ASP.NET MVC 5: Building Your First Web Application (A Beginner's Guide)

Vincent Maverick
(C# Corner MVP)

About Author

Vincent Maverick is a Microsoft ASP.NET MVP since 2009, C# Corner MVP and DZone MVB. He works as a Technical Lead Developer in a research and development company. He works on ASP.NET, C#, MS SQL, Entity Framework, LINQ, AJAX, JavaScript, JQuery, HTML5, CSS, and other technologies.

Vincent Maverick

INDEX

Page no

ASP.NET MVC 5: Building Your First Web Application (A Beginner's Guide)

Introduction

Technologies are constantly evolving and as developer we need to cope up with what's the latest or at least popular nowadays. As a starter you might find yourself having a hard-time catching up with latest technologies because it will give you more confusion as to what sets of technologies to use and where to start. We know that there are tons of resources out there that you can use as a reference to learn but you still find it hard to connect the dots in the picture. Sometimes you might thought of losing the interest to learn and gave up. If you are confused and no idea how to start building a web app from scratch then this book is for you.

ASP.NET MVC 5: Building Your First Web Application is targeted to beginners who want to jump on ASP.NET MVC 5 and get their hands dirty with practical example. I've written this book in such a way that it's easy to follow and understand by providing step-by-step process on creating a simple web application from scratch and deploying it to IIS Web Server. As you go along and until such time you finished following the book, you will learn how to create a database using SQL Server, learn the concept of ASP.NET MVC and what it is all about, learn Entity Framework using Database-First approach, learn basic jQuery and AJAX, learn to create pages such as Registration, Login, Profile and Admin page where user can modify, add and delete information. You will also learn how to install and deploy your application in IIS Web Server.

Prerequisites

Before you go any further make sure that you have basic knowledge on the following technologies:

- SQL Server
- Visual Studio
- ASP.NET in general
- Basic understanding of ASP.NET MVC
- Entity Framework
- C#
- Basics on HTML, CSS and JavaScript/jQuery

Environment and Development Tools

The following are the tools and environment settings that I am using upon building the web app.

- Windows 8.1
- IIS8
- Visual Studio 2015
- SQL Express 2014

Getting Started

This book will guide you through the basic steps on creating a simple web application using ASP.NET MVC 5 with real-world example using Entity Framework Database-First approach. I'll try to keep this demo as simple as possible so starters can easily follow. By "simple" I mean limit the talking about theories and concepts, but instead jumping directly into the mud and get your hands dirty with code examples.

ASP.NET MVC Overview

Before we start building an MVC application let's talk about a bit of MVC first because it is very important to know how the MVC framework works.

What is ASP.NET MVC?

ASP.NET MVC is part of ASP.NET framework. The figure below will give you a high level look to where ASP.NET MVC resides within the ASP.NET framework.

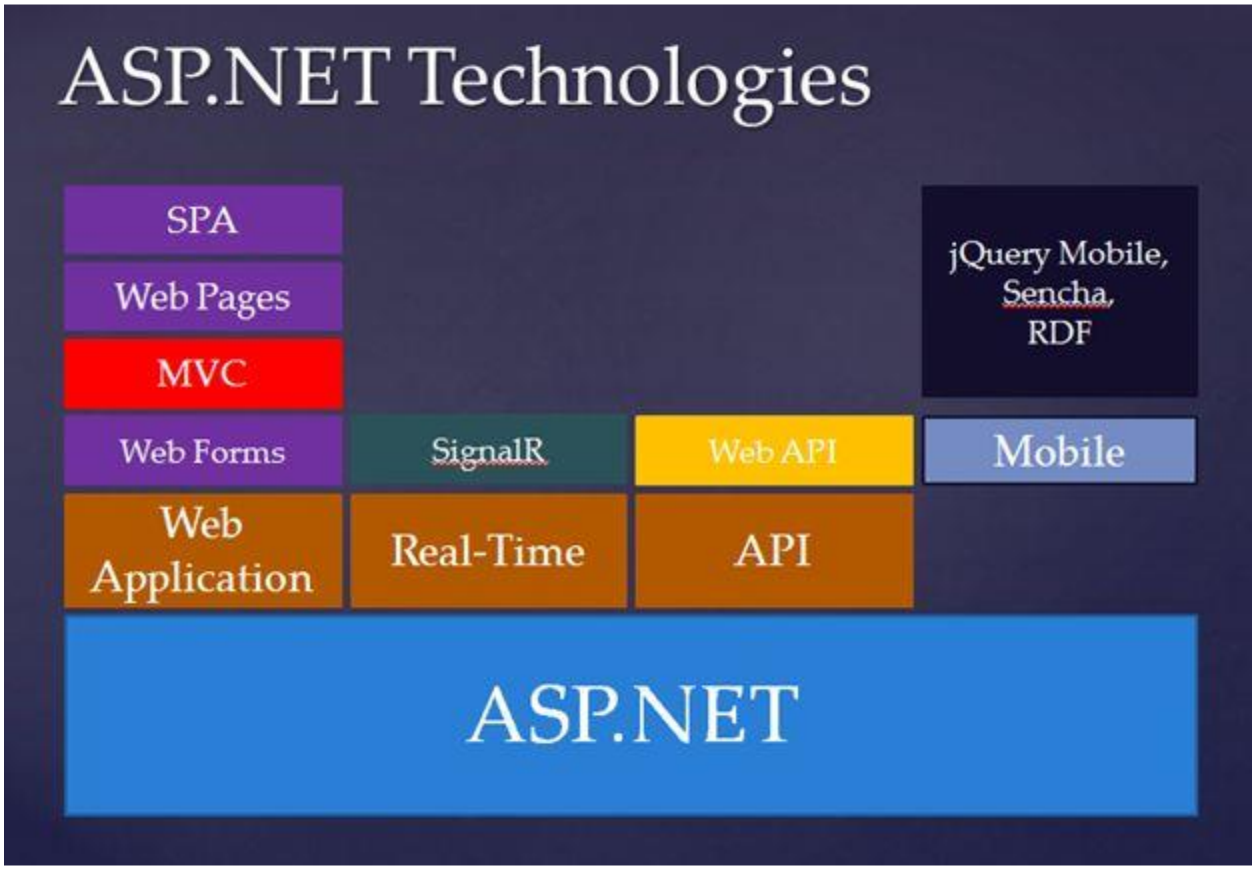


Figure 1: The ASP.NET technologies

You will see that ASP.NET MVC sits on top of ASP.NET. ASP.NET MVC gives you a powerful, pattern-based way to build dynamic websites that enables a clean separation of concerns and that gives you full control over mark-up for enjoyable and agile development.

To make it more clear, here's how I view the high-level process of MVC:

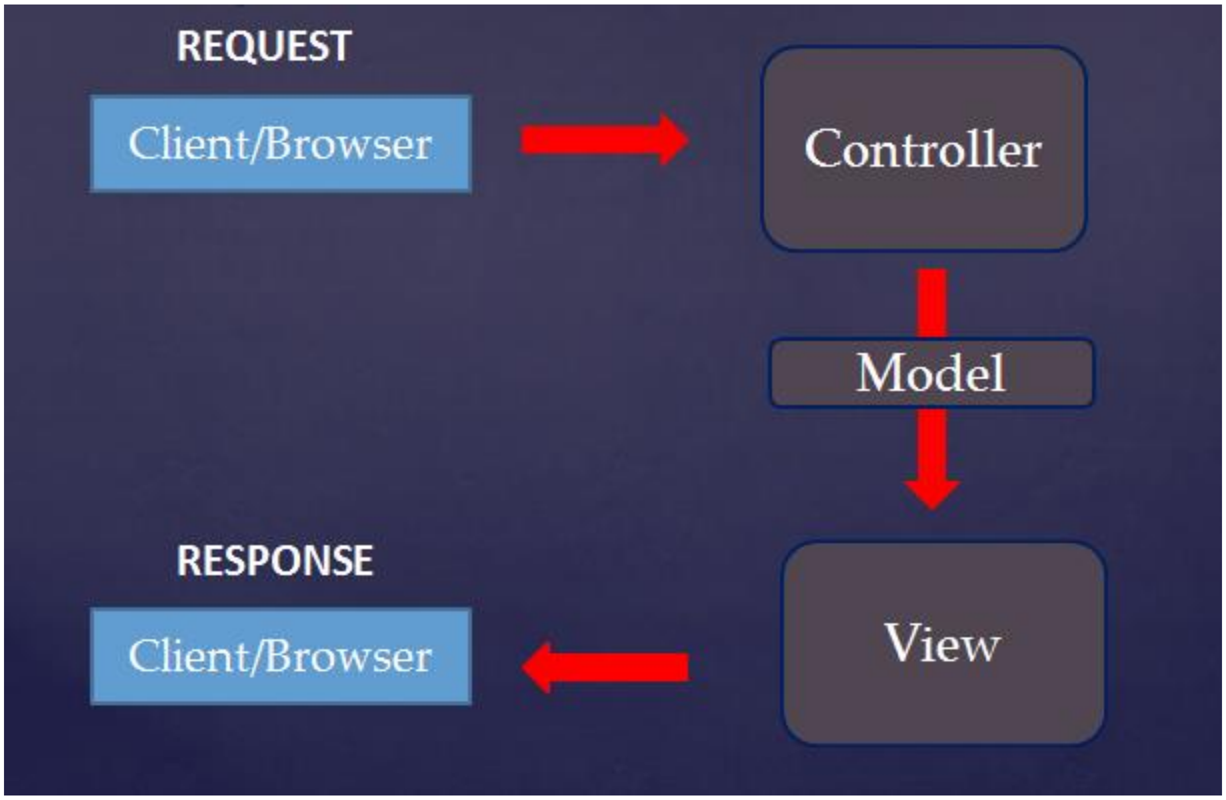


Figure 2: MVC architecture flow

Unlike in ASP.NET WebForms that a request is going directly to a page file (.ASPX), in MVC when a user request a page it will first talk to the Controller , process data when necessary and returns a Model to the View for the user to see.

What are Models?

Model objects are the parts of the application that implement the logic for the application domain data. Often, model objects retrieved and store model state in database.

What are Controllers?

Controllers are the components that handle user interaction, work with the model, and ultimately select a view to render in the browser.

What are Views?

Views are the components that display the application's user interface (UI), typically this UI is created from the model data.

To put them up together, the **M** is for Model, which is typically where the BO (Business Objects), BL (Business Layer) and DAL (Data Access) will live. Note that in typical layered architecture, your BL and DAL should be in separate project. The **V** is for View, which is what the user sees. This could simply mean that any UI and client-side related development will live in the View including HTML, CSS and JavaScript. The **C** is the Controller, which orchestrates the flow of logic. For example if a user clicks a button that points to a specific URL, that request is mapped to a Controller Action method that is responsible for handling any logic required to service the request, and returning a response- typically a new View, or an update to the existing View.

If you are still confused about Models, Views and Controllers then don't worry because I will be covering how each of them relates to each other by providing code examples. So keep reading ☺

Creating a Database

Open SQL Server or SQL Server Express Management Studio and then create a database by doing the following:

- Right click on the Databases folder
- Select New Database
- Enter a database name and then click OK. Note that in this demo I used "DemoDB" as my database name.

The "DemoDB" database should be created as shown in the figure below:

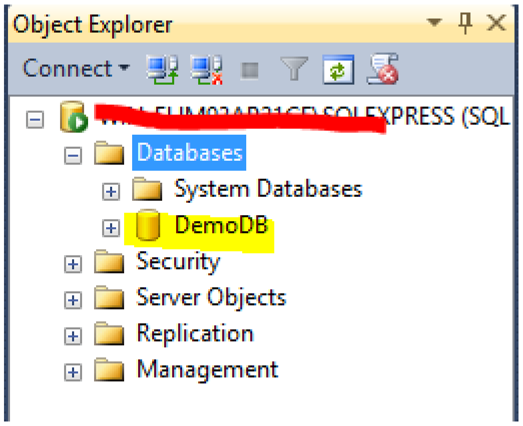


Figure 3: New database created

Alternatively, you can also write a SQL script to create a database. For example:

```
CREATE DATABASE DemoDB;
```

Creating Database Tables

Now open a New Query window or just press CTRL + N to launch the query window and then run the following scripts:

LOOKUPRole table

```
USE [DemoDB]
GO

CREATE TABLE [dbo].[LOOKUPRole](
       [LOOKUPRoleID] [int] IDENTITY(1,1) NOT NULL,
       [RoleName] [varchar](100) DEFAULT '',
       [RoleDescription] [varchar](500) DEFAULT '',
       [RowCreatedSYSUserID] [int] NOT NULL,
       [RowCreatedDateTime] [datetime]  DEFAULT GETDATE(),
       [RowModifiedSYSUserID] [int] NOT NULL,
       [RowModifiedDateTime] [datetime] DEFAULT GETDATE(),
       PRIMARY KEY (LOOKUPRoleID)
       )
GO
```

Adding test data to LOOKUPRole table

```
INSERT INTO LOOKUPRole
(RoleName,RoleDescription,RowCreatedSYSUserID,RowModifiedSYSUserID)
          VALUES ('Admin','Can Edit, Update, Delete',1,1)
INSERT INTO LOOKUPRole
(RoleName,RoleDescription,RowCreatedSYSUserID,RowModifiedSYSUserID)
          VALUES ('Member','Read only',1,1)
```

SYSUser table

```
USE [DemoDB]
GO

CREATE TABLE [dbo].[SYSUser](
       [SYSUserID] [int] IDENTITY(1,1) NOT NULL,
       [LoginName] [varchar](50) NOT NULL,
       [PasswordEncryptedText] [varchar](200) NOT NULL,
       [RowCreatedSYSUserID] [int] NOT NULL,
       [RowCreatedDateTime] [datetime] DEFAULT GETDATE(),
       [RowModifiedSYSUserID] [int] NOT NULL,
       [RowModifiedDateTime] [datetime] DEFAULT GETDATE(),
       PRIMARY KEY (SYSUserID)
)

GO
```

SYSUserProfile table

```
USE [DemoDB]
GO
```

```
CREATE TABLE [dbo].[SYSUserProfile](
	[SYSUserProfileID] [int] IDENTITY(1,1) NOT NULL,
	[SYSUserID] [int] NOT NULL,
	[FirstName] [varchar](50) NOT NULL,
	[LastName] [varchar](50) NOT NULL,
	[Gender] [char](1) NOT NULL,
	[RowCreatedSYSUserID] [int] NOT NULL,
	[RowCreatedDateTime] [datetime] DEFAULT GETDATE(),
	[RowModifiedSYSUserID] [int] NOT NULL,
	[RowModifiedDateTime] [datetime] DEFAULT GETDATE(),
	PRIMARY KEY (SYSUserProfileID)
	)
GO

ALTER TABLE [dbo].[SYSUserProfile]  WITH CHECK ADD FOREIGN KEY([SYSUserID])
REFERENCES [dbo].[SYSUser] ([SYSUserID])
GO
```

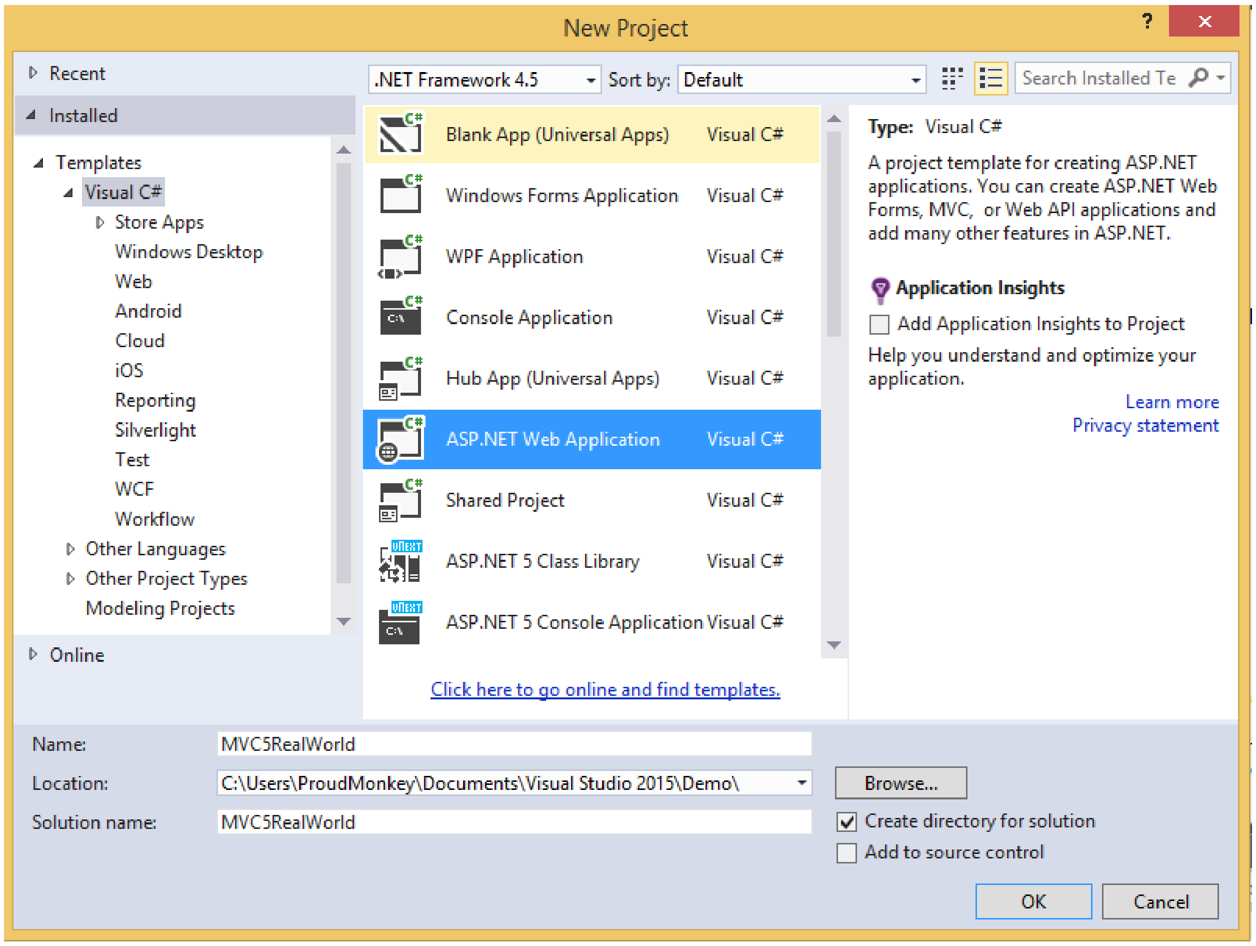
And finally, the SYSUserRole table

```
USE [DemoDB]
GO

CREATE TABLE [dbo].[SYSUserRole](
	[SYSUserRoleID] [int] IDENTITY(1,1) NOT NULL,
	[SYSUserID] [int] NOT NULL,
	[LOOKUPRoleID] [int] NOT NULL,
	[IsActive] [bit] DEFAULT (1),
	[RowCreatedSYSUserID] [int] NOT NULL,
	[RowCreatedDateTime] [datetime] DEFAULT GETDATE(),
	[RowModifiedSYSUserID] [int] NOT NULL,
	[RowModifiedDateTime] [datetime] DEFAULT GETDATE(),
	PRIMARY KEY (SYSUserRoleID)
)
GO

ALTER TABLE [dbo].[SYSUserRole]  WITH CHECK ADD FOREIGN KEY([LOOKUPRoleID])
REFERENCES [dbo].[LOOKUPRole] ([LOOKUPRoleID])
GO

ALTER TABLE [dbo].[SYSUserRole]  WITH CHECK ADD FOREIGN KEY([SYSUserID])
REFERENCES [dbo].[SYSUser] ([SYSUserID])
GO
```

That's it. We have just created four (4) database tables. The next step is to create the web application.

Adding a New ASP.NET MVC 5 Project

Go ahead and fire up Visual Studio 2015 and select File > New > Project. Under "New Project" dialog, select Templates > Visual C# > ASP.NET Web Application. See the figure below for your reference.



New Project

.NET Framework 4.5 | Sort by: Default | Search Installed Te

Recent

Installed

- Templates
  - Visual C#
    - Store Apps
    - Windows Desktop
    - Web
    - Android
    - Cloud
    - iOS
    - Reporting
    - Silverlight
    - Test
    - WCF
    - Workflow
  - Other Languages
  - Other Project Types
  - Modeling Projects

Online

Blank App (Universal Apps)	Visual C#
Windows Forms Application	Visual C#
WPF Application	Visual C#
Console Application	Visual C#
Hub App (Universal Apps)	Visual C#
ASP.NET Web Application	Visual C#
Shared Project	Visual C#
ASP.NET 5 Class Library	Visual C#
ASP.NET 5 Console Application	Visual C#

Click here to go online and find templates.

Type: Visual C#

A project template for creating ASP.NET applications. You can create ASP.NET Web Forms, MVC, or Web API applications and add many other features in ASP.NET.

Application Insights

☐ Add Application Insights to Project

Help you understand and optimize your application.

Learn more

Privacy statement

Name: MVC5RealWorld

Location: C:\Users\ProudMonkey\Documents\Visual Studio 2015\Demo\

Solution name: MVC5RealWorld

Browse...

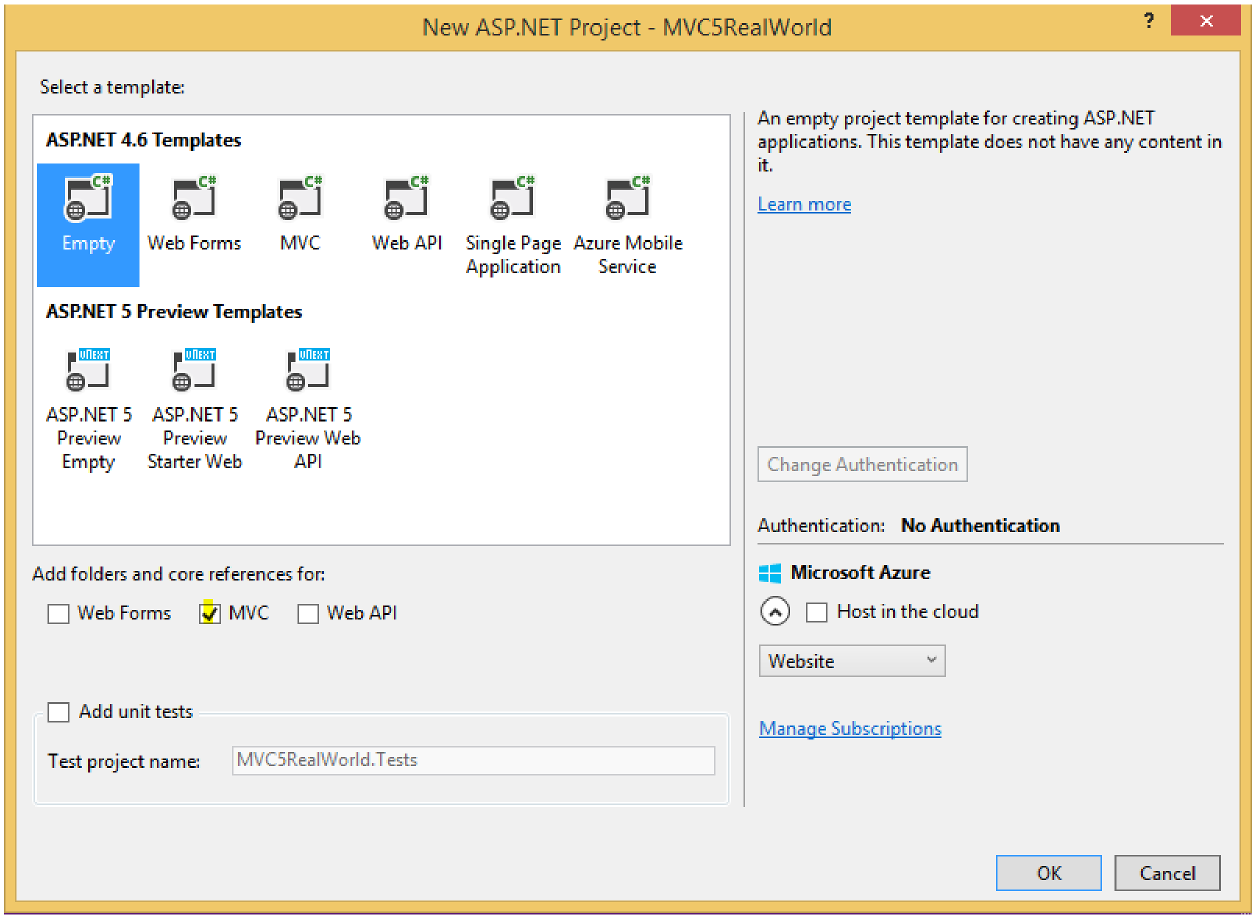
☑ Create directory for solution

☐ Add to source control

OK | Cancel

Figure 4: ASP.NET Web Application template

Name your project to whatever you like and then click OK. Note that for this demo I have named the project as "**MVC5RealWorld**". Now after that you should be able to see the "New ASP.NET Project" dialog as shown in the figure below:


New ASP.NET Project - MVC5RealWorld

Select a template:

ASP.NET 4.6 Templates

Empty | Web Forms | MVC | Web API | Single Page Application | Azure Mobile Service

ASP.NET 5 Preview Templates

ASP.NET 5 Preview Empty | ASP.NET 5 Preview Starter Web | ASP.NET 5 Preview Web API

An empty project template for creating ASP.NET applications. This template does not have any content in it.

Learn more

Change Authentication

Authentication: **No Authentication**

Microsoft Azure

☐ Host in the cloud

Website

Manage Subscriptions

Add folders and core references for:

☐ Web Forms ☑ MVC ☐ Web API

☐ Add unit tests

Test project name: MVC5RealWorld.Tests

OK | Cancel

Figure 5: New ASP.NET Project dialog

The New ASP.NET Project dialog for ASP.NET 4.6 templates allows you to select what type of project you want to create, configure any combination of ASP.NET technologies such as WebForms, MVC or Web API, configure unit test project, configure authentication option and also offers a new option to host your website in Azure cloud. Adding to that it also provide templates for ASP.NET 5.

In this book I will only be covering on creating an ASP.NET MVC 5 application. So the details of each configuration like unit testing, authentication, hosting in cloud, etc. will not be covered.

Now select “Empty” under ASP.NET 4.6 templates and then check the “MVC” option under folders and core reference as shown in Figure 5. The reason for this is that we will create an empty MVC application from scratch. Click OK to let Visual Studio generate the necessary files and templates needed for you to run an MVC application.

You should now be seeing something like below:

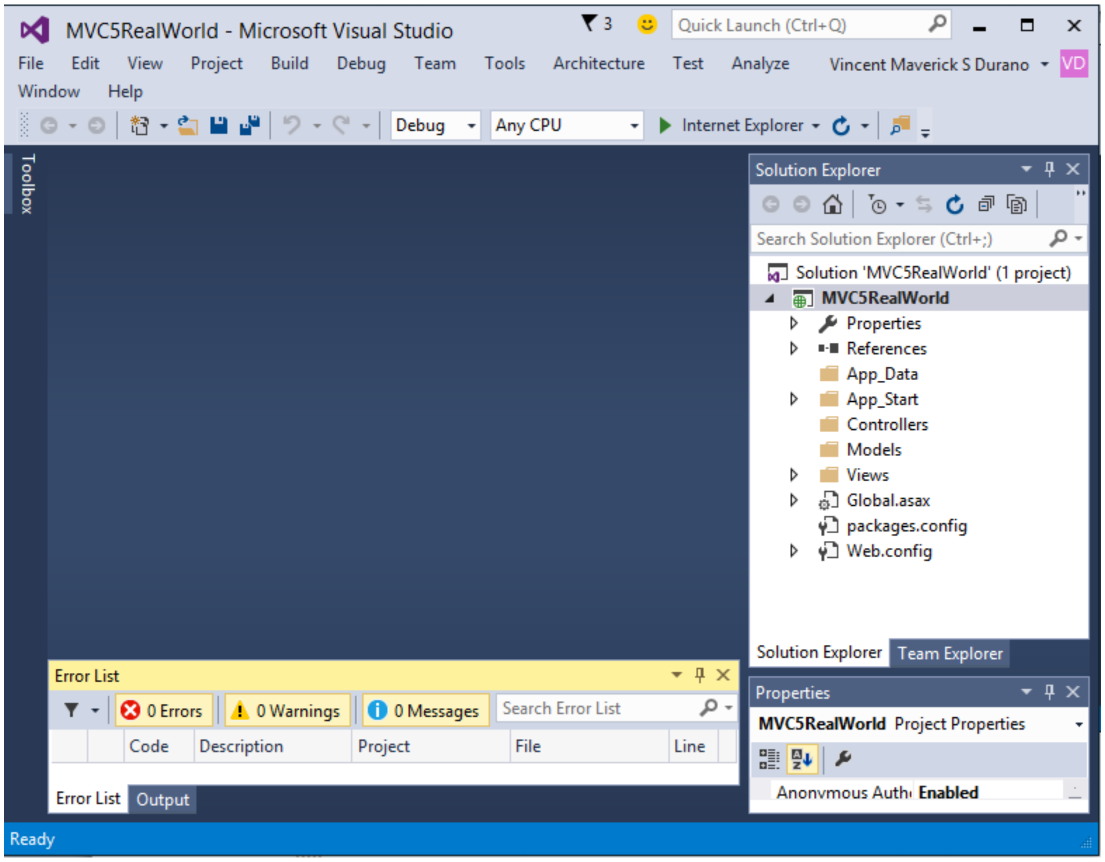


Figure 6: The MVC5RealWorld project

Setting Up the Data Access

For this example, I'm going to use Database-First with Entity Framework 6 (EF) as our data access mechanism so that we can just program against the conceptual application model instead of programming directly against our database.

Umm Huh? What do you mean?

This could simply mean that using EF you will be working with entities (class/object representation of your data structure) and letting the framework handle the basic select, update, insert & delete. In traditional ADO.NET you will write the SQL queries directly against tables/columns/procedures and you don't have entities so it's much less objecting oriented.

I prefer using EF because it provides the following benefits:

- Applications can work in terms of a more application-centric conceptual model, including types with inheritance, complex members, and relationships.

- Applications are freed from hard-coded dependencies on a particular data engine or storage schema.
- Mappings between the conceptual model and the storage-specific schema can change without changing the application code.
- Developers can work with a consistent application object model that can be mapped to various storage schemas, possibly implemented in different database management systems.
- Multiple conceptual models can be mapped to a single storage schema.
- Language-integrated query (LINQ) support provides compile-time syntax validation for queries against a conceptual model.

Creating the Entity Models

As a quick recap, a Model is just a class. Yes it's a class that implements the logic for your application's domain data. Often, model objects retrieved and store model state in database.

Now let's setup our Model folder structure by adding the following sub-folders under the "Models" folder:

- DB
- EntityManager
- ViewModel

Our model structure should look something like below:

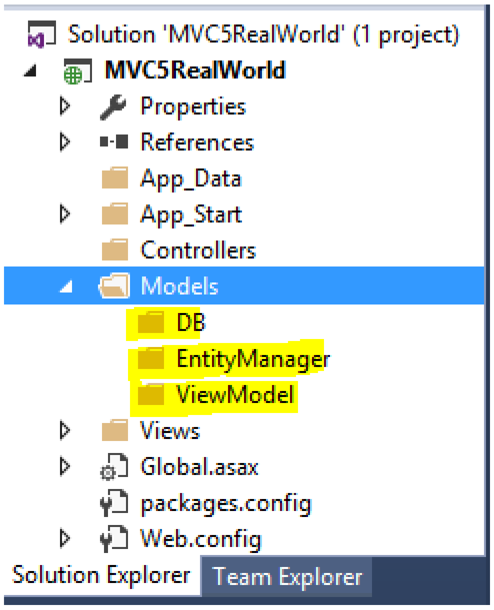


Figure 7: Creating the Models folder

The **DB** folder is where we store our entity data model (.EDMX). You can think of it as a conceptual database that contains some tables. To add an entity, right click on the DB folder and select Add > New Item > Data > ADO.NET Entity Data Mode as shown in the figure below.

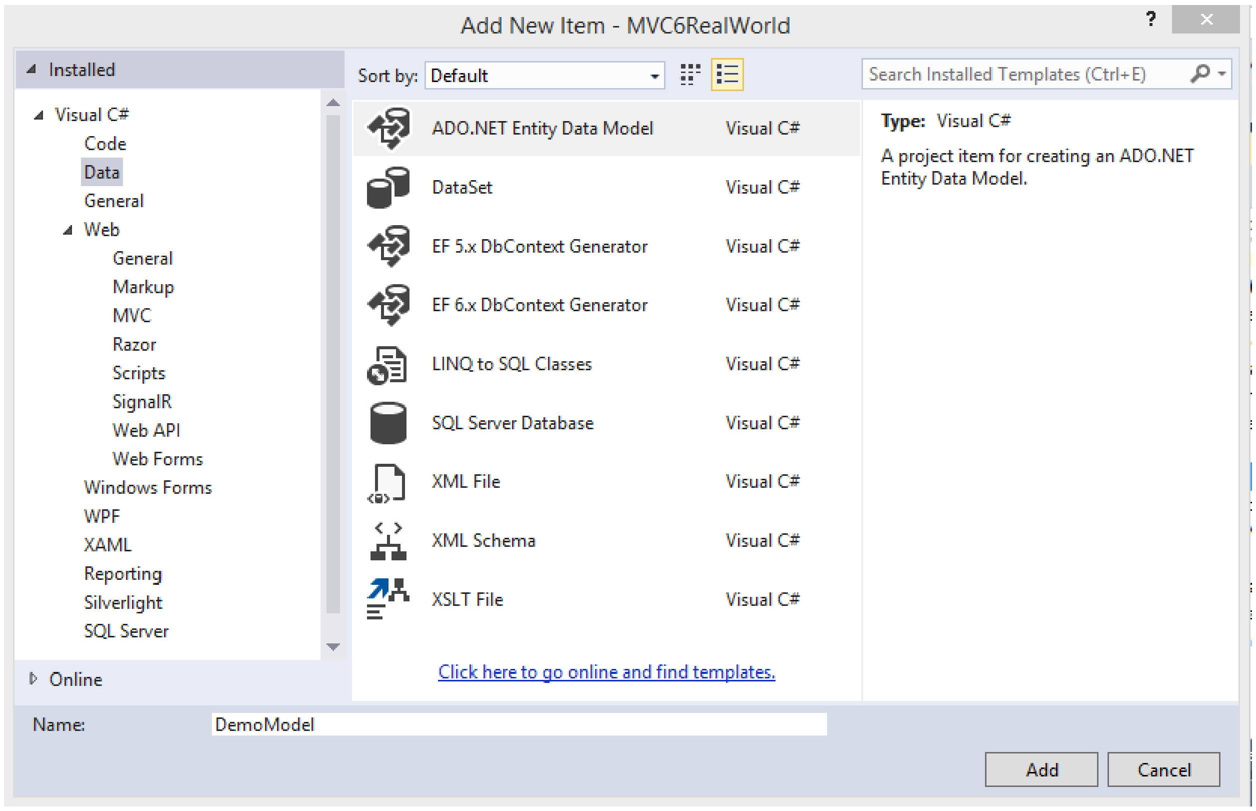
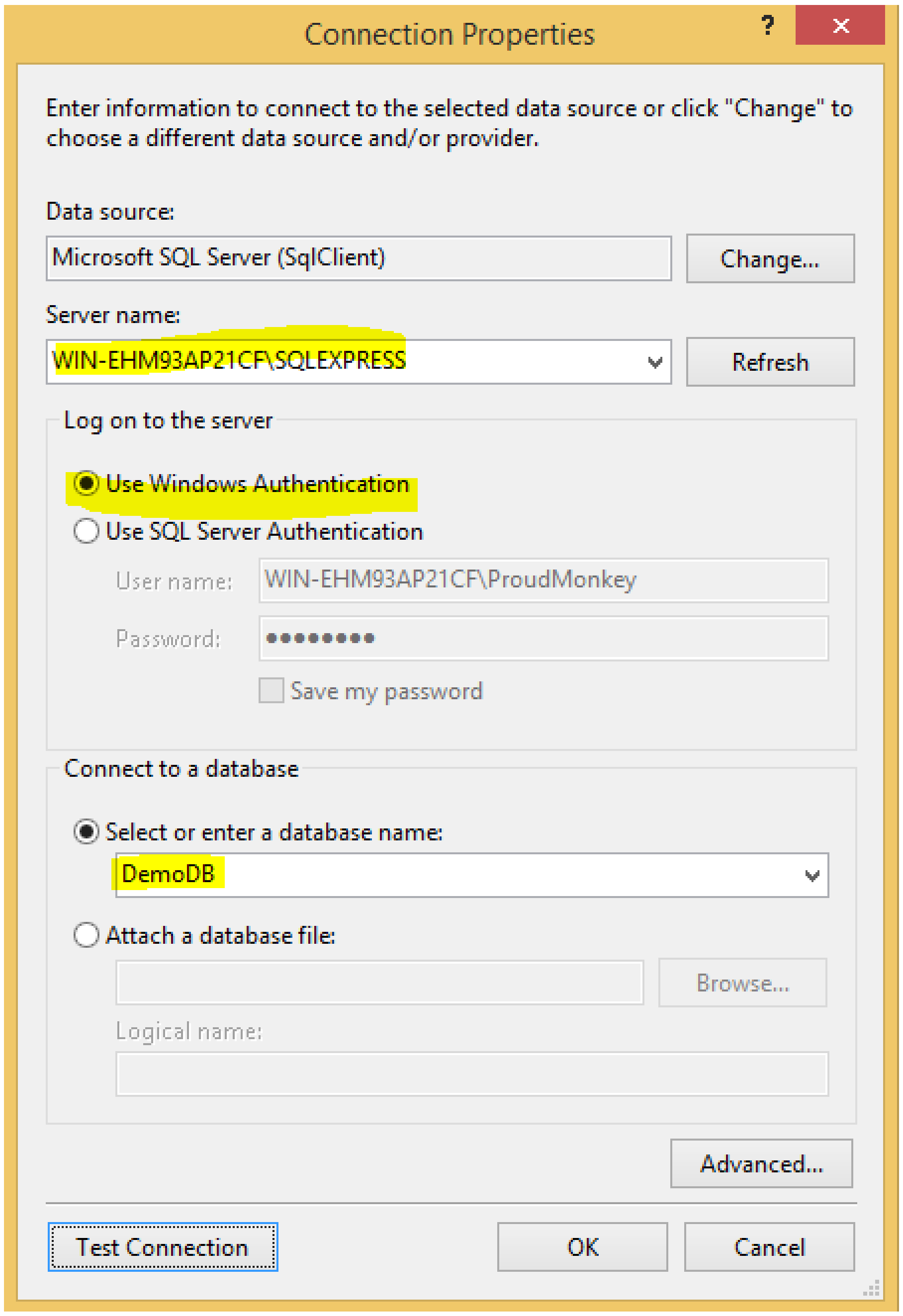

Add New Item - MVC6RealWorld
Installed
Visual C#
Code
Data
General
Web
General
Markup
MVC
Razor
Scripts
SignalR
Web API
Web Forms
Windows Forms
WPF
XAML
Reporting
Silverlight
SQL Server
Online
Sort by: Default
Search Installed Templates (Ctrl+E)
ADO.NET Entity Data Model Visual C#
DataSet Visual C#
EF 5.x DbContext Generator Visual C#
EF 6.x DbContext Generator Visual C#
LINQ to SQL Classes Visual C#
SQL Server Database Visual C#
XML File Visual C#
XML Schema Visual C#
XSLT File Visual C#
Click here to go online and find templates.
Type: Visual C#
A project item for creating an ADO.NET Entity Data Model.
Name: DemoModel
Add
Cancel

Figure 8: Adding Entity Data Model

You can name your entity model as you would like but for this example I just named it as "**DemoModel**" for simplicity. Now click "Add" to continue and on the next step select "EF Designer from Database" as we are going to use database first approach to work with existing database. Click "Next" to proceed. In the next step click on "New Connection" button and then select "Microsoft SQL Server (SqlClient)" as the data source, then click "Next". You should see this dialog below:


Connection Properties

Enter information to connect to the selected data source or click "Change" to choose a different data source and/or provider.

