**Entity Framework in ASP.NET MVC**

**Entity Framework in ASP.NET MVC**

**Creating the Database with Required Table and Data:**

Please use below SQL Script to create the  **MVC\_DB**Database, **Employee** table and populate the Employee table with test data:

--Create a database called MVC\_DB

**CREATE** **DATABASE** MVC\_DB

**GO**

--Use MVC\_DB

**USE** MVC\_DB

**GO**

--Create a Table called Employee

**CREATE** **TABLE** Employee

(

EmployeeId int Primary Key Identity(1,1),

Name nvarchar(50),

Gender nvarchar(10),

City nvarchar(50),

Salary decimal(18,2)

)

**GO**

--Insert some values into Employee Table

**INSERT** **INTO** Employee values('Pranaya','Male','Mumbai', 2000)

**INSERT** **INTO** Employee values('Sambit','Male','Chennai', 3000)

**INSERT** **INTO** Employee values('Priyanka','Female','Hydrabad', 4000)

**INSERT** **INTO** Employee values('Anurag','Male','Cheenai', 4000)

**INSERT** **INTO** Employee values('Subrat','Male','Mumbai', 3000)

**INSERT** **INTO** Employee values('Preety','Female','Mumbai', 2000)

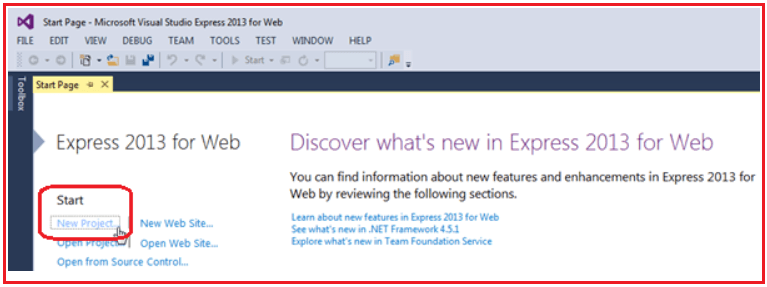
**INSERT** **INTO** Employee values('Trupti','Female','Hyderabad', 5000)

**GO**

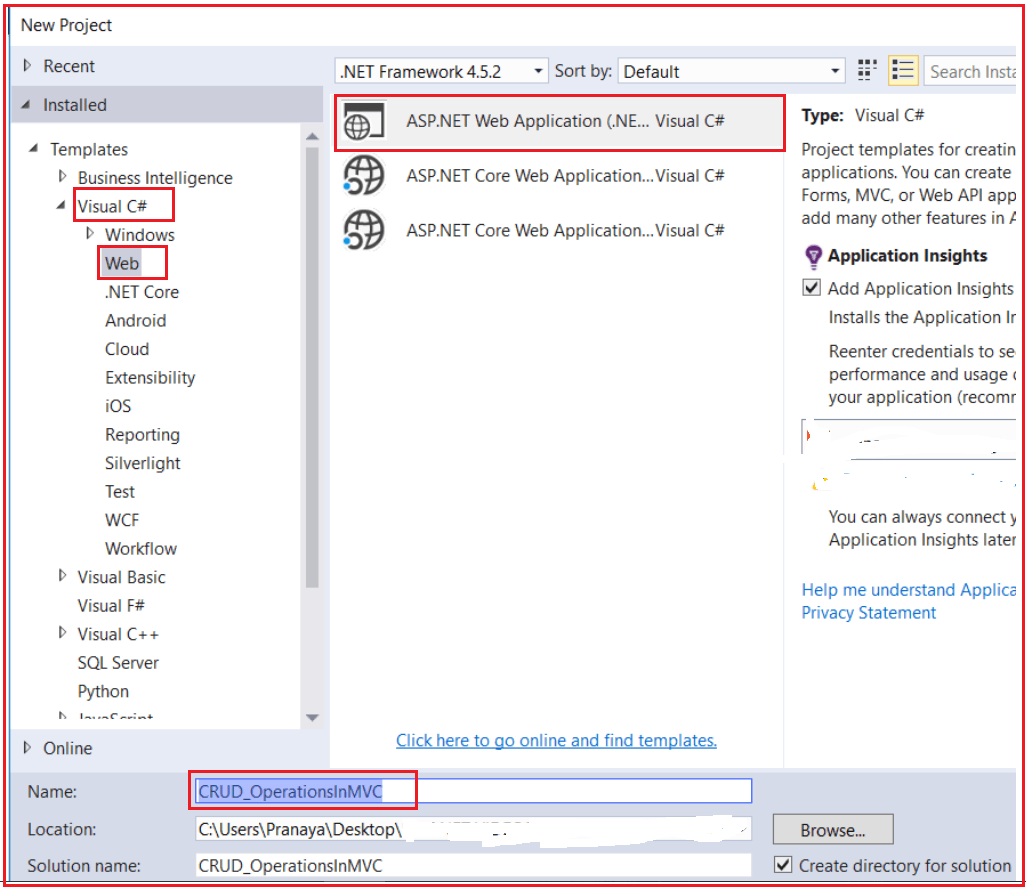
Let us understand how to use Entity Framework in ASP.NET MVC Application step by step.

**Step1: Create a new Empty ASP.NET MVC Application**

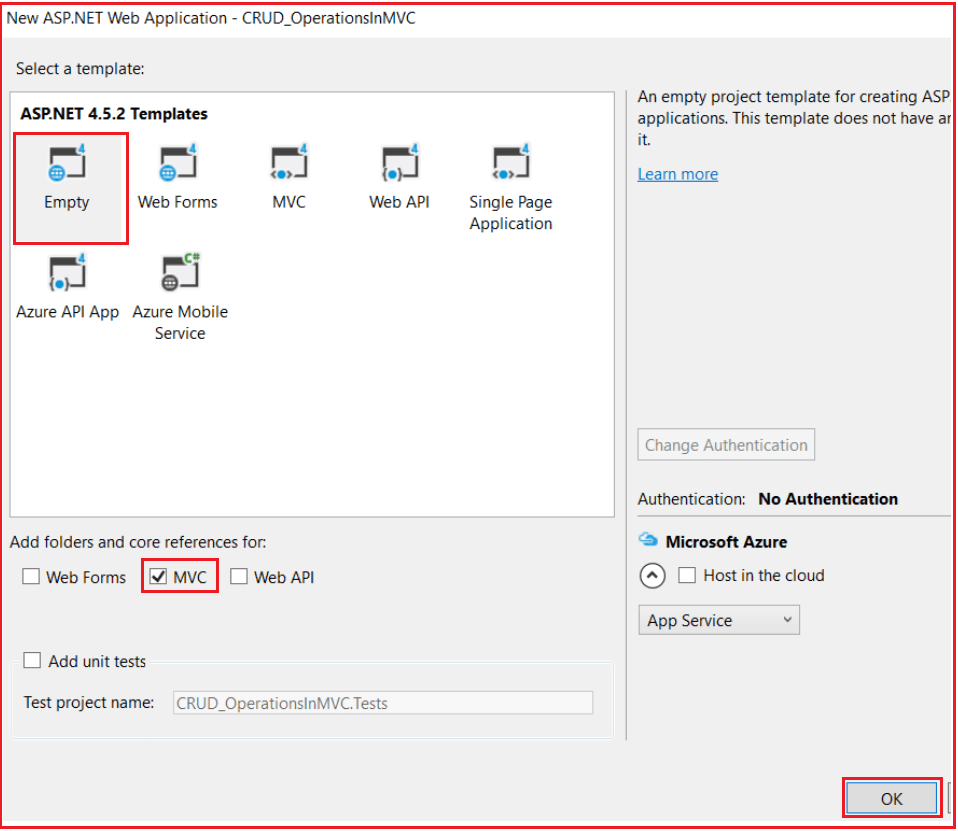
Open Visual Studio and click on the New Project link on the Startup page as shown below. Alternatively, you can also select the File menu -> New Project option from the Context menu.



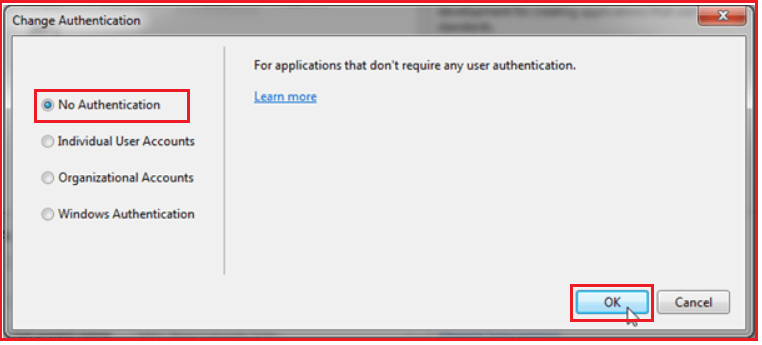
From the New Project window as shown in the below image, expand the Visual C# node and select Web from the left pane. From the middle pane select ASP.NET Web Application. Provide a meaningful name to your project such as **CRUD\_OperationsInMVC**. Select the location where you want to create the project by clicking on the Browse button. Finally, click on the OK button as shown in the below image.



Once you click on the OK button a new dialog will pop up for selecting project Templates as shown in the below image. From the below window, we are going to choose the Empty Project Template and Select MVC from Add Folders and core reference section.



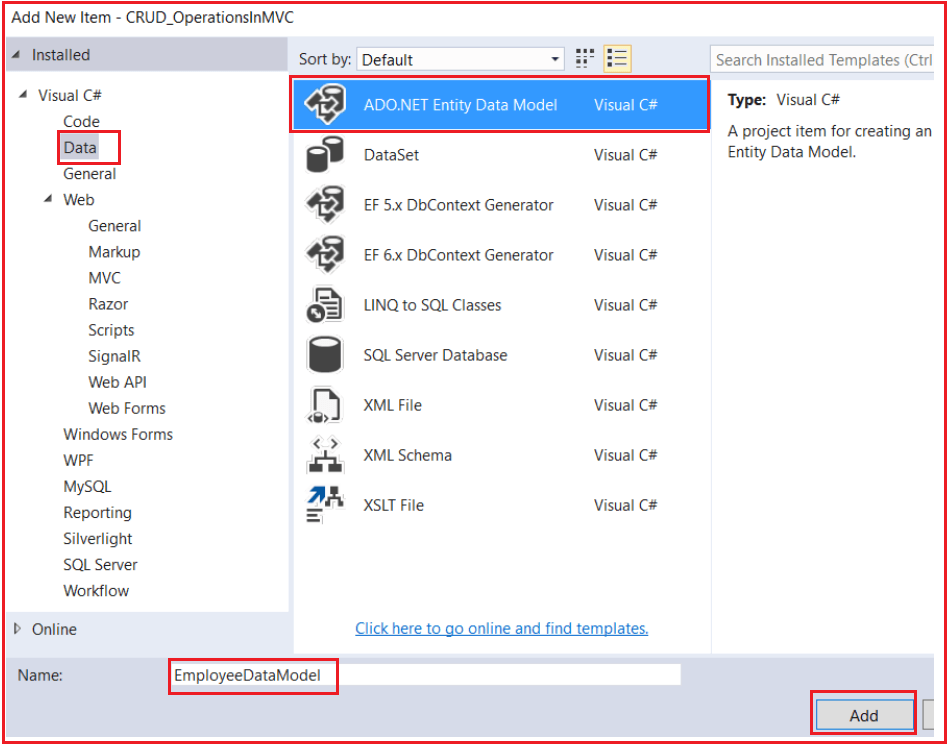
You can also change the authentication by clicking on the **Change Authentication** button. You can select the appropriate authentication mode for your application. Here, we are not going to have any authentication for our application. So select the **No Authentication** radio button and click **OK**. (By default, Individual User Accounts would be selected to authenticate users stored in the SQL Server database.)



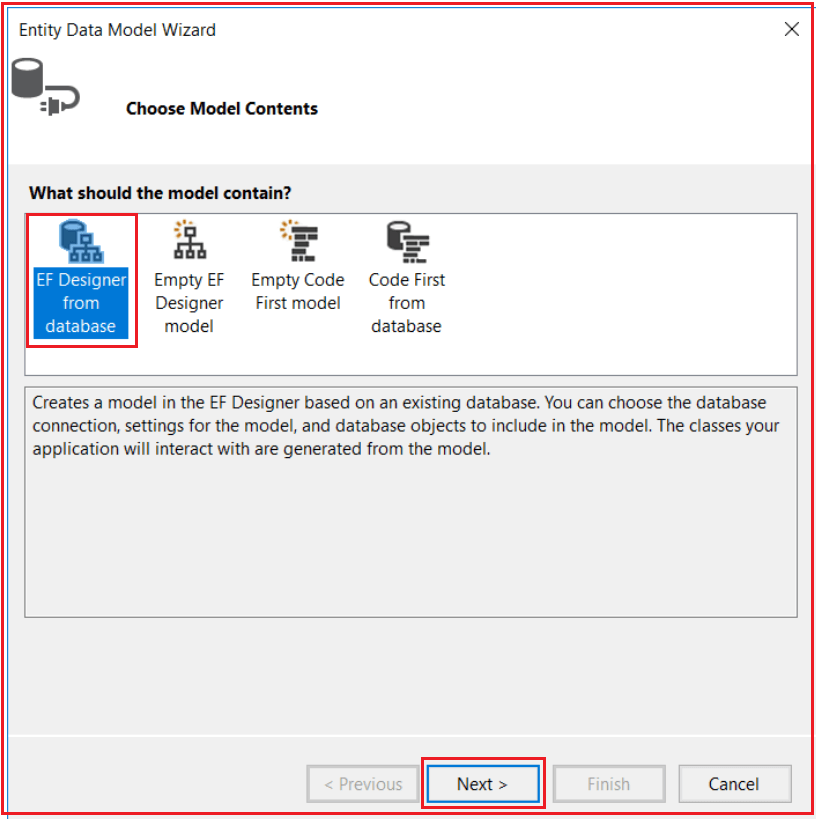
Wait for some time till Visual Studio creates a simple MVC project using the default template.

**Step2: Adding ADO.NET Entity Data Model in ASP.NET MVC Application**

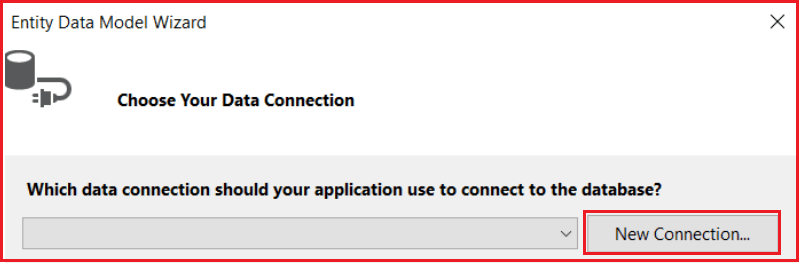
Right-Click on Models Folder, and then select **Add => New Item** from the context menu which will open the Add New Item window. From the “Add New Item” window, from the left pane expand **Installed => Visual C# => Data** option. From the middle pane select the ADO.NET Entity Data Model template. Provide a meaningful name to your data model such as **EmployeeDataModel**and finally click on the**Add**button as shown in the below image.



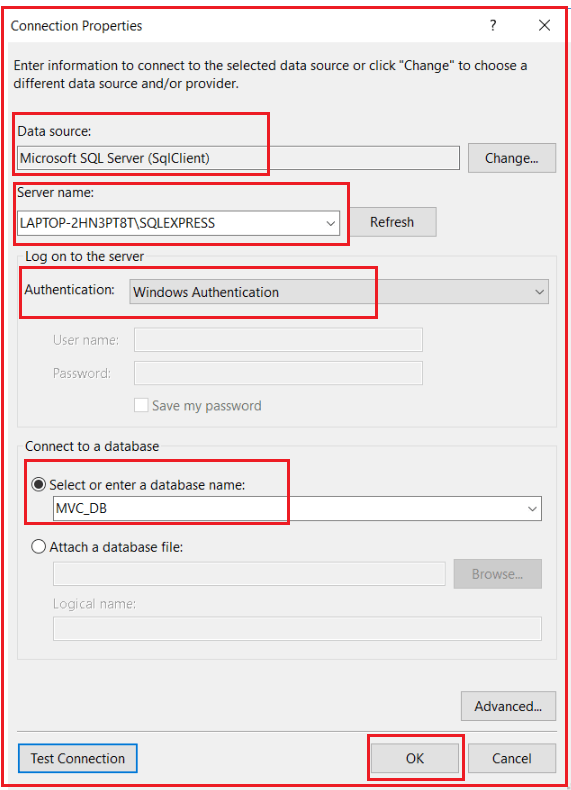
From the next Entity Data Model Wizard screen, as we are going to use the **Entity Framework Database First** approach, so select**EF Designer from Database**option and click on the **Next** button as shown in the image below.



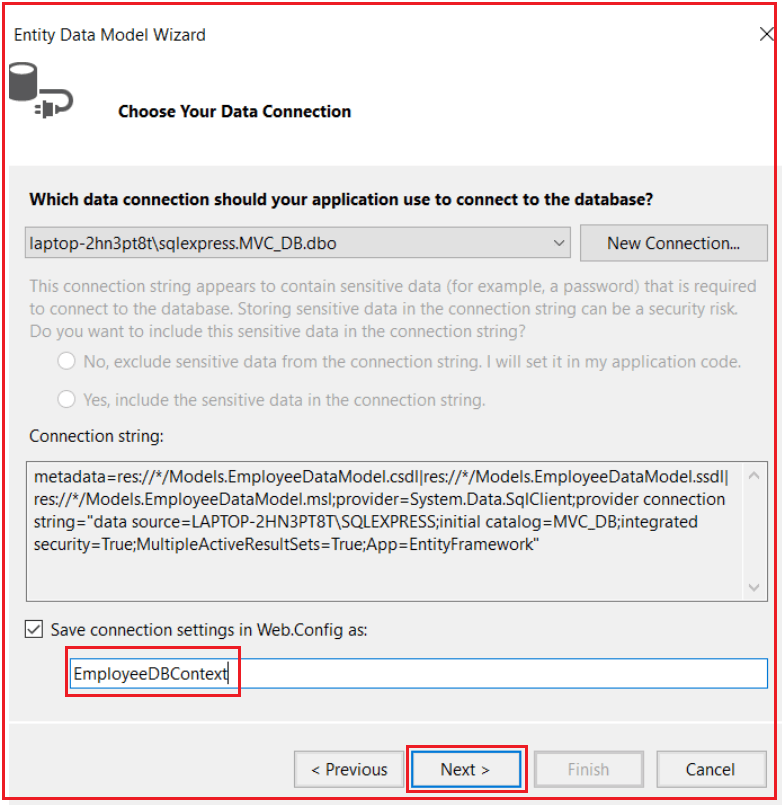
In the next step, click on the new connection From Choose your data connection wizard as shown below.



Provide the necessary details to communicate with the database such as**Server name**, select the **Authentication Type**, select the **Database** and click on the **Ok**button as shown below.

