When the browser closes it automatically deletes session based cookies (non-persistent cookies), but no cookies are cleared when an individual tab is closed. The server is not notified of tab or browser close events.

React to back-end changes

Once a cookie is created, the cookie is the single source of identity. If a user account is disabled in back-end systems:

- The app's cookie authentication system continues to process requests based on the authentication cookie.
- The user remains signed into the app as long as the authentication cookie is valid.

The ValidatePrincipal event can be used to intercept and override validation of the cookie identity. Validating the cookie on every request mitigates the risk of revoked users accessing the app.

One approach to cookie validation is based on keeping track of when the user database changes. If the database hasn't been changed since the user's cookie was issued, there's no need to re-authenticate the user if their cookie is still valid. In the sample app, the database is implemented in `IUserRepository` and stores a `LastChanged` value. When a user is updated in the database, the `LastChanged` value is set to the current time.

In order to invalidate a cookie when the database changes based on the `LastChanged` value, create the cookie with a `LastChanged` claim containing the current `LastChanged` value from the database:

C#

```
var claims = new List<Claim>
{
    new Claim(ClaimTypes.Name, user.Email),
    new Claim("LastChanged", {Database Value})
};

var claimsIdentity = new ClaimsIdentity(
    claims,
    CookieAuthenticationDefaults.AuthenticationScheme);

await HttpContext.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    new ClaimsPrincipal(claimsIdentity));
```

To implement an override for the `ValidatePrincipal` event, write a method with the following signature in a class that derives from CookieAuthenticationEvents:

C#

```
ValidatePrincipal(CookieValidatePrincipalContext)
```

The following is an example implementation of `CookieAuthenticationEvents`:

C#

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authentication.Cookies;

public class CustomCookieAuthenticationEvents : CookieAuthenticationEvents
{
    private readonly IUserRepository _userRepository;

    public CustomCookieAuthenticationEvents(IUserRepository userRepository)
    {
        _userRepository = userRepository;
    }

    public override async Task
ValidatePrincipal(CookieValidatePrincipalContext context)
    {
        var userPrincipal = context.Principal;

        // Look for the LastChanged claim.
        var lastChanged = (from c in userPrincipal.Claims
                           where c.Type == "LastChanged"
                           select c.Value).FirstOrDefault();

        if (string.IsNullOrEmpty(lastChanged) ||
            !_userRepository.ValidateLastChanged(lastChanged))
        {
            context.RejectPrincipal();

            await context.HttpContext.SignOutAsync(
                CookieAuthenticationDefaults.AuthenticationScheme);
        }
    }
}
```

Register the events instance during cookie service registration. Provide a scoped service registration for your `CustomCookieAuthenticationEvents` class:

C#

```
using Microsoft.AspNetCore.Authentication.Cookies;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
```

```
builder.Services.AddControllersWithViews();

builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(options =>
    {
        options.EventsType = typeof(CustomCookieAuthenticationEvents);
    });

builder.Services.AddScoped<CustomCookieAuthenticationEvents>();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

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

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

app.MapRazorPages();
app.MapDefaultControllerRoute();

app.Run();
```

Consider a situation in which the user's name is updated—a decision that doesn't affect security in any way. If you want to non-destructively update the user principal, call `context.ReplacePrincipal` and set the `context.ShouldRenew` property to `true`.

> ⚠ **Warning**
>
> The approach described here is triggered on every request. Validating authentication cookies for all users on every request can result in a large performance penalty for the app.

Persistent cookies

You may want the cookie to persist across browser sessions. This persistence should only be enabled with explicit user consent with a "Remember Me" checkbox on sign in or a similar mechanism.

The following code snippet creates an identity and corresponding cookie that survives through browser closures. Any sliding expiration settings previously configured are honored. If the cookie expires while the browser is closed, the browser clears the cookie once it's restarted.

Set IsPersistent to `true` in AuthenticationProperties:

```
C#

// using Microsoft.AspNetCore.Authentication;

await HttpContext.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    new ClaimsPrincipal(claimsIdentity),
    new AuthenticationProperties
    {
        IsPersistent = true
    });
```

Absolute cookie expiration

An absolute expiration time can be set with ExpiresUtc. To create a persistent cookie, `IsPersistent` must also be set. Otherwise, the cookie is created with a session-based lifetime and could expire either before or after the authentication ticket that it holds. When `ExpiresUtc` is set, it overrides the value of the ExpireTimeSpan option of CookieAuthenticationOptions, if set.

The following code snippet creates an identity and corresponding cookie that lasts for 20 minutes. This ignores any sliding expiration settings previously configured.

```
C#

// using Microsoft.AspNetCore.Authentication;

await HttpContext.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    new ClaimsPrincipal(claimsIdentity),
    new AuthenticationProperties
    {
        IsPersistent = true,
        ExpiresUtc = DateTime.UtcNow.AddMinutes(20)
    });
```

Configure OpenID Connect Web (UI) authentication in ASP.NET Core

Article • 12/02/2024

By Damien Bowden

View or download sample code

This article covers the following areas:

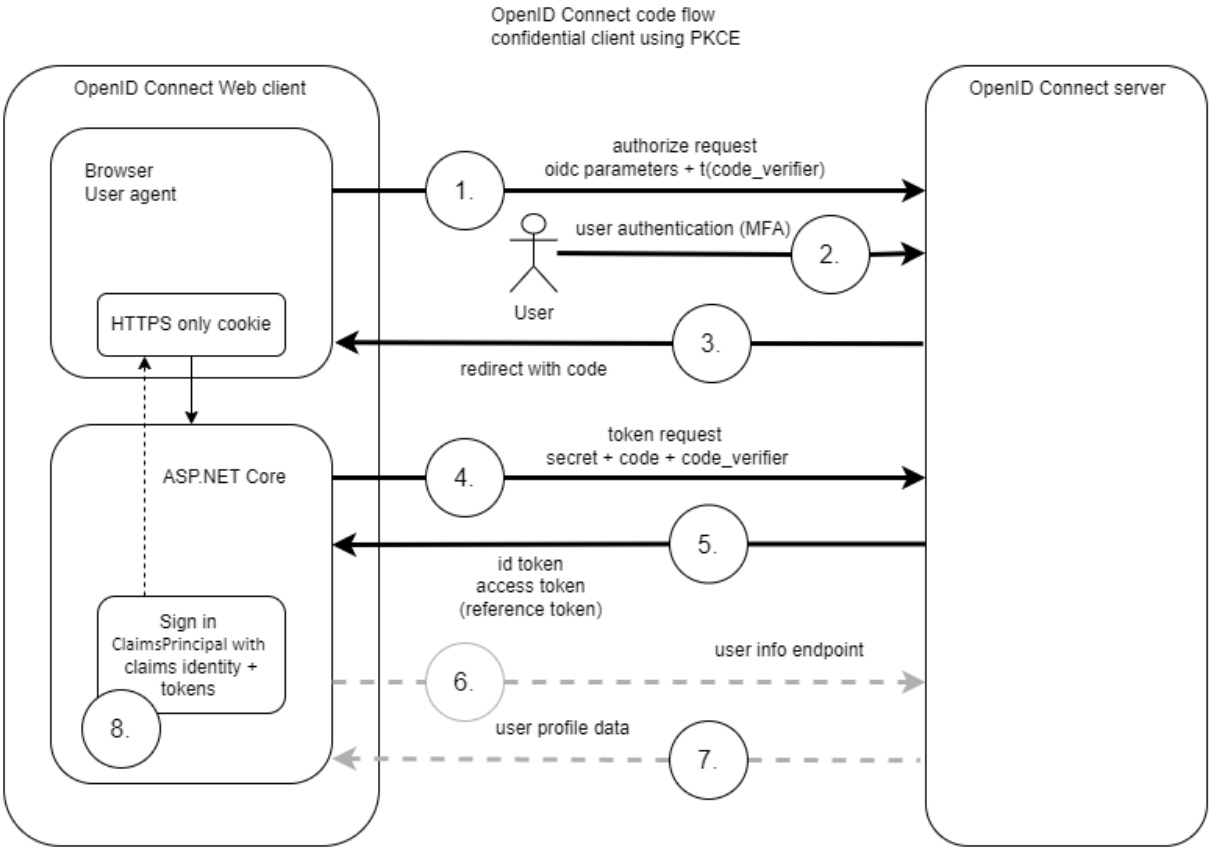
- What is an OpenID Connect confidential interactive client
- Create an OpenID Connect client in ASP.NET Core
- Examples of OpenID Connect client with code snippets
- Using third party OpenID Connect provider clients
- Backend for frontend (BFF) security architecture
- Advanced features, standards, extending the an OpenID Connect client

What is an OpenID Connect confidential interactive client

OpenID Connect can be used to implement authentication in ASP.NET Core applications. The recommended way is to use an OpenID Connect confidential client using the code flow. Using the Proof Key for Code Exchange by OAuth Public Clients (PKCE) is recommended for this implementation. Both the application client and the user of the application are authenticated in the confidential flow. The application client uses a client secret or a client assertion to authenticate.

Public OpenID Connect/OAuth clients are no longer recommended for web applications.

The default flow works as shown in the following diagram:



OpenID Connect comes in many variations and all server implementations have slightly different parameters and requirements. Some servers don't support the user info endpoint, some still don't support PKCE and others require special parameters in the token request. Client assertions can be used instead of client secrets. New standards also exist which add extra security on top of the OpenID Connect Core, for example FAPI, CIBA or DPoP for downstream APIs.

> **Note**
>
> From .NET 9, OAuth 2.0 Pushed Authorization Requests (PAR) RFC 9126 is used per default, if the OpenID Connect server supports this. This is a three step flow and not a two step flow as shown above. (User Info request is an optional step.)

Create an Open ID Connect code flow client using Razor Pages

The following section shows how to implement an OpenID Connect client in an empty ASP.NET Core Razor page project. The same logic can be applied to any ASP.NET Core web project with only the UI integration being different.

Add OpenID Connect support

Add the Microsoft.AspNetCore.Authentication.OpenIdConnect Nuget packages to the ASP.NET Core project.

Setup the OpenID Connect client

Add the authentication to the web application using the builder.Services in the **Program.cs** file. The configuration is dependent on the OpenID Connect server. Each OpenID Connect server requires small differences in the setup.

The OpenID Connect handler is used for challenges and signout. The cookie is used to handle the session in the web application. The default schemes for the authentication can be specified as required.

See the ASP.NET Core [authentication-handler](xref: security/authentication/index?view=aspnetcore-8.0#authentication-handler) for details.

C#

```
builder.Services.AddAuthentication(options =>
{
    options.DefaultScheme =
CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme =
OpenIdConnectDefaults.AuthenticationScheme;
})
.AddCookie()
.AddOpenIdConnect(options =>
{
    var oidcConfig =
builder.Configuration.GetSection("OpenIDConnectSettings");

    options.Authority = oidcConfig["Authority"];
    options.ClientId = oidcConfig["ClientId"];
    options.ClientSecret = oidcConfig["ClientSecret"];

    options.SignInScheme =
CookieAuthenticationDefaults.AuthenticationScheme;
    options.ResponseType = OpenIdConnectResponseType.Code;

    options.SaveTokens = true;
    options.GetClaimsFromUserInfoEndpoint = true;

    options.MapInboundClaims = false;
    options.TokenValidationParameters.NameClaimType =
JwtRegisteredClaimNames.Name;
    options.TokenValidationParameters.RoleClaimType = "roles";
});
```

See Secure an ASP.NET Core Blazor Web App with OpenID Connect (OIDC) for details on the different OpenID Connect options.

See Mapping, customizing, and transforming claims in ASP.NET Core for the different claims mapping possibilities.

> **Note**
>
> The following namespaces are required:

C#

```
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.AspNetCore.Authentication.OpenIdConnect;
using Microsoft.IdentityModel.Protocols.OpenIdConnect;
using Microsoft.IdentityModel.Tokens;
```

Setup the configuration properties

Add the OpenID Connect client settings to the application configuration properties. The settings must match the client configuration in the OpenID Connect server. No secrets should be persisted in application settings where they might get accidently checked in. Secrets should be stored in a secure location like Azure Key Vault in production environments or in user secrets in a development environment. See App Secrets.

JSON

```
"OpenIDConnectSettings": {
  // OpenID Connect URL. (The base URL for the /.well-known/openid-
configuration)
  "Authority": "<Authority>",
  // client ID from the OpenID Connect server
  "ClientId": "<Client ID>",
  //"ClientSecret": "--stored-in-user-secrets-or-key-vault--"
},
```

Update the ASP.NET Core pipeline method in the program class.

The UseRouting must be implemented before the UseAuthorization method.

C#

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

app.UseRouting();
app.UseAuthentication();
// Authorization is applied for middleware after the UseAuthorization method
app.UseAuthorization();
app.MapRazorPages();
```

Force authorization

Add the Authorize attribute to the protected razor pages, for example the Index.cshtml.cs file

C#

```
[Authorize]
```

A better way would be to force the whole application to be authorized and opt out for unsecure pages

C#

```
var requireAuthPolicy = new AuthorizationPolicyBuilder()
    .RequireAuthenticatedUser()
    .Build();

builder.Services.AddAuthorizationBuilder()
    .SetFallbackPolicy(requireAuthPolicy);
```

Add a new Logout.cshtml and SignedOut.cshtml Razor page to the project

A logout is required to sign-out both the cookie session and the OpenID Connect session. The whole application needs to redirect to the OpenID Connect server to sign-out. After a successful sign-out, the application will open the RedirectUri route.

Implement a default sign-out page and change the Logout razor page code with this:

C#

```
[Authorize]
public class LogoutModel : PageModel
{
```

```
    public IActionResult OnGetAsync()
    {
        return SignOut(new AuthenticationProperties
        {
            RedirectUri = "/SignedOut"
        },
        // Clear auth cookie
        CookieAuthenticationDefaults.AuthenticationScheme,
        // Redirect to OIDC provider signout endpoint
        OpenIdConnectDefaults.AuthenticationScheme);
    }
}
```

The `SignedOut.cshtml` requires the AllowAnonymous attribute.

C#

```
[AllowAnonymous]
public class SignedOutModel : PageModel
{
    public void OnGet()
    {
    }
}
```

Implement Login Page

A Login Razor Page can also be implemented to call the **ChallengeAsync** directly with the required AuthProperties. This is not required if the whole web application requires authentication and the default Challenge is used.

The `login.cshtml` requires the AllowAnonymous attribute.

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace RazorPageOidc.Pages;

[AllowAnonymous]
public class LoginModel : PageModel
{
    [BindProperty(SupportsGet = true)]
    public string? ReturnUrl { get; set; }

    public async Task OnGetAsync()
```

```
    {
        var properties = GetAuthProperties(ReturnUrl);
        await HttpContext.ChallengeAsync(properties);
    }

    private static AuthenticationProperties GetAuthProperties(string?
returnUrl)
    {
        const string pathBase = "/";

        // Prevent open redirects.
        if (string.IsNullOrEmpty(returnUrl))
        {
            returnUrl = pathBase;
        }
        else if (!Uri.IsWellFormedUriString(returnUrl, UriKind.Relative))
        {
            returnUrl = new Uri(returnUrl, UriKind.Absolute).PathAndQuery;
        }
        else if (returnUrl[0] != '/')
        {
            returnUrl = $"{pathBase}{returnUrl}";
        }

        return new AuthenticationProperties { RedirectUri = returnUrl };
    }
}
```

Add a login, logout button for the user.

```
@if (Context.User.Identity!.IsAuthenticated)
{
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="" asp-
page="/Logout">Logout</a>
    </li>

    <span class="nav-link text-dark">Hi @Context.User.Identity.Name</span>
}
else
{
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="" asp-
page="/Index">Login</a>
    </li>
}
```

Examples with code snippets

Example Using User Info endpoint

The OpenID Connect options can be used to map claims, implement handlers or even save the tokens in the session for later usage.

The **Scope** option can be used to request different claims or a refresh token which is sent as information to the OpenID Connect server. Requesting the **offline_access** is asking the server to return a reference token which can be used to refresh the session without authenticating the user of the application again.

C#

```
services.AddAuthentication(options =>
{
    options.DefaultScheme =
CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme =
OpenIdConnectDefaults.AuthenticationScheme;
})
.AddCookie()
.AddOpenIdConnect(OpenIdConnectDefaults.AuthenticationScheme, options =>
{
    var oidcConfig =
builder.Configuration.GetSection("OpenIDConnectSettings");
    options.Authority = oidcConfig["IdentityProviderUrl"];
    options.ClientSecret = oidcConfig["ClientSecret"];
    options.ClientId = oidcConfig["Audience"];
    options.ResponseType = OpenIdConnectResponseType.Code;

    options.Scope.Clear();
    options.Scope.Add("openid");
    options.Scope.Add("profile");
    options.Scope.Add("email");
    options.Scope.Add("offline_access");

    options.ClaimActions.Remove("amr");
    options.ClaimActions.MapUniqueJsonKey("website", "website");

    options.GetClaimsFromUserInfoEndpoint = true;
    options.SaveTokens = true;

    // .NET 9 feature
    options.PushedAuthorizationBehavior =
PushedAuthorizationBehavior.Require;

    options.TokenValidationParameters.NameClaimType = "name";
```

```
    options.TokenValidationParameters.RoleClaimType = "role";
});
```

Implementing Microsoft identity providers

Microsoft has multiple identity providers and OpenID Connect implementations. Microsoft has different OpenID Connect servers:

- Microsoft Entra ID
- Microsoft Entra External ID
- Azure AD B2C

If authenticating using one of the Microsoft identity providers in ASP.NET Core, it is recommended to use the Microsoft.Identity.Web Nuget packages.

The Microsoft.Identity.Web Nuget packages is a Microsoft specific client built on top on the ASP.NET Core OpenID Connect client with some changes to the default client.

Using third party OpenID Connect provider clients

Many OpenID Connect server implementations create Nuget packages which are optimized for the same OpenID Connect implementation. These packages implement the OpenID Connect client specifics with the extras required by the specific OpenID Connect server. Microsoft.Identity.Web is one example of this.

If implementing multiple OpenID Connect clients from different OpenID Connect servers in a single application, it is normally better to revert to the default ASP.NET Core implementation as the different clients overwrite some options which affect the other clients.

OpenIddict Web providers is a client implementation which supports many different server implementations.

IdentityModel is a .NET standard helper library for claims-based identity, OAuth 2.0 and OpenID Connect. This can also be used to help with the client implementation.

Backend for frontend (BFF) security architecture

It is no longer recommended to implement OpenID Connect public clients for any web applications.

See the draft OAuth 2.0 for Browser-Based Applications for further details.

If implementing **web** applications which have no independent backend, it is recommended to use the Backend for Frontend (BFF) pattern security architecture. This pattern can be implemented in different ways, but the authentication is always implemented in the backend and no sensitive data is sent to the web client for further authorization or authentication flows.

Advanced features, standards, extending the OIDC client

Logging

Debugging OpenID Connect clients can be hard. Personally identifiable information (PII) data is not logged by default. If debugging in development mode, the ** IdentityModelEventSource.ShowPII** can be used to log sensitive personal data. This should never by deployed to productive servers.

C#

```
//using ...

using Microsoft.IdentityModel.Logging;

var builder = WebApplication.CreateBuilder(args);

//... code

var app = builder.Build();

IdentityModelEventSource.ShowPII = true;

//... code

app.Run();
```

See Logging for further information on logging.

> ⓘ Note

You may want to lower the configured log level to see all the required logs.

OIDC and OAuth Parameter Customization

The OAuth and OIDC authentication handlers AdditionalAuthorizationParameters option allows customization of authorization message parameters that are usually included as part of the redirect query string.

Map claims from OpenID Connect

Refer to the following document:

Mapping, customizing, and transforming claims in ASP.NET Core

Blazor OpenID Connect

Refer to the following document:

Secure an ASP.NET Core Blazor Web App with OpenID Connect (OIDC)

Standards

OpenID Connect 1.0

Proof Key for Code Exchange by OAuth Public Clients

The OAuth 2.0 Authorization Framework

OAuth 2.0 Pushed Authorization Requests (PAR) RFC 9126

Configure certificate authentication in ASP.NET Core

Article • 09/10/2024

`Microsoft.AspNetCore.Authentication.Certificate` contains an implementation similar to Certificate Authentication for ASP.NET Core. Certificate authentication happens at the TLS level, long before it ever gets to ASP.NET Core. More accurately, this is an authentication handler that validates the certificate and then gives you an event where you can resolve that certificate to a `ClaimsPrincipal`.

You ***must*** configure your server for certificate authentication, be it IIS, Kestrel, Azure Web Apps, or whatever else you're using.

Proxy and load balancer scenarios

Certificate authentication is a stateful scenario primarily used where a proxy or load balancer doesn't handle traffic between clients and servers. If a proxy or load balancer is used, certificate authentication only works if the proxy or load balancer:

- Handles the authentication.
- Passes the user authentication information to the app (for example, in a request header), which acts on the authentication information.

An alternative to certificate authentication in environments where proxies and load balancers are used is Active Directory Federated Services (ADFS) with OpenID Connect (OIDC).

Get started

Acquire an HTTPS certificate, apply it, and configure your server to require certificates.

In the web app:

- Add a reference to the Microsoft.AspNetCore.Authentication.Certificate NuGet package.
- In `Program.cs`, call `builder.Services.AddAuthentication(CertificateAuthenticationDefaults.AuthenticationScheme).AddCertificate(...);`. Provide a delegate for `OnCertificateValidated` to do any supplementary validation on the client

certificate sent with requests. Turn that information into a `ClaimsPrincipal` and set it on the `context.Principal` property.

If authentication fails, this handler returns a `403 (Forbidden)` response rather a `401 (Unauthorized)`, as you might expect. The reasoning is that the authentication should happen during the initial TLS connection. By the time it reaches the handler, it's too late. There's no way to upgrade the connection from an anonymous connection to one with a certificate.

`UseAuthentication` is required to set `HttpContext.User` to a `ClaimsPrincipal` created from the certificate. For example:

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthentication(
        CertificateAuthenticationDefaults.AuthenticationScheme)
    .AddCertificate();

var app = builder.Build();

app.UseAuthentication();

app.MapGet("/", () => "Hello World!");

app.Run();
```

The preceding example demonstrates the default way to add certificate authentication. The handler constructs a user principal using the common certificate properties.

Configure certificate validation

The `CertificateAuthenticationOptions` handler has some built-in validations that are the minimum validations you should perform on a certificate. Each of these settings is enabled by default.

AllowedCertificateTypes = Chained, SelfSigned, or All (Chained | SelfSigned)

Default value: `CertificateTypes.Chained`

This check validates that only the appropriate certificate type is allowed. If the app is using self-signed certificates, this option needs to be set to `CertificateTypes.All` or

`CertificateTypes.SelfSigned`.

ChainTrustValidationMode

Default value: X509ChainTrustMode.System

The certificate presented by the client must chain to a trusted root certificate. This check controls which trust store contains these root certificates.

By default, the handler uses the system trust store. If the presented client certificate needs to chain to a root certificate which doesn't appear in the system trust store, this option can be set to X509ChainTrustMode.CustomRootTrust to make the handler use the `CustomTrustStore`.

CustomTrustStore

Default value: Empty X509Certificate2Collection

If the handler's ChainTrustValidationMode property is set to `X509ChainTrustMode.CustomRootTrust`, this X509Certificate2Collection contains every certificate which will be used to validate the client certificate up to a trusted root, including the trusted root.

When the client presents a certificate which is part of a multi-level certificate chain, `CustomTrustStore` must contain every issuing certificate in the chain.

ValidateCertificateUse

Default value: `true`

This check validates that the certificate presented by the client has the Client Authentication extended key use (EKU), or no EKUs at all. As the specifications say, if no EKU is specified, then all EKUs are deemed valid.

ValidateValidityPeriod

Default value: `true`

This check validates that the certificate is within its validity period. On each request, the handler ensures that a certificate that was valid when it was presented hasn't expired during its current session.

RevocationFlag

Default value: `X509RevocationFlag.ExcludeRoot`

A flag that specifies which certificates in the chain are checked for revocation.

Revocation checks are only performed when the certificate is chained to a root certificate.

RevocationMode

Default value: `X509RevocationMode.Online`

A flag that specifies how revocation checks are performed.

Specifying an online check can result in a long delay while the certificate authority is contacted.

Revocation checks are only performed when the certificate is chained to a root certificate.

Can I configure my app to require a certificate only on certain paths?

This isn't possible. Remember the certificate exchange is done at the start of the HTTPS conversation, it's done by the server before the first request is received on that connection so it's not possible to scope based on any request fields.

Handler events

The handler has two events:

- `OnAuthenticationFailed`: Called if an exception happens during authentication and allows you to react.
- `OnCertificateValidated`: Called after the certificate has been validated, passed validation and a default principal has been created. This event allows you to perform your own validation and augment or replace the principal. For examples include:
    - Determining if the certificate is known to your services.
    - Constructing your own principal. Consider the following example:

C#

```
builder.Services.AddAuthentication(
        CertificateAuthenticationDefaults.AuthenticationScheme)
    .AddCertificate(options =>
    {
        options.Events = new CertificateAuthenticationEvents
        {
            OnCertificateValidated = context =>
            {
                var claims = new[]
                {
                    new Claim(
                        ClaimTypes.NameIdentifier,
                        context.ClientCertificate.Subject,
                        ClaimValueTypes.String,
context.Options.ClaimsIssuer),
                    new Claim(
                        ClaimTypes.Name,
                        context.ClientCertificate.Subject,
                        ClaimValueTypes.String,
context.Options.ClaimsIssuer)
                };

                context.Principal = new ClaimsPrincipal(
                    new ClaimsIdentity(claims,
context.Scheme.Name));
                context.Success();

                return Task.CompletedTask;
            }
        };
    });
```

If you find the inbound certificate doesn't meet your extra validation, call `context.Fail("failure reason")` with a failure reason.

For better functionality, call a service registered in dependency injection that connects to a database or other type of user store. Access the service by using the context passed into the delegate. Consider the following example:

C#

```
builder.Services.AddAuthentication(
        CertificateAuthenticationDefaults.AuthenticationScheme)
    .AddCertificate(options =>
    {
        options.Events = new CertificateAuthenticationEvents
        {
            OnCertificateValidated = context =>
            {
```

```
                var validationService = context.HttpContext.RequestServices
                    .GetRequiredService<ICertificateValidationService>();

                if
(validationService.ValidateCertificate(context.ClientCertificate))
                {
                    var claims = new[]
                    {
                        new Claim(
                            ClaimTypes.NameIdentifier,
                            context.ClientCertificate.Subject,
                            ClaimValueTypes.String,
context.Options.ClaimsIssuer),
                        new Claim(
                            ClaimTypes.Name,
                            context.ClientCertificate.Subject,
                            ClaimValueTypes.String,
context.Options.ClaimsIssuer)
                    };

                    context.Principal = new ClaimsPrincipal(
                        new ClaimsIdentity(claims, context.Scheme.Name));
                    context.Success();
                }

                return Task.CompletedTask;
            }
        };
    });
```

Conceptually, the validation of the certificate is an authorization concern. Adding a check on, for example, an issuer or thumbprint in an authorization policy, rather than inside `OnCertificateValidated`, is perfectly acceptable.

Configure your server to require certificates

Kestrel

In `Program.cs`, configure Kestrel as follows:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.Configure<KestrelServerOptions>(options =>
{
    options.ConfigureHttpsDefaults(options =>
        options.ClientCertificateMode =
```

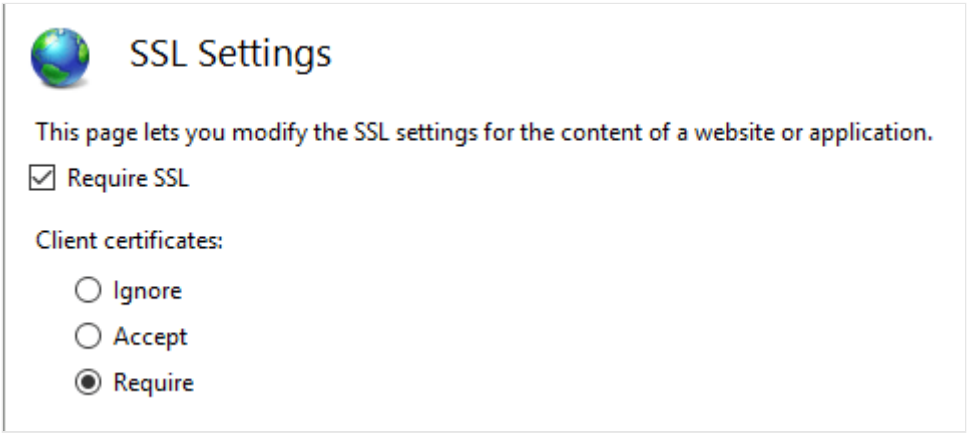
```
ClientCertificateMode.RequireCertificate);
});
```

> ⓘ Note
>
> Endpoints created by calling Listen **before** calling ConfigureHttpsDefaults won't have the defaults applied.

IIS

Complete the following steps in IIS Manager:

1. Select your site from the **Connections** tab.
2. Double-click the **SSL Settings** option in the **Features View** window.
3. Check the **Require SSL** checkbox, and select the **Require** radio button in the **Client certificates** section.



Azure and custom web proxies

See the host and deploy documentation for how to configure the certificate forwarding middleware.

Use certificate authentication in Azure Web Apps

No forwarding configuration is required for Azure. Forwarding configuration is set up by the Certificate Forwarding Middleware.

> ⓘ Note
>
> Certificate Forwarding Middleware is required for this scenario.

For more information, see Use a TLS/SSL certificate in your code in Azure App Service (Azure documentation).

Use certificate authentication in custom web proxies

The `AddCertificateForwarding` method is used to specify:

- The client header name.
- How the certificate is to be loaded (using the `HeaderConverter` property).

In custom web proxies, the certificate is passed as a custom request header, for example `X-SSL-CERT`. To use it, configure certificate forwarding in `Program.cs`:

C#

```
builder.Services.AddCertificateForwarding(options =>
{
    options.CertificateHeader = "X-SSL-CERT";

    options.HeaderConverter = headerValue =>
    {
        X509Certificate2? clientCertificate = null;

        if (!string.IsNullOrWhiteSpace(headerValue))
        {
            clientCertificate = new
X509Certificate2(StringToByteArray(headerValue));
        }

        return clientCertificate!;

        static byte[] StringToByteArray(string hex)
        {
            var numberChars = hex.Length;
            var bytes = new byte[numberChars / 2];

            for (int i = 0; i < numberChars; i += 2)
            {
                bytes[i / 2] = Convert.ToByte(hex.Substring(i, 2), 16);
            }

            return bytes;
        }
    };
});
```

If the app is reverse proxied by NGINX with the configuration `proxy_set_header ssl-client-cert $ssl_client_escaped_cert` or deployed on Kubernetes using NGINX Ingress,

the client certificate is passed to the app in URL-encoded form. To use the certificate, decode it as follows:

C#

```
builder.Services.AddCertificateForwarding(options =>
{
    options.CertificateHeader = "ssl-client-cert";

    options.HeaderConverter = (headerValue) =>
    {
        X509Certificate2? clientCertificate = null;

        if (!string.IsNullOrWhiteSpace(headerValue))
        {
            clientCertificate = X509Certificate2.CreateFromPem(
                WebUtility.UrlDecode(headerValue));
        }

        return clientCertificate!;
    };
});
```

Add the middleware in `Program.cs`. `UseCertificateForwarding` is called before the calls to `UseAuthentication` and `UseAuthorization`:

C#

```
var app = builder.Build();

app.UseCertificateForwarding();

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

A separate class can be used to implement validation logic. Because the same self-signed certificate is used in this example, ensure that only your certificate can be used. Validate that the thumbprints of both the client certificate and the server certificate match, otherwise any certificate can be used and will be enough to authenticate. This would be used inside the `AddCertificate` method. You could also validate the subject or the issuer here if you're using intermediate or child certificates.

C#

```
using System.Security.Cryptography.X509Certificates;

namespace CertAuthSample.Snippets;
```

```
public class SampleCertificateValidationService :
ICertificateValidationService
{
    public bool ValidateCertificate(X509Certificate2 clientCertificate)
    {
        // Don't hardcode passwords in production code.
        // Use a certificate thumbprint or Azure Key Vault.
        var expectedCertificate = new X509Certificate2(
            Path.Combine("/path/to/pfx"), "1234");

        return clientCertificate.Thumbprint ==
expectedCertificate.Thumbprint;
    }
}
```

Implement an HttpClient using a certificate and IHttpClientFactory

In the following example, a client certificate is added to a `HttpClientHandler` using the `ClientCertificates` property from the handler. This handler can then be used in a named instance of an `HttpClient` using the ConfigurePrimaryHttpMessageHandler method. This is setup in `Program.cs`:

C#

```
var clientCertificate =
    new X509Certificate2(
      Path.Combine(_environment.ContentRootPath, "sts_dev_cert.pfx"),
"1234");

builder.Services.AddHttpClient("namedClient", c =>
{
}).ConfigurePrimaryHttpMessageHandler(() =>
{
    var handler = new HttpClientHandler();
    handler.ClientCertificates.Add(clientCertificate);
    return handler;
});
```

The `IHttpClientFactory` can then be used to get the named instance with the handler and the certificate. The `CreateClient` method with the name of the client defined in `Program.cs` is used to get the instance. The HTTP request can be sent using the client as required:

C#

```
public class SampleHttpService
{
    private readonly IHttpClientFactory _httpClientFactory;
```

```
    public SampleHttpService(IHttpClientFactory httpClientFactory)
        => _httpClientFactory = httpClientFactory;

    public async Task<JsonDocument> GetAsync()
    {
        var httpClient = _httpClientFactory.CreateClient("namedClient");
        var httpResponseMessage = await
httpClient.GetAsync("https://example.com");

        if (httpResponseMessage.IsSuccessStatusCode)
        {
            return JsonDocument.Parse(
                await httpResponseMessage.Content.ReadAsStringAsync());
        }

        throw new ApplicationException($"Status code:
{httpResponseMessage.StatusCode}");
    }
}
```

If the correct certificate is sent to the server, the data is returned. If no certificate or the wrong certificate is sent, an HTTP 403 status code is returned.

Create certificates in PowerShell

Creating the certificates is the hardest part in setting up this flow. A root certificate can be created using the `New-SelfSignedCertificate` PowerShell cmdlet. When creating the certificate, use a strong password. It's important to add the `KeyUsageProperty` parameter and the `KeyUsage` parameter as shown.

Create root CA

PowerShell

```
New-SelfSignedCertificate -DnsName "root_ca_dev_damienbod.com",
"root_ca_dev_damienbod.com" -CertStoreLocation "cert:\LocalMachine\My" -
NotAfter (Get-Date).AddYears(20) -FriendlyName "root_ca_dev_damienbod.com" -
KeyUsageProperty All -KeyUsage CertSign, CRLSign, DigitalSignature

$mypwd = ConvertTo-SecureString -String "1234" -Force -AsPlainText

Get-ChildItem -Path cert:\localMachine\my\"The thumbprint..." | Export-
PfxCertificate -FilePath C:\git\root_ca_dev_damienbod.pfx -Password $mypwd

Export-Certificate -Cert cert:\localMachine\my\"The thumbprint..." -FilePath
root_ca_dev_damienbod.crt
```

> **Note**
>
> The `-DnsName` parameter value must match the deployment target of the app. For example, "localhost" for development.

Install in the trusted root

The root certificate must be trusted on your host system. Only root certificates created by a certificate authority are trusted by default. For information on how to trust the root certificate on Windows, see the Windows documentation or the Import-Certificate PowerShell cmdlet.

Intermediate certificate

An intermediate certificate can now be created from the root certificate. This isn't required for all use cases, but you might need to create many certificates or need to activate or disable groups of certificates. The `TextExtension` parameter is required to set the path length in the basic constraints of the certificate.

The intermediate certificate can then be added to the trusted intermediate certificate in the Windows host system.

PowerShell

```
$mypwd = ConvertTo-SecureString -String "1234" -Force -AsPlainText

$parentcert = ( Get-ChildItem -Path cert:\LocalMachine\My\"The thumbprint of
the root..." )

New-SelfSignedCertificate -certstorelocation cert:\localmachine\my -dnsname
"intermediate_dev_damienbod.com" -Signer $parentcert -NotAfter (Get-
Date).AddYears(20) -FriendlyName "intermediate_dev_damienbod.com" -
KeyUsageProperty All -KeyUsage CertSign, CRLSign, DigitalSignature -
TextExtension @("2.5.29.19={text}CA=1&pathlength=1")

Get-ChildItem -Path cert:\localMachine\my\"The thumbprint..." | Export-
PfxCertificate -FilePath
C:\git\AspNetCoreCertificateAuth\Certs\intermediate_dev_damienbod.pfx -
Password $mypwd

Export-Certificate -Cert cert:\localMachine\my\"The thumbprint..." -FilePath
intermediate_dev_damienbod.crt
```

Create child certificate from intermediate certificate

A child certificate can be created from the intermediate certificate. This is the end entity and doesn't need to create more child certificates.

