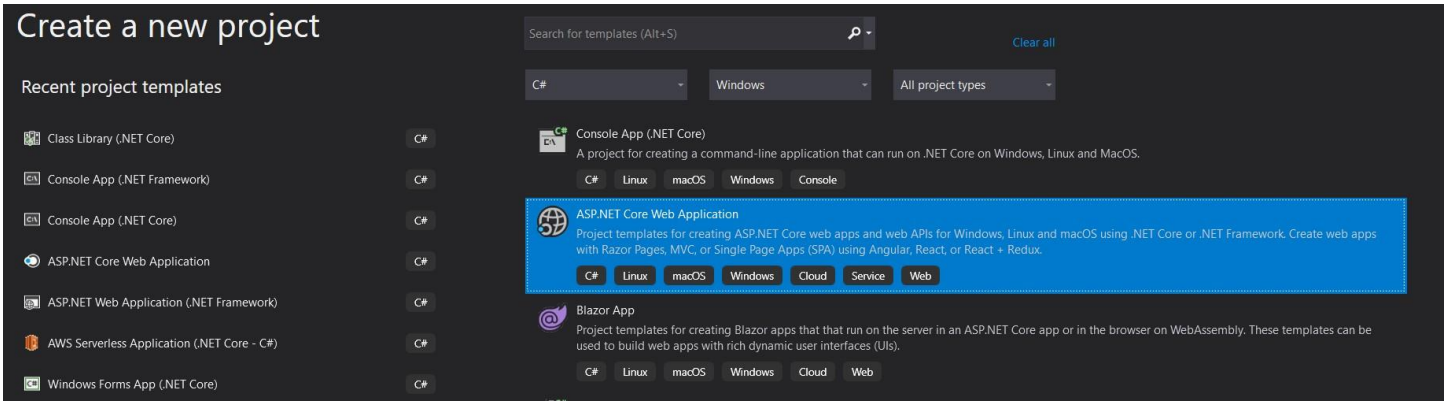
Identity in ASP.NET Core

Introducing Identity and Implementing Roles

By implementing roles in an ASP.NET MVC Web Application we can further separate the types of users on our application. This allows us to determine what information to show as well as restrict or allow access to parts of our application respectively.

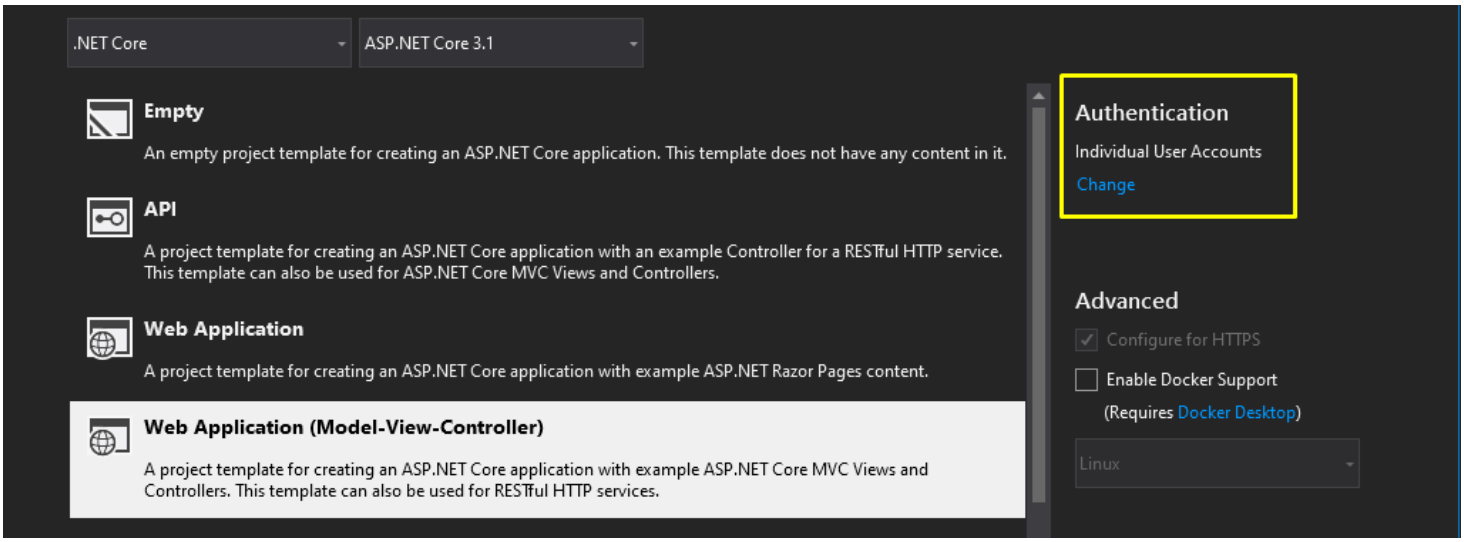
Create a new ASP.NET Core MVC Application

