Diving into Knockout JS

(A Quick start with Knockout JS)

Akhil Mittal

Sr. Analyst (Magic Software)

https://www.facebook.com/csharppulse

https://in.linkedin.com/in/akhilmittal

https://twitter.com/AkhilMittal20

google.com/+AkhilMittal

https://www.codeteddy.com

https://github.com/akhilmittal

About Author

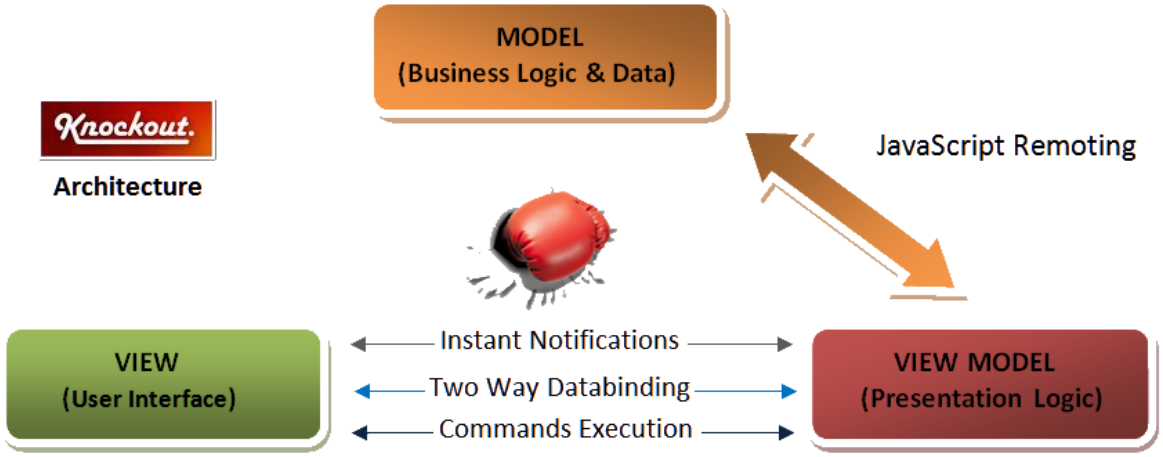
*Akhil Mittal is a Microsoft MVP, C# Corner MVP, a Code project MVP, blogger, programmer by heart and currently working as a Sr. Analyst in **Magic Software** and have an experience of more than 9 years in C#.Net. He is a B.Tech in Computer Science and holds a diploma in Information Security and Application Development. His work experience includes Development of Enterprise Applications using C#, .Net and SQL Server, Analysis as well as Research and Development. He is a MCP in Web Applications (MCTS-70-528, MCTS-70-515) and .Net Framework 2.0 (MCTS-70-536). Visit his personal blog **CodeTeddy** for more informative articles.*

Akhil Mittal
Sr. Analyst (Magic Software)

Introduction to Knockout.js

Knockout :

In today's changing trend of development data driven apps depends largely on JavaScript and JS based libraries such as jQuery. The client side programming appears to become complex and complex because the user interface becomes more and more rich. In scenarios like this the data binding and dependency tracking are highly desirable in the applications for further extensibility of the application. Knockout JS fulfils these rich requirements on client side programming and makes a developers life easy and joyfull. Lets discuss KO in detail.



Knockout.JS (KO) is basically a JS library that enables Declarative Bindings using an 'Observable' ViewModel on the client (browser) following observer pattern approach, enabling UI to bind and refresh itself automatically whenever the data bound is modified. Knockout.JS provides its own templating pattern that helps us to bind our view model data easily. KO works on MVVM pattern i.e. Model-View-ViewModel.
As the architecture is shown, Views interact with View Models in a two way binding manner i.e. when model is changed view updates itself and when view is updated , model too updates itself instantaneously.
KO provides 3 most important features like,

- Automatic Refresh of UI
- Two way binding
- Templating

The whole idea of KO derives from these three major functionalities. KO also helps in developing Single page applications (SPA's).SPA's are out of the box new way of developing rich internet applications(RIA's) in todays era.

Model-View-View Model (MVVM):

When we develop a rich UI internet based application, we create Views(UI like html and aspx pages) using server controls, html controls and then extend our application by writing business logic behind those views like event handling, property binding, creating entities. This approach increases complexities when application

is too large.Here we require separation of concerns and maintainability of the application,specially on client side.

The MVVM pattern includes three key parts:

Model (Business rule, data access, model classes,Data displayed in UI)
View (User interface (html, aspx,cshtml…))
ViewModel (Event handling, binding, business logic)

Model refers to our application data and domain model i.e. entities. In a traditional ASP.NET web application, the data is basically stored inside database or files and UI fetches the data using client-server request like ajax or direct bind itself.

View Model contains the User Interface level operations/methods/functions, performed on model data to bind the outcome to view. The operations include business logic validations and checks to be performed before binding data to UI. View models act as interface between model and views and acts as a wrapper over model prior binding to Views.

View is the user interface of our application. View talks to View Model to invoke certain methods/operations as explained above. View gets updated automatically whenever data from the View Model changes.

MVVM provides a clear separation of concerns between the user interface (UI) and the business logic.
In the MVC pattern, a view acts as the broker agent between the Model (the data displayed in the View) and the Controller (server-side endpoint that takes data from the View and performs some action on that data and provides a response.

Observables for two way binding:

KO provides Observables in the library to be bound to UI elements and simultaneously code is written to view models,so that when view updates the data the model updates itself and vice versa, for eg, in the following code,

```
<tr>
  <td>Batch :</td>
  <td>
    <input data-bind="value: Batch" /></td>
  <td><span data-bind="text: Batch" /></td>
</tr>
<tr>
  <td>Address :</td>
  <td>
    <input data-bind="value: Address" /></td>
  <td><span data-bind="text: Address" /></td>
</tr>
<tr>
  <td>Class :</td>
```

```
        <td>
          <input data-bind="value: Class" /></td>
        <td><span data-bind="text: Class" /></td>
      </tr>
```

Above code shows a part of view, you can see the elements are bound to properties like text and value, these properties are provided by KO, and the right side of these properties are property key names which are bind in view-models with the help of observables like shown below,

```
    var self = this;
  self.Batch = ko.observable();
  self.Address = ko.observable();
  self.Class = ko.observable();
```

So this would be the code in View model, any-ways we'll be discussing all this in detail.

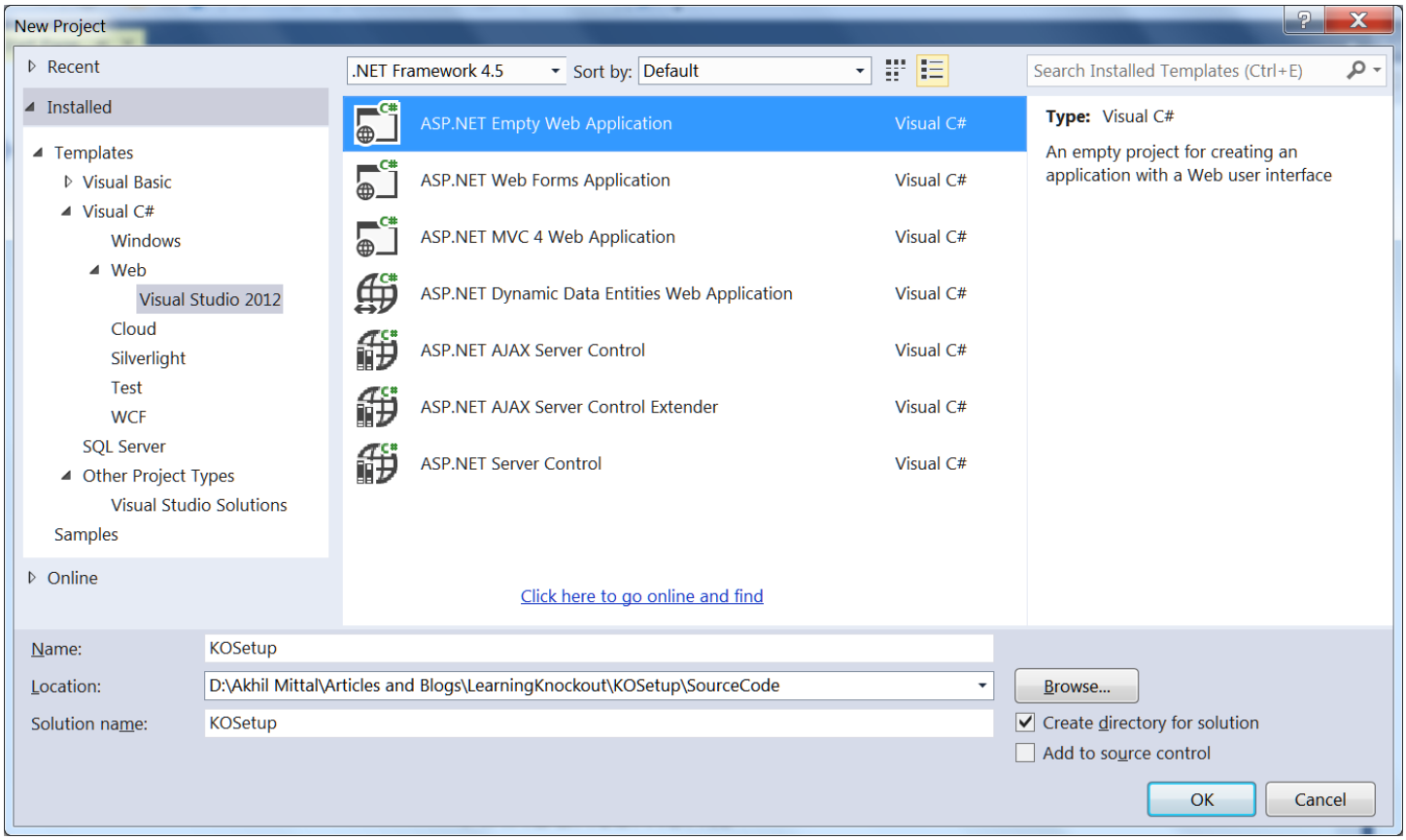
NOTE: data-bind is an htmnl5 attribute.

CRUD Operations in Asp.Net Web Forms using Knockout.JS

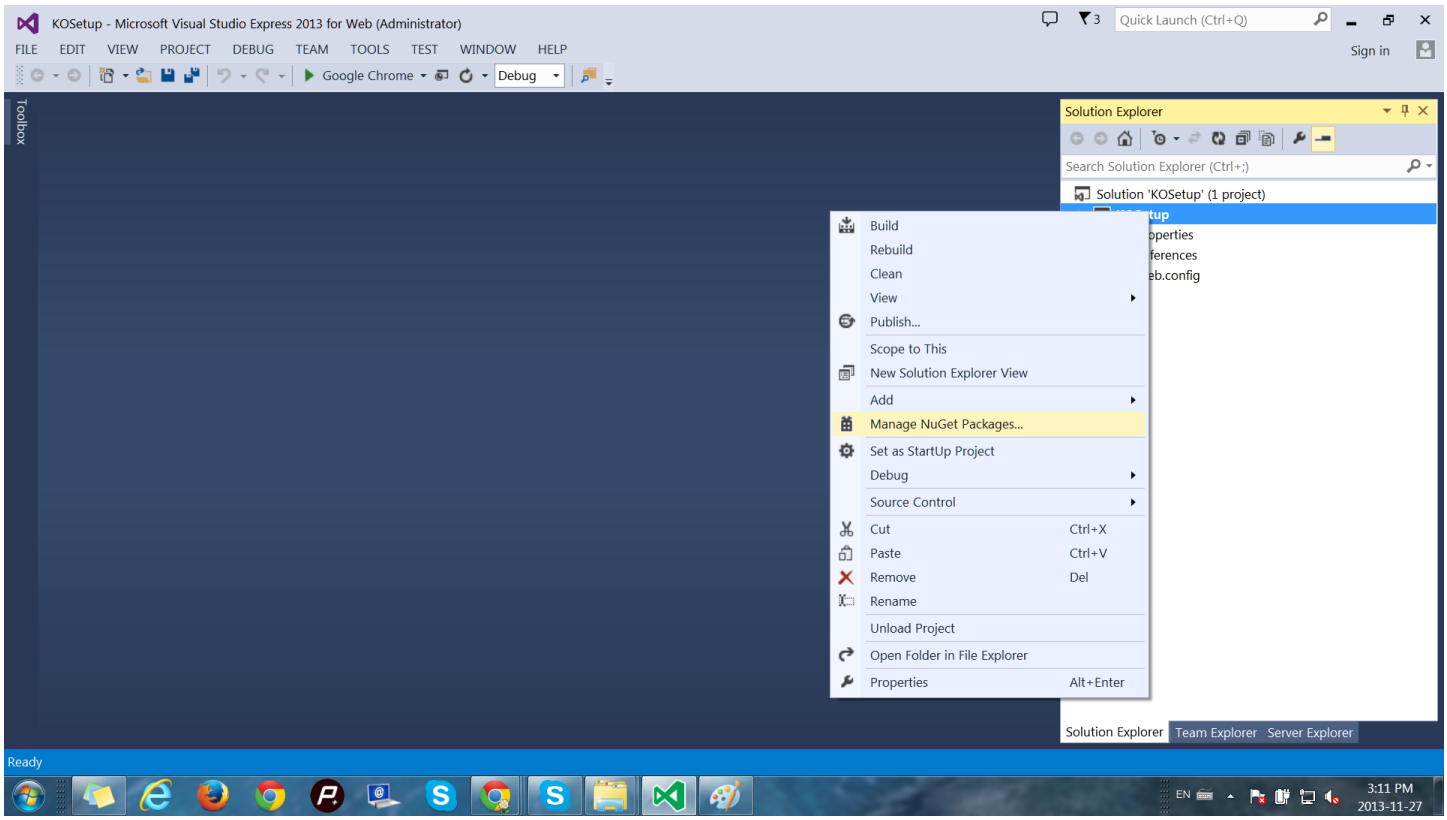
Setting up Environment in Visual Studio for KO

We go step by step to set up Knockout js environment in Visual Studio.
The pre-requisite is Visual Studio must be version greater than or equal to 12. I am using Visual Studio 2013 Express.

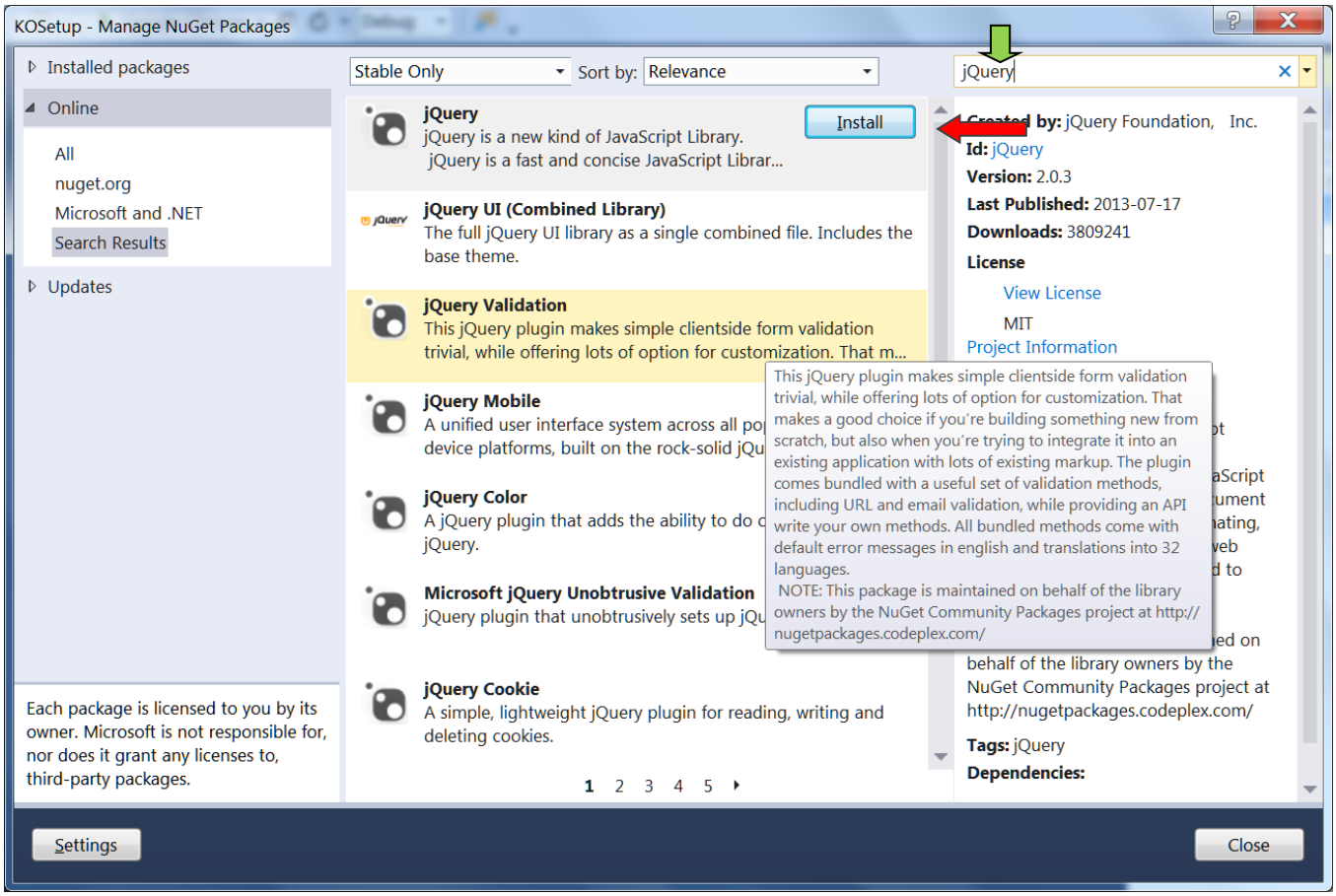
1. **Step1:** Open Visual Studio and create a simple asp.net application, I have given it a name KOSetup



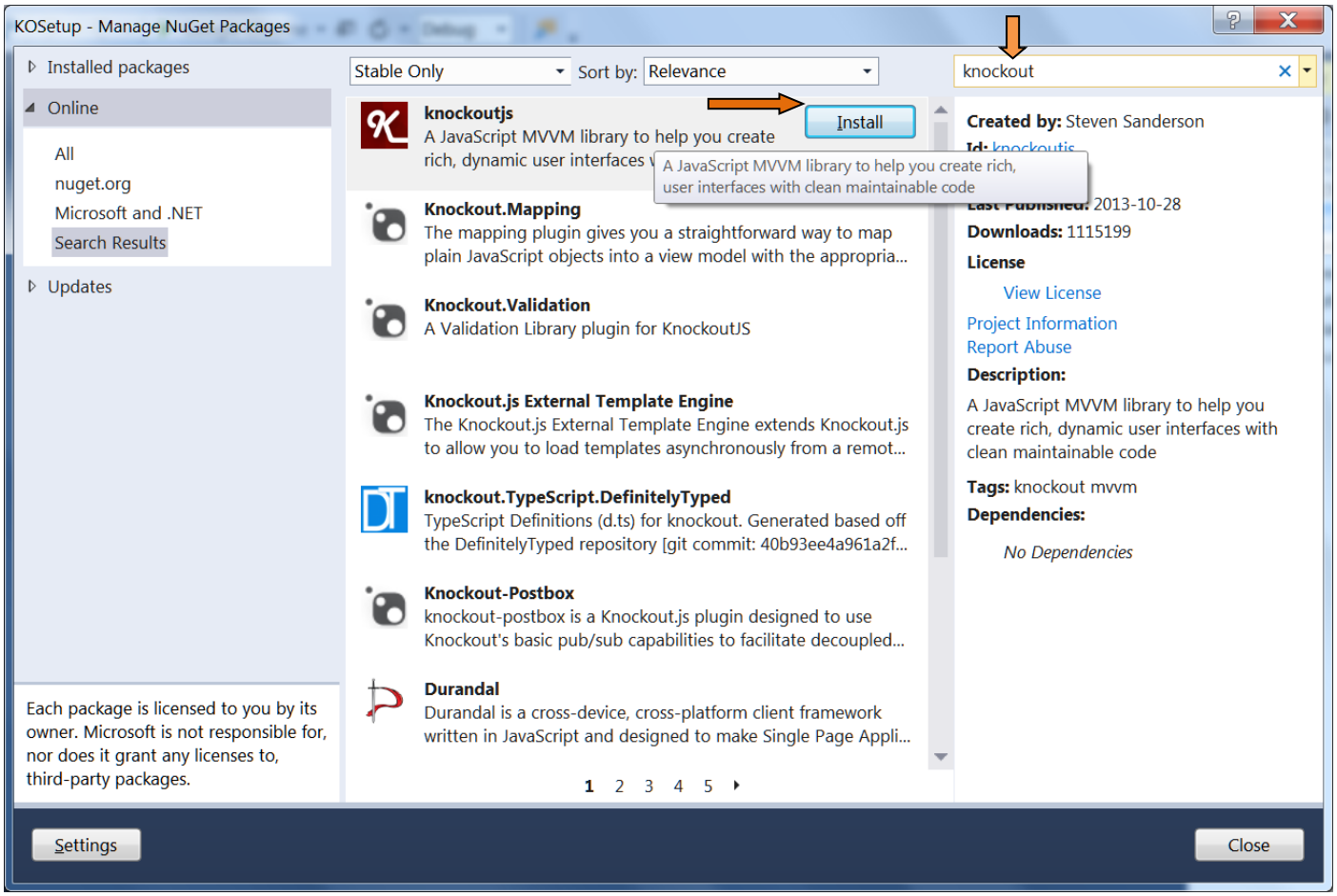
2. **Step2:** Right click on project, and in context menu select manage Nuget packages.. to install JQuery and KO.



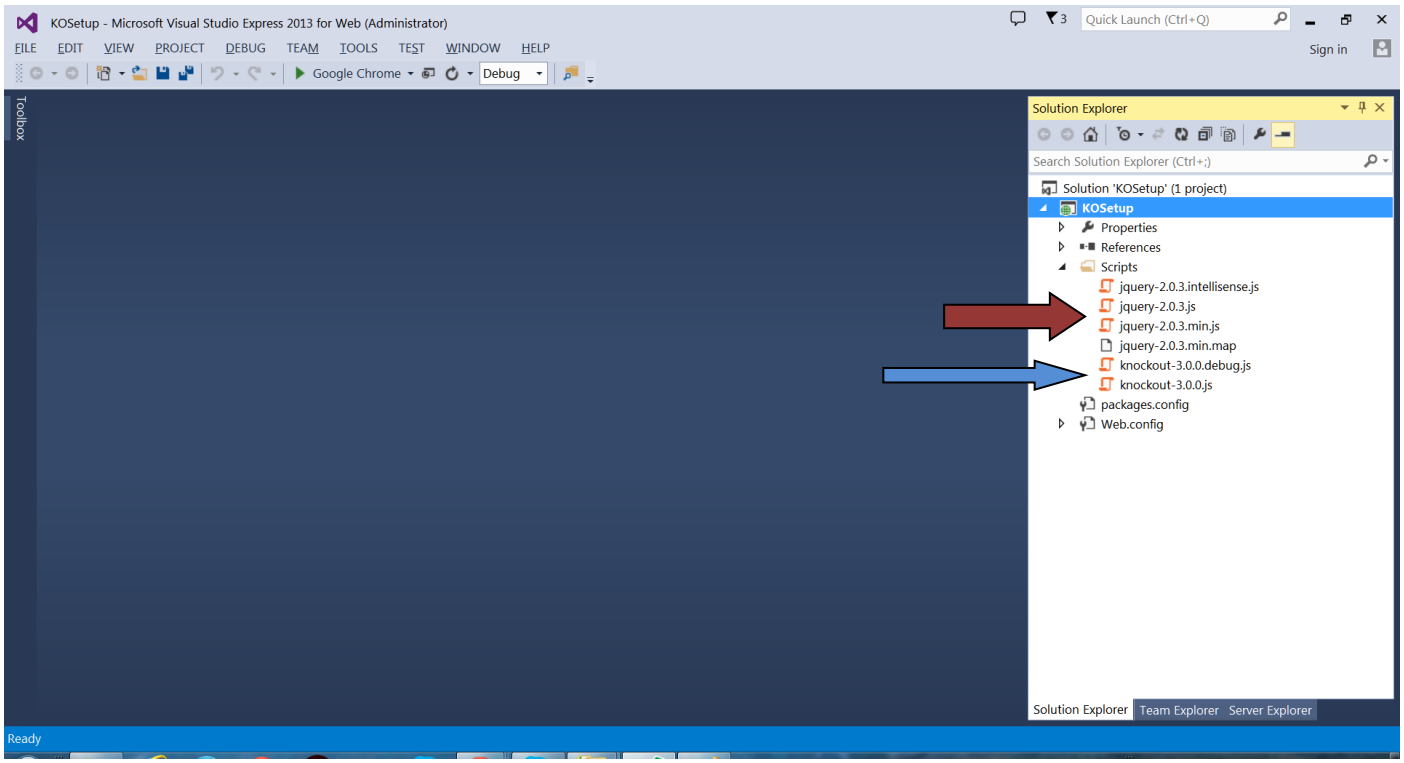
3. **Step3:** Type jQuery in search text box to get the latest compatible jQuery library.Click install to install the library.



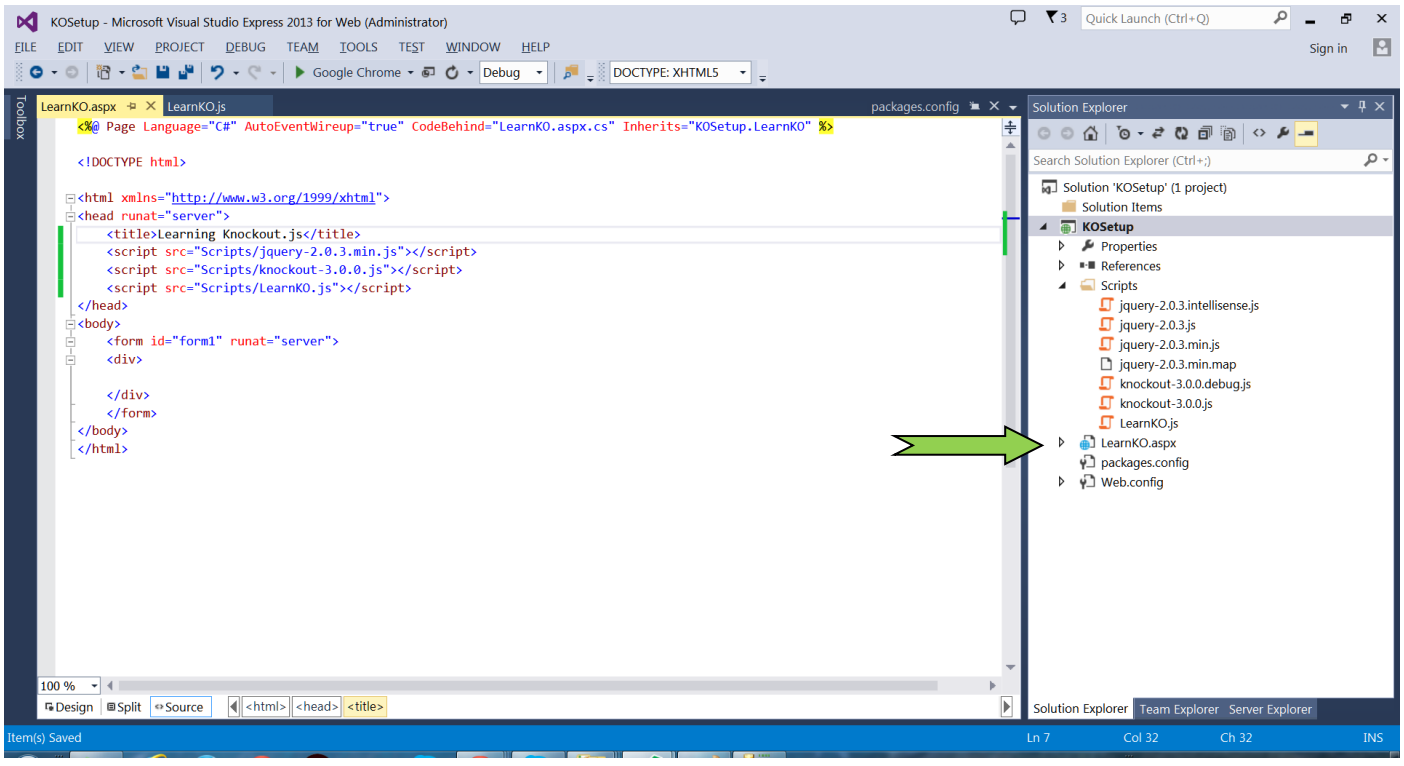
4. **Step4:** In the similar fashion, search 'knockout' in search textbox and install knockoutjs library in your application.



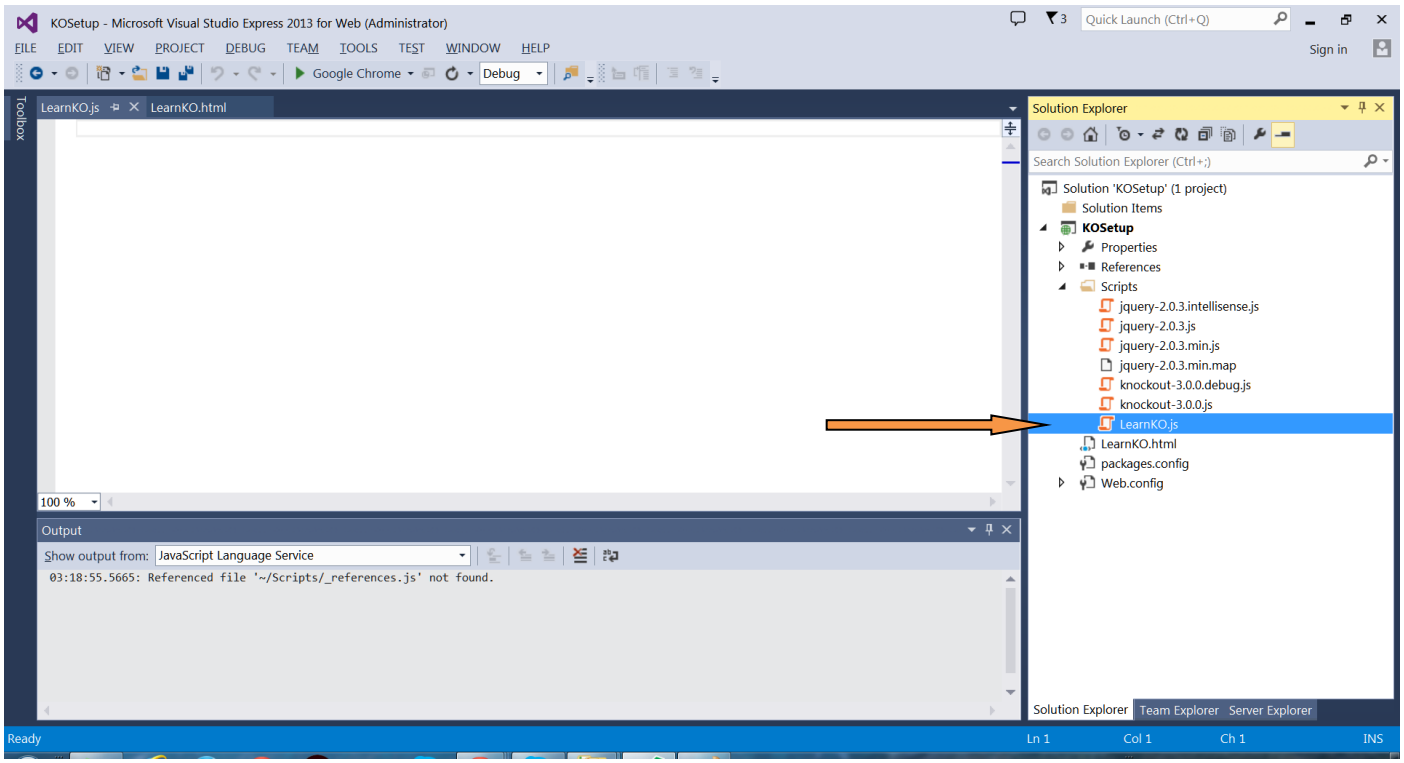
5. **Step5:** Our solution will look like,We have a folder ctreatet named Scripts and that contains jQuery and knockout libraries.



