**Implement Security Using ASP.NET Core Identity In 10 Easy Steps**

Many web applications need to authenticate and authorize its users. A common approach is to accept user name and password from the user and validate them against some data store. As far as ASP.NET Core web applications are concerned the recommended way to implement such a security using ASP.NET Core Identity. In this article you will learn to implement user authentication as well as role based security using ASP.NET Core Identity. You will do so by building a sample application from scratch using the empty project template.

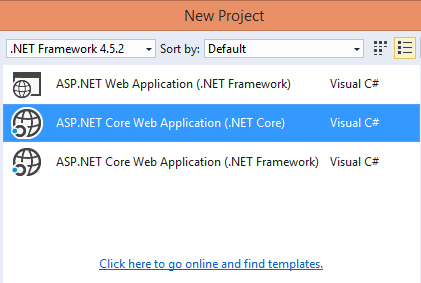
The steps required to build this application are listed below:

* Step 1 : Create a new ASP.NET Core project using Empty project template
* Step 2 : Add the required NuGet packages
* Step 3 : Create Identity DbContext, user and role classes
* Step 4 : Configure application startup
* Step 5 : Create view models required by the application
* Step 6 : Create AccountController - the controller containing registration / login / logout code.
* Step 7 : Create Register and Login views
* Step 8 : Create HomeController - the controller that contains actions to be secured
* Step 9 : Create Index view
* Step 10 : Create database tables using dotnet EF Core migrations commands

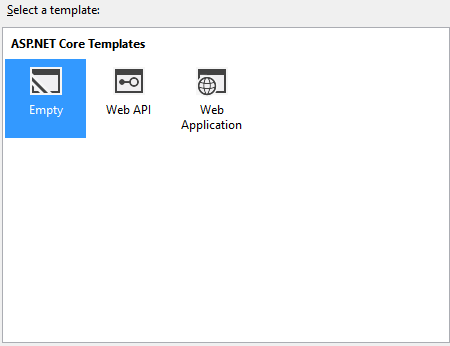
Let's begin!

**Step 1 : Creating a new project**

Create a new ASP.NET Core Web Application using Visual Studio 2015.



Make sure to select the Empty project template while creating the project.



**Step 2 : Adding NuGet packages**

Since you have selected the Empty project template, the Project.json won't have any mention of the NuGet packages. Open the Project.json file and modify the dependencies and tools sections as shown below:

"dependencies": {

"Microsoft.NETCore.App": {

"version": "1.0.0",

"type": "platform"

},

**"Microsoft.AspNetCore.Authentication.Cookies": "1.0.0",**

"Microsoft.AspNetCore.Diagnostics": "1.0.0",

"Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore": "1.0.0",

**"Microsoft.AspNetCore.Identity.EntityFrameworkCore": "1.0.0",**

**"Microsoft.AspNetCore.Mvc": "1.0.0",**

**"Microsoft.AspNetCore.Razor.Tools": {**

**"version": "1.0.0-preview2-final",**

**"type": "build"**

**},**

"Microsoft.AspNetCore.Server.IISIntegration": "1.0.0",

"Microsoft.AspNetCore.Server.Kestrel": "1.0.0",

"Microsoft.AspNetCore.StaticFiles": "1.0.0",

**"Microsoft.EntityFrameworkCore.SqlServer": "1.0.0",**

"Microsoft.EntityFrameworkCore.SqlServer.Design": {

"version": "1.0.0",

"type": "build"

},

"Microsoft.EntityFrameworkCore.Tools": {

"version": "1.0.0-preview2-final",

"type": "build"

},

"Microsoft.Extensions.Configuration.EnvironmentVariables": "1.0.0",

"Microsoft.Extensions.Configuration.Json": "1.0.0",

"Microsoft.Extensions.Configuration.UserSecrets": "1.0.0",

"Microsoft.Extensions.Logging": "1.0.0",

"Microsoft.Extensions.Logging.Console": "1.0.0",

"Microsoft.Extensions.Logging.Debug": "1.0.0",

"Microsoft.Extensions.Options.ConfigurationExtensions": "1.0.0",

"Microsoft.VisualStudio.Web.BrowserLink.Loader": "14.0.0",

"Microsoft.VisualStudio.Web.CodeGeneration.Tools": {

"version": "1.0.0-preview2-final",

"type": "build"

