Introduction to Identity on ASP.NET Core

07/15/2020 • 17 minutes to read • +20

In this article

- Create a Web app with authentication
- Scaffold Register, Login, LogOut, and RegisterConfirmation
- Test Identity
- Identity Components
- Migrating to ASP.NET Core Identity
- Setting password strength
- AddDefaultIdentity and AddIdentity
- Prevent publish of static Identity assets
- Next Steps

By Rick Anderson

ASP.NET Core Identity:

- Is an API that supports user interface (UI) login functionality.
- Manages users, passwords, profile data, roles, claims, tokens, email confirmation, and more.

Users can create an account with the login information stored in Identity or they can use an external login provider. Supported external login providers include Facebook, Google, Microsoft Account, and Twitter.

For information on how to globally require all users to be authenticated, see Require authenticated users.

The Identity source code is available on GitHub. Scaffold Identity and view the generated files to review the template interaction with Identity.

Identity is typically configured using a SQL Server database to store user names, passwords, and profile data. Alternatively, another persistent store can be used, for example, Azure Table Storage.

In this topic, you learn how to use Identity to register, log in, and log out a user. Note: the templates treat username and email as the same for users. For more detailed instructions about creating apps that use Identity, see Next Steps.

Microsoft identity platform is:

- An evolution of the Azure Active Directory (Azure AD) developer platform.
- Unrelated to ASP.NET Core Identity.

ASP.NET Core Identity adds user interface (UI) login functionality to ASP.NET Core web apps. To secure web APIs and SPAs, use one of the following:

- Azure Active Directory
- Azure Active Directory B2C (Azure AD B2C)
- IdentityServer4

IdentityServer4 is an OpenID Connect and OAuth 2.0 framework for ASP.NET Core. IdentityServer4 enables the following security features:

- Authentication as a Service (AaaS)
- Single sign-on/off (SSO) over multiple application types
- Access control for APIs
- Federation Gateway

For more information, see Welcome to IdentityServer4.

View or download the sample code (how to download).

Create a Web app with authentication

Create an ASP.NET Core Web Application project with Individual User Accounts.

Visual Studio | .NET Core CLI

- Select **File** > **New** > **Project**.
- Select **ASP.NET Core Web Application**. Name the project **WebApp1** to have the same namespace as the project download. Click **OK**.
- Select an ASP.NET Core **Web Application**, then select **Change Authentication**.
- Select **Individual User Accounts** and click **OK**.

The generated project provides ASP.NET Core Identity as a Razor Class Library. The Identity Razor Class Library exposes endpoints with the `Identity` area. For example:

- /Identity/Account/Login
- /Identity/Account/Logout
- /Identity/Account/Manage

Apply migrations

Apply the migrations to initialize the database.

Visual Studio | .NET Core CLI

Run the following command in the Package Manager Console (PMC):

```
PM> Update-Database
```