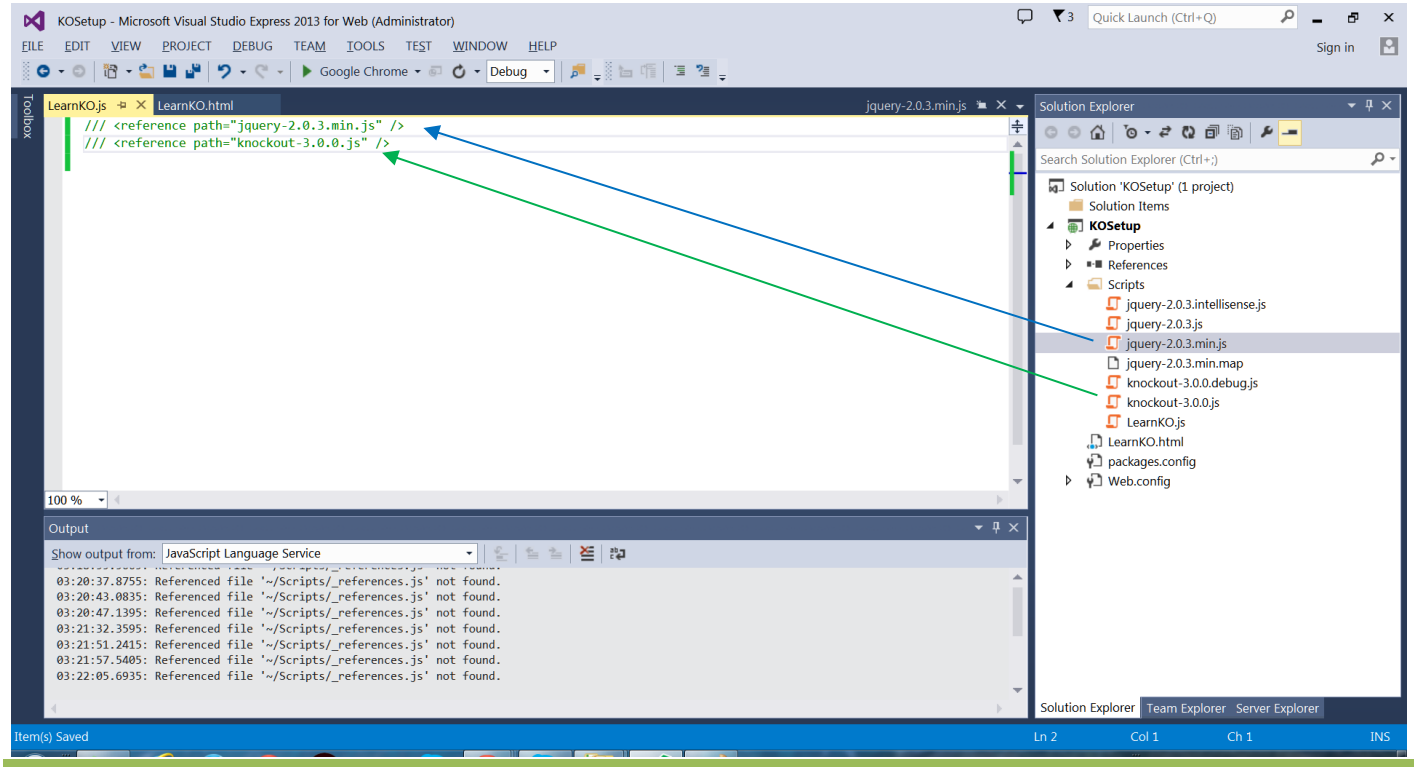
6. **Step6:** Now right click the project and add and aspx page,I named that page LearnKO.aspx.



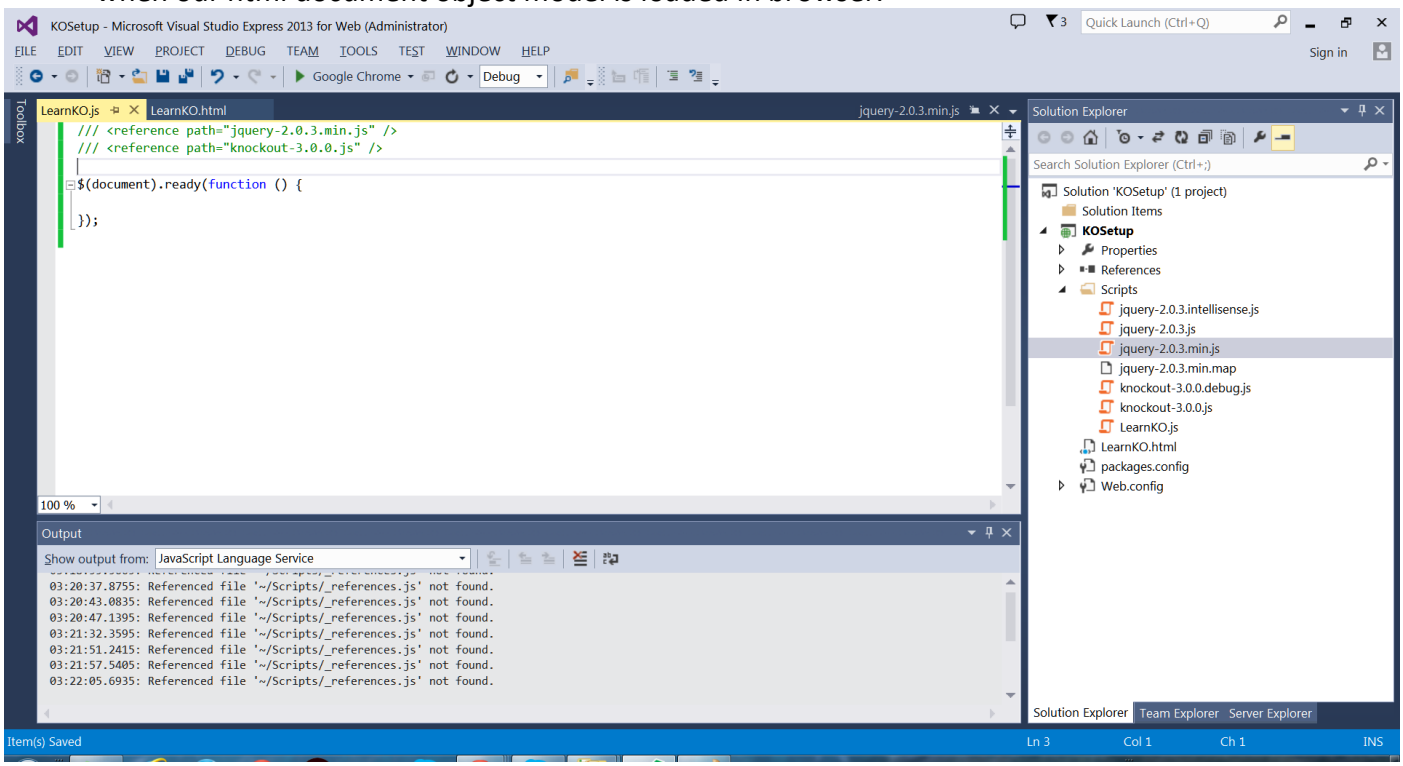
7. **Step7:** Similarly create a javascript file and add that to the project , I named that file as LearnKO.js



8. **Step8:** Open the learnKO.js file and drag the jQuery file and knockout.js library file to the LearKO.js file, we see in the below picture that reference of both the files is created on the js file.We did this because, it will proved us intellisense support for jQuery and knockout on our LearnKO.js file.



9. **Step9:** Write document.ready fuction of jquery in our LearnKO.js file. Document.ready function is fired when our html document object model is loaded in browser.

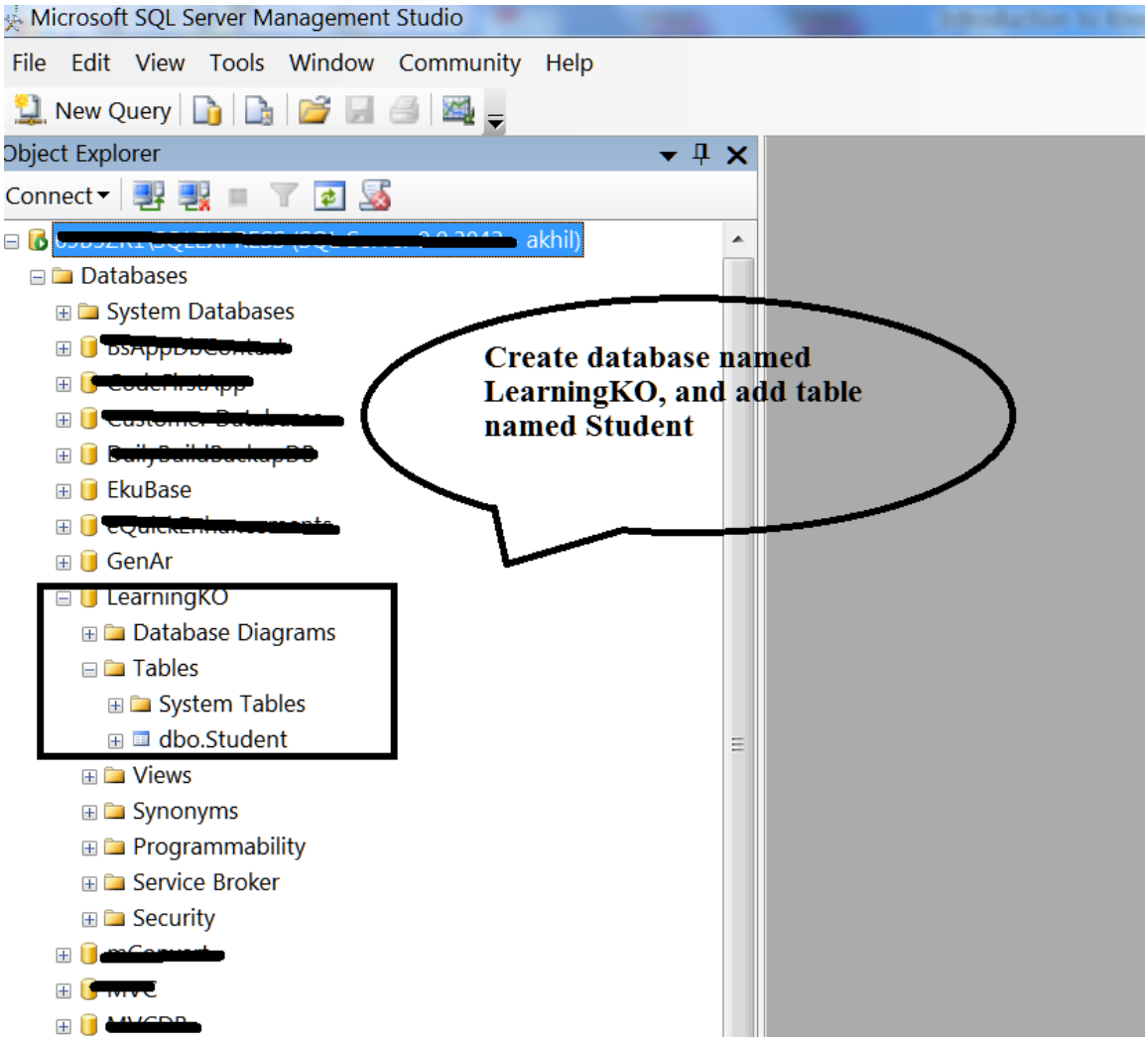


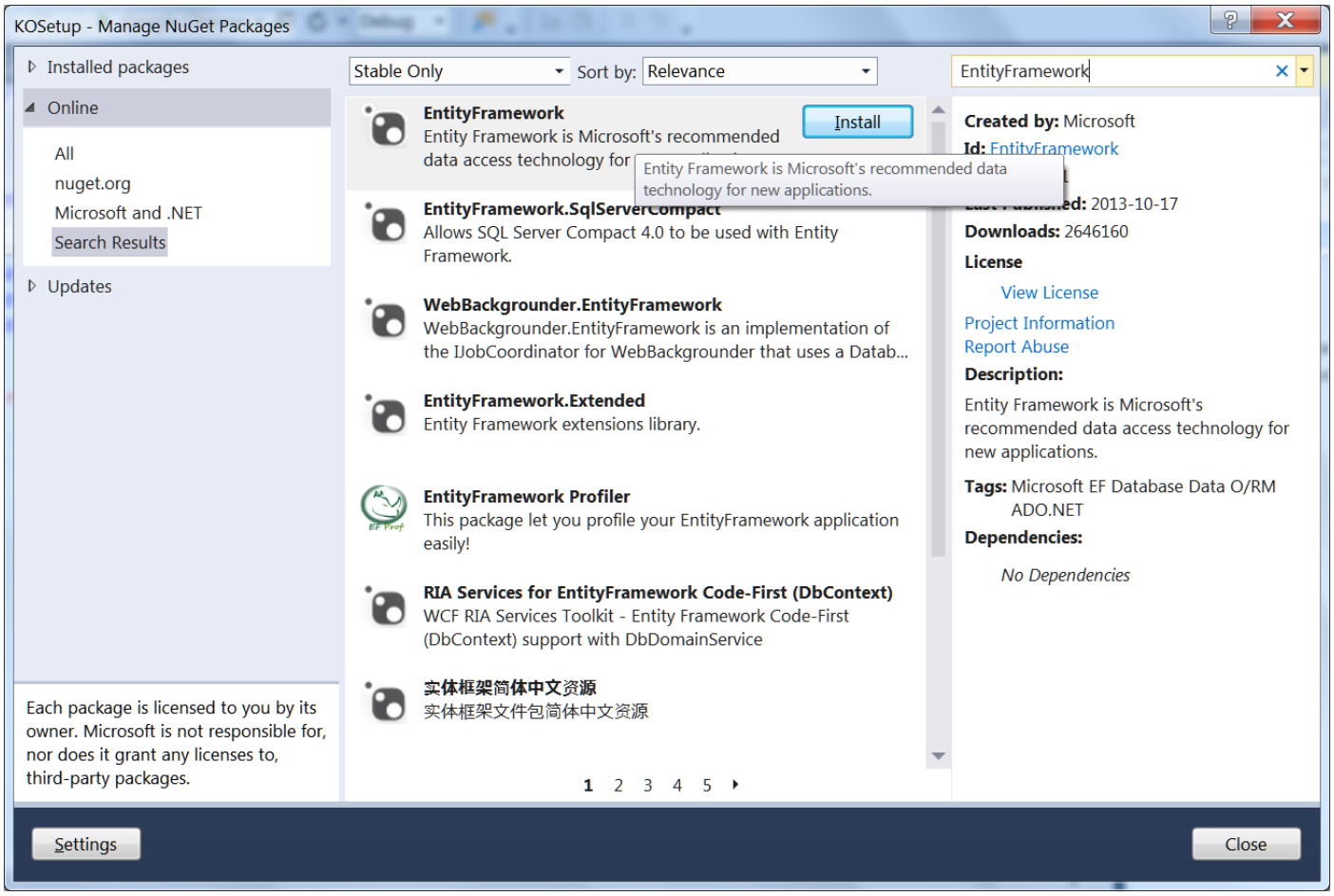
This is all we have to do to setup knockout, now we know how to setup initial environment to use knockout.js in our application.

We proceed now to create the application, talk to data base and create template and view model.

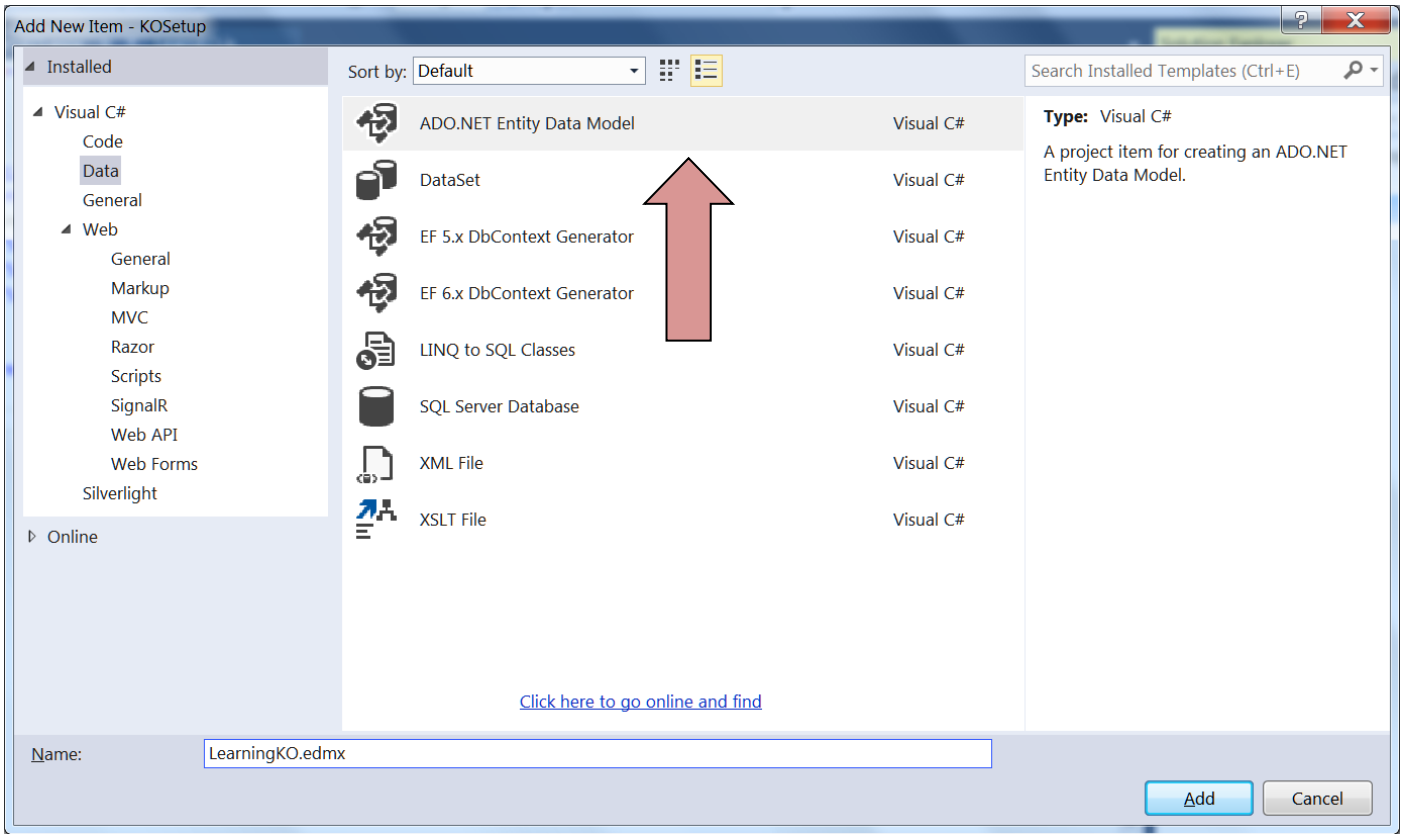
Creating Knockout application:

10. **Step10:** For communication to data base, add Entity Framework library in the same manner as we added JQuery and KO, installing Entity Framework library will add the EF dll to the project. We'll talk to data base using Entity Framework of Microsoft.Alternatively there are number of ways to talk to database like ADO.Net, LINQ to SQl etc. But first things first, create a data base you have to use in Sql Server.I've provided the script for the same,

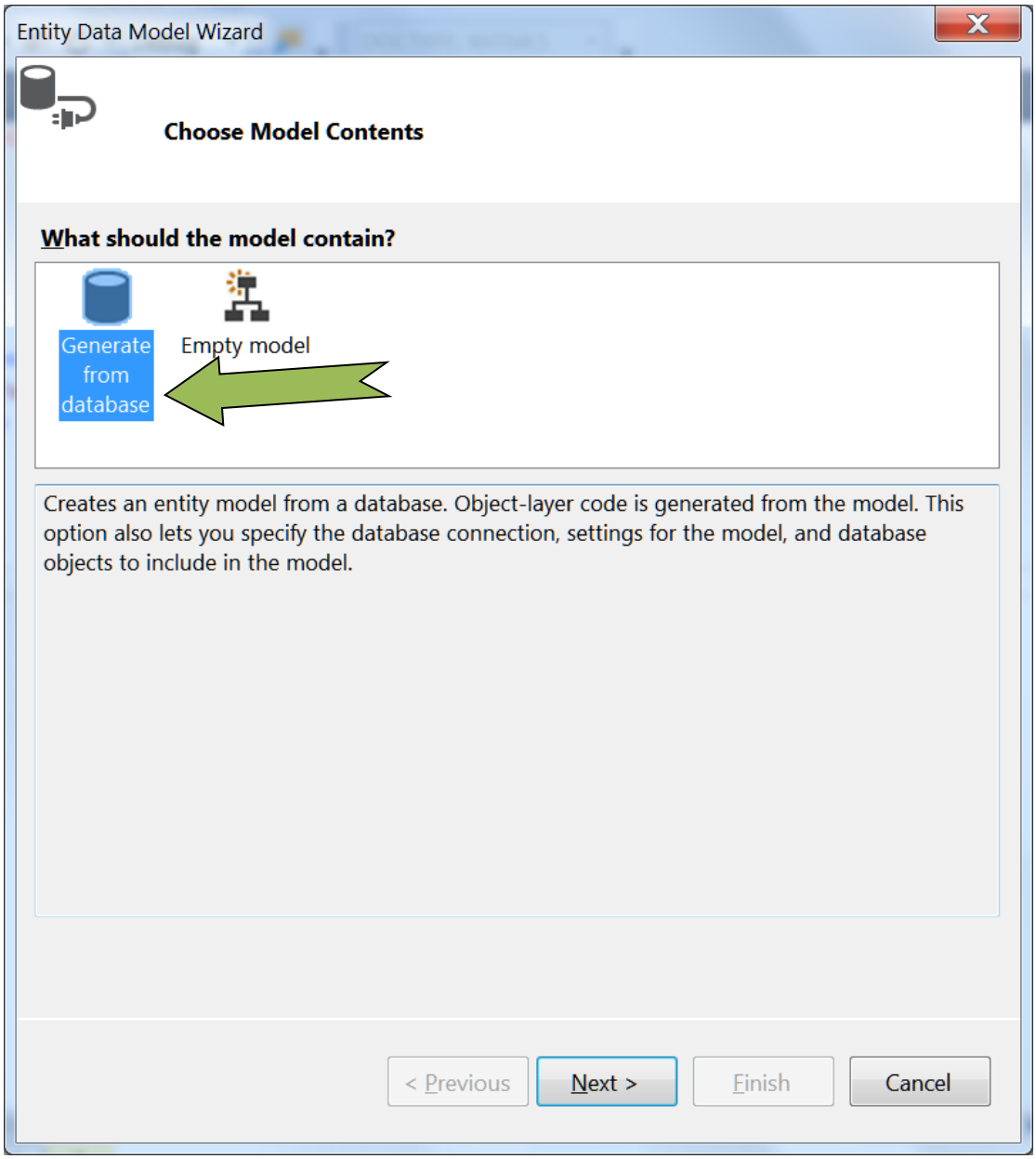

Microsoft SQL Server Management Studio
File Edit View Tools Window Community Help
New Query
Object Explorer
Connect
akhil)
Databases
System Databases
EkuBase
GenAr
LearningKO
Database Diagrams
Tables
System Tables
dbo.Student
Views
Synonyms
Programmability
Service Broker
Security
Create database named LearningKO, and add table named Student



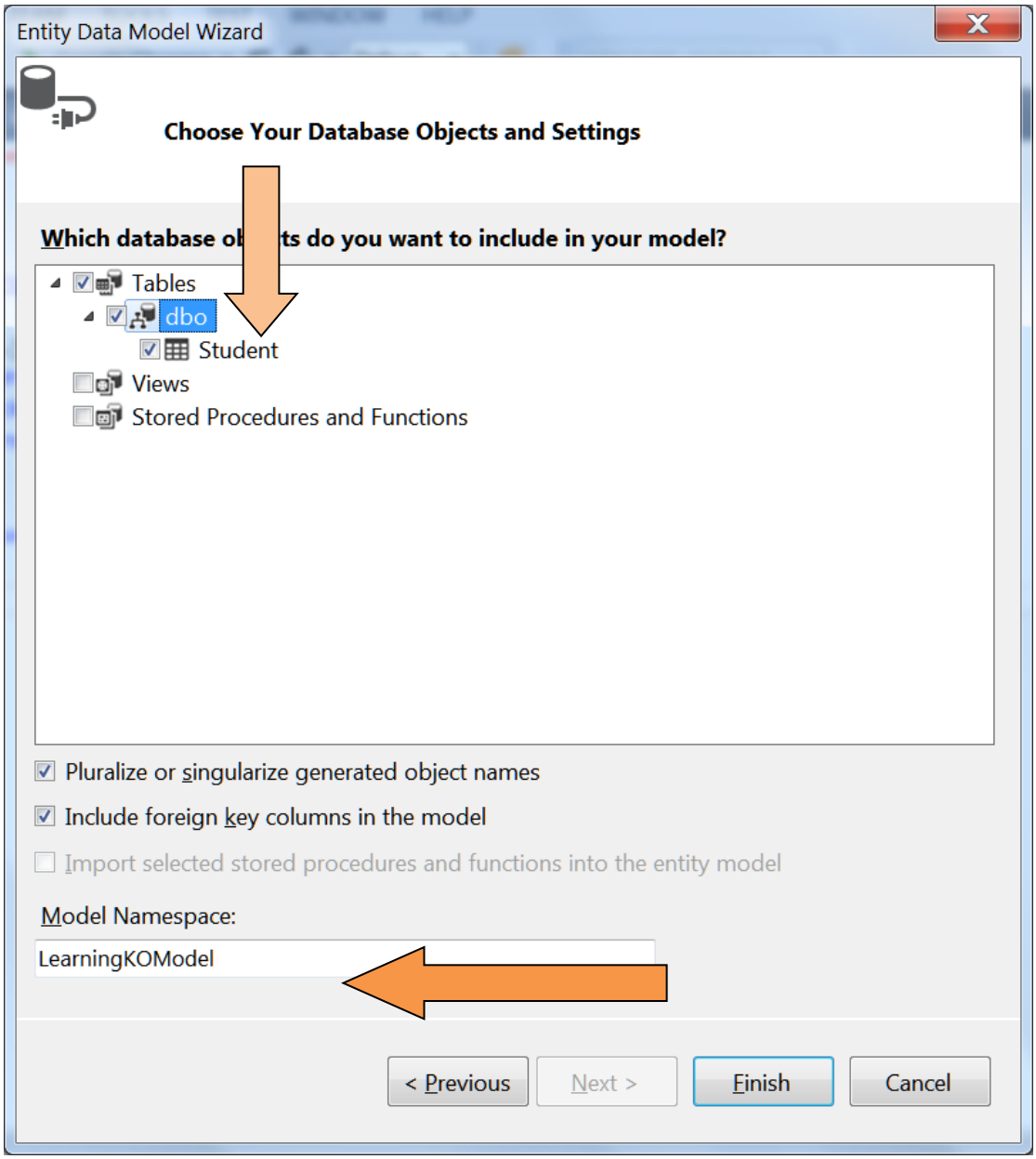
11. **Step11:** Right click project and add ADO.NET Entity data Model, click Add, and follow these below steps,



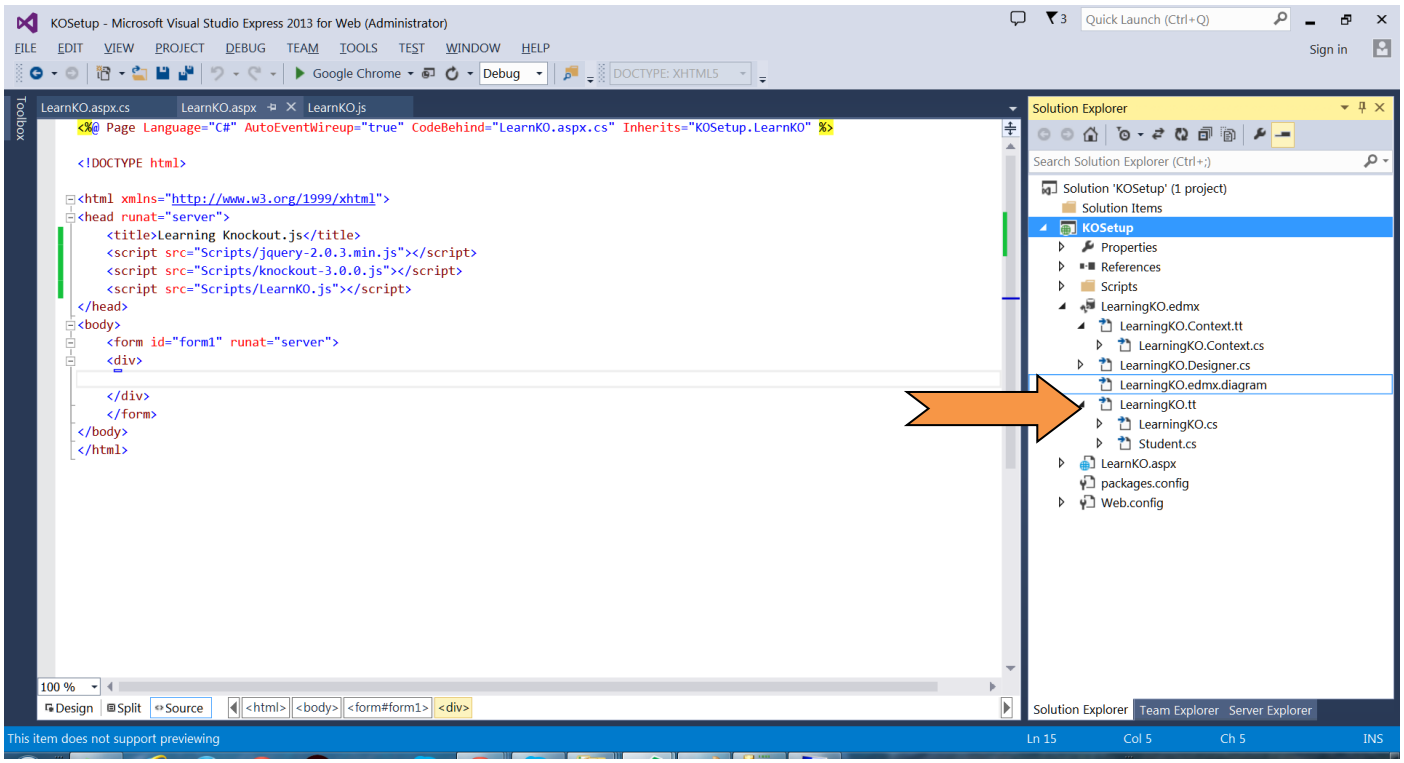
12. **Step12:** Following is step two of Entity Data Model, You can choose model contents from database you already created. So select “Generate From database” option.Click Next.



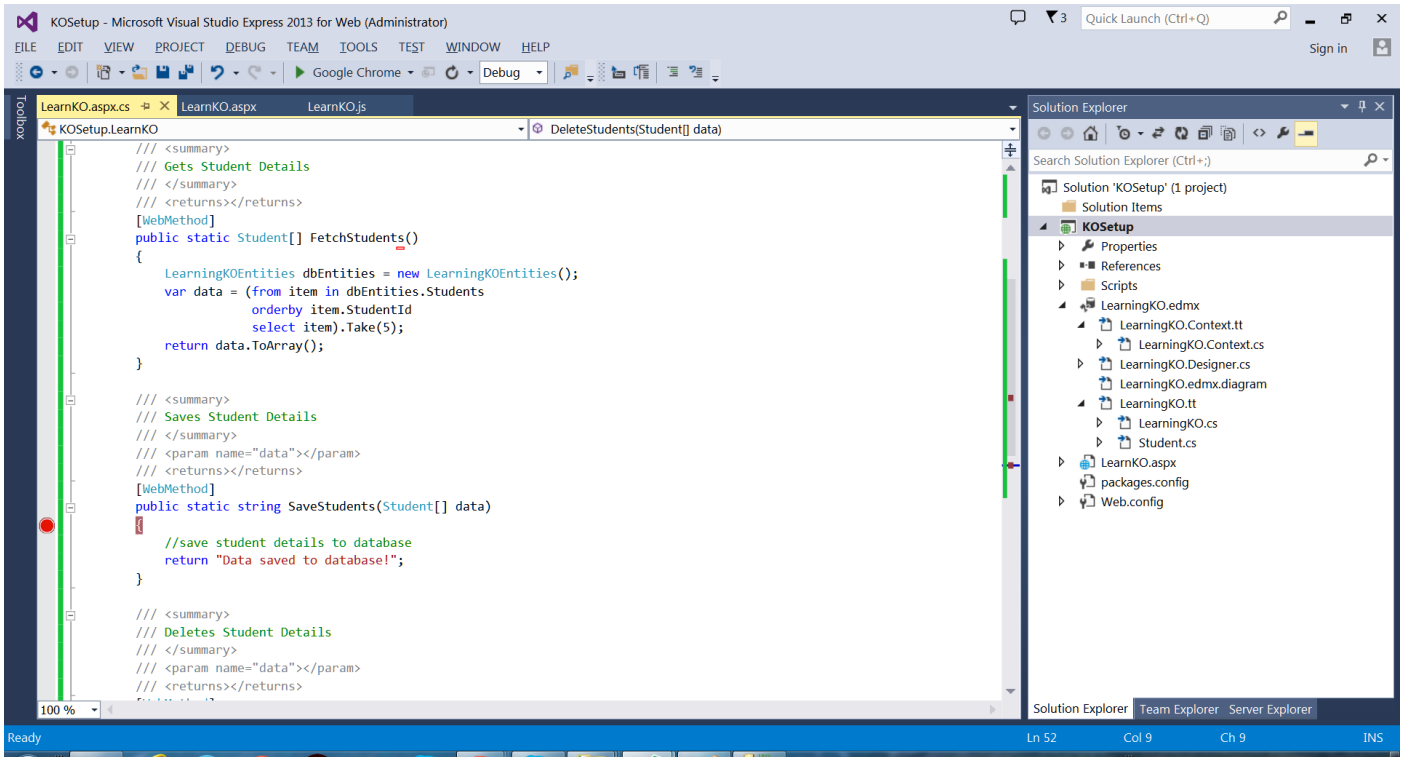
13. **Step13:** Choose the table you want to add i.e. Student table as shown below in the figure.Name the model as LearningKOModel.Click Finish.