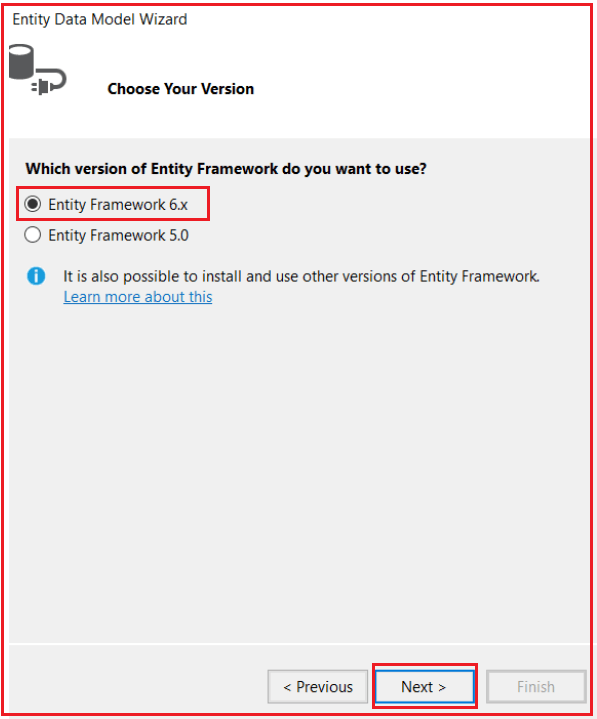


Then provide a meaningful name for the connection string that is going to save in the **web.config** file. Here, I am naming the connection string as **EmployeeDBContext** and then click on the **Next** button as shown in the below image.



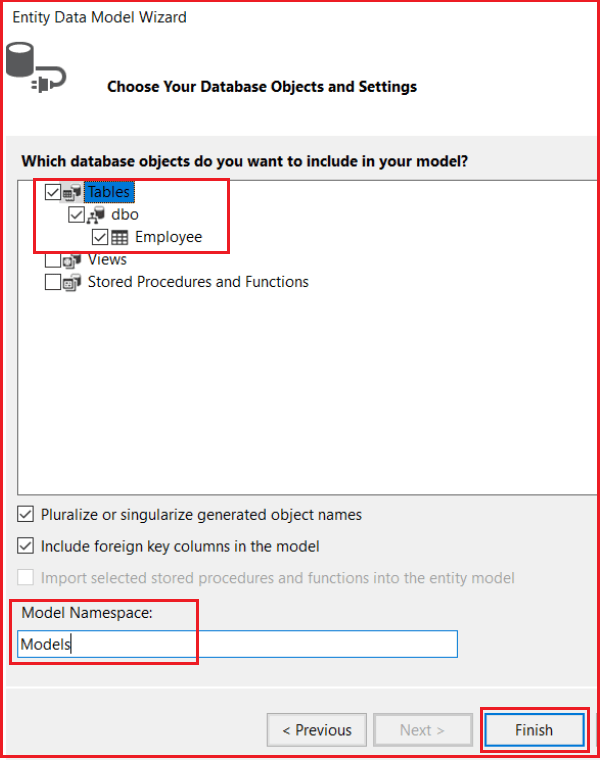
**Select the version of Entity Framework**

Here I am going to use **Entity Framework 6.x,** so I am selecting the **Entity Framework 6.x** radio button and click on the **Next** button as shown in the below image.

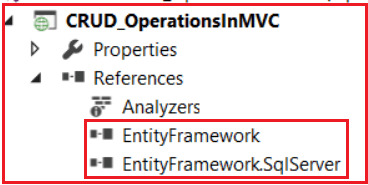


**Selecting the Database Objects**

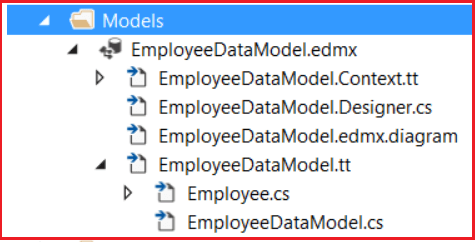
Then we need to select the database object for our application. As our database has one table, so we need to select that **Employee** Table. Provide a meaningful namespace to your **EDMX** file and finally click on the **Finish** button as shown in the below image.



Once you click on the **Finish** button, let’s see what the things are created by Entity Framework. It will add the reference to the Entity Framework in the reference folder.



The framework will create the EDMX file within the Models folder.



**It will create the Employee Model.**

To see the Employee Model click on **Employee.cs** file, this is inside the **EmployeeDataModel.tt** File

**namespace** *CRUD\_OperationsInMVC.Models*

**{**

**using** System;

**using** System.Collections.Generic;

**public** **partial** **class** Employee

**{**

**public** **int** EmployeeId **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Gender **{** **get**; **set**; **}**

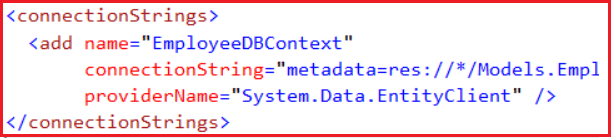
**public** string City **{** **get**; **set**; **}**

**public** Nullable**<decimal>** Salary **{** **get**; **set**; **}**

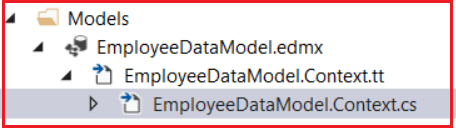
**}**

**}**

**It will create the connection string in the web.config file as shown below.**



It will create the **DBContext** class for us whose name is the same as the name of the connection string that is **EmployeeDBContext.**The **EmployeeDBContext**class derives from the **DbContext**class and is responsible for establishing a connection to the database. The **EmployeeDataModel.Context.cs** which is inside **EmployeedataModel.Context.tt** as sown below.



**Below is the auto-generated code for the EmployeeDBContext class**

**namespace** *CRUD\_OperationsInMVC.Models*

**{**

**using** System;

**using** System.Data.Entity;

**using** System.Data.Entity.Infrastructure;

**public** **partial** **class** EmployeeDBContext : DbContext

**{**

**public** EmployeeDBContext**()**

: **base(**"name=EmployeeDBContext"**)**

**{**

**}**

**protected** **override** **void** OnModelCreating**(**DbModelBuilder modelBuilder**)**

**{**

**throw** new UnintentionalCodeFirstException**()**;

**}**

**public** **virtual** DbSet**<**Employee**>** Employees **{** **get**; **set**; **}**

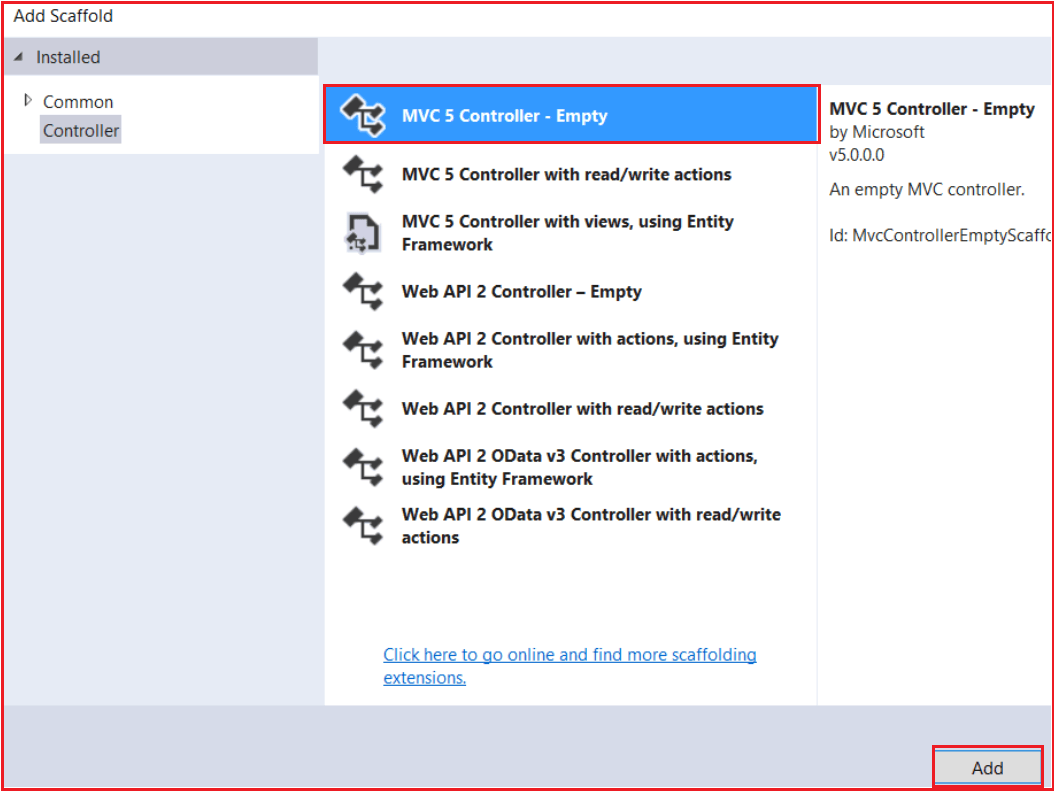
**}**

**}**

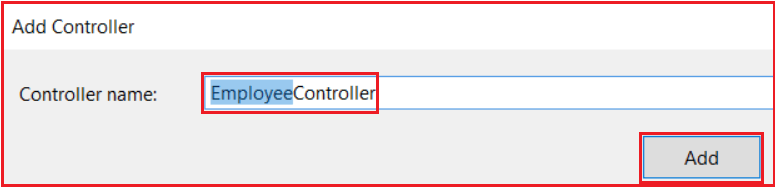
That’s it. We successfully created our Entity Data Model. Now it’s time to use this model in our application.

**Creating Controller:**

Right-Click on Controllers Folder. Select **Add => Controller => Select MVC5 Controller Empty**and then click on add button as shown in the below image.



In the next Screen provide the controller name as **EmployeeController** and click on the **Add** button as shown below.



Let’s modify the default route in the **RouteConfig** class so that when the application runs for the first time it will redirect to the Index Action method of Employee Controller.

After modifying the **RouteConfig** class the codes looks like as shown below

**namespace** *CRUD\_OperationsInMVC*

**{**

**public** **class** RouteConfig

**{**

**public** **static** **void** RegisterRoutes**(**RouteCollection routes**)**

**{**

routes.IgnoreRoute**(**"{resource}.axd/{\*pathInfo}"**)**;

routes.MapRoute**(**

name: "Default",

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

defaults: new **{** controller = "Employee", action = "Index", id = UrlParameter.Optional **}**

**)**;

**}**

**}**

**}**

**Let’s modify the index action method of employee controller to use entity framework as shown below**

**namespace** *CRUD\_OperationsInMVC.Controllers*

**{**

**public** **class** EmployeeController : Controller

**{**

**public** ActionResult Index**()**

**{**

EmployeeDBContext dbContext = new EmployeeDBContext**()**;

List**<**Employee**>** empList = dbContext.Employees.ToList**()**;

**return** View**(**empList**)**;

**}**

**}**

**}**

**Let’s create the index view.**

Right-click on the Index Action Method and click on Add View. Then paste the following code in the Index,cshtml file.

@model IEnumerable**<CRUD**\_OperationsInMVC.Models.Employee**>**

**<table** class="table"**>**

**<tr>**

**<th>**

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

**</th>**

**<th>**

@Html.DisplayNameFor(model => model.Gender)

**</th>**

**<th>**

@Html.DisplayNameFor(model => model.City)

**</th>**

**<th>**

@Html.DisplayNameFor(model => model.Salary)

**</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.City)

**</td>**

**<td>**

@Html.DisplayFor(modelItem => item.Salary)

**</td>**

**</tr>**

}

**</table>**

That’s it. Now run the application and you should get the result as expected as shown in the below image.



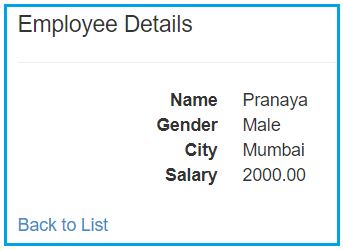
**ActionLink HTML Helper in ASP.NET MVC**

**ActionLink HTML Helper in ASP.NET MVC to Generate Hyperlinks**

Let us understand the need and use of ActionLink HTML Helper in ASP.NET MVC Application with an example. We want to display all the employees in a bulleted list as shown in the below image. Please have a look at the employee’s name, here we rendering the employee name as hyperlinks.



When we click on the above name hyperlink, then we need to redirect to the employee details page where we will display the full details of the employee as shown in the below image.



Again, when we click on the “**Back to List**” button, we will be redirected to the employee list page.

**Let’s see how to achieve this.**

Please modify the Employee controller as shown below. Here we add the Details action method to Employee Controller. Now we have two action methods i.e. Index and Details as shown below.

**namespace** *CRUD\_OperationsInMVC.Controllers*

**{**

**public** **class** EmployeeController : Controller

**{**

**public** ActionResult Index**()**

**{**

EmployeeDBContext dbContext = new EmployeeDBContext**()**;

List**<**Employee**>** empList = dbContext.Employees.ToList**()**;

**return** View**(**empList**)**;

**}**

**public** ActionResult Details**(int** id**)**

**{**

EmployeeDBContext dbContext = new EmployeeDBContext**()**;

Employee employee = dbContext.Employees.FirstOrDefault**(**x =**>** x.EmployeeId == id**)**;

**return** View**(**employee**)**;

**}**

**}**

**}**

**ActionLink HTML Help Method:**

There are many overloaded versions available for the ActionLink HTML Helper method. But we are going to use the following overloaded version which takes link text, action method name, and route values as parameters. The first parameter specifies that it is an extension method and we can access this using the HtmlHelper object.

**public static MvcHtmlString ActionLink(this HtmlHelper htmlHelper, string linkText, string actionName, object routeValues);**

**Modifying the Index View:**

In our example, the Index action method retrieves the list of employees which is then passed to the Index view for rendering. So, let’s change the Index view as shown below which displays the employee names as Hyperlinks using the ActionLink HTML Helper method.

@model IEnumerable**<CRUD**\_OperationsInMVC.Models.Employee**>**

@using CRUD\_OperationsInMVC.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>**

As you can see in the above code, we set the **@model** to **IEnumerable<CRUD\_OperationsInMVC.Models.Employee>**and we are generating the hyperlinks using **Html.ActionLink HTML** helper method. This method takes three parameters, the first parameter is the link text that will be rendered in the browser, the second parameter is the action method name and the third parameter is the route values that will be passed to the Details action method.

**Adding Details view:**

The Details action method takes a parameter as EmployeeId and then gets the employee details based on the Employee ID. Once it gets the employee details then it passes that employee object to the view and then the view displays the employee details. Let’s Add Details View and then copy and paste the following codes into it.

@model CRUD\_OperationsInMVC.Models.Employee

**<div>**

**<h4>**Employee Details**</h4>**

**<hr** **/>**

**<dl** class="dl-horizontal"**>**

**<dt>**

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

**</dt>**

**<dd>**

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

**</dd>**

**<dt>**

@Html.DisplayNameFor(model => model.Gender)

**</dt>**

**<dd>**

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

**</dd>**

**<dt>**

@Html.DisplayNameFor(model => model.City)

**</dt>**

**<dd>**

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

**</dd>**

**<dt>**

@Html.DisplayNameFor(model => model.Salary)

**</dt>**

**<dd>**

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

**</dd>**

**</dl>**

**</div>**

**<p>**

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

**</p>**

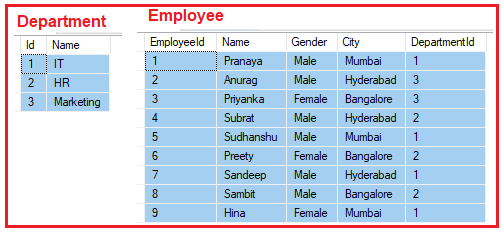
That’s it. We are done with our implementation. So now it’s time to run the application and see everything is working as expected. In this article, we just see how to use the **ActionLink HTML Helper** in the ASP.NET MVC application. In our upcoming articles, we will discuss the [**HTML Helpers in detail**](https://dotnettutorials.net/lesson/html-helpers-asp-net-mvc/) with some real-time examples.

# Working with Multiple Tables in ASP.NET MVC using Entity Framework

## ****Working with Multiple Tables in ASP.NET MVC using Entity Framework****

##### ****Database tables used in this demo:****

We are going to work with the below two tables i.e. Department and Employee.



Please use the below SQL script to create and populate Department and Employee tables with the required test data.

-- Create Department Table

**Create** **table** Department

(

Id int primary key identity,

Name nvarchar(50)

)

**GO**

-- Insert some test data into Department table

**Insert** **into** Department values('IT')

**Insert** **into** Department values('HR')

**Insert** **into** Department values('Marketing')

**GO**

<br>-- First Drop the Existing Employee table<br>DROP Table Employee;<br>

-- Create Employee Table

**Create** **table** Employee

(

EmployeeId int Primary Key Identity(1,1),

Name nvarchar(50),

Gender nvarchar(10),

City nvarchar(50),

DepartmentId int

)

**GO**

-- Add Foreign Key into Employee Table Reference to the Department Table

**Alter** **table** Employee

add foreign key (DepartmentId)

references Department(Id)

**GO**

-- Insert Some Test data into Employee Table

**Insert** **into** Employee values('Pranaya','Male','Mumbai',1)

**Insert** **into** Employee values('Anurag','Male','Hyderabad',3)

**Insert** **into** Employee values('Priyanka','Female','Bangalore',3)

**Insert** **into** Employee values('Subrat','Male','Hyderabad',2)

**Insert** **into** Employee values('Sudhanshu','Male','Mumbai',1)

**Insert** **into** Employee values('Preety','Female','Bangalore',2)

**Insert** **into** Employee values('Sandeep','Male','Hyderabad',1)

**Insert** **into** Employee values('Sambit','Male','Bangalore',2)

**Insert** **into** Employee values('Hina','Female','Mumbai',1)

**GO**

##### ****Example to understand working with multiple tables in MVC Application:****

Following is our business requirement.

1. We need to display all the departments from the Department table. The Department names should be rendered as hyperlinks. This is going to be our department list page.
2. On clicking the department name link, all the employees in that particular department should be displayed. The employee names also here going to be rendered as hyperlinks. This is going to be our employee list page.
3. When the user clicks on the employee name link the full details of that employee should be displayed in the browser. This is going to be our employee details page.
4. A link should also be provided on the employee full details page to navigate back to the Employee List page. Along the same lines, a link should also be provided on the employee list page to navigate back to the Departments’s list page.

###### **The following image gives you the overall workflow of our requirement**



**Note:** We are going to work with the same example that we started in our previous two articles. So please read the below two articles before proceeding to this article.