PowerShell

```
$parentcert = ( Get-ChildItem -Path cert:\LocalMachine\My\"The thumbprint
from the Intermediate certificate..." )

New-SelfSignedCertificate -certstorelocation cert:\localmachine\my -dnsname
"child_a_dev_damienbod.com" -Signer $parentcert -NotAfter (Get-
Date).AddYears(20) -FriendlyName "child_a_dev_damienbod.com"

$mypwd = ConvertTo-SecureString -String "1234" -Force -AsPlainText

Get-ChildItem -Path cert:\localMachine\my\"The thumbprint..." | Export-
PfxCertificate -FilePath
C:\git\AspNetCoreCertificateAuth\Certs\child_a_dev_damienbod.pfx -Password
$mypwd

Export-Certificate -Cert cert:\localMachine\my\"The thumbprint..." -FilePath
child_a_dev_damienbod.crt
```

Create child certificate from root certificate

A child certificate can also be created from the root certificate directly.

PowerShell

```
$rootcert = ( Get-ChildItem -Path cert:\LocalMachine\My\"The thumbprint from
the root cert..." )

New-SelfSignedCertificate -certstorelocation cert:\localmachine\my -dnsname
"child_a_dev_damienbod.com" -Signer $rootcert -NotAfter (Get-
Date).AddYears(20) -FriendlyName "child_a_dev_damienbod.com"

$mypwd = ConvertTo-SecureString -String "1234" -Force -AsPlainText

Get-ChildItem -Path cert:\localMachine\my\"The thumbprint..." | Export-
PfxCertificate -FilePath
C:\git\AspNetCoreCertificateAuth\Certs\child_a_dev_damienbod.pfx -Password
$mypwd

Export-Certificate -Cert cert:\localMachine\my\"The thumbprint..." -FilePath
child_a_dev_damienbod.crt
```

Example root - intermediate certificate - certificate

PowerShell

```
$mypwdroot = ConvertTo-SecureString -String "1234" -Force -AsPlainText
$mypwd = ConvertTo-SecureString -String "1234" -Force -AsPlainText

New-SelfSignedCertificate -DnsName "root_ca_dev_damienbod.com",
"root_ca_dev_damienbod.com" -CertStoreLocation "cert:\LocalMachine\My" -
NotAfter (Get-Date).AddYears(20) -FriendlyName "root_ca_dev_damienbod.com" -
KeyUsageProperty All -KeyUsage CertSign, CRLSign, DigitalSignature

Get-ChildItem -Path
cert:\localMachine\my\0C89639E4E2998A93E423F919B36D4009A0F9991 | Export-
PfxCertificate -FilePath C:\git\root_ca_dev_damienbod.pfx -Password
$mypwdroot

Export-Certificate -Cert
cert:\localMachine\my\0C89639E4E2998A93E423F919B36D4009A0F9991 -FilePath
root_ca_dev_damienbod.crt

$rootcert = ( Get-ChildItem -Path
cert:\LocalMachine\My\0C89639E4E2998A93E423F919B36D4009A0F9991 )

New-SelfSignedCertificate -certstorelocation cert:\localmachine\my -dnsname
"child_a_dev_damienbod.com" -Signer $rootcert -NotAfter (Get-
Date).AddYears(20) -FriendlyName "child_a_dev_damienbod.com" -
KeyUsageProperty All -KeyUsage CertSign, CRLSign, DigitalSignature -
TextExtension @("2.5.29.19={text}CA=1&pathlength=1")

Get-ChildItem -Path
cert:\localMachine\my\BA9BF91ED35538A01375EFC212A2F46104B33A44 | Export-
PfxCertificate -FilePath
C:\git\AspNetCoreCertificateAuth\Certs\child_a_dev_damienbod.pfx -Password
$mypwd

Export-Certificate -Cert
cert:\localMachine\my\BA9BF91ED35538A01375EFC212A2F46104B33A44 -FilePath
child_a_dev_damienbod.crt

$parentcert = ( Get-ChildItem -Path
cert:\LocalMachine\My\BA9BF91ED35538A01375EFC212A2F46104B33A44 )

New-SelfSignedCertificate -certstorelocation cert:\localmachine\my -dnsname
"child_b_from_a_dev_damienbod.com" -Signer $parentcert -NotAfter (Get-
Date).AddYears(20) -FriendlyName "child_b_from_a_dev_damienbod.com"

Get-ChildItem -Path
cert:\localMachine\my\141594A0AE38CBBECED7AF680F7945CD51D8F28A | Export-
PfxCertificate -FilePath
C:\git\AspNetCoreCertificateAuth\Certs\child_b_from_a_dev_damienbod.pfx -
Password $mypwd

Export-Certificate -Cert
cert:\localMachine\my\141594A0AE38CBBECED7AF680F7945CD51D8F28A -FilePath
child_b_from_a_dev_damienbod.crt
```

When using the root, intermediate, or child certificates, the certificates can be validated using the Thumbprint or PublicKey as required:

C#

```
using System.Security.Cryptography.X509Certificates;

namespace CertAuthSample.Snippets;

public class SampleCertificateThumbprintsValidationService :
ICertificateValidationService
{
    private readonly string[] validThumbprints = new[]
    {
        "141594A0AE38CBBECED7AF680F7945CD51D8F28A",
        "0C89639E4E2998A93E423F919B36D4009A0F9991",
        "BA9BF91ED35538A01375EFC212A2F46104B33A44"
    };

    public bool ValidateCertificate(X509Certificate2 clientCertificate)
        => validThumbprints.Contains(clientCertificate.Thumbprint);
}
```

Certificate validation caching

ASP.NET Core 5.0 and later versions support the ability to enable caching of validation results. The caching dramatically improves performance of certificate authentication, as validation is an expensive operation.

By default, certificate authentication disables caching. To enable caching, call `AddCertificateCache` in `Program.cs`:

C#

```
builder.Services.AddAuthentication(
        CertificateAuthenticationDefaults.AuthenticationScheme)
    .AddCertificate()
    .AddCertificateCache(options =>
    {
        options.CacheSize = 1024;
        options.CacheEntryExpiration = TimeSpan.FromMinutes(2);
    });
```

The default caching implementation stores results in memory. You can provide your own cache by implementing `ICertificateValidationCache` and registering it with dependency injection. For example, `services.AddSingleton<ICertificateValidationCache, YourCache>()`.

Optional client certificates

This section provides information for apps that must protect a subset of the app with a certificate. For example, a Razor Page or controller in the app might require client certificates. This presents challenges as client certificates:

- Are a TLS feature, not an HTTP feature.
- Are negotiated per-connection and usually at the start of the connection before any HTTP data is available.

There are two approaches to implementing optional client certificates:

1. Using separate host names (SNI) and redirecting. While more work to configure, this is recommended because it works in most environments and protocols.
2. Renegotiation during an HTTP request. This has several limitations and is not recommended.

Separate Hosts (SNI)

At the start of the connection, only the Server Name Indication (SNI)† is known. Client certificates can be configured per host name so that one host requires them and another does not.

- Set up binding for the domain and subdomain:
  - For example, set up bindings on `contoso.com` and `myClient.contoso.com`. The `contoso.com` host doesn't require a client certificate but `myClient.contoso.com` does.
  - For more information, see:
    - Kestrel web server in ASP.NET Core:
      - ListenOptions.UseHttps
      - ClientCertificateMode
    - IIS
      - Hosting IIS
      - Configure security on IIS
    - HTTP.sys: Configure Windows Server

ASP.NET Core 5 and later adds more convenient support for redirecting to acquire optional client certificates. For more information, see the Optional certificates sample.

- For requests to the web app that require a client certificate and don't have one:
  - Redirect to the same page using the client certificate protected subdomain.

    - For example, redirect to `myClient.contoso.com/requestedPage`. Because the request to `myClient.contoso.com/requestedPage` is a different hostname than `contoso.com/requestedPage`, the client establishes a different connection and the client certificate is provided.
    - For more information, see Introduction to authorization in ASP.NET Core.

† Server Name Indication (SNI) is a TLS extension to include a virtual domain as a part of SSL negotiation. This effectively means the virtual domain name, or a hostname, can be used to identify the network end point.

Renegotiation

TLS renegotiation is a process by which the client and server can re-assess the encryption requirements for an individual connection, including requesting a client certificate if not previously provided. TLS renegotiation is a security risk and isn't recommended because:

- In HTTP/1.1 the server must first buffer or consume any HTTP data that is in flight such as POST request bodies to make sure the connection is clear for the renegotiation. Otherwise the renegotiation can stop responding or fail.
- HTTP/2 and HTTP/3 explicitly prohibit renegotiation.
- There are security risks associated with renegotiation. TLS 1.3 removed renegotiation of the whole connection and replaced it with a new extension for requesting only the client certificate after the start of the connection. This mechanism is exposed via the same APIs and is still subject to the prior constraints of buffering and HTTP protocol versions.

The implementation and configuration of this feature varies by server and framework version.

IIS

IIS manages the client certificate negotiation on your behalf. A subsection of the application can enable the `SslRequireCert` option to negotiate the client certificate for those requests. See Configuration in the IIS documentation for details.

IIS will automatically buffer any request body data up to a configured size limit before renegotiating. Requests that exceed the limit are rejected with a 413 response. This limit defaults to 48KB and is configurable by setting the uploadReadAheadSize.

HttpSys

HttpSys has two settings which control the client certificate negotiation and both should be set. The first is in netsh.exe under `http add sslcert clientcertnegotiation=enable/disable`. This flag indicates if the client certificate should be negotiated at the start of a connection and it should be set to `disable` for optional client certificates. See the netsh docs for details.

The other setting is ClientCertificateMethod. When set to `AllowRenegotation`, the client certificate can be renegotiated during a request.

NOTE The application should buffer or consume any request body data before attempting the renegotiation, otherwise the request may become unresponsive.

An application can first check the ClientCertificate property to see if the certificate is available. If it is not available, ensure the request body has been consumed before calling GetClientCertificateAsync to negotiate one. Note `GetClientCertificateAsync` can return a null certificate if the client declines to provide one.

NOTE The behavior of the `ClientCertificate` property changed in .NET 6. For more information, see this GitHub issue.

Kestrel

Kestrel controls client certificate negotiation with the ClientCertificateMode option.

ClientCertificateMode.DelayCertificate is new option available in .NET 6 or later. When set, an app can check the ClientCertificate property to see if the certificate is available. If it isn't available, ensure the request body has been consumed before calling GetClientCertificateAsync to negotiate one. Note `GetClientCertificateAsync` can return a null certificate if the client declines to provide one.

NOTE The application should buffer or consume any request body data before attempting the renegotiation, otherwise `GetClientCertificateAsync` may throw `InvalidOperationException: Client stream needs to be drained before renegotiation.`.

If you're programmatically configuring the TLS settings per SNI host name, call the UseHttps overload (.NET 6 or later) that takes TlsHandshakeCallbackOptions and controls client certificate renegotiation via TlsHandshakeCallbackContext.AllowDelayedClientCertificateNegotation.

Leave questions, comments, and other feedback on optional client certificates in this GitHub discussion issue.

Configure Windows Authentication in ASP.NET Core

Article • 09/12/2024

By Rick Anderson and Kirk Larkin

Windows Authentication (also known as Negotiate, Kerberos, or NTLM authentication) can be configured for ASP.NET Core apps hosted with IIS, Kestrel, or HTTP.sys.

Windows Authentication relies on the operating system to authenticate users of ASP.NET Core apps. Windows Authentication is used for servers that run on a corporate network using Active Directory domain identities or Windows accounts to identify users. Windows Authentication is best suited to intranet environments where users, client apps, and web servers belong to the same Windows domain.

> **Note**
>
> Windows Authentication isn't supported with HTTP/2. Authentication challenges can be sent on HTTP/2 responses, but the client must downgrade to HTTP/1.1 before authenticating.

Proxy and load balancer scenarios

Windows Authentication is a stateful scenario primarily used in an intranet, where a proxy or load balancer doesn't usually handle traffic between clients and servers. If a proxy or load balancer is used, Windows Authentication only works if the proxy or load balancer:

- Handles the authentication.
- Passes the user authentication information to the app (for example, in a request header), which acts on the authentication information.

An alternative to Windows Authentication in environments where proxies and load balancers are used is Active Directory Federated Services (ADFS) with OpenID Connect (OIDC).

IIS/IIS Express

Add the NuGet package Microsoft.AspNetCore.Authentication.Negotiate and authentication services by calling AddAuthentication in `Program.cs`:

C#

```
using Microsoft.AspNetCore.Authentication.Negotiate;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthentication(NegotiateDefaults.AuthenticationScheme)
    .AddNegotiate();

builder.Services.AddAuthorization(options =>
{
    options.FallbackPolicy = options.DefaultPolicy;
});
builder.Services.AddRazorPages();

var app = builder.Build();
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

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

app.UseRouting();

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

app.MapRazorPages();

app.Run();
```

The preceding code was generated by the ASP.NET Core Razor Pages template with **Windows Authentication** specified.

Launch settings (debugger)

Configuration for launch settings only affects the `Properties/launchSettings.json` file for IIS Express and doesn't configure IIS for Windows Authentication. Server configuration is explained in the IIS section.

The **Web Application** templates available via Visual Studio or the .NET CLI can be configured to support Windows Authentication, which updates the `Properties/launchSettings.json` file automatically.

Visual Studio

New project

Create a new Razor Pages or MVC app. In the **Additional information** dialog, set the **Authentication type** to **Windows**.

Run the app. The username appears in the rendered app's user interface.

Existing project

The project's properties enable Windows Authentication and disable Anonymous Authentication. Open the launch profiles dialog:

1. In Solution Explorer, right click the project and select **Properties**.
2. Select the **Debug** > **General** tab and select **Open debug launch profiles UI**.
3. Clear the checkbox for **Enable Anonymous Authentication**.
4. Select the checkbox for **Enable Windows Authentication**.

Alternatively, the properties can be configured in the `iisSettings` node of the `launchSettings.json` file:

JSON

```
"iisSettings": {
    "windowsAuthentication": true,
    "anonymousAuthentication": false,
    "iisExpress": {
        "applicationUrl": "http://localhost:52171/",
        "sslPort": 44308
    }
}
```

IIS

IIS uses the ASP.NET Core Module to host ASP.NET Core apps. Windows Authentication is configured for IIS via the *web.config* file. The following sections show how to:

- Provide a local *web.config* file that activates Windows Authentication on the server when the app is deployed.
- Use the IIS Manager to configure the *web.config* file of an ASP.NET Core app that has already been deployed to the server.

If you haven't already done so, enable IIS to host ASP.NET Core apps. For more information, see Host ASP.NET Core on Windows with IIS.

Enable the IIS Role Service for Windows Authentication. For more information, see Enable Windows Authentication in IIS Role Services (see Step 2).

IIS Integration Middleware is configured to automatically authenticate requests by default. For more information, see Host ASP.NET Core on Windows with IIS: IIS options (AutomaticAuthentication).

The ASP.NET Core Module is configured to forward the Windows Authentication token to the app by default. For more information, see ASP.NET Core Module configuration reference: Attributes of the aspNetCore element.

Use **either** of the following approaches:

- **Before publishing and deploying the project,** add the following *web.config* file to the project root:

  XML

  ```
  <?xml version="1.0" encoding="utf-8"?>
  <configuration>
    <location path="." inheritInChildApplications="false">
      <system.webServer>
        <security>
          <authentication>
            <anonymousAuthentication enabled="false" />
            <windowsAuthentication enabled="true" />
          </authentication>
        </security>
      </system.webServer>
    </location>
  </configuration>
  ```

  When the project is published by the .NET Core SDK (without the `<IsTransformWebConfigDisabled>` property set to `true` in the project file), the published *web.config* file includes the `<location><system.webServer><security><authentication>` section. For more information on the `<IsTransformWebConfigDisabled>` property, see Host ASP.NET Core on Windows with IIS.

- **After publishing and deploying the project,** perform server-side configuration with the IIS Manager:

1. In IIS Manager, select the IIS site under the **Sites** node of the **Connections** sidebar.
2. Double-click **Authentication** in the **IIS** area.
3. Select **Anonymous Authentication**. Select **Disable** in the **Actions** sidebar.
4. Select **Windows Authentication**. Select **Enable** in the **Actions** sidebar.

When these actions are taken, IIS Manager modifies the app's *web.config* file. A `<system.webServer><security><authentication>` node is added with updated settings for `anonymousAuthentication` and `windowsAuthentication`:

XML

```
<system.webServer>
  <security>
    <authentication>
      <anonymousAuthentication enabled="false" />
      <windowsAuthentication enabled="true" />
    </authentication>
  </security>
</system.webServer>
```

The `<system.webServer>` section added to the *web.config* file by IIS Manager is outside of the app's `<location>` section added by the .NET Core SDK when the app is published. Because the section is added outside of the `<location>` node, the settings are inherited by any sub-apps to the current app. To prevent inheritance, move the added `<security>` section inside of the `<location><system.webServer>` section that the .NET Core SDK provided.

When IIS Manager is used to add the IIS configuration, it only affects the app's *web.config* file on the server. A subsequent deployment of the app may overwrite the settings on the server if the server's copy of *web.config* is replaced by the project's *web.config* file. Use **either** of the following approaches to manage the settings:

- Use IIS Manager to reset the settings in the *web.config* file after the file is overwritten on deployment.
- Add a *web.config file* to the app locally with the settings.

Kestrel

The Microsoft.AspNetCore.Authentication.Negotiate NuGet package can be used with Kestrel to support Windows Authentication using Negotiate and Kerberos on Windows, Linux, and macOS.

> **⚠ Warning**
>
> Credentials can be persisted across requests on a connection. *Negotiate authentication must not be used with proxies unless the proxy maintains a 1:1 connection affinity (a persistent connection) with Kestrel.*

> **ⓘ Note**
>
> The Negotiate handler detects if the underlying server supports Windows Authentication natively and if it is enabled. If the server supports Windows Authentication but it is disabled, an error is thrown asking you to enable the server implementation. When Windows Authentication is enabled in the server, the Negotiate handler transparently forwards authentication requests to it.

Authentication is enabled by the following highlighted code to `Program.cs`:

C#

```
using Microsoft.AspNetCore.Authentication.Negotiate;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthentication(NegotiateDefaults.AuthenticationScheme)
   .AddNegotiate();

builder.Services.AddAuthorization(options =>
{
    options.FallbackPolicy = options.DefaultPolicy;
});
builder.Services.AddRazorPages();

var app = builder.Build();
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

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

app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();
app.MapRazorPages();
```

```
app.Run();
```

The preceding code was generated by the ASP.NET Core Razor Pages template with **Windows Authentication** specified. The following APIs are used in the preceding code:

- AddAuthentication
- AddNegotiate
- UseAuthentication

Kerberos authentication and role-based access control (RBAC)

Kerberos authentication on Linux or macOS doesn't provide any role information for an authenticated user. To add role and group information to a Kerberos user, the authentication handler must be configured to retrieve the roles from an LDAP domain. The most basic configuration only specifies an LDAP domain to query against and uses the authenticated user's context to query the LDAP domain:

C#

```
using Microsoft.AspNetCore.Authentication.Negotiate;
using System.Runtime.InteropServices;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthentication(NegotiateDefaults.AuthenticationScheme)
    .AddNegotiate(options =>
    {
        if (RuntimeInformation.IsOSPlatform(OSPlatform.Linux))
        {
            options.EnableLdap("contoso.com");
        }
    });
```

Some configurations may require specific credentials to query the LDAP domain. The credentials can be specified in the following highlighted options:

C#

```
using Microsoft.AspNetCore.Authentication.Negotiate;
using System.Runtime.InteropServices;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthentication(NegotiateDefaults.AuthenticationScheme)
        .AddNegotiate(options =>
```

```
        {
            if (RuntimeInformation.IsOSPlatform(OSPlatform.Linux))
            {
                options.EnableLdap(settings =>
                {
                    settings.Domain = "contoso.com";
                    settings.MachineAccountName = "machineName";
                    settings.MachineAccountPassword =
                                    builder.Configuration["Password"];
                });
            }
        });

builder.Services.AddRazorPages();
```

By default, the negotiate authentication handler resolves nested domains. In a large or complicated LDAP environment, resolving nested domains may result in a slow lookup or a lot of memory being used for each user. Nested domain resolution can be disabled using the `IgnoreNestedGroups` option.

Anonymous requests are allowed. Use ASP.NET Core Authorization to challenge anonymous requests for authentication.

Windows environment configuration

The Microsoft.AspNetCore.Authentication.Negotiate component performs User Mode authentication. Service Principal Names (SPNs) must be added to the user account running the service, not the machine account. Execute `setspn -S HTTP/myservername.mydomain.com myuser` in an administrative command shell.

Kerberos vs NTLM

The Negotiate package on Kestrel for ASP.NET Core attempts to use Kerberos, which is a more secure and peformant authentication scheme than NTLM:

C#

```
using Microsoft.AspNetCore.Authentication.Negotiate;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthentication(NegotiateDefaults.AuthenticationScheme)
   .AddNegotiate();

builder.Services.AddAuthorization(options =>
{
    options.FallbackPolicy = options.DefaultPolicy;
```

```
});
builder.Services.AddRazorPages();

var app = builder.Build();
```

NegotiateDefaults.AuthenticationScheme specifies Kerberos because it's the default.

IIS, IISExpress, and Kestrel support both Kerberos and NTLM.

Examining the WWW-Authenticate: header using IIS or IISExpress with a tool like Fiddler shows either `Negotiate` or NTLM.

Kestrel only shows `WWW-Authenticate: Negotiate`

The `WWW-Authenticate: Negotiate` header means that the server can use NTLM or Kerberos. Kestrel requires the Negotiate header prefix, it doesn't support directly specifying `NTLM` in the request or response auth headers. NTLM is supported in Kestrel, but it must be sent as `Negotiate`.

On Kestrel, to see if NTLM or Kerberos is used, Base64 decode the header and it shows either `NTLM` or `HTTP`. `HTTP` indicates Kerberos was used.

Linux and macOS environment configuration

Instructions for joining a Linux or macOS machine to a Windows domain are available in the Connect Azure Data Studio to your SQL Server using Windows authentication - Kerberos article. The instructions create a machine account for the Linux machine on the domain. SPNs must be added to that machine account.

> **Note**
>
> When following the guidance in the Connect Azure Data Studio to your SQL Server using Windows authentication - Kerberos article, replace `python-software-properties` with `python3-software-properties` if needed.

Once the Linux or macOS machine is joined to the domain, additional steps are required to provide a keytab file with the SPNs:

- On the domain controller, add new web service SPNs to the machine account:
  - `setspn -S HTTP/mywebservice.mydomain.com mymachine`
  - `setspn -S HTTP/mywebservice@MYDOMAIN.COM mymachine`
- Use ktpass to generate a keytab file:

    - `ktpass -princ HTTP/mywebservice.mydomain.com@MYDOMAIN.COM -pass myKeyTabFilePassword -mapuser MYDOMAIN\mymachine$ -pType KRB5_NT_PRINCIPAL -out c:\temp\mymachine.HTTP.keytab -crypto AES256-SHA1`
    - Some fields must be specified in uppercase as indicated.
- Copy the keytab file to the Linux or macOS machine.
- Select the keytab file via an environment variable: `export KRB5_KTNAME=/tmp/mymachine.HTTP.keytab`
- Invoke `klist` to show the SPNs currently available for use.

> ⓘ Note
>
> A keytab file contains domain access credentials and must be protected accordingly.

HTTP.sys

HTTP.sys supports Kernel Mode Windows Authentication using Negotiate, NTLM, or Basic authentication.

The following code adds authentication and configures the app's web host to use HTTP.sys with Windows Authentication:

C#

```
using Microsoft.AspNetCore.Server.HttpSys;
using System.Runtime.InteropServices;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthentication(HttpSysDefaults.AuthenticationScheme);

if (RuntimeInformation.IsOSPlatform(OSPlatform.Windows))
{
    builder.WebHost.UseHttpSys(options =>
        {
            options.Authentication.Schemes =
                AuthenticationSchemes.NTLM |
                AuthenticationSchemes.Negotiate;
            options.Authentication.AllowAnonymous = false;
        });
}
```

> ⓘ Note

HTTP.sys delegates to Kernel Mode authentication with the Kerberos authentication protocol. User Mode authentication isn't supported with Kerberos and HTTP.sys. The machine account must be used to decrypt the Kerberos token/ticket that's obtained from Active Directory and forwarded by the client to the server to authenticate the user. Register the Service Principal Name (SPN) for the host, not the user of the app.

> **Note**
>
> HTTP.sys isn't supported on Nano Server version 1709 or later. To use Windows Authentication and HTTP.sys with Nano Server, use a Server Core (microsoft/windowsservercore) container (see `https://hub.docker.com/_/microsoft-windows-servercore`). For more information on Server Core, see What is the Server Core installation option in Windows Server?.

Authorize users

The configuration state of anonymous access determines the way in which the [Authorize] and [AllowAnonymous] attributes are used in the app. The following two sections explain how to handle the disallowed and allowed configuration states of anonymous access.

Disallow anonymous access

When Windows Authentication is enabled and anonymous access is disabled, the [Authorize] and [AllowAnonymous] attributes have no effect. If an IIS site is configured to disallow anonymous access, the request never reaches the app. For this reason, the [AllowAnonymous] attribute isn't applicable.

Allow anonymous access

When both Windows Authentication and anonymous access are enabled, use the [Authorize] and [AllowAnonymous] attributes. The [Authorize] attribute allows you to secure endpoints of the app which require authentication. The [AllowAnonymous] attribute overrides the [Authorize] attribute in apps that allow anonymous access. For attribute usage details, see Simple authorization in ASP.NET Core.

> **Note**

By default, users who lack authorization to access a page are presented with an empty HTTP 403 response. The StatusCodePages Middleware can be configured to provide users with a better "Access Denied" experience.

Impersonation

ASP.NET Core doesn't implement impersonation. Apps run with the app's identity for all requests, using app pool or process identity. If the app should perform an action on behalf of a user, use WindowsIdentity.RunImpersonated or RunImpersonatedAsync in a terminal inline middleware in `Program.cs`. Run a single action in this context and then close the context.

C#

```
app.Run(async (context) =>
{
    try
    {
        var user = (WindowsIdentity)context.User.Identity!;

        await context.Response
            .WriteAsync($"User: {user.Name}\tState:
{user.ImpersonationLevel}\n");

        await WindowsIdentity.RunImpersonatedAsync(user.AccessToken, async
() =>
        {
            var impersonatedUser = WindowsIdentity.GetCurrent();
            var message =
                $"User: {impersonatedUser.Name}\t" +
                $"State: {impersonatedUser.ImpersonationLevel}";

            var bytes = Encoding.UTF8.GetBytes(message);
            await context.Response.Body.WriteAsync(bytes, 0, bytes.Length);
        });
    }
    catch (Exception e)
    {
        await context.Response.WriteAsync(e.ToString());
    }
});
```

While the Microsoft.AspNetCore.Authentication.Negotiate package enables authentication on Windows, Linux, and macOS, impersonation is only supported on Windows.

Claims transformations

When hosting with IIS, AuthenticateAsync isn't called internally to initialize a user. Therefore, an IClaimsTransformation implementation used to transform claims after every authentication isn't activated by default. For more information and a code example that activates claims transformations, see Differences between in-process and out-of-process hosting.

Additional resources

- dotnet publish
- Host ASP.NET Core on Windows with IIS
- ASP.NET Core Module (ANCM) for IIS
- Visual Studio publish profiles (.pubxml) for ASP.NET Core app deployment

Authenticate users with WS-Federation in ASP.NET Core

Article • 06/18/2024

This tutorial demonstrates how to enable users to sign in with a WS-Federation authentication provider like Active Directory Federation Services (ADFS) or Microsoft Entra ID. It uses the ASP.NET Core sample app described in Facebook, Google, and external provider authentication.

For ASP.NET Core apps, WS-Federation support is provided by Microsoft.AspNetCore.Authentication.WsFederation. This component is ported from Microsoft.Owin.Security.WsFederation and shares many of that component's mechanics. However, the components differ in a couple of important ways.

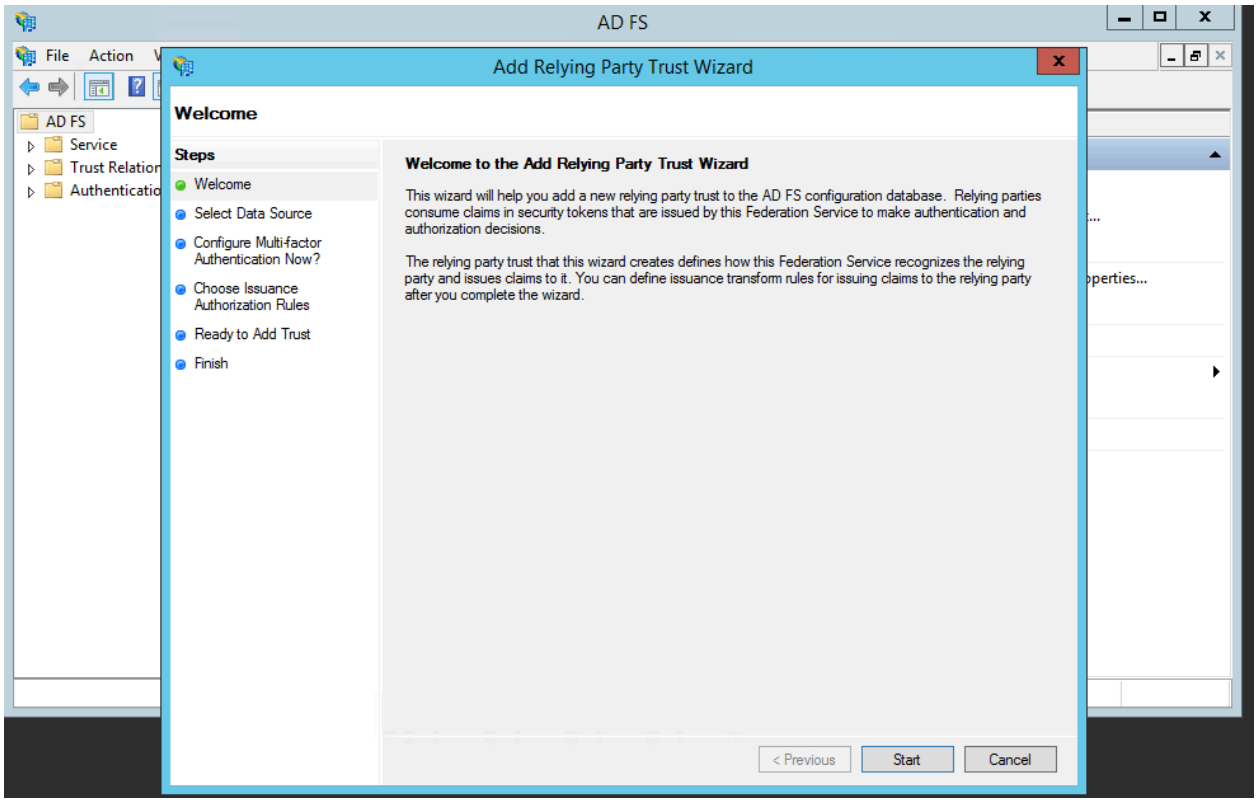
By default, the new middleware:

- Doesn't allow unsolicited logins. This feature of the WS-Federation protocol is vulnerable to XSRF attacks. However, it can be enabled with the `AllowUnsolicitedLogins` option.
- Doesn't check every form post for sign-in messages. Only requests to the `CallbackPath` are checked for sign-ins. `CallbackPath` defaults to `/signin-wsfed` but can be changed via the inherited RemoteAuthenticationOptions.CallbackPath property of the WsFederationOptions class. This path can be shared with other authentication providers by enabling the SkipUnrecognizedRequests option.