},

"Microsoft.VisualStudio.Web.CodeGenerators.Mvc": {

"version": "1.0.0-preview2-final",

"type": "build"

}

},

"tools": {

"BundlerMinifier.Core": "2.0.238",

"Microsoft.AspNetCore.Razor.Tools":

"1.0.0-preview2-final",

"Microsoft.AspNetCore.Server.IISIntegration.Tools":

"1.0.0-preview2-final",

"Microsoft.EntityFrameworkCore.Tools":

"1.0.0-preview2-final",

"Microsoft.Extensions.SecretManager.Tools":

"1.0.0-preview2-final",

"Microsoft.VisualStudio.Web.CodeGeneration.Tools": {

"version": "1.0.0-preview2-final",

"imports": [

"portable-net45+win8"

]

}

},

Notice the NuGet packages marked in bold letters. They are required for ASP.NET Core MVC, Entity Framework Core and ASP.NET Core Identity.

**Step 3 : Create Identity DbContext, User and Role classes**

Our sample application being built stored user names and passwords in a local SQL server database. To talk with this database you need an IdentityDbContext class. The IdentityDbContext needs to know what "type" of users and roles it will be dealing with. Thus you also need to create user and role classes.

The MyIdentityUser class shown below represents our application user.

public class MyIdentityUser:IdentityUser

{

public string FullName { get; set; }

public DateTime BirthDate { get; set; }

}

As you can see the MyIdentityUser class inherits from IdentityUser base class (Microsoft.AspNetCore.Identity.EntityFrameworkCore namespace). The IdentityUser base class contains basic user details such as UserName, Password and Email. We also want to capture FullName and BirthDate of a user. So, we add these additional properties in MyIdentityUser class.

Now add MyIdentityRole class as shown below:

public class MyIdentityRole:IdentityRole

{

public string Description { get; set; }

}

The MyIdentityRole class inherits from IdentityRole base class. The base class provides details such as RoleName and we add Description as an extra piece of information.

If you don't want any additional details to be captured you could have used IdentityUser and IdentityRole classes directly.

Now that user and role classes are ready, let's create IdentityDbContext class.

public class MyIdentityDbContext:

dentityDbContext<MyIdentityUser,MyIdentityRole,string>

{

public MyIdentityDbContext

(DbContextOptions<MyIdentityDbContext> options)

: base(options)

{

//nothing here

}

}

The MyIdentityDbContext class inherits from IdentityDbContext base class. Notice how MyIdentityUser and MyIdentityRole types are passed while creating the MyIdentityDbContext class. The third parameter is the data type of the primary key for the user and role classes.

**Step 4 : Configure application startup**

Open Startup.cs file and modify the ConfigureServices() and Configure() methods as shown below:

private IConfiguration config;

public Startup(IHostingEnvironment env)

{

var builder = new ConfigurationBuilder();

builder.SetBasePath(env.ContentRootPath);

**builder.AddJsonFile("appsettings.json");**

config = builder.Build();

}

public void ConfigureServices(IServiceCollection services)

{

**services.AddDbContext<MyIdentityDbContext>(options =>**

**options.UseSqlServer(config.GetConnectionString**

**("DefaultConnection")));**

**services.AddIdentity<MyIdentityUser, MyIdentityRole>()**

**.AddEntityFrameworkStores<MyIdentityDbContext>()**

**.AddDefaultTokenProviders();**

services.AddMvc();

}

public void Configure(IApplicationBuilder app,

IHostingEnvironment env, ILoggerFactory loggerFactory)

{

app.UseStaticFiles();

**app.UseIdentity();**

app.UseMvc(routes =>

{

routes.MapRoute(

name: "default",

template: "{controller=Home}/

{action=Index}/{id?}");

});

}

Notice the code marked in bold letters.