14. **Step14:** We get certain files in our solution like context and tt files. We also get Student.CS file, that will act as our server side domain model.The context class contains the data communication methods of Entity Framework.



15. **Step15:** Write three methods with the help of Entity Framework in our aspx.cs page.One method to Fetch all the Students and another methods to save and delete a student to/from database, as shown below. Mark them as web method so that they could be called from client side.



Code:

```
#region Public Web Methods.
    /// <summary>
    /// Gets Student Details
    /// </summary>
    /// <returns></returns>
    [WebMethod]
    public static Student[] FetchStudents()
    {
      LearningKOEntities dbEntities = new LearningKOEntities();
      var data = (from item in dbEntities.Students
                  orderby item.StudentId
                  select item).Take(5);
      return data.ToArray();
    }

    /// <summary>
    /// Saves Student Details
    /// </summary>
    /// <param name="data"></param>
    /// <returns></returns>
    [WebMethod]
    public static string SaveStudent(Student[] data)
    {
      try
      {
        var dbContext = new LearningKOEntities();
        var studentList = from dbStududent in dbContext.Students select dbStududent;
        foreach (Student userDetails in data)
        {
          var student = new Student();
          if (userDetails != null)
          {
            student.StudentId = userDetails.StudentId;
            student.FirstName = userDetails.FirstName;
            student.LastName = userDetails.LastName;
            student.Address = userDetails.Address;
            student.Age = userDetails.Age;
            student.Gender = userDetails.Gender;
            student.Batch = userDetails.Batch;
            student.Class = userDetails.Class;
            student.School = userDetails.School;
            student.Domicile = userDetails.Domicile;
          }
          Student stud=(from st in studentList where st.StudentId==student.StudentId select st).FirstOrDefault();
          if (stud == null)
            dbContext.Students.Add(student);
          dbContext.SaveChanges();
        }
        return "Data saved to database!";
      }
      catch (Exception ex)
      {
        return "Error: " + ex.Message;
```

```
        }
    }

    /// <summary>
    /// Deletes Student Details
    /// </summary>
    /// <param name="data"></param>
    /// <returns></returns>
    [WebMethod]
    public static string DeleteStudent(Student data)
    {
        try
        {
            var dbContext = new LearningKOEntities();
            var student = dbContext.Students.FirstOrDefault(userId => userId.StudentId == data.StudentId);
            if (student != null)
            {
                if (student != null)
                {
                    dbContext.Students.Remove(student);
                    dbContext.SaveChanges();
                }
            }
            return "Data deleted from database!";

        }
        catch (Exception ex)
        {
            return "Error: " + ex.Message;
        }
    }
            #endregion
```

16. **Step16:** Open the aspx page we created and add following code to it, the code provides templates in html bound to model properties, one to add Student and other displaying Student List.

```
<table style="width:100%;" >
        <tbody>
            <tr>
                <th style="width:100px;">Property Name</th>
                <th style="width:100px;">Enter Value</th>
                <th style="width:100px;">Example of two Way Binding</th>
            </tr>
            </tbody>
        <tr>
            <td>Student ID (int):</td>
            <td>
                <input data-bind="value: StudentId" /></td> <!--,valueUpdate:'keypress'-->
            <td><span data-bind="text: StudentId" /></td>
        </tr>
        <tr>
            <td>First Name :</td>
            <td>
                <input data-bind="value: FirstName" /></td>
            <td  ><span data-bind="text: FirstName" /></td>
```

```
</tr>
<tr>
  <td>Last Name :</td>
  <td>
    <input data-bind="value: LastName" /></td>
  <td><span data-bind="text: LastName" /></td>
</tr>

<tr>
  <td>Student Age (int) :</td>
  <td>
    <input data-bind="value: Age" /></td>
  <td><span data-bind="text: Age" /></td>
</tr>
<tr>
  <td>Gender :</td>
  <td>
    <select data-bind="options: Genders, value: Gender, optionsCaption: 'Select Gender...'"></select></td>
  <td><span data-bind="text: Gender" /></td>
</tr>
<tr>
  <td>Batch :</td>
  <td>
    <input data-bind="value: Batch" /></td>
  <td><span data-bind="text: Batch" /></td>
</tr>
<tr>
  <td>Address :</td>
  <td>
    <input data-bind="value: Address" /></td>
  <td><span data-bind="text: Address" /></td>
</tr>
<tr>
  <td>Class :</td>
  <td>
    <input data-bind="value: Class" /></td>
  <td><span data-bind="text: Class" /></td>
</tr>
<tr>
  <td>School :</td>
  <td>
    <input data-bind="value: School" /></td>
  <td><span data-bind="text: School" /></td>
</tr>
<tr>
  <td>Domicile :</td>
  <td>
    <select data-bind="options: Domiciles, value: Domicile, optionsCaption: 'Select Domicile...'"></select>
  </td>
  <td><span data-bind="text: Domicile" /></td>
</tr>
<tr>
  <td colspan="3">
    <button type="button" data-bind="click: AddStudent">Add Student</button>
    <button type="button" data-bind="click: SaveStudent">Save Student To Database</button>
  </td>
</tr>
```

```
</table>
  </div>

 <div style="width:70%;float:left;display:inline-block;">
   <h2>List of Students</h2>
<table style="width:100%;" data-bind="visible: Students().length > 0" border="0">
  <tr>
    <th>Student Id</th>
    <th>First Name</th>
    <th>Last Name</th>
    <th>Age</th>
    <th>Gender</th>
    <th>Batch</th>
    <th>Address</th>
    <th>Class</th>
    <th>School</th>
    <th>Domicile</th>
  </tr>
  <tbody data-bind="foreach: Students">
    <tr>
      <td><span data-bind="text: StudentId" /></td>
      <td>
        <input data-bind="value: FirstName" /></td>
      <td>
        <input data-bind="value: LastName" /></td>
      <td>
        <input data-bind="value: Age" /></td>

      <td>
        <select data-bind="options: $root.Genders, value: Gender"></select></td>
      <td>
        <input data-bind="value: Batch" /></td>
      <td>
        <input data-bind="value: Address" /></td>
      <td>
        <input data-bind="value: Class" /></td>
      <td>
        <input data-bind="value: School" /></td>
      <td>
        <select data-bind="options: $root.Domiciles, value: Domicile"></select></td>

      <td><a href="#" data-bind="click: $root.DeleteStudent">Delete</a></td>
    </tr>
  </tbody>
</table>
```

There are two html tables one for adding student to database and other showing all the students having delete anchor link to delete the student, these template properties will be bound in view model, where we write method to communicate with data base and call Web Methods we created in aspx.cs page.The Viewmodel also contains observables to be bound to these properties.

17. **Step17:** Now it's time to create ViewModel, Open the learnKO.js file and add codes to fetch, save and delete student, and observables bound to properties binded on controls of html page.

```
/// <reference path="jquery-2.0.3.min.js" />
/// <reference path="knockout-3.0.0.js" />



function Student(data) {
    this.StudentId = ko.observable(data.StudentId);
    this.FirstName = ko.observable(data.FirstName);
    this.LastName = ko.observable(data.LastName);
    this.Age = ko.observable(data.Age);
    this.Gender = ko.observable(data.Gender);
    this.Batch = ko.observable(data.Batch);
    this.Address = ko.observable(data.Address);
    this.Class = ko.observable(data.Class);
    this.School = ko.observable(data.School);
    this.Domicile = ko.observable(data.Domicile);

}


function StudentViewModel() {
    var self = this;
    self.Domiciles = ko.observableArray(['Delhi', 'Outside Delhi']);
    self.Genders = ko.observableArray(['Male', 'Female']);
    self.Students = ko.observableArray([]);
    self.StudentId = ko.observable();
    self.FirstName = ko.observable();
    self.LastName = ko.observable();
    self.Age = ko.observable();
    self.Batch = ko.observable();
    self.Address = ko.observable();
    self.Class = ko.observable();
    self.School = ko.observable();
    self.Domicile = ko.observable();
    self.Gender = ko.observable();


    self.AddStudent = function () {
        self.Students.push(new Student({
            StudentId: self.StudentId(),
            FirstName: self.FirstName(),
            LastName: self.LastName(),
            Domicile: self.Domicile(),
            Age: self.Age(),
            Batch: self.Batch(),
            Address: self.Address(),
            Class: self.Class(),
            School: self.School(),
            Gender: self.Gender()
        }));
         self.StudentId(""),
         self.FirstName(""),
         self.LastName(""),
         self.Domicile(""),
         self.Age(""),
         self.Batch(""),
         self.Address(""),
```

```
        self.Class(""),
        self.School(""),
       self.Gender("")
    };

    self.DeleteStudent = function (student) {

        $.ajax({
            type: "POST",
            url: 'LearnKO.aspx/DeleteStudent',
            data: ko.toJSON({ data: student }),
            contentType: "application/JSON; charset=utf-8",
            success: function (result) {
                alert(result.d);
                self.Students.remove(student)
            },
            error: function (err) {
                alert(err.status + " - " + err.statusText);
            }
        });

    };

    self.SaveStudent = function () {
        $.ajax({
            type: "POST",
            url: 'LearnKO.aspx/SaveStudent',
            data: ko.toJSON({ data: self.Students }),
            contentType: "application/JSON; charset=utf-8",
            success: function (result) {
                alert(result.d);
            },
            error: function (err) {
            alert(err.status + " - " + err.statusText);
        }
        });
    };

    $.ajax({
        type: "POST",
        url: 'LearnKO.aspx/FetchStudents',
        contentType: "application/JSON; charset=utf-8",
        dataType: "JSON",
        success: function (results) {
            var students = $.map(results.d, function (item) {
                return new Student(item)
            });
            self.Students(students);
        },
        error: function (err) {
            alert(err.status + " - " + err.statusText);
        }
    })
}

$(document).ready(function () {
    ko.applyBindings(new StudentViewModel());
```

```
});
```

We create StudentViewModel() as our primary view model javascript function, that contains all the business logic and operations.
We bind this View model on document ready function by the KO method named **applyBindings.**This initializes our view model, `ko.applyBindings(new StudentViewModel());`
Function `function Student(data)` contains observables bound to model properties.
We can create observable arrays of Domiciles and genders to bind to the dropdown list of our html.Ko provides these observables and other such properties to bind to model.

. observable: Used to define model/entity properties. If these properties are bound with user interface and when value for these properties gets updated, automatically the UI elements bound with these properties will be updated with the new value instantaneously.

E.g. this.StudentId = ko.observable("1"); - => StudentId is the observable property. KO represent an object for the Knockout.js library.

The value of the observable is read as var id= this. StudentId ();

· **observableArray:** observableArray represents a collection of data elements which required notifications. It's used to bind with the List kind of elements.

E.g this.Students = ko.observableArray([]);

· **applyBindings:** This is used to activate knockout for the current HTML document or a specific UI element in HTML document. The parameter for this method is the view-model which is defined in JavaScript. This ViewModel contains the observable, observableArray and various methods.

Various other types of binding are used in this chapter:

o click: Represents a click event handler added to the UI element so that JavaScript function is called.

o value: This represents the value binding with the UI element's value property to the property defined into the ViewModel.

The value binding should be used with <input> , <select> , <textarea>

o visible: This is used to hide or unhide the UI element based upon the value passed to it's binding.

o Text: This represent the text value of the parameter passed to the UI element.

18. **Step18:** Include js files and stylesheet(you can create your own stylesheet file) to head section of aspx page,

```
<head runat="server">
    <title>Learning Knockout.js</title>
    <script src="Scripts/jquery-2.0.3.min.js"></script>
    <script src="Scripts/knockout-3.0.0.js"></script>
    <script src="Scripts/LearnKO.js"></script>
    <link href="Styles/Style.css" rel="stylesheet" />
</head>
```

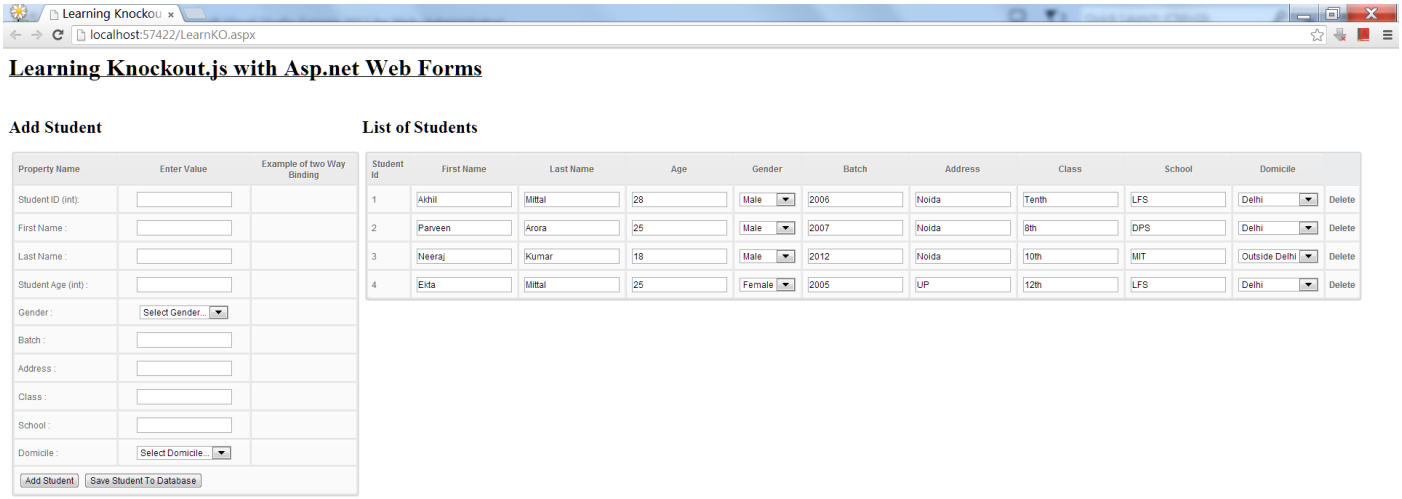
Include jQuery File

Include Knockout.js file

Include LearKO.js file

Include custom Stylesheet

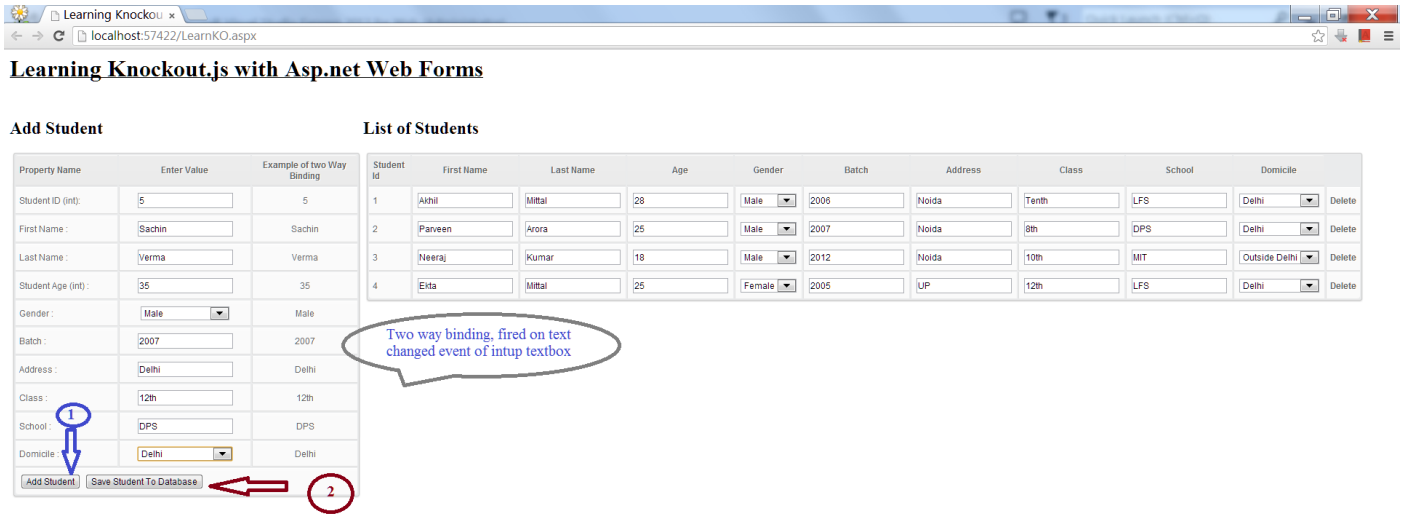
19. **Step19:** Press F5 to run the application, and we'll be shown a page having html controls as follows,



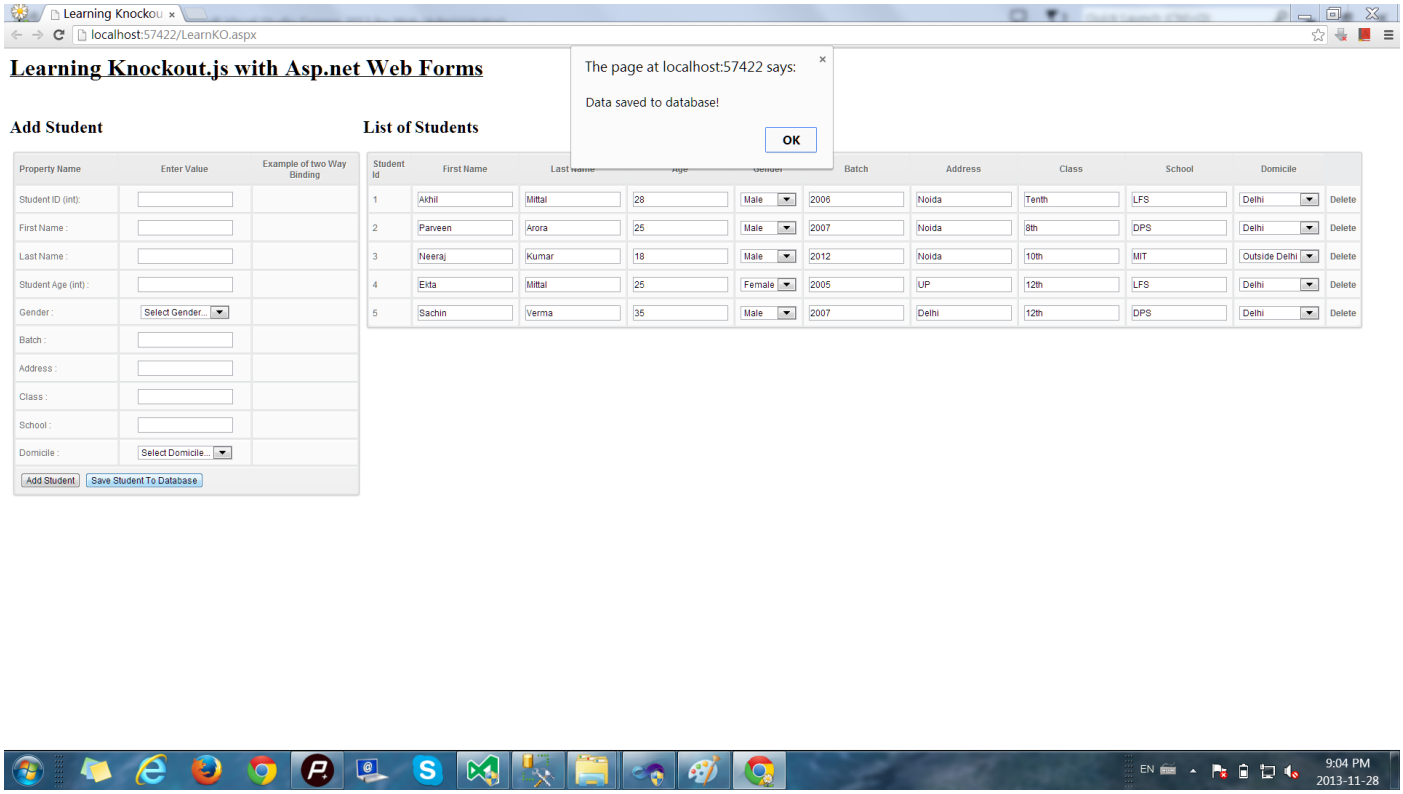
List of Student Shows the list of students from database, bound to viewmodel's event.
Try to create a new student and add student then save it to database, it will automatically be added to the right hand side list.
We can see that Domicile and Genders drop down lists are bound to our Viewmodel's properties.

Note: Do not give string in StudentId and Age as no validation is put on those fields, code may break.

Create Student:



Student added to list and database:



Job Done !

Now you can say that you have become a knockout.js developer.

Conclusion:

We learnt a lot in this chapter about how to set up knockout.js in visual studio, lots of theory and also created a sample application just to hands-on the concept. In the next chapter I'll explain creating a sample application and performing CRUD operations in MVC4 with knockout js and Entity Framework, now pat your back to have done a great job by learning a new concept.

Complete end to end CRUD operations using Knockout.JS and EntityFramework 5 in MVC application

)

Introduction:

This chapter will be a kind of tutorial to explain how we can set up knockout.js environment in MVC4 application and perform CRUD operations on the same.

MVC:

Model: The business entity on which the overall application operates. Many applications use a persistent storage mechanism (such as a database) to store data. MVC does not specifically mention the data access layer because it is understood to be encapsulated by the Model.

View: The user interface that renders the model into a form of interaction.

Controller: Handles a request from a view and updates the model that results a change in Model's state.

To implement MVC in .NET we need mainly three classes (View, Controller and the Model).

Entity Framework:

Let's have a look on standard definition of Entity Framework given by Microsoft:

"The Microsoft ADO.NET Entity Framework is an Object/Relational Mapping (ORM) framework that enables developers to work with relational data as domain-specific objects, eliminating the need for most of the data access plumbing code that developers usually need to write. Using the Entity Framework, developers issue queries using LINQ, then retrieve and manipulate data as strongly typed objects. The Entity Framework's ORM implementation provides services like change tracking, identity resolution, lazy loading, and query translation so that developers can focus on their application-specific business logic rather than the data access fundamentals."

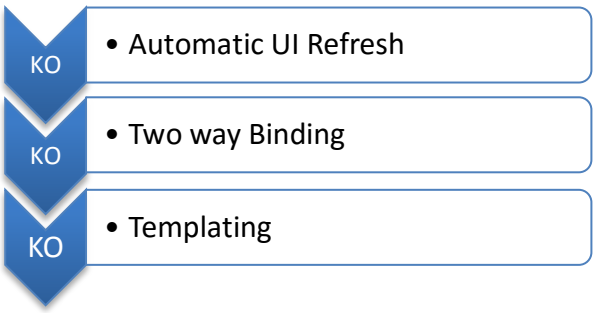
In a simple language, Entity framework is an Object/Relational Mapping (ORM) framework. It is an enhancement to ADO.NET, an upper layer to ADO.Net that gives developers an automated mechanism for accessing & storing the data in the database.
Hope this gives a glimpse of an ORM and EntityFramework.

Knockout.JS:

Knockout.JS (KO) is basically a JS library that enables Declarative Bindings using an 'Observable' View Model on the client (browser) following observer pattern approach, enabling UI to bind and refresh itself automatically whenever the data bound is modified. Knockout.JS provides its own templating pattern that helps us to bind our view model data easily. KO works on MVVM pattern i.e. Model-View-View Model.

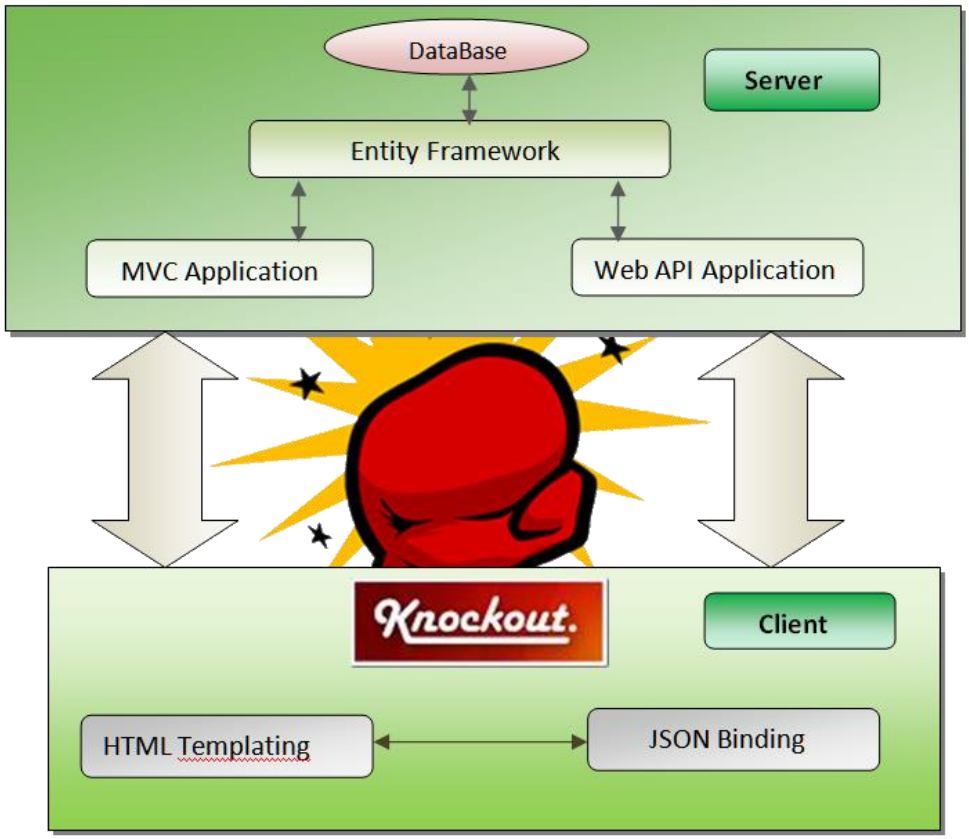
As the architecture is shown, Views interact with View Models in a two way binding manner i.e. when model is changed view updates itself and when view is updated, model too updates itself instantaneously.

KO provides 3 most important features like,



The whole idea of KO derives from these three major functionalities. KO also helps in developing Single page applications (SPA's).SPA's are out of the box new way of developing rich internet applications(RIA's) in todays era.

Application Architecture:



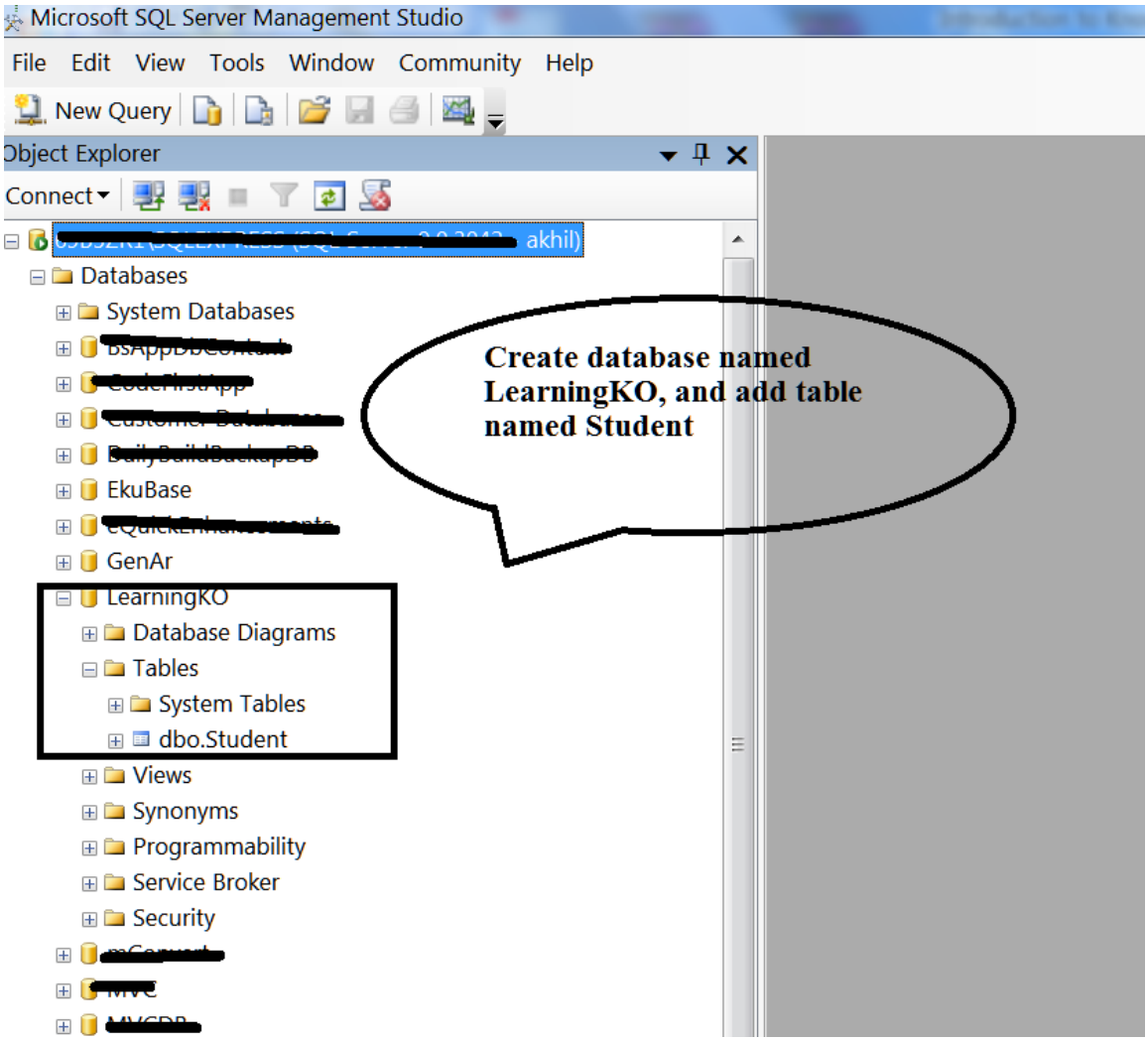
The architecture is very much self explanatory. The application works on client-server model, where our MVC application or Web API application (not covered in this tutorial) will interact with EntityFramework layer on server side. Entity Framework layer will be responsible for data transactions with data base.
On client side we have HTML templates which will communicate with server through Ajax calls and the templates will be bind to data via JSON objects through knockout observables (already discussed in first part).