Data source: Microsoft SQL Server (SqlClient)

Server name: WIN-EHM93AP21CF\SQLEXPRESS

Log on to the server

(●) Use Windows Authentication

( ) Use SQL Server Authentication

User name: WIN-EHM93AP21CF\ProudMonkey

Password: ●●●●●●●●

[ ] Save my password

Connect to a database

(●) Select or enter a database name: DemoDB

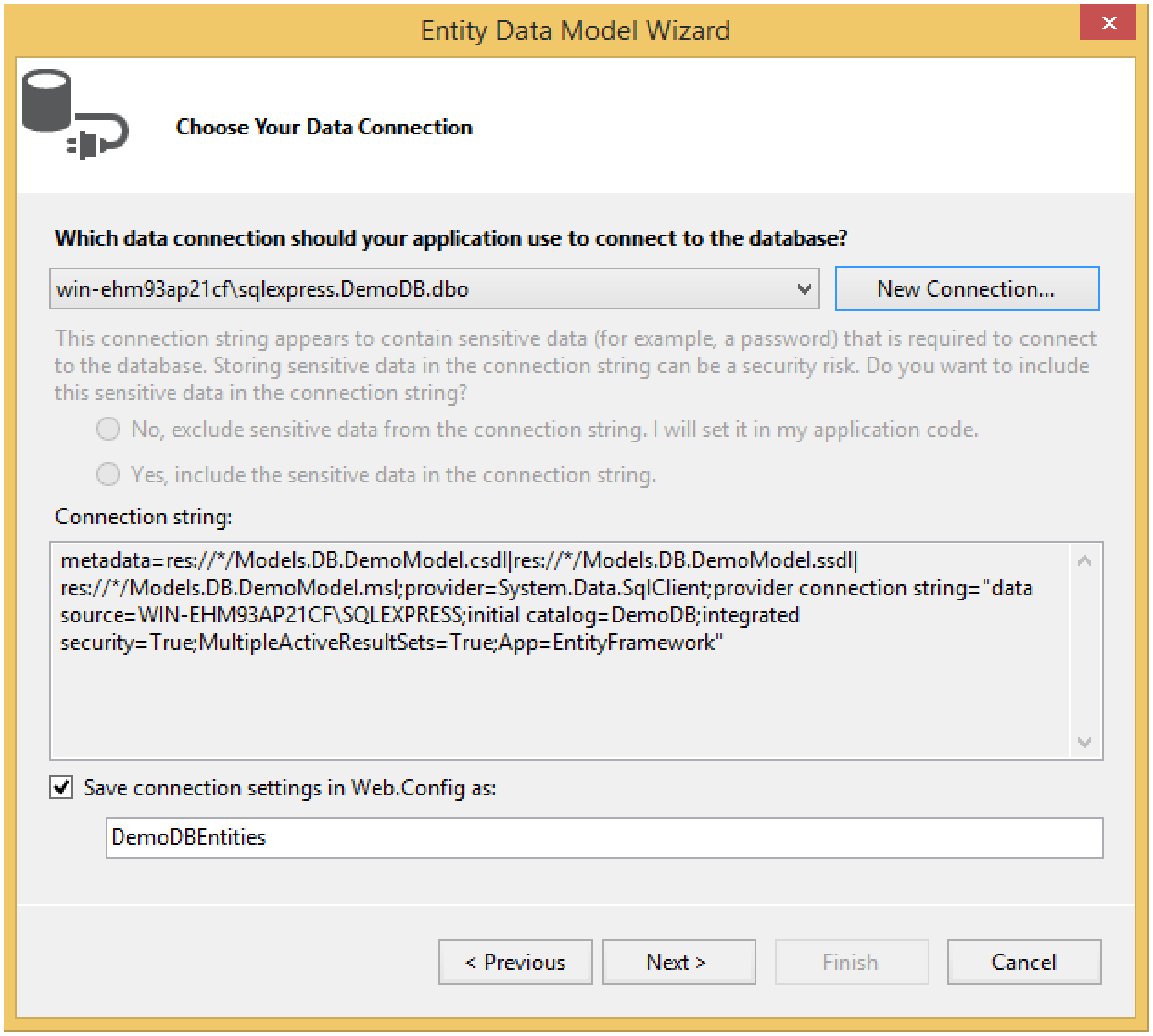
( ) Attach a database file:

Logical name:

Figure 9: Connection Properties dialog

Enter the SQL server name and select the database that we have just created in previous steps. If you have an existing database, then use that instead. Also note that I am using windows authentication for logging in to my SQL Server. Once you've done supplying the necessary fields, you can then click on "Test Connection" to verify the connectivity. If it is successful then just click "OK".

You should now see the following dialog below:



Which data connection should your application use to connect to the database?

win-ehm93ap21cf\sqlexpress.DemoDB.dbo

This connection string appears to contain sensitive data (for example, a password) that is required to connect to the database. Storing sensitive data in the connection string can be a security risk. Do you want to include this sensitive data in the connection string?

- No, exclude sensitive data from the connection string. I will set it in my application code.
- Yes, include the sensitive data in the connection string.

Connection string:

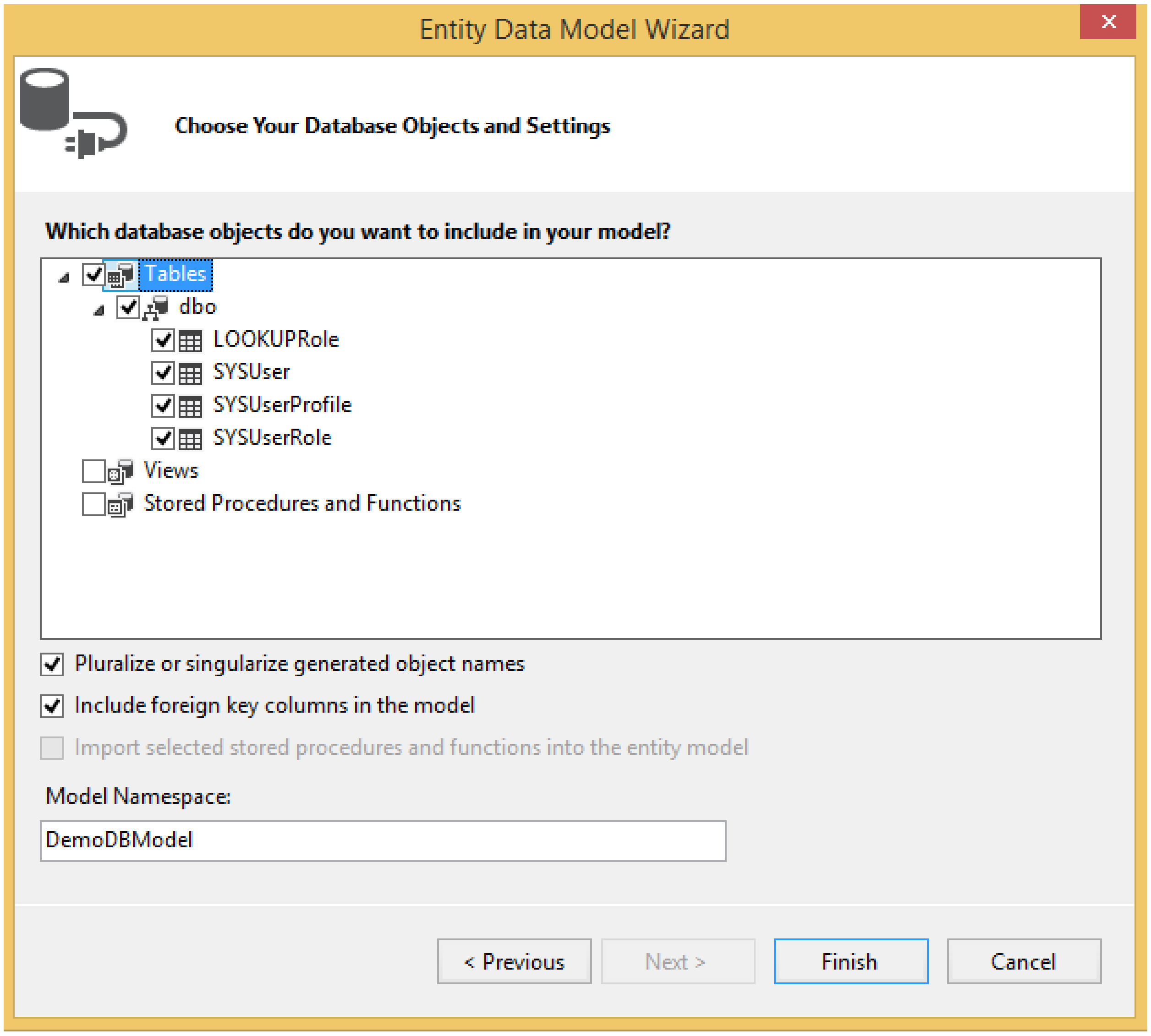
```
metadata=res://*/Models.DB.DemoModel.csdl|res://*/Models.DB.DemoModel.ssdl|
res://*/Models.DB.DemoModel.msl;provider=System.Data.SqlClient;provider connection string="data
source=WIN-EHM93AP21CF\SQLEXPRESS;initial catalog=DemoDB;integrated
security=True;MultipleActiveResultSets=True;App=EntityFramework"
```

✓ Save connection settings in Web.Config as:

DemoDBEntities

Figure 10: Choose Your Data Connection dialog

Notice that the connection string was automatically generated for you. Click “Next” and then select “Entity Framework 6.x” to bring up the following dialog below:



Which database objects do you want to include in your model?

- [✓] Tables
  - [✓] dbo
    - [✓] LOOKUPRole
    - [✓] SYSUser
    - [✓] SYSUserProfile
    - [✓] SYSUserRole
- [ ] Views
- [ ] Stored Procedures and Functions

[✓] Pluralize or singularize generated object names
[✓] Include foreign key columns in the model
[ ] Import selected stored procedures and functions into the entity model

Model Namespace: DemoDBModel

Figure 11: Entity Data Model Wizard dialog

Now select the table(s) that you want to use in your application. For this example I selected all tables because we are going to use those in our application. Clicking the "Finish" button will generate the entity model for you as shown in the figure below:

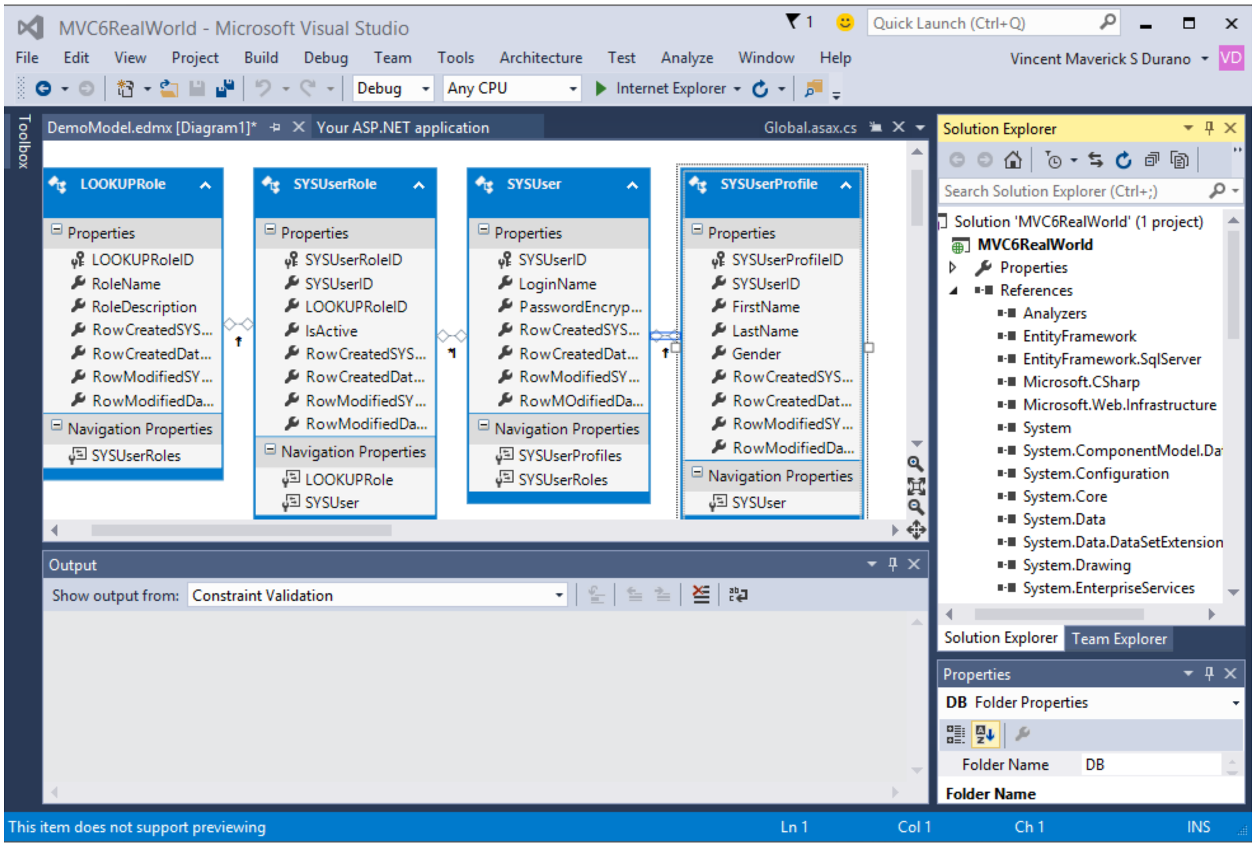


Figure 12: The Entity Data Model

What happened there is that EF automatically generates the business objects for you and let you query against it. The EDMX or the entity data model will serve as the main gateway by which you retrieve objects from database and resubmit changes.

Creating a Signup Page

Adding ViewModels

Again, Entity Framework will generate the business model objects and manage Data Access within the application. As a result, the class LOOKUPRole, SYSUserRole, SYSUser and SYSUserProfile are automatically created by EF and it features all the fields from the database table as properties of each class.

I don't want to use these classes directly in the View so I’ve decided to create a separate class that just holds the properties I needed in the View. Now let's add the “**UserModel**” class by right-clicking on the "ViewModel" folder and then selecting Add > Class. The "UserModel.cs" file is where we put all user related model views. For the Signup page we are going to add the "UserSignUpView" class. In the "UserModel.cs" file add the following code below:

```
using System.ComponentModel.DataAnnotations;

namespace MVC5RealWorld.Models.ViewModel
{

    public class UserSignUpView
    {
        [Key]
        public int SYSUserID { get; set; }
        public int LOOKUPRoleID { get; set; }
        public string RoleName { get; set; }
        [Required(ErrorMessage = "*")]
        [Display(Name = "Login ID")]
        public string LoginName { get; set; }
        [Required(ErrorMessage = "*")]
        [Display(Name = "Password")]
        public string Password { get; set; }
        [Required(ErrorMessage = "*")]
        [Display(Name = "First Name")]
        public string FirstName { get; set; }
        [Required(ErrorMessage = "*")]
        [Display(Name = "Last Name")]
        public string LastName { get; set; }
        public string Gender { get; set; }
    }
}
```

Notice that I have added the "Required" and "DisplayName" attributes for each property in the UserSignUpView class. This attributes is called Data Annotations. Data annotations are attribute classes that lives under *System.ComponentModel.DataAnnotations* namespace that you can use to decorate classes or properties to enforce pre-defined validation rules.

I'll use this validation technique because I want to keep a clear separation of concerns by using the MVC pattern and couple that with data annotations in the model, then your validation code becomes much simpler to write, maintain, and test.

For more information about Data Annotations then you can refer this article from MSDN: Data Annotations . And of course you can find more examples about it by doing a simple search at google 😉.

Adding the UserManager Class

The next step that we are going to do is to create the "**UserManger**" class that would handle the CRUD operations (Create, Read, Update and Delete operations) of a certain table. The purpose of this class is to separate the actual data operations from our controller and to have a central class for handling insert, update, fetch and delete operations.

Notes:

Please keep in mind that in this section I'm only be doing the insert part in which a user can add new data from the View to the database. I'll talk about how to do update, fetch and delete with MVC in the next section. So this time we'll just focus on the insertion part first.

Since this demo is intended to make web application as simple as possible then I will not be using TransactionScope and Repository pattern. In real complex web app you may want to consider using TransactionScope and Repository for your Data Access.

Now right click on the "EntityManager" folder and then add a new class by selecting Add > Class and name the class as "UserManager". Here's the code block for the "UserManager" class:

```
using System;
using System.Linq;
using MVC5RealWorld.Models.DB;
using MVC5RealWorld.Models.ViewModel;

namespace MVC5RealWorld.Models.EntityManager
{
    public class UserManager
    {
        public void AddUserAccount(UserSignUpView user) {

            using (DemoDBEntities db = new DemoDBEntities()) {

                SYSUser SU = new SYSUser();
                SU.LoginName = user.LoginName;
                SU.PasswordEncryptedText = user.Password;
                SU.RowCreatedSYSUserID = user.SYSUserID > 0 ? user.SYSUserID : 1;
                SU.RowModifiedSYSUserID = user.SYSUserID > 0 ? user.SYSUserID : 1; ;
                SU.RowCreatedDateTime = DateTime.Now;
                SU.RowMOdifiedDateTime = DateTime.Now;

                db.SYSUsers.Add(SU);
                db.SaveChanges();

                SYSUserProfile SUP = new SYSUserProfile();
                SUP.SYSUserID = SU.SYSUserID;
                SUP.FirstName = user.FirstName;
                SUP.LastName = user.LastName;
                SUP.Gender = user.Gender;
                SUP.RowCreatedSYSUserID = user.SYSUserID > 0 ? user.SYSUserID : 1;
                SUP.RowModifiedSYSUserID = user.SYSUserID > 0 ? user.SYSUserID : 1;
                SUP.RowCreatedDateTime = DateTime.Now;
                SUP.RowModifiedDateTime = DateTime.Now;
                db.SYSUserProfiles.Add(SUP);
                db.SaveChanges();


                if (user.LOOKUPRoleID > 0) {
                    SYSUserRole SUR = new SYSUserRole();
                    SUR.LOOKUPRoleID = user.LOOKUPRoleID;
                    SUR.SYSUserID = user.SYSUserID;
                    SUR.IsActive = true;
                    SUR.RowCreatedSYSUserID = user.SYSUserID > 0 ? user.SYSUserID : 1;
                    SUR.RowModifiedSYSUserID = user.SYSUserID > 0 ? user.SYSUserID : 1;
                    SUR.RowCreatedDateTime = DateTime.Now;
                    SUR.RowModifiedDateTime = DateTime.Now;

                    db.SYSUserRoles.Add(SUR);
                    db.SaveChanges();
                }
            }
        }

        public bool IsLoginNameExist(string loginName) {
            using (DemoDBEntities db = new DemoDBEntities()) {
                return db.SYSUsers.Where(o => o.LoginName.Equals(loginName)).Any();
            }
        }
    }
}
```

The **AddUserAccount()** is a method that inserts data to the database using Entity Framework. The **IsLoginNameExist()** is a method that returns boolean. It basically checks the database for an existing data using LINQ syntax.

Adding the Controllers

Since our model was already set then let's go ahead and add the "**AccountController**" class. To do this, just right click on the "Controllers" folder and select Add > Controller > MVC 5 Controller -Empty and then click “Add”. In the next dialog name the controller as "AccountController" and then click “Add” to generate class for you.

Here’s the code block for the "AccountController" class:

```
using System.Web.Mvc;
using System.Web.Security;
using MVC5RealWorld.Models.ViewModel;
using MVC5RealWorld.Models.EntityManager;

namespace MVC5RealWorld.Controllers
{
    public class AccountController : Controller
    {
        public ActionResult SignUp() {
            return View();
        }

        [HttpPost]
        public ActionResult SignUp(UserSignUpView USV) {
            if (ModelState.IsValid) {
                UserManager UM = new UserManager();
                if (!UM.IsLoginNameExist(USV.LoginName)) {
                    UM.AddUserAccount(USV);
                    FormsAuthentication.SetAuthCookie(USV.FirstName, false);
                    return RedirectToAction("Welcome", "Home");

                }
                else
                    ModelState.AddModelError("", "Login Name already taken.");
            }
            return View();
        }
    }
}
```

The “**AccountController”** class has two main methods. The first one is the "SignUp" which returns the "SignUp.cshtml" View when that action is requested. The second one also named as "SignUp" but it is decorated with the "[HttpPost]" attribute. This attribute specifies that the overload of the "SignUp" method can be invoked only for POST requests.

The second method is responsible for inserting new entry to the database and automatically authenticate the users using **FormsAuthentication.SetAuthCookie()** method. This method creates an authentication ticket for the supplied user name and adds it to the cookies collection of the response or to the URL if you are using cookieless authentication. After authenticating, we then redirect the users to the “Welcome.cshtml” page.

Now add another Controller and name it as "**HomeController**". This controller would be our controller for our default page. We will create the "Index" and the "Welcome" View for this controller in the next step. Here's the code for the "HomeController" class:

```
using System.Web.Mvc;

namespace MVC5RealWorld.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index() {
            return View();
        }

        [Authorize]
        public ActionResult Welcome() {
            return View();
        }

    }
}
```

The **HomeController** class consists of two ActionResult methods such as **Index** and **Welcome**. The "Index" method serves as our default redirect page and the "Welcome" method will be the page where we redirect users after they have authenticated successfully. We also decorated it with the "[Authorize]" attribute so that this method will only be available for the logged-in or authenticated users.

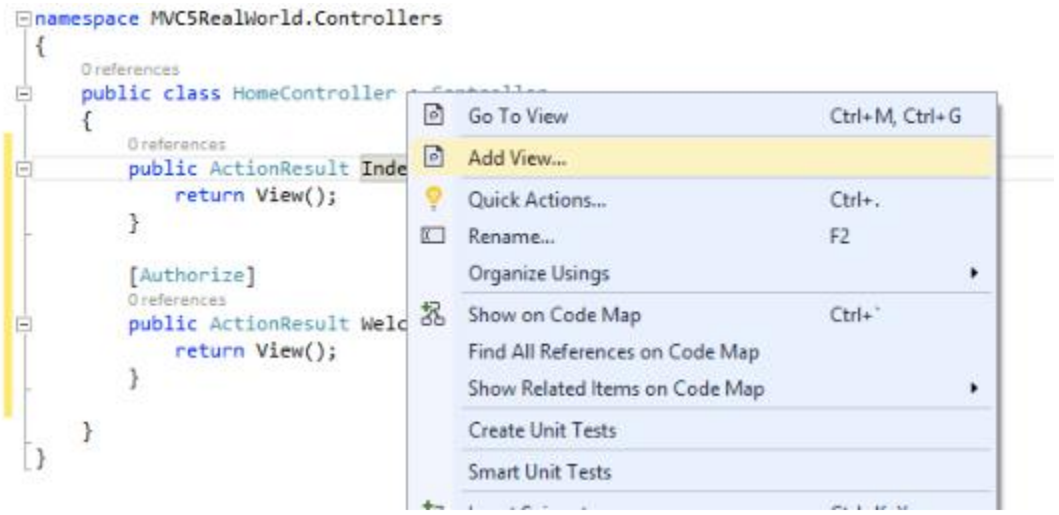
To configure a default page route, you can go to App_Start > RouteConfig. From there you should be able to see something like this:

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

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

The code above signifies that the URL path /Home/Index is the default page for our application. For more information about Routing, visit: ASP.NET MVC Routing Overview

Adding the Views

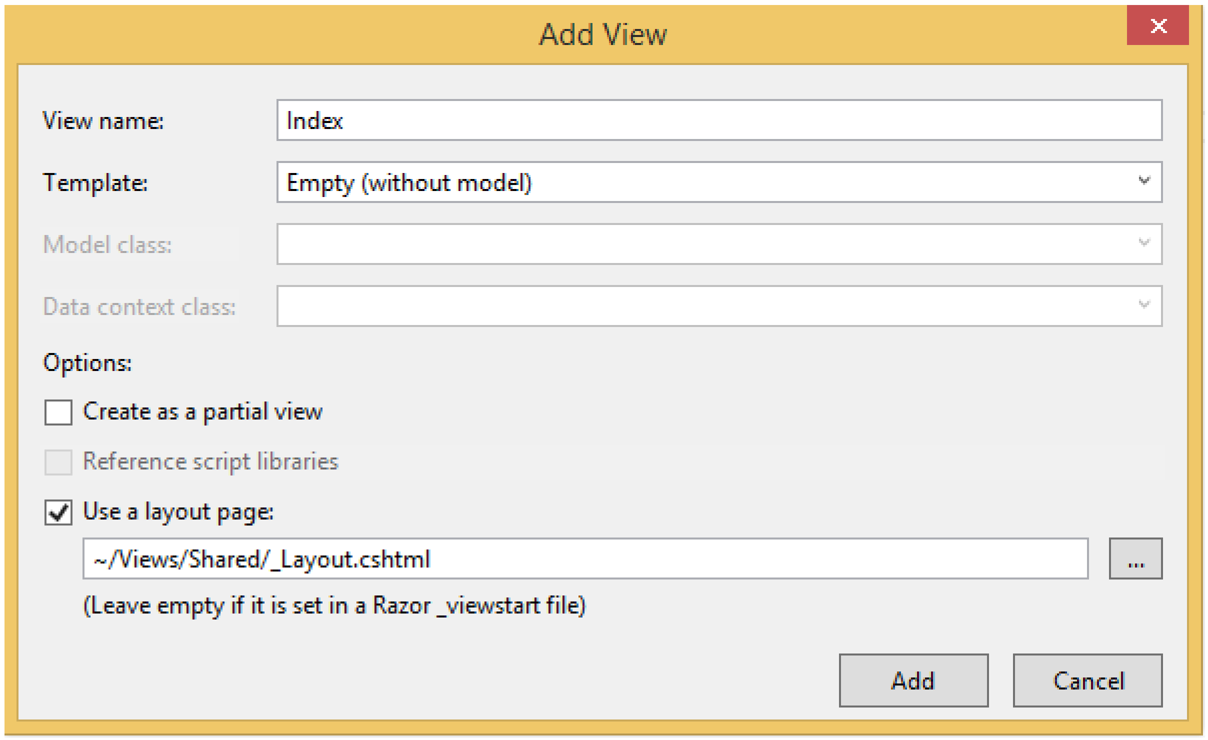
There are two possible ways to add Views. Either you can manually create the Views folder by yourself and add the corresponding .CSHTML files or by right clicking on the Controller's action method just like in the figure shown below:



```
namespace MVC5RealWorld.Controllers
{
    0 references
    public class HomeController
    {
        0 references
        public ActionResult Inde
            return View();
        }

        [Authorize]
        0 references
        public ActionResult Welc
            return View();
        }
    }
}
```

Figure 13: Adding new View

Clicking "Add" View will show this dialog below:



Add View

View name: Index

Template: Empty (without model)

Model class:

Data context class:

Options:

☐ Create as a partial view

☐ Reference script libraries

☑ Use a layout page:

~/Views/Shared/_Layout.cshtml

(Leave empty if it is set in a Razor _viewstart file)

Figure 14: Add View dialog

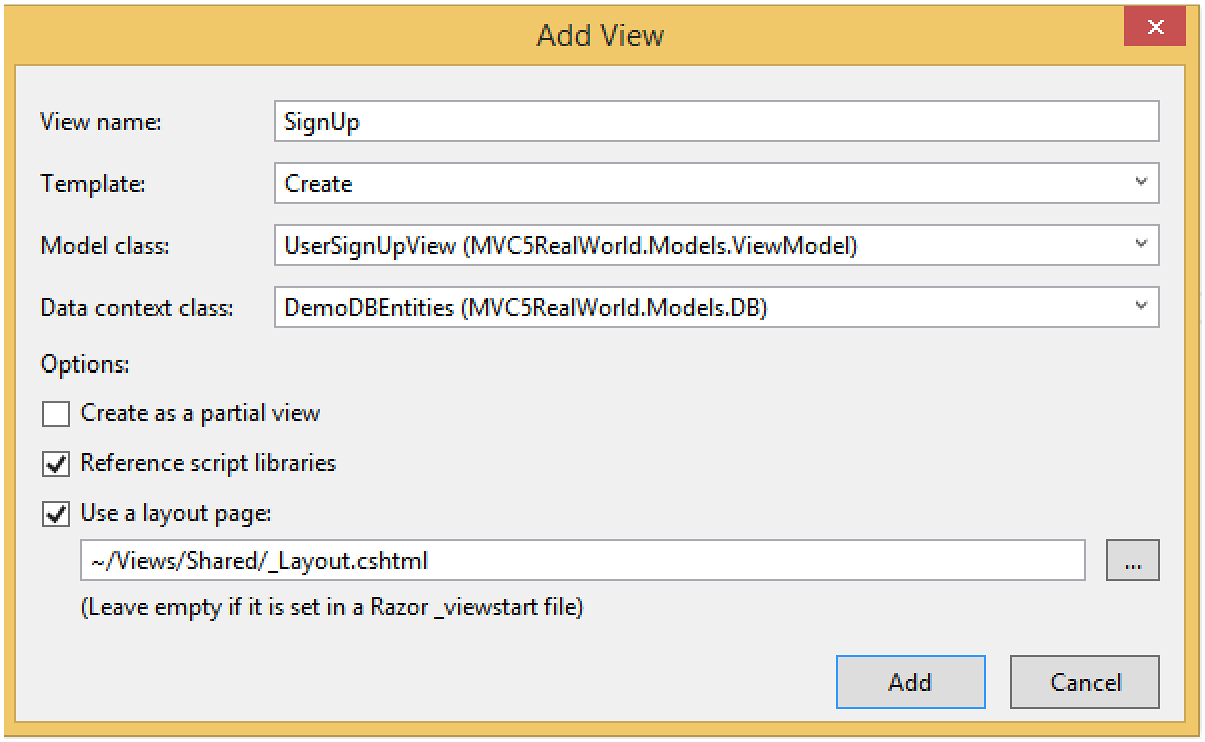
Just click "Add" since we don't need to do anything with the Index page at this point. Now modify the **Index** page and replace it with the following HTML markup:

```
@{
    ViewBag.Title = "Index";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<h2>Index</h2>
<br/>
No Account yet? @Html.ActionLink("Signup Now!", "SignUp", "Account")
```

The ActionLink in the markup above allows you to navigate to the **SignUp** page which lives under AccountController. Now add a View to the Welcome action by doing the same as what we did by adding the Index page. Here's the Welcome page HTML markup:

```
@{
    ViewBag.Title = "Welcome";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<h2>Hi <b>@Context.User.Identity.Name</b>! Welcome to my first MVC 5 Web App!</h2>
```

Now switch back to "**AccountController**" class and add a new View for the "**SignUp**" page. In the Add View dialog select "Create" as the scaffold template, select the "UserSignUpView" as the model and the "DemoDBEntities" as the data context as shown in the figure below:



View name: SignUp
Template: Create
Model class: UserSignUpView (MVC5RealWorld.Models.ViewModel)
Data context class: DemoDBEntities (MVC5RealWorld.Models.DB)
☐ Create as a partial view
☑ Reference script libraries
☑ Use a layout page:
~/Views/Shared/_Layout.cshtml
(Leave empty if it is set in a Razor _viewstart file)

Figure 15: Add View dialog

Click “Add” to let Visual Studio scaffolds the UI for you. The term “Scaffolding” allow you to quickly generate the UI that you can edit and customize.