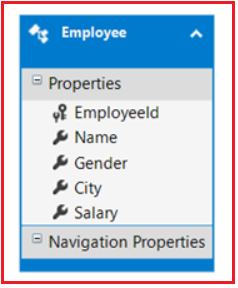
[**Entity Framework in ASP.NET MVC**](https://dotnettutorials.net/lesson/entity-framework-in-asp-dot-net-mvc/)

[**Generating hyperlinks using Action Link HTML helper**](https://dotnettutorials.net/lesson/actionlink-html-helper-mvc/)

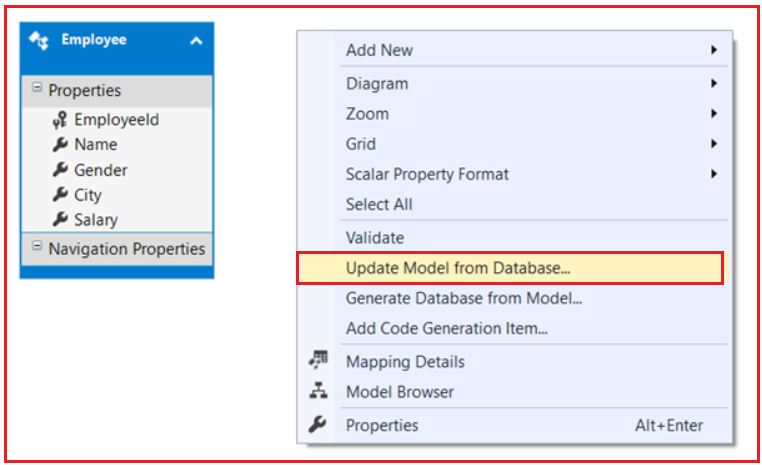
To implement the above example first we need to update the EmployeeDataModel.edmx file.

##### Update the EDMX file

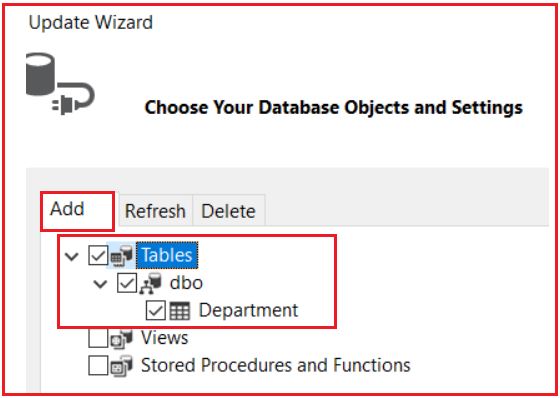
Double click on the **EmployeeDataModel.edmx** file which is in the Models folder. Once you click on the edmx file the following screen will open.



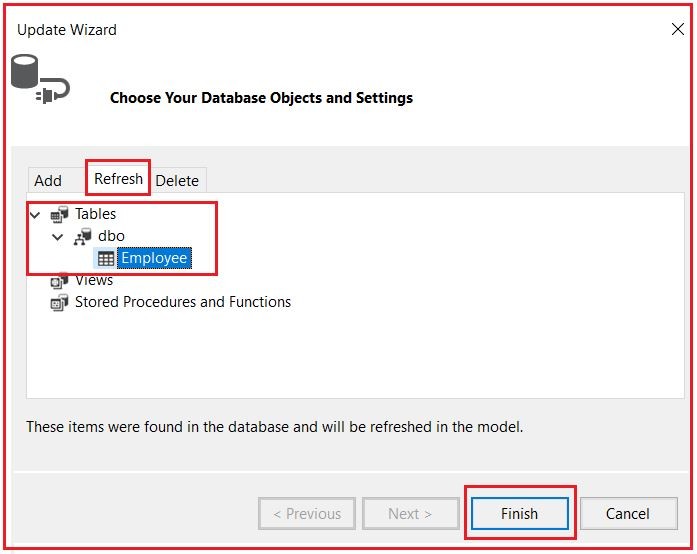
Right-click anywhere in the edmx file and then click on the “**update model from the database”** option as shown in the below image.



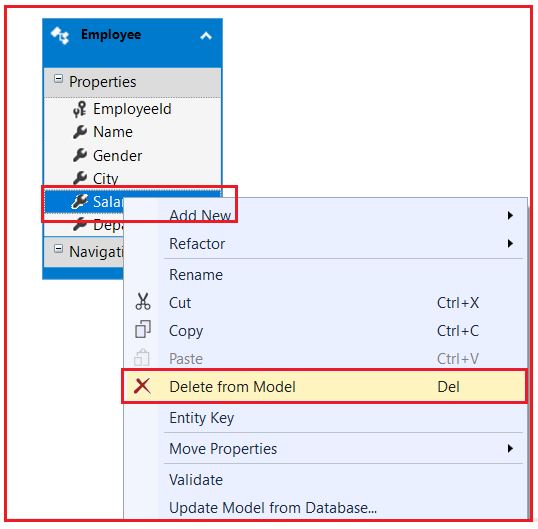
Then choose the add button and then select the Department table as shown in the below image.



Next, choose the Refresh button and select the Employee table and click on the Finish button as shown in the below image.



Once you click on the Finish button, you will get an error saying Salary Property is not mapped. Simply select the Salary Property from the Employee Model of the edmx file, right-click on it, and then click on delete from the model as shown in the below image.



That’s it. Save the edmx file and build the solution. Let’s have a look of the files that are generated and modified by Entity Framework.

##### Department.cs (This file is added by Entity Framework)

**namespace** *CRUD\_OperationsInMVC.Models*

**{**

**using** System;

**using** System.Collections.Generic;

**public** **partial** **class** Department

**{**

**[**System.Diagnostics.CodeAnalysis.SuppressMessage**(**"Microsoft.Usage", "CA2214:DoNotCallOverridableMethodsInConstructors"**)]**

**public** Department**()**

**{**

this.Employees = new HashSet**<**Employee**>()**;

**}**

**public** **int** Id **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**[**System.Diagnostics.CodeAnalysis.SuppressMessage**(**"Microsoft.Usage", "CA2227:CollectionPropertiesShouldBeReadOnly"**)]**

**public** **virtual** ICollection**<**Employee**>** Employees **{** **get**; **set**; **}**

**}**

**}**

##### Employee.cs (This file is modified by Entity Framework)

**namespace** *CRUD\_OperationsInMVC.Models*

**{**

**using** System;

**using** System.Collections.Generic;

**public** **partial** **class** Employee

**{**

**public** **int** EmployeeId **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Gender **{** **get**; **set**; **}**

**public** string City **{** **get**; **set**; **}**

**public** Nullable**<int>** DepartmentId **{** **get**; **set**; **}**

**public** **virtual** Department Department **{** **get**; **set**; **}**

**}**

**}**

##### EmployeeDataModel.Context.cs (This file is modified by Entity framework)

**namespace** *CRUD\_OperationsInMVC.Models*

**{**

**using** System;

**using** System.Data.Entity;

**using** System.Data.Entity.Infrastructure;

**public** **partial** **class** EmployeeDBContext : DbContext

**{**

**public** EmployeeDBContext**()**

: **base(**"name=EmployeeDBContext"**)**

**{**

**}**

**protected** **override** **void** OnModelCreating**(**DbModelBuilder modelBuilder**)**

**{**

**throw** new UnintentionalCodeFirstException**()**;

**}**

**public** **virtual** DbSet**<**Employee**>** Employees **{** **get**; **set**; **}**

**public** **virtual** DbSet**<**Department**>** Departments **{** **get**; **set**; **}**

**}**

**}**

These are the changes done by entity framework.

##### ****Add Department Controller****

Right click on the “Controllers” folder and add a MVC5 Empty Controller with the name DepartmentController and then Copy and paste the following code into it.

**namespace** *CRUD\_OperationsInMVC.Controllers*

**{**

**public** **class** DepartmentController : Controller

**{**

**public** ActionResult Index**()**

**{**

EmployeeDBContext dbContext = new EmployeeDBContext**()**;

List**<**Department**>** listDepartments = dbContext.Departments.ToList**()**;

**return** View**(**listDepartments**)**;

**}**

**}**

**}**

##### ****Adding Index View Of Department Controller:****

Right-click on the Index() action method in DepartmentController class and select “Add View” from the context menu. Set all the default values as it is. Copy and paste the following code in Index.cshtml view file in Department folder.

@using CRUD\_OperationsInMVC.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>**

In the above code, we are using the following ActionLink HTML helper extension which takes five parameters.

**public static MvcHtmlString ActionLink(this HtmlHelper htmlHelper, string linkText, string actionName, string controllerName, object routeValues, object htmlAttributes);**

##### ****Modify Employee Controller:****

Add “departmentId” parameter to Index() action method in “EmployeeController” class. Use the “departmentId” parameter to filter the list of employees. After changes Employee Controller looks as shown below.

**namespace** *CRUD\_OperationsInMVC.Controllers*

**{**

**public** **class** EmployeeController : Controller

**{**

**public** ActionResult Index**(int** departmentId**)**

**{**

EmployeeDBContext dbContext = new EmployeeDBContext**()**;

List**<**Employee**>** employees = dbContext.Employees.Where**(**emp =**>** emp.DepartmentId == departmentId**)**.ToList**()**;

**return** View**(**employees**)**;

**}**

**public** ActionResult Details**(int** id**)**

**{**

EmployeeDBContext dbContext = new EmployeeDBContext**()**;

Employee employee = dbContext.Employees.FirstOrDefault**(**x =**>** x.EmployeeId == id**)**;

**return** View**(**employee**)**;

**}**

**}**

**}**

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

@model IEnumerable**<CRUD**\_OperationsInMVC.Models.Employee**>**

@using CRUD\_OperationsInMVC.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>**

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

**</div>**

##### ****Modify the Details.cshtml file that is present in Employee Folder.****

Here, we are just removing the Salary property.

@model CRUD\_OperationsInMVC.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>**

**<p>**

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

**</p>**

Change the RouteConfig file as shown below where we provide the default Route as Index Action Method of Department Controller.

**namespace** *CRUD\_OperationsInMVC*

**{**

**public** **class** RouteConfig

**{**

**public** **static** **void** RegisterRoutes**(**RouteCollection routes**)**

**{**

routes.IgnoreRoute**(**"{resource}.axd/{\*pathInfo}"**)**;

routes.MapRoute**(**

name: "Default",

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

defaults: new **{** controller = "Department", action = "Index", id = UrlParameter.Optional **}**

**)**;

**}**

**}**

**}**

That’s it we have done with our implementation. Now run the application and see everything is working as expected or not.

# Business Objects as Model in ASP.NET MVC

## ****Business Objects as Model in ASP.NET MVC****

##### ****Create an Empty ASP.NET MVC Application:****

First, create an Empty MVC Application with the name **MVC\_DEMO.**Once you create the application, then create a controller with the name as **HomeController** within the Controllers Folder and then copy and paste the below code in it.

**namespace** *MVC\_DEMO.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**public** ViewResult Index**()**

**{**

ViewData**[**"Countries"**]** = new List**<**string**>()**

**{**

"India",

"US",

"Canada",

"Brazil"

**}**;

**return** View**()**;

**}**

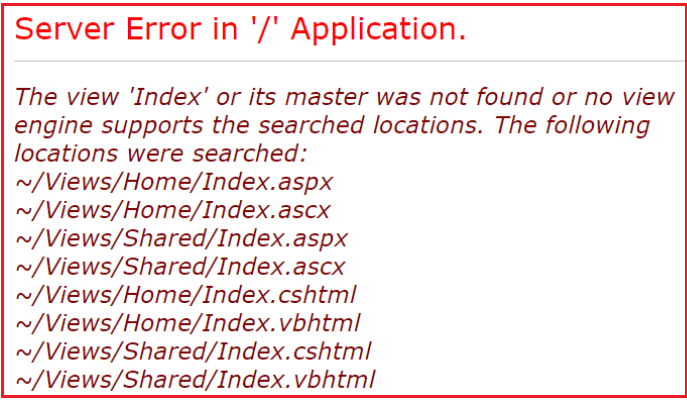
**}**

**}**

The following URL will invoke the **Index()** action method of the **HomeController**. Notice that, the HomeController class is inherited from the base **Controller** class which in turn inherits from **ControllerBase** class. ControllerBase, in turn, inherits from the **IController** interface.

**http://localhost:53657/Home/Index**

The**return View()** statement within the index action method by default looks for a view with the name “**Index**” in “**/Views/Home/**” and “**/Views/Shared/**” folders. If a view with the name “**Index**” is not found then we will get the following error.



So, In the ASP.NET MVC application, there are several conventions that we need to follow while working. For example, controllers need to have the word controller in them and should implement the IController interface either directly or indirectly. Views should be placed in a specific location that MVC can find them.

But with models, there are no strict rules. In fact, the “**Models**” folder is optional and they can place anywhere within the application. They can even be present in a separate project. Let’s now turn our attention to using business objects as the model. We will be using the table “**Employee**” for this demo.

##### ****Step1: Create the Required Database****

Please use the below SQL script to create and populate the Employee table with some test data. Also, we are creating one stored procedure to retrieve the employee data.

-- Create Employee Table

**Create** **table** Employee

(

Id int Primary Key Identity(1,1),

Name nvarchar(50),

Gender nvarchar(10),

City nvarchar(50),

Salary decimal(18,2),

DateOfBirth DateTime

)

**GO**

-- Insert some test data into Employee table

**Insert** **into** Employee values('Pranaya','Male','Mumbai',4000,'02/03/1977')

**Insert** **into** Employee values('Anurag','Male','Hyderabad',5000,'04/06/1979')

**Insert** **into** Employee values('Priyanka','Female','Bangalore',1000,'01/05/1979')

**Insert** **into** Employee values('Subrat','Male','Hyderabad',2000,'03/07/1981')

**Insert** **into** Employee values('Sudhanshu','Male','Mumbai',3000,'02/04/1978')

**Insert** **into** Employee values('Preety','Female','Bangalore',4000,'02/03/1974')

**Insert** **into** Employee values('Sandeep','Male','Hyderabad',5000,'04/06/1972')

**Insert** **into** Employee values('Sambit','Male','Bangalore',6000,'07/05/1975')

**Insert** **into** Employee values('Hina','Female','Mumbai',3000,'09/08/1976')

**GO**

--Stored procedure to retrieve data

**Create** procedure spGetAllEmployees

**as**

Begin

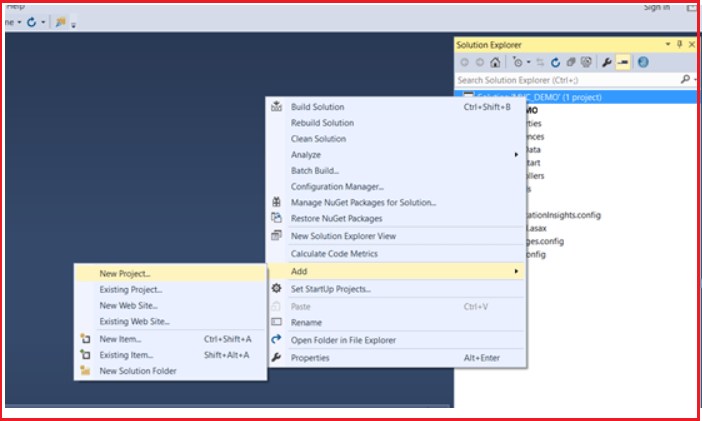
**Select** Id, Name, Gender, City, Salary, DateOfBirth

**from** Employee

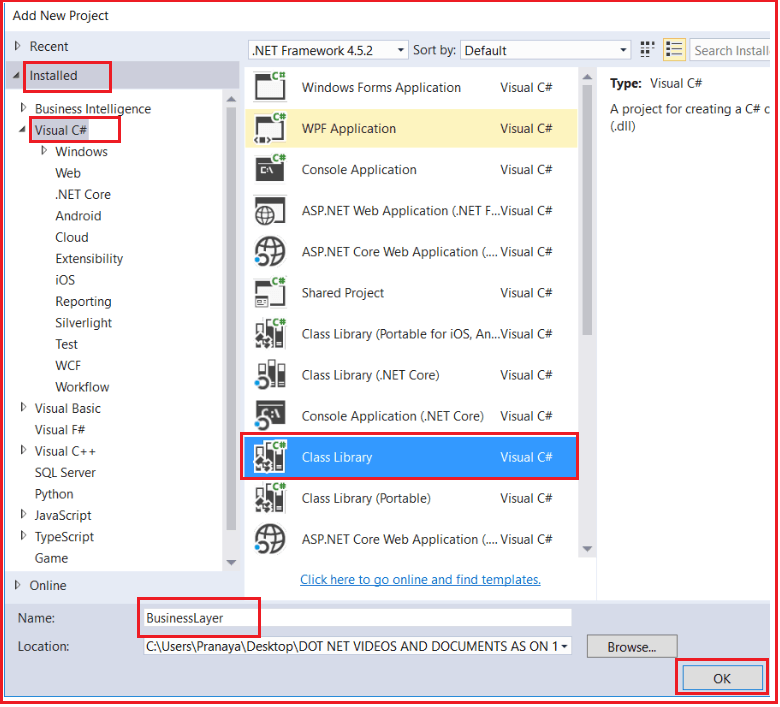
End

**GO**

**Step2: Add a Class Library project with Name=”BusinessLayer” to the Solution**  
Right-click on the Solution Folder => Add => New Project as shown in the below image.



From the new project window, select **Visual C#** from Installed Template from the left pane and then select **Class Library** Template from the middle pane. Provide the name as **BusinessLayer** and click on the **OK** as shown in the below image.



Now it will add the BusinessLayer class library project to our existing solution.

##### ****Step3: Adding Models to the Class Library Project****

Right-click on the business layer class library project and add a class file with the name **Employee.cs**. Once you created the **Employee** class then copy and paste the following code into it. The following class is very straightforward. We simply created the class with 6 properties.

**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** **decimal** Salary **{** **get**; **set**; **}**

**public** DateTime DateOfBirth **{** **get**; **set**; **}**

**}**

**}**

##### ****Step4: Adding Required References****

Right-click on the “**References**” folder of the business layer class library project and add a reference to the “**System.Configuration**” assembly. This is required as we want to read the connection string from the web config file using the ConfigurationManager class and this class belongs to **System.Configuration namespace.**

##### ****Step5: Adding EmployeeBusinessLayer class****

Right-click on the business layer class library project and add a class file with the name **EmployeeBusinessLayer.cs**. Once you created the **EmployeeBusinessLayer** class then copy and paste the following code into it. In the following class, we define one method i..e. GetAllEmployess(). This method is used to get the employee details from the database. The following code is self-explained, so please go through the comment lines.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Configuration;*

**using** *System.Data;*

**using** *System.Data.SqlClient;*

**namespace** *BusinessLayer*

**{**

**public** **class** EmployeeBusinessLayer

**{**

**public** List**<**Employee**>** GetAllEmployess**()**

**{**

//Reads the connection string from web.config file. The connection string name is DBCS

string connectionString = ConfigurationManager.ConnectionStrings**[**"DBCS"**]**.ConnectionString;

//Create List of employees collection object which can store list of employees

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

//Establish the Connection to the database

**using** **(**SqlConnection con = new SqlConnection**(**connectionString**))**

**{**

//Creating the command object by passing the stored procedure that is used to

//retrieve all the employess from the tblEmployee table and the connection object

//on which the stored procedure is going to execute

SqlCommand cmd = new SqlCommand**(**"spGetAllEmployees", con**)**;

//Specify the command type as stored procedure

cmd.CommandType = CommandType.StoredProcedure;

//Open the connection

con.Open**()**;

//Execute the command and stored the result in Data Reader as the method ExecuteReader

//is going to return a Data Reader result set

SqlDataReader rdr = cmd.ExecuteReader**()**;

//Read each employee from the SQL Data Reader and stored in employee object

**while** **(**rdr.Read**())**

**{**

//Creating the employee object to store employee information

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.Salary = Convert.ToDecimal**(**rdr**[**"Salary"**])**;

employee.DateOfBirth = Convert.ToDateTime**(**rdr**[**"DateOfBirth"**])**;

//Adding that employee into List of employees collection object

employees.Add**(**employee**)**;

**}**

**}**

//Return the list of employees that is stored in the list collection of employees

**return** employees;

**}**

**}**

**}**

##### ****Step6: Adding a Reference to class Library Project in ASP.NET MVC Application:****

Right-click on the “**References**” folder of the “**MVC\_DEMO**” project and add a reference to the “**BusinessLayer**” class library project. Then Include a connection string with name = “**DBCS**” in **Web.Config** file as shown below.

