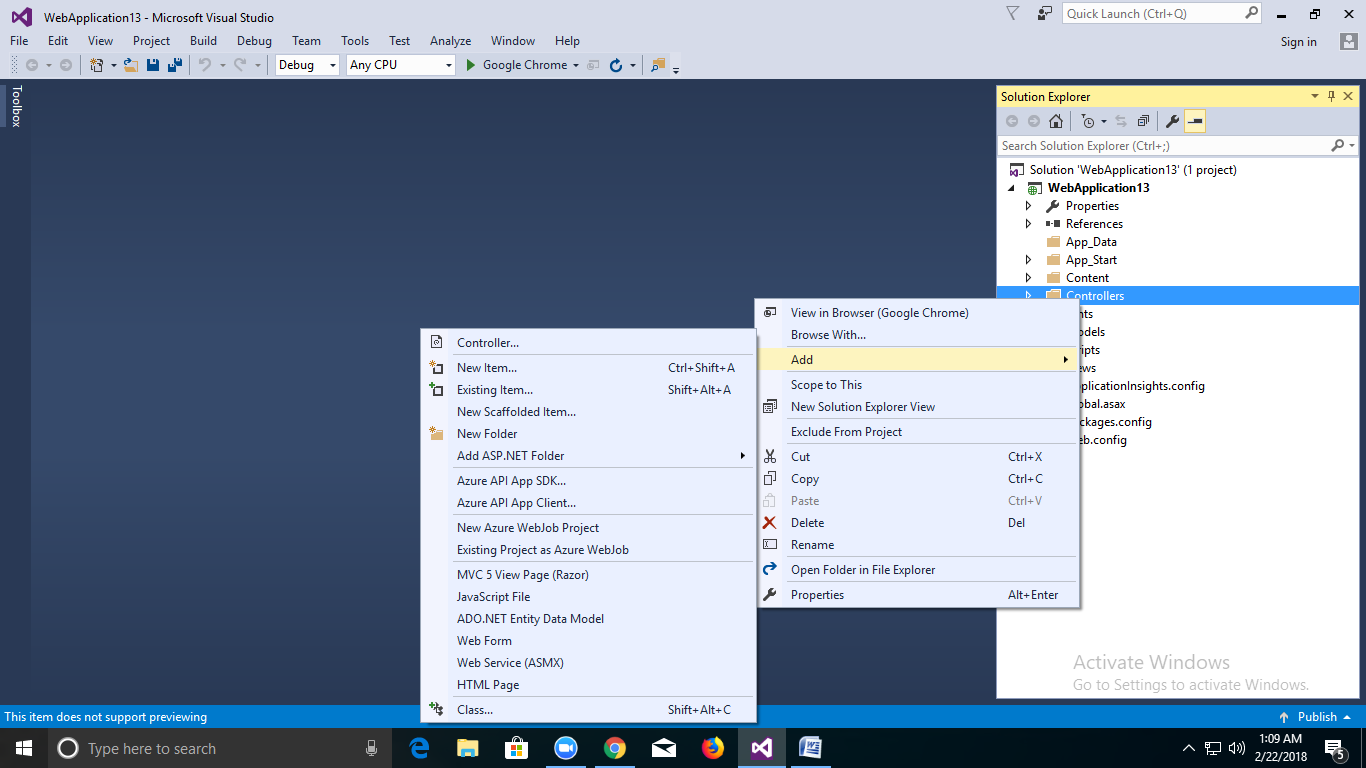
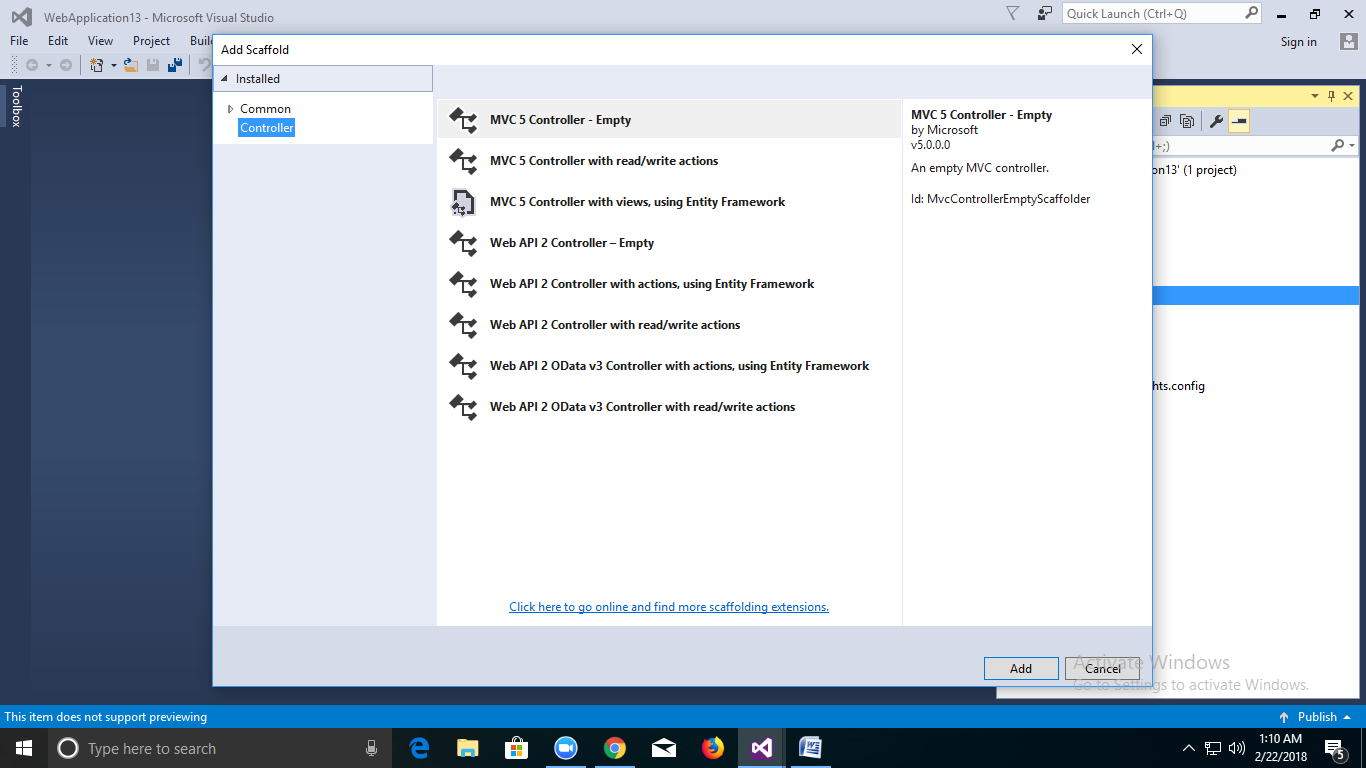
**Asp.Net Mvc Notes**

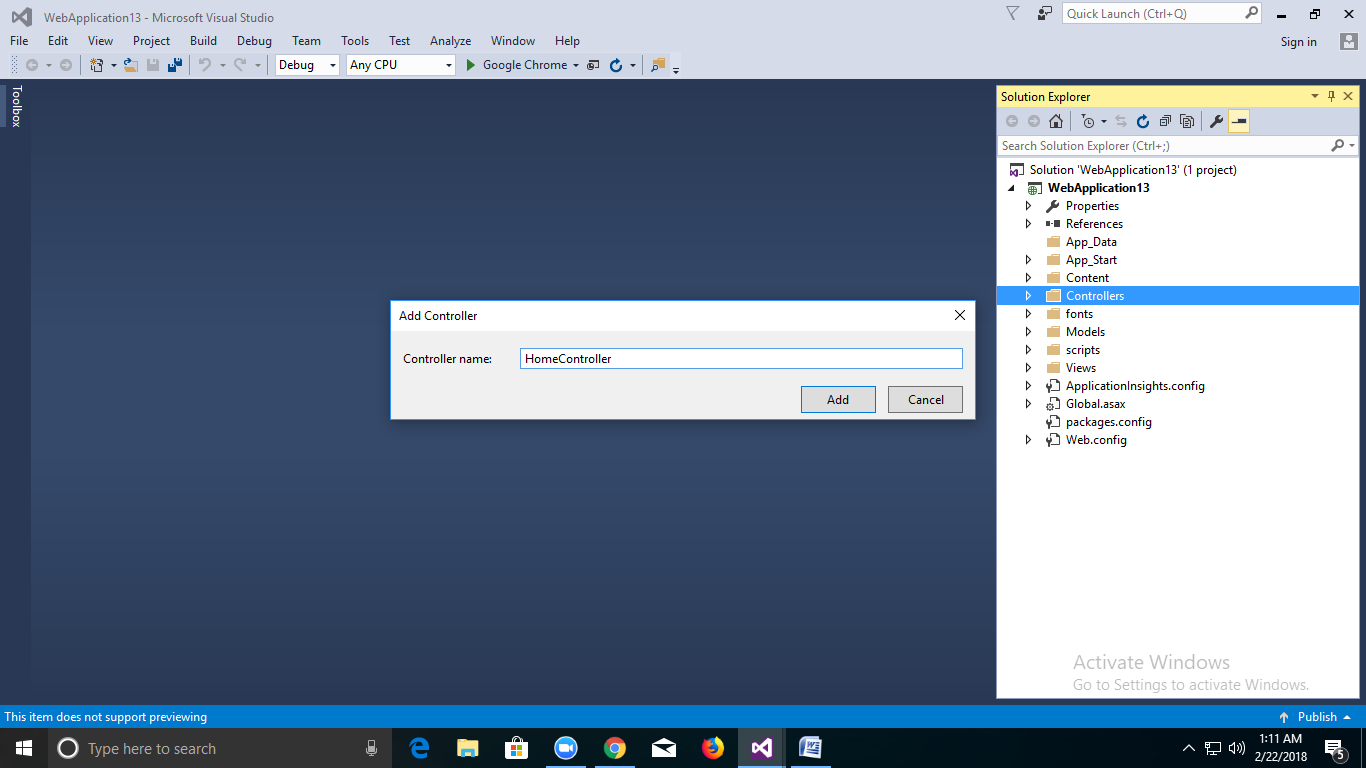
MVC stands for *model-view-controller*. MVC is a pattern for developing applications that are well architected, testable and easy to maintain. MVC-based applications contain:

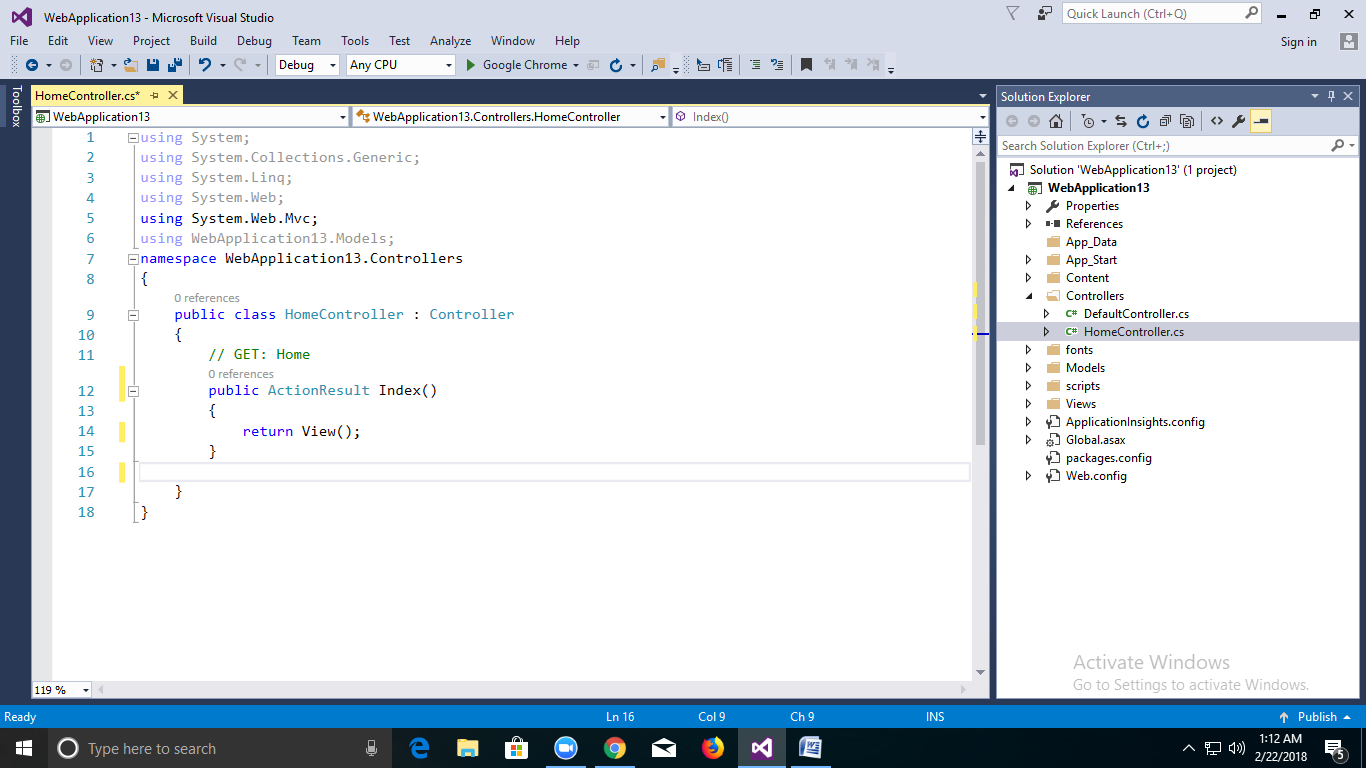
* **M** odels: Classes that represent the data of the application and that use validation logic to enforce business rules for that data.
* **V** iews: Template files that your application uses to dynamically generate HTML responses.
* **C** ontrollers: Classes that handle incoming browser requests, retrieve model data, and then specify view templates that return a response to the browser.

Let's begin by creating a controller class. In **Solution Explorer**, right-click the Controllers folder and then click **Add**, then **Controller**.

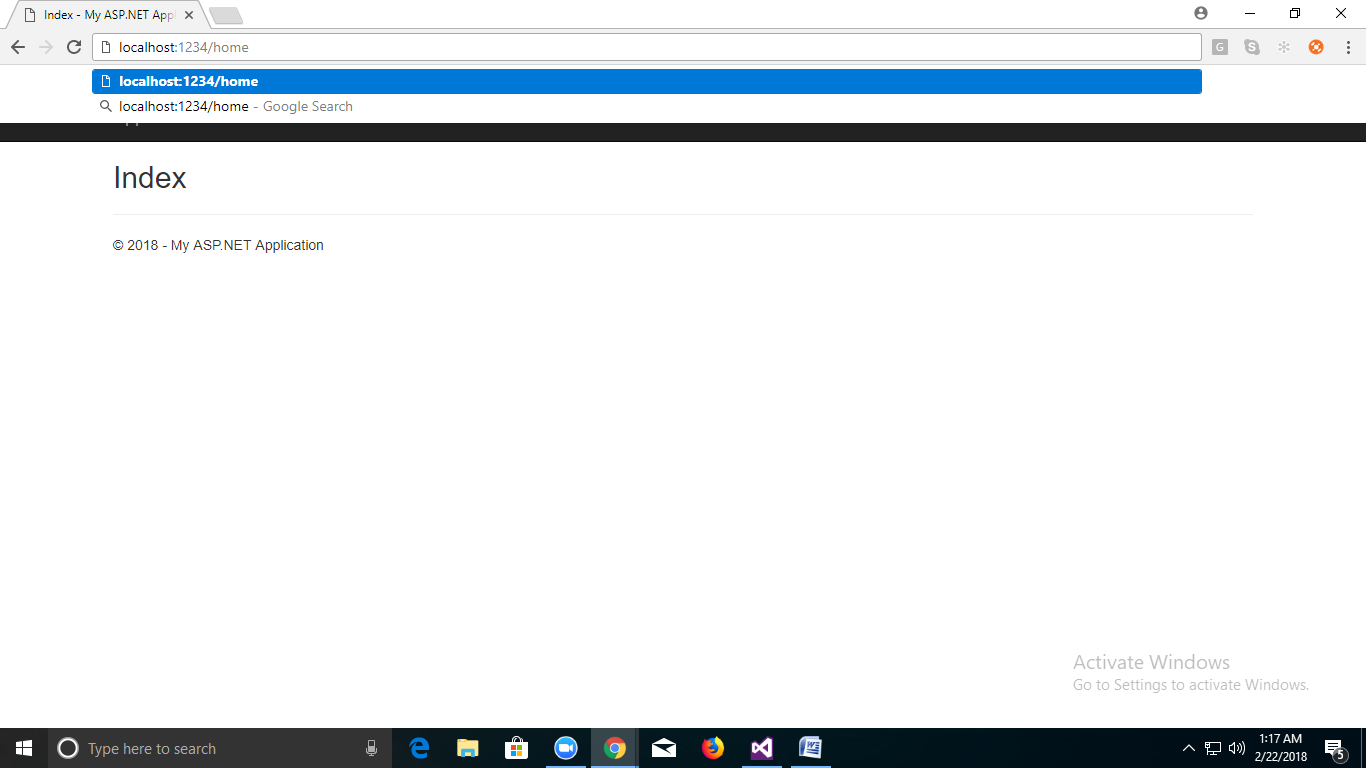
****

****

****

****

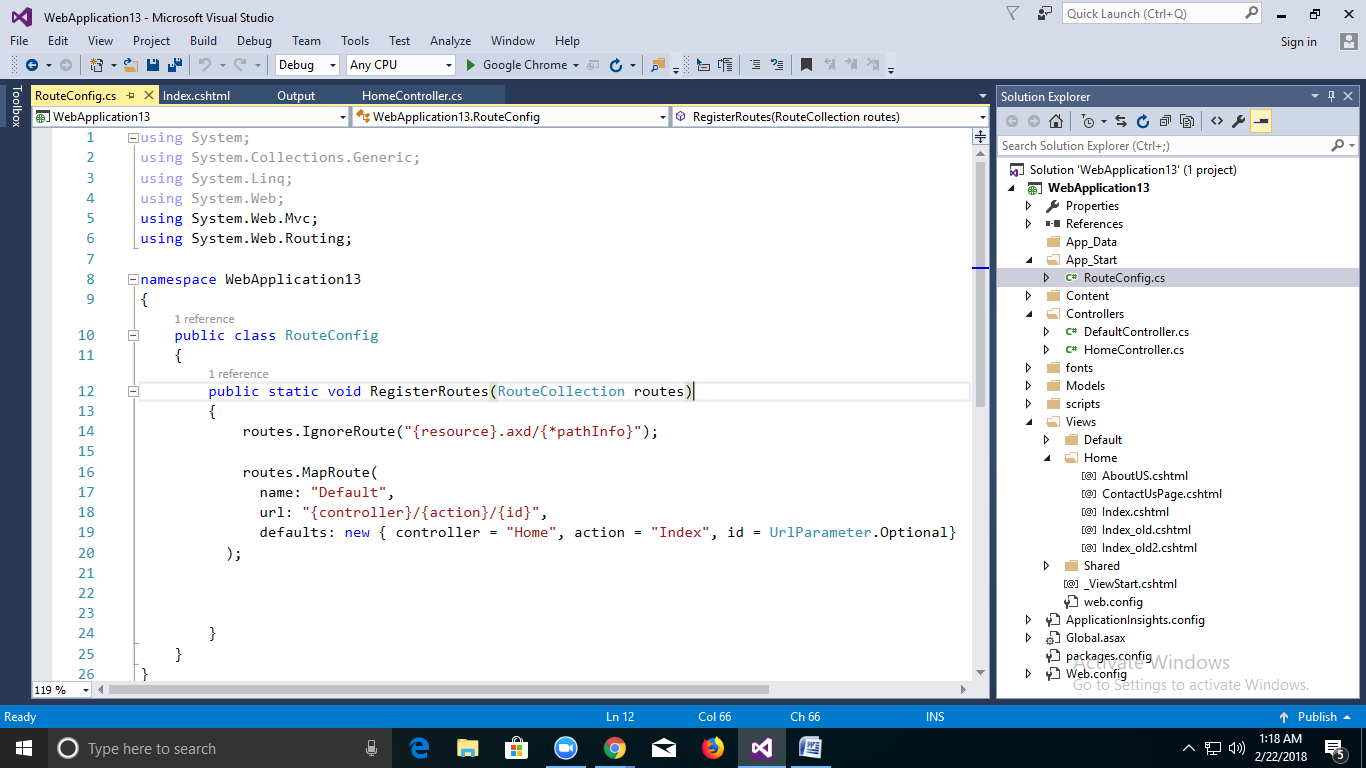
The controller methods will return CSHTML Page as an example. The controller is named HomeController and the first method is named Index. Let's invoke it from a browser. Run the application (press F5 or Ctrl+F5). In the browser, append "Home" to the path in the address bar. (For example, in the illustration below, it's http://localhost:1234/Home.) The page in the browser will look like the following screenshot. In the method above, the code returned a string directly. You told the system to just return some HTML, and it did!

****

ASP.NET MVC invokes different controller classes (and different action methods within them) depending on the incoming URL. The default URL routing logic used by ASP.NET MVC uses a format like this to determine what code to invoke:1

/[Controller]/[ActionName]/[Parameters]

You set the format for routing in the App\_Start/RouteConfig.cs file.

****

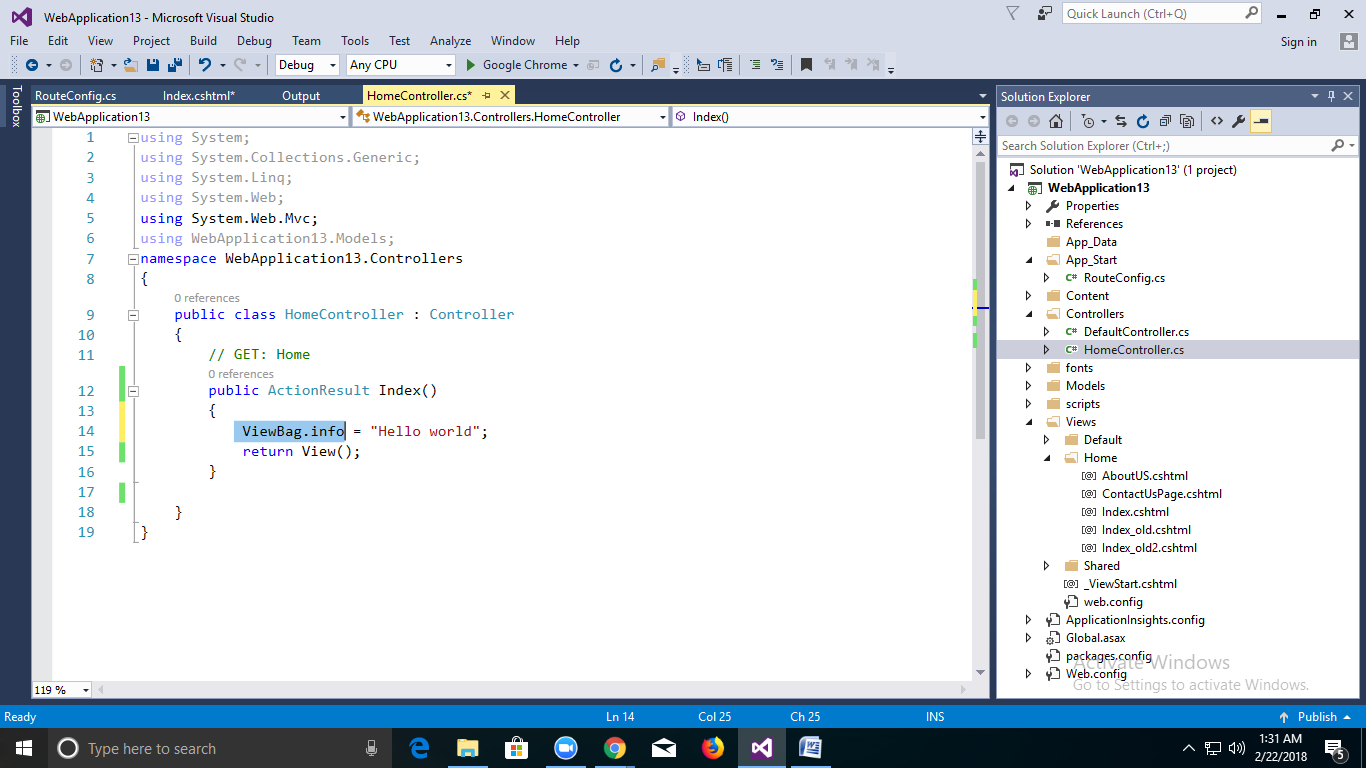
When you run the application and don't supply any URL segments, it defaults to the "Home" controller and the "Index" action method specified in the defaults section of the code above.

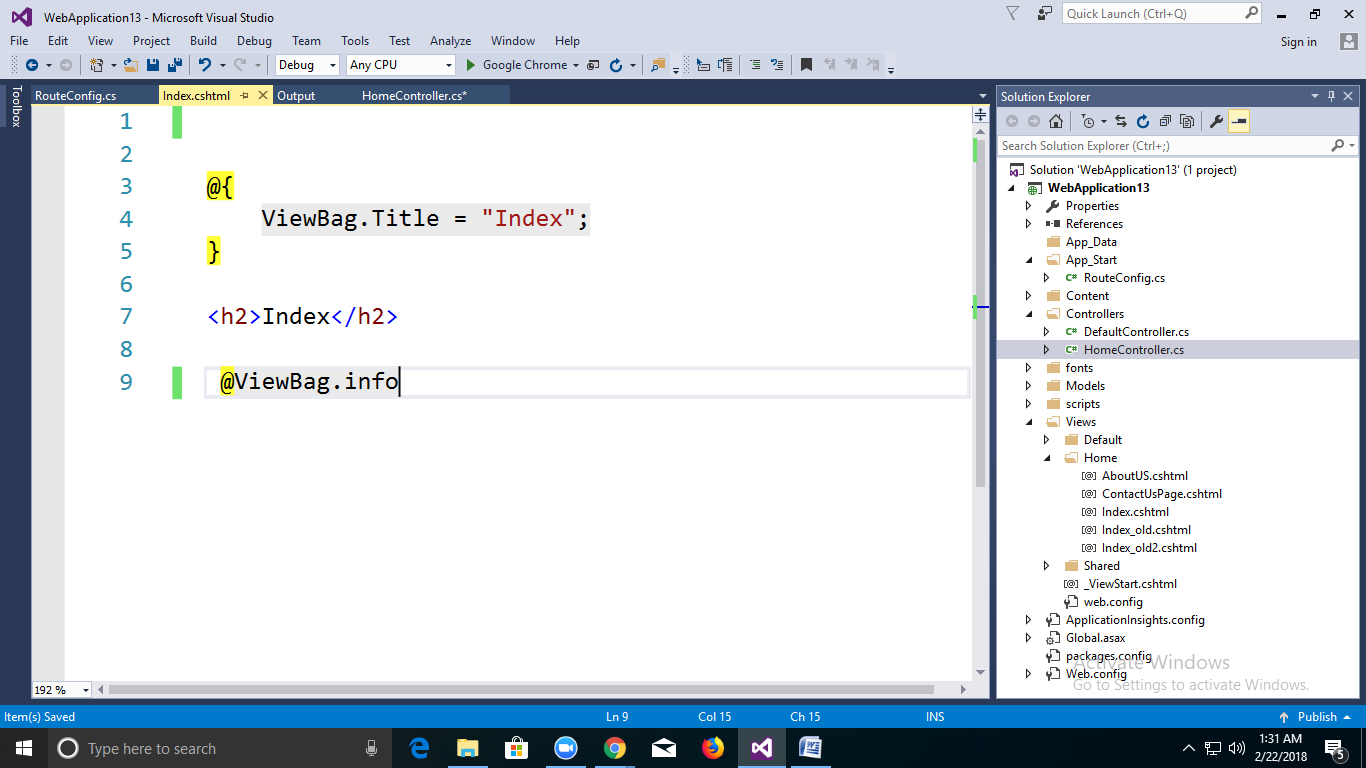
The first part of the URL determines the controller class to execute. So /Home  maps to the HomeController class. The second part of the URL determines the action method on the class to execute. So /HelloWorld/Index would cause the Index method of the HomeController class to execute. Notice that we only had to browse to /Home and the Index method was used by default. This is because a method named Index is the default method that will be called

**Lets Learn How to pass data from Controller to View**

There are various ways to pass data from a Controller to a View. I'm going to discuss how Controllers interact with Views and specifically cover ways you can pass data from a Controller to a View to render a response back to a client. So, let's get started.  
  