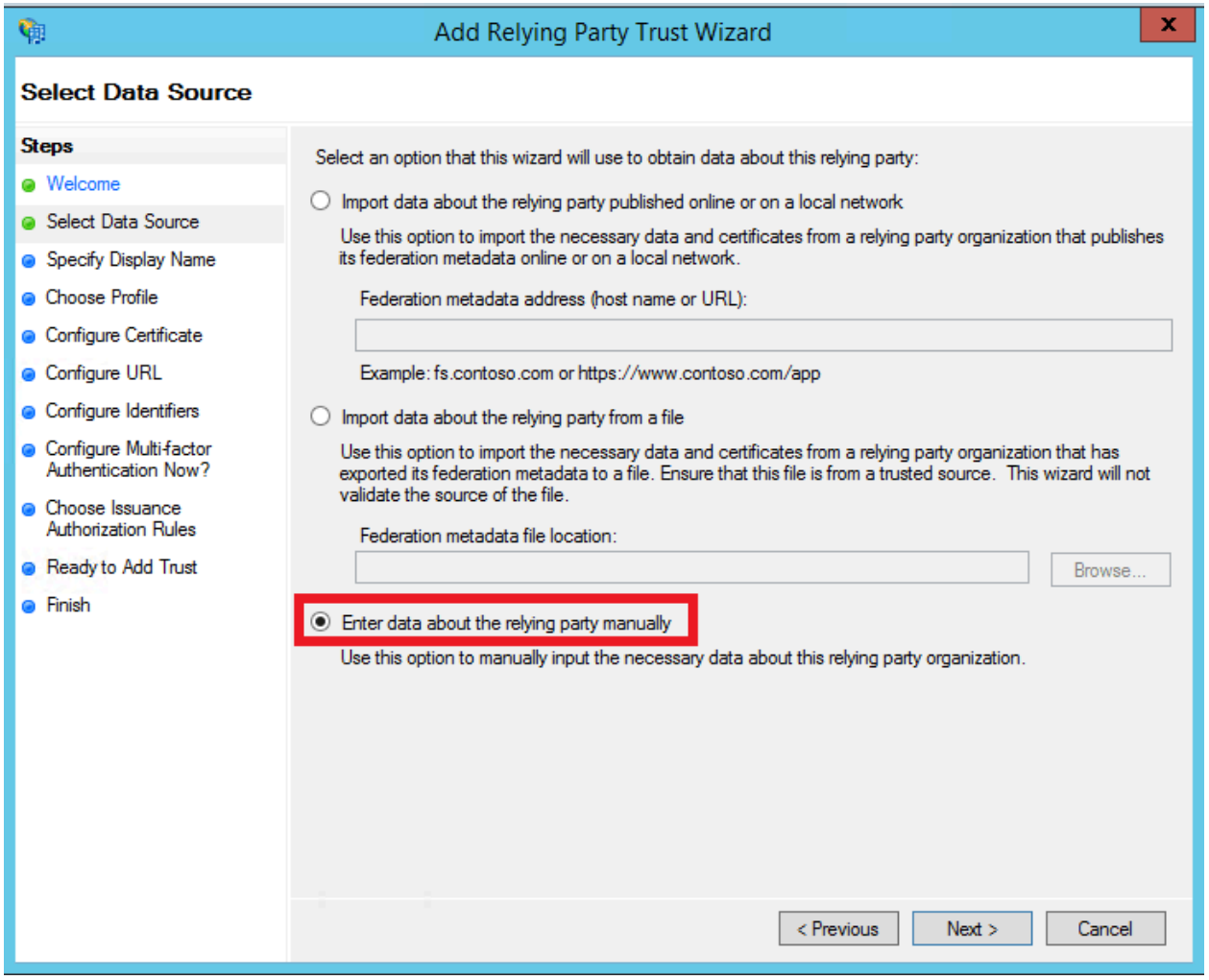
Register the app with Active Directory

Active Directory Federation Services

- Open the server's **Add Relying Party Trust Wizard** from the ADFS Management console:

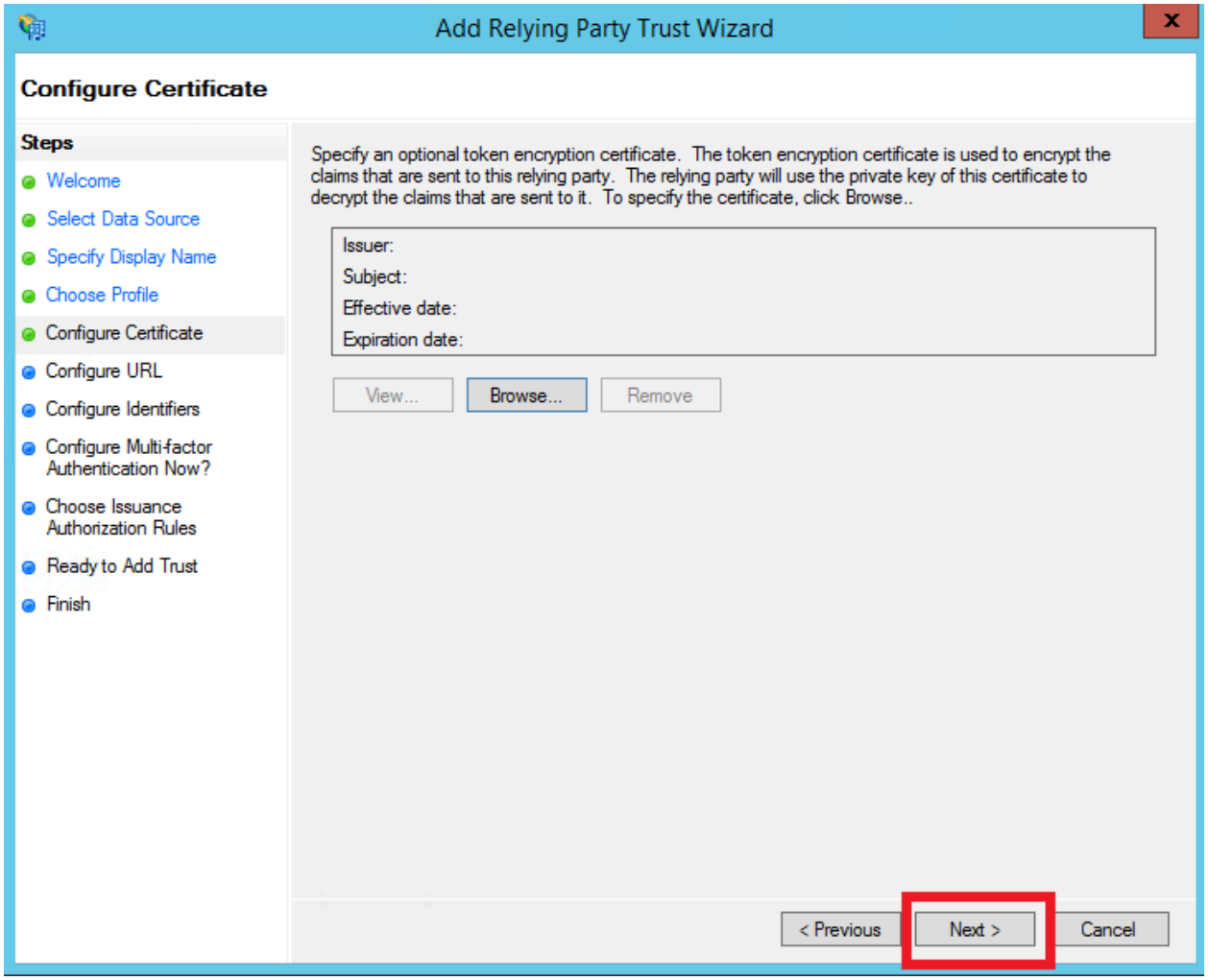


- Choose to enter data manually:

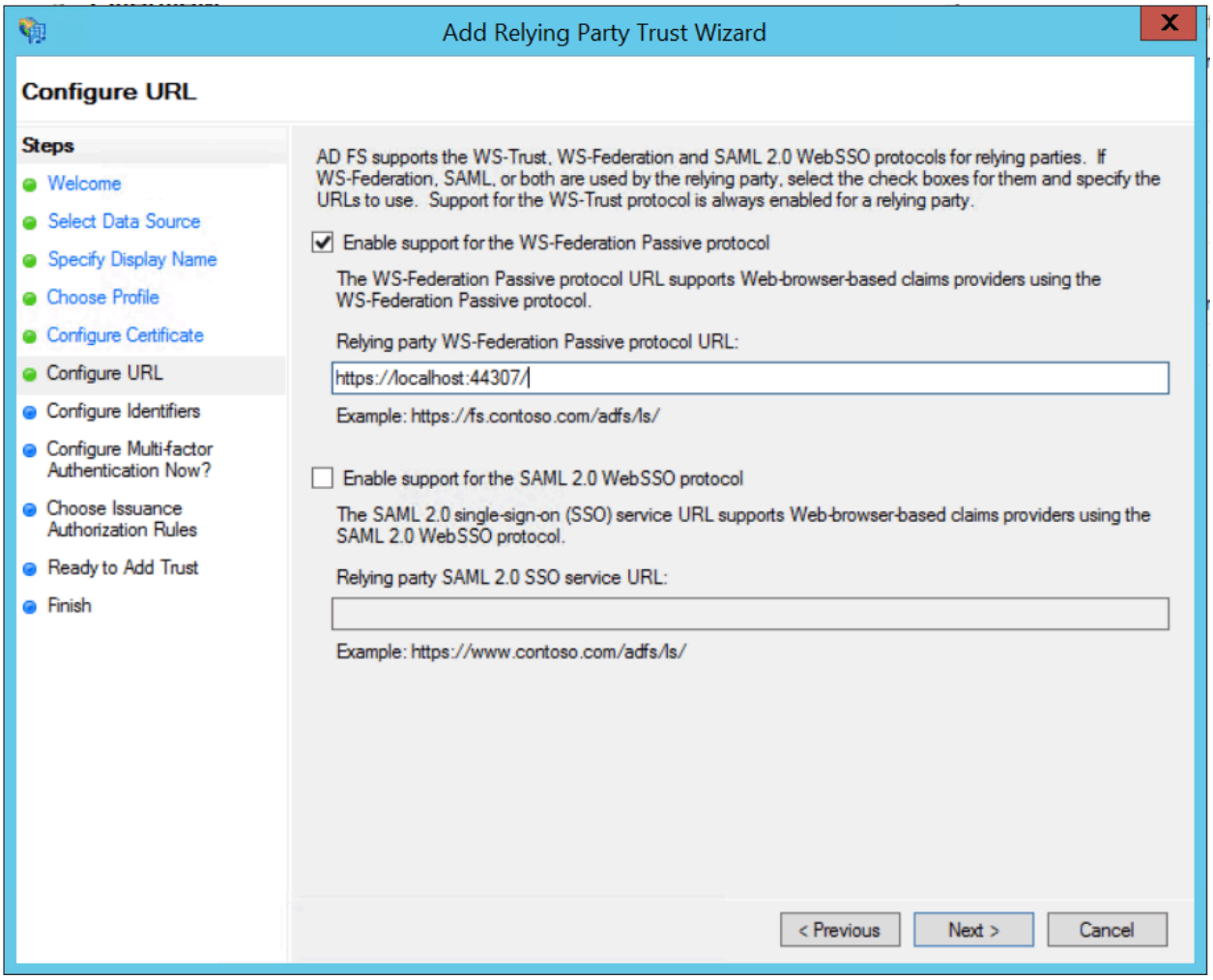


- Enter a display name for the relying party. The name isn't important to the ASP.NET Core app.

- Microsoft.AspNetCore.Authentication.WsFederation lacks support for token encryption, so don't configure a token encryption certificate:

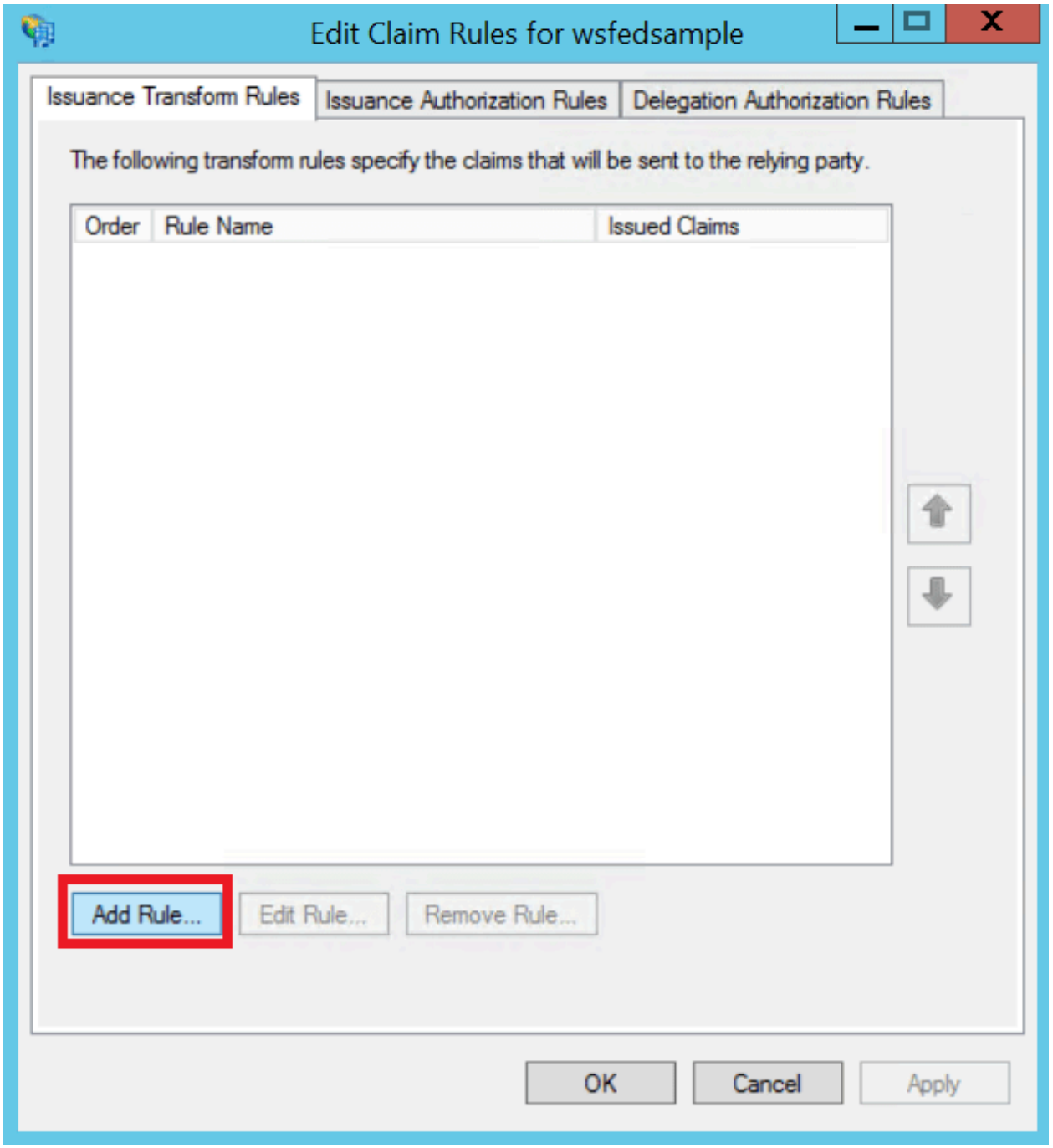


- Enable support for WS-Federation Passive protocol, using the app's URL. Verify the port is correct for the app:

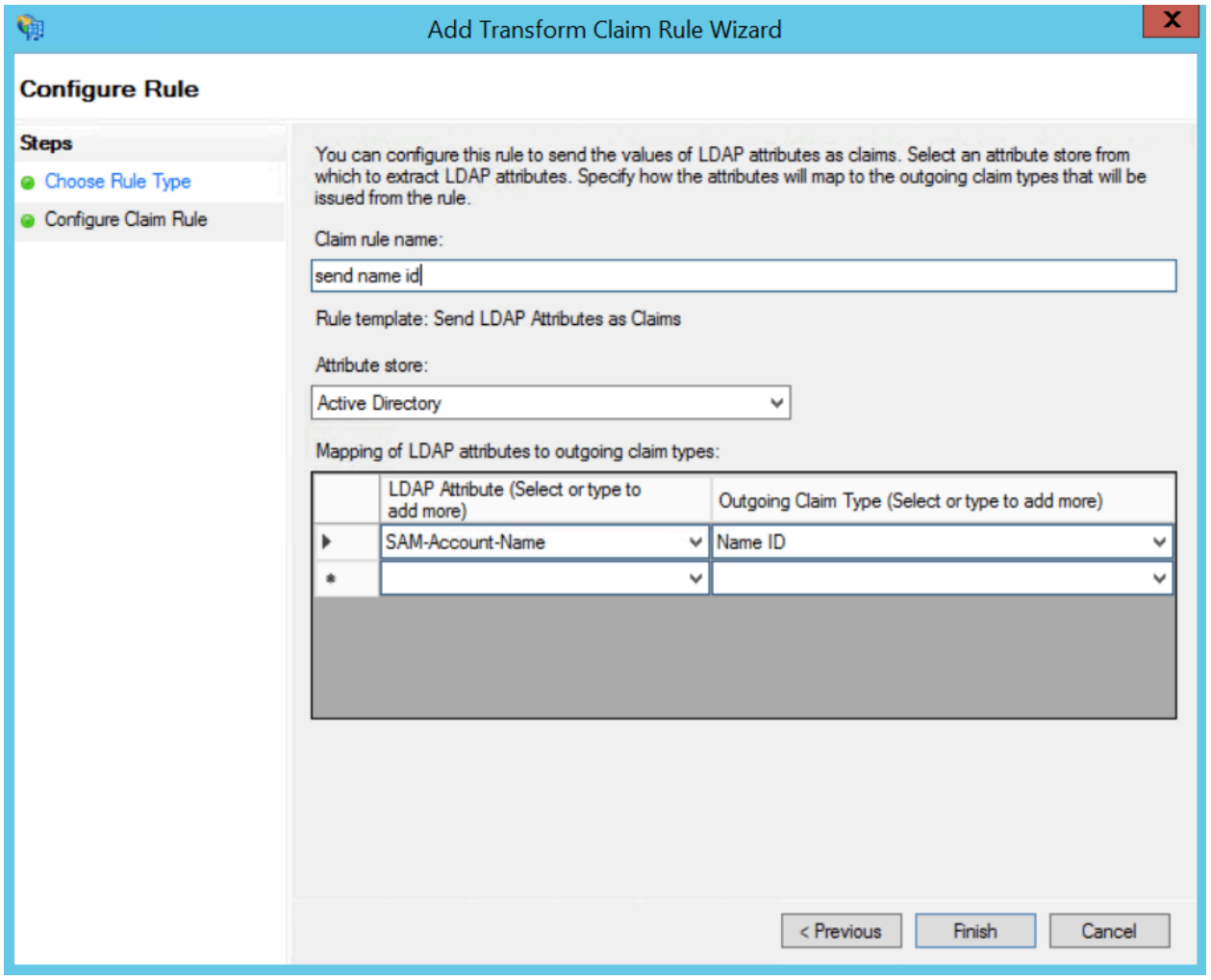


> ⓘ **Note**
>
> This must be an HTTPS URL. IIS Express can provide a self-signed certificate when hosting the app during development. Kestrel requires manual certificate configuration. See the Kestrel documentation for more details.

- Click **Next** through the rest of the wizard and **Close** at the end.
- ASP.NET Core Identity requires a **Name ID** claim. Add one from the **Edit Claim Rules** dialog:



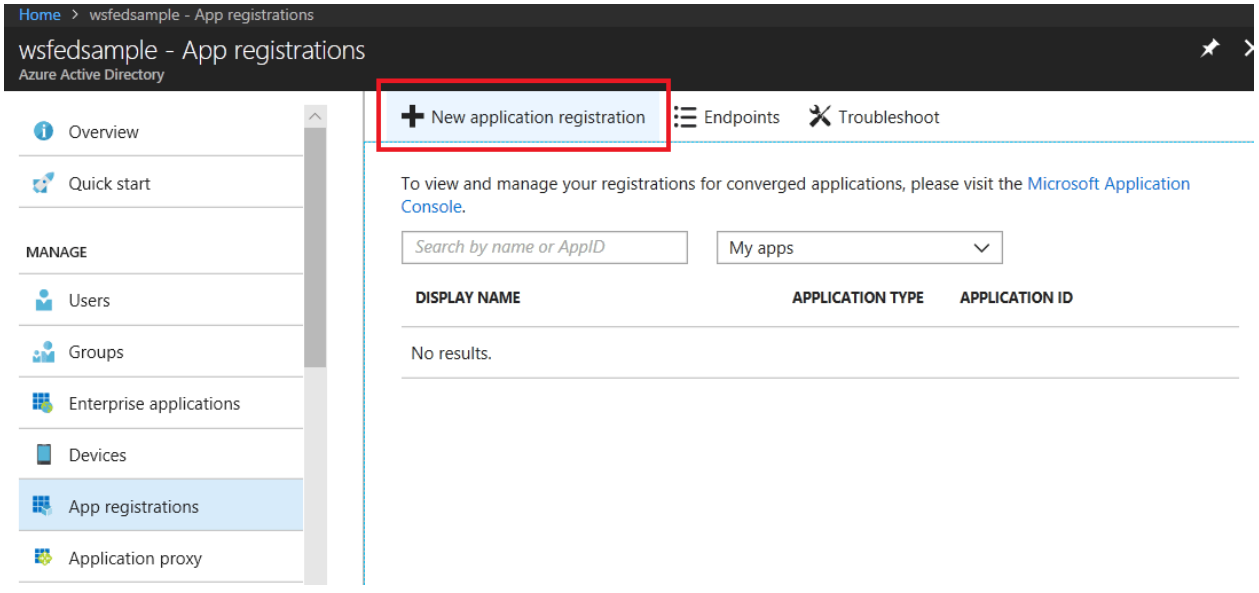
- In the **Add Transform Claim Rule Wizard**, leave the default **Send LDAP Attributes as Claims** template selected, and click **Next**. Add a rule mapping the **SAM-Account-Name** LDAP attribute to the **Name ID** outgoing claim:



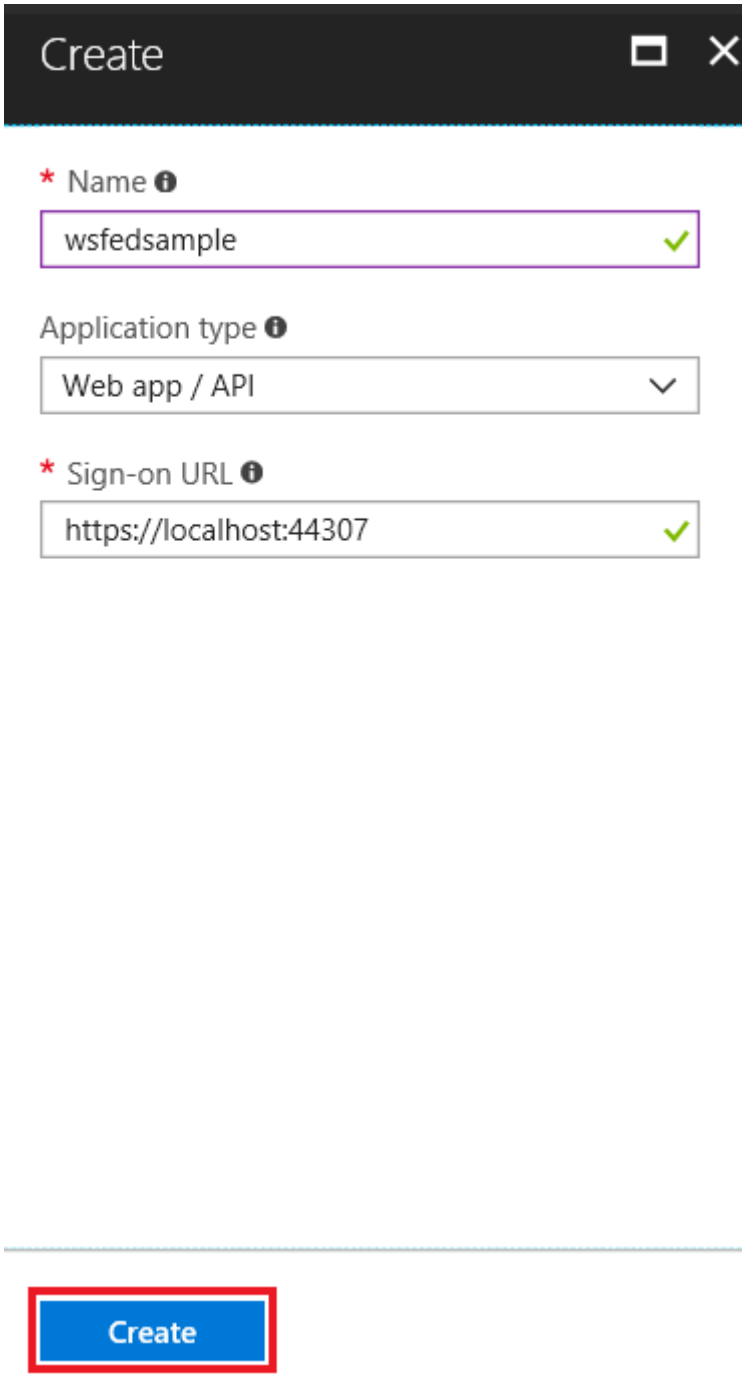
- Click **Finish** > **OK** in the **Edit Claim Rules** window.

Microsoft Entra ID

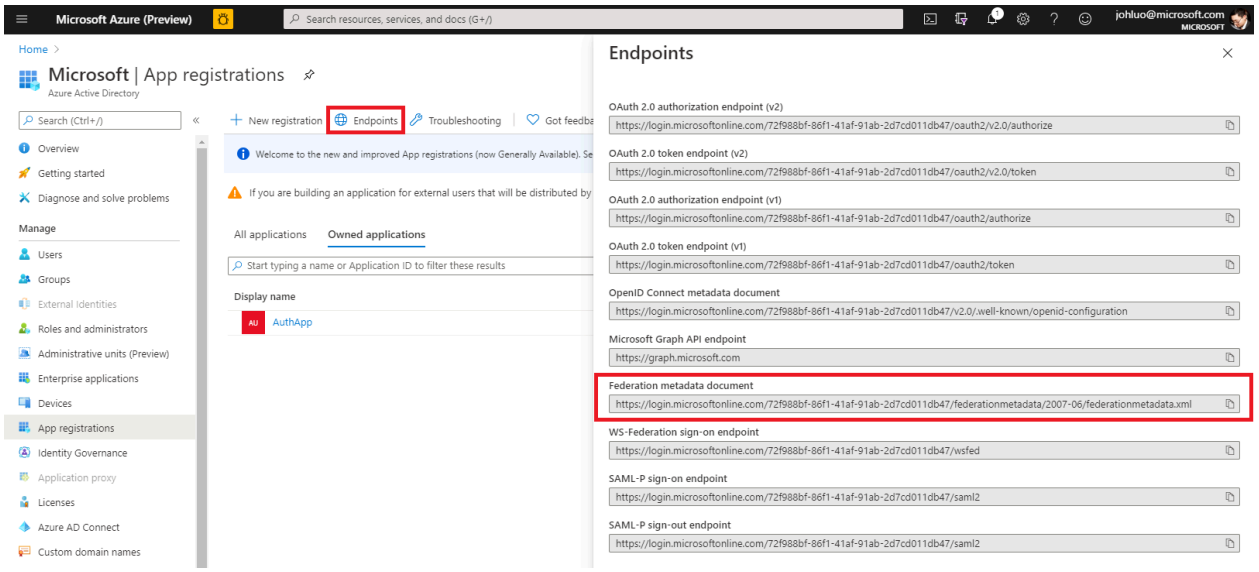
- Navigate to the Microsoft Entra ID tenant's app registrations blade. Click **New application registration**:



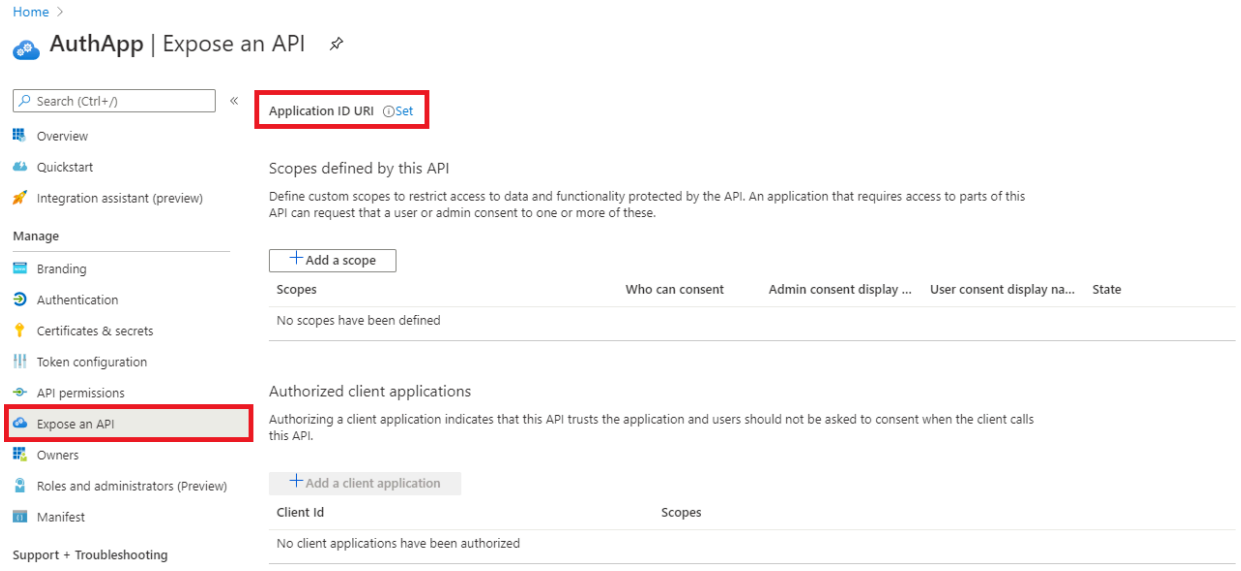
- Enter a name for the app registration. This isn't important to the ASP.NET Core app.
- Enter the URL the app listens on as the **Sign-on URL**:



- Click **Endpoints** and note the **Federation Metadata Document** URL. This is the WS-Federation middleware's `MetadataAddress`:



- Navigate to the new app registration. Click **Expose an API**. Click Application ID URI **Set** > **Save**. Make note of the **Application ID URI**. This is the WS-Federation middleware's `Wtrealm`:



Use WS-Federation without ASP.NET Core Identity

The WS-Federation middleware can be used without Identity. For example:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication(sharedOptions =>
    {
        sharedOptions.DefaultScheme =
CookieAuthenticationDefaults.AuthenticationScheme;
        sharedOptions.DefaultChallengeScheme =
WsFederationDefaults.AuthenticationScheme;
    })
```

```
     .AddWsFederation(options =>
     {
         options.Wtrealm = Configuration["wsfed:realm"];
         options.MetadataAddress = Configuration["wsfed:metadata"];
     })
     .AddCookie();

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

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }
    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

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

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
        endpoints.MapRazorPages();
    });
}
```

Add WS-Federation as an external login provider for ASP.NET Core Identity

- Add a dependency on Microsoft.AspNetCore.Authentication.WsFederation to the project.
- Add WS-Federation to `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
```

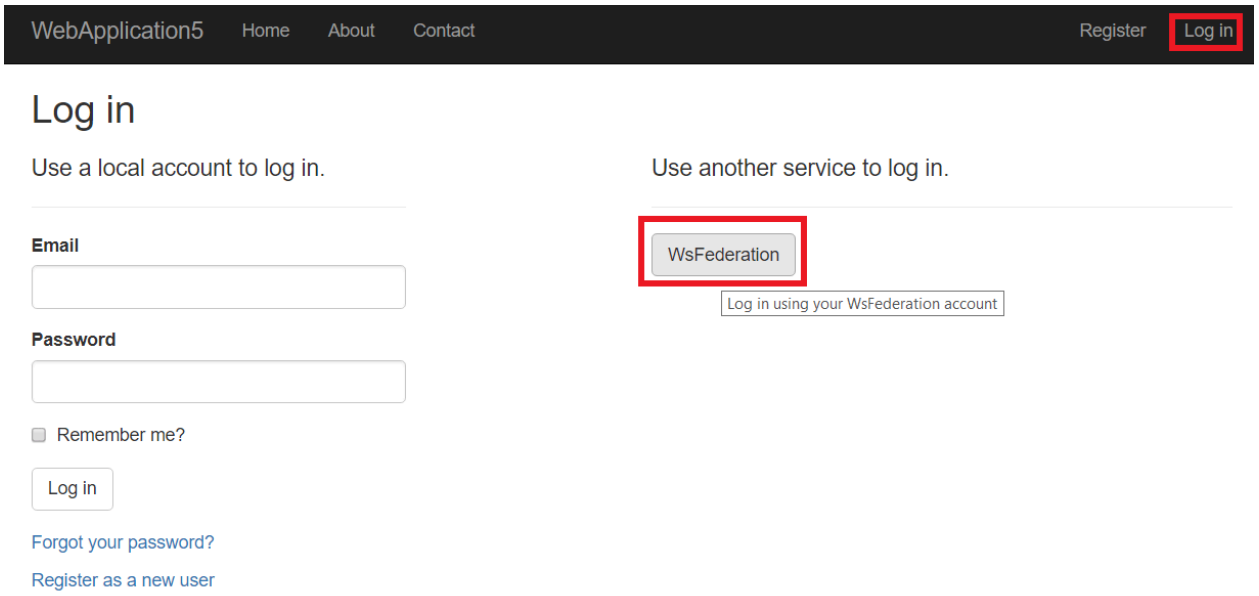
```
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationDbContext>();


    services.AddAuthentication()
        .AddWsFederation(options =>
        {
            // MetadataAddress represents the Active Directory instance used
to authenticate users.
            options.MetadataAddress = "https://<ADFS FQDN or AAD
tenant>/FederationMetadata/2007-06/FederationMetadata.xml";

            // Wtrealm is the app's identifier in the Active Directory
instance.
            // For ADFS, use the relying party's identifier, its WS-
Federation Passive protocol URL:
            options.Wtrealm = "https://localhost:44307/";

            // For AAD, use the Application ID URI from the app
registration's Overview blade:
            options.Wtrealm = "api://bbd35166-7c13-49f3-8041-9551f2847b69";
        });

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

The AddAuthentication(IServiceCollection, String) overload sets the DefaultScheme property. The AddAuthentication(IServiceCollection, Action<AuthenticationOptions>) overload allows configuring authentication options, which can be used to set up default authentication schemes for different purposes. Subsequent calls to `AddAuthentication` override previously configured AuthenticationOptions properties.

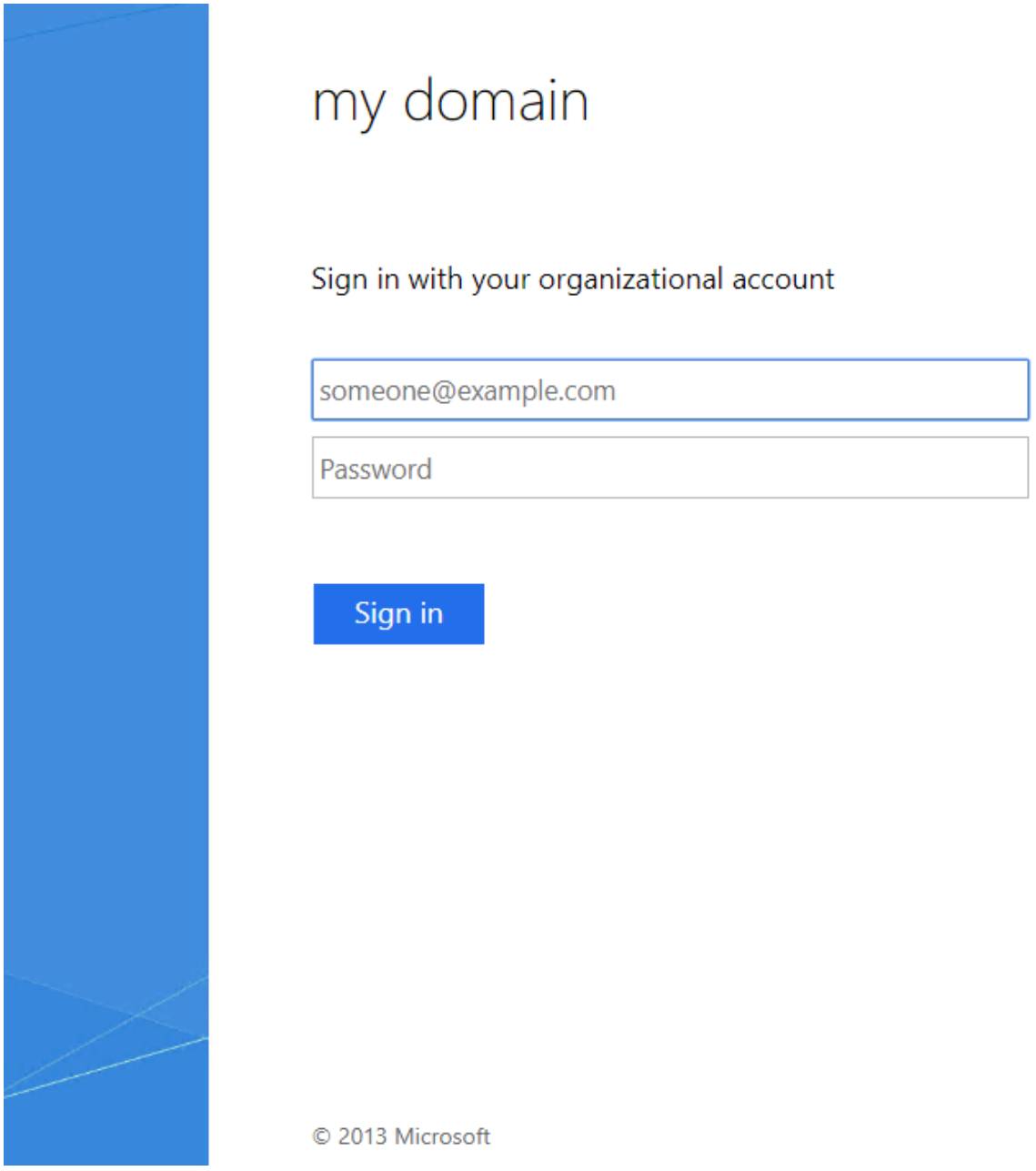
AuthenticationBuilder extension methods that register an authentication handler may only be called once per authentication scheme. Overloads exist that allow configuring the scheme properties, scheme name, and display name.

Log in with WS-Federation

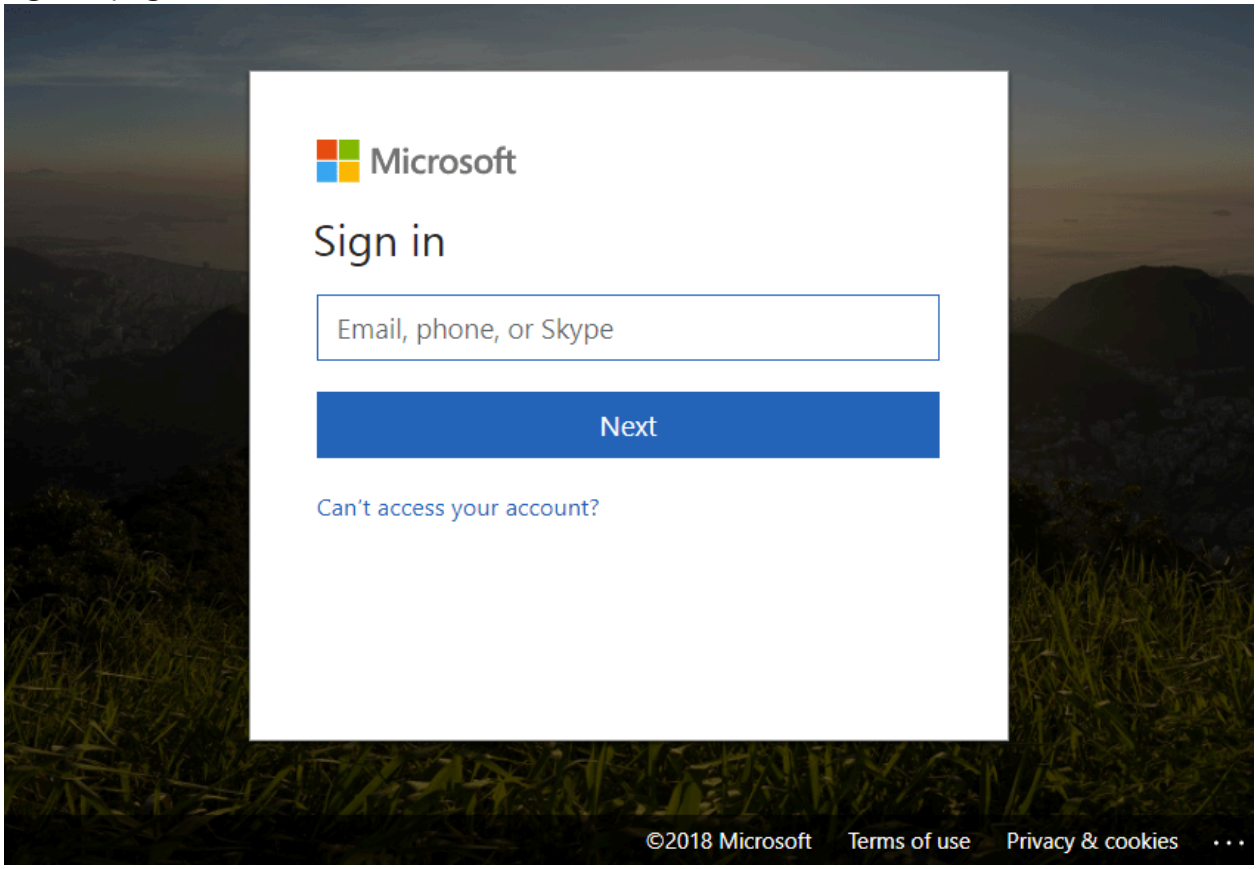
Browse to the app and click the **Log in** link in the nav header. There's an option to log in with WsFederation:



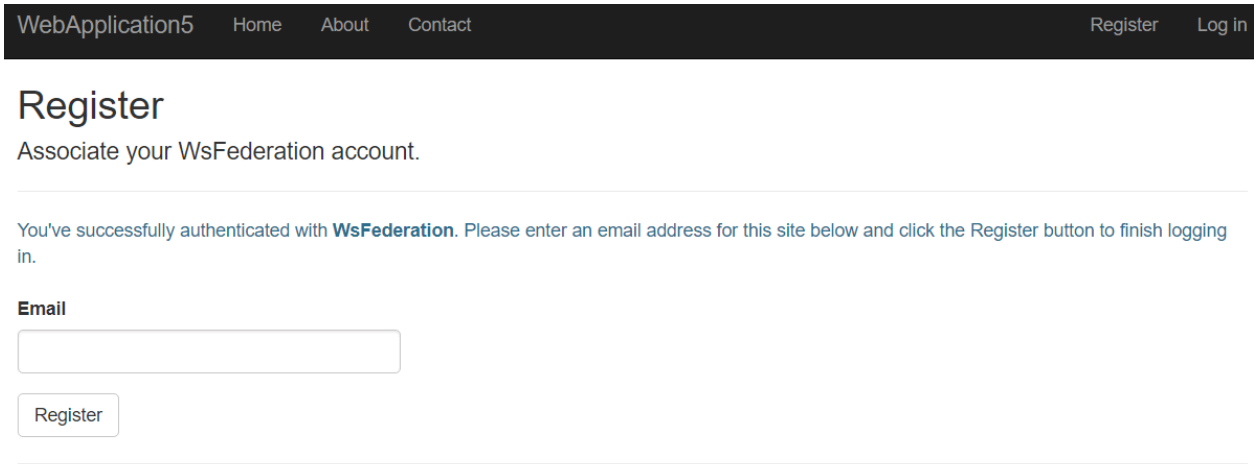
With ADFS as the provider, the button redirects to an ADFS sign-in page:



With Microsoft Entra ID as the provider, the button redirects to a Microsoft Entra ID sign-in page:



A successful sign-in for a new user redirects to the app's user registration page:



Use social sign-in provider authentication without ASP.NET Core Identity

Article • 06/03/2022

By Kirk Larkin and Rick Anderson

Facebook and Google authentication in ASP.NET Core describes how to enable users to sign in using OAuth 2.0 with credentials from external authentication providers. The approach described in that article includes ASP.NET Core Identity as an authentication provider.

This sample demonstrates how to use an external authentication provider **without** ASP.NET Core Identity. This approach is useful for apps that don't require all of the features of ASP.NET Core Identity, but still require integration with a trusted external authentication provider.

This sample uses Google authentication for authenticating users. Using Google authentication shifts many of the complexities of managing the sign-in process to Google. To integrate with a different external authentication provider, see the following articles:

- Facebook authentication
- Microsoft authentication
- Twitter authentication
- Other providers

Configuration

In `Program.cs`, configure the app's authentication schemes with the AddAuthentication, AddCookie, and AddGoogle methods:

C#

```
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.AspNetCore.Authentication.Google;

var builder = WebApplication.CreateBuilder(args);

builder.Services
    .AddAuthentication(options =>
    {
```

```
        options.DefaultScheme =
CookieAuthenticationDefaults.AuthenticationScheme;
        options.DefaultChallengeScheme =
GoogleDefaults.AuthenticationScheme;
    })
    .AddCookie()
    .AddGoogle(options =>
    {
        options.ClientId =
builder.Configuration["Authentication:Google:ClientId"];
        options.ClientSecret =
builder.Configuration["Authentication:Google:ClientSecret"];
    });

builder.Services.AddRazorPages();
```

The call to AddAuthentication sets the app's DefaultScheme. The `DefaultScheme` is the default scheme used by the following `HttpContext` authentication extension methods:

- AuthenticateAsync
- ChallengeAsync
- ForbidAsync
- SignInAsync
- SignOutAsync

Setting the app's `DefaultScheme` to CookieAuthenticationDefaults.AuthenticationScheme ("Cookies") configures the app to use Cookies as the default scheme for these extension methods. Setting the app's DefaultChallengeScheme to GoogleDefaults.AuthenticationScheme ("Google") configures the app to use Google as the default scheme for calls to `ChallengeAsync`. `DefaultChallengeScheme` overrides `DefaultScheme`. See AuthenticationOptions for more properties that override `DefaultScheme` when set.

In `Program.cs`, call UseAuthentication and UseAuthorization. This middleware combination sets the HttpContext.User property and runs the Authorization Middleware for requests:

C#

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

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

app.MapRazorPages();
```

To learn more about authentication schemes, see Authentication Concepts. To learn more about cookie authentication, see Use cookie authentication without ASP.NET Core Identity.