The Startup constructor loads AppSettings.json file containing a database connection string. This database stores the user accounts and other details. Make sure to add the AppSettings.json file and specify a valid database connection string there. An example is given below:

{

"ConnectionStrings": {

"DefaultConnection": "Server=.;Database=MyDb;

Trusted\_Connection=True;MultipleActiveResultSets=true"

}

}

The ConfigureServices() method calls AddDbContext() method to add MyIdentityDbContext to the services collection. Notice how the database connection string is specified. Then AddIdentity() method is used to add ASP.NET Core Identity services to the container. This is where MyIdentityUser and MyIdentityRole classes are also mentioned.

The Configure() method calls UserIdentity() method to add ASP.NET Core Identity to the request pipeline.

**Step 5 :  Create RegisterViewModel and LoginViewModel classes**

We need two view models - RegisterViewModel and LoginViewModel - in our application. These view models hold the data entered on the register and login views respectively and are used by the AccountController we create later.

These two view model classes are shown below:

public class RegisterViewModel

{

[Required]

public string UserName { get; set; }

[Required]

[DataType(DataType.Password)]

public string Password { get; set; }

[Required]

[DataType(DataType.Password)]

public string ConfirmPassword { get; set; }

[Required]

[DataType(DataType.EmailAddress)]

public string Email { get; set; }

[Required]

public string FullName { get; set; }

[Required]

[DataType(DataType.Date)]

public DateTime BirthDate { get; set; }

}

public class LoginViewModel

{

[Required]

public string UserName { get; set; }

[Required]

[DataType(DataType.Password)]

public string Password { get; set; }

[Required]

public bool RememberMe { get; set; }

}

As you can see, the view model classes are quite straightforward. The RegisterViewModel class has six properties namely UserName, Password, ConfirmPassword, Email, FullName and BirthDate. The LoginViewModel class contains three properties namely UserName, Password and RememberMe. Various properties of the view model classes are decorated with data annotation attributes for the sake of model validation.

**Step 6 : Create AccountController class**

The AccountController class is a controller that performs registration, login and logout operations. The AccountController uses the functionality of ASP.NET Core Identity for creating user accounts and signing the user in and out of the application.

We declare a few private members for this class and assign them in the constructor as shown below:

public class AccountController : Controller

{

private readonly UserManager<MyIdentityUser> userManager;

private readonly SignInManager<MyIdentityUser> loginManager;

private readonly RoleManager<MyIdentityRole> roleManager;

public AccountController(UserManager<MyIdentityUser> userManager,

SignInManager<MyIdentityUser> loginManager,

RoleManager<MyIdentityRole> roleManager)

{

this.userManager = userManager;

this.loginManager = loginManager;

this.roleManager = roleManager;

}

....

....

}

The AccountController declares three private variables of type UserManager<T>, SignInManager<T> and RoleManager<T> respectively. The UserManager class is used for creating and managing users. The RoleManager class is used for creating and managing roles. The SignInManager class is used to log a user in and out of the system.

Object instances of these three classes are injected into the AccountController constructor as shown above. The constructor code simply assigns the injected objects to the respective variables declared earlier.

The AccountController has in all five actions :

* Two Register() actions - GET and POST
* Two Login() actions - GET and POST
* One Logout() actions.- POST

The Register() actions are responsible for creating user accounts and are shown below:

public IActionResult Register()

{

return View();

}

[HttpPost]

[ValidateAntiForgeryToken]

public IActionResult Register(RegisterViewModel obj)

{

if (ModelState.IsValid)

{

MyIdentityUser user = new MyIdentityUser();

user.UserName = obj.UserName;

user.Email = obj.Email;

user.FullName = obj.FullName;

user.BirthDate = obj.BirthDate;

IdentityResult result = userManager.CreateAsync

(user, obj.Password).Result;

if (result.Succeeded)

{

if(!roleManager.RoleExistsAsync("NormalUser").Result)

{

MyIdentityRole role = new MyIdentityRole();

role.Name = "NormalUser";

role.Description = "Perform normal operations.";

IdentityResult roleResult = roleManager.