MVC Application:

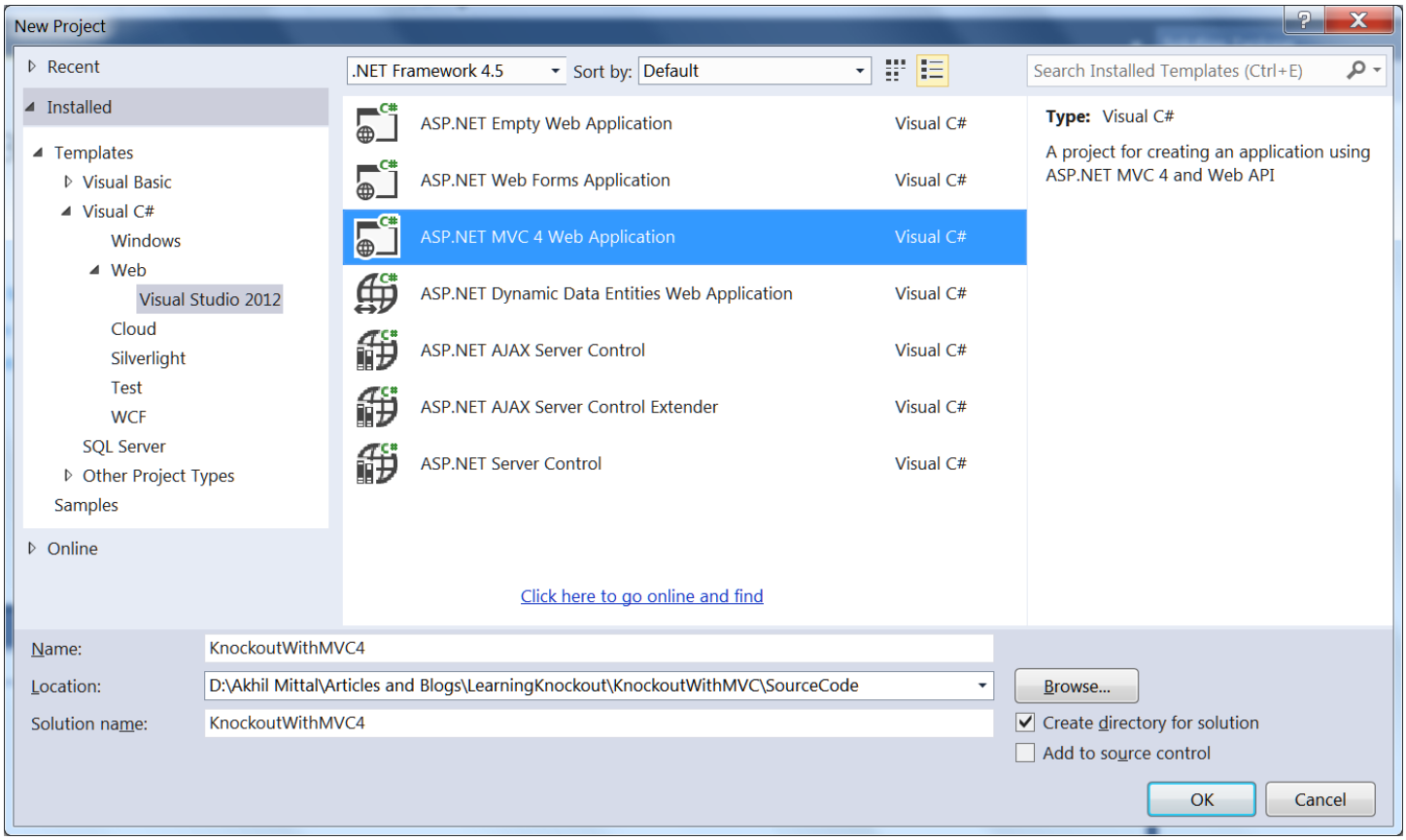
1. **Step1:** Create a data base named LearningKO and add a table named student to it, script of the table is as follows,

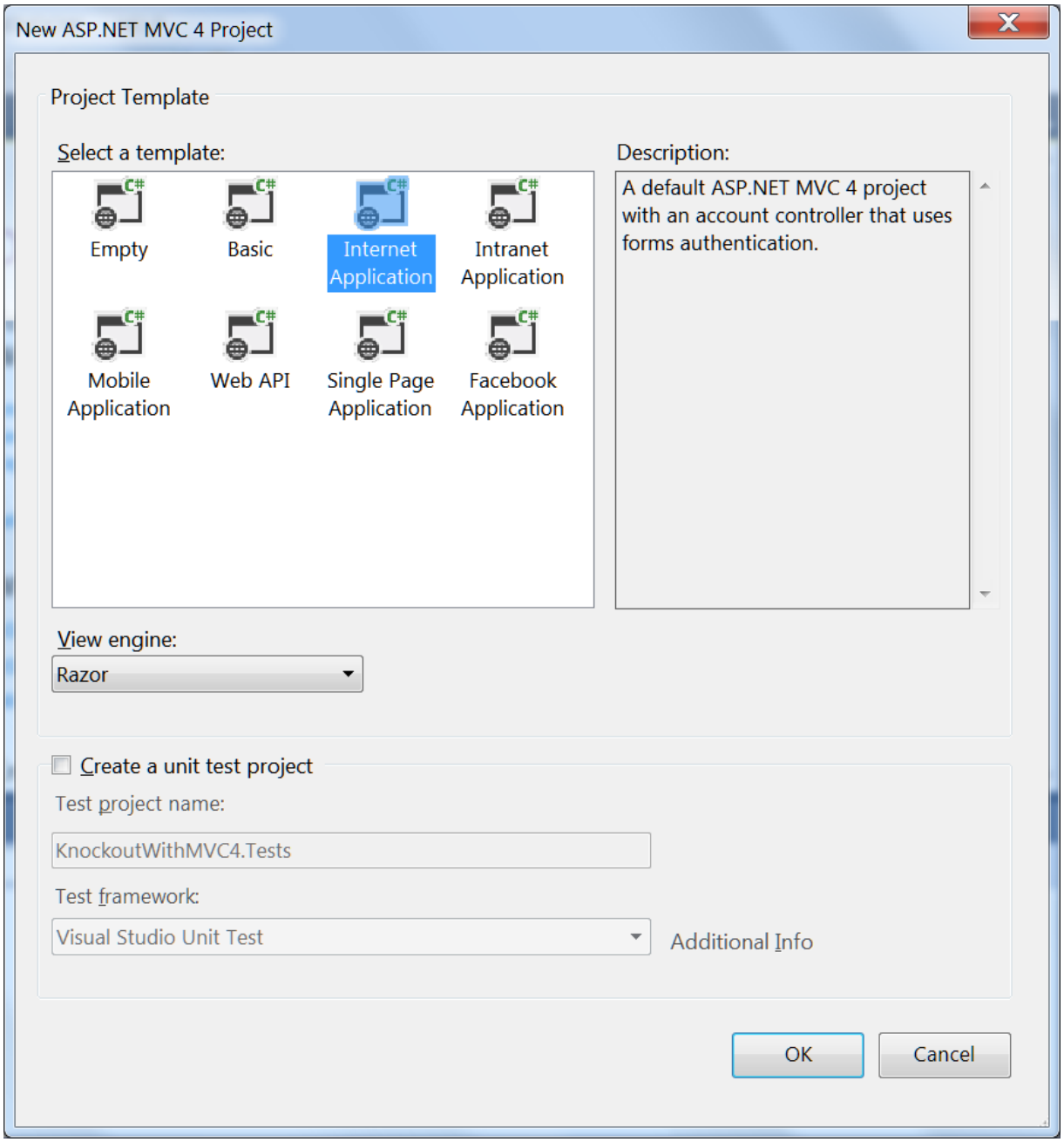
```
USE [LearningKO]
GO
/****** Object:  Table [dbo].[Student]    Script Date: 12/04/2013 23:58:12 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Student](
```

```
	[StudentId] [nvarchar](10) NOT NULL,
	[FirstName] [nvarchar](50) NULL,
	[LastName] [nvarchar](50) NULL,
	[Age] [int] NULL,
	[Gender] [nvarchar](50) NULL,
	[Batch] [nvarchar](50) NULL,
	[Address] [nvarchar](50) NULL,
	[Class] [nvarchar](50) NULL,
	[School] [nvarchar](50) NULL,
	[Domicile] [nvarchar](50) NULL,
 CONSTRAINT [PK_Student] PRIMARY KEY CLUSTERED
(
	[StudentId] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON,
ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
) ON [PRIMARY]
GO
INSERT [dbo].[Student] ([StudentId], [FirstName], [LastName], [Age], [Gender], [Batch], [Address], [Class], [School], [Domicile])
VALUES (N'1', N'Akhil', N'Mittal', 28, N'Male', N'2006', N'Noida', N'Tenth', N'LFS', N'Delhi')
INSERT [dbo].[Student] ([StudentId], [FirstName], [LastName], [Age], [Gender], [Batch], [Address], [Class], [School], [Domicile])
VALUES (N'2', N'Parveen', N'Arora', 25, N'Male', N'2007', N'Noida', N'8th', N'DPS', N'Delhi')
INSERT [dbo].[Student] ([StudentId], [FirstName], [LastName], [Age], [Gender], [Batch], [Address], [Class], [School], [Domicile])
VALUES (N'3', N'Neeraj', N'Kumar', 38, N'Male', N'2011', N'Noida', N'10th', N'MIT', N'Outside Delhi')
INSERT [dbo].[Student] ([StudentId], [FirstName], [LastName], [Age], [Gender], [Batch], [Address], [Class], [School], [Domicile])
VALUES (N'4', N'Ekta', N'Mittal', 25, N'Female', N'2005', N' Noida', N'12th', N'LFS', N'Delhi')
```



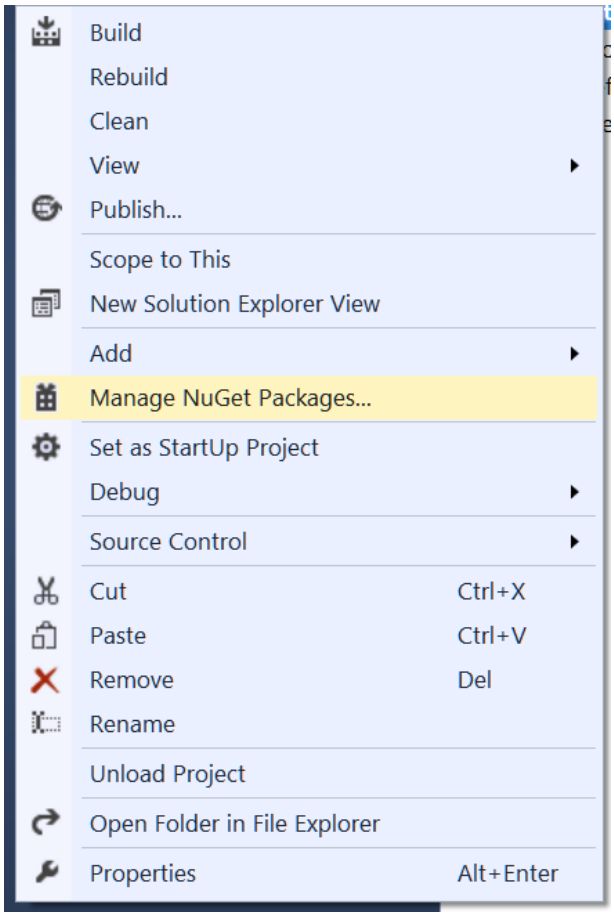
2. **Step2:** Open your Visual Studio (Visual Studio Version should be greater than or equal to 12) and add an MVC Internet application,

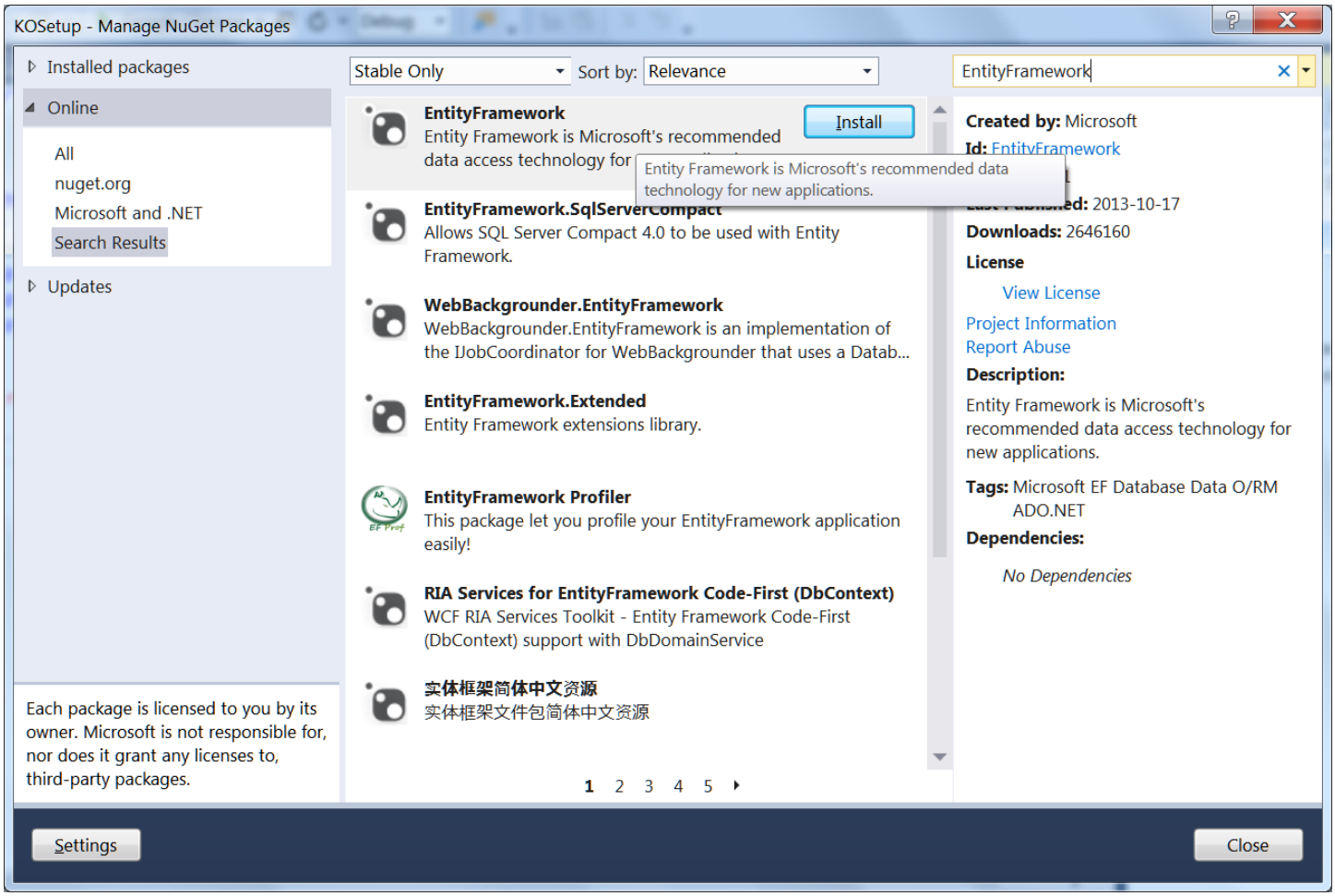




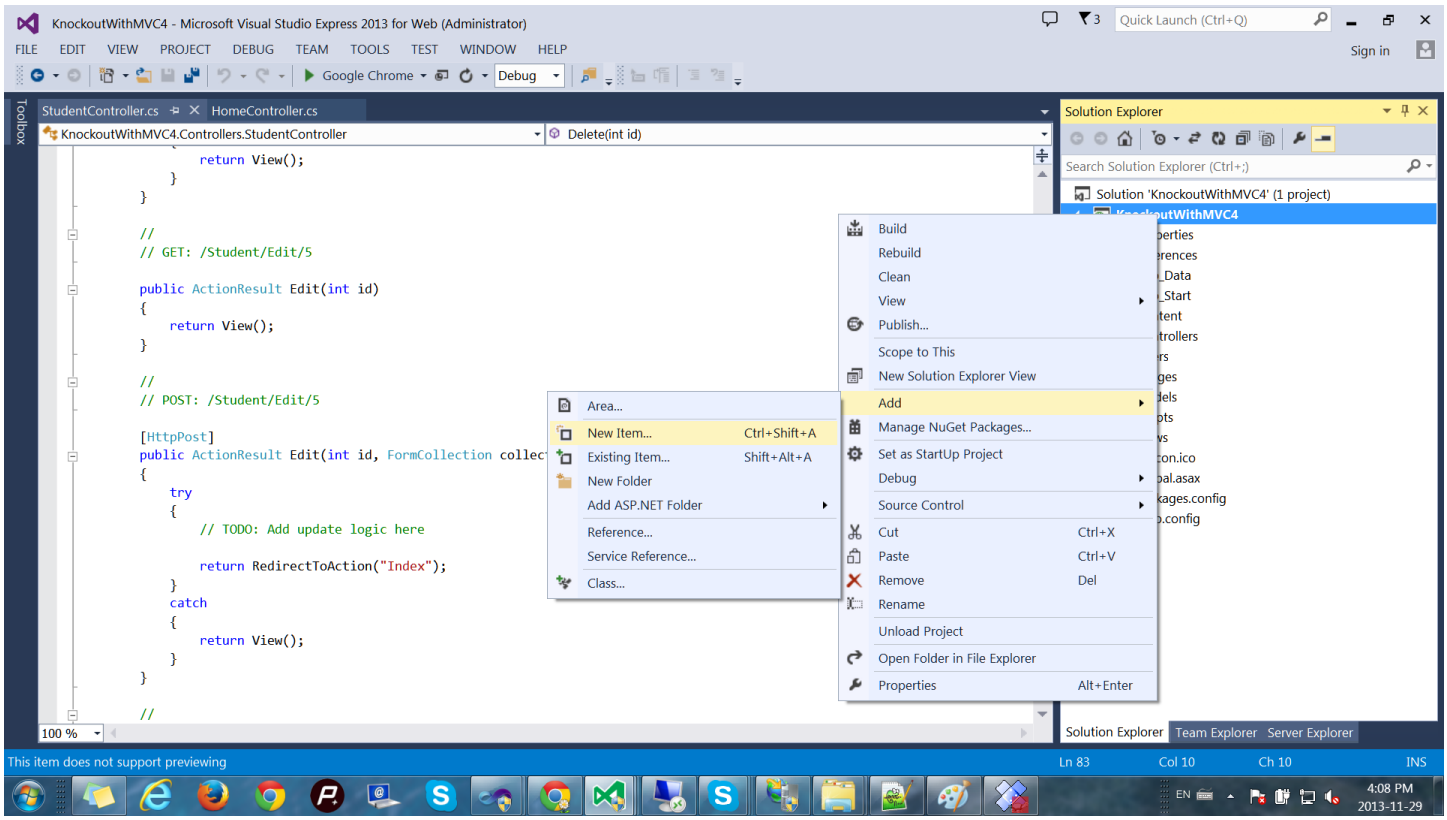
I have given it a name "KnockoutWithMVC4".

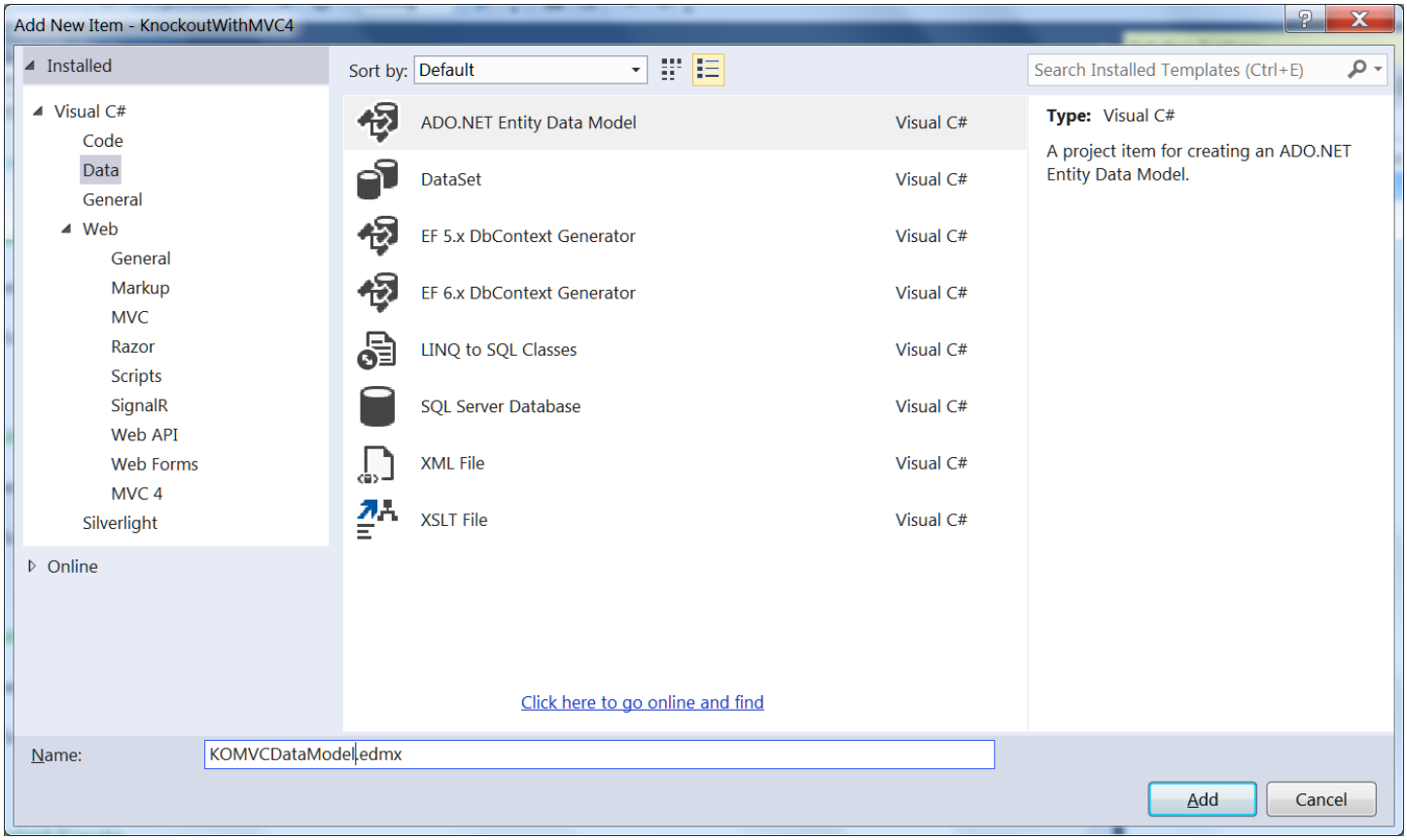
3. **Step3:** You'll get a full structured MVC application with default Home controller in the Controller folder. By default entity framework is downloaded as a package inside application folder but if not, you can add entity framework package by right click the project, select manage nugget packages and search and install Entity Framework,

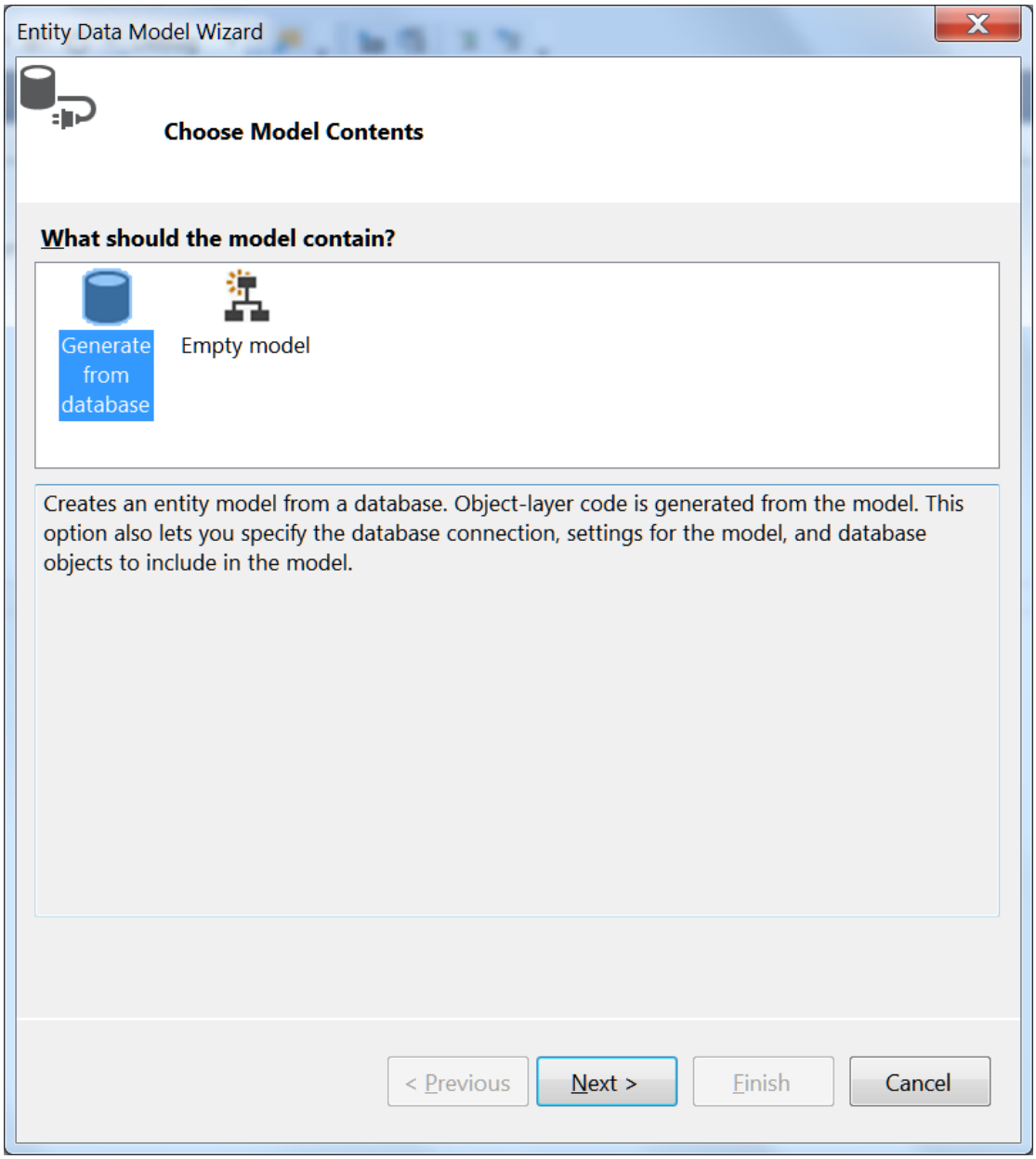




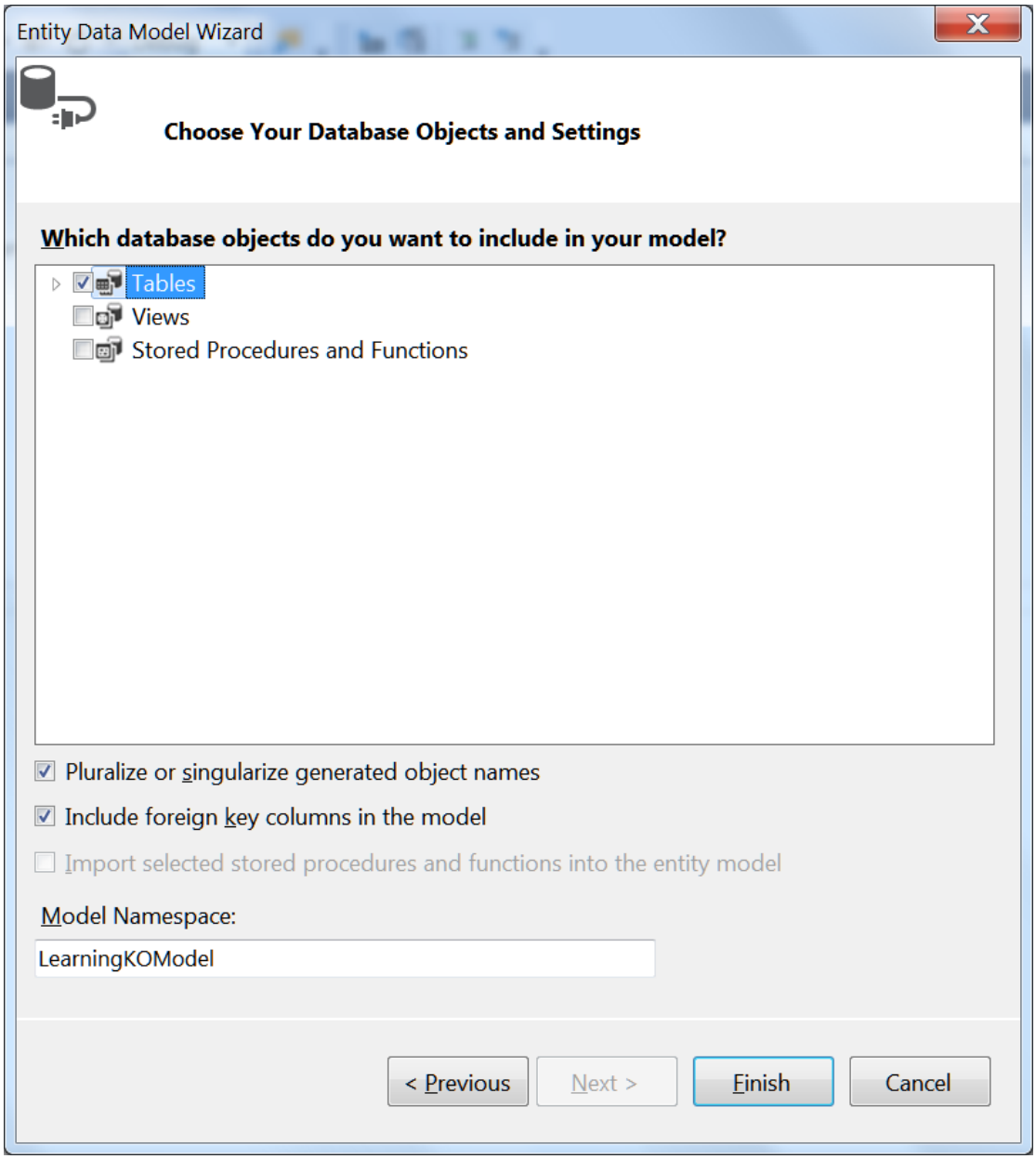
4. **Step 4:** Right click project file, select add new item and add ADO.net entity data model, follow the steps in the wizard as shown below,