Apply authorization

Test the app's authentication configuration by applying the [Authorize] attribute to a controller, action, or page. The following code limits access to the *Privacy* page to users that have been authenticated:

C#

```
[Authorize]
public class PrivacyModel : PageModel
{

}
```

Save the access token

SaveTokens defines whether access and refresh tokens should be stored in the AuthenticationProperties after a successful authorization. `SaveTokens` is set to `false` by default to reduce the size of the final authentication cookie.

To save access and refresh tokens after a successful authorization, set `SaveTokens` to `true` in `Program.cs`:

C#

```
builder.Services
    .AddAuthentication(options =>
    {
        options.DefaultScheme =
CookieAuthenticationDefaults.AuthenticationScheme;
        options.DefaultChallengeScheme =
GoogleDefaults.AuthenticationScheme;
    })
    .AddCookie()
    .AddGoogle(options =>
    {
        options.ClientId =
builder.Configuration["Authentication:Google:ClientId"];
        options.ClientSecret =
builder.Configuration["Authentication:Google:ClientSecret"];
```

```
            options.SaveTokens = true;
        });
```

To retrieve a saved token, use GetTokenAsync. The following example retrieves the token named `access_token`:

```
C#

public async Task OnGetAsync()
{
    var accessToken = await HttpContext.GetTokenAsync(
        GoogleDefaults.AuthenticationScheme, "access_token");

    // ...
}
```

Sign out

To sign out the current user and delete their cookie, call SignOutAsync. The following code adds a `Logout` page handler to the *Index* page:

```
C#

public class IndexModel : PageModel
{
    public async Task<IActionResult> OnPostLogoutAsync()
    {
        // using Microsoft.AspNetCore.Authentication;
        await HttpContext.SignOutAsync();
        return RedirectToPage();
    }
}
```

Notice that the call to `SignOutAsync` doesn't specify an authentication scheme. The app uses the `DefaultScheme`, `CookieAuthenticationDefaults.AuthenticationScheme`, as a fallback.

Additional resources

- Simple authorization in ASP.NET Core
- Persist additional claims and tokens from external providers in ASP.NET Core

Policy schemes in ASP.NET Core

Article • 06/03/2022

Authentication policy schemes make it easier to have a single logical authentication scheme potentially use multiple approaches. For example, a policy scheme might use Google authentication for challenges, and cookie authentication for everything else. Authentication policy schemes make it:

- Easy to forward any authentication action to another scheme.
- Forward dynamically based on the request.

All authentication schemes that use derived AuthenticationSchemeOptions and the associated AuthenticationHandler<TOptions>:

- Are automatically policy schemes in ASP.NET Core 2.1 and later.
- Can be enabled via configuring the scheme's options.

C#

```
public class AuthenticationSchemeOptions
{
    /// <summary>
    /// If set, this specifies a default scheme that authentication handlers
should
    /// forward all authentication operations to, by default. The default
forwarding
    /// logic checks in this order:
    /// 1. The most specific
ForwardAuthenticate/Challenge/Forbid/SignIn/SignOut
    /// 2. The ForwardDefaultSelector
    /// 3. ForwardDefault
    /// The first non null result is used as the target scheme to forward
to.
    /// </summary>
    public string ForwardDefault { get; set; }

    /// <summary>
    /// If set, this specifies the target scheme that this scheme should
forward
    /// AuthenticateAsync calls to. For example:
    /// Context.AuthenticateAsync("ThisScheme") =>
    ///
Context.AuthenticateAsync("ForwardAuthenticateValue");
    /// Set the target to the current scheme to disable forwarding and allow
    /// normal processing.
    /// </summary>
    public string ForwardAuthenticate { get; set; }

    /// <summary>
```

```
    /// If set, this specifies the target scheme that this scheme should
forward
    /// ChallengeAsync calls to. For example:
    /// Context.ChallengeAsync("ThisScheme") =>
    ///
Context.ChallengeAsync("ForwardChallengeValue");
    /// Set the target to the current scheme to disable forwarding and allow
normal
    /// processing.
    /// </summary>
    public string ForwardChallenge { get; set; }

    /// <summary>
    /// If set, this specifies the target scheme that this scheme should
forward
    /// ForbidAsync calls to.For example:
    /// Context.ForbidAsync("ThisScheme")
    ///                                =>
Context.ForbidAsync("ForwardForbidValue");
    /// Set the target to the current scheme to disable forwarding and allow
normal
    /// processing.
    /// </summary>
    public string ForwardForbid { get; set; }

    /// <summary>
    /// If set, this specifies the target scheme that this scheme should
forward
    /// SignInAsync calls to. For example:
    /// Context.SignInAsync("ThisScheme") =>
    ///
Context.SignInAsync("ForwardSignInValue");
    /// Set the target to the current scheme to disable forwarding and allow
normal
    /// processing.
    /// </summary>
    public string ForwardSignIn { get; set; }

    /// <summary>
    /// If set, this specifies the target scheme that this scheme should
forward
    /// SignOutAsync calls to. For example:
    /// Context.SignOutAsync("ThisScheme") =>
    ///
Context.SignOutAsync("ForwardSignOutValue");
    /// Set the target to the current scheme to disable forwarding and allow
normal
    /// processing.
    /// </summary>
    public string ForwardSignOut { get; set; }

    /// <summary>
    /// Used to select a default scheme for the current request that
authentication
    /// handlers should forward all authentication operations to by default.
```

```
The
    /// default forwarding checks in this order:
    /// 1. The most specific
ForwardAuthenticate/Challenge/Forbid/SignIn/SignOut
    /// 2. The ForwardDefaultSelector
    /// 3. ForwardDefault.
    /// The first non null result will be used as the target scheme to
forward to.
    /// </summary>
    public Func<HttpContext, string> ForwardDefaultSelector { get; set; }
}
```

Examples

The following example shows a higher level scheme that combines lower level schemes. Google authentication is used for challenges, and cookie authentication is used for everything else:

C#

```
public void ConfigureServices(IServiceCollection services)
{

services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme
)
       .AddCookie(options => options.ForwardChallenge = "Google")
       .AddGoogle(options => { });
}
```

The following example enables dynamic selection of schemes on a per request basis. That is, how to mix cookies and API authentication:

C#

```
public void ConfigureServices(IServiceCollection services)
{

services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme
)
        .AddCookie(options =>
        {
            // For example, can foward any requests that start with /api
            // to the api scheme.
            options.ForwardDefaultSelector = ctx =>
               ctx.Request.Path.StartsWithSegments("/api") ? "Api" : null;
        })
        .AddYourApiAuth("Api");
}
```

Manage JSON Web Tokens in development with dotnet user-jwts

Article • 09/27/2024

By Rick Anderson

The `dotnet user-jwts` command line tool can create and manage app specific local JSON Web Tokens (JWTs).

Synopsis

.NET CLI

```
dotnet user-jwts [<PROJECT>] [command]
dotnet user-jwts [command] -h|--help
```

Description

Creates and manages project specific local JSON Web Tokens.

Arguments

`PROJECT | SOLUTION`

The MSBuild project to apply a command on. If a project is not specified, MSBuild searches the current working directory for a file that has a file extension that ends in *proj* and uses that file.

Commands

Expand table

Command	Description
clear	Delete all issued JWTs for a project.
create	Issue a new JSON Web Token.
remove	Delete a given JWT.

Command	Description
key	Display or reset the signing key used to issue JWTs.
list	Lists the JWTs issued for the project.
print	Display the details of a given JWT.

Create

Usage: `dotnet user-jwts create [options]`

Expand table

Option	Description
-p | --project	The path of the project to operate on. Defaults to the project in the current directory.
--scheme	The scheme name to use for the generated token. Defaults to 'Bearer'.
-n | --name	The name of the user to create the JWT for. Defaults to the current environment user.
--audience	The audiences to create the JWT for. Defaults to the URLs configured in the project's launchSettings.json.
--issuer	The issuer of the JWT. Defaults to 'dotnet-user-jwts'.
--scope	A scope claim to add to the JWT. Specify once for each scope.
--role	A role claim to add to the JWT. Specify once for each role.
--claim	Claims to add to the JWT. Specify once for each claim in the format "name=value".
--not-before	The UTC date & time the JWT should not be valid before in the format 'yyyy-MM-dd [[HH:mm[[:ss]]]]'. Defaults to the date & time the JWT is created.
--expires-on	The UTC date & time the JWT should expire in the format 'yyyy-MM-dd [[[[HH:mm]]:ss]]'. Defaults to 6 months after the --not-before date. Do not use this option in conjunction with the --valid-for option.
--valid-for	The period the JWT should expire after. Specify using a number followed by duration type like 'd' for days, 'h' for hours, 'm' for minutes, and 's' for seconds, for example 365d'. Do not use this option in conjunction with the --expires-on option.
-o | --output	The format to use for displaying output from the command. Can be one of 'default', 'token', or 'json'.
-h | --help	Show help information

Examples

Run the following commands to create an empty web project and add the Microsoft.AspNetCore.Authentication.JwtBearer NuGet package:

.NET CLI

```
dotnet new web -o MyJWT
cd MyJWT
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
```

Replace the contents of `Program.cs` with the following code:

C#

```
using System.Security.Claims;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthorization();
builder.Services.AddAuthentication("Bearer").AddJwtBearer();

var app = builder.Build();

app.UseAuthorization();

app.MapGet("/", () => "Hello, World!");
app.MapGet("/secret", (ClaimsPrincipal user) => $"Hello
{user.Identity?.Name}. My secret")
    .RequireAuthorization();

app.Run();
```

In the preceding code, a GET request to `/secret` returns an `401 Unauthorized` error. A production app might get the JWT from a Security token service (STS), perhaps in response to logging in via a set of credentials. For the purpose of working with the API during local development, the `dotnet user-jwts` command line tool can be used to create and manage app-specific local JWTs.

The `user-jwts` tool is similar in concept to the user-secrets tool, it can be used to manage values for the app that are only valid for the developer on the local machine. In fact, the user-jwts tool utilizes the `user-secrets` infrastructure to manage the key that the JWTs are signed with, ensuring it's stored safely in the user profile.

The `user-jwts` tool hides implementation details, such as where and how the values are stored. The tool can be used without knowing the implementation details. The values

are stored in a JSON file in the local machine's user profile folder:

Windows

File system path:

`%APPDATA%\Microsoft\UserSecrets\<secrets_GUID>\user-jwts.json`

Create a JWT

The following command creates a local JWT:

.NET CLI

```
dotnet user-jwts create
```

The preceding command creates a JWT and updates the project's `appsettings.Development.json` file with JSON similar to the following:

C#

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "Authentication": {
    "Schemes": {
      "Bearer": {
        "ValidAudiences": [
          "http://localhost:8401",
          "https://localhost:44308",
          "http://localhost:5182",
          "https://localhost:7076"
        ],
        "ValidIssuer": "dotnet-user-jwts"
      }
    }
  }
}
```

Copy the JWT and the `ID` created in the preceding command. Use a tool like Curl to test `/secret`:

.NET CLI

```
curl -i -H "Authorization: Bearer {token}" https://localhost:{port}/secret
```

Where `{token}` is the previously generated JWT.

Display JWT security information

The following command displays the JWT security information, including expiration, scopes, roles, token header and payload, and the compact token:

.NET CLI

```
dotnet user-jwts print {ID} --show-all
```

Create a token for a specific user and scope

See Create in this topic for supported create options.

The following command creates a JWT for a user named `MyTestUser`:

.NET CLI

```
dotnet user-jwts create --name MyTestUser --scope "myapi:secrets"
```

The preceding command has output similar to the following:

.NET CLI

```
New JWT saved with ID '43e0b748'.
Name: MyTestUser
Scopes: myapi:secrets

Token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.{Remaining token deleted}
```

The preceding token can be used to test the `/secret2` endpoint in the following code:

C#

```
using System.Security.Claims;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthorization();
```

```
builder.Services.AddAuthentication("Bearer").AddJwtBearer();

var app = builder.Build();

app.MapGet("/", () => "Hello, World!");
app.MapGet("/secret", (ClaimsPrincipal user) => $"Hello
{user.Identity?.Name}. My secret")
    .RequireAuthorization();
app.MapGet("/secret2", () => "This is a different secret!")
    .RequireAuthorization(p => p.RequireClaim("scope", "myapi:secrets"));

app.Run();
```

Mapping, customizing, and transforming claims in ASP.NET Core

Article • 08/16/2024

By Damien Bowden

Claims can be created from any user or identity data which can be issued using a trusted identity provider or ASP.NET Core identity. A claim is a name value pair that represents what the subject is, not what the subject can do. This article covers the following areas:

- How to configure and map claims using an OpenID Connect client
- Set the name and role claim
- Reset the claims namespaces
- Customize, extend the claims using TransformAsync

Mapping claims using OpenID Connect authentication

The profile claims can be returned in the `id_token`, which is returned after a successful authentication. The ASP.NET Core client app only requires the profile scope. When using the `id_token` for claims, no extra claims mapping is required.

C#

```
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.AspNetCore.Authentication.OpenIdConnect;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
```

```
builder.Services.AddAuthentication(options =>
{
    options.DefaultScheme =
CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme =
OpenIdConnectDefaults.AuthenticationScheme;
})
   .AddCookie()
   .AddOpenIdConnect(options =>
   {
       options.SignInScheme = "Cookies";
       options.Authority = "-your-identity-provider-";
       options.RequireHttpsMetadata = true;
       options.ClientId = "-your-clientid-";
       options.ClientSecret = "-your-client-secret-from-user-secrets-or-
keyvault";
       options.ResponseType = "code";
       options.UsePkce = true;
       options.Scope.Add("profile");
       options.SaveTokens = true;
   });
var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

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

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

app.MapRazorPages();

app.Run();
```

The preceding code requires the Microsoft.AspNetCore.Authentication.OpenIdConnect NuGet package.

Another way to get the user claims is to use the OpenID Connect User Info API. The ASP.NET Core client app uses the `GetClaimsFromUserInfoEndpoint` property to configure this. One important difference from the first settings, is that you must specify the claims you require using the `MapUniqueJsonKey` method, otherwise only the `name`, `given_name` and `email` standard claims will be available in the client app. The claims included in the `id_token` are mapped per default. This is the major difference to the first option. You must explicitly define some of the claims you require.

```
```

C#

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.AspNetCore.Authentication.OpenIdConnect;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddAuthentication(options =>
{
    options.DefaultScheme =
CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme =
OpenIdConnectDefaults.AuthenticationScheme;
})
   .AddCookie()
   .AddOpenIdConnect(options =>
   {
       options.SignInScheme = "Cookies";
       options.Authority = "-your-identity-provider-";
       options.RequireHttpsMetadata = true;
       options.ClientId = "-your-clientid-";
       options.ClientSecret = "-client-secret-from-user-secrets-or-
keyvault";
       options.ResponseType = "code";
       options.UsePkce = true;
       options.Scope.Add("profile");
       options.SaveTokens = true;
       options.GetClaimsFromUserInfoEndpoint = true;
       options.ClaimActions.MapUniqueJsonKey("preferred_username",
                                             "preferred_username");
       options.ClaimActions.MapUniqueJsonKey("gender", "gender");
   });

var app = builder.Build();

// Code removed for brevity.
```

> **Note**
>
> The default Open ID Connect handler uses Pushed Authorization Requests (PAR) if the identity provider's discovery document advertises support for PAR. The identity provider's discovery document is usually found at `.well-known/openid-configuration`. If you cannot use PAR in the client configuration on the identity provider, PAR can be disabled by using the **PushedAuthorizationBehavior** option.

C#

```
builder.Services
    .AddAuthentication(options =>
    {
        options.DefaultScheme =
CookieAuthenticationDefaults.AuthenticationScheme;
        options.DefaultChallengeScheme =
OpenIdConnectDefaults.AuthenticationScheme;
    })
    .AddCookie()
    .AddOpenIdConnect("oidc", oidcOptions =>
    {
        // Other provider-specific configuration goes here.

        // The default value is PushedAuthorizationBehavior.UseIfAvailable.

        // 'OpenIdConnectOptions' does not contain a definition for
'PushedAuthorizationBehavior'
        // and no accessible extension method 'PushedAuthorizationBehavior'
accepting a first argument
        // of type 'OpenIdConnectOptions' could be found
        oidcOptions.PushedAuthorizationBehavior =
PushedAuthorizationBehavior.Disable;
    });
```

To ensure that authentication only succeeds if PAR is used, use PushedAuthorizationBehavior.Require instead. This change also introduces a new OnPushAuthorization event to OpenIdConnectEvents which can be used to customize the pushed authorization request or handle it manually. See the API proposal for more details.

Name claim and role claim mapping

The **Name** claim and the **Role** claim are mapped to default properties in the ASP.NET Core HTTP context. Sometimes it is required to use different claims for the default properties, or the name claim and the role claim do not match the default values. The claims can be mapped using the **TokenValidationParameters** property and set to any claim as required. The values from the claims can be used directly in the HttpContext **User.Identity.Name** property and the roles.

If the `User.Identity.Name` has no value or the roles are missing, please check the values in the returned claims and set the `NameClaimType` and the `RoleClaimType` values. The returned claims from the client authentication can be viewed in the HTTP context.

```
C#
```

```
builder.Services.AddAuthentication(options =>
{
    options.DefaultScheme =
CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme =
OpenIdConnectDefaults.AuthenticationScheme;
})
  .AddCookie()
  .AddOpenIdConnect(options =>
  {
        // Other options...
        options.TokenValidationParameters = new TokenValidationParameters
        {
           NameClaimType = "email"
           //, RoleClaimType = "role"
        };
  });
```

Claims namespaces, default namespaces

ASP.NET Core adds default namespaces to some known claims, which might not be required in the app. Optionally, disable these added namespaces and use the exact claims that the OpenID Connect server created.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

JsonWebTokenHandler.DefaultInboundClaimTypeMap.Clear();

builder.Services.AddAuthentication(options =>
{
    options.DefaultScheme =
CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme =
OpenIdConnectDefaults.AuthenticationScheme;
})
   .AddCookie()
   .AddOpenIdConnect(options =>
   {
       options.SignInScheme = "Cookies";
       options.Authority = "-your-identity-provider-";
       options.RequireHttpsMetadata = true;
       options.ClientId = "-your-clientid-";
       options.ClientSecret = "-your-client-secret-from-user-secrets-or-
keyvault";
       options.ResponseType = "code";
       options.UsePkce = true;
```

```
        options.Scope.Add("profile");
        options.SaveTokens = true;
    });

var app = builder.Build();

// Code removed for brevity.
```

If you need to disable the namespaces per scheme and not globally, you can use the **MapInboundClaims = false** option.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddAuthentication(options =>
{
    options.DefaultScheme =
CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme =
OpenIdConnectDefaults.AuthenticationScheme;
})
    .AddCookie()
    .AddOpenIdConnect(options =>
    {
        options.SignInScheme = "Cookies";
        options.Authority = "-your-identity-provider-";
        options.RequireHttpsMetadata = true;
        options.ClientId = "-your-clientid-";
        options.ClientSecret = "-your-client-secret-from-user-secrets-or-
keyvault";
        options.ResponseType = "code";
        options.UsePkce = true;
        options.MapInboundClaims = false;
        options.Scope.Add("profile");
        options.SaveTokens = true;
    });

var app = builder.Build();

// Code removed for brevity.
```

Extend or add custom claims using `IClaimsTransformation`

The IClaimsTransformation interface can be used to add extra claims to the ClaimsPrincipal class. The interface requires a single method TransformAsync. This

method might get called multiple times. Only add a new claim if it does not already exist in the `ClaimsPrincipal`. A `ClaimsIdentity` is created to add the new claims and this can be added to the `ClaimsPrincipal`.

C#

```
using Microsoft.AspNetCore.Authentication;
using System.Security.Claims;

public class MyClaimsTransformation : IClaimsTransformation
{
    public Task<ClaimsPrincipal> TransformAsync(ClaimsPrincipal principal)
    {
        ClaimsIdentity claimsIdentity = new ClaimsIdentity();
        var claimType = "myNewClaim";
        if (!principal.HasClaim(claim => claim.Type == claimType))
        {
            claimsIdentity.AddClaim(new Claim(claimType, "myClaimValue"));
        }

        principal.AddIdentity(claimsIdentity);
        return Task.FromResult(principal);
    }
}
```

The IClaimsTransformation interface and the `MyClaimsTransformation` class can be registered as a service:

C#

```
builder.Services.AddTransient<IClaimsTransformation, MyClaimsTransformation>
();
```

Map claims from external identity providers

Refer to the following document:

Persist additional claims and tokens from external providers in ASP.NET Core

Community OSS authentication options for ASP.NET Core

Article • 05/06/2024

This page shows community-provided open-source software (OSS) authentication options for ASP.NET Core. This page is periodically updated as new providers become available.

OSS authentication providers [.NET]

Expand table

Name	Description
Duende IdentityServer	IdentityServer is an OpenID Connect and OAuth 2.0 framework for ASP.NET Core.
OpenIddict	OAuth 2.0/OpenID Connect server for ASP.NET Core and ASP.NET 4.x.
FIDO2 .NET Library, WebAuthn	FIDO2 .NET library for FIDO2 / WebAuthn Attestation and Assertion using .NET

OSS authentication provider clients [.NET]

Expand table

Name	Description
OpenIddict	OAuth 2.0/OpenID Connect client for ASP.NET Core, ASP.NET 4.x and Windows/Linux desktop apps with built-in integrations for 80+ services such as Auth0, Microsoft Entra ID, GitHub, Google, Twitter or Yahoo.
AspNet.Security.OAuth.Providers	A collection of security middleware for ASP.NET Core apps to support social authentication.
AspNet.Security.OpenId.Providers	A collection of security middleware for ASP.NET Core apps to support OpenID 2.0 authentication providers like Steam.

Other OSS authentication providers

Expand table

Name	Description
Gluu Server	Enterprise ready, open source software for identity, access management (IAM), and single sign-on (SSO). For more information, see the Gluu Product Documentation.
Keycloak	Open Source Identity and Access Management For Modern Applications and Services.
node-oidc-provider	OpenID Certified™ OAuth 2.0 Authorization Server implementation for Node.js.
Authentik	Authentik is an open-source Identity Provider focused on flexibility and versatility.

To add a provider, edit this page.

Identity management solutions for .NET web apps

Article • 10/18/2024

The following table provides an overview of various identity management solutions that can be used in ASP.NET Core apps. These solutions offer features and capabilities to manage user authentication, authorization, and user identity within an app. It includes options for apps that are:

- Container-based
- Self-hosted, where you manage the installation and infrastructure to support it.
- Managed, such as cloud-based services like Microsoft Entra

The following table lists both open source and commercial solutions in alphabetical order. Each line contains details such as license type, website, and documentation that is specific to ASP.NET Core integration. The table can help identify the identity management solutions that best align with your app's needs.

Many of the commercial licenses provide "community" or free options that may be available depending on your company size and app requirements.

Expand table

Name	Type	License Type	Documentation
ASP.NET Core Identity	Self host	OSS (MIT)	Secure a web app with ASP.NET Core Identity
Auth0	Managed	Commercial	Get started
Duende IdentityServer	Self host	Commercial	ASP.NET Identity integration
Keycloak	Container	OSS (Apache 2.0)	Keycloak securing apps documentation
Microsoft Entra ID	Managed	Commercial	Entra documentation
Okta	Managed	Commercial	Okta for ASP.NET Core
OpenIddict	Self host	OSS (Apache 2.0)	OpenIddict documentation

Is there a solution that should be added to this list? Do you have a correction, suggestion, or feedback? We welcome your contributions. Learn how to contribute.

Multi-factor authentication in ASP.NET Core

Article • 10/30/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By Damien Bowden

View or download sample code (damienbod/AspNetCoreHybridFlowWithApi GitHub repository)

Multi-factor authentication (MFA) is a process in which a user is requested during a sign-in event for additional forms of identification. This prompt could be to enter a code from a cellphone, use a FIDO2 key, or to provide a fingerprint scan. When you require a second form of authentication, security is enhanced. The additional factor isn't easily obtained or duplicated by a cyberattacker.

This article covers the following areas:

- What is MFA and what MFA flows are recommended
- Configure MFA for administration pages using ASP.NET Core Identity
- Send MFA sign-in requirement to OpenID Connect server
- Force ASP.NET Core OpenID Connect client to require MFA

MFA, 2FA

MFA requires at least two or more types of proof for an identity like something you know, something you possess, or biometric validation for the user to authenticate.

Two-factor authentication (2FA) is like a subset of MFA, but the difference being that MFA can require two or more factors to prove the identity.

2FA is supported by default when using ASP.NET Core Identity. To enable or disable 2FA for a specific user, set the IdentityUser<TKey>.TwoFactorEnabled property. The ASP.NET Core Identity Default UI includes pages for configuring 2FA.

MFA TOTP (Time-based One-time Password Algorithm)

MFA using TOTP is supported by default when using ASP.NET Core Identity. This approach can be used together with any compliant authenticator app, including:

- Microsoft Authenticator
- Google Authenticator

For implementation details, see Enable QR Code generation for TOTP authenticator apps in ASP.NET Core.

To disable support for MFA TOTP, configure authentication using AddIdentity instead of AddDefaultIdentity. `AddDefaultIdentity` calls AddDefaultTokenProviders internally, which registers multiple token providers including one for MFA TOTP. To register only specific token providers, call AddTokenProvider for each required provider. For more information about available token providers, see the AddDefaultTokenProviders source on GitHub.

MFA passkeys/FIDO2 or passwordless

passkeys/FIDO2 is currently:

- The most secure way of achieving MFA.
- MFA that protects against phishing attacks. (As well as certificate authentication and Windows for business)

At present, ASP.NET Core doesn't support passkeys/FIDO2 directly. Passkeys/FIDO2 can be used for MFA or passwordless flows.

Microsoft Entra ID provides support for passkeys/FIDO2 and passwordless flows. For more information, see Passwordless authentication options.

Other forms of passwordless MFA do not or may not protect against phishing.

MFA SMS

MFA with SMS increases security massively compared with password authentication (single factor). However, using SMS as a second factor is no longer recommended. Too many known attack vectors exist for this type of implementation.

NIST guidelines

Configure MFA for administration pages using ASP.NET Core Identity

MFA could be forced on users to access sensitive pages within an ASP.NET Core Identity app. This could be useful for apps where different levels of access exist for the different identities. For example, users might be able to view the profile data using a password login, but an administrator would be required to use MFA to access the administrative pages.

Extend the login with an MFA claim

The demo code is setup using ASP.NET Core with Identity and Razor Pages. The `AddIdentity` method is used instead of `AddDefaultIdentity` one, so an `IUserClaimsPrincipalFactory` implementation can be used to add claims to the identity after a successful login.

> ⚠ **Warning**
>
> This article shows the use of connection strings. With a local database the user doesn't have to be authenticated, but in production, connection strings sometimes include a password to authenticate. A resource owner password credential (ROPC) is a security risk that should be avoided in production databases. Production apps should use the most secure authentication flow available. For more information on authentication for apps deployed to test or production environments, see Secure authentication flows.

C#

```
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlite(
        Configuration.GetConnectionString("DefaultConnection")));

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

builder.Services.AddSingleton<IEmailSender, EmailSender>();
builder.Services.AddScoped<IUserClaimsPrincipalFactory<IdentityUser>,
    AdditionalUserClaimsPrincipalFactory>();

builder.Services.AddAuthorization(options =>
    options.AddPolicy("TwoFactorEnabled", x => x.RequireClaim("amr",
"mfa")));
```

```
builder.Services.AddRazorPages();
```

The `AdditionalUserClaimsPrincipalFactory` class adds the `amr` claim to the user claims only after a successful login. The claim's value is read from the database. The claim is added here because the user should only access the higher protected view if the identity has logged in with MFA. If the database view is read from the database directly instead of using the claim, it's possible to access the view without MFA directly after activating the MFA.

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.Options;
using System.Collections.Generic;
using System.Security.Claims;
using System.Threading.Tasks;

namespace IdentityStandaloneMfa
{
    public class AdditionalUserClaimsPrincipalFactory :
        UserClaimsPrincipalFactory<IdentityUser, IdentityRole>
    {
        public AdditionalUserClaimsPrincipalFactory(
            UserManager<IdentityUser> userManager,
            RoleManager<IdentityRole> roleManager,
            IOptions<IdentityOptions> optionsAccessor)
            : base(userManager, roleManager, optionsAccessor)
        {
        }

        public async override Task<ClaimsPrincipal> CreateAsync(IdentityUser
user)
        {
            var principal = await base.CreateAsync(user);
            var identity = (ClaimsIdentity)principal.Identity;

            var claims = new List<Claim>();

            if (user.TwoFactorEnabled)
            {
                claims.Add(new Claim("amr", "mfa"));
            }
            else
            {
                claims.Add(new Claim("amr", "pwd"));
            }

            identity.AddClaims(claims);
            return principal;
        }
```

```
    }
}
```

Because the Identity service setup changed in the `Startup` class, the layouts of the Identity need to be updated. Scaffold the Identity pages into the app. Define the layout in the `Identity/Account/Manage/_Layout.cshtml` file.

CSHTML

```
@{
    Layout = "/Pages/Shared/_Layout.cshtml";
}
```

Also assign the layout for all the manage pages from the Identity pages:

CSHTML

```
@{
    Layout = "_Layout.cshtml";
}
```

Validate the MFA requirement in the administration page

The administration Razor Page validates that the user has logged in using MFA. In the `OnGet` method, the identity is used to access the user claims. The `amr` claim is checked for the value `mfa`. If the identity is missing this claim or is `false`, the page redirects to the Enable MFA page. This is possible because the user has logged in already, but without MFA.

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace IdentityStandaloneMfa
{
    public class AdminModel : PageModel
    {
        public IActionResult OnGet()
        {
            var claimTwoFactorEnabled =
                User.Claims.FirstOrDefault(t => t.Type == "amr");
```

```
            if (claimTwoFactorEnabled != null &&
                "mfa".Equals(claimTwoFactorEnabled.Value))
            {
                // You logged in with MFA, do the administrative stuff
            }
            else
            {
                return Redirect(
                    "/Identity/Account/Manage/TwoFactorAuthentication");
            }

            return Page();
        }
    }
}
```

UI logic to toggle user login information

An authorization policy was added at startup. The policy requires the `amr` claim with the value `mfa`.

C#

```
services.AddAuthorization(options =>
    options.AddPolicy("TwoFactorEnabled",
        x => x.RequireClaim("amr", "mfa")));
```

This policy can then be used in the `_Layout` view to show or hide the **Admin** menu with the warning:

CSHTML

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

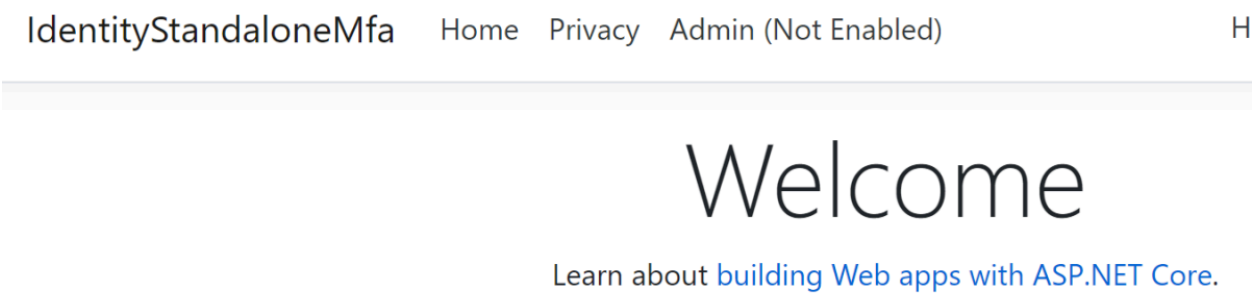
If the identity has logged in using MFA, the **Admin** menu is displayed without the tooltip warning. When the user has logged in without MFA, the **Admin (Not Enabled)** menu is displayed along with the tooltip that informs the user (explaining the warning).

