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

10/30/2020 • 16 minutes to read •      +1

In this article

An ASP.NET Core app can establish additional claims and tokens from external authentication providers, such as Facebook, Google, Microsoft, and Twitter. Each provider reveals different information about users on its platform, but the pattern for receiving and transforming user data into additional claims is the same.

View or download sample code (how to download)

Prerequisites

Decide which external authentication providers to support in the app. For each provider, register the app and obtain a client ID and client secret. For more information, see Facebook, Google, and external provider authentication in ASP.NET Core. The sample app uses the Google authentication provider.

Set the client ID and client secret

The OAuth authentication provider establishes a trust relationship with an app using a client ID and client secret. Client ID and client secret values are created for the app by the

external authentication provider when the app is registered with the provider. Each external provider that the app uses must be configured independently with the provider's client ID and client secret. For more information, see the external authentication provider topics that apply to your scenario:

- Facebook authentication
- Google authentication
- Microsoft authentication
- Twitter authentication
- Other authentication providers
- OpenIdConnect

The sample app configures the Google authentication provider with a client ID and client secret provided by Google:

C# Copy

```
services.AddAuthentication().AddGoogle(options =>
{
    // Provide the Google Client ID
    options.ClientId = "XXXXXXXXXXXXXXX.apps.googleusercontent.com";
    // Register with User Secrets using:
    // dotnet user-secrets set "Authentication:Google:ClientId" "
{Client ID}"

    // Provide the Google Client Secret
    options.ClientSecret = "{Client Secret}";
    // Register with User Secrets using:
    // dotnet user-secrets set "Authentication:Google:ClientSecret" "
{Client Secret}"

    options.ClaimActions.MapJsonKey("urn:google:picture", "picture",
"url");
    options.ClaimActions.MapJsonKey("urn:google:locale", "locale",
"string");
    options.SaveTokens = true;

    options.Events.OnCreatingTicket = ctx =>
    {
        List<AuthenticationToken> tokens =
ctx.Properties.GetTokens().ToList();

        tokens.Add(new AuthenticationToken()
        {
            Name = "TicketCreated",
            Value = DateTime.UtcNow.ToString()
        });
```

```
            ctx.Properties.StoreTokens(tokens);

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

Establish the authentication scope

Specify the list of permissions to retrieve from the provider by specifying the Scope. Authentication scopes for common external providers appear in the following table.

Provider	Scope
Facebook	`https://www.facebook.com/dialog/oauth`
Google	`profile`, `email`, `openid`
Microsoft	`https://login.microsoftonline.com/common/oauth2/v2.0/authorize`
Twitter	`https://api.twitter.com/oauth/authenticate`

In the sample app, Google's `profile`, `email`, and `openid` scopes are automatically added by the framework when AddGoogle is called on the AuthenticationBuilder. If the app requires additional scopes, add them to the options. In the following example, the Google `https://www.googleapis.com/auth/user.birthday.read` scope is added to retrieve a user's birthday:

C# Copy

```
options.Scope.Add("https://www.googleapis.com/auth/user.birthday.read");
```

Map user data keys and create claims

In the provider's options, specify a MapJsonKey or MapJsonSubKey for each key/subkey in the external provider's JSON user data for the app identity to read on sign in. For more information on claim types, see ClaimTypes.

The sample app creates locale (`urn:google:locale`) and picture (`urn:google:picture`) claims from the `locale` and `picture` keys in Google user data:

C# Copy

```
services.AddAuthentication().AddGoogle(options =>
{
    // Provide the Google Client ID
    options.ClientId = "XXXXXXXXXXXXXXX.apps.googleusercontent.com";
    // Register with User Secrets using:
    // dotnet user-secrets set "Authentication:Google:ClientId" "
{Client ID}"

    // Provide the Google Client Secret
    options.ClientSecret = "{Client Secret}";
    // Register with User Secrets using:
    // dotnet user-secrets set "Authentication:Google:ClientSecret" "
{Client Secret}"

    options.ClaimActions.MapJsonKey("urn:google:picture", "picture",
"url");
    options.ClaimActions.MapJsonKey("urn:google:locale", "locale",
"string");
    options.SaveTokens = true;

    options.Events.OnCreatingTicket = ctx =>
    {
        List<AuthenticationToken> tokens =
ctx.Properties.GetTokens().ToList();

        tokens.Add(new AuthenticationToken()
        {
            Name = "TicketCreated",
            Value = DateTime.UtcNow.ToString()
        });

        ctx.Properties.StoreTokens(tokens);

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

In `Microsoft.AspNetCore.Identity.UI.Pages.Account.Internal.ExternalLoginModel.OnPostConfirmationAsync`, an IdentityUser (`ApplicationUser`) is signed into the app with SignInAsync. During the sign in process, the UserManager<TUser> can store an `ApplicationUser` claims for user data available from the Principal.