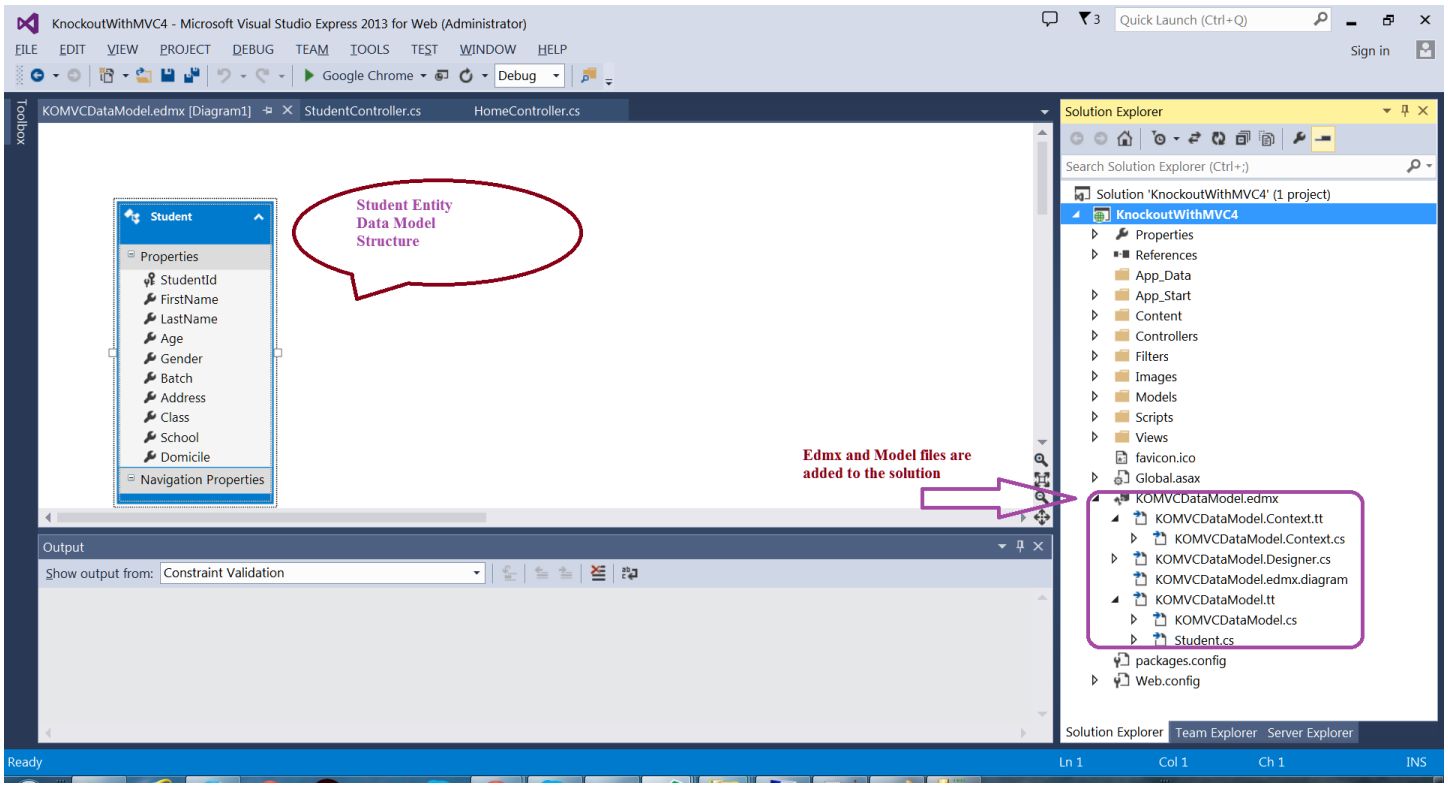




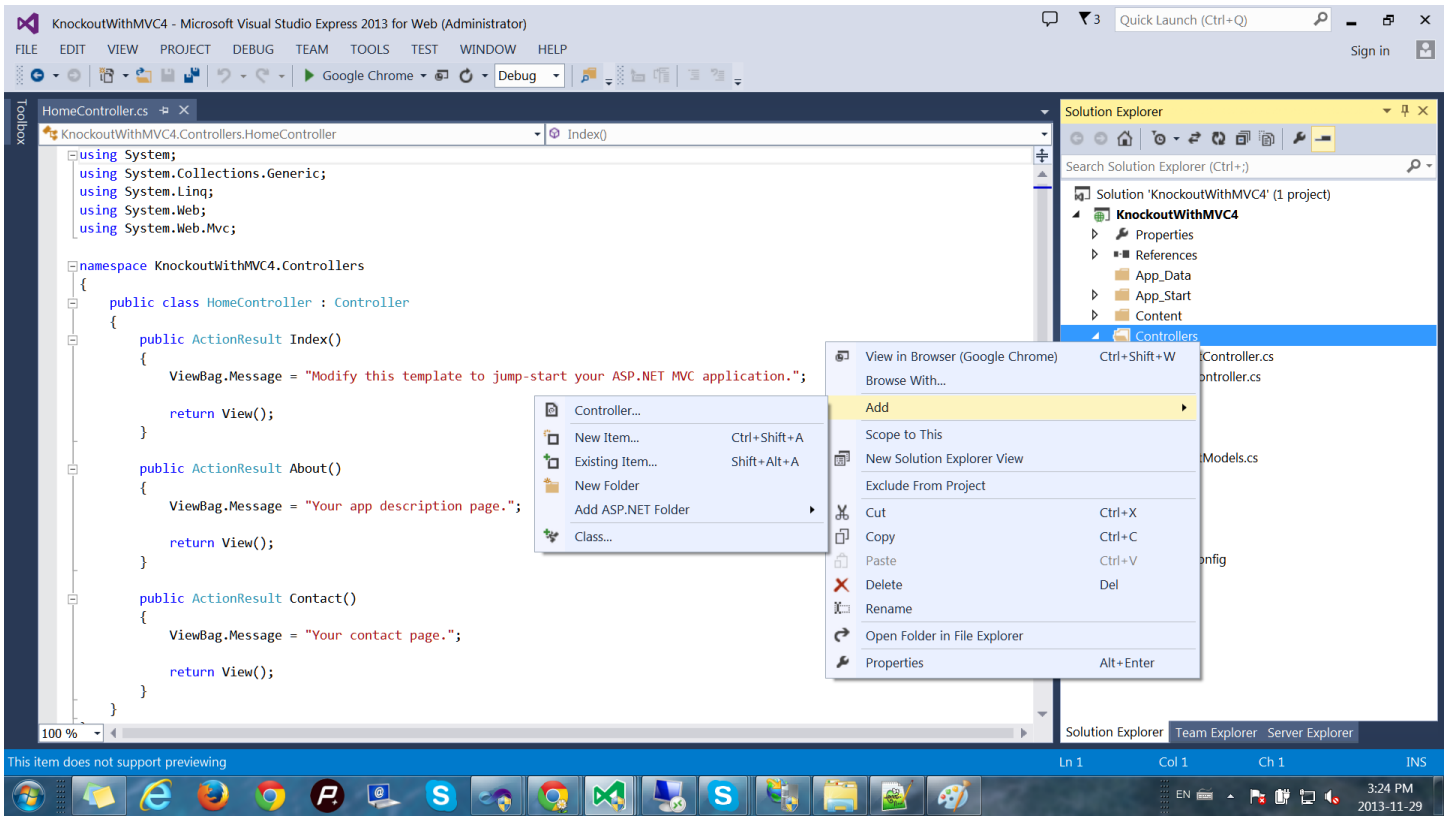
Generate model from data base, select your server and LearningKO data base name, the connection string will automatically be added to your Web.Config, name that connection string as **LearningKOEntities.**

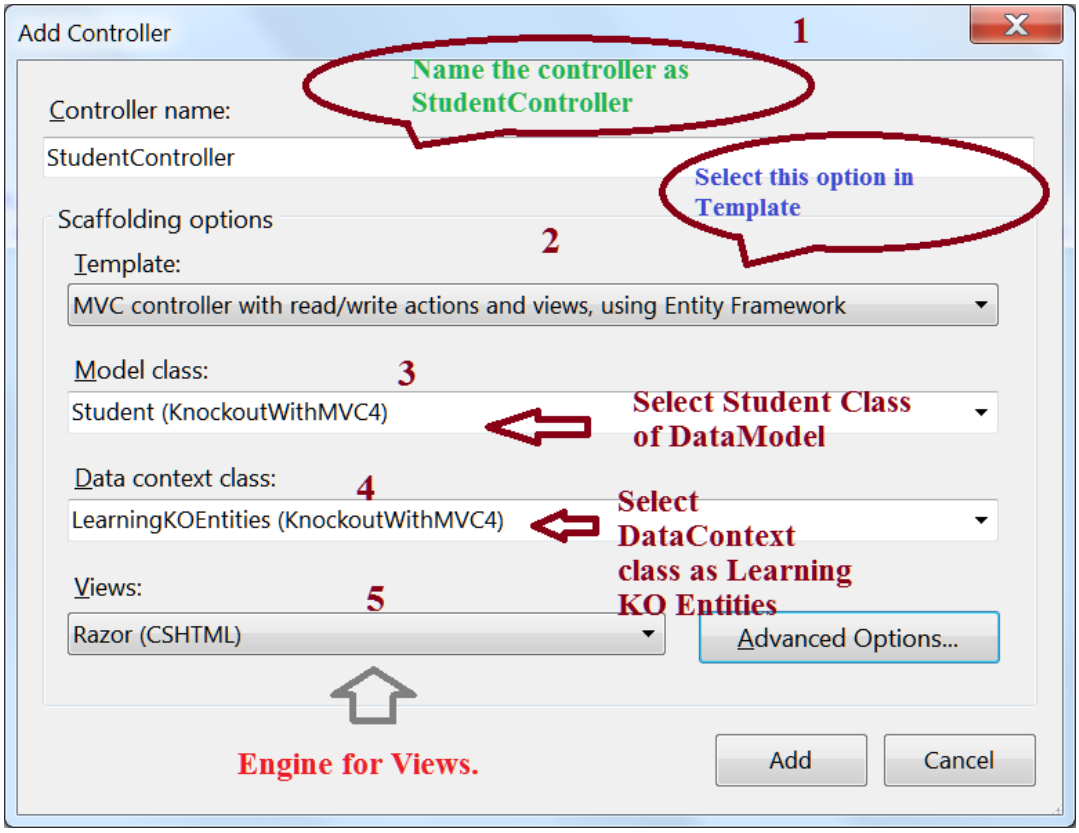


Select tables to be added to the model. In our case it's Student Table.



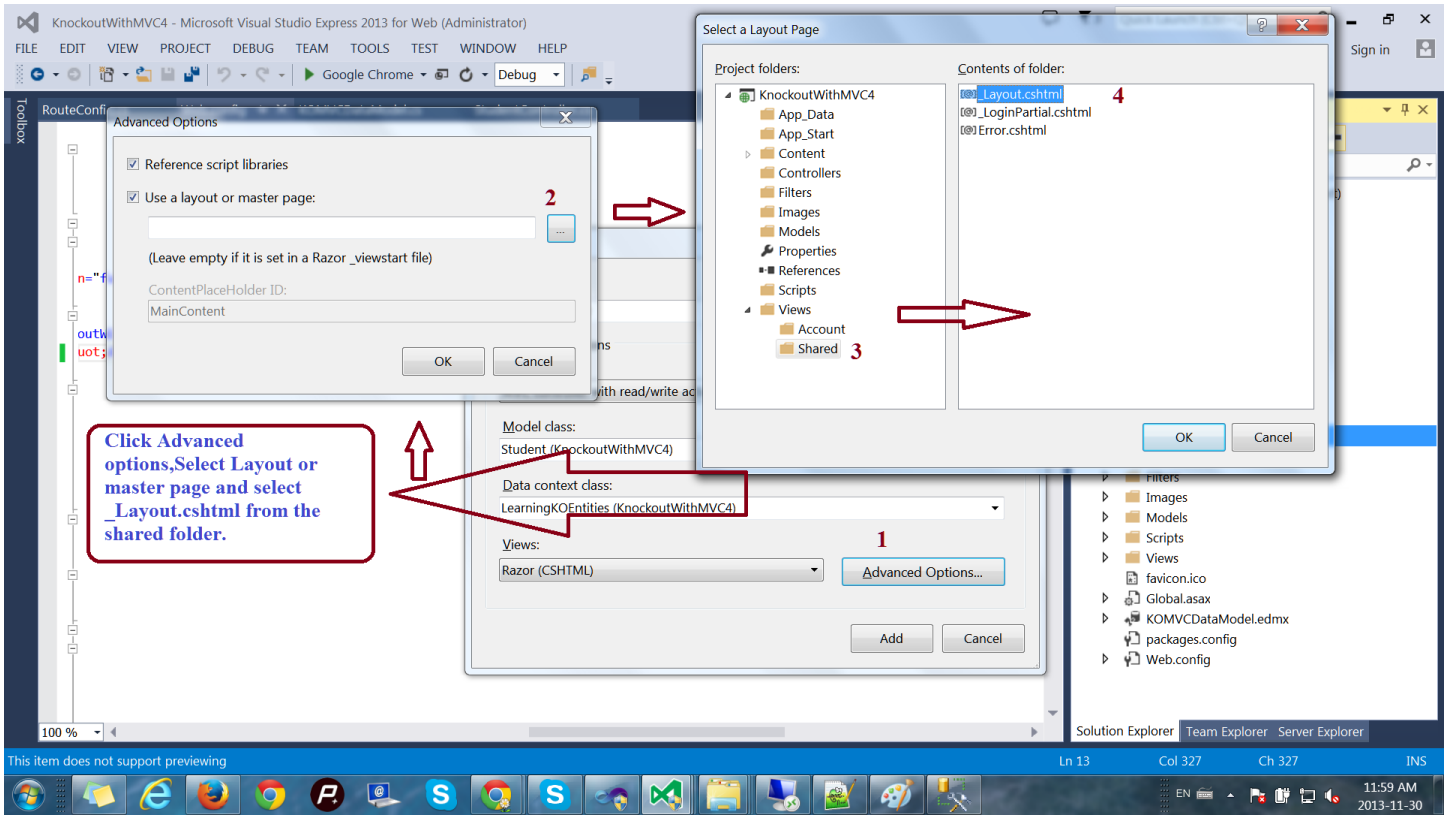
5. **Step5:** Now add a new controller to the Controller folder, right click controller folder and add a controller named Student. Since we have already created our Datamodel, we can choose for an option where CRUD actions are created by chosen Entity Framework Datamodel,





- Name your controller as StudentController,

- from Scaffolding Options, select “MVC controller with read/write actions and views,using Entity Framework”.
- Select Model class as Student, that lies in our solution.
- Select Data context class as LearningKOEntities that is added to outr solution when we added EF data model.
- Select Razor as rendering engine for views.
- Click Advanced options,Select Layout or master page and select _Layout.cshtml from the shared folder.



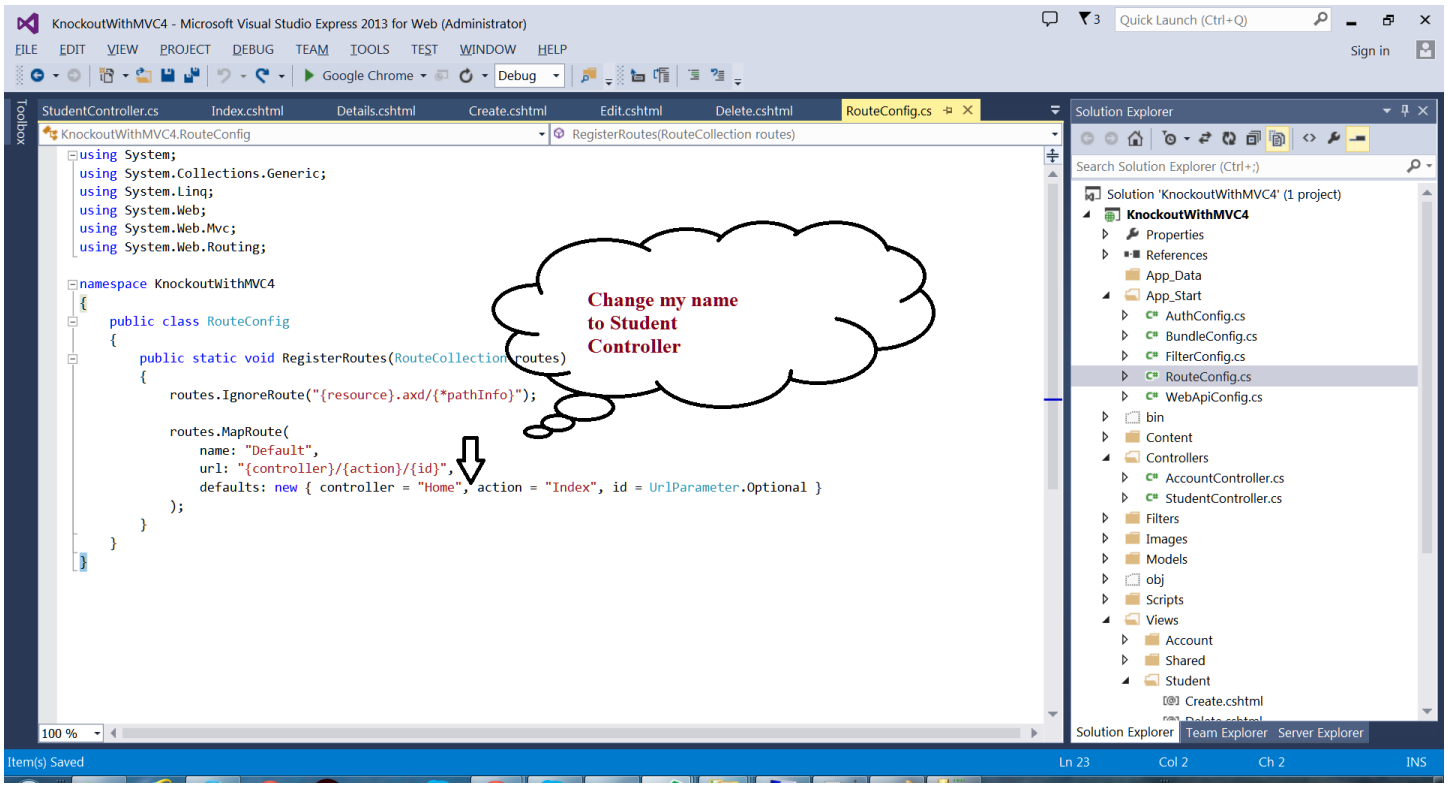
6. **Step6:** We see out student controller prepared with all the CRUD operation actions as shown below,

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.Entity;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace KnockoutWithMVC4.Controllers
{
    public class StudentController : Controller
    {
        private LearningKOEntities db = new LearningKOEntities();

        //
        // GET: /Student/
```

```
23.
24.         public ActionResult Index()
25.         {
26.             return View(db.Students.ToList());
27.         }
28.
29.         //
30.         // GET: /Student/Details/5
31.
32.         public ActionResult Details(string id = null)
33.         {
34.             Student student = db.Students.Find(id);
35.             if (student == null)
36.             {
37.                 return HttpNotFound();
38.             }
39.             return View(student);
40.         }
41.
42.         //
43.         // GET: /Student/Create
44.
45.         public ActionResult Create()
46.         {
47.             return View();
48.         }
49.
50.         //
51.         // POST: /Student/Create
52.
53.         [HttpPost]
54.         [ValidateAntiForgeryToken]
55.         public ActionResult Create(Student student)
56.         {
57.             if (ModelState.IsValid)
58.             {
59.                 db.Students.Add(student);
60.                 db.SaveChanges();
61.                 return RedirectToAction("Index");
62.             }
63.
64.             return View(student);
65.         }
66.
67.         //
68.         // GET: /Student/Edit/5
69.
70.         public ActionResult Edit(string id = null)
71.         {
72.             Student student = db.Students.Find(id);
73.             if (student == null)
74.             {
75.                 return HttpNotFound();
76.             }
77.             return View(student);
78.         }
79.
```

```
80.         //
81.         // POST: /Student/Edit/5
82.
83.         [HttpPost]
84.         [ValidateAntiForgeryToken]
85.         public ActionResult Edit(Student student)
86.         {
87.             if (ModelState.IsValid)
88.             {
89.                 db.Entry(student).State = EntityState.Modified;
90.                 db.SaveChanges();
91.                 return RedirectToAction("Index");
92.             }
93.             return View(student);
94.         }
95.
96.         //
97.         // GET: /Student/Delete/5
98.
99.         public ActionResult Delete(string id = null)
100.        {
101.            Student student = db.Students.Find(id);
102.            if (student == null)
103.            {
104.                return HttpNotFound();
105.            }
106.            return View(student);
107.        }
108.
109.        //
110.        // POST: /Student/Delete/5
111.
112.        [HttpPost, ActionName("Delete")]
113.        [ValidateAntiForgeryToken]
114.        public ActionResult DeleteConfirmed(string id)
115.        {
116.            Student student = db.Students.Find(id);
117.            db.Students.Remove(student);
118.            db.SaveChanges();
119.            return RedirectToAction("Index");
120.        }
121.
122.        protected override void Dispose(bool disposing)
123.        {
124.            db.Dispose();
125.            base.Dispose(disposing);
126.        }
127.    }
128.            }
```

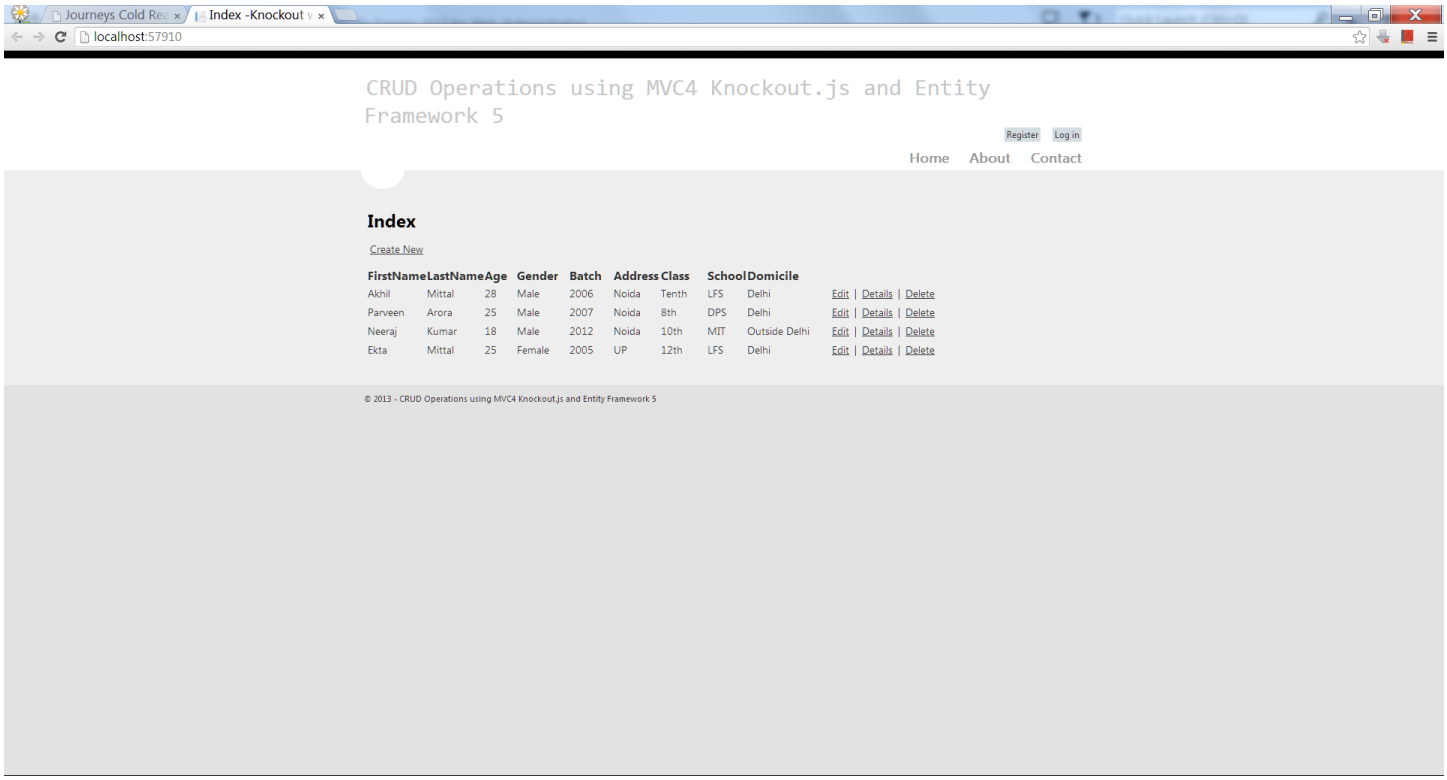
7.Step7: Open App_Start folder and, change the name of controller from Home to Student,



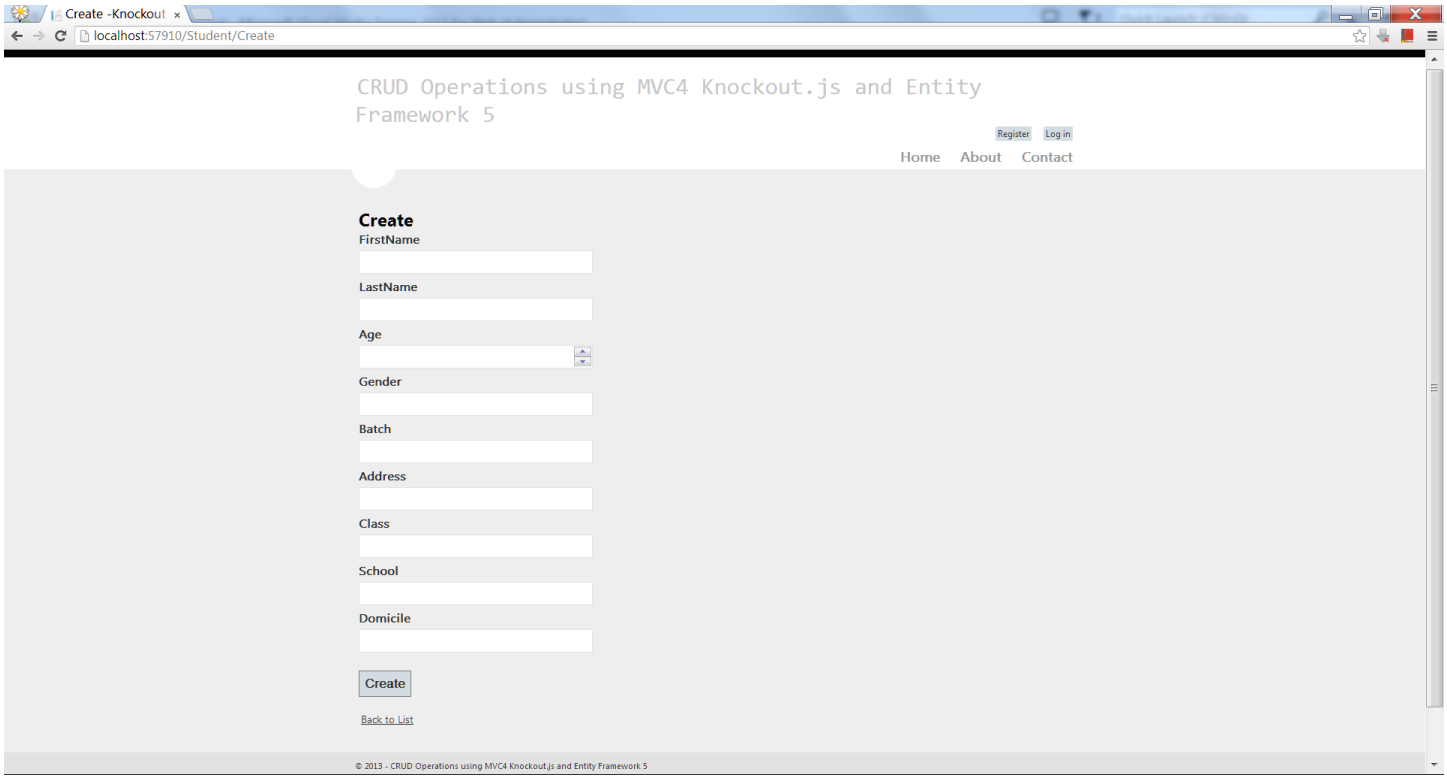
the code will become as,

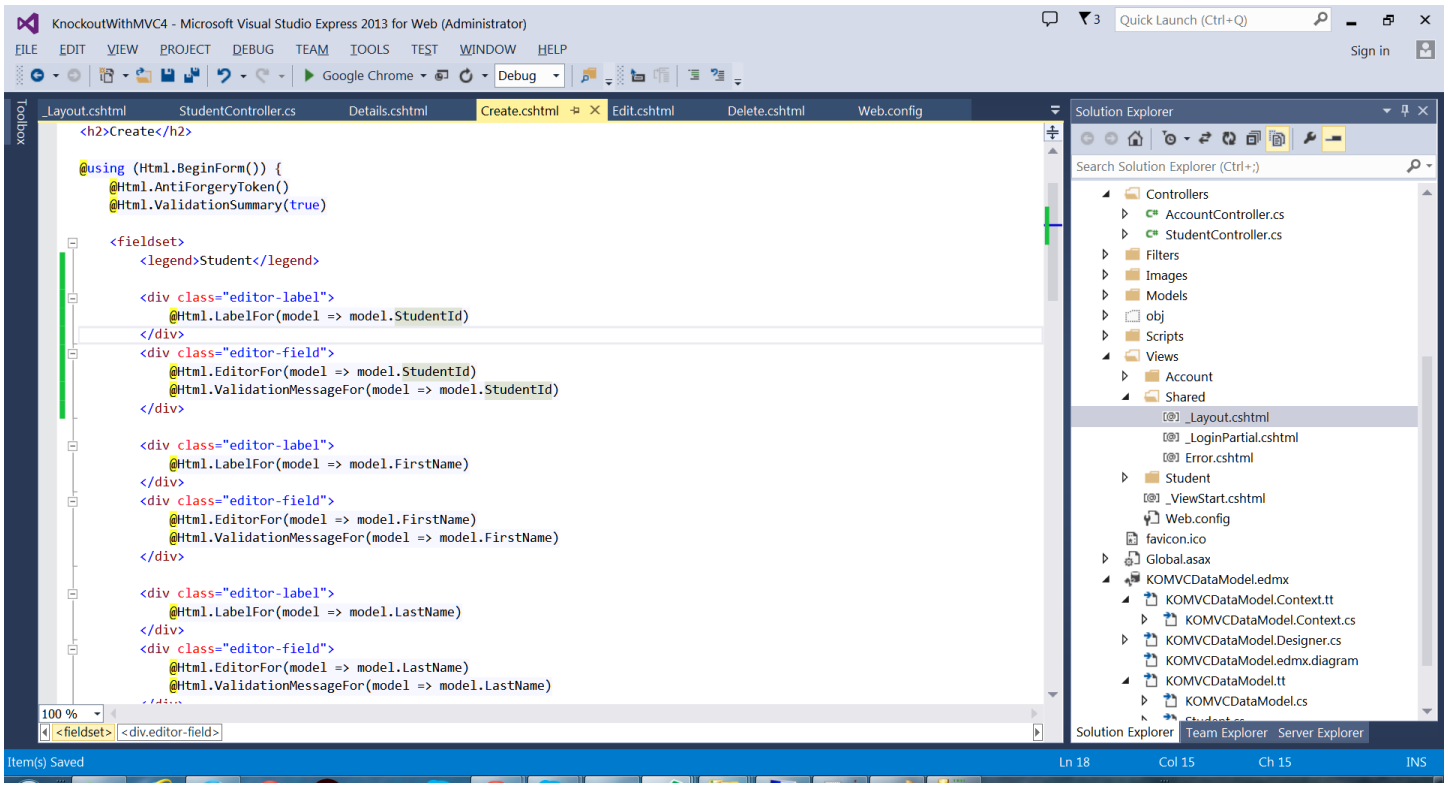
```
public static void RegisterRoutes(RouteCollection routes)
  {
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

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

8. Step8: Now press F5 to run the application, and you'll see the list of all students we added in to table Student while creating it is displayed. Since the CRUD operations are automatically written, we have action results for display list and other Edit, Delete and Create operations. Note that views for all the operations are created in Views Folder under Student Folder name.

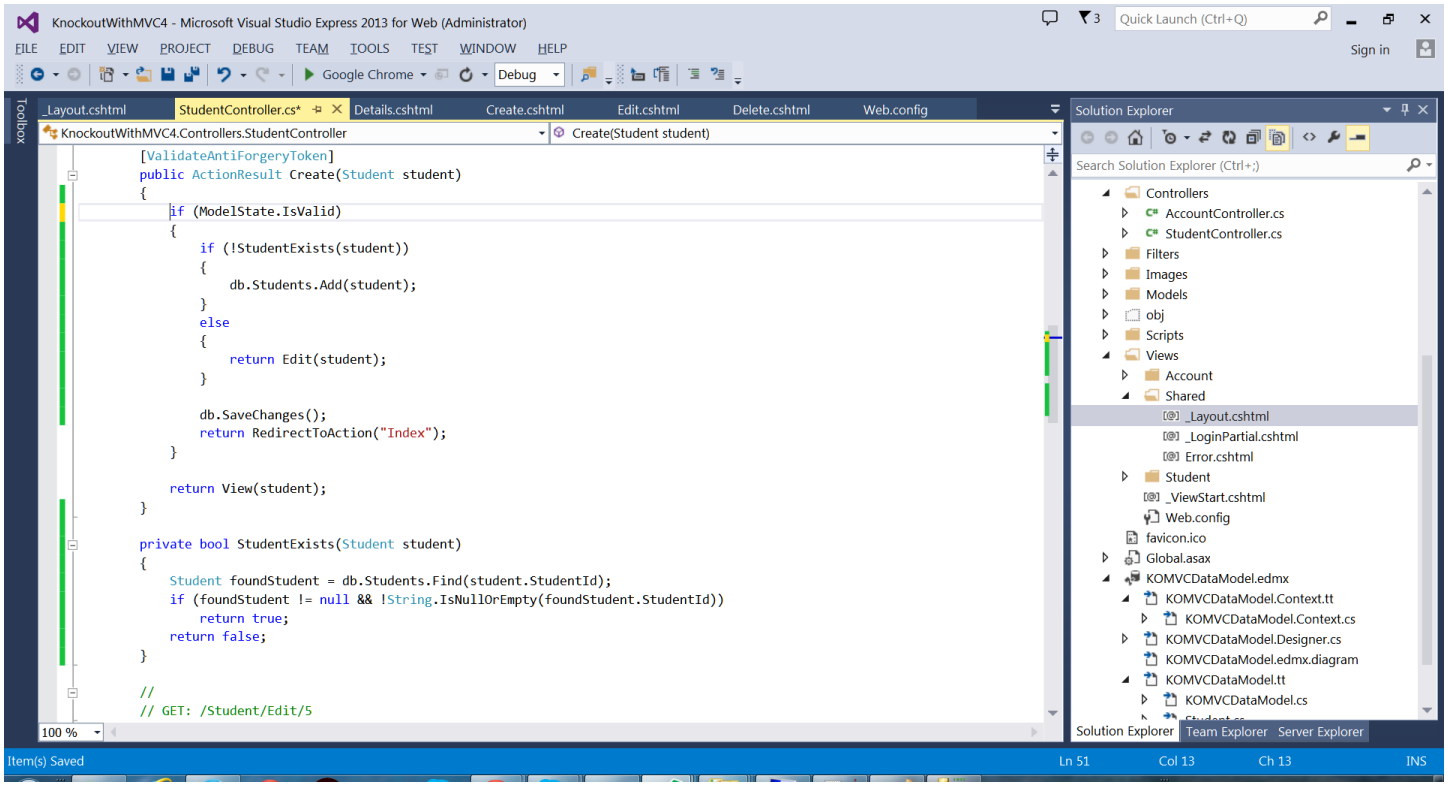


Now you can perform all the operations on this list.