CreateAsync(role).Result;

if(!roleResult.Succeeded)

{

ModelState.AddModelError("",

"Error while creating role!");

return View(obj);

}

}

userManager.AddToRoleAsync(user,

"NormalUser").Wait();

return RedirectToAction("Login", "Account");

}

}

return View(obj);

}

The Register() action with no parameters simply returns a blank registration page to the user. The second Register() action accepts RegisterViewModel object as its parameter. Inside, the code checks whether the model contains valid data or not using the IsValid property of ModelState object. If the model is valid we create a new MyIdentityUser object and assign its UserName, Email, FullName and BirthDate properties. Then the code calls the CreateAsync() method of the UserManager class in an attempt to create a user account. The password is passed to the CreateAsync() method. Since the CreateAsync() is an asynchronous call, we block the execution by calling the Result property. The CreateAsync() method returns IdentityResult - an object that tells us whether the user creation was successful or not.

If the user creation succeeds (result.Succeeded property) we proceed to create the roles required by the application. The role creation is a one time operation. It should happen only if a role doesn't exist already. So, the code uses RoleManager to check whether a role already exists or not. This is done using RoleExistsAsync() method and by passing role name as its parameter.

If a role doesn't exist already, MyIdentityRole object representing that role is created. Its RoleName and Description properties are assigned the required values (NormalUser is the role name in this example). Then CreateAsync() is called on RoleManager in an attempt to create the role. If the role creation succeeds (Succeeded property returns true) we add the newly created user to the NormalUser role. This is done using AddToRoleAsync() method of UserManager.

After creating the account and adding the new user to the NormalUser role we redirect the user to the login page. The two Login() actions are discussed next.

public IActionResult Login()

{

return View();

}

[HttpPost]

[ValidateAntiForgeryToken]

public IActionResult Login(LoginViewModel obj)

{

if (ModelState.IsValid)

{

**var result = loginManager.PasswordSignInAsync**

**(obj.UserName, obj.Password,**

**obj.RememberMe,false).Result;**

if (result.Succeeded)

{

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

}

ModelState.AddModelError("", "Invalid login!");

}

return View(obj);

}

The first Login() action simply returns a blank login view to the user. When a user enters user name and password and submits the form the form is posted to the second Login() action.

The code then attempts to log the user in by calling the PasswordSignInAsync() method of the SignInManager. The PasswordSignInAsync() method accepts user name, password and remember me flag and account lockout flag. ASP.NET Core Identity uses cookie based authentication scheme to authenticate requests. The remember me flag controls whether the authentication cookie is to be persisted on the client machine when the browser is closed. A value of true will persist  the cookie. We accept this flag from the end user using a checkbox (as you will see later). The lockout flag comes into picture only when you are using account lockout feature (account is locked after certain number of unsuccessful attempts). In this example we aren't using account lockout and hence we pass false.

If the user name and password are valid the PasswordSignInAsync() method will issue the authentication cookie and Succeeded property will be true. We then redirect the user to the /home/index page.

Finally, the Logout() action of the AccountController() removes the authentication cookie issued earlier. This action is shown below:

[HttpPost]

[ValidateAntiForgeryToken]

public IActionResult LogOff()

{

loginManager.SignOutAsync().Wait();

return RedirectToAction("Login","Account");

}

The SignOutAsync() method of SignInManager removes the authentication cookie. Since this method is a void method we call Wait() on it to block the call. Once a user signs out of the application we redirect the control to the login page.

**Step 7 : Create Register and Login views**

The Register and Login views are relatively straightforward. The markup of Register view is given below:

@model RegisterViewModel

<script src="~/Scripts/jquery.js"></script>

<script src="~/Scripts/jquery.validate.js"></script>

<script src="~/Scripts/jquery.validate.unobtrusive.js">

</script>

<h1>Register</h1>

<form asp-controller="Account" asp-action="Register" method="post">

<table>

<tr>

<td><label asp-for="UserName"></label></td>

<td><input asp-for="UserName" /></td>

</tr>

<tr>

<td><label asp-for="Password"></label></td>

<td><input asp-for="Password" /></td>

</tr>

<tr>

<td><label asp-for="ConfirmPassword"></label></td>

<td><input asp-for="ConfirmPassword" /></td>

</tr>

<tr>

<td><label asp-for="Email"></label></td>

<td><input asp-for="Email" /></td>

</tr>

<tr>

<td><label asp-for="FullName"></label></td>

<td><input asp-for="FullName" /></td>

</tr>

<tr>

<td><label asp-for="BirthDate"></label></td>

<td><input asp-for="BirthDate" /></td>

</tr>

<tr>

<td colspan="2"><input type="submit"

value="Register" /></td>

</tr>

</table>

<div asp-validation-summary="All" ></div>

</form>

The Register view consists of a form tag helper that submits to Register action of the Account controller. The form further uses label and input tag helpers to render the respective data entry fields. Notice that the form also uses certain script files that are used during the client side validation. You can place these files inside the Scripts folder under wwwroot folder. There is a <div> element at the bottom that uses validation summary tag helper and displays all the validation related errors.