CSHTML

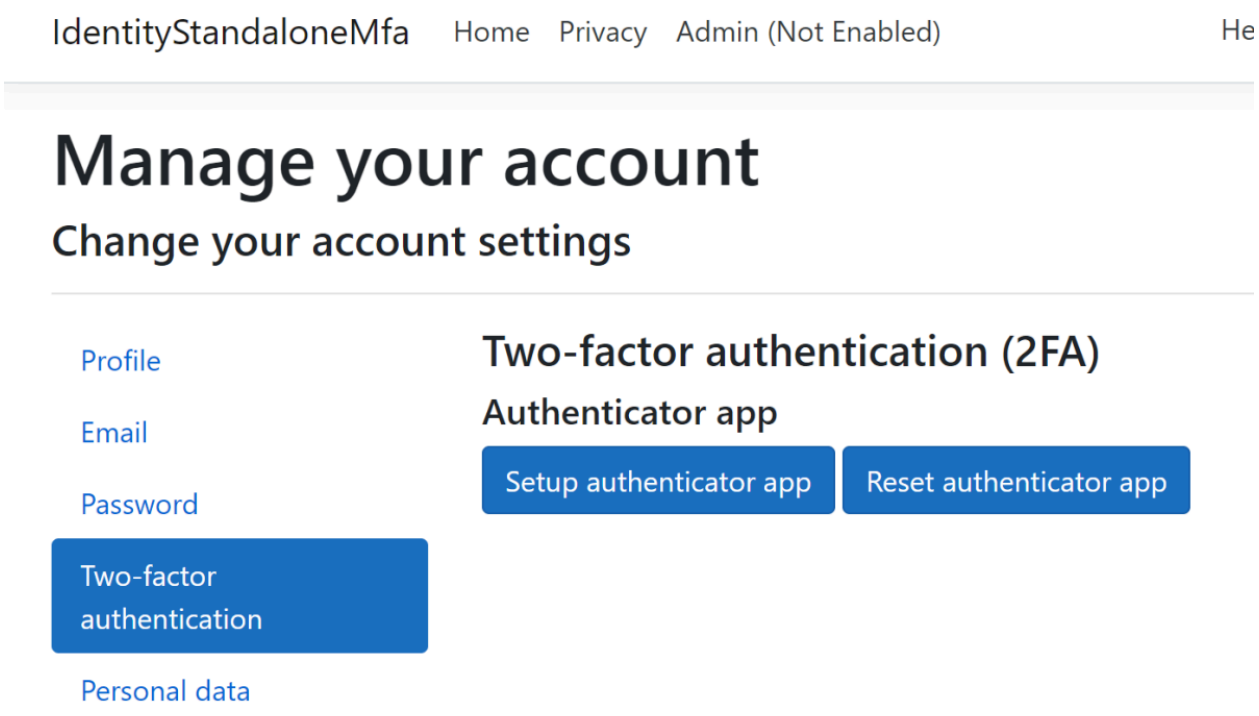
```
@if (SignInManager.IsSignedIn(User))
{
```

```
    @if ((AuthorizationService.AuthorizeAsync(User,
"TwoFactorEnabled")).Result.Succeeded)
    {
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="" asp-
page="/Admin">Admin</a>
        </li>
    }
    else
    {
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="" asp-page="/Admin"
               id="tooltip-demo"
               data-toggle="tooltip"
               data-placement="bottom"
               title="MFA is NOT enabled. This is required for the Admin
Page. If you have activated MFA, then logout, login again.">
                Admin (Not Enabled)
            </a>
        </li>
    }
}
```

If the user logs in without MFA, the warning is displayed:



The user is redirected to the MFA enable view when clicking the **Admin** link:



Send MFA sign-in requirement to OpenID Connect server

The `acr_values` parameter can be used to pass the `mfa` required value from the client to the server in an authentication request.

> **Note**
>
> The `acr_values` parameter needs to be handled on the OpenID Connect server for this to work.

OpenID Connect ASP.NET Core client

The ASP.NET Core Razor Pages OpenID Connect client app uses the `AddOpenIdConnect` method to login to the OpenID Connect server. The `acr_values` parameter is set with the `mfa` value and sent with the authentication request. The `OpenIdConnectEvents` is used to add this.

For recommended `acr_values` parameter values, see Authentication Method Reference Values.

```
C#
```

```
build.Services.AddAuthentication(options =>
{
    options.DefaultScheme =
        CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme =
        OpenIdConnectDefaults.AuthenticationScheme;
})
.AddCookie()
.AddOpenIdConnect(options =>
{
    options.SignInScheme =
        CookieAuthenticationDefaults.AuthenticationScheme;
    options.Authority = "<OpenID Connect server URL>";
    options.RequireHttpsMetadata = true;
    options.ClientId = "<OpenID Connect client ID>";
    options.ClientSecret = "<>";
    options.ResponseType = "code";
    options.UsePkce = true;
    options.Scope.Add("profile");
    options.Scope.Add("offline_access");
    options.SaveTokens = true;
    options.AdditionalAuthorizationParameters.Add("acr_values", "mfa");
});
```

Example OpenID Connect Duende IdentityServer server with ASP.NET Core Identity

On the OpenID Connect server, which is implemented using ASP.NET Core Identity with Razor Pages, a new page named `ErrorEnable2FA.cshtml` is created. The view:

- Displays if the Identity comes from an app that requires MFA but the user hasn't activated this in Identity.
- Informs the user and adds a link to activate this.

CSHTML

```
@{
    ViewData["Title"] = "ErrorEnable2FA";
}

<h1>The client application requires you to have MFA enabled. Enable this,
try login again.</h1>

<br />

You can enable MFA to login here:

<br />
```

```
<a href="~/Identity/Account/Manage/TwoFactorAuthentication">Enable MFA</a>
```

In the `Login` method, the `IIdentityServerInteractionService` interface implementation `_interaction` is used to access the OpenID Connect request parameters. The `acr_values` parameter is accessed using the `AcrValues` property. As the client sent this with `mfa` set, this can then be checked.

If MFA is required, and the user in ASP.NET Core Identity has MFA enabled, then the login continues. When the user has no MFA enabled, the user is redirected to the custom view `ErrorEnable2FA.cshtml`. Then ASP.NET Core Identity signs the user in.

The Fido2Store is used to check if the user has activated MFA using a custom FIDO2 Token Provider.

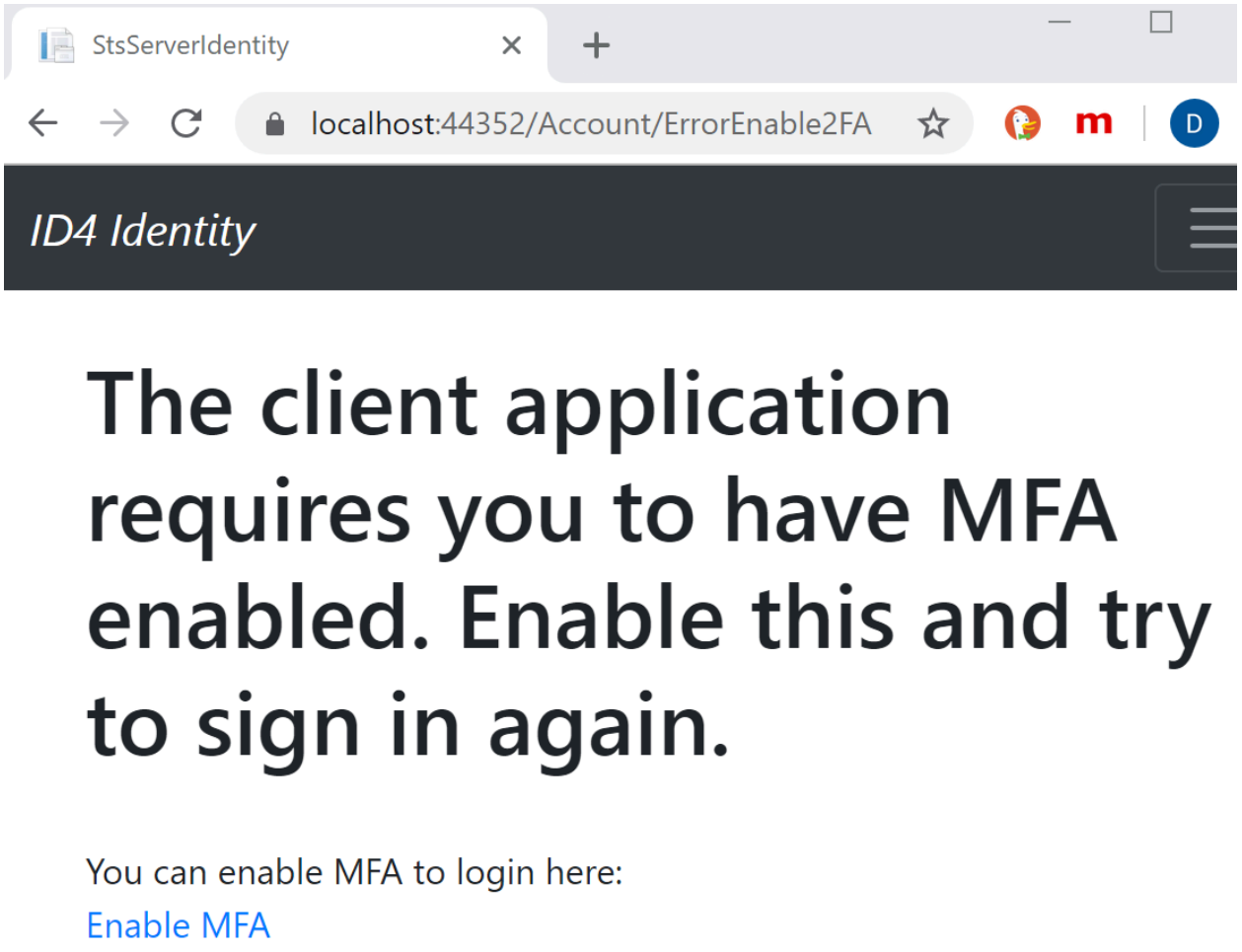
C#

```
public async Task<IActionResult> OnPost()
{
    // check if we are in the context of an authorization request
    var context = await
_interaction.GetAuthorizationContextAsync(Input.ReturnUrl);

    var requires2Fa = context?.AcrValues.Count(t => t.Contains("mfa")) >= 1;

    var user = await _userManager.FindByNameAsync(Input.Username);
    if (user != null && !user.TwoFactorEnabled && requires2Fa)
    {
        return RedirectToPage("/Home/ErrorEnable2FA/Index");
    }

    // code omitted for brevity

    if (ModelState.IsValid)
    {
        var result = await
_signInManager.PasswordSignInAsync(Input.Username, Input.Password,
Input.RememberLogin, lockoutOnFailure: true);
        if (result.Succeeded)
        {
            // code omitted for brevity
        }
        if (result.RequiresTwoFactor)
        {
            var fido2ItemExistsForUser = await
_fido2Store.GetCredentialsByUserNameAsync(user.UserName);
            if (fido2ItemExistsForUser.Count > 0)
            {
                return RedirectToPage("/Account/LoginFido2Mfa", new { area =
"Identity", Input.ReturnUrl, Input.RememberLogin });
```

```
            }

            return RedirectToPage("/Account/LoginWith2fa", new { area =
"Identity", Input.ReturnUrl, RememberMe = Input.RememberLogin });
        }

        await _events.RaiseAsync(new UserLoginFailureEvent(Input.Username,
"invalid credentials", clientId: context?.Client.ClientId));
        ModelState.AddModelError(string.Empty,
LoginOptions.InvalidCredentialsErrorMessage);
    }

    // something went wrong, show form with error
    await BuildModelAsync(Input.ReturnUrl);
    return Page();
}
```

If the user is already logged in, the client app:

- Still validates the `amr` claim.
- Can set up the MFA with a link to the ASP.NET Core Identity view.



Force ASP.NET Core OpenID Connect client to require MFA

This example shows how an ASP.NET Core Razor Page app, which uses OpenID Connect to sign in, can require that users have authenticated using MFA.

To validate the MFA requirement, an `IAuthorizationRequirement` requirement is created. This will be added to the pages using a policy that requires MFA.

C#

```
using Microsoft.AspNetCore.Authorization;

namespace AspNetCoreRequireMfaOidc;

public class RequireMfa : IAuthorizationRequirement{}
```

An `AuthorizationHandler` is implemented that will use the `amr` claim and check for the value `mfa`. The `amr` is returned in the `id_token` of a successful authentication and can have many different values as defined in the Authentication Method Reference Values specification.

The returned value depends on how the identity authenticated and on the OpenID Connect server implementation.

The `AuthorizationHandler` uses the `RequireMfa` requirement and validates the `amr` claim. The OpenID Connect server can be implemented using Duende Identity Server with ASP.NET Core Identity. When a user logs in using TOTP, the `amr` claim is returned with an MFA value. If using a different OpenID Connect server implementation or a different MFA type, the `amr` claim will, or can, have a different value. The code must be extended to accept this as well.

C#

```
public class RequireMfaHandler : AuthorizationHandler<RequireMfa>
{
    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context,
        RequireMfa requirement)
    {
        if (context == null)
            throw new ArgumentNullException(nameof(context));
        if (requirement == null)
            throw new ArgumentNullException(nameof(requirement));
```

```
        var amrClaim =
            context.User.Claims.FirstOrDefault(t => t.Type == "amr");

        if (amrClaim != null && amrClaim.Value == Amr.Mfa)
        {
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }
}
```

In the program file, the `AddOpenIdConnect` method is used as the default challenge scheme. The authorization handler, which is used to check the `amr` claim, is added to the Inversion of Control container. A policy is then created which adds the `RequireMfa` requirement.

C#

```
builder.Services.ConfigureApplicationCookie(options =>
        options.Cookie.SecurePolicy =
            CookieSecurePolicy.Always);

builder.Services.AddSingleton<IAuthorizationHandler, RequireMfaHandler>();

builder.Services.AddAuthentication(options =>
{
    options.DefaultScheme =
        CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme =
        OpenIdConnectDefaults.AuthenticationScheme;
})
.AddCookie()
.AddOpenIdConnect(options =>
{
    options.SignInScheme =
        CookieAuthenticationDefaults.AuthenticationScheme;
    options.Authority = "https://localhost:44352";
    options.RequireHttpsMetadata = true;
    options.ClientId = "AspNetCoreRequireMfaOidc";
    options.ClientSecret = "AspNetCoreRequireMfaOidcSecret";
    options.ResponseType = "code";
    options.UsePkce = true;
    options.Scope.Add("profile");
    options.Scope.Add("offline_access");
    options.SaveTokens = true;
});

builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("RequireMfa", policyIsAdminRequirement =>
    {
```

```
        policyIsAdminRequirement.Requirements.Add(new RequireMfa());
    });
});

builder.Services.AddRazorPages();
```

This policy is then used in the Razor page as required. The policy could be added globally for the entire app as well.

C#

```
[Authorize(Policy= "RequireMfa")]
public class IndexModel : PageModel
{
    public void OnGet()
    {
    }
}
```

If the user authenticates without MFA, the `amr` claim will probably have a `pwd` value. The request won't be authorized to access the page. Using the default values, the user will be redirected to the *Account/AccessDenied* page. This behavior can be changed or you can implement your own custom logic here. In this example, a link is added so that the valid user can set up MFA for their account.

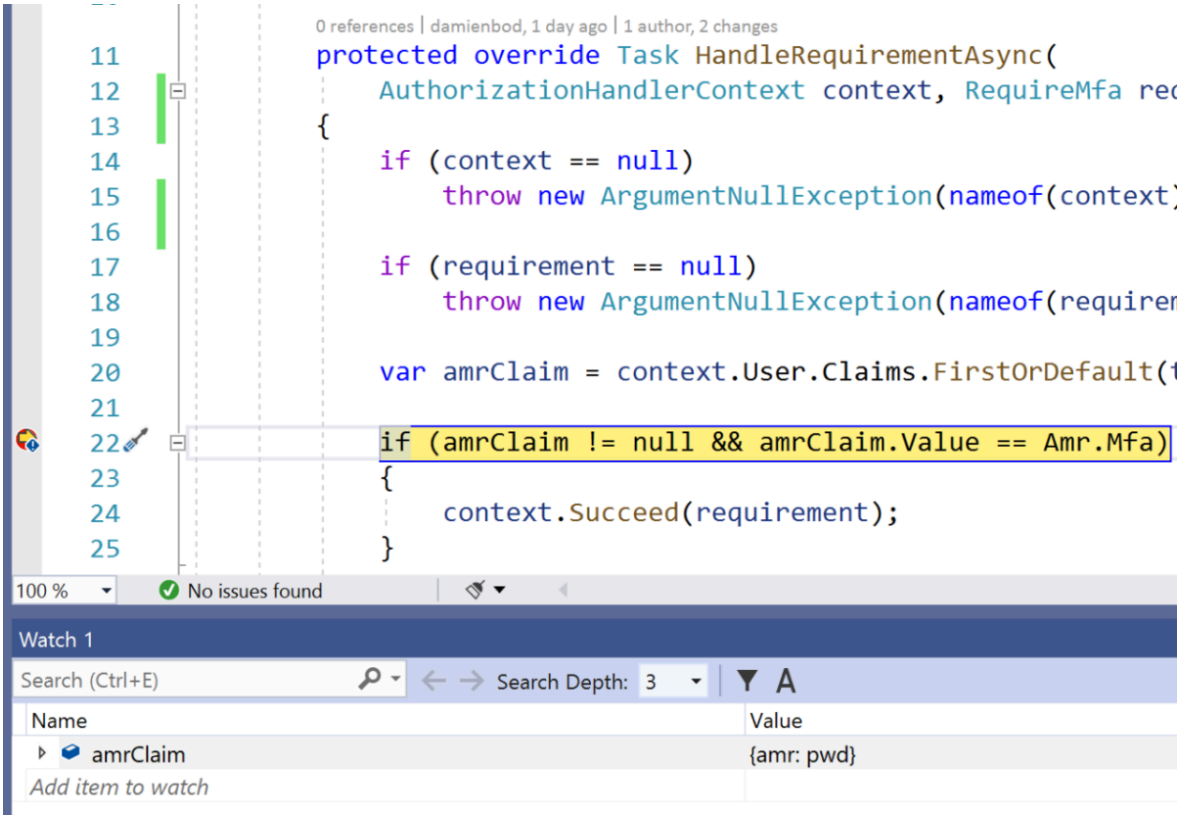
CSHTML

```
@page
@model AspNetCoreRequireMfaOidc.AccessDeniedModel
@{
    ViewData["Title"] = "AccessDenied";
    Layout = "~/Pages/Shared/_Layout.cshtml";
}

<h1>AccessDenied</h1>

You require MFA to login here

<a href="https://localhost:44352/Manage/TwoFactorAuthentication">Enable
MFA</a>
```

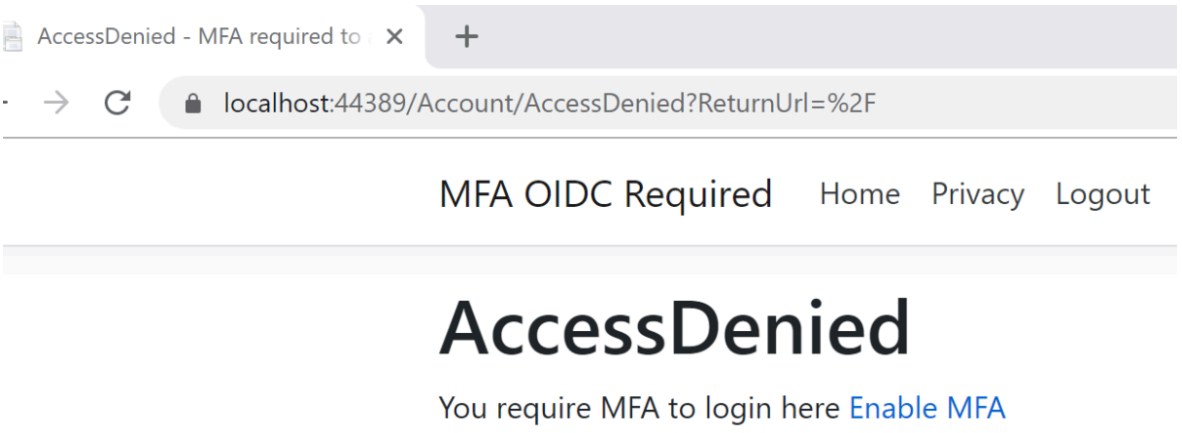
Now only users that authenticate with MFA can access the page or website. If different MFA types are used or if 2FA is okay, the `amr` claim will have different values and needs to be processed correctly. Different OpenID Connect servers also return different values for this claim and might not follow the Authentication Method Reference Values specification.

When logging in without MFA (for example, using just a password):

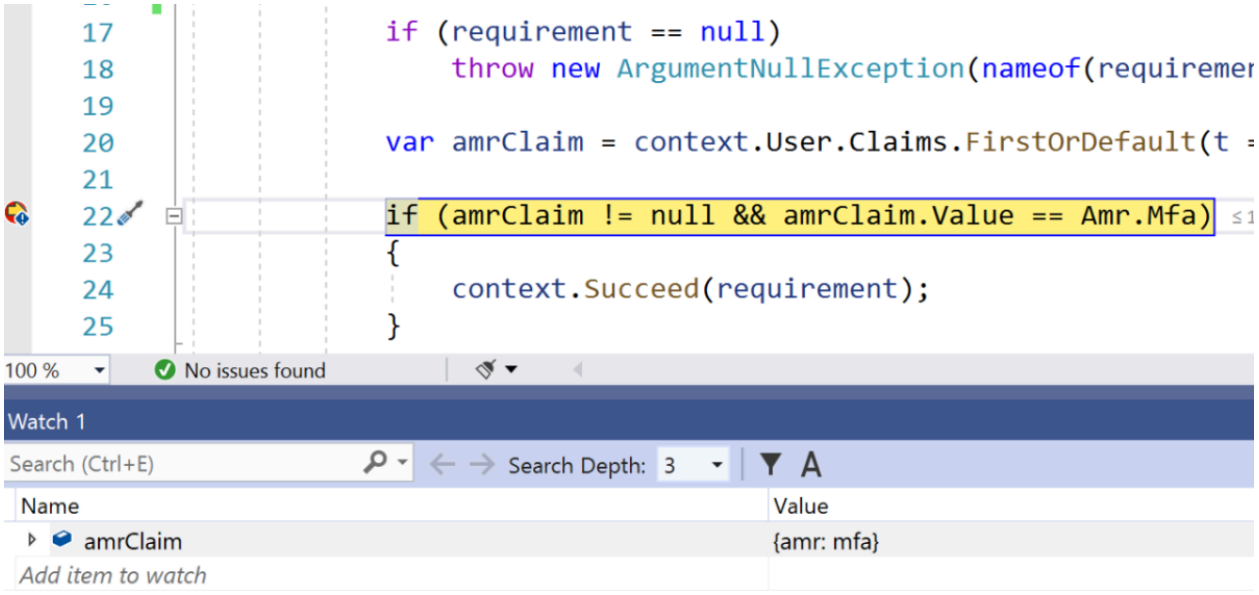
- The `amr` has the `pwd` value:



- Access is denied:



Alternatively, logging in using OTP with Identity:



OIDC and OAuth Parameter Customization

The OAuth and OIDC authentication handlers AdditionalAuthorizationParameters option allows customization of authorization message parameters that are usually included as part of the redirect query string:

C#

```
builder.Services.AddAuthentication().AddOpenIdConnect(options =>
{
    options.AdditionalAuthorizationParameters.Add("prompt", "login");
    options.AdditionalAuthorizationParameters.Add("audience",
"https://api.example.com");
});
```

Additional resources

- Enable QR Code generation for TOTP authenticator apps in ASP.NET Core
- Passwordless authentication options for Azure Active Directory
- FIDO2 .NET library for FIDO2 / WebAuthn Attestation and Assertion using .NET
- WebAuthn Awesome

Introduction to authorization in ASP.NET Core

Article • 12/02/2024

Authorization refers to the process that determines what a user is able to do. For example, an administrative user is allowed to create a document library, add documents, edit documents, and delete them. A non-administrative user working with the library is only authorized to read the documents.

Authorization is separate and distinct from authentication. However, authorization relies on an authentication mechanism. Authentication is the process of verifying a user's identity, which may result in the creation of one or more identity objects for the user.

For more information about authentication in ASP.NET Core, see Overview of ASP.NET Core Authentication.

Authorization types

ASP.NET Core authorization provides a simple, declarative role and a rich policy-based model. Authorization is expressed in requirements, and handlers evaluate a user's claims against requirements. Imperative checks can be based on simple policies or policies which evaluate both the user identity and properties of the resource that the user is attempting to access.

Namespaces

Authorization components, including the `AuthorizeAttribute` and `AllowAnonymousAttribute` attributes, are found in the `Microsoft.AspNetCore.Authorization` namespace.

Consult the documentation on simple authorization.

Create an ASP.NET Core web app with user data protected by authorization

Article • 10/04/2024

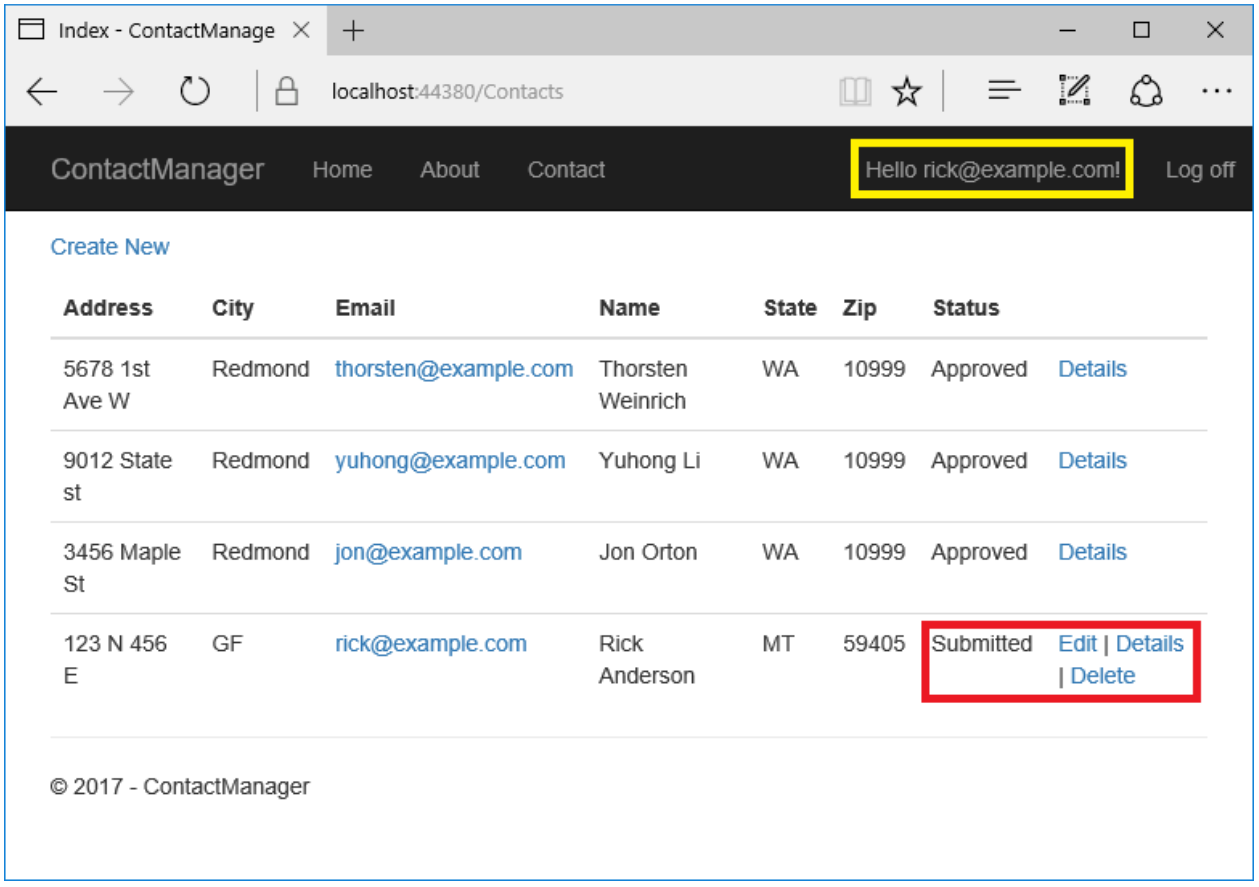
By Rick Anderson and Joe Audette

This tutorial shows how to create an ASP.NET Core web app with user data protected by authorization. It displays a list of contacts that authenticated (registered) users have created. There are three security groups:

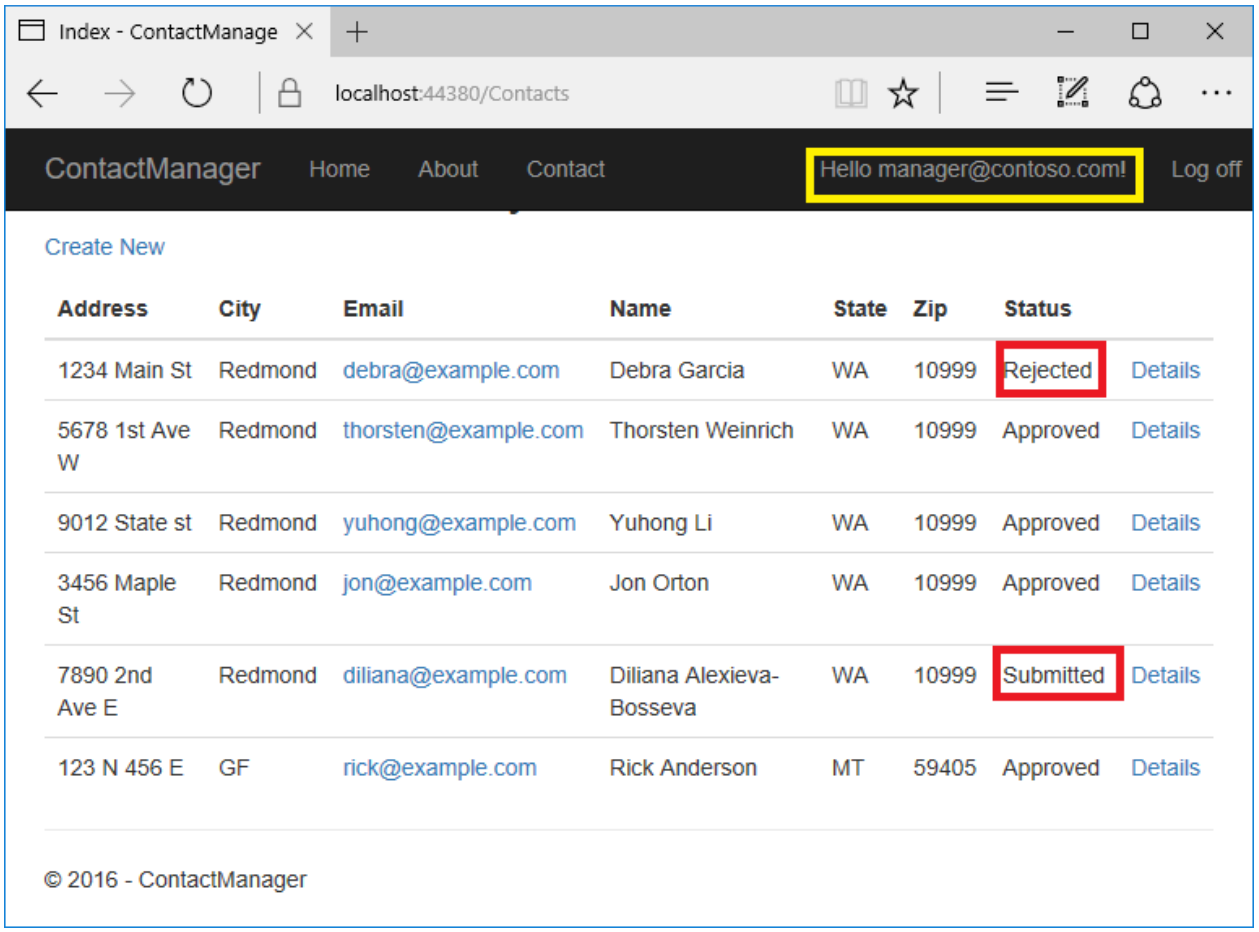
- **Registered users** can view all the approved data and can edit/delete their own data.
- **Managers** can approve or reject contact data. Only approved contacts are visible to users.
- **Administrators** can approve/reject and edit/delete any data.

The images in this document don't exactly match the latest templates.

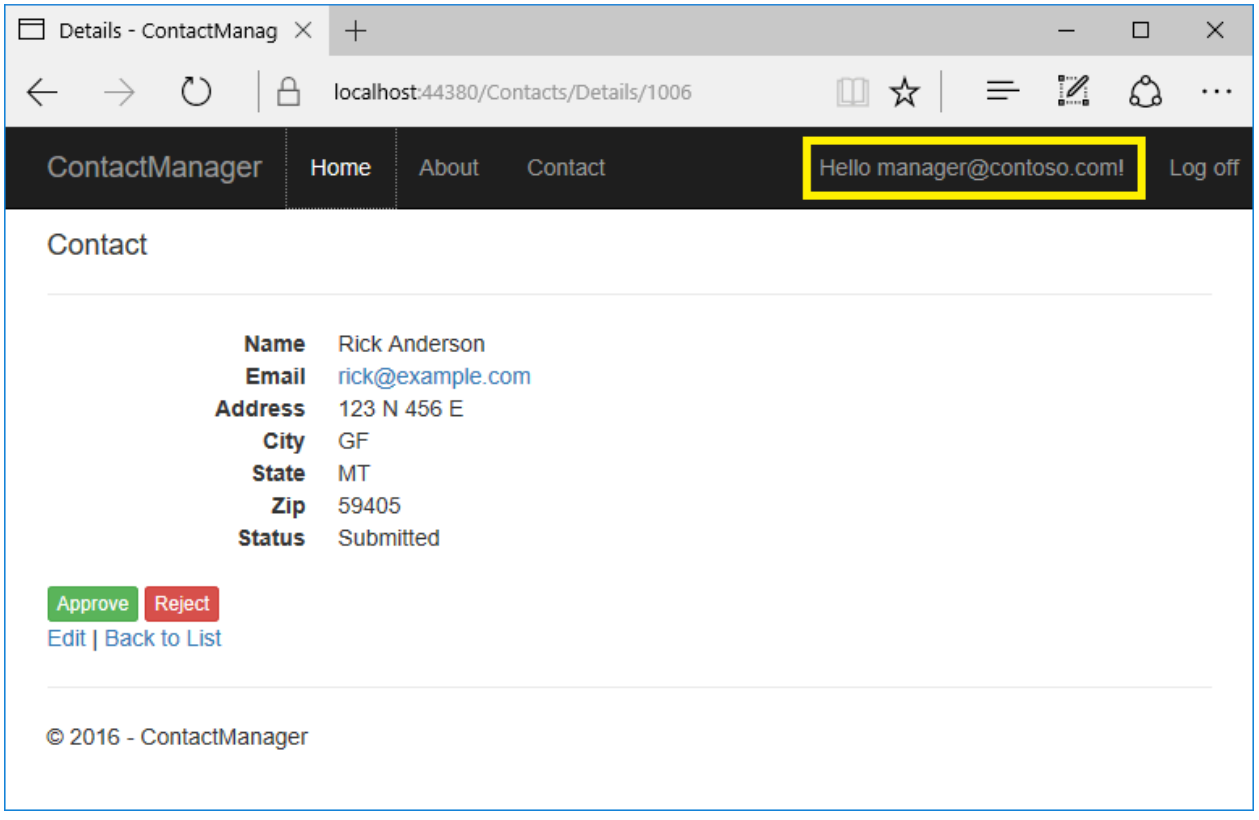
In the following image, user Rick (`rick@example.com`) is signed in. Rick can only view approved contacts and **Edit/Delete/Create New** links for his contacts. Only the last record, created by Rick, displays **Edit** and **Delete** links. Other users won't see the last record until a manager or administrator changes the status to "Approved".