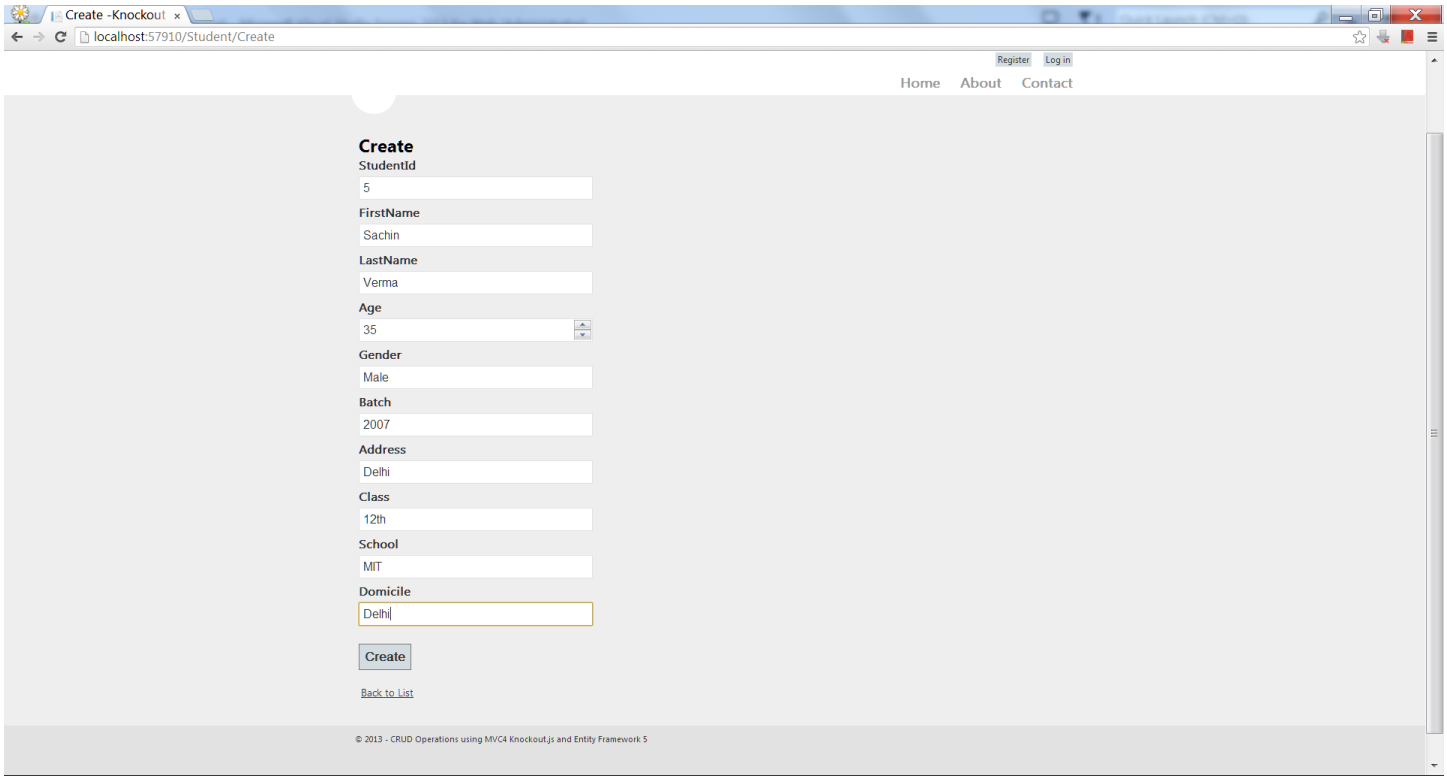




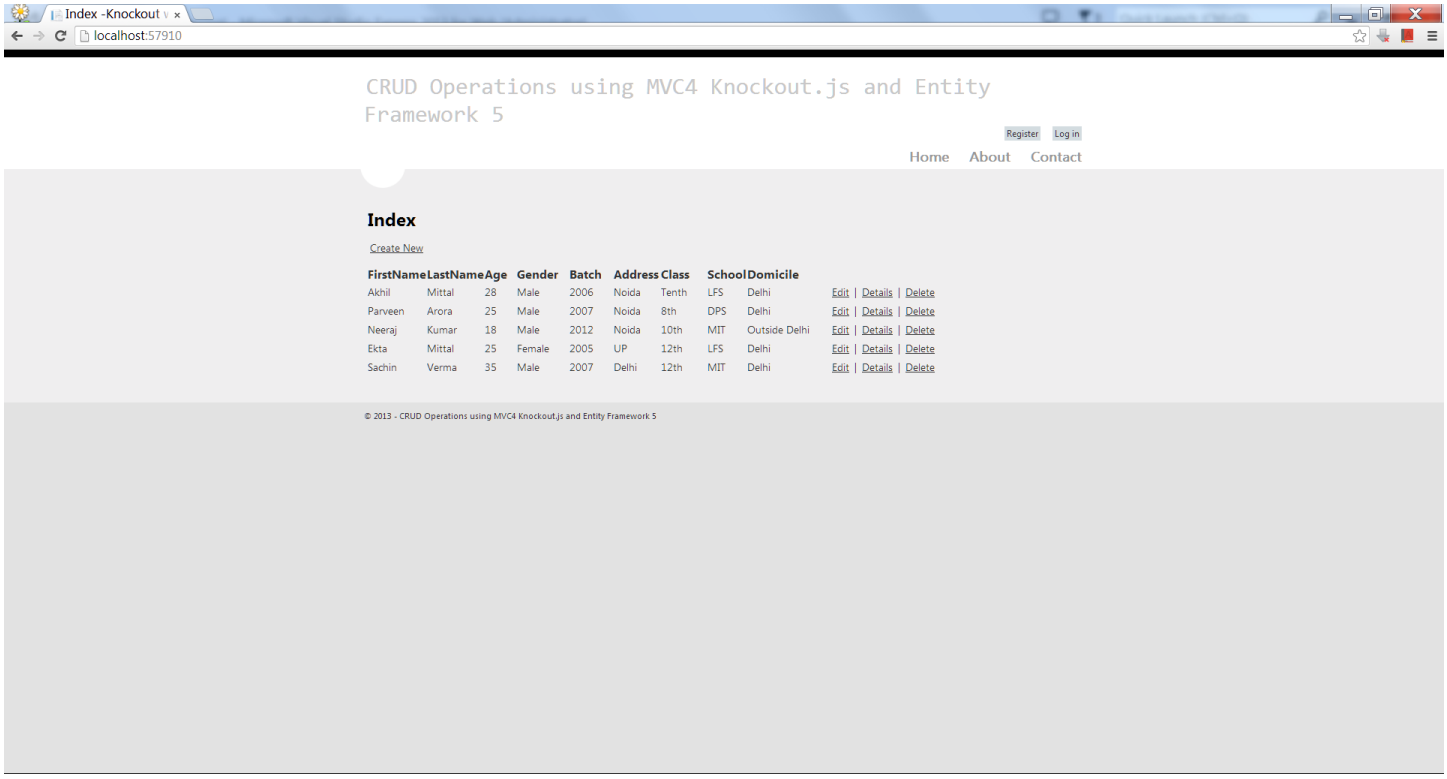
Since I have not provided any validation checks on model or creating an existing student id, the code may break, so I am calling Edit Action in create when we find that id already exists,



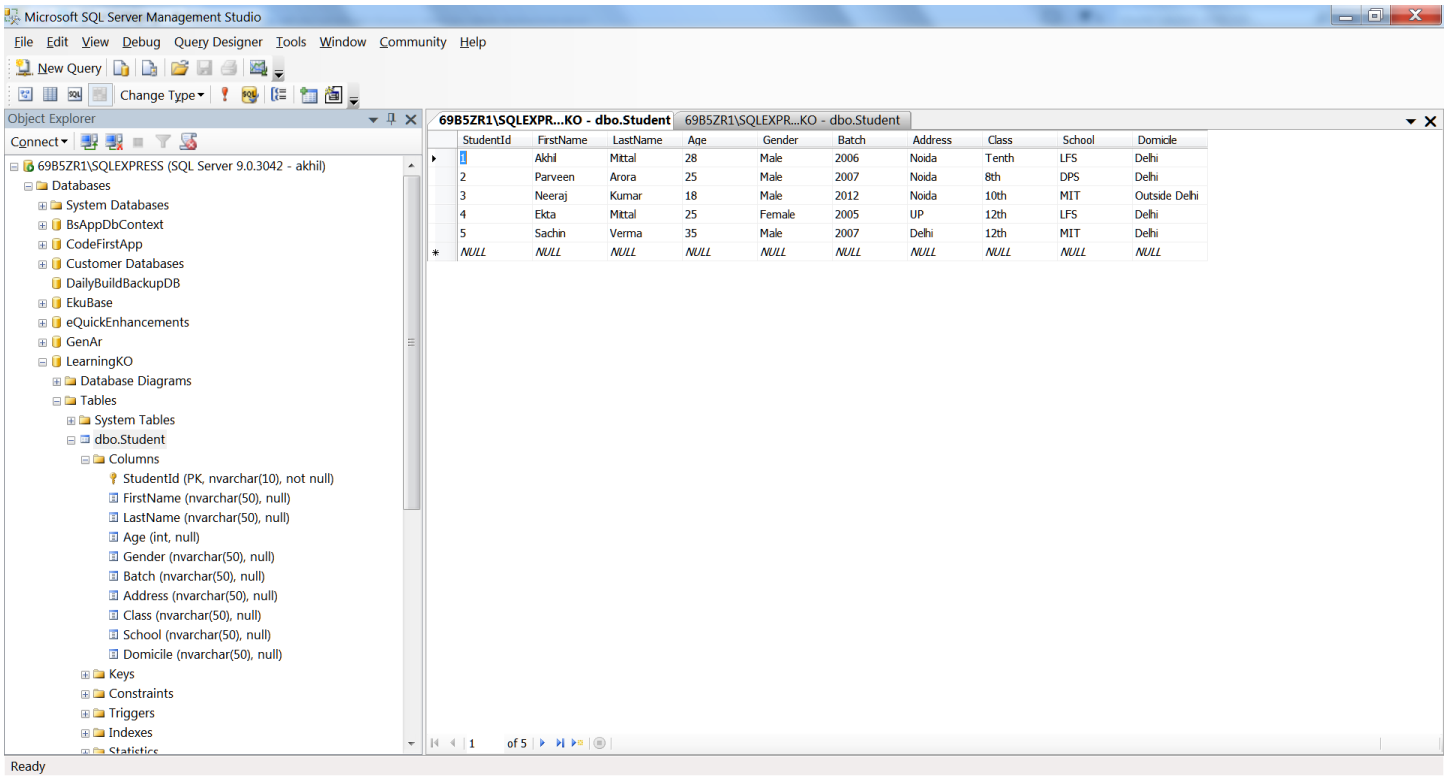
Now create new student ,



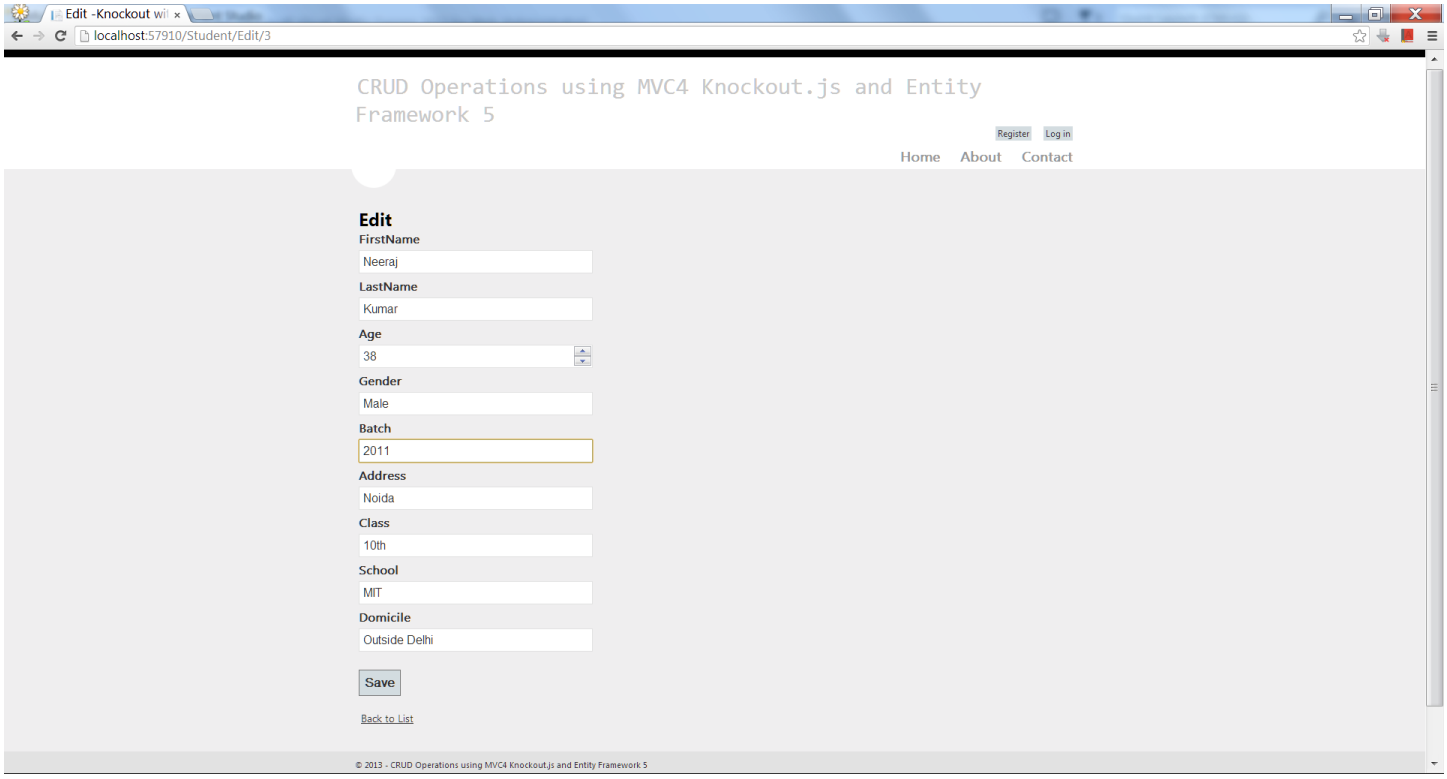
We see that the student is created successfully and added to the list,



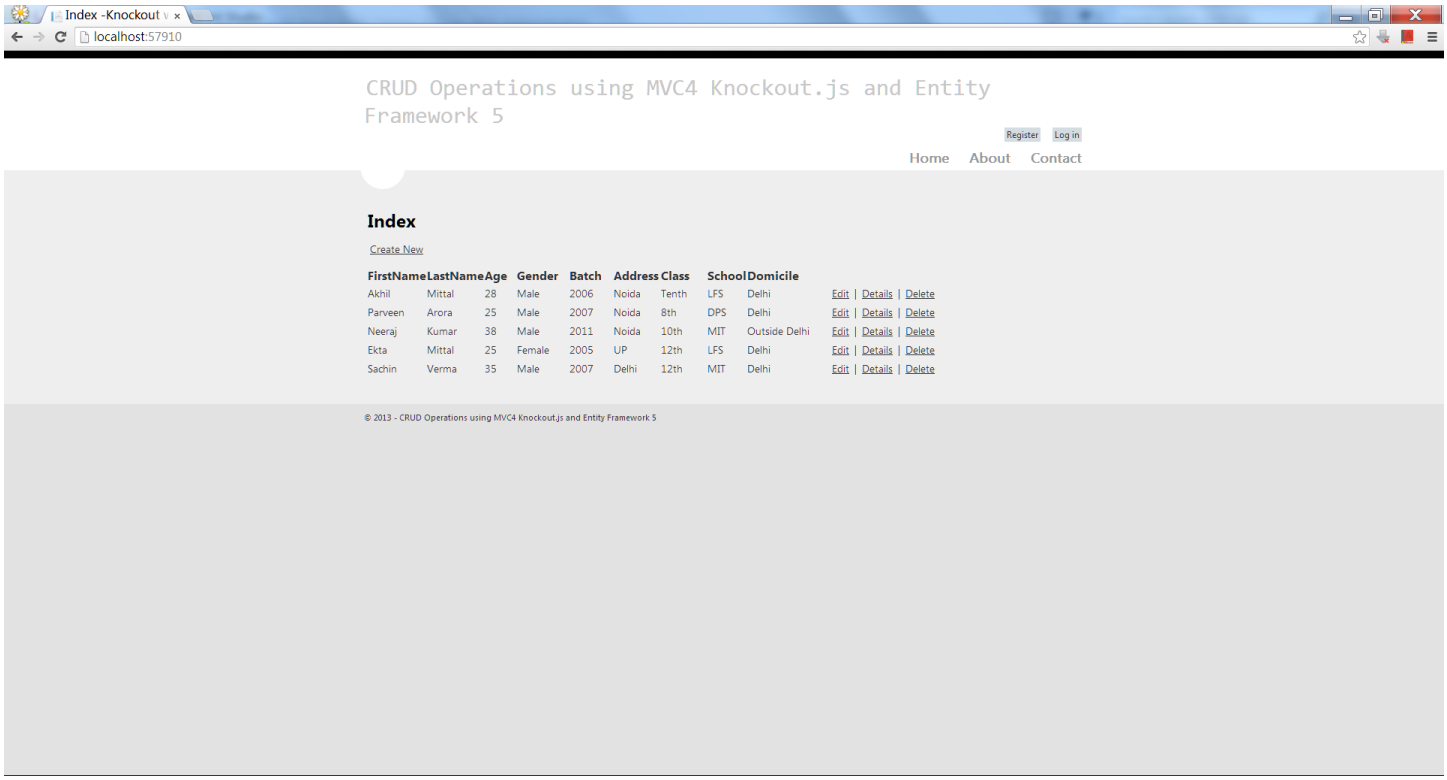
In data base,



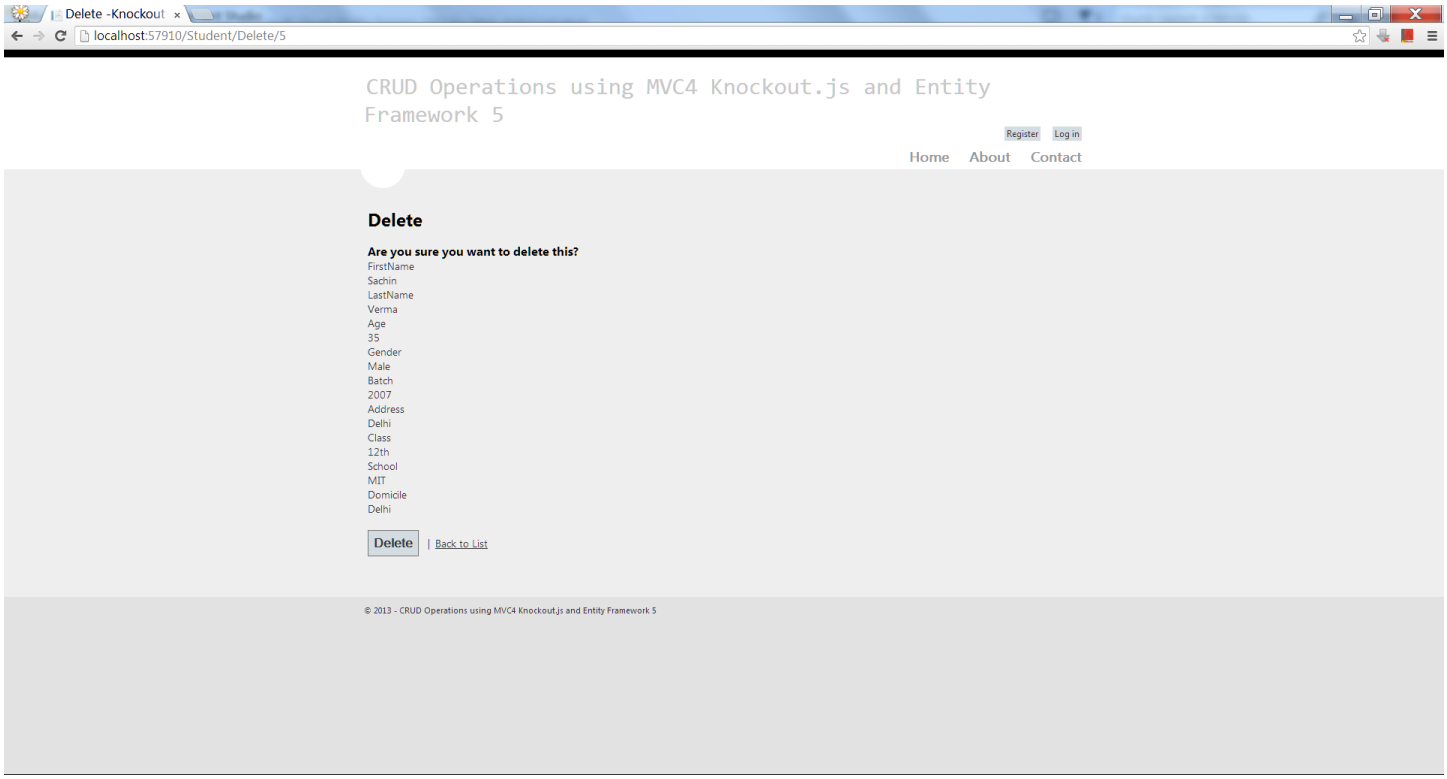
Similarly for Edit,



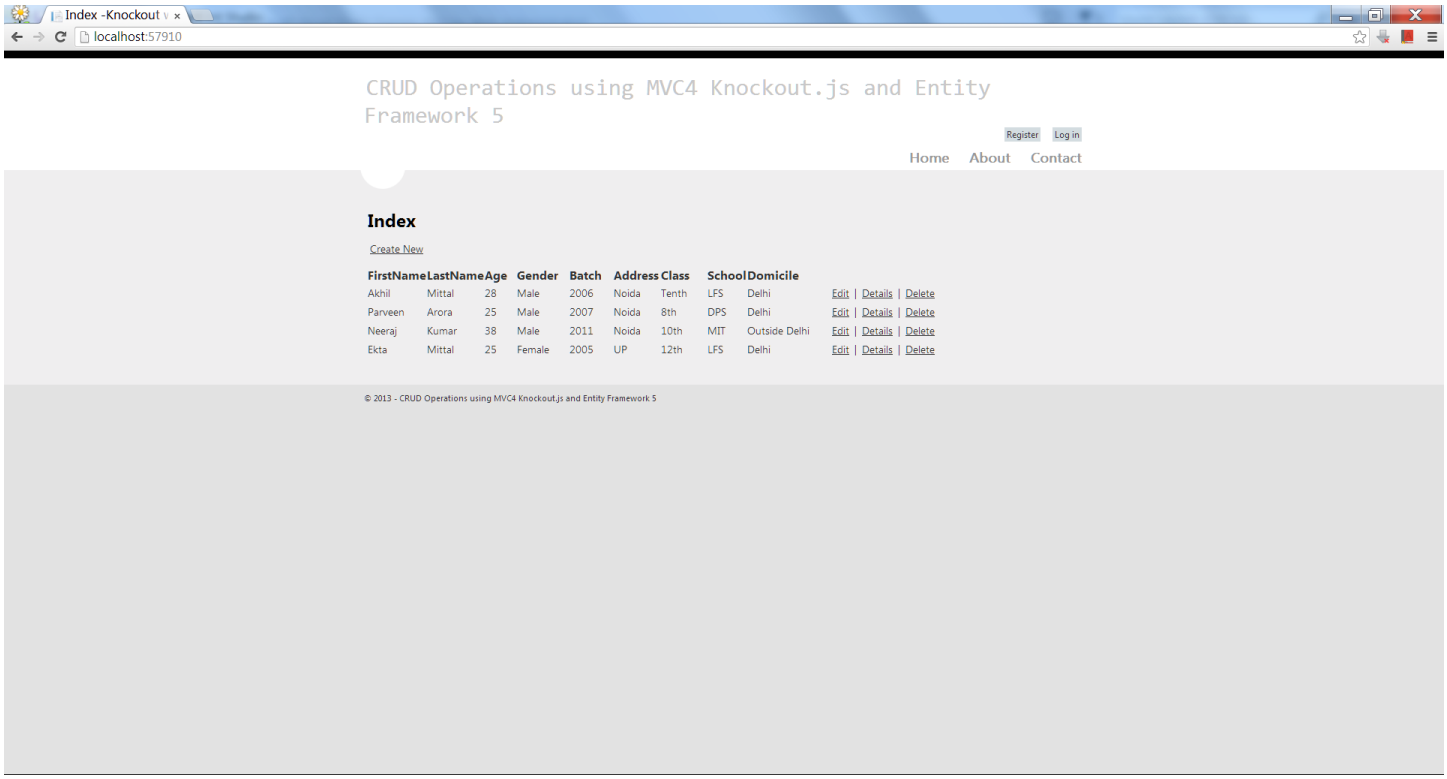
Change any field and press save.The change will be reflected in the list and data base,



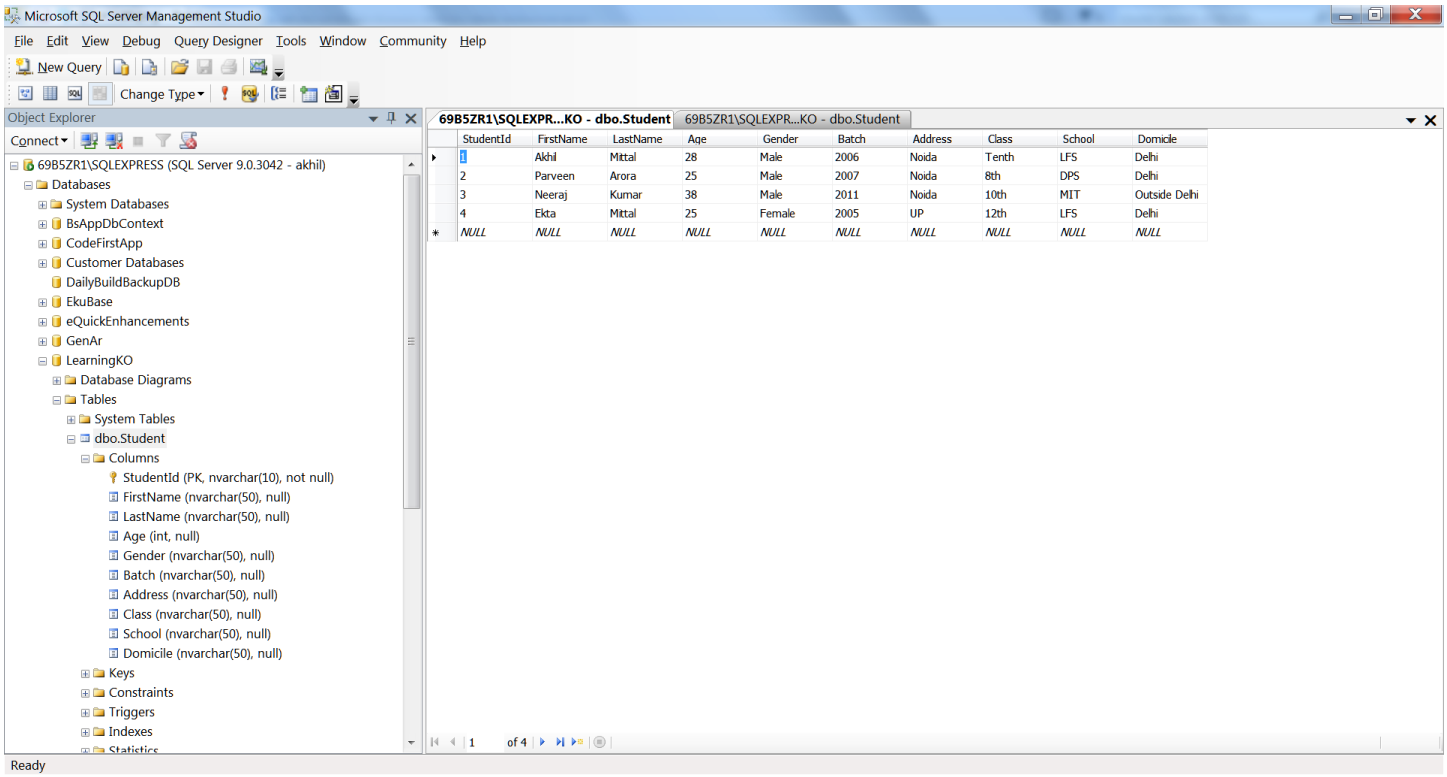
For Delete,



Student Deleted.



And in database,



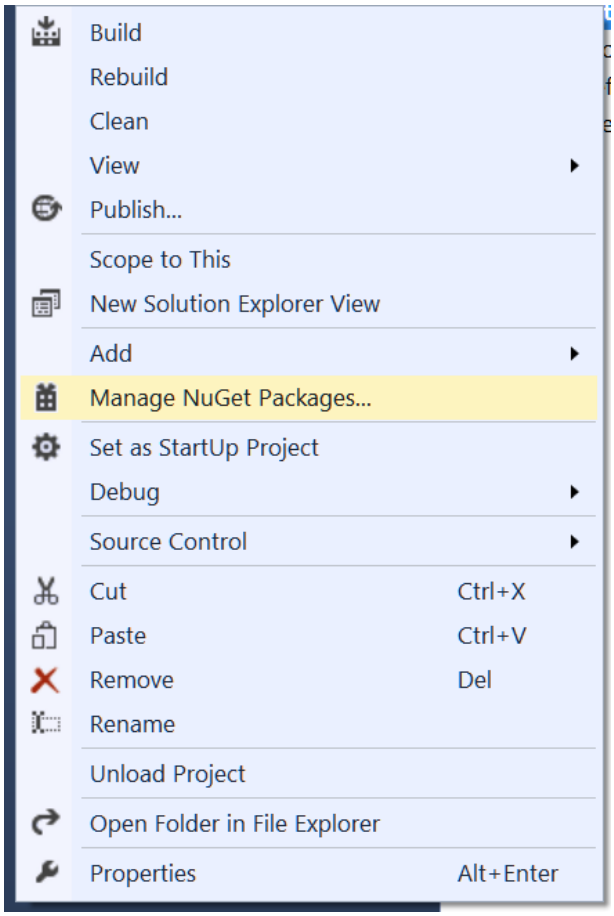
So, that's it, our first job is completed i.e. to create an MVC application, and perform CRUD operations using Entity Framework 5.You can see that till now we have not written a single line of code. Yes that's the magic of MVC and EF. Cheers !!!