**ViewBag**  
  
ViewBag is a very well known way to pass the data from Controller to View & even View to View. ViewBag uses the dynamic feature that was added in C# 4.0. We can say ViewBag=ViewData + Dynamic wrapper around the ViewData dictionary. Let's see how it is used.

**Between Controller and View**

****

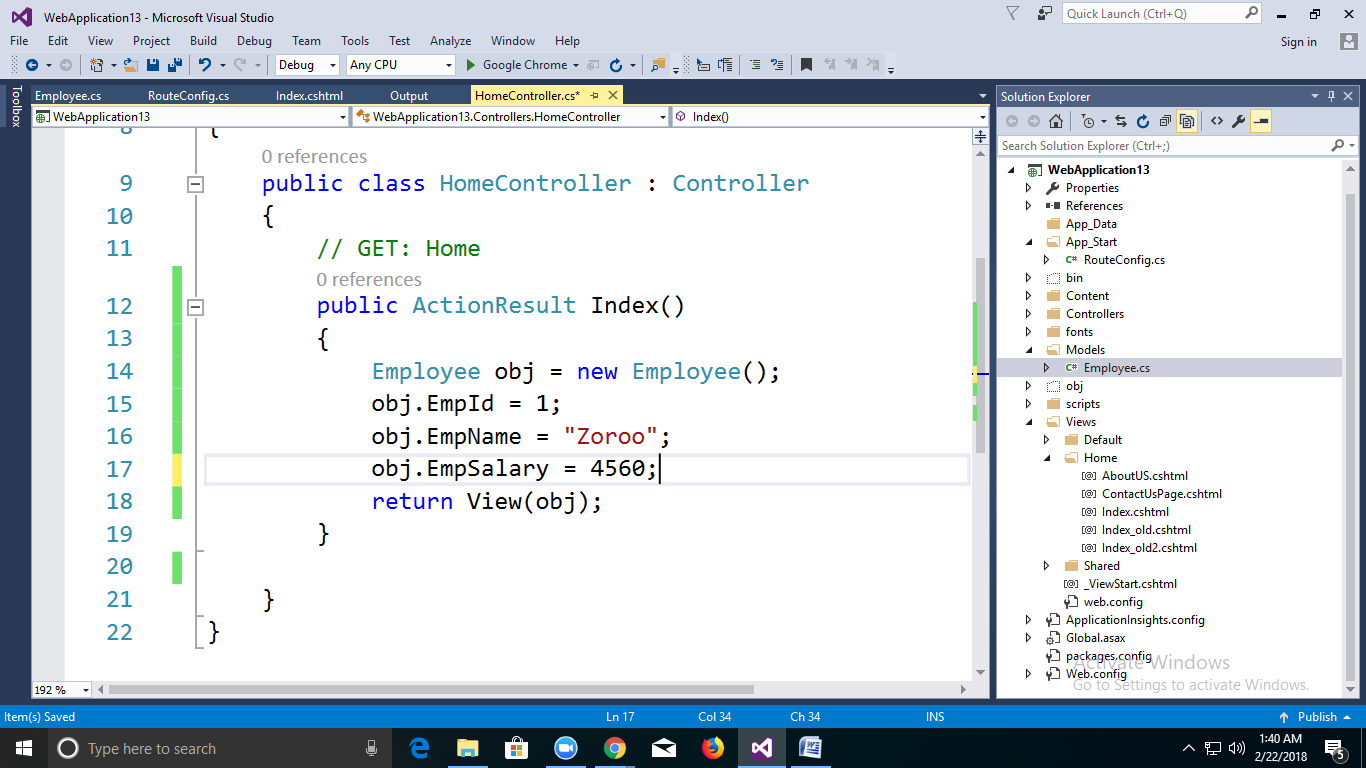
****

**ViewModel**  
  
Using ViewModel we can also pass the data from the Controller to View; let's look at the image.

**Step 1:**

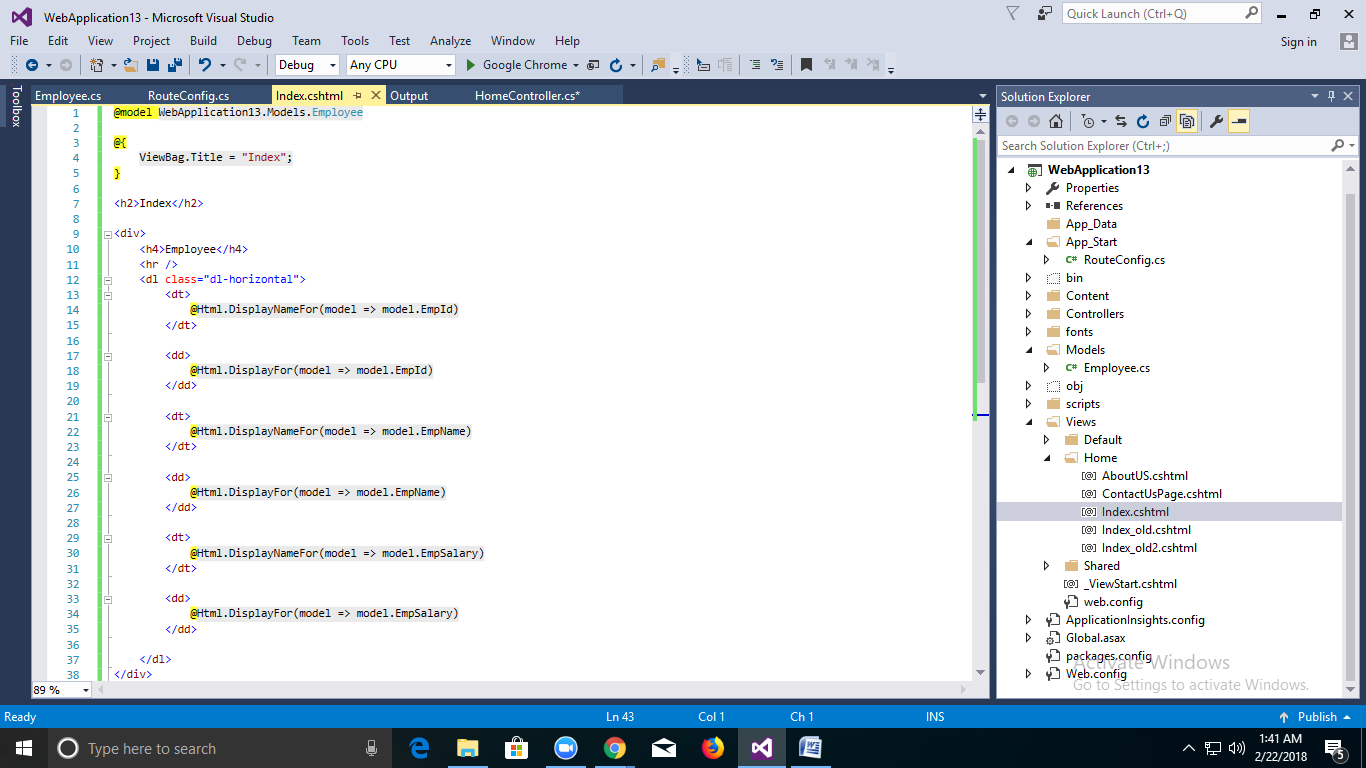
**Create Model Employee Class in Model Folder:**

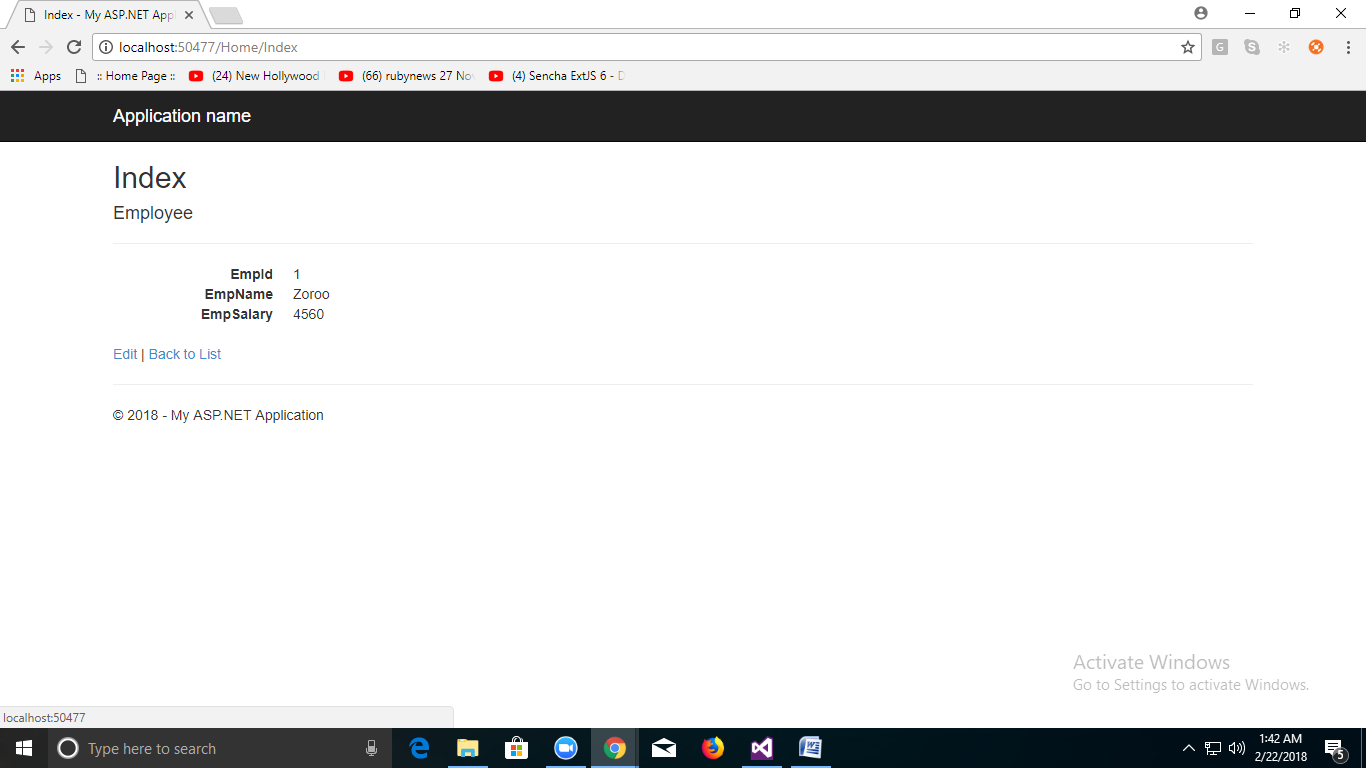
****

****

**Step 2:**

**Create View:**

****

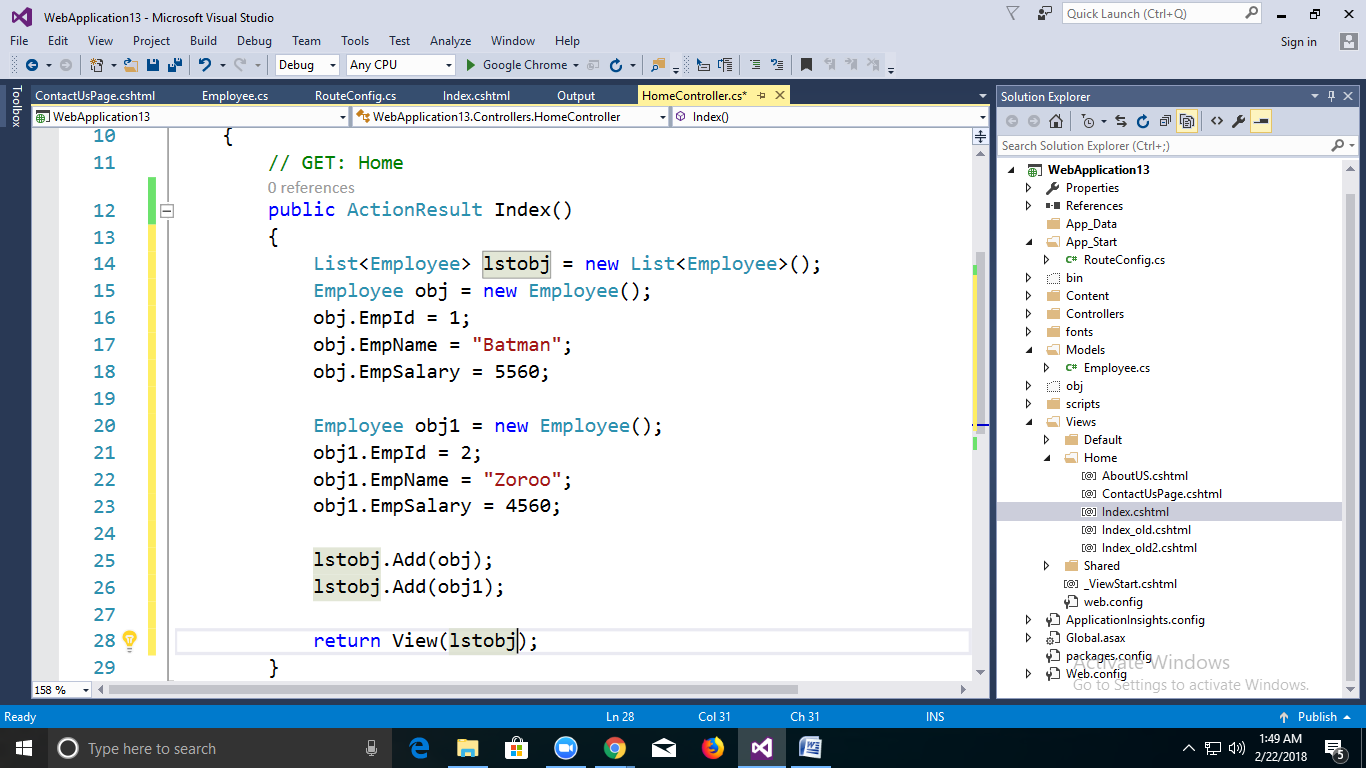
****

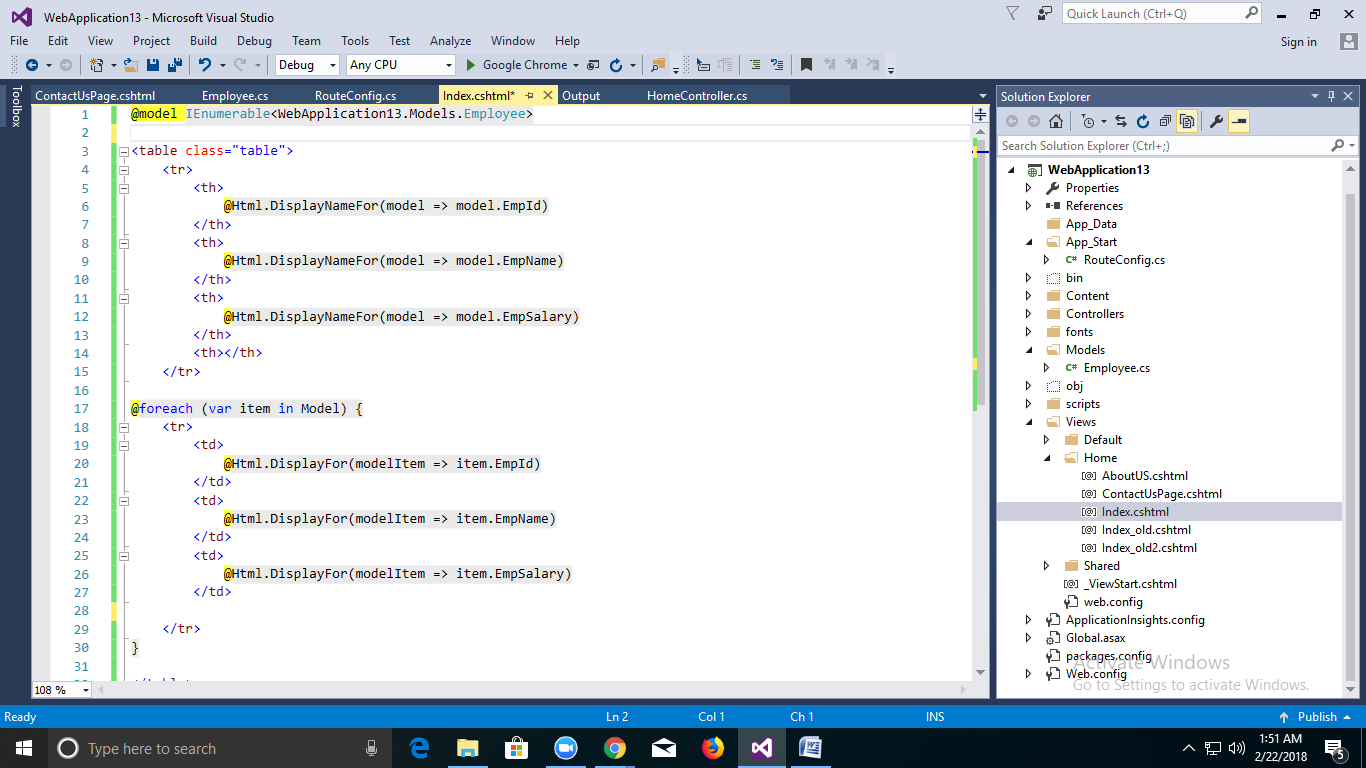
Here @model is variable which going to get the value from controller. Which is of type Webapplication13.Model.Employee

@Html.DisplayNameFor is HtmlHelper to display model Name

@Html.DisplayFor is HtmlHelper to display model Value.

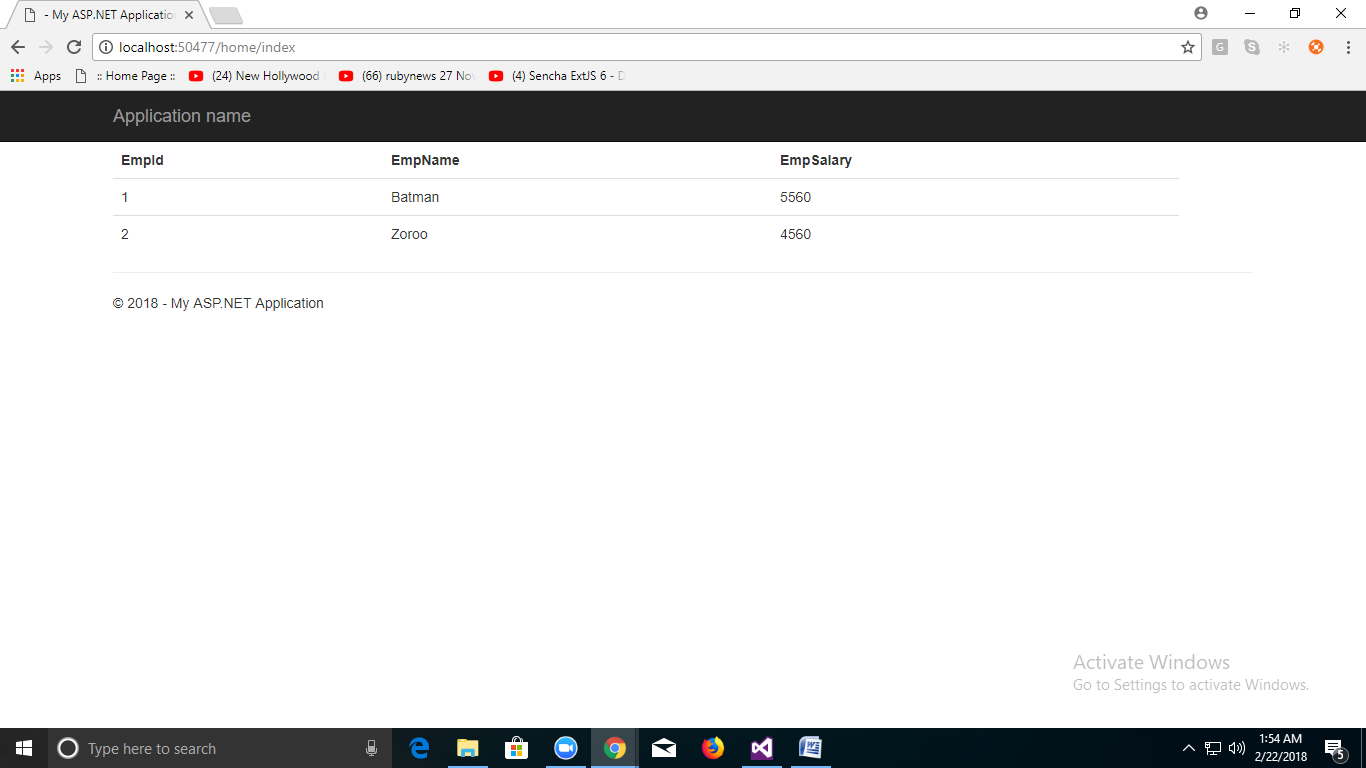
B)Multiple Name from Controller to View.





Now @model Variable is Holding Multiple Values using Listobj

@model IEnumerable<WebApplication13.Models.Employee>



**Action Result In Mvc**

In ASP.NET, MVC has different types of Action Results. Each action result returns a different format of output. A programmer uses different action results to get expected output. Action Results return the result to view the page for the given request.

**Definition**Action Result is a result of action methods or return types of action methods. Action result is an abstract class. It is a base class for all type of action results.

ActionResult

Redirect

Content

RedirectToAction

FileResult

JsonResult

Partial View

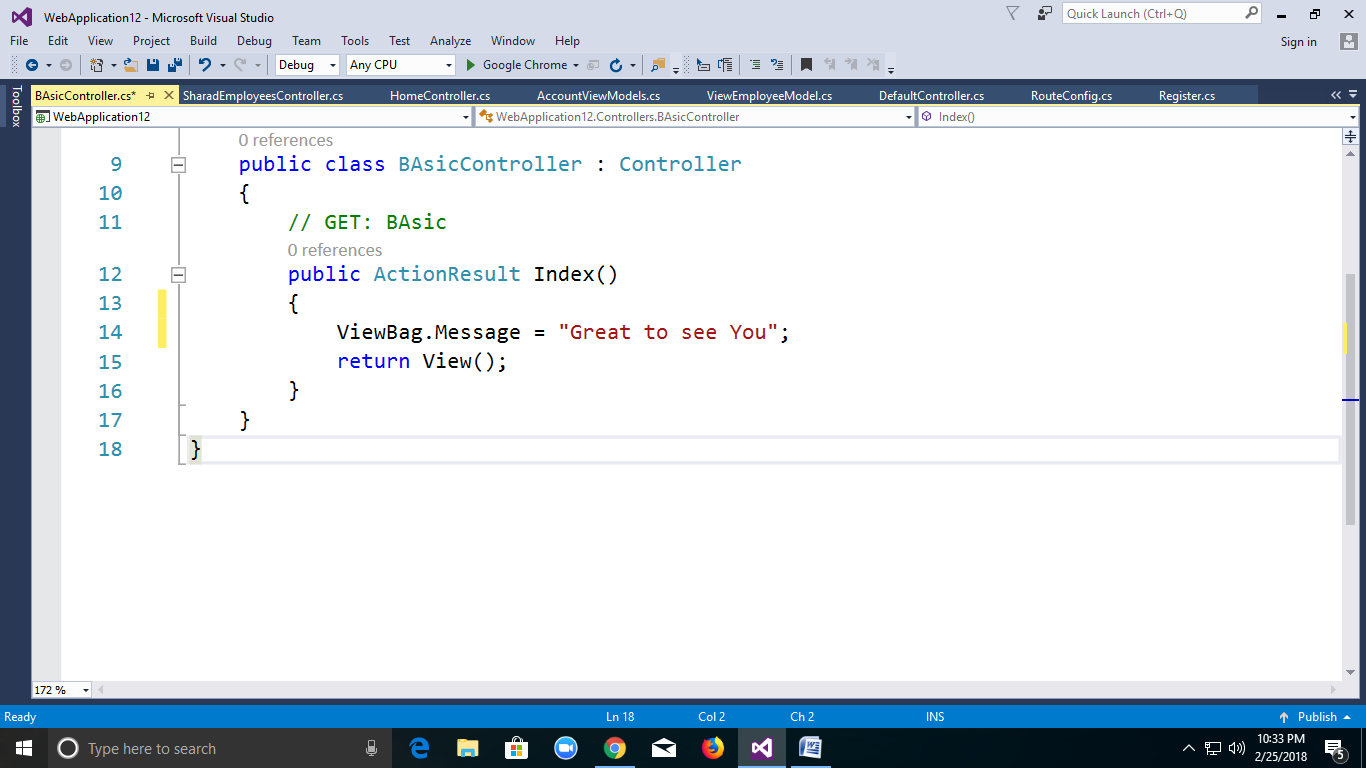
ViewResult

There are different Types of action results in ASP.NET MVC. Each result has a different type of result format to view page.