Open Visual Studio and select a new ASP.NET Core Web Application from the list of templates and select Next.



Name the project and hit create.

Select the Web Application (Model-View-Contoller) template from the list of options. Change the Authentication type from “No Authentication” to “Individual User Accounts” and click create.



Identity Review

ASP.NET Core Identity is an API that supports login functionality and manages user, passwords, roles, claims, tokens, and more. By incorporating Identity into an ASP.NET project we gain access to many helper functionalities and a popular database schema.

By creating an MVC application with Individual User Accounts Authentication, the template includes the Identity library. When adding Identity to an existing application that was created without the Individual User Account Authentication, you will need to install the "Microsoft.AspNetCore.Identity.EntityFrameworkCore" library from the NuGet Package Manager.

Scaffold Identity

In our case, we want to change some of the classes and views that are inside of the Identity library. Currently, we do not have access to those files because Identity is an external library, meaning its files do not show up in our project folders.

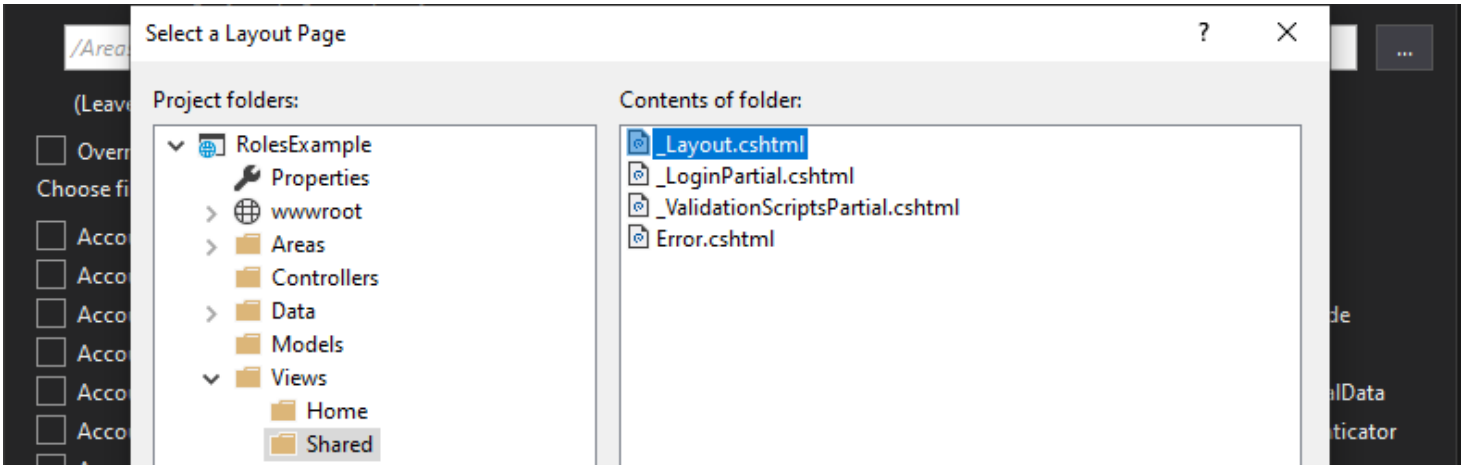
We can choose to have Visual Studio scaffold (generate) some of the files inside of the Identity library so we can edit them to add custom functionality.