The following figure shows a sample run of the Register view:



The complete markup of the Login view is as follows:

@model LoginViewModel

<script src="~/Scripts/jquery.js"></script>

<script src="~/Scripts/jquery.validate.js"></script>

<script src="~/Scripts/jquery.validate.unobtrusive.js">

</script>

<h1>Login</h1>

<form asp-controller="Account" asp-action="Login" method="post">

<table>

<tr>

<td><label asp-for="UserName"></label></td>

<td><input asp-for="UserName" /></td>

</tr>

<tr>

<td><label asp-for="Password"></label></td>

<td><input asp-for="Password" /></td>

</tr>

<tr>

<td><label asp-for="RememberMe"></label></td>

<td><input asp-for="RememberMe" /></td>

</tr>

<tr>

<td colspan="2"><input type="submit"

value="Register" /></td>

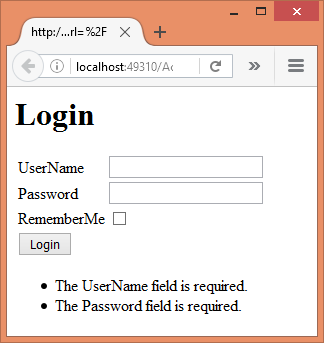
</tr>

</table>

<div asp-validation-summary="All"></div>

</form>

We won't go into the details of this markup as it's straightforward. A sample run of Login view looks like this:



**Step 8 : Create HomeController and a secured action**

The HomeController contains one secured action. The secured action simply forms messages based on current user's name and role. These actions are shown below:

private readonly UserManager<MyIdentityUser> userManager;

public HomeController(UserManager<MyIdentityUser>

userManager)

{

this.userManager = userManager;

}

[Authorize]

public IActionResult Index()

{

MyIdentityUser user = userManager.GetUserAsync

(HttpContext.User).Result;

ViewBag.Message = $"Welcome {user.FullName}!";

if(userManager.IsInRoleAsync(user,"NormalUser").Result)

{

ViewBag.RoleMessage = "You are a NormalUser.";

}

return View();

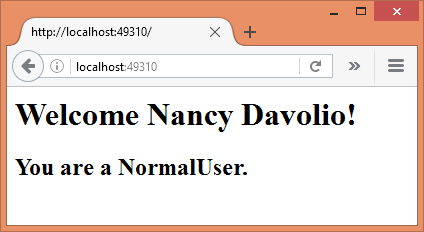
}

The HomeController's constructor receives a UserManager object through ASP.NET Core DI framework. It stores the injected UserManager in a private variable. The Index() action is secured by decorating it with [Authorize] attribute. This way you won't be able to invoke Index() unless you log-in first. The Index() action retrieves the name of the current user. This is done using the GetUserAsync() method of UserManager. Notice the use of HttpContext.User property to retrieve the underlying security principle of the current user.

A welcome message is then formed using the FullName of the user. Further, the IsInRoleAsync() method checks whether the current user belongs to NormalUser role. If so, a message is stored in the RoleMessage property of the ViewBag.

**Step 9 : Creating the Index view**

The Index view simply outputs the Message and RoleMessage properties on the page. A sample run of Index view is shown below:



**Step 10 : Create the database tables**

ASP.NET Core Identity stores the user and role information in certain database tables. To create these tables you need to issue dotnet EF migrations commands at the command prompt. So, open a command prompt and go to the project's root folder. Then issue the following commands:

>> dotnet ef migrations add MyMigrations

>> dotnet ef database update

After issuing the first command Migrations folder will be added to your project and EF migrations related code will be generated for you. The second command applies those migrations to the underlying database.