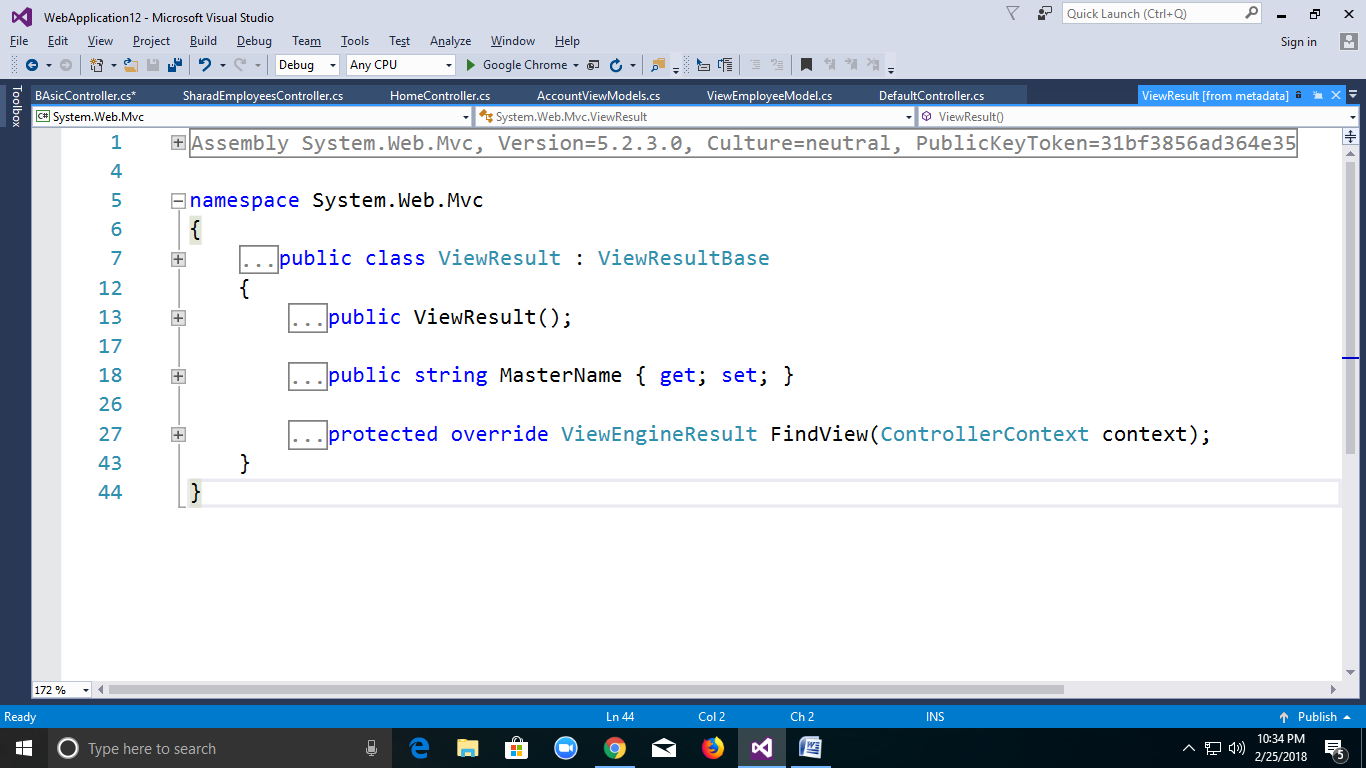
* View Result
* Partial View Result
* Redirect Result
* Redirect To Action Result
* Redirect To Route Result
* Json Result
* File Result
* Content Result

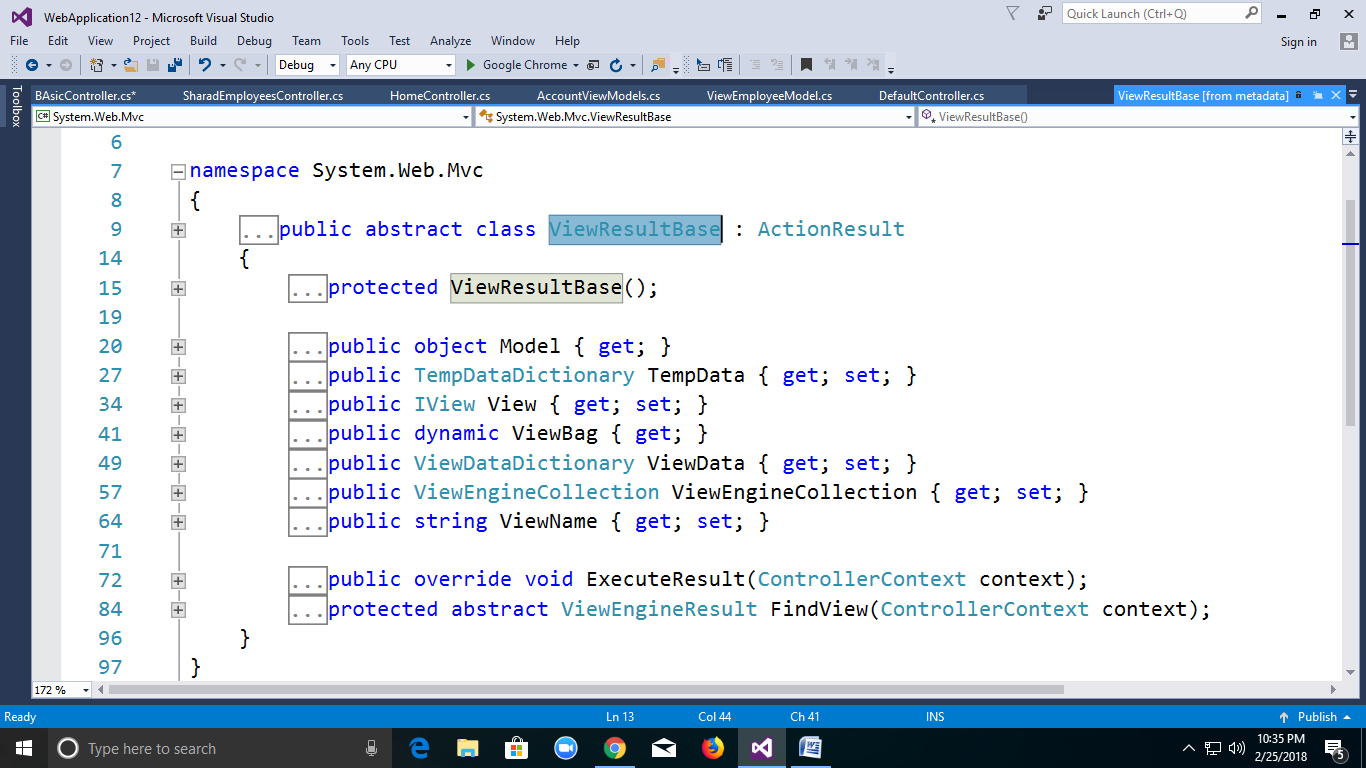
View Result

View result is a basic view result. It returns basic results to view page. View result can return data to view page through which class is defined in the model. View page is a simple HTML page. Here view page has “.cshtm” extension.



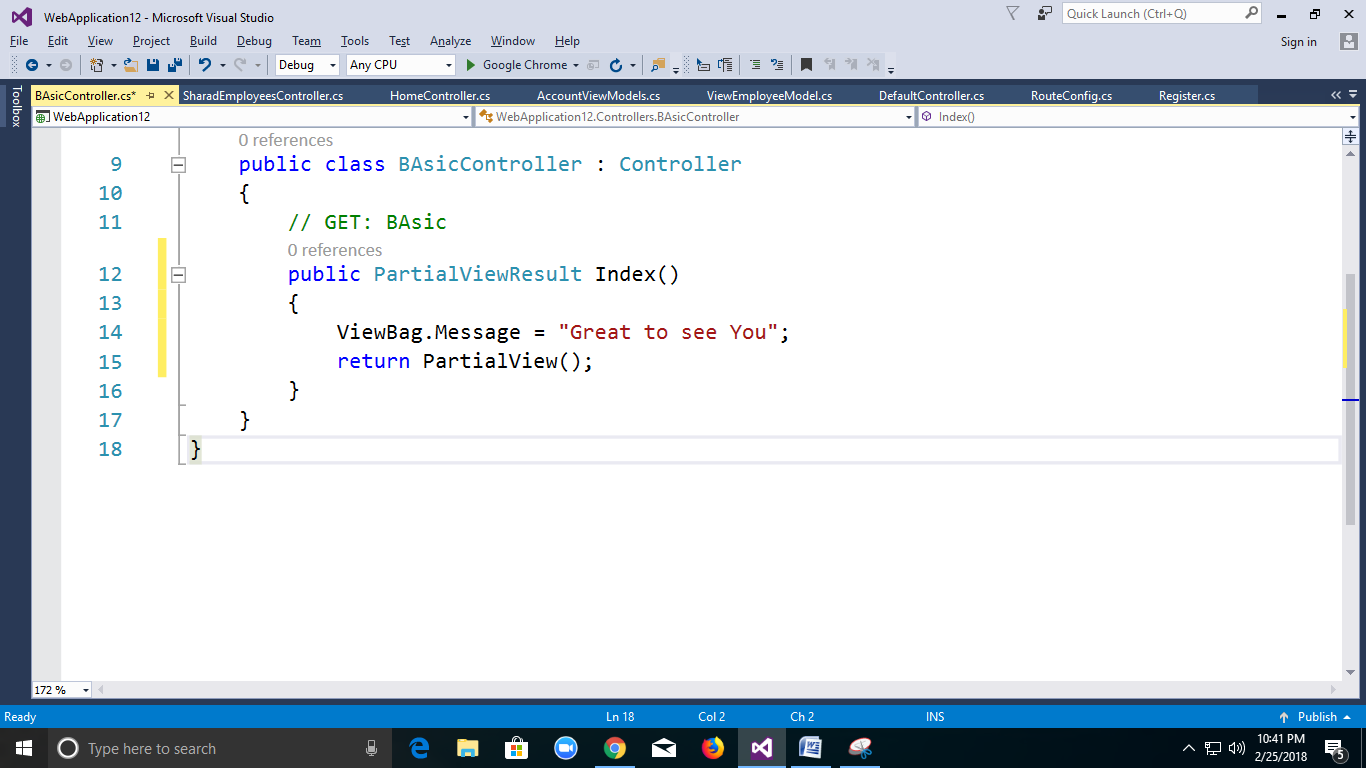
View Result is a class and is derived from “ViewResultBase” class. “ViewResultBase” is derived from Action Result. View Result base class is an Action Result. Action Result is a base class of different action result.

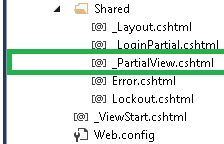




View Result class is inherited from Action Result class by View Result Base class. The ScreenShort shown above describes inheritance of Action Results.

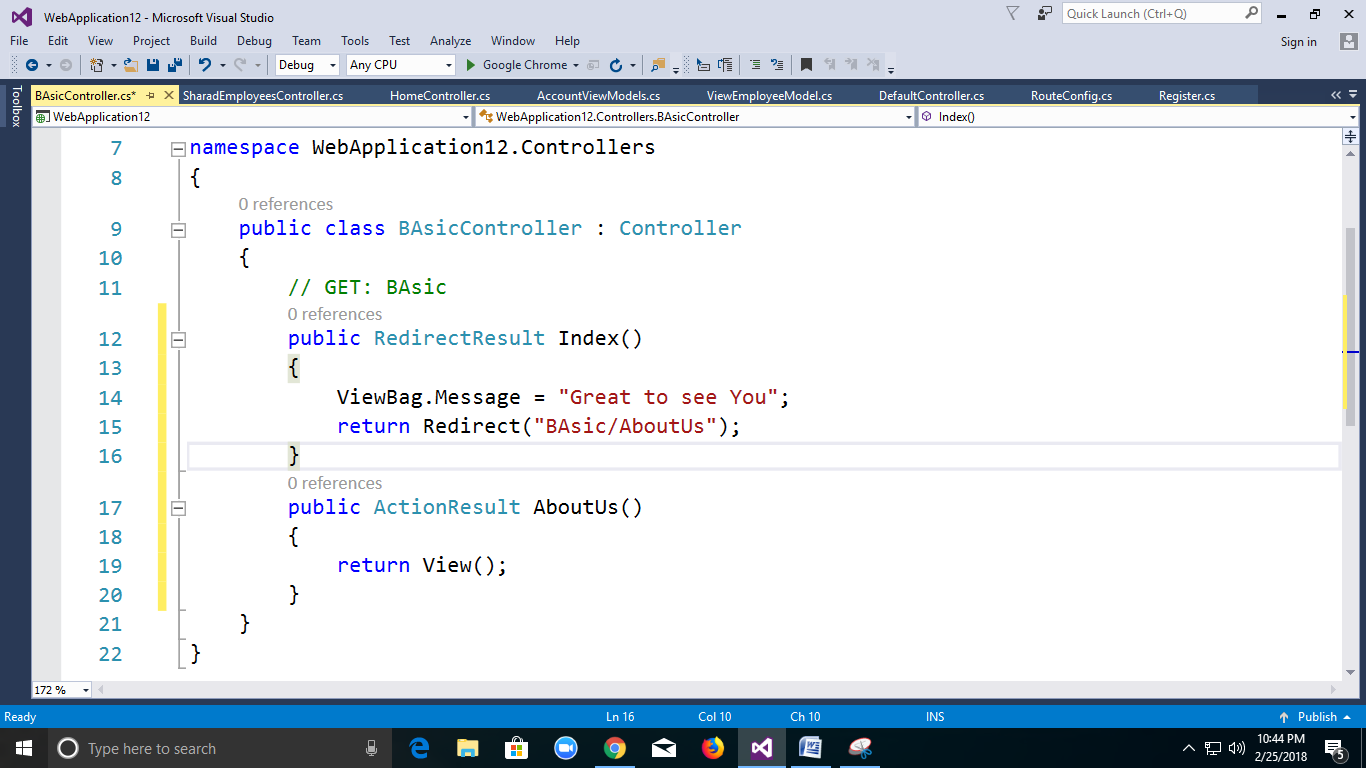
**Partial View Result**  
Partial View Result is returning the result to Partial view page. Partial view is one of the views that we can call inside Normal view page.



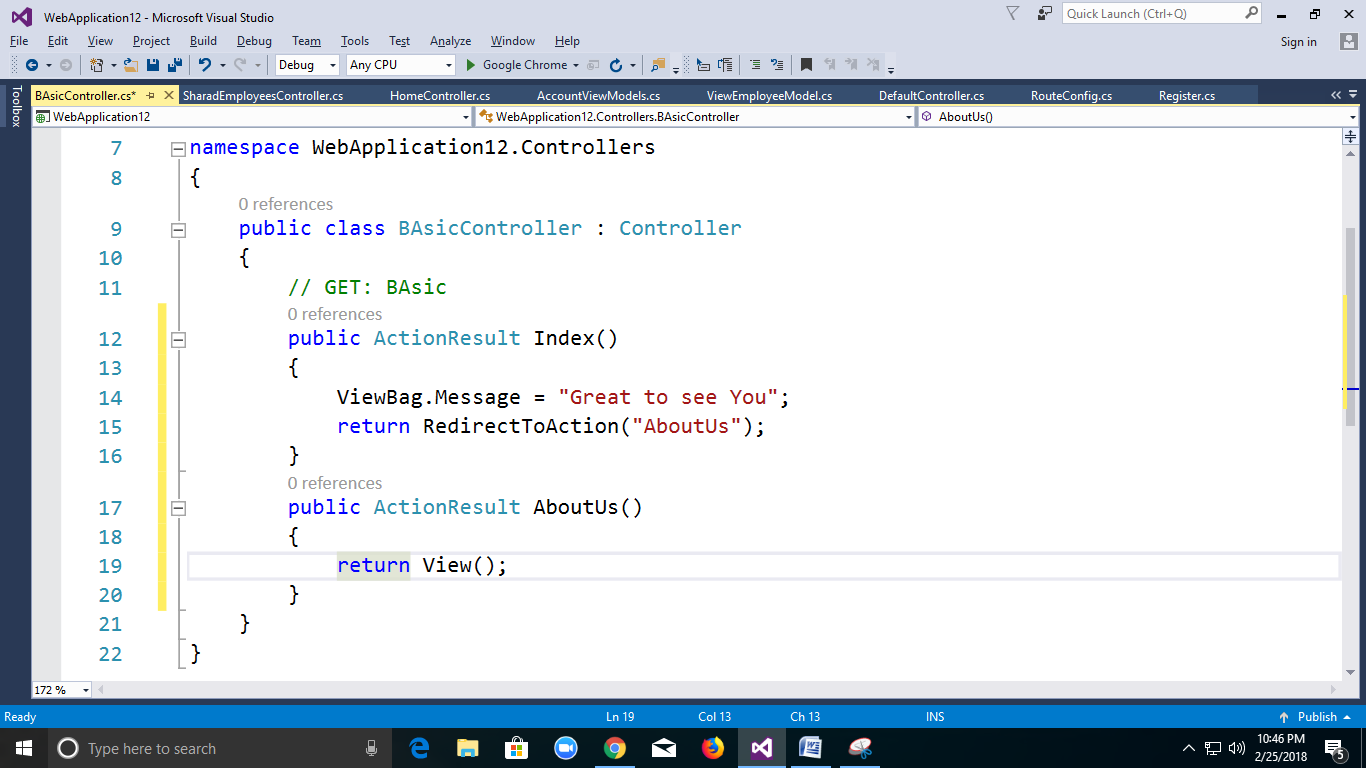


We should create a Partial view inside shared folder, otherwise we cannot access the Partial view. The diagram is shown above the Partial view page and Layout page because layout page is a Partial view. Partial View Result class is also derived from Action Result.

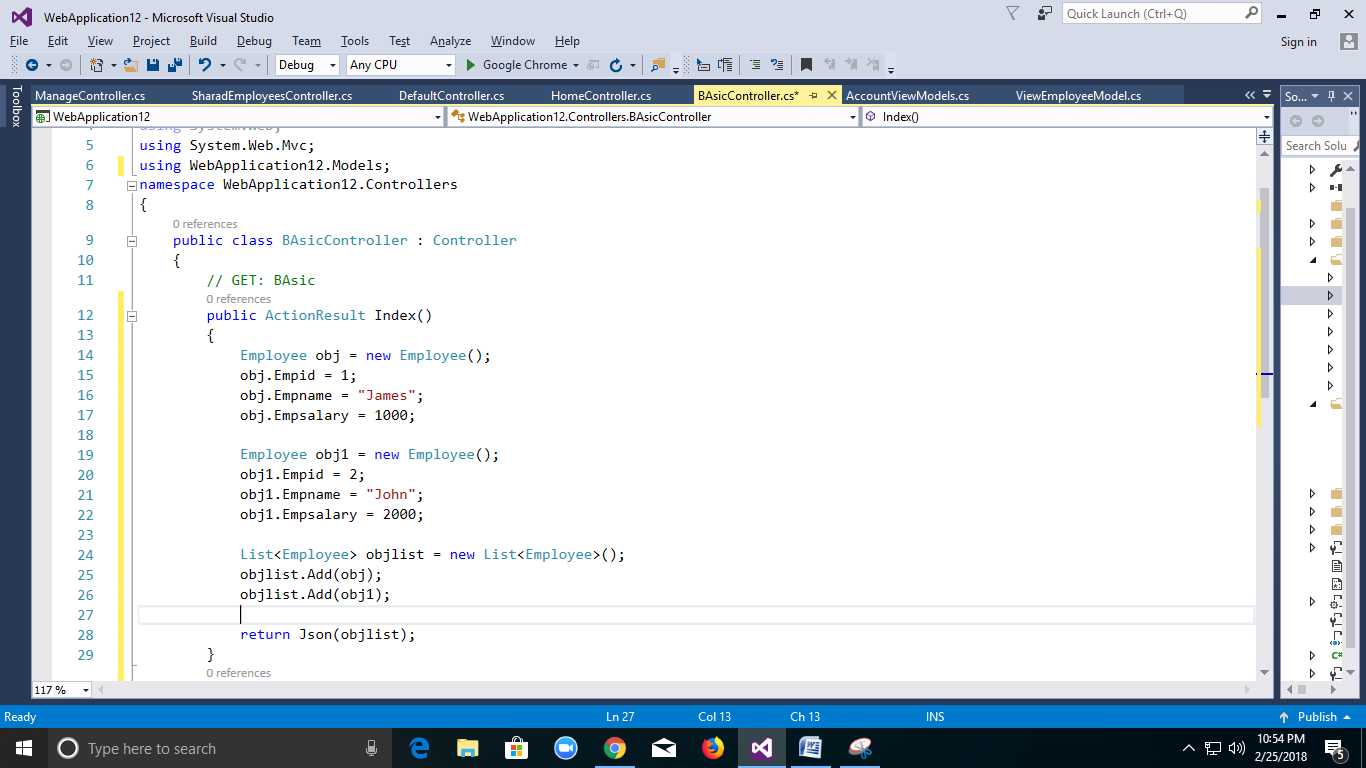
**Redirect Result**Redirect result is returning the result to specific URL. It is rendered to the page by URL. If it gives wrong URL, it will show 404 page errors.

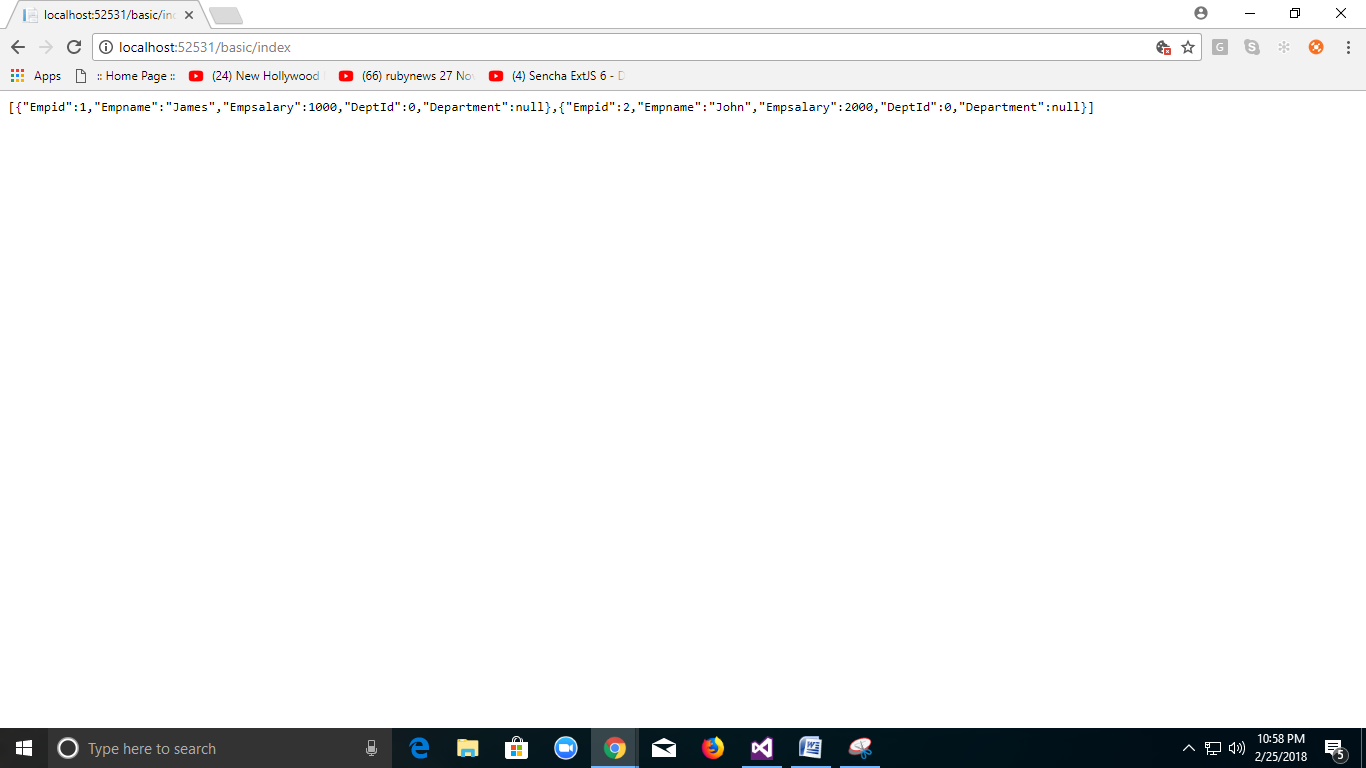


**Redirect to Action Result**Redirect to Action result is returning the result to a specified controller and action method. Controller name is optional in Redirect to Action method. If not mentioned, Controller name redirects to a mentioned action method in current Controller. Suppose action name is not available but mentioned in the current controller, then it will show 404 page error.

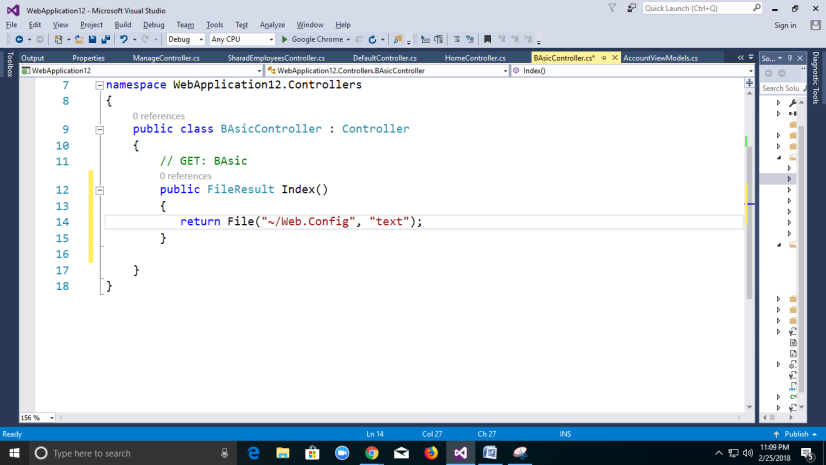


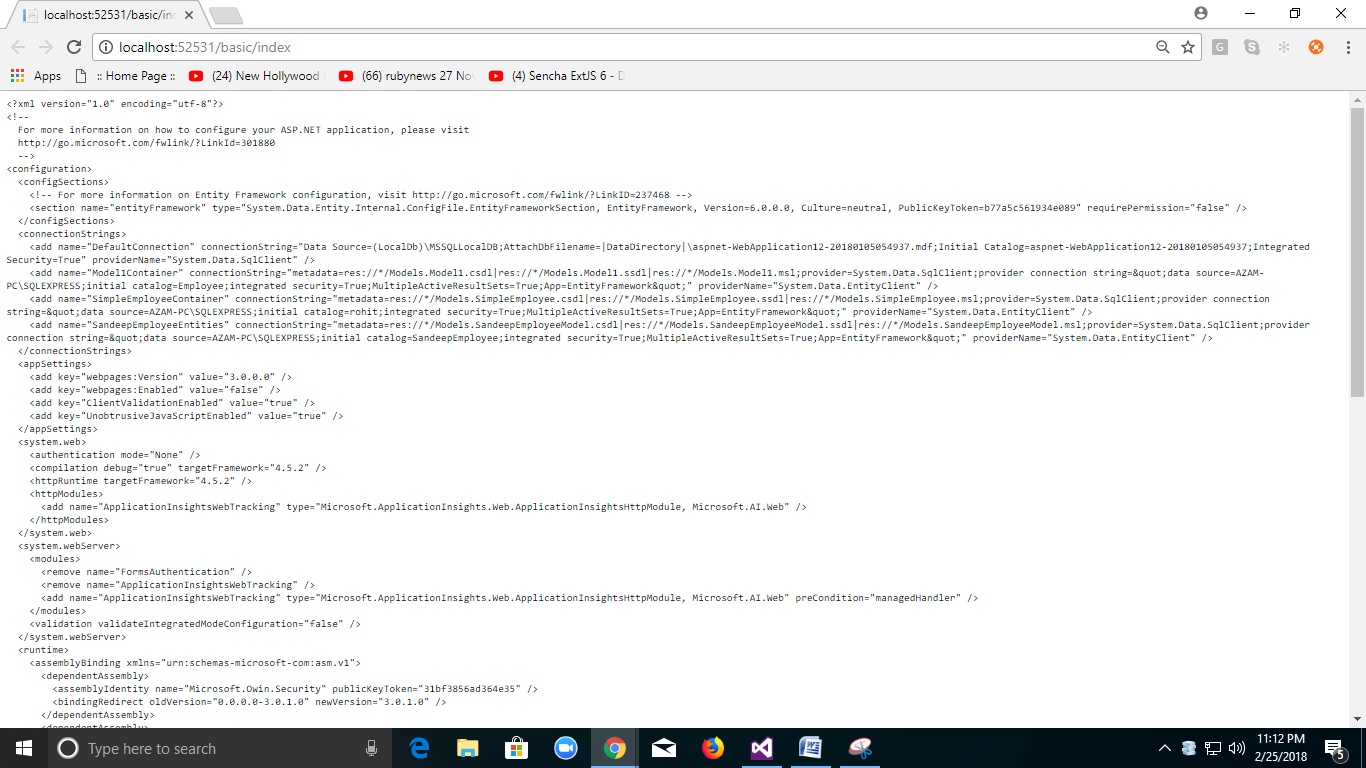
**Json Result**  
Json result is a significant Action Result in MVC. It will return simple text file format and key value pairs. If we call action method, using Ajax, it should return Json result.