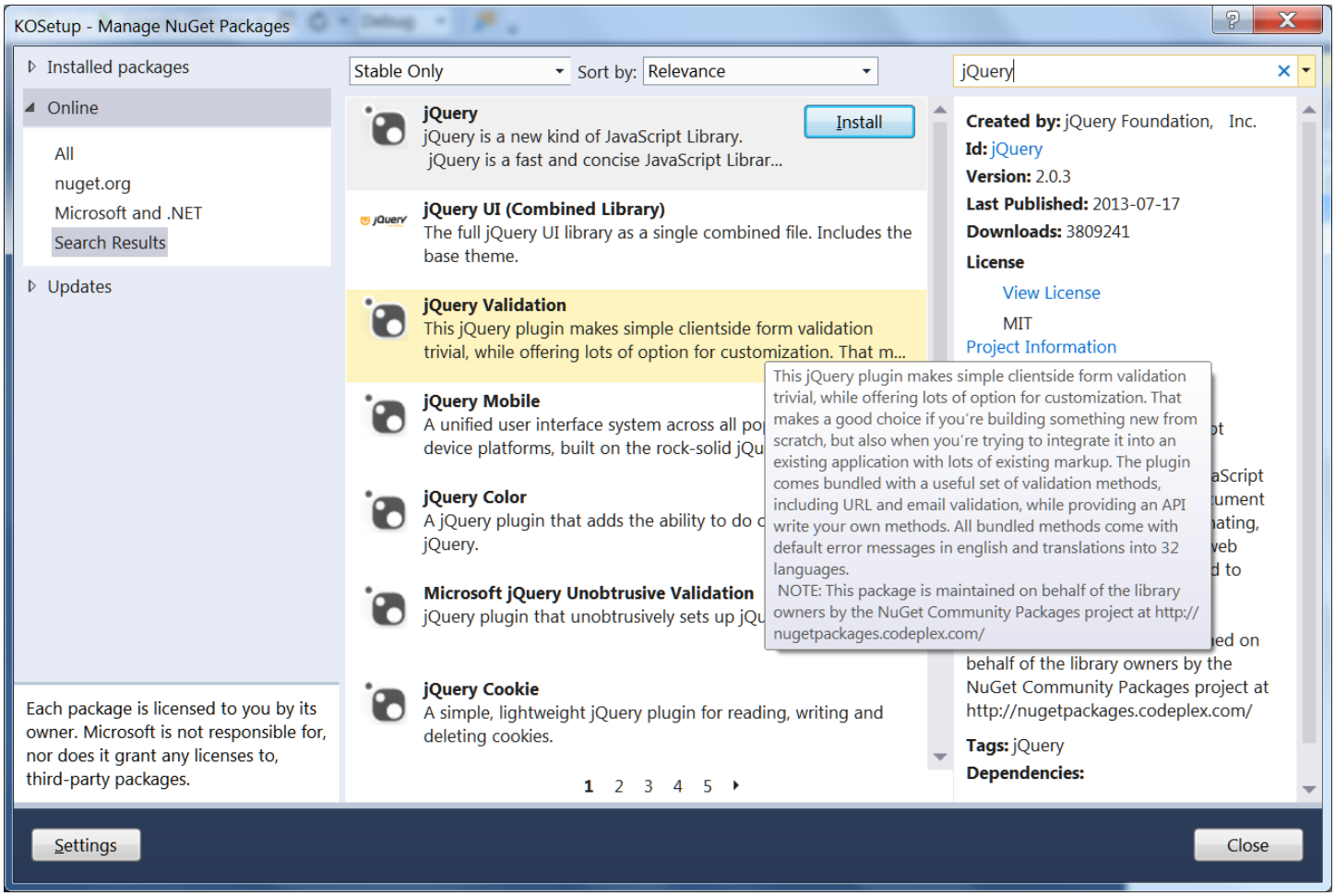
Knockout Application :

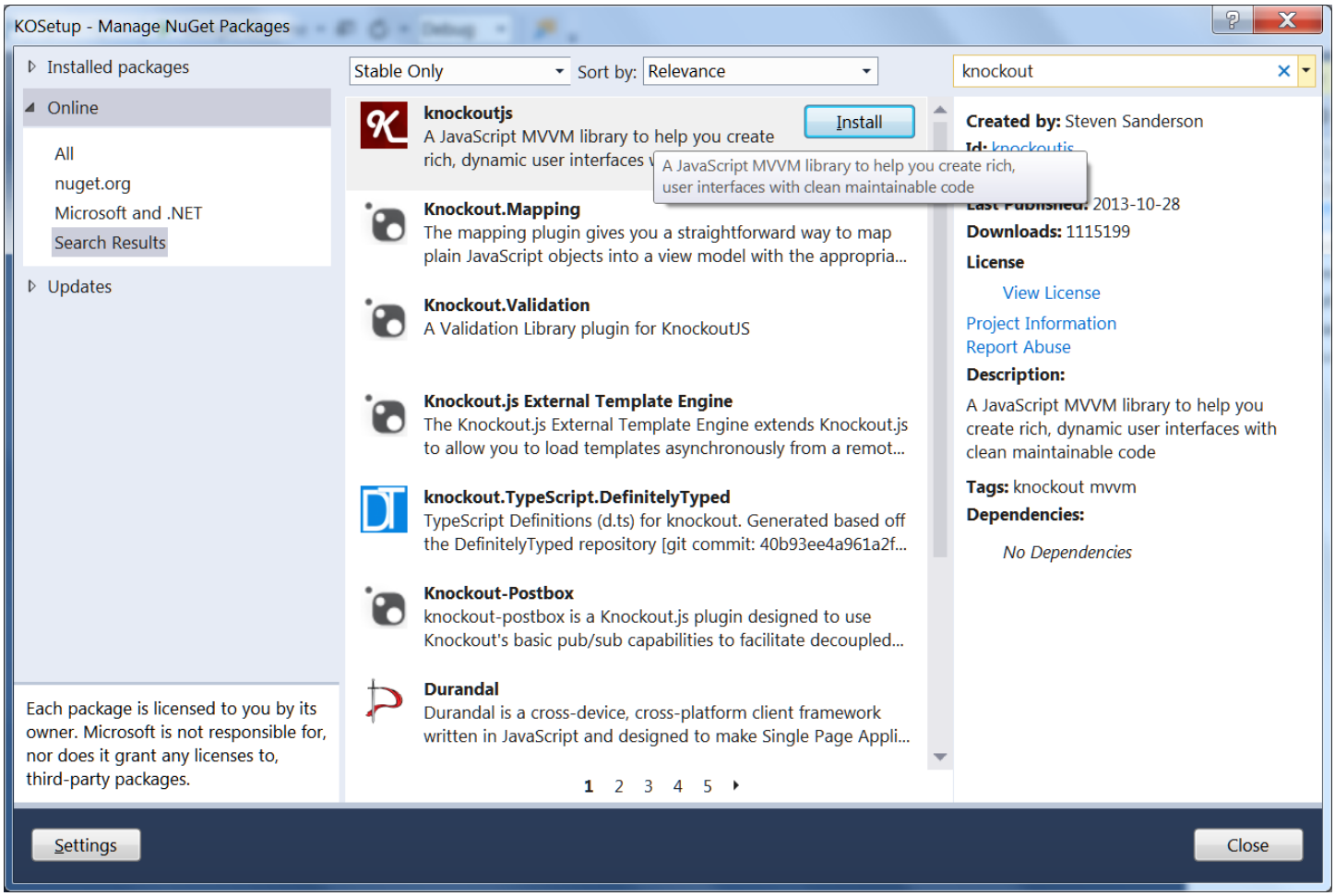
Our first job is well done, now moving on to our primary target, i.e. KO. Since KO depends largely on MVVM pattern, we'll take MVVM at client side, and use our controller to be as same just modified little bit for returning JSON logic. You can learn about MVVM pattern and KO theory in the first chapter.

1. **Step1:**

   JQuery and Knockout.js files are very important to be in the solution's script folder. Check to them and if you do not find them, then add the packages for jQuery and Knockout, in the same fashion as you added Entity Framework. Right click project, select "Manage nugget packages" and search for jQuery then install it , then search for knockout package and install it,







1. **Step2:** Right click Scripts folder and a folder named ViewModel.Add four js files to that folder, and name them as CreateVM.js, EditVM.js,DeleteVM.js and StudentListVM.js respectively. These are View Model files added to communicate to Controller and render our View templates.

2. **Step3:** Add some code to CreateVm.js as,

```
var urlPath = window.location.pathname;
$(function () {
   ko.applyBindings(CreateVM);
});

var CreateVM = {
   Domiciles: ko.observableArray(['Delhi', 'Outside Delhi']),
   Genders: ko.observableArray(['Male', 'Female']),
   Students: ko.observableArray([]),
   StudentId: ko.observable(),
   FirstName: ko.observable(),
   LastName: ko.observable(),
   Age: ko.observable(),
   Batch: ko.observable(),
```

```
    Address: ko.observable(),
    Class: ko.observable(),
    School: ko.observable(),
    Domicile: ko.observable(),
    Gender: ko.observable(),
    SaveStudent: function () {
        $.ajax({
            url: '/Student/Create',
            type: 'post',
            dataType: 'JSON',
            data: ko.toJSON(this),
            contentType: 'application/JSON',
            success: function (result) {
            },
            error: function (err) {
                if (err.responseText == "Creation Failed")
                { window.location.href = '/Student/Index/'; }
                else {
                    alert("Status:"+err.responseText);
                    window.location.href = '/Student/Index/';;
                }
            },
            complete: function () {
                window.location.href = '/Student/Index/';
            }
        });
    }
};
```

On document load we apply bindings for CreateVM, then inside the view model method we initialize the observables to properties of Student, that will be bind to respective view. You can read more about observables in KO in first chapter. There is a save function, that sends an ajax request to Student Controller's Create method, and gets string result. data: ko.toJSON(this), means sending the object in JSON format to controller method.

Student/Create Controller Method:

Modify the code of controller method of Create, to return JSON to the caller. The html templates bound with objects are actually bound to JSON properties, set in the methods of view model using Knockout Observables. The work on observer pattern, so that when model is updated the views automatically gets updated and when views get updated the models update itself, this is called two way binding.

Controller Code:

```
    [HttpPost]
     public string Create(Student student)
     {
        if (ModelState.IsValid)
        {
            if (!StudentExists(student))
                db.Students.Add(student);
            else
                return Edit(student);
```

```
            db.SaveChanges();
            return "Student Created";
        }
        return "Creation Failed";
    }
```

View Code:

Change the code of the already created views to work with KO,

For create.cshtml,

```
<h2>Create</h2>
<fieldset>
    <legend>Create Student</legend>

    <div class="editor-label">
        Student id
    </div>
    <div class="editor-field">
        <input data-bind="value: StudentId" />
    </div>

    <div class="editor-label">
        First Name
    </div>
    <div class="editor-field">
        <input data-bind="value: FirstName" />
    </div>

    <div class="editor-label">
        Last Name
    </div>
    <div class="editor-field">
        <input data-bind="value: LastName" />
    </div>

    <div class="editor-label">
        Age
    </div>
    <div class="editor-field">
        <input data-bind="value: Age" />
    </div>

    <div class="editor-label">
        Gender
    </div>
    <div class="editor-field">
        <select data-bind="options: Genders, value: Gender, optionsCaption: 'Select Gender...'"></select>
    </div>

    <div class="editor-label">
        Batch
    </div>
```

```
    <div class="editor-field">
        <input data-bind="value: Batch" />
    </div>

    <div class="editor-label">
        Address
    </div>
    <div class="editor-field">
        <input data-bind="value: Address" />
    </div>

    <div class="editor-label">
        Class
    </div>
    <div class="editor-field">
        <input data-bind="value: Class" />
    </div>

    <div class="editor-label">
        School
    </div>
    <div class="editor-field">
        <input data-bind="value: School" />
    </div>

    <div class="editor-label">
        Domicile
    </div>
    <div class="item ">
        <select data-bind="options: Domiciles, value: Domicile, optionsCaption: 'Select Domicile...'"></select>
    </div>

    <p>
        <button type="button" data-bind="click: SaveStudent">Save Student To Database</button>
    </p>
</fieldset>
<div>
    <a href="@Url.Action("Index", "Student")" >Back to List</a>
</div>

@section Scripts {
    @Scripts.Render("~/Scripts/ViewModels/CreateVM.js")
}
```

You can see I have used data-bind attribute of HTML5 to bind the View elements to View Models properties like data-bind="value: StudentId" , the same applies to all the editable elements.Click button is bound to SaveStudent method of view model.

At the end of the page we have registered the CreateVM.js view model for this particular view by

```
@section Scripts {
    @Scripts.Render("~/Scripts/ViewModels/CreateVM.js")
}
```

Tag.

3. **Step3:** We do the same set of operations for all the views, View models and Controller method, the code is as below,

For Edit:

View Model:

Code added in StudentListVM for Edit View Model, since it only perform get when it loads.

Controller methods:

```
public ActionResult Edit(string id=null)
    {
      Student student = db.Students.Find(id);
      if (student == null)
      {
        return null;
      }
      JavaScriptSerializer serializer = new JavaScriptSerializer();
      ViewBag.InitialData = serializer.Serialize(student);
      return View();
    }

    /// <summary>
    /// Edits particular student details
    /// </summary>
    /// <param name="student"></param>
    /// <returns></returns>
    [HttpPost]
    public string Edit(Student student)
    {
      if (ModelState.IsValid)
      {
        db.Entry(student).State = EntityState.Modified;
        db.SaveChanges();
        return "Student Edited";
      }
      return "Edit Failed";
    }
```

View:

```
<h2>Edit</h2>
<fieldset>
  <legend>Edit Student</legend>

  <div class="editor-label">
    Student id
  </div>
  <div class="editor-field">
```

```
    <input data-bind="value: StudentId" readonly="readonly" />
</div>

<div class="editor-label">
    First Name
</div>
<div class="editor-field">
    <input data-bind="value: FirstName" />
</div>

<div class="editor-label">
    Last Name
</div>
<div class="editor-field">
    <input data-bind="value: LastName" />
</div>

<div class="editor-label">
    Age
</div>
<div class="editor-field">
    <input data-bind="value: Age" />
</div>

<div class="editor-label">
    Gender
</div>
<div class="editor-field">
    <select data-bind="options: Genders, value: Gender, optionsCaption: 'Select Gender...'"></select>
</div>

<div class="editor-label">
    Batch
</div>
<div class="editor-field">
    <input data-bind="value: Batch" />
</div>

<div class="editor-label">
    Address
</div>
<div class="editor-field">
    <input data-bind="value: Address" />
</div>

<div class="editor-label">
    Class
</div>
<div class="editor-field">
    <input data-bind="value: Class" />
</div>

<div class="editor-label">
    School
</div>
<div class="editor-field">
    <input data-bind="value: School" />
```

```
    </div>

    <div class="editor-label">
        Domicile
    </div>
    <div class="editor-field">
        <select data-bind="options: Domiciles, value: Domicile, optionsCaption: 'Select Domicile...'"></select>
    </div>
    <p>
        <button type="button" data-bind="click: SaveStudent">Save Student To Database</button>
    </p>
</fieldset>
<div>
    <a href="@Url.Action("Index", "Student")">Back to List</a>
</div>
@section Scripts {
    <script>

        $(function () {
            ko.applyBindings(EditVM);
        });

        var initialData = '@Html.Raw(ViewBag.InitialData)'; //get the raw JSON
        var parsedJSON = $.parseJSON(initialData); //parse the JSON client side
        var EditVM = {
            Domiciles: ko.observableArray(['Delhi', 'Outside Delhi']),
            Genders: ko.observableArray(['Male', 'Female']),
            Students: ko.observableArray([]),
            StudentId: ko.observable(parsedJSON.StudentId),
            FirstName: ko.observable(parsedJSON.FirstName),
            LastName: ko.observable(parsedJSON.LastName),
            Age: ko.observable(parsedJSON.Age),
            Batch: ko.observable(parsedJSON.Batch),
            Address: ko.observable(parsedJSON.Address),
            Class: ko.observable(parsedJSON.Class),
            School: ko.observable(parsedJSON.School),
            Domicile: ko.observable(parsedJSON.Domicile),
            Gender: ko.observable(parsedJSON.Gender),
            SaveStudent: function () {
                $.ajax({
                    url: '/Student/Edit',
                    type: 'post',
                    dataType: 'JSON',
                    data: ko.toJSON(this),
                    contentType: 'application/JSON',
                    success: function (result) {
                    },
                    error: function (err) {
                        if (err.responseText == "Creation Failed")
                        { window.location.href = '/Student/Index/'; }
                        else {
                            alert("Status:" + err.responseText);
                            window.location.href = '/Student/Index/';;
                        }
                    },
                    complete: function () {
                        window.location.href = '/Student/Index/';
```

```
            }
        });
    }
};

</script>
}
```

For Delete:

View Model:

Code added in StudentListVM for Edit View Model, since it only perform get when it loads.

Controller methods:

```
public ActionResult Delete(string id = null)
{
    Student student = db.Students.Find(id);
    if (student == null)
    {
        return null;
    }
    JavaScriptSerializer serializer = new JavaScriptSerializer();
    ViewBag.InitialData = serializer.Serialize(student);
    return View();
}

/// <summary>
/// Delete particular student details
/// </summary>
/// <param name="student"></param>
/// <returns></returns>
[HttpPost]
public string Delete(Student student)
{
    Student studentDetail = db.Students.Find(student.StudentId);
    db.Students.Remove(studentDetail);
    db.SaveChanges();
    return "Student Deleted";
}
```

View:

```
@model KnockoutWithMVC4.Student

@{
    ViewBag.Title = "Delete";
}

<h2>Delete Student</h2>
```

```
<h3>Are you sure you want to delete this?</h3>
<fieldset>
  <legend>Delete</legend>
  <div class="display-label">
    Student Id
  </div>
  <div class="display-field">
    <input data-bind="value: StudentId" />

  </div>
  <div class="display-label">
    First Name
  </div>
  <div class="display-field">
    <input data-bind="value: FirstName" />
  </div>

  <div class="display-label">
    Last Name
  </div>
  <div class="display-field">
    <input data-bind="value: LastName" />

  </div>

  <div class="display-label">
    Age
  </div>
  <div class="display-field">
    <input data-bind="value: Age" />

  </div>

  <div class="display-label">
    Gender
  </div>
  <div class="display-field">
    <input data-bind="value: Gender" />

  </div>

  <div class="display-label">
    Batch
  </div>
  <div class="display-field">
    <input data-bind="value: Batch" />

  </div>

  <div class="display-label">
    Address

  </div>
  <div class="display-field">
    <input data-bind="value: Address" />
```

```
    </div>

    <div class="display-label">
        Class
    </div>
    <div class="display-field">
        <input data-bind="value: Class" />

    </div>

    <div class="display-label">
        School
    </div>
    <div class="display-field">
        <input data-bind="value: School" />

    </div>

    <div class="display-label">
        Domicile
    </div>
    <div class="display-field">
        <input data-bind="value: Domicile" />

    </div>
</fieldset>

<p>
    <button type="button" data-bind="click: DeleteStudent">Delete Student</button> |
    <a href="@Url.Action("Index", "Student")">Back to List</a>
</p>

@section Scripts {
    <script>

        $(function () {
            ko.applyBindings(DeleteVM);
        });

        var initialData = '@Html.Raw(ViewBag.InitialData)'; //get the raw JSON
        var parsedJSON = $.parseJSON(initialData); //parse the JSON client side
        var DeleteVM = {
            Domiciles: ko.observableArray(['Delhi', 'Outside Delhi']),
            Genders: ko.observableArray(['Male', 'Female']),
            Students: ko.observableArray([]),
            StudentId: ko.observable(parsedJSON.StudentId),
            FirstName: ko.observable(parsedJSON.FirstName),
            LastName: ko.observable(parsedJSON.LastName),
            Age: ko.observable(parsedJSON.Age),
            Batch: ko.observable(parsedJSON.Batch),
            Address: ko.observable(parsedJSON.Address),
            Class: ko.observable(parsedJSON.Class),
            School: ko.observable(parsedJSON.School),
            Domicile: ko.observable(parsedJSON.Domicile),
            Gender: ko.observable(parsedJSON.Gender),
            DeleteStudent: function () {
                $.ajax({
```

```
            url: '/Student/Delete',
            type: 'post',
            dataType: 'JSON',
            data: ko.toJSON(this),
            contentType: 'application/JSON',
            success: function (result) {
            },
            error: function (err) {
              if (err.responseText == "Creation Failed")
              { window.location.href = '/Student/Index/'; }
              else {
                alert("Status:" + err.responseText);
                window.location.href = '/Student/Index/';;
              }
            },
            complete: function () {
              window.location.href = '/Student/Index/';
            }
          });
        }
    };

  </script>
}
```

For Index(To display list):

View Model:

```
var urlPath = window.location.pathname;
$(function () {
  ko.applyBindings(StudentListVM);
  StudentListVM.getStudents();
});

//View Model
var StudentListVM = {
  Students: ko.observableArray([]),
  getStudents: function () {
    var self = this;
    $.ajax({
      type: "GET",
      url: '/Student/FetchStudents',
      contentType: "application/JSON; charset=utf-8",
      dataType: "JSON",
      success: function (data) {
        self.Students(data); //Put the response in ObservableArray
      },
      error: function (err) {
        alert(err.status + " : " + err.statusText);
      }
    });
  },
};
```

```
self.editStudent = function (student) {
   window.location.href =  '/Student/Edit/' + student.StudentId;
};
self.deleteStudent = function (student) {
   window.location.href = '/Student/Delete/' + student.StudentId;
};

//Model
function Students(data) {
   this.StudentId = ko.observable(data.StudentId);
   this.FirstName = ko.observable(data.FirstName);
   this.LastName = ko.observable(data.LastName);
   this.Age = ko.observable(data.Age);
   this.Gender = ko.observable(data.Gender);
   this.Batch = ko.observable(data.Batch);
   this.Address = ko.observable(data.Address);
   this.Class = ko.observable(data.Class);
   this.School = ko.observable(data.School);
   this.Domicile = ko.observable(data.Domicile);
}
```

Controller methods:

```
public JSONResult FetchStudents()
      {
         return JSON(db.Students.ToList(), JSONRequestBehavior.AllowGet);
      }
```

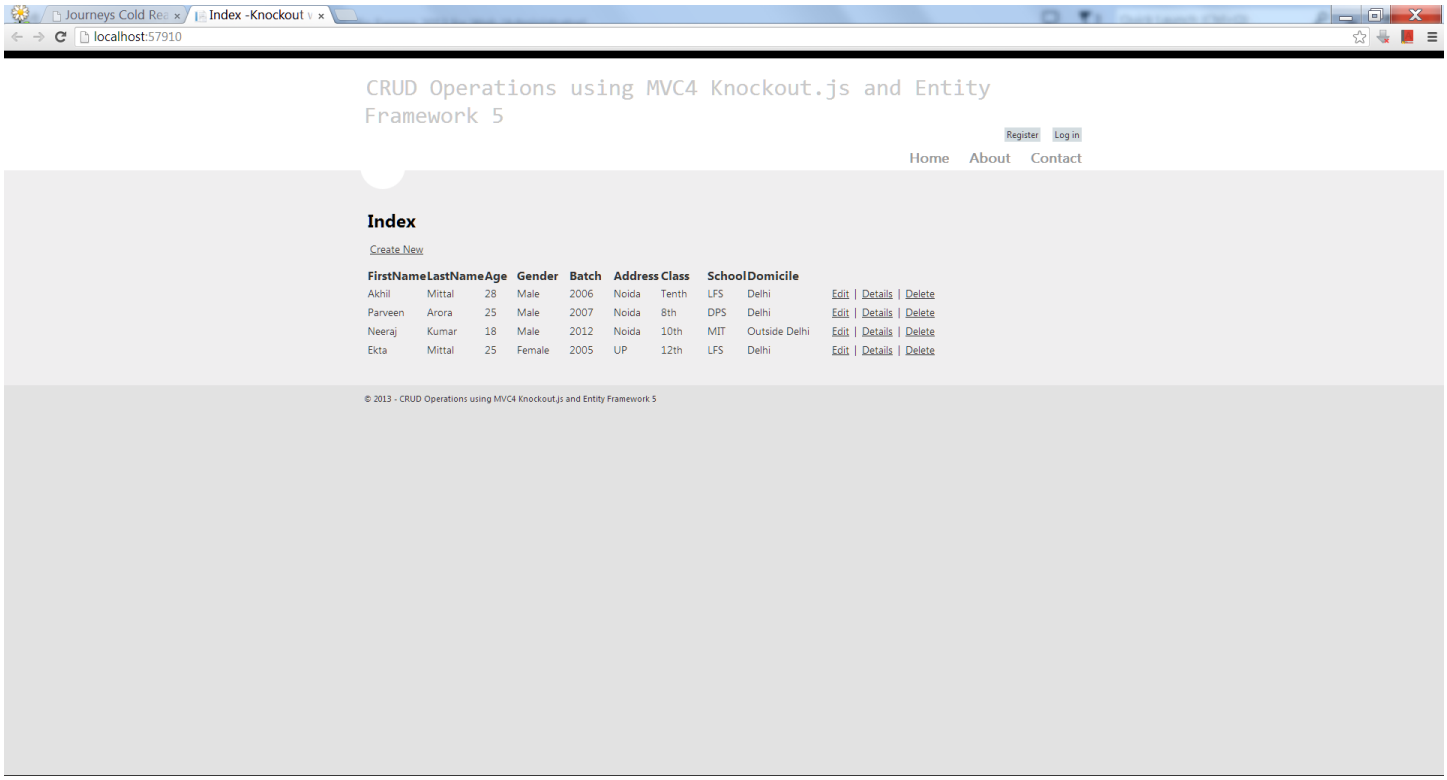
View:

```
@model IEnumerable<KnockoutWithMVC4.Student>

@{
   ViewBag.Title = "Index";
}

<h2>Students List</h2>

<p>
   <a href="@Url.Action("Create", "Student")" >Create Student</a>
</p>
<table>
   <thead>
      <tr>
         <th>
            Student Id
         </th>
         <th>
            First Name
         </th>
         <th>
            Last Name
         </th>
```

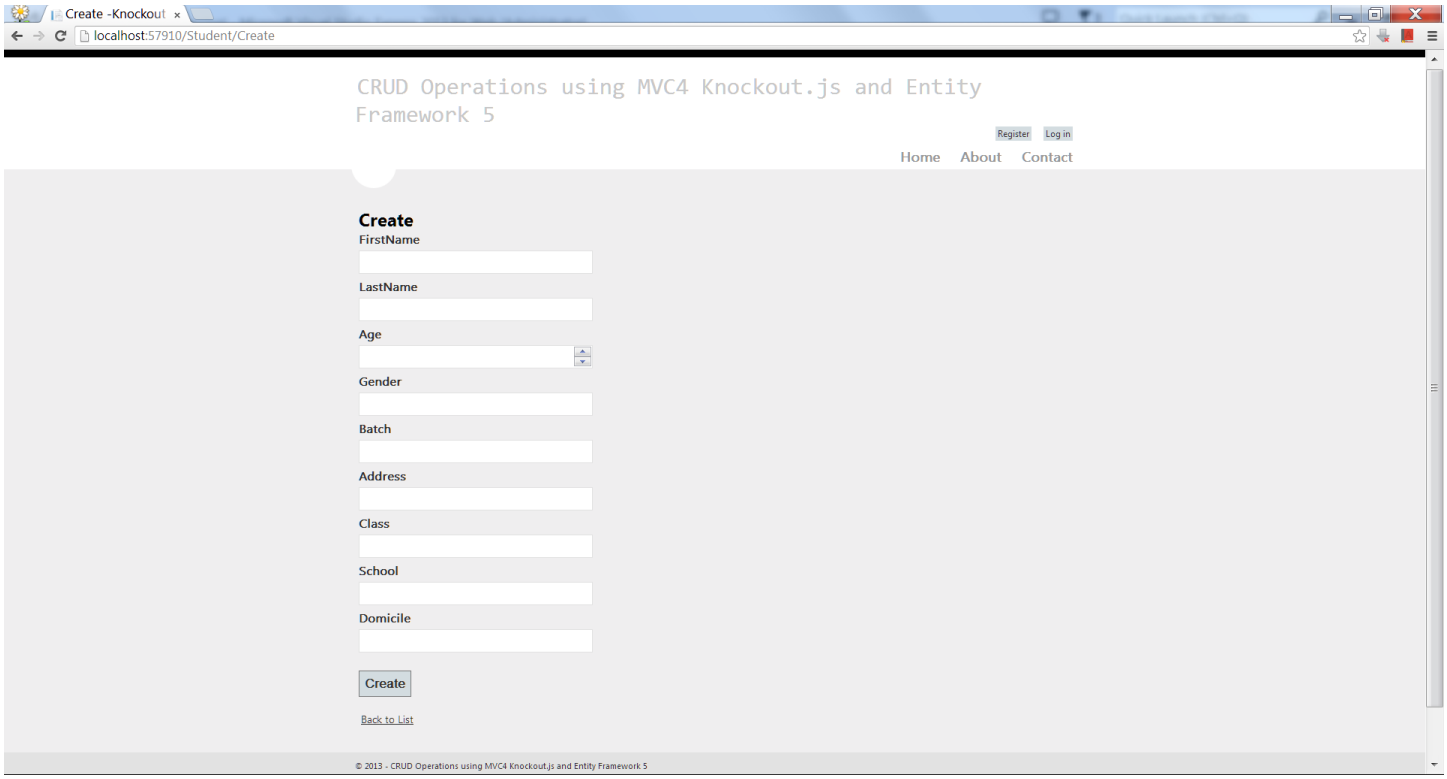
```
            <th>
              Age
            </th>
            <th>
              Gender
            </th>
            <th>
              Batch
            </th>
            <th>
              Address
            </th>
            <th>
              Class
            </th>
            <th>
              School
            </th>
            <th>
              Domicile
            </th>
            <th></th>
          </tr>
        </thead>
        <tbody data-bind="foreach: Students">
          <tr>
            <td data-bind="text: StudentId"></td>
            <td data-bind="text: FirstName"></td>
            <td data-bind="text: LastName"></td>
            <td data-bind="text: Age"></td>
            <td data-bind="text: Gender"></td>
            <td data-bind="text: Batch"></td>
            <td data-bind="text: Address"></td>
            <td data-bind="text: Class"></td>
            <td data-bind="text: School"></td>
            <td data-bind="text: Domicile"></td>
            <td>
              <a data-bind="click: editStudent">Edit</a>
              <a data-bind="click: deleteStudent">Delete</a>
            </td>
          </tr>
        </tbody>
    </table>
    @section Scripts {
        @Scripts.Render("~/Scripts/ViewModels/StudentListVM.js")
    }
```

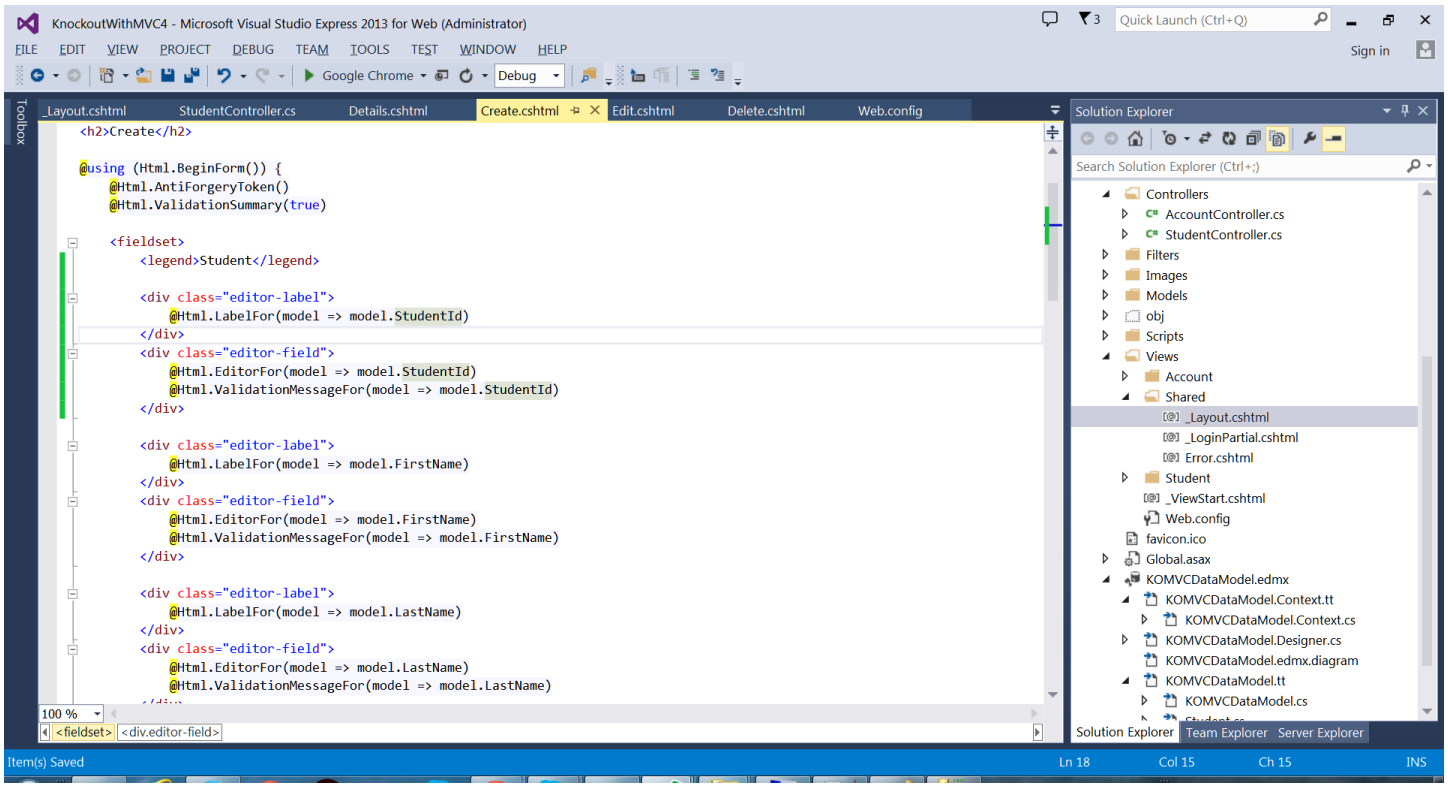
The `return JSON(db.Students.ToList(), JSONRequestBehavior.AllowGet);` code returns the student object in JSON format, for binding to the view.