That's it! You can now run the application. Test the functionality by creating a couple of new user accounts and then try signing in with those credentials

**Seed Users And Roles Data In ASP.NET Core Identity**

ASP.NET Core identity allows you to implement authentication and authorization for your web applications. While working with ASP.NET Core Identity at times you need to create default user accounts and roles in the system. In ASP.NET MVC you could have easily done this in Global.asax and Application\_Start event handler. In ASP.NET Core the process is bit different since the application startup process is different. To that end this article explains a way to seed such user and roles data in your applications.

Before you go any further ensure that you have a web application configured to use ASP.NET Core Identity. I am going to assume that you have followed my earlier article [**here**](http://www.binaryintellect.net/articles/b957238b-e2dd-4401-bfd7-f0b8d984786d.aspx) to implement ASP.NET Core Identity in your web application. The article just mentioned is for ASP.NET Core 1.x but most of the part is same under ASP.NET Core 2.0. So, it should be an easy upgrade for you. Of course, you can use your own implementation also.

If you followed my article mentioned above, you already have MyIdentityDbContext, MyIdentityUser and MyIdentityRole classes ready with you. Our aim is to create a few user accounts and a few roles when the application runs for the first time. Obviously, we do this only if those users and roles doesn't exists in the database yet.

In the following sections we discuss two ways of seeding the data :

* Using Configure() method
* Using Main() method

The former method is simple and straightforward with minimal coding. Just like you inject objects in other parts of your application, you can inject them into the Configure() method also. The later approach is bit complex and requires you to shift the seeding operation to the Main() from Program.cs.

Ok. Let's get going!

Modify your Configure() method signature as shown below :

public void Configure(IApplicationBuilder app,

IHostingEnvironment env,

UserManager<MyIdentityUser> userManager,

RoleManager<MyIdentityRole> roleManager)

{

...

...

}

As you can see we now inject UserManager and RoleManager into the Configure() method. This way you can use UserManager to create user accounts and RoleManager to create roles at the application startup. Instead of writing code directly inside the Configure() method, we will isolate it in a separate class and then call it from here.

So, add a new class called MyIdentityDataInitializer into the project. The following code shows the skeleton of this class :

public static class MyIdentityDataInitializer

{

public static void SeedData

(UserManager<MyIdentityUser> userManager,

RoleManager<MyIdentityRole> roleManager)

{

}

public static void SeedUsers

(UserManager<MyIdentityUser> userManager)

{

}

public static void SeedRoles

(RoleManager<MyIdentityRole> roleManager)

{

}

}

The MyIdentityDataInitializer class has three static methods - SeedRoles(), SeedUsers() and SeedData(). If you wish to create both - users and roles - then you would call SeedData(). You can also create only users or only roles using the respective methods.

Now, let's see the code that goes inside these methods. This code should be familiar to you because you already wrote something similar in the AccountController (see my earlier article [**here**](http://www.binaryintellect.net/articles/b957238b-e2dd-4401-bfd7-f0b8d984786d.aspx) for example).

The SeedRoles() creates desired default roles in the system and looks like this :

public static void SeedRoles

(RoleManager<MyIdentityRole> roleManager)

{

if (!roleManager.RoleExistsAsync

("NormalUser").Result)

{

MyIdentityRole role = new MyIdentityRole();

role.Name = "NormalUser";

role.Description = "Perform normal operations.";

IdentityResult roleResult = roleManager.

CreateAsync(role).Result;

}

if (!roleManager.RoleExistsAsync

("Administrator").Result)

{

MyIdentityRole role = new MyIdentityRole();

role.Name = "Administrator";

role.Description = "Perform all the operations.";

IdentityResult roleResult = roleManager.

CreateAsync(role).Result;

}

}

The SeedRoles() method accepts RoleManager as its parameter. Inside, it creates two roles in the system - NormalUser and Administrator. You should change the role names and their description as per your need. Note that we first check whether a role already exists or not. If it doesn't exist only then we create it.