Now we need to trim down the generated fields because there are some fields that we don’t actually need users to see like the RoleName and ID’s. Adding to that I also modified the Password to use the PasswordFor HTML helper and use DropDownListFor for displaying the Gender. Here’s the modified and trimmed down HTML markup for the SignUp page:

```
@model MVC5RealWorld.Models.ViewModel.UserSignUpView

@{
    ViewBag.Title = "SignUp";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<h2>SignUp</h2>

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div class="form-horizontal">
        <hr />
        @Html.ValidationSummary(true, "", new { @class = "text-danger" })
        <div class="form-group">
            @Html.LabelFor(model => model.LoginName, htmlAttributes: new { @class =
"control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.LoginName, new { htmlAttributes = new {
@class = "form-control" } })
                @Html.ValidationMessageFor(model => model.LoginName, "", new { @class =
"text-danger" })
            </div>
        </div>

        <div class="form-group">
            @Html.LabelFor(model => model.Password, htmlAttributes: new { @class =
"control-label col-md-2" })
            <div class="col-md-10">
                @Html.PasswordFor(model => model.Password, new  { @class = "form-control"
} )
                @Html.ValidationMessageFor(model => model.Password, "", new { @class =
"text-danger" })
            </div>
        </div>

        <div class="form-group">
            @Html.LabelFor(model => model.FirstName, htmlAttributes: new { @class =
"control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.FirstName, new { htmlAttributes = new {
@class = "form-control" } })
```

```
  @Html.ValidationMessageFor(model => model.FirstName, "", new { @class = "text-danger" 
})
            </div>
        </div>

        <div class="form-group">
            @Html.LabelFor(model => model.LastName, htmlAttributes: new { @class = 
"control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.LastName, new { htmlAttributes = new { 
@class = "form-control" } })
                @Html.ValidationMessageFor(model => model.LastName, "", new { @class = 
"text-danger" })
            </div>
        </div>

        <div class="form-group">
            @Html.LabelFor(model => model.Gender, htmlAttributes: new { @class = 
"control-label col-md-2" })
            <div class="col-md-10">
                @Html.DropDownListFor(model => model.Gender, new List<SelectListItem> {
                    new SelectListItem { Text="Male", Value="M" },
                    new SelectListItem { Text="Female", Value="F" }
                }, new { @class = "form-control" })
            </div>
        </div>

        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Register" class="btn btn-default" />
            </div>
        </div>
    </div>
}

<div>
    @Html.ActionLink("Back to Main", "Index","Home")
</div>
```

The markup above is a strongly-type view. This strongly typed approach enables better compile-time checking of your code and richer IntelliSense in the Visual Studio editor. By including a @model statement at the top of the view template file, you can specify the type of object that the view expects. In this case it uses the **MVC5RealWorld.Models.ViewModel.UserSignUpView.**

If you also noticed, after adding the views, Visual Studio automatically structures the folders for your Views. See the figure below for your reference:

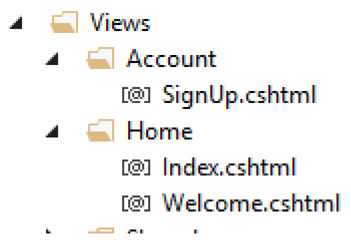
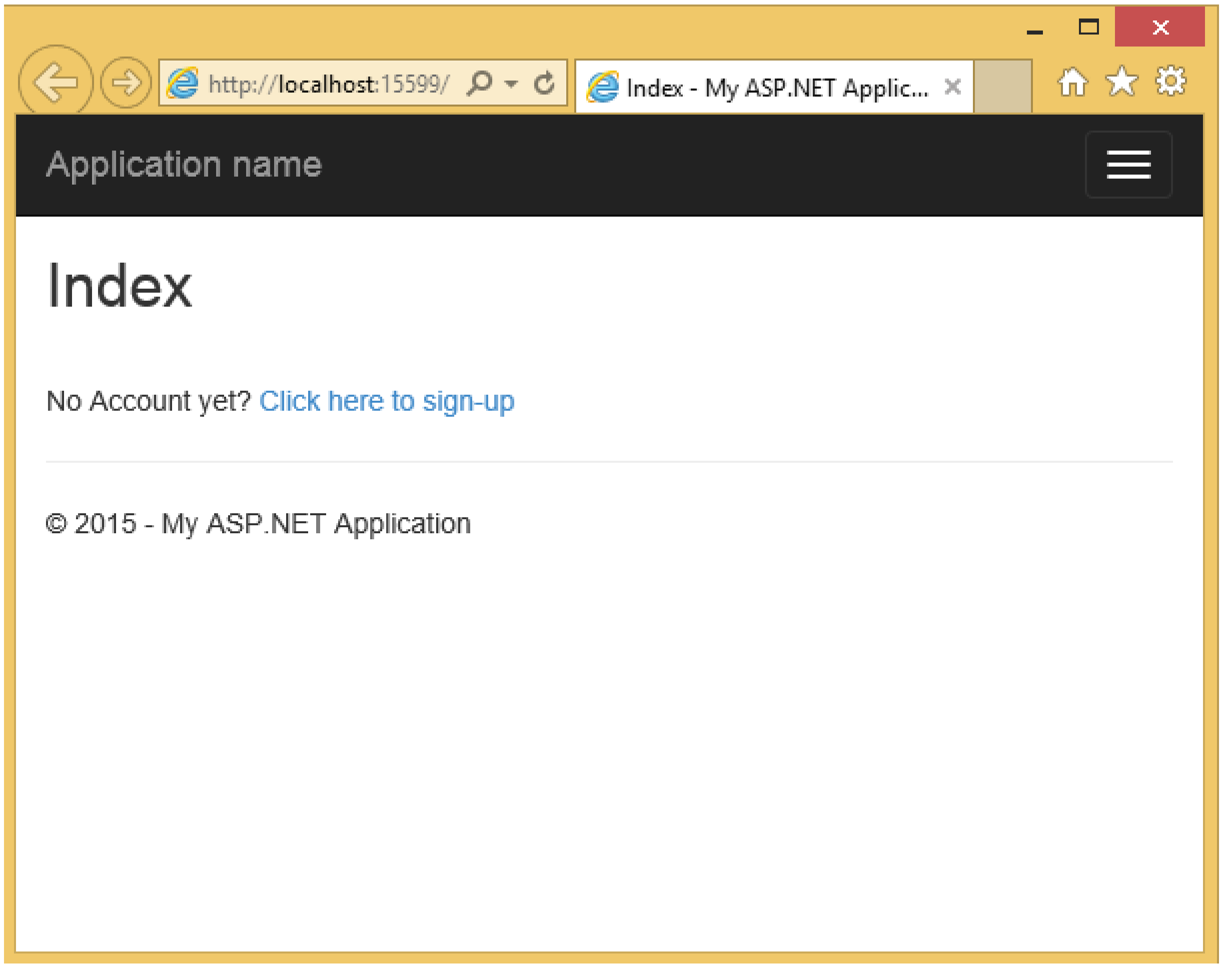

Views
Account
SignUp.cshtml
Home
Index.cshtml
Welcome.cshtml

Figure 16: The newly added Views

Running the Application

Here are the following outputs when you run the page in the browser:

On initial load



Application name

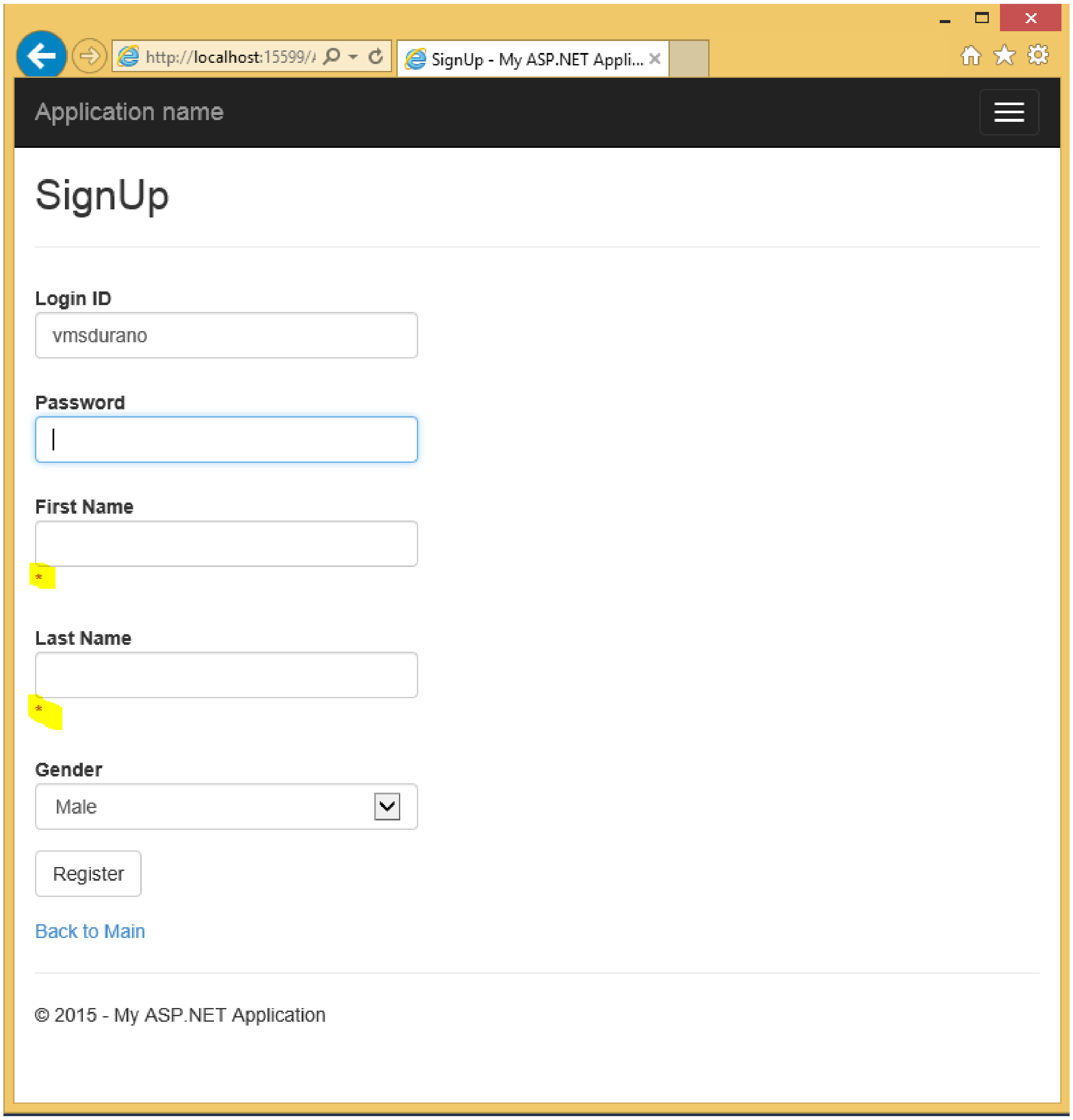
Index

No Account yet? Click here to sign-up

© 2015 - My ASP.NET Application

Figure 17: Initial request

Page validation triggers



http://localhost:15599/

SignUp - My ASP.NET Appli...

Application name

SignUp

Login ID

vmsdurano

Password

First Name

*

Last Name

*

Gender

Male

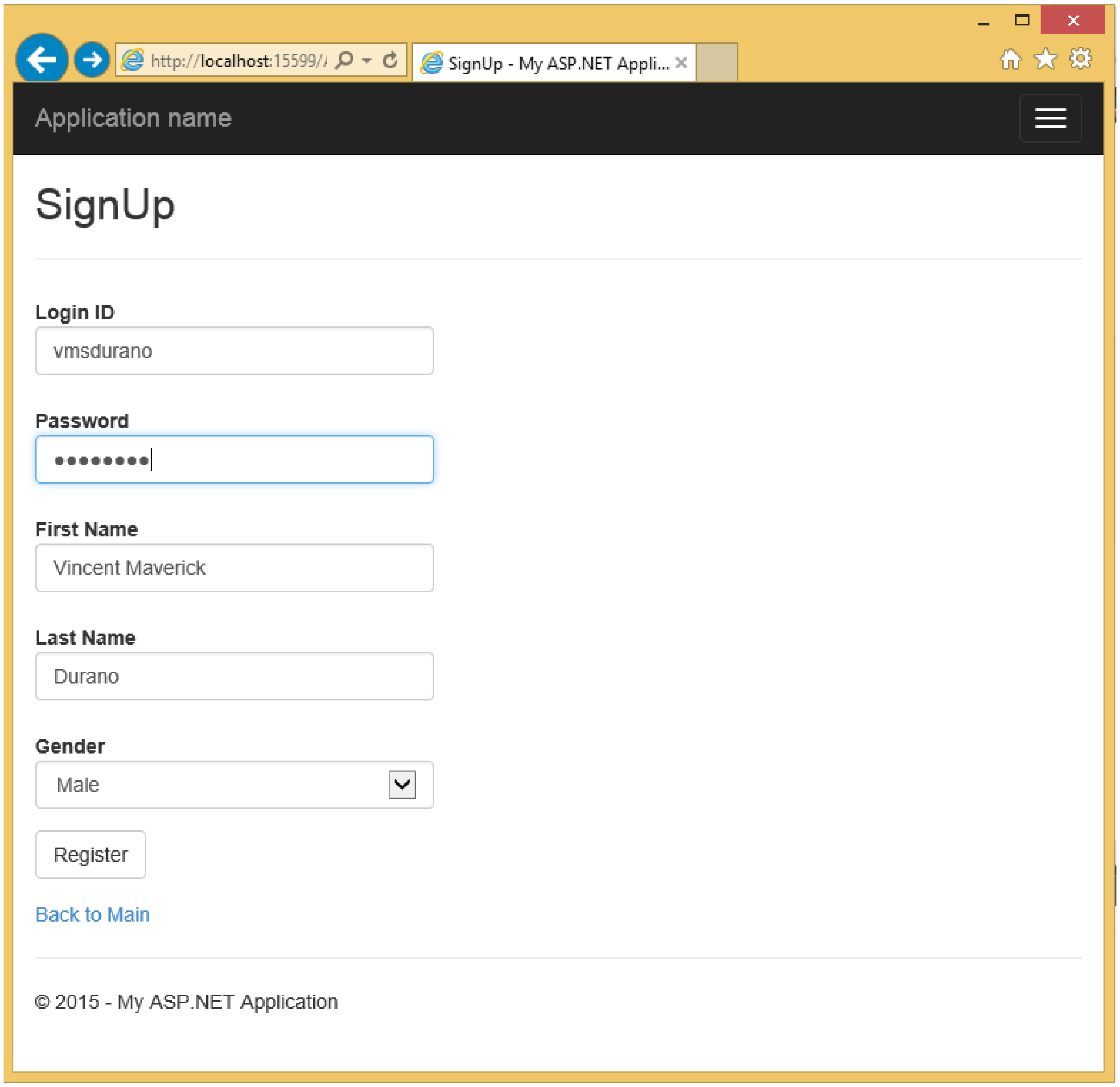
Register

Back to Main

© 2015 - My ASP.NET Application

Figure 18: Page Validation

Supplying the required fields



SignUp

Login ID
vmsdurano

Password
●●●●●●●●●

First Name
Vincent Maverick

Last Name
Durano

Gender
Male

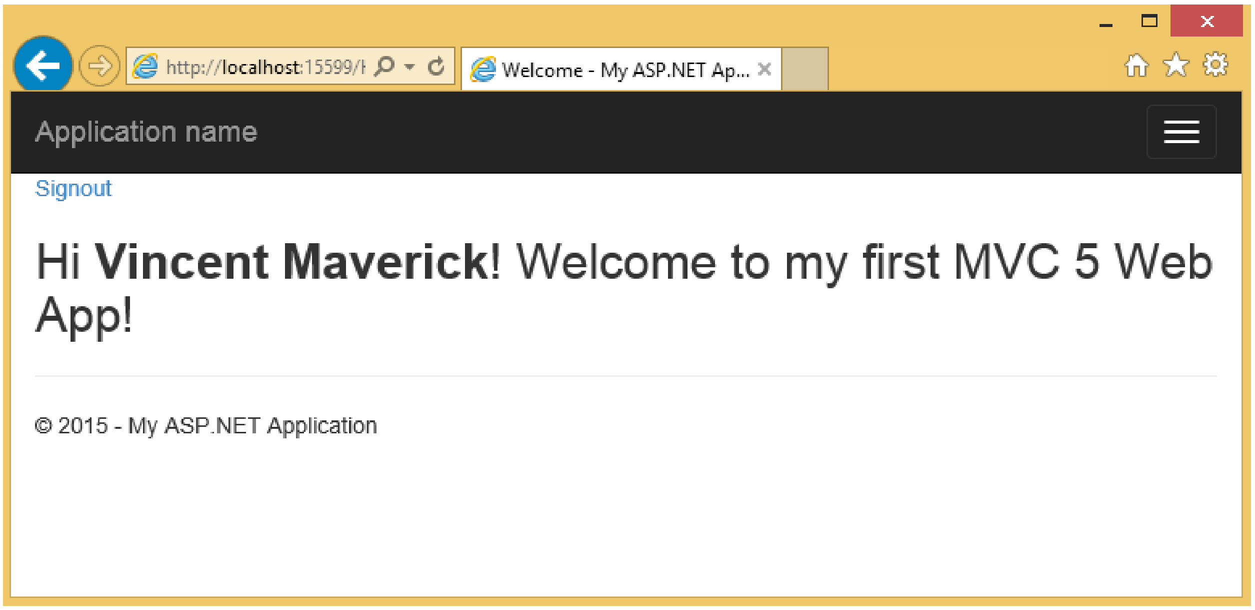
Register

Back to Main

© 2015 - My ASP.NET Application

Figure 19: Supplying the Required fields

And after successful registration



Hi **Vincent Maverick**! Welcome to my first MVC 5 Web App!

© 2015 - My ASP.NET Application

Figure 20: Successful registration

Creating the Login Page

In this section you will learn the following:

- Creating a Login page that would validate and authenticate user using Forms Authentication
- Creating a custom role-based page authorization using custom Authorize filter

In this section, I will show how to create a simple Login page by implementing a custom authentication and role-based page authorization without using ASP.NET Membership or ASP.NET Identity. If you want to build an app that allow users to login using their social media accounts like Facebook, Twitter, Google Plus, etc. then you may want explore on ASP.NET Identity instead.

Before we get our hands dirty let's talk about a bit of security in general.

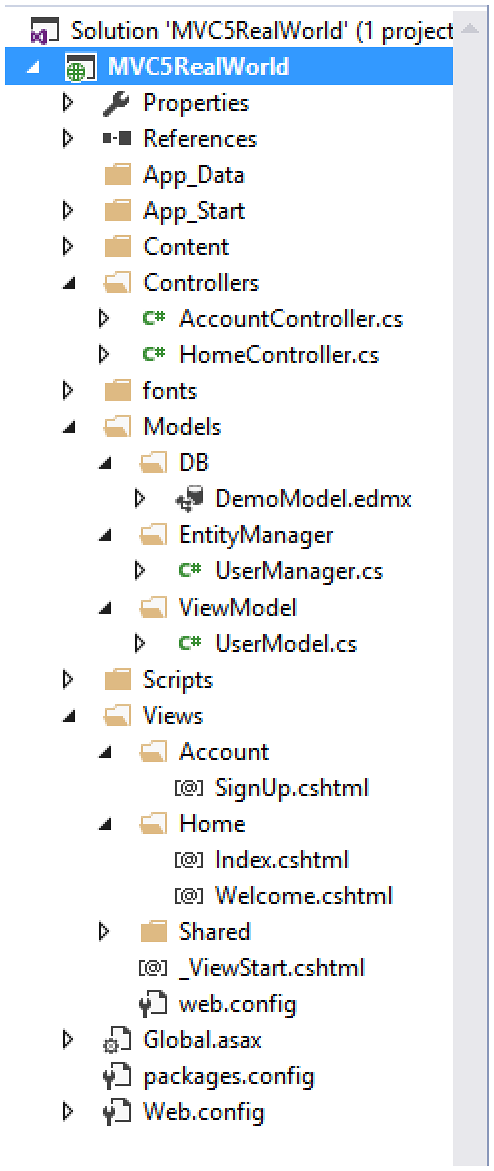
Forms Authentication Overview

Security is an integral part of any Web-based application. Majority of the web sites nowadays heavily relies on authentication and authorization for securing their application. You can think of a web site as somewhat analogous to a company office where an office is open for people like applicants or messenger to come, but there are certain parts of the facility, such as workstations and conference rooms, that are accessible only to people with certain credentials, such as employees. An example is when you build a shopping cart application that accepts users' credit card information for payment purposes and stores them to your database; ASP.NET helps protect your database from public access by providing authentication and authorization mechanism.

Forms authentication lets you authenticate users by using your own code and then maintain an authentication token in a cookie or in the page URL. To use forms authentication, you create a login page that collect credentials from the user and that includes code to authenticate the credentials. Typically you configure the application to redirect requests to the login page when users try to access a protected resource, such as a page that requires authentication. If the user's credentials are valid, you can call the method of the **FormsAuthentication** class to redirect the request back to the originally requested resource with an appropriate authentication ticket (cookie).

Let’s get our hands dirty!

As a recap, here's the previous project structure below:



```
Solution 'MVC5RealWorld' (1 project
  MVC5RealWorld
    Properties
    References
    App_Data
    App_Start
    Content
    Controllers
      AccountController.cs
      HomeController.cs
    fonts
    Models
      DB
        DemoModel.edmx
      EntityManager
        UserManager.cs
      ViewModel
        UserModel.cs
    Scripts
    Views
      Account
        SignUp.cshtml
      Home
        Index.cshtml
        Welcome.cshtml
      Shared
      _ViewStart.cshtml
      web.config
    Global.asax
    packages.config
    Web.config
```

Figure 21: The Project structure

Enabling Forms Authentication

The very first thing you do to allow forms authentication in your application is to configure FormsAuthentication which manages forms authentication services to your web application. The default authentication mode for ASP.NET is “windows”. To enable forms authentication, add the <authentication> and <forms> elements under <system.web> element in your web.config like:

```
<system.web>
    <authentication mode="Forms">
      <forms loginUrl="~/Account/Login" defaultUrl="~/Home/Welcome"></forms>
    </authentication>

</system.web>
```

Setting the **loginUrl** enables the application to determine where to redirect an un-authenticated user who attempts to access a secured page. The **defaultUrl** redirects users to the specified page after they have successfully logging-in into the web site.

Adding the UserLoginView Model

Let's go ahead and create a View Model class for our Login page by adding the following code below within the “UserModel” class:

```
public class UserLoginView
{
        [Key]
        public int SYSUserID { get; set; }
        [Required(ErrorMessage = "*")]
        [Display(Name = "Login ID")]
        public string LoginName { get; set; }
        [Required(ErrorMessage = "*")]
        [DataType(DataType.Password)]
        [Display(Name = "Password")]
        public string Password { get; set; }

}
```

The fields defined above will be used in our Login page. You may also notice that the fields are decorated with Required, Display and DataType attributes. Again these attributes are called Data Annotations. Adding these attributes will allow you to do pre-validation on the model. For example the LoginName and Password field should not be empty.

Adding the GetUserPassword() Method

Add the following code below under “UserManager.cs” class:

```
public string GetUserPassword(string loginName) {
            using (DemoDBEntities db = new DemoDBEntities()) {
                var user = db.SYSUsers.Where(o =>
o.LoginName.ToLower().Equals(loginName));
                if (user.Any())
                    return user.FirstOrDefault().PasswordEncryptedText;
                else
                    return string.Empty;
            }

}
```

As the method name suggests, it gets the corresponding password from the database for a particular user using LINQ query.

Adding the Login Action Method

Add the following code below under “AccountController” class:

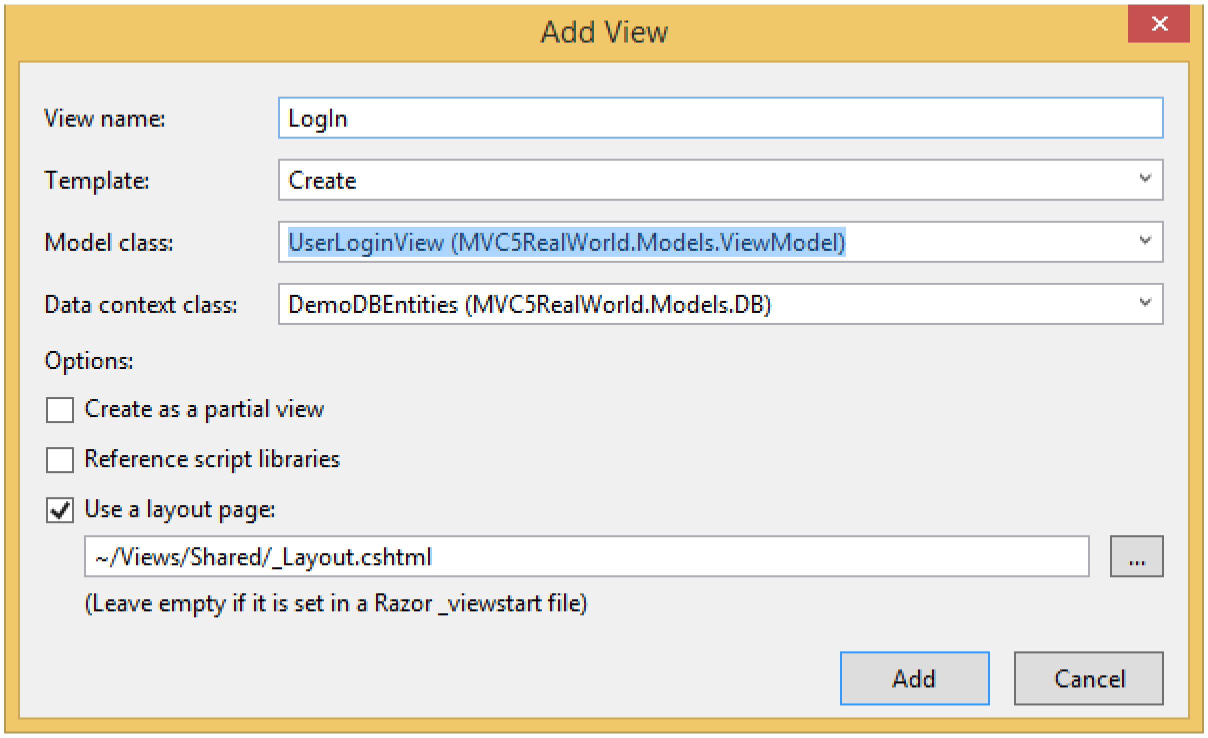
```
public ActionResult LogIn() {
            return View();
 }

 [HttpPost]
 public ActionResult LogIn(UserLoginView ULV, string returnUrl) {
            if (ModelState.IsValid) {
                UserManager UM = new UserManager();
                string password = UM.GetUserPassword(ULV.LoginName);

                if (string.IsNullOrEmpty(password))
                    ModelState.AddModelError("", "The user login or password provided is 
incorrect.");
                else {
                    if (ULV.Password.Equals(password)) {
                        FormsAuthentication.SetAuthCookie(ULV.LoginName, false);
                        return RedirectToAction("Welcome", "Home");
                    }
                    else {
                        ModelState.AddModelError("", "The password provided is 
incorrect.");
                    }
                }
            }

            // If we got this far, something failed, redisplay form
            return View(ULV);

}
```

There are two methods above with the same name. The first one is the "Login" method that simply returns the LogIn.cshtml view. We will create this view in the next step. The second one also named as "Login" but it is decorated with the "[HttpPost]" attribute. If you still remember from previous section, this attribute specifies an overload of the "Login" method that can be invoked for POST requests only.

The second method will be triggered once the Button "LogIn" is clicked. What it does is, first it will check if the required fields are supplied so it checks for ModelState.IsValid condition. It will then create an instance of the **UserManager** class and call the **GetUserPassword()** method by passing the user LoginName value supplied by the user. If the password returns an empty string then it will display an error to the View. If the password supplied is equal to the password retrieved from the database then it will redirect the user to the Welcome page, otherwise displays an error stating that the login name or password supplied was invalid.

Adding the Login View

Before adding the view, make sure to build your application first to ensure that the application is error free. After a successful build, navigate to "AccountController" class and right click on the Login Action method and then select "Add View". This will bring up the following dialog below:



View name: LogIn
Template: Create
Model class: UserLoginView (MVC5RealWorld.Models.ViewModel)
Data context class: DemoDBEntities (MVC5RealWorld.Models.DB)
Options:
- [ ] Create as a partial view
- [ ] Reference script libraries
- [✓] Use a layout page:

~/Views/Shared/_Layout.cshtml
(Leave empty if it is set in a Razor _viewstart file)

Figure 22: Add View dialog

Take note of the values supplied for each field above. Now click on "Add" to let Visual Studio scaffolds the UI for you. Here's the modified HTML markup below:

```
@model MVC5RealWorld.Models.ViewModel.UserLoginView

@{
    ViewBag.Title = "LogIn";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<h2>LogIn</h2>

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()
    <div class="form-horizontal">
        <hr />
        @Html.ValidationSummary(true, "", new { @class = "text-danger" })
        <div class="form-group">
            @Html.LabelFor(model => model.LoginName, htmlAttributes: new { @class =
"control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.LoginName, new { htmlAttributes = new {
@class = "form-control" } })
                @Html.ValidationMessageFor(model => model.LoginName, "", new { @class =
"text-danger" })
            </div>
        </div>

        <div class="form-group">
            @Html.LabelFor(model => model.Password, htmlAttributes: new { @class =
"control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.Password, new { htmlAttributes = new {
@class = "form-control" } })
                @Html.ValidationMessageFor(model => model.Password, "", new { @class =
"text-danger" })
            </div>
        </div>

        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Login" class="btn btn-default" />
            </div>
        </div>
    </div>
}

<div>
    @Html.ActionLink("Back to Main", "Index", "Home")
</div>
```

Implementing the Logout Functionality

The logout code is pretty much easy. Just add the following method below within the "AccountController "class.

```
[Authorize]
public ActionResult SignOut() {
            FormsAuthentication.SignOut();
            return RedirectToAction("Index", "Home");

}
```

The **FormsAuthentication.SignOut** method removes the forms-authentication ticket from the browser. We then redirect user to **Index** page after signing out.

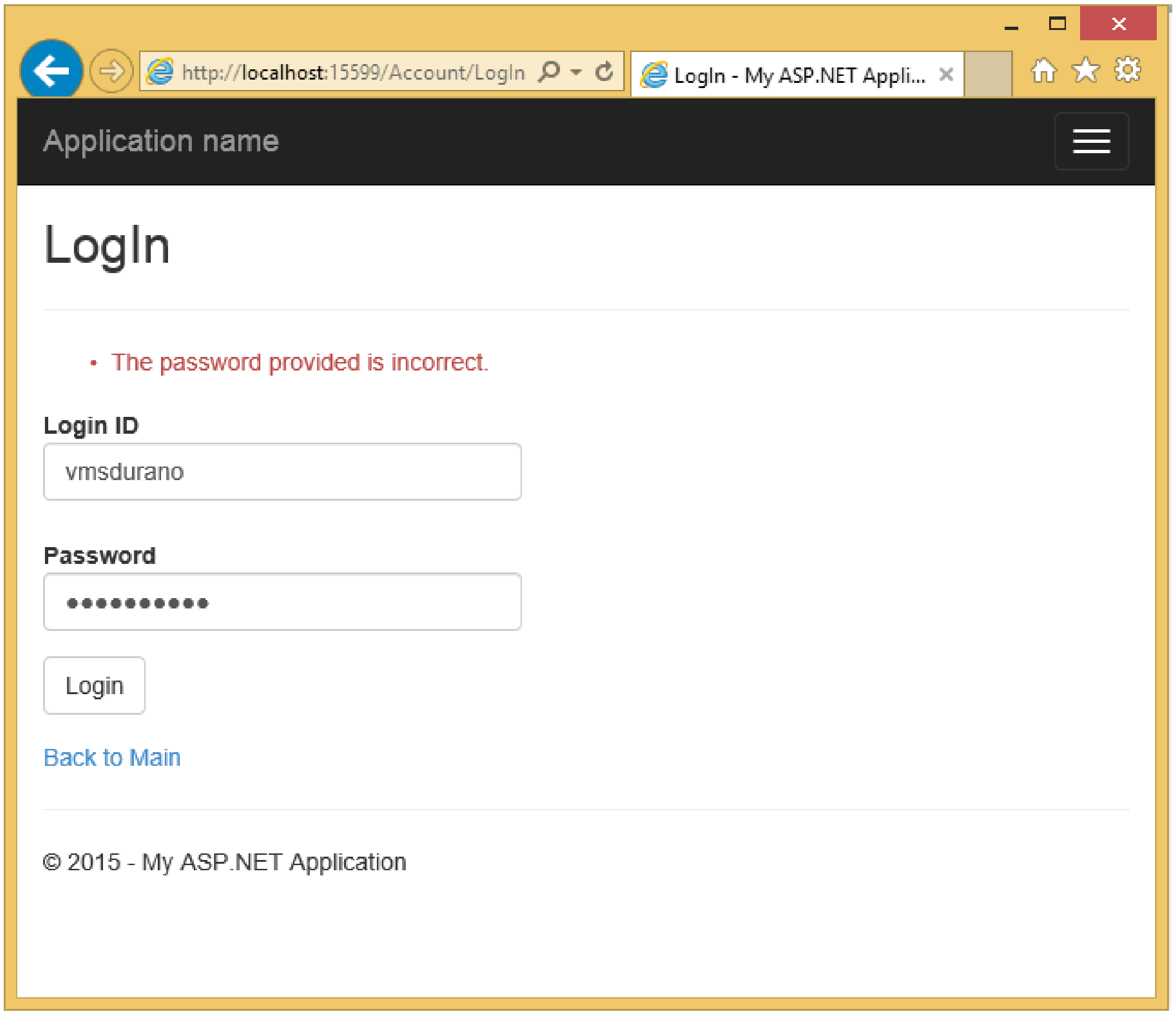
Here's the corresponding action link for the Logout that you can add within your Home page:

```
@Html.ActionLink("Signout","SignOut","Account")
```

Running the Application

Now try to navigate to this URL: http://localhost:15599/Account/LogIn. It should display something like these:

When validation triggers


http://localhost:15599/Account/LogIn
LogIn - My ASP.NET Appli...
Application name

LogIn

- The password provided is incorrect.

Login ID

vmsdurano

Password

●●●●●●●●●●●

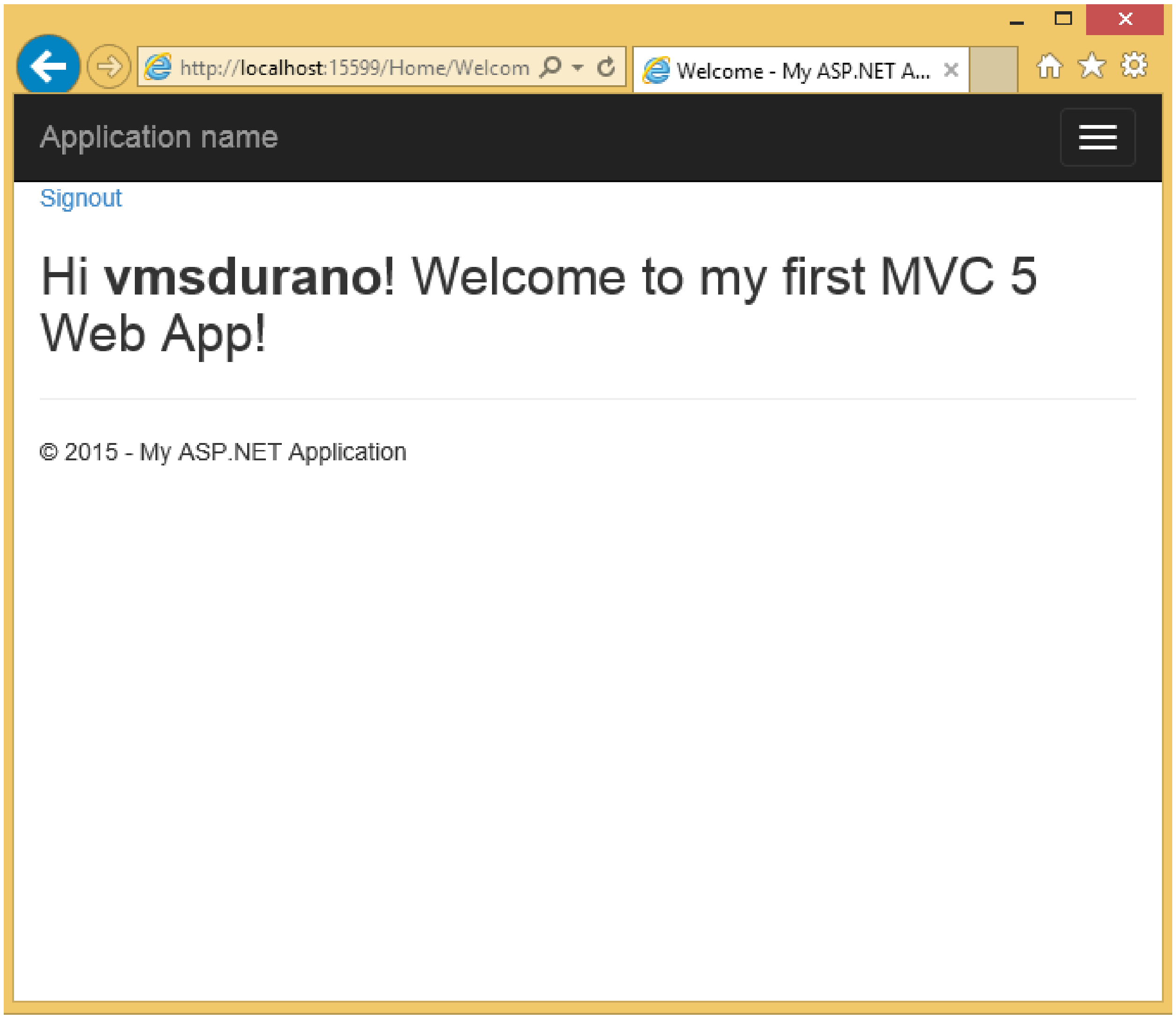
Login

Back to Main

© 2015 - My ASP.NET Application

Figure 23: Validation triggers

After successful Logging-in


http://localhost:15599/Home/Welcom

Welcome - My ASP.NET A...

Application name

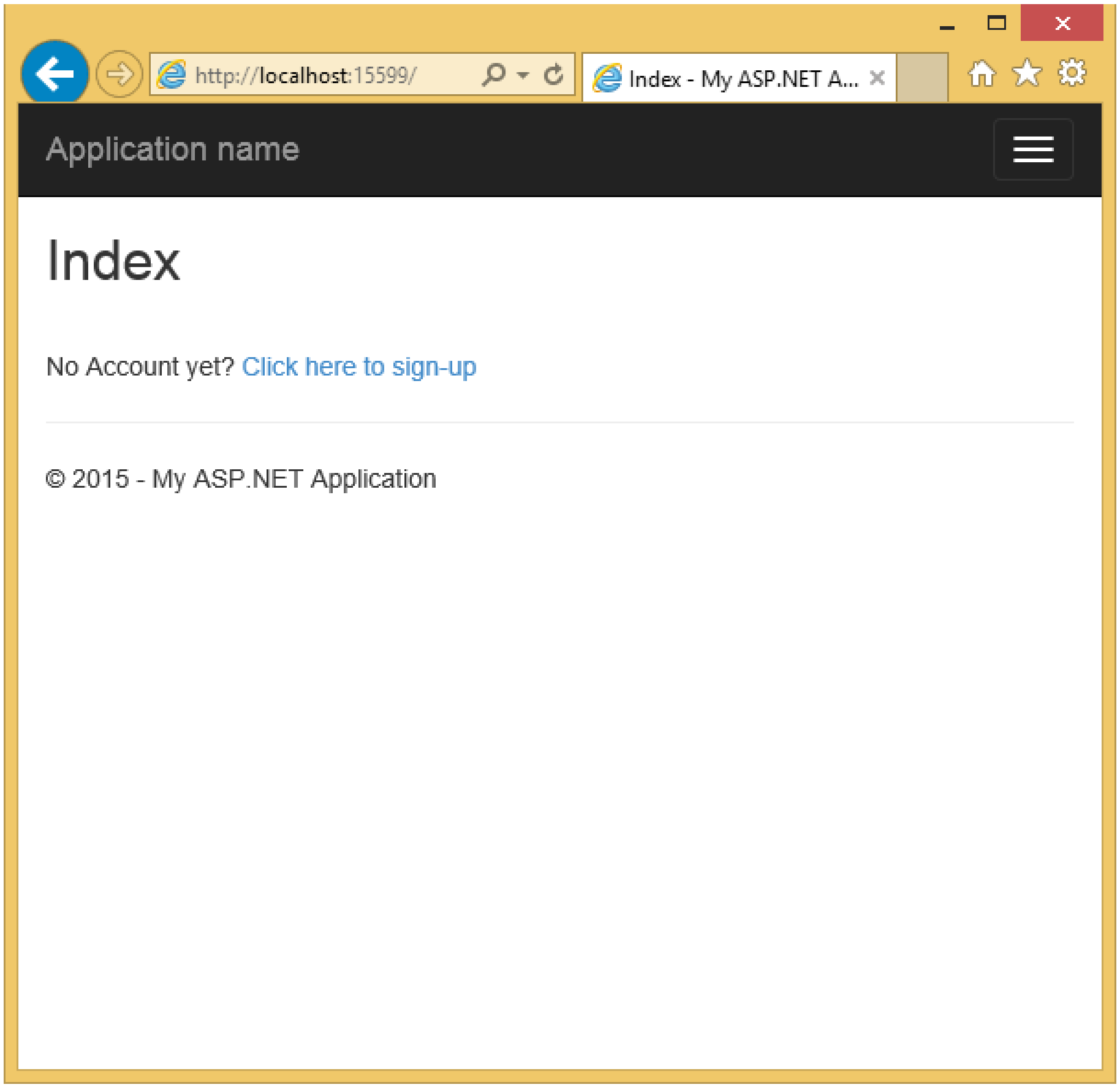
Signout

Hi **vmsdurano**! Welcome to my first MVC 5 Web App!