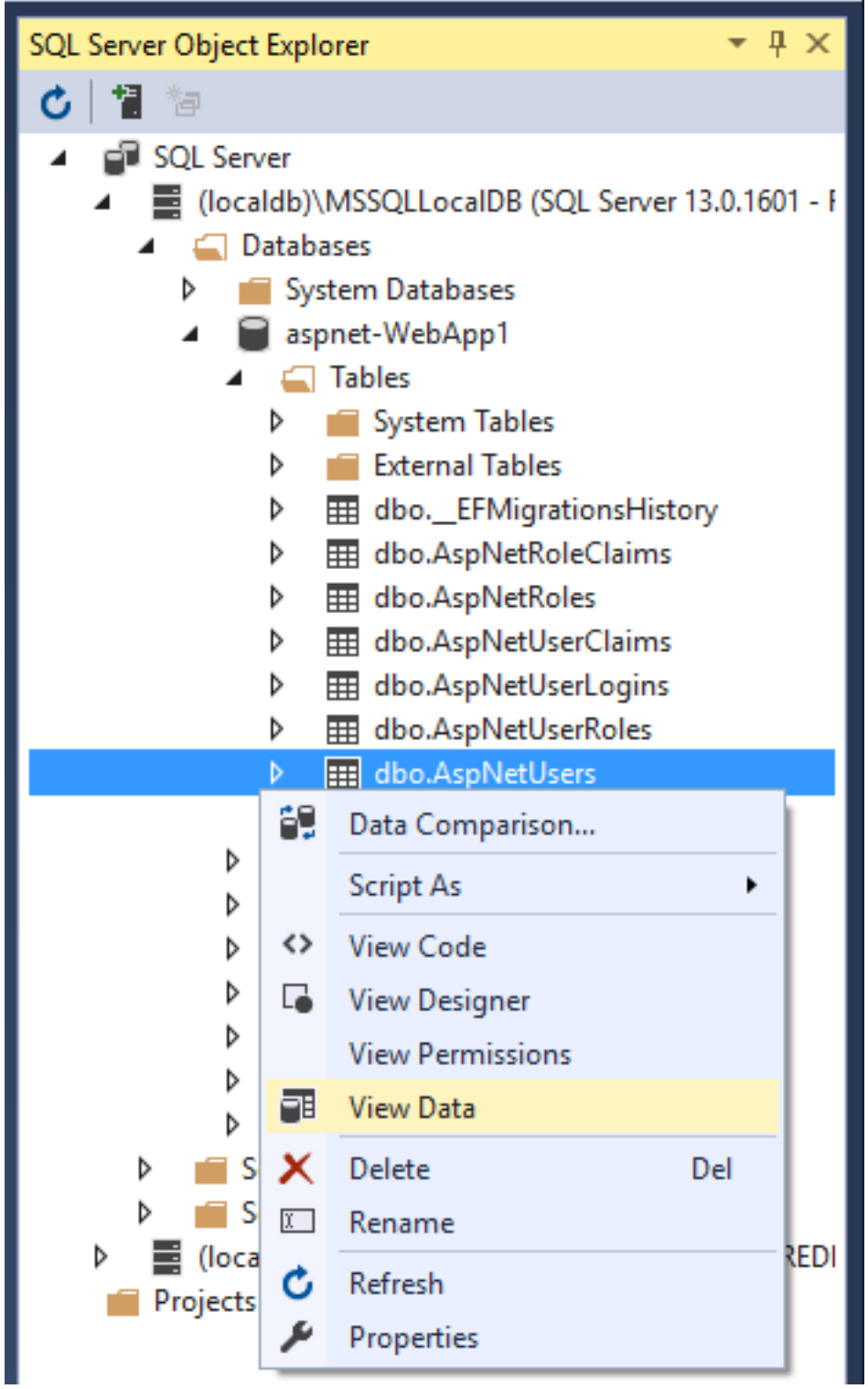
Test Register and Login

Run the app and register a user. Depending on your screen size, you might need to select the navigation toggle button to see the **Register** and **Login** links.

View the Identity database

Visual Studio | .NET Core CLI

- From the **View** menu, select **SQL Server Object Explorer** (SSOX).
- Navigate to **(localdb)MSSQLLocalDB(SQL Server 13)**. Right-click on **dbo.AspNetUsers** > **View Data**:



Configure Identity services

Services are added in `ConfigureServices`. The typical pattern is to call all the `Add{Service}` methods, and then call all the `services.Configure{Service}` methods.

C# Copy

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        // options.UseSqlite(
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDatabaseDeveloperPageExceptionFilter();
    services.AddDefaultIdentity<IdentityUser>(options => options.Sign-
In.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationDbContext>();
```

```
    services.AddRazorPages();

    services.Configure<IdentityOptions>(options =>
    {
        // Password settings.
        options.Password.RequireDigit = true;
        options.Password.RequireLowercase = true;
        options.Password.RequireNonAlphanumeric = true;
        options.Password.RequireUppercase = true;
        options.Password.RequiredLength = 6;
        options.Password.RequiredUniqueChars = 1;