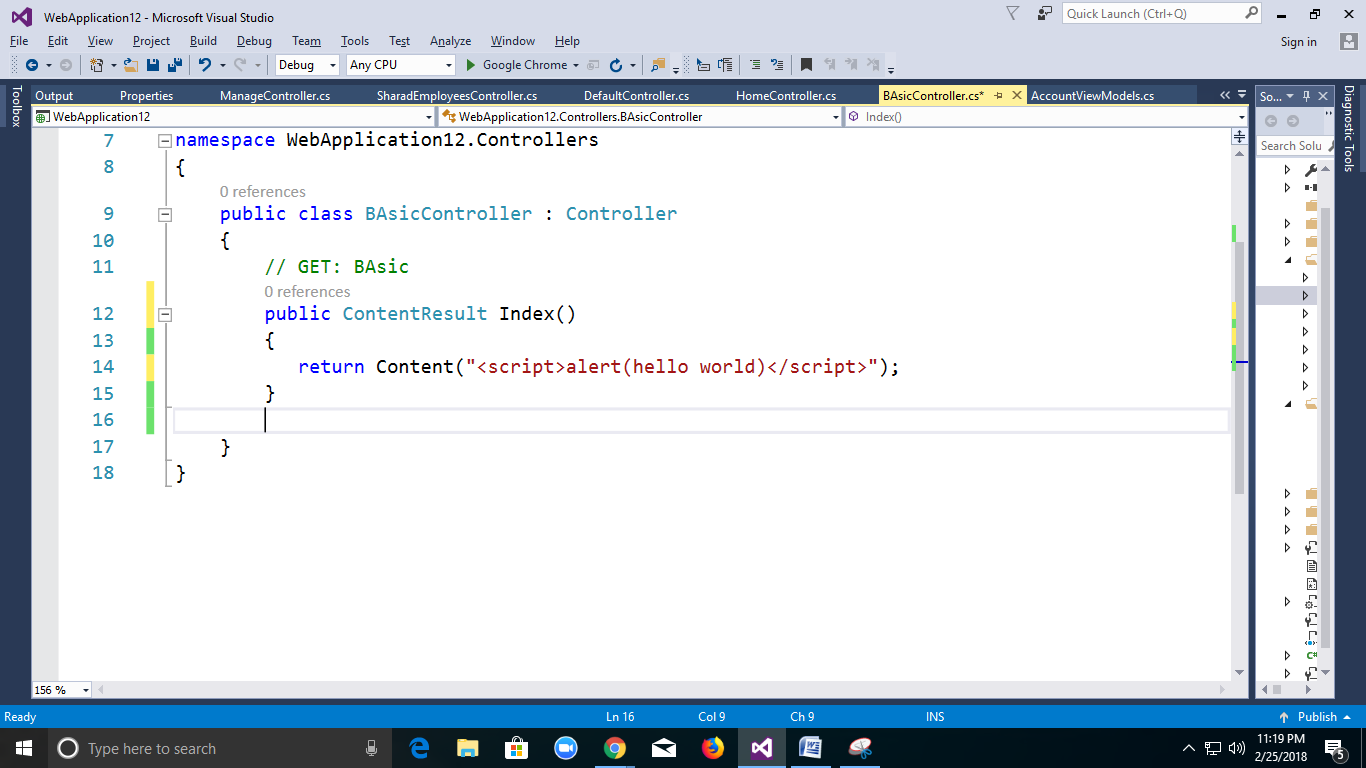


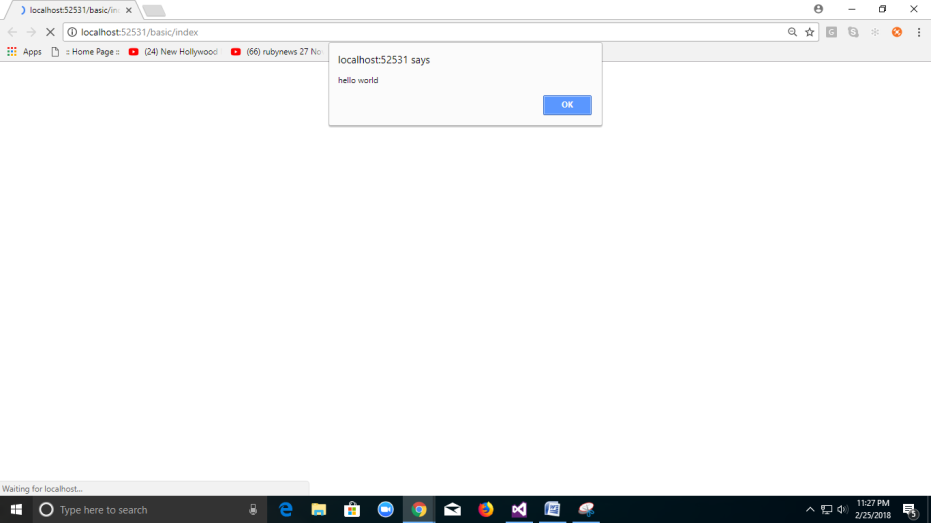
**File Result**File Result returns different file format view page when we implement file download concept in MVC using file result. Simple examples for file result are shown below:

****

****

**Content Result**Content result returns different content's format to view. MVC returns different format using content return like HTML format, Java Script format and any other format.

****

****

**Crud Operation Using Ado.net in MVC.**

**STEP1**

**Create Model Class Employee**

public class Employee

{

public int EmpId { get; set; }

public string EmpName { get; set; }

public int EmpSalary { get; set; }

}

**Displaying Employee Details:**

**Step2:**

using System.Data;

using System.Data.SqlClient;

**Add a method EmployeeDetail to Employee Class:**

SqlConnection con = new SqlConnection("Data Source=LocalHost\\SQLEXPRESS;Initial Catalog=Employee; Integrated Security=true;");

public List<Employee> EmployeeDetail()

{

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

SqlCommand cmd = new SqlCommand("sp\_getprathimaEmployeeDetails", con);

cmd.CommandType = CommandType.StoredProcedure;

con.Open();

SqlDataAdapter da = new SqlDataAdapter(cmd);

DataTable dt = new DataTable();

da.Fill(dt);

foreach (DataRow dr in dt.Rows)

{

objlist.Add(new Employee {

EmpId=Convert.ToInt32(dr["empid"]),

EmpName =(dr["empname"].ToString()),

EmpSalary = Convert.ToInt32(dr["empsalary"]),

}

);

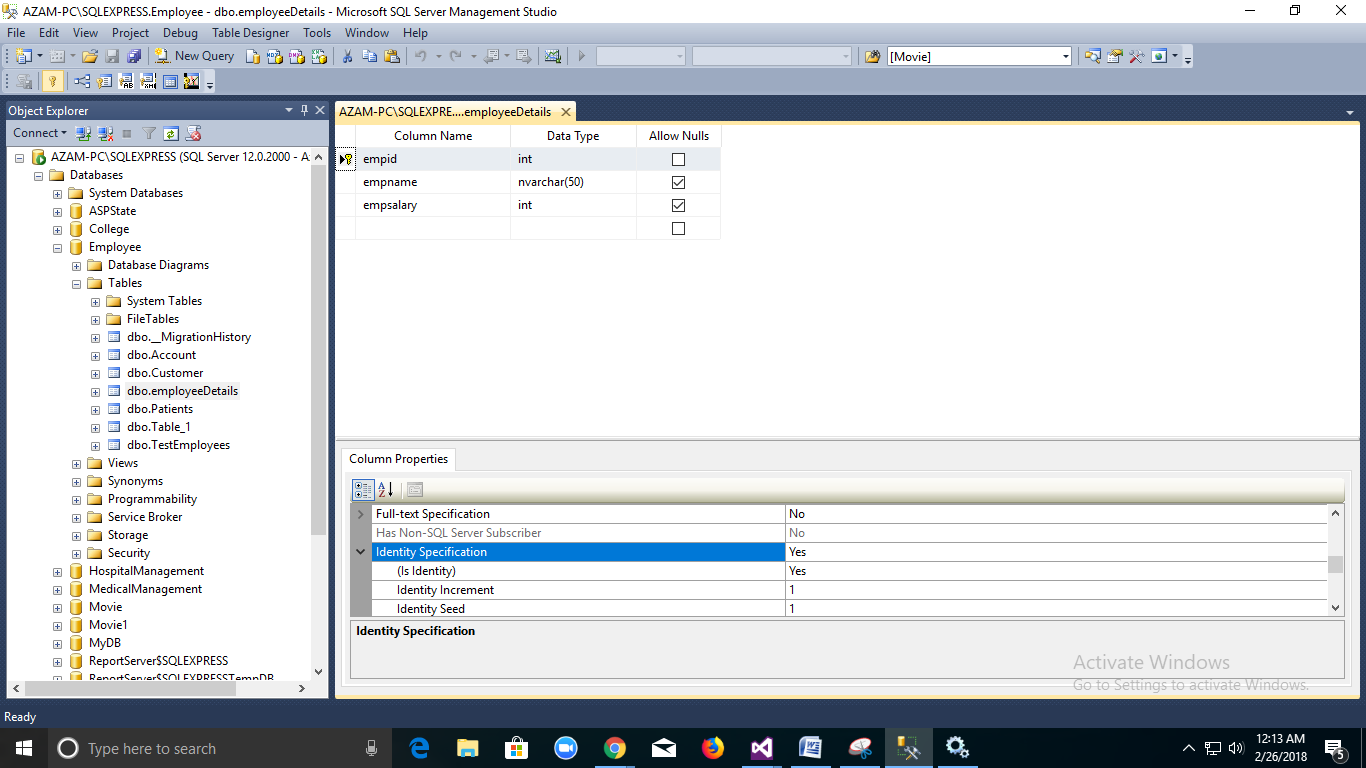
}

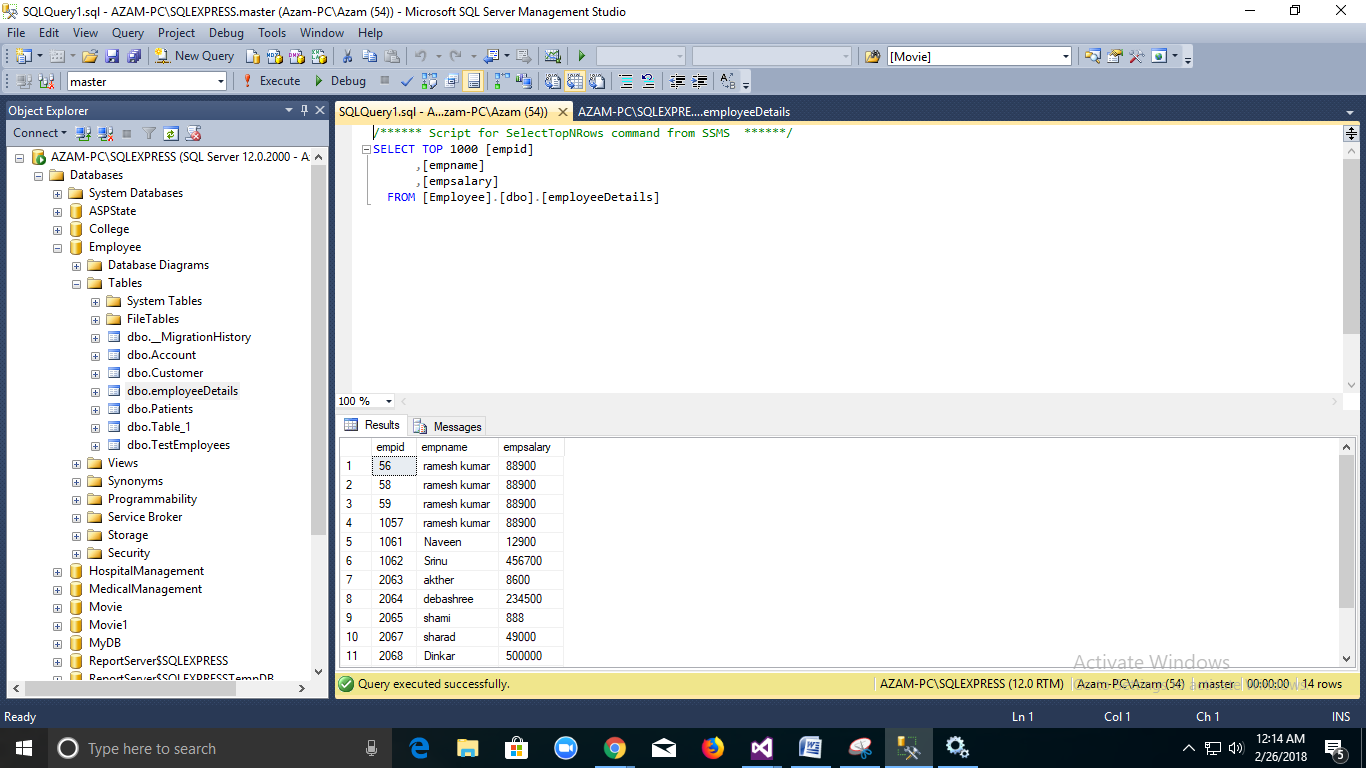
return objlist;

}

**Step3:**

**Create Table:**

****

****

**Create Procedure:**

CREATE PROCEDURE [dbo].[sp\_getprathimaEmployeeDetails]

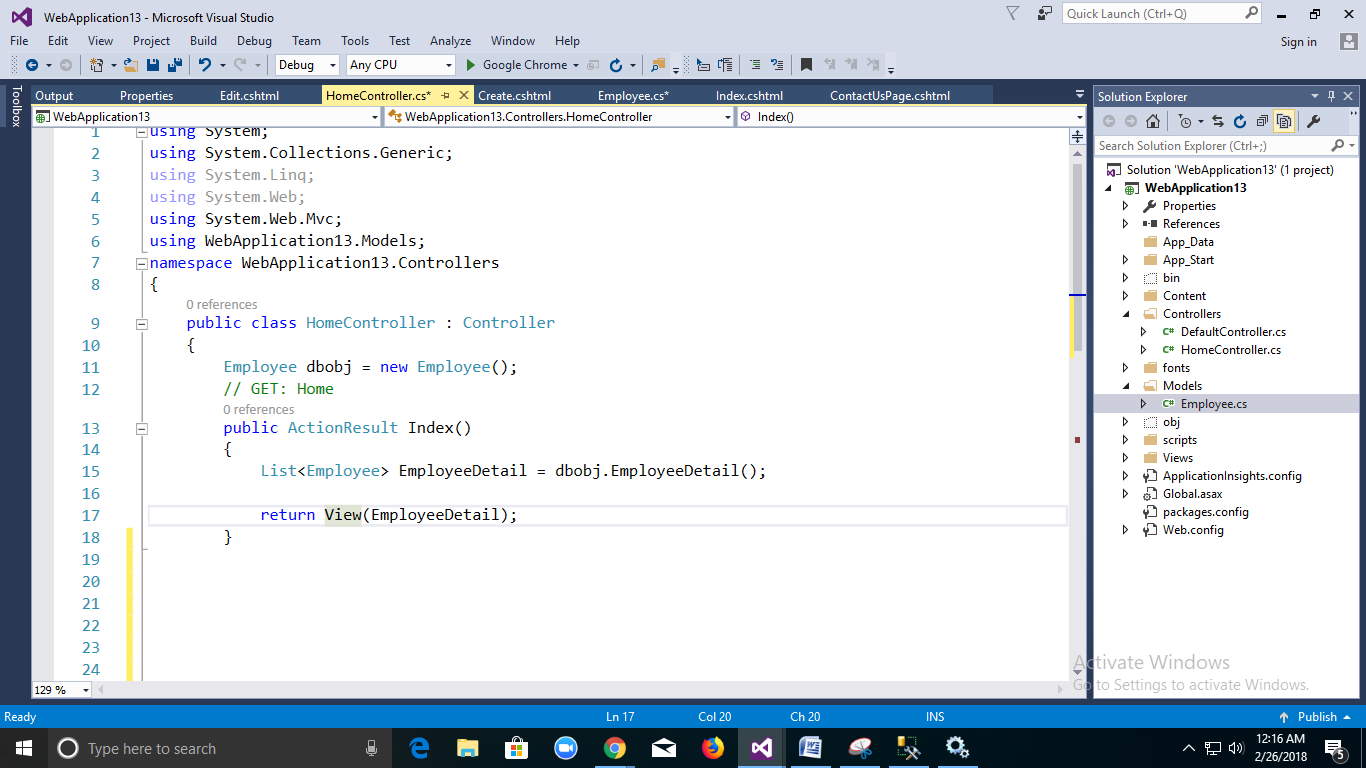
As

BEGIN

SELECT \* FROM Employeedetails

END

**Add Controller:**

****

public class HomeController : Controller

{

Employee dbobj = new Employee();

// GET: Home

public ActionResult Index()

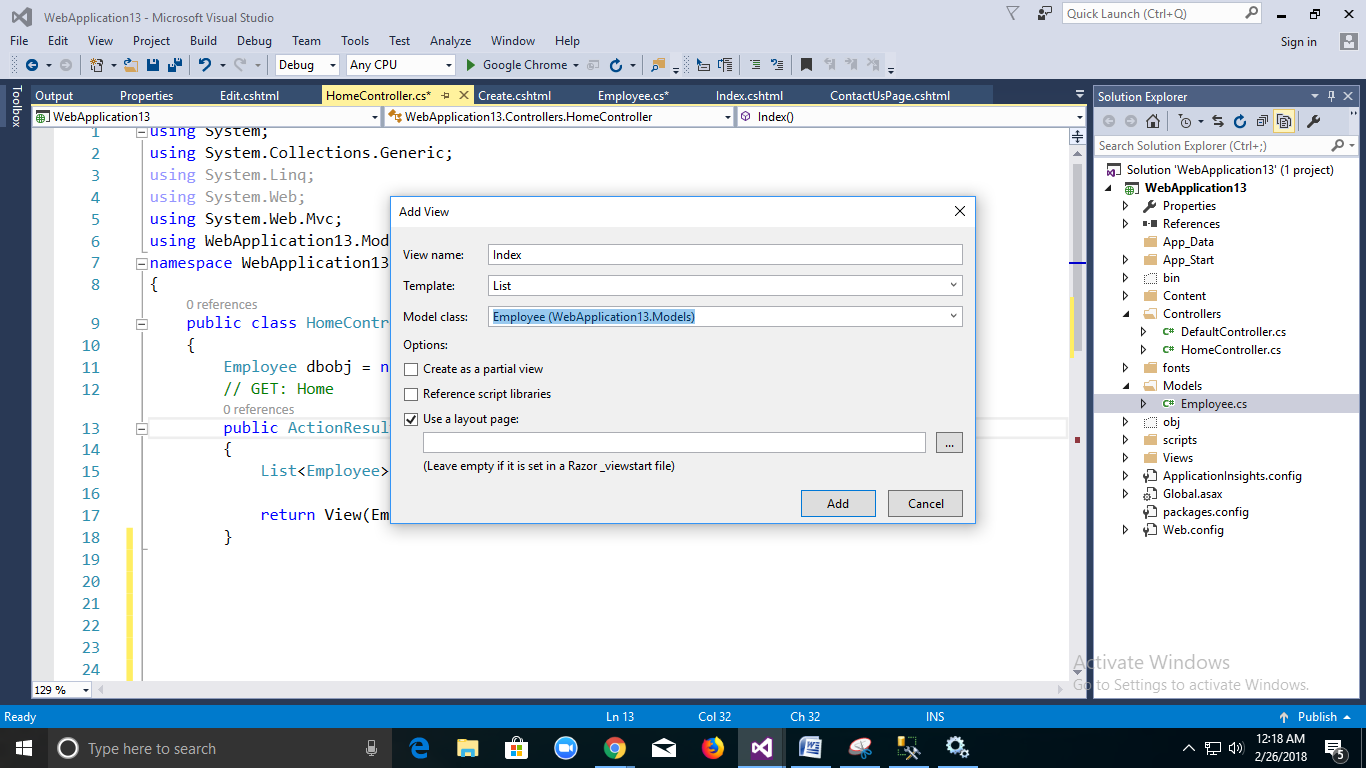
{

List<Employee> EmployeeDetail = dbobj.EmployeeDetail();

return View(EmployeeDetail);

}

Add View:



@model IEnumerable<WebApplication13.Models.Employee>

@{

ViewBag.Title = "Index";

}

<h2>Index</h2>

<p>

@Html.ActionLink("Create Employee Details","Create")

</p>

<table class="table">

<tr>

<th>

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

</th>

<th>

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

</th>

<th>

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

</th>

<th></th>

</tr>

@foreach (var item in Model) {

<tr>

<td>

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

</td>

<td>

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

</td>

<td>

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

</td>

<td>

@Html.ActionLink("Edit", "Edit", new { id=item.EmpId }) |

@Html.ActionLink("Details", "Details", new { id=item.EmpId }) |

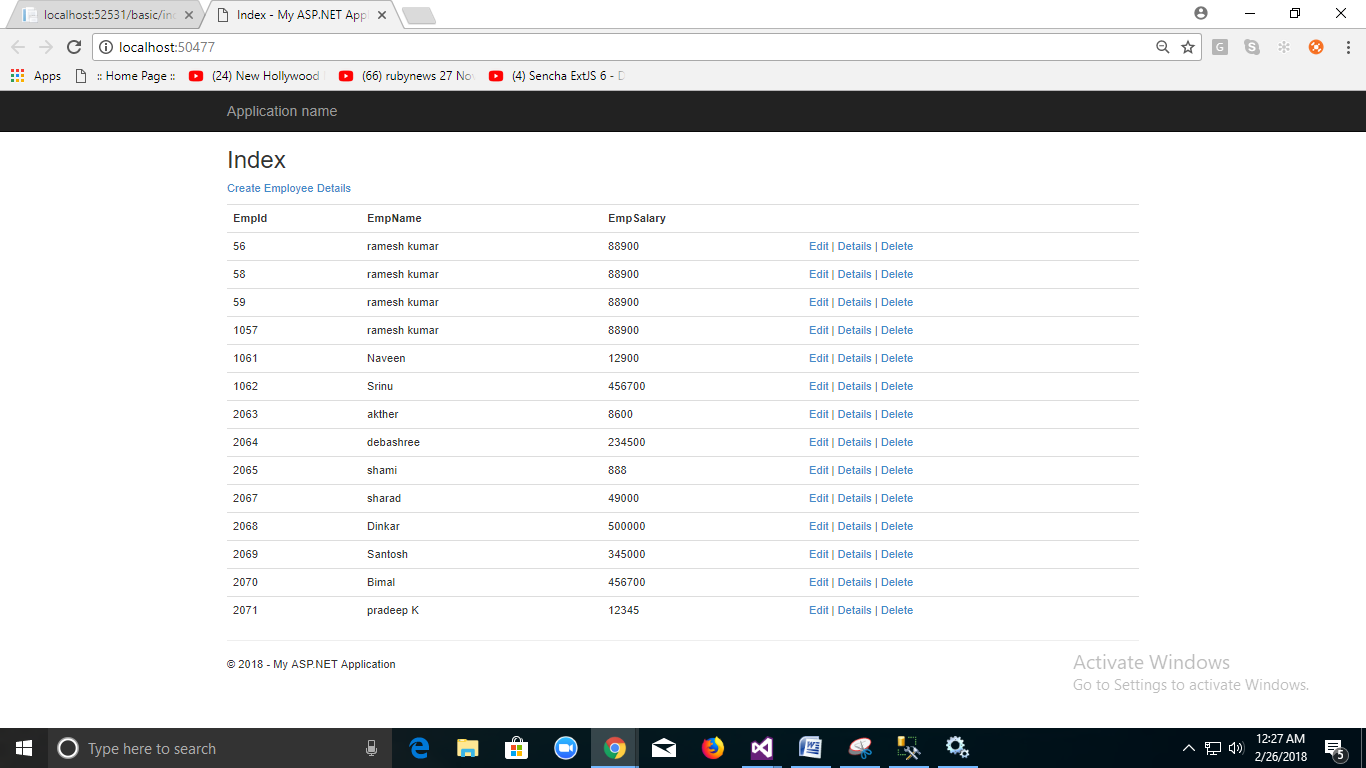
@Html.ActionLink("Delete", "Delete", new id=item.EmpId })

</td>

</tr>

}

</table>



Insert Operation Using Ado.net:

Step1:

CREATE PROCEDURE [dbo].[spr\_InsertEmployeeDetails]

@empid int,

@empname nvarchar(50),

@empsalary int

AS

BEGIN

insert into employeeDetails(empid,empname,empsalary)values(@empid,@empname,@empsalary)

END

**ADD EmployeeDetail Class** **insertEmployeeDetail Method**

public int insertEmployeeDetail(string EmpName,int Empsalary)

{

SqlCommand cmd = new SqlCommand("spr\_InsertprathimaEmployeeDetails", con);

con.Open();

cmd.CommandType = CommandType.StoredProcedure;

cmd.Parameters.AddWithValue("@empname", EmpName);

cmd.Parameters.AddWithValue("@empsalary", EmpSalary);

object i = cmd.ExecuteScalar();

int result = Convert.ToInt32(i);

return result;

}

Create Method in HomeController

[HttpGet]

public ActionResult Create()

{

return View();

}

For Post Method you can use Two operation Either FormCollection Class or Employee Class

[HttpPost]

public ActionResult Create(FormCollection formKey)

{

string empName = formKey["Name"];

int empSalary = Convert.ToInt32(formKey["Salary"]);

int result = dbobj.insertEmployeeDetail(empName,empSalary);

if (result > 0)

{

return RedirectToAction("Index");

}

else

{

return View();

}

}

OR

[HttpPost]

public ActionResult Create(Employee Empobj)

{

string empName = Empobj.EmpName;

int empSalary = Convert.ToInt32(Empobj.EmpSalary);

int result = dbobj.insertEmployeeDetail(empName,empSalary);

if (result > 0)

{

return RedirectToAction("Index");

}

else

{

return View();

}

}

FormCollection is a Class Which will catch all form values By Html Contol Name Property.