**<connectionStrings>**

**<add** name="DBCS"

connectionString="Data Source=LAPTOP-2HN3PT8T\SQLEXPRESS;Initial Catalog=MVC\_DB;Integrated Security=True"

providerName="System.Data.SqlClient"**/>**

**</connectionStrings>**

##### ****Step8: Creating Controller****

Right-click on the “**Controllers**” folder and add a Controller with the name “**EmployeeController**” and then copy and paste the following code into it. In the below controller we have only one action method i.e. Index. This method creates an instance of **EmoloyeeBusinessLayer** class and then calls the **GetAllEmployees** method which will return the list of employees. The list of employees is then handed over to the **Index** view.

**using** *BusinessLayer;*

**using** *System.Collections.Generic;*

**using** *System.Web.Mvc;*

**namespace** *MVC\_DEMO.Controllers*

**{**

**public** **class** EmployeeController : Controller

**{**

**public** ActionResult Index**()**

**{**

EmployeeBusinessLayer employeeBusinessLayer = new EmployeeBusinessLayer**()**;

List**<**Employee**>** employees = employeeBusinessLayer.GetAllEmployess**()**;

**return** View**(**employees**)**;

**}**

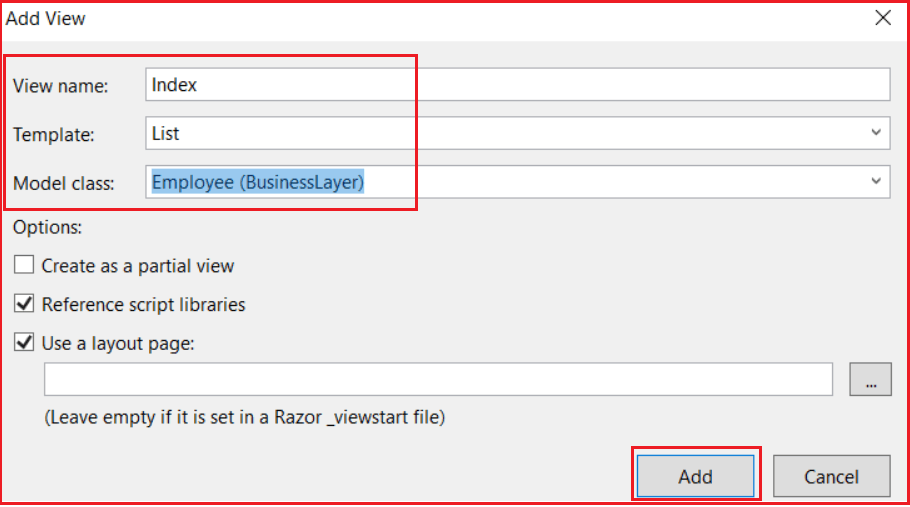
**}**

**}**

##### ****Step9: Adding Index View****

Right-click on the **Index()** action method in the “**EmployeeController**” class and then select “**Add View**” from the context menu. Set

**View name = Index**  
**Model class = Employee (BusinessLayer)**  
**Template = List**  
**Click on the “Add” button as shown below**



###### **Change the RouteConfig.cs as shown below**

We are setting the controller as Employee and the default action method as Index.

**namespace** *MVC\_DEMO*

**{**

**public** **class** RouteConfig

**{**

**public** **static** **void** RegisterRoutes**(**RouteCollection routes**)**

**{**

routes.IgnoreRoute**(**"{resource}.axd/{\*pathInfo}"**)**;

routes.MapRoute**(**

name: "Default",

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

defaults: new **{** controller = "Employee", action = "Index", id = UrlParameter.Optional **}**

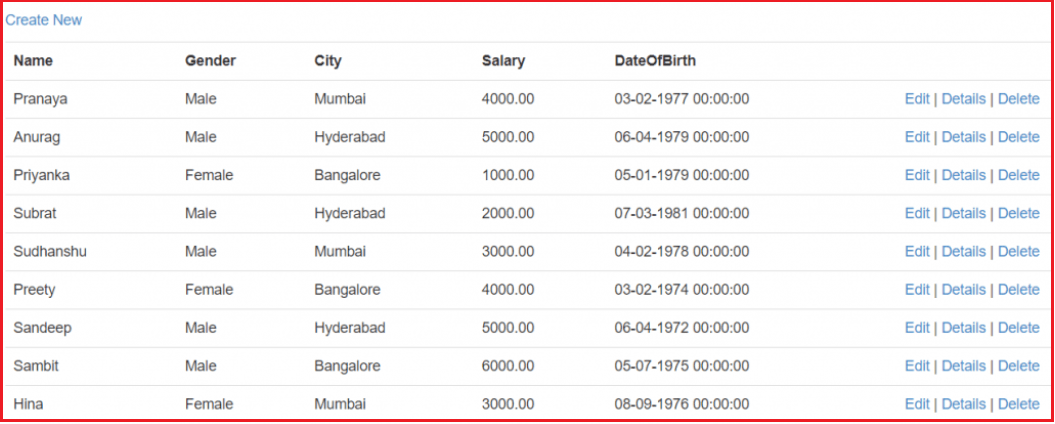
**)**;

**}**

**}**

**}**

Set MVC\_DEMO as your startup project and run the application, then navigate to **http://localhost:54094/Employee/Index**. It should display the output as expected as shown in the below image.

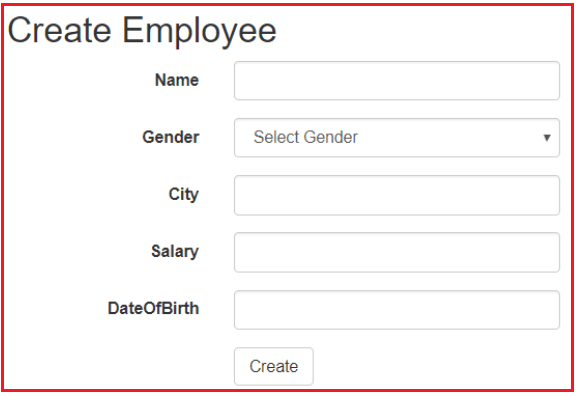


# FormCollection in ASP.NET MVC

## ****FormCollection in ASP.NET MVC Application****

##### ****Creating a view to inserting data using FormCollection in MVC****

We want to create a view as shown below to create a new employee.



##### ****Creating the Create Action Method:****

Please Copy and paste the following “Create” action method in the EmployeeController class. Please note the following action method is decorated with the **HttpGet** attribute. This makes the Create Action Method to respond only to the “**GET**” request.

**[**HttpGet**]**

**public** ActionResult Create**()**

**{**

**return** View**()**;

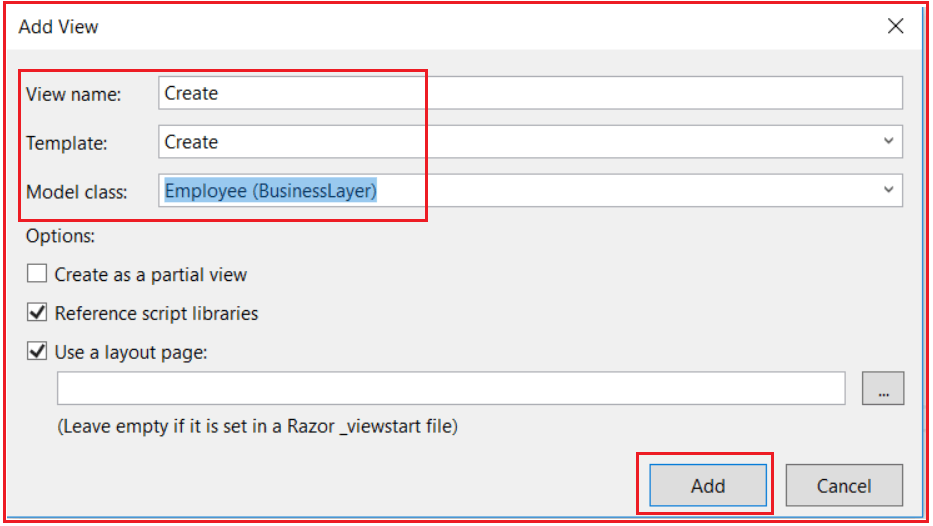
**}**

##### ****Adding the Create View:****

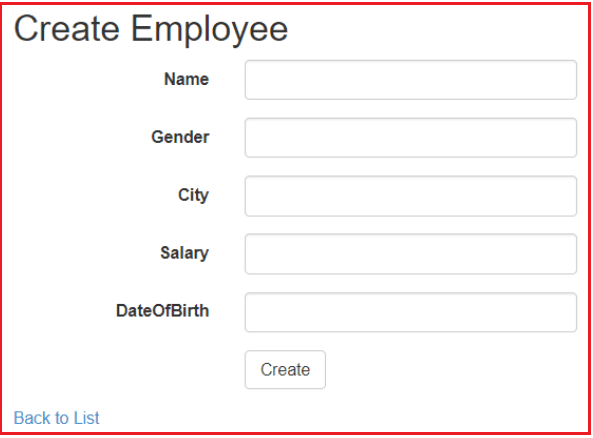
Now let’s add the “**Create**” view. To do this right-click on the “**Create**” action method and select “**Add View**” from the context menu. Set

**View name = “Create”**  
**Template = “Create”**  
**Model class = Employee (BusinessLayer)**

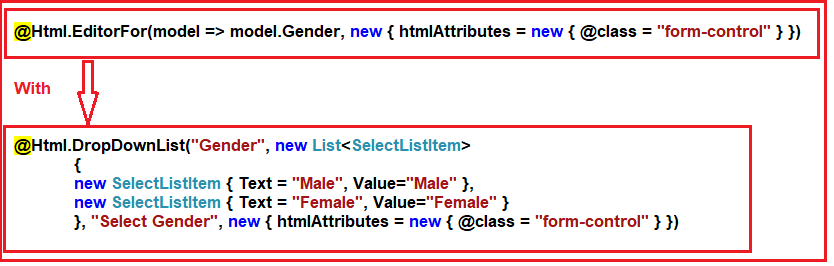
Click on the “**Add**” button as shown below.



At this point “**Create.cshtml**” view will be added to the “**Employee**” folder. Run the application and navigate to the  URL “**http://localhost:54094/Employee/Index**“. Click on the “**Create New**” link. It will navigate to the URL **http://localhost:54094/Employee/Create**URL and will display the following page.



A form with text boxes to add a new employee is rendered. For employee “**Gender**“, it is ideal to have a dropdown list instead of a text box. To achieve this replace the following line of code.



##### ****After the changes, the complete code for the create view as shown below****

@model BusinessLayer.Employee

@{

ViewBag.Title = "Create";

}

**<h2>**Create Employee**</h2>**

@using (Html.BeginForm())

{

@Html.AntiForgeryToken()

**<div** class="form-horizontal"**>**

@Html.ValidationSummary(true, "", new { @class = "text-danger" })

**<div** class="form-group"**>**

@Html.LabelFor(model => model.Name, htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.EditorFor(model => model.Name, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.Name, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

@Html.LabelFor(model => model.Gender, htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.DropDownList("Gender", new List**<SelectListItem>**

{

new SelectListItem { Text = "Male", Value="Male" },

new SelectListItem { Text = "Female", Value="Female" }

}, "Select Gender", new { @class = "form-control" })

@Html.ValidationMessageFor(model => model.Gender, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

@Html.LabelFor(model => model.City, htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.EditorFor(model => model.City, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.City, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

@Html.LabelFor(model => model.Salary, htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.EditorFor(model => model.Salary, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.Salary, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

@Html.LabelFor(model => model.DateOfBirth, htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.EditorFor(model => model.DateOfBirth, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.DateOfBirth, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

**<div** class="col-md-offset-2 col-md-10"**>**

**<input** type="submit" value="Create" class="btn btn-default" **/>**

**</div>**

**</div>**

**</div>**

}

**<div>**

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

**</div>**

**<script** src="~/Scripts/jquery-1.10.2.min.js"**></script>**

**<script** src="~/Scripts/jquery.validate.min.js"**></script>**

**<script** src="~/Scripts/jquery.validate.unobtrusive.min.js"**></script>**

Now, run the application and notice that a dropdown list is rendered for “Gender”. Now, if you click on the “**Create**” button you will get an error message stating – **The resource cannot be found.**This is because we don’t have the “**Create**” action method that can handle **HTTPPost**requests.

##### ****What is the FormCollection Class in ASP.NET MVC?****

The FormCollection class in ASP.NET MVC will automatically receive the posted form values in the controller action method in the form of **key/value** pairs. The values can be accessed using either key names or indexes. We can use the FormCollection to loop through each key and its value that is posted to the server. Let’s add the following Create Post method in the employee Controller class.

**[**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/>"**)**;

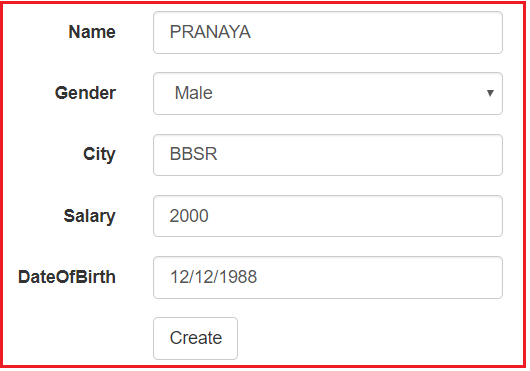
**}**

**}**

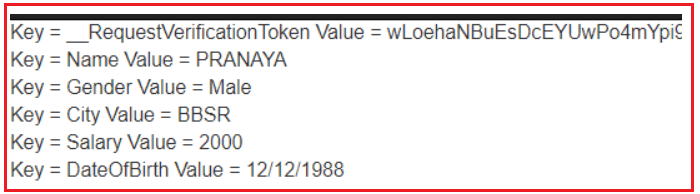
**return** View**()**;

**}**

**Now run the application and fill the view and click on the create button as shown below.**



**The output is as shown below.**



###### **Let’s create a stored procedure to insert the employee object in the Employee table**

**Create** procedure spAddEmployee

@Name nvarchar(50),

@Gender nvarchar (10),

@City nvarchar (50),

@Salary decimal(18,2),

@DateOfBirth DateTime

**As**

Begin

**Insert** **into** Employee(Name, Gender, City, Salary, DateOfBirth)

Values (@Name, @Gender, @City,@Salary, @DateOfBirth)

End

##### ****Add the following method to the EmployeeBusinessLayer.cs file.****

//Add employee into the database. This method takes an argument of Employee type which contains the

//employee that is going to stored in the database

**public** **void** AddEmmployee**(**Employee employee**)**

**{**

//Creating the connection string

string connectionString = ConfigurationManager.ConnectionStrings**[**"DBCS"**]**.ConnectionString;

//Establishing the connection to the database

**using** **(**SqlConnection con = new SqlConnection**(**connectionString**))**

**{**

//Creating the command object by passing the stored procedure and connection object as argument

//This stored procedure is used to store the employee in to the database

SqlCommand cmd = new SqlCommand**(**"spAddEmployee", con**)**

**{**

//Specifying the command as stored procedure

CommandType = CommandType.StoredProcedure

**}**;

//Creating SQL parameters because that stored procedure accept some input values

SqlParameter paramName = new SqlParameter

**{**

//Storing the parameter name of the stored procedure into the SQL parameter

//By using ParameterName property

ParameterName = "@Name",

//storing the parameter value into sql parameter by using Value ptoperty

Value = employee.Name

**}**;

//Adding that parameter into Command objects Parameter collection by using Add method

//which will take the SQL parameter name as argument

cmd.Parameters.Add**(**paramName**)**;

//Same for all other parameters (Gender, City, DateOfBirth )

SqlParameter paramGender = new SqlParameter

**{**

ParameterName = "@Gender",

Value = employee.Gender

**}**;

cmd.Parameters.Add**(**paramGender**)**;

SqlParameter paramCity = new SqlParameter

**{**

ParameterName = "@City",

Value = employee.City

**}**;

cmd.Parameters.Add**(**paramCity**)**;

SqlParameter paramSalary = new SqlParameter

**{**

ParameterName = "@Salary",

Value = employee.Salary

**}**;

cmd.Parameters.Add**(**paramSalary**)**;

SqlParameter paramDateOfBirth = new SqlParameter

**{**

ParameterName = "@DateOfBirth",

Value = employee.DateOfBirth

**}**;

cmd.Parameters.Add**(**paramDateOfBirth**)**;

//Open the connection and execute the command on ExecuteNonQuery method

con.Open**()**;

cmd.ExecuteNonQuery**()**;

**}**

**}**

To save form data to a database table modify the create (HttpPost) action method **EmployeeController.cs**file as shown below.

//FormCollection will store the submitted form data automatically when the form is submitted

**[**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.Salary =Convert.ToDecimal**(**formCollection**[**"Salary"**])**;

employee.DateOfBirth = Convert.ToDateTime**(**formCollection**[**"DateOfBirth"**])**;

EmployeeBusinessLayer employeeBusinessLayer = new EmployeeBusinessLayer**()**;

employeeBusinessLayer.AddEmmployee**(**employee**)**;

**return** RedirectToAction**(**"Index"**)**;

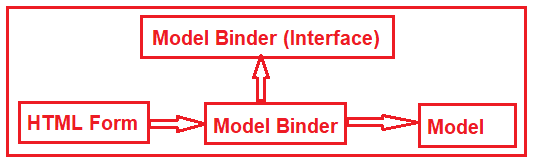
**}**

Now Run the application and see everything is working as expected.

**Model Binding in ASP.NET MVC**

**Model Binding in ASP.NET MVC Application with Examples**

Here in this article, we will discuss how to map ASP.NET MVC posted form data to controller action method using simple parameter types as well as using complex parameters using Model Binding.