All set now, you can press F5 to run the application and we see, that application runs in the same manner as it executed before,

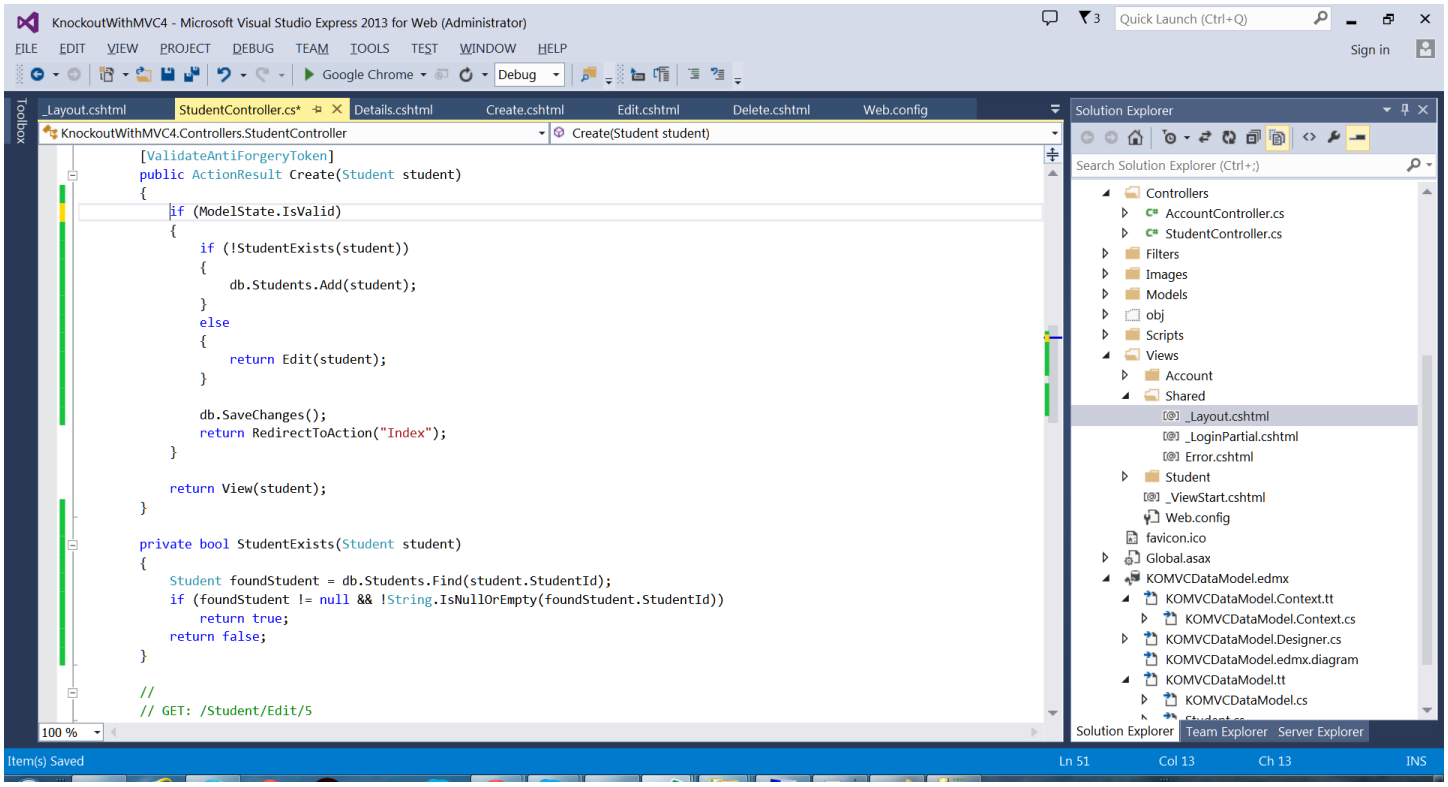


Now you can perform all the operations on this list.

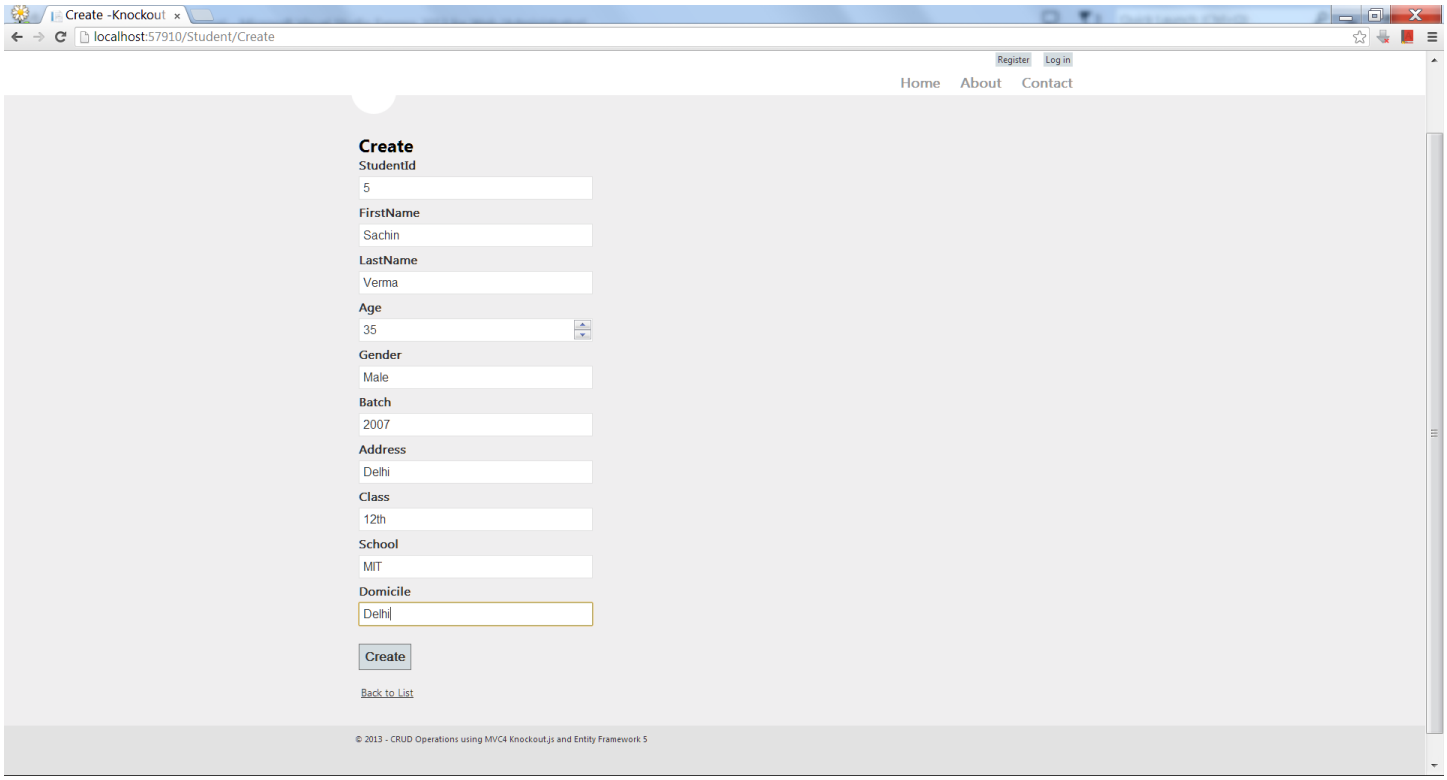




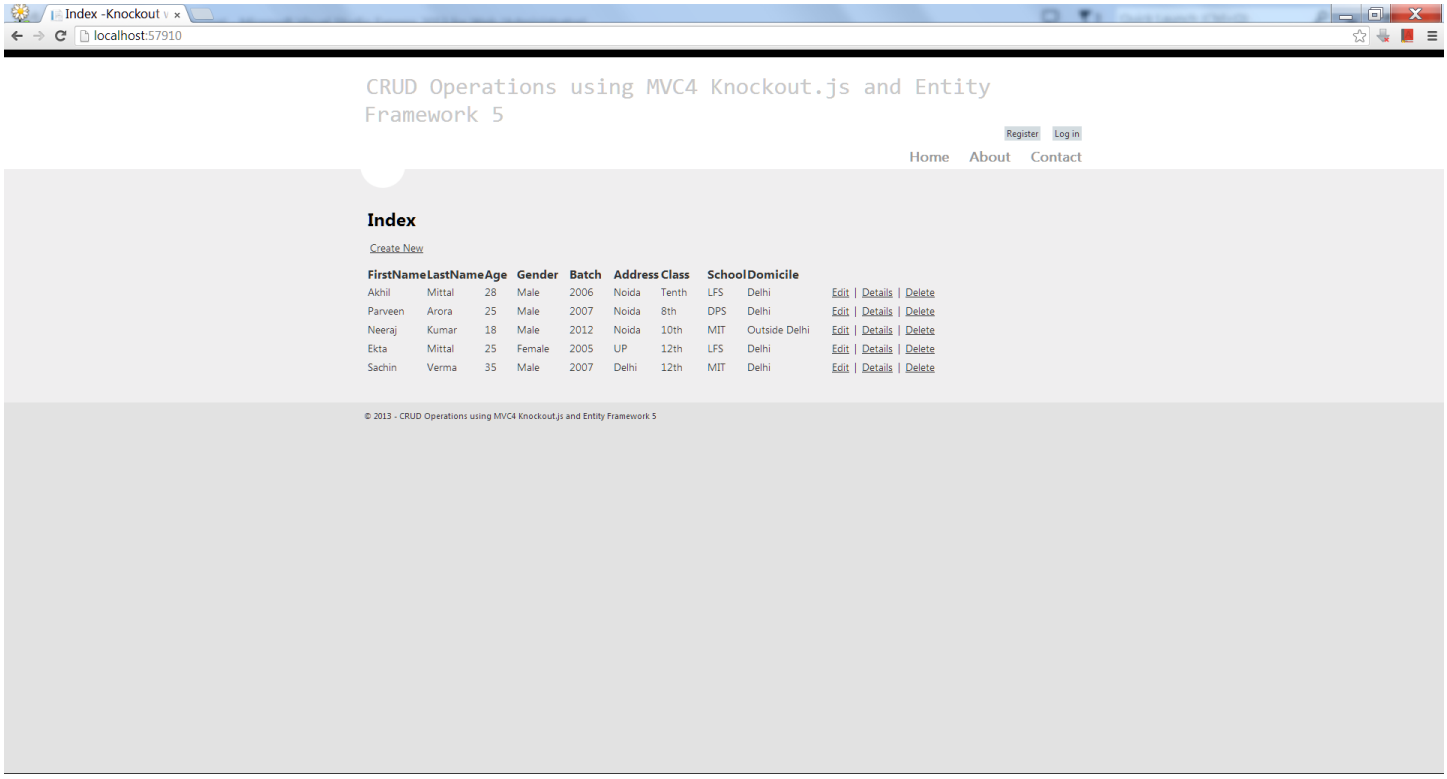
Do not type anything else other than int for student id and Age, since validation checks are missing, they may cause error.



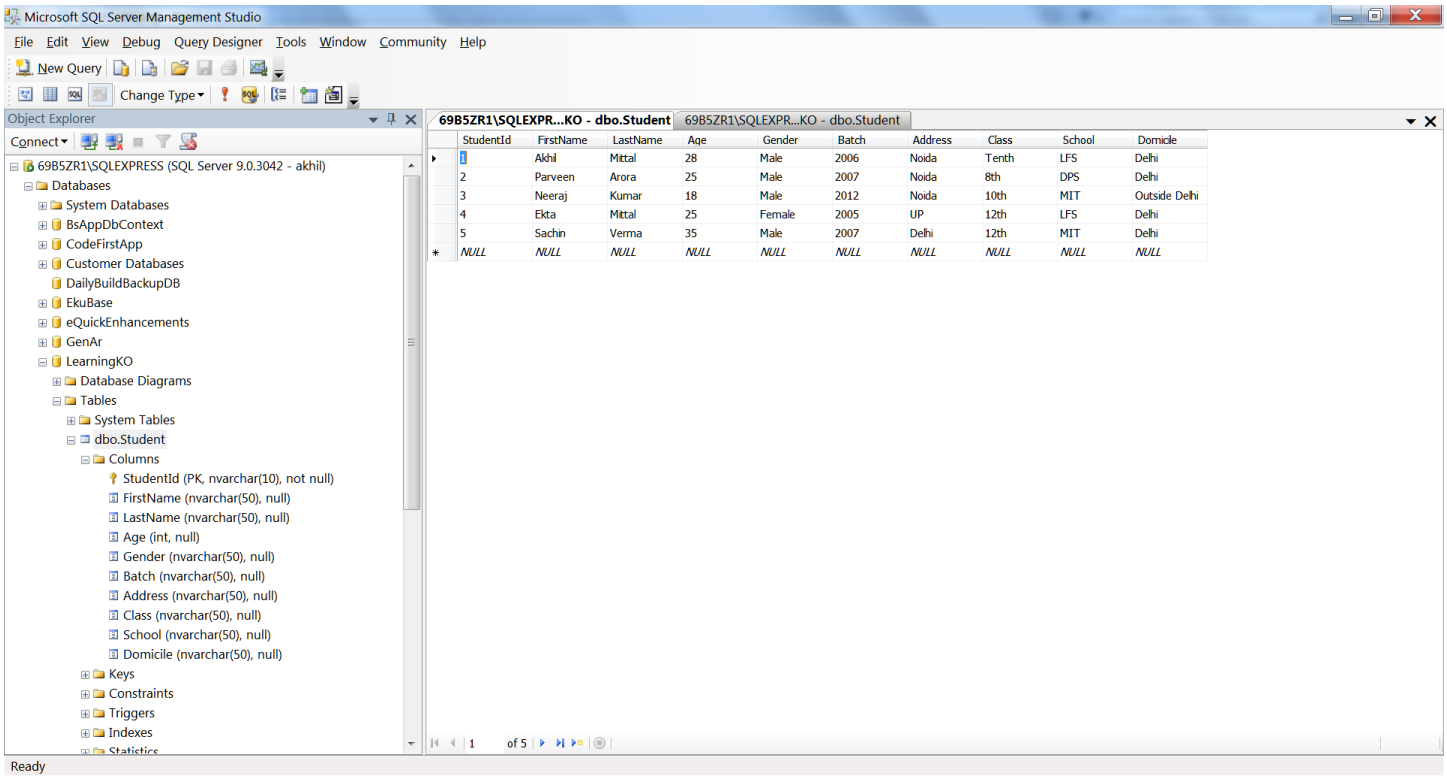
Now create new student ,



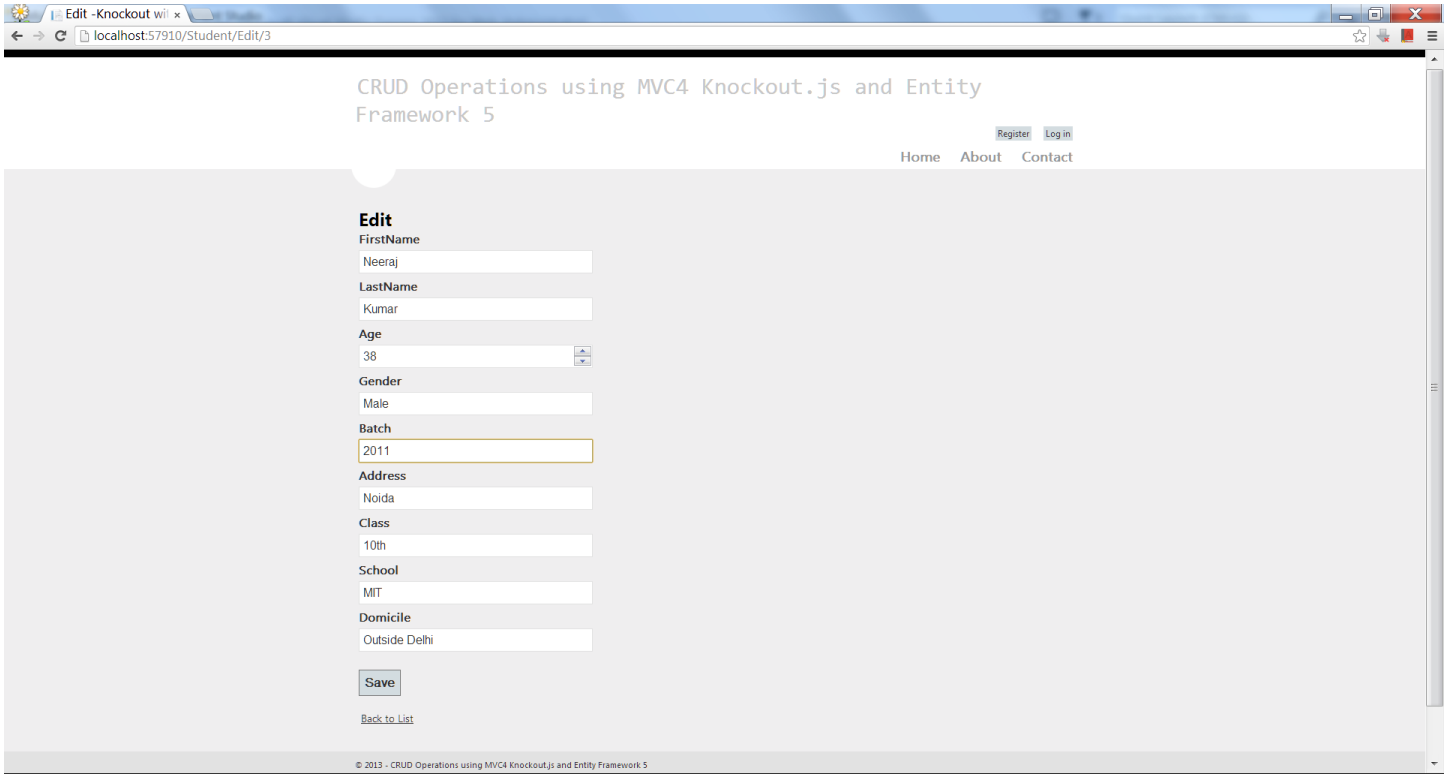
We see that the student is created successfully and added to the list,



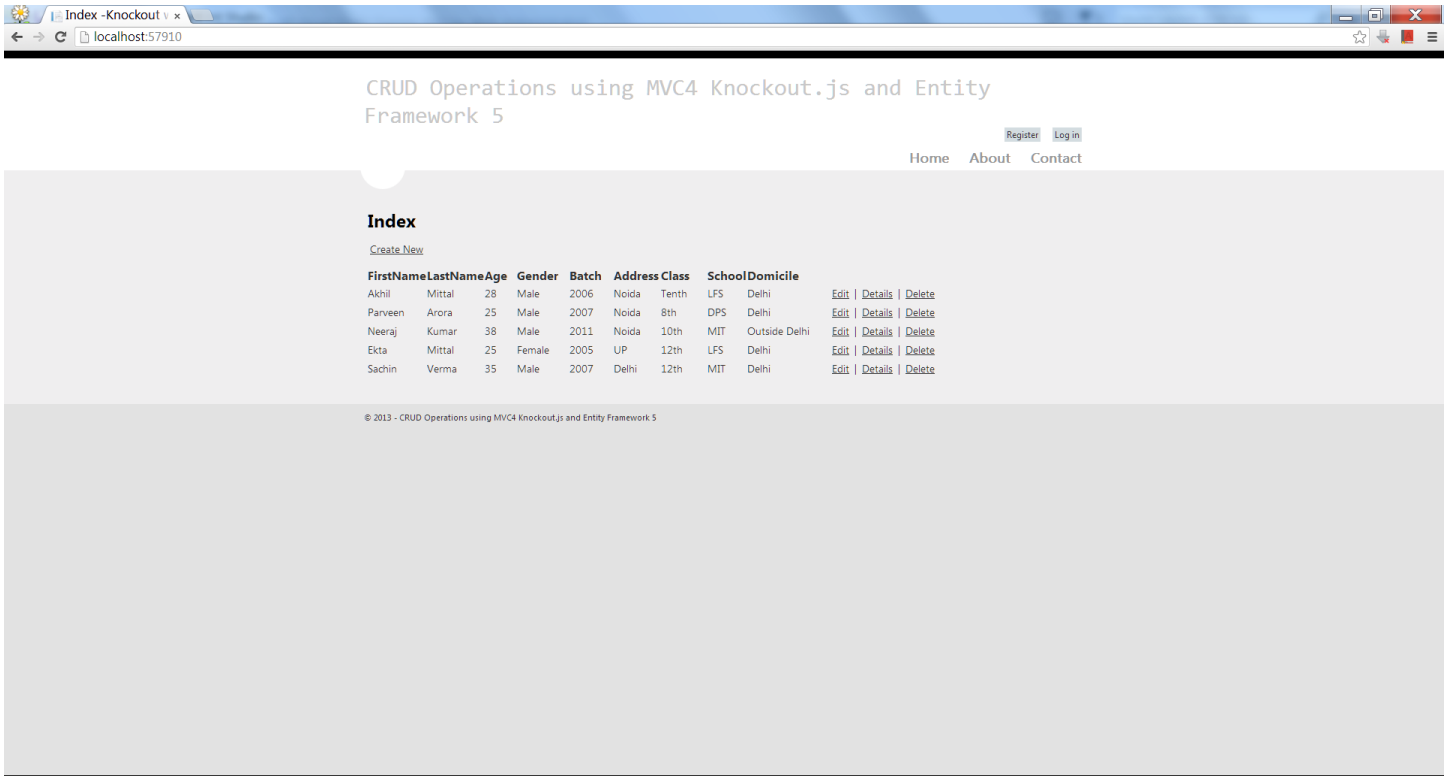
In data base,



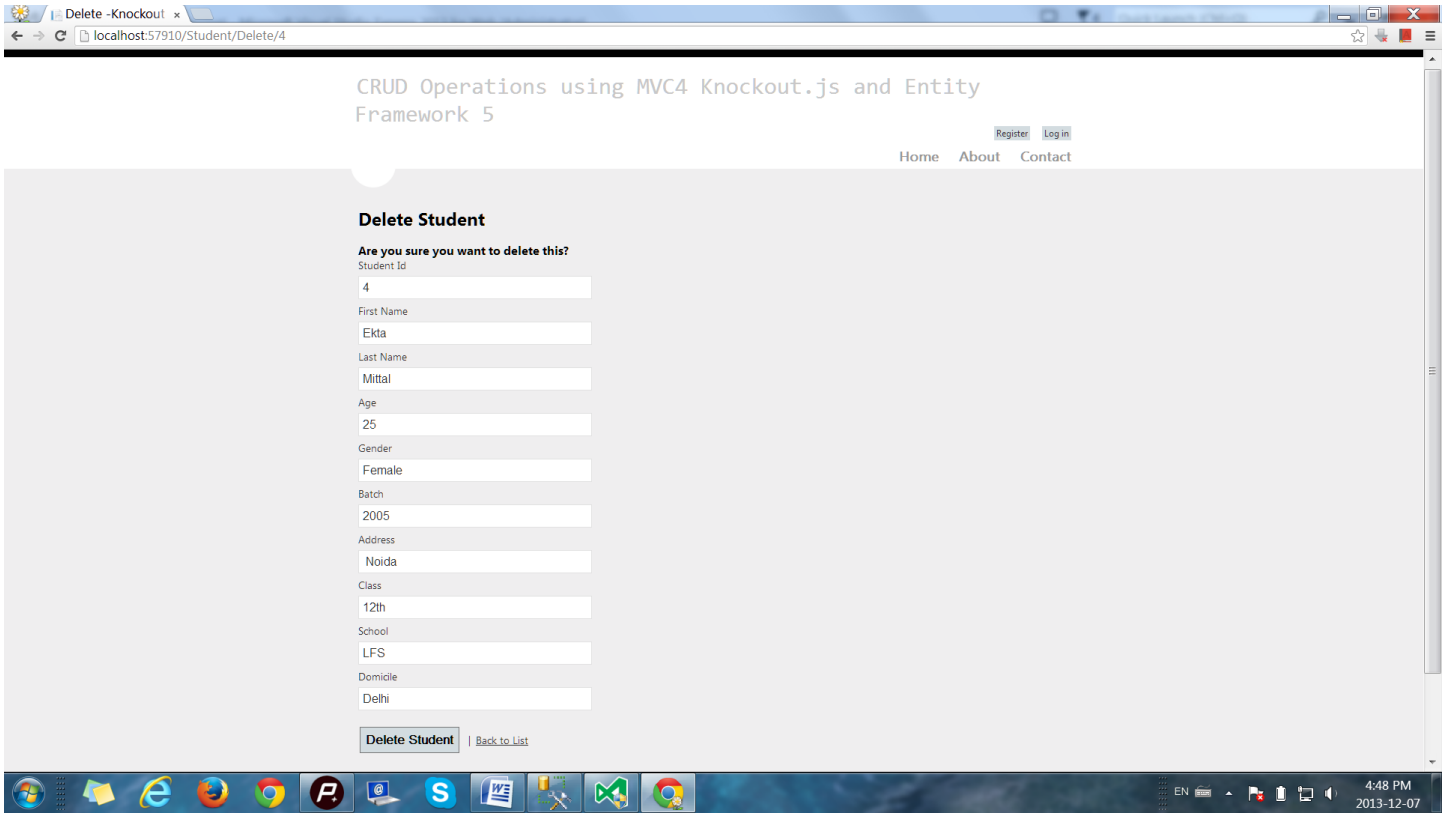
Similarly for Edit,



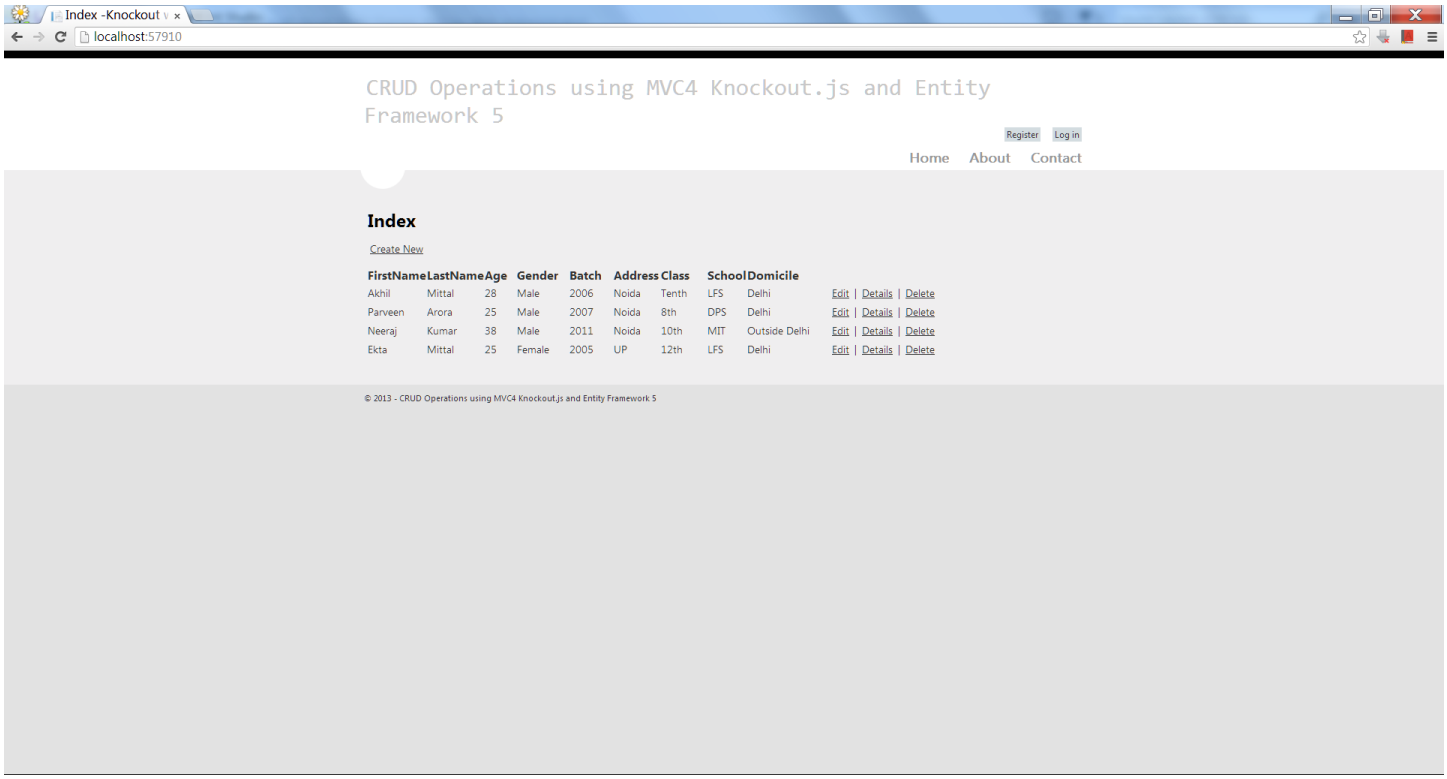
Change any field and press save.The change will be reflected in the list and data base,



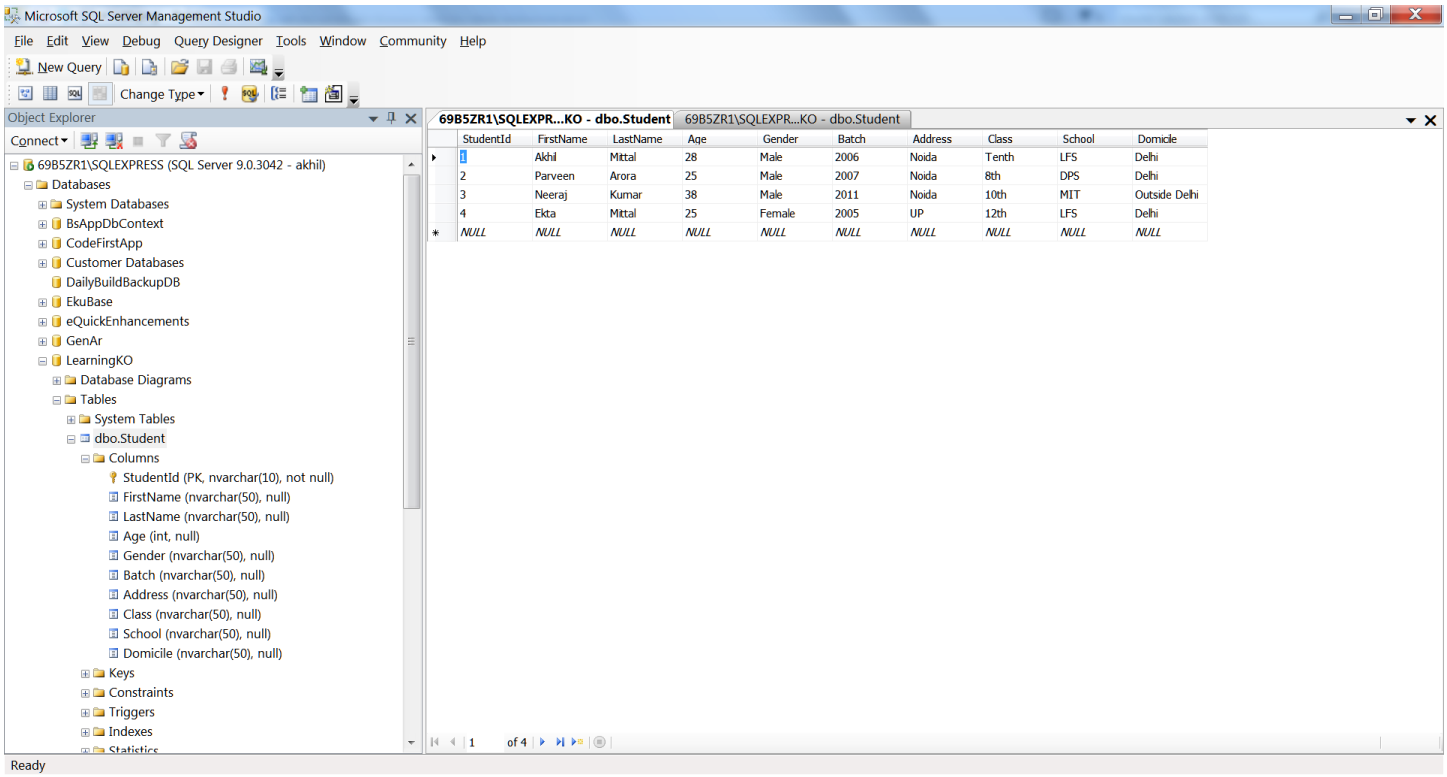
For Delete,



Student Deleted.



And in database,



Knockout Attributes Glossary :

. observable: Used to define model/entity properties. If these properties are bound with user interface and when value for these properties gets updated, automatically the UI elements bound with these properties will be updated with the new value instantaneously.

E.g. this.StudentId = ko.observable("1"); - => StudentId is the observable property. KO represent an object for the Knockout.js library.

The value of the observable is read as var id= this. StudentId ();

· **observableArray:** observableArray represents a collection of data elements which required notifications. It's used to bind with the List kind of elements.

E.g this.Students = ko.observableArray([]);

· **applyBindings**: This is used to activate knockout for the current HTML document or a specific UI element in HTML document. The parameter for this method is the view-model which is defined in JavaScript. This ViewModel contains the observable, observableArray and various methods.

Various other types of binding are used in this chapter:

. click: Represents a click event handler added to the UI element so that JavaScript function is called.

. value: This represents the value binding with the UI element's value property to the property defined into the ViewModel.

The value binding should be used with <input> , <select> , <textarea>

. visible: This is used to hide or unhide the UI element based upon the value passed to it's binding.

. Text: This represent the text value of the parameter passed to the UI element.

Conclusion:

In this book , we learnt a lot of things about MVC, Entity Framework and Knockout.JS .We did practical hands on by creating a CRUD operations application too.

Therefore we can mark it as tasks done.

Index