2000
})

```

Part 97 - Implement autocomplete textbox functionality in mvc

Suggested Videos

[Part 94 - Providing visual feedback using LoadingElementId AjaxOption](#)

[Part 95 - OnBegin, OnComplete, OnSuccess and OnFailure properties of AjaxOptions class](#)

[Part 96 - LoadingElementDuration property of AjaxOptions class](#)

In this video, we will discuss **implementing auto-complete functionality** in an asp.net mvc application using jQuery Autocomplete Widget

Name:

Name	Gender	TotalMarks
Mark Hastings	Male	900
Mary Ward	Female	870
Pam Nicholas	Female	760
John Stenson	Male	980
Ram Gerald	Male	990
Ron Simpson	Male	440
Able Wicht	Male	320
Steve Thompson	Male	983
James Byne	Male	720
Mary Ward	Female	870
Nick Niron	Male	680

Step 1: We will be using **tblStudents** table in this demo. Please find the sql script below, to create and populate this table with some data.

Create Table tblStudents

```

(
ID int identity primary key,
Name nvarchar(50),
Gender nvarchar(20),
TotalMarks int
)

```

```

Insert into tblStudents values('Mark Hastings','Male',900)
Insert into tblStudents values('Pam Nicholas','Female',760)
Insert into tblStudents values('John Stenson','Male',980)
Insert into tblStudents values('Ram Gerald','Male',990)
Insert into tblStudents values('Ron Simpson','Male',440)
Insert into tblStudents values('Able Wicht','Male',320)
Insert into tblStudents values('Steve Thompson','Male',983)
Insert into tblStudents values('James Byne','Male',720)
Insert into tblStudents values('Mary Ward','Female',870)
Insert into tblStudents values('Nick Niron','Male',680)

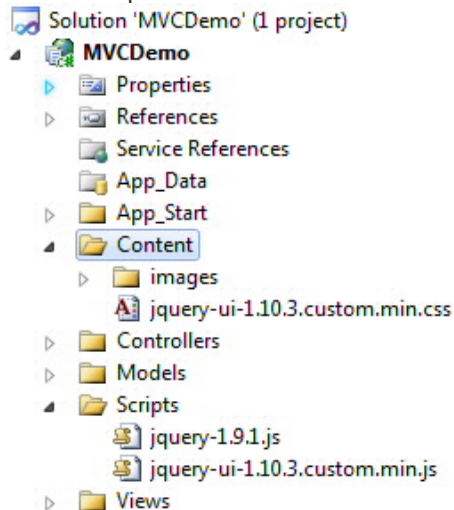
```

Step 2: Create an ado.net entity data model using table **tblStudents**. Change the name of the generated entity from **tblStudent** to **Student**.

Step 3: Download autocomplete widget from <http://jqueryui.com/download>. The following folders and files will be downloaded.

Name	Date modified	Type	Size
css	19/09/201...	File folder	
development-bundle	19/09/201...	File folder	
js	19/09/201...	File folder	
index.html	19/09/201...	Chrome H...	25 KB

Step 4: Open "js" folder copy "jquery-1.9.1.js" and "jquery-ui-1.10.3.custom.min.js" files and paste them in the "Scripts" folder of your mvc project. Now open "css" folder. This folder will be present in "ui-lightness" folder. Copy "images" folder and "jquery-ui-1.10.3.custom.min.css" file and paste them in "Content" folder in your mvc project. If you are following along, at this point your solution explorer should look as shown below.



Step 5: Right click on the "Controllers" folder and add "Home" controller. Copy and paste the following code. Please make sure to include "MVCDemo.Models" namespace.

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        DbContext db = new DbContext();
        return View(db.Students);
    }

    [HttpPost]
    public ActionResult Index(string searchTerm)
    {
        DbContext db = new DbContext();
        List<Student> students;
        if (string.IsNullOrEmpty(searchTerm))
        {
            students = db.Students.ToList();
        }
        else
        {
            students = db.Students
                .Where(s => s.Name.StartsWith(searchTerm)).ToList();
        }
        return View(students);
    }

    public JsonResult GetStudents(string term)
    {
        DbContext db = new DbContext();
        List<string> students = db.Students.Where(s => s.Name.StartsWith(term))
            .Select(x => x.Name).ToList();
        return Json(students, JsonRequestBehavior.AllowGet);
    }
}
```

```
}
```