**Let us first recap of what we did in our previous article.**

In our previous article, in order to save the form data to a database table, we use FormCollection class as shown in the below code. The 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.Salary = Convert.ToDecimal**(**formCollection**[**"Salary"**])**;

employee.DateOfBirth = Convert.ToDateTime**(**formCollection**[**"DateOfBirth"**])**;

EmployeeBusinessLayer employeeBusinessLayer = new EmployeeBusinessLayer**()**;

employeeBusinessLayer.AddEmmployee**(**employee**)**;

**return** RedirectToAction**(**"Index"**)**;

**}**

The above “**Create**” HttpPost action method can be re-written using simple types as shown below. Notice that, here the create action method has got parameter names that match with the names of the form controls. The model binder in ASP.NET MVC maps the values of these controls to the respective parameters.

**[**HttpPost**]**

**public** ActionResult Create**(**string name, string gender, string city, **decimal** Salary, DateTime dateOfBirth**)**

**{**

Employee employee = new Employee**()**;

employee.Name = name;

employee.Gender = gender;

employee.City = city;

employee.Salary = Salary;

employee.DateOfBirth = dateOfBirth;

EmployeeBusinessLayer employeeBusinessLayer = new EmployeeBusinessLayer**()**;

employeeBusinessLayer.AddEmmployee**(**employee**)**;

**return** RedirectToAction**(**"Index"**)**;

**}**

Run the application. Create one employee and see everything is working as expected.

**Note:** 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.

**Do we really have to do these mappings manually?**

The answer is no. Instead of creating simple parameters we can create one parameter of Employee type which will automatically hold the posted form values. Let’s modify the create HttpPost method to accept a complex parameter of Employee Type as shown below.

**[**HttpPost**]**

**public** ActionResult Create**(**Employee employee**)**

**{**

**if** **(**ModelState.IsValid**)**

**{**

EmployeeBusinessLayer employeeBusinessLayer = new EmployeeBusinessLayer**()**;

employeeBusinessLayer.AddEmmployee**(**employee**)**;

**return** RedirectToAction**(**"Index"**)**;

**}**

**return** View**()**;

**}**

**Points to Note:**

1. The model state is being checked using the **IsValid** boolean property of the **ModelState** object. We will discuss ModelState in a later article.
2. Instead of passing the individual properties of the “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 the AddEmployee() method of the “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.

**What is Model Binding in ASP.NET MVC?**

The ASP.NET MVC Model Binding is a mechanism that maps the HTTP request data with a model. It is the process of creating .NET objects using the data sent by the browser in an HTTP request. The ASP.NET Web Forms developers who are new to ASP.NET MVC are mostly confused about how the values from View get converted to the Model class when it reaches the Action method of the Controller class, so this conversion is done by the Model Binder.

Model binding is a well-designed bridge between the HTTP request and the C# action methods. It makes it easy for developers to work with data on forms (views) because POST and GET are automatically transferred into a data model we specify. ASP.NET MVC uses default binders to complete this behind the scene.

**UpdateModel and TryUpdateModel in ASP.NET MVC**

**UpdateModel and TryUpdateModel in ASP.NET MVC Application**

1. **Understanding the UpdateModel Function in ASP.NET MVC.**
2. **Understanding the TryUpdateModel() in ASP.NET MVC.**
3. **Difference between UpdateModel and TryUpdateModel function.**
4. **Is it mandatory to use “UpdateModel()” or “Try”UpdateModel()” function to update the Model?**
5. **Why do we need to explicitly invoke model binding?**

In the Model Binding article, we discuss how to save the model data using the complex object as a parameter as shown below

**[**HttpPost**]**

**public** ActionResult Create**(**Employee employee**)**

**{**

**if** **(**ModelState.IsValid**)**

**{**

EmployeeBusinessLayer employeeBusinessLayer = new EmployeeBusinessLayer**()**;

employeeBusinessLayer.AddEmmployee**(**employee**)**;

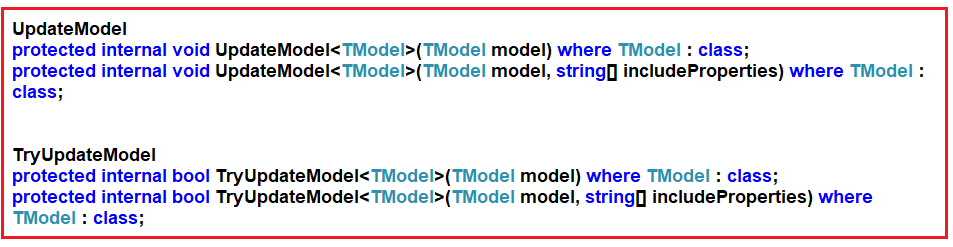
**return** RedirectToAction**(**"Index"**)**;

**}**

**return** View**()**;

**}**

**UpdateModel and TryUpdateModel in ASP.NET MVC Application:**



**UpdateModel Function in ASP.NET MVC**

Let us first understand how to use the UpdateModel function to capture the posted form data. In order to do this, please modify the Create (HttpPost) action method as shown below.

**[**HttpPost**]**

**public** ActionResult Create**()**

**{**

**if** **(**ModelState.IsValid**)**

**{**

EmployeeBusinessLayer employeeBusinessLayer = new EmployeeBusinessLayer**()**;

Employee employee = new Employee**()**;

UpdateModel**<**Employee**>(**employee**)**;

employeeBusinessLayer.AddEmmployee**(**employee**)**;

**return** RedirectToAction**(**"Index"**)**;

**}**

**return** View**()**;

**}**

When we make this change we get a compilation error stating – **The.EmployeeController’ already defines a member called ‘Create’ with the same parameter types.**

Our intention here is to overload the “**Create**” action method based on the “**HttpGet**” and “**HttpPost**“. To fix this error use the “**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.AddEmmployee**(**employee**)**;

**return** RedirectToAction**(**"Index"**)**;

**}**

**return** View**()**;

**}**

**Understanding the code:**

Here, first, we changed the names of the “**Create**” action methods to “**Create\_Get**” and “**Create\_Post**” depending on the actions they respond to.

The “ActionName” is specified as “Create” for both of these methods. So if a “GET” request is made to the “URL – http://localhost:54094/Employee/Create” then the “Create\_Get()” action method is invoked. On the other hand, if a “POST” request is made to the same URL then the “Create\_Post()” action method is invoked.

Instead of passing the “Employee” object as a parameter to the “Create\_Post()” action method we are creating an instance of an “Employee” object within the function and updating it using the “UpdateModel()” function.

The “UpdateModel()” function inspects all the HttpRequest inputs such as Posted Form data, QueryString, Cookies, and Server variables and populates the employee object.

**Understanding the TryUpdateModel() in ASP.NET MVC:**

Now let’s understand how to use the TryUpdateModel function in ASP.NET MVC Application. Modify the create (HttpPost) action method as shown below. Here we use TryUpdateModel() instead of UpdateModel().

**[**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**()**;

**}**

**}**

Now run the application and see everything is working as expected. Now let us understand the difference between them.

The difference is UpdateModel() throws an exception if validation fails whereas 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 “Try”UpdateModel()” function to update the Model?**

The answer is **NO**. The above method can be re-written as shown below and we get the same behavior.

**[**HttpPost**]**

**[**ActionName**(**"Create"**)]**

**public** ActionResult Create\_Post**(**Employee employee**)**

**{**

**if** **(**ModelState.IsValid**)**

**{**

EmployeeBusinessLayer employeeBusinessLayer = new EmployeeBusinessLayer**()**;

employeeBusinessLayer.AddEmmployee**(**employee**)**;

**return** RedirectToAction**(**"Index"**)**;

**}**

**else**

**{**

**return** View**()**;

**}**

**}**

**Why do we need to explicitly invoke model binding?**

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

# Editing a Model in ASP.NET MVC

## ****How to Edit a Model in ASP.NET MVC Application****

##### ****Step1: Creating the Edit action method.****

Please copy and paste the following “**Edit**” action method within the “**EmployeeController**” and include the **System.Linq** namespace. Here, first, we are retrieving all the employees and then finding the employees by using the Linq FirstOrDefault method. But in real-time you need to create one method in your business layer and that method should return the employee based on the id.

**[**HttpGet**]**

**public** ActionResult Edit**(int** id**)**

**{**

EmployeeBusinessLayer employeeBusinessLayer = new EmployeeBusinessLayer**()**;

Employee employee = employeeBusinessLayer.GetAllEmployess**()**.FirstOrDefault**(**emp =**>** emp.ID == id**)**;

**return** View**(**employee**)**;

**}**

**Points to note:**

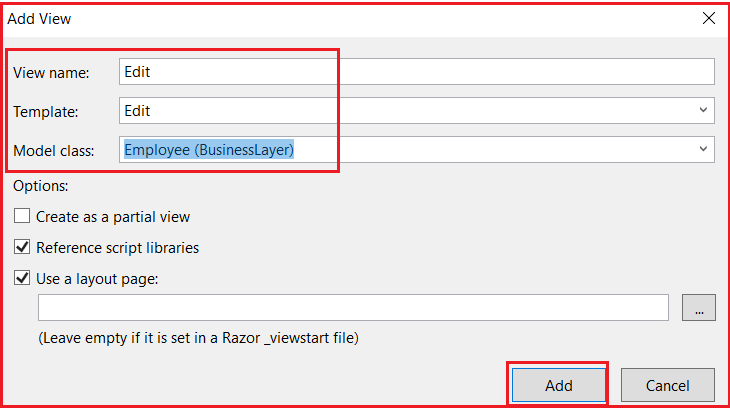
1. This method is decorated with the [HttpGet] attribute. So this method only responds to HTTP get requests when editing the data.
2. The Edit action method also receives the id of the employee that is being edited. This “id” is used to retrieve the employee details.
3. The employee object is passed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

###### **Step2: Creating the “Edit” view**

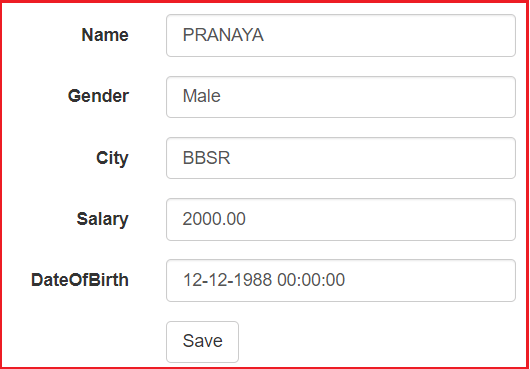
Now let’s add the **Edit** view. To do this right-click on the **Edit** action method and then select **Add View** option from the context menu and then Set

**View name = “Edit”**  
**Template = “Edit”**  
**Model class = Employee (BusinessLayer)**

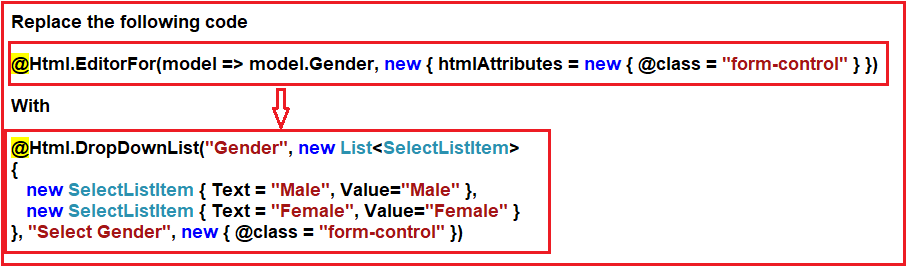
Click on the “**Add**” button as shown below. This should add the “**Edit.cshtml**” view within the “**Employee**” folder which is within the “**Views**” folder.



Run the application and navigate to **http://localhost:54094/Employee/Index**. This page should display the list of all the employees. Click on the “**Edit**” link which is available on the right side of the employee. The “**Edit**” page should display the details of the “**Employee**” that is being edited as shown below.



Notice that by default “**textboxes**” are used for editing. It is ideal to have a **drop-down list for gender** rather than a textbox. To achieve this make the following changes to “Edit.cshtml”.



With the above changes in place, the complete code in Edit.cshtml view as below.

@model BusinessLayer.Employee

@{

ViewBag.Title = "Edit";

}

**<h2>**Edit**</h2>**

@using (Html.BeginForm())

{

@Html.AntiForgeryToken()

**<div** class="form-horizontal"**>**

**<h4>**Employee**</h4>**

**<hr** **/>**

@Html.ValidationSummary(true, "", new { @class = "text-danger" })

@Html.HiddenFor(model => model.ID)

**<div** class="form-group"**>**

@Html.LabelFor(model => model.Name, htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.EditorFor(model => model.Name, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.Name, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

@Html.LabelFor(model => model.Gender, htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.DropDownList("Gender", new List**<SelectListItem>**

{

new SelectListItem { Text = "Male", Value="Male" },

new SelectListItem { Text = "Female", Value="Female" }

}, "Select Gender", new { @class = "form-control" })

**</div>**

**</div>**

**<div** class="form-group"**>**

@Html.LabelFor(model => model.City, htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.EditorFor(model => model.City, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.City, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

@Html.LabelFor(model => model.Salary, htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.EditorFor(model => model.Salary, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.Salary, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

@Html.LabelFor(model => model.DateOfBirth, htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.EditorFor(model => model.DateOfBirth, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.DateOfBirth, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

**<div** class="col-md-offset-2 col-md-10"**>**

**<input** type="submit" value="Save" class="btn btn-default" **/>**

**</div>**

**</div>**

**</div>**

}

**<div>**

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

**</div>**

**<script** src="~/Scripts/jquery-1.10.2.min.js"**></script>**

**<script** src="~/Scripts/jquery.validate.min.js"**></script>**

**<script** src="~/Scripts/jquery.validate.unobtrusive.min.js"**></script>**

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

###### **Create a stored procedure to update employee data.**

**Create** procedure spUpdateEmployee

@Id int,

@Name nvarchar(50),

@Gender nvarchar (10),

@City nvarchar (50),

@Salary decimal(18,2),

@DateOfBirth DateTime

**as**

Begin

**Update** Employee Set

Name = @Name,

Gender = @Gender,

City = @City,

Salary = @Salary,

DateOfBirth = @DateOfBirth

**Where** Id = @Id

End

###### **Add the following “UpdateEmployee()” method to the “EmployeeBusinessLayer” class in the “BusinessLayer” project.**

This method is used to update employee data to the database table.

**public** **void** UpdateEmmployee**(**Employee employee**)**

**{**

string connectionString = ConfigurationManager.ConnectionStrings**[**"DBCS"**]**.ConnectionString;

**using** **(**SqlConnection con = new SqlConnection**(**connectionString**))**

**{**

SqlCommand cmd = new SqlCommand**(**"spUpdateEmployee", 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 paramSalary = new SqlParameter**()**;

paramSalary.ParameterName = "@Salary";

paramSalary.Value = employee.Salary;

cmd.Parameters.Add**(**paramSalary**)**;

SqlParameter paramDateOfBirth = new SqlParameter**()**;

paramDateOfBirth.ParameterName = "@DateOfBirth";

paramDateOfBirth.Value = employee.DateOfBirth;

cmd.Parameters.Add**(**paramDateOfBirth**)**;

con.Open**()**;

cmd.ExecuteNonQuery**()**;

**}**

**}**

##### ****Creating the Edit Post Method:****

Please copy and paste the following “**Edit**” (HttpPost) action method in “**EmployeeController**“.

**[**HttpPost**]**

**public** ActionResult Edit**(**Employee employee**)**

**{**

**if** **(**ModelState.IsValid**)**

**{**

EmployeeBusinessLayer employeeBusinessLayer = new EmployeeBusinessLayer**()**;

employeeBusinessLayer.UpdateEmmployee**(**employee**)**;

**return** RedirectToAction**(**"Index"**)**;

**}**

**return** View**(**employee**)**;

**}**

##### ****Code Explanation:****

1. The above Edit action method is decorated with the [**HttpPost**] attribute. So this method only responds to HTTP post requests when updating data.
2. This method receives the modified “**Employee**” object as a parameter. This object is then passed to **UpdateEmmployee()** method which updates 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 the “**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.

# Unintended Updates in ASP.NET MVC

## Unintended****Updates in ASP.NET MVC Application****

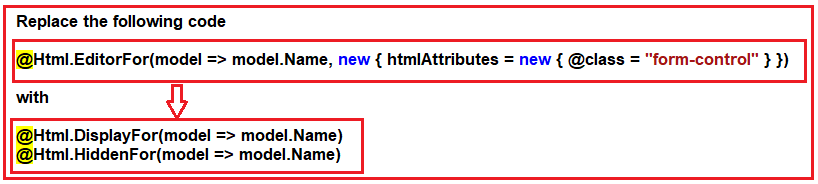
1. **What are Unintended Updates in ASP.NET MVC?**
2. **Example to understand Unintended Updates in ASP.NET MVC.**
3. **How to prevent Unintended Updates?**

##### Let’s understand ****Unintended Updates****with an example.

At the moment, within the “**Employee Edit”**view, we are allowing to change all of the following fields.

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

Let’s make the “**Name**” field is non-editable. To achieve this change the following code in the **Edit.cshtml** file.



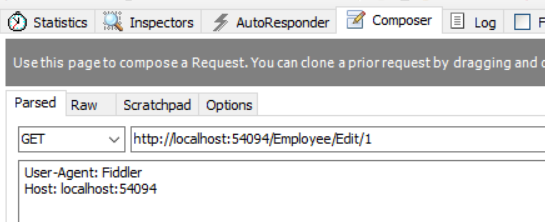
Run the application and edit an employee. Notice that the Nameof the employee is no longer rendered using a textbox. At this point, you may think that it is impossible for the user to change the name of the employee using the “**Edit**“view. That is not true. Because of the way we have written our code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

##### ****Using Fiddler to Post data:****

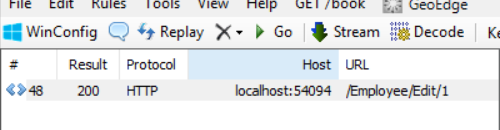
Fiddler can be downloaded from the following URL

<https://www.telerik.com/download/fiddler>

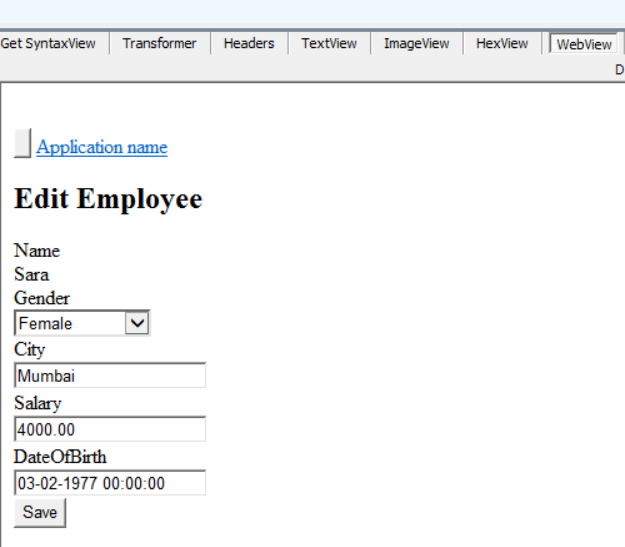
Once you downloaded and installed the fiddler, then run fiddler. Select the Composer Tab and then select the method as GET. Provide the URL as **http://localhost:54094/Employee/Edit/1**and click on the **execute** button as shown below



In the fiddler in**the**web sessionswindow, select the URL. Under the“**Inspectors**“tab we can see Request headers and responses. We will discuss more on fiddler in a later session. To see this click on the below URL



Then select the web view as shown below



Now click on the “Save”button on the “Edit”view. Notice that under“Web Sessions”in fiddler another request is captured for the same URL – **http://localhost:54094/Employee/Edit/1**

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

1. Click on**the**“**Composer**“tab in the fiddler
2. Drag and drop the following URL from the “**Web Sessions**“window onto the Composer window.
3. In“**Request Body**“under**the**“**Composer**“tab change“**Name**“of the employee to“**XYZ**“
4. Finally, clickthe “**Execute**“button

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

##### ****How to prevent unintended updates in ASP.NET MVC?****

Modify the “**Edit**” action method of **EmployeeController** that is decorated with [**HttpPost**] attribute as shown below.