In the following image, `manager@contoso.com` is signed in and in the manager's role:

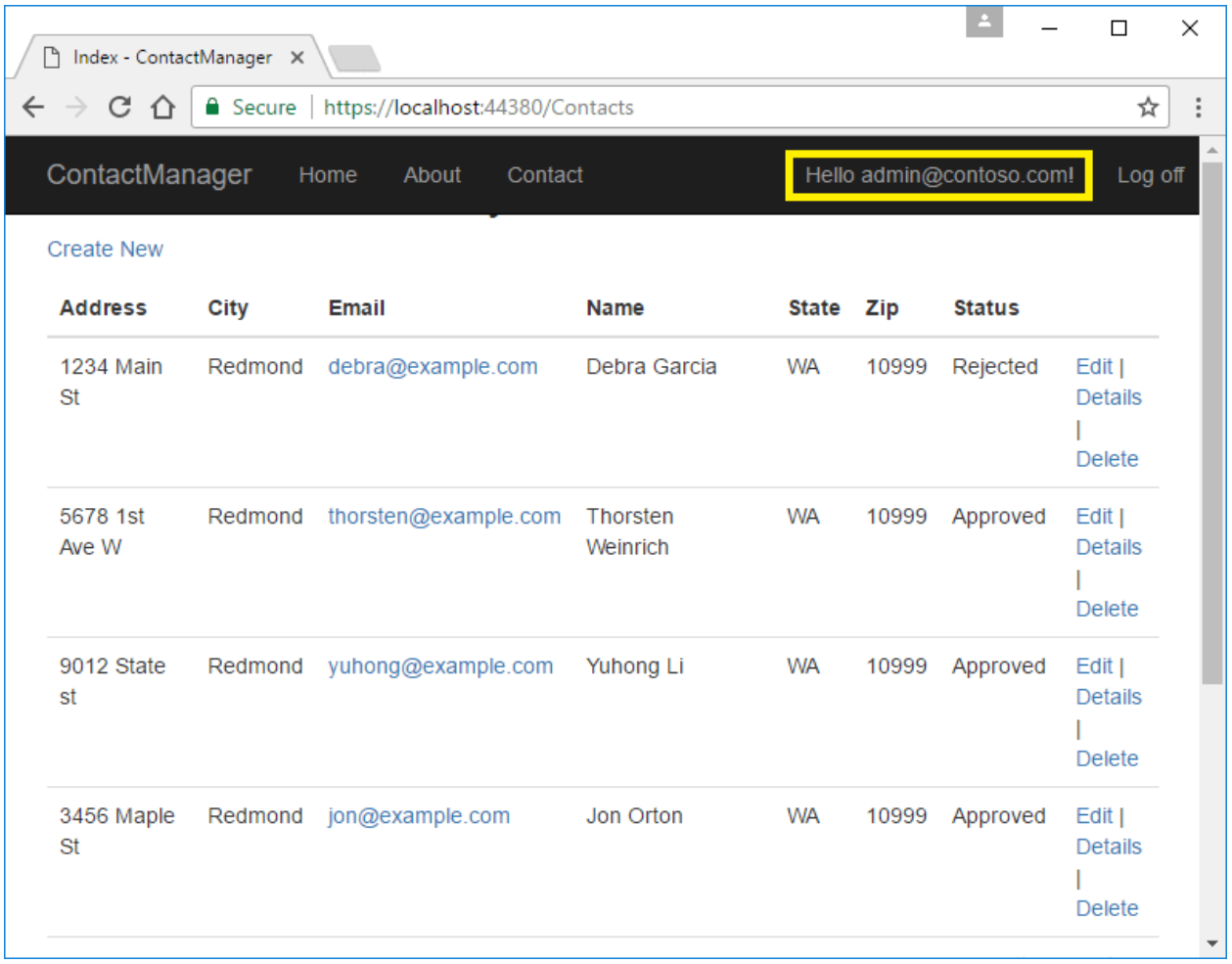


The following image shows the managers details view of a contact:



The **Approve** and **Reject** buttons are only displayed for managers and administrators.

In the following image, `admin@contoso.com` is signed in and in the administrator's role:



The administrator has all privileges. She can read, edit, or delete any contact and change the status of contacts.

The app was created by scaffolding the following `Contact` model:

C#

```
public class Contact
{
    public int ContactId { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Zip { get; set; }
    [DataType(DataType.EmailAddress)]
    public string Email { get; set; }
}
```

The sample contains the following authorization handlers:

- `ContactIsOwnerAuthorizationHandler`: Ensures that a user can only edit their data.

- `ContactManagerAuthorizationHandler`: Allows managers to approve or reject contacts.
- `ContactAdministratorsAuthorizationHandler`: Allows administrators to approve or reject contacts and to edit/delete contacts.

Prerequisites

This tutorial is advanced. You should be familiar with:

- ASP.NET Core
- Authentication
- Account Confirmation and Password Recovery
- Authorization
- Entity Framework Core

The starter and completed app

Download the completed app. Test the completed app so you become familiar with its security features.

The starter app

Download the starter app.

Run the app, tap the **ContactManager** link, and verify you can create, edit, and delete a contact. To create the starter app, see Create the starter app.

Secure user data

The following sections have all the major steps to create the secure user data app. You may find it helpful to refer to the completed project.

Tie the contact data to the user

Use the ASP.NET Identity user ID to ensure users can edit their data, but not other users data. Add `OwnerID` and `ContactStatus` to the `Contact` model:

C#

```
public class Contact
{
```

```
    public int ContactId { get; set; }

    // user ID from AspNetUser table.
    public string? OwnerID { get; set; }

    public string? Name { get; set; }
    public string? Address { get; set; }
    public string? City { get; set; }
    public string? State { get; set; }
    public string? Zip { get; set; }
    [DataType(DataType.EmailAddress)]
    public string? Email { get; set; }

    public ContactStatus Status { get; set; }
}

public enum ContactStatus
{
    Submitted,
    Approved,
    Rejected
}
```

`OwnerID` is the user's ID from the `AspNetUser` table in the Identity database. The `Status` field determines if a contact is viewable by general users.

Create a new migration and update the database:

.NET CLI

```
dotnet ef migrations add userID_Status
dotnet ef database update
```

Add Role services to Identity

Append AddRoles to add Role services:

C#

```
var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(
    options => options.SignIn.RequireConfirmedAccount = true)
```

```
    .AddRoles<IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>();
```

Require authenticated users

Set the fallback authorization policy to require users to be authenticated:

C#

```
var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(
    options => options.SignIn.RequireConfirmedAccount = true)
    .AddRoles<IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>();

builder.Services.AddRazorPages();

builder.Services.AddAuthorization(options =>
{
    options.FallbackPolicy = new AuthorizationPolicyBuilder()
        .RequireAuthenticatedUser()
        .Build();
});
```

The preceding highlighted code sets the fallback authorization policy. The fallback authorization policy requires ***all*** users to be authenticated, except for Razor Pages, controllers, or action methods with an authorization attribute. For example, Razor Pages, controllers, or action methods with `[AllowAnonymous]` or `[Authorize(PolicyName="MyPolicy")]` use the applied authorization attribute rather than the fallback authorization policy.

RequireAuthenticatedUser adds DenyAnonymousAuthorizationRequirement to the current instance, which enforces that the current user is authenticated.

The fallback authorization policy:

- Is applied to all requests that don't explicitly specify an authorization policy. For requests served by endpoint routing, this includes any endpoint that doesn't specify an authorization attribute. For requests served by other middleware after

the authorization middleware, such as static files, this applies the policy to all requests.

Setting the fallback authorization policy to require users to be authenticated protects newly added Razor Pages and controllers. Having authorization required by default is more secure than relying on new controllers and Razor Pages to include the `[Authorize]` attribute.

The AuthorizationOptions class also contains AuthorizationOptions.DefaultPolicy. The `DefaultPolicy` is the policy used with the `[Authorize]` attribute when no policy is specified. `[Authorize]` doesn't contain a named policy, unlike `[Authorize(PolicyName="MyPolicy")]`.

For more information on policies, see Policy-based authorization in ASP.NET Core.

An alternative way for MVC controllers and Razor Pages to require all users be authenticated is adding an authorization filter:

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using ContactManager.Data;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc.Authorization;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(
    options => options.SignIn.RequireConfirmedAccount = true)
    .AddRoles<IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>();

builder.Services.AddRazorPages();

builder.Services.AddControllers(config =>
{
    var policy = new AuthorizationPolicyBuilder()
                     .RequireAuthenticatedUser()
                     .Build();
    config.Filters.Add(new AuthorizeFilter(policy));
});
```

```
var app = builder.Build();
```

The preceding code uses an authorization filter, setting the fallback policy uses endpoint routing. Setting the fallback policy is the preferred way to require all users be authenticated.

Add AllowAnonymous to the `Index` and `Privacy` pages so anonymous users can get information about the site before they register:

C#

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

namespace ContactManager.Pages;

[AllowAnonymous]
public class IndexModel : PageModel
{
    private readonly ILogger<IndexModel> _logger;

    public IndexModel(ILogger<IndexModel> logger)
    {
        _logger = logger;
    }

    public void OnGet()
    {

    }
}
```

Configure the test account

The `SeedData` class creates two accounts: administrator and manager. Use the Secret Manager tool to set a password for these accounts. Set the password from the project directory (the directory containing `Program.cs`):

.NET CLI

```
dotnet user-secrets set SeedUserPW <PW>
```

If a weak password is specified, an exception is thrown when `SeedData.Initialize` is called.

Update the app to use the test password:

C#

```
var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(
    options => options.SignIn.RequireConfirmedAccount = true)
    .AddRoles<IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>();

builder.Services.AddRazorPages();

builder.Services.AddAuthorization(options =>
{
    options.FallbackPolicy = new AuthorizationPolicyBuilder()
        .RequireAuthenticatedUser()
        .Build();
});

// Authorization handlers.
builder.Services.AddScoped<IAuthorizationHandler,
                      ContactIsOwnerAuthorizationHandler>();

builder.Services.AddSingleton<IAuthorizationHandler,
                      ContactAdministratorsAuthorizationHandler>();

builder.Services.AddSingleton<IAuthorizationHandler,
                      ContactManagerAuthorizationHandler>();

var app = builder.Build();

using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;
    var context = services.GetRequiredService<ApplicationDbContext>();
    context.Database.Migrate();
    // requires using Microsoft.Extensions.Configuration;
    // Set password with the Secret Manager tool.
    // dotnet user-secrets set SeedUserPW <pw>

    var testUserPw = builder.Configuration.GetValue<string>("SeedUserPW");

   await SeedData.Initialize(services, testUserPw);
}
```

Create the test accounts and update the contacts

Update the `Initialize` method in the `SeedData` class to create the test accounts:

C#

```
public static async Task Initialize(IServiceProvider serviceProvider, string
testUserPw)
{
    using (var context = new ApplicationDbContext(

serviceProvider.GetRequiredService<DbContextOptions<ApplicationDbContext>>
()))
    {
        // For sample purposes seed both with the same password.
        // Password is set with the following:
        // dotnet user-secrets set SeedUserPW <pw>
        // The admin user can do anything

        var adminID = await EnsureUser(serviceProvider, testUserPw,
"admin@contoso.com");
        await EnsureRole(serviceProvider, adminID,
Constants.ContactAdministratorsRole);

        // allowed user can create and edit contacts that they create
        var managerID = await EnsureUser(serviceProvider, testUserPw,
"manager@contoso.com");
        await EnsureRole(serviceProvider, managerID,
Constants.ContactManagersRole);

        SeedDB(context, adminID);
    }
}

private static async Task<string> EnsureUser(IServiceProvider
serviceProvider,
                                            string testUserPw, string
UserName)
{
    var userManager = serviceProvider.GetService<UserManager<IdentityUser>>
();

    var user = await userManager.FindByNameAsync(UserName);
    if (user == null)
    {
        user = new IdentityUser
        {
            UserName = UserName,
            EmailConfirmed = true
        };
        await userManager.CreateAsync(user, testUserPw);
    }
```

```
    if (user == null)
    {
        throw new Exception("The password is probably not strong enough!");
    }

    return user.Id;
}

private static async Task<IdentityResult> EnsureRole(IServiceProvider
serviceProvider,
                                                     string uid,
string role)
{
    var roleManager = serviceProvider.GetService<RoleManager<IdentityRole>>
();

    if (roleManager == null)
    {
        throw new Exception("roleManager null");
    }

    IdentityResult IR;
    if (!await roleManager.RoleExistsAsync(role))
    {
        IR = await roleManager.CreateAsync(new IdentityRole(role));
    }

    var userManager = serviceProvider.GetService<UserManager<IdentityUser>>
();

    //if (userManager == null)
    //{
    //    throw new Exception("userManager is null");
    //}

    var user = await userManager.FindByIdAsync(uid);

    if (user == null)
    {
        throw new Exception("The testUserPw password was probably not strong
enough!");
    }

    IR = await userManager.AddToRoleAsync(user, role);

    return IR;
}
```

Add the administrator user ID and `ContactStatus` to the contacts. Make one of the contacts "Submitted" and one "Rejected". Add the user ID and status to all the contacts. Only one contact is shown:

```
C#
```

```
public static void SeedDB(ApplicationDbContext context, string adminID)
{
    if (context.Contact.Any())
    {
        return;   // DB has been seeded
    }

    context.Contact.AddRange(
        new Contact
        {
            Name = "Debra Garcia",
            Address = "1234 Main St",
            City = "Redmond",
            State = "WA",
            Zip = "10999",
            Email = "debra@example.com",
            Status = ContactStatus.Approved,
            OwnerID = adminID
        },
```

Create owner, manager, and administrator authorization handlers

Create a `ContactIsOwnerAuthorizationHandler` class in the *Authorization* folder. The `ContactIsOwnerAuthorizationHandler` verifies that the user acting on a resource owns the resource.

C#

```
using ContactManager.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Authorization.Infrastructure;
using Microsoft.AspNetCore.Identity;
using System.Threading.Tasks;

namespace ContactManager.Authorization
{
    public class ContactIsOwnerAuthorizationHandler
                  : AuthorizationHandler<OperationAuthorizationRequirement,
Contact>
    {
        UserManager<IdentityUser> _userManager;

        public ContactIsOwnerAuthorizationHandler(UserManager<IdentityUser>
            userManager)
        {
            _userManager = userManager;
        }
```

```
        protected override Task
            HandleRequirementAsync(AuthorizationHandlerContext context,
                                   OperationAuthorizationRequirement
requirement,
                                   Contact resource)
        {
            if (context.User == null || resource == null)
            {
                return Task.CompletedTask;
            }

            // If not asking for CRUD permission, return.

            if (requirement.Name != Constants.CreateOperationName &&
                requirement.Name != Constants.ReadOperationName   &&
                requirement.Name != Constants.UpdateOperationName &&
                requirement.Name != Constants.DeleteOperationName )
            {
                return Task.CompletedTask;
            }

            if (resource.OwnerID == _userManager.GetUserId(context.User))
            {
                context.Succeed(requirement);
            }

            return Task.CompletedTask;
        }
    }
}
```

The `ContactIsOwnerAuthorizationHandler` calls context.Succeed if the current authenticated user is the contact owner. Authorization handlers generally:

- Call `context.Succeed` when the requirements are met.
- Return `Task.CompletedTask` when requirements aren't met. Returning `Task.CompletedTask` without a prior call to `context.Success` or `context.Fail`, is not a success or failure, it allows other authorization handlers to run.

If you need to explicitly fail, call context.Fail.

The app allows contact owners to edit/delete/create their own data. `ContactIsOwnerAuthorizationHandler` doesn't need to check the operation passed in the requirement parameter.

Create a manager authorization handler

Create a `ContactManagerAuthorizationHandler` class in the *Authorization* folder. The `ContactManagerAuthorizationHandler` verifies the user acting on the resource is a manager. Only managers can approve or reject content changes (new or changed).

C#

```
using System.Threading.Tasks;
using ContactManager.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Authorization.Infrastructure;
using Microsoft.AspNetCore.Identity;

namespace ContactManager.Authorization
{
    public class ContactManagerAuthorizationHandler :
        AuthorizationHandler<OperationAuthorizationRequirement, Contact>
    {
        protected override Task
            HandleRequirementAsync(AuthorizationHandlerContext context,
                                   OperationAuthorizationRequirement
requirement,
                                   Contact resource)
        {
            if (context.User == null || resource == null)
            {
                return Task.CompletedTask;
            }

            // If not asking for approval/reject, return.
            if (requirement.Name != Constants.ApproveOperationName &&
                requirement.Name != Constants.RejectOperationName)
            {
                return Task.CompletedTask;
            }

            // Managers can approve or reject.
            if (context.User.IsInRole(Constants.ContactManagersRole))
            {
                context.Succeed(requirement);
            }

            return Task.CompletedTask;
        }
    }
}
```

Create an administrator authorization handler

Create a `ContactAdministratorsAuthorizationHandler` class in the *Authorization* folder. The `ContactAdministratorsAuthorizationHandler` verifies the user acting on the resource

is an administrator. Administrator can do all operations.

C#

```
using System.Threading.Tasks;
using ContactManager.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Authorization.Infrastructure;

namespace ContactManager.Authorization
{
    public class ContactAdministratorsAuthorizationHandler
                      :
AuthorizationHandler<OperationAuthorizationRequirement, Contact>
    {
        protected override Task HandleRequirementAsync(
                                                AuthorizationHandlerContext
context,
                                       OperationAuthorizationRequirement
requirement,
                                        Contact resource)
        {
            if (context.User == null)
            {
                return Task.CompletedTask;
            }

            // Administrators can do anything.
            if (context.User.IsInRole(Constants.ContactAdministratorsRole))
            {
                context.Succeed(requirement);
            }

            return Task.CompletedTask;
        }
    }
}
```

Register the authorization handlers

Services using Entity Framework Core must be registered for dependency injection using AddScoped. The `ContactIsOwnerAuthorizationHandler` uses ASP.NET Core Identity, which is built on Entity Framework Core. Register the handlers with the service collection so they're available to the `ContactsController` through dependency injection. Add the following code to the end of `ConfigureServices`:

C#

```
var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(
    options => options.SignIn.RequireConfirmedAccount = true)
    .AddRoles<IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>();

builder.Services.AddRazorPages();

builder.Services.AddAuthorization(options =>
{
    options.FallbackPolicy = new AuthorizationPolicyBuilder()
        .RequireAuthenticatedUser()
        .Build();
});

// Authorization handlers.
builder.Services.AddScoped<IAuthorizationHandler,
                      ContactIsOwnerAuthorizationHandler>();

builder.Services.AddSingleton<IAuthorizationHandler,
                      ContactAdministratorsAuthorizationHandler>();

builder.Services.AddSingleton<IAuthorizationHandler,
                      ContactManagerAuthorizationHandler>();

var app = builder.Build();

using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;
    var context = services.GetRequiredService<ApplicationDbContext>();
    context.Database.Migrate();
    // requires using Microsoft.Extensions.Configuration;
    // Set password with the Secret Manager tool.
    // dotnet user-secrets set SeedUserPW <pw>

    var testUserPw = builder.Configuration.GetValue<string>("SeedUserPW");

   await SeedData.Initialize(services, testUserPw);
}
```

`ContactAdministratorsAuthorizationHandler` and `ContactManagerAuthorizationHandler` are added as singletons. They're singletons because they don't use EF and all the information needed is in the `Context` parameter of the `HandleRequirementAsync` method.

Support authorization

In this section, you update the Razor Pages and add an operations requirements class.

Review the contact operations requirements class

Review the `ContactOperations` class. This class contains the requirements the app supports:

C#

```
using Microsoft.AspNetCore.Authorization.Infrastructure;

namespace ContactManager.Authorization
{
    public static class ContactOperations
    {
        public static OperationAuthorizationRequirement Create =
          new OperationAuthorizationRequirement
{Name=Constants.CreateOperationName};
        public static OperationAuthorizationRequirement Read =
          new OperationAuthorizationRequirement
{Name=Constants.ReadOperationName};
        public static OperationAuthorizationRequirement Update =
          new OperationAuthorizationRequirement
{Name=Constants.UpdateOperationName};
        public static OperationAuthorizationRequirement Delete =
          new OperationAuthorizationRequirement
{Name=Constants.DeleteOperationName};
        public static OperationAuthorizationRequirement Approve =
          new OperationAuthorizationRequirement
{Name=Constants.ApproveOperationName};
        public static OperationAuthorizationRequirement Reject =
          new OperationAuthorizationRequirement
{Name=Constants.RejectOperationName};
    }

    public class Constants
    {
        public static readonly string CreateOperationName = "Create";
        public static readonly string ReadOperationName = "Read";
        public static readonly string UpdateOperationName = "Update";
        public static readonly string DeleteOperationName = "Delete";
        public static readonly string ApproveOperationName = "Approve";
        public static readonly string RejectOperationName = "Reject";

        public static readonly string ContactAdministratorsRole =

"ContactAdministrators";
        public static readonly string ContactManagersRole =
"ContactManagers";
```

```
    }
}
```

Create a base class for the Contacts Razor Pages

Create a base class that contains the services used in the contacts Razor Pages. The base class puts the initialization code in one location:

C#

```
using ContactManager.Data;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace ContactManager.Pages.Contacts
{
    public class DI_BasePageModel : PageModel
    {
        protected ApplicationDbContext Context { get; }
        protected IAuthorizationService AuthorizationService { get; }
        protected UserManager<IdentityUser> UserManager { get; }

        public DI_BasePageModel(
            ApplicationDbContext context,
            IAuthorizationService authorizationService,
            UserManager<IdentityUser> userManager) : base()
        {
            Context = context;
            UserManager = userManager;
            AuthorizationService = authorizationService;
        }
    }
}
```

The preceding code:

- Adds the `IAuthorizationService` service to access to the authorization handlers.
- Adds the Identity `UserManager` service.
- Add the `ApplicationDbContext`.

Update the CreateModel

Update the create page model:

- Constructor to use the `DI_BasePageModel` base class.
- `OnPostAsync` method to:

- Add the user ID to the `Contact` model.
- Call the authorization handler to verify the user has permission to create contacts.

C#

```
using ContactManager.Authorization;
using ContactManager.Data;
using ContactManager.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;

namespace ContactManager.Pages.Contacts
{
    public class CreateModel : DI_BasePageModel
    {
        public CreateModel(
            ApplicationDbContext context,
            IAuthorizationService authorizationService,
            UserManager<IdentityUser> userManager)
            : base(context, authorizationService, userManager)
        {
        }

        public IActionResult OnGet()
        {
            return Page();
        }

        [BindProperty]
        public Contact Contact { get; set; }

        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            Contact.OwnerID = UserManager.GetUserId(User);

            var isAuthorized = await AuthorizationService.AuthorizeAsync(
                                                        User, Contact,

ContactOperations.Create);
            if (!isAuthorized.Succeeded)
            {
                return Forbid();
            }

            Context.Contact.Add(Contact);
            await Context.SaveChangesAsync();
```

```
            return RedirectToPage("./Index");
        }
    }
}
```

Update the IndexModel

Update the `OnGetAsync` method so only approved contacts are shown to general users:

C#

```
public class IndexModel : DI_BasePageModel
{
    public IndexModel(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<IdentityUser> userManager)
        : base(context, authorizationService, userManager)
    {
    }

    public IList<Contact> Contact { get; set; }

    public async Task OnGetAsync()
    {
        var contacts = from c in Context.Contact
                       select c;

        var isAuthorized = User.IsInRole(Constants.ContactManagersRole) ||

User.IsInRole(Constants.ContactAdministratorsRole);

        var currentUserId = UserManager.GetUserId(User);

        // Only approved contacts are shown UNLESS you're authorized to see
them
        // or you are the owner.
        if (!isAuthorized)
        {
            contacts = contacts.Where(c => c.Status ==
ContactStatus.Approved
                                        || c.OwnerID == currentUserId);
        }

        Contact = await contacts.ToListAsync();
    }
}
```

Update the EditModel

Add an authorization handler to verify the user owns the contact. Because resource authorization is being validated, the `[Authorize]` attribute is not enough. The app doesn't have access to the resource when attributes are evaluated. Resource-based authorization must be imperative. Checks must be performed once the app has access to the resource, either by loading it in the page model or by loading it within the handler itself. You frequently access the resource by passing in the resource key.

C#

```
public class EditModel : DI_BasePageModel
{
    public EditModel(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<IdentityUser> userManager)
        : base(context, authorizationService, userManager)
    {
    }

    [BindProperty]
    public Contact Contact { get; set; }

    public async Task<IActionResult> OnGetAsync(int id)
    {
        Contact? contact = await Context.Contact.FirstOrDefaultAsync(
                                                          m => m.ContactId ==
id);
        if (contact == null)
        {
            return NotFound();
        }

        Contact = contact;

        var isAuthorized = await AuthorizationService.AuthorizeAsync(
                                                  User, Contact,
                                                  ContactOperations.Update);
        if (!isAuthorized.Succeeded)
        {
            return Forbid();
        }

        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int id)
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }
```

```
        // Fetch Contact from DB to get OwnerID.
        var contact = await Context
            .Contact.AsNoTracking()
            .FirstOrDefaultAsync(m => m.ContactId == id);

        if (contact == null)
        {
            return NotFound();
        }

        var isAuthorized = await AuthorizationService.AuthorizeAsync(
                                                 User, contact,
                                                 ContactOperations.Update);
        if (!isAuthorized.Succeeded)
        {
            return Forbid();
        }

        Contact.OwnerID = contact.OwnerID;

        Context.Attach(Contact).State = EntityState.Modified;

        if (Contact.Status == ContactStatus.Approved)
        {
            // If the contact is updated after approval,
            // and the user cannot approve,
            // set the status back to submitted so the update can be
            // checked and approved.
            var canApprove = await AuthorizationService.AuthorizeAsync(User,
                                     Contact,
                                     ContactOperations.Approve);

            if (!canApprove.Succeeded)
            {
                Contact.Status = ContactStatus.Submitted;
            }
        }

        await Context.SaveChangesAsync();

        return RedirectToPage("./Index");
    }
}
```

Update the DeleteModel

Update the delete page model to use the authorization handler to verify the user has delete permission on the contact.

```
C#
```

```
public class DeleteModel : DI_BasePageModel
{
    public DeleteModel(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<IdentityUser> userManager)
        : base(context, authorizationService, userManager)
    {
    }

    [BindProperty]
    public Contact Contact { get; set; }

    public async Task<IActionResult> OnGetAsync(int id)
    {
        Contact? _contact = await Context.Contact.FirstOrDefaultAsync(
                                            m => m.ContactId == id);

        if (_contact == null)
        {
            return NotFound();
        }
        Contact = _contact;

        var isAuthorized = await AuthorizationService.AuthorizeAsync(
                                                User, Contact,
                                                ContactOperations.Delete);
        if (!isAuthorized.Succeeded)
        {
            return Forbid();
        }

        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int id)
    {
        var contact = await Context
            .Contact.AsNoTracking()
            .FirstOrDefaultAsync(m => m.ContactId == id);

        if (contact == null)
        {
            return NotFound();
        }

        var isAuthorized = await AuthorizationService.AuthorizeAsync(
                                                User, contact,
                                                ContactOperations.Delete);
        if (!isAuthorized.Succeeded)
        {
            return Forbid();
        }
```

```
            Context.Contact.Remove(contact);
            await Context.SaveChangesAsync();

            return RedirectToPage("./Index");
        }
    }
```

Inject the authorization service into the views

Currently, the UI shows edit and delete links for contacts the user can't modify.

Inject the authorization service in the `Pages/_ViewImports.cshtml` file so it's available to all views:

CSHTML

```
@using Microsoft.AspNetCore.Identity
@using ContactManager
@using ContactManager.Data
@namespace ContactManager.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@using ContactManager.Authorization;
@using Microsoft.AspNetCore.Authorization
@using ContactManager.Models
@inject IAuthorizationService AuthorizationService
```

The preceding markup adds several `using` statements.

Update the **Edit** and **Delete** links in `Pages/Contacts/Index.cshtml` so they're only rendered for users with the appropriate permissions:

CSHTML

```
@page
@model ContactManager.Pages.Contacts.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
```

```
                @Html.DisplayNameFor(model => model.Contact[0].Name)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].Address)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].City)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].State)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].Zip)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].Email)
            </th>
             <th>
                @Html.DisplayNameFor(model => model.Contact[0].Status)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model.Contact) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Name)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Address)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.City)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.State)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Zip)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Email)
            </td>
                        <td>
```

```
                    @Html.DisplayFor(modelItem => item.Status)
                </td>
                <td>
                    @if ((await AuthorizationService.AuthorizeAsync(
                     User, item,
                     ContactOperations.Update)).Succeeded)
                    {
                        <a asp-page="./Edit" asp-route-
id="@item.ContactId">Edit</a>
                        <text> | </text>
                    }

                    <a asp-page="./Details" asp-route-
id="@item.ContactId">Details</a>

                    @if ((await AuthorizationService.AuthorizeAsync(
                     User, item,
                     ContactOperations.Delete)).Succeeded)
                    {
                        <text> | </text>
                        <a asp-page="./Delete" asp-route-
id="@item.ContactId">Delete</a>
                    }
                </td>
            </tr>
        }
    </tbody>
</table>
```

> ⚠ **Warning**
>
> Hiding links from users that don't have permission to change data doesn't secure the app. Hiding links makes the app more user-friendly by displaying only valid links. Users can hack the generated URLs to invoke edit and delete operations on data they don't own. The Razor Page or controller must enforce access checks to secure the data.

Update Details

Update the details view so managers can approve or reject contacts:

CSHTML

```
        @*Preceding markup omitted for brevity.*@
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Contact.Email)
        </dt>
```

```
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Contact.Email)
        </dd>
    <dt>
            @Html.DisplayNameFor(model => model.Contact.Status)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Contact.Status)
        </dd>
    </dl>
</div>

@if (Model.Contact.Status != ContactStatus.Approved)
{
    @if ((await AuthorizationService.AuthorizeAsync(
     User, Model.Contact, ContactOperations.Approve)).Succeeded)
    {
        <form style="display:inline;" method="post">
            <input type="hidden" name="id" value="@Model.Contact.ContactId"
/>
            <input type="hidden" name="status"
value="@ContactStatus.Approved" />
            <button type="submit" class="btn btn-xs btn-
success">Approve</button>
        </form>
    }
}

@if (Model.Contact.Status != ContactStatus.Rejected)
{
    @if ((await AuthorizationService.AuthorizeAsync(
     User, Model.Contact, ContactOperations.Reject)).Succeeded)
    {
        <form style="display:inline;" method="post">
            <input type="hidden" name="id" value="@Model.Contact.ContactId"
/>
            <input type="hidden" name="status"
value="@ContactStatus.Rejected" />
            <button type="submit" class="btn btn-xs btn-
danger">Reject</button>
        </form>
    }
}

<div>
    @if ((await AuthorizationService.AuthorizeAsync(
         User, Model.Contact,
         ContactOperations.Update)).Succeeded)
    {
        <a asp-page="./Edit" asp-route-
id="@Model.Contact.ContactId">Edit</a>
        <text> | </text>
    }
    <a asp-page="./Index">Back to List</a>
</div>
```

Update the details page model

C#

```
public class DetailsModel : DI_BasePageModel
{
    public DetailsModel(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<IdentityUser> userManager)
        : base(context, authorizationService, userManager)
    {
    }

    public Contact Contact { get; set; }

    public async Task<IActionResult> OnGetAsync(int id)
    {
        Contact? _contact = await Context.Contact.FirstOrDefaultAsync(m =>
m.ContactId == id);

        if (_contact == null)
        {
            return NotFound();
        }
        Contact = _contact;

        var isAuthorized = User.IsInRole(Constants.ContactManagersRole) ||

User.IsInRole(Constants.ContactAdministratorsRole);

        var currentUserId = UserManager.GetUserId(User);

        if (!isAuthorized
            && currentUserId != Contact.OwnerID
            && Contact.Status != ContactStatus.Approved)
        {
            return Forbid();
        }

        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int id, ContactStatus
status)
    {
        var contact = await Context.Contact.FirstOrDefaultAsync(
                                                  m => m.ContactId == id);

        if (contact == null)
        {
```

```
            return NotFound();
        }

        var contactOperation = (status == ContactStatus.Approved)
                                                           ?
ContactOperations.Approve
                                                           :
ContactOperations.Reject;

        var isAuthorized = await AuthorizationService.AuthorizeAsync(User,
contact,
                                         contactOperation);
        if (!isAuthorized.Succeeded)
        {
            return Forbid();
        }
        contact.Status = status;
        Context.Contact.Update(contact);
        await Context.SaveChangesAsync();

        return RedirectToPage("./Index");
    }
}
```

Add or remove a user to a role

See this issue for information on:

- Removing privileges from a user. For example, muting a user in a chat app.
- Adding privileges to a user.

Differences between Challenge and Forbid

This app sets the default policy to require authenticated users. The following code allows anonymous users. Anonymous users are allowed to show the differences between Challenge vs Forbid.

C#

```
[AllowAnonymous]
public class Details2Model : DI_BasePageModel
{
    public Details2Model(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<IdentityUser> userManager)
        : base(context, authorizationService, userManager)
    {
```

```
    }

    public Contact Contact { get; set; }

    public async Task<IActionResult> OnGetAsync(int id)
    {
        Contact? _contact = await Context.Contact.FirstOrDefaultAsync(m =>
m.ContactId == id);

        if (_contact == null)
        {
            return NotFound();
        }
        Contact = _contact;

        if (!User.Identity!.IsAuthenticated)
        {
            return Challenge();
        }

        var isAuthorized = User.IsInRole(Constants.ContactManagersRole) ||

User.IsInRole(Constants.ContactAdministratorsRole);

        var currentUserId = UserManager.GetUserId(User);

        if (!isAuthorized
            && currentUserId != Contact.OwnerID
            && Contact.Status != ContactStatus.Approved)
        {
            return Forbid();
        }

        return Page();
    }
}
```

In the preceding code:

- When the user is **not** authenticated, a `ChallengeResult` is returned. When a `ChallengeResult` is returned, the user is redirected to the sign-in page.
- When the user is authenticated, but not authorized, a `ForbidResult` is returned. When a `ForbidResult` is returned, the user is redirected to the access denied page.

Test the completed app

> ⚠ **Warning**

This article uses the Secret Manager tool to store the password for the seeded user accounts. The Secret Manager tool is used to store sensitive data during local development. For information on authentication procedures that can be used when an app is deployed to a test or production environment, see Secure authentication flows.

If you haven't already set a password for seeded user accounts, use the Secret Manager tool to set a password:

- Choose a strong password:
  - At least 12 characters long but 14 or more is better.
  - A combination of uppercase letters, lowercase letters, numbers, and symbols.
  - Not a word that can be found in a dictionary or the name of a person, character, product, or organization.
  - Significantly different from your previous passwords.
  - Easy for you to remember but difficult for others to guess. Consider using a memorable phrase like "6MonkeysRLooking^".
- Execute the following command from the project's folder, where `<PW>` is the password:

  .NET CLI

  ```
  dotnet user-secrets set SeedUserPW <PW>
  ```

If the app has contacts:

- Delete all of the records in the `Contact` table.
- Restart the app to seed the database.

An easy way to test the completed app is to launch three different browsers (or incognito/InPrivate sessions). In one browser, register a new user (for example, `test@example.com`). Sign in to each browser with a different user. Verify the following operations:

- Registered users can view all of the approved contact data.
- Registered users can edit/delete their own data.
- Managers can approve/reject contact data. The `Details` view shows **Approve** and **Reject** buttons.
- Administrators can approve/reject and edit/delete all data.

Expand table

User	Approve or reject contacts	Options
test@example.com	No	Edit and delete their data.
manager@contoso.com	Yes	Edit and delete their data.
admin@contoso.com	Yes	Edit and delete ***all*** data.

Create a contact in the administrator's browser. Copy the URL for delete and edit from the administrator contact. Paste these links into the test user's browser to verify the test user can't perform these operations.

Create the starter app

- Create a Razor Pages app named "ContactManager"
  - Create the app with **Individual User Accounts**.
  - Name it "ContactManager" so the namespace matches the namespace used in the sample.
  - `-uld` specifies LocalDB instead of SQLite

.NET CLI

```
dotnet new webapp -o ContactManager -au Individual -uld
```

- Add `Models/Contact.cs`: secure-data\samples\starter6\ContactManager\Models\Contact.cs

C#

```
using System.ComponentModel.DataAnnotations;

namespace ContactManager.Models
{
    public class Contact
    {
        public int ContactId { get; set; }
        public string? Name { get; set; }
        public string? Address { get; set; }
        public string? City { get; set; }
        public string? State { get; set; }
        public string? Zip { get; set; }
        [DataType(DataType.EmailAddress)]
        public string? Email { get; set; }
    }
}
```

- Scaffold the `Contact` model.
- Create initial migration and update the database:

.NET CLI

```
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
dotnet tool install -g dotnet-aspnet-codegenerator
dotnet-aspnet-codegenerator razorpage -m Contact -udl -dc
ApplicationDbContext -outDir Pages\Contacts --referenceScriptLibraries
dotnet ef database drop -f
dotnet ef migrations add initial
dotnet ef database update
```

> **Note**
>
> By default the architecture of the .NET binaries to install represents the currently running OS architecture. To specify a different OS architecture, see dotnet tool install, --arch option. For more information, see GitHub issue dotnet/AspNetCore.Docs #29262.

