Create n-tier Architecture in ASP.NET MVC Application

Download Source Code

Introduction

In this article we will learn how to create n-tier architecture in ASP.NET MVC application. You should have basic understanding creating an MVC Application in .NET. We will also create a generic Database Layer which can be used in any other application. I will not be using any ORM framework for Database connection and CRUD operations. Lets stick to the basics.

Lets Start the Fun

1. Create the MVC Application in Visual Studio. You will see usual Models, View and Controller folder in the project.
2. Now lets create our database table. We will name it Employee Table and it would like this

```
CREATE TABLE [dbo].[EmployeeTable](
    [EmpId] [int] NOT NULL,
    [EmpName] [varchar](50) NOT NULL,
    [Designation] [char](10) NULL)
```

3. Now we will create our Model class. Right click on the Models folder and Add a class and name it Employee. Our Employee class should look like thi

Recent Posts

› Understanding .NET Core, .NET Standard and .NET Framework

› Extend Factory Design Pattern to implement Base and Derive class

› Implement Dependency Injection using Unity in ASP.NET MVC Controller

› Create n-tier Architecture in ASP.NET MVC Application

› Custom validation of Date in ASP.NET MVC

Categories

› .NET Core

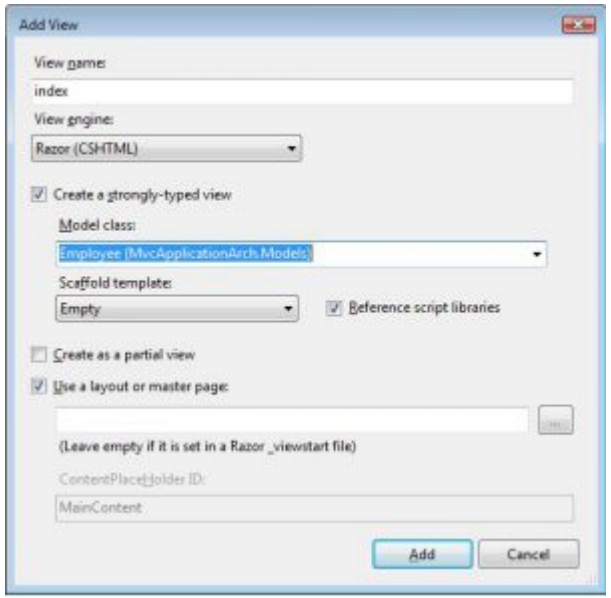
› ASP.NET MVC

```
        public string Department { get; set; }

    }
```

So far So good. We have created a class which has a similar structure to our database table.

4. Now we will Add our View which would take the input from User and insert those values in the database table Employee. I am not adding any validation on the HTML form in this article, if you want to learn how to perform validations in HTML form in ASP.NET MVC please read my article Custom validation of Date in ASP.NET MVC.
   Before adding a view make sure to compile your project. Add a folder in Views folder and name it Home. Now Add a view in this Home folder and name it index and select the checkbox “Create a strongly typed view” and Select the Employee class from the drop down. Your screen should look like this:



5. Our index.cshtml file should look like this:

```
@model MvcApplicationArch.Models.Employee

```

```
@{ using (Html.BeginForm("Index", "Home", FormMethod.Post, new { e
 {
       @Html.ValidationSummary(true)
    <table>
    <tr><td>Employee Id</td><td>@Html.TextBoxFor(m => m.EmployeeId
    <tr><td>Employee Name</td><td>@Html.TextBoxFor(m => m.Employee
    <tr><td>Department</td><td>@Html.TextBoxFor(m => m.Department)

    <tr><td></td><td><input type="submit" value="Submit"/></td></t
    </table>
 }
}

<div>
@ViewData["Message"]
</div>
```

So now our Model and View is ready but without controller the application is incomplete. In next steps we will not add the Controller rather we will add Layers which will make our application n-tier. In normal MVC applications many Developers would write the business logic in the Controller class and some even write the Database CRUD operations in the controller class. This violates the principle of “Single Responsibility”. The responsibility of Controller is like an entry gate. It acts as an entry to the applications. User inputs come to the controller but controller should not perform any business logic on this data. There should be separate classes to perform Business Logic and Database handling. In next steps we will create our Database class and Business Class. That was bit lengthy explanation but it was worth it. Wasn’t it?

6. In your project Add a new Folder and name it DataAccessLayer. Now add a new class in this folder and name it SQLDBAccess. Now this class will perform all the activities related to the Database. This class will connect to the Database, Call the SQL Stored Procedures to get the Output, Call the SQL Stored Procedures which will perform Insert / Delete / Update statements on the Database tables. Our code of SQLDBAccess class look like this:

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

```
8   using System.Configuration;
9
10  namespace MvcApplicationArch.DataAccessLayer
11  {
12      public class SQLDBAccess
13      {
14          #region Class Variables
15          private string _connString = string.Empty;
16          private SqlConnection _sqlConn = null;
17
18          #endregion
19
20          #region Private methods
21          private string GetConnectionString()
22          {
23              try
24              {
25                  if(String.IsNullOrEmpty(_connString))
26                  _connString = ConfigurationManager.ConnectionStri
27                  return _connString;
28              }
29              catch (Exception)
30              {
31
32                  throw;
33              }
34          }
35
36          private SqlConnection GetConnection()
37          {
38              var sqlConn = new SqlConnection(GetConnectionString()
39              return sqlConn;
40          }
41
42          private void CloseConnection()
43          {
44              try
45              {
46                  if (_sqlConn != null && _sqlConn.State != Connect
47                  {
48                  _sqlConn.Close();
49                  }
50              }
51              catch (Exception)
52              {
53
```

```
58
59          #region Public Mehtods
60
61          /// <summary>
62          /// Call stored procedure with Sql Command. This will cor
63          /// </summary>
64          /// <param name="spName"></param>
65          /// <param name="command"></param>
66          /// <returns></returns>
67          public DataSet GetDataSet(string spName, SqlCommand comma
68          {
69              var dataSet = new DataSet();
70              //var dataAdapter = new SqlDataAdapter();
71              try
72              {
73                  using (var sqlConn = GetConnection())
74                  {
75                      command.Connection = sqlConn;
76                      command.CommandText = spName;
77                      command.CommandType = CommandType.StoredProce
78                      sqlConn.Open();
79                      using (var dataAdapter = new SqlDataAdapter()
80                      {
81                          dataAdapter.SelectCommand = command;
82                          dataAdapter.Fill(dataSet);
83                      }
84                  }
85              }
86              catch (Exception)
87              {
88
89                  throw;
90              }
91              return dataSet;
92          }
93
94          /// <summary>
95          /// Call stored proc without params
96          /// </summary>
97          /// <param name="spName"></param>
98          /// <returns></returns>
99          public DataSet GetDataSet(string spName)
100         {
101             var dataSet = new DataSet();
102             try
```

```
107                 }
108             }
109             catch (Exception)
110             {
111 
112                 throw;
113             }
114             return dataSet;
115         }
116 
117         public void ExecuteNonQuery(SqlCommand command)
118         {
119             try
120             {
121                 using (var sqlConn = GetConnection())
122                 {
123                     command.Connection = sqlConn;
124                     sqlConn.Open();
125                     command.ExecuteNonQuery();
126                 }
127             }
128             catch (Exception)
129             {
130 
131                 throw;
132             }
133         }
134 
135         public void ExecuteNonQuery(string spName, SqlCommand com
136         {
137             try
138             {
139                 command.CommandText = spName;
140                 command.CommandType = CommandType.StoredProcedure
141                 ExecuteNonQuery(command);
142             }
143             catch (Exception)
144             {
145 
146                 throw;
147             }
148         }
149         #endregion
150     }
151 }
```

is ready. This code is very generic in nature and can be reused in multiple applications where MS SQL Server is used. To interact with Oracle Database we can write a new class in a similar fashion but which will use Oracle driver to connect to Oracle Database.

7. So now our Database table is ready, Our Model class is ready, we have created the view also and our Database layer is also ready. Now we will create the Business Layer. This layer will act as a medium between Controller and Database layer. When user inputs the data in the HTML form and clicks on the Submit, that data comes to the Controller. So now Controller will not perform any action on this data rather it will pass the Data to the Business Layer. Any Business Logic on this data should be handled by the Business Layer.

8. In the Project Add a new Folder and name it Business Layer. Add a new Class in this folder and name it BusinessAccess. Now we will create a method in this BusinessAccess class which get the Input as the data entered by the user and pass this data to the Database Layer which in turn will call the Stored Procedure to insert the data in the database. If any Business logic needs to be performed on this data then BusinessAccess layer is place to write that logic.
   Our BusinessAccess code look like this:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Web;
5  using System.Data;
6  using System.Data.SqlClient;
7  using MvcApplicationArch.Models;
8  using MvcApplicationArch.DataAccessLayer;
9
10 namespace MvcApplicationArch.BusinessLayer
11 {
12     public class BusinessAccess
13     {
14         public bool InsertEmployee(Employee emp)
15         {
16             bool flag = false;
17             var sqlDb = new SQLDBAccess();
18             using (var command = new SqlCommand())
19             {
```

```
                flag = true;
            }
            return flag;
        }
    }
}
```

The InsertEmployee method gets the Employee class object as input parameter and pass the values to the Database layer.

9. Now only thing left is to create the Controller and link it with the BusinessAccess layer. In the Controller folder add a new Controller and Name it Home. Create a new method in the Controller which will take the object of the Employee class as an input and will be called by the HTML Post method from the View. Our Controller code look like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using MvcApplicationArch.Models;
using MvcApplicationArch.BusinessLayer;

namespace MvcApplicationArch.Controllers
{
    public class HomeController : Controller
    {

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

        [HttpPost]
        public ActionResult Index(Employee employee)
        {
            var business = new BusinessAccess();
            if (business.InsertEmployee(employee))
            {
                @ViewData["Message"] = "Employee inserted";
            }
            else
```

```
32              }
33          }
34      }
```

Conclusion

In this article we learnt how to create n-tier architecture in ASP.NET MVC applications. This article only explained how to insert the data in the Database table. The code of SQL Server Stored Procedure is available in the downloadable zip file. Here i have explained how to create different classes and implement the Single Responsibility Principle. I hope you enjoyed reading this article. Any issues or suggestions please let me know in the comment section.

← CUSTOM VALIDATION OF DATE IN ASP.NET MVC

IMPLEMENT DEPENDENCY INJECTION USING UNITY IN ASP.NET MVC CONTROLLER →

One thought on “Create n-tier Architecture in ASP.NET MVC Application”

Generic cialis says:

26/03/2018 at 12:41 pm

Hi there, I found your site via Google while searching for a related topic, your web site came up, it looks great. I’ve bookmarked it in my google bookmarks.

Reply

Comment

Name *

Email *

Website

Post Comment

Back To Top