**[**HttpPost**]**

**[**ActionName**(**"Edit"**)]**

**public** ActionResult Edit\_Post**(int** id**)**

**{**

EmployeeBusinessLayer employeeBusinessLayer = new EmployeeBusinessLayer**()**;

Employee employee = employeeBusinessLayer.GetAllEmployess**()**.FirstOrDefault**(**x =**>** x.ID == id**)**;

UpdateModel**(**employee, new string**[]** **{** "ID", "Gender", "City", "Salary", "DateOfBirth" **})**;

**if** **(**ModelState.IsValid**)**

**{**

employeeBusinessLayer.UpdateEmmployee**(**employee**)**;

**return** RedirectToAction**(**"Index"**)**;

**}**

**return** View**(**employee**)**;

**}**

###### **Code Explanation:**

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 the “**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, Salary & DateOfBirth**) from the database.
5. We then call**UpdateModel()** function. This should automatically update the“**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 including a listorwhite list. Notice that we did not include the “**Name**“property on the list. This means even if the posted form data contains the value for the “**Name**“property it will not be used to update the“**Name**“property of the“**Employee**“object.

So, if you generated a post request using the fiddler “**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.GetAllEmployess**()**.Single**(**x =**>** x.ID == id**)**;

UpdateModel**(**employee, **null**, **null**, new string**[]** **{** "Name" **})**;

**if** **(**ModelState.IsValid**)**

**{**

employeeBusinessLayer.UpdateEmmployee**(**employee**)**;

**return** RedirectToAction**(**"Index"**)**;

**}**

**return** View**(**employee**)**;

**}**

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

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

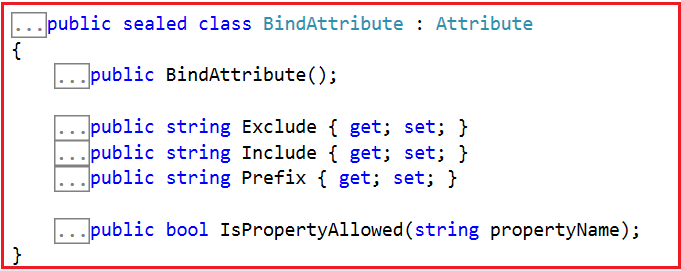
**Bind Attribute in ASP.NET MVC**

**Bind Attribute in ASP.NET MVC**

1. **Understanding the Bind Attribute class.**
2. **Including Model Properties using Bind Attributes.**
3. **Excluding Model Properties using Bind Attribute.**

**Understanding the Bind Attribute in ASP.NET MVC:**

Let us first have a look at the definition of the BindAttribute class in the ASP.NET MVC Framework. As you can see the BindAttribute class is inherited from the Attribute class and also contain some properties i.e. Exclude, Include, Prefix, and IsPropertyAllowed method.



1. **Exclude**: The Exclude property is used to get or set a comma-delimited list of property names for which binding is not allowed.
2. **Include**: The Include property is used to get or set a comma-delimited list of property names for which binding is allowed.
3. **Prefix**: The Prefix property is used to gets or sets the prefix to use when markup is rendered for binding to an action argument or to a model property.
4. **IsPropertyAllowed**: The IsPropertyAllowed method is used to determines whether the specified property is allowed. It returns true if the specified property is allowed; otherwise, false.

**Bind Attribute Example in ASP.NET MVC Application:**

Let us understand how to use the Bind Attribute in ASP.NET MVC Application with an example. First, modify the **“Edit\_Post()”** action method of EmployeeController.cs as shown below.

**[**HttpPost**]**

**[**ActionName**(**"Edit"**)]**

**public** ActionResult Edit\_Post**([**Bind**(**Include = "Id, Gender, City, Salary, DateOfBirth"**)]** Employee employee**)**

**{**

EmployeeBusinessLayer employeeBusinessLayer = new EmployeeBusinessLayer**()**;

employee.Name = employeeBusinessLayer.GetAllEmployess**()**.FirstOrDefault**(**x =**>** x.ID == employee.ID**)**.Name;

**if** **(**ModelState.IsValid**)**

**{**

employeeBusinessLayer.UpdateEmmployee**(**employee**)**;

**return** RedirectToAction**(**"Index"**)**;

**}**

**return** View**(**employee**)**;

**}**

As you can see in the above action code, using the **“BIND”** attribute we are specifying the properties that we want to include in our model binding using the Include property of BindAttribute class. As the **“Name”** property of the Employee model is not specified in the **Include**list, so it will be excluded from model binding. Now, if you generate a post request using fiddler as we did in the previous session, the **“Name”** property of the **“Employee”** object will not be updated.

**Using the Exclude Property of BindAttribute class in ASP.NET MVC:**

You can also achieve the same thing using the **Exclude** property of the Bind Attribute class in the ASP.NET MVC Application. Here, you need to specify the properties which you want to exclude from model binding as shown below in the below code.

**[**HttpPost**]**

**[**ActionName**(**"Edit"**)]**

**public** ActionResult Edit\_Post**([**Bind**(**Exclude = "Name"**)]** Employee employee**)**

**{**

// Rest of the method implementation remains the same

**}**

As you can see in the above code, we specify the Name property in the Exclude list of the Bind Attribute class. So, here the Name property will be excluded from model binding.

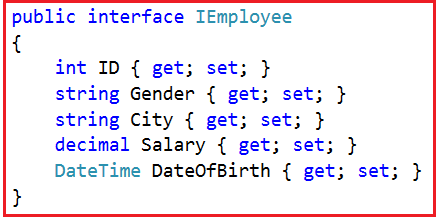
# Model Binding Using Interface

## ****Including and Excluding Properties from Model Binding Using Interface****

In this article, I am going to discuss **Including and Excluding Properties from Model Binding using Interface** **in ASP.NET MVC** Application. Please read our previous article before proceeding to this article where we discussed [**How to include and Exclude Properties using Bind Attribute**](https://dotnettutorials.net/lesson/bind-attribute-asp-net-mvc/)in ASP.NET MVC Application. This is also a continuation part of our previous article. Here, I will show you how to include and exclude properties in Model Binding using Interface.

##### ****Include and Exclude Properties from Model Binding using Interface****

First Create an interface “**IEmployee**” as shown below.



Notice that the above **IEmployee** interface has got only the properties that we want to include in model binding. The “**Name**” property is not present. This means “**Name**” property will be excluded from model binding. So, modify the “**Employee.cs**” class file in the “**BusinessLayer**” project as shown below.

**namespace** *BusinessLayer*

**{**

**public** **interface** IEmployee

**{**

**int** ID **{** **get**; **set**; **}**

string Gender **{** **get**; **set**; **}**

string City **{** **get**; **set**; **}**

**decimal** Salary **{** **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**; **}**

**public** string Gender **{** **get**; **set**; **}**

**public** string City **{** **get**; **set**; **}**

**public** **decimal** Salary **{** **get**; **set**; **}**

**public** DateTime DateOfBirth **{** **get**; **set**; **}**

**}**

**}**

###### **Next, Modify the “Edit\_Post()” action method of EmployeeController as shown below.**

**[**HttpPost**]**

**[**ActionName**(**"Edit"**)]**

**public** ActionResult Edit\_Post**(int** id**)**

**{**

EmployeeBusinessLayer employeeBusinessLayer = new EmployeeBusinessLayer**()**;

Employee employee = employeeBusinessLayer.GetAllEmployess**()**.Single**(**x =**>** x.ID == id**)**;

UpdateModel**<**IEmployee**>(**employee**)**;

**if** **(**ModelState.IsValid**)**

**{**

employeeBusinessLayer.UpdateEmmployee**(**employee**)**;

**return** RedirectToAction**(**"Index"**)**;

**}**

**return** View**(**employee**)**;

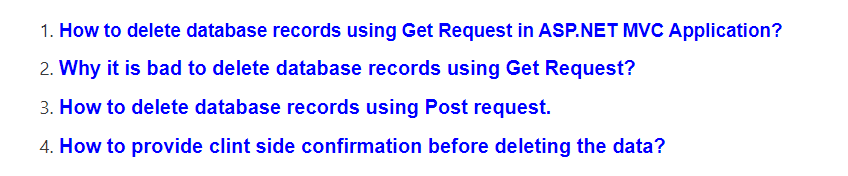
**}**

Notice that we are explicitly calling the model binder by calling **UpdateModel()** function passing our interface **IEmployee**. The model binder will update only the properties that are present in the interface. So if you were generating 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 in ASP.NET MVC Application. Depending on the architecture and requirements of your project you need to choose the approach that best fits your needs.

**Deleting Database Records in ASP.NET MVC**

**Deleting Database Records in ASP.NET MVC Application**



**Deleting Database Records using Get Request:**

Let’s first understand how to delete database records in ASP.NET MVC Application using a GET request and then we will discuss why it is bad to do so.

**Step1: Create a stored procedure to delete employee data by “ID”**

**Create** Procedure spDeleteEmployee

@Id int

**as**

Begin

**Delete** **from** Employee **where** Id = @Id

End

**Step2:**Add the following **DeleteEmployee**() method to the **“EmployeeBusinessLayer.cs”**file in the “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**()**;

**}**

**}**

**Step3: Add the following “DELETE” 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 the **“Index”** action. Click the **“Delete”** link. This issues the **“GET”** request to the following URL and deletes the record.

**http://localhost/MVC\_DEMO/Employee/Delete/1**

Deleting database records using a 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.

**<img src=”http://localhost/MVC\_DEMO/Employee/Delete/2″ />**

Also when search engines index our page they issue a **GET**request which would delete the data. In general, **GET** requests should be free of any side effects meaning they should not change the state. Deletes should always be performed using a **POST** request.

**Deleting Database Records using the POST Request in ASP.NET MVC Application:**

Showing the client-side javascript confirmation dialog box before deleting.

**Step1:** Mark the “**Delete**” action method in the “**Employee**” controller with the [**HttpPost**] attribute. With this change, the “**Delete**” method will no longer respond to the “**GET**” request. At this point, if we run the application and click on the “**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"**)**;

**}**

**Step2:** Modifying the **“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.Salary)

**</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.Salary)

**</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 }) |

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

**</td>**

**</tr>**

}

}

Notice that we are using **“Html.BeginForm()”** HTML helper to generate a form tag.

**Step3:** To include client-side confirmation before the data can be deleted add the **“onclick”** attribute to the **“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’);” />**

That’s it run the application and see everything is working as expected.

**CRUD Operations using Entity Framework**

**CRUD Operations using Entity Framework in ASP.NET MVC**

**Database Tables used in this Demo:**

In this demo, we are going to use the following **Department**and **Employee** table. So, please use the below SQL script to create and populate these 2 tables

**Create** **table** Department

(

Id int primary key identity,

Name nvarchar(50)

)

**Insert** **into** Department values('IT')

**Insert** **into** Department values('HR')

**Insert** **into** Department values('Payroll')

**Create** **table** Employee

(

EmployeeId int Primary Key Identity(1,1),

Name nvarchar(50),

Gender nvarchar(10),

City nvarchar(50),

Salary decimal(18,2),

DepartmentId int

)

**Alter** **table** Employee add foreign key (DepartmentId) references Department(Id)

**Insert** **into** Employee values('Mark','Male','London',1000,1)

**Insert** **into** Employee values('John','Male','Chennai',2000,3)

**Insert** **into** Employee values('Mary','Female','New York',3000,3)

**Insert** **into** Employee values('Mike','Male','Sydeny',4000,2)

**Insert** **into** Employee values('Scott','Male','London',3000,1)

**Insert** **into** Employee values('Pam','Female','Falls Church',2000,2)

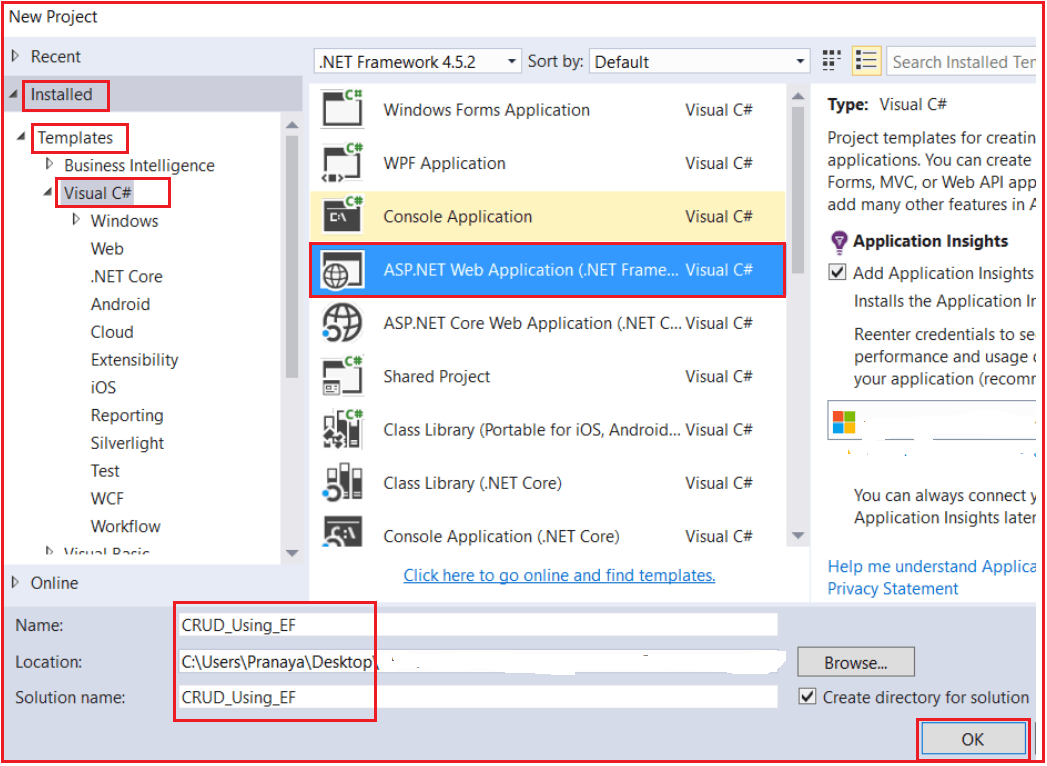
**Insert** **into** Employee values('Todd','Male','Sydney',1000,1)

**Insert** **into** Employee values('Ben','Male','New Delhi',4000,2)

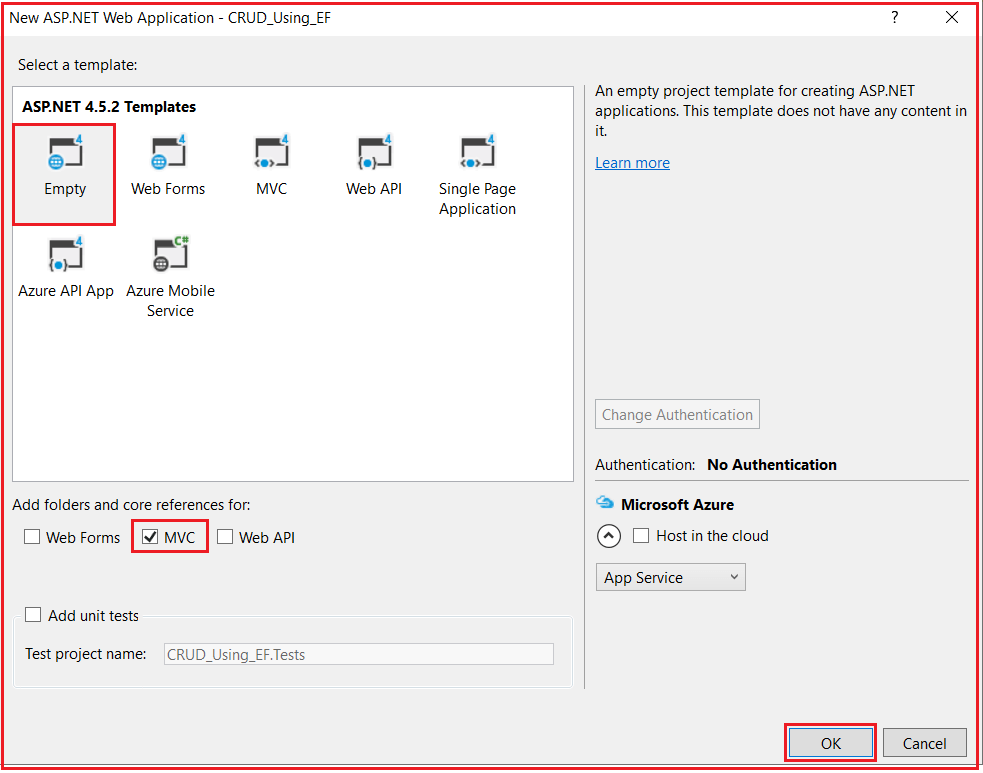
**Insert** **into** Employee values('Sara','Female','London',5000,1)

**Create a new ASP.NET MVC 5 Web application:**

Open **File => New =>Project** and then select **Installed => Templates => Visual C#.**Then select **ASP.NET Web Application**. Provide the project name and location where you want to save your application and finally click on the **OK** button as shown in the below image.