In the sample app, `OnPostConfirmationAsync` (*Account/ExternalLogin.cshtml.cs*) establishes the locale (`urn:google:locale`) and picture (`urn:google:picture`) claims for the signed in `ApplicationUser`, including a claim for GivenName:

C# Copy

```
public async Task<IActionResult> OnPostConfirmationAsync(string retur-
nUrl = null)
{
    returnUrl = returnUrl ?? Url.Content("~/");
    // Get the information about the user from the external login
provider
    var info = await _signInManager.GetExternalLoginInfoAsync();

    if (info == null)
    {
        ErrorMessage =
            "Error loading external login information during
confirmation.";

        return RedirectToPage("./Login", new { ReturnUrl = returnUrl
});
    }

    if (ModelState.IsValid)
    {
        var user = new IdentityUser
        {
            UserName = Input.Email,
            Email = Input.Email
        };

        var result = await _userManager.CreateAsync(user);

        if (result.Succeeded)
        {
            result = await _userManager.AddLoginAsync(user, info);

            if (result.Succeeded)
            {
                // If they exist, add claims to the user for:
                //    Given (first) name
                //    Locale
                //    Picture
```

```
                if (info.Principal.HasClaim(c => c.Type ==
ClaimTypes.GivenName))
                {
                    await _userManager.AddClaimAsync(user,

info.Principal.FindFirst(ClaimTypes.GivenName));
                }

                if (info.Principal.HasClaim(c => c.Type ==
"urn:google:locale"))
                {
                    await _userManager.AddClaimAsync(user,
                        info.Principal.FindFirst("urn:google:locale"));
                }

                if (info.Principal.HasClaim(c => c.Type ==
"urn:google:picture"))
                {
                    await _userManager.AddClaimAsync(user,

info.Principal.FindFirst("urn:google:picture"));
                }

                // Include the access token in the properties
                var props = new AuthenticationProperties();
                props.StoreTokens(info.AuthenticationTokens);
                props.IsPersistent = true;

                await _signInManager.SignInAsync(user, props);

                _logger.LogInformation(
                    "User created an account using {Name} provider.",
                    info.LoginProvider);

                return LocalRedirect(returnUrl);
            }
        }

        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(string.Empty, error.Description);
        }
    }

    LoginProvider = info.LoginProvider;
    ReturnUrl = returnUrl;
    return Page();
}
```

By default, a user's claims are stored in the authentication cookie. If the authentication cookie is too large, it can cause the app to fail because:

- The browser detects that the cookie header is too long.
- The overall size of the request is too large.

If a large amount of user data is required for processing user requests:

- Limit the number and size of user claims for request processing to only what the app requires.
- Use a custom ITicketStore for the Cookie Authentication Middleware's SessionStore to store identity across requests. Preserve large quantities of identity information on the server while only sending a small session identifier key to the client.

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.

The sample app sets the value of `SaveTokens` to `true` in GoogleOptions:

C# Copy

```
services.AddAuthentication().AddGoogle(options =>
{
    // Provide the Google Client ID
    options.ClientId = "XXXXXXXXXXXXXXX.apps.googleusercontent.com";
    // Register with User Secrets using:
    // dotnet user-secrets set "Authentication:Google:ClientId" "
{Client ID}"

    // Provide the Google Client Secret
    options.ClientSecret = "{Client Secret}";
    // Register with User Secrets using:
    // dotnet user-secrets set "Authentication:Google:ClientSecret" "
{Client Secret}"

    options.ClaimActions.MapJsonKey("urn:google:picture", "picture",
"url");
    options.ClaimActions.MapJsonKey("urn:google:locale", "locale",
"string");
    options.SaveTokens = true;

    options.Events.OnCreatingTicket = ctx =>
```

```
    {
        List<AuthenticationToken> tokens =
ctx.Properties.GetTokens().ToList();

        tokens.Add(new AuthenticationToken()
        {
            Name = "TicketCreated",
            Value = DateTime.UtcNow.ToString()
        });

        ctx.Properties.StoreTokens(tokens);

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

When `OnPostConfirmationAsync` executes, store the access token (ExternalLoginInfo.AuthenticationTokens) from the external provider in the `ApplicationUser`'s `AuthenticationProperties`.

The sample app saves the access token in `OnPostConfirmationAsync` (new user registration) and `OnGetCallbackAsync` (previously registered user) in *Account/ExternalLogin.cshtml.cs*:

C# Copy

```
public async Task<IActionResult> OnPostConfirmationAsync(string retur-
nUrl = null)
{
    returnUrl = returnUrl ?? Url.Content("~/");
    // Get the information about the user from the external login
provider
    var info = await _signInManager.GetExternalLoginInfoAsync();

    if (info == null)
    {
        ErrorMessage =
            "Error loading external login information during
confirmation.";

        return RedirectToPage("./Login", new { ReturnUrl = returnUrl
});
    }

    if (ModelState.IsValid)
    {
        var user = new IdentityUser
        {
```

```
            UserName = Input.Email,
            Email = Input.Email
        };

        var result = await _userManager.CreateAsync(user);

        if (result.Succeeded)
        {
            result = await _userManager.AddLoginAsync(user, info);

            if (result.Succeeded)
            {
                // If they exist, add claims to the user for:
                //    Given (first) name
                //    Locale
                //    Picture
                if (info.Principal.HasClaim(c => c.Type ==
ClaimTypes.GivenName))
                {
                    await _userManager.AddClaimAsync(user,

info.Principal.FindFirst(ClaimTypes.GivenName));
                }

                if (info.Principal.HasClaim(c => c.Type ==
"urn:google:locale"))
                {
                    await _userManager.AddClaimAsync(user,
                        info.Principal.FindFirst("urn:google:locale"));
                }

