Lab: Creating a Blog with ASP.NET MVC

In this lab, we shall create a fully-functional **Blog System** in ASP.NET MVC with SQL Server database using Entity Framework and MVC. This tutorial is part of the "Software Technologies" course @ SoftUni.

I. Overview

ASP.MVC – this is a **web application framework** developed by Microsoft, which implements the **model–view–controller** (**MVC**) pattern, with which you should already be familiar with. In other words, this gives you a **bare bone working web app** (you will see that you can start it immediately after creating the project) out of the box, on top of which you can **build your own app**. Consider it our **foundation**.

Entity Framework – basically, this gives you a way to **interact with a database** by making you see database objects (**tables**) as classes (It is analogous to Doctrine, which you have already used but is very different). Once familiar with **object-oriented programming** you should appreciate how handy this is.

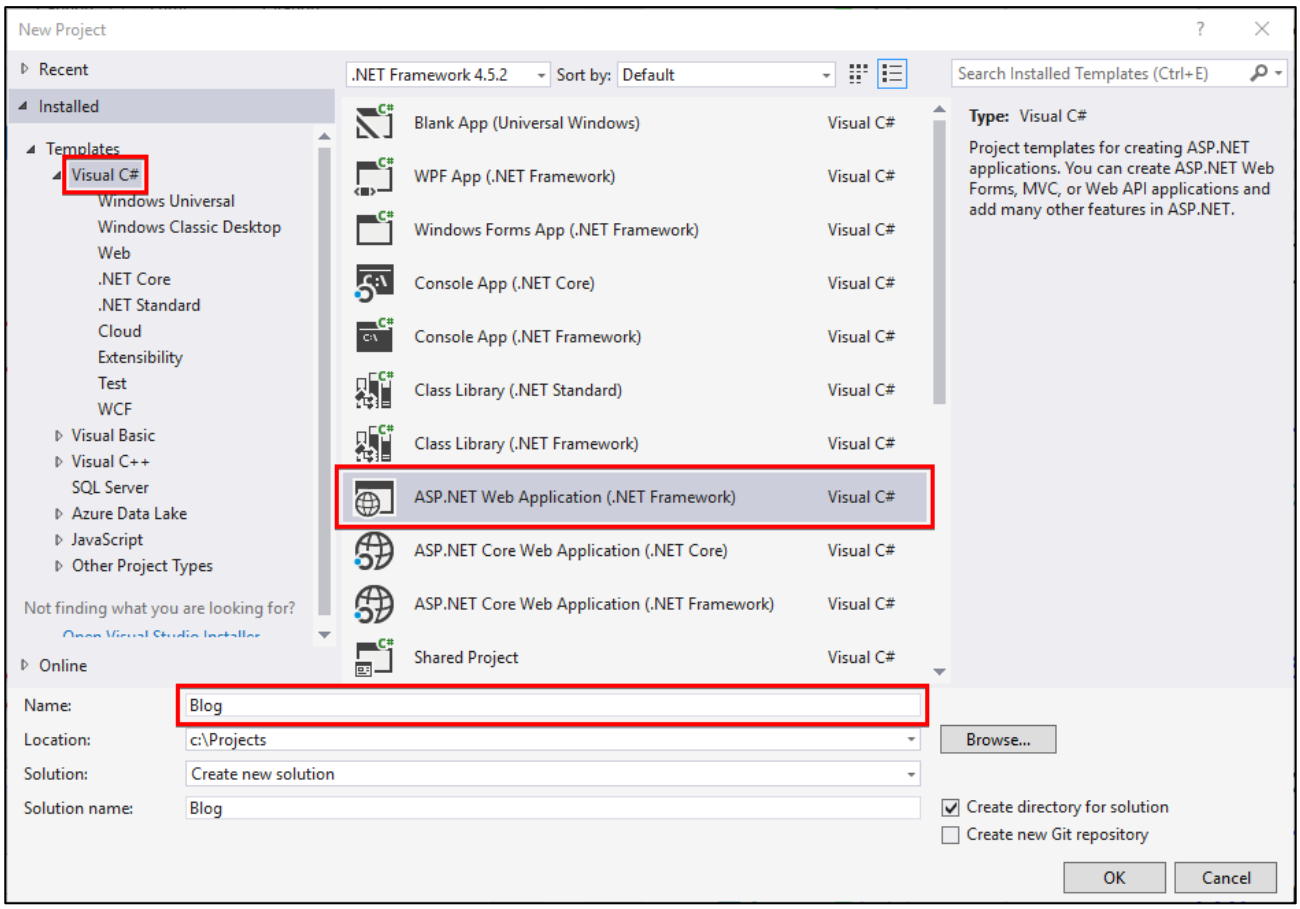
SQL – query language used for **managing a database**. In our case, Entity Framework will take care of this.

II. Initial Setup

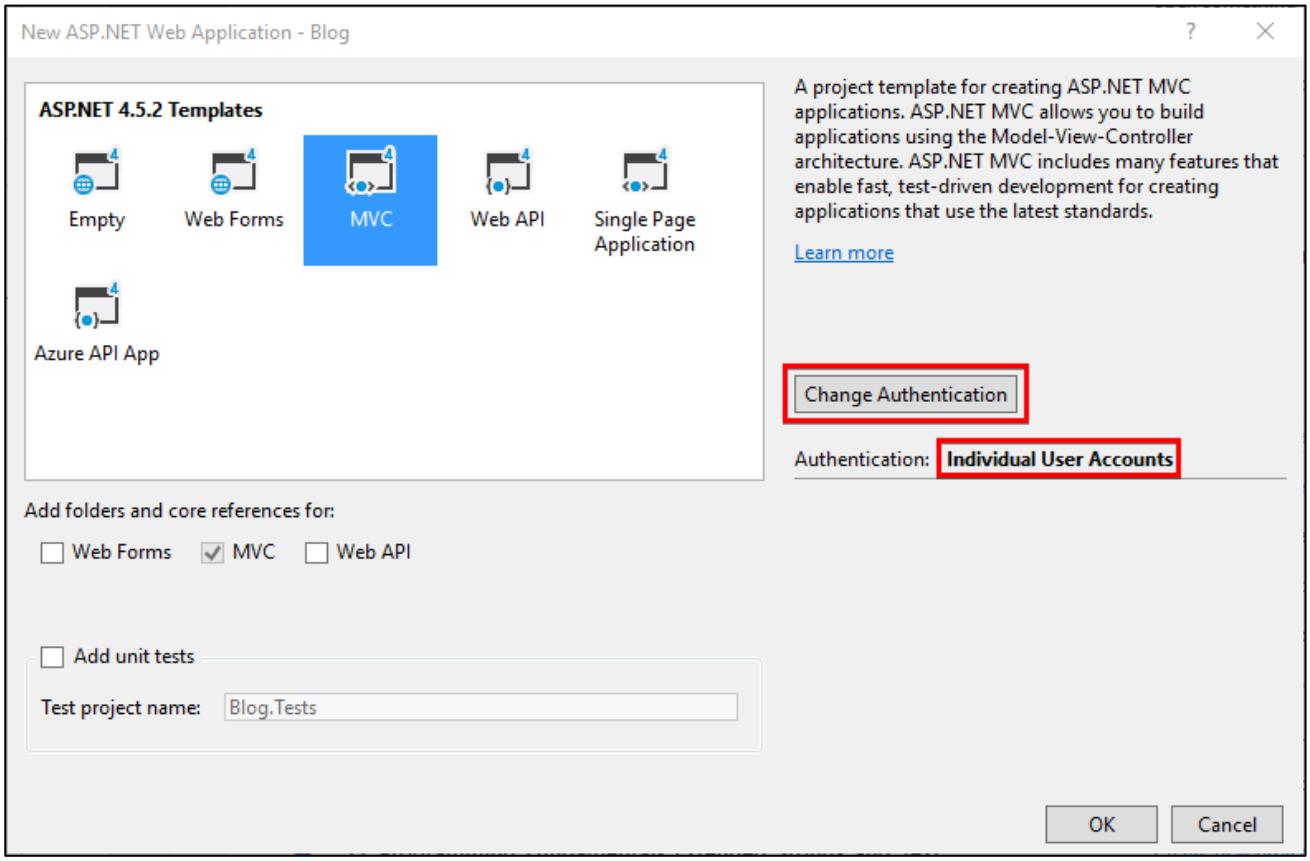
In this section we will setup our project and lay the foundations.

1. Create a New ASP.NET MVC Application

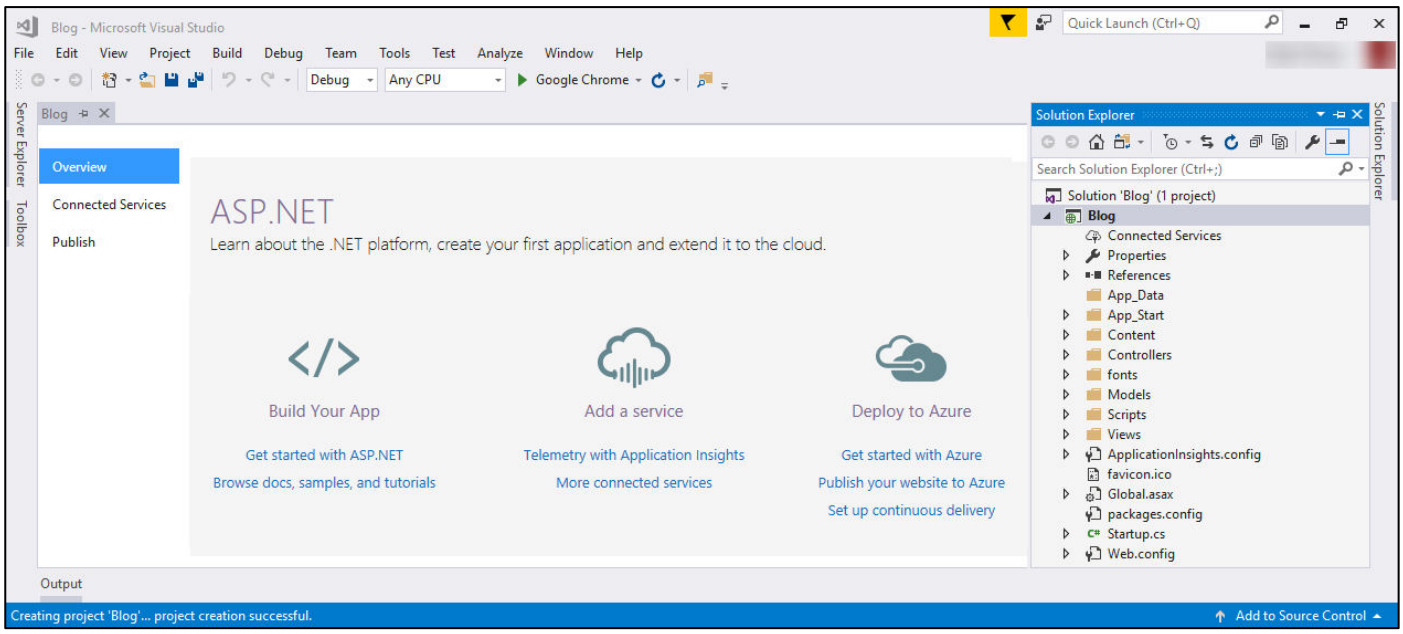
First, let's start by creating an **ASP.NET MVC Application** in Visual Studio. Don't forget to name the project appropriately, as if you leave this for later, you can encounter major problems. All code in the guide is made in a project with the name "**Blog**":



In the next window, choose "**MVC**" and untick the "**Host in the cloud**" checkbox (if you use Visual Studio 2015). Also change the authentication method to "**Individual User Account**":

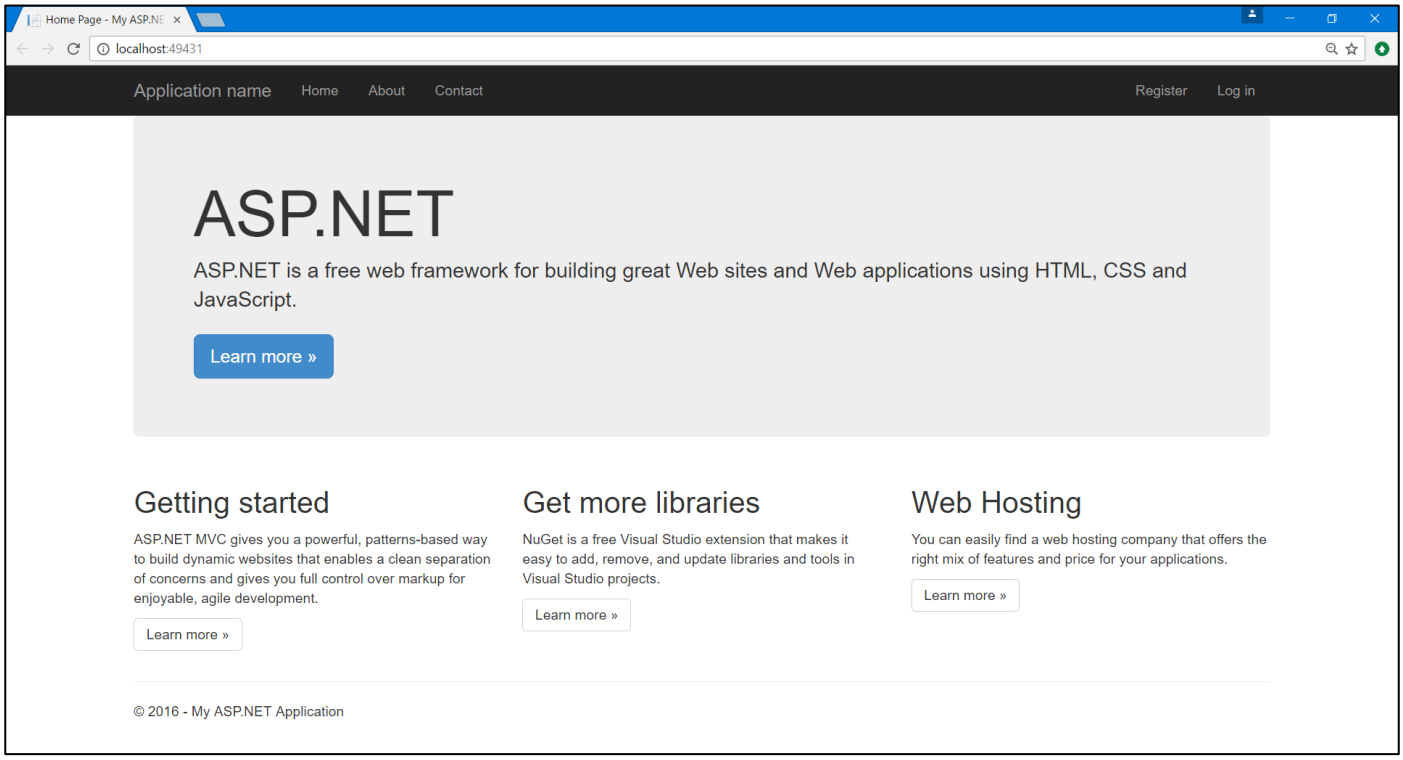


Click on **[OK]** and you should see the following window:



2. Run the Application

Run the application to see what was generated by the Visual Studio MVC application template. Press **[Ctrl+F5]**.



3. Register a User

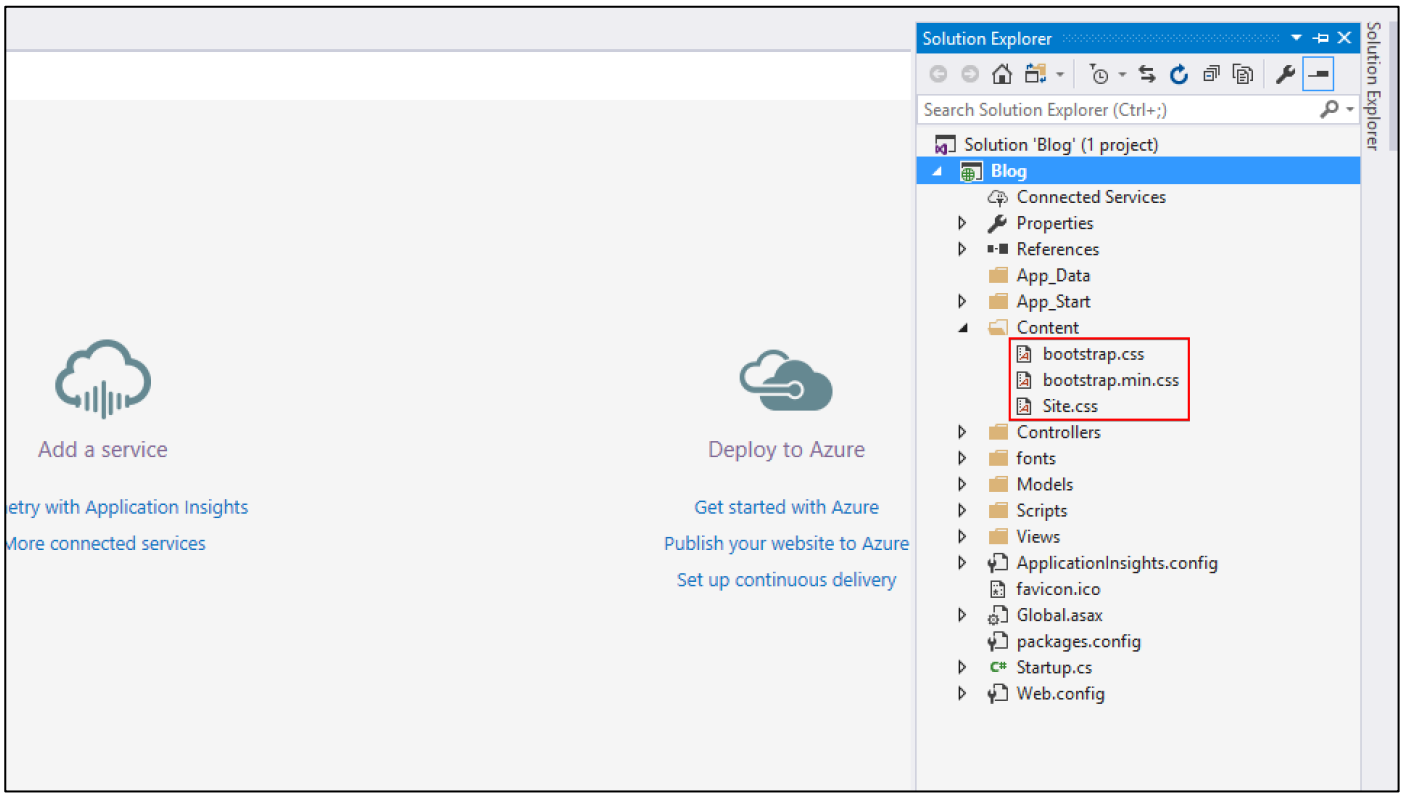
Click on the button in the upper right corner and **register a user**. **If you register** a user successfully **proceed with the next steps**.

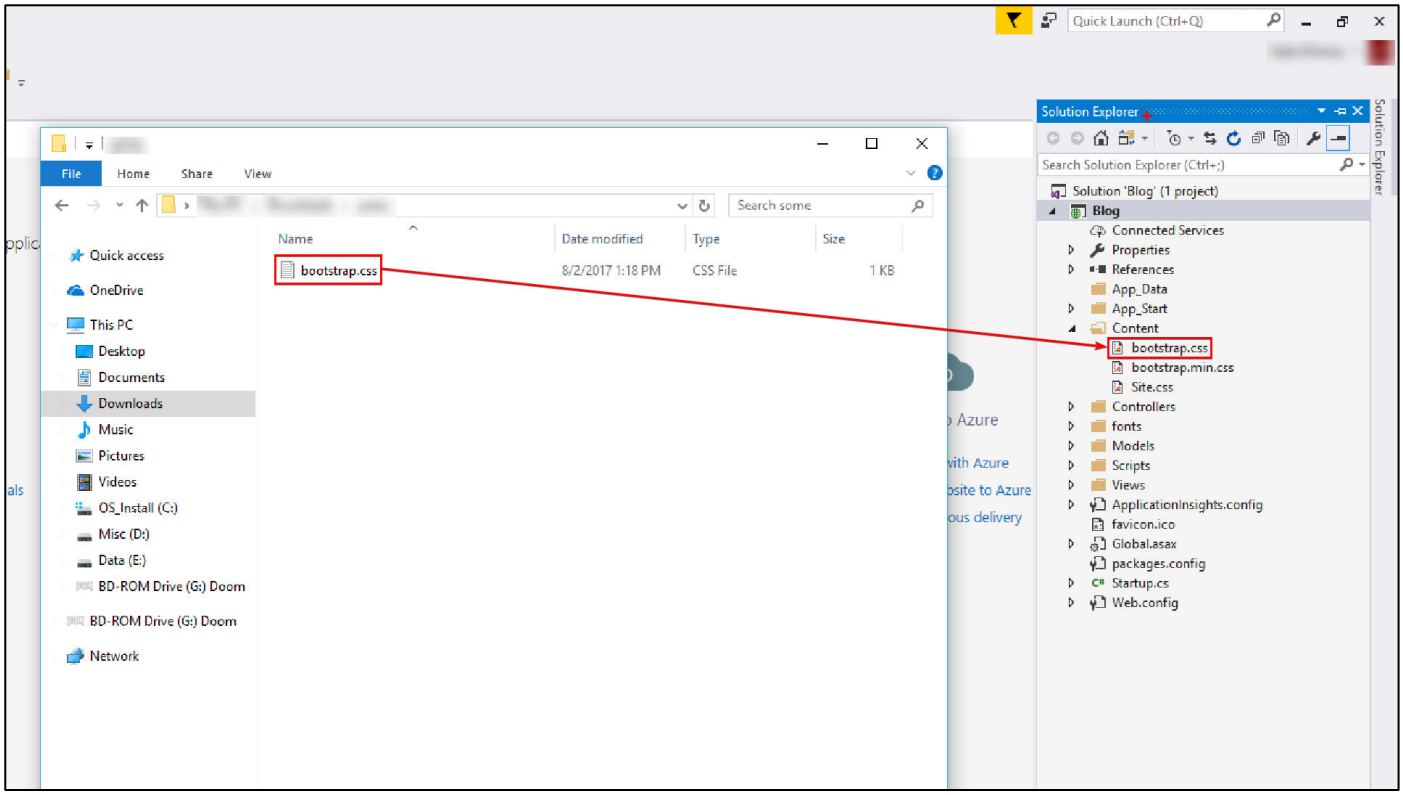
If you get an exception, you need to **install MSSQL Server LocalDB**. You can get the instructions from the course instance.