        // Lockout settings.
        options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(5);
        options.Lockout.MaxFailedAccessAttempts = 5;
        options.Lockout.AllowedForNewUsers = true;

        // User settings.
        options.User.AllowedUserNameCharacters =
        "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTU-
VWXYZ0123456789-._@+";
        options.User.RequireUniqueEmail = false;
    });

    services.ConfigureApplicationCookie(options =>
    {
        // Cookie settings
        options.Cookie.HttpOnly = true;
        options.ExpireTimeSpan = TimeSpan.FromMinutes(5);

        options.LoginPath = "/Identity/Account/Login";
        options.AccessDeniedPath = "/Identity/Account/AccessDenied";
        options.SlidingExpiration = true;
    });
}
```

The preceding code configures Identity with default option values. Services are made available to the app through dependency injection.

Identity is enabled by calling UseAuthentication. `UseAuthentication` adds authentication middleware to the request pipeline.

C# Copy

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseMigrationsEndPoint();
    }
```

```
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

The template-generated app doesn't use authorization. `app.UseAuthorization` is included to ensure it's added in the correct order should the app add authorization. `UseRouting`, `UseAuthentication`, `UseAuthorization`, and `UseEndpoints` must be called in the order shown in the preceding code.

For more information on `IdentityOptions` and `Startup`, see IdentityOptions and Application Startup.

Scaffold Register, Login, LogOut, and RegisterConfirmation

Visual Studio | .NET Core CLI

Add the `Register`, `Login`, `LogOut`, and `RegisterConfirmation` files. Follow the Scaffold identity into a Razor project with authorization instructions to generate the code shown in this section.

Examine Register

When a user clicks the **Register** button on the `Register` page, the `RegisterModel.OnPostAsync` action is invoked. The user is created by CreateAsync on the `_userManager` object:

C# Copy

```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    returnUrl = returnUrl ?? Url.Content("~/");
    ExternalLogins = (await _signInManager.GetExternalAuthentication-
SchemesAsync())
                                            .ToList();
    if (ModelState.IsValid)
    {
        var user = new IdentityUser { UserName = Input.Email, Email = In-
put.Email };
        var result = await _userManager.CreateAsync(user,
Input.Password);
        if (result.Succeeded)
        {
            _logger.LogInformation("User created a new account with pass-
word.");

            var code = await _userManager.GenerateEmailConfirmationToken-
Async(user);
            code = WebEncoders.Base64UrlEncode(Encoding.UTF8.Get-
Bytes(code));
            var callbackUrl = Url.Page(
                "/Account/ConfirmEmail",
                pageHandler: null,
                values: new { area = "Identity", userId = user.Id, code =
code },
                protocol: Request.Scheme);

            await _emailSender.SendEmailAsync(Input.Email, "Confirm your
email",
                $"Please confirm your account by <a href='{HtmlEn-
coder.Default.Encode(callbackUrl)}'>clicking here</a>.");

            if (_userManager.Options.SignIn.RequireConfirmedAccount)
            {
                return RedirectToPage("RegisterConfirmation",
                                      new { email = Input.Email });
            }
            else
            {
                await _signInManager.SignInAsync(user, isPersistent:
false);
                return LocalRedirect(returnUrl);
            }
        }
        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(string.Empty, error.Description);
        }
    }
```

```
    // If we got this far, something failed, redisplay form
    return Page();
}
```

Disable default account verification

With the default templates, the user is redirected to the `Account.RegisterConfirmation` where they can select a link to have the account confirmed. The default `Account.RegisterConfirmation` is used ***only*** for testing, automatic account verification should be disabled in a production app.

To require a confirmed account and prevent immediate login at registration, set `DisplayConfirmAccountLink = false` in */Areas/Identity/Pages/Account/RegisterConfirmation.cshtml.cs*:

C# Copy