Right click on the project name and select Add > New Scaffolded Item

From the left-hand side of the modal select Identity and then Identity in the main box.

After hitting Add you will see the Add Identity modal.

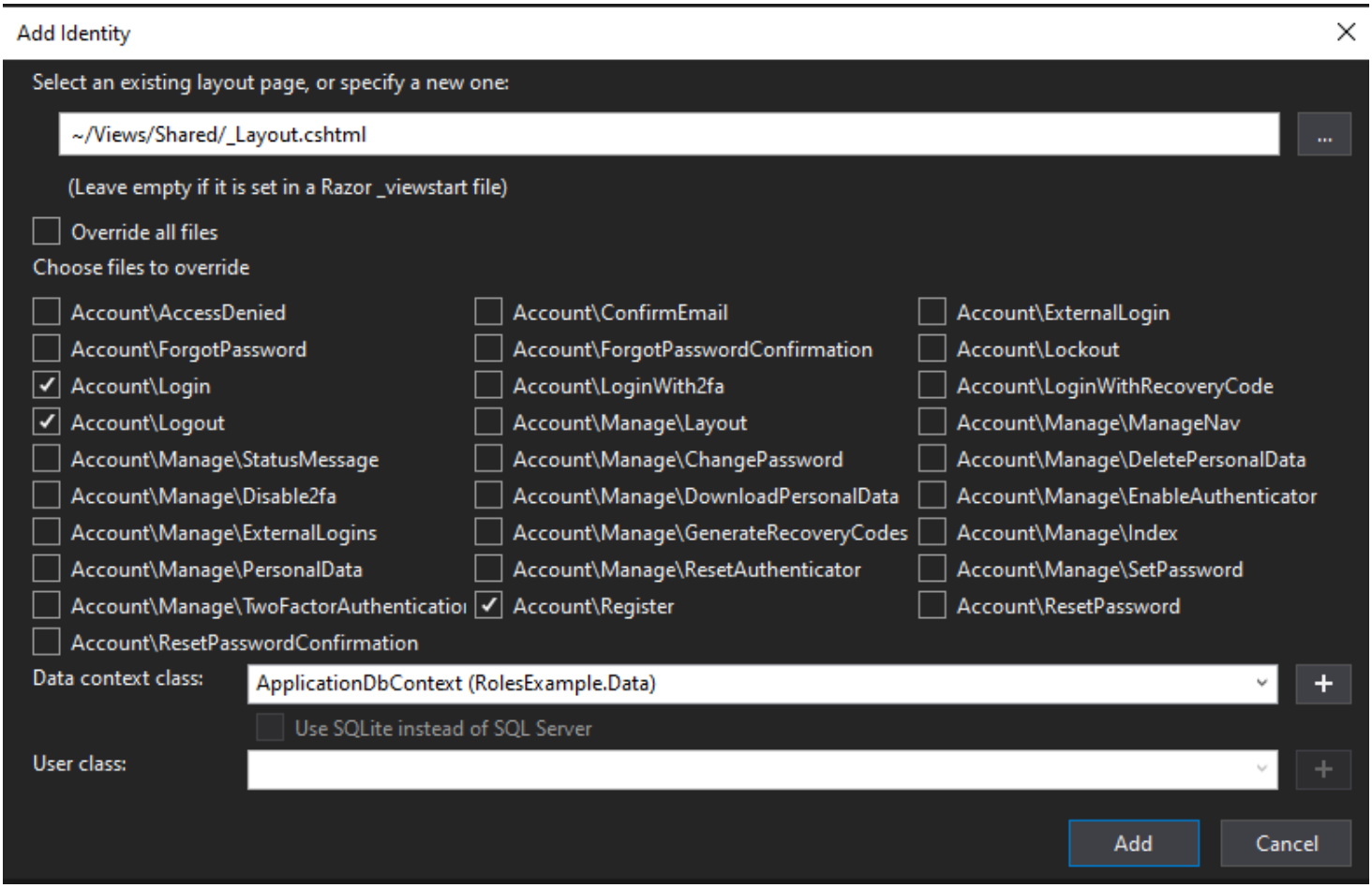
Select _Layout.cshtml for the "Select an existing layout page, or specify a new one:" input box. See photo below.