© 2015 - My ASP.NET Application

Figure 24: Successful logging-in

After logging out



http://localhost:15599/

Index - My ASP.NET A...

Application name

Index

No Account yet? Click here to sign-up

© 2015 - My ASP.NET Application

Figure 25: After Logging-out

That simple! Now let's take a look at how we are going to implement a simple role-based page authorization.

Implementing a Simple Role-Based Page Authorization

Authorization is a function that specifies access rights to a certain resource or page. One practical example is having a page that only a certain user role can have access to it. For example, only allow administrator to access the maintenance page for your application. In this section we will create a simple implementation on how to achieve that.

Creating the IsUserInRole() Method

Add the following code below at “UserManager” class:

```
public bool IsUserInRole(string loginName, string roleName) {
            using (DemoDBEntities db = new DemoDBEntities()) {
                SYSUser SU = db.SYSUsers.Where(o =>
o.LoginName.ToLower().Equals(loginName))?.FirstOrDefault();
                if (SU != null) {
                    var roles = from q in db.SYSUserRoles
                                join r in db.LOOKUPRoles on q.LOOKUPRoleID equals
r.LOOKUPRoleID
                                where r.RoleName.Equals(roleName) &&
q.SYSUserID.Equals(SU.SYSUserID)
                                select r.RoleName;

                    if (roles != null) {
                        return roles.Any();
                    }
                }

                return false;
            }

}
```

The method above takes the loginName and roleName as parameters. What it does is it checks for the existing records in the “SYSUser” table and then validates if the corresponding user has roles assigned to it.

Creating a Custom Authorization Attribute Filter

If you remember we are using the [Authorize] attribute to restrict anonymous users from accessing a certain action method. The [Authorize] attribute provides filters for users and roles and it’s fairly easy to implement it if you are using membership provider. Since we are using our own database for storing users and roles then we need to implement our own authorization filter by extending the AuthorizeAttribute class.

AuthorizeAttribute specifies that access to a controller or action method is restricted to users who meet the authorization requirement. Our goal here to allow page authorization based on user roles and nothing else. If you want to implement custom filters to do certain task and value separation of concerns then you may want to look at IAutenticationFilter instead.

To start, add a new folder and name it as "Security". Then add the "AuthorizeRoleAttribute" class. Here's a screen shot of the structure below:

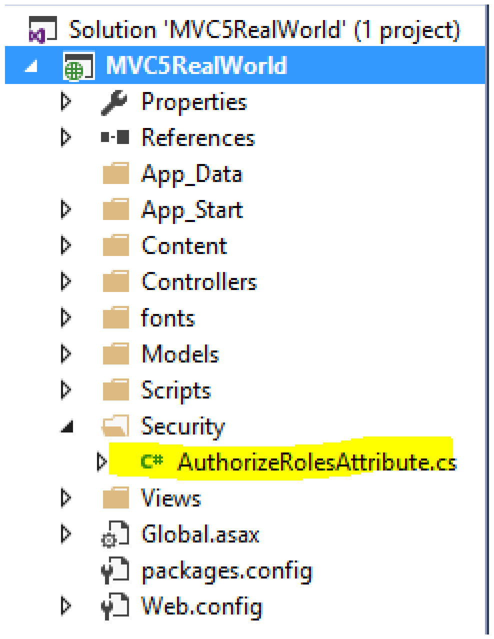


Figure 26: The AuthorizeRoleAttribute class location

Here's the code block for our custom filter:

```
using System.Web;
using System.Web.Mvc;
using MVC5RealWorld.Models.DB;
using MVC5RealWorld.Models.EntityManager;

namespace MVC5RealWorld.Security
{
    public class AuthorizeRolesAttribute : AuthorizeAttribute
    {
        private readonly string[] userAssignedRoles;
        public AuthorizeRolesAttribute(params string[] roles) {
            this.userAssignedRoles = roles;
        }
        protected override bool AuthorizeCore(HttpContextBase httpContext) {
            bool authorize = false;
            using (DemoDBEntities db = new DemoDBEntities()) {
                UserManager UM = new UserManager();
                foreach (var roles in userAssignedRoles) {
                    authorize = UM.IsUserInRole(httpContext.User.Identity.Name, roles);
                    if (authorize)
                        return authorize;
                }
            }
            return authorize;
        }
        protected override void HandleUnauthorizedRequest(AuthorizationContext
filterContext) {
            filterContext.Result = new RedirectResult("~/Home/UnAuthorized");
        }
    }

}
```

There are two main methods in the class above that we have overridden. The **AuthorizeCore()** method is the entry point for the authentication check. This is where we check the roles assigned for a certain users and returns the result if the user is allowed to access a page or not. The **HandleUnuathorizedRequest()** is a method in which we redirect un-authorized users to a certain page.

Adding the AdminOnly and UnAuthorized page

Now switch back to "HomeController" and add the following code:

```
[AuthorizeRoles("Admin")]
public ActionResult AdminOnly() {
       return View();
}

public ActionResult UnAuthorized() {
       return View();

}
```

If you notice we decorated the **AdminOnly** action with our custom authorization filter by passing the value of "Admin" as the role name. This means that only allow admin users to access the "AdminOnly" page. To support multiple role access, just add another role name by separating it with comma for example [AuthorizeRoles("Admin","Manager")]. Note that the value of "Admin" and "Manager" should match with the role names from your database for it to work. And finally, make sure to reference the namespace below before using the AuthorizeRoles attribute:

```
using MVC5RealWorld.Security;
```

Here's the AdminOnly.cshtml view:

```
@{
    ViewBag.Title = "AdminOnly";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>For Admin users only!</h2>
```

And here's the UnAuthorized.cshtml view:

```
@{
    ViewBag.Title = "UnAuthorized";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<h2>Unauthorized Access!</h2>
<p>Oops! You don't have permission to access this page.</p>

<div>

    @Html.ActionLink("Back to Main", "Welcome", "Home")

</div>
```

Adding Test Roles Data

Before we test the functionality lets add an admin user to the database first. For this demo I have inserted the following data to the database:

```
INSERT INTO SYSUser (LoginName,PasswordEncryptedText, RowCreatedSYSUserID,
RowModifiedSYSUserID)
VALUES ('Admin','Admin',1,1)
GO
INSERT INTO SYSUserProfile (SYSUserID,FirstName,LastName,Gender,RowCreatedSYSUserID,
RowModifiedSYSUserID)
VALUES (2,'Vinz','Durano','M',1,1)
GO

INSERT INTO SYSUserRole (SYSUserID,LOOKUPRoleID,IsActive,RowCreatedSYSUserID,
RowModifiedSYSUserID)
VALUES (2,1,1,1,1)
```

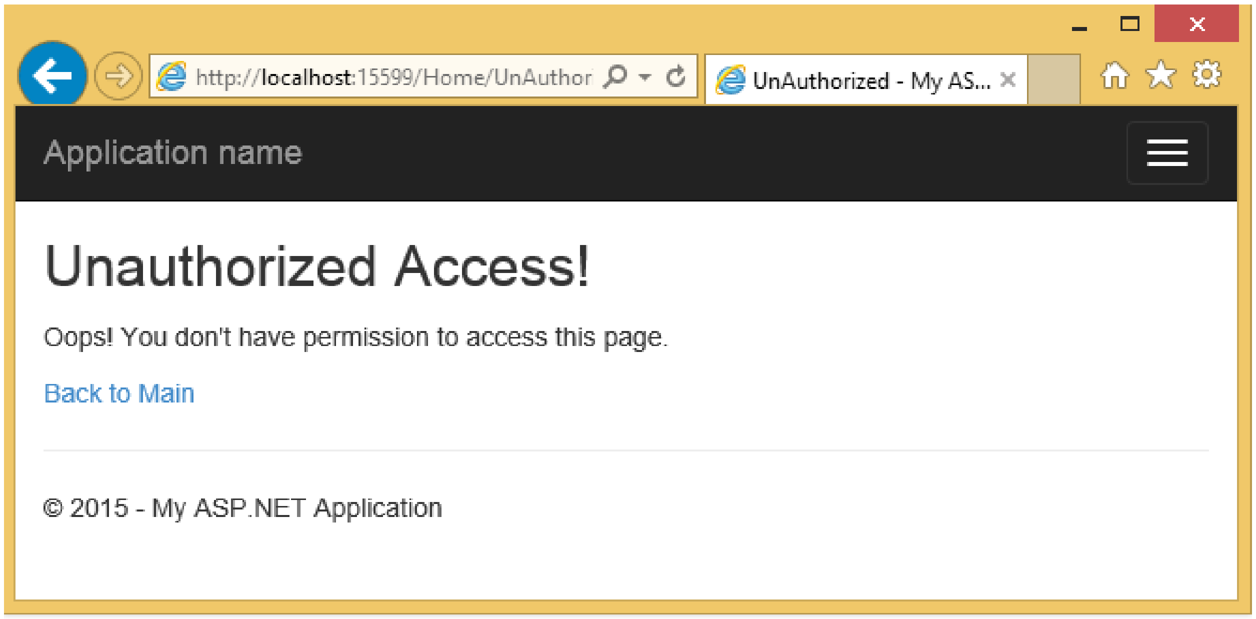
Okay now we have some data to test and we are ready to run the application.

Running the Application

Here are some of the screenshots captured during my test:

When logging in as normal user and accessing the following URL:

http://localhost:15599/Home/AdminOnly



http://localhost:15599/Home/UnAuthor

UnAuthorized - My AS...

Application name

Unauthorized Access!

Oops! You don't have permission to access this page.

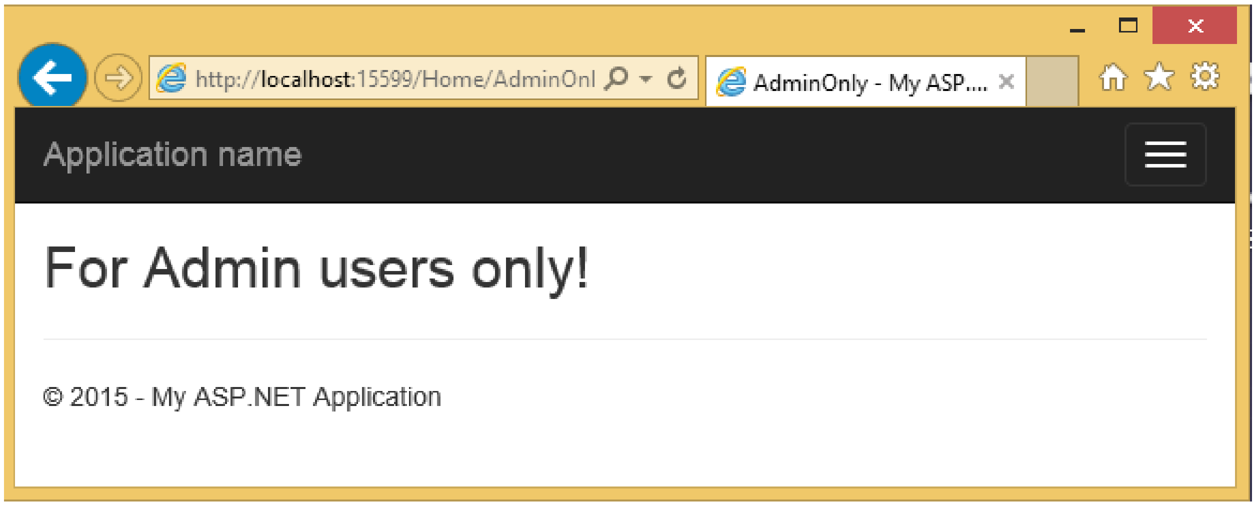
Back to Main

© 2015 - My ASP.NET Application

Figure 27: Unauthorized access

When logging in as an Admin user and accessing the following URL:

http://localhost:15599/Home/AdminOnly



http://localhost:15599/Home/AdminOnl

AdminOnly - My ASP....

Application name

For Admin users only!

© 2015 - My ASP.NET Application

Figure 28: Admin page

Implementing Fetch, Edit, Update and Delete Operations

In previous sections you've learned about creating a simple database from scratch using MS SQL Server, a brief overview about ASP.NET MVC in general, creating a data access using Entity Framework database first approach and a simple implementation of a Signup page in ASP.NET MVC . You've also learned the step-by-step process on creating a basic login page and creating a simple role-based page authorization within your ASP.NET MVC application.

In this section, I'm going to walk you through about how to perform Fetch, Edit, Update and Delete (FEUD) operations in our application. The idea is to create a maintenance page where admin users can modify user profiles. There are many possible ways to implement FEUD operations in you MVC app depending on your business needs. For this particular demo, I'm going to use jQuery and jQuery AJAX to perform asynchronous operation in our page.

Let's get started!

Fetching and Displaying the Data

For this example, I'm going to create a **PartialView** for displaying the list of users from the database. Partial Views allow you to define a view that will be rendered inside a main view. If you are using WebForms before then you can think of partial views as user-controls (.ascx).

Adding the View Models

The first thing we need is to create view models for our view. Add the following code below within "UserModel.cs" class:

```
public class UserProfileView
{
    [Key]
    public int SYSUserID { get; set; }
    public int LOOKUPRoleID { get; set; }
    public string RoleName { get; set; }
    public bool? IsRoleActive { get; set; }
    [Required(ErrorMessage = "*")]
    [Display(Name = "Login ID")]
    public string LoginName { get; set; }
    [Required(ErrorMessage = "*")]
    [Display(Name = "Password")]
    public string Password { get; set; }
    [Required(ErrorMessage = "*")]
    [Display(Name = "First Name")]
    public string FirstName { get; set; }
    [Required(ErrorMessage = "*")]
    [Display(Name = "Last Name")]
    public string LastName { get; set; }
    public string Gender { get; set; }
}

public class LOOKUPAvailableRole
{
    [Key]
    public int LOOKUPRoleID { get; set; }
    public string RoleName { get; set; }
    public string RoleDescription { get; set; }
}

public class Gender
{
    public string Text { get; set; }
    public string Value { get; set; }
}
public class UserRoles
{
    public int? SelectedRoleID { get; set; }
    public IEnumerable<LOOKUPAvailableRole> UserRoleList { get; set; }
}

public class UserGender
{
    public string SelectedGender { get; set; }
    public IEnumerable<Gender> Gender { get; set; }
}
public class UserDataView
{
    public IEnumerable<UserProfileView> UserProfile { get; set; }
    public UserRoles UserRoles { get; set; }
    public UserGender UserGender { get; set; }
}
```

If you still remember, View Model is a model that houses some properties that we only need for the view or page.

Now Open “UserManager” class and declare the namespace below:

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

The namespace above contain interfaces and classes that define generic collections, which allow us to create strongly-typed collections. Now add the following code below in “UserManager” class:

```
      public List<LOOKUPAvailableRole> GetAllRoles() {
            using (DemoDBEntities db = new DemoDBEntities()) {
                var roles = db.LOOKUPRoles.Select(o => new LOOKUPAvailableRole {
                    LOOKUPRoleID = o.LOOKUPRoleID,
                    RoleName = o.RoleName,
                    RoleDescription = o.RoleDescription
                }).ToList();

                return roles;
            }
```

```
        }

        public int GetUserID(string loginName) {
            using (DemoDBEntities db = new DemoDBEntities()) {
                var user = db.SYSUsers.Where(o => o.LoginName.Equals(loginName));
                if (user.Any())
                    return user.FirstOrDefault().SYSUserID;
            }
            return 0;
        }
        public List<UserProfileView> GetAllUserProfiles() {
            List<UserProfileView> profiles = new List<UserProfileView>();
            using(DemoDBEntities db = new DemoDBEntities()) {
                UserProfileView UPV;
                var users = db.SYSUsers.ToList();

                foreach(SYSUser u in db.SYSUsers) {
                    UPV = new UserProfileView();
                    UPV.SYSUserID = u.SYSUserID;
                    UPV.LoginName = u.LoginName;
                    UPV.Password = u.PasswordEncryptedText;

                    var SUP = db.SYSUserProfiles.Find(u.SYSUserID);
                    if(SUP != null) {
                        UPV.FirstName = SUP.FirstName;
                        UPV.LastName = SUP.LastName;
                        UPV.Gender = SUP.Gender;
                    }

                    var SUR = db.SYSUserRoles.Where(o =>
o.SYSUserID.Equals(u.SYSUserID));
                    if (SUR.Any()) {
                        var userRole = SUR.FirstOrDefault();
                        UPV.LOOKUPRoleID = userRole.LOOKUPRoleID;
                        UPV.RoleName = userRole.LOOKUPRole.RoleName;
                        UPV.IsRoleActive = userRole.IsActive;
                    }

                    profiles.Add(UPV);
                }
            }

            return profiles;
        }

        public UserDataView GetUserDataView(string loginName) {
            UserDataView UDV = new UserDataView();
            List<UserProfileView> profiles = GetAllUserProfiles();
            List<LOOKUPAvailableRole> roles = GetAllRoles();

            int? userAssignedRoleID = 0, userID = 0;
            string userGender = string.Empty;

            userID = GetUserID(loginName);
            using (DemoDBEntities db = new DemoDBEntities()) {
                userAssignedRoleID = db.SYSUserRoles.Where(o => o.SYSUserID ==
userID)?.FirstOrDefault().LOOKUPRoleID;
                userGender = db.SYSUserProfiles.Where(o => o.SYSUserID ==
userID)?.FirstOrDefault().Gender;
            }

            List<Gender> genders = new List<Gender>();
            genders.Add(new Gender { Text = "Male", Value = "M" });
            genders.Add(new Gender { Text = "Female", Value = "F" });

            UDV.UserProfile = profiles;
            UDV.UserRoles = new UserRoles { SelectedRoleID = userAssignedRoleID,
UserRoleList = roles };
            UDV.UserGender = new UserGender { SelectedGender = userGender, Gender =
genders };
            return UDV;
        }
```

The methods shown from the code above is pretty much self-explanatory as their method names suggest. The main method there is the **GetUserDataView ()** which gets all user profiles and roles. The **UserRoles** and **UserGender** properties will be used during editing and updating of user data. We will use those values to populate the dropdown lists for roles and gender.

Adding the ManageUserPartial Action Method

Open “HomeController.cs” class and add the following namespaces below:

```
using System.Web.Security;
using MVC5RealWorld.Models.ViewModel;
using MVC5RealWorld.Models.EntityManager;
```

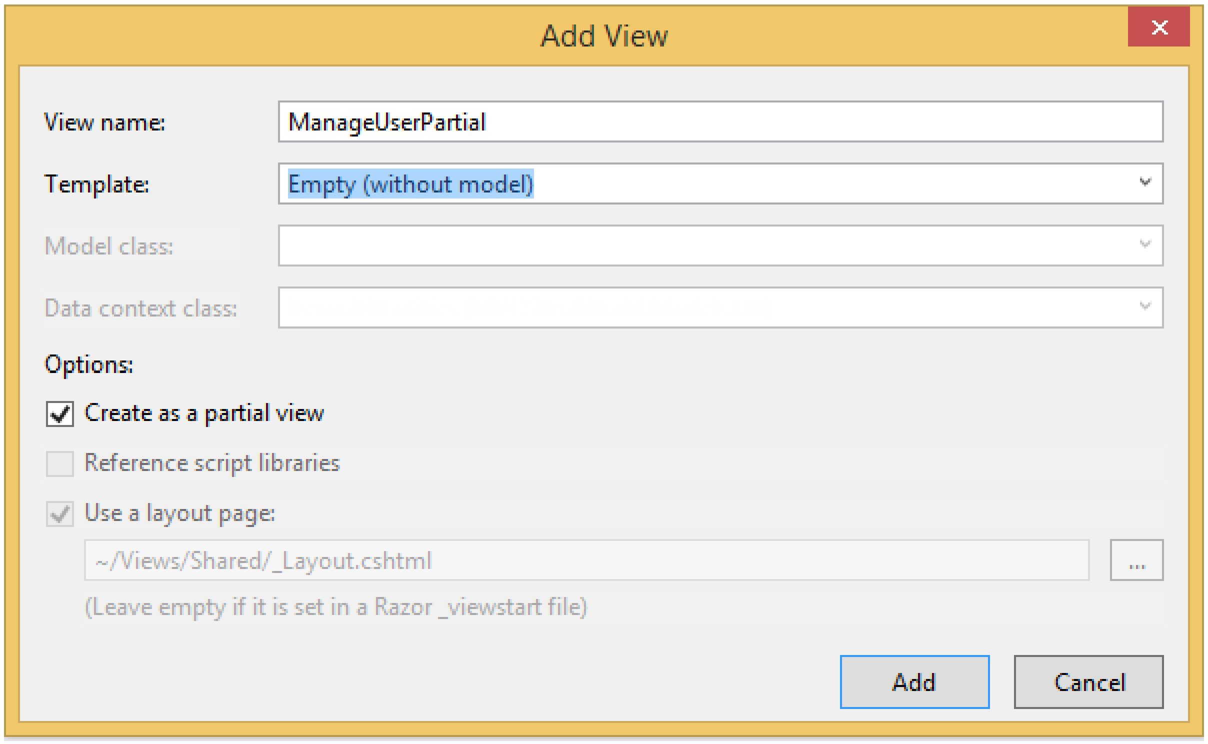
And then add the following action method below:

```
       [AuthorizeRoles("Admin")]
        public ActionResult ManageUserPartial() {
            if (User.Identity.IsAuthenticated) {
                string loginName = User.Identity.Name;
                UserManager UM = new UserManager();
                UserDataView UDV = UM.GetUserDataView(loginName);
                return PartialView(UDV);
            }

            return View();
        }
```

The code above is decorated with the custom Authorize attribute so that only admin users can invoke that method. What it does is it calls the **GetUserDataView()** method by passing in the loginName as the parameter and return the result in the Partial View.

Adding the ManageUserPartial Partial View

Now let's create the Partial View. Right click on the "ManageUserPartial" method and select "Add New" view. This will bring up the following dialog:



View name: ManageUserPartial

Template: Empty (without model)

Model class:

Data context class:

Options:

[✓] Create as a partial view

[ ] Reference script libraries

[✓] Use a layout page:

~/Views/Shared/_Layout.cshtml

(Leave empty if it is set in a Razor _viewstart file)

Figure 29: Admin page

Since we are going to create a custom view for managing the users then just select an "Empty" template and make sure to tick the "**Create as a partial view**" option. Click "Add" and then copy the following HTML markup below:

```
@model MVC5RealWorld.Models.ViewModel.UserDataView

<div>
    <h1>List of Users</h1>
    <span class="alert-success">@ViewBag.Message</span>
    <table class="table table-striped table-condensed table-hover">
        <thead>
            <tr>
                <th>ID</th>
                <th>Login ID</th>
                <th>Password</th>
                <th>First Name</th>
                <th>Last Name</th>
                <th>Gender</th>
                <th colspan="2">Role</th>
                <th></th>
                <th></th>
            </tr>
        </thead>
        <tbody>
            @foreach (var i in Model.UserProfile) {
                <tr>
                    <td> @Html.DisplayFor(m => i.SYSUserID)</td>
                    <td> @Html.DisplayFor(m => i.LoginName)</td>
                    <td> @Html.DisplayFor(m => i.Password)</td>
                    <td> @Html.DisplayFor(m => i.FirstName)</td>
                    <td> @Html.DisplayFor(m => i.LastName)</td>
                    <td> @Html.DisplayFor(m => i.Gender)</td>
                    <td> @Html.DisplayFor(m => i.RoleName)</td>
                    <td> @Html.HiddenFor(m => i.LOOKUPRoleID)</td>
                    <td><a href="javacript:void(0)" class="lnkEdit">Edit</a></td>
                    <td><a href="javacript:void(0)" class="lnkDelete">Delete</a></td>
                </tr>
            }
        </tbody>
    </table>
</div>
```

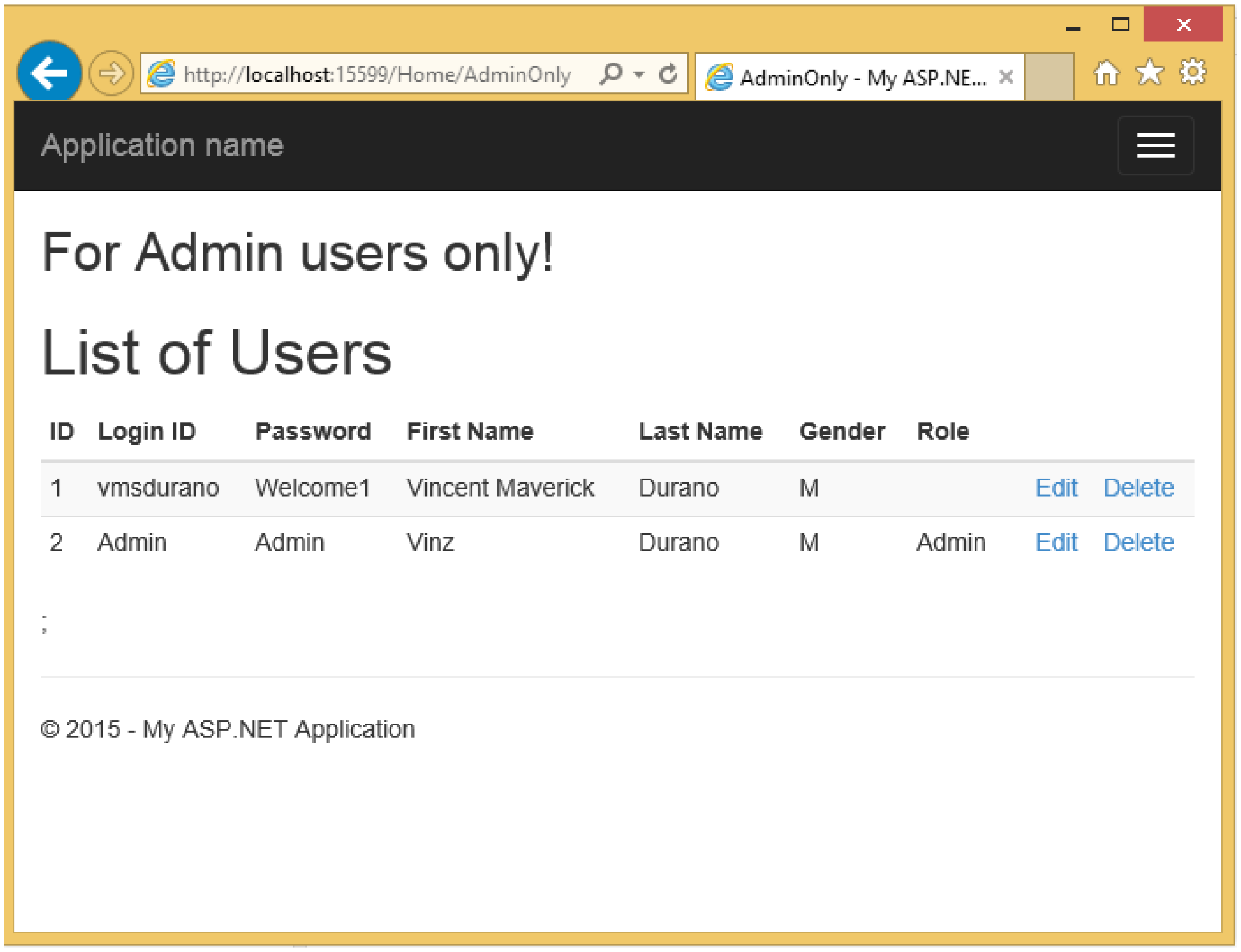
The markup above is a strongly-typed View which renders the **UserDataView** model. By specifying the type of data, you can get access to data associated within the model instead of using the general ViewData/ViewBag structure and most importantly able to use IntelliSense feature in Visual Studio.

Now open the “AdminOnly.cshtml” view and add the following markup:

```
<div id="divUserListContainer">
    @Html.Action("ManageUserPartial", "Home");
</div>
```

Running the Application

Now try to login to your web page then navigate to: http://localhost:15599/Home/AdminOnly . The output should look something like this:



http://localhost:15599/Home/AdminOnly

AdminOnly - My ASP.NE...

Application name

For Admin users only!

List of Users

ID	Login ID	Password	First Name	Last Name	Gender	Role		
1	vmsdurano	Welcome1	Vincent Maverick	Durano	M		Edit	Delete
2	Admin	Admin	Vinz	Durano	M	Admin	Edit	Delete

;

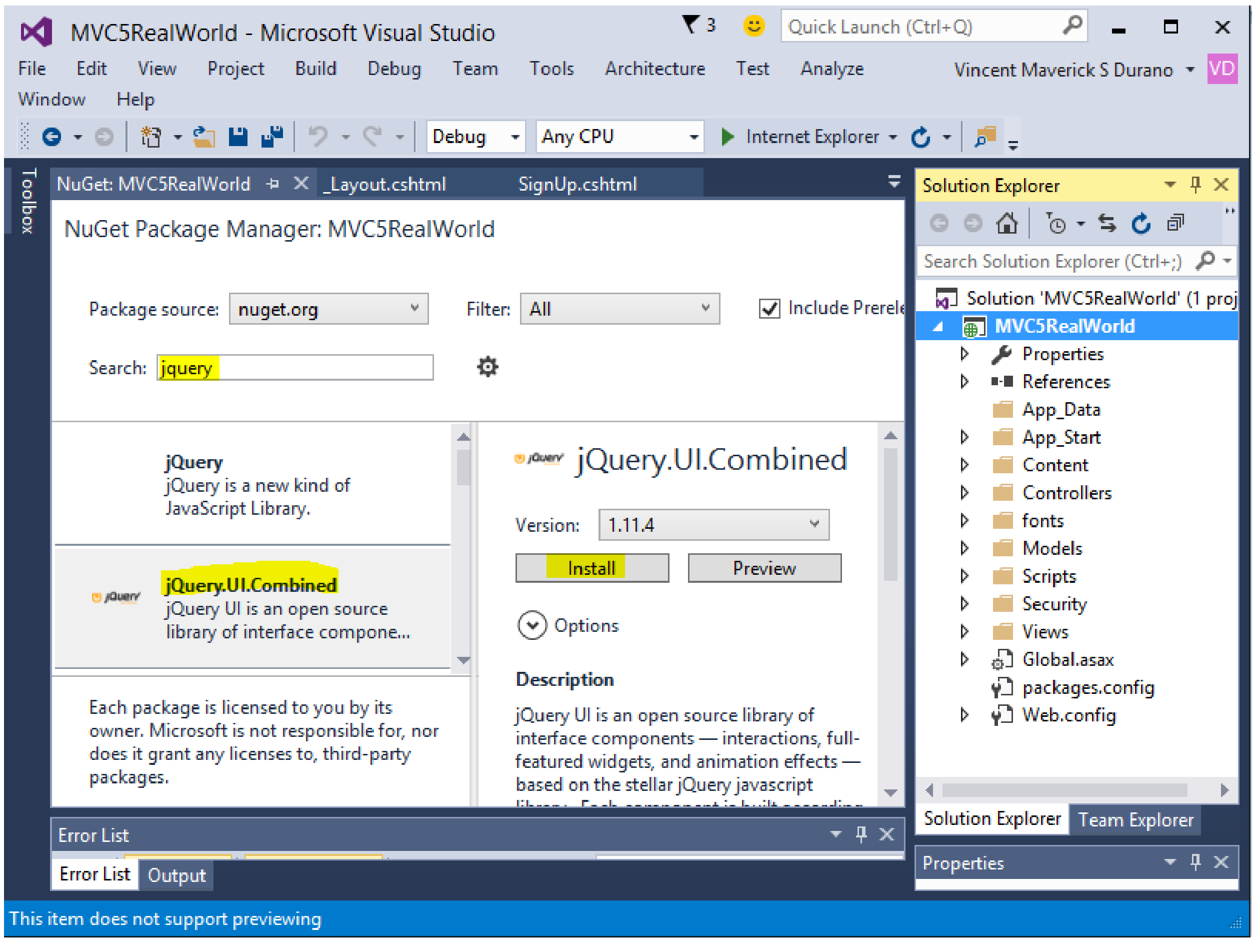
© 2015 - My ASP.NET Application

Figure 29: List of Users

Pretty much easy right? ☺ Now let's move to the next step.

Editing and Updating the Data

Since we are going to use jQueryUI for presenting a dialog box for the user to edit the data, then we need to add a reference to it first. To do that, just right click on your project and then select "Manage Nuget Packages". In the search box type in "jquery" and select "jQuery.UI.Combined" as shown in the image below:



NuGet Package Manager: MVC5RealWorld

Package source: nuget.org Filter: All ☑ Include Prerele

Search: jquery

jQuery
jQuery is a new kind of JavaScript Library.

jQuery.UI.Combined
jQuery UI is an open source library of interface compone...

Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.

jQuery.UI.Combined

Version: 1.11.4

Description

jQuery UI is an open source library of interface components — interactions, full-featured widgets, and animation effects — based on the stellar jQuery javascript

This item does not support previewing

Figure 30: Adding jQuery as NuGet package

Once installed the jQueryUI library should be added in your project under the “Scripts” folder:

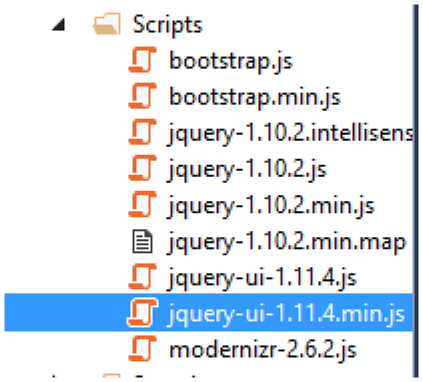

Scripts
bootstrap.js
bootstrap.min.js
jquery-1.10.2.intellisens
jquery-1.10.2.js
jquery-1.10.2.min.js
jquery-1.10.2.min.map
jquery-ui-1.11.4.js
jquery-ui-1.11.4.min.js
modernizr-2.6.2.js

Figure 31: The jQuery and jQueryUI scripts

Now go to Views > Shared >_Layout.cshtml and add the jQueryUI reference in the following order:

```
    <script src="~/Scripts/jquery-1.10.2.min.js"></script>
    <script src="~/Scripts/jquery-ui-1.11.4.min.js"></script>
```

The jQueryUI should be referenced after jQuery library since jQueryUI uses the core jQuery library under the hood.

Now add the jQueryUI CSS reference:

```
<link href="~/Content/themes/base/all.css" rel="stylesheet" />
```

Your **_Layout.cshtml** markup should look something like below with the added references to jQuery and jQueryUI:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@ViewBag.Title - My ASP.NET Application</title>
    <link href="~/Content/Site.css" rel="stylesheet" type="text/css" />
    <link href="~/Content/bootstrap.min.css" rel="stylesheet" type="text/css" />
    <script src="~/Scripts/modernizr-2.6.2.js"></script>
    <script src="~/Scripts/jquery-1.10.2.min.js"></script>
    <script src="~/Scripts/jquery-ui-1.11.4.min.js"></script>
    <link href="~/Content/themes/base/all.css" rel="stylesheet" />
</head>
<body>
    <div class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-
target=".navbar-collapse">
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                @Html.ActionLink("Application name", "Index", "Home", new { area = "" },
new { @class = "navbar-brand" })
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                </ul>
            </div>
        </div>
    </div>

    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
        </footer>
    </div>


    <script src="~/Scripts/bootstrap.min.js"></script>
</body>
</html>
```

Adding the UpdateUserAccount() Method

Keep in mind that this demo is intended to make an app as simple as possible. In complex real-scenarios I would strongly suggest you to use a Repository pattern and Unit-of-Work for your data access layer.

Add the following code below within “UserManager.cs” class:

```
public void UpdateUserAccount(UserProfileView user) {

  using (DemoDBEntities db = new DemoDBEntities()) {
   using (var dbContextTransaction = db.Database.BeginTransaction()) {
    try {

      SYSUser SU = db.SYSUsers.Find(user.SYSUserID);
      SU.LoginName = user.LoginName;
      SU.PasswordEncryptedText = user.Password;
      SU.RowCreatedSYSUserID = user.SYSUserID;
      SU.RowModifiedSYSUserID = user.SYSUserID;
      SU.RowCreatedDateTime = DateTime.Now;
      SU.RowMOdifiedDateTime = DateTime.Now;

      db.SaveChanges();

      var userProfile = db.SYSUserProfiles.Where(o => o.SYSUserID == user.SYSUserID);
        if (userProfile.Any()) {
          SYSUserProfile SUP = userProfile.FirstOrDefault();
          SUP.SYSUserID = SU.SYSUserID;
          SUP.FirstName = user.FirstName;
          SUP.LastName = user.LastName;
          SUP.Gender = user.Gender;
          SUP.RowCreatedSYSUserID = user.SYSUserID;
          SUP.RowModifiedSYSUserID = user.SYSUserID;
          SUP.RowCreatedDateTime = DateTime.Now;
          SUP.RowModifiedDateTime = DateTime.Now;

            db.SaveChanges();
              }

              if (user.LOOKUPRoleID > 0) {
                var userRole = db.SYSUserRoles.Where(o => o.SYSUserID == user.SYSUserID);
                    SYSUserRole SUR = null;
                      if (userRole.Any()) {
                        SUR = userRole.FirstOrDefault();
                        SUR.LOOKUPRoleID = user.LOOKUPRoleID;
```

```
                                    SUR.SYSUserID = user.SYSUserID;
                                    SUR.IsActive = true;
                                    SUR.RowCreatedSYSUserID = user.SYSUserID;
                                    SUR.RowModifiedSYSUserID = user.SYSUserID;
                                    SUR.RowCreatedDateTime = DateTime.Now;
                                     SUR.RowModifiedDateTime = DateTime.Now;
                                      }
                                      else {
                                       SUR = new SYSUserRole();
                                       SUR.LOOKUPRoleID = user.LOOKUPRoleID;
                                       SUR.SYSUserID = user.SYSUserID;
                                       SUR.IsActive = true;
                                       SUR.RowCreatedSYSUserID = user.SYSUserID;
                                       SUR.RowModifiedSYSUserID = user.SYSUserID;
                                       SUR.RowCreatedDateTime = DateTime.Now;
                                       SUR.RowModifiedDateTime = DateTime.Now;
                                        db.SYSUserRoles.Add(SUR);
                                        }

                                        db.SaveChanges();
                                    }
                                    dbContextTransaction.Commit();
                                }
                                catch {
                                    dbContextTransaction.Rollback();
                                }
                            }
                        }
                    }
```

The method above takes **UserProfileView** object as the parameter. This parameter object is coming from a strongly-typed View. What it does is it first issues a query to the database using the LINQ syntax to get the specific user data by passing the SYSUserID. It then updates the **SYSUser** object with the corresponding data from the **UserProfileView** object. The second query gets the associated **SYSUserProfiles** data and then updates the corresponding values. After that it then looks for the associated **LOOKUPRoleID** for a certain user. If the user doesn't have role assigned to it then it adds a new record to the database otherwise just update the table.

If you also noticed, I used a simple transaction within that method. This is because the tables SYSUser, SYSUserProfile and SYSUserRole have dependencies to each other and we need to make sure that we only commit changes to the database if the operation for each table is successful. The **Database.BeginTransaction()** is only available in EF 6 onwards.

Adding the UpdateUserData Action Method

Add the following code within "HomeController" class:

```
       [AuthorizeRoles("Admin")]
        public ActionResult UpdateUserData(int userID, string loginName, string password,
string firstName, string lastName, string gender, int roleID = 0) {
            UserProfileView UPV = new UserProfileView();
            UPV.SYSUserID = userID;
            UPV.LoginName = loginName;
            UPV.Password = password;
            UPV.FirstName = firstName;
            UPV.LastName = lastName;
            UPV.Gender = gender;

            if (roleID > 0)
                UPV.LOOKUPRoleID = roleID;

            UserManager UM = new UserManager();
            UM.UpdateUserAccount(UPV);

            return Json(new { success = true });
        }
```

The method above is responsible for collecting data that is sent from the View for update. It then calls the method **UpdateUserAccount()** and pass the **UserProfileView** model view as the parameter. The **UpdateUserData** method will be called through an AJAX request.

Modifying the UserManagePartial View

Add the following HTML markup within “UserManagePartial.cshtml”:

```
<div id="divEdit" style="display:none">
        <input type="hidden" id="hidID" />
        <table>
            <tr>
                <td>Login Name</td>
                <td><input type="text" id="txtLoginName" class="form-control" /></td>
            </tr>
            <tr>
                <td>Password</td>
                <td><input type="text" id="txtPassword" class="form-control" /></td>
            </tr>
            <tr>
                <td>First Name</td>
                <td><input type="text" id="txtFirstName" class="form-control" /></td>
            </tr>
            <tr>
                <td>Last Name</td>
                <td><input type="text" id="txtLastName" class="form-control" /></td>
            </tr>
            <tr>
                <td>Gender</td>
                <td>@Html.DropDownListFor(o => o.UserGender.SelectedGender,
                       new SelectList(Model.UserGender.Gender, "Value", "Text"),
                       "",
                       new { id = "ddlGender", @class="form-control" })
                </td>
            </tr>
            <tr>
                <td>Role</td>
                <td>@Html.DropDownListFor(o => o.UserRoles.SelectedRoleID,
new SelectList(Model.UserRoles.UserRoleList, "LOOKUPRoleID", "RoleName"),
                       "",
                       new { id = "ddlRoles", @class="form-control" })
                </td>
            </tr>
        </table>
    </div>
```

Integrating jQuery and jQuery AJAX

Before we go to the implementation it's important to know what these technologies are.

jQuery is a light weight and feature-rich JavaScript library that enable DOM manipulation, even handling, animation and Ajax much simpler with powerful API that works across all major browsers.

jQueryUI provides a set of UI interactions, effects, widgets and themes built on top of the jQuery library.

jQuery AJAX enables you to use functions and methods to communicate with your data from the server and loads your data to the client/browser.

Now switch back to "UserManagePartial" View and add the following script block at the very bottom:

```
<script type="text/javascript">
    $(function () {

        var initDialog = function (type) {
            var title = type;
            $("#divEdit").dialog({
                autoOpen: false,
                modal: true,
                title: type + ' User',
                width: 360,
                buttons: {
                    Save: function () {
                        var id = $("#hidID").val();
                        var role = $("#ddlRoles").val();
                        var loginName = $("#txtLoginName").val();
                        var loginPass = $("#txtPassword").val();
                        var fName = $("#txtFirstName").val();
                        var lName = $("#txtLastName").val();
                        var gender = $("#ddlGender").val();

                        UpdateUser(id, loginName, loginPass, fName, lName, gender, role);
                        $(this).dialog("destroy");
                    },
                    Cancel: function () { $(this).dialog("destroy"); }
                }
            });
        }

        function UpdateUser(id, logName, logPass, fName, lName, gender, role) {
            $.ajax({
                type: "POST",
                url: "@(Url.Action("UpdateUserData","Home"))",
                data: { userID: id, loginName: logName, password: logPass, firstName:
fName, lastName: lName, gender: gender, roleID: role },
                success: function (data) {

$("#divUserListContainer").load("@(Url.Action("ManageUserPartial","Home", new { status
="update" }))");
                },
                error: function (error) {
                    //to do:
                }
            });
        }

        $("a.lnkEdit").on("click", function () {
            initDialog("Edit");
            $(".alert-success").empty();
            var row = $(this).closest('tr');

            $("#hidID").val(row.find("td:eq(0)").html().trim());
            $("#txtLoginName").val(row.find("td:eq(1)").html().trim())
            $("#txtPassword").val(row.find("td:eq(2)").html().trim())
            $("#txtFirstName").val(row.find("td:eq(3)").html().trim())
            $("#txtLastName").val(row.find("td:eq(4)").html().trim())
            $("#ddlGender").val(row.find("td:eq(5)").html().trim())
            $("#ddlRoles").val(row.find("td:eq(7) > input").val().trim());

            $("#divEdit").dialog("open");
            return false;
        });
    });

</script>
```

The **initDialog** initializes the jQueryUI dialog by customizing the dialog. We customized it by adding our own Save and Cancel button for us to write custom code implementation for each event. In the Save function we extracted each values from the edit form and pass these values to the **UpdateUser()** JavaScript function.

The **UpdateUser()** function issues an AJAX request using jQuery AJAX. The "type" parameter indicates what form method the request requires, in this case we set the type as "POST". The "url" is the path to the controller's method which we created in previous step. Note that the value of “url” can be a web service, web API or anything that host your data. The "data" is where we assign values to the method that requires parameter. If your method in the server doesn't require any parameter then you can leave this as empty using the value "{}". The "success" function is usually used when you do certain process if the request succeeds. In this case we load the Partial

View to reflect the changes on the View after we update the data. Keep in mind that we are passing a new parameter to the "ManageUserPartial" action that indicates the status of the request.

The last function is where we open the dialog when the user clicks on the "edit" link from the grid. This is also where we extract the data from the grid using jQuery selectors and populate the dialog fields with the extracted data.

Modifying the UserManagePartial Action Method

If you remember, we've added the new parameter "status" to the "UserManagePartial"method in our AJAX request so we need to update the method signature to accept a parameter. The new method should now look something like this:

```
        [AuthorizeRoles("Admin")]
         public ActionResult ManageUserPartial(string status = "") {
             if (User.Identity.IsAuthenticated) {
                 string loginName = User.Identity.Name;
                 UserManager UM = new UserManager();
                 UserDataView UDV = UM.GetUserDataView(loginName);

                 string message = string.Empty;
                 if (status.Equals("update"))
                     message = "Update Successful";
                 else if (status.Equals("delete"))
                     message = "Delete Successful";

                 ViewBag.Message = message;

                 return PartialView(UDV);
             }

             return RedirectToAction("Index", "Home");
     }
```

Displaying the Status Result

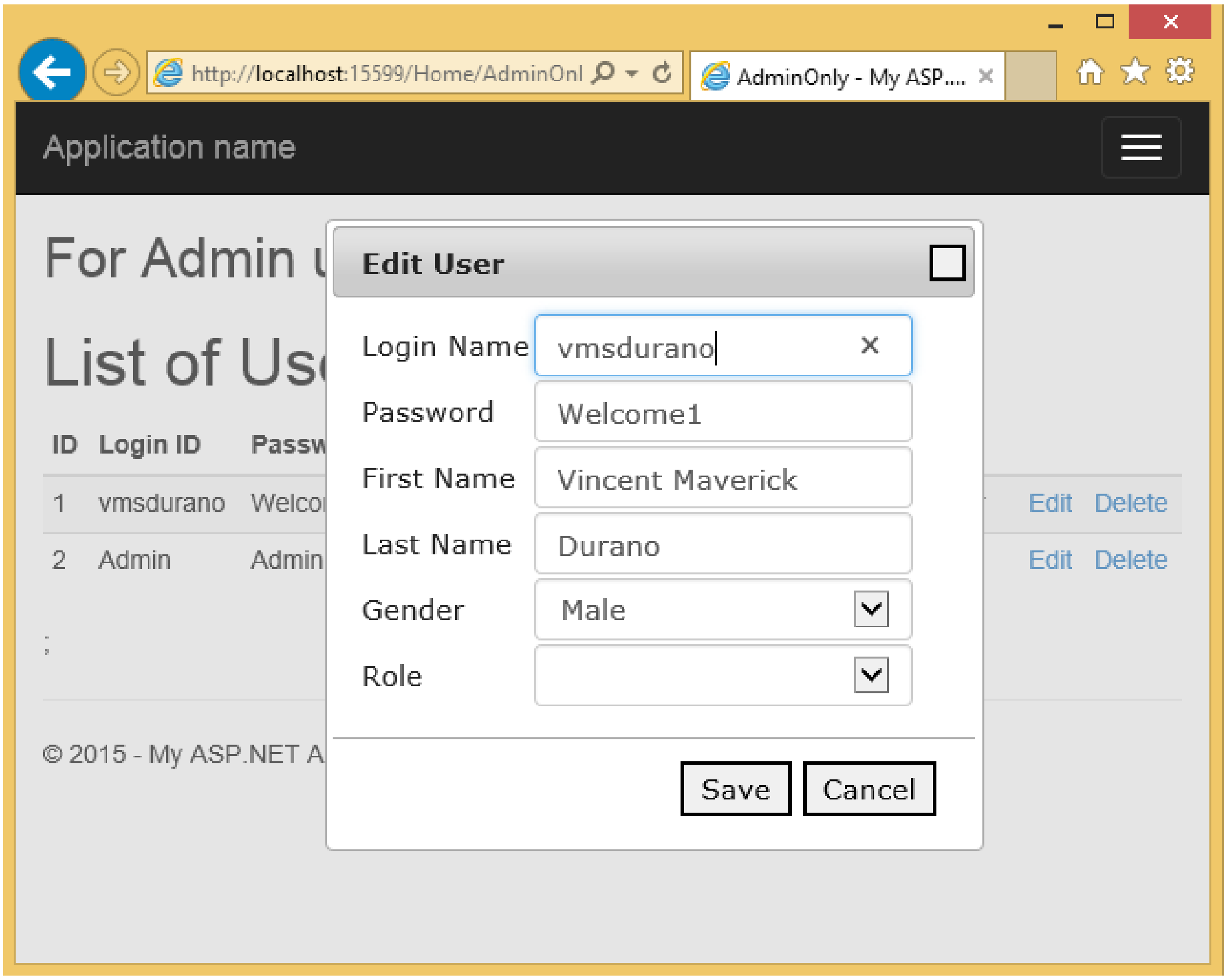
If you notice we are creating a message string based on a certain operation and store the result in ViewBag . This is to let user see if a certain operation succeeds. Now add the following markup below within "ManageUserPartial" view:

```
<span class="alert-success">@ViewBag.Message</span>
```

Running the Application

Here are the outputs below:

After clicking the edit dialog



For Admin u

List of Use

ID	Login ID	Passw		
1	vmsdurano	Welco	Edit	Delete
2	Admin	Admin	Edit	Delete

;

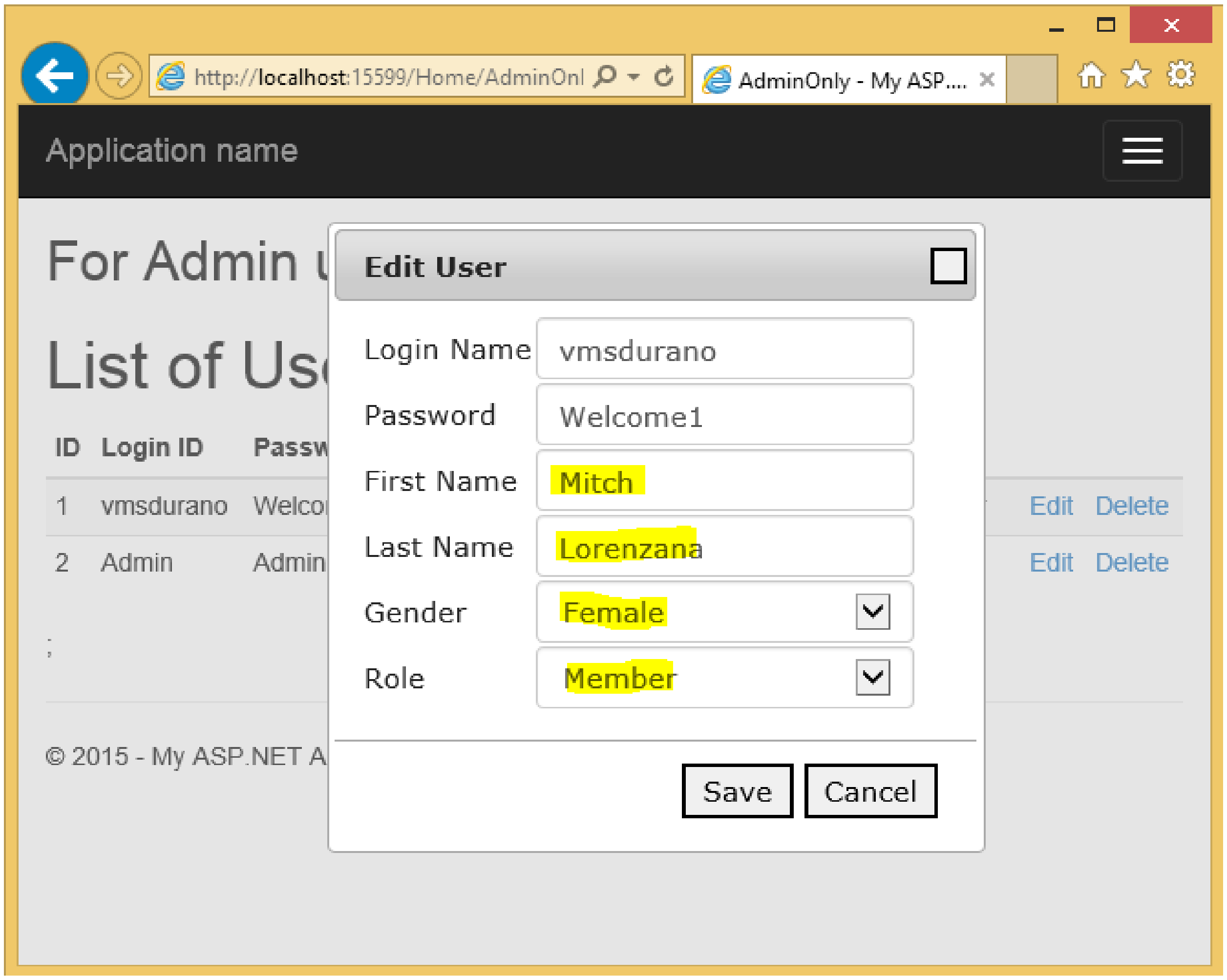
© 2015 - My ASP.NET A

Edit User

Login Name	vmsdurano
Password	Welcome1
First Name	Vincent Maverick
Last Name	Durano
Gender	Male
Role	

Figure 32: Editing the data

Editing the data

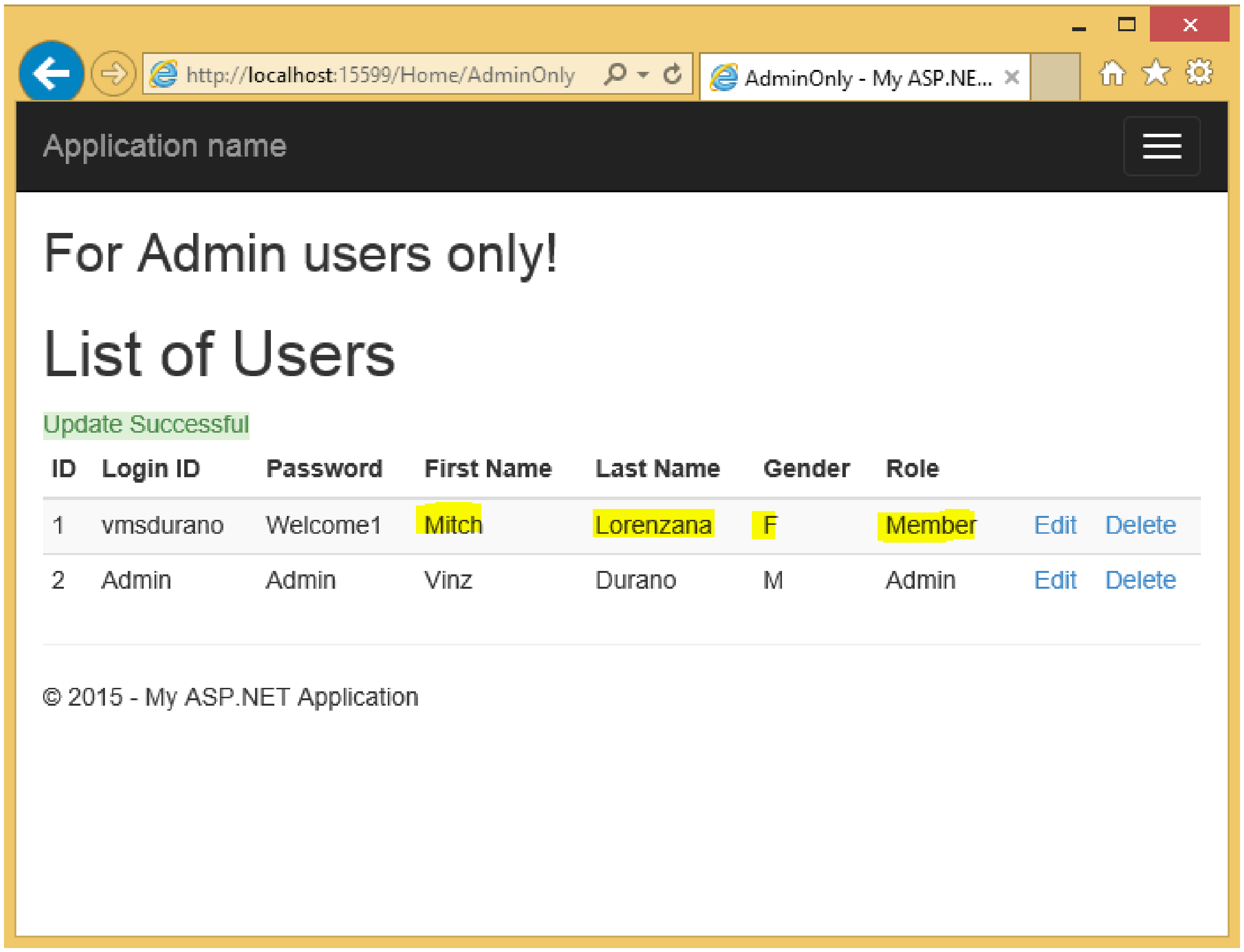


ID	Login ID	Passw
1	vmsdurano	Welco
2	Admin	Admin

Login Name	vmsdurano
Password	Welcome1
First Name	Mitch
Last Name	Lorenzana
Gender	Female
Role	Member

Figure 33: Modifying the data

After updating the data



For Admin users only!

List of Users

Update Successful

ID	Login ID	Password	First Name	Last Name	Gender	Role		
1	vmsdurano	Welcome1	Mitch	Lorenzana	F	Member	Edit	Delete
2	Admin	Admin	Vinz	Durano	M	Admin	Edit	Delete

© 2015 - My ASP.NET Application

Figure 34: Update successful

If you've made it this far then congratulations, you're now ready for the next step. Now down to the last part of this series. ☺

Deleting Data

Adding the DeleteUser() Method

Add the following method in "UserManager" class:

```
public void DeleteUser(int userID) {
     using (DemoDBEntities db = new DemoDBEntities()) {
         using (var dbContextTransaction = db.Database.BeginTransaction()) {
             try {

                 var SUR = db.SYSUserRoles.Where(o => o.SYSUserID == userID);
                 if (SUR.Any()) {
                     db.SYSUserRoles.Remove(SUR.FirstOrDefault());
```

```
                            db.SaveChanges();
                        }

                        var SUP = db.SYSUserProfiles.Where(o => o.SYSUserID == userID);
                        if (SUP.Any()) {
                            db.SYSUserProfiles.Remove(SUP.FirstOrDefault());
                            db.SaveChanges();
                        }

                        var SU = db.SYSUsers.Where(o => o.SYSUserID == userID);
                        if (SU.Any()) {
                            db.SYSUsers.Remove(SU.FirstOrDefault());
                            db.SaveChanges();
                        }

                        dbContextTransaction.Commit();
                    }
                    catch {
                        dbContextTransaction.Rollback();
                    }
                }
            }
        }
```

The method above deletes the record for a particular user in the SYSUserRole, SYSUserProfile and SYSUser tables by passing the SYSUserID as the parameter.

Adding the DeleteUser() Action Method

Add the following code within "HomeController" class:

```
        [AuthorizeRoles("Admin")]
         public ActionResult DeleteUser(int userID) {
             UserManager UM = new UserManager();
             UM.DeleteUser(userID);
             return Json(new { success = true });
         }
```

Integrating jQuery and jQuery AJAX

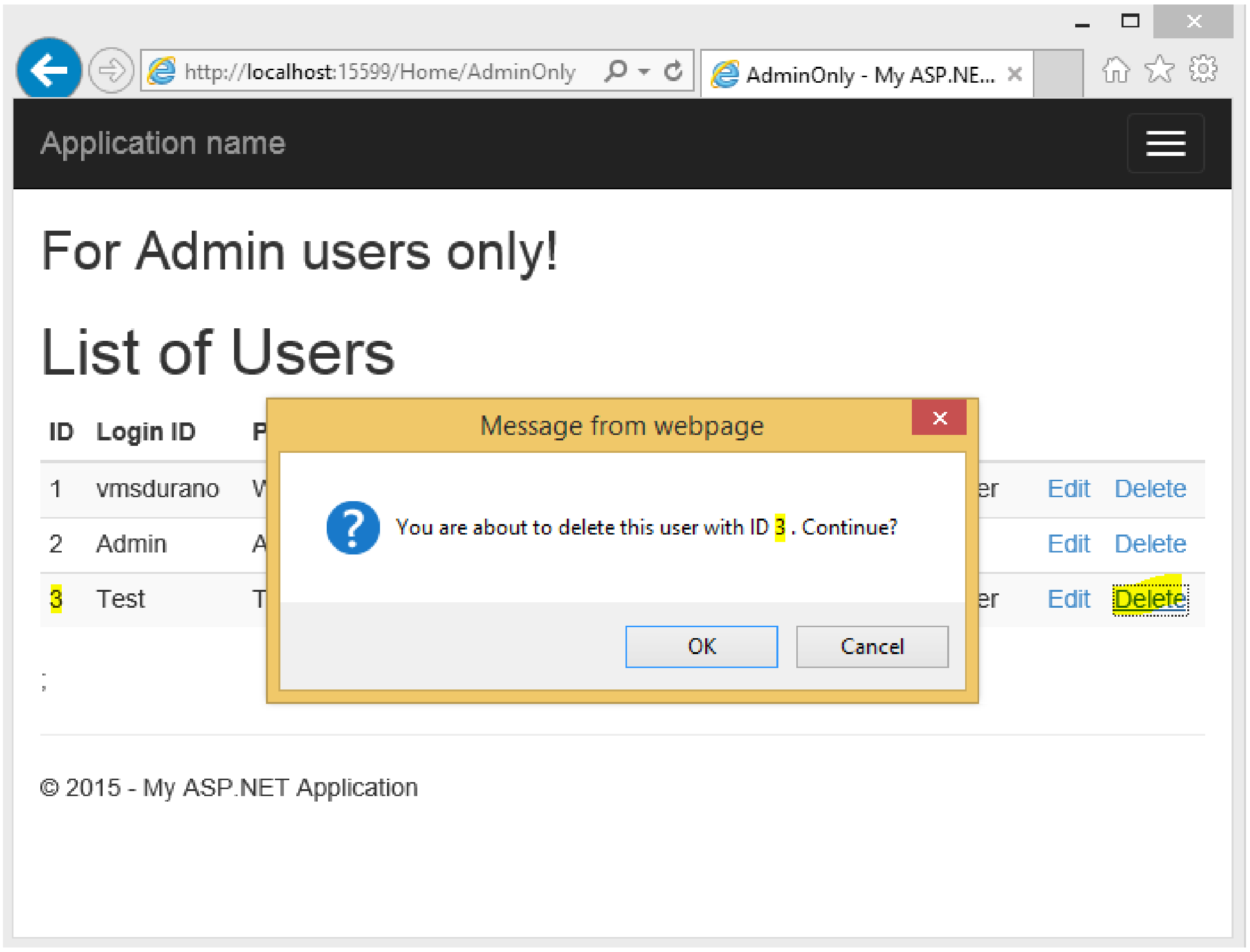
Add the following script within the <script> tag in "UserManagePartial" view:

```
        function DeleteUser(id) {
            $.ajax({
                type: "POST",
                url: "@(Url.Action("DeleteUser","Home"))",
                data: { userID: id },
                success: function (data) {

$("#divUserListContainer").load("@(Url.Action("ManageUserPartial","Home", new { status
="delete" }))");
                },
                error: function (error) { }
            });
        }

        $("a.lnkDelete").on("click", function () {
            var row = $(this).closest('tr');
            var id = row.find("td:eq(0)").html().trim();
            var answer = confirm("You are about to delete this user with ID " + id + " . 
Continue?");
            if (answer)
                DeleteUser(id);
            return false;
        });
```

Running the Application

Here are the outputs below:

After clicking the delete link



http://localhost:15599/Home/AdminOnly

AdminOnly - My ASP.NE...

Application name

For Admin users only!

List of Users

ID	Login ID	P			
1	vmsdurano	V	er	Edit	Delete
2	Admin	A		Edit	Delete
3	Test	T	er	Edit	Delete

Message from webpage

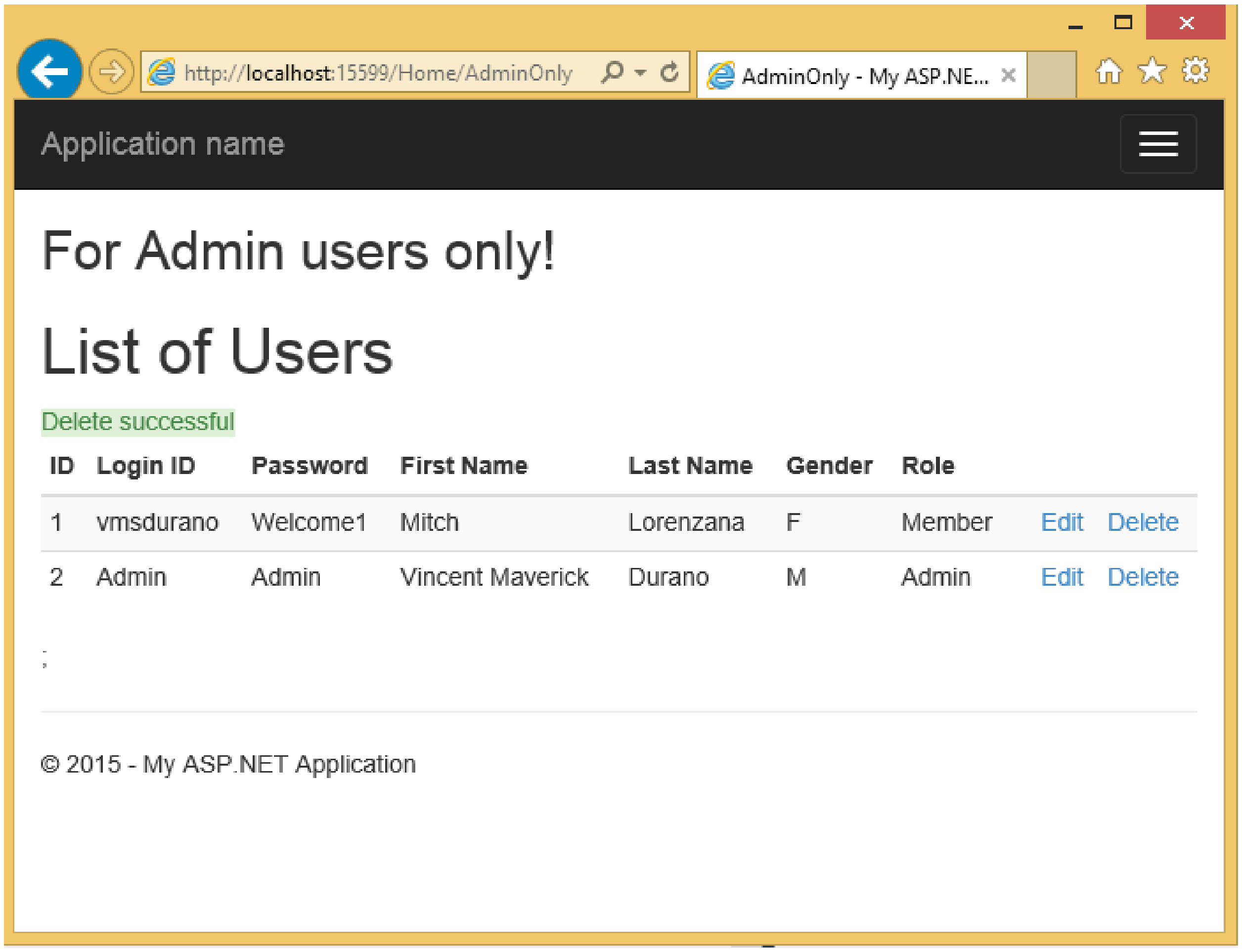
You are about to delete this user with ID 3 . Continue?

OK Cancel

© 2015 - My ASP.NET Application

Figure 35: About to Delete data

After deletion



For Admin users only!

List of Users

Delete successful

ID	Login ID	Password	First Name	Last Name	Gender	Role		
1	vmsdurano	Welcome1	Mitch	Lorenzana	F	Member	Edit	Delete
2	Admin	Admin	Vincent Maverick	Durano	M	Admin	Edit	Delete

;

© 2015 - My ASP.NET Application

Figure 35: After successful deletion

That's it. Now you have an admin page that manages user information.

Creating a User Profile Page

Up to this point you've learned how to create a simple admin page that manages user's data. In this section we will create a page to allow users to modify their profile data.

Adding the GetUserProfile() Method

To begin, open "UserManager" class and add the following method below:

```
public UserProfileView GetUserProfile(int userID) {
     UserProfileView UPV = new UserProfileView();
     using (DemoDBEntities db = new DemoDBEntities()) {
         var user = db.SYSUsers.Find(userID);
         if (user != null) {
             UPV.SYSUserID = user.SYSUserID;
```

```
                    UPV.LoginName = user.LoginName;
                    UPV.Password = user.PasswordEncryptedText;

                    var SUP = db.SYSUserProfiles.Find(userID);
                    if (SUP != null) {
                        UPV.FirstName = SUP.FirstName;
                        UPV.LastName = SUP.LastName;
                        UPV.Gender = SUP.Gender;
                    }

                    var SUR = db.SYSUserRoles.Find(userID);
                    if (SUR != null) {
                        UPV.LOOKUPRoleID = SUR.LOOKUPRoleID;
                        UPV.RoleName = SUR.LOOKUPRole.RoleName;
                        UPV.IsRoleActive = SUR.IsActive;
                    }
                }
            }
            return UPV;
        }
```

The method above gets the specific user information from the database by the passing the SYSUserID as a parameter. You may have noticed that the method returns a **UserProfileView** type which holds some properties from different tables.

Adding the EditProfile() Action Method

Now open “HomeController” class and add the following action methods:

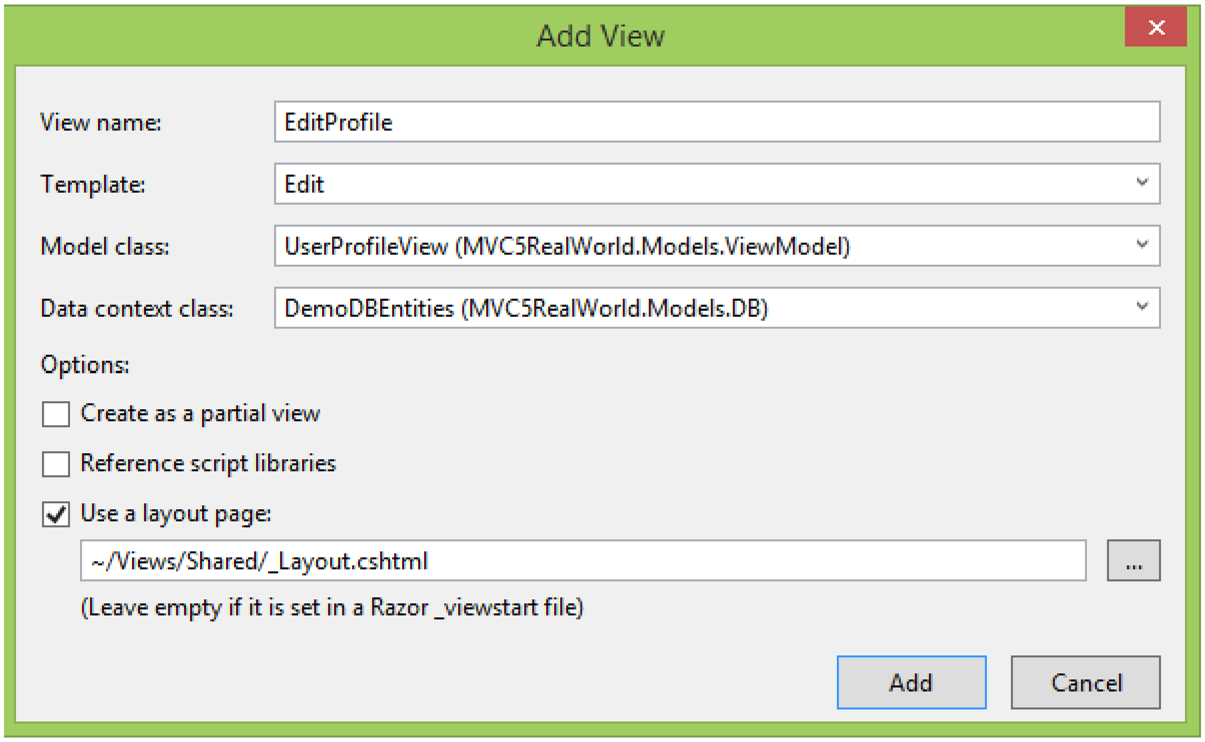
```
        [Authorize]
        public ActionResult EditProfile()
        {
            string loginName = User.Identity.Name;
            UserManager UM = new UserManager();
            UserProfileView UPV = UM.GetUserProfile(UM.GetUserID(loginName));
            return View(UPV);
        }


        [HttpPost]
        [Authorize]
        public ActionResult EditProfile(UserProfileView profile)
        {
            if (ModelState.IsValid)
            {
                UserManager UM = new UserManager();
                UM.UpdateUserAccount(profile);

                ViewBag.Status = "Update Sucessful!";
            }
            return View(profile);
        }
```

The code above is composed of two action methods; the first **EditProfile()** method will be invoked once the page is requested and loaded to the browser. What it does is it gets the user profile data by calling the **GetUserProfile()** method and passing the SYSUserID as the parameter. The second is the overload method which will be invoked during POST request and that is when you hit the Button to save the data. What it does it is first checks for validity of the fields if they are valid and not empty. It then calls the method **UpdateUserAccount()** and passes the **UserProfileView** model from the View to that method. If you still remember from previous section, the UpdateUserAccount() method is where it executes the actual saving of data to your database.

You may also have noticed that both action methods are decorated with the [Authorize] attribute to ensure that both methods should only be accessible by authenticated users.

Adding the View

The next step is to generate the View for the profile page. To do this, right click on the EditProfile() method and select "Add View". In the Add View dialog supply the needed fields as shown in the figure below:



View name: EditProfile

Template: Edit

Model class: UserProfileView (MVC5RealWorld.Models.ViewModel)

Data context class: DemoDBEntities (MVC5RealWorld.Models.DB)

Options:

☐ Create as a partial view

☐ Reference script libraries

☑ Use a layout page:

~/Views/Shared/_Layout.cshtml

(Leave empty if it is set in a Razor _viewstart file)

Figure 36: Add View dialog

Take note of the Model class field value. It should be "UserProfileView". Now click Add to scaffold the UI for you.

Visual Studio will generate all the controls in the View based on the fields you defined from your Model (*UserProfileView*). This means that it will also generate unnecessary fields that we don't want to edit such as the LOOKUPRoleID and IsRoleActive. Aside from that we will also need to provide a drop-down list for displaying the Gender field, so make sure to update the generated HTML markup with the following:

```
@model MVC5RealWorld.Models.ViewModel.UserProfileView

@{
    ViewBag.Title = "EditProfile";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<h2>Edit Your Profile</h2>

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div class="form-horizontal">
        <hr />
        <span class="alert-success">@ViewBag.Status</span>
        @Html.ValidationSummary(true, "", new { @class = "text-danger" })
        @Html.HiddenFor(model => model.SYSUserID)

        <div class="form-group">
            @Html.LabelFor(model => model.RoleName, htmlAttributes: new { @class =
"control-label col-md-2" })
            <div class="col-md-10">
                @Html.DisplayFor(model => model.RoleName)
                @Html.ValidationMessageFor(model => model.RoleName, "", new { @class =
"text-danger" })
            </div>
        </div>

        <div class="form-group">
            @Html.LabelFor(model => model.LoginName, htmlAttributes: new { @class =
"control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.LoginName, new { htmlAttributes = new {
@class = "form-control" } })
                @Html.ValidationMessageFor(model => model.LoginName, "", new { @class =
"text-danger" })
            </div>
        </div>

        <div class="form-group">
            @Html.LabelFor(model => model.Password, htmlAttributes: new { @class =
"control-label col-md-2" })
```

```
                <div class="col-md-10">
                    @Html.EditorFor(model => model.Password, new { htmlAttributes = new {
@class = "form-control" } })
                    @Html.ValidationMessageFor(model => model.Password, "", new { @class =
"text-danger" })
                </div>
            </div>

            <div class="form-group">
                @Html.LabelFor(model => model.FirstName, htmlAttributes: new { @class =
"control-label col-md-2" })
                <div class="col-md-10">
                    @Html.EditorFor(model => model.FirstName, new { htmlAttributes = new {
@class = "form-control" } })
                    @Html.ValidationMessageFor(model => model.FirstName, "", new { @class =
"text-danger" })
                </div>
            </div>

            <div class="form-group">
                @Html.LabelFor(model => model.LastName, htmlAttributes: new { @class =
"control-label col-md-2" })
                <div class="col-md-10">
                    @Html.EditorFor(model => model.LastName, new { htmlAttributes = new {
@class = "form-control" } })
                    @Html.ValidationMessageFor(model => model.LastName, "", new { @class =
"text-danger" })
                </div>
            </div>

            <div class="form-group">
                @Html.LabelFor(model => model.Gender, htmlAttributes: new { @class =
"control-label col-md-2" })
                <div class="col-md-10">
                    @Html.DropDownListFor(model => model.Gender, new List<SelectListItem> {
                        new SelectListItem { Text="Male", Value="M" },
                        new SelectListItem { Text="Female", Value="F" }
                    }, new { @class = "form-control" })
                </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", "Welcome")
</div>
```

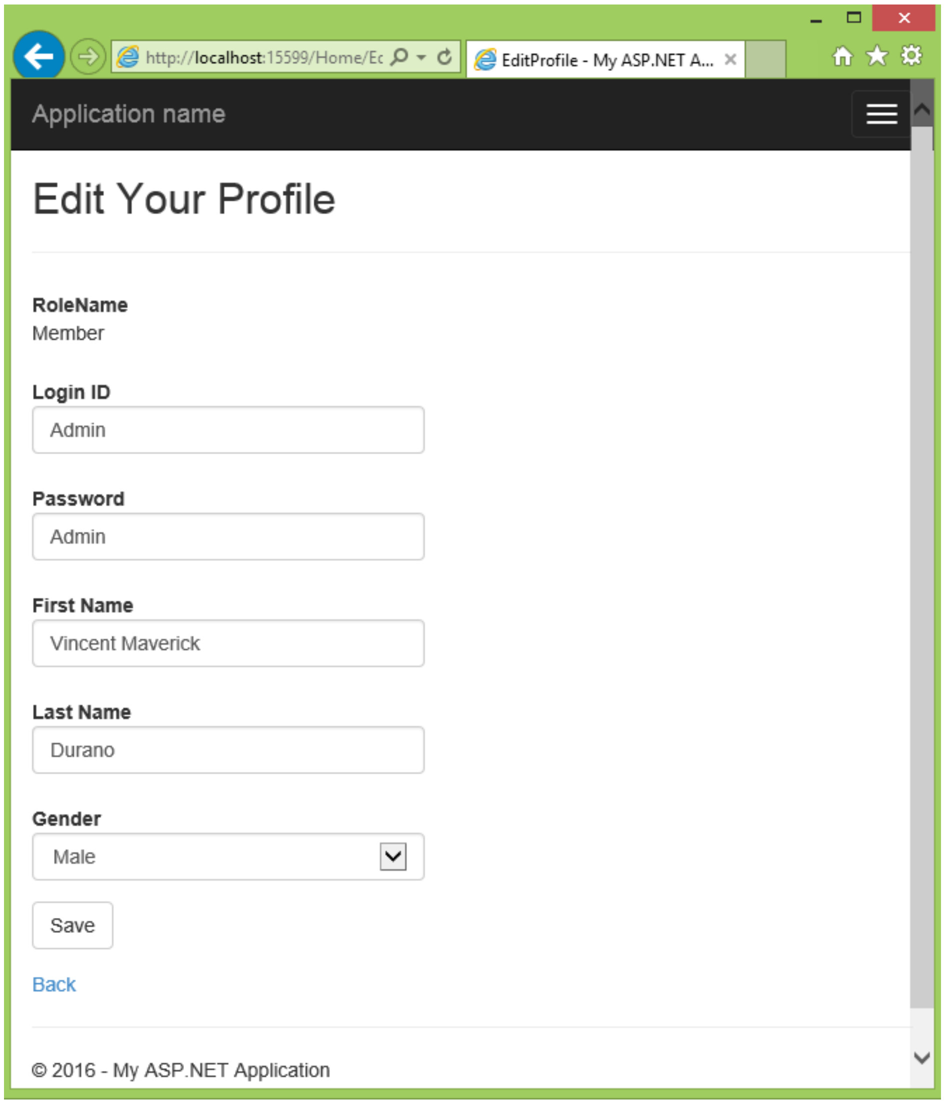
The markup above is another strongly-typed View which renders the **UserProfileView** model. Now add the following markup below within “Welcome.cshtml”.

```
@Html.ActionLink("Edit Profile", "EditProfile", "Home")
```

The markup above is nothing but a link to the Edit Profile page so that when you logged in you can easily navigate to your profile page and start modifying data.

Running the Application

Now try to build your code and then run your application. The output should look similar to the figure below:



Edit Your Profile

RoleName
Member

Login ID
Admin

Password
Admin

First Name
Vincent Maverick

Last Name
Durano

Gender
Male

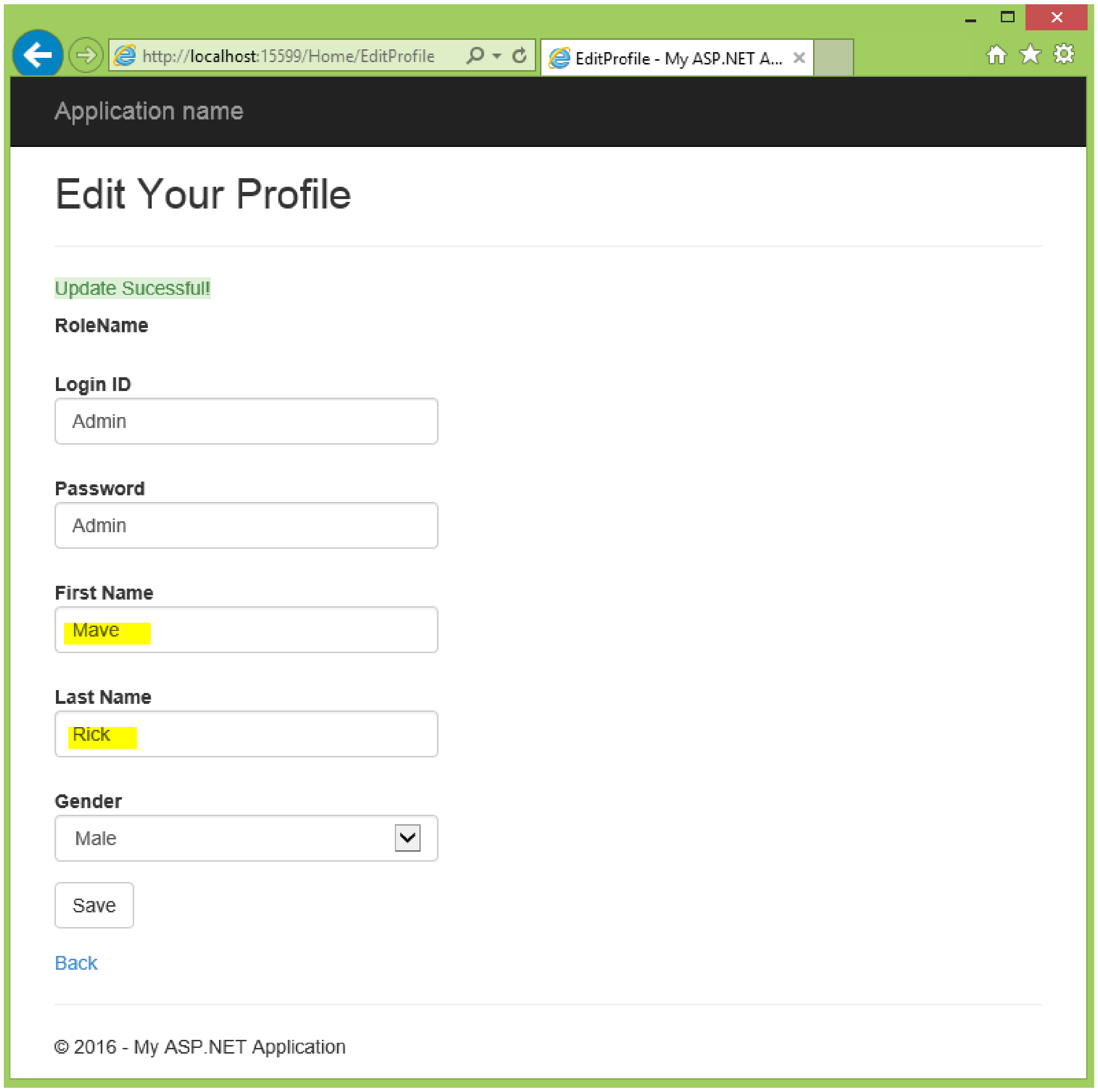
Save

Back

© 2016 - My ASP.NET Application

Figure 37: The Edit Profile page

After modifying the data


http://localhost:15599/Home/EditProfile

EditProfile - My ASP.NET A...

Application name

Edit Your Profile

Update Sucessful!

RoleName

Login ID: Admin

Password: Admin

First Name: Mave

Last Name: Rick

Gender: Male

Save

Back

© 2016 - My ASP.NET Application

Figure 38: After successful update

That simple! Now let's try to move further and do a bit of advance feature.

Implementing a ShoutBox Feature

This section will walk you through on how to implement a simple "shoutbox" feature in your ASP.NET MVC application. I call the feature as "shoutbox" because users within your web site can exchange conversation with each other. You can think of it as a comment board or pretty much similar to a group chat window. Please keep in mind that a "shoutbox" is not a full blown implementation of a chat feature, if you are looking for a chat application then you can refer my other article about Building a Simple Real-Time Chat Application using ASP.NET SignalR

There are many possible ways to implement this feature, but since this article is targeted for beginners to intermediate developers then I decided to use a simple and typical way of performing asynchronous operations using jQuery and AJAX. If you want a simple and clean API that allows you to create real-time web applications where the server needs to continuously push data to clients/browsers then you may want to look at ASP.NET SignalR instead.

Let's get started!

Creating the Message Table

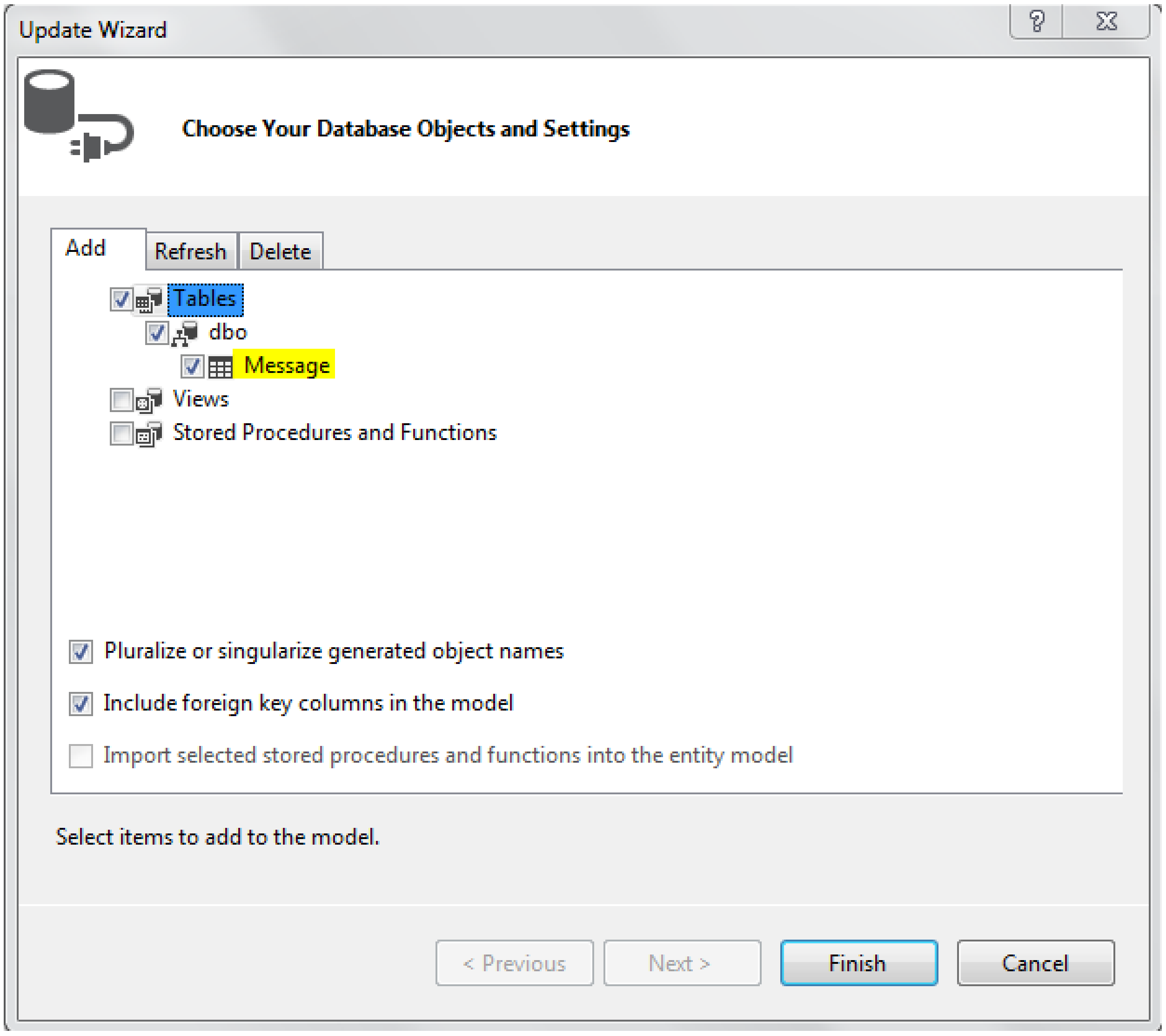
The very first thing we need to do is to create a new table in the database for storing the message of each users. So go ahead and launch SQL Server Management Studio and create a Message table by running the following SQL script below:

```
CREATE TABLE [dbo].[Message](
    [MessageID] [int] IDENTITY(1,1) NOT NULL,
    [SYSUserID] [int] NULL,
    [MessageText] [varchar](max) NULL,
    [DatePosted] [datetime] NULL,
CONSTRAINT [PK_Message] PRIMARY KEY CLUSTERED
(
    [MessageID] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]

GO
```

Updating the Entity Data Model

Now switch back to Visual Studio and then open your EF designer by going to the Models folder > DB > DemoModel.edmx.

Right-click on the design surface and then select "Update Model from Database". Select the Message table to add it to your entity set and click “Finish” as shown in the figure below:



- [✓] Tables
  - [✓] dbo
    - [✓] Message
- [ ] Views
- [ ] Stored Procedures and Functions

[✓] Pluralize or singularize generated object names

[✓] Include foreign key columns in the model

[ ] Import selected stored procedures and functions into the entity model

Select items to add to the model.

Figure 39: Adding the Message table to the Entity

Updating the UserModel

Add the following class under Models folder > ViewModel > UserModel.cs

```
public class UserMessage
{
        public int MessageID { get; set; }
        public int SYSUserID { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string MessageText { get; set; }
        public DateTime? LogDate { get; set; }
}
```

The code above is just a simple class that houses some properties to store data from the database.

Updating the UserManager Class

Add the following code block under Models folder > ObjectManager > UserManager.cs

```
public List<UserMessage> GetAllMessages()
{
    using (DemoDBEntities db = new DemoDBEntities())
    {
        var m = (from q in db.SYSUsers
                 join q2 in db.Messages on q.SYSUserID equals q2.SYSUserID
                 join q3 in db.SYSUserProfiles on q.SYSUserID equals q3.SYSUserID
                 select new UserMessage
                 {
                     MessageID = q2.MessageID,
                     SYSUserID = q.SYSUserID,
                     FirstName = q3.FirstName,
                     LastName = q3.LastName,
                     MessageText = q2.MessageText,
                     LogDate = q2.DatePosted
                 }).OrderBy(o => o.LogDate);

        return m.ToList();
    }
}

public void AddMessage(int userID, string messageText)
{
    using (DemoDBEntities db = new DemoDBEntities())
    {
        Message m = new Message();
        m.MessageText = messageText;
        m.SYSUserID = userID;
        m.DatePosted = DateTime.UtcNow;

        db.Messages.Add(m);
        db.SaveChanges();
    }
}

public int GetUserID(string loginName)
{
    using (DemoDBEntities db = new DemoDBEntities())
    {
        return db.SYSUsers.Where(o => o.LoginName.Equals(loginName))
                          .SingleOrDefault().SYSUserID;
    }
}
```

The **GetAllMessages()** method fetches all messages that was stored from the database and assigning each field values to the corresponding properties of the **UserMessage** model. **AddMessage()** method simply add new sets of data to the database. Finally, **GetUserID()** method gets the user id of the current logged user by passing the login name as the parameter.

Updating the HomeController Class

Add the following action methods below under Controllers folder > HomeController.cs

```
[Authorize]
public ActionResult Index()
{
    UserManager UM = new UserManager();
    ViewBag.UserID = UM.GetUserID(User.Identity.Name);
    return View();
}

[Authorize]
public ActionResult ShoutBoxPartial()
{
    return PartialView();
}

[Authorize]
public ActionResult SendMessage(int userID, string message)
{
    UserManager UM = new UserManager();
    UM.AddMessage(userID, message);
    return Json(new { success = true });
}

[Authorize]
public ActionResult GetMessages()
{
    UserManager UM = new UserManager();
    return Json(UM.GetAllMessages(), JsonRequestBehavior.AllowGet);
}
```

In Index action method we call the **GetUserID()** method by passing the login name as the parameter to get the user ID of the current logged user. We then store the value in ViewBag so we can reference it in our View later on. The **SendMessage()** action method simply calls the **AddMessage()** method to insert new records to the database. The **GetMessages()** method fetches all user messages from the database.

Creating the ShoutBoxPartial Partial View

Create a new partial view under Views folder > Home and name it as "ShoutBoxPartial.cshtml". And then add the following markup below:

```
<style type="text/css">
    #divShoutBox {
        position: relative;
        width: 400px;
        height: 300px;
        overflow: auto;
    }

    #txtMessageText {
        width: 400px;
        height: 100px;
    }
</style>

<div id="divShoutBox">
    <div id="divUserMessage"></div>
</div>

<br />
<textarea id="txtMessageText"></textarea>
<br />
<input type="button" id="btnPost" value="Post" />

<script>

    var _isScrolling = false;
    var _lastScrollPos = 0;
    var _counter = 0;

    $(function () {

        GetMessages();
        setInterval(Fetch, 5000);

        $("#divShoutBox").on("scroll", function () {
            _isScrolling = true;
            _lastScrollPos = this.scrollHeight;
        });
        $("#btnPost").on("click", function () {
            var msg = $("#txtMessageText");
            var user = $("#hidUserID");

            if (msg.val().length > 0) {
                $.ajax({
                    type: "POST",
                    url: '@(Url.Action("SendMessage","Home"))',
                    data: { userID: user.val(), message: msg.val() },
                    success: function (d) { msg.val(""); GetMessages(); },
                    error: function (err) { }
                });
            }
        });

    });


    function Fetch() {
        if (!_isScrolling) {
            GetMessages();
            $("#divShoutBox").scrollTop(_lastScrollPos);
        };
        _isScrolling = false;
    }

    function GetMessages() {
        $.ajax({
            type: "POST",
            url: '@(Url.Action("GetMessages","Home"))',
               data: {},
               success: function (d) {
                   $("#divUserMessage").empty();
                   $.each(d, function (index, i) {
                       GenerateHTML(i.FirstName, i.LastName, i.MessageText,
FormatDateString(i.LogDate));
                   });
               },
               error: function (err) { }
        });
    }

    function GenerateHTML(fName, lName, msgText, logDate) {
        var divMsg = $("#divUserMessage");
        divMsg.append("Posted by: " + fName + " " + lName + "<br/>");
        divMsg.append("Posted on: " + logDate + "<br/>");
        divMsg.append(msgText);
        divMsg.append("<hr/>");
    }

    function FormatDateString(logDate) {
        var d = new Date(parseInt(logDate.substr(6)));
        var year = d.getFullYear();
        var month = d.getMonth() + 1;
        var day = d.getDate();
        var hour = d.getHours();
        var minutes = d.getMinutes();
        var sec = d.getSeconds();

        return month + "/" + day + "/" + year + " " + hour + ":" + minutes + ":" + sec;
    }

</script>
```

The HTML markup above is fairly simple and nothing really fancy about it. It just contains some div elements, textarea and a button. I also applied few CSS style for the div and textbox elements. Keep in mind that the look and feel doesn't really matter for this tutorial as we are focusing mainly on the functionality itself.

Down to the JavaScript Functions

There are four (4) main JavaScript functions from the markup above. The first one is the **GetMessages()** function. This function uses jQuery AJAX to issue an asynchronous post request to the server to get all available messages from the database. If the AJAX call is successful then we iterate to each items from the JSON response and call the **GenerateHTML()** function to build up the UI with the result set. The GenerateHTML() function uses jQuery function to build up the HTML and append the values to the existing div element. The **FormatDateString()** funtion is a method that converts JSON date format to JavaScript date format and return our own date format to the UI for the users to see. The **Fetch()** function calls the GetMessages() function and handles the scroll position of the div. This means that we auto scroll to the bottom part of the div element once there's a new message coming.

The **$(function (){})** is the short-hand syntax for jQuery's document ready function which fires once all DOM elements are loaded in the browser. This is where we register the onscroll event of div and the “onclick” event of button using jQuery. In “onscroll” event we just set some values to some global variables for future use. In onclick event we just issued an AJAX request to the server to add new data to the database. When the DOM is ready we also call the GetMessages() function to display all messages on initial load of the browser. You may also noticed there that I have used the **setInterval()** function to automatically pull data from the server after every five (5) seconds. So if other users from your web site send a message then it will automatically be available for other users after 5 seconds cycle. This is the traditional way of using AJAX to pull data from the server for a given period of time.

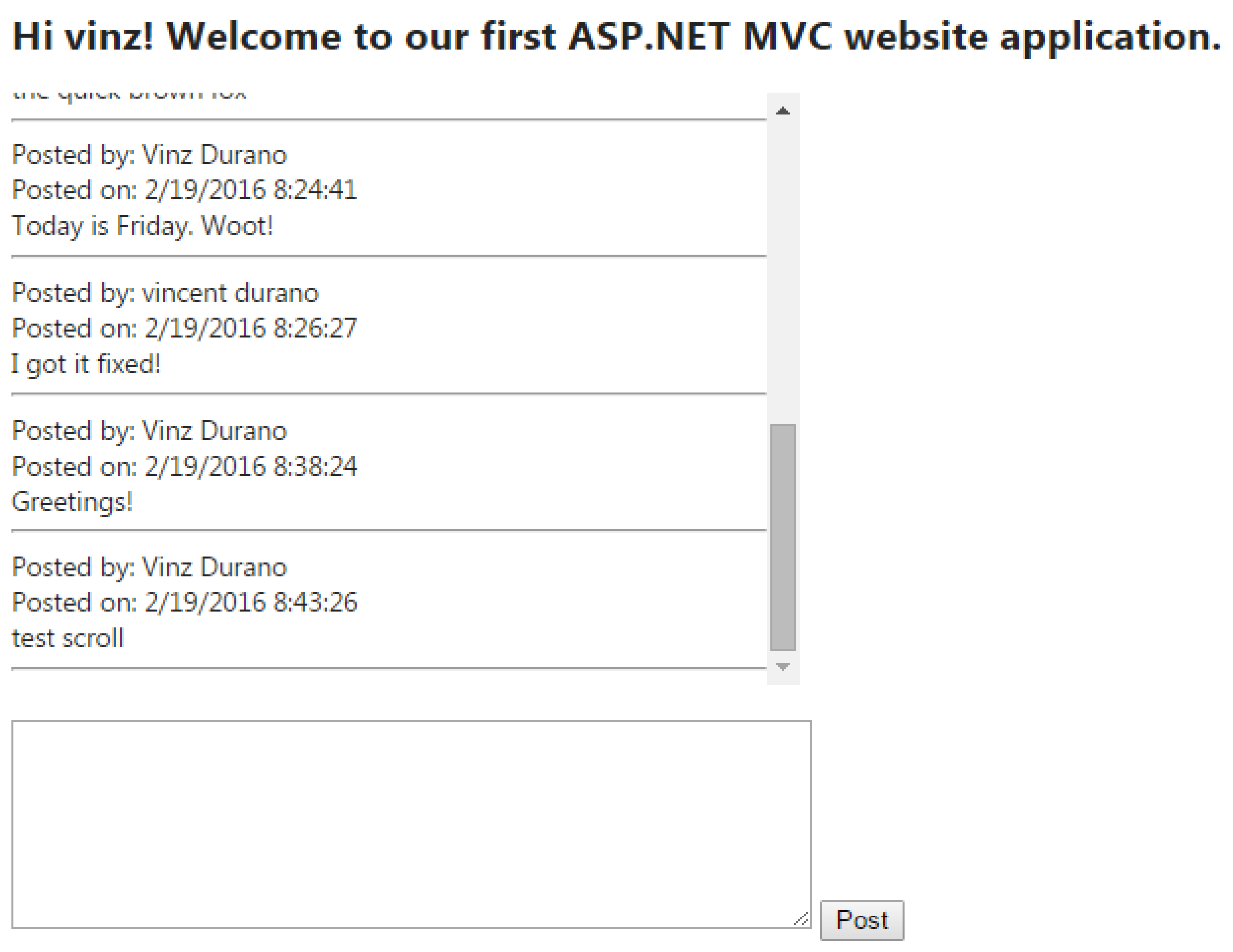
Wrapping Up

Add the following markup below in Index.cshtml file:

```
<input type="hidden" id="hidUserID" value="@ViewBag.UserID" />
@Html.Action("ShoutBoxPartial", "Home")
```

Running the Application

Running the code should look something like this:



Hi vinz! Welcome to our first ASP.NET MVC website application.

Posted by: Vinz Durano
Posted on: 2/19/2016 8:24:41
Today is Friday. Woot!

Posted by: vincent durano
Posted on: 2/19/2016 8:26:27
I got it fixed!

Posted by: Vinz Durano
Posted on: 2/19/2016 8:38:24
Greetings!

Posted by: Vinz Durano
Posted on: 2/19/2016 8:43:26
test scroll

Figure 40: The ShoutBox

Deploying Your ASP.NET MVC 5 App to IIS8

Web Developers today build and test ASP.NET sites and applications using one of the two web-servers:

- The IIS Express that comes built-into Visual Studio
- The IIS Web Server that comes built-into Windows

If you have noticed the URL displayed in the browser shows *http://localhost:15599*. The integer value in the URL represents the port number used in IIS Express. IIS Express is the default web server for web application projects in Visual Studio 2012 and higher versions. The default internal web server in Visual Studio typically used to build and run your app during development for you to test and debug codes. You can see the IIS Express configuration by right clicking on the project and then by clicking on the "Web" tab. The figure below shows how it looks like:

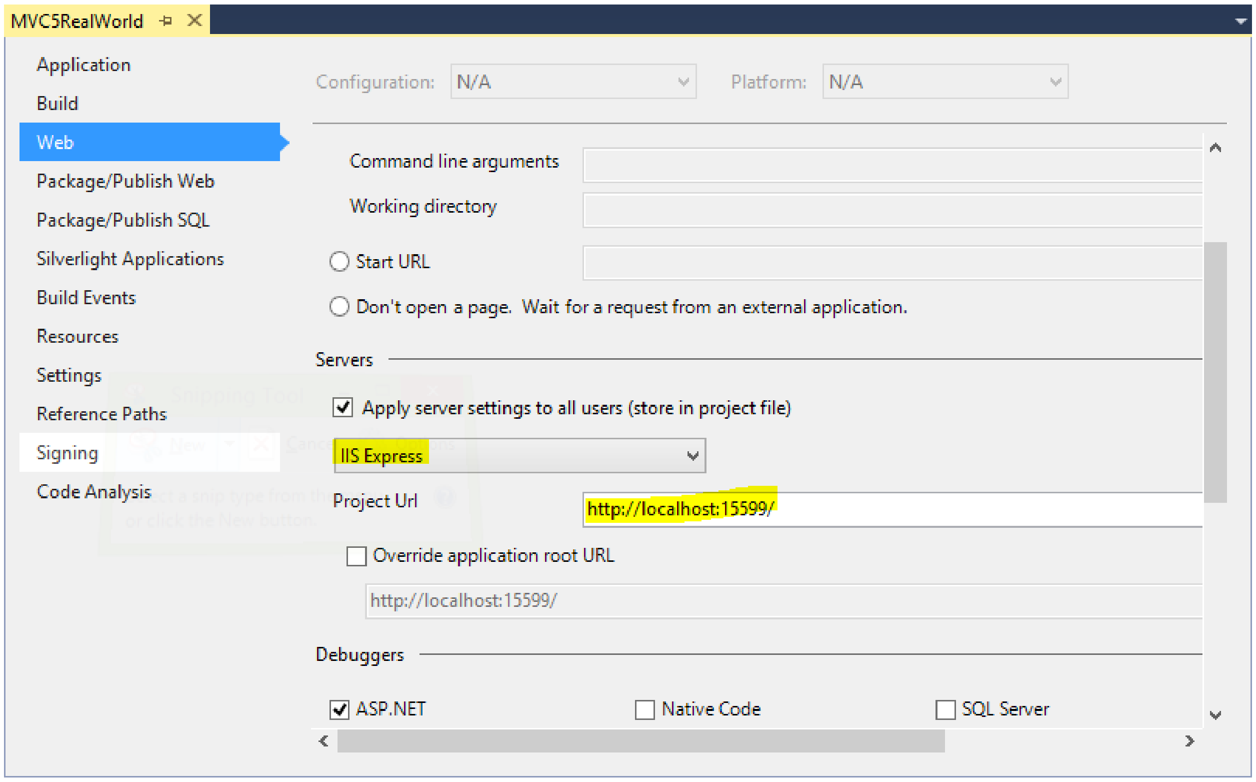


Figure 41: Web Settings

You will use IIS Web Server when you want to test your web application using the server environment that is closest to what the live site will run under, and it is practical for you to install and work with IIS on your development computer. This section will walk you through on how to host your ASP.NET MVC 5 web application in your local IIS Web Server.

Before deploying your app, verify that you have IIS installed in your machine. If you already have IIS installed then you can skip this step otherwise if you don't then just follow through.

In this particular project I used Windows 8.1 as my Windows Operating System. If you are using a different version of Windows OS then I'm sure there are plenty of resources from the web that demonstrate the installation of IIS in your Windows machine.

Installing IIS8 on Windows 8.1

Open Control Panel and click on "Programs" as shown in the figure below:

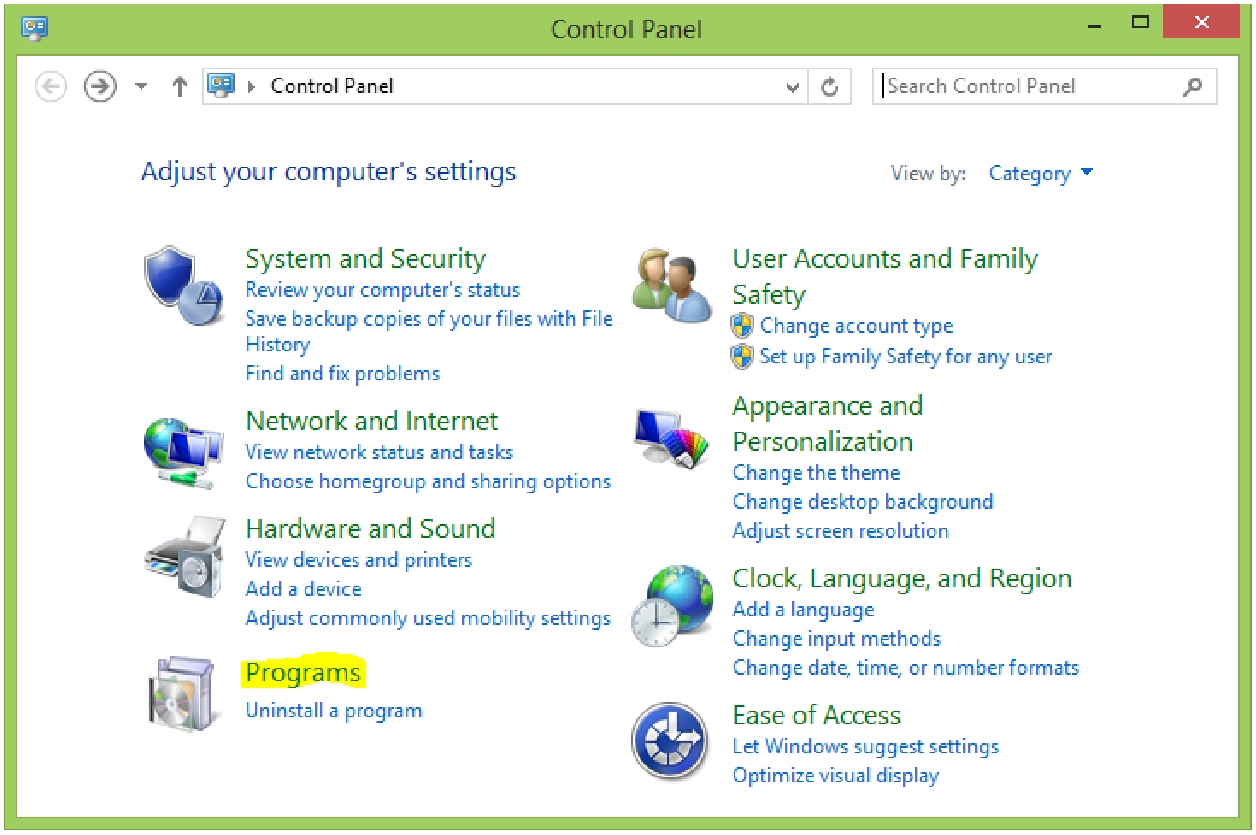
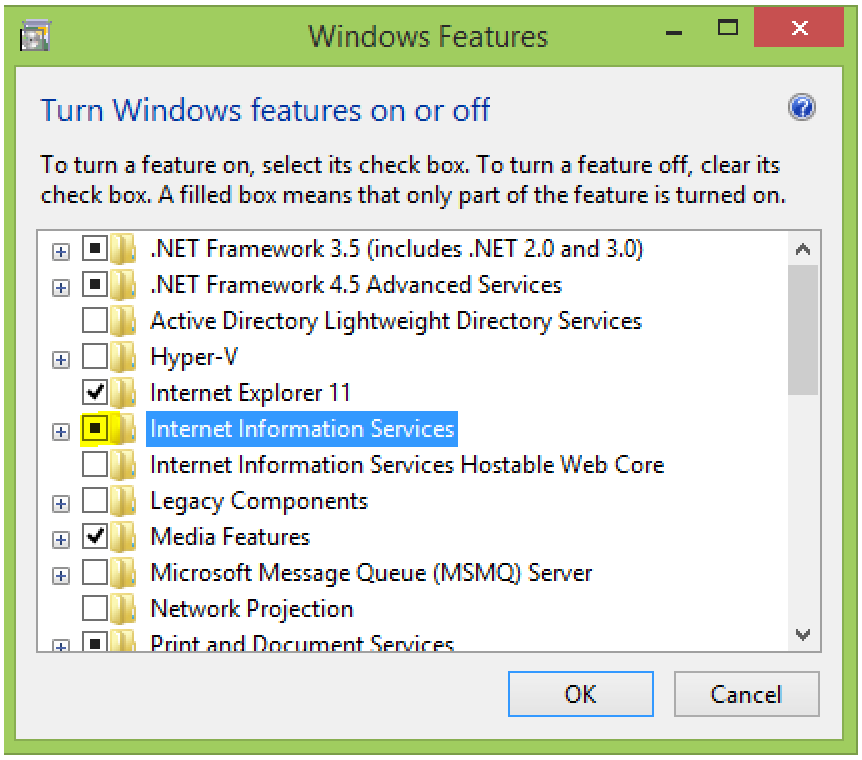

Control Panel
Control Panel
Search Control Panel
Adjust your computer's settings
View by: Category
System and Security
Review your computer's status
Save backup copies of your files with File History
Find and fix problems
Network and Internet
View network status and tasks
Choose homegroup and sharing options
Hardware and Sound
View devices and printers
Add a device
Adjust commonly used mobility settings
Programs
Uninstall a program
User Accounts and Family Safety
Change account type
Set up Family Safety for any user
Appearance and Personalization
Change the theme
Change desktop background
Adjust screen resolution
Clock, Language, and Region
Add a language
Change input methods
Change date, time, or number formats
Ease of Access
Let Windows suggest settings
Optimize visual display

Figure 42: Windows Control Panel

Then click on "Turn Windows features on or off" from the Programs and Features dialog and select "Internet Information Services" from the list as shown in the figure below:



Windows Features

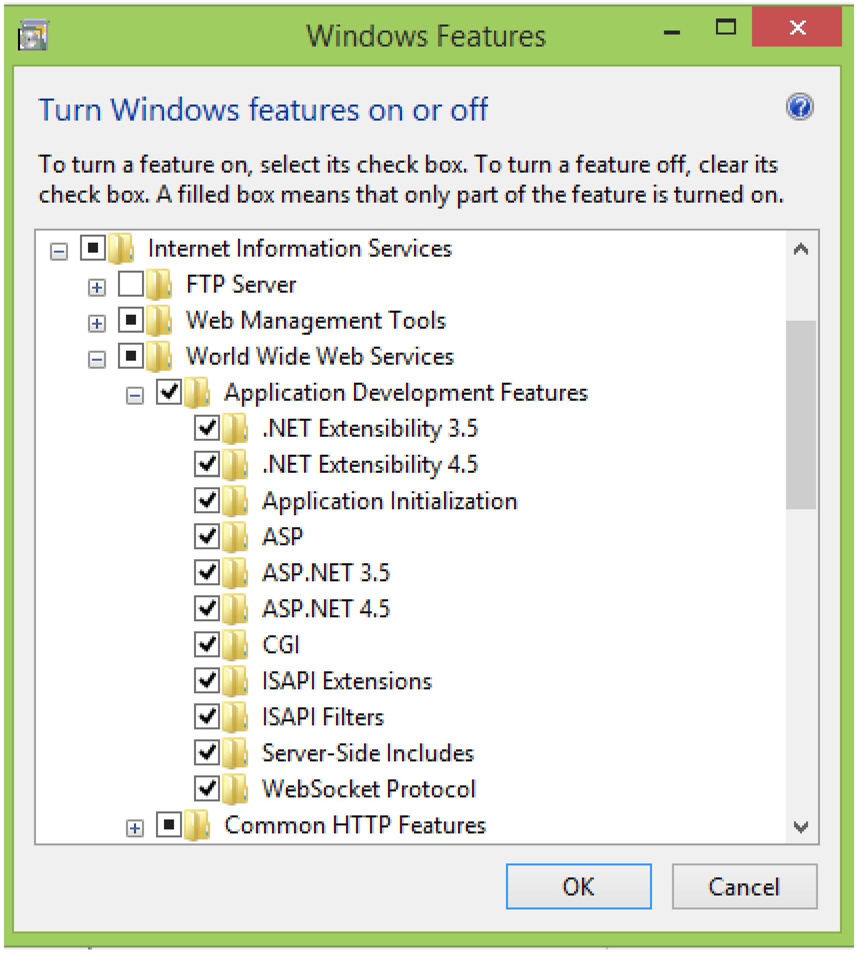
Turn Windows features on or off

To turn a feature on, select its check box. To turn a feature off, clear its check box. A filled box means that only part of the feature is turned on.

- [■] .NET Framework 3.5 (includes .NET 2.0 and 3.0)
- [■] .NET Framework 4.5 Advanced Services
- [ ] Active Directory Lightweight Directory Services
- [ ] Hyper-V
- [✓] Internet Explorer 11
- [■] Internet Information Services
- [ ] Internet Information Services Hostable Web Core
- [ ] Legacy Components
- [✓] Media Features
- [ ] Microsoft Message Queue (MSMQ) Server
- [ ] Network Projection
- [■] Print and Document Services

Figure 43: Windows Features dialog

Expand IIS and check/enable all components under “World Wide Web Services” > “Application Development Features” as shown in the figure below.


Windows Features

Turn Windows features on or off

To turn a feature on, select its check box. To turn a feature off, clear its check box. A filled box means that only part of the feature is turned on.

- ■ Internet Information Services
  - ☐ FTP Server
  - ■ Web Management Tools
  - ■ World Wide Web Services
    - ✓ Application Development Features
      - ✓ .NET Extensibility 3.5
      - ✓ .NET Extensibility 4.5
      - ✓ Application Initialization
      - ✓ ASP
      - ✓ ASP.NET 3.5
      - ✓ ASP.NET 4.5
      - ✓ CGI
      - ✓ ISAPI Extensions
      - ✓ ISAPI Filters
      - ✓ Server-Side Includes
      - ✓ WebSocket Protocol
    - ■ Common HTTP Features

Figure 44: Windows Features dialog

Click “OK” to let Windows install the need files. Once installed you may now close the dialog. Now open an internet browser and type-in “localhost” to verify that IIS was indeed installed. It should bring up the following page below:

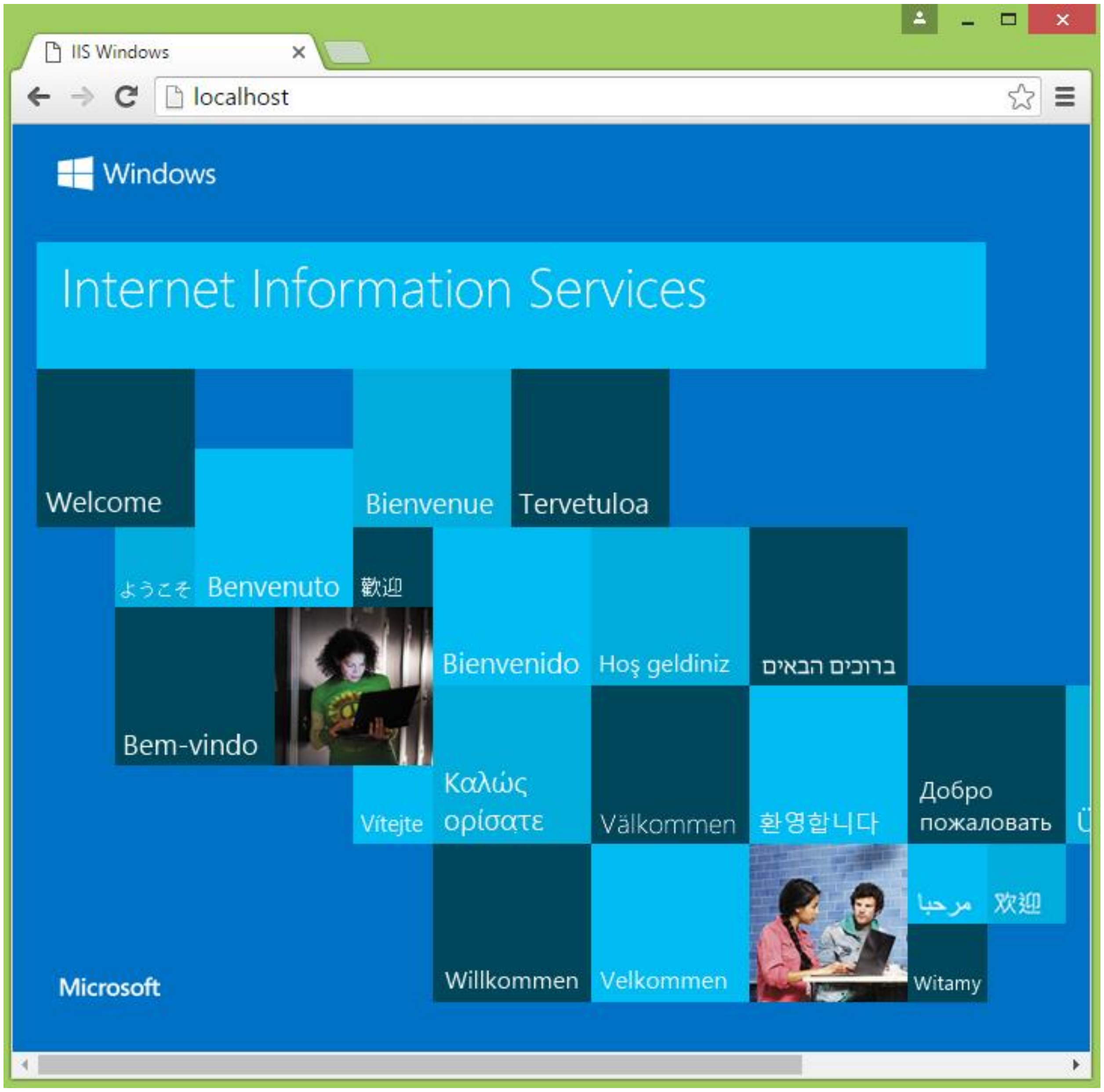


Figure 45: IIS page

Publishing from Visual Studio

If you've seen that in the browser then we are ready to deploy and host our app in IIS. Now switch back to Visual Studio 2015 and then right click on your project, in this case "MVC5RealWorld" and then select "Publish". It should bring up the following dialog below:

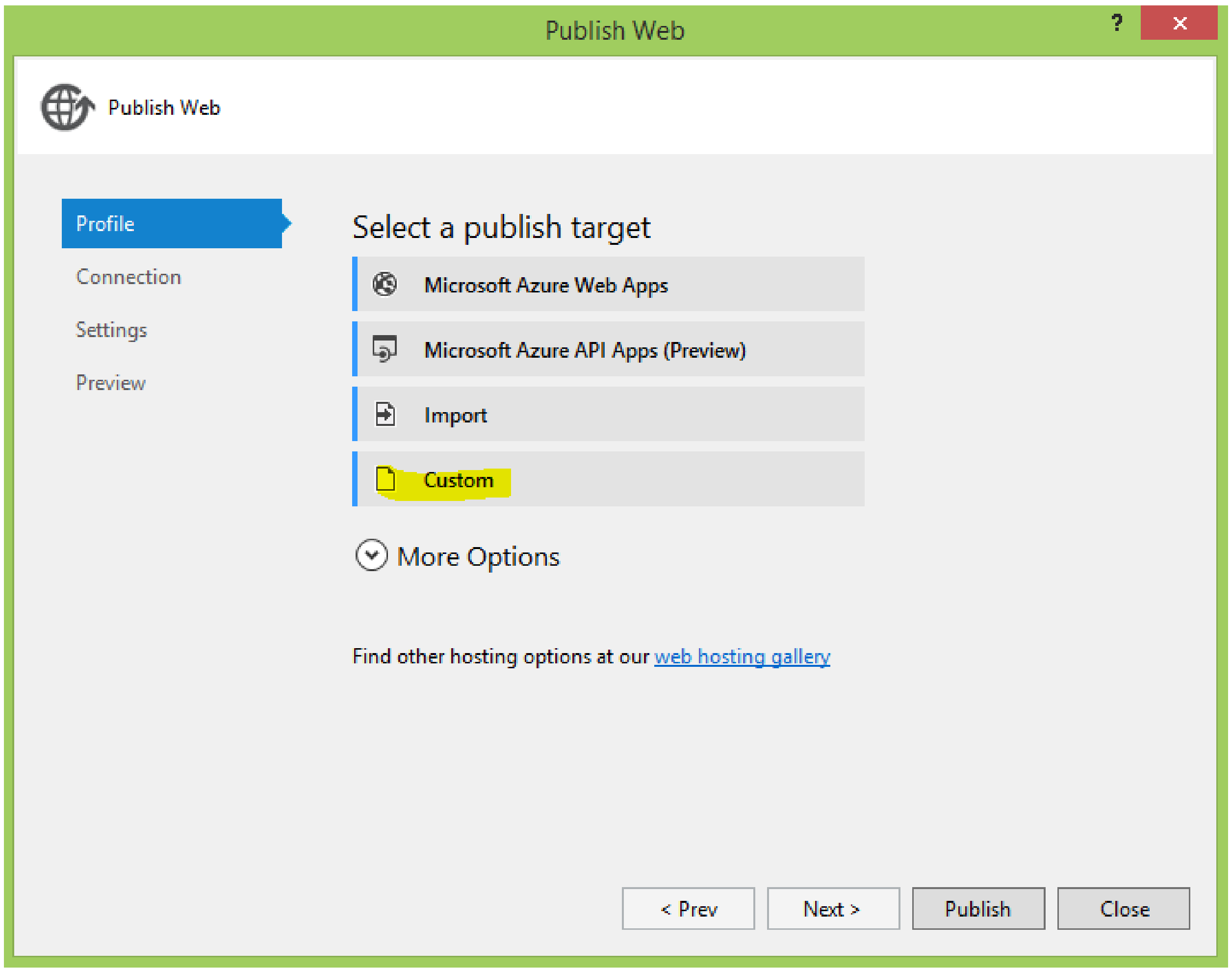
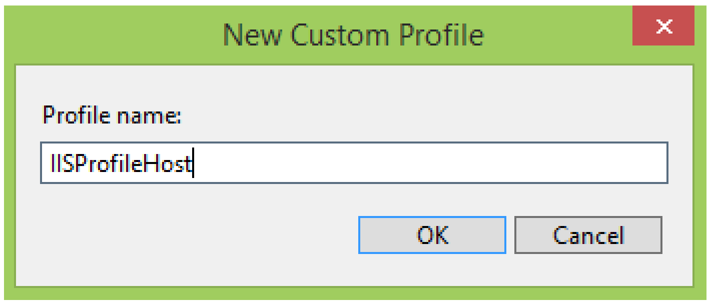


Figure 46: Publish Web dialog

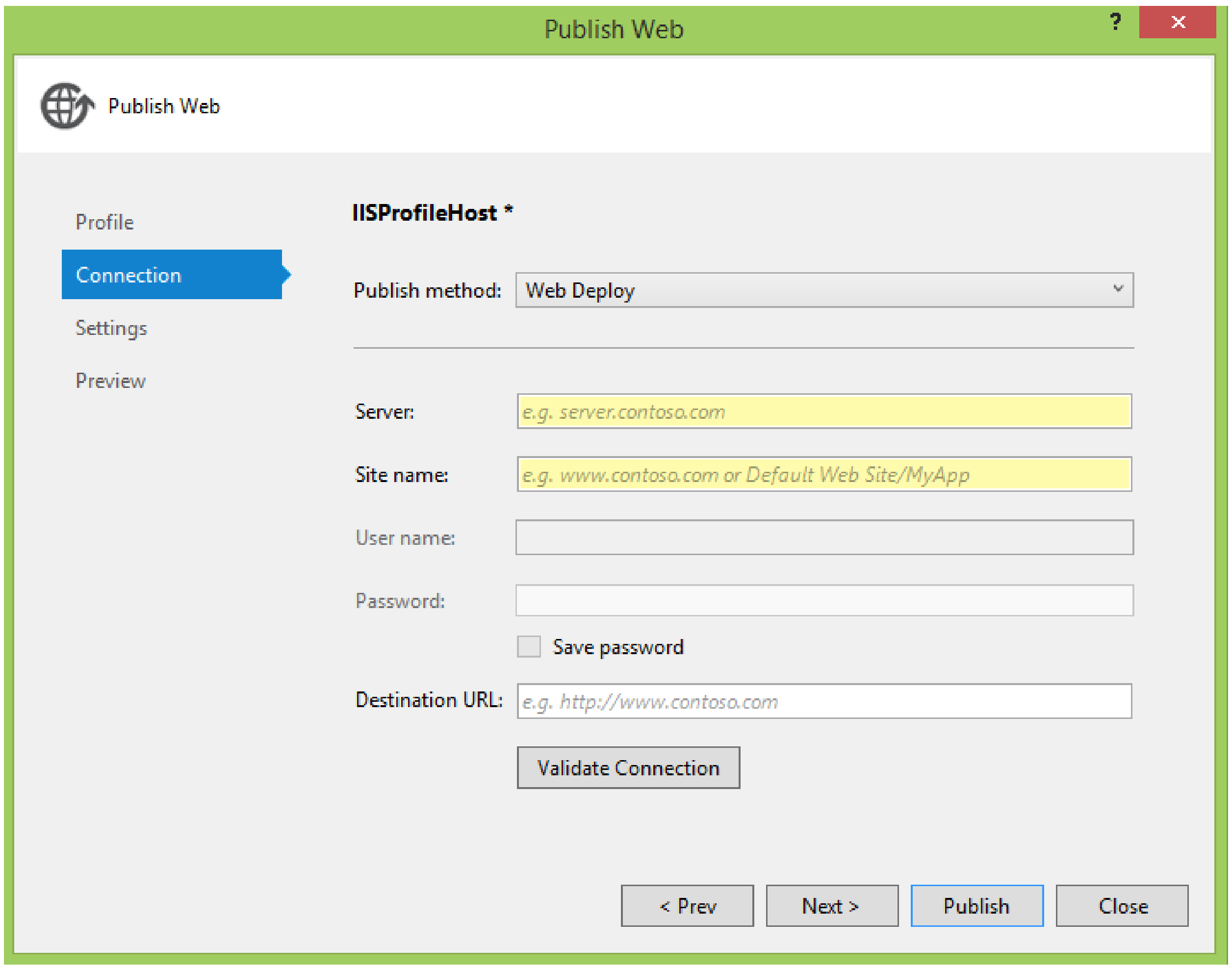
Select "Custom" from the options and enter a profile name for your host as shown in the figure below:



Profile name: IISProfileHost

Figure 47: New Custom Profile dialog

Click “OK” to bring up the following dialog below:


Publish Web

Publish Web

IISProfileHost *

Publish method: Web Deploy

Server: e.g. server.contoso.com

Site name: e.g. www.contoso.com or Default Web Site/MyApp

User name:

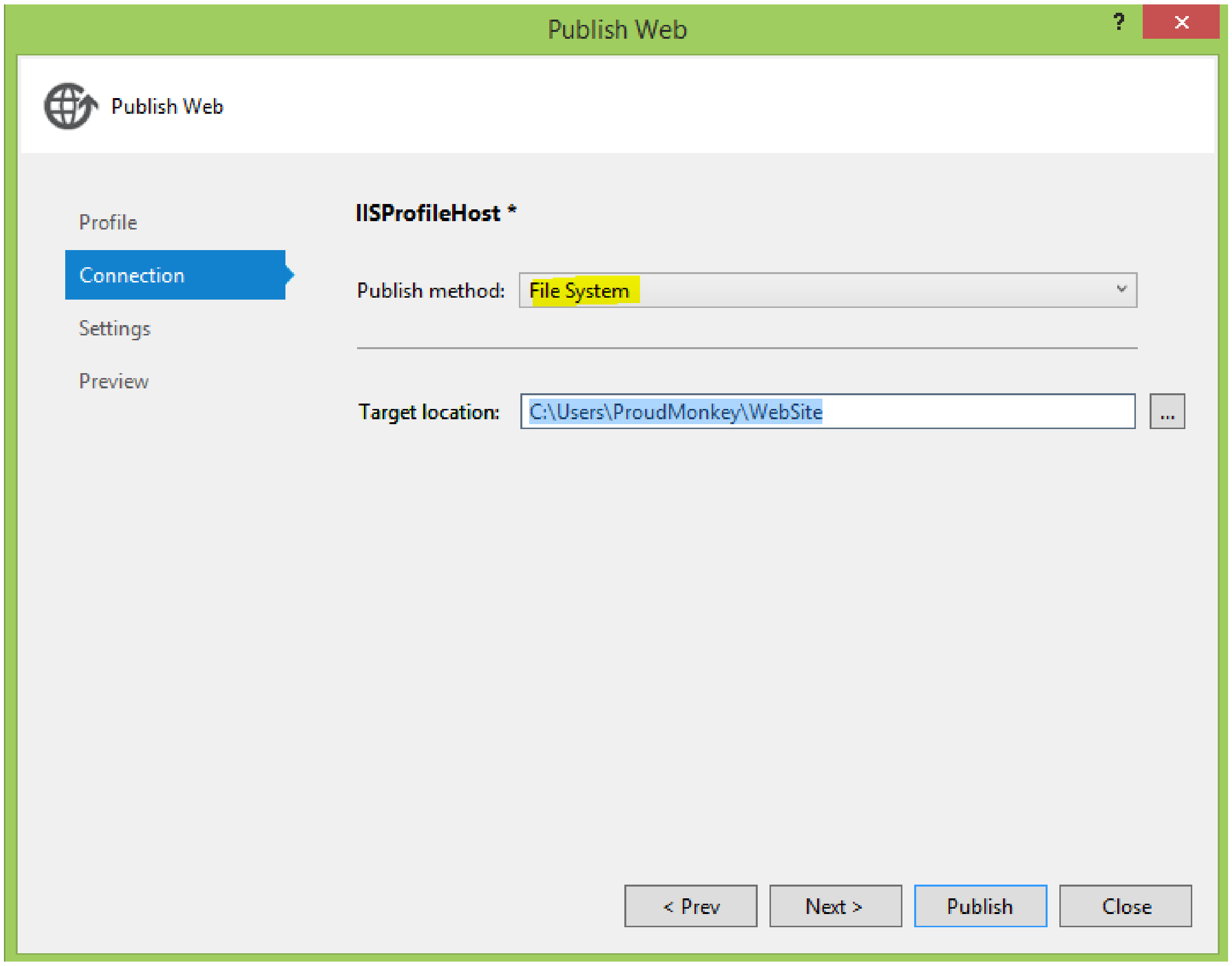
Password:

Save password

Destination URL: e.g. http://www.contoso.com

Figure 48: Publish Method

Now select “File System” as publish method and enter your preferred deployment location. In my case I target it at this location “C:\Users\ProudMonkey\WebSite” in my local drive. See the figure below for your reference:



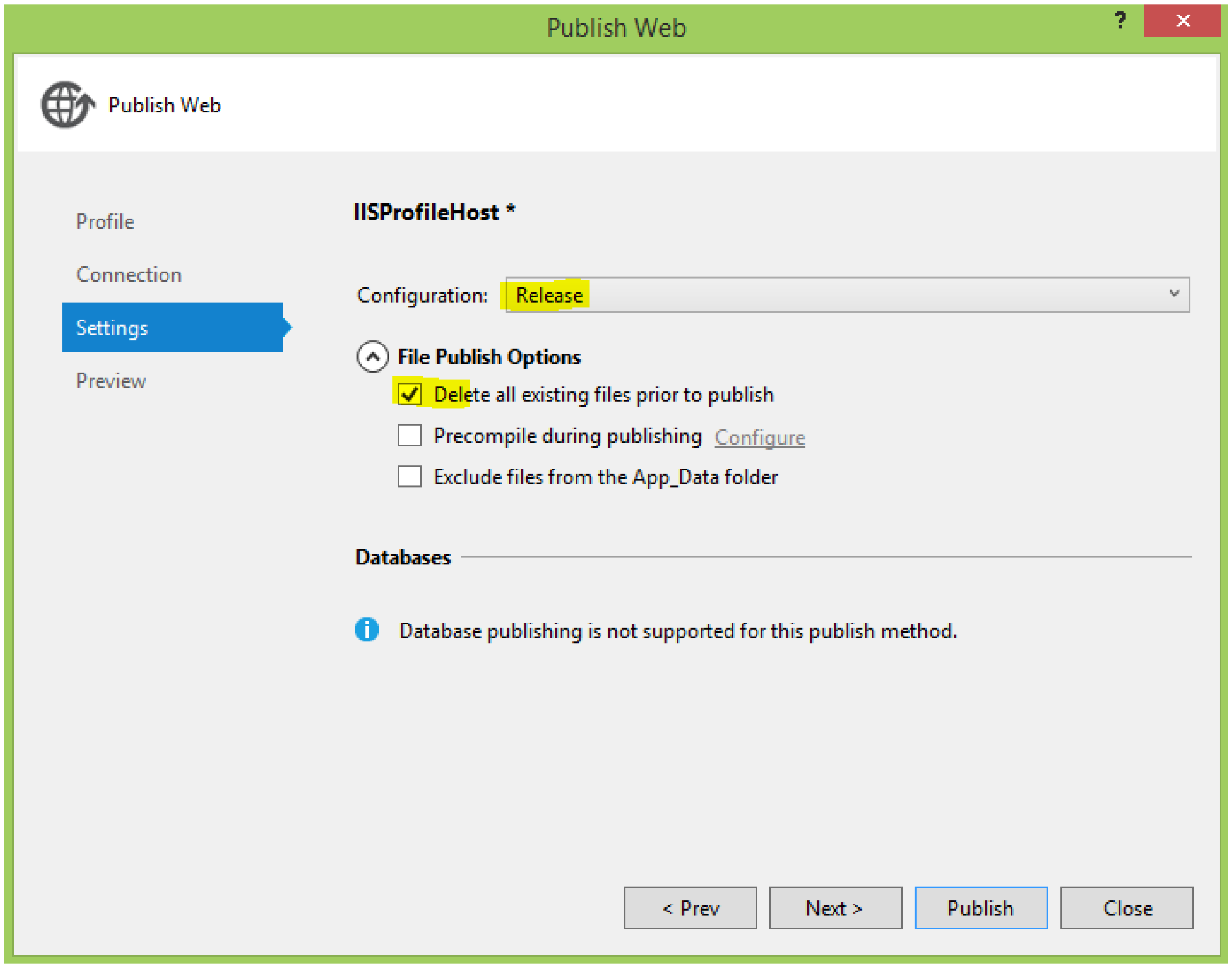
IISProfileHost *

Publish method: File System

Target location: C:\Users\ProudMonkey\WebSite

Figure 49: Publish Method

Click "Next" and then select "Release" as the configuration and check the "Delete all existing files prior to publish" option to make sure that Visual Studio will generate fresh files once you re-publish your app.


Publish Web

Publish Web

IISProfileHost *

Configuration: Release

File Publish Options

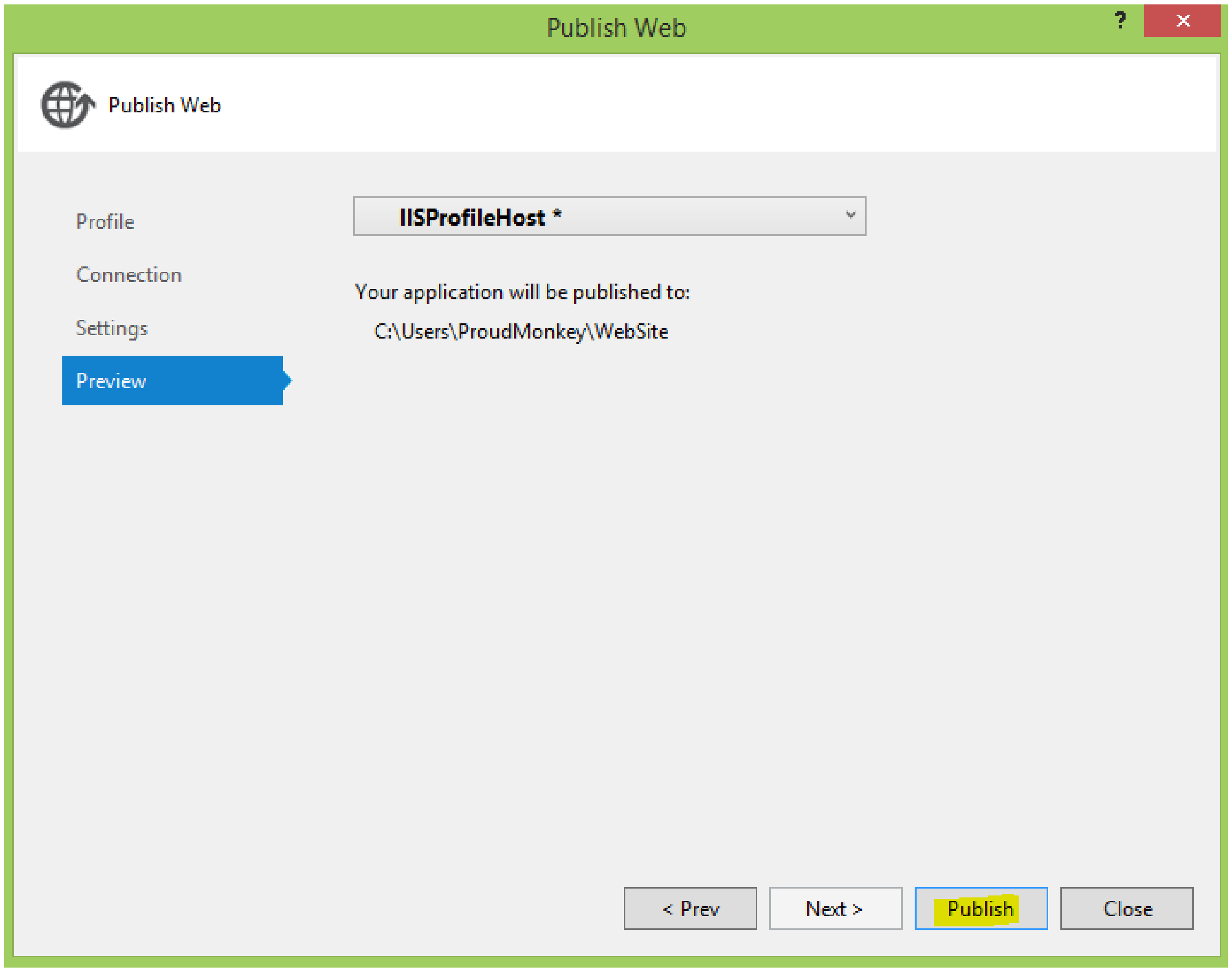
- [✓] Delete all existing files prior to publish
- [ ] Precompile during publishing Configure
- [ ] Exclude files from the App_Data folder

Databases

Database publishing is not supported for this publish method.

Figure 50: Publishing

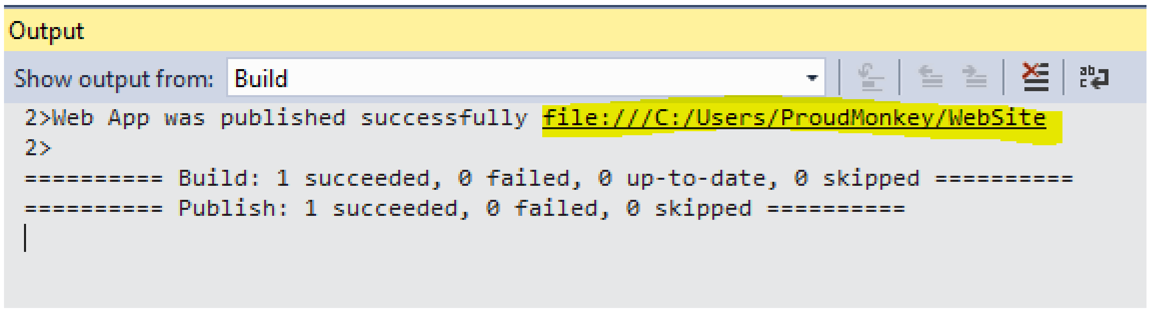
Click “Next” and it should take you to the next step where it will inform you that your web app will be deployed to the location you supplied from the previous step. If you are sure about it then just click “Publish” as shown in the figure below.



Your application will be published to:
C:\Users\ProudMonkey\WebSite

Figure 51: Publishing

Visual Studio will compile and publish your app to the desired location. When it's succeeded then it show something like this in the output window.



```
2>Web App was published successfully file:///C:/Users/ProudMonkey/WebSite
2>
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
========== Publish: 1 succeeded, 0 failed, 0 skipped ==========
```

Figure 52: Publish succeeded

Now browse the location to where you point your files to be published. In this example the file was published in “C:\Users\ProudMonkey\WebSite” (it could a different location in your case). To verify that the location is accessible in IIS then make sure that the folder containing the published files is not “Read-Only”. You can verify it by right-clicking on the folder and see the read-only option. Make sure it is unchecked.

We’re Not Done Yet!

Converting Your App to Web Application

Yup, we’re not done yet. The last step is to configure IIS to convert your app as a web application. To do this open IIS Manager or simply type “**inetmgr**” in Windows 8 search box. It should bring up the following window below:

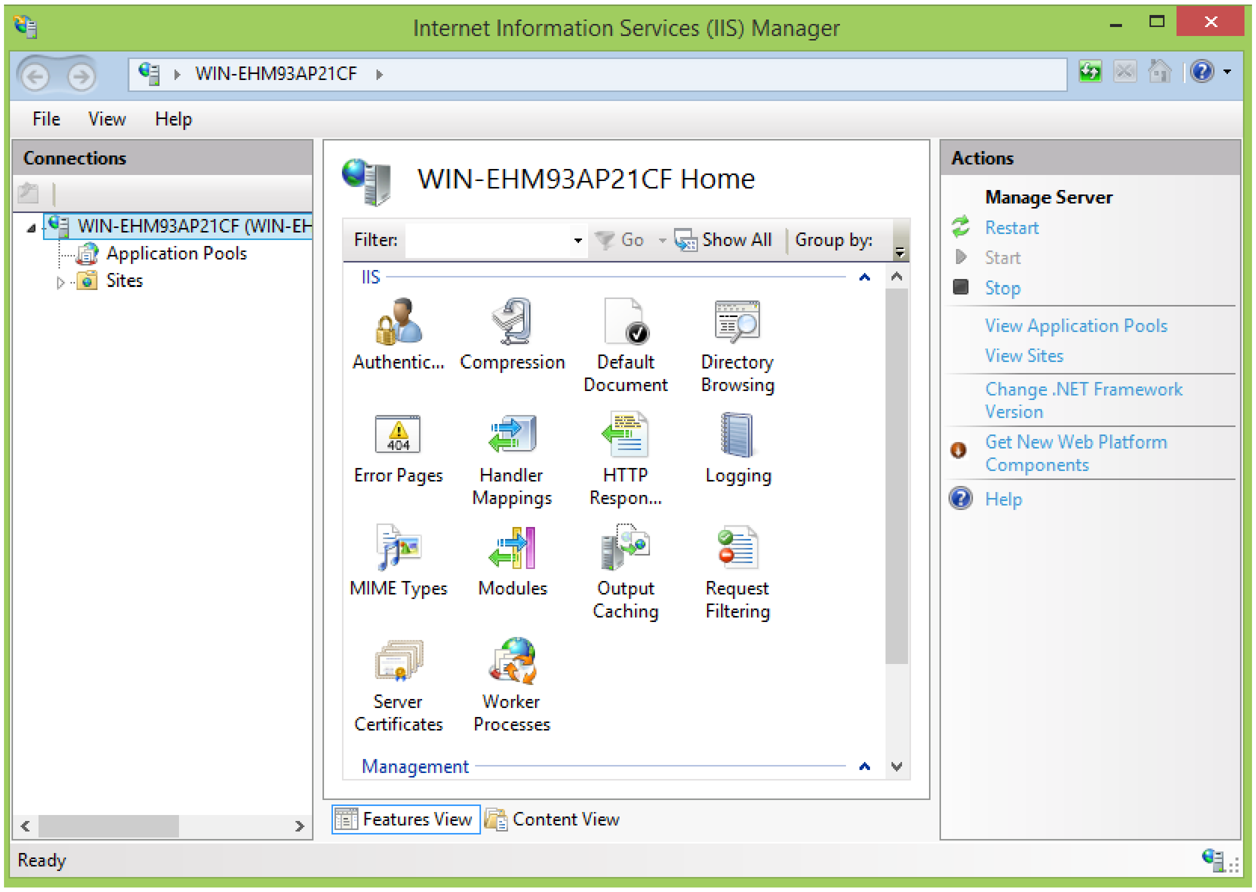
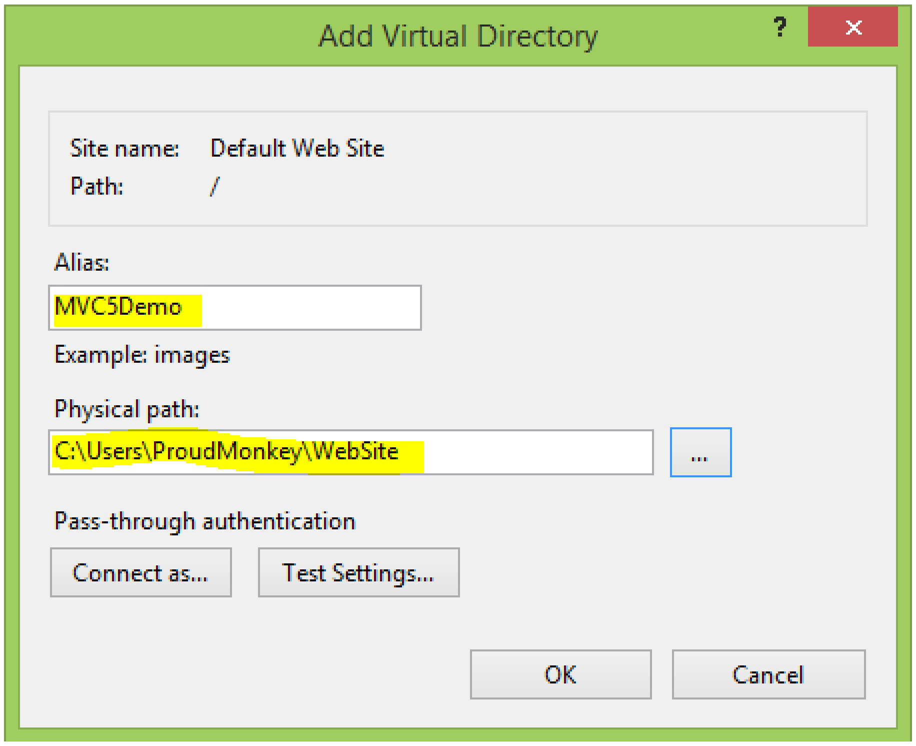


Figure 53: IIS Manager

Expand the “Sites” folder and then right click on the “Default Web Site” and select “Add Virtual Directory”. You should be able to see the following dialog below.



Site name: Default Web Site
Path: /

Alias: MVC5Demo

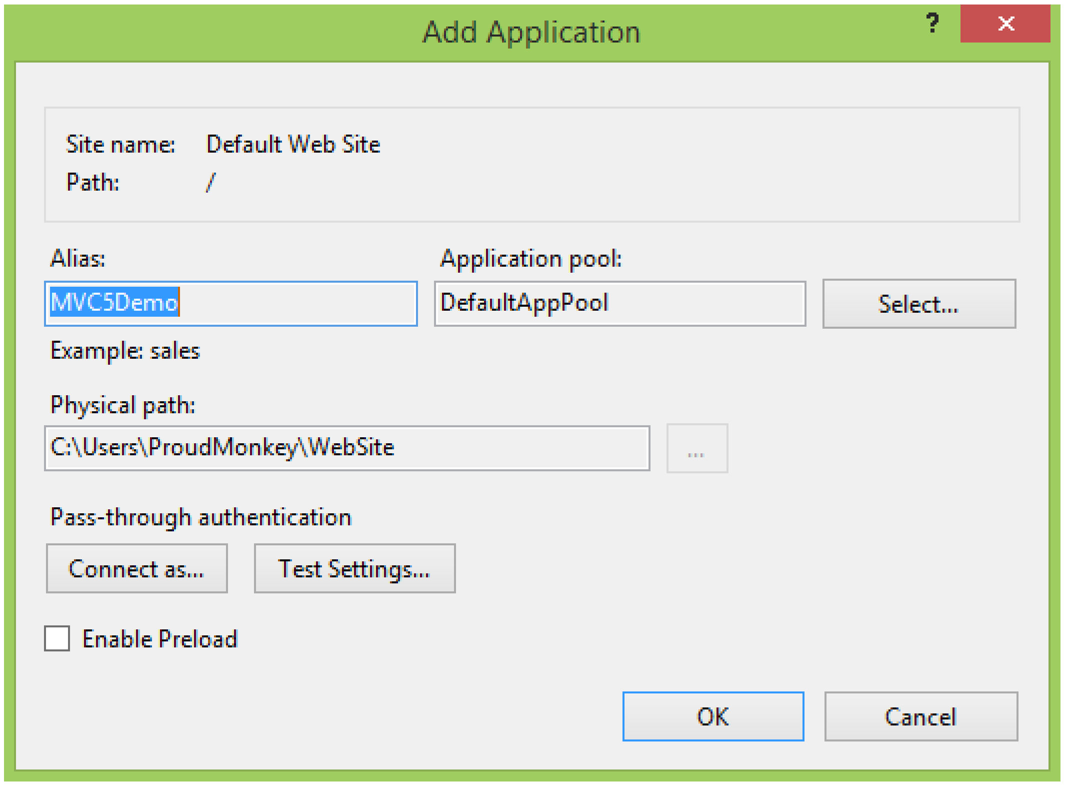
Example: images

Physical path: C:\Users\ProudMonkey\WebSite

Figure 54: Add Virtual Directory

Enter an alias name and then browse the location where you publish the source files for your web app. In this case "C:\Users\ProudMonkey\WebSite". Now click "OK". The "MVC5Demo" folder should be added under "Default Web Site".

Now right click on "MVC5Demo" folder and select "Convert to Web Application". It should bring up the following dialog.



Site name: Default Web Site
Path: /

Alias: MVC5Demo
Application pool: DefaultAppPool
Example: sales

Physical path: C:\Users\ProudMonkey\WebSite

Pass-through authentication

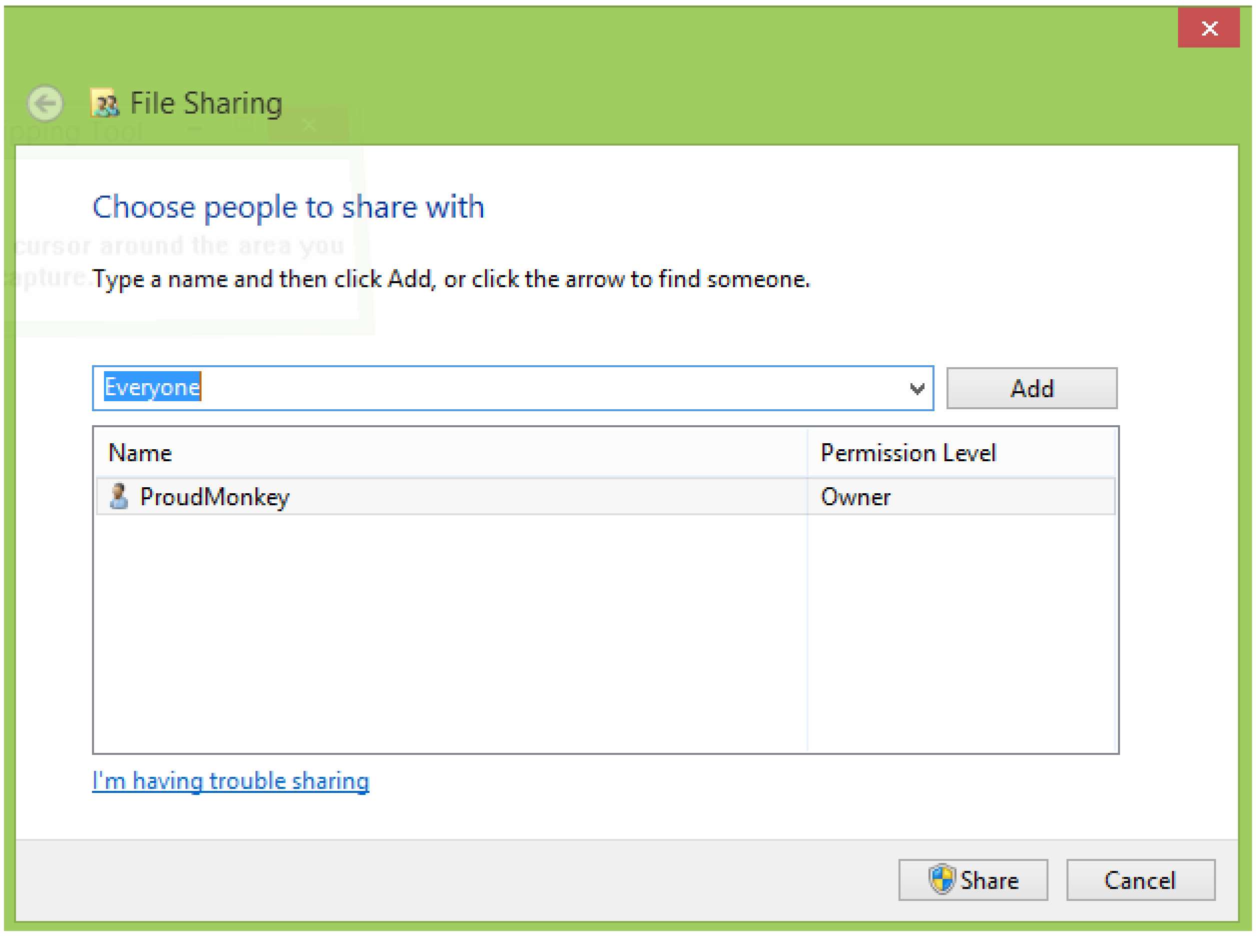
- [ ] Enable Preload

Figure 55: Add Application

Click “OK” to convert your folder into a Web Application.

Enable File Sharing in IIS

Now to ensure that our virtual location to where we publish the web site is accessible to IIS then we need to enable “Sharing” so IIS users can have access it. To do this, right-click on “MVC5Demo” and select “Edit Permissions”. In the dialog click on the “Sharing” tab and click the “Share” button. It should bring up the following dialog below.


File Sharing

Choose people to share with

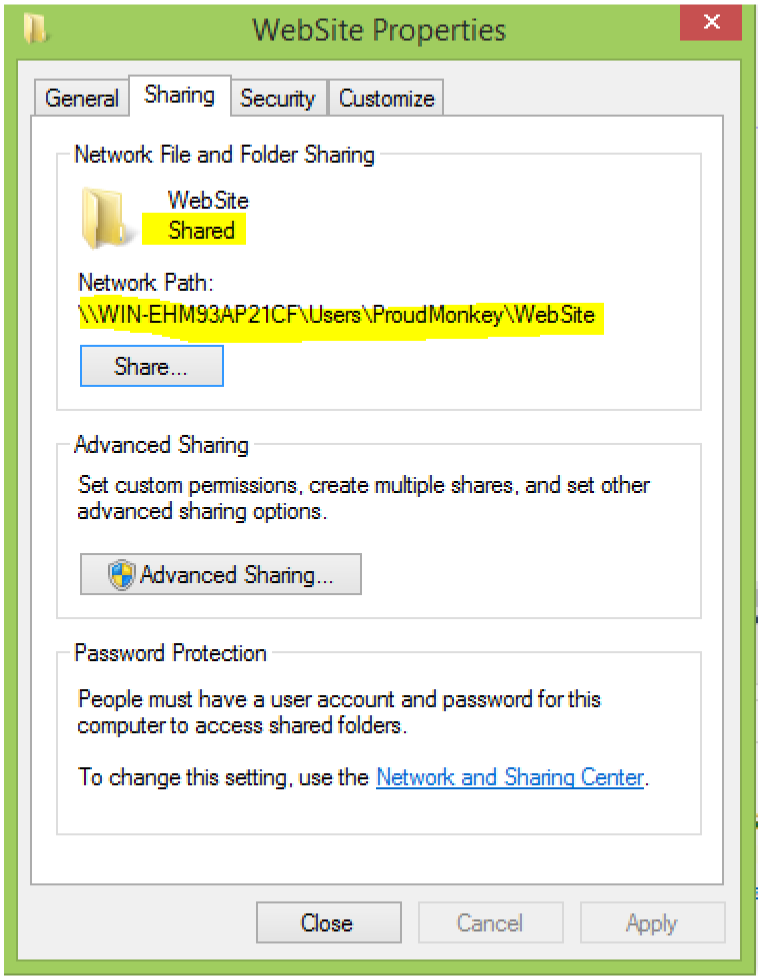
Type a name and then click Add, or click the arrow to find someone.

Everyone

Name	Permission Level
ProudMonkey	Owner

Figure 55: File Sharing

Add “Everyone” and click “Share” to add it to the list. You should now be able to see the something like below after you’ve added the users.


WebSite Properties

General | Sharing | Security | Customize

Network File and Folder Sharing

WebSite
Shared

Network Path:
\\WIN-EHM93AP21CF\Users\ProudMonkey\WebSite

Share...

Advanced Sharing

Set custom permissions, create multiple shares, and set other advanced sharing options.

Advanced Sharing...

Password Protection

People must have a user account and password for this computer to access shared folders.

To change this setting, use the Network and Sharing Center.

Close | Cancel | Apply

Figure 56: WebSite Properties

Now open up internet browser and try to access this URL:

http://localhost/MVC5Demo/Account/Login

It should show up the Login page just like in the figure below:

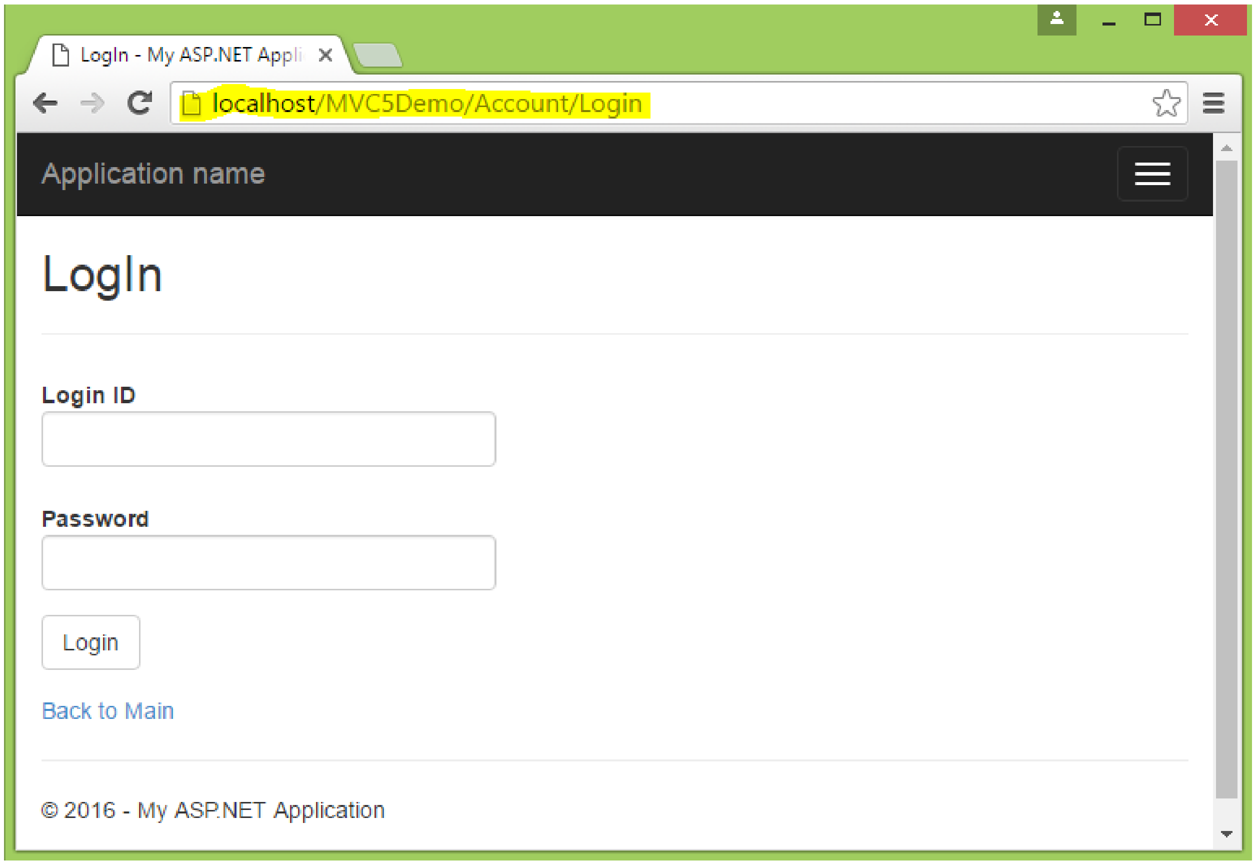
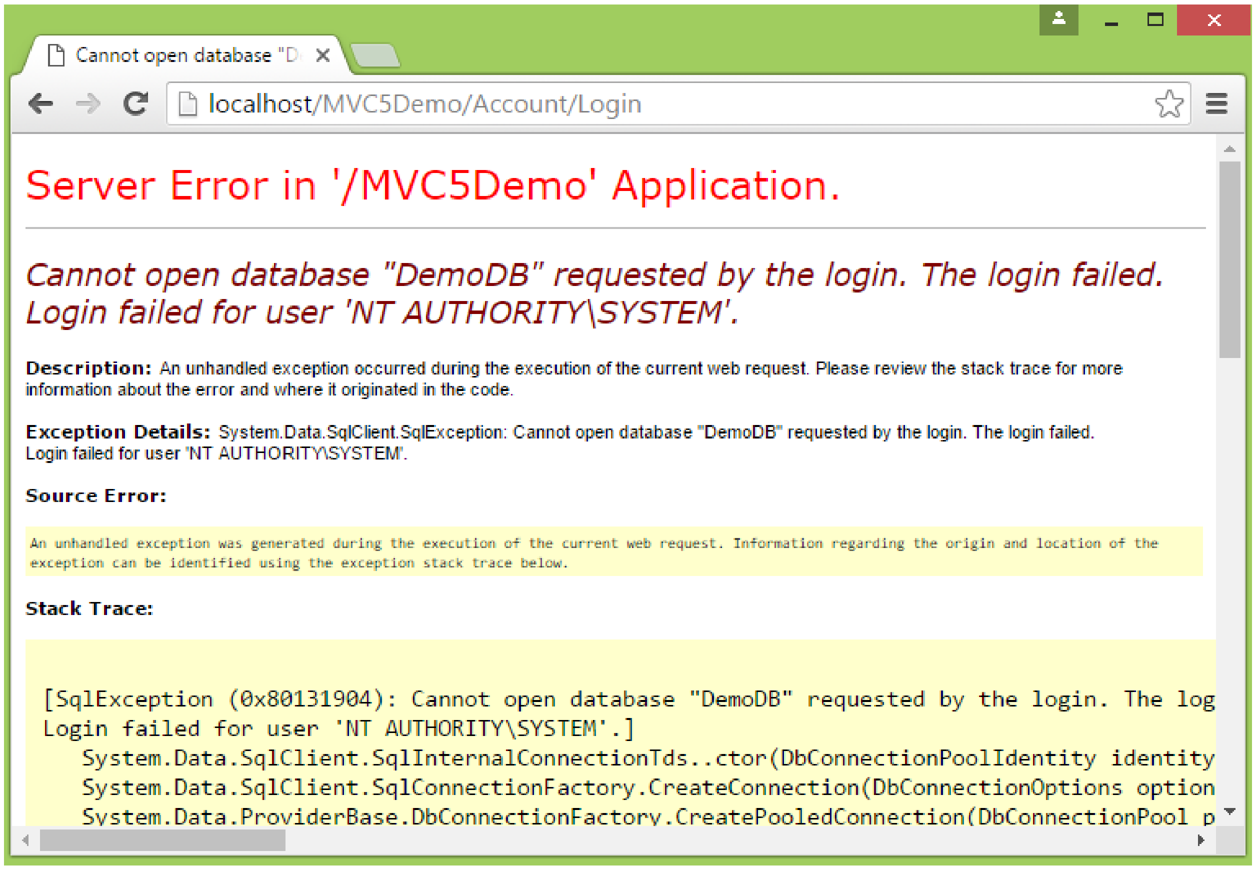


Figure 57: Hosted App in IIS

Now try to enter an account credentials and click "Login". If you are seeing the following error below, don't panic! ☺



Server Error in '/MVC5Demo' Application.

Cannot open database "DemoDB" requested by the login. The login failed.
Login failed for user 'NT AUTHORITY\SYSTEM'.

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

Exception Details: System.Data.SqlClient.SqlException: Cannot open database "DemoDB" requested by the login. The login failed.
Login failed for user 'NT AUTHORITY\SYSTEM'.

Source Error:

```
An unhandled exception was generated during the execution of the current web request. Information regarding the origin and location of the
exception can be identified using the exception stack trace below.
```

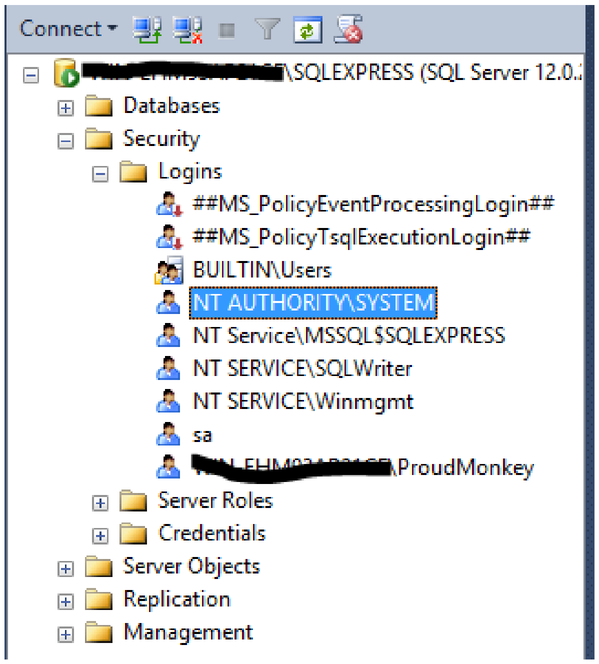
Stack Trace:

```
[SqlException (0x80131904): Cannot open database "DemoDB" requested by the login. The log
Login failed for user 'NT AUTHORITY\SYSTEM'.]
   System.Data.SqlClient.SqlInternalConnectionTds..ctor(DbConnectionPoolIdentity identity
   System.Data.SqlClient.SqlConnectionFactory.CreateConnection(DbConnectionOptions option
   System.Data.ProviderBase.DbConnectionFactory.CreatePooledConnection(DbConnectionPool p
```

Figure 58: Cannot open database error

Configuring SQL Server Logins

Open SQL Express Management Studio as an “Administrator” and navigate to Security > Logins > NT AUTHORITY\SYSTEM as shown in the figure below.



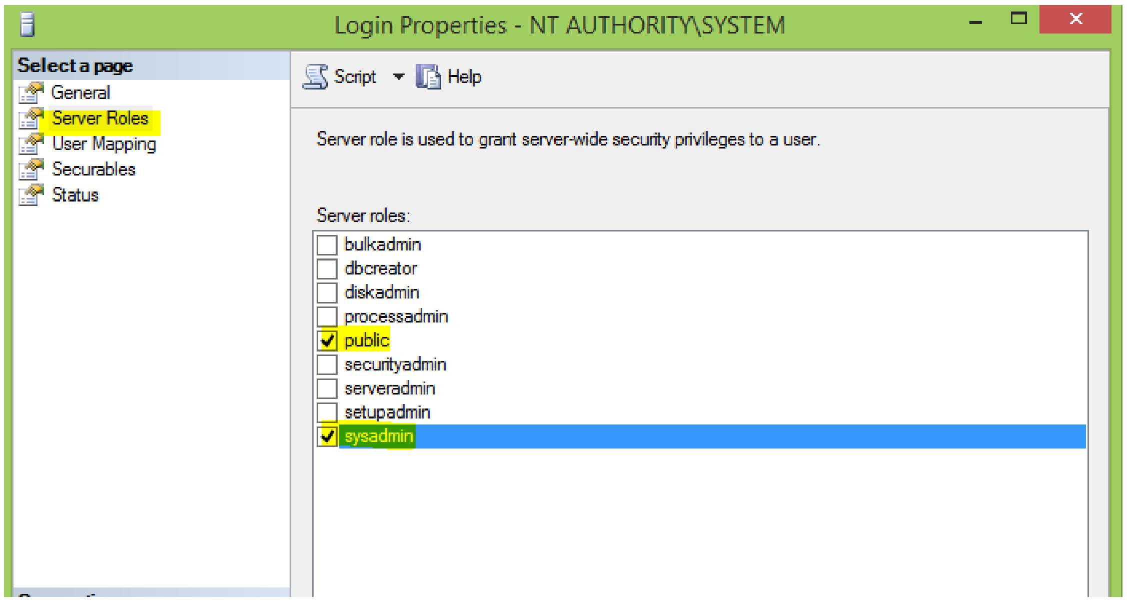
Connect

[redacted]\SQLEXPRESS (SQL Server 12.0.

- Databases
- Security
  - Logins
    - ##MS_PolicyEventProcessingLogin##
    - ##MS_PolicyTsqlExecutionLogin##
    - BUILTIN\Users
    - NT AUTHORITY\SYSTEM
    - NT Service\MSSQL$SQLEXPRESS
    - NT SERVICE\SQLWriter
    - NT SERVICE\Winmgmt
    - sa
    - [redacted]\ProudMonkey
  - Server Roles
  - Credentials
- Server Objects
- Replication
- Management

Figure 59: Configuring Logins

Right click on "NT AUTHORITY\SYSTEM" and select Properties. Select "Server Roles" from the left panel and make sure that "public" and "sysadmin" are checked as shown in the figure below.



Login Properties - NT AUTHORITY\SYSTEM

Select a page
- General
- Server Roles
- User Mapping
- Securables
- Status

Script Help

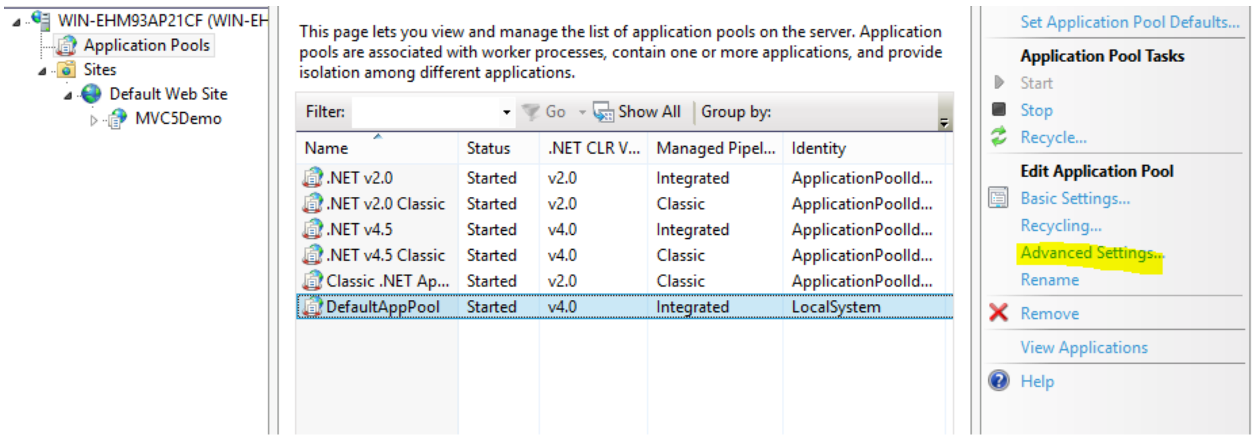
Server role is used to grant server-wide security privileges to a user.

Server roles:
- [ ] bulkadmin
- [ ] dbcreator
- [ ] diskadmin
- [ ] processadmin
- [✓] public
- [ ] securityadmin
- [ ] serveradmin
- [ ] setupadmin
- [✓] sysadmin

Figure 60: Login Properties

Configuring Application Pool's Identity

Now open IIS Manager. Select "Application Pools" and select "DefaultAppPool" from the list since our app uses this default application pool. If you are using a different application pool for your app then select that instead. On the left panel, select the link "Advance Settings" as shown in the figure below.

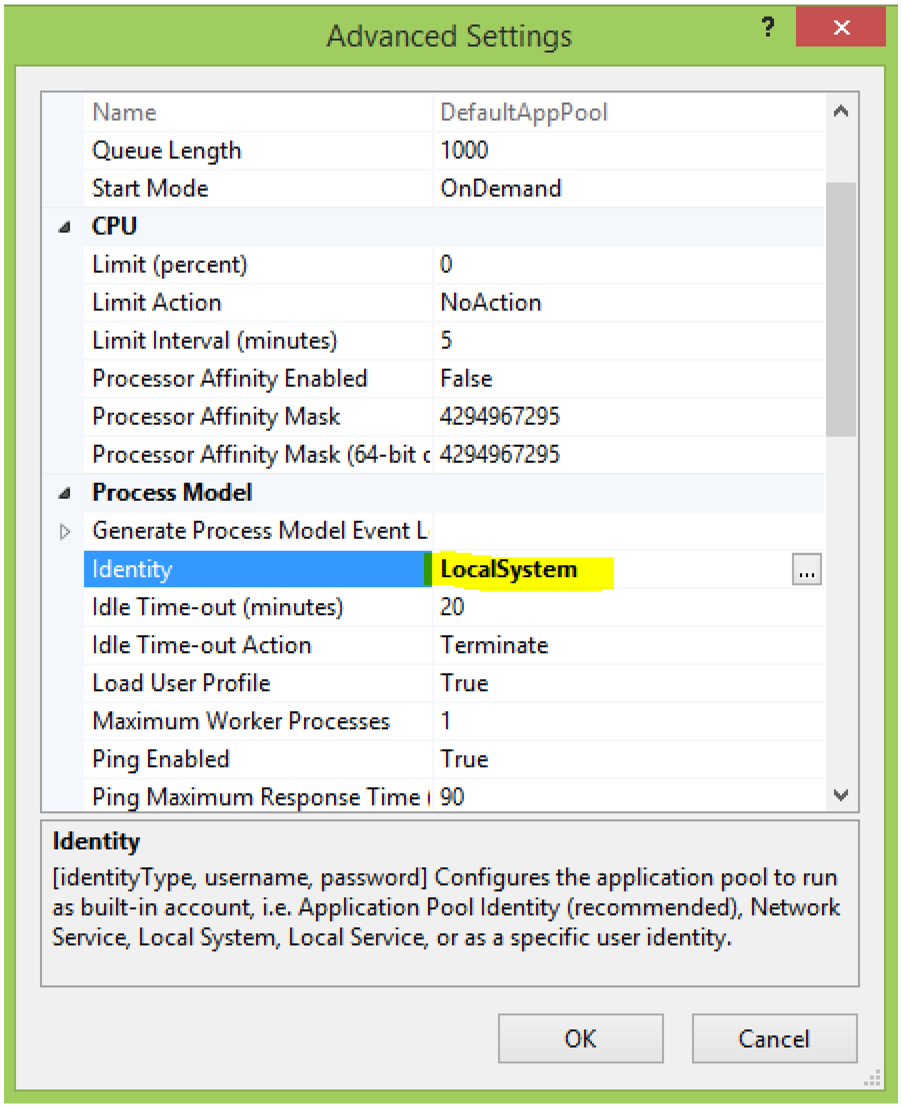


This page lets you view and manage the list of application pools on the server. Application pools are associated with worker processes, contain one or more applications, and provide isolation among different applications.

Name	Status	.NET CLR V...	Managed Pipel...	Identity
.NET v2.0	Started	v2.0	Integrated	ApplicationPoolId...
.NET v2.0 Classic	Started	v2.0	Classic	ApplicationPoolId...
.NET v4.5	Started	v4.0	Integrated	ApplicationPoolId...
.NET v4.5 Classic	Started	v4.0	Classic	ApplicationPoolId...
Classic .NET Ap...	Started	v2.0	Classic	ApplicationPoolId...
DefaultAppPool	Started	v4.0	Integrated	LocalSystem

Figure 61: AppPool Advance Settings

Make sure that you select "Local System" as the Identity from the Advance Settings dialog as shown in the figure below.



Advanced Settings

Name	DefaultAppPool
Queue Length	1000
Start Mode	OnDemand
CPU	
Limit (percent)	0
Limit Action	NoAction
Limit Interval (minutes)	5
Processor Affinity Enabled	False
Processor Affinity Mask	4294967295
Processor Affinity Mask (64-bit c	4294967295
Process Model	
Generate Process Model Event L	
Identity	LocalSystem
Idle Time-out (minutes)	20
Idle Time-out Action	Terminate
Load User Profile	True
Maximum Worker Processes	1
Ping Enabled	True
Ping Maximum Response Time (	90

Identity
[identityType, username, password] Configures the application pool to run as built-in account, i.e. Application Pool Identity (recommended), Network Service, Local System, Local Service, or as a specific user identity.

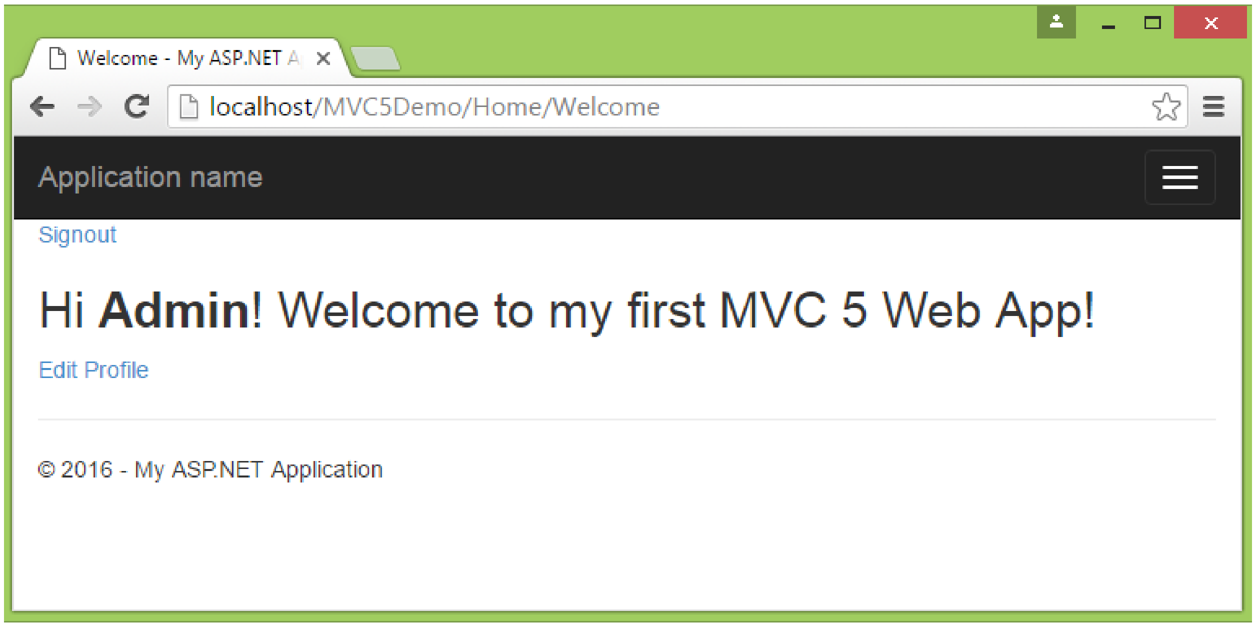
Figure 62: Advance Settings

Click “OK” and try to browse your page again using the same URL.

Running Your Application

You should now be able to connect to your database. Here are some screen shots of the page hosted in IIS.

After logging-in

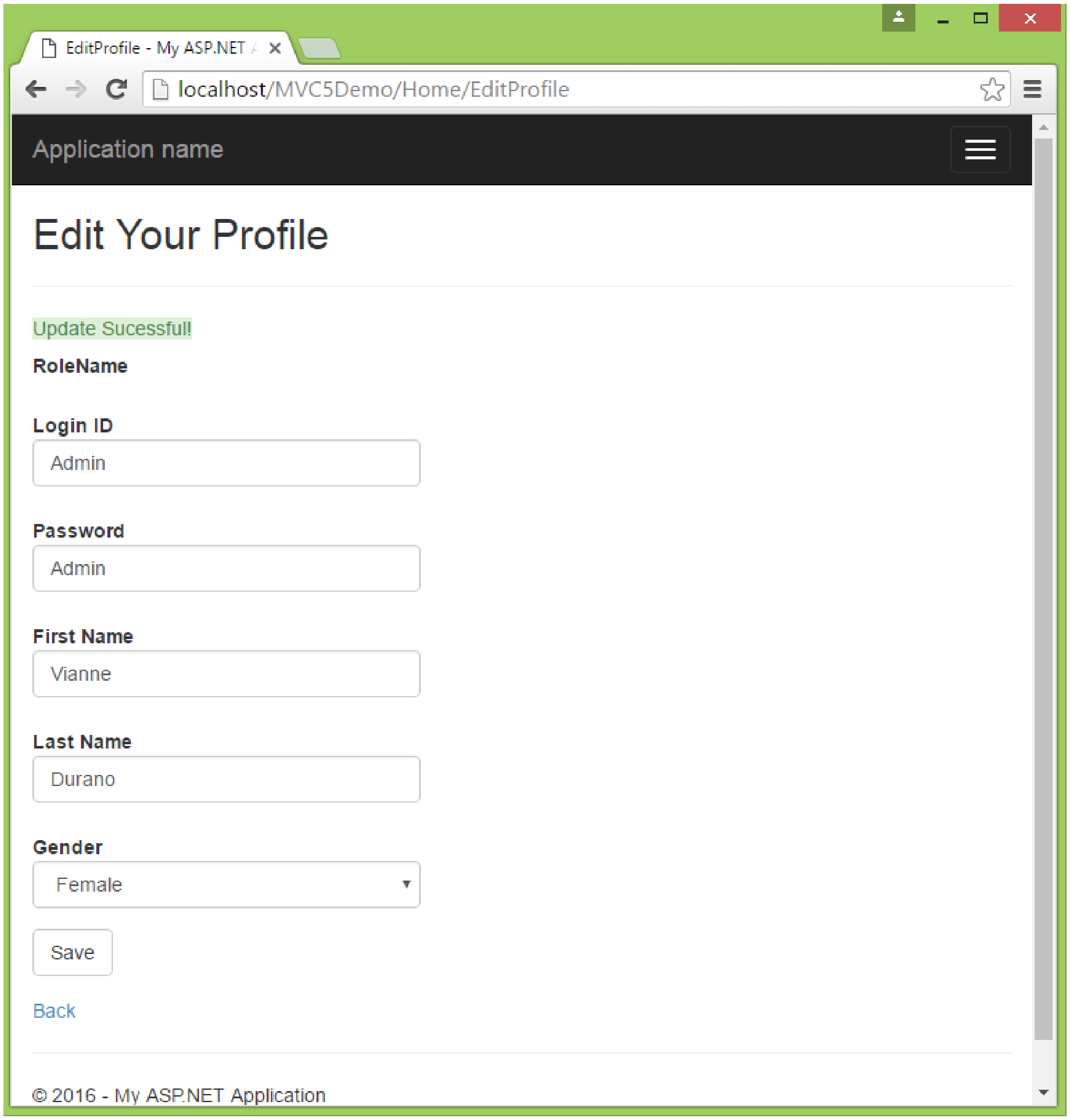


Hi **Admin**! Welcome to my first MVC 5 Web App!

© 2016 - My ASP.NET Application

Figure 63: After Successful Login

After updating the database


EditProfile - My ASP.NET

localhost/MVC5Demo/Home/EditProfile

Application name

Edit Your Profile

Update Sucessful!

RoleName

Login ID: Admin

Password: Admin

First Name: Vianne

Last Name: Durano

Gender: Female

Save

Back

© 2016 - My ASP.NET Application

Figure 64: After successful update to the database

That’s it. You now have a web app hosted in IIS Web Server that is up and running. ☺

Summary

This book was targeted to beginners and to developers who are still confused on how to start building an ASP.NET MVC 5 application from scratch. I’ve demonstrated the basics on creating a database and how to perform basic CRUD operations in MVC5 using Entity Framework as the data access mechanism. Along the way, I have shown how to integrate jQuery and jQueryUI to perform client-side way of manipulating the data from the UI to the database. I have also shown how to use jQuery AJAX to perform asynchronous operations and real-time update using the ShoutBox as an example. Deploying an application to local IIS Web Server was also included in the exercise.

The features demonstrated in this book are not full-blown and there are a lot of rooms for improvement. What I’ve shown was just the basic and to guide you to get something working. You can always enhance and add more features to it if you’d like to and apply the things that wasn’t included in this book, for example enhancing the look and feel of the page or even extend the database to support shopping cart. That’s just few of the examples that you can integrate.

I hope somehow you find this book useful.