The SeedUsers() creates desired default user accounts and looks like this :

public static void SeedUsers

(UserManager<MyIdentityUser> userManager)

{

if (userManager.FindByNameAsync

("user1").Result == null)

{

MyIdentityUser user = new MyIdentityUser();

user.UserName = "user1";

user.Email = "user1@localhost";

user.FullName = "Nancy Davolio";

user.BirthDate = new DateTime(1960, 1, 1);

IdentityResult result = userManager.CreateAsync

(user, "password\_goes\_here").Result;

if (result.Succeeded)

{

userManager.AddToRoleAsync(user,

"NormalUser").Wait();

}

}

if (userManager.FindByNameAsync

("user2").Result == null)

{

MyIdentityUser user = new MyIdentityUser();

user.UserName = "user2";

user.Email = "user2@localhost";

user.FullName = "Mark Smith";

user.BirthDate = new DateTime(1965, 1, 1);

IdentityResult result = userManager.CreateAsync

(user, "password\_goes\_here").Result;

if (result.Succeeded)

{

userManager.AddToRoleAsync(user,

"Administrator").Wait();

}

}

}

The SeedUsers() method accepts a UserManager and creates two users - user1 and user2. Notice that we first check whether a user with the same name exists or not. If it doesn't exist then we create it with default values of l, full name and birth date. Change these values as per your requirement. Also notice that the user accounts have been assigned certain default roles.

The SeedUsers() and SeedRoles() don't return any value. But you can change the signature to return some success or failure status if you so wish.

The SeedData() basically calls SeedRoles() and SeedUsers() and looks like this :

public static void SeedData

(UserManager<MyIdentityUser> userManager,

RoleManager<MyIdentityRole> roleManager)

{

SeedRoles(roleManager);

SeedUsers(userManager);

}

The SeedData() method accepts a UserManager and a RoleManager. Inside, it calls SeedRoles() and SeedUsers(). Note that SeedRoles() is called first because SeedUsers() assigns certain roles to the users being added and those roles must exist in the system prior to adding the users.

**Seeding data in Configure()**

Finally, it's time to use the MyIdentityDataInitializer class. Open Startup class, go to Configure() method and add this line of code :

public void Configure(IApplicationBuilder app,

IHostingEnvironment env,

UserManager<MyIdentityUser> userManager,

RoleManager<MyIdentityRole> roleManager)

{

...

**app.UseAuthentication();**

**MyIdentityDataInitializer.SeedData(userManager, roleManager);**

app.UseStaticFiles();

app.UseMvc(routes =>

{

routes.MapRoute(

name: "default",

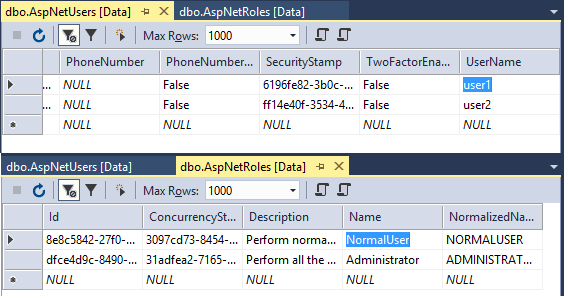
template: "{controller=Home}/{action=Index}/{id?}");

});

}

As you can see after calling UseAuthentication() we call SeedData() on MyIdentityDataInitializer.

Run the application and see whether the users get created in the database or not. The following figure shows these sample user accounts getting created in the database :



**Seeding data in Main()**

In the preceding code you created default users and roles in the Configure() method. You can also do that work in Main() method. You might want to use this later approach if you are also seeding data for EF core (won't go into those details here). The following code from Program.cs shows how the Main() code looks like :

public static void Main(string[] args)

{

var host = BuildWebHost(args);

using (var scope = host.Services.CreateScope())

{

**var serviceProvider = scope.ServiceProvider;**

try

{

**var userManager = serviceProvider.**

**GetRequiredService<UserManager<MyIdentityUser>>();**

**var roleManager = serviceProvider.**

**GetRequiredService<RoleManager<MyIdentityRole>>();**

MyIdentityDataInitializer.SeedData

(userManager, roleManager);

}

catch

{

}

}

host.Run();

}

Here, instead of injecting the UserManager and RoleManager into Configure() you  grab them from GetRequiredService() method. The userManager and roleManager objects are then passed to SeedData() as before.