Next, check the boxes for:

- Account\Login
- Account\Logout
- Account\Register

These are the only files we will be editing therefore the only files we will scaffold from Identity. Finally, select "ApplicaitonDbContext" for the "Data context class:" input box and hit Add.



Razor Pages vs MVC

Microsoft scaffolds Identity as a Razor Class Library (Razor Pages). This is different from the MVC approach.

In the MVC approach, functionality is grouped by function, by what a thing does. Controllers contain actions, models contain data, and views display information.

In the Razor Pages approach, files are grouped by purpose. A razor Page is both view and method. The main difference is that the C# code that is tied to the view is grouped together with the .cshtml file.

We can view this difference inside of the folder Areas > Identity > Pages > Account

You will notice that the Login.cshtml, Logout.cshtml, and Register.cshtml files have arrows and are expandable. By expanding the files, we see that there are .cshtml.cs files attached to them.

Inside of these .cshtml.cs files there are two main methods: OnGetAsync() and OnPostAsync.

You can relate these methods to the methods you use in your controllers to handle showing a view and submitting data from a view (HttpPost).

The main difference you need to be aware of is that the properties on the Register.cshtml.cs file are the "view's" model properties.

You can read about the difference between MVC and Razor Pages here: https://exceptionnotfound.net/razor-pages-how-does-it-differ-from-mvc-in-asp-net-core/

Next Steps

Inside the Areas > Identity > Pages > Account folder is where we will be making the changes to implement roles into our application.

The steps we need to implement are (we will be doing these in the sections below):

1. Edit the Register.cshtml.cs file to add the RoleManager class to tie a user to a role when a user registers with the application
2. Edit the Register.cshtml file to include a drop-down list of the available roles to choose from when registering. (This would not be the case if this were a real application. In a real application a user would not have the ability to choose his or her own role. This would most likely be assigned by an admin or the user would be given direct links to registrations relative to the role type)
3. Seed the database with roles

Edit Register.cshtml.cs

We will begin by opening the Register.cshtml.cs file located in the Areas > Identity > Pages > Account folder.

By utilizing Dependency Injection, we are going to inject the RoleManager class into the RegisterModel class via its constructor.

Modify the global member variables and the RegisterModel class constructor to include the following.

```
        …
        private readonly IEmailSender _emailSender;
        private readonly RoleManager<IdentityRole> _roleManager;

        public RegisterModel(
            UserManager<IdentityUser> userManager,
            SignInManager<IdentityUser> signInManager,
            ILogger<RegisterModel> logger,
            IEmailSender emailSender,
            RoleManager<IdentityRole> roleManager)
        {
            _userManager = userManager;
            _signInManager = signInManager;
            _logger = logger;
            _emailSender = emailSender;
            _roleManager = roleManager;
        }

        …
```

We have now injected the RoleManager class into the RegisterModel class for us to utilize.

Next, lets modify the InputModel class located inside of the RegisterModel class. We are going to add a new property that we expect back from the view. That way we can edit our Register.cshtml view to include this new input and option. This is the model you would change if you were to add further changes to the registration form.

Add the following additional property to your InputModel class:

```
            …
            public string ConfirmPassword { get; set; }

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

        }
```

Next, we need to add an additional property to the RegisterModel class. We will use this property to populate a drop-down list on our register html page so the user can choose what role to assign to their account.

Add the following lines of code to the RegisterModel class:

```
        …
        [BindProperty]
        public InputModel Input { get; set; }
        public SelectList Roles { get; set; }

        …
```

You will need to bring in a new using statement to gain access to the SelectList data type.

Next, we need to edit the OnGetAsync method within the RegisterModel class to grab all of the roles in our database and initialize our new Roles property with some values for the drop-down list.