Step 6: Right click on the **"Index"** action method in the **"HomeController"** and add **"Index"** view.
Copy and paste the following code.

```
@model IEnumerable<MVCDemo.Models.Student>
```

```
<link href="~/Content/jquery-ui-1.10.3.custom.min.css" rel="stylesheet" type="text/css" />
```

```
<script src="~/Scripts/jquery-1.9.1.js" type="text/javascript"></script>
```

```
<script src="~/Scripts/jquery-ui-1.10.3.custom.min.js" type="text/javascript"></script>
```

```
<script type="text/javascript">
```

```
$(function () {
    $("#txtSearch").autocomplete({
        source: '@Url.Action("GetStudents")',
        minLength: 1
    });
});
```

```
</script>
```

```
<div style="font-family:Arial">
```

```
@using (@Html.BeginForm())
```

```
{
    <b>Name: </b>
    @Html.TextBox("searchTerm", null, new { id = "txtSearch" })
    <input type="submit" value="Search" />
}
```

```
<table border="1">
```

```
<tr>
    <th>
        @Html.DisplayNameFor(model => model.Name)
    </th>
    <th>
        @Html.DisplayNameFor(model => model.Gender)
    </th>
    <th>
        @Html.DisplayNameFor(model => model.TotalMarks)
    </th>
</tr>
```

```
@foreach (var item in Model)
```

```
{
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Name)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Gender)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.TotalMarks)
        </td>
    </tr>
}
```

```
</table>
```

```
</div>
```

Part 98 - What is JavaScript minification

Suggested Videos

[Part 95 - OnBegin, OnComplete, OnSuccess and OnFailure properties of AjaxOptions class](#)

[Part 96 - LoadingElementDuration property of AjaxOptions class](#)
[Part 97 - Implement autocomplete textbox functionality in mvc](#)

What is JavaScript minification?

JavaScript minification is the process of reducing the size of JavaScript file, by removing comments, extra white spaces, new line characters and using shorter variable names.

What are the advantages of JavaScript minification?

As the minified JavaScript files are very small, they will be downloaded much faster and consumes less bandwidth. Search engines like google considers page load time as one of the parameters to rank the pages.

Is any of the functionality lost because of minification?

No, the functionality will be exactly the same as before.

Are there any tools available to minify JavaScript?

There are lot of free tools available on the web. Just GOOGLE by using "Minify JavaScript" search term.

What is the difference between jquery.js and jquery.min.js?

jquery.min.js is the minified version, where as jquery.js is the non-minified version. In your production website always use minified files as they download faster and consumes less bandwidth.

What is the downside of JavaScript minification?

They are harder to read and debug. However, for development and debugging we can use non-minified versions. Just before deployment, minify and use the minified versions on the production environment.

Part 99 - What is CDN - Content Delivery Network

Suggested Videos

[Part 96 - LoadingElementDuration property of AjaxOptions class](#)
[Part 97 - Implement autocomplete textbox functionality in mvc](#)
[Part 98 - What is JavaScript minification](#)

CDN stands for Content Delivery Network. CDN is a network of computers that exist all over the world. For a web application there are 2 ways we can make the jQuery library available

1. Have a copy of the jQuery library on your web server and reference it

```
<script src="/Scripts/jquery-1.7.1.min.js" type="text/javascript">
</script>
```

2. Reference the jQuery library from a CDN. (Microsoft, Google, or jQuery)

```
<script src="http://code.jquery.com/jquery-1.7.1.min.js" type="text/javascript">
</script>
```

Google CDN: <http://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js>

Microsoft CDN: <http://ajax.aspnetcdn.com/ajax/jquery/jquery-1.7.1.min.js>

Jquery CDN: <http://code.jquery.com/jquery-1.7.1.min.js>

Benefits or advantages of using a CDN:

1. Caching benefits - If other web sites use jquery and if the user has already visited those websites first, jquery file is cached. If the user then visits your website, the cached jquery file is used without having the need to download it again.

2. Speed benefits - The jquery file from the nearest geographical server will be downloaded.

3. Reduces network traffic - As the jQuery file is loaded from a CDN, this reduces the network traffic to your web server.

4. Parallelism benefit - There is a limitation on how many components can be downloaded in parallel. This limitation is per hostname. For example,

1. If the browser allows only 2 components to be downloaded in parallel per hostname and
2. If a web page requires 4 components and
3. If all of them are hosted on a single host and
4. If 2 components take 1 second to download

Then to download all the 4 components, it is going to take 2 seconds.

However, if 2 of the components are hosted on a different host(server), then all the 4 components can be downloaded in parallel and in 1 second.

Part 100 - What if CDN is down

Suggested Videos

[Part 97 - Implement autocomplete textbox functionality in mvc](#)

[Part 98 - What is JavaScript minification](#)

[Part 99 - What is CDN - Content Delivery Network](#)

In this video we will discuss

1. What if CDN is down?
2. How to fallback from CDN to use local copy of JQuery?

There are several benefits of referencing a resource from a CDN. We discussed these in [Part 99](#) of [asp.net mvc tutorial](#).

A CDN is an external resource and beyond our control. So, what if, CDN is down? How can we fall back from CDN to use local copy of JQuery.

The following code checks if jQuery is loaded and if it isn't, the jquery script will be loaded from our web server.

```
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js"
  type="text/javascript">
</script>
```

```
<script type="text/javascript">
  window.jQuery || document.write('<script src="/MVCDemo/Scripts/jquery-
1.7.1.min.js">\x3C/script>')
</script>
```

\x3C is hexadecimal for <