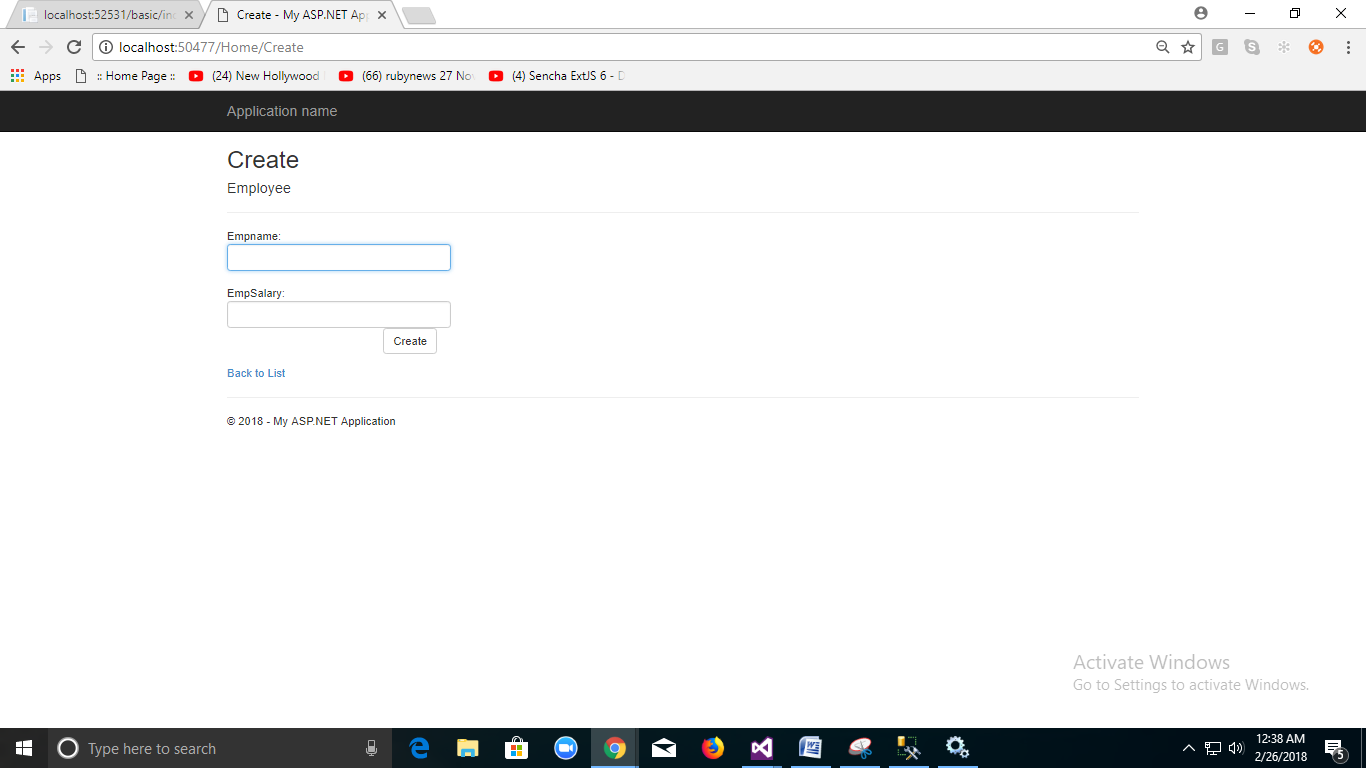
Ex: <input type=”text” name=”EmpName” />

Here We Have To Create Two Create Method ie..

Create For HttpGet and HttpPost

HttpGet for loading Html Form

HttpPost Create Method will be Executed When User Click on Submit Button



**CREATE View Code:**

@model WebApplication13.Models.Employee

@{

ViewBag.Title = "Create";

}

<h2>Create</h2>

@using (Html.BeginForm()) //<form method="post" action="Home/Create"></form>

{

@Html.AntiForgeryToken()

<div class="form-horizontal">

<h4>Employee</h4>

<hr />

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

Empname:<input type="text" id="txtName" name="Name" class="form-control"/><br />

EmpSalary:<input type="text" id="txtsalary" name="Salary" class="form-control" />

<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>

Edit Operation Using Ado.net:

Before Editing Operation Show All Values in control

Add Procedure:

CREATE PROCEDURE [dbo].[spr\_getEmployeeDetailsbyId]

@empid int

AS

BEGIN

SELECT \* FROM employeeDetails where empid=@empid

END

**Add EmployeeDetail Class getEmployeeDetailById Method**.

public Employee getEmployeeDetailById(int? id)

{

Employee obj = new Employee();

SqlCommand cmd = new SqlCommand("spr\_getEmployeeDetailsbyId", con);

cmd.CommandType = CommandType.StoredProcedure;

cmd.Parameters.AddWithValue("@empid",id);

con.Open();

SqlDataAdapter da = new SqlDataAdapter(cmd);

DataTable dt = new DataTable();

da.Fill(dt);

if (dt.Rows.Count>0)

{

obj.EmpId = Convert.ToInt32(dt.Rows[0]["empid"]);

obj.EmpName = dt.Rows[0]["empname"].ToString();

obj.EmpSalary = Convert.ToInt32(dt.Rows[0]["empsalary"]);

}

con.Close();

return obj;

}

[HttpGet]

public ActionResult Edit(int? id)

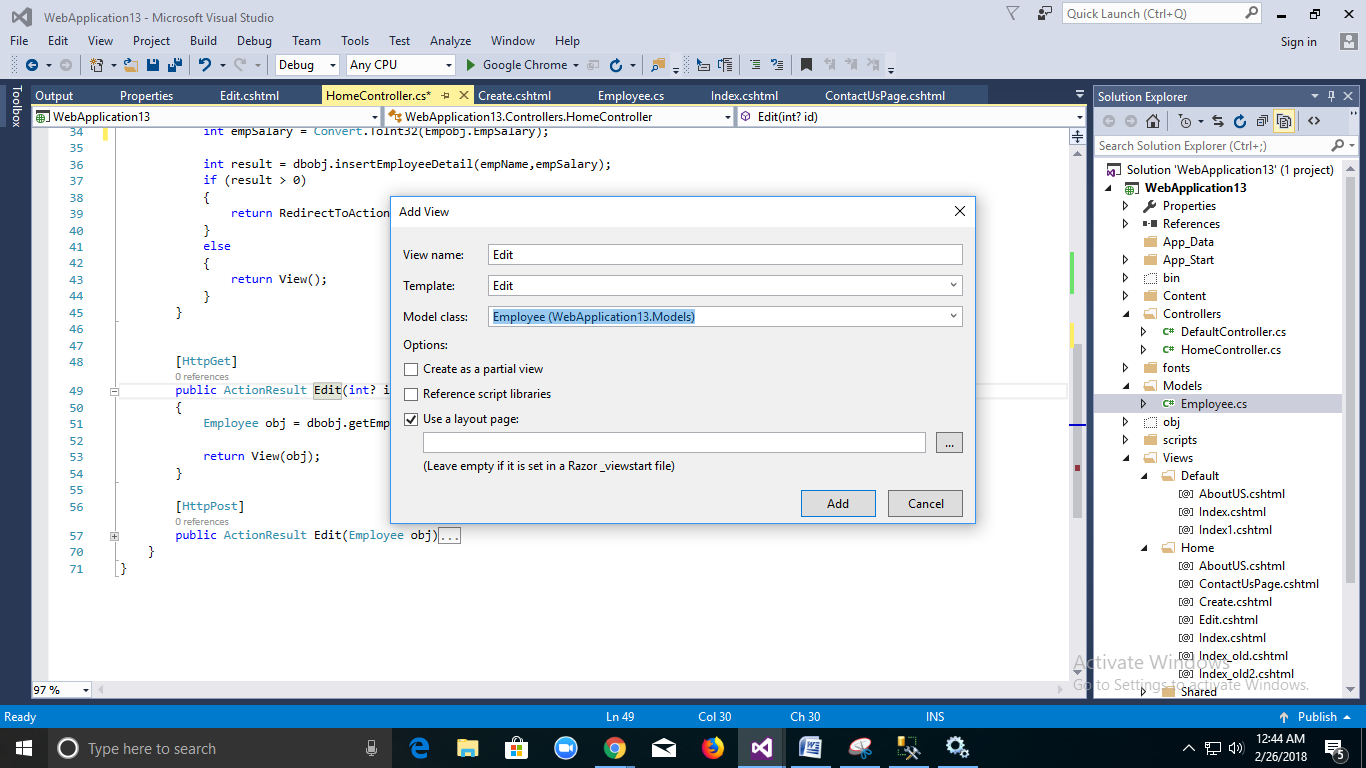
{

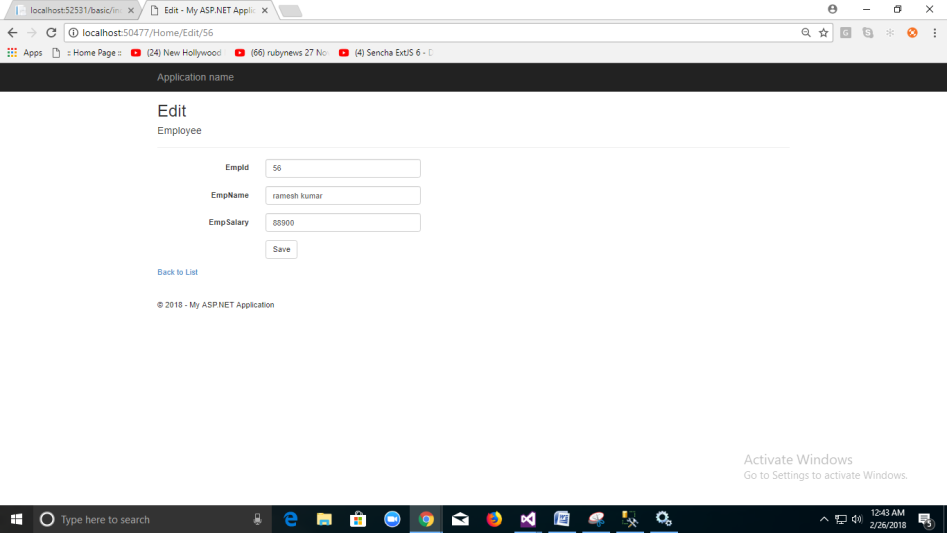
Employee obj = dbobj.getEmployeeDetailById(id);

return View(obj);

}

AddView:





CREATE PROCEDURE [dbo].[spr\_updateEmployeeDetails]

@empid int,

@empname nvarchar(50),

@empsalary int

AS

BEGIN

update employeeDetails set empname=@empname,empsalary=@empsalary where empid=@empid

END

**Add Employee Class updateEmployeeDetails Method.**

public int updateEmployeeDetails(Employee obj)

{

Employee objupdate = new Employee();

SqlCommand cmd = new SqlCommand("spr\_updateEmployeeDetails", con);

cmd.CommandType = CommandType.StoredProcedure;

cmd.Parameters.AddWithValue("@empid", obj.EmpId);

cmd.Parameters.AddWithValue("@empname", obj.EmpName);

cmd.Parameters.AddWithValue("@empsalary", obj.EmpSalary);

con.Open();

int i = cmd.ExecuteNonQuery();

return i;

}

[HttpPost]

public ActionResult Edit(Employee obj)

{

int result = dbobj.updateEmployeeDetails(obj);

if (result > 0)

{

return RedirectToAction("Index");

}

else {

return View(obj);

}

}

}

Add View Code:

@model WebApplication13.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" })

<div class="form-group">

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

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

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

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

</div>

</div>

<div class="form-group">

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

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

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

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

</div>

</div>

<div class="form-group">

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

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

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

@Html.ValidationMessageFor(model => model.EmpSalary, "", 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>

**Asp.net with Entity Framework Approach.**

Entity Framework is an open-source [ORM framework](http://en.wikipedia.org/wiki/Object-relational_mapping) for .NET applications supported by Microsoft. It enables developers to work with data using objects of domain specific classes without focusing on the underlying database tables and columns where this data is stored. With the Entity Framework, developers can work at a higher level of abstraction when they deal with data, and can create and maintain data-oriented applications with less code compared with traditional applications.

Official Definition: “Entity Framework is an object-relational mapper (O/RM) that enables .NET developers to work with a database using .NET objects. It eliminates the need for most of the data-access code that developers usually need to write.”

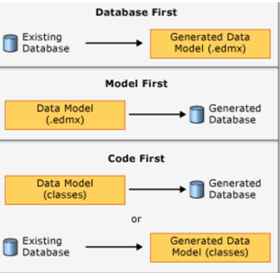
Entity Framework Features:

* **Cross-platform:** EF Core is a cross-platform framework which can run on Windows, Linux and Mac.
* **Modelling:** EF (Entity Framework) creates an EDM (Entity Data Model) based on POCO (Plain Old CLR Object) entities with get/set properties of different data types. It uses this model when querying or saving entity data to the underlying database.
* **Querying:** EF allows us to use LINQ queries (C#/VB.NET) to retrieve data from the underlying database. The database provider will translate this LINQ queries to the database-specific query language (e.g. SQL for a relational database). EF also allows us to execute raw SQL queries directly to the database.
* **Change Tracking:** EF keeps track of changes occurred to instances of your entities (Property values) which need to be submitted to the database.
* **Saving:** EF executes INSERT, UPDATE, and DELETE commands to the database based on the changes occurred to your entities when you call the SaveChanges() method. EF also provides the asynchronous SaveChangesAsync() method.
* **Concurrency:** EF uses Optimistic Concurrency by default to protect overwriting changes made by another user since data was fetched from the database.
* **Transactions:** EF performs automatic transaction management while querying or saving data. It also provides options to customize transaction management.
* **Caching:** EF includes first level of caching out of the box. So, repeated querying will return data from the cache instead of hitting the database.
* **Built-in Conventions:** EF follows conventions over the configuration programming pattern, and includes a set of default rules which automatically configure the EF model.
* **Configurations:** EF allows us to configure the EF model by using data annotation attributes or Fluent API to override default conventions.
* **Migrations:** EF provides a set of migration commands that can be executed on the NuGet Package Manager Console or the Command Line Interface to create or manage underlying database Schema.

| **EF Version** | **Release Year** | **.NET Framework** |
| --- | --- | --- |
| EF 6 | 2013 | .NET 4.0 & .NET 4.5, VS 2012 |
| EF 5 | 2012 | .NET 4.0, VS 2012 |
| EF 4.3 | 2011 | .NET 4.0, VS 2012 |
| EF 4.0 | 2010 | .NET 4.0, VS 2010 |
| EF 1.0 (or 3.5) | 2008 | .NET 3.5 SP1, VS 2008 |

## Entity Framework Development Approaches

As shown in the following diagram, there are three ways you can work with data models and databases in the Entity Framework: *Database First*, *Model First*, and *Code First*.



* **Database First**

If you already have a database, the Entity Framework designer built into Visual Studio can automatically generate a data model that consists of classes and properties that correspond to existing database objects such as tables and columns. The information about your database structure (*store schema*), your data model (*conceptual model*), and the mapping between them is stored in XML in an *.edmx* file. The Entity Framework designer provides a graphical UI that you can use to display and edit the .edmx file.

* **Model First**

If you don't have a database yet, you can begin by creating a model in an .edmx file by using the Entity Framework graphical designer in Visual Studio. When the model is finished, the Entity Framework designer can generate DDL (data definition language) statements to create the database. As in Database First, the .edmx file stores model and mapping information.

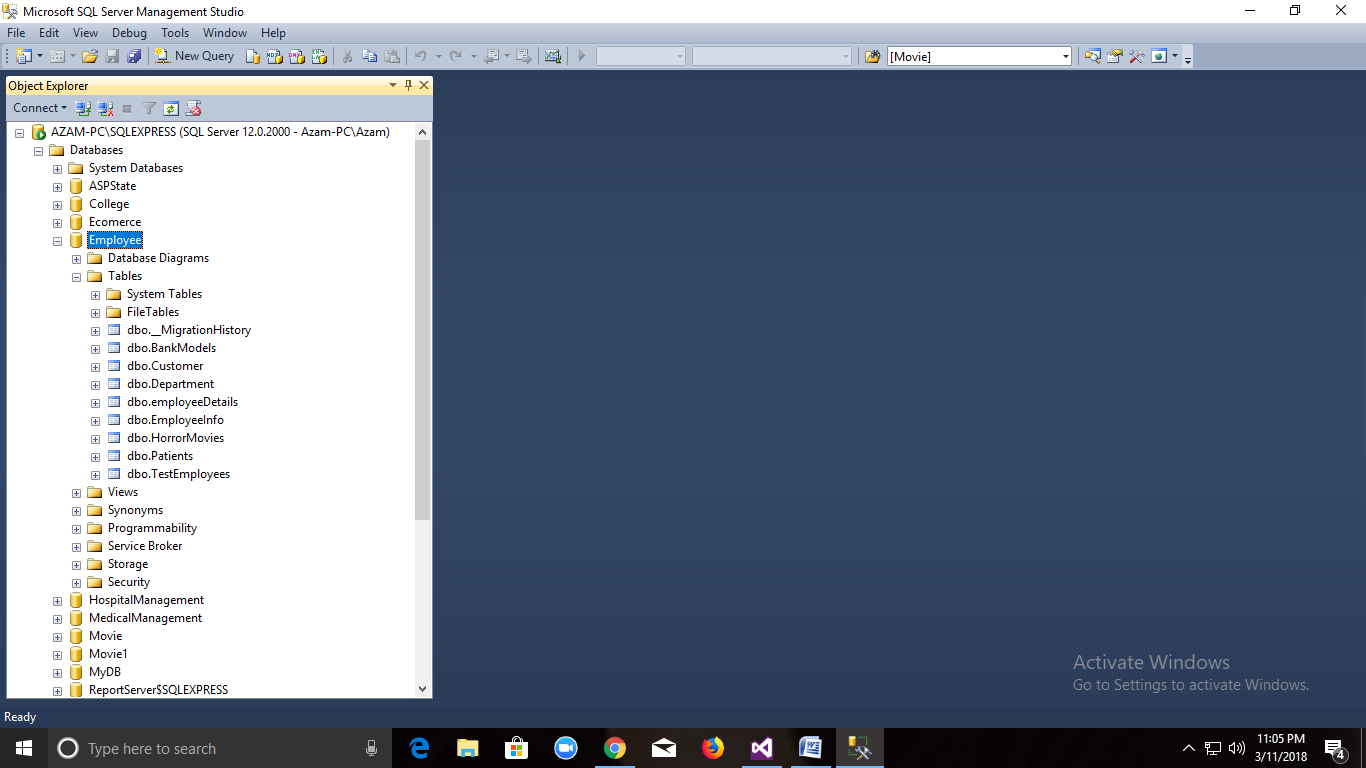
* **Code First**

Whether you have an existing database or not, you can use the Entity Framework without using the designer or an .edmx file. If you don't have a database, you can code your own classes and properties that correspond to tables and columns. If you do have a database, Entity Framework tools can generate the classes and properties that correspond to existing tables and columns. The mapping between the store schema and the conceptual model represented by your code is handled by convention and by a special mapping API. If you let Code First create the database, you can use Code First Migrations to automate the process of deploying the database to production. Migrations can also automate the deployment of database schema changes to production when your data model changes.

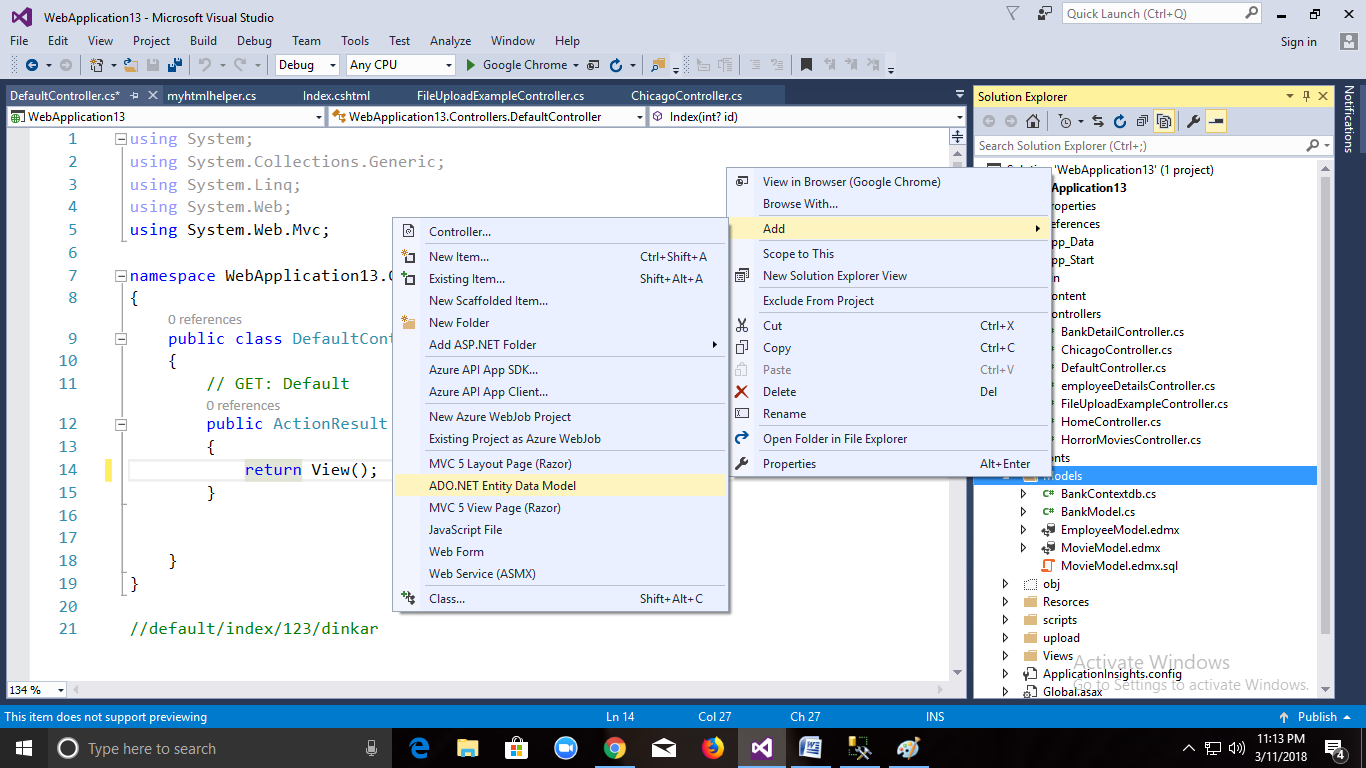
Choose Code First for new development unless you want to use a graphical designer to model database objects and relationships. The Entity Framework designer only works with Database First and Model First. Before you choose Database First or Model First, however, consider how you want to handle updates to the data model after you create the database, and how you want to deploy the database and deploy updates to it. Code First Migrations automates the process of implementing and deploying database schema changes that result from data model changes. The advantages of Code First Migrations might outweigh the advantages of the Entity Framework designer.

Let us see all this by using Enitity Framework Database Approach:

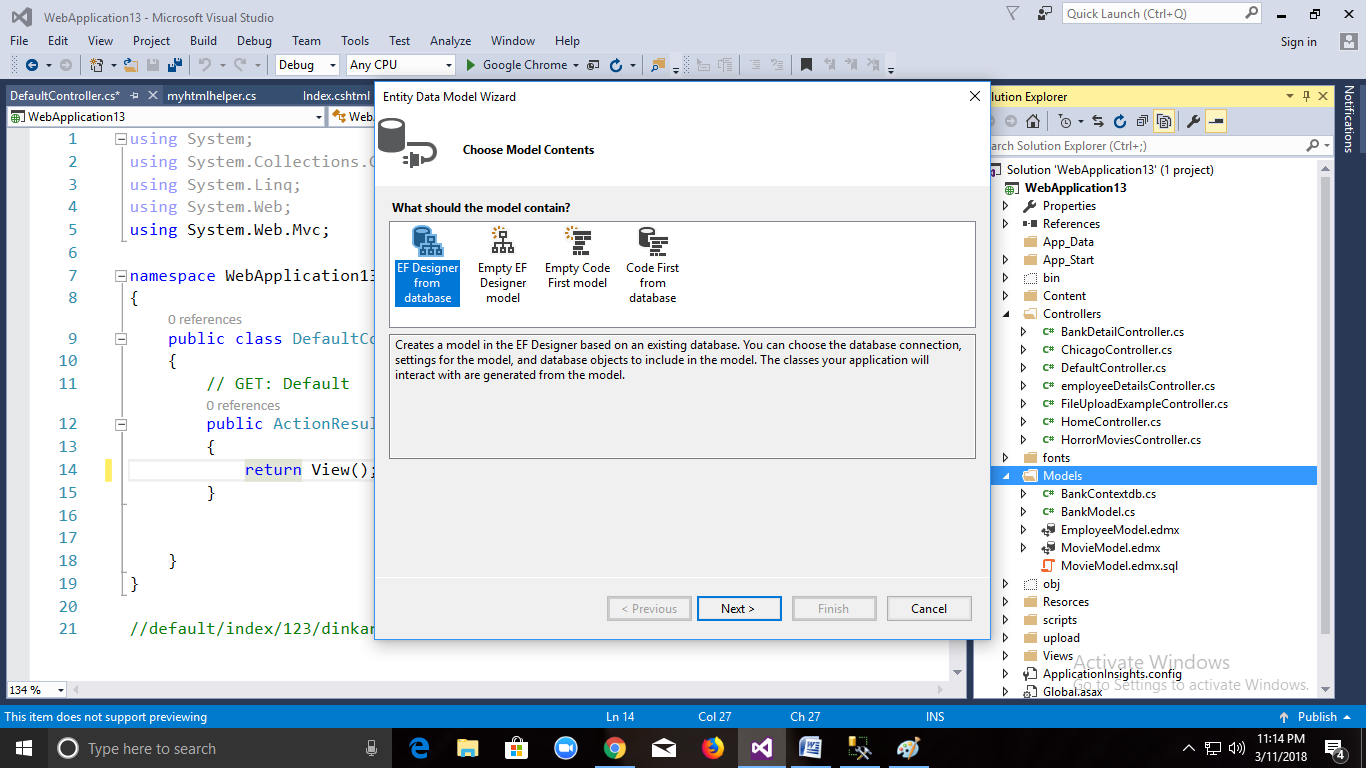
* For database approach using EnitityFramework you need to have Existing Database.