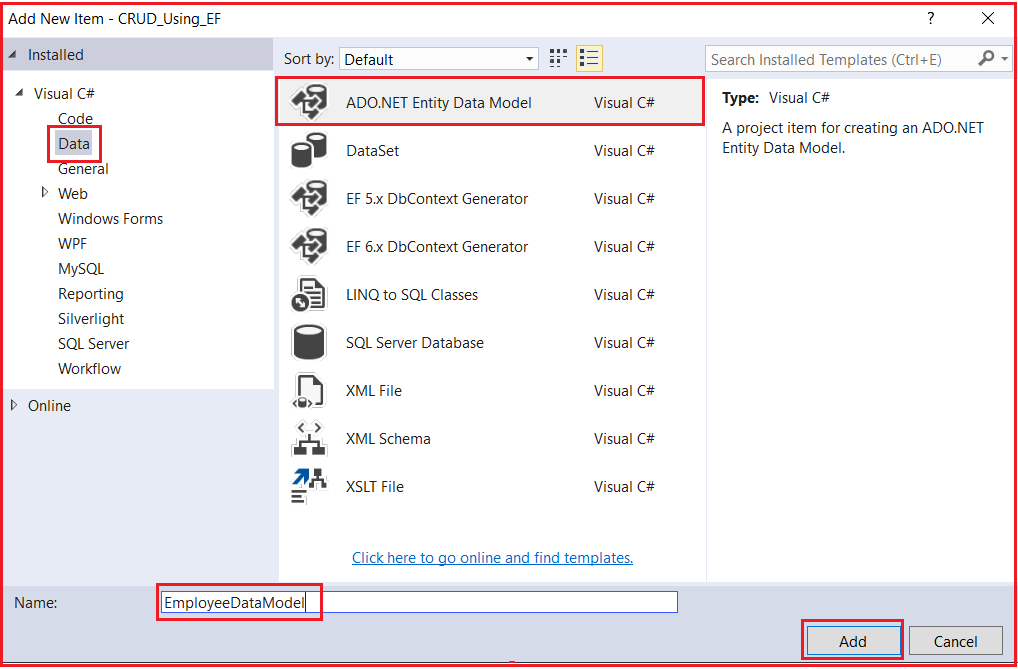
From the next screen that is from select a template screen, select **Empty** as the project template. From add folder and core reference section check the **MVC** checkbox and click on **OK** as shown below.



Once you click on the OK button, it will take some time to create the project for us.

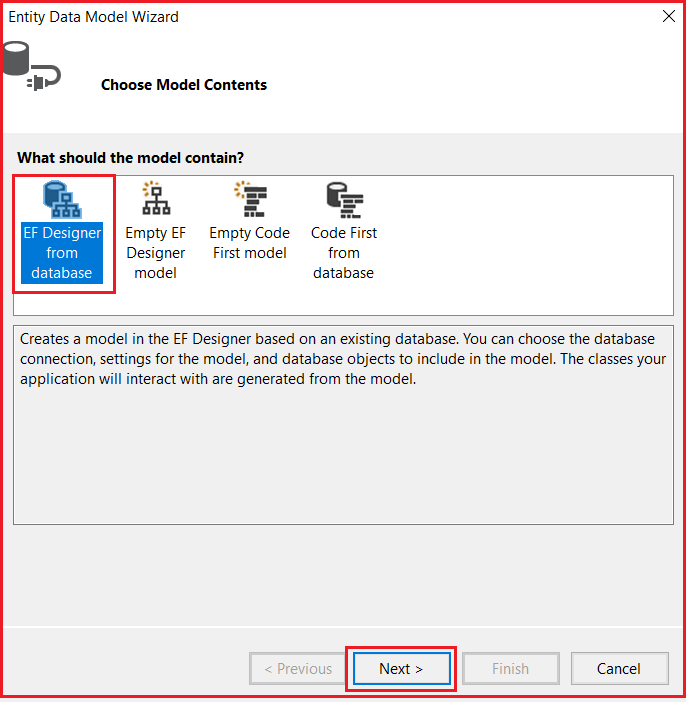
**Adding ADO.NET Entity Data Model**

Right-click on Models folder then select **Add => New Item** from the context menu that will open the **Add New Item** window. Select the **Data** tab from the left panel and then choose **ADO.NET Entity Data Model** from the middle panel. Provide a meaningful name for your data model and click on the **Add** button as shown in the below image.



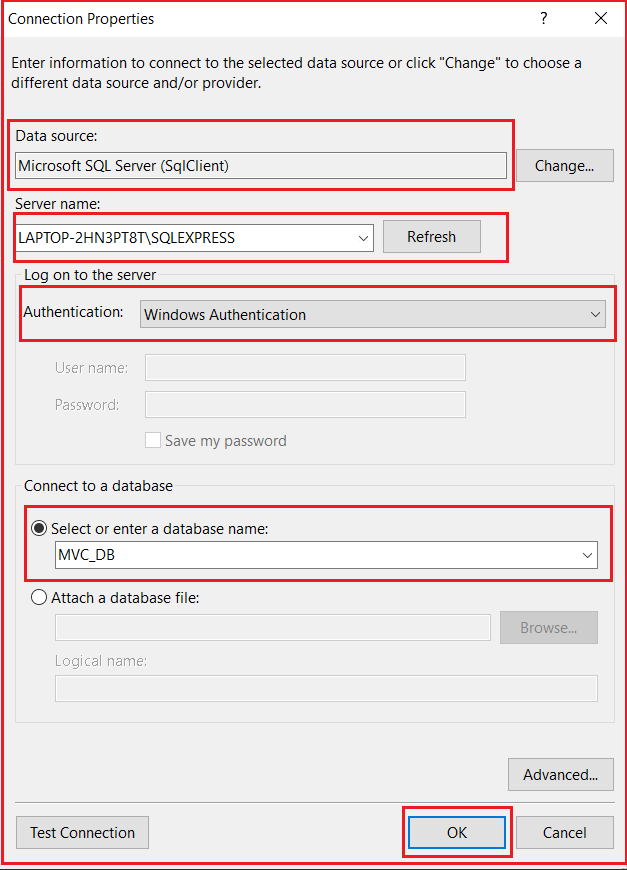
**Selecting The Entity Framework Approach to interact with the database.**

Once you click on the Add button it will ask you to choose the entity framework approach. Here, we are going to use the database first approach as we already created the required database tables. So, from the Entity Data Model Wizard, select “**Generate from database**” option and click “**Next**” as shown below.

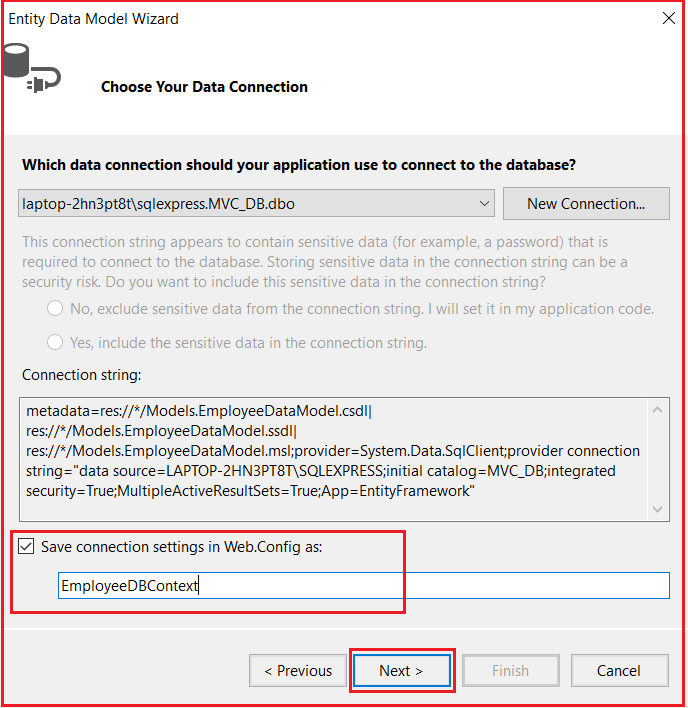


**Creating Database Connection:**

On “**Choose your data connection screen**” and click on the “**New Connection**” button which will open the connection properties window. Here, we are going to interact with the SQL Server database. So, from the Data Source select **Microsoft SQL Server (SqlClient)**. Provide your SQL Server name. Choose the Authentication type. Here, I am choosing **Windows Authentication**. Then select the database to which you are going to interact from the **select or enter a database name** drop-down list. Finally, click on the “**OK**” as shown below.

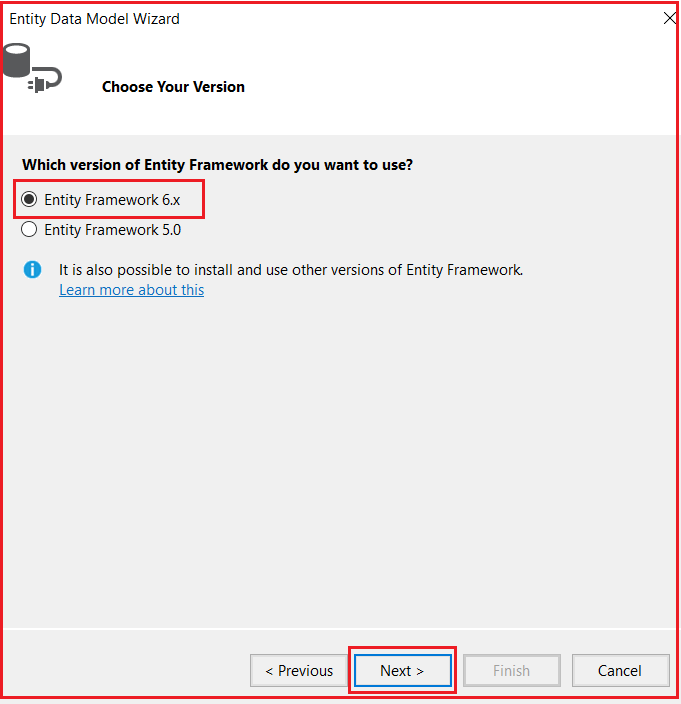


Then provide a meaningful connection string name such as “**EmployeeDBContext**” and click on the “**Next**” button as shown below.



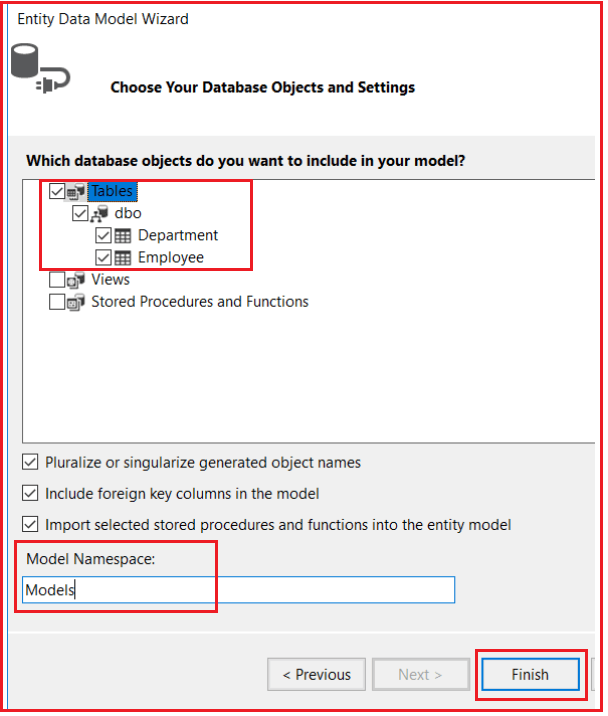
**Choose Entity Framework Version:**

Here, we are going to use Entity Framework 6. So, from the Choose Your Version screen, choose the **Entity framework 6.x** and click on the **Next** button as shown below.

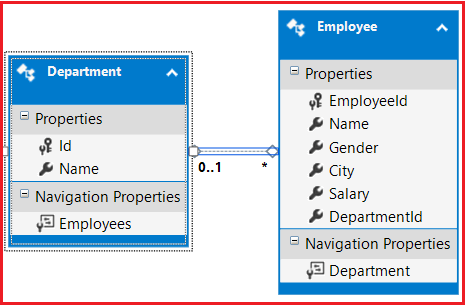


**Selecting Database Objects and Settings:**

On “**Choose your database objects and Settings**” screen, expand the “**Tables**” and then select “**Department**” and “**Employee**” tables. Set Model Namespace as **Models** and click on the “**Finish**” button as shown in the below image.

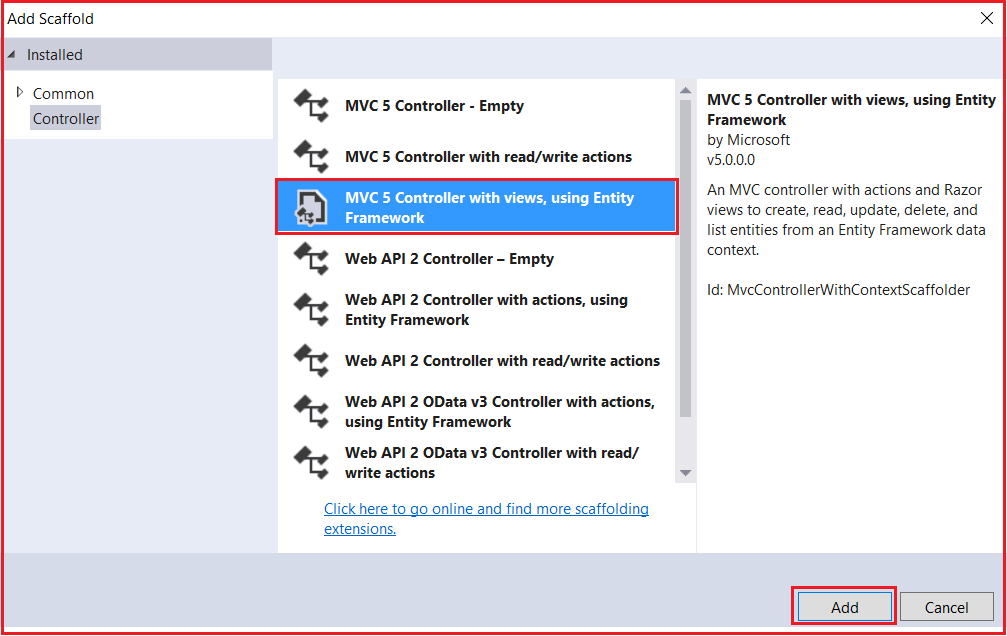


At this point, we should have Departmentand Employee entities generated as shown below.



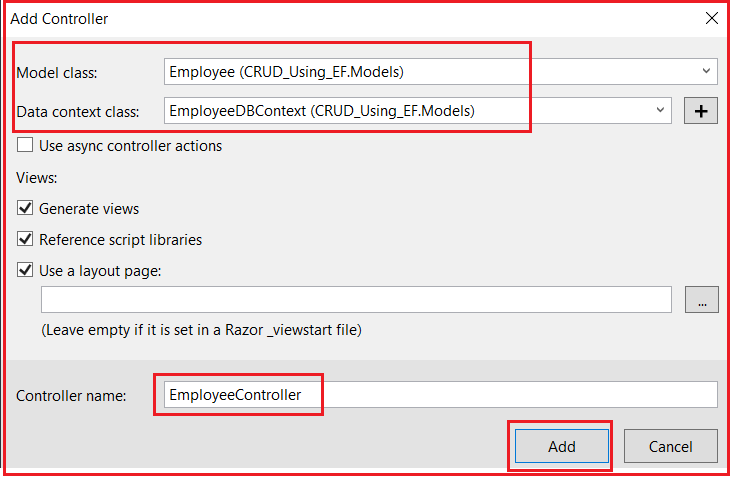
**Creating MVC 5 Controller:**

Right-click on the “**Controllers**” folder and select **Add – Controller** from the context menu. Then select **MVC 5 Controller with views, using Entity Framework** and click on the **Add** button as shown in the below image.



**On the next screen set the below details**

1. **Model class = Employee (CRUD\_Using\_EF.Models)**
2. **Data Context Class = EmployeeDBContext(CRUD\_Using\_EF.Models)**
3. **Controller Name = EmployeeController**
4. **Rest values are as it is and click on the Add button as shown in the below image**



At this point, we should have the following files automatically added.

**EmployeeController.cs** file in “**Controllers**” folder. **Index, Create, Edit, Detail and Delete** views in the “**Employee**” folder which is inside the Views folder.

**Below is the EmployeeController code**

**namespace** *CRUD\_Using\_EF.Controllers*

**{**

**public** **class** EmployeeController : Controller

**{**

**private** EmployeeDBContext db = new EmployeeDBContext**()**;

// GET: Employee

**public** ActionResult Index**()**

**{**

var employees = db.Employees.Include**(**e =**>** e.Department**)**;

**return** View**(**employees.ToList**())**;

**}**

// GET: Employee/Details/5

**public** ActionResult Details**(int**? id**)**

**{**

**if** **(**id == **null)**

**{**

**return** new HttpStatusCodeResult**(**HttpStatusCode.BadRequest**)**;

**}**

Employee employee = db.Employees.Find**(**id**)**;

**if** **(**employee == **null)**

**{**

**return** HttpNotFound**()**;

**}**

**return** View**(**employee**)**;

**}**

// GET: Employee/Create

**public** ActionResult Create**()**

**{**

ViewBag.DepartmentId = new SelectList**(**db.Departments, "Id", "Name"**)**;

**return** View**()**;

**}**

// POST: Employee/Create

// To protect from overposting attacks, please enable the specific properties you want to bind to, for

// more details see http://go.microsoft.com/fwlink/?LinkId=317598.

**[**HttpPost**]**

**[**ValidateAntiForgeryToken**]**

**public** ActionResult Create**([**Bind**(**Include = "EmployeeId,Name,Gender,City,Salary,DepartmentId"**)]** Employee employee**)**

**{**

**if** **(**ModelState.IsValid**)**

**{**

db.Employees.Add**(**employee**)**;

db.SaveChanges**()**;

**return** RedirectToAction**(**"Index"**)**;

**}**

ViewBag.DepartmentId = new SelectList**(**db.Departments, "Id", "Name", employee.DepartmentId**)**;

**return** View**(**employee**)**;

**}**

// GET: Employee/Edit/5

**public** ActionResult Edit**(int**? id**)**

**{**

**if** **(**id == **null)**

**{**

**return** new HttpStatusCodeResult**(**HttpStatusCode.BadRequest**)**;

**}**

Employee employee = db.Employees.Find**(**id**)**;

**if** **(**employee == **null)**

**{**

**return** HttpNotFound**()**;

**}**

ViewBag.DepartmentId = new SelectList**(**db.Departments, "Id", "Name", employee.DepartmentId**)**;

**return** View**(**employee**)**;

**}**

// POST: Employee/Edit/5

// To protect from overposting attacks, please enable the specific properties you want to bind to, for

// more details see http://go.microsoft.com/fwlink/?LinkId=317598.