Inside the OnGetAsync method add:

```
        public async Task OnGetAsync(string returnUrl = null)
        {
            ReturnUrl = returnUrl;
            ExternalLogins = (await
_signInManager.GetExternalAuthenticationSchemesAsync()).ToList();
            var roles = _roleManager.Roles;
            Roles = new SelectList(roles, "Name", "Name");
        }
```

Lastly, we need to edit the OnPostAsync method to add the new user role relationship to the database when a new user is successfully created.

Go to the OnPostAsync method and add:

```
      …
      if (result.Succeeded)
      {
            if(await _roleManager.RoleExistsAsync(Input.Role))
            {
                await _userManager.AddToRoleAsync(user, Input.Role);
            }
            _logger.LogInformation("User created a new account with
            password.");

       …
```

We have now made all the changes needed to the Register.cshtml.cs file. Every time a new user registers, we confirm the role selected from the drop-down list exists in the database. If the role does exist, we call the AddToRoleAsync method on the UserManager class and pass in the user and role name. The AddToRoleAsync method then creates a new record in the AspNetUserRoles table which creates a relationship between a user and role.

Editing Services to Reflect RoleManager

Now that we are injecting the RoleManager into the RegisterModel class, we need to update the services to reflect the changes. At the point of first receiving this tutorial, you will not understand what services are. This lecture will come later in the course. For now, implement the changes in the Startup.cs class within the ConfigureServices method.

```
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<ApplicationDbContext>(options =>
                options.UseSqlServer(
                    Configuration.GetConnectionString("DefaultConnection")));

                  services.AddIdentity<IdentityUser, IdentityRole>(options =>
      options.SignIn.RequireConfirmedAccount = false)
                .AddEntityFrameworkStores<ApplicationDbContext>()
                .AddDefaultUI()
                .AddDefaultTokenProviders();

            services.AddControllersWithViews();
            services.AddRazorPages();
        }
```

Notice how we changed the value for RequireConfirmedAccount equal to false (…`SignIn.RequireConfirmedAccount = false).` Out of the box, the Identity template forces the user to verify their email. We need to turn this off, so we don't run into problems later on.

Now we can inject the RoleManager class into classes that request it.

Editing the Register.cshtml View

Now that we require a user to select a role when registering, we need to add a drop-down list input to our register form.

Add the following code to the Register.cshtml file:

```
…
<div class="form-group">
      <label asp-for="Input.ConfirmPassword"></label>
      <input asp-for="Input.ConfirmPassword" class="form-control" />
      <span asp-validation-for="Input.ConfirmPassword" class="text-
      danger"></span>
</div>
<div class="form-group">
      <label asp-for="Input.Role"></label>
      <select asp-for="Input.Role" class="form-control" asp-
      items="@Model.Roles"></select>
      <span asp-validation-for="Input.Role" class="text-danger"></span>
</div>
<button type="submit" class="btn btn-primary">Register</button>

…
```

Seeding Roles in the Database

We can use the ModelBuilder class which is available in the OnModelCreating method in our ApplicationDbContext class. We must override the OnModelCreating method from the IdentityDbContext class which is the parent of our ApplicationDbContext class. This gives us a variety of EF Core configurations options that we can use while configuring our entities. For our purposes, we are going to use the ModelBuilder class to seed some roles into our database.

Add the following method to your ApplicationDbContext class:

```
public class ApplicationDbContext : IdentityDbContext
    {
        public
ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
            : base(options)
        {
        }

        protected override void OnModelCreating(ModelBuilder builder)
        {
            base.OnModelCreating(builder);

            builder.Entity<IdentityRole>()
                .HasData(
                    new IdentityRole
                    {
                        Name = "Admin",
                        NormalizedName = "ADMIN"
                    }
                );
        }
    }
```

This is the preferred way to seed data into our database. We can further improve the organization of our code by creating sub configuration classes to handle the seed data for each table. The next time we run a migration, it will seed this data into our database. If you have not run a migration yet, do so now. (There may be nothing in your up and down methods within the new migration scaffolding. That is okay.)

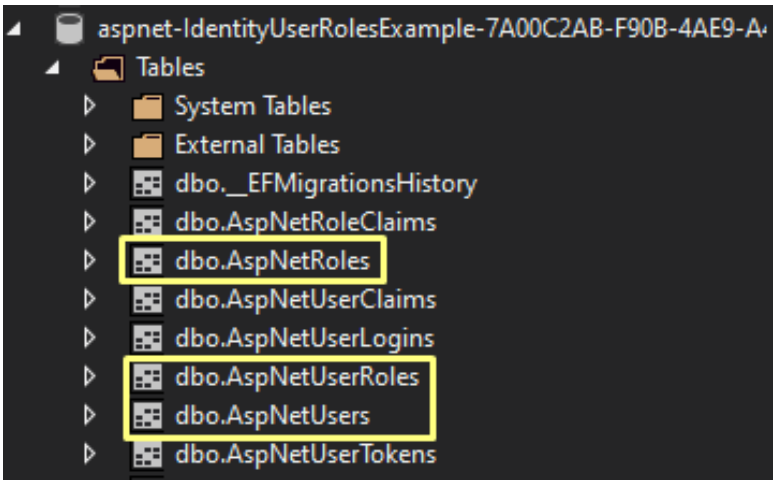
Enter the following commands inside of the Package Manager Console.

Add-Migration Initial

Update-Database

Identity Users and Roles Tables

By taking a look at the database after the migration, we can see multiple tables have been created. Most importantly, take notice of the AspNetRoles, AspNetUserRoles, and AspNetUsers tables.



AspNetRoles

AspNetRoles table holds the data pertaining to the roles that exist in our database. The model that represents this table is IdentityRole. If you look at the Seeding Roles in the Database section, we are seeding the admin role into this AspNetRoles table.

Table preview:

Id	Name	NormalizedName	ConcurrencyStamp
76a38594-a9e5-430f-9399-22399677e131	Admin	ADMIN	a8483e81-7497-41b8-bb3b-3033149752e4

AspNetUsers

The AspNetUsers table holds the users that register from the Register.cshtml view form. If you take a look at the Register.cshtml.cs file within the RegisterModel class inside the OnPostAsync method, you will find this code:

```
if (ModelState.IsValid)
{
    var user = new IdentityUser { UserName = Input.Email, Email = Input.Email };
    var result = await _userManager.CreateAsync(user, Input.Password);
    if (result.Succeeded)
    {
        if (await _roleManager.RoleExistsAsync(Input.Role))
        {
            await _userManager.AddToRoleAsync(user, Input.Role);
        }
```

Outlined in yellow are the two lines of code that take the input from the Register.cshtml view form, create a new IdentityUser object, and add the new IdentityUser to the AspNetUsers table.

Table preview:

Id (string)	UserName	NormalizedUserName	Email	…

0ca45457-9c21-4bbc-b8bf-370c1ac8c8a3	david@awesome.com	DAVID@AWESOME.COM	david@awesome.com	…

AspNetUserRoles

AspNetUserRoles is a junction table that holds relationships between a user and role. Inside of the RegisterModel class OnPostAsync method we are first checking if the selected role from the drop-down list exists, and if it does, add a new row in the AspNetUserRoles table with the user and role ids. Outlined in red are the lines of code responsible for this functionality:

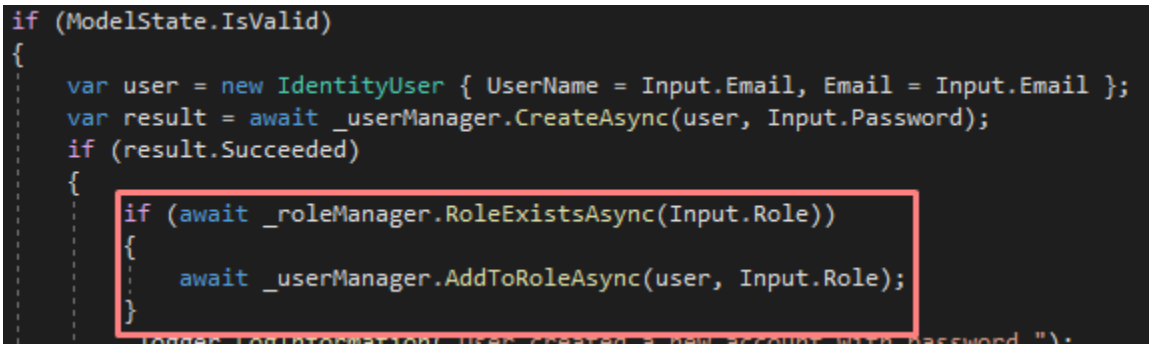


Table preview:

UserId	**RoleId**
0ca45457-9c21-4bbc-b8bf-370c1ac8c8a3	76a38594-a9e5-430f-9399-22399677e131

You can see in the table preview, the id for user david@awesome.com is tied to the id for role Admin. This is what creates the relationship between the user and the role.

Accessing Current Signed-In User

We can gain access to information about the user that is currently signed in by using the following line of code:

```
var userId = this.User.FindFirstValue(ClaimTypes.NameIdentifier);
```

If a user is signed in, this line of code will return the users primary key id from the AspNetUsers table. If a user is not signed in, userId will be null.

This will be used to query the database for information specific to the currently signed in user.

Adding Foreign Key Reference to IdentityUser

There will be many cases where you will need to add information to the database that is tied to a specific user that is signed in. Since all of our users that register are stored in the AspNetUsers table, which utilizes the IdentityUser model, any other data that needs to be stored with a relationship to the user must store a connection to the IdentityUser's model id. Lets say we have a Customer class that represents a customer using our application, and we want to add a row in the Customers table that has a relationship to the current signed in user, then we would need to add a foreign key reference from the AspNetUsers table to our Customers table.

In our Customer model, we would include the following code:

```
public class Customer
{
    [Key]
    public int Id { get; set; }
    public string Name { get; set; }

    [ForeignKey("IdentityUser")]
    public string IdentityUserId { get; set; }
    public IdentityUser IdentityUser { get; set; }
}
```

When a customer is being added to the database, we must now get the current signed in user's id and attach it to our Customer model before adding it as a new record in the table.

An example of this code would look something like this:

```
var userId = this.User.FindFirstValue(ClaimTypes.NameIdentifier);
customer.IdentityUserId = userId;
_context.Add(customer);
_context.SaveChanges();
```

Now, if we needed to get information from the Customer table that was directly related to the user who is currently signed in, we would do something like this:

```
var userId = this.User.FindFirstValue(ClaimTypes.NameIdentifier);
var customer = _context.Customers.Where(c => c.IdentityUserId ==
userId).SingleOrDefault();
```

Routing Based on User Role Action Filter

Now that we have successfully implemented roles within our application, the next thing we want to do is route the user to the correct home page based on what role the user is tied to. We are going to be creating a new global routing configuration that all our controllers will utilize. If a user is signed in, we ensure that they are directed to their respective home page. We are going to do this by creating a custom Action Filter. We use Action Filters to extract code which can be reused and allows us to make our actions cleaner and more maintainable.

There are several built-in action filters given to us by .NET. Some of these include:

- Authorization Filters – Help determine authorized access to the requested resource or page
- Resource Filters – Run after authorization and are used for performance and caching
- Action Filters – Used to run code right before and after the execution of action methods
- Exception Filters – Used to handle exceptions
- Result Filters – like action filters, before and after the invoking of action methods results

We are going to create our own custom Action Filter. In order to create a custom Action Filter, we need to create a class that inherits from one of the following: IActionFilter, IAsyncActionFilter, or ActionFilterAttribute. We will then be forced to implement two methods: OneActionExecuting and OnActionExecuted.

Create a new folder in your project and name it ActionFilters. Create a new class called GlobalRouting that inherits from IActionFilter and add the following code:

```
public class GlobalRouting : IActionFilter
{
    private readonly ClaimsPrincipal _claimsPrincipal;
    public GlobalRouting(ClaimsPrincipal claimsPrincipal)
    {
        _claimsPrincipal = claimsPrincipal;
    }

    public void OnActionExecuting(ActionExecutingContext context)
    {
        var controller = context.RouteData.Values["controller"];
        if(controller.Equals("Home"))
        {
            if (_claimsPrincipal.IsInRole("Customer"))
            {
                context.Result = new RedirectToActionResult("Index",
                            "Customers", null);
            }
            else if (_claimsPrincipal.IsInRole("Employee"))
            {
                context.Result = new RedirectToActionResult("Index",
                     "Employees", null);
            }
        }
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {

    }
}
```

We are putting logic inside of the OnActionExecuting method because we want this code to run before an action method is executed. Each method has a parameter of type ActionExecut-ing/edContext. This gives us access to the context of the request. Inside of our logic we are accessing what controller the request is trying to access. In this case, we are assuming that the Home controller is only being used for users that are not signed in. If they are signed in, then we want to direct the user to their respective home page that is specific to what role that user is attached to. If a user signs in and has a role of Customer, then rather than having them access the Home controller (which is used for non-signed in users) we send them to the CustomersController home index page. Same thing with if a user is attached to the role "Employee".

Notice how we are injecting the ClaimsPrinciple class into this GlobalRouting action filter. We will need to register ClaimsPrinciple as a scoped service to our application to user it with dependency injection. We will do this in a moment.

There are multiple ways we can use this new custom Action Filter. We can decide to use it for specific controllers only, for specific action methods only, or globally for all controllers and actions methods.

To use for all action methods on a specific controller add this code as an attribute above the controller class (do not do this yet):

```
[ServiceFilter(typeof(GlobalRouting))]
public class HomeController : Controller
{
```

Or if you want to use it for specific action methods (do not do this yet):

```
[ServiceFilter(typeof(GlobalRouting))]
public IActionResult Index()
{
```

We are going to set it up to be used globally between all controllers. This will allow you to add custom logic pertaining to your specific application if you so please.

In order to add it as a global action filter, we need to add some configuration to all controllers within the ConfigureServices method inside of the Startup class.

Add the following code to the ConfigureServices method within the Startup class:

```
    …
    .AddDefaultUI()
    .AddDefaultTokenProviders();

    services.AddScoped<ClaimsPrincipal>(s =>
          s.GetService<IHttpContextAccessor>().HttpContext.User);
    services.AddControllers(config =>
    {
          config.Filters.Add(typeof(GlobalRouting));
    });

    …
```

First, we are adding ClaimsPrincipal to our services so we can successfully inject it into our GlobalRouting action filter. Next we are adding the action filter to every controller using services.AddControllers().

Now, every time we execute an action method, our GlobalRouting OnActionExecuting method will run before the action method and redirect the user to the correct page if need be.

Authorize Attribute

We can use the Authorize attribute above methods and controllers to either allow or deny access to a page or resource. In our case, because our application is utilizing roles, we will use the Authorize attribute to check whether a user has the appropriate role to access the specified controllers and action methods.

If you have a CustomersController that is used for users that are attached to a Customer role, we can choose to only allow users that have the Customer role attached to them to access the controller and its action methods. We can do this by placing the following code above the controller class:

```
[Authorize(Roles = "Customer")]
public class CustomersController : Controller
{
```

If any user tries to access the CustomersController that does not have the role of Customer, they will be either re-directed to the login page, or shown an Access Denied page. If you have other controllers you would like only authorized users to access, you can use this same concept checking for the specified required role.

Conclusion

Implementing Roles is very easy using ASP.NET Identity. In this tutorial we have learned:

- Introduction to ASP.NET Core Identity
- Scaffold and Override Identity Library Files
- Edit the RegisterModel Properties and Form Model
- Implement a SelectList with a Drop-down Select Form
- Register RoleManager with the IOC Container
- Seed Roles into a Database
- Getting current signed in user's IdentityUser id
- Adding a foreign key reference to the IdentityUser model (AspNetUsers table)
- Routing by creating a custom action filter
- Authorization filter attribute

Identity is a vast library with much more functionalities than shown in this tutorial. If you would like to read more about Identity visit: https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-3.1&tabs=visual-studio