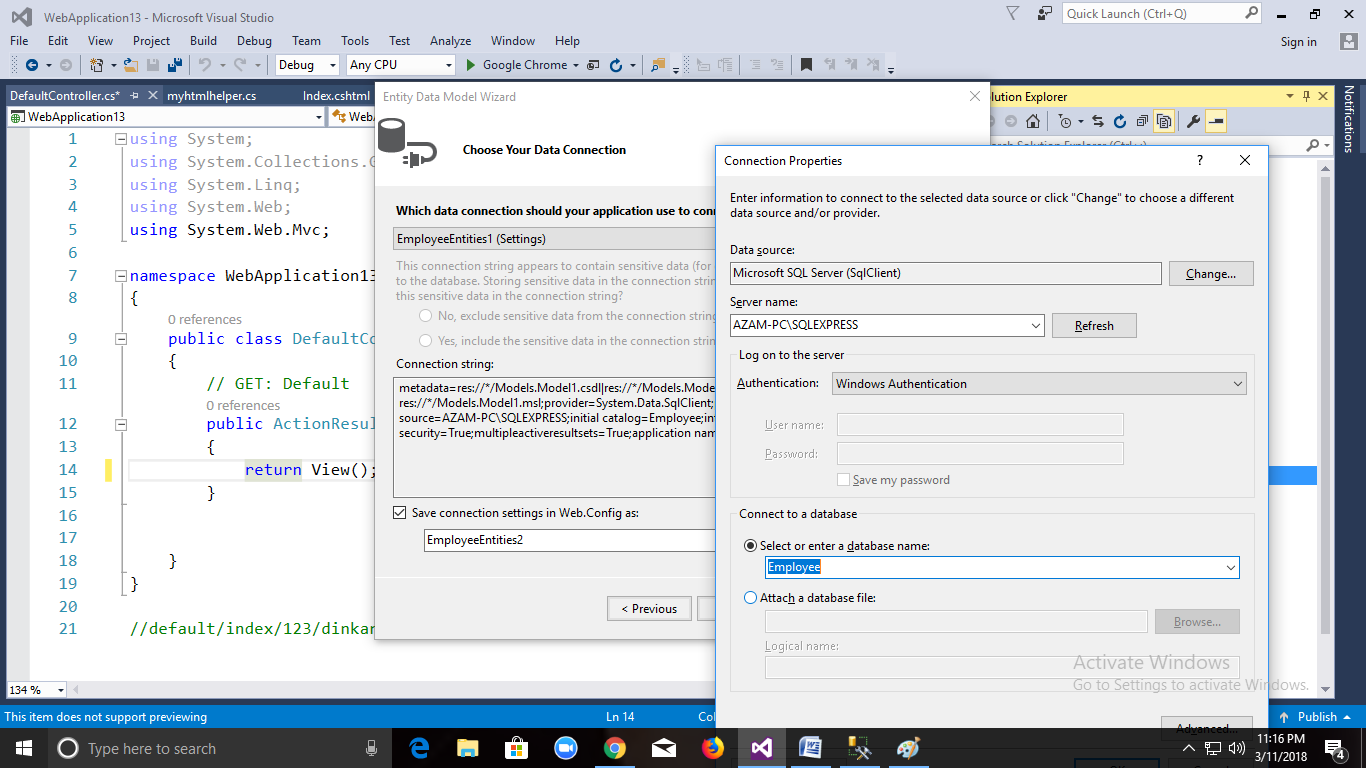


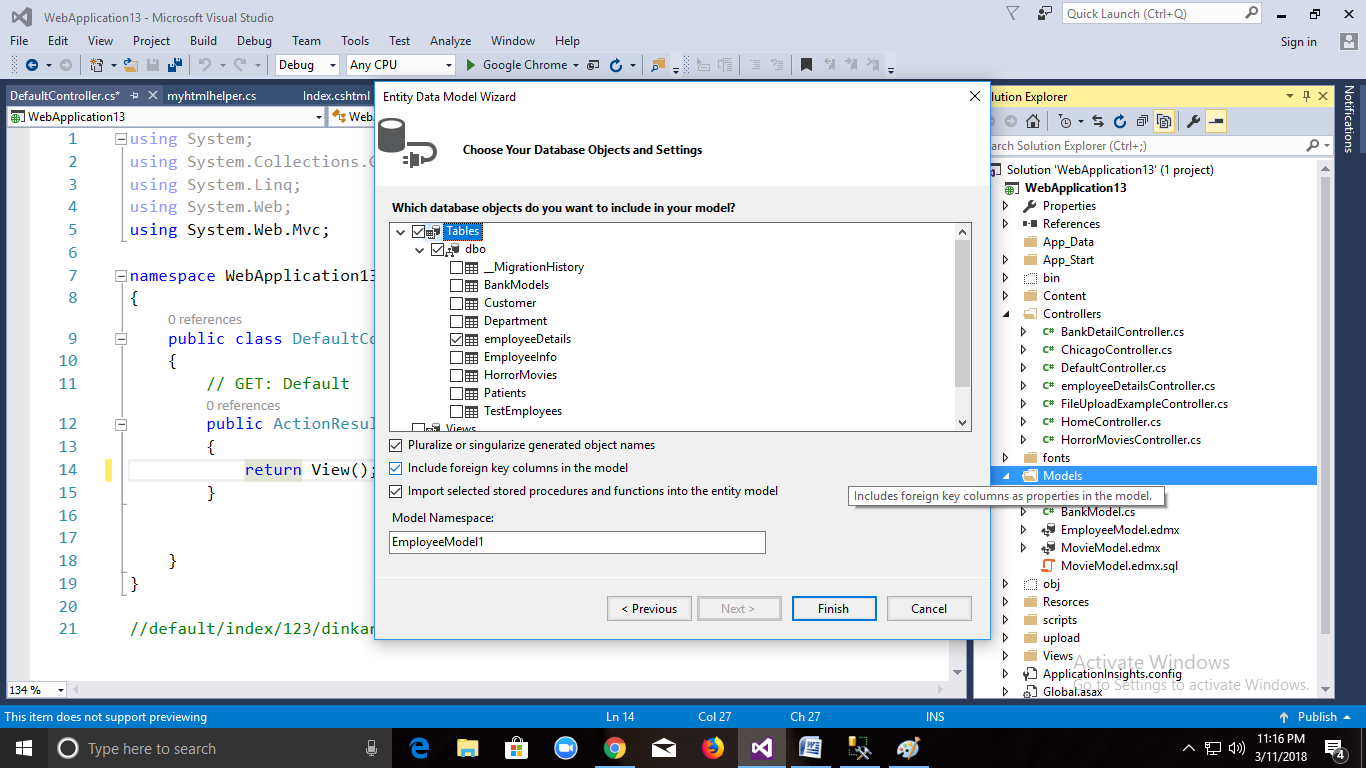
Employee Database is Approach given below.

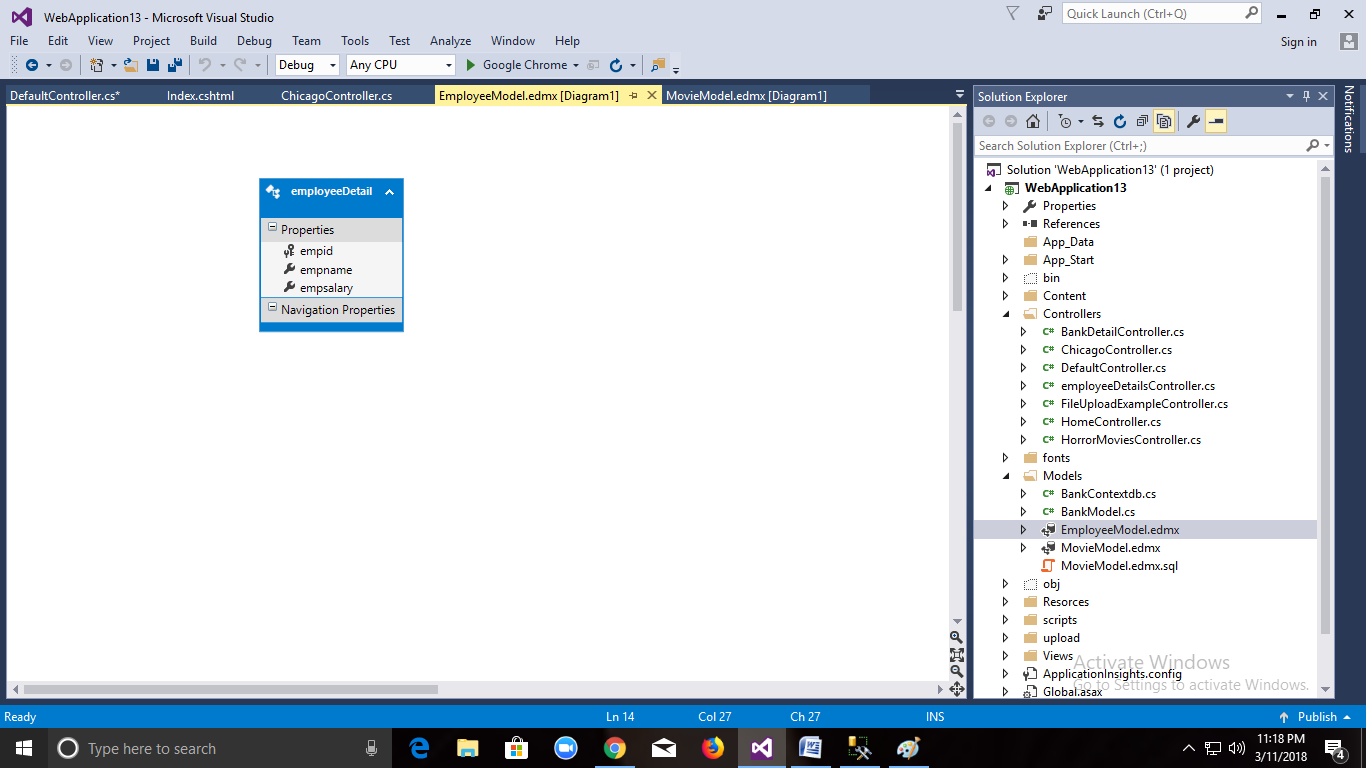
Step1: 

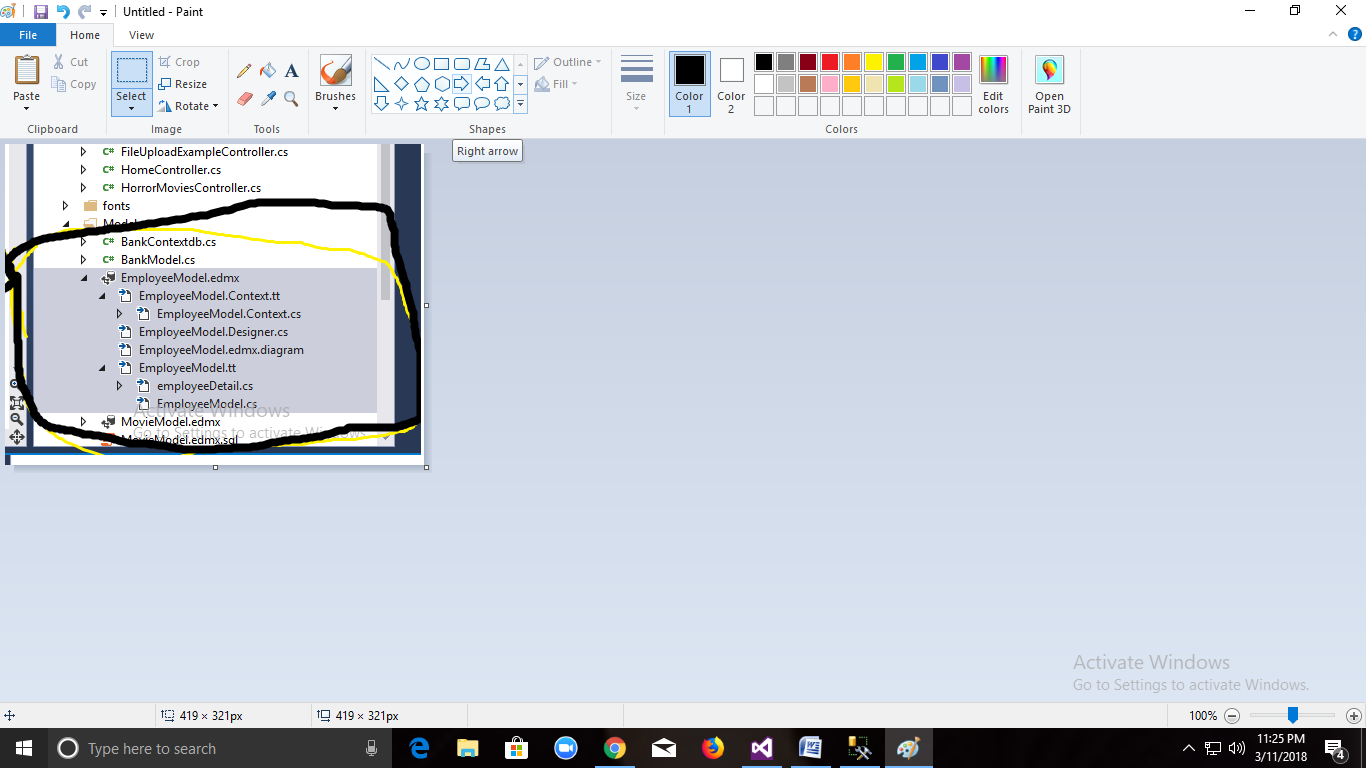












Entity Framework EmployeeFramework Edmx. Will be Created.

There two are the important Class

//------------------------------------------------------------------------------

// <auto-generated>

// This code was generated from a template.

//

// Manual changes to this file may cause unexpected behavior in your application.

// Manual changes to this file will be overwritten if the code is regenerated.

// </auto-generated>

//------------------------------------------------------------------------------

**1)EmployeeModel.Context.Cs**

namespace WebApplication13.Models

{

using System;

using System.Data.Entity;

using System.Data.Entity.Infrastructure;

public partial class EmployeeEntities : DbContext

{

public EmployeeEntities()

: base("name=EmployeeEntities")

{

}

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

throw new UnintentionalCodeFirstException();

}

public virtual DbSet<employeeDetail> employeeDetails { get; set; }

}

}

**2) employeeDetail Model Class**

namespace WebApplication13.Models

{

using System;

using System.Collections.Generic;

public partial class employeeDetail

{

public int empid { get; set; }

public string empname { get; set; }

public Nullable<int> empsalary { get; set; }

}

}

These Two class will be Created Automatically.

**class EmployeeEntities** will contain ConnectionString Information, to make any class Context Class we need to inherit with DbContext Class, and DbContext class will be present in System.Data.Enitity namesapace.

base("name=EmployeeEntities"): it used to invoke ConnectionString.In Web Config file we already defined Connectionstring with EmployeeEntities.

You will file your database ConnectionString in Web.config below

<connectionStrings>

<add name="EmployeeEntities1" connectionString="data source=AZAM-PC\SQLEXPRESS;initial catalog=Employee;integrated security=True;" providerName="System.Data.SqlClient" />

</ connectionStrings>

DataBase Name AuthentcationType(windows) SqlServer Name

public virtual DbSet<employeeDetail> employeeDetails { get; set; }

it is alias name through

which we can access table.

public partial class employeeDetail

{

public int empid { get; set; }

public string empname { get; set; }

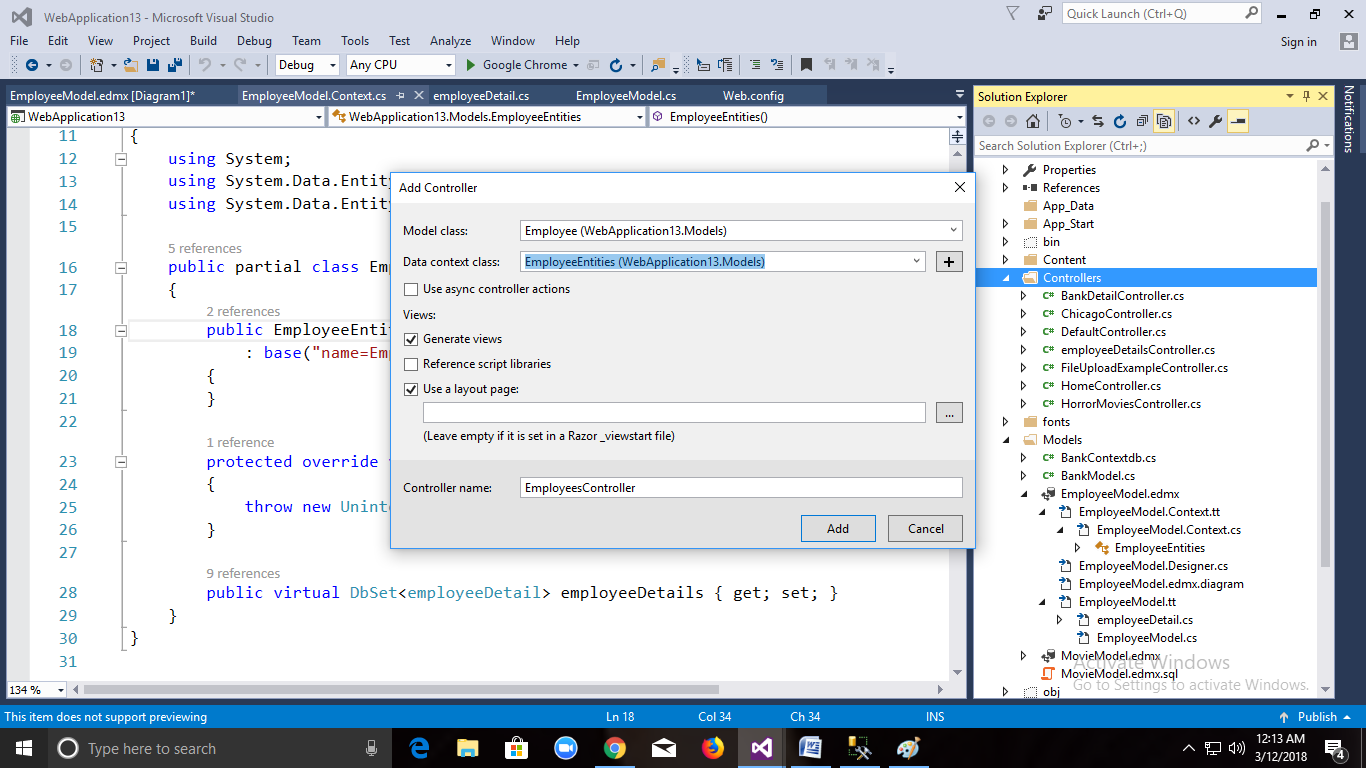
public Nullable<int> empsalary { get; set; }

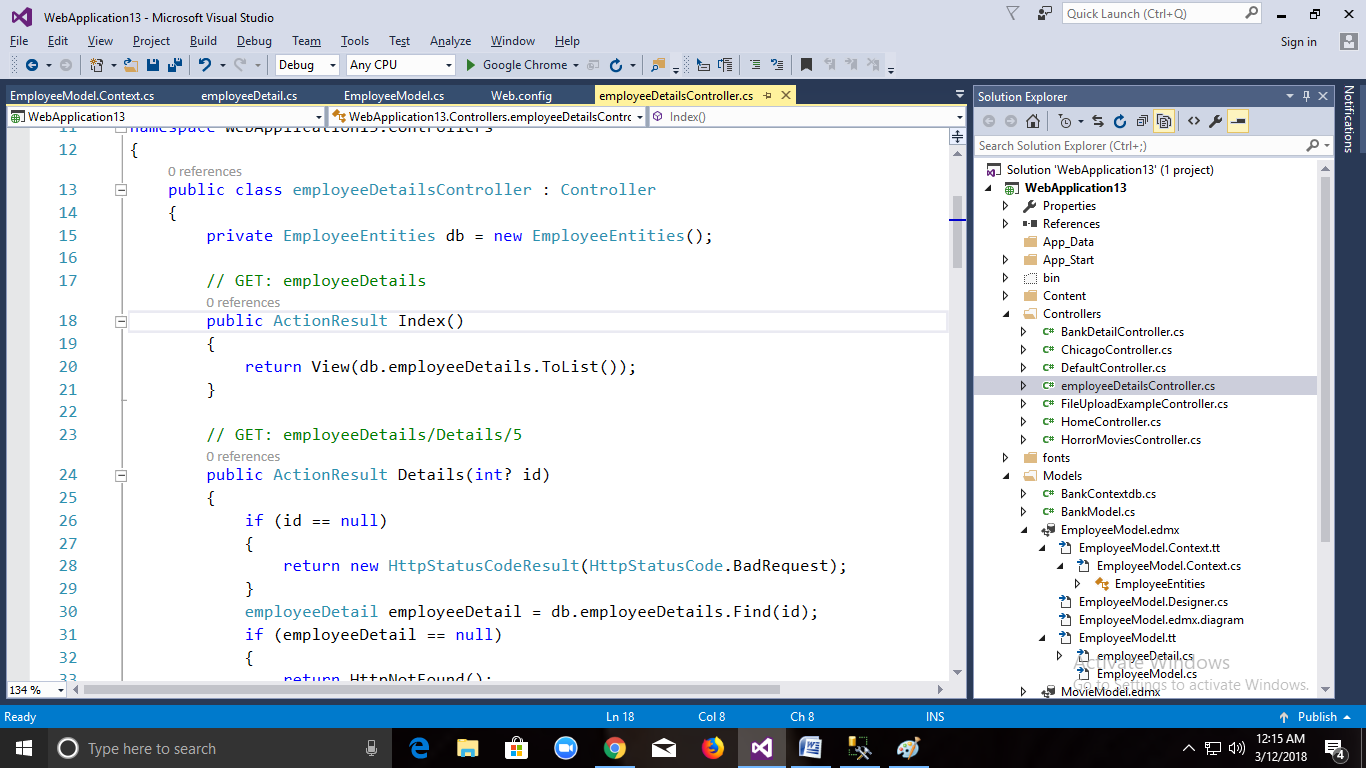
}

2) employeeDetail **is a second class which will be created automatically will contain the property same as table which is there in sql server.**

**So till Now what we have done is we just Created EntityFramework Now we will add Controller.**

****

****



EmployeeDetailsController with Predefined code for crud operation and View Will be Created (insert,delete,update)

public class employeeDetailsController : Controller

{

private EmployeeEntities db = new EmployeeEntities();

// GET: employeeDetails

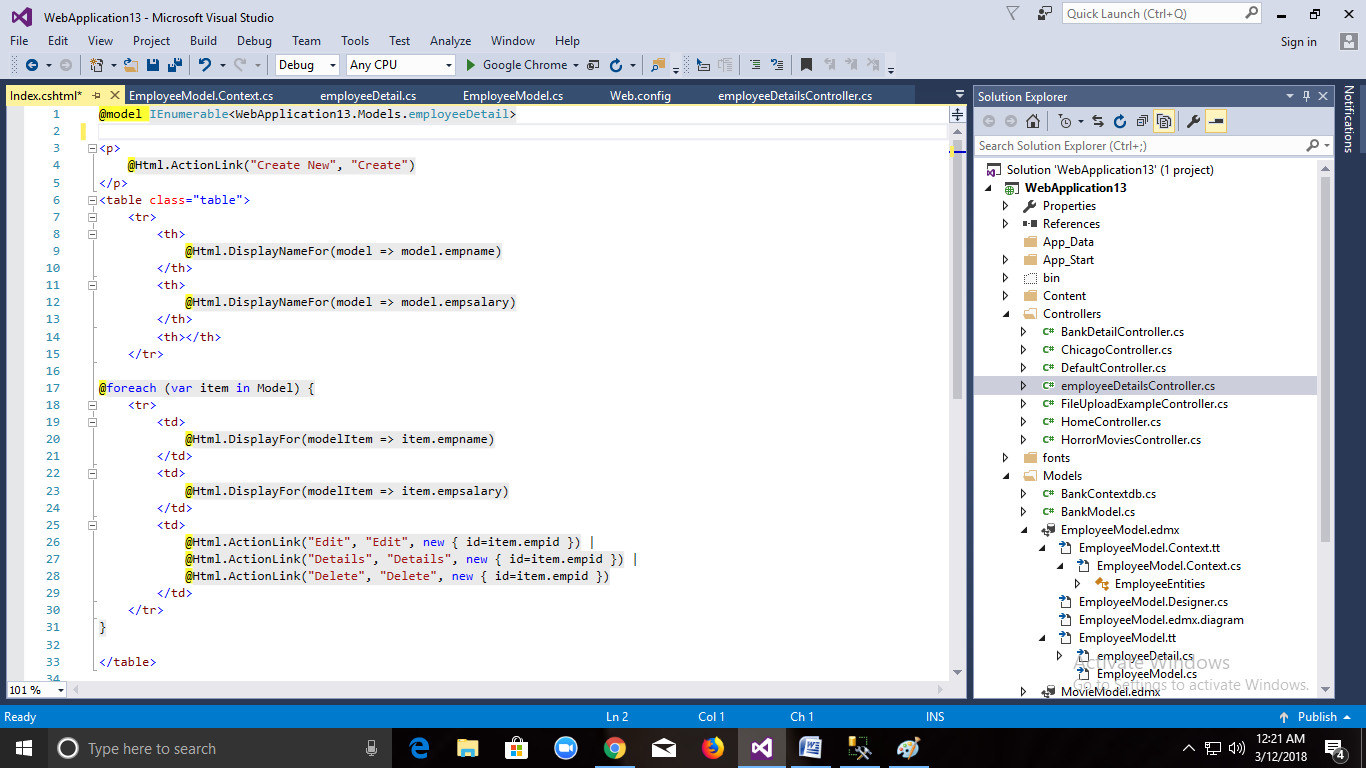
public ActionResult Index() this code will reterive data in form list

{

return View(db.employeeDetails.ToList());Tolist will Convert data of Table to List

}

Showing all Data.



// GET: employeeDetails/Details/5

public ActionResult Details(int? id)(It is used to display Single Detail)

{

if (id == null)

{

return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

}

employeeDetail employeeDetail = db.employeeDetails.Find(id);

if (employeeDetail == null)

{

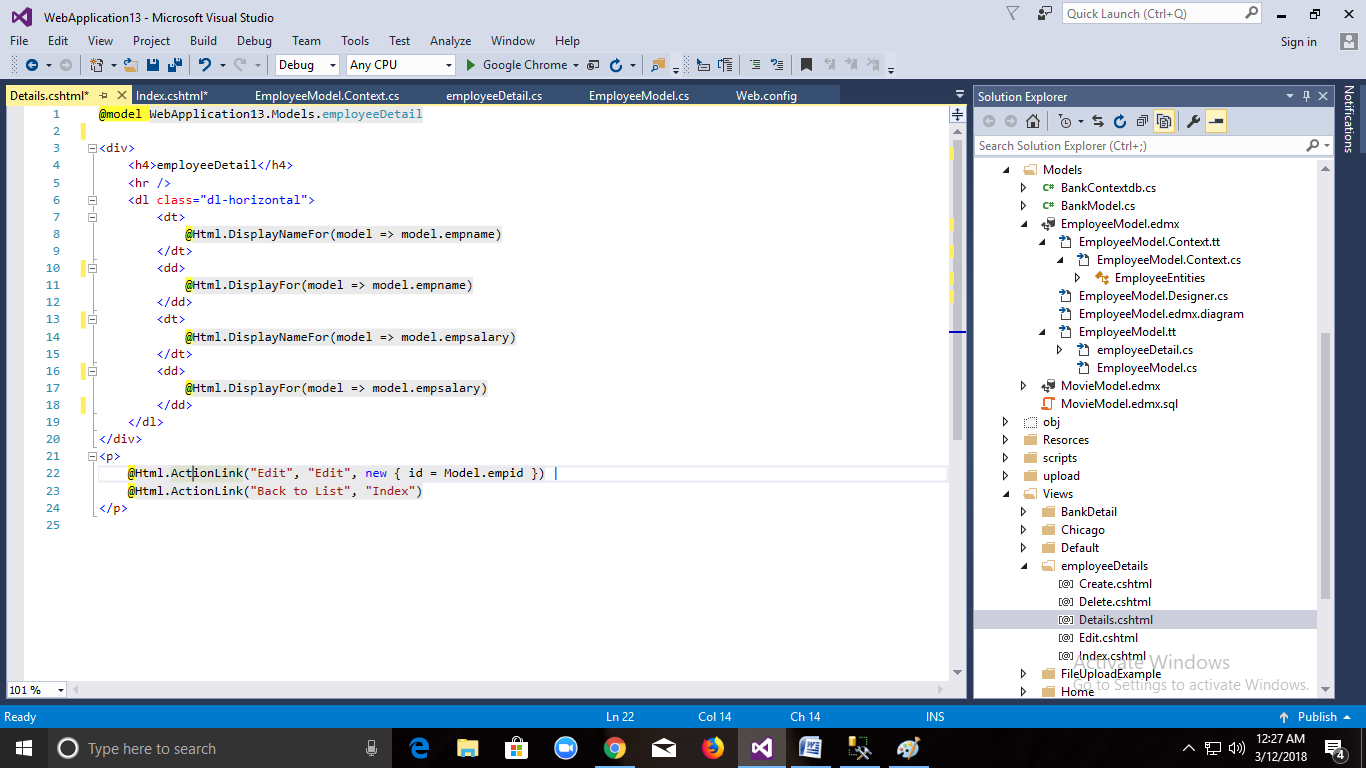
return HttpNotFound();

}

return View(employeeDetail);

}

View:



// GET: employeeDetails/Create

public ActionResult Create()//this Method Load Create Page

{

return View();

}

// POST: employeeDetails/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 = "empid,empname,empsalary")] employeeDetail employeeDetail) //When you Click button total form will be post and get values in employeeDetail variable.

{

employeeDetail.Empid=1;

employeeDetail.EmpName=”xyz”;

employeeDetail.Salary=1234;

if (ModelState.IsValid)

{

db.employeeDetails.Add(employeeDetail); employeeDetail add to employeeDetails table

db.SaveChanges();//permenantely Saving Data.

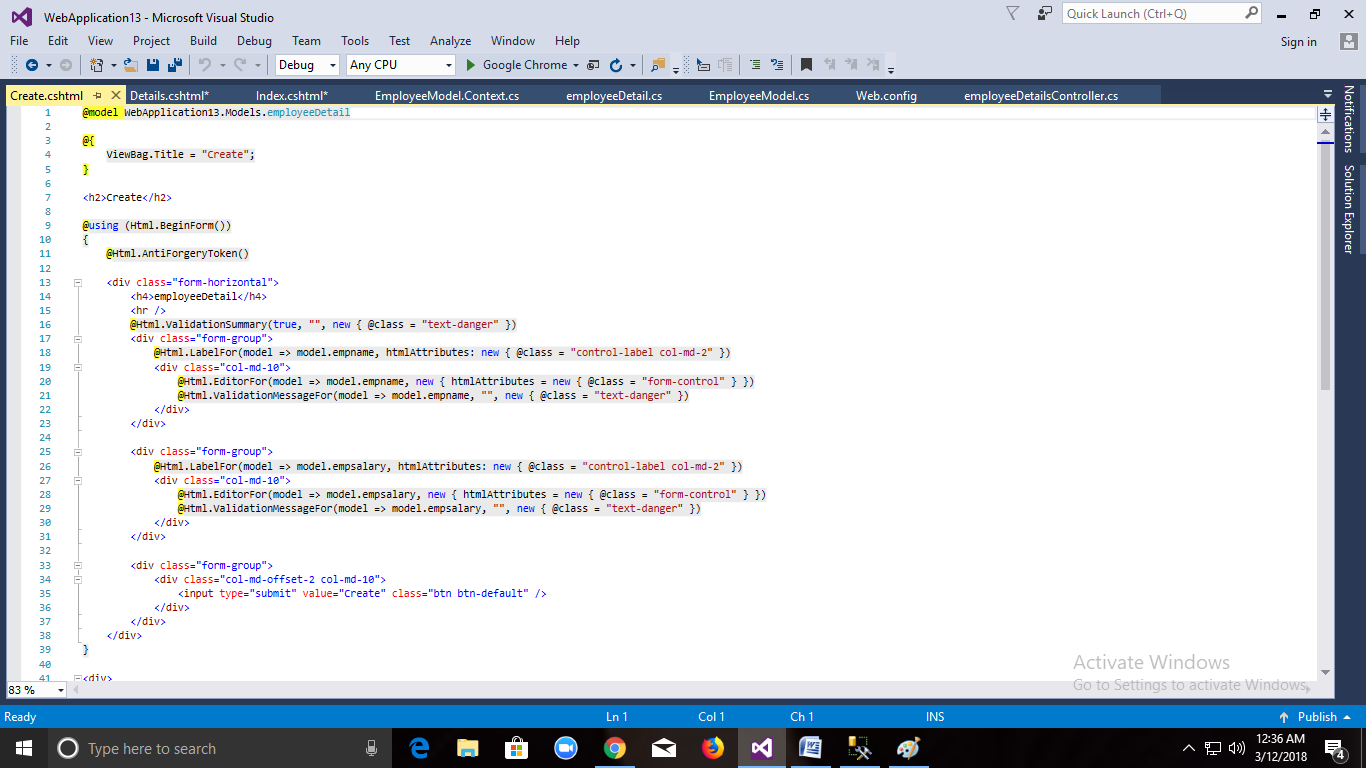
return RedirectToAction("Index");

}

return View(employeeDetail);

}

**View :**



// GET: employeeDetails/Edit/5

public ActionResult Edit(int? id)//When we Click Edit Button u will get value in Id

{

if (id == null)

{

return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

}

employeeDetail employeeDetail = db.employeeDetails.Find(id);//based on Id finding Record

if (employeeDetail == null)

It will send to edit view so that in that textbox we will get values.

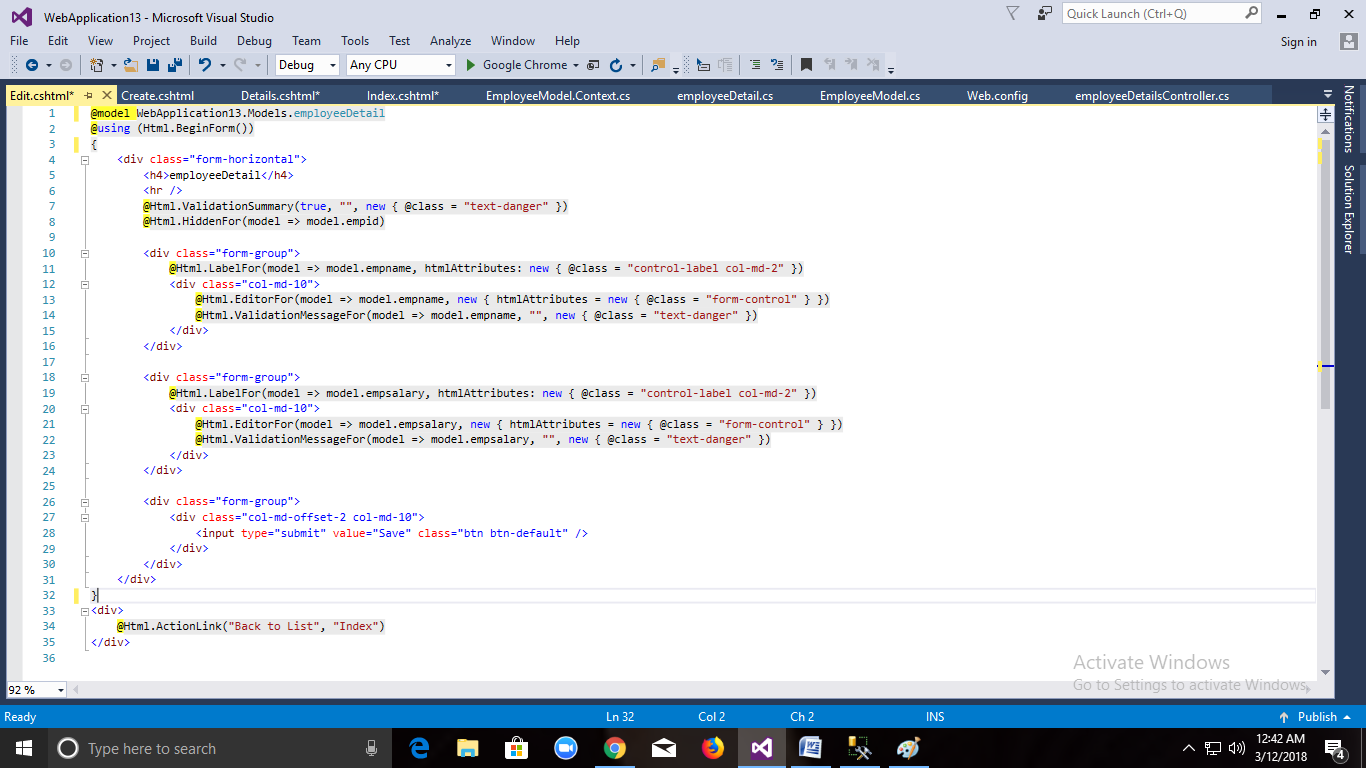
{

return HttpNotFound();

}

return View(employeeDetail);

}



// POST: employeeDetails/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 = "empid,empname,empsalary")] employeeDetail employeeDetail)

{

if (ModelState.IsValid)

{

db.Entry(employeeDetail).State = EntityState.Modified;

db.SaveChanges();//Updating table Values using savechanges method.

return RedirectToAction("Index");

}

return View(employeeDetail);

}

// GET: employeeDetails/Delete/5

public ActionResult Delete(int? id)//Load single Record and display Delete View

{

if (id == null)

{

return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

}

employeeDetail employeeDetail = db.employeeDetails.Find(id);

if (employeeDetail == null)

{

return HttpNotFound();

}

return View(employeeDetail);

}

// POST: employeeDetails/Delete/5

[HttpPost, ActionName("Delete")]

[ValidateAntiForgeryToken]

public ActionResult DeleteConfirmed(int id)

{

employeeDetail employeeDetail = db.employeeDetails.Find(id);

db.employeeDetails.Remove(employeeDetail);//By using Remove Method DeletingRecord

db.SaveChanges();

return RedirectToAction("Index");

}

protected override void Dispose(bool disposing)

{

if (disposing)

{

db.Dispose();

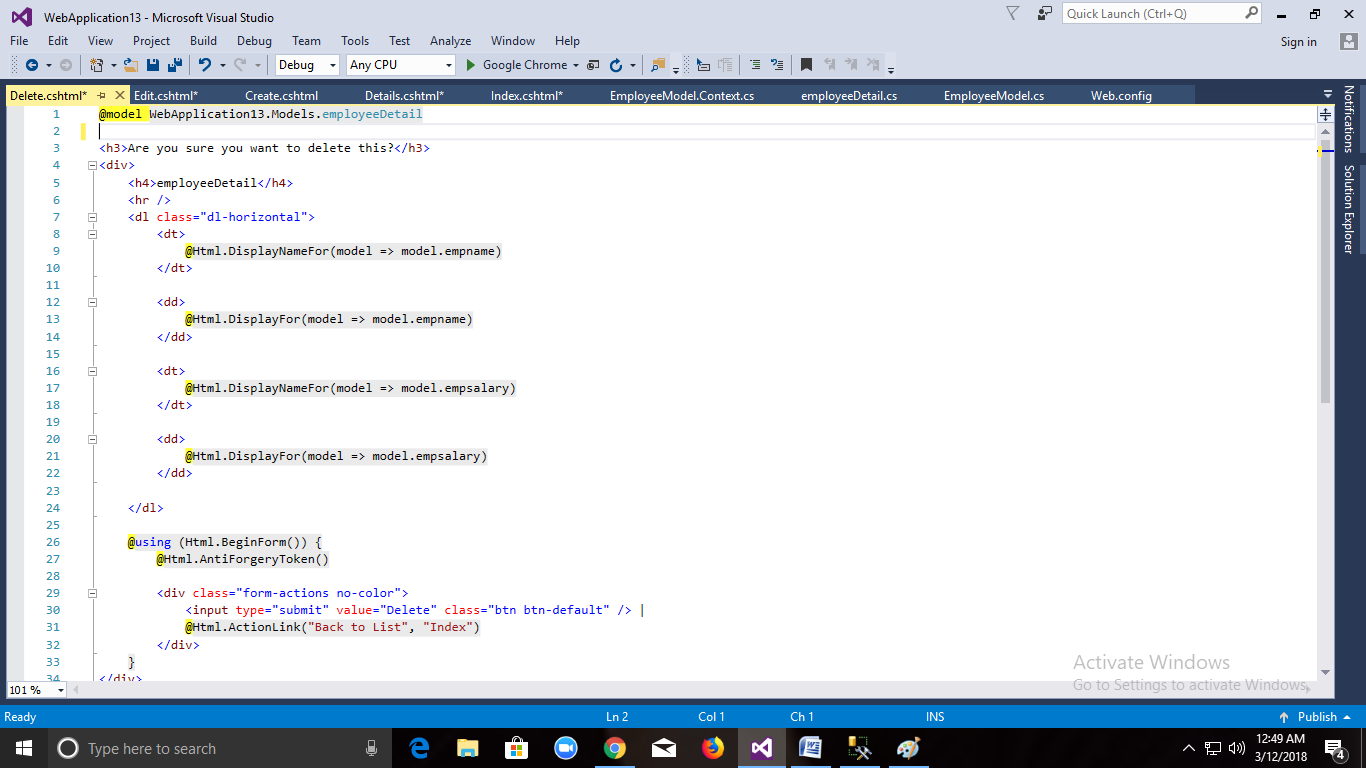
}

base.Dispose(disposing);

}

}

View :



We have Seen

@model WebApplication13.Models.employeeDetail

@model variable **which will be present top in a view .it is used to store single employee Record**

@model IEnumerable<WebApplication13.Models.employeeDetail>

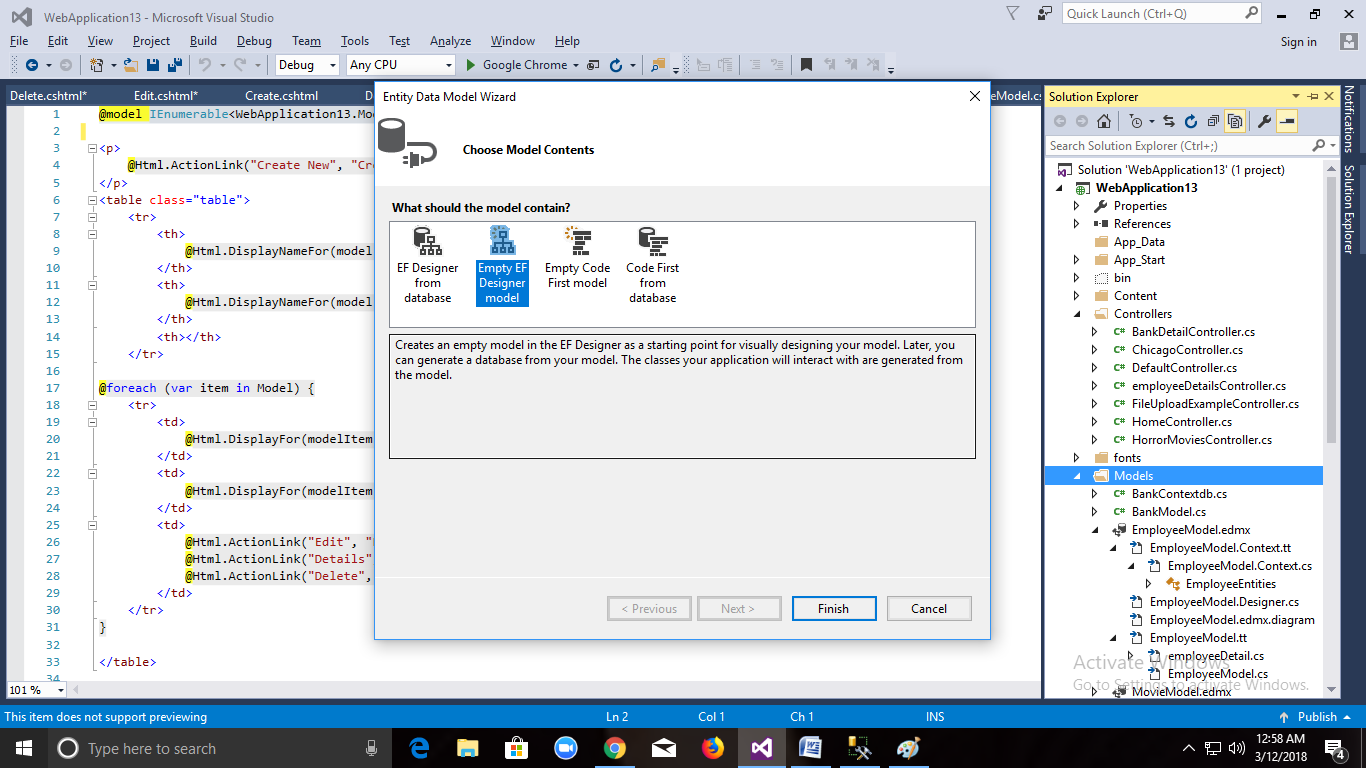
**Here @model variable will contain collection of Employee details**

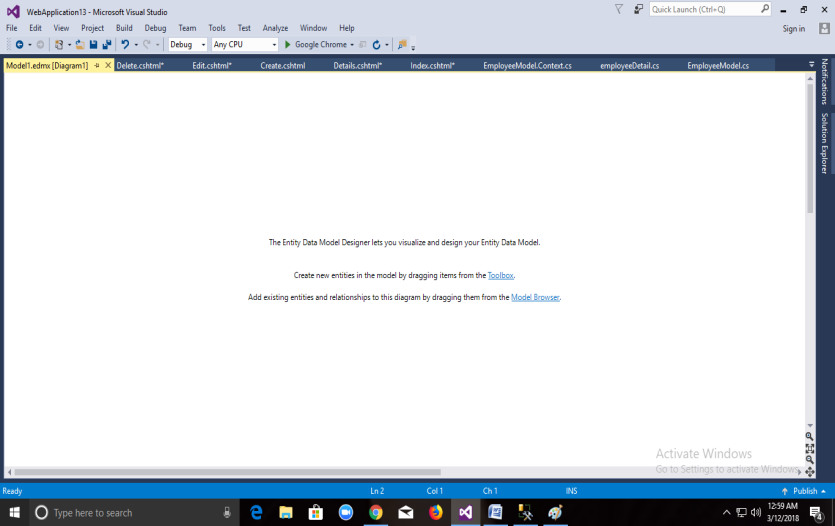
**Here by using database approach Entity framework will create Controller and View Automatically.**

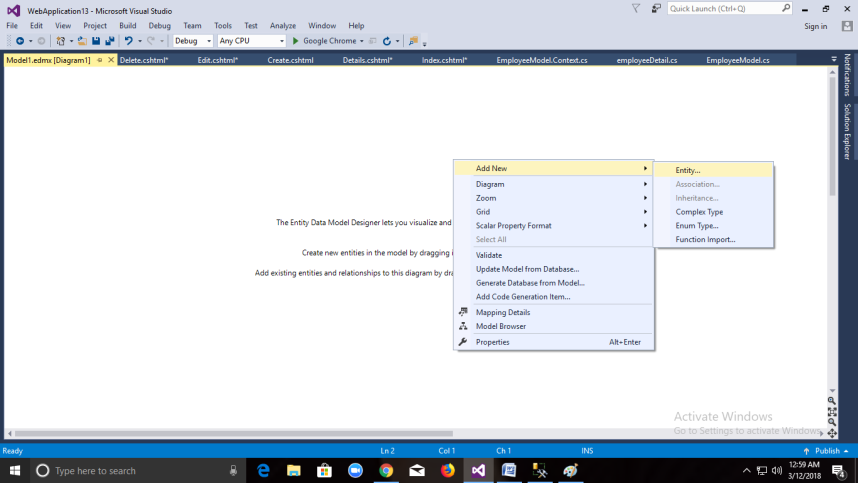
**Model Based Approach:**

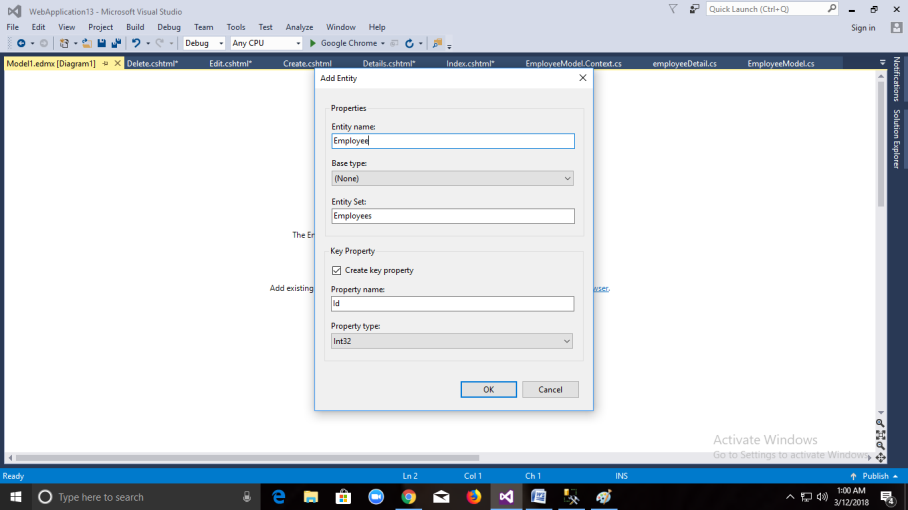
**Step 1:**

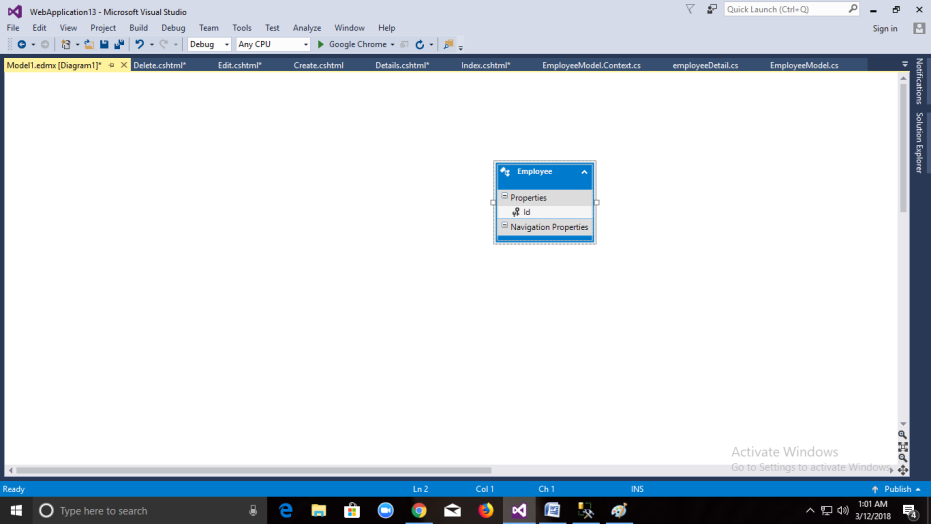
**First add Ado.net Framework and Select Model Based Approach**