**[**HttpPost**]**

**[**ValidateAntiForgeryToken**]**

**public** ActionResult Edit**([**Bind**(**Include = "EmployeeId,Name,Gender,City,Salary,DepartmentId"**)]** Employee employee**)**

**{**

**if** **(**ModelState.IsValid**)**

**{**

db.Entry**(**employee**)**.State = EntityState.Modified;

db.SaveChanges**()**;

**return** RedirectToAction**(**"Index"**)**;

**}**

ViewBag.DepartmentId = new SelectList**(**db.Departments, "Id", "Name", employee.DepartmentId**)**;

**return** View**(**employee**)**;

**}**

// GET: Employee/Delete/5

**public** ActionResult Delete**(int**? id**)**

**{**

**if** **(**id == **null)**

**{**

**return** new HttpStatusCodeResult**(**HttpStatusCode.BadRequest**)**;

**}**

Employee employee = db.Employees.Find**(**id**)**;

**if** **(**employee == **null)**

**{**

**return** HttpNotFound**()**;

**}**

**return** View**(**employee**)**;

**}**

// POST: Employee/Delete/5

**[**HttpPost, ActionName**(**"Delete"**)]**

**[**ValidateAntiForgeryToken**]**

**public** ActionResult DeleteConfirmed**(int** id**)**

**{**

Employee employee = db.Employees.Find**(**id**)**;

db.Employees.Remove**(**employee**)**;

db.SaveChanges**()**;

**return** RedirectToAction**(**"Index"**)**;

**}**

**protected** **override** **void** Dispose**(bool** disposing**)**

**{**

**if** **(**disposing**)**

**{**

db.Dispose**()**;

**}**

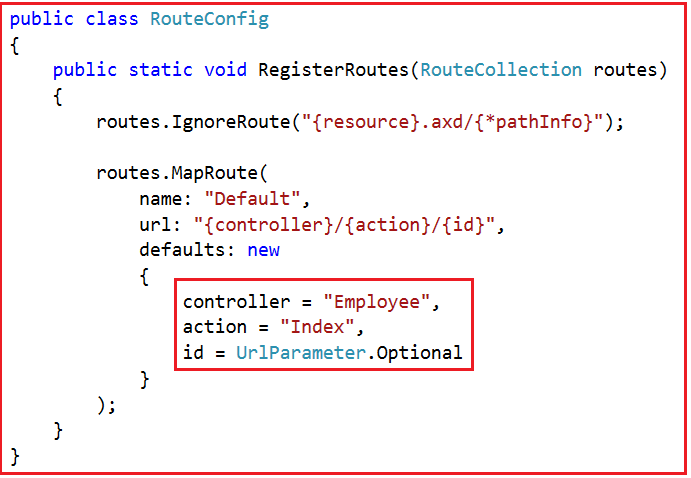
**base**.Dispose**(**disposing**)**;

**}**

**}**

**}**

At this point if you run the application you will get an error stating – **The resource cannot be found**. This is because by default the application goes to the “**Home**” controller and “**Index**” action. To fix this, open “**RouteConfig.cs**” file from the “App\_Start” folder and set the controller as “**Employee**” as shown below.



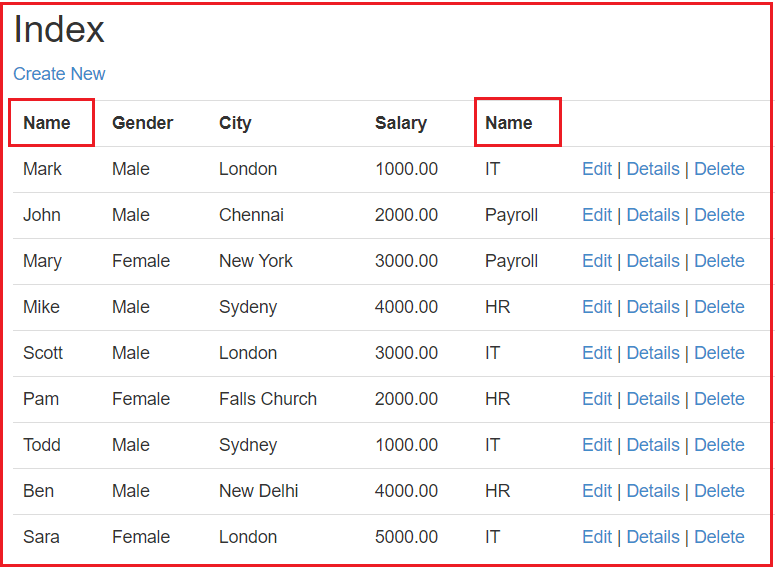
Run the application again. Notice that all the employees are listed on the index view. We can also create a new employee, edit an employee, view their full details and delete an employee as well. However, there are a few issues with each of the views which we will address in our upcoming articles.

# Customizing Auto Generated Index and Create Views

## ****Customizing Auto-Generated Index and Create View****

##### ****Customizing Index View:****

Run the application and navigate to the index view which should display the data as shown below.



As shown in the above “**Index**” view, it is using “**Name**” as the column header for both **employee** and **department** name. This is because the “**Name**” column is used in both the database tables (**Employee and Department**) and the entity framework used these column names to generate the “**Name**” property in both **Employee and Department** classes that are auto-generated as shown below.

###### **Employee.cs File**

**namespace** *CRUD\_Using\_EF.Models*

**{**

**using** System;

**using** System.Collections.Generic;

**public** **partial** **class** Employee

**{**

**public** **int** EmployeeId **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Gender **{** **get**; **set**; **}**

**public** string City **{** **get**; **set**; **}**

**public** Nullable**<decimal>** Salary **{** **get**; **set**; **}**

**public** Nullable**<int>** DepartmentId **{** **get**; **set**; **}**

**public** **virtual** Department Department **{** **get**; **set**; **}**

**}**

**}**

###### **Department.cs File**

**namespace** *CRUD\_Using\_EF.Models*

**{**

**using** System;

**using** System.Collections.Generic;

**public** **partial** **class** Department

**{**

**[**System.Diagnostics.CodeAnalysis.SuppressMessage**(**"Microsoft.Usage", "CA2214:DoNotCallOverridableMethodsInConstructors"**)]**

**public** Department**()**

**{**

this.Employees = new HashSet**<**Employee**>()**;

**}**

**public** **int** Id **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**[**System.Diagnostics.CodeAnalysis.SuppressMessage**(**"Microsoft.Usage", "CA2227:CollectionPropertiesShouldBeReadOnly"**)]**

**public** **virtual** ICollection**<**Employee**>** Employees **{** **get**; **set**; **}**

**}**

**}**

Here, we need to change the department column header to “**Department Name**” instead of just “**Name**“.  In order to do this add a class file with the name **PDepartment.cs** within the “**Models**” folder. Once you created the **PDepartment.cs** class file then simply copy and paste the following code in it.

**namespace** *CRUD\_Using\_EF.Models*

**{**

**[**MetadataType**(**typeof**(**DepartmentMetaData**))]**

**public** **partial** **class** Department

**{**

**}**

**public** **class** DepartmentMetaData

**{**

**[**Display**(**Name = "Department Name"**)]**

**public** string Name **{** **get**; **set**; **}**

**}**

**}**

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

If you are wondering why can’t we apply the “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.

##### ****Customizing the auto-generated create view.****

At the moment none of the fields on the “**Create**” view are required. That means when we click on the “**Create**” button without filling out any data NULL values are stored in all the columns of the Employee table.

##### ****How to make these fields on the “Create” view required?****

Add **[Required]** attribute to the properties of 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 the name **PEmployee.cs** within the “**Models**” folder. Once you created the **PEmployee.cs** class file, then simply copy and paste the following code into it.

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

**namespace** *CRUD\_Using\_EF.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**]**

**public** string Salary **{** **get**; **set**; **}**

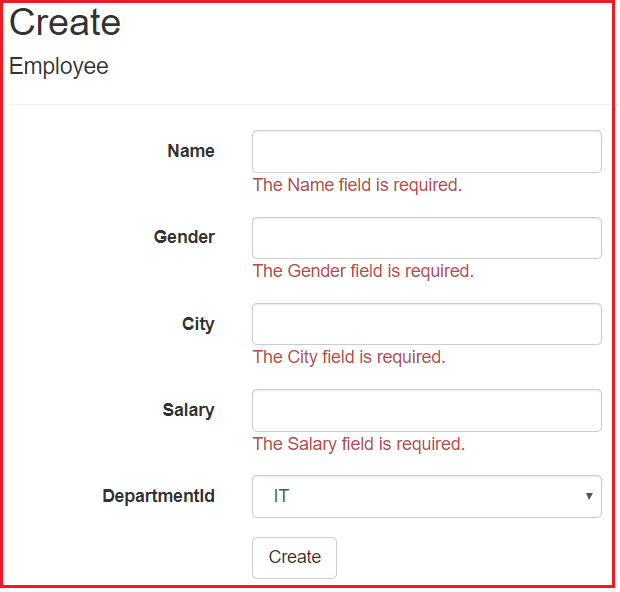
**[**Required**]**

**public** **int** DepartmentId **{** **get**; **set**; **}**

**}**

**}**

At this point run the application and click on the “**Create**” button without filling out any data. Notice that we get validation error messages as expected as shown in the below image.



If you want “**Select Department**” as the first item in the “**Department**” dropdownlist on “**Create**” view then Replace the following code in the Create.cshtml view.

**@Html.DropDownList(“DepartmentId”,null , htmlAttributes: new { @class = “form-control” })**  
with  
**@Html.DropDownList(“DepartmentId”,null , “Select Department”, htmlAttributes: new { @class = “form-control” })**

Notice that a textbox is used for gender. It is an ideal situation to have a drop down list for gender rather than a textbox. To achieve this make the following changes to the “**Create.cshtml**” view. Replace the following code.

**@Html.EditorFor(model => model.Gender, new { htmlAttributes = new { @class = “form-control” } })**  
with  
**@Html.DropDownList(“Gender”, new List<SelectListItem>**  
**{**  
**new SelectListItem { Text = “Male”, Value=”Male” },**  
**new SelectListItem { Text = “Female”, Value=”Female” }**  
**}, “Select Gender”, new { @class = “form-control” })**

##### ****Below is the complete code for Create.cshtml view****

@model CRUD\_Using\_EF.Models.Employee

@{

ViewBag.Title = "Create";

}

**<h2>**Create**</h2>**

@using (Html.BeginForm())

{

@Html.AntiForgeryToken()

**<div** class="form-horizontal"**>**

**<h4>**Employee**</h4>**

**<hr** **/>**

@Html.ValidationSummary(true, "", new { @class = "text-danger" })

**<div** class="form-group"**>**

@Html.LabelFor(model => model.Name, htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.EditorFor(model => model.Name, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.Name, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

@Html.LabelFor(model => model.Gender, htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.DropDownList("Gender", new List**<SelectListItem>**

{

new SelectListItem { Text = "Male", Value="Male" },

new SelectListItem { Text = "Female", Value="Female" }

}, "Select Gender", new { @class = "form-control" })

@Html.ValidationMessageFor(model => model.Gender, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

@Html.LabelFor(model => model.City, htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.EditorFor(model => model.City, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.City, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

@Html.LabelFor(model => model.Salary, htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.EditorFor(model => model.Salary, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.Salary, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

@Html.LabelFor(model => model.DepartmentId, "DepartmentId", htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.DropDownList("DepartmentId",null , "Select Department", htmlAttributes: new { @class = "form-control" })

@Html.ValidationMessageFor(model => model.DepartmentId, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

**<div** class="col-md-offset-2 col-md-10"**>**

**<input** type="submit" value="Create" class="btn btn-default" **/>**

**</div>**

**</div>**

**</div>**

}

**<div>**

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

**</div>**

**<script** src="~/Scripts/jquery-1.10.2.min.js"**></script>**

**<script** src="~/Scripts/jquery.validate.min.js"**></script>**

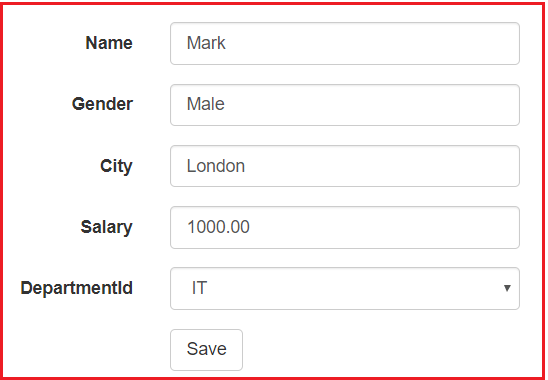
**<script** src="~/Scripts/jquery.validate.unobtrusive.min.js"**></script>**

# Customizing Auto Generated Edit View

## ****Customizing Auto-Generated Edit View in ASP.NET MVC Application****

##### ****Customizing Auto-Generated Edit View****

Run the application and navigate to the Index view. Click on the Edit button which will take you to edit view to edit an employee which should display as shown below.



In the edit view if you want **“Select Department”** as the first item in the **“Department”** dropdown list then replace the following line.

**@Html.DropDownList(“DepartmentId”,null , htmlAttributes: new { @class = “form-control” })**  
**with**  
**@Html.DropDownList(“DepartmentId”,null , “Select Department”, htmlAttributes: new { @class = “form-control” })**

Notice that a textbox is used for gender. It is ideal to have a dropdown list for gender rather than a textbox. To achieve this, make the following changes to the **“Edit.cshtml”** view. Replace the following code

**@Html.EditorFor(model => model.Gender, new { htmlAttributes = new { @class = “form-control” } })**  
**with**  
**@Html.DropDownList(“Gender”, new List<SelectListItem>**  
**{**  
**new SelectListItem { Text = “Male”, Value=”Male” },**  
**new SelectListItem { Text = “Female”, Value=”Female” }**  
**}, “Select Gender”, new { @class = “form-control” })**

##### ****Creating a Field as Non Editable:****

Let’s make employee name as non-editable. To achieve this, change the following code in Edit.cshtml view  
**@Html.EditorFor(model =>model.Name)**  
**TO**  
**@Html.DisplayFor(model =>model.Name)**  
**@Html.HiddenFor(model =>model.Name)**

**After all the modifications below is our code for Edit.cshtml view**

@model CRUD\_Using\_EF.Models.Employee

@{

ViewBag.Title = "Edit";

}

**<h2>**Edit**</h2>**

@using (Html.BeginForm())

{

@Html.AntiForgeryToken()

**<div** class="form-horizontal"**>**

**<h4>**Employee**</h4>**

**<hr** **/>**

@Html.ValidationSummary(true, "", new { @class = "text-danger" })

@Html.HiddenFor(model => model.EmployeeId)

**<div** class="form-group"**>**

@Html.LabelFor(model => model.Name, htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.DisplayFor(model => model.Name, new { htmlAttributes = new { @class = "form-control" } })

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

@Html.ValidationMessageFor(model => model.Name, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

@Html.LabelFor(model => model.Gender, htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.DropDownList("Gender", new List**<SelectListItem>**

{

new SelectListItem { Text = "Male", Value="Male" },

new SelectListItem { Text = "Female", Value="Female" }

}, "Select Gender", new { @class = "form-control" })

@Html.ValidationMessageFor(model => model.Gender, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

@Html.LabelFor(model => model.City, htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.EditorFor(model => model.City, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.City, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

@Html.LabelFor(model => model.Salary, htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.EditorFor(model => model.Salary, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.Salary, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

@Html.LabelFor(model => model.DepartmentId, "DepartmentId", htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.DropDownList("DepartmentId", null, "Select Department", htmlAttributes: new { @class = "form-control" })

@Html.ValidationMessageFor(model => model.DepartmentId, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

**<div** class="col-md-offset-2 col-md-10"**>**

**<input** type="submit" value="Save" class="btn btn-default" **/>**

**</div>**

**</div>**

**</div>**

}

**<div>**

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

**</div>**

**<script** src="~/Scripts/jquery-1.10.2.min.js"**></script>**

**<script** src="~/Scripts/jquery.validate.min.js"**></script>**

**<script** src="~/Scripts/jquery.validate.unobtrusive.min.js"**></script>**

At this point, we will still be able to change the **“Name”** property of the employee using tools like Fiddler. There are several ways to prevent **“Name”** property from being updated.

1. Use [**UpdateModel**()](https://dotnettutorials.net/lesson/updatemodel-and-tryupdatemodel-asp-net-mvc/) function and pass include and exclude list as a parameter
2. Use [**Bind**](https://dotnettutorials.net/lesson/bind-attribute-asp-net-mvc/)attribute
3. Use [**Interfaces**](https://dotnettutorials.net/lesson/model-binding-using-interface/)

##### ****Prevent Updating using Bind Attribute:****

Now let’s discuss using the **BIND**attribute to prevent updating the **“Name”** property using tools like Fiddler. Along the way, I will demonstrate adding model validation errors dynamically. Change the implementation of the **“Edit”** action method of **Employee** Controller that responds to [**HttpPost**] request as shown below

**[**HttpPost**]**

**[**ValidateAntiForgeryToken**]**

**public** ActionResult Edit**([**Bind**(**Include = "EmployeeId,Gender,City,Salary,DepartmentId"**)]** Employee employee**)**

**{**

Employee employeeFromDB = db.Employees.Single**(**emp =**>** emp.EmployeeId == employee.EmployeeId**)**;

employeeFromDB.Gender = employee.Gender;

employeeFromDB.City = employee.City;

employeeFromDB.Salary = employee.Salary;

employeeFromDB.DepartmentId = employee.DepartmentId;

**if** **(**ModelState.IsValid**)**

**{**

db.Entry**(**employeeFromDB**)**.State = EntityState.Modified;

db.SaveChanges**()**;

**return** RedirectToAction**(**"Index"**)**;

**}**

ViewBag.DepartmentId = new SelectList**(**db.Departments, "Id", "Name", employee.DepartmentId**)**;

**return** View**(**employee**)**;

**}**

Here, we have not included the **“Name”** property from model binding using the **“Bind”**attribute. Even without the **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 the **“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 the “**Save**” button on the “**Edit**” view we get a validation error stating –**The Name field is required**. This is because the “**Name**” property is decorated with the [**Required**] attribute in “**PEmployee.cs**” file. To prevent the validation error, remove the [**Required**] attribute.

###### **Now the PEmployee.cs file looks as follows**

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

**namespace** *CRUD\_Using\_EF.Models*

**{**

**[**MetadataType**(**typeof**(**EmployeeMetaData**))]**

**public** **partial** **class** Employee

**{**

**}**

**public** **class** EmployeeMetaData

**{**

**public** string Name **{** **get**; **set**; **}**

**[**Required**]**

**public** string Gender **{** **get**; **set**; **}**

**[**Required**]**

**public** string City **{** **get**; **set**; **}**

**[**Required**]**

**public** string Salary **{** **get**; **set**; **}**

**[**Required**]**

**public** **int** DepartmentId **{** **get**; **set**; **}**

**}**

**}**

Now run the application and see Edit view is working as expected.

##### ****Adding Model Error Dynamically:****

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

**[**HttpPost**]**

**[**ValidateAntiForgeryToken**]**

**public** ActionResult Create**([**Bind**(**Include = "EmployeeId,Name,Gender,City,Salary,DepartmentId"**)]** Employee employee**)**

**{**

**if** **(**string.IsNullOrEmpty**(**employee.Name**))**

**{**

ModelState.AddModelError**(**"Name", "The Name field is required."**)**;

**}**

**if** **(**ModelState.IsValid**)**

**{**

db.Employees.Add**(**employee**)**;

db.SaveChanges**()**;

**return** RedirectToAction**(**"Index"**)**;

**}**

ViewBag.DepartmentId = new SelectList**(**db.Departments, "Id", "Name", employee.DepartmentId**)**;

**return** View**(**employee**)**;

**}**

That’s it. Run the application and see Create View is also working as expected.