- Update the **ContactManager** anchor in the `Pages/Shared/_Layout.cshtml` file:

CSHTML

```
<a class="nav-link text-dark" asp-area="" asp-
page="/Contacts/Index">Contact Manager</a>
```

- Test the app by creating, editing, and deleting a contact

Seed the database

Add the SeedData class to the *Data* folder:

C#

```
using ContactManager.Models;
using Microsoft.EntityFrameworkCore;

// dotnet aspnet-codegenerator razorpage -m Contact -dc ApplicationDbContext
-udl -outDir Pages\Contacts --referenceScriptLibraries

namespace ContactManager.Data
{
    public static class SeedData
    {
```

```
        public static async Task Initialize(IServiceProvider
serviceProvider, string testUserPw="")
        {
            using (var context = new ApplicationDbContext(

serviceProvider.GetRequiredService<DbContextOptions<ApplicationDbContext>>
()))
            {
                SeedDB(context, testUserPw);
            }
        }

        public static void SeedDB(ApplicationDbContext context, string
adminID)
        {
            if (context.Contact.Any())
            {
                return;   // DB has been seeded
            }

            context.Contact.AddRange(
                new Contact
                {
                    Name = "Debra Garcia",
                    Address = "1234 Main St",
                    City = "Redmond",
                    State = "WA",
                    Zip = "10999",
                    Email = "debra@example.com"
                },
                new Contact
                {
                    Name = "Thorsten Weinrich",
                    Address = "5678 1st Ave W",
                    City = "Redmond",
                    State = "WA",
                    Zip = "10999",
                    Email = "thorsten@example.com"
                },
                new Contact
                {
                    Name = "Yuhong Li",
                    Address = "9012 State st",
                    City = "Redmond",
                    State = "WA",
                    Zip = "10999",
                    Email = "yuhong@example.com"
                },
                new Contact
                {
                    Name = "Jon Orton",
                    Address = "3456 Maple St",
                    City = "Redmond",
                    State = "WA",
                    Zip = "10999",
```

```
                    Email = "jon@example.com"
                },
                new Contact
                {
                    Name = "Diliana Alexieva-Bosseva",
                    Address = "7890 2nd Ave E",
                    City = "Redmond",
                    State = "WA",
                    Zip = "10999",
                    Email = "diliana@example.com"
                }
             );
            context.SaveChanges();
        }

    }
}
```

Call `SeedData.Initialize` from `Program.cs`:

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using ContactManager.Data;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddRazorPages();

var app = builder.Build();

using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;

    await SeedData.Initialize(services);
}

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
```

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

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

app.UseRouting();

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

app.MapRazorPages();

app.Run();
```

Test that the app seeded the database. If there are any rows in the contact DB, the seed method doesn't run.

Additional resources

- Tutorial: Build an ASP.NET Core and Azure SQL Database app in Azure App Service
- ASP.NET Core Authorization Lab. This lab goes into more detail on the security features introduced in this tutorial.
- Introduction to authorization in ASP.NET Core
- Custom policy-based authorization

Razor Pages authorization conventions in ASP.NET Core

Article • 06/03/2022

One way to control access in your Razor Pages app is to use authorization conventions at startup. These conventions allow you to authorize users and allow anonymous users to access individual pages or folders of pages. The conventions described in this topic automatically apply authorization filters to control access.

View or download sample code (how to download)

The sample app uses cookie authentication without ASP.NET Core Identity. The concepts and examples shown in this topic apply equally to apps that use ASP.NET Core Identity. To use ASP.NET Core Identity, follow the guidance in Introduction to Identity on ASP.NET Core.

Require authorization to access a page

Use the AuthorizePage convention to add an AuthorizeFilter to the page at the specified path:

C#

```
services.AddRazorPages(options =>
{
    options.Conventions.AuthorizePage("/Contact");
    options.Conventions.AuthorizeFolder("/Private");
    options.Conventions.AllowAnonymousToPage("/Private/PublicPage");
    options.Conventions.AllowAnonymousToFolder("/Private/PublicPages");
});
```

The specified path is the View Engine path, which is the Razor Pages root relative path without an extension and containing only forward slashes.

To specify an authorization policy, use an AuthorizePage overload:

C#

```
options.Conventions.AuthorizePage("/Contact", "AtLeast21");
```

Note

An AuthorizeFilter can be applied to a page model class with the `[Authorize]` filter attribute. For more information, see Authorize filter attribute.

Require authorization to access a folder of pages

Use the AuthorizeFolder convention to add an AuthorizeFilter to all of the pages in a folder at the specified path:

C#

```
services.AddRazorPages(options =>
{
    options.Conventions.AuthorizePage("/Contact");
    options.Conventions.AuthorizeFolder("/Private");
    options.Conventions.AllowAnonymousToPage("/Private/PublicPage");
    options.Conventions.AllowAnonymousToFolder("/Private/PublicPages");
});
```

The specified path is the View Engine path, which is the Razor Pages root relative path.

To specify an authorization policy, use an AuthorizeFolder overload:

C#

```
options.Conventions.AuthorizeFolder("/Private", "AtLeast21");
```

Require authorization to access an area page

Use the AuthorizeAreaPage convention to add an AuthorizeFilter to the area page at the specified path:

C#

```
options.Conventions.AuthorizeAreaPage("Identity", "/Manage/Accounts");
```

The page name is the path of the file without an extension relative to the pages root directory for the specified area. For example, the page name for the file `Areas/Identity/Pages/Manage/Accounts.cshtml` is */Manage/Accounts*.

To specify an authorization policy, use an AuthorizeAreaPage overload:

C#

```
options.Conventions.AuthorizeAreaPage("Identity", "/Manage/Accounts",
"AtLeast21");
```

Require authorization to access a folder of areas

Use the AuthorizeAreaFolder convention to add an AuthorizeFilter to all of the areas in a folder at the specified path:

C#

```
options.Conventions.AuthorizeAreaFolder("Identity", "/Manage");
```

The folder path is the path of the folder relative to the pages root directory for the specified area. For example, the folder path for the files under *Areas/Identity/Pages/Manage/* is */Manage*.

To specify an authorization policy, use an AuthorizeAreaFolder overload:

C#

```
options.Conventions.AuthorizeAreaFolder("Identity", "/Manage", "AtLeast21");
```

Allow anonymous access to a page

Use the AllowAnonymousToPage convention to add an AllowAnonymousFilter to a page at the specified path:

C#

```
services.AddRazorPages(options =>
{
    options.Conventions.AuthorizePage("/Contact");
    options.Conventions.AuthorizeFolder("/Private");
    options.Conventions.AllowAnonymousToPage("/Private/PublicPage");
    options.Conventions.AllowAnonymousToFolder("/Private/PublicPages");
});
```

The specified path is the View Engine path, which is the Razor Pages root relative path without an extension and containing only forward slashes.

Allow anonymous access to a folder of pages

Use the AllowAnonymousToFolder convention to add an AllowAnonymousFilter to all of the pages in a folder at the specified path:

```
C#

services.AddRazorPages(options =>
{
    options.Conventions.AuthorizePage("/Contact");
    options.Conventions.AuthorizeFolder("/Private");
    options.Conventions.AllowAnonymousToPage("/Private/PublicPage");
    options.Conventions.AllowAnonymousToFolder("/Private/PublicPages");
});
```

The specified path is the View Engine path, which is the Razor Pages root relative path.

Note on combining authorized and anonymous access

It's valid to specify that a folder of pages requires authorization and then specify that a page within that folder allows anonymous access:

```
C#

// This works.
.AuthorizeFolder("/Private").AllowAnonymousToPage("/Private/Public")
```

The reverse, however, isn't valid. You can't declare a folder of pages for anonymous access and then specify a page within that folder that requires authorization:

```
C#

// This doesn't work!
.AllowAnonymousToFolder("/Public").AuthorizePage("/Public/Private")
```

Requiring authorization on the Private page fails. When both the AllowAnonymousFilter and AuthorizeFilter are applied to the page, the AllowAnonymousFilter takes precedence and controls access.

Additional resources

- Razor Pages route and app conventions in ASP.NET Core

- PageConventionCollection

Simple authorization in ASP.NET Core

Article • 05/02/2024

Authorization in ASP.NET Core is controlled with the [Authorize] attribute and its various parameters. In its most basic form, applying the `[Authorize]` attribute to a controller, action, or Razor Page, limits access to that component to authenticated users.

Prerequisites

This article assumes that you have a basic understanding of ASP.NET Core Razor Pages and MVC. If you're new to ASP.NET Core, see the following resources:

- Introduction to Razor Pages in ASP.NET Core
- Overview of ASP.NET Core MVC
- Tutorial: Get started with Razor Pages in ASP.NET Core
- Introduction to Identity on ASP.NET Core

Use the `[Authorize]` attribute

The following code limits access to the `AccountController` to authenticated users:

C#

```
[Authorize]
public class AccountController : Controller
{
    public ActionResult Login()
    {
    }

    public ActionResult Logout()
    {
    }
}
```

If you want to apply authorization to an action rather than the controller, apply the `AuthorizeAttribute` attribute to the action itself:

C#

```
public class AccountController : Controller
{
   public ActionResult Login()
```

```
    {
    }

    [Authorize]
    public ActionResult Logout()
    {
    }
}
```

Now only authenticated users can access the `Logout` function.

You can also use the `AllowAnonymous` attribute to allow access by non-authenticated users to individual actions. For example:

C#

```
[Authorize]
public class AccountController : Controller
{
    [AllowAnonymous]
    public ActionResult Login()
    {
    }

    public ActionResult Logout()
    {
    }
}
```

This would allow only authenticated users to the `AccountController`, except for the `Login` action, which is accessible by everyone, regardless of their authenticated or unauthenticated / anonymous status.

> ⚠ **Warning**
>
> `[AllowAnonymous]` bypasses authorization statements. If you combine `[AllowAnonymous]` and an `[Authorize]` attribute, the `[Authorize]` attributes are ignored. For example if you apply `[AllowAnonymous]` at the controller level:
>
> - Any authorization requirements from `[Authorize]` attributes on the same controller or action methods on the controller are ignored.
> - Authentication middleware is not short-circuited but doesn't need to succeed.

The following code limits access to the `LogoutModel` Razor Page to authenticated users:

C#

```
[Authorize]
public class LogoutModel : PageModel
{
    public async Task OnGetAsync()
    {

    }

    public async Task<IActionResult> OnPostAsync()
    {

    }
}
```

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

Authorize attribute and Razor Pages

The AuthorizeAttribute can ***not*** be applied to Razor Page handlers. For example, `[Authorize]` can't be applied to `OnGet`, `OnPost`, or any other page handler. Consider using an ASP.NET Core MVC controller for pages with different authorization requirements for different handlers. Using an MVC controller when different authorization requirements are required:

- Is the least complex approach.
- Is the approach recommended by Microsoft.

If you decide not to use an MVC controller, the following two approaches can be used to apply authorization to Razor Page handler methods:

- Use separate pages for page handlers requiring different authorization. Move shared content into one or more partial views. When possible, this is the recommended approach.

- For content that must share a common page, write a filter that performs authorization as part of IAsyncPageFilter.OnPageHandlerSelectionAsync. The PageHandlerAuth GitHub project demonstrates this approach:
  - The AuthorizeIndexPageHandlerFilter implements the authorization filter:

    C#

    ```
    [TypeFilter(typeof(AuthorizeIndexPageHandlerFilter))]
    public class IndexModel : PageModel
    ```

```
{
    private readonly ILogger<IndexModel> _logger;

    public IndexModel(ILogger<IndexModel> logger)
    {
        _logger = logger;
    }

    public void OnGet()
    {

    }

    public void OnPost()
    {

    }

    [AuthorizePageHandler]
    public void OnPostAuthorized()
    {

    }
}
```

- The [AuthorizePageHandler] attribute is applied to the `OnPostAuthorized` page handler:

C#

```
public class AuthorizeIndexPageHandlerFilter : IAsyncPageFilter,
IOrderedFilter
{
    private readonly IAuthorizationPolicyProvider policyProvider;
    private readonly IPolicyEvaluator policyEvaluator;

    public AuthorizeIndexPageHandlerFilter(
        IAuthorizationPolicyProvider policyProvider,
        IPolicyEvaluator policyEvaluator)
    {
        this.policyProvider = policyProvider;
        this.policyEvaluator = policyEvaluator;
    }

    // Run late in the selection pipeline
    public int Order => 10000;

    public Task OnPageHandlerExecutionAsync(PageHandlerExecutingContext
context, PageHandlerExecutionDelegate next) => next();

    public async Task
OnPageHandlerSelectionAsync(PageHandlerSelectedContext context)
    {
```

```
        var attribute =
context.HandlerMethod?.MethodInfo?.GetCustomAttribute<AuthorizePageHand
lerAttribute>();
        if (attribute is null)
        {
            return;
        }

        var policy = await
AuthorizationPolicy.CombineAsync(policyProvider, new[] { attribute });
        if (policy is null)
        {
            return;
        }

        await AuthorizeAsync(context, policy);
    }

    #region AuthZ - do not change
    private async Task AuthorizeAsync(ActionContext actionContext,
AuthorizationPolicy policy)
    {
        var httpContext = actionContext.HttpContext;
        var authenticateResult = await
policyEvaluator.AuthenticateAsync(policy, httpContext);
        var authorizeResult = await
policyEvaluator.AuthorizeAsync(policy, authenticateResult, httpContext,
actionContext.ActionDescriptor);
        if (authorizeResult.Challenged)
        {
            if (policy.AuthenticationSchemes.Count > 0)
            {
                foreach (var scheme in policy.AuthenticationSchemes)
                {
                    await httpContext.ChallengeAsync(scheme);
                }
            }
            else
            {
                await httpContext.ChallengeAsync();
            }

            return;
        }
        else if (authorizeResult.Forbidden)
        {
            if (policy.AuthenticationSchemes.Count > 0)
            {
                foreach (var scheme in policy.AuthenticationSchemes)
                {
                    await httpContext.ForbidAsync(scheme);
                }
            }
            else
            {
```

```
                await httpContext.ForbidAsync();
            }

            return;
        }
    }
```

> ⚠ **Warning**
>
> The PageHandlerAuth sample approach does ***not***:
>
> - Compose with authorization attributes applied to the page, page model, or globally. Composing authorization attributes results in authentication and authorization executing multiple times when you have one more `AuthorizeAttribute` or `AuthorizeFilter` instances also applied to the page.
> - Work in conjunction with the rest of ASP.NET Core authentication and authorization system. You must verify using this approach works correctly for your application.

There are no plans to support the `AuthorizeAttribute` on Razor Page handlers.

Custom authorization policies with IAuthorizationRequirementData

Article • 06/16/2023

Consider the following sample that implements a custom `MinimumAgeAuthorizationHandler`:

C#

```
using AuthRequirementsData.Authorization;
using Microsoft.AspNetCore.Authorization;

var builder = WebApplication.CreateBuilder();

builder.Services.AddAuthentication().AddJwtBearer();
builder.Services.AddAuthorization();
builder.Services.AddControllers();
builder.Services.AddSingleton<IAuthorizationHandler,
MinimumAgeAuthorizationHandler>();

var app = builder.Build();

app.MapControllers();

app.Run();
```

The `MinimumAgeAuthorizationHandler` class:

C#

```
using Microsoft.AspNetCore.Authorization;
using System.Globalization;
using System.Security.Claims;

namespace AuthRequirementsData.Authorization;

class MinimumAgeAuthorizationHandler :
AuthorizationHandler<MinimumAgeAuthorizeAttribute>
{
    private readonly ILogger<MinimumAgeAuthorizationHandler> _logger;

    public
MinimumAgeAuthorizationHandler(ILogger<MinimumAgeAuthorizationHandler>
logger)
    {
        _logger = logger;
    }
```

```
    // Check whether a given MinimumAgeRequirement is satisfied or not for a
particular
    // context.
    protected override Task
HandleRequirementAsync(AuthorizationHandlerContext context,
                                                MinimumAgeAuthorizeAttribute
requirement)
    {
        // Log as a warning so that it's very clear in sample output which
authorization
        // policies(and requirements/handlers) are in use.
        _logger.LogWarning("Evaluating authorization requirement for age >=
{age}",

requirement.Age);

        // Check the user's age.
        var dateOfBirthClaim = context.User.FindFirst(c => c.Type ==

ClaimTypes.DateOfBirth);
        if (dateOfBirthClaim != null)
        {
            // If the user has a date of birth claim, check their age.
            var dateOfBirth = Convert.ToDateTime(dateOfBirthClaim.Value,

CultureInfo.InvariantCulture);
            var age = DateTime.Now.Year - dateOfBirth.Year;
            if (dateOfBirth > DateTime.Now.AddYears(-age))
            {
                // Adjust age if the user hasn't had a birthday yet this
year.
                age--;
            }

            // If the user meets the age criterion, mark the authorization
requirement
            // succeeded.
            if (age >= requirement.Age)
            {
                _logger.LogInformation(
                    "Minimum age authorization requirement {age} satisfied",
                      requirement.Age);
                context.Succeed(requirement);
            }
            else
            {
                _logger.LogInformation("Current user's DateOfBirth claim
({dateOfBirth})"
                   + " does not satisfy the minimum age authorization
requirement {age}",
                    dateOfBirthClaim.Value,
                    requirement.Age);
            }
        }
        else
```

```
        {
            _logger.LogInformation("No DateOfBirth claim present");
        }

        return Task.CompletedTask;
    }
}
```

The custom `MinimumAgePolicyProvider`:

C#

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Authorization;
using Microsoft.Extensions.Options;

namespace AuthRequirementsData.Authorization;

class MinimumAgePolicyProvider : IAuthorizationPolicyProvider
{
    const string POLICY_PREFIX = "MinimumAge";
    public DefaultAuthorizationPolicyProvider FallbackPolicyProvider { get;
}
    public MinimumAgePolicyProvider(IOptions<AuthorizationOptions> options)
    {
        FallbackPolicyProvider = new
DefaultAuthorizationPolicyProvider(options);
    }
    public Task<AuthorizationPolicy> GetDefaultPolicyAsync() =>
                            FallbackPolicyProvider.GetDefaultPolicyAsync();
    public Task<AuthorizationPolicy?> GetFallbackPolicyAsync() =>
                            FallbackPolicyProvider.GetFallbackPolicyAsync();

    public Task<AuthorizationPolicy?> GetPolicyAsync(string policyName)
    {
        if (policyName.StartsWith(POLICY_PREFIX,
StringComparison.OrdinalIgnoreCase) &&
            int.TryParse(policyName.Substring(POLICY_PREFIX.Length), out var
age))
        {
            var policy = new AuthorizationPolicyBuilder(

JwtBearerDefaults.AuthenticationScheme);
            policy.AddRequirements(new MinimumAgeRequirement(age));
            return Task.FromResult<AuthorizationPolicy?>(policy.Build());
        }

        return Task.FromResult<AuthorizationPolicy?>(null);
    }
}
```

ASP.NET Core only uses one authorization policy provider. If the custom implementation doesn't handle all policies, including default policies, etc., it should fall back to an alternate provider. In the preceding sample, a default authorization policy provider is:

- Constructed with options from the dependency injection container.
- Used if this custom provider isn't able to handle a given policy name.

If a custom policy provider is able to handle all expected policy names, setting the fallback policy with GetFallbackPolicyAsync() isn't required.

C#

```
class MinimumAgePolicyProvider : IAuthorizationPolicyProvider
{
    const string POLICY_PREFIX = "MinimumAge";
    public DefaultAuthorizationPolicyProvider FallbackPolicyProvider { get;
}
    public MinimumAgePolicyProvider(IOptions<AuthorizationOptions> options)
    {
        FallbackPolicyProvider = new
DefaultAuthorizationPolicyProvider(options);
    }
    public Task<AuthorizationPolicy> GetDefaultPolicyAsync() =>
                            FallbackPolicyProvider.GetDefaultPolicyAsync();
    public Task<AuthorizationPolicy?> GetFallbackPolicyAsync() =>
                            FallbackPolicyProvider.GetFallbackPolicyAsync();
```

Policies are looked up by string name, therefore parameters, for example, `age`, are embedded in the policy names. This is abstracted away from developers by the more strongly-typed attributes derived from AuthorizeAttribute. For example, the `[MinimumAgeAuthorize()]` attribute in this sample looks up policies by string name.

C#

```
public Task<AuthorizationPolicy?> GetPolicyAsync(string policyName)
{
    if (policyName.StartsWith(POLICY_PREFIX,
StringComparison.OrdinalIgnoreCase) &&
        int.TryParse(policyName.Substring(POLICY_PREFIX.Length), out var
age))
    {
        var policy = new AuthorizationPolicyBuilder(

JwtBearerDefaults.AuthenticationScheme);
        policy.AddRequirements(new MinimumAgeRequirement(age));
        return Task.FromResult<AuthorizationPolicy?>(policy.Build());
    }
```

```
    return Task.FromResult<AuthorizationPolicy?>(null);
}
```

The `MinimumAgeAuthorizeAttribute` uses the IAuthorizationRequirementData interface that allows the attribute definition to specify the requirements associated with the authorization policy:

C#

```
using Microsoft.AspNetCore.Authorization;

namespace AuthRequirementsData.Authorization;

class MinimumAgeAuthorizeAttribute : AuthorizeAttribute,
IAuthorizationRequirement,
                                     IAuthorizationRequirementData
{
    public MinimumAgeAuthorizeAttribute(int age) => Age = age;
    public int Age { get; }

    public IEnumerable<IAuthorizationRequirement> GetRequirements()
    {
        yield return this;
    }
}
```

The `GreetingsController` displays the user's name when they satisfy the minimum age policy:

C#

```
using AuthRequirementsData.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace AuthRequirementsData.Controllers;

[ApiController]
[Route("api/[controller]")]
public class GreetingsController : Controller
{
    [MinimumAgeAuthorize(16)]
    [HttpGet("hello")]
    public string Hello() => $"Hello {(HttpContext.User.Identity?.Name ??
"world")}!";
}
```

The complete sample can be found in the AuthRequirementsData folder of the AspNetCore.Docs.Samples repository.

The sample can be tested with dotnet user-jwts and curl:

- `dotnet user-jwts create --claim http://schemas.xmlsoap.org/ws/2005/05/identity/claims/dateofbirth=1989-01-01`
- `curl -i -H "Authorization: Bearer <token from dotnet user-jwts>" http://localhost:<port>/api/greetings/hello`

Role-based authorization in ASP.NET Core

Article • 09/27/2024

When an identity is created it may belong to one or more roles. For example, Tracy may belong to the `Administrator` and `User` roles while Scott may only belong to the `User` role. How these roles are created and managed depends on the backing store of the authorization process. Roles are exposed to the developer through the IsInRole method on the ClaimsPrincipal class. AddRoles must be added to Role services.

While roles are claims, not all claims are roles. Depending on the identity issuer a role may be a collection of users that may apply claims for group members, as well as an actual claim on an identity. However, claims are meant to be information about an individual user. Using roles to add claims to a user can confuse the boundary between the user and their individual claims. This confusion is why the SPA templates are not designed around roles. In addition, for organizations migrating from an on-premises legacy system the proliferation of roles over the years can mean a role claim may be too large to be contained within a token usable by SPAs. To secure SPAs, see Use Identity to secure a Web API backend for SPAs.

Add Role services to Identity

Register role-based authorization services in `Program.cs` by calling AddRoles with the role type in the app's Identity configuration. The role type in the following example is `IdentityRole`:

C#

```
builder.Services.AddDefaultIdentity<IdentityUser>( ... )
    .AddRoles<IdentityRole>()
    ...
```

The preceding code requires the Microsoft.AspNetCore.Identity.UI package and a `using` directive for `Microsoft.AspNetCore.Identity`.

Adding role checks

Role based authorization checks:

- Are declarative and specify roles which the current user must be a member of to access the requested resource.
- Are applied to Razor Pages, controllers, or actions within a controller.
- Can ***not*** be applied at the Razor Page handler level, they must be applied to the Page.

For example, the following code limits access to any actions on the `AdministrationController` to users who are a member of the `Administrator` role:

C#

```
[Authorize(Roles = "Administrator")]
public class AdministrationController : Controller
{
    public IActionResult Index() =>
        Content("Administrator");
}
```

Multiple roles can be specified as a comma separated list:

C#

```
[Authorize(Roles = "HRManager,Finance")]
public class SalaryController : Controller
{
    public IActionResult Payslip() =>
                    Content("HRManager || Finance");
}
```

The `SalaryController` is only accessible by users who are members of the `HRManager` role ***or*** the `Finance` role.

When multiple attributes are applied, an accessing user must be a member of ***all*** the roles specified. The following sample requires that a user must be a member of ***both*** the `PowerUser` ***and*** `ControlPanelUser` role:

C#

```
[Authorize(Roles = "PowerUser")]
[Authorize(Roles = "ControlPanelUser")]
public class ControlPanelController : Controller
{
    public IActionResult Index() =>
        Content("PowerUser && ControlPanelUser");
}
```

Access to an action can be limited by applying additional role authorization attributes at the action level:

C#

```
[Authorize(Roles = "Administrator, PowerUser")]
public class ControlAllPanelController : Controller
{
    public IActionResult SetTime() =>
        Content("Administrator || PowerUser");

    [Authorize(Roles = "Administrator")]
    public IActionResult ShutDown() =>
        Content("Administrator only");
}
```

In the preceding `ControlAllPanelController` controller:

- Members of the `Administrator` role or the `PowerUser` role can access the controller and the `SetTime` action.
- Only members of the `Administrator` role can access the `ShutDown` action.

A controller can be secured but allow anonymous, unauthenticated access to individual actions:

C#

```
[Authorize]
public class Control3PanelController : Controller
{
    public IActionResult SetTime() =>
        Content("[Authorize]");

    [AllowAnonymous]
    public IActionResult Login() =>
        Content("[AllowAnonymous]");
}
```

For Razor Pages, [Authorize] can be applied by either:

- Using a convention, or
- Applying the [Authorize] to the `PageModel` instance:

C#

```
[Authorize(Policy = "RequireAdministratorRole")]
public class UpdateModel : PageModel
{
```

```
    public IActionResult OnPost() =>
        Content("OnPost RequireAdministratorRole");
}
```

> **ⓘ Important**
>
> Filter attributes, including `AuthorizeAttribute`, can only be applied to PageModel and cannot be applied to specific page handler methods.

Policy based role checks

Role requirements can also be expressed using the Policy syntax, where a developer registers a policy at application startup as part of the Authorization service configuration. This typically occurs in the `Program.cs` file:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("RequireAdministratorRole",
        policy => policy.RequireRole("Administrator"));
});

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

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

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

app.MapDefaultControllerRoute();
app.MapRazorPages();

app.Run();
```

Policies are applied using the Policy property on the [Authorize] attribute:

```
C#

[Authorize(Policy = "RequireAdministratorRole")]
public IActionResult Shutdown()
{
    return View();
}
```

To specify multiple allowed roles in a requirement, specify them as parameters to the RequireRole method:

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("ElevatedRights", policy =>
          policy.RequireRole("Administrator", "PowerUser",
"BackupAdministrator"));
});

var app = builder.Build();
```

The preceding code authorizes users who belong to the `Administrator`, `PowerUser` or `BackupAdministrator` roles.

Claims-based authorization in ASP.NET Core

Article • 11/10/2023

When an identity is created it may be assigned one or more claims issued by a trusted party. A claim is a name value pair that represents what the subject is, not what the subject can do. For example, you may have a driver's license, issued by a local driving license authority. Your driver's license has your date of birth on it. In this case the claim name would be `DateOfBirth`, the claim value would be your date of birth, for example `8th June 1970` and the issuer would be the driving license authority. Claims-based authorization, at its simplest, checks the value of a claim and allows access to a resource based upon that value. For example if you want access to a night club the authorization process might be:

The door security officer would evaluate the value of your date of birth claim and whether they trust the issuer (the driving license authority) before granting you access.

An identity can contain multiple claims with multiple values and can contain multiple claims of the same type.

Adding claims checks

Claim-based authorization checks:

- Are declarative.
- Are applied to Razor Pages, controllers, or actions within a controller.
- Can ***not*** be applied at the Razor Page handler level, they must be applied to the Page.

Claims in code specify claims which the current user must possess, and optionally the value the claim must hold to access the requested resource. Claims requirements are policy based; the developer must build and register a policy expressing the claims requirements.

The simplest type of claim policy looks for the presence of a claim and doesn't check the value.

Build and register the policy and call UseAuthorization. Registering the policy takes place as part of the Authorization service configuration, typically in the `Program.cs` file:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.AddAuthorization(options =>
{
   options.AddPolicy("EmployeeOnly", policy =>
policy.RequireClaim("EmployeeNumber"));
});

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

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

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

app.MapDefaultControllerRoute();
app.MapRazorPages();

app.Run();
```

In this case the `EmployeeOnly` policy checks for the presence of an `EmployeeNumber` claim on the current identity.

Apply the policy using the `Policy` property on the [Authorize] attribute to specify the policy name.

C#

```
[Authorize(Policy = "EmployeeOnly")]
public IActionResult VacationBalance()
{
    return View();
}
```

The `[Authorize]` attribute can be applied to an entire controller or Razor Page, in which case only identities matching the policy are allowed access to any Action on the controller.

C#

```
[Authorize(Policy = "EmployeeOnly")]
public class VacationController : Controller
{
    public IActionResult Index()
    {
        return View();
    }

    public ActionResult VacationBalance()
    {
        return View();
    }

    [AllowAnonymous]
    public ActionResult VacationPolicy()
    {
        return View();
    }
}
```

The following code applies the `[Authorize]` attribute to a Razor Page:

C#

```
[Authorize(Policy = "EmployeeOnly")]
public class IndexModel : PageModel
{
    public void OnGet()
    {

    }
}
```

Policies can ***not*** be applied at the Razor Page handler level, they must be applied to the Page.

If you have a controller that's protected by the `[Authorize]` attribute but want to allow anonymous access to particular actions, you apply the `AllowAnonymousAttribute` attribute.

C#

```
[Authorize(Policy = "EmployeeOnly")]
public class VacationController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
```

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

    [AllowAnonymous]
    public ActionResult VacationPolicy()
    {
        return View();
    }
}
```

Because policies can ***not*** be applied at the Razor Page handler level, we recommend using a controller when policies must be applied at the page handler level. The rest of the app that doesn't require policies at the Razor Page handler level can use Razor Pages.

Most claims come with a value. You can specify a list of allowed values when creating the policy. The following example would only succeed for employees whose employee number was 1, 2, 3, 4, or 5.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("Founders", policy =>
                      policy.RequireClaim("EmployeeNumber", "1", "2", "3",
"4", "5"));
});

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

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

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

app.MapDefaultControllerRoute();
```

```
app.MapRazorPages();

app.Run();
```

Add a generic claim check

If the claim value isn't a single value or a transformation is required, use RequireAssertion. For more information, see Use a func to fulfill a policy.

Multiple Policy Evaluation

If multiple policies are applied at the controller and action levels, ***all*** policies must pass before access is granted:

C#

```
[Authorize(Policy = "EmployeeOnly")]
public class SalaryController : Controller
{
    public IActionResult Index()
    {
        return View();
    }

    public IActionResult Payslip()
    {
        return View();
    }

    [Authorize(Policy = "HumanResources")]
    public IActionResult UpdateSalary()
    {
        return View();
    }
}
```

In the preceding example any identity which fulfills the `EmployeeOnly` policy can access the `Payslip` action as that policy is enforced on the controller. However, in order to call the `UpdateSalary` action the identity must fulfill *both* the `EmployeeOnly` policy and the `HumanResources` policy.

If you want more complicated policies, such as taking a date of birth claim, calculating an age from it, and then checking that the age is 21 or older, then you need to write custom policy handlers.

In the following sample, both page handler methods must fulfill *both* the `EmployeeOnly` policy and the `HumanResources` policy:

C#

```
[Authorize(Policy = "EmployeeOnly")]
[Authorize(Policy = "HumanResources")]
public class SalaryModel : PageModel
{
    public ContentResult OnGetPayStub()
    {
        return Content("OnGetPayStub");
    }

    public ContentResult OnGetSalary()
    {
        return Content("OnGetSalary");
    }
}
```

Policy-based authorization in ASP.NET Core

Article • 08/08/2024

Underneath the covers, role-based authorization and claims-based authorization use a requirement, a requirement handler, and a preconfigured policy. These building blocks support the expression of authorization evaluations in code. The result is a richer, reusable, testable authorization structure.

An authorization policy consists of one or more requirements. Register it as part of the authorization service configuration, in the app's `Program.cs` file:

C#

```
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("AtLeast21", policy =>
        policy.Requirements.Add(new MinimumAgeRequirement(21)));
});
```

In the preceding example, an "AtLeast21" policy is created. It has a single requirement—that of a minimum age, which is supplied as a parameter to the requirement.

IAuthorizationService

The primary service that determines if authorization is successful is IAuthorizationService:

C#

```
/// <summary>
/// Checks policy based permissions for a user
/// </summary>
public interface IAuthorizationService
{
    /// <summary>
    /// Checks if a user meets a specific set of requirements for the
specified resource
    /// </summary>
    /// <param name="user">The user to evaluate the requirements against.
</param>
    /// <param name="resource">
    /// An optional resource the policy should be checked with.
    /// If a resource is not required for policy evaluation you may pass
null as the value
```

```
    /// </param>
    /// <param name="requirements">The requirements to evaluate.</param>
    /// <returns>
    /// A flag indicating whether authorization has succeeded.
    /// This value is <value>true</value> when the user fulfills the policy;
    /// otherwise <value>false</value>.
    /// </returns>
    /// <remarks>
    /// Resource is an optional parameter and may be null. Please ensure
that you check
    /// it is not null before acting upon it.
    /// </remarks>
    Task<AuthorizationResult> AuthorizeAsync(ClaimsPrincipal user, object
resource,
                                        IEnumerable<IAuthorizationRequirement>
requirements);

    /// <summary>
    /// Checks if a user meets a specific authorization policy
    /// </summary>
    /// <param name="user">The user to check the policy against.</param>
    /// <param name="resource">
    /// An optional resource the policy should be checked with.
    /// If a resource is not required for policy evaluation you may pass
null as the value
    /// </param>
    /// <param name="policyName">The name of the policy to check against a
specific
    /// context.</param>
    /// <returns>
    /// A flag indicating whether authorization has succeeded.
    /// Returns a flag indicating whether the user, and optional resource
has fulfilled
    /// the policy.
    /// <value>true</value> when the policy has been fulfilled;
    /// otherwise <value>false</value>.
    /// </returns>
    /// <remarks>
    /// Resource is an optional parameter and may be null. Please ensure
that you check
    /// it is not null before acting upon it.
    /// </remarks>
    Task<AuthorizationResult> AuthorizeAsync(
                                    ClaimsPrincipal user, object resource,
string policyName);
}
```

The preceding code highlights the two methods of the IAuthorizationService.

IAuthorizationRequirement is a marker service with no methods, and the mechanism for tracking whether authorization is successful.

Each IAuthorizationHandler is responsible for checking if requirements are met:

C#

```
/// <summary>
/// Classes implementing this interface are able to make a decision if
authorization
/// is allowed.
/// </summary>
public interface IAuthorizationHandler
{
    /// <summary>
    /// Makes a decision if authorization is allowed.
    /// </summary>
    /// <param name="context">The authorization information.</param>
    Task HandleAsync(AuthorizationHandlerContext context);
}
```

The AuthorizationHandlerContext class is what the handler uses to mark whether requirements have been met:

C#

```
 context.Succeed(requirement)
```

The following code shows the simplified (and annotated with comments) default implementation of the authorization service:

C#

```
public async Task<AuthorizationResult> AuthorizeAsync(ClaimsPrincipal user,
             object resource, IEnumerable<IAuthorizationRequirement>
requirements)
{
    // Create a tracking context from the authorization inputs.
    var authContext = _contextFactory.CreateContext(requirements, user,
resource);

    // By default this returns an IEnumerable<IAuthorizationHandler> from
DI.
    var handlers = await _handlers.GetHandlersAsync(authContext);

    // Invoke all handlers.
    foreach (var handler in handlers)
    {
        await handler.HandleAsync(authContext);
    }

    // Check the context, by default success is when all requirements have
been met.
```

```
    return _evaluator.Evaluate(authContext);
}
```

The following code shows a typical authorization service configuration:

C#

```
// Add all of your handlers to DI.
builder.Services.AddSingleton<IAuthorizationHandler, MyHandler1>();
// MyHandler2, ...

builder.Services.AddSingleton<IAuthorizationHandler, MyHandlerN>();

// Configure your policies
builder.Services.AddAuthorization(options =>
      options.AddPolicy("Something",
      policy => policy.RequireClaim("Permission", "CanViewPage",
"CanViewAnything")));
```

Use IAuthorizationService, `[Authorize(Policy = "Something")]`, or `RequireAuthorization("Something")` for authorization.

Apply policies to MVC controllers

For apps that use Razor Pages, see the Apply policies to Razor Pages section.

Apply policies to controllers by using the `[Authorize]` attribute with the policy name:

C#

```
[Authorize(Policy = "AtLeast21")]
public class AtLeast21Controller : Controller
{
    public IActionResult Index() => View();
}
```

If multiple policies are applied at the controller and action levels, ***all*** policies must pass before access is granted:

C#

```
[Authorize(Policy = "AtLeast21")]
public class AtLeast21Controller2 : Controller
{
    [Authorize(Policy = "IdentificationValidated")]
```

```
    public IActionResult Index() => View();
}
```

Apply policies to Razor Pages

Apply policies to Razor Pages by using the `[Authorize]` attribute with the policy name. For example:

C#

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

namespace AuthorizationPoliciesSample.Pages;

[Authorize(Policy = "AtLeast21")]
public class AtLeast21Model : PageModel { }
```

Policies can ***not*** be applied at the Razor Page handler level, they must be applied to the Page.

Policies can also be applied to Razor Pages by using an authorization convention.

Apply policies to endpoints

Apply policies to endpoints by using RequireAuthorization with the policy name. For example:

C#

```
app.MapGet("/helloworld", () => "Hello World!")
    .RequireAuthorization("AtLeast21");
```

Requirements

An authorization requirement is a collection of data parameters that a policy can use to evaluate the current user principal. In our "AtLeast21" policy, the requirement is a single parameter—the minimum age. A requirement implements IAuthorizationRequirement, which is an empty marker interface. A parameterized minimum age requirement could be implemented as follows:

C#

```
using Microsoft.AspNetCore.Authorization;

namespace AuthorizationPoliciesSample.Policies.Requirements;

public class MinimumAgeRequirement : IAuthorizationRequirement
{
    public MinimumAgeRequirement(int minimumAge) =>
        MinimumAge = minimumAge;

    public int MinimumAge { get; }
}
```

If an authorization policy contains multiple authorization requirements, all requirements must pass in order for the policy evaluation to succeed. In other words, multiple authorization requirements added to a single authorization policy are treated on an **AND** basis.

> ⓘ **Note**
>
> A requirement doesn't need to have data or properties.

Authorization handlers

An authorization handler is responsible for the evaluation of a requirement's properties. The authorization handler evaluates the requirements against a provided AuthorizationHandlerContext to determine if access is allowed.

A requirement can have multiple handlers. A handler may inherit AuthorizationHandler<TRequirement>, where `TRequirement` is the requirement to be handled. Alternatively, a handler may implement IAuthorizationHandler directly to handle more than one type of requirement.

Use a handler for one requirement

The following example shows a one-to-one relationship in which a minimum age handler handles a single requirement:

C#

```
using System.Security.Claims;
using AuthorizationPoliciesSample.Policies.Requirements;
using Microsoft.AspNetCore.Authorization;
```

```
namespace AuthorizationPoliciesSample.Policies.Handlers;

public class MinimumAgeHandler : AuthorizationHandler<MinimumAgeRequirement>
{
    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context, MinimumAgeRequirement
requirement)
    {
        var dateOfBirthClaim = context.User.FindFirst(
            c => c.Type == ClaimTypes.DateOfBirth && c.Issuer ==
"http://contoso.com");

        if (dateOfBirthClaim is null)
        {
            return Task.CompletedTask;
        }

        var dateOfBirth = Convert.ToDateTime(dateOfBirthClaim.Value);
        int calculatedAge = DateTime.Today.Year - dateOfBirth.Year;
        if (dateOfBirth > DateTime.Today.AddYears(-calculatedAge))
        {
            calculatedAge--;
        }

        if (calculatedAge >= requirement.MinimumAge)
        {
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }
}
```

The preceding code determines if the current user principal has a date of birth claim that has been issued by a known and trusted Issuer. Authorization can't occur when the claim is missing, in which case a completed task is returned. When a claim is present, the user's age is calculated. If the user meets the minimum age defined by the requirement, authorization is considered successful. When authorization is successful, `context.Succeed` is invoked with the satisfied requirement as its sole parameter.

Use a handler for multiple requirements

The following example shows a one-to-many relationship in which a permission handler can handle three different types of requirements:

C#

```
using System.Security.Claims;
using AuthorizationPoliciesSample.Policies.Requirements;
using Microsoft.AspNetCore.Authorization;
```

```
namespace AuthorizationPoliciesSample.Policies.Handlers;

public class PermissionHandler : IAuthorizationHandler
{
    public Task HandleAsync(AuthorizationHandlerContext context)
    {
        var pendingRequirements = context.PendingRequirements.ToList();

        foreach (var requirement in pendingRequirements)
        {
            if (requirement is ReadPermission)
            {
                if (IsOwner(context.User, context.Resource)
                    || IsSponsor(context.User, context.Resource))
                {
                    context.Succeed(requirement);
                }
            }
            else if (requirement is EditPermission || requirement is
DeletePermission)
            {
                if (IsOwner(context.User, context.Resource))
                {
                    context.Succeed(requirement);
                }
            }
        }

        return Task.CompletedTask;
    }

    private static bool IsOwner(ClaimsPrincipal user, object? resource)
    {
        // Code omitted for brevity
        return true;
    }

    private static bool IsSponsor(ClaimsPrincipal user, object? resource)
    {
        // Code omitted for brevity
        return true;
    }
}
```

The preceding code traverses PendingRequirements—a property containing requirements not marked as successful. For a `ReadPermission` requirement, the user must be either an owner or a sponsor to access the requested resource. For an `EditPermission` or `DeletePermission` requirement, they must be an owner to access the requested resource.

Handler registration

Register handlers in the services collection during configuration. For example:

```
C#

builder.Services.AddSingleton<IAuthorizationHandler, MinimumAgeHandler>();
```

The preceding code registers `MinimumAgeHandler` as a singleton. Handlers can be registered using any of the built-in service lifetimes.

It's possible to bundle both a requirement and a handler into a single class implementing both IAuthorizationRequirement and IAuthorizationHandler. This bundling creates a tight coupling between the handler and requirement and is only recommended for simple requirements and handlers. Creating a class that implements both interfaces removes the need to register the handler in DI because of the built-in PassThroughAuthorizationHandler that allows requirements to handle themselves.

See the AssertionRequirement class for a good example where the `AssertionRequirement` is both a requirement and the handler in a fully self-contained class.

What should a handler return?

Note that the `Handle` method in the handler example returns no value. How is a status of either success or failure indicated?

- A handler indicates success by calling `context.Succeed(IAuthorizationRequirement requirement)`, passing the requirement that has been successfully validated.
- A handler doesn't need to handle failures generally, as other handlers for the same requirement may succeed.
- To guarantee failure, even if other requirement handlers succeed, call `context.Fail`.

If a handler calls `context.Succeed` or `context.Fail`, all other handlers are still called. This allows requirements to produce side effects, such as logging, which takes place even if another handler has successfully validated or failed a requirement. When set to `false`, the InvokeHandlersAfterFailure property short-circuits the execution of handlers when `context.Fail` is called. `InvokeHandlersAfterFailure` defaults to `true`, in which case all handlers are called.

> ⓘ Note
>
> Authorization handlers are called even if authentication fails. Also handlers can execute in any order, so do ***not*** depend on them being called in any particular order.

Why would I want multiple handlers for a requirement?

In cases where you want evaluation to be on an **OR** basis, implement multiple handlers for a single requirement. For example, Microsoft has doors that only open with key cards. If you leave your key card at home, the receptionist prints a temporary sticker and opens the door for you. In this scenario, you'd have a single requirement, *BuildingEntry*, but multiple handlers, each one examining a single requirement.

`BuildingEntryRequirement.cs`

C#

```
using Microsoft.AspNetCore.Authorization;

namespace AuthorizationPoliciesSample.Policies.Requirements;

public class BuildingEntryRequirement : IAuthorizationRequirement { }
```

`BadgeEntryHandler.cs`

C#

```
using AuthorizationPoliciesSample.Policies.Requirements;
using Microsoft.AspNetCore.Authorization;

namespace AuthorizationPoliciesSample.Policies.Handlers;

public class BadgeEntryHandler :
AuthorizationHandler<BuildingEntryRequirement>
{
    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context, BuildingEntryRequirement
requirement)
    {
        if (context.User.HasClaim(
            c => c.Type == "BadgeId" && c.Issuer ==
"https://microsoftsecurity"))
        {
```

```
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }
}
```

`TemporaryStickerHandler.cs`

C#

```
using AuthorizationPoliciesSample.Policies.Requirements;
using Microsoft.AspNetCore.Authorization;

namespace AuthorizationPoliciesSample.Policies.Handlers;

public class TemporaryStickerHandler :
AuthorizationHandler<BuildingEntryRequirement>
{
    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context, BuildingEntryRequirement
requirement)
    {
        if (context.User.HasClaim(
            c => c.Type == "TemporaryBadgeId" && c.Issuer ==
"https://microsoftsecurity"))
        {
            // Code to check expiration date omitted for brevity.
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }
}
```

Ensure that both handlers are registered. If either handler succeeds when a policy evaluates the `BuildingEntryRequirement`, the policy evaluation succeeds.

Use a func to fulfill a policy

There may be situations in which fulfilling a policy is simple to express in code. It's possible to supply a `Func<AuthorizationHandlerContext, bool>` when configuring a policy with the `RequireAssertion` policy builder.

For example, the previous `BadgeEntryHandler` could be rewritten as follows:

C#

```
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("BadgeEntry", policy =>
        policy.RequireAssertion(context => context.User.HasClaim(c =>
            (c.Type == "BadgeId" || c.Type == "TemporaryBadgeId")
            && c.Issuer == "https://microsoftsecurity")));
});
```

Access MVC request context in handlers

The `HandleRequirementAsync` method has two parameters: an `AuthorizationHandlerContext` and the `TRequirement` being handled. Frameworks such as MVC or SignalR are free to add any object to the `Resource` property on the `AuthorizationHandlerContext` to pass extra information.

When using endpoint routing, authorization is typically handled by the Authorization Middleware. In this case, the `Resource` property is an instance of HttpContext. The context can be used to access the current endpoint, which can be used to probe the underlying resource to which you're routing. For example:

C#

```
if (context.Resource is HttpContext httpContext)
{
    var endpoint = httpContext.GetEndpoint();
    var actionDescriptor =
endpoint.Metadata.GetMetadata<ControllerActionDescriptor>();
    ...
}
```

With traditional routing, or when authorization happens as part of MVC's authorization filter, the value of `Resource` is an AuthorizationFilterContext instance. This property provides access to `HttpContext`, `RouteData`, and everything else provided by MVC and Razor Pages.

The use of the `Resource` property is framework-specific. Using information in the `Resource` property limits your authorization policies to particular frameworks. Cast the `Resource` property using the `is` keyword, and then confirm the cast has succeeded to ensure your code doesn't crash with an `InvalidCastException` when run on other frameworks:

C#

```
// Requires the following import:
//     using Microsoft.AspNetCore.Mvc.Filters;
if (context.Resource is AuthorizationFilterContext mvcContext)
{
    // Examine MVC-specific things like routing data.
}
```

Globally require all users to be authenticated

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

Authorization with external service sample

The sample code on AspNetCore.Docs.Samples shows how to implement additional authorization requirements with an external authorization service. The sample `Contoso.API` project is secured with Azure AD. An additional authorization check from the `Contoso.Security.API` project returns a payload describing whether the `Contoso.API` client app can invoke the `GetWeather` API.

Configure the sample

- Create an application registration in your Microsoft Entra ID tenant:
- Assign it an AppRole.
- Under API permissions, add the AppRole as a permission and grant Admin consent. Note that in this setup, this app registration represents both the API and the client invoking the API. If you like, you can create two app registrations. If you are using this setup, be sure to only perform the API permissions, add AppRole as a permission step for only the client. Only the client app registration requires a client secret to be generated.
- Configure the `Contoso.API` project with the following settings:

C#

```
{
  "AzureAd": {
    "Instance": "https://login.microsoftonline.com/",
    "Domain": "<Tenant name from AAD properties>.onmicrosoft.com",
    "TenantId": "<Tenant Id from AAD properties>",
    "ClientId": "<Client Id from App Registration representing the API>"
```

```
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

- Configure `Contoso.Security.API` with the following settings:

C#

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "AllowedClients": [
    "<Use the appropriate Client Id representing the Client calling the
API>"
  ]
}
```

- Open the ContosoAPI.collection.json file and configure an environment with the following:
  - `ClientId`: Client Id from app registration representing the client calling the API.
  - `clientSecret`: Client Secret from app registration representing the client calling the API.
  - `TenantId`: Tenant Id from AAD properties

- Extract the commands from the `ContosoAPI.collection.json` file and use them to construct cURL commands to test the app.

- Run the solution and use cURL to invoke the API. You can add breakpoints in the `Contoso.Security.API.SecurityPolicyController` and observe the client Id is being passed in that is used to assert whether it is allowed to Get Weather.

Additional resources

- Quickstart: Configure an application to expose a web API
- AspNetCore.Docs.Samples code

Custom Authorization Policy Providers using IAuthorizationPolicyProvider in ASP.NET Core

Article • 06/03/2022

By Mike Rousos

Typically when using policy-based authorization, policies are registered by calling `AuthorizationOptions.AddPolicy` as part of authorization service configuration. In some scenarios, it may not be possible (or desirable) to register all authorization policies in this way. In those cases, you can use a custom IAuthorizationPolicyProvider to control how authorization policies are supplied.

Examples of scenarios where a custom IAuthorizationPolicyProvider may be useful include:

- Using an external service to provide policy evaluation.
- Using a large range of policies (for different room numbers or ages, for example), so it doesn't make sense to add each individual authorization policy with an `AuthorizationOptions.AddPolicy` call.
- Creating policies at runtime based on information in an external data source (like a database) or determining authorization requirements dynamically through another mechanism.

View or download sample code from the AspNetCore GitHub repository. Download the dotnet/AspNetCore repository ZIP file. Unzip the file. Navigate to the *src/Security/samples/CustomPolicyProvider* project folder.

Customize policy retrieval

ASP.NET Core apps use an implementation of the `IAuthorizationPolicyProvider` interface to retrieve authorization policies. By default, DefaultAuthorizationPolicyProvider is registered and used. `DefaultAuthorizationPolicyProvider` returns policies from the `AuthorizationOptions` provided in an `IServiceCollection.AddAuthorization` call.

Customize this behavior by registering a different `IAuthorizationPolicyProvider` implementation in the app's dependency injection container.

The `IAuthorizationPolicyProvider` interface contains three APIs:

- GetPolicyAsync returns an authorization policy for a given name.
- GetDefaultPolicyAsync returns the default authorization policy (the policy used for `[Authorize]` attributes without a policy specified).
- GetFallbackPolicyAsync returns the fallback authorization policy (the policy used by the Authorization Middleware when no policy is specified).

By implementing these APIs, you can customize how authorization policies are provided.

Parameterized authorize attribute example

One scenario where `IAuthorizationPolicyProvider` is useful is enabling custom `[Authorize]` attributes whose requirements depend on a parameter. For example, in policy-based authorization documentation, an age-based ("AtLeast21") policy was used as a sample. If different controller actions in an app should be made available to users of *different* ages, it might be useful to have many different age-based policies. Instead of registering all the different age-based policies that the application will need in `AuthorizationOptions`, you can generate the policies dynamically with a custom `IAuthorizationPolicyProvider`. To make using the policies easier, you can annotate actions with custom authorization attribute like `[MinimumAgeAuthorize(20)]`.

Custom Authorization attributes

Authorization policies are identified by their names. The custom `MinimumAgeAuthorizeAttribute` described previously needs to map arguments into a string that can be used to retrieve the corresponding authorization policy. You can do this by deriving from `AuthorizeAttribute` and making the `Age` property wrap the `AuthorizeAttribute.Policy` property.

C#

```
internal class MinimumAgeAuthorizeAttribute : AuthorizeAttribute
{
    const string POLICY_PREFIX = "MinimumAge";

    public MinimumAgeAuthorizeAttribute(int age) => Age = age;

    // Get or set the Age property by manipulating the underlying Policy
property
    public int Age
    {
        get
```

```
        {
            if (int.TryParse(Policy.Substring(POLICY_PREFIX.Length), out var
age))
            {
                return age;
            }
            return default(int);
        }
        set
        {
            Policy = $"{POLICY_PREFIX}{value.ToString()}";
        }
    }
}
```

This attribute type has a `Policy` string based on the hard-coded prefix (`"MinimumAge"`) and an integer passed in via the constructor.

You can apply it to actions in the same way as other `Authorize` attributes except that it takes an integer as a parameter.

C#

```
[MinimumAgeAuthorize(10)]
public IActionResult RequiresMinimumAge10()
```

Custom IAuthorizationPolicyProvider

The custom `MinimumAgeAuthorizeAttribute` makes it easy to request authorization policies for any minimum age desired. The next problem to solve is making sure that authorization policies are available for all of those different ages. This is where an `IAuthorizationPolicyProvider` is useful.

When using `MinimumAgeAuthorizationAttribute`, the authorization policy names will follow the pattern `"MinimumAge" + Age`, so the custom `IAuthorizationPolicyProvider` should generate authorization policies by:

- Parsing the age from the policy name.
- Using `AuthorizationPolicyBuilder` to create a new `AuthorizationPolicy`
- In this and following examples it will be assumed that the user is authenticated via a cookie. The `AuthorizationPolicyBuilder` should either be constructed with at least one authorization scheme name or always succeed. Otherwise there is no information on how to provide a challenge to the user and an exception will be thrown.

- Adding requirements to the policy based on the age with `AuthorizationPolicyBuilder.AddRequirements`. In other scenarios, you might use `RequireClaim`, `RequireRole`, or `RequireUserName` instead.

C#

```
internal class MinimumAgePolicyProvider : IAuthorizationPolicyProvider
{
    const string POLICY_PREFIX = "MinimumAge";

    // Policies are looked up by string name, so expect 'parameters' (like
age)
    // to be embedded in the policy names. This is abstracted away from
developers
    // by the more strongly-typed attributes derived from AuthorizeAttribute
    // (like [MinimumAgeAuthorize()] in this sample)
    public Task<AuthorizationPolicy> GetPolicyAsync(string policyName)
    {
        if (policyName.StartsWith(POLICY_PREFIX,
StringComparison.OrdinalIgnoreCase) &&
            int.TryParse(policyName.Substring(POLICY_PREFIX.Length), out var
age))
        {
            var policy = new
AuthorizationPolicyBuilder(CookieAuthenticationDefaults.AuthenticationScheme
);
            policy.AddRequirements(new MinimumAgeRequirement(age));
            return Task.FromResult(policy.Build());
        }

        return Task.FromResult<AuthorizationPolicy>(null);
    }
}
```

Multiple authorization policy providers

When using custom `IAuthorizationPolicyProvider` implementations, keep in mind that ASP.NET Core only uses one instance of `IAuthorizationPolicyProvider`. If a custom provider isn't able to provide authorization policies for all policy names that will be used, it should defer to a backup provider.

For example, consider an application that needs both custom age policies and more traditional role-based policy retrieval. Such an app could use a custom authorization policy provider that:

- Attempts to parse policy names.

- Calls into a different policy provider (like `DefaultAuthorizationPolicyProvider`) if the policy name doesn't contain an age.

The example `IAuthorizationPolicyProvider` implementation shown above can be updated to use the `DefaultAuthorizationPolicyProvider` by creating a backup policy provider in its constructor (to be used in case the policy name doesn't match its expected pattern of 'MinimumAge' + age).

C#

```
private DefaultAuthorizationPolicyProvider BackupPolicyProvider { get; }

public MinimumAgePolicyProvider(IOptions<AuthorizationOptions> options)
{
    // ASP.NET Core only uses one authorization policy provider, so if the
custom implementation
    // doesn't handle all policies it should fall back to an alternate
provider.
    BackupPolicyProvider = new DefaultAuthorizationPolicyProvider(options);
}
```

Then, the `GetPolicyAsync` method can be updated to use the `BackupPolicyProvider` instead of returning null:

C#

```
...
return BackupPolicyProvider.GetPolicyAsync(policyName);
```

Default policy

In addition to providing named authorization policies, a custom `IAuthorizationPolicyProvider` needs to implement `GetDefaultPolicyAsync` to provide an authorization policy for `[Authorize]` attributes without a policy name specified.

In many cases, this authorization attribute only requires an authenticated user, so you can make the necessary policy with a call to `RequireAuthenticatedUser`:

C#

```
public Task<AuthorizationPolicy> GetDefaultPolicyAsync() =>
    Task.FromResult(new
AuthorizationPolicyBuilder(CookieAuthenticationDefaults.AuthenticationScheme
).RequireAuthenticatedUser().Build());
```

As with all aspects of a custom `IAuthorizationPolicyProvider`, you can customize this, as needed. In some cases, it may be desirable to retrieve the default policy from a fallback `IAuthorizationPolicyProvider`.

Fallback policy

A custom `IAuthorizationPolicyProvider` can optionally implement `GetFallbackPolicyAsync` to provide a policy that's used when combining policies and when no policies are specified. If `GetFallbackPolicyAsync` returns a non-null policy, the returned policy is used by the Authorization Middleware when no policies are specified for the request.

If no fallback policy is required, the provider can return `null` or defer to the fallback provider:

```
C#
```

```
public Task<AuthorizationPolicy> GetFallbackPolicyAsync() =>
    Task.FromResult<AuthorizationPolicy>(null);
```

Use a custom IAuthorizationPolicyProvider

To use custom policies from an `IAuthorizationPolicyProvider`, you ***must***:

- Register the appropriate `AuthorizationHandler` types with dependency injection (described in policy-based authorization), as with all policy-based authorization scenarios.
- Register the custom `IAuthorizationPolicyProvider` type in the app's dependency injection service collection in `Startup.ConfigureServices` to replace the default policy provider.

  ```
  C#
  ```

  ```
  services.AddSingleton<IAuthorizationPolicyProvider,
  MinimumAgePolicyProvider>();
  ```

A complete custom `IAuthorizationPolicyProvider` sample is available in the dotnet/aspnetcore GitHub repository.

Customize the behavior of `AuthorizationMiddleware`

Article • 06/03/2022

Apps can register an IAuthorizationMiddlewareResultHandler to customize how AuthorizationMiddleware handles authorization results. Apps can use `IAuthorizationMiddlewareResultHandler` to:

- Return customized responses.
- Enhance the default challenge or forbid responses.

The following code shows an example implementation of `IAuthorizationMiddlewareResultHandler` that returns a custom response for specific authorization failures:

C#

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Authorization.Policy;

public class SampleAuthorizationMiddlewareResultHandler :
IAuthorizationMiddlewareResultHandler
{
    private readonly AuthorizationMiddlewareResultHandler defaultHandler =
new();

    public async Task HandleAsync(
        RequestDelegate next,
        HttpContext context,
        AuthorizationPolicy policy,
        PolicyAuthorizationResult authorizeResult)
    {
        // If the authorization was forbidden and the resource had a
specific requirement,
        // provide a custom 404 response.
        if (authorizeResult.Forbidden
            && authorizeResult.AuthorizationFailure!.FailedRequirements
                .OfType<Show404Requirement>().Any())
        {
            // Return a 404 to make it appear as if the resource doesn't
exist.
            context.Response.StatusCode = StatusCodes.Status404NotFound;
            return;
        }

        // Fall back to the default implementation.
        await defaultHandler.HandleAsync(next, context, policy,
authorizeResult);
```

```
    }
}

public class Show404Requirement : IAuthorizationRequirement { }
```

Register this implementation of `IAuthorizationMiddlewareResultHandler` in `Program.cs`:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddSingleton<
    IAuthorizationMiddlewareResultHandler,
SampleAuthorizationMiddlewareResultHandler>();

var app = builder.Build();
```

Dependency injection in requirement handlers in ASP.NET Core

Article • 06/03/2022

Authorization handlers must be registered in the service collection during configuration using dependency injection.

Suppose you had a repository of rules you wanted to evaluate inside an authorization handler and that repository was registered in the service collection. Authorization resolves and injects that into the constructor.

For example, to use the .NET logging infrastructure, inject ILoggerFactory into the handler, as shown in the following example:

C#

```
public class SampleAuthorizationHandler :
AuthorizationHandler<SampleRequirement>
{
    private readonly ILogger _logger;

    public SampleAuthorizationHandler(ILoggerFactory loggerFactory)
        => _logger = loggerFactory.CreateLogger(GetType().FullName);

    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context, SampleRequirement requirement)
    {
        _logger.LogInformation("Inside my handler");

        // ...

        return Task.CompletedTask;
    }
}
```

The preceding handler can be registered with any service lifetime. The following code uses AddSingleton to register the preceding handler:

C#

```
builder.Services.AddSingleton<IAuthorizationHandler,
SampleAuthorizationHandler>();
```

An instance of the handler is created when the app starts, and DI injects the registered `ILoggerFactory` into its constructor.

> **Note**
>
> Don't register authorization handlers that use Entity Framework (EF) as singletons.

Resource-based authorization in ASP.NET Core

Article • 06/18/2024

Authorization approach depends on the resource. For example, only the author of a document is authorized to update the document. Consequently, the document must be retrieved from the data store before authorization evaluation can occur.

Attribute evaluation occurs before data binding and before execution of the page handler or action that loads the document. For these reasons, declarative authorization with an `[Authorize]` attribute doesn't suffice. Instead, you can invoke a custom authorization method—a style known as *imperative authorization*.

View or download sample code (how to download).

Create an ASP.NET Core app with user data protected by authorization contains a sample app that uses resource-based authorization.

Use imperative authorization

Authorization is implemented as an IAuthorizationService service and is registered in the service collection at application startup. The service is made available via dependency injection to page handlers or actions.

C#

```
public class DocumentController : Controller
{
    private readonly IAuthorizationService _authorizationService;
    private readonly IDocumentRepository _documentRepository;

    public DocumentController(IAuthorizationService authorizationService,
                              IDocumentRepository documentRepository)
    {
        _authorizationService = authorizationService;
        _documentRepository = documentRepository;
    }
```

`IAuthorizationService` has two `AuthorizeAsync` method overloads: one accepting the resource and the policy name and the other accepting the resource and a list of requirements to evaluate.

C#

```
Task<AuthorizationResult> AuthorizeAsync(ClaimsPrincipal user,
                          object resource,
                          IEnumerable<IAuthorizationRequirement>
requirements);
Task<AuthorizationResult> AuthorizeAsync(ClaimsPrincipal user,
                          object resource,
                          string policyName);
```

In the following example, the resource to be secured is loaded into a custom `Document` object. An `AuthorizeAsync` overload is invoked to determine whether the current user is allowed to edit the provided document. A custom "EditPolicy" authorization policy is factored into the decision. See Custom policy-based authorization for more on creating authorization policies.

> **Note**
>
> The following code samples assume authentication has run and set the `User` property.

C#

```
public async Task<IActionResult> OnGetAsync(Guid documentId)
{
    Document = _documentRepository.Find(documentId);

    if (Document == null)
    {
        return new NotFoundResult();
    }

    var authorizationResult = await _authorizationService
            .AuthorizeAsync(User, Document, "EditPolicy");

    if (authorizationResult.Succeeded)
    {
        return Page();
    }
    else if (User.Identity.IsAuthenticated)
    {
        return new ForbidResult();
    }
    else
    {
        return new ChallengeResult();
    }
}
```

Write a resource-based handler

Writing a handler for resource-based authorization isn't much different than writing a plain requirements handler. Create a custom requirement class, and implement a requirement handler class. For more information on creating a requirement class, see Requirements.

The handler class specifies both the requirement and resource type. For example, a handler utilizing a `SameAuthorRequirement` and a `Document` resource follows:

C#

```
public class DocumentAuthorizationHandler :
    AuthorizationHandler<SameAuthorRequirement, Document>
{
    protected override Task
HandleRequirementAsync(AuthorizationHandlerContext context,
                                                    SameAuthorRequirement
requirement,
                                                    Document resource)
    {
        if (context.User.Identity?.Name == resource.Author)
        {
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }
}

public class SameAuthorRequirement : IAuthorizationRequirement { }
```

In the preceding example, imagine that `SameAuthorRequirement` is a special case of a more generic `SpecificAuthorRequirement` class. The `SpecificAuthorRequirement` class (not shown) contains a `Name` property representing the name of the author. The `Name` property could be set to the current user.

Register the requirement and handler in `Program.cs`:

C#

```
builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();
```

```
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("EditPolicy", policy =>
        policy.Requirements.Add(new SameAuthorRequirement()));
});

builder.Services.AddSingleton<IAuthorizationHandler,
DocumentAuthorizationHandler>();
builder.Services.AddSingleton<IAuthorizationHandler,
DocumentAuthorizationCrudHandler>();
builder.Services.AddScoped<IDocumentRepository, DocumentRepository>();
```

Operational requirements

If you're making decisions based on the outcomes of CRUD (Create, Read, Update, Delete) operations, use the OperationAuthorizationRequirement helper class. This class enables you to write a single handler instead of an individual class for each operation type. To use it, provide some operation names:

C#

```
public static class Operations
{
    public static OperationAuthorizationRequirement Create =
        new OperationAuthorizationRequirement { Name = nameof(Create) };
    public static OperationAuthorizationRequirement Read =
        new OperationAuthorizationRequirement { Name = nameof(Read) };
    public static OperationAuthorizationRequirement Update =
        new OperationAuthorizationRequirement { Name = nameof(Update) };
    public static OperationAuthorizationRequirement Delete =
        new OperationAuthorizationRequirement { Name = nameof(Delete) };
}
```

The handler is implemented as follows, using an `OperationAuthorizationRequirement` requirement and a `Document` resource:

C#

```
public class DocumentAuthorizationCrudHandler :
    AuthorizationHandler<OperationAuthorizationRequirement, Document>
{
    protected override Task
HandleRequirementAsync(AuthorizationHandlerContext context,

OperationAuthorizationRequirement requirement,
                                                    Document resource)
    {
        if (context.User.Identity?.Name == resource.Author &&
            requirement.Name == Operations.Read.Name)
```

```
        {
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }
}
```

The preceding handler validates the operation using the resource, the user's identity, and the requirement's `Name` property.

Challenge and forbid with an operational resource handler

This section shows how the challenge and forbid action results are processed and how challenge and forbid differ.

To call an operational resource handler, specify the operation when invoking `AuthorizeAsync` in your page handler or action. The following example determines whether the authenticated user is permitted to view the provided document.

> ⓘ Note
>
> The following code samples assume authentication has run and set the `User` property.

C#

```
public async Task<IActionResult> OnGetAsync(Guid documentId)
{
    Document = _documentRepository.Find(documentId);

    if (Document == null)
    {
        return new NotFoundResult();
    }

    var authorizationResult = await _authorizationService
            .AuthorizeAsync(User, Document, Operations.Read);

    if (authorizationResult.Succeeded)
    {
        return Page();
    }
    else if (User.Identity.IsAuthenticated)
    {
```

```
            return new ForbidResult();
        }
        else
        {
            return new ChallengeResult();
        }
    }
```

If authorization succeeds, the page for viewing the document is returned. If authorization fails but the user is authenticated, returning `ForbidResult` informs any authentication middleware that authorization failed. A `ChallengeResult` is returned when authentication must be performed. For interactive browser clients, it may be appropriate to redirect the user to a login page.

View-based authorization in ASP.NET Core MVC

Article • 09/27/2024

A developer often wants to show, hide, or otherwise modify a UI based on the current user identity. You can access the authorization service within MVC views via dependency injection. To inject the authorization service into a Razor view, use the `@inject` directive:

CSHTML

```
@using Microsoft.AspNetCore.Authorization
@inject IAuthorizationService AuthorizationService
```

If you want the authorization service in every view, place the `@inject` directive into the `_ViewImports.cshtml` file of the `Views` directory. For more information, see Dependency injection into views.

Use the injected authorization service to invoke `AuthorizeAsync` in exactly the same way you would check during resource-based authorization:

CSHTML

```
@if ((await AuthorizationService.AuthorizeAsync(User,
"PolicyName")).Succeeded)
{
    <p>This paragraph is displayed because you fulfilled PolicyName.</p>
}
```

In some cases, the resource will be your view model. Invoke `AuthorizeAsync` in exactly the same way you would check during resource-based authorization:

CSHTML

```
@if ((await AuthorizationService.AuthorizeAsync(User, Model,
Operations.Edit)).Succeeded)
{
    <p><a class="btn btn-default" role="button"
        href="@Url.Action("Edit", "Document", new { id = Model.Id
})">Edit</a></p>
}
```

In the preceding code, the model is passed as a resource the policy evaluation should take into consideration.

> **⚠ Warning**
>
> Don't rely on toggling visibility of your app's UI elements as the sole authorization check. Hiding a UI element may not completely prevent access to its associated controller action. For example, consider the button in the preceding code snippet. A user can invoke the `Edit` action method if they know the relative resource URL is `/Document/Edit/1`. For this reason, the `Edit` action method should perform its own authorization check.

Authorize with a specific scheme in ASP.NET Core

Article • 06/03/2022

For an introduction to authentication schemes in ASP.NET Core, see Authentication scheme.

In some scenarios, such as Single Page Applications (SPAs), it's common to use multiple authentication methods. For example, the app may use cookie-based authentication to log in and JWT bearer authentication for JavaScript requests. In some cases, the app may have multiple instances of an authentication handler. For example, two cookie handlers where one contains a basic identity and one is created when a multi-factor authentication (MFA) has been triggered. MFA may be triggered because the user requested an operation that requires extra security. For more information on enforcing MFA when a user requests a resource that requires MFA, see the GitHub issue Protect section with MFA.

An authentication scheme is named when the authentication service is configured during authentication. For example:

C#

```
using Microsoft.AspNetCore.Authentication;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthentication()
        .AddCookie(options =>
        {
            options.LoginPath = "/Account/Unauthorized/";
            options.AccessDeniedPath = "/Account/Forbidden/";
        })
        .AddJwtBearer(options =>
        {
            options.Audience = "http://localhost:5001/";
            options.Authority = "http://localhost:5000/";
        });

builder.Services.AddAuthentication()
        .AddIdentityServerJwt();

builder.Services.AddControllersWithViews();
builder.Services.AddRazorPages();

var app = builder.Build();
```

```
if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseHsts();
}

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

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

app.MapDefaultControllerRoute();
app.MapRazorPages();

app.MapFallbackToFile("index.html");

app.Run();
```

In the preceding code, two authentication handlers have been added: one for cookies and one for bearer.

> **Note**
>
> Specifying the default scheme results in the `HttpContext.User` property being set to that identity. If that behavior isn't desired, disable it by invoking the parameterless form of `AddAuthentication`.

Selecting the scheme with the Authorize attribute

At the point of authorization, the app indicates the handler to be used. Select the handler with which the app will authorize by passing a comma-delimited list of authentication schemes to `[Authorize]`. The [Authorize] attribute specifies the authentication scheme or schemes to use regardless of whether a default is configured. For example:

```
C#
```

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Mvc;

namespace AuthScheme.Controllers;

[Authorize(AuthenticationSchemes = AuthSchemes)]
public class MixedController : Controller
{
    private const string AuthSchemes =
        CookieAuthenticationDefaults.AuthenticationScheme + "," +
        JwtBearerDefaults.AuthenticationScheme;
    public ContentResult Index() => Content(MyWidgets.GetMyContent());

}
```

In the preceding example, both the cookie and bearer handlers run and have a chance to create and append an identity for the current user. By specifying a single scheme only, the corresponding handler runs:

C#

```
[Authorize(AuthenticationSchemes=JwtBearerDefaults.AuthenticationScheme)]
public class Mixed2Controller : Controller
{
    public ContentResult Index() => Content(MyWidgets.GetMyContent());
}
```

In the preceding code, only the handler with the "Bearer" scheme runs. Any cookie-based identities are ignored.

Selecting the scheme with policies

If you prefer to specify the desired schemes in policy, you can set the AuthenticationSchemes collection when adding a policy:

C#

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authentication.JwtBearer;

var builder = WebApplication.CreateBuilder(args);
```

```
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("Over18", policy =>
    {

policy.AuthenticationSchemes.Add(JwtBearerDefaults.AuthenticationScheme);
        policy.RequireAuthenticatedUser();
        policy.Requirements.Add(new MinimumAgeRequirement(18));
    });
});
builder.Services.AddAuthentication()
                .AddIdentityServerJwt();

builder.Services.AddControllersWithViews();
builder.Services.AddRazorPages();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseHsts();
}

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

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

app.MapDefaultControllerRoute();
app.MapRazorPages();

app.MapFallbackToFile("index.html");

app.Run();
```

In the preceding example, the "Over18" policy only runs against the identity created by the "Bearer" handler. Use the policy by setting the `[Authorize]` attribute's `Policy` property:

C#

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace AuthScheme.Controllers;
```