****

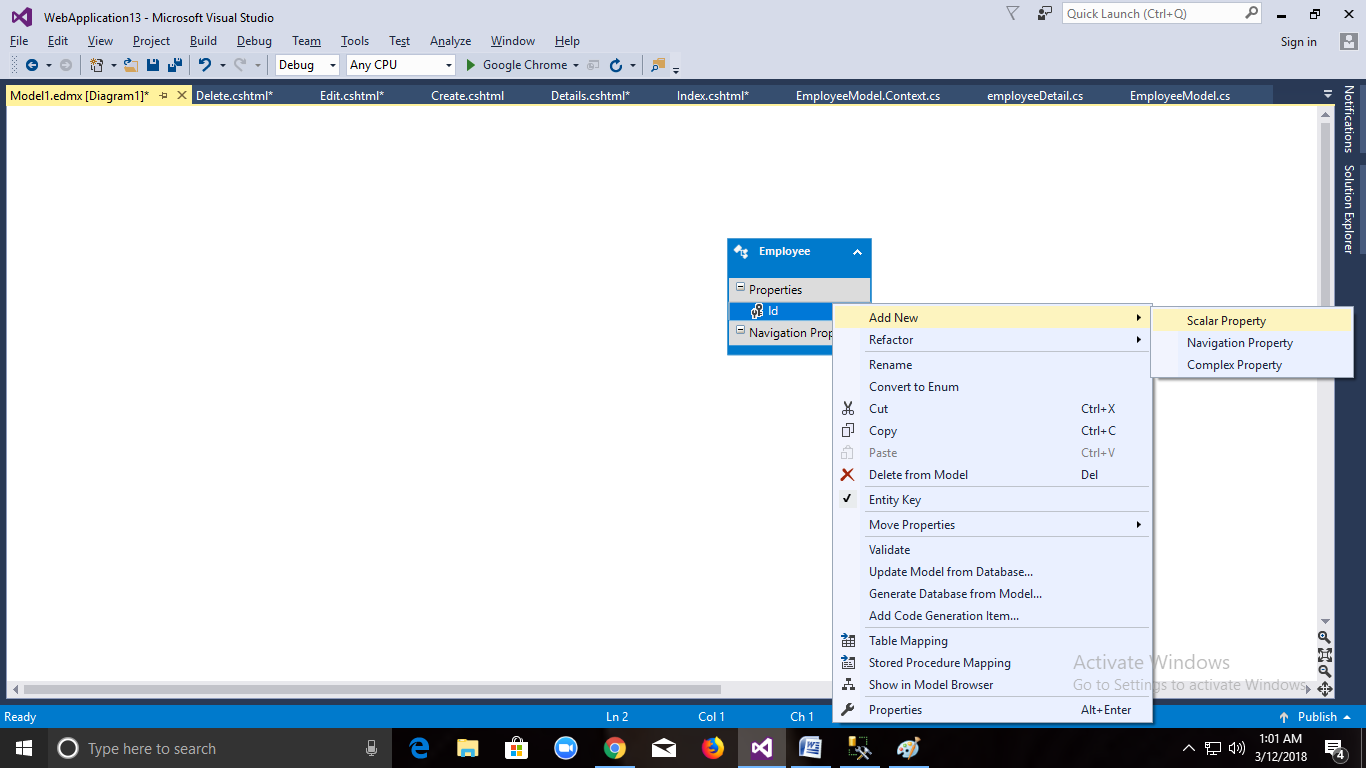
****

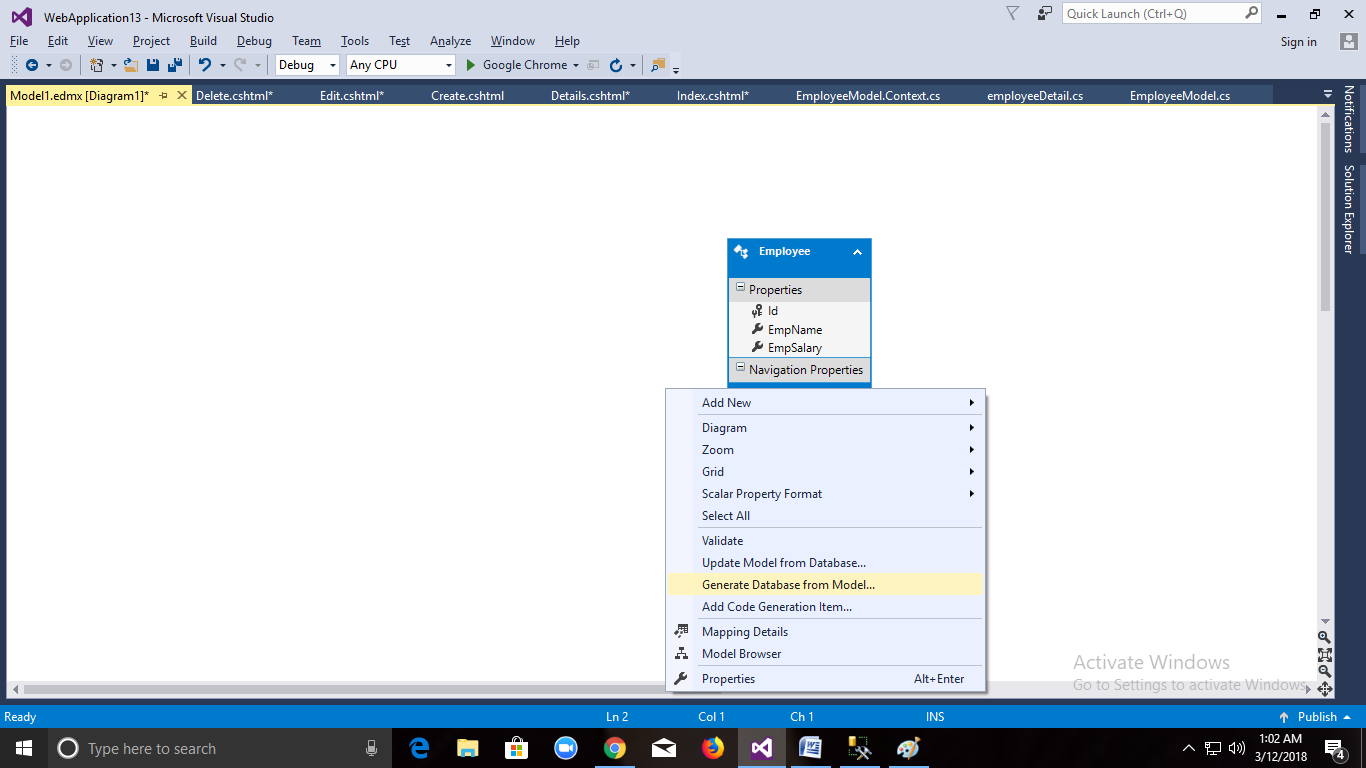
****

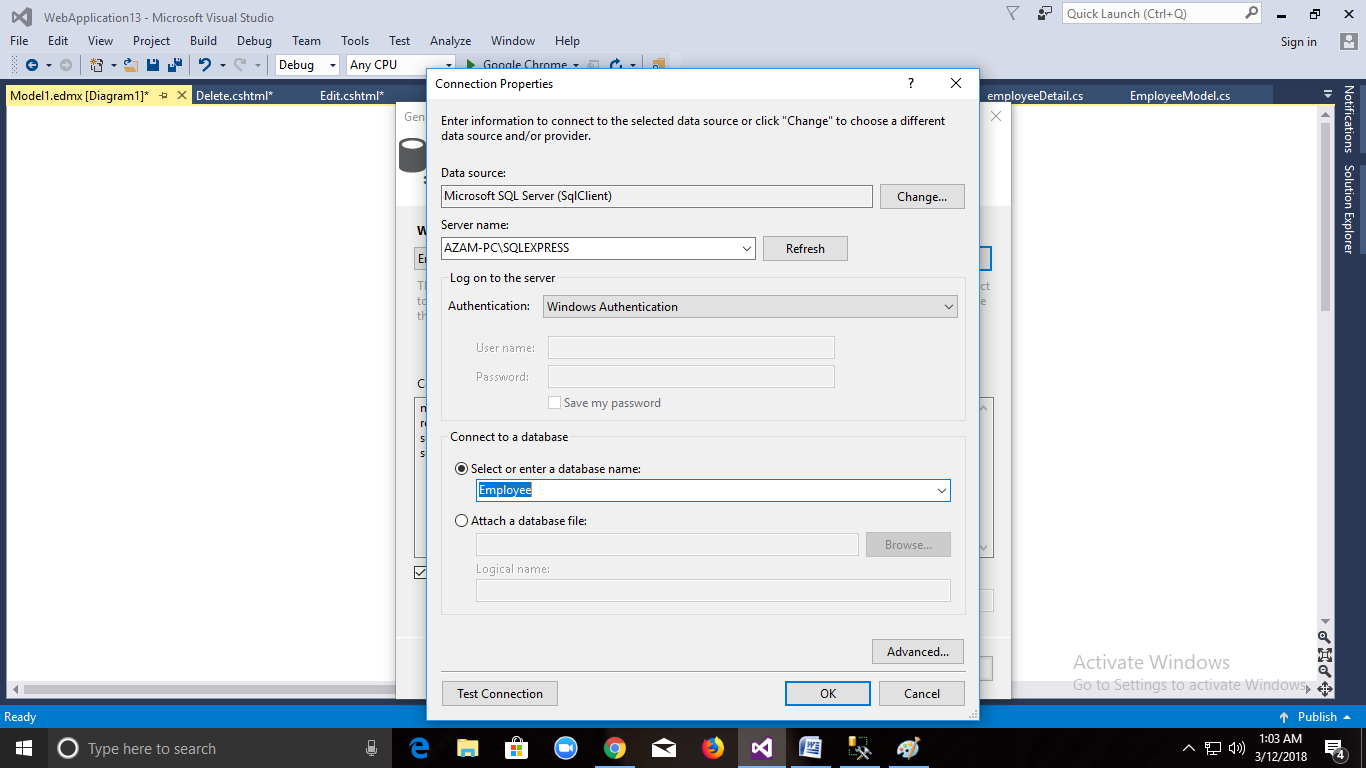
****

****

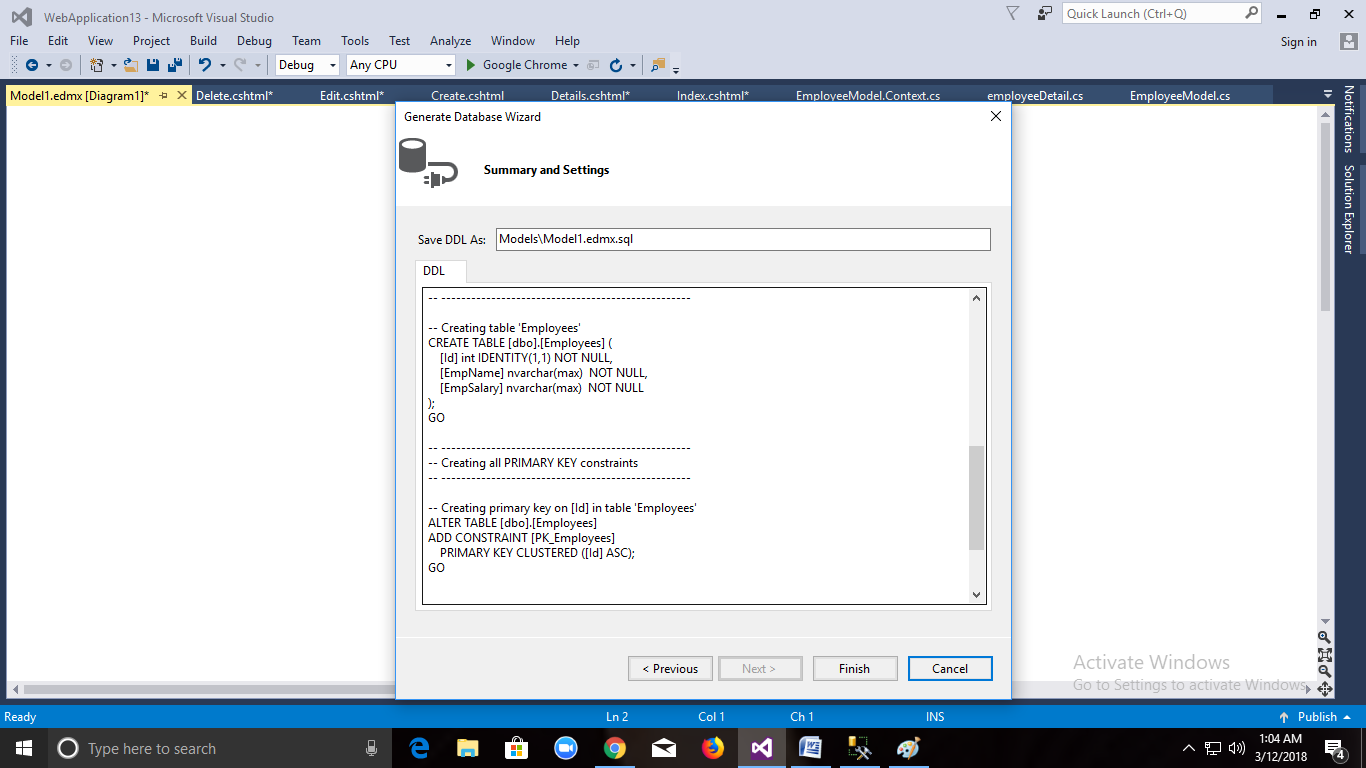
**To add Column**

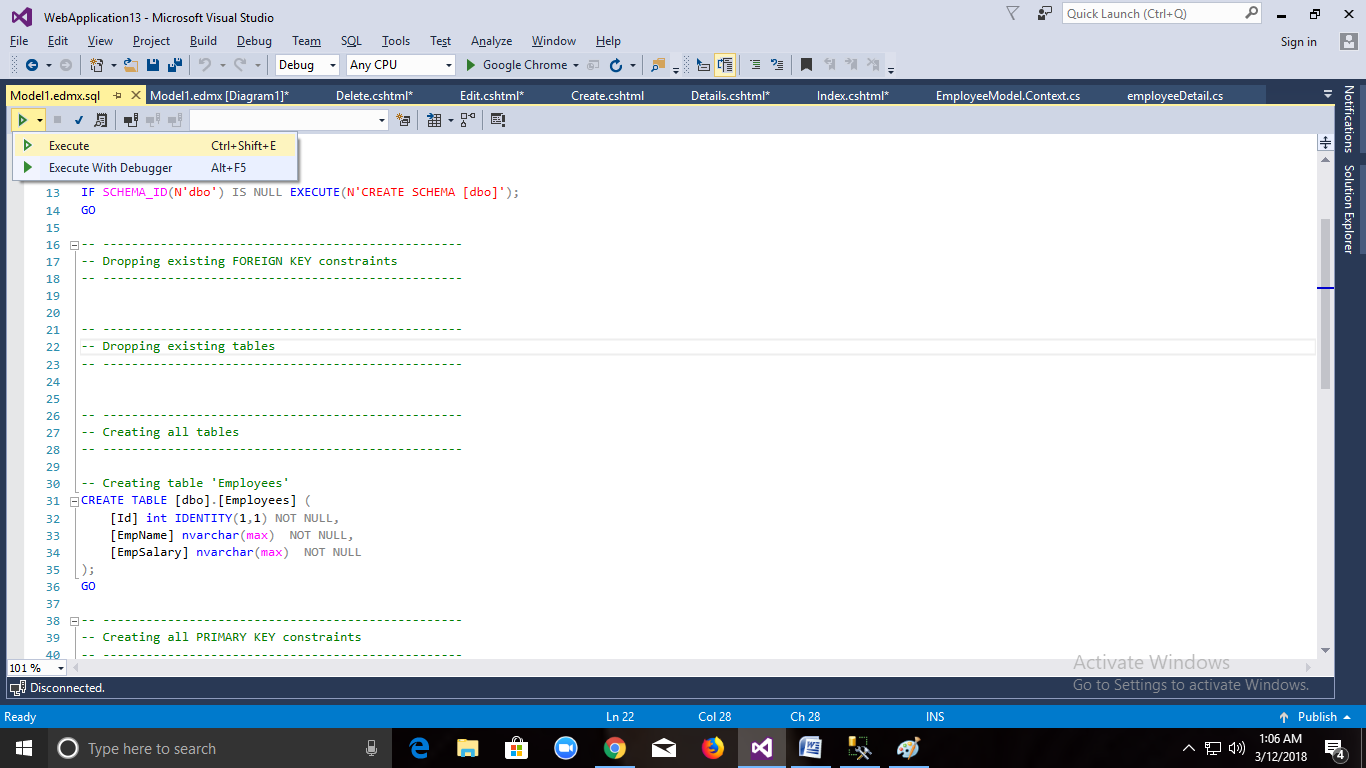
****

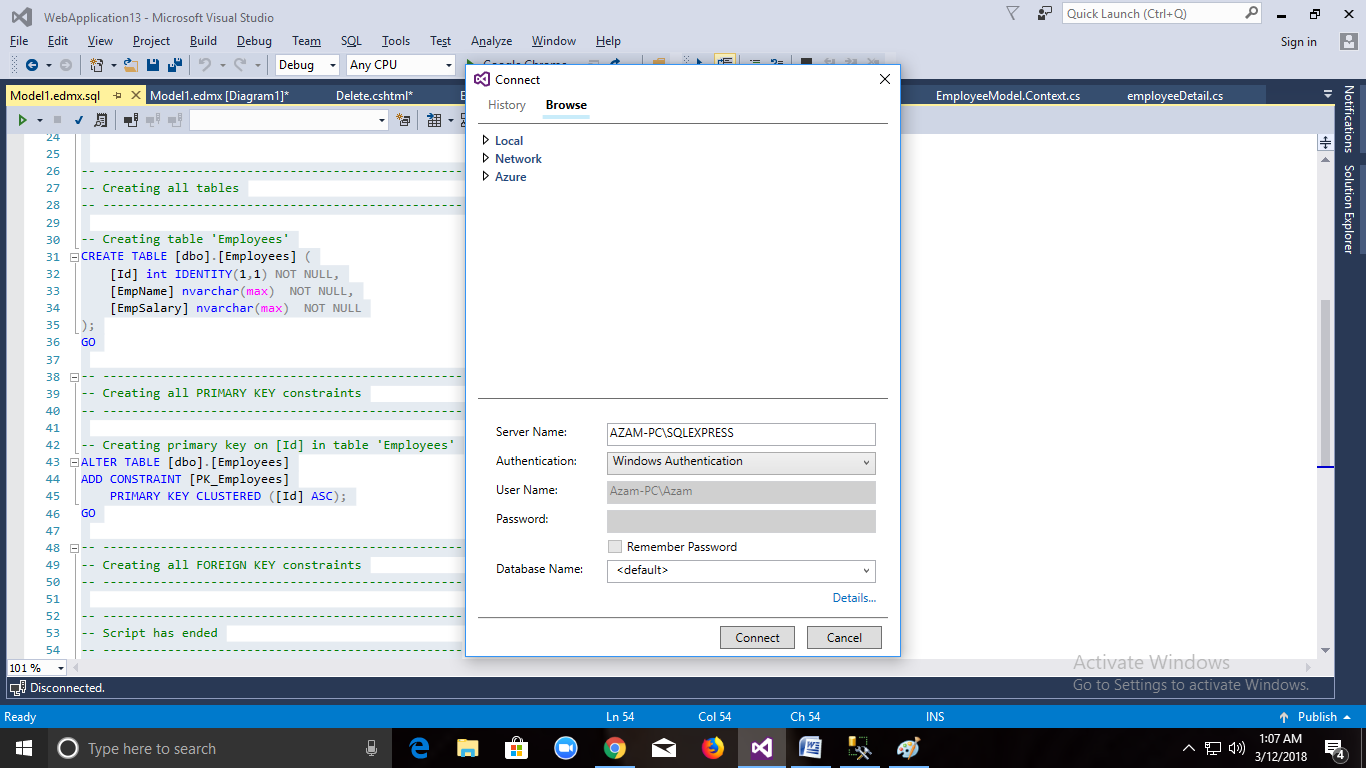
****

****

**Edmx.sql will be created**

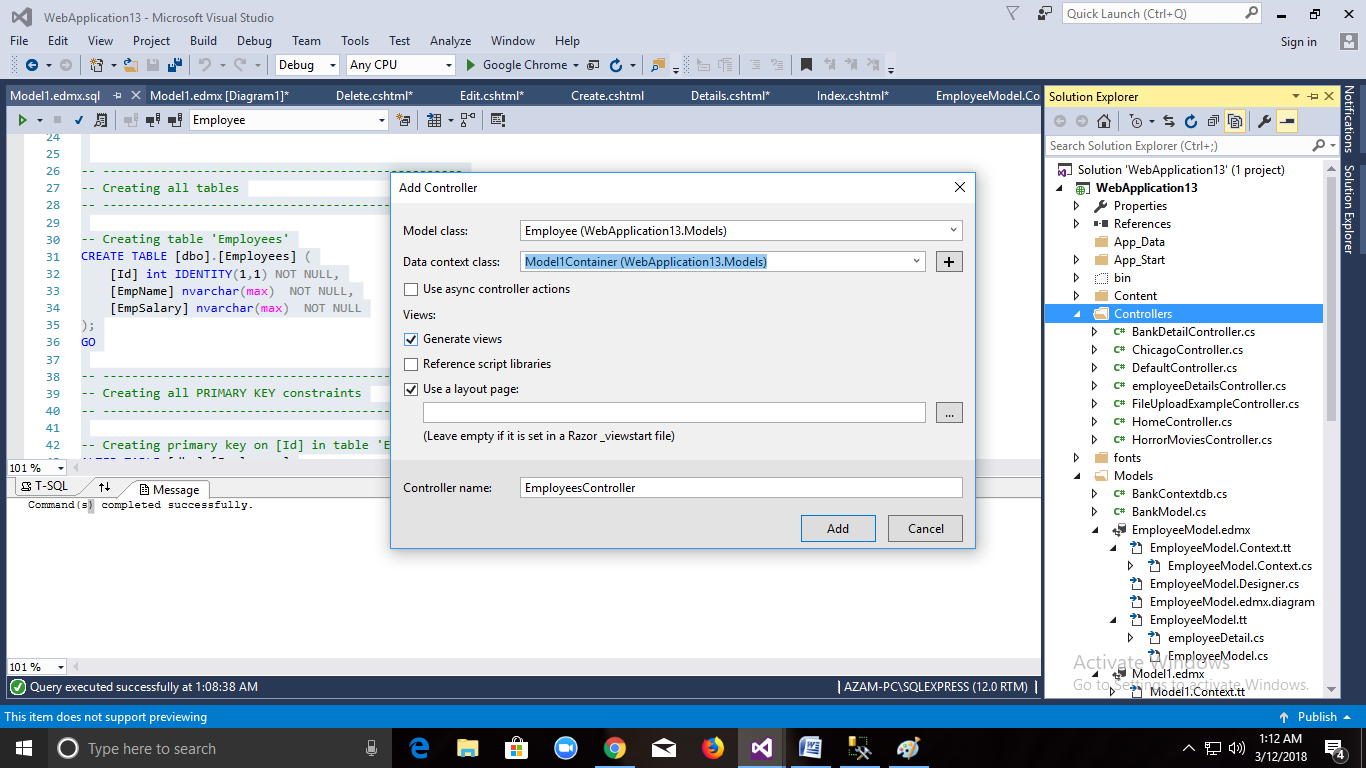
****

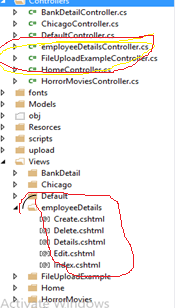
****

****

**Table will be created in the database which you have selected Now.**

**Now Create Controller and View**

****



**Introduction To ViewBag,ViewData,TempData,Session.**

ViewData is a dictionary object that is derived from ViewDataDictionary class.

public ViewDataDictionary ViewData { get; set; }

ViewData is a property of ControllerBase class.

ViewData is used to pass data from controller to corresponding view.

It’s life lies only during the current request.

If redirection occurs then it’s value becomes null.

It’s required typecasting for getting data and check for null values to avoid error.

Viewdata is used for transferring the data from controller to view for a corresponded or single request

public ActionResult Index()

{

List<string> obj = new List<string>();

obj.Add("pratiusha");

obj.Add("deepti");

obj.Add("Nagini");

obj.Add("Anusha");

ViewData["Student"] = obj;

return View();

}

<ul>

@foreach (string item in ViewData["Student"] as List<string>)

{

<li>@item</li>

}

</ul>

2)ViewBag:

ViewBag is a dynamic property that takes advantage of the new dynamic features in C# 4.0.

Basically it is a wrapper around the ViewData and also used to pass data from controller to corresponding view.

public Object ViewBag { get; }

ViewBag is a property of ControllerBase class.

It’s life also lies only during the current request.

If redirection occurs then it’s value becomes null.

It doesn’t required typecasting for getting data.

we are creating model in model folder

public class Employee

{

public int Empid { get; set; }

public string EmpName { get; set; }

public string Department { get; set; }

}

--->

creating Controller

public ActionResult Index()

{

Employee emp = new Employee();

emp.Empid = 1;

emp.EmpName = "prfatiusha";

emp.Department = "IT";

ViewBag.Details = emp;

return View();

}

under view

<ul>

<li> @ViewBag.Details.Empid</li>

<li> @ViewBag.Details.EmpName</li>

<li> @ViewBag.Details.Department</li>

</ul>

or

@{

ViewBag.Title = "Index";

var myfellow = ViewBag.Details;

}

<h2>Index</h2>

<ul>

<li> @myfellow.Empid</li>

<li> @myfellow.EmpName</li>

<li> @myfellow.Department</li>

</ul>

**TempData:**

TempData is a dictionary object that is derived from TempDataDictionary class and stored in short lives session.

public **TempDataDictionary** TempData { get; set; }

TempData is a property of ControllerBase class.

**TempData** is used to pass data from current request to subsequent request (means redirecting from one page to another).

It’s life is very short and lies only till the target view is fully loaded.

It’s required typecasting for getting data and check for null values to avoid error.

It is used to store only one time messages like error messages, validation messages. To persist data with TempData refer this article: Persisting Data with TempData

**TempData**:it is used to transfer the data from one controller to another controller then you can use tempdata

Tempdata is also Tempdictionary object

Tempdata["key"]=value;

Tempdata is used to retain the values by two ways

1)**keep** method: it is used to retain the values which is present in tempdata

return type is void

Tempdata.keep();

**2)peek method:**

it is used for getting the object by using the key and will retain the values

by not allowing the key to be deleted

Tempdata.peek("key");both accessing value and for retaing the value

public ActionResult Index()

{

TempData["Student"] = "Pratiusha";

return RedirectToAction("About");

}

public ActionResult About()

{

var b = TempData.Peek("Student");

var c = TempData.Peek("Student");

ViewData["b"] = b;

ViewData["c"] = c;

return View();

}

Session is very well known concept in any web application. It is used to pass data from page to page. Basically web application is stateless in nature. So, to maintain state across request and response we need to use few technique, Session is one of them.

Using session variable we can pass data from model to controller or controller to view.

Create Model:

namespace MVC3.Models

{

public class Customer

{

public String Name { get; set; }

public String Surname { get; set;

}

}

}

Create Controller:

namespace MVC3.Controllers

{

public class CustomerController : Controller

{

public ActionResult Index()

{

Customer Obj = new Customer();

Obj.Name = "Dinkar";

Obj.Surname = "Santosh";

Session["Customer"] = Obj;

return View();

}

}

}

Within controller we are creating object of model class(Customer) and populating with data. Then we are inserting object of customer class within session variable.

<div>

@{

var CustomerInfo = (MVC3.Models.Customer) Session["Customer"];

}

Customer Name is :- @CustomerInfo.Name;

<br />

Customer Surname is :-@CustomerInfo.Surname;

</div>

Html Helper in MVC:

It provides an easy way to Render [HTML](http://www.c-sharpcorner.com/search/html) in our View.

The following is the list of Html Helper controls.

* Html.Beginform
* Html.EndForm
* Html.Label
* Html.TextBox
* Html.TextArea
* Html.Password
* Html.DropDownList
* Html.CheckBox
* Html.RedioButton
* Html.ListBox
* Html.Hidden

Below are Strongly Type Html Helper methods, this will allow us to check compile time errors. We get Model's Property intelligence at Runtime.

* Html.LabelFor
* Html.TextBoxFor
* Html.TextAreaFor
* Html.DropDownListFor
* Html.CheckBoxFor
* Html.RadioButtonFor
* Html.ListBoxFor
* Html.HiddenFor

Html helpers

**TextBox:**

@Html.Label("Username")

@Html.TextBox("txtUsername","pratiusha",new { @class="mytest", @readonly="true"})<br/>

**Password:**

<label>Password:</label>

@Html.Password("txtpwd")<br />

**RadioButton**

<label>Gender</label>

@Html.RadioButton("rbgender","Male",true)<span>Male</span>

@Html.RadioButton("rbgender", "Female")<span>Female</span><br />

**CheckBox**

<label>Choose Courses</label>

@Html.CheckBox("dotnet")<span>Dotnet</span>

@Html.CheckBox("Java")<span>Java</span><br />

**DropdownList**

<label>Select Country</label>@Html.DropDownList("Country",new List<SelectListItem>()

{

new SelectListItem {Text="India",Value="1" },

new SelectListItem {Text="China",Value="2",Selected=true},

new SelectListItem {Text="Bangladesh",Value="3" }

},"select Country")<br />

**TextArea:**

<label>Comments</label> @Html.TextArea("comments",null,5,10,null)

**FileUpload Control**

@using (Html.BeginForm("Index","Home",FormMethod.Post,new {enctype="multipart/form-data"}))

{

<span>File</span>

<input type="file" id="fileupload" name="fileupload" />

<input type="submit" value="upload" />

}

@ViewBag.msg

public ActionResult Index()

{

return View();

}

[HttpPost]

public ActionResult Index(HttpPostedFileBase fileupload)

{

string filename = Path.GetFileName(fileupload.FileName);

string path = Server.MapPath("~/upload");

fileupload.SaveAs(Path.Combine(path, filename));

ViewBag.msg = "uploaded successfully";

return View();

}

Customize Helper Example:

@using HtmlHelpersExample.CustomHelpers;

@myhtmlhelper.MyLabel("pratiusha")

@Html.createUrControl("file")

@helper labelbyrazzorcode(string content)

{

<label>@content</label>

}

@labelbyrazzorcode("hi this is ali")

Create class for customize control

using System.Web.Mvc;

namespace HtmlHelpersExample.CustomHelpers

{

public static class myhtmlhelper

{

public static IHtmlString MyLabel(string content)

{

string htmlstring = String.Format("<label>{0}</label>", content);

return new HtmlString(htmlstring);

}

public static IHtmlString createUrControl(this HtmlHelper helper, string content)

{

string htmlstring ="<input type="+content+">";

return new HtmlString(htmlstring);

}

}

}

CheckboxList in Mvc

Step1:

Create Employee Model

Public int Id {get;set;}

Public string EmpName {get;set};

Public bool checked{get;set;}

Step2:

Add Controller:

[httpGet]

Public ActionResult Index()

{

Var list=new List<EmployeeModel>

{

new EmployeeModel{Id=1,Name=”dinkar”,Checked=false},

new EmployeeModel{Id=2,Name=”salman”,Checked=false},

new EmployeeModel{Id=3,Name=”santosh”,Checked=false},

}

return View(list);

}

[HttpPost]

Public ActionResult Index(List<EmployeeModel> liobj)

{

return View(liobj);

}

**Strongly Type Helper:**

@model WebHelper.Models.Register

@{

var MyList = new List

<SelectListItem>(){

new SelectListItem(){Value="1",Text="India"},

new SelectListItem(){Value="2",Text="UK"}

};

}

@using (Html.BeginForm("Index", "Home", FormMethod.Post))

{

@Html.LabelFor(m => m.Name)

@Html.TextBoxFor(m => m.Name)

<br />

@Html.LabelFor(m => m.Phone)

@Html.TextBoxFor(m => m.Phone)

<br />

@Html.LabelFor(m => m.Email)

@Html.TextBoxFor(m => m.Email)

<br />

@Html.LabelFor(m => m.Password)

@Html.PasswordFor(m => m.Password)

<br />

@Html.Label("Male")

@Html.RadioButtonFor(m => m.Gender, "Male", new { value = "Male" })

@Html.Label("Female")

@Html.RadioButtonFor(m => m.Gender, "Female", new { value = "Female" })

<br />

@Html.LabelFor(m => m.Country)

@Html.DropDownListFor(m => m.Country, MyList)

<br />

@Html.LabelFor(m => m.Terms)

@Html.CheckBoxFor(m => m.Terms)

<br />

<input type="submit" value="Submit" />

}