```
[AllowAnonymous]
public class RegisterConfirmationModel : PageModel
{
    private readonly UserManager<IdentityUser> _userManager;
    private readonly IEmailSender _sender;

    public RegisterConfirmationModel(UserManager<IdentityUser> userManag-
er, IEmailSender sender)
    {
        _userManager = userManager;
        _sender = sender;
    }

    public string Email { get; set; }

    public bool DisplayConfirmAccountLink { get; set; }

    public string EmailConfirmationUrl { get; set; }

    public async Task<IActionResult> OnGetAsync(string email, string re-
turnUrl = null)
    {
        if (email == null)
        {
            return RedirectToPage("/Index");
        }

        var user = await _userManager.FindByEmailAsync(email);
        if (user == null)
        {
            return NotFound($"Unable to load user with email
```

```
'{email}'.");
        }

        Email = email;
        // Once you add a real email sender, you should remove this code
that lets you confirm the account
        DisplayConfirmAccountLink = false;
        if (DisplayConfirmAccountLink)
        {
            var userId = await _userManager.GetUserIdAsync(user);
            var code = await _userManager.GenerateEmailConfirmationToken-
Async(user);
            code = WebEncoders.Base64UrlEncode(Encoding.UTF8.Get-
Bytes(code));
            EmailConfirmationUrl = Url.Page(
                "/Account/ConfirmEmail",
                pageHandler: null,
                values: new { area = "Identity", userId = userId, code =
code, returnUrl = returnUrl },
                protocol: Request.Scheme);
        }

        return Page();
    }
}
```

Log in

The Login form is displayed when:

- The **Log in** link is selected.
- A user attempts to access a restricted page that they aren't authorized to access **or** when they haven't been authenticated by the system.

When the form on the Login page is submitted, the `OnPostAsync` action is called. `PasswordSignInAsync` is called on the `_signInManager` object.

C# Copy

```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    returnUrl = returnUrl ?? Url.Content("~/");

    if (ModelState.IsValid)
    {
        // This doesn't count login failures towards account lockout
        // To enable password failures to trigger account lockout,
        // set lockoutOnFailure: true
```

```
        var result = await _signInManager.PasswordSignInAsync(Input.E-
mail,
                                Input.Password, Input.RememberMe, lockoutOn-
Failure: true);
        if (result.Succeeded)
        {
            _logger.LogInformation("User logged in.");
            return LocalRedirect(returnUrl);
        }
        if (result.RequiresTwoFactor)
        {
            return RedirectToPage("./LoginWith2fa", new
            {
                ReturnUrl = returnUrl,
                RememberMe = Input.RememberMe
            });
        }
        if (result.IsLockedOut)
        {
            _logger.LogWarning("User account locked out.");
            return RedirectToPage("./Lockout");
        }
        else
        {
            ModelState.AddModelError(string.Empty, "Invalid login at-
tempt.");
            return Page();
        }
    }

    // If we got this far, something failed, redisplay form
    return Page();
}
```

For information on how to make authorization decisions, see Introduction to authorization in ASP.NET Core.

Log out

The **Log out** link invokes the `LogoutModel.OnPost` action.

C# Copy

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.Extensions.Logging;
using System.Threading.Tasks;
```

```
namespace WebApp1.Areas.Identity.Pages.Account
{
    [AllowAnonymous]
    public class LogoutModel : PageModel
    {
        private readonly SignInManager<IdentityUser> _signInManager;
        private readonly ILogger<LogoutModel> _logger;

        public LogoutModel(SignInManager<IdentityUser> signInManager,
ILogger<LogoutModel> logger)
        {
            _signInManager = signInManager;
            _logger = logger;
        }

        public void OnGet()
        {
        }

        public async Task<IActionResult> OnPost(string returnUrl = null)
        {
            await _signInManager.SignOutAsync();
            _logger.LogInformation("User logged out.");
            if (returnUrl != null)
            {
                return LocalRedirect(returnUrl);
            }
            else
            {
                return RedirectToPage();
            }
        }
    }
}
```

In the preceding code, the code `return RedirectToPage();` needs to be a redirect so that the browser performs a new request and the identity for the user gets updated.

SignOutAsync clears the user's claims stored in a cookie.

Post is specified in the *Pages/Shared/_LoginPartial.cshtml*:

CSHTML Copy

```
@using Microsoft.AspNetCore.Identity
@inject SignInManager<IdentityUser> SignInManager
@inject UserManager<IdentityUser> UserManager

<ul class="navbar-nav">
@if (SignInManager.IsSignedIn(User))
{
```

```
    <li class="nav-item">
        <a  class="nav-link text-dark" asp-area="Identity" asp-page="/Ac-
count/Manage/Index"
                                                title="Manage">Hello
@User.Identity.Name!</a>
    </li>
    <li class="nav-item">
        <form class="form-inline" asp-area="Identity" asp-page="/Ac-
count/Logout"
                                    asp-route-returnUrl="@Url.Page("/", new
{ area = "" })"
                                    method="post" >
            <button  type="submit" class="nav-link btn btn-link text-
dark">Logout</button>
        </form>
    </li>
}
else
{
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="Identity" asp-page="/Ac-
count/Register">Register</a>
    </li>
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="Identity" asp-page="/Ac-
count/Login">Login</a>
    </li>
}
</ul>
```

Test Identity

The default web project templates allow anonymous access to the home pages. To test Identity, add [Authorize]:

C# Copy

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.Extensions.Logging;

namespace WebApp1.Pages
{
    [Authorize]
    public class PrivacyModel : PageModel
    {
        private readonly ILogger<PrivacyModel> _logger;

        public PrivacyModel(ILogger<PrivacyModel> logger)
        {
```

```
            _logger = logger;
        }

        public void OnGet()
        {
        }
    }
}
```

If you are signed in, sign out. Run the app and select the **Privacy** link. You are redirected to the login page.

Explore Identity

To explore Identity in more detail:

- Create full identity UI source
- Examine the source of each page and step through the debugger.

Identity Components

All the Identity-dependent NuGet packages are included in the ASP.NET Core shared framework.

The primary package for Identity is Microsoft.AspNetCore.Identity. This package contains the core set of interfaces for ASP.NET Core Identity, and is included by `Microsoft.AspNetCore.Identity.EntityFrameworkCore`.

Migrating to ASP.NET Core Identity

For more information and guidance on migrating your existing Identity store, see Migrate Authentication and Identity.

Setting password strength

See Configuration for a sample that sets the minimum password requirements.

AddDefaultIdentity and AddIdentity

AddDefaultIdentity was introduced in ASP.NET Core 2.1. Calling `AddDefaultIdentity` is similar to calling the following:

- AddIdentity
- AddDefaultUI
- AddDefaultTokenProviders

See AddDefaultIdentity source for more information.

Prevent publish of static Identity assets

To prevent publishing static Identity assets (stylesheets and JavaScript files for Identity UI) to the web root, add the following `ResolveStaticWebAssetsInputsDependsOn` property and `RemoveIdentityAssets` target to the app's project file:

XML Copy

```xml
<PropertyGroup>
  <ResolveStaticWebAssetsInputsDependsOn>RemoveIdentityAssets</Resolve-
StaticWebAssetsInputsDependsOn>
</PropertyGroup>

<Target Name="RemoveIdentityAssets">
  <ItemGroup>
    <StaticWebAsset Remove="@(StaticWebAsset)" Condition="%(SourceId) ==
'Microsoft.AspNetCore.Identity.UI'" />
  </ItemGroup>
</Target>
```

Next Steps

- ASP.NET Core Identity source code
- See this GitHub issue for information on configuring Identity using SQLite.
- Configure Identity
- Create an ASP.NET Core app with user data protected by authorization
- Add, download, and delete user data to Identity in an ASP.NET Core project
- Enable QR Code generation for TOTP authenticator apps in ASP.NET Core
- Migrate Authentication and Identity to ASP.NET Core
- Account confirmation and password recovery in ASP.NET Core
- Two-factor authentication with SMS in ASP.NET Core
- Host ASP.NET Core in a web farm

Is this page helpful?

Yes No