After you are done installing, **proceed with blog creation**.

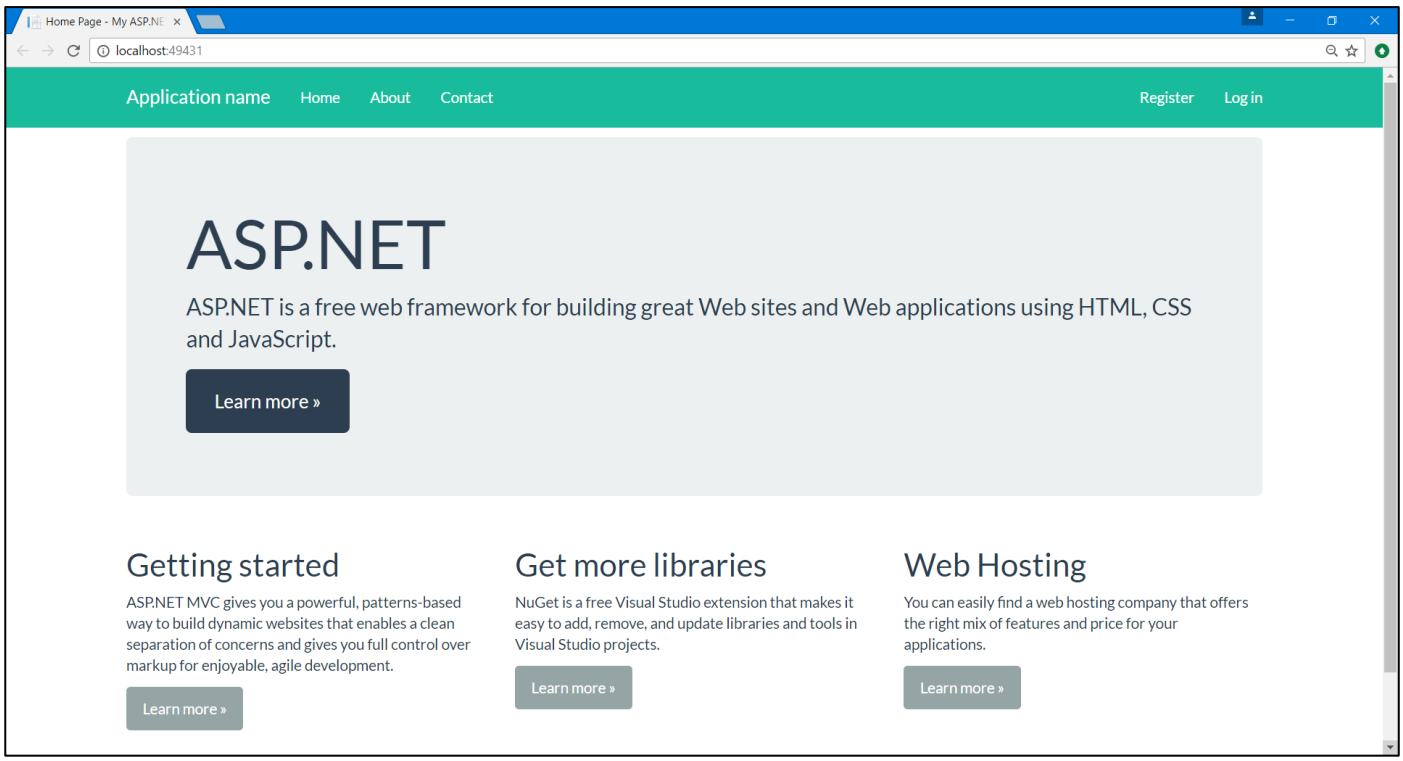
4. Plug in Custom Bootstrap File

We need to **insert our bootstrap file** in the project so we can start using it. This can be done by replacing the old one (because **ASP.NET MVC** already uses bootstrap) by heading to the "**Content**" folder. Just delete all files there and place the provided **bootstrap.css**:





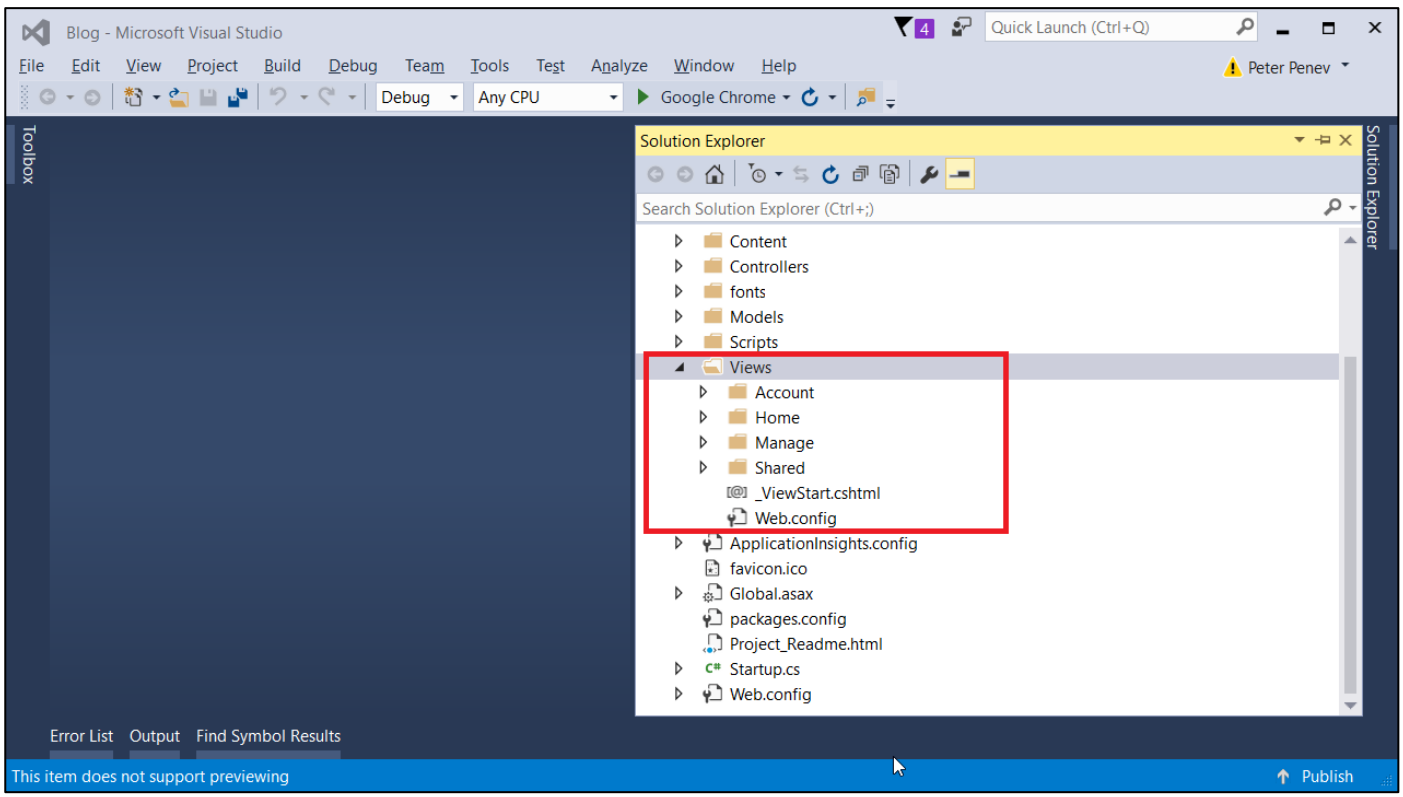
After you do this, **start the application** again to make sure everything works and that you have the new style. Make sure that you **build the project** again with **[Ctrl + Shift +B]**, because you need to recompile in order to see the changes. You can also just save it **[Ctrl + Shift + S]** and start it again **[Ctrl + F5]**:



Now, obviously the navbar looks different, but that is just because the default layout uses the "**navbar-inverse**" bootstrap class. When we **edit the general layout of the blog**, everything will start to look as it should.

5. Setup the General Layout of the Blog

Now we need to set the **general layout**. Just head to the "**Views**" folder. This is the folder that **holds all the views** for the project. Inside you can see that there are some other folders:



- **Account** - This folder holds all views that are related to **user registration** and **login**
- **Home** - Contains the **home page views**

- **Manage** - Views related to **managing user accounts** (changing password, adding a phone number, etc.)
- **Shared** - folder containing views that are **common for all of the project** (general layout, contact page, etc.)

As you probably guessed we need the "**Shared**" folder. In it you will find a file named **"_Layout.cshtml**". It begins with an underscore because it is a **partial view** (more on that later).

In it just **delete all code** and **paste the provided** one:

\Views\Shared_Layout.cshtml

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@ViewBag.Title - SoftUni Blog</title>
    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/modernizr")
</head>
<body>
    <div class="navbar navbar-default 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("SOFTUNI BLOG", "Index", "Home", new { area = "" }, new {
@class = "navbar-brand" })
            </div>
            <div class="navbar-collapse collapse">
                @Html.Partial("_LoginPartial")
            </div>
        </div>
    </div>
    <div class="container body-content">
        @RenderBody()
        <footer class="pull-right">
            <p>&copy; @DateTime.Now.Year - SoftUni Blog</p>
        </footer>
    </div>

    @Scripts.Render("~/bundles/jquery")
    @Scripts.Render("~/bundles/bootstrap")
    @RenderSection("scripts", required: false)
</body>
</html>
```

Test again and now you should see that the navbar is the good old navbar that you see for the fourth time in this course:

Now, there are many **new** and **interesting** things in the code that you have just copied and pasted.

For example, you can see **some of the code has** the **"@"** sign in front of it. This is part of the **Razor** engine and is analogous to **Twig** (you saw that in PHP and Symphony). Basically, it gives you the ability to **mix HTML code with C#**

code. Everywhere that there is a **"@"** means that there is a **language switch** (it is a little bit more complex, if you are interested in the syntax, google it).

For example:

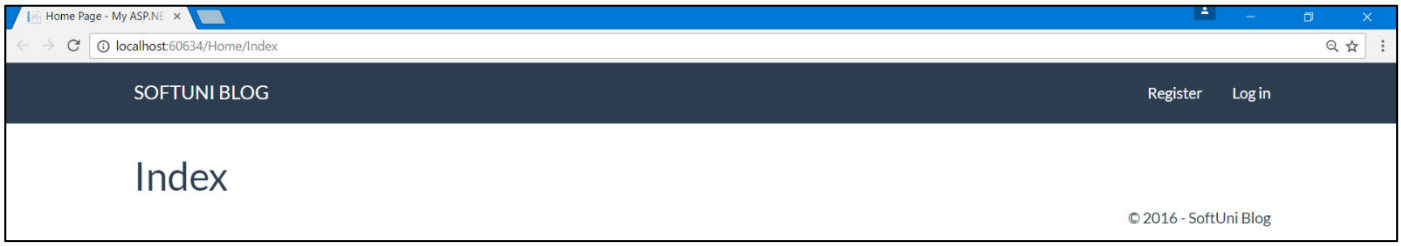
- **@Styles.Render("~/Content/css")** – links the page to the CSS style files.
- **@Html.ActionLink("SOFTUNI BLOG", "Index", "Home", new { area = "" }, new { @class = "navbar-brand" })** – creates a link with name "SOFTUNI BLOG", that will be accessible through the "**Index**" method of the "**Home**" controller.
- **@Html.Partial("_LoginPartial")** - this renders the partial view (a view inside another view) **_LoginPartial**. You can find it at "**Shared/_LoginPartial.cshtml**" folder.
- **@RenderBody()** - renders the view that is currently passed to the browser. For example, if you are accessing the index action in the home controller, the view of the body will be "**Views/Home/Index.cshtml**".

6. Edit the Home Page

We are just going to put a placeholder on the home page. Go to "**Views/Index.cshtml**" and paste this:

```
@{
    ViewBag.Title = "Home Page";
}

<h1>Index</h1>
```

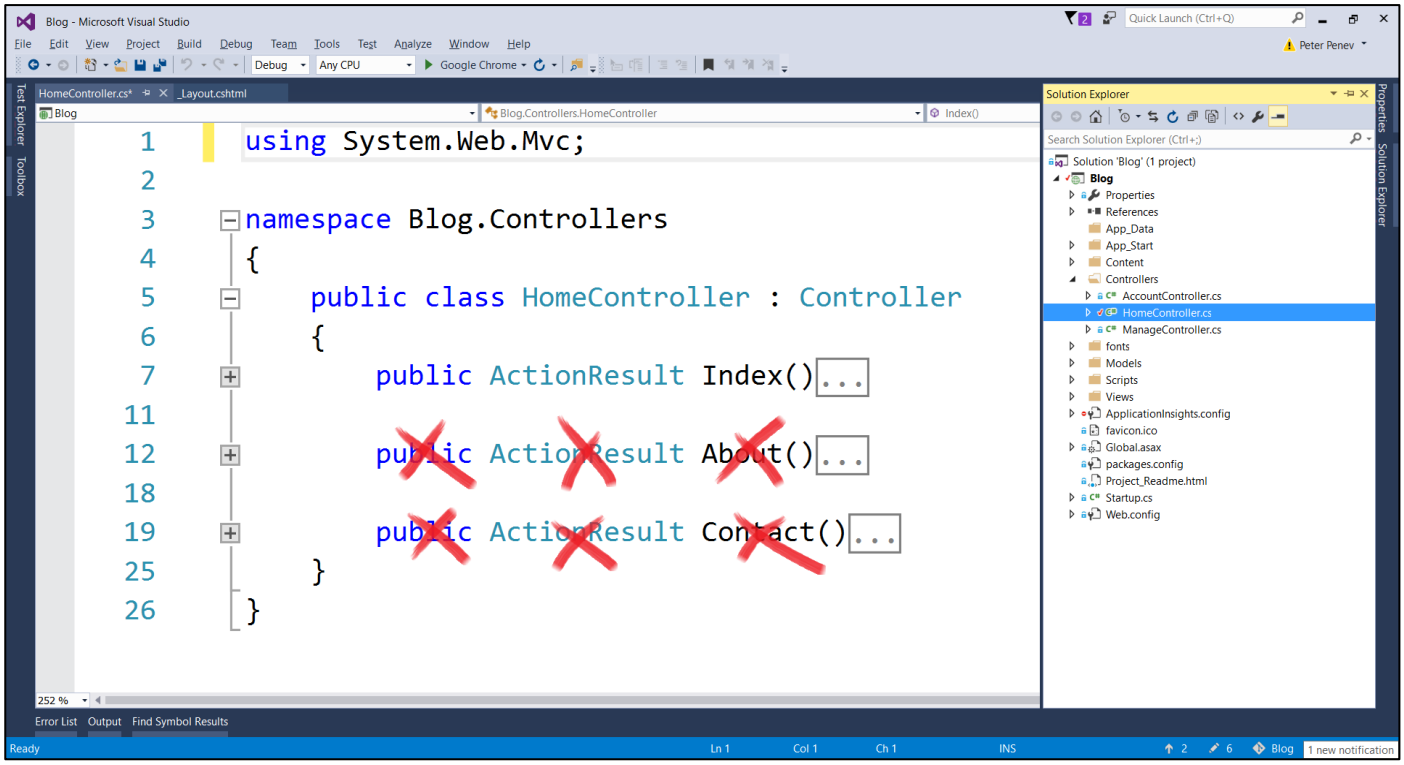
It will just make the home page look like this:



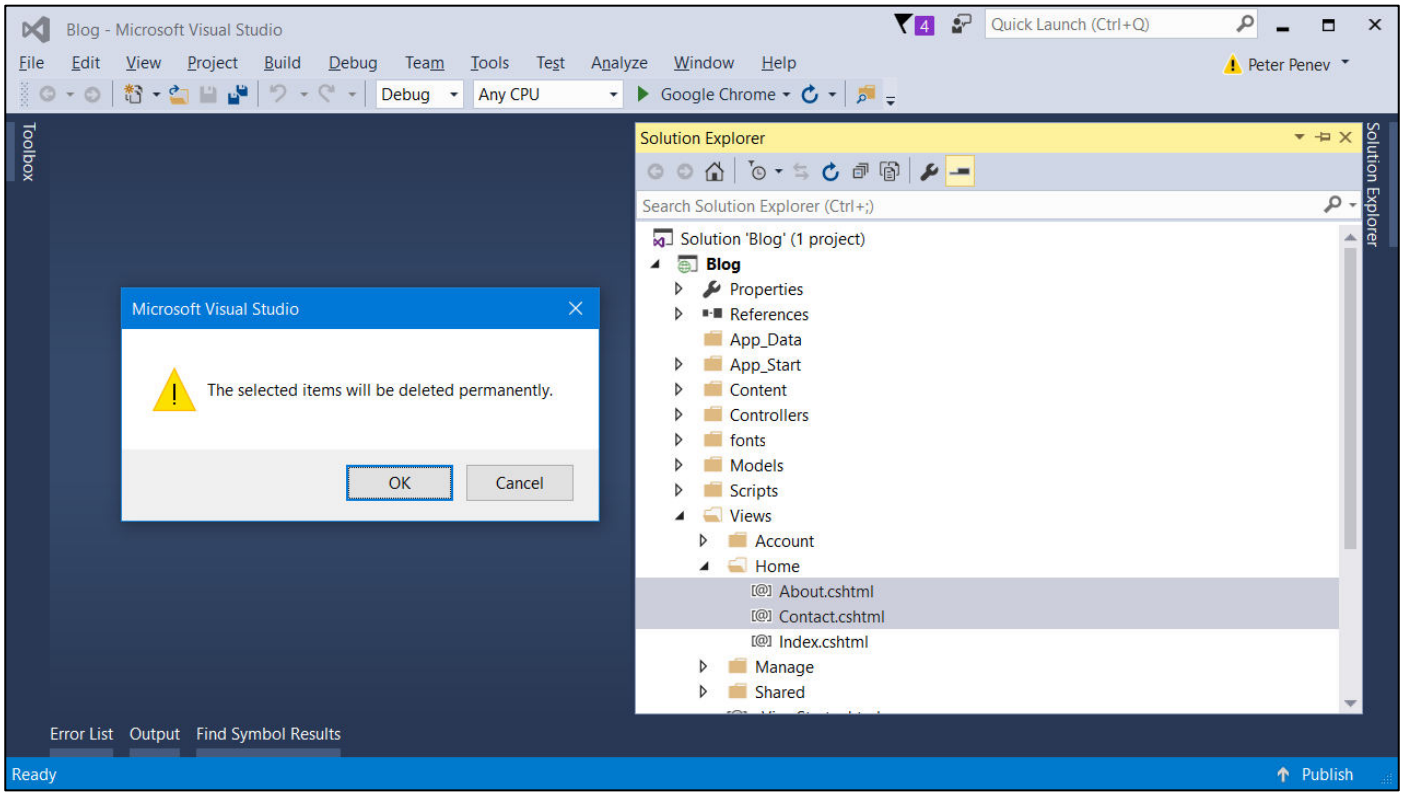
7. Delete Unnecessary Actions and Views

Now if you head to the Home Controller (**"Controllers/HomeController.cs"**) you should see that there are three methods (actions - **Index()**, **About()** and **Contact()**). You don't need **About()** and **Contact()**.

Also, if you look at the home page **we don't have links** to them anymore. You can **safely delete them**:



This also means that **you can safely delete the views** that are linked to them. You can find them at "**Views/Home**":



III. Edit Class Structure

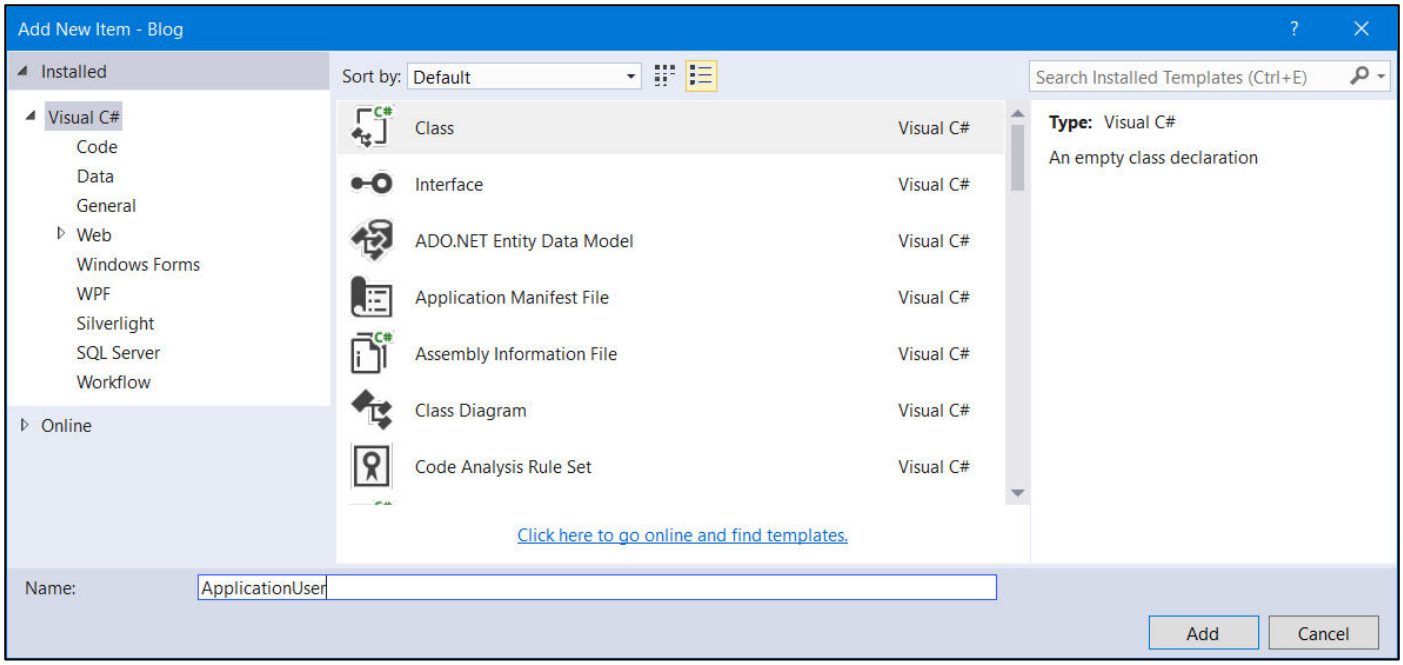
1. Separate Application User and Database Context

In **ASP.NET MVC** you get the user **authentication** as part of the package, but let's take a brief examination.

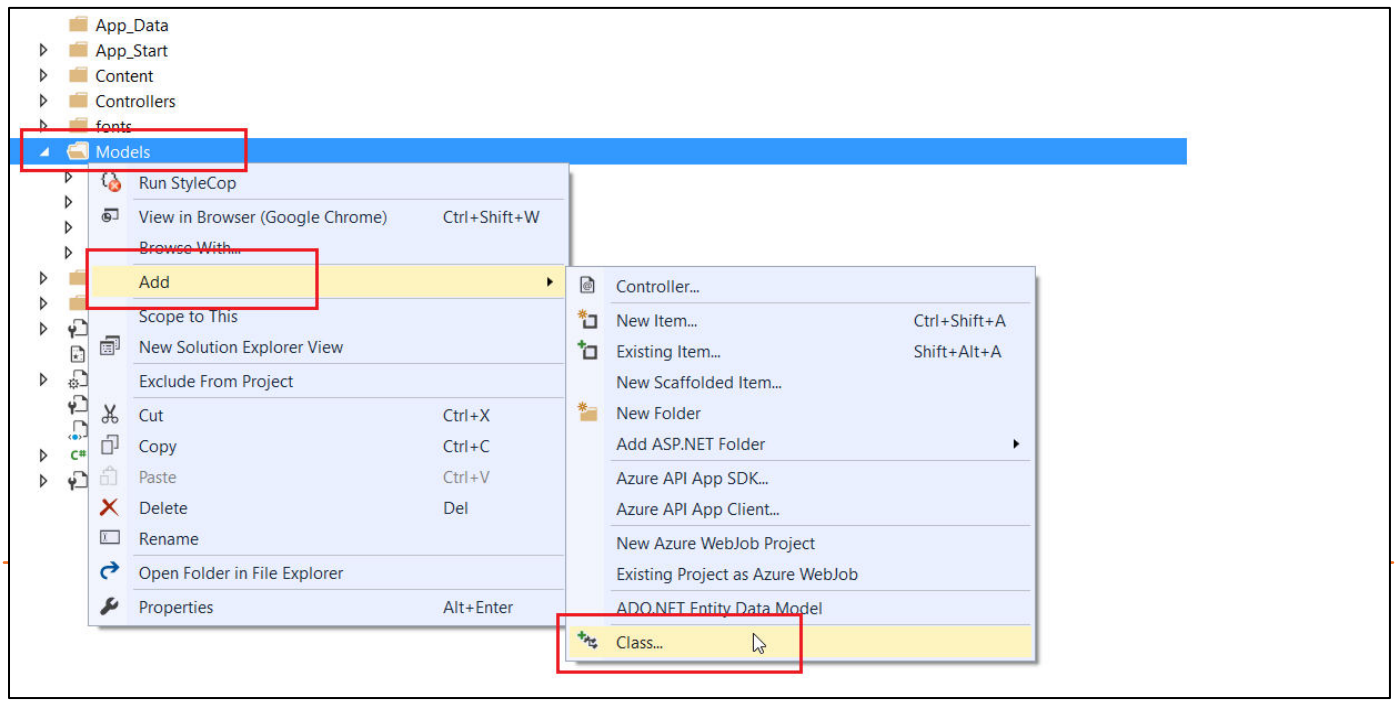
You can see the user class at "**Models/IdentityModels.cs**". There are **two classes inside the file** and one of them is the **ApplicationUser** class. It **holds information about the user** such as his user name, email, password etc. However, it is a **good practice to have only one class inside a file**, so let's do exactly this:

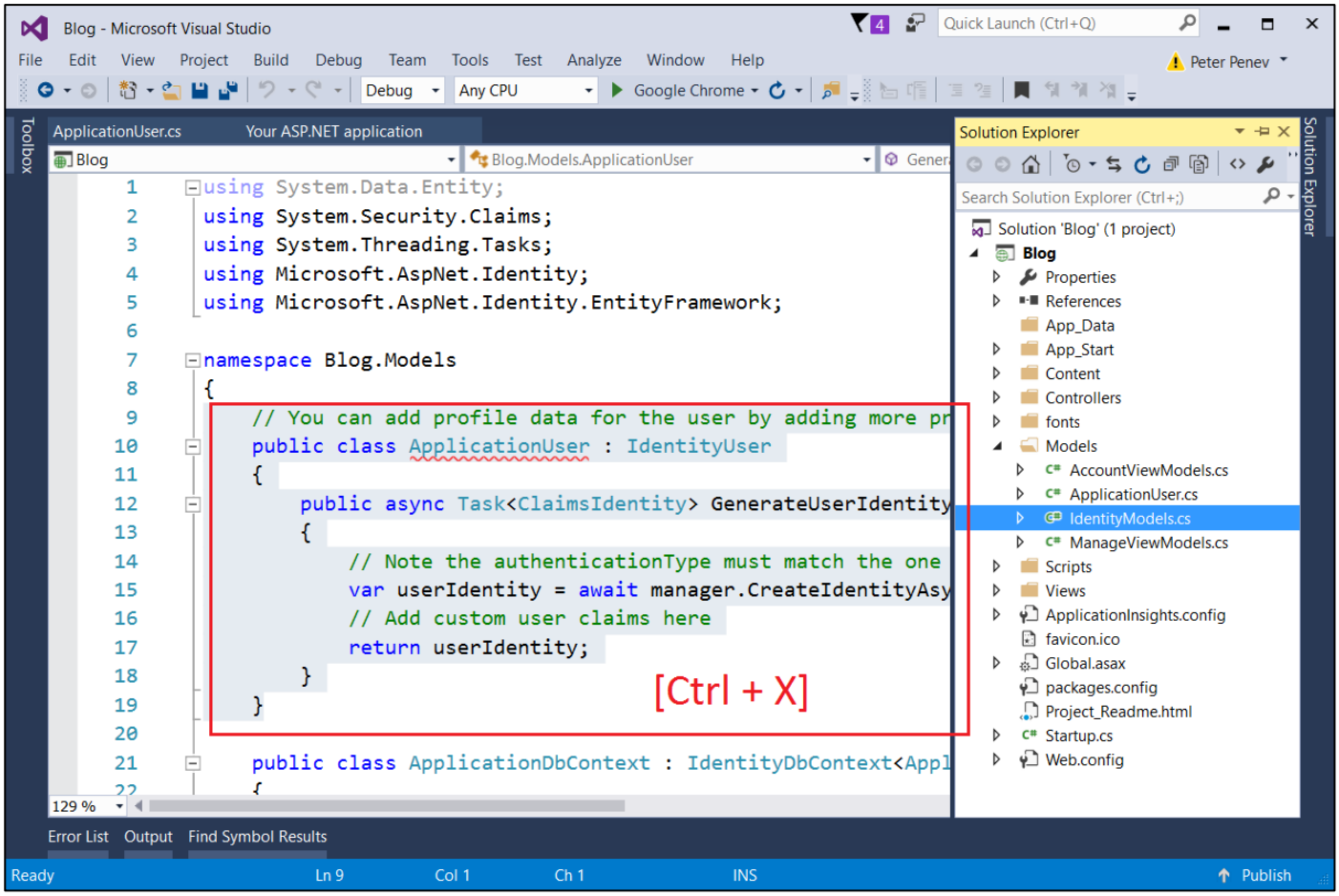
Create a new class inside the same folder "**IdentityModels**":

Call it "**ApplicationUser**":

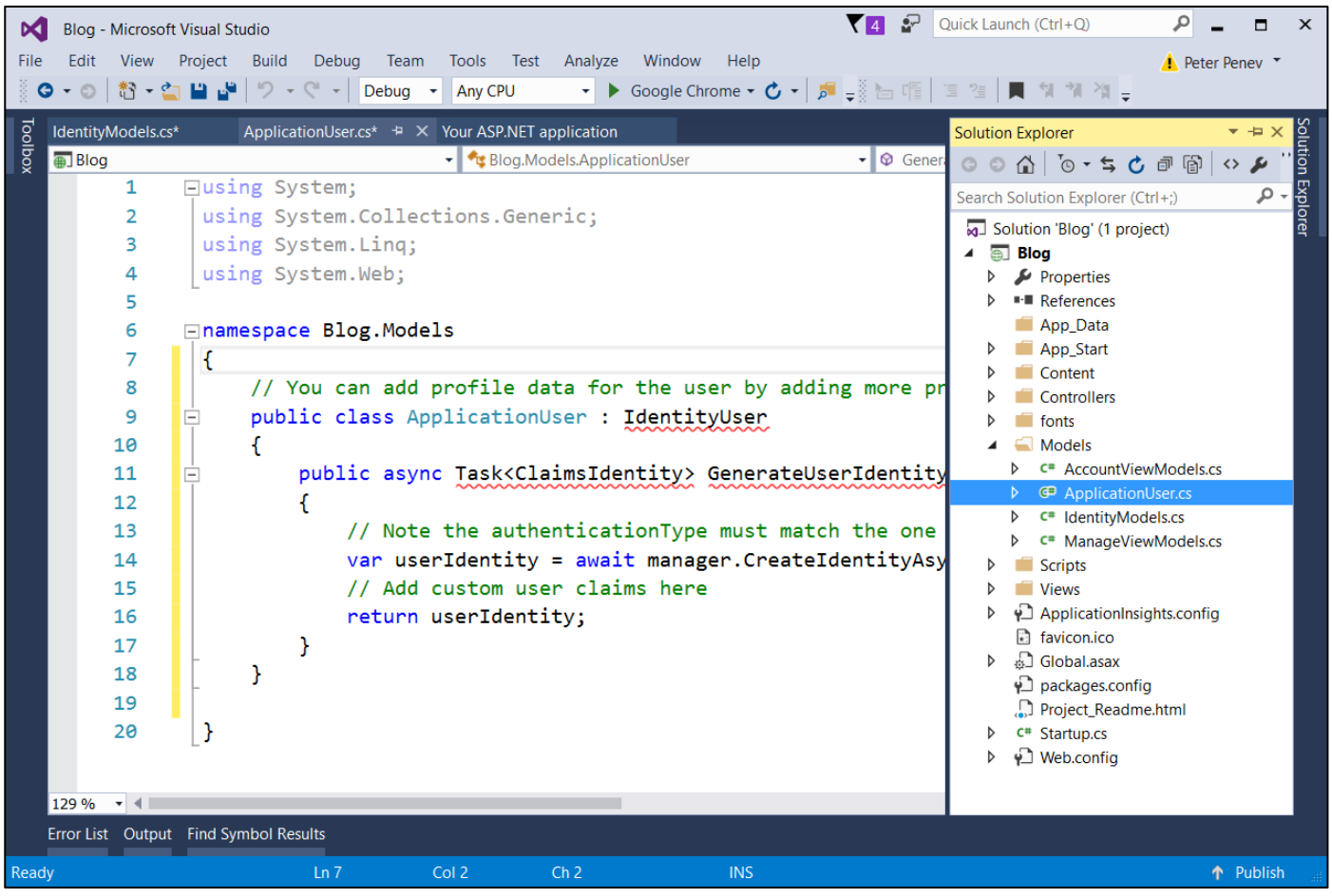


Now, **take all the code** that is the **definition of this class** from "**IdentityModels**" file and paste it inside the new file:

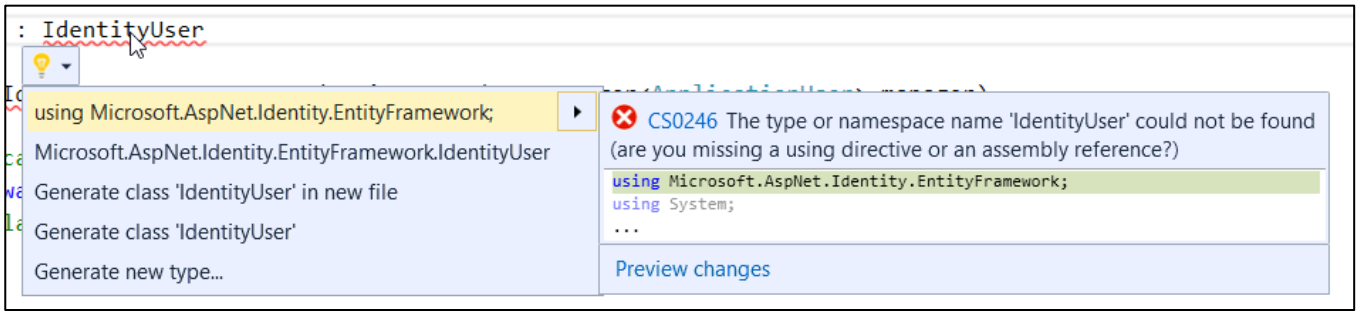




Place it inside the new file (**ApplicationUser**) you have just created:



You can see that there are a lot of red underlining. This is because, inside the new file, **there are no references** to the **libraries that are used**. Just **click on top of an underlined class** (for example **IdentityUser**) and hit **[Ctrl + .]**, this should show a menu:



Choose "using…", hit enter and this should add the specified assembly as a using statement **in the beginning of the file**.

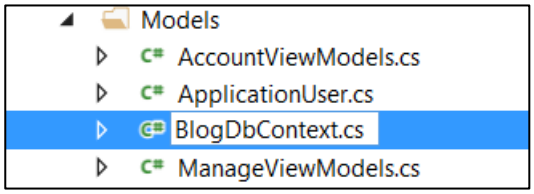
```
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Security.Claims;
using System.Threading.Tasks;
using System.Web;
```

Do this for the rest of the underlined classes or methods **until you can rebuild your solution**.

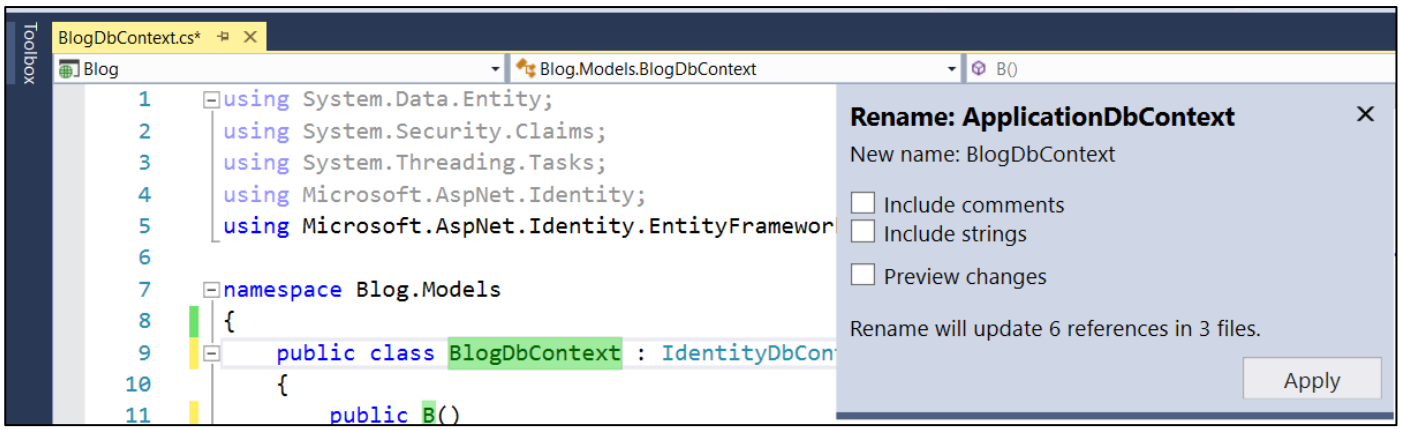
Rebuild with **[Ctrl + Shift + B].**

2. Rename ApplicationDbContext

First, rename the file "**IdentityModels.cs**" to "**BlogDbContext.cs**", because now it holds exactly that, the **blog database context**:



And then **rename the class** (Rename with **[Ctrl + R, R]** after selecting the name of the class):



Rebuild the solution and if there are any errors, fix them by adding the missing libraries.

3. Modify Default Password Requirements

ASP.NET MVC has really complex password requirements. We can change this by editing three separate files:

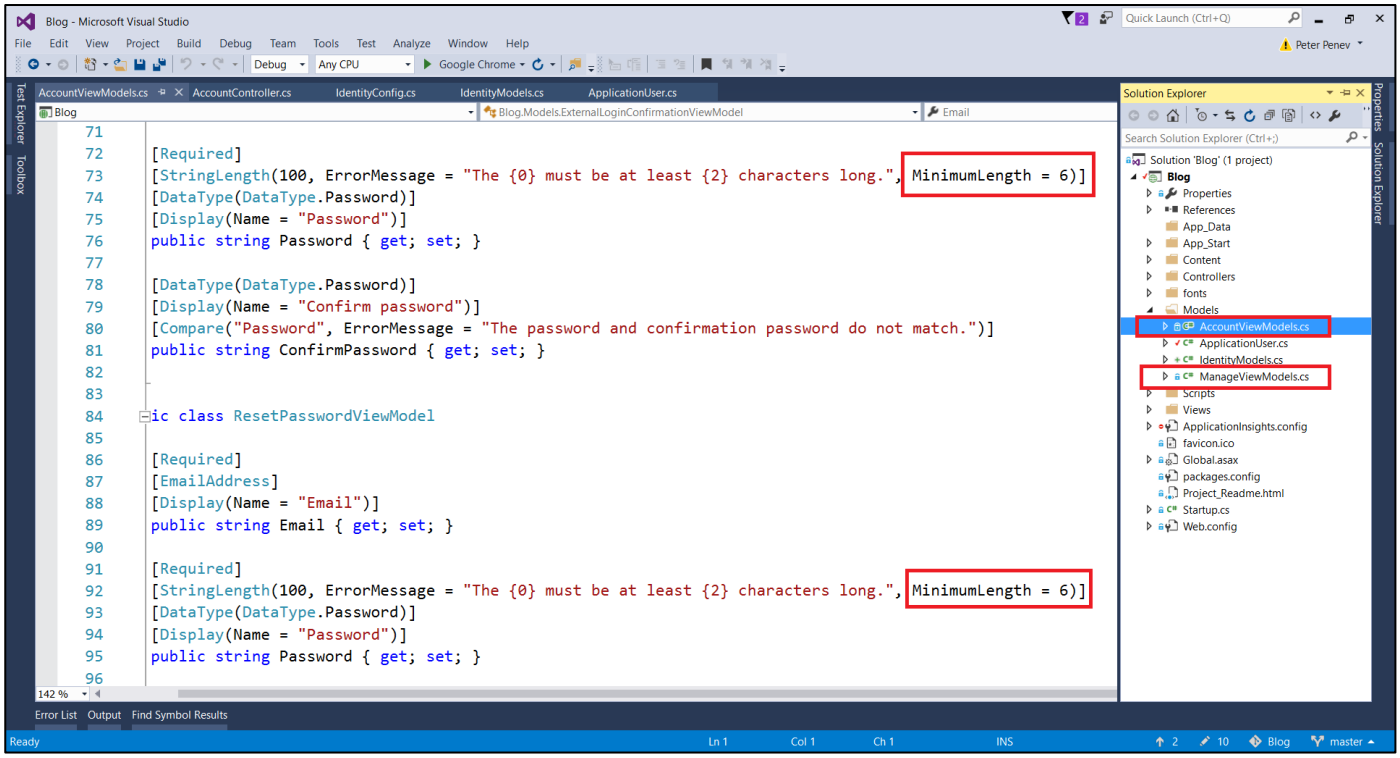
First go to **"App_Start/IdentityConfig.cs"** and look for this code:

```
// Configure validation logic for passwords
manager.PasswordValidator = new PasswordValidator
{
    RequiredLength = 6,
    RequireNonLetterOrDigit = true,
    RequireDigit = true,
    RequireLowercase = true,
    RequireUppercase = true,
};
```

Make the **required length equal to 1** and **the rest equal to false**. This will be much more comfortable for testing.

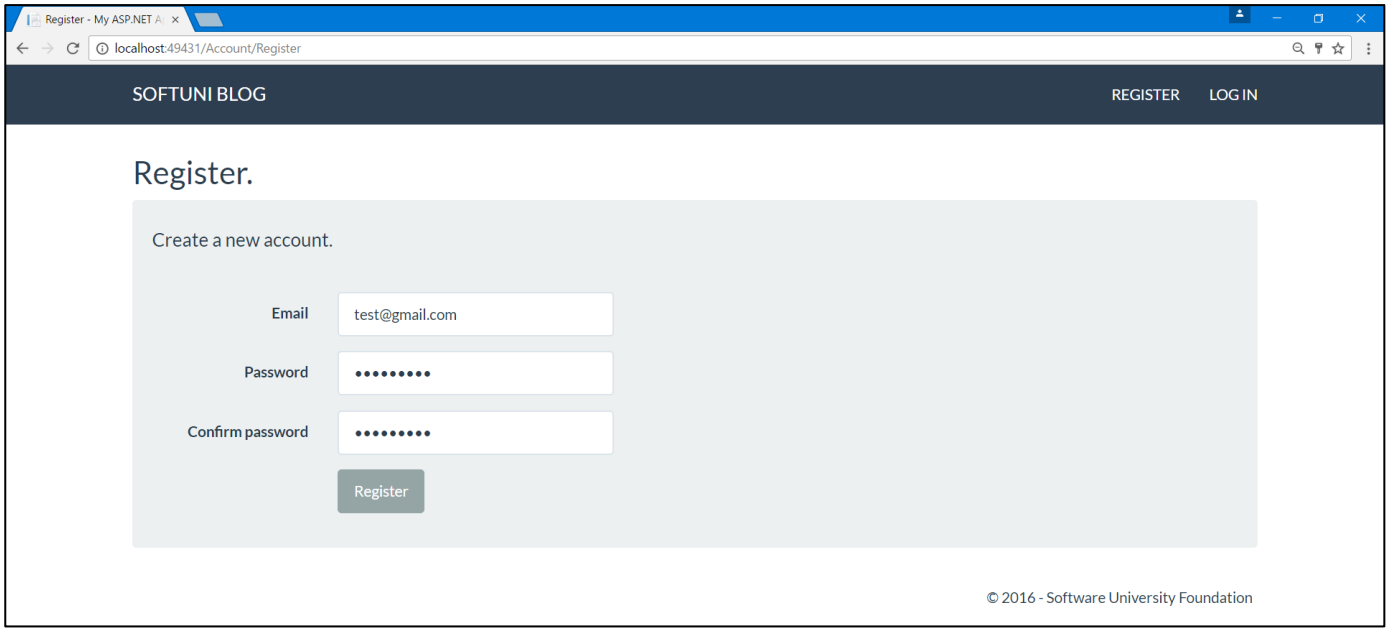
```
// Configure validation logic for passwords
manager.PasswordValidator = new PasswordValidator
{
    RequiredLength = 1,
    RequireNonLetterOrDigit = false,
    RequireDigit = false,
    RequireLowercase = false,
    RequireUppercase = false,
};
```

Then go to "**Models/AccountViewModels.cs**" and "**Models/ManageViewModels.cs**" and find these and **edit them to be equal to 1**:



4. Test Identity

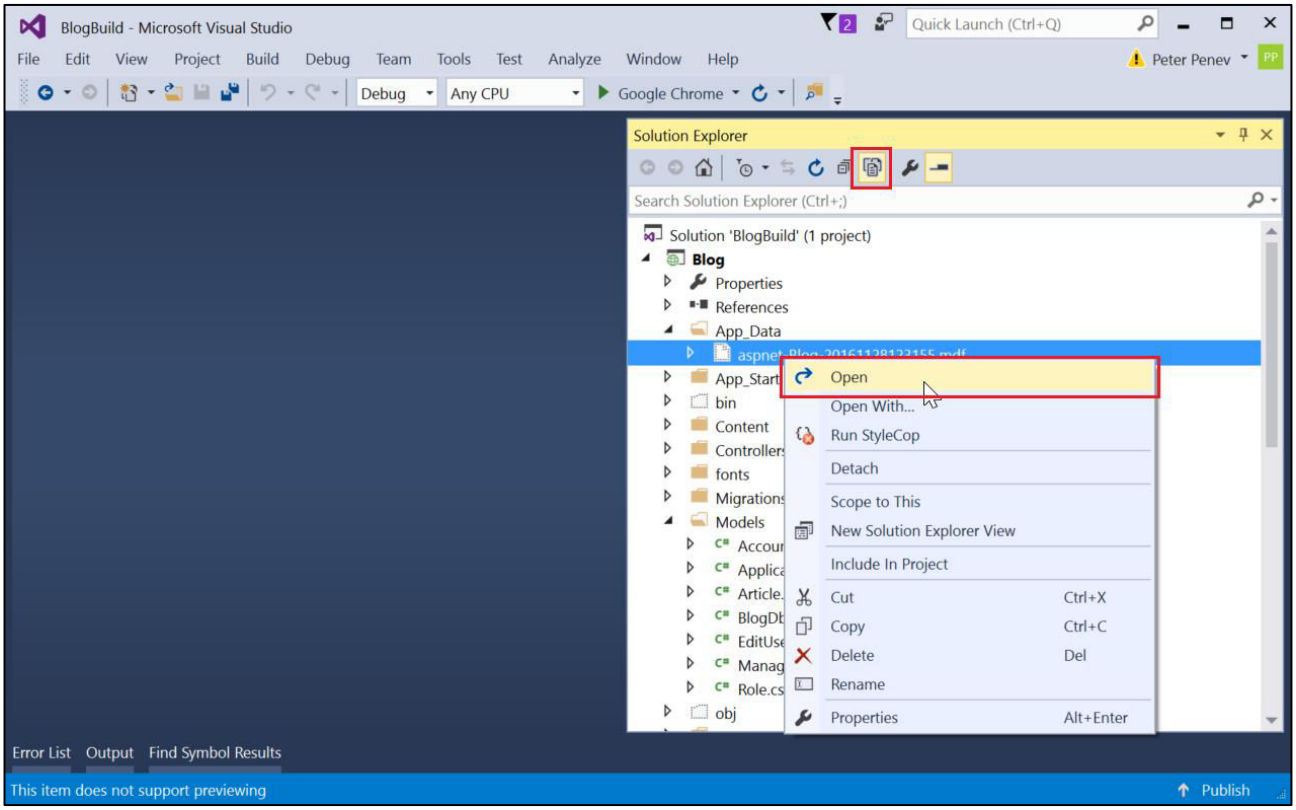
You can test if the authentication module actually works. Just **start the application** and **click on register**. Register a new user and try to log off and to log in again:



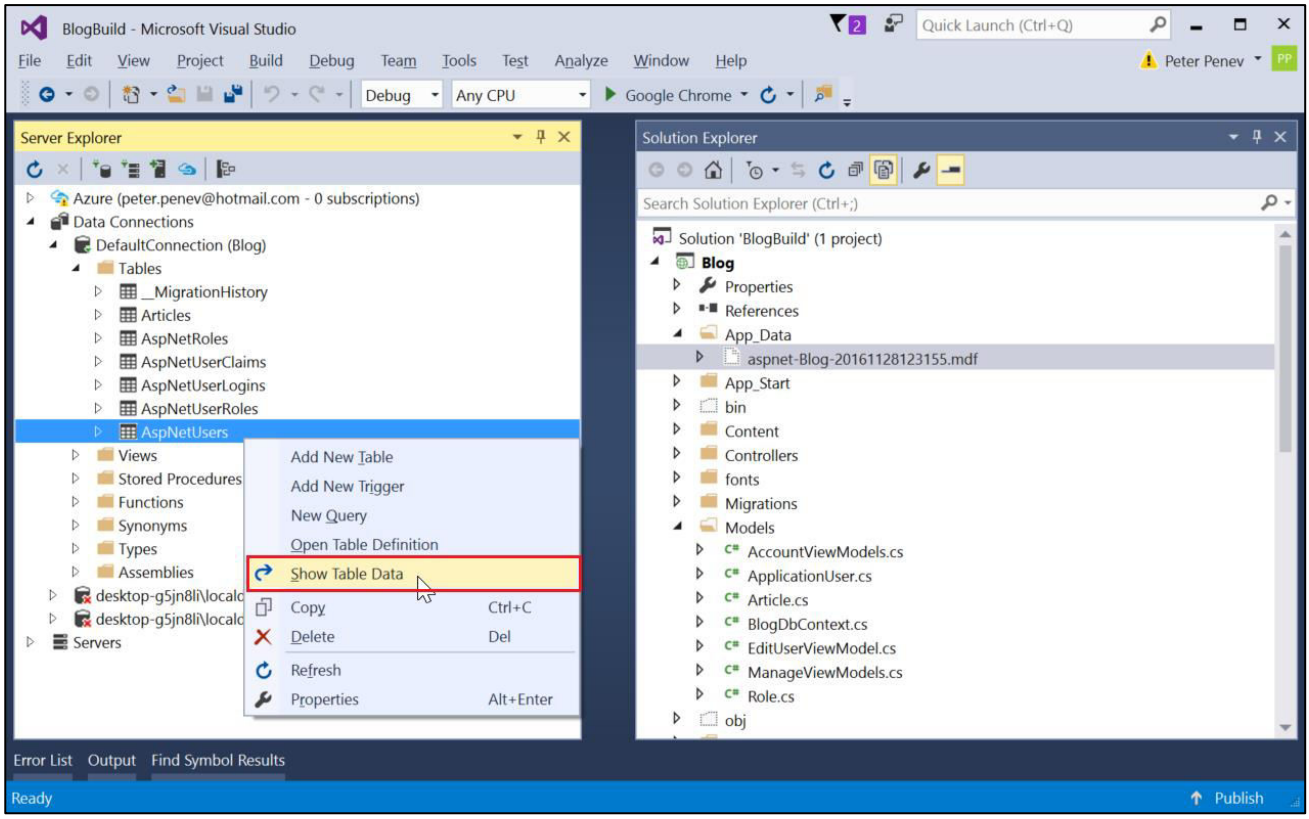
5. Inspect the Database

Open the **automatically-generated database** in the **App_Data** folder and view the **`AspNetUsers`** table:

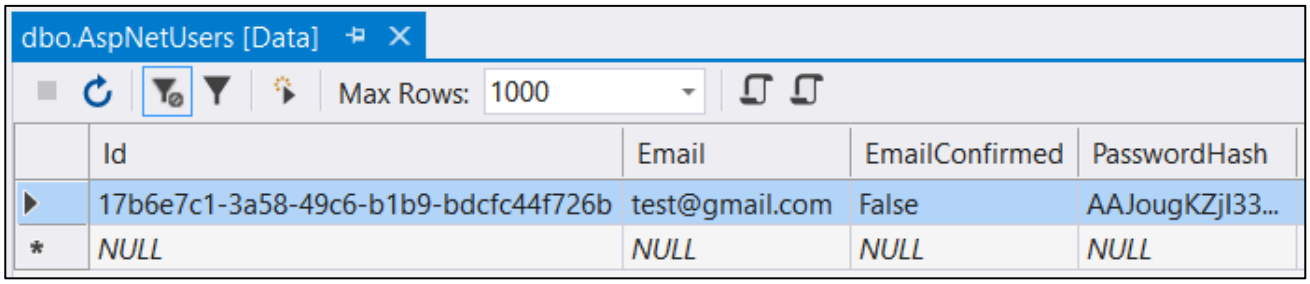
1. Click on the icon in the upper right corner to show all files
2. Double click on the file that now appears in the "**App_Data**" folder. This is your database



3. Right click on the table AspNetUsers and click on Show Table Data:



4. **Inspect the table**. There should be a user with the information you have entered earlier:



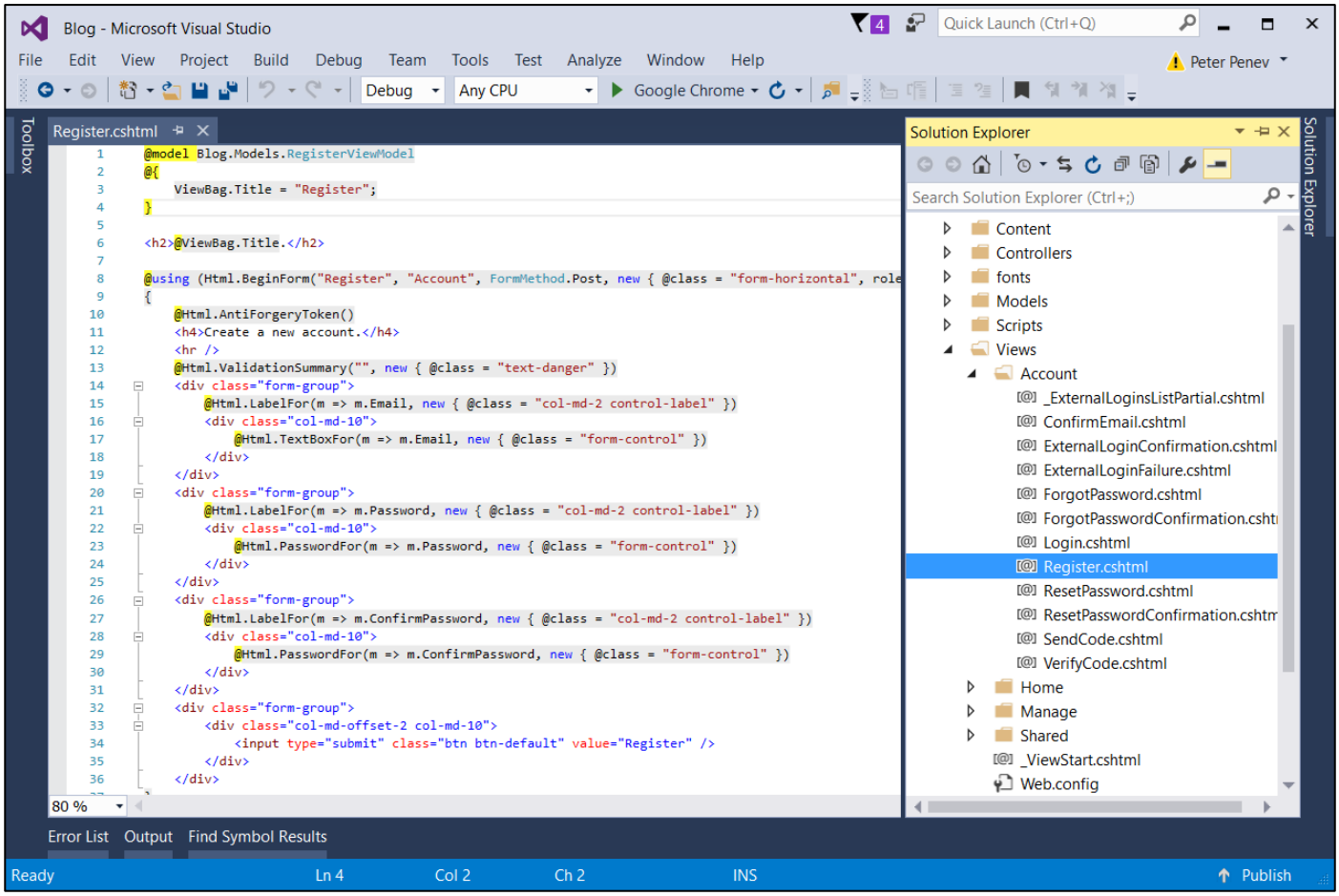
You can see that the **user Id** is a long string of letters and digits. This is a **GUID** or **Globally Universal Id**. Globally universal IDs are guaranteed to be **almost unique in the whole world**.

IV. Get Familiar with Razor Views

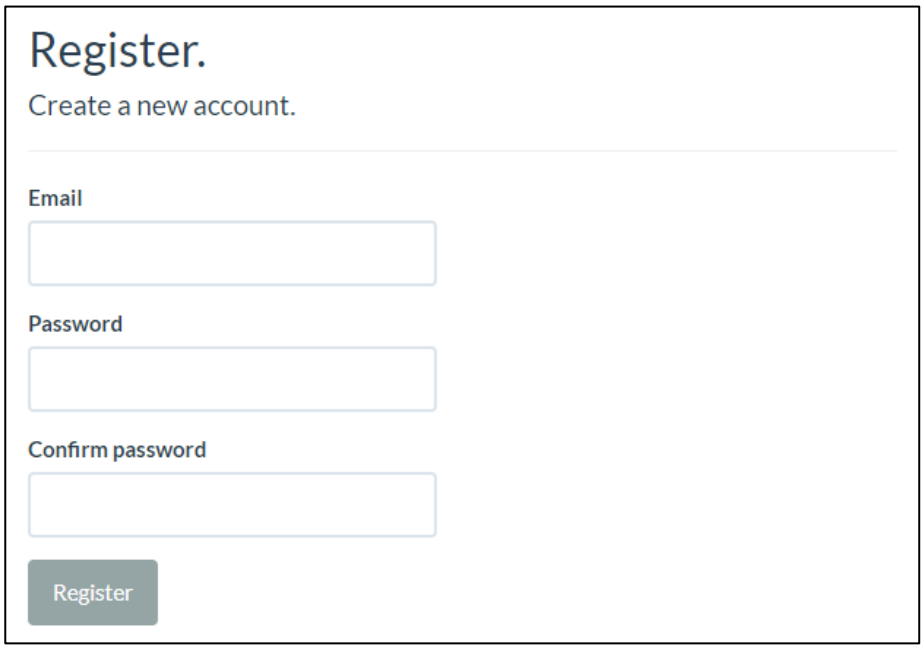
1. Inspect the Register View

Razor is the engine that **handles Views in ASP**. It **mixes HTML language with C#**.

Go to file "**Views/Account/Register**". This is the file that renders the view for Action "**Register**" in the "**AccountController**":



So, the above code renders this:



The first thing to notice is the first line:

```
@model Blog.Models.RegisterViewModel
```

This means that this view works with a class called **RegisterViewModel**, located in the models folder (the view's **View Model**). More on that later.

There is a single **HTML form** in the view (**forms** are used to **gather information** from the user):

```
@using (Html.BeginForm("Register", "Account", FormMethod.Post, new { @class = "form-horizontal", role = "form" }))
{
```

This is a form that is related to the "**Register**" action, in the "**Account**" Controller. It is used to "**Post**" information and has some **classes** for styling.

Inside you can see three **`<div>`**s:

```
<div class="form-group">
    @Html.LabelFor(m => m.Email, new { @class = "col-md-2 control-label" })
    <div class="col-md-10">
        @Html.TextBoxFor(m => m.Email, new { @class = "form-control" })
    </div>
</div>
```

A **form group** represents some **grouped elements inside a form**. There are two elements in it - a **label** and a **text box**. The above div represents this:

The other **`<div>`**s are analogous.

The fourth one has a **button** in it. This button submits the form along the information gathered.

2. Editing a Razor View

Let's make the register page to look and feel familiar to the blogs that we have already made.

Create a **`<div class="container">`** with a **`<div class="well">`** in it. Place the form and the title inside of them (the hole form, with all four divs):

```
1   @model Blog.Models.RegisterViewModel
2   @{
3       ViewBag.Title = "Register";
4   }
5
6   <div class="container">
7       <div class="well">
8           <h2>@ViewBag.Title.</h2>
9
10          @using (Html.BeginForm("Register", "Account", FormMethod.Post, new { @class = "form-horizo
11          {
12              @Html.AntiForgeryToken()
13              <h4>Create a new account.</h4>
14              <hr />
15              @Html.ValidationSummary("", new { @class = "text-danger" })
16              <div class="form-group">
17                  @Html.LabelFor(m => m.Email, new { @class = "col-md-2 control-label" })
18                  <div class="col-md-10">
19                      @Html.TextBoxFor(m => m.Email, new { @class = "form-control" })
20                  </div>
21              </div>
```

Now go through the view and make all **`"col-md-2"`** or whatever number they have in all of the divs, to be **`"col-sm-4"`**:

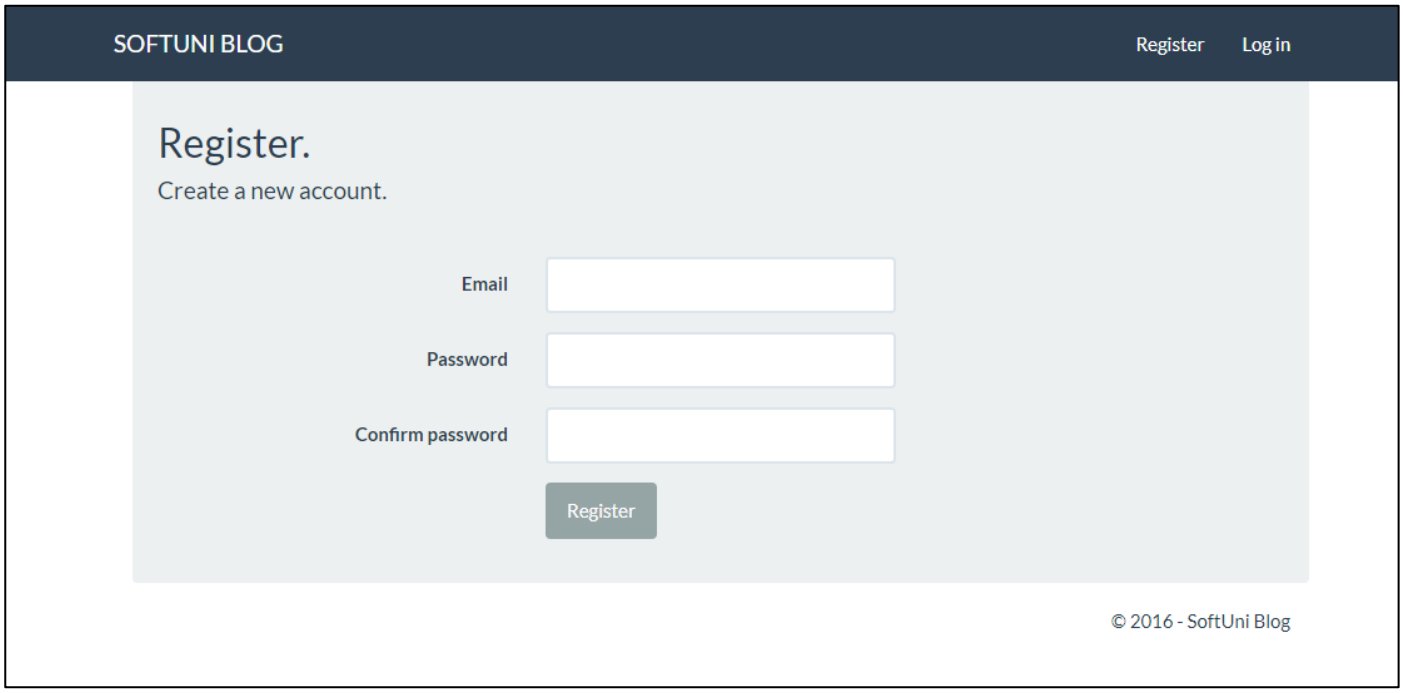
```
<div class="form-group">
    @Html.LabelFor(m => m.Email, new { @class = "col-sm-4 control-label" })
    <div class="col-sm-4">
        @Html.TextBoxFor(m => m.Email, new { @class = "form-control" })
    </div>
</div>
```

Also, make sure to correct the offset for the buttons:

```
<div class="form-group">
    <div class="col-sm-offset-4 col-sm-4">
        <input type="submit" class="btn btn-default" value="Register" />
    </div>
</div>
```

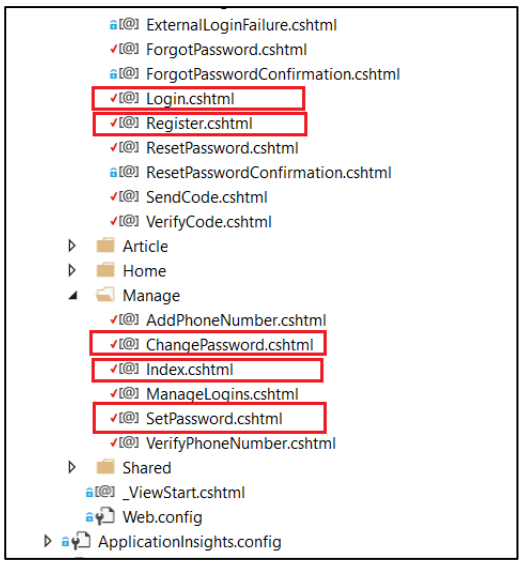
This should make **every label** and **text box** to **take 4 columns of the grid** when the window is of size small and above.

So, the result should be this:



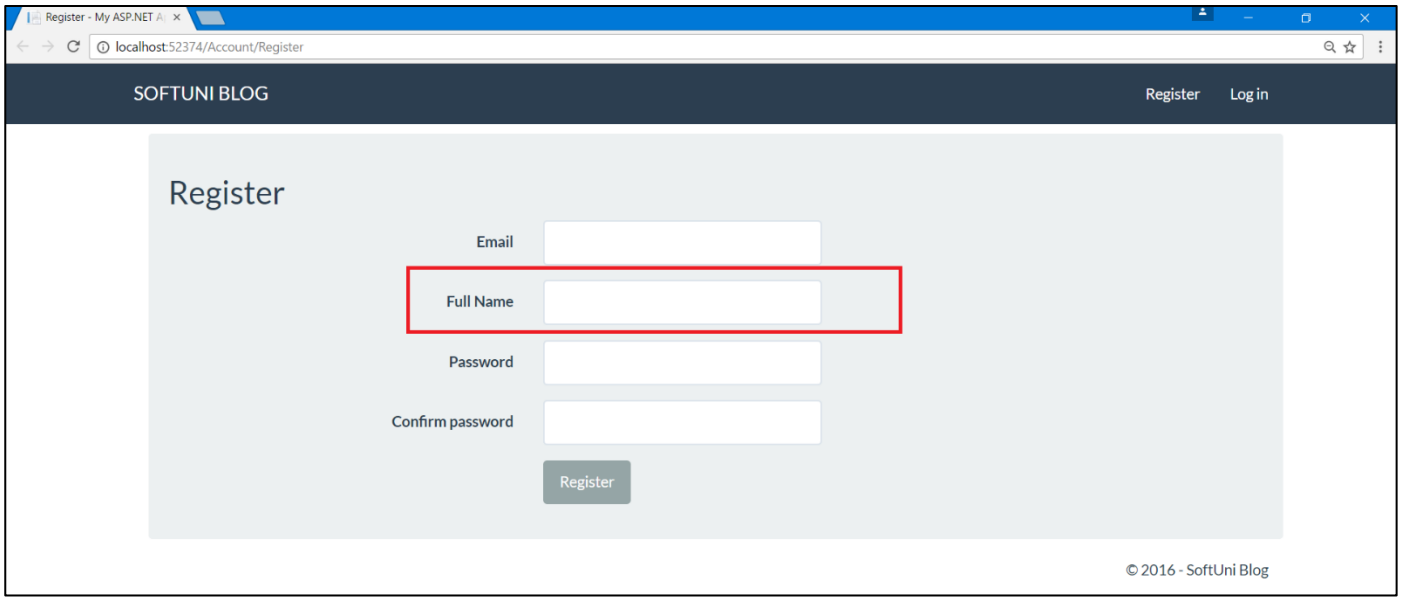
3. Editing the Rest of the Views

Now we won't need all of the views that come with **ASP.NET** but we need some of them so let's edit them to have the same layout. Go through all views that you think you will need and edit them, following the same steps as above:



V. Start Working with Project Models

There is a problem with the application user. The model has no full name. We need to add this:



In order to have a full name saved for every user in the database we need to:

1. Edit the **view**
2. Edit the **model**
3. Edit the **controller**

As we recently worked with the views, let's start from there.

1. Edit the Register View

Go back to "**Views/Account/Register.cshtml**". Remember that there were some divs **representing every element on the register page** like this one:

```
<div class="form-group">
    @Html.LabelFor(m => m.Email, new { @class = "col-sm-4 control-label" })
    <div class="col-sm-4">
        @Html.TextBoxFor(m => m.Email, new { @class = "form-control" })
    </div>
</div>
```

We need to **add additional one** for the full name. Insert one **between** the **email** and the **password**:

```
<div class="form-group">
    @Html.LabelFor(m => m.FullName, new { @class = "col-sm-4 control-label" })
    <div class="col-sm-4">
        @Html.TextBoxFor(m => m.FullName, new { @class = "form-control" })
    </div>
</div>
```

But there is a problem, our model has no **property** for "**FullName**". That's why you don't get any autocompletion and "**FullName**" has red underlining.

2. Edit the Register View Model

There are **two kinds of models** in ASP.NET. **Data models** and **View models**. In short, **data models are meant to interact with the controller** and **view models with the views**.

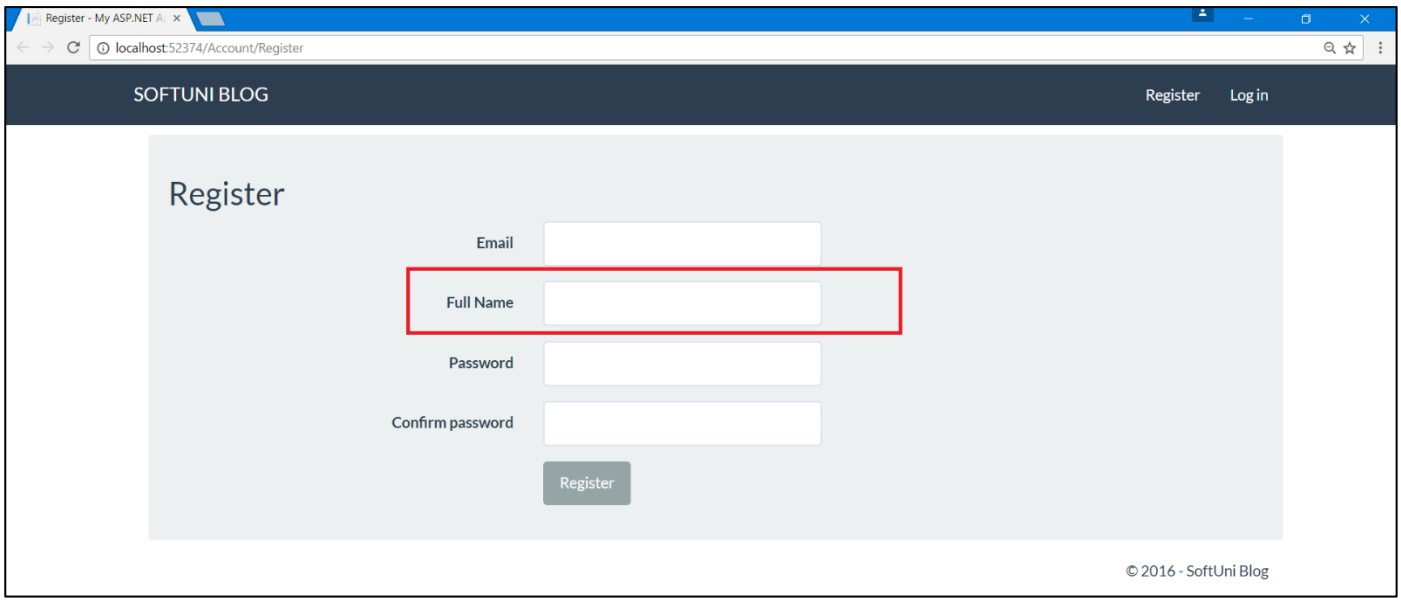
Go to "**Models/AccountViewModels.cs**" and search for **RegisterViewModel**. This is the model that our view works with.

Add the following lines of code:

```
public class RegisterViewModel
{
    [Required]
    [EmailAddress]
    [Display(Name = "Email")]
    public string Email { get; set; }

    [Required]
    [StringLength(50)]
    [Display(Name = "Full Name")]
    public string FullName { get; set; }

    [Required]
    [StringLength(100, ErrorMessage = "The {0} must be at least {2} characters long.", MinimumLength = 1)]
    [DataType(DataType.Password)]
    [Display(Name = "Password")]
    public string Password { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "Confirm password")]
    [Compare("Password", ErrorMessage = "The password and confirmation password do not match.")]
    public string ConfirmPassword { get; set; }
}
```

This will **add a new property** called **FullName**, that is required and its maximum length is 50.

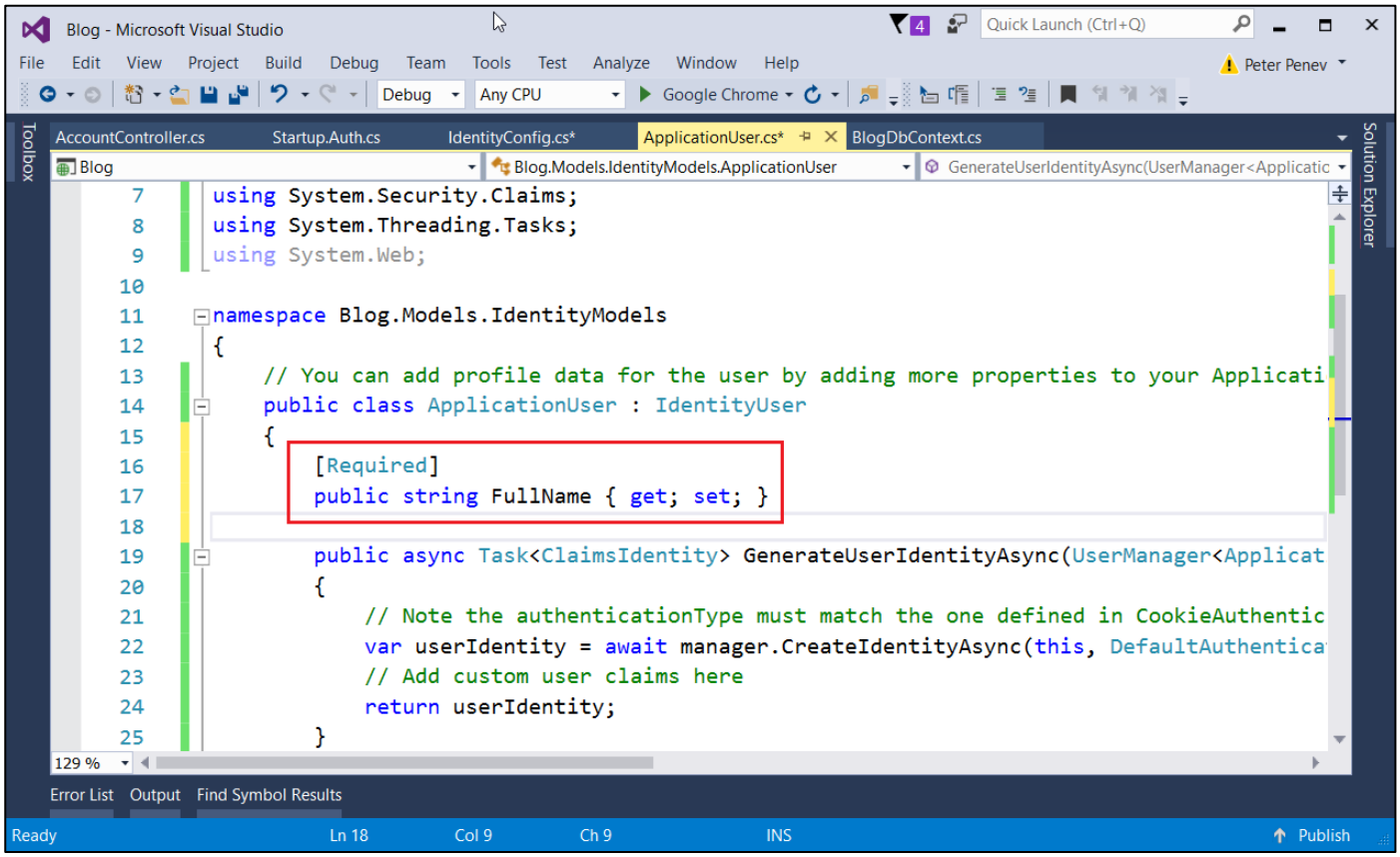
Now you can **compile the project** and see that you have a **new textbox on the register view**:



But **it is not functional**. It doesn't save data in the database.

3. Edit the Application User

Head to the **"Models/ApplicationUser.cs"** and **add a string property called FullName**:



The **property should be public**. Include an attribute [**Required**]. In this case the attribute is **used for validation** that makes sure that a user without full name **cannot be inserted into the database**.

Don't forget to **add missing libraries** for the attribute.

4. Edit Register Action

The only thing left to do is to set the full name when the user is registered.

Go to "**Controllers/AccountController.cs**" and find the **`Register`** method.

Add the following code:

```
//
// POST: /Account/Register
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = model.Email, FullName = model.FullName, Email = model.Email };
        var result = await UserManager.CreateAsync(user, model.Password);

        if (result.Succeeded)
        {
            await SignInManager.SignInAsync(user, isPersistent:false, rememberBrowser:false);
```

5. Add Database Migrations

Now, if you try to register a new user, you should **get an error**:

Server Error in '/' Application.

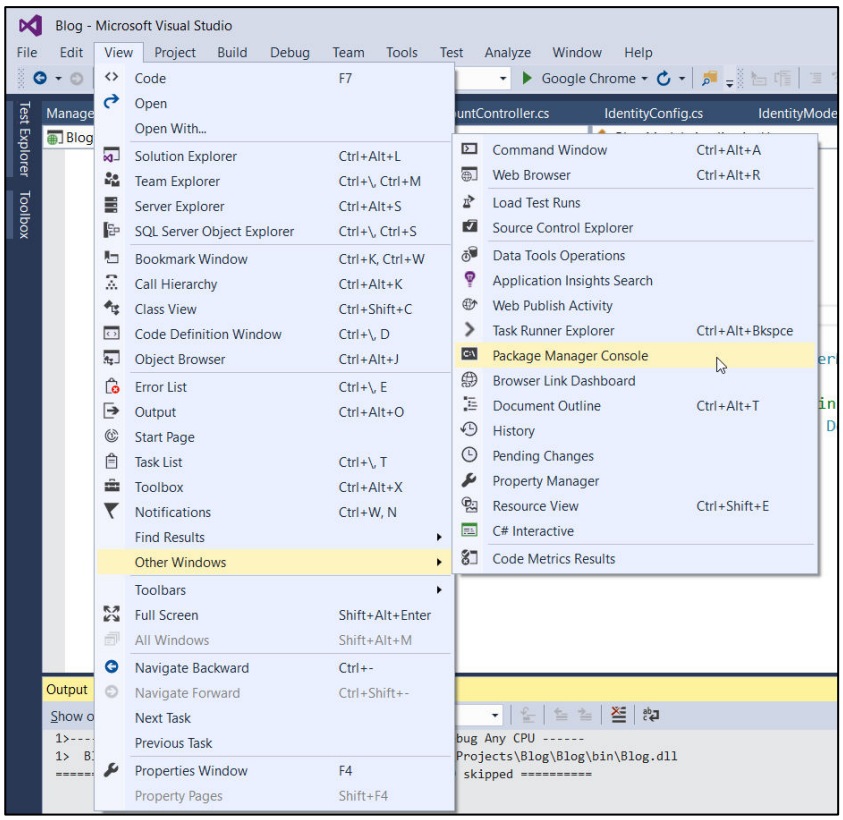
The model backing the 'ApplicationDbContext' context has changed since the database was created. Consider using Code First Migrations to update the database (http://go.microsoft.com/fwlink/?LinkId=238269).

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.InvalidOperationException: The model backing the 'ApplicationDbContext' context has changed since the database was created. Consider using Code First Migrations to update the database (http://go.microsoft.com/fwlink/?LinkId=238269).

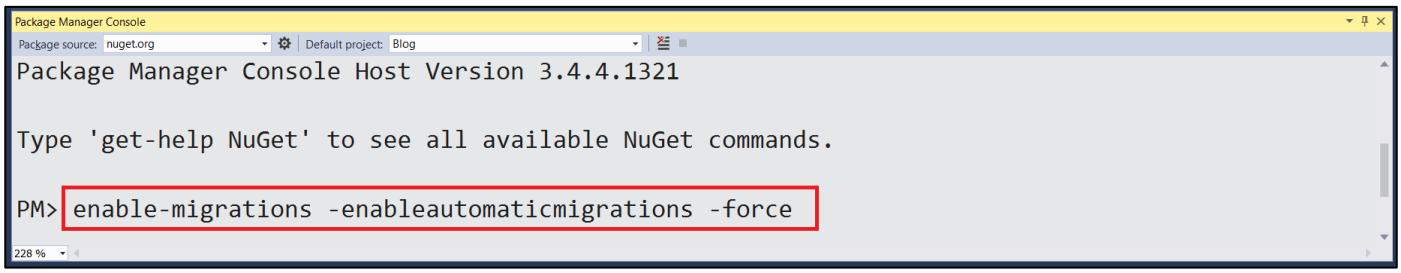
This error tells us that the database has changed. We need to explicitly state that we want the **database to be modified if any changes are made** to the entities.

First, open the package manager. In the top left corner of the screen click "**View -> Other Windows -> Package Manager Console**":



Type:

```
Enable-Migrations -EnableAutomaticMigrations -Force
```

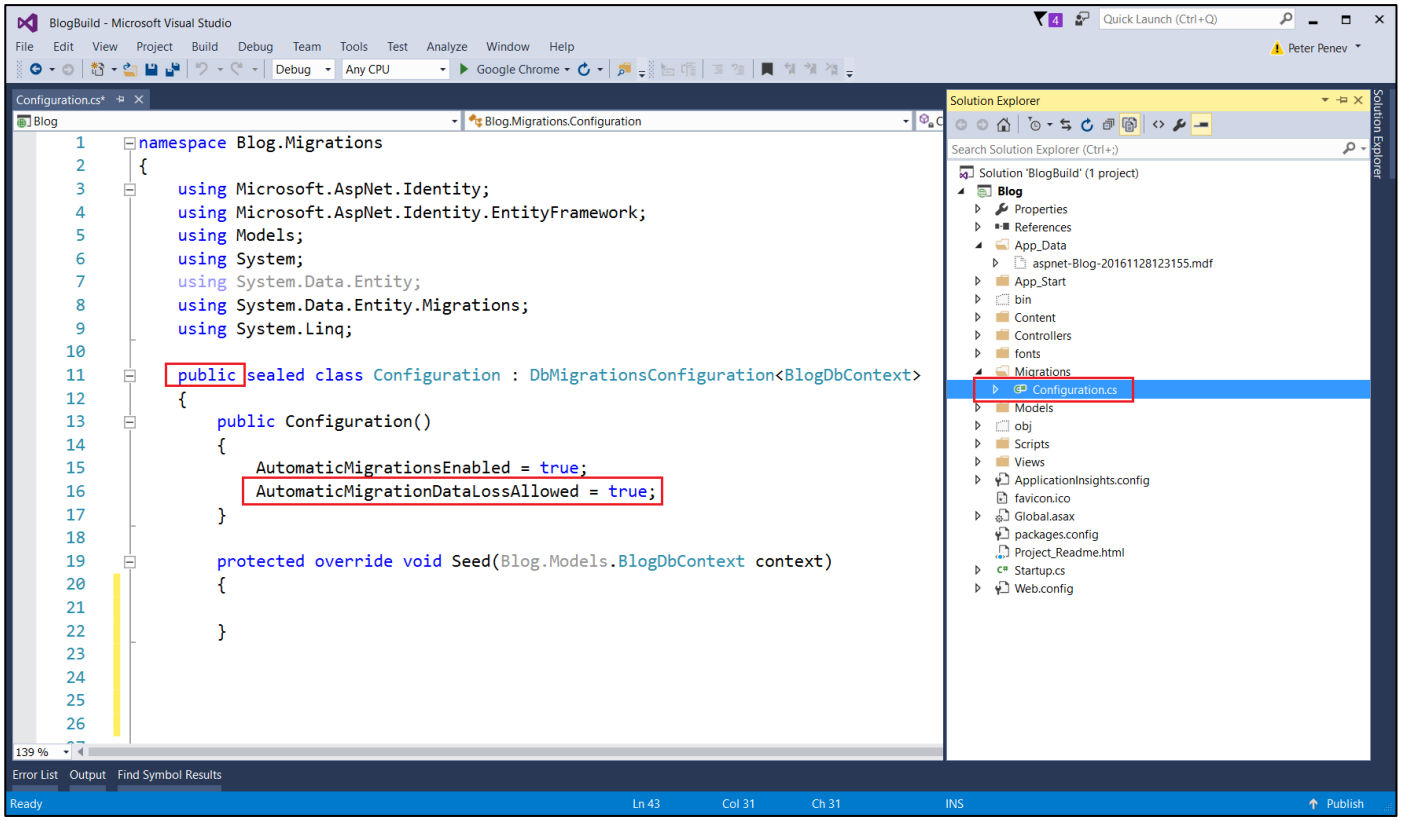


Next, go to the newly created file "**Migrations/Configurations.cs**".

1. Make the class public
2. Add:

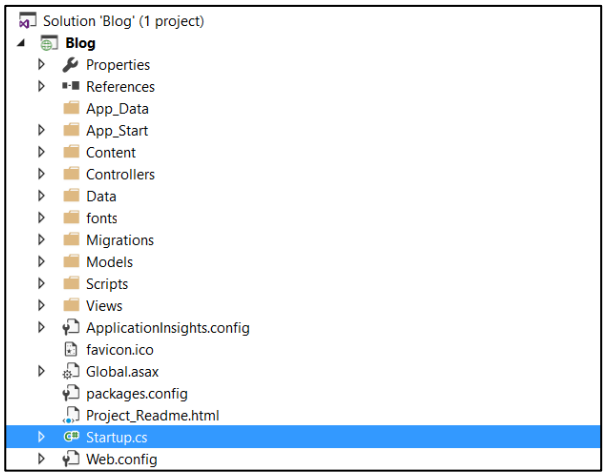
```
AutomaticMigrationDataLossAllowed = true;
```

3. Don't forget to turn off the above option, once you have sensitive information in the database ☺

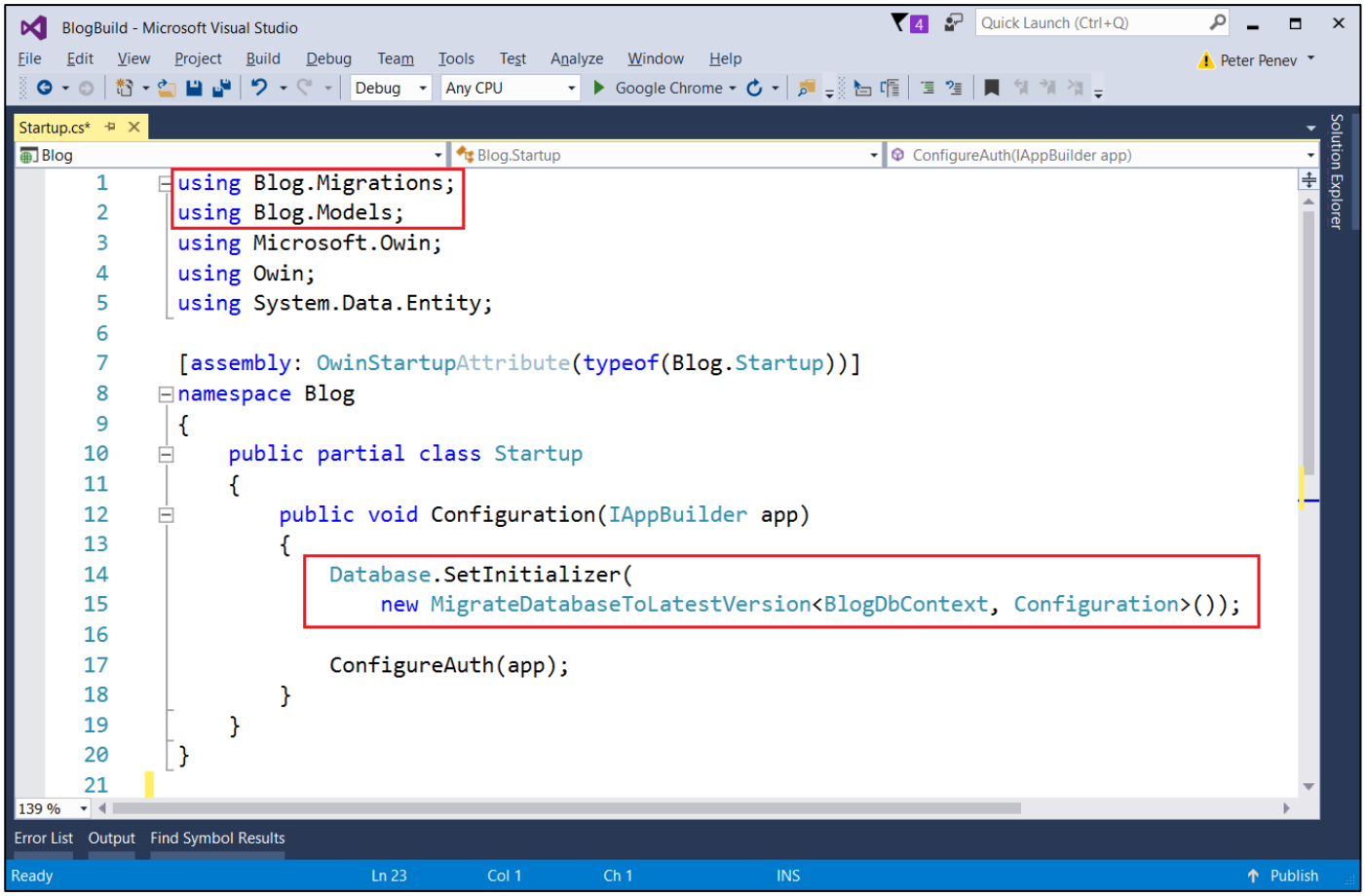



```
namespace Blog.Migrations
{
    using Microsoft.AspNet.Identity;
    using Microsoft.AspNet.Identity.EntityFramework;
    using Models;
    using System;
    using System.Data.Entity;
    using System.Data.Entity.Migrations;
    using System.Linq;

    public sealed class Configuration : DbMigrationsConfiguration<BlogDbContext>
    {
        public Configuration()
        {
            AutomaticMigrationsEnabled = true;
            AutomaticMigrationDataLossAllowed = true;
        }

        protected override void Seed(Blog.Models.BlogDbContext context)
        {

        }
```

The last thing that you need to do is head to "**Startup.cs**" (it is in the root directory):



And add the following lines:



Don't forget to include the new usings with **[Ctrl + .]**

Make sure that you can **start the app**, **register a new user** and the User entity in the database has a full name column.

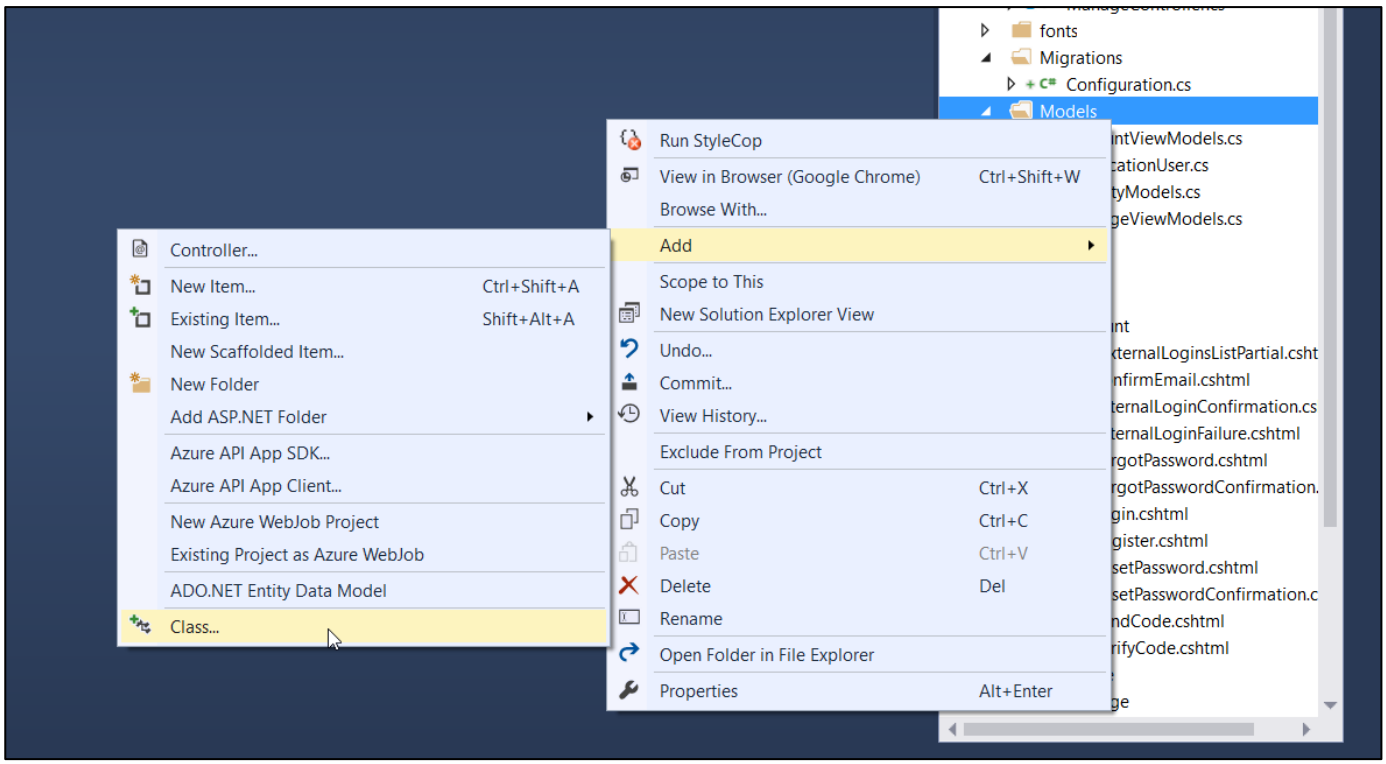
VI. Creating Article Entities

This is the moment we can actually **create something**. Let's start with articles.

For the articles, we will need a **controller** and a **model**. In the controller, we will have action for **listing all articles**, action for **creating an article**, action for **displaying a single article** and actions for **editing** and **deleting articles**. For each action, we need a **view**.

1. Creating the Article Model

This is a class that will **hold information about a single article** and will be **saved in the database** (e.g. an **entity**). So, **create a new class** in the "**Models**" folder and name it "**Article**":



Add a public int property "**Id**" with an attribute [**Key**]:

```
using System;
using System.ComponentModel.DataAnnotations;

namespace Blog.Models
{
    public class Article
    {
        [Key]
        public int Id { get; set; }
    }
}
```

The attribute [**Key**] specifies that this is the primary key and is used to **identify** a record in the table (think of it as an **id**).

Now we need the other properties of an article: **Title**, **Content**, **Date Added** and **Author**.

The **title** will be with a maximum length of 255 characters and will be **required**, so again, we can use an attribute to specify that:

```
[Required]
[StringLength(50)]
public string Title { get; set; }
```

The **content** won't have any restriction to length and it can be an **empty string**, so:

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

And we need an **author id**, which will be of type **string**:

```
[ForeignKey("Author")]
public string AuthorId { get; set; }
```

An author for the article, which will be of type **`ApplicationUser`**:

```
public virtual ApplicationUser Author { get; set; }
```

You should end up with something like this:

```
public class Article
{
    [Key]
    public int Id { get; set; }

    [Required]
    [StringLength(50)]
    public string Title { get; set; }

    [Required]
    public string Content { get; set; }

    [ForeignKey("Author")]
    public string AuthorId { get; set; }

    public virtual ApplicationUser Author { get; set; }
}
```

2. Inserting Articles Manually

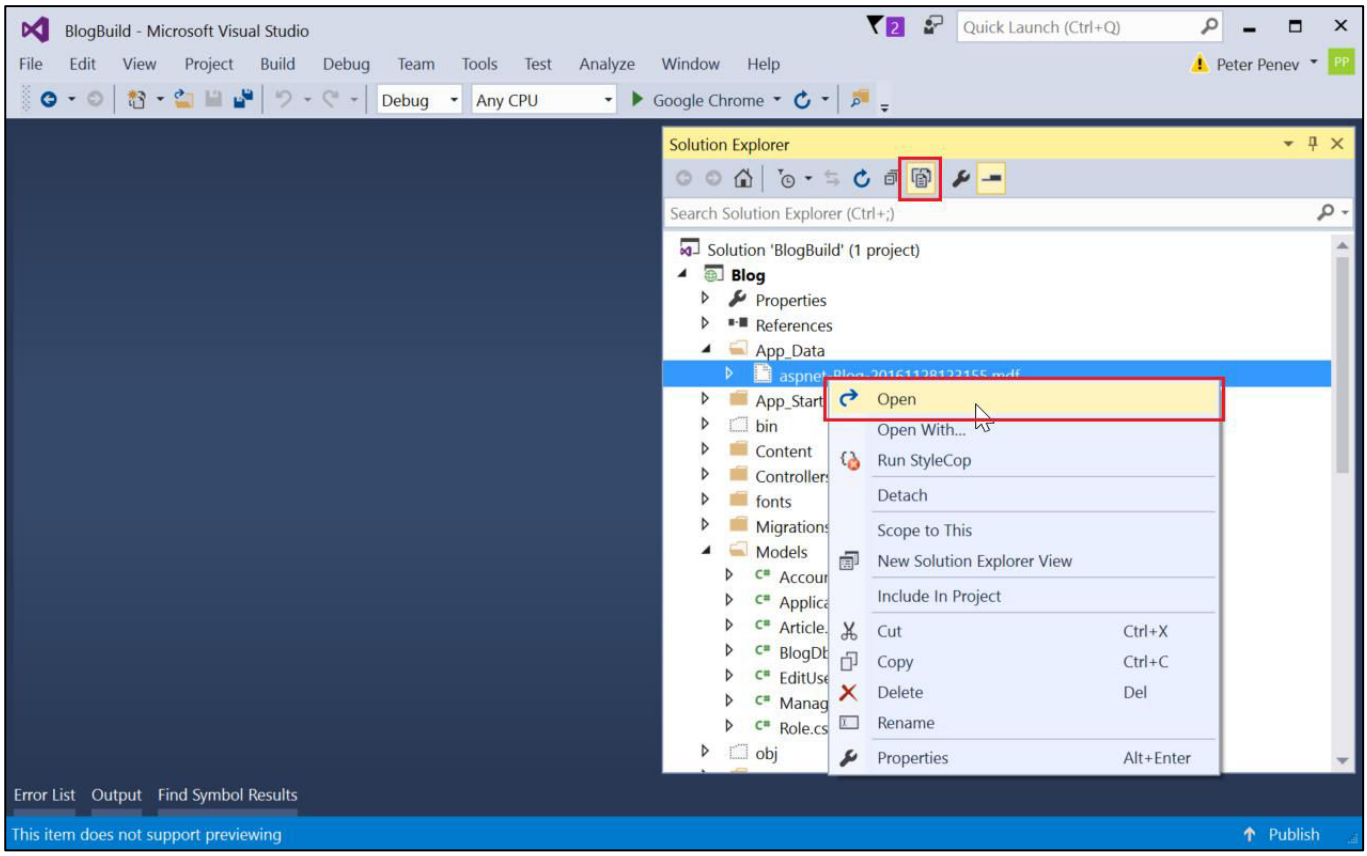
We want to test if the **entity is created properly** in the database. In order to check this, the database should know about articles. We can do this in the file "**`Models/BlogDbContext.cs`**".

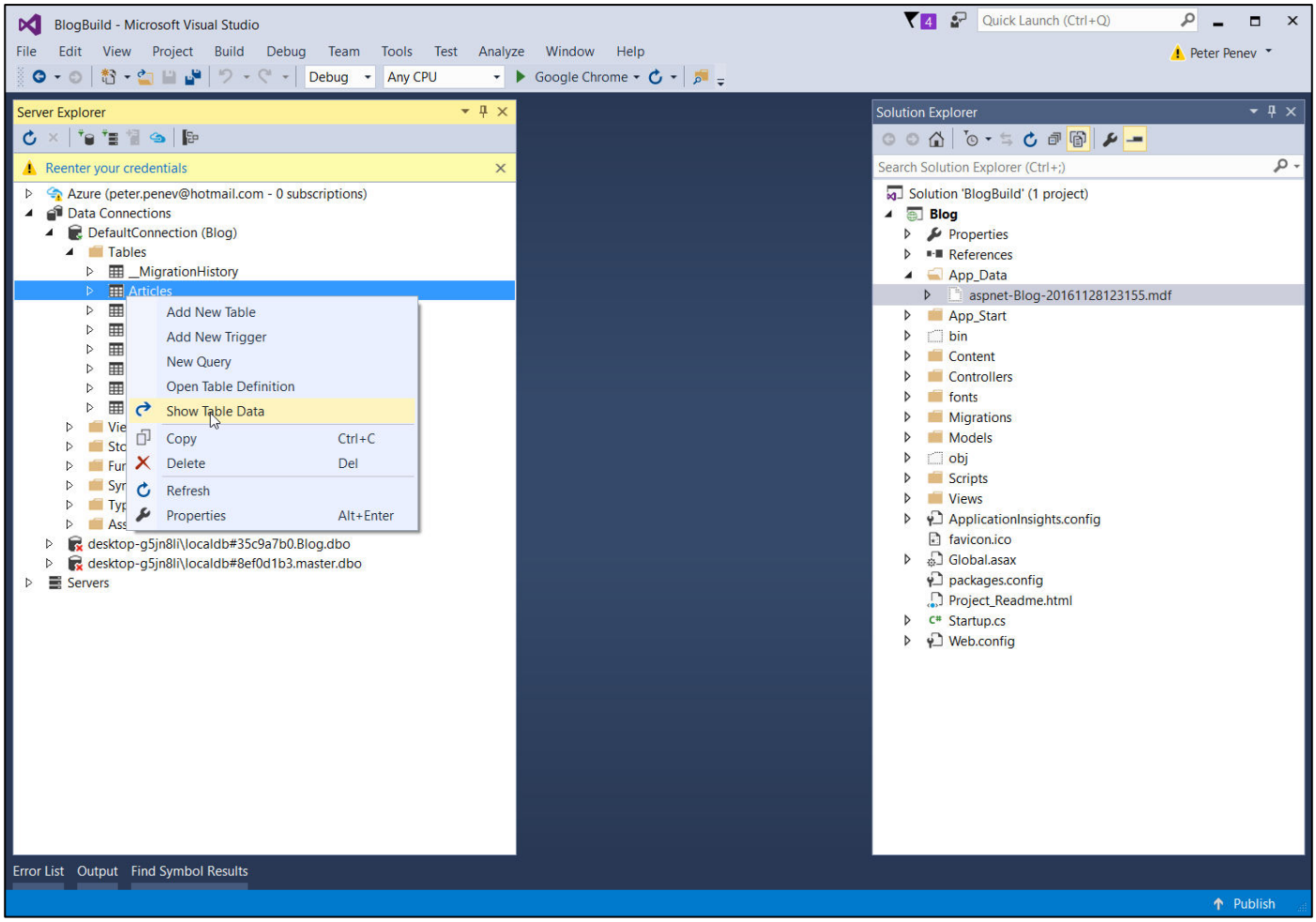
The **database context** is the layer in our application that **communicates with the database**.

Just create a new property that is of type **`public virtual IDbSet<Article>`** and name it **`Articles`**:

```
namespace BlogSystem.Data
{
    using Microsoft.AspNet.Identity.EntityFramework;
    using Models.DataModels;
    using Models.Identity;
    using System.Data.Entity;
    public class BlogDbContext : IdentityDbContext<ApplicationUser>
    {
        public BlogDbContext()
            : base("DefaultConnection", throwIfV1Schema: false)
        {
        }

        public virtual IDbSet<Article> Articles { get; set; }

        public static BlogDbContext Create()
        {
            return new BlogDbContext();
        }
    }
}
```

Build the application and then do something involving **modifying the database**, like registering a new user. After that you can see that in there is a new table called Articles:





	Id	Title	Content	DateAdded	Author_Id
▶*	NULL	NULL	NULL	NULL	NULL

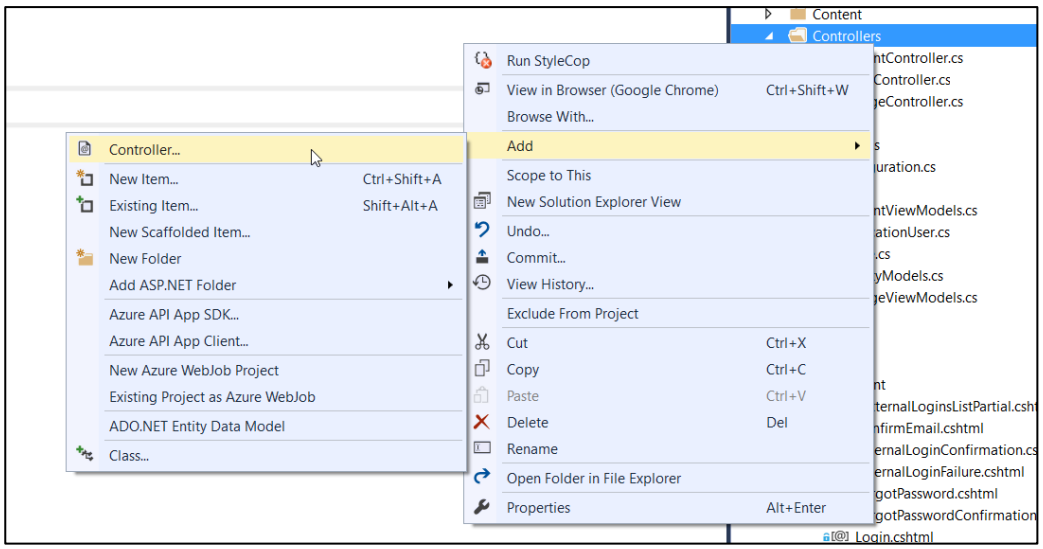
So, try to **insert some articles**. Just click on a field and **type some information in**. Make sure you copy and paste the **GUID** of some existing user, otherwise it won't let you to insert an article with an invalid or missing author.

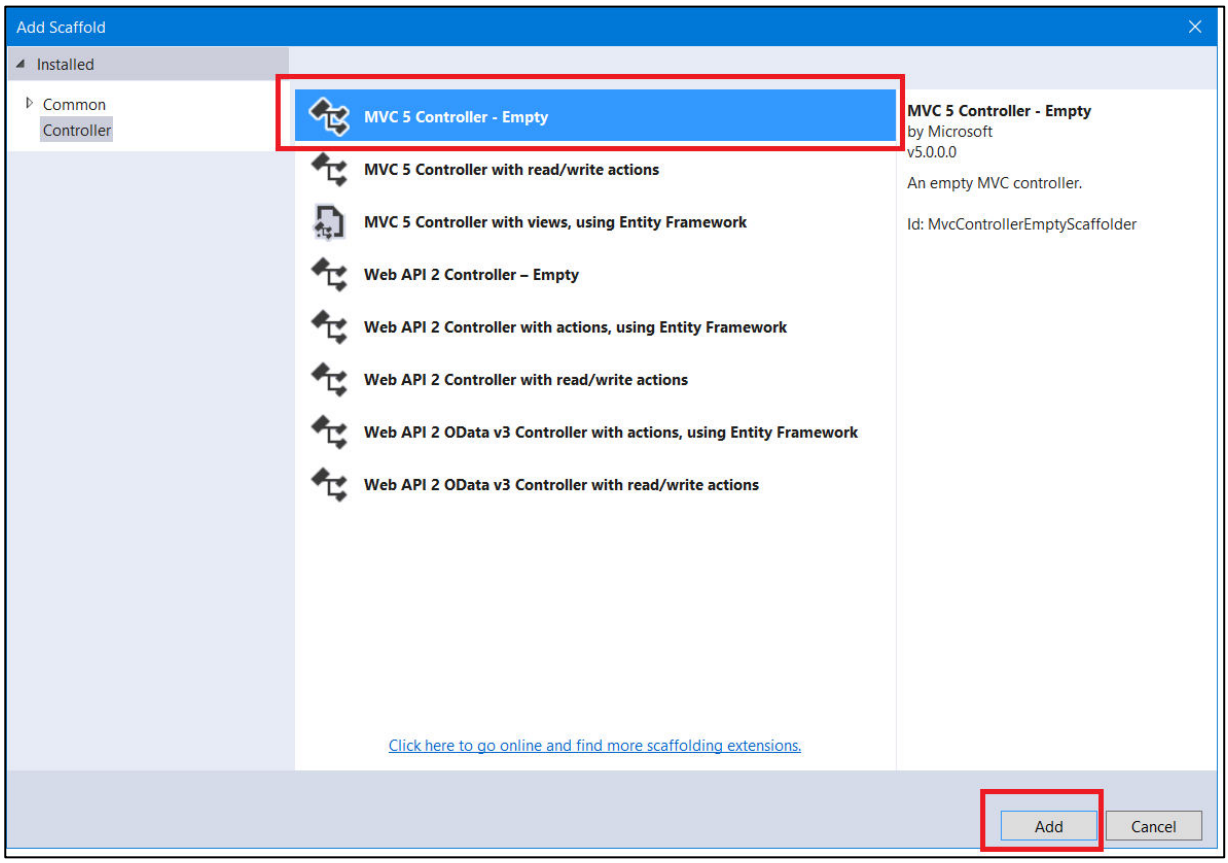
	Id	Title	Content	DateAdded	Author_Id
	1	Article	Content	01/01/2016 ...	214f0e27-2c7d-4982-a624-6b91a5ec5e55
▶*	NULL	NULL	NULL	NULL	NULL

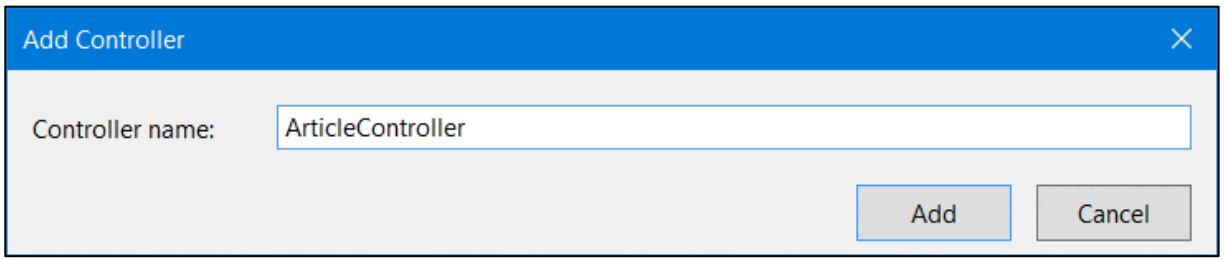
The only problem is that we don't have a place in the blog where we see the articles. For this we will need an **Article Controller**.

3. Creating the Article Controller

Head to the "**Controllers**" folder and create a new **Controller**:







Create a new **method** (**Action**) in it called "**List**" and make the **Index()** action to redirect to it:

```
public class ArticleController : Controller
{
    //
    // GET: Article
    public ActionResult Index()
    {
        return RedirectToAction("List");
    }

    //
    // GET: Article/List
    public ActionResult List()
    {
        return View();
    }
}
```

So, when the **Index()** action is called, it will automatically redirect to **List()**.

VII. List Articles

1. Creating the Get Method

Now, head to the **List()** action:

```
//
// GET: Article/List
public ActionResult List()
{
    return View();
}
```

We need to **get all articles** from the database.

To do this, we need an instance of the class **BlogDbContext** and we need to **dispose** it after we are done with it, so we are going to place it inside a using block:

```
//
// GET: Article/List
public ActionResult List()
{
    using (var database = new BlogDbContext())
    {
        // Get articles from database

        return View();
    }
}
```

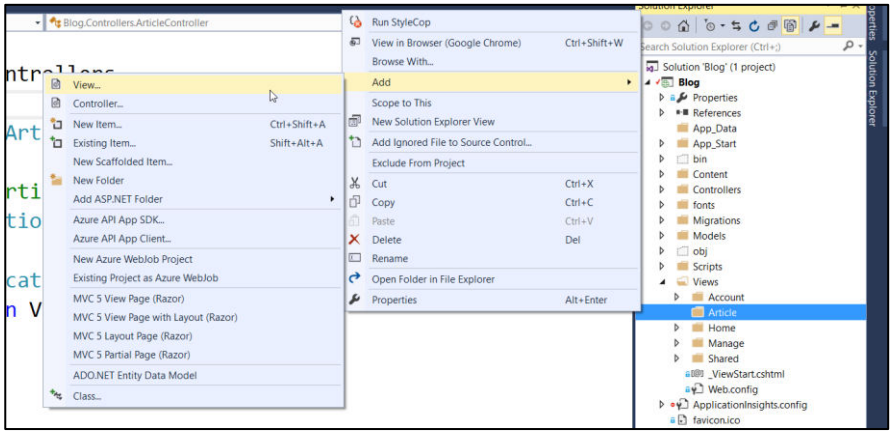
Now, using **LINQ**, we can **get all articles** and then **pass them to the view**:

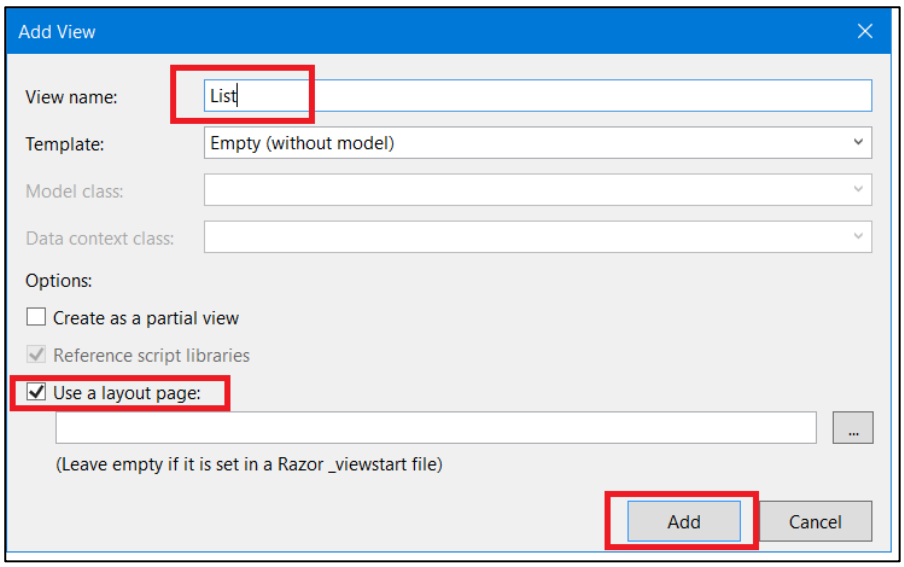
```
//
// GET: Article/List
public ActionResult List()
{
    using (var database = new BlogDbContext())
    {
        // Get articles from database
        var articles = database.Articles
            .Include(a => a.Author)
            .ToList();

        return View(articles);
    }
}
```

Notice the "**.Include()**" statement. What it does is when we **materialize** the articles from the database, we need to **include information about their authors**. If we didn't do that, the author would be left **null** and if we try to get information about the author inside the view, an **exception will be thrown**.

2. Creating the View

Create a new view inside the "**Views/Article**":





Paste this code in the view:

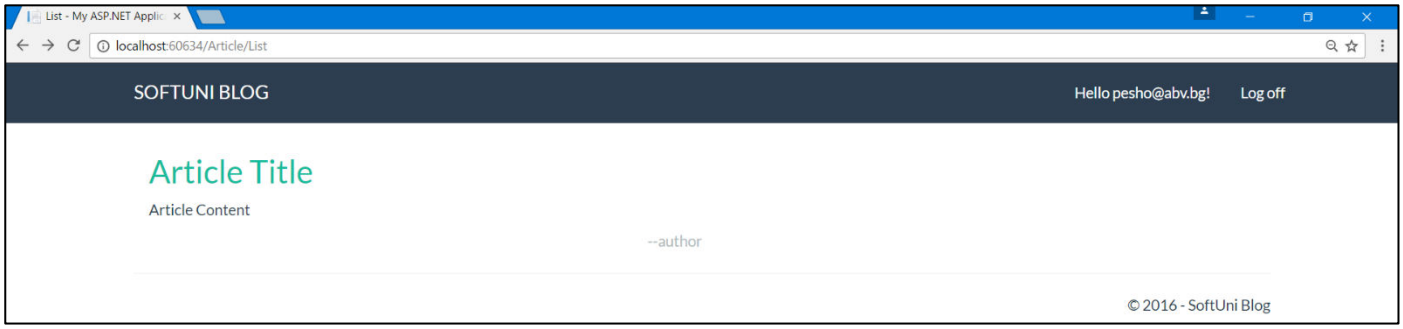
```
@model List<Blog.Models.Article>

@{
    ViewBag.Title = "List";
}

<div class="container">
    <div class="row">
        @foreach (var article in Model)
        {
            <div class="col-sm-6">
                <article>
                    <header>
                        <h2>
                            @Html.ActionLink(@article.Title, "Details", "Article", new { @id
= article.Id }, null)
                        </h2>
                    </header>
                    <p>
                        @article.Content
                    </p>
                    <footer class="pull-right">
                        <small class="author">
                            --author @article.Author.FullName
                        </small>
                    </footer>
                </article>
            </div>
        }
    </div>
</div>
<hr />
```

Examine the code.

The view uses a **model**. On the first line is the statement that specifies the **type of the model** (which is **List<Article>**). Then, every time the **@Model** or **Model** is used, it basically is a **List<Article>**. So, we can iterate through it with a **foreach loop** and use the properties of the article to show them on screen.

Notice also that there is an **ActionLink** pointing to **Article/Details**. If you try to click it, it won't work, because we haven't yet created neither the Details action, nor the view.

While you are in the List view, start the application and see what happens:



If you start the application from somewhere else, just type "**/Article/List**" or "**/Article**" after the localhost and you should see the above screen.

If you click the title you should get an error. We will fix that in a moment.

3. Redirect to List Articles

The **listing of all articles** should be the **home page** of the blog. It is really easy to do that. We need to redirect to the **Article/List** action from **Home/Index** action:

```
namespace Blog.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return RedirectToAction("List", "Article");
        }
    }
}
```

This should **redirect** you to the **List()** method of the **Article Controller** whenever you try to access the home page.

But there is a problem. When you **click on the article title** to view it, an **exception is thrown**.

We need to implement **Article Details**.

VIII. Article Details

1. Creating the Get Method

In the **Article controller**, create an action called "**`Details`**".

The integer parameter should be **nullable** (**int?**), e.g. it can have a null value (unlike a normal **int**), just like an object:

```
//
// GET: Article/Details
public ActionResult Details(int? id)
{

}
```

We should **check if it is null** (e.g. something went wrong in the request):

```
//
// GET: Article/Details
public ActionResult Details(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
}
```

Now we need an **instance** of the **database context** and a query for the article by its id:

```
//
// GET: Article/Details
public ActionResult Details(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }

    using (var database = new BlogDbContext())
    {
        // Get the article from database

        if (article == null)
        {
            return HttpNotFound();
        }

        return View(article);
    }
}
```

Fetch the article using this **LINQ** query:

```
var article = database.Articles
    .Where(a => a.Id == id)
    .Include(a => a.Author)
    .First();
```

You can see how the **id is passed** to the action inside the List view:

```
<header>
    <h2>
        @Html.ActionLink(@article.Title, "Details", "Article", new { @id = article.Id }, null)
    </h2>
</header>
```

It is passed as a **parameter in the action link**.

2. Creating the View

The above action gives the view an article. Now we need to show it in the browser. Create a view called "**`Details`**"and paste the following code:

```
@model Blog.Models.Article

@{
    ViewBag.Title = "Details";
}

<div class="container">
    <article>
        <header>
            <h2>
                @Model.Title
            </h2>
        </header>
        <p>
            @Model.Content
        </p>
        <small class="author">
```

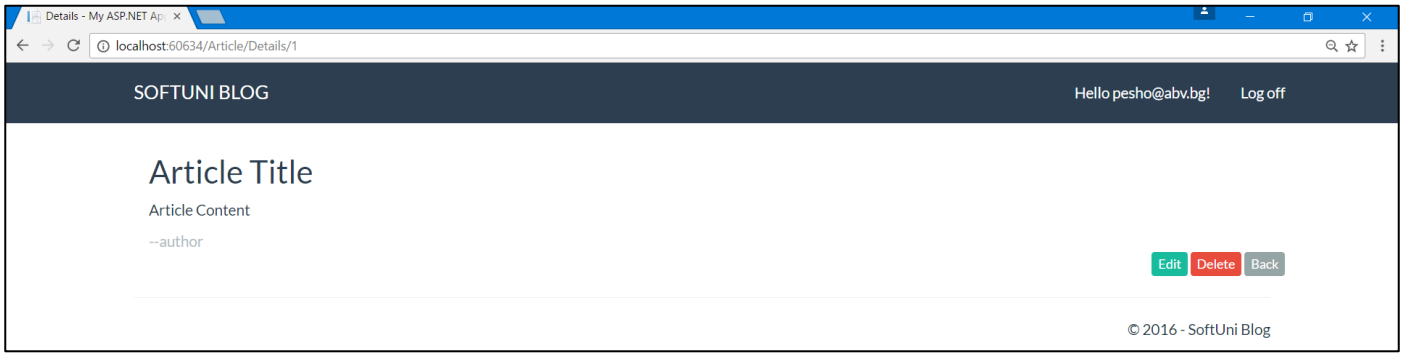
```
            --author @Model.Author.FullName
        </small>
        <footer class="pull-right">
            @Html.ActionLink("Edit", "Edit", "Article", new { @id = Model.Id }, new { @class
= "btn btn-success btn-xs" })
            @Html.ActionLink("Delete", "Delete", "Article", new { @id = Model.Id }, new {
@class = "btn btn-danger btn-xs" })
            @Html.ActionLink("Back", "Index", "Article", null, new { @class = "btn btn-
default btn-xs" })
        </footer>
    </article>
</div>
<hr/>
```

There are some new elements in the code.

Inspect the buttons in the footer. There are three **Action Links**.

They are created with the helper method **@Html.ActionLink** which creates a link. The method takes as parameters the **display value** of the button (what it will show in the browser), **the action** and **the controller** to which the link points and **two anonymous classes**.

You can now test the **Details** view:



IX. Creating Articles

1. Creating the Get Method

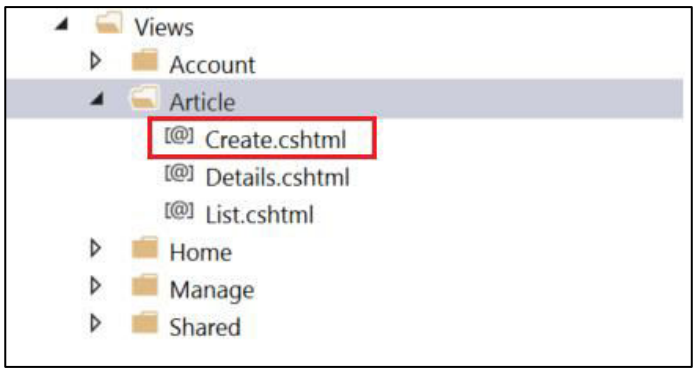
For article creation, we need two actions (**get** and **post**) and a view.

In **ArticleController**, create the first action called **Create**:

```
//
// GET: Article/Create
public ActionResult Create()
{
    return View();
}
```

2. Creating the View

Before we create the second method, lets create the view -> "**`Views/Article/Create.cshtml`**":

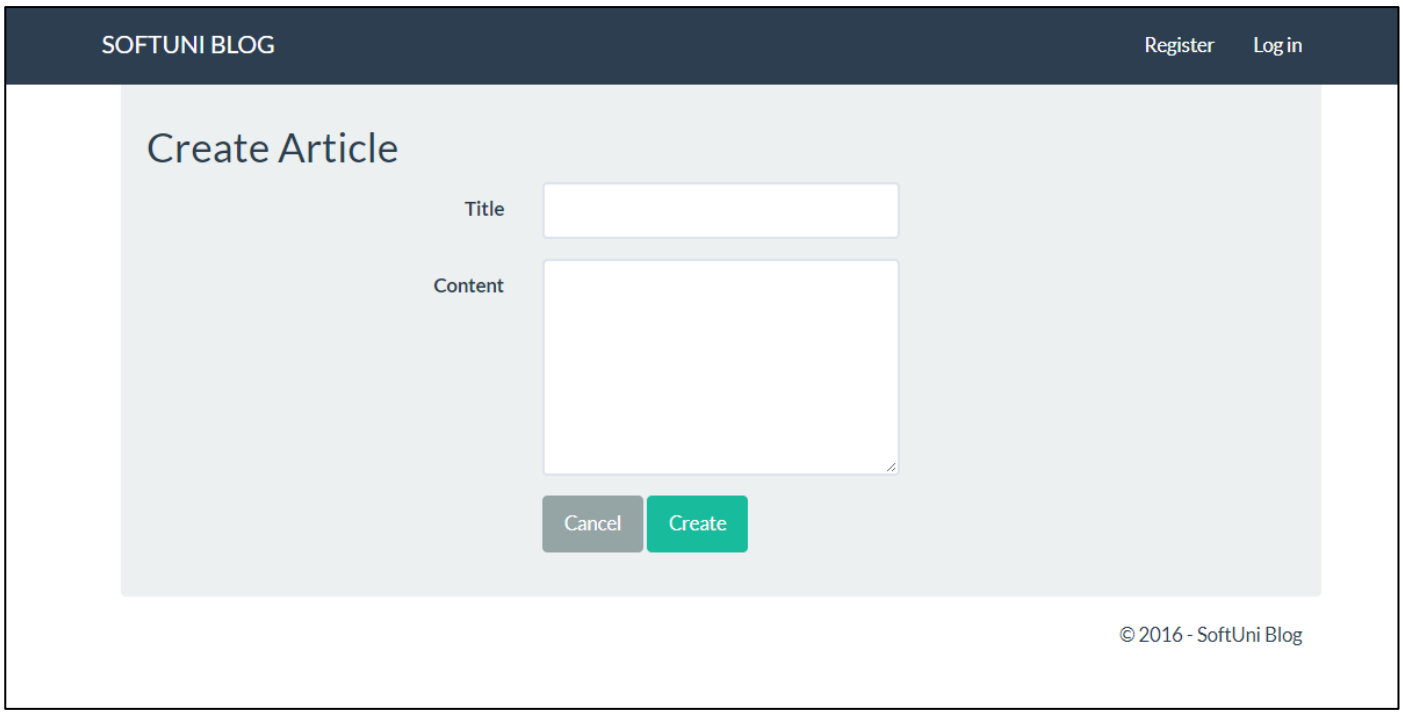


Paste the following code there:

```
@model Blog.Models.Article
@{
    ViewBag.Title = "Create";
}

<div class="container">
    <div class="well">
        <h2>Create Article</h2>
        @using (Html.BeginForm("Create", "Article", FormMethod.Post, new { @class = "form-
horizontal" }))
        {
            @Html.AntiForgeryToken()
            @Html.ValidationSummary("", new { @class = "text-danger" })

            <div class="form-group">
                @Html.LabelFor(m => m.Title, new { @class = "control-label col-sm-4" })
                <div class="col-sm-4">
                    @Html.TextBoxFor(m => m.Title, new { @class = "form-control" })
                </div>
            </div>
            <div class="form-group">
                @Html.LabelFor(m => m.Content, new { @class = "control-label col-sm-4" })
                <div class="col-sm-4">
                    @Html.TextAreaFor(m => m.Content, new { @class = "form-control", @rows =
"7" })
                </div>
            </div>

            <div class="form-group">
                <div class="col-sm-4 col-sm-offset-4">
                    @Html.ActionLink("Cancel", "Index", "Article", null, new { @class = "btn
btn-default" })
                    <input type="submit" value="Create" class="btn btn-success" />
                </div>
            </div>
        }
    </div>
</div>
```

This should be already familiar. It's a **view** with a single form in which there are **text box** for title, **text area** for content and **two buttons**. One of the buttons **submits the form** and the other redirects to the home of the blog.

The final **view** should look something like this:



3. Creating the Post Method

Adding articles to the database is done through the **post action method** that we mentioned earlier. Basically, once you hit the submit button you **send the information from the form to the action**.

Start with method creation:

```
//
// POST: Article/Create
[HttpPost]
public ActionResult Create(Article article)
{
    if (ModelState.IsValid)
    {
        // Insert article in DB
    }

    return View(article);
}
```

The **method receives** an **article model**. If the model is valid we can **insert it in the db**. If not, we just display the same page with the info from the model.

We can chop the article insertion into smaller problems:

```
//
// POST: Article/Create
[HttpPost]
public ActionResult Create(Article article)
{
    if (ModelState.IsValid)
    {
        using (var database = new BlogDbContext())
        {
            // Get author id

            // Set articles author

            // Save article in DB

            return RedirectToAction("Index");
        }
    }

    return View(article);
}
```

1. We can **get the author from the db** this way:

```
// Get author id
var authorId = database.Users
    .Where(u => u.UserName == this.User.Identity.Name)
    .First()
    .Id;
```

2. Now, **set the author of the article**:

```
// Set articles author
article.AuthorId = authorId;
```

3. And **add the article** to the db:

```
// Save article in DB
database.Articles.Add(article);
database.SaveChanges();
```

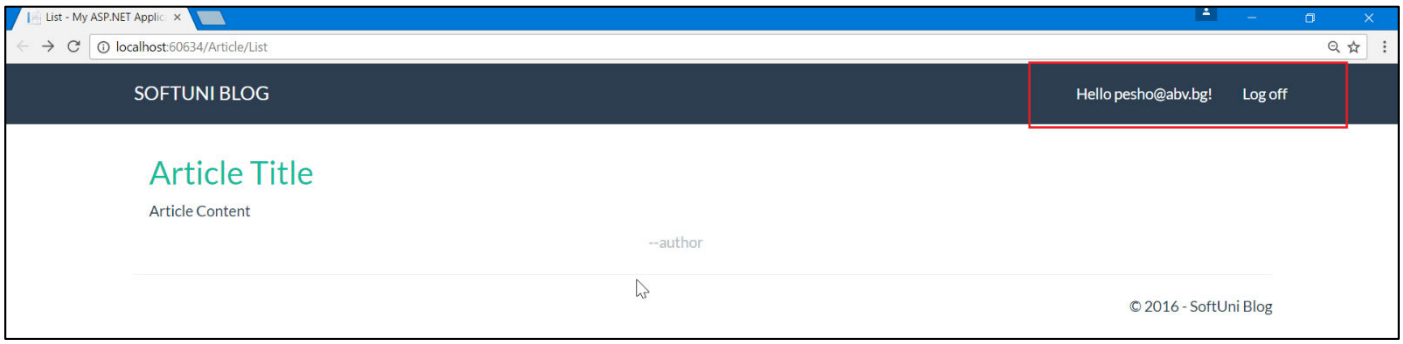
In the end, the **method should look like this**:

```
//
// POST: Article/Create
[HttpPost]
public ActionResult Create(Article article)
{
    if (ModelState.IsValid)
    {
        using (var database = new BlogDbContext())
        {
            // Get author id
            var authorId = database.Users
                .Where(u => u.UserName == this.User.Identity.Name)
                .First()
                .Id;

            // Set articles author
            article.AuthorId = authorId;

            // Save article in DB
            database.Articles.Add(article);
            database.SaveChanges();

            return RedirectToAction("Index");
        }
    }

    return View(article);
}
```

4. Create Article Action Link

We need a link through which we can go to the create article page.

Just go to **"Views/Shared/_LoginPartial.cshtml"**. This is the view that handles the upper right navigation.



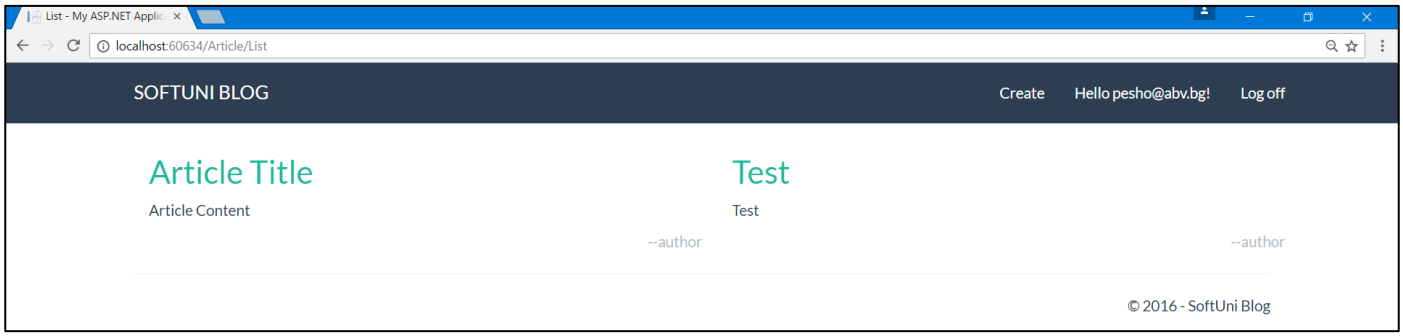
If you **inspect the view**, you will see that there is a **conditional statement** there. If the user is logged, it shows some links. If no one is logged, it shows different links.

Inside the logged user part add a new list item:

```
@using Microsoft.AspNet.Identity
@if (Request.IsAuthenticated)
{
    using (Html.BeginForm("LogOff", "Account", FormMethod.Post, new { id = "logoutForm", @class = "navbar-right" }))
    {
    @Html.AntiForgeryToken()

    <ul class="nav navbar-nav navbar-right">
        <li>
            @Html.ActionLink("Create", "Create", "Article")
        </li>
        <li>
            @Html.ActionLink("Hello " + User.Identity.GetUserName() + "!", "Index", "Manage", routeValues: null, htmlAtt
        </li>
        <li><a href="javascript:document.getElementById('logoutForm').submit()">Log off</a></li>
    </ul>
    }
}
```

This will create a link with display in the browser "**Create**", pointing to "**Create**" action inside the "**Article**" controller.

You can now test if the article creation actually works.



X. Deleting Articles

1. Create the Get Method

Deleting articles follow the same pattern as creating them. A **get** and a **post** method will be needed.

Create the **GET Method** in the **Article Controller**:

```
//
// GET: Article/Delete
public ActionResult Delete(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }

    using (var database = new BlogDbContext())
    {
        // Get article from database

        // Check if article exists

        // Pass article to view
    }
}
```

It should **receive an article id** so it would know **which article to delete**. It also checks if the id is **valid**.

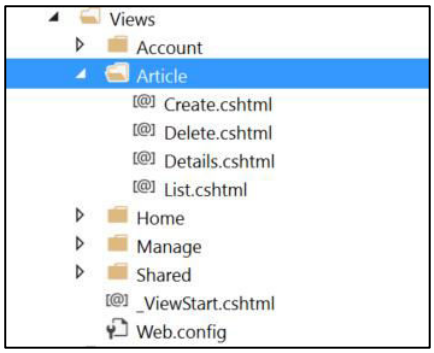
Then, **get the article** from database and if it exists, pass it to the **Delete View**:

```
    //
    // GET: Article/Delete
    public ActionResult Delete(int? id)
    {
        if (id == null)
        {
            return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
        }

        using (var database = new BlogDbContext())
        {
            // Get article from database
            var article = database.Articles
                .Where(a => a.Id == id)
                .Include(a => a.Author)
                .First();

            // Check if article exists
            if (article == null)
            {
                return HttpNotFound();
            }

            // Pass article to view
            return View(article);
        }
    }
```

2. Create the View

The above method directs to a delete view, but it doesn't exist yet.

Create a "**Delete**" view:

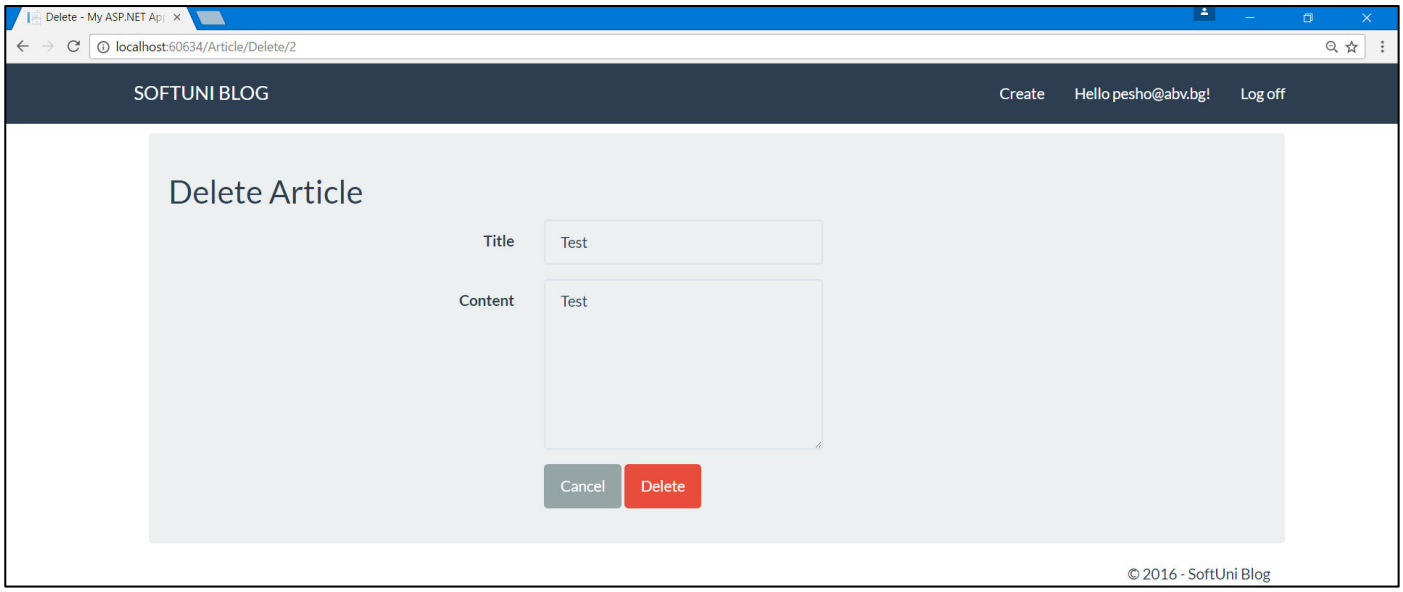


Paste the following code:

```
@model Blog.Models.Article

@{
    ViewBag.Title = "Delete";
}

<div class="container">
    <div class="well">
        <h2>Delete Article</h2>
        @using (Html.BeginForm("Delete", "Article", FormMethod.Post, new { @class = "form-
horizontal" }))
        {
            @Html.AntiForgeryToken()

            <div class="form-group">
                @Html.LabelFor(m => m.Title, new { @class = "control-label col-sm-4" })
                <div class="col-sm-4">
```

```
                    @Html.TextBoxFor(m => m.Title, new { @class = "form-control", @readonly =
"readonly" })
                </div>
            </div>
            <div class="form-group">
                @Html.LabelFor(m => m.Content, new { @class = "control-label col-sm-4" })
                <div class="col-sm-4">
                    @Html.TextAreaFor(m => m.Content, new { @class = "form-control",
@readonly = "readonly", @rows = "7" })
                </div>
            </div>

            <div class="form-group">
                <div class="col-sm-4 col-sm-offset-4">
                    @Html.ActionLink("Cancel", "Index", "Article", null, new { @class = "btn
btn-default" })
                    <input type="submit" value="Delete" class="btn btn-danger" />
                </div>
            </div>
        }
    </div>
</div>
```

This is the same as the Details view, but the **text boxes are populated** with the **articles information** and you can't edit it.



3. Create the Post Method

When you click on the "**Delete**" button (this is actually the submit of the form) you are sent to the **POST Method** in the controller.

Let's create it:

```
//
// POST: Article/Delete
[HttpPost]
[ActionName("Delete")]
public ActionResult DeleteConfirmed(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }

    using (var database = new BlogDbContext())
    {
        // Get article from database

        // Check if article exists

        // Delete article from database

        // Redirect to index page
    }
}
```

Make sure you add the attribute [**HttpPost**].

You can see that if we name the method "**Delete**" it will have the same signature as the **GET Method**. So we can name it **DeleteConfirmed**. But this way, the route to the method will not be valid (it responds to **Article/Delete/Id**). So, we fixed this by adding the attribute [**ActionName("Delete")**].

1. Now, let's **get the article** form the database:

```
// Get article from db
var article = database.Articles
    .Where(a => a.Id == id)
    .Include(a => a.Author)
    .First();
```

2. **Check** if the article exists:

```
// Check if article exists
if (article == null)
{
    return HttpNotFound();
}
```

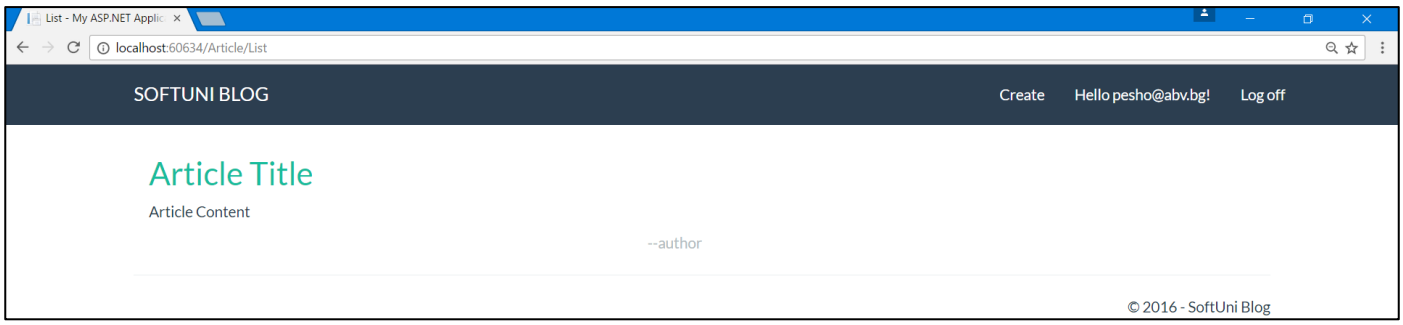
3. **Delete** the article:

```
// Remove article from db
database.Articles.Remove(article);
database.SaveChanges();
```

4. And, **redirect** to the index page:

```
// Redirect to index page
return RedirectToAction("Index");
```

Test if you can delete an article:

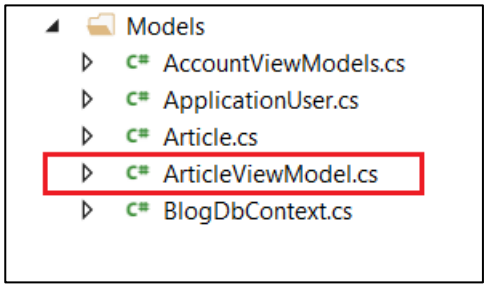


XI. Editing Articles

For editing articles, we are going to create a view model.

1. Creating the View Model

This is the same as creating regular models. Just create a class named **ArticleViewModel** in the models folder:



Inside the new class, we will have the properties we need to be edited in the view - **Title** and **Content**. But aside from them we will need properties for **article id** and **author id**, so we can keep track of them:

```
public class ArticleViewModel
{
    public int Id { get; set; }

    [Required]
    [StringLength(50)]
    public string Title { get; set; }

    [Required]
    public string Content { get; set; }

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

We are going to use the **ArticleViewModel** to edit an article and later on, to create one.

2. Creating the Get Method

Again, editing articles follow the same pattern. A **get** and a **post** method.

Create the **GET Method**:

```
//
// GET: Article/Edit
public ActionResult Edit(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }

    using (var database = new BlogDbContext())
    {
        // Get article from the database

        // Check if article exists

        // Create the view model

        // Pass the view model to view
    }
}
```

1. **Get** the article

```
// Get article from the database
var article = database.Articles
    .Where(a => a.Id == id)
    .First();
```

2. **Check** if it exists

```
// Check if article exists
if (article == null)
{
    return HttpNotFound();
}
```

3. **Create** the view model

```
// Create the view model
var model = new ArticleViewModel();
model.Id = article.Id;
model.Title = article.Title;
model.Content = article.Content;
```

4. **Pass** it to the view

```
// Pass the view model to view
return View(model);
```

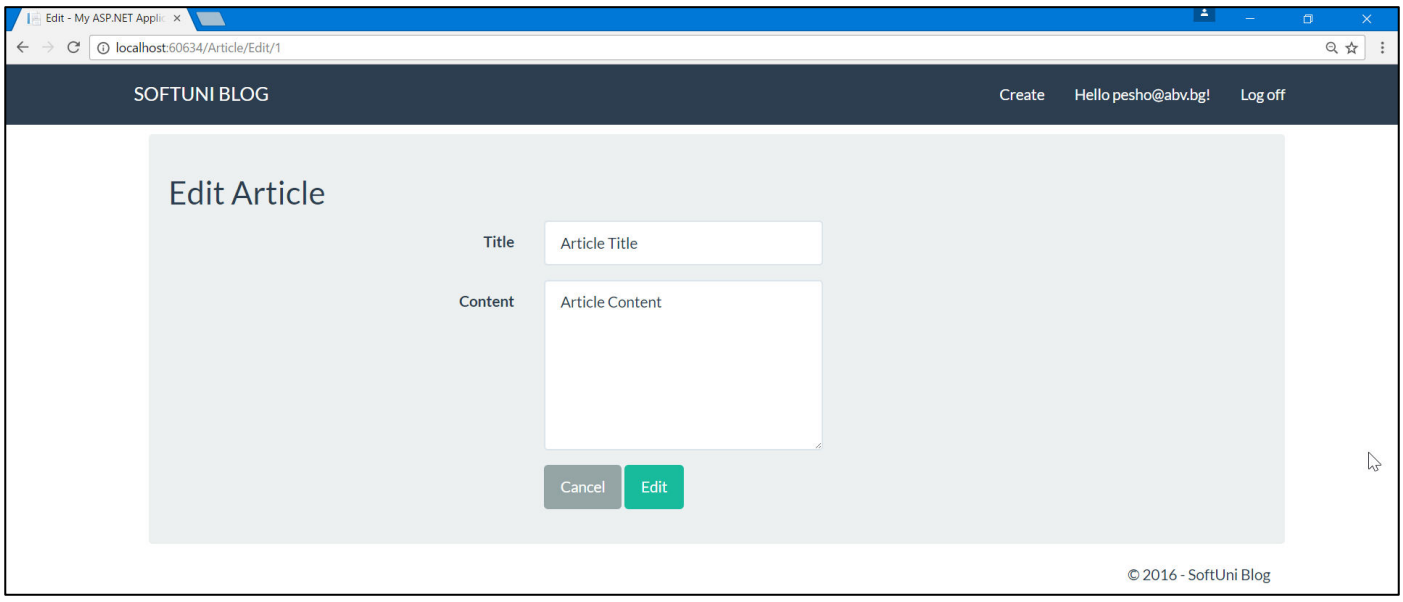
3. Creating the View

You should be able to create the view by **yourself**.

Just kidding, paste the following code in the view:

```
@model Blog.Models.ArticleViewModel

@{
    ViewBag.Title = "Edit";
}

<div class="container">
    <div class="well">
        <h2>Edit Article</h2>
        @using (Html.BeginForm("Edit", "Article", FormMethod.Post, new { @class = "form-
horizontal" }))
        {
            @Html.AntiForgeryToken()
            @Html.ValidationSummary()

            @Html.HiddenFor(m => m.Id)
            @Html.HiddenFor(m => m.AuthorId)

            <div class="form-group">
                @Html.LabelFor(m => m.Title, new { @class = "control-label col-sm-4" })
                <div class="col-sm-4">
                    @Html.TextBoxFor(m => m.Title, new { @class = "form-control" })
                </div>
            </div>

            <div class="form-group">
                @Html.LabelFor(m => m.Content, new { @class = "control-label col-sm-4" })
                <div class="col-sm-4">
                    @Html.TextAreaFor(m => m.Content, new { @class = "form-control", @rows =
"7" })
                </div>
            </div>

            <div class="form-group">
                <div class="col-sm-4 col-sm-offset-4">
                    @Html.ActionLink("Cancel", "Index", "Article", null, new { @class = "btn
btn-default" })
                    <input type="submit" value="Edit" class="btn btn-success" />
                </div>
            </div>
        }
    </div>
</div>
```

It should result in this:



4. Creating the Post Method

We can also use the view model In the **Post Method**. This is done **through method parameters**:

Create the method:

```
//
// POST: Article/Edit
[HttpPost]
public ActionResult Edit(ArticleViewModel model)
{
    // Check if model state is valid
    if (ModelState.IsValid)
    {
        using (var database = new BlogDbContext())
        {
            // Get article from database

            // Set article properties

            // Save article state in database

            // Redirect to the index page
        }
    }

    // If model state is invalid, return the same view
    return View(model);
}
```

To implement the method:

1. First, **get the article** from the database

```
// Get article from database
var article = database.Articles
    .FirstOrDefault(a => a.Id == model.Id);
```

2. Next, **set article's** new **values**

```
// Set article properties
article.Title = model.Title;
article.Content = model.Content;
```

3. **Set the article state** to modified

```
// Save article state in database
database.Entry(article).State = EntityState.Modified;
database.SaveChanges();
```

4. **Redirect** to the **index page**

```
// Redirect to the index page
return RedirectToAction("Index");
```

XII. Validations

1. Validating Create Article

Right now, **everyone can use the routing** to get to article creation and **to create an article**. Only registered users should be able to do that.

The only thing we need to do is to **add an attribute** over the **Create Article Actions**:

```
//
// GET: Article/Create
[Authorize]
public ActionResult Create()...

//
// POST: Article/Create
[HttpPost]
[Authorize]
public ActionResult Create(ArticleViewModel model)...
```

If you now try to create an article when you are not registered, you won't be able to.

2. Creating a Helper Methods

Let's create a method inside the **Article Class** that checks **if a given user is its author**:

It is a really simple method:

```
public bool IsAuthor(string name)
{
    return this.Author.UserName.Equals(name);
}
```

We can use the name of a user because usernames are unique.

The second method will be inside the **Article Controller**. Again, it is pretty simple:

```
private bool IsUserAuthorizedToEdit(Article article)
{
    bool isAdmin = this.User.IsInRole("Admin");
    bool isAuthor = article.IsAuthor(this.User.Identity.Name);

    return isAdmin || isAuthor;
}
```

It **receives an article** and **checks weather the user is authorized** to edit it in any way. If he is the author the author of the article or the admin, he gets the permission.

3. Hiding Edit and Delete

We want only article authors and admins to see the buttons for editing and deleting articles.

We can do this in the "**Views/Articles/Details** ".

Find the footer:

```
<footer class="pull-right">
    @Html.ActionLink("Edit", "Edit", "Article", new { @id = Model.Id }, new { @class = "btn btn-success btn-xs" })
    @Html.ActionLink("Delete", "Delete", "Article", new { @id = Model.Id }, new { @class = "btn btn-danger btn-xs" })
    @Html.ActionLink("Back", "Index", "Article", null, new { @class = "btn btn-default btn-xs" })
</footer>
```

Add the following conditional and place only the edit and delete buttons:

```
<footer class="pull-right">
    @if (User.IsInRole("Admin") || Model.IsAuthor(User.Identity.Name))
    {
        @Html.ActionLink("Edit", "Edit", "Article", new { @id = Model.Id }, new { @class = "btn btn-success btn-xs" })
        @Html.ActionLink("Delete", "Delete", "Article", new { @id = Model.Id }, new { @class = "btn btn-danger btn-xs" })
    }

    @Html.ActionLink("Back", "Index", "Article", null, new { @class = "btn btn-default btn-xs" })
</footer>
```

4. Validating Delete Requests

The only thing left to do is to validate any requests for editing and deleting articles.

```
//
// GET: Article/Delete
public ActionResult Delete(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }

    using (var database = new BlogDbContext())
    {
        var article = database.Articles
            .Where(a => a.Id == id)
            .Include(a => a.Author)
            .First();

        if (! IsUserAuthorizedToEdit(article))
        {
            return new HttpStatusCodeResult(HttpStatusCode.Forbidden);
        }

        if (article == null)
        {
            return HttpNotFound();
        }

        return View(article);
    }
}
```

Go to the **Article/Delete** Action and **add the following validation**:

5. Validating Edit Requests

Do the same as above but this time for the action **Article/Edit**.

6. Test Edit and Delete Article

Test if you can **edit** or **delete** any articles, while you are not logged, logged as a user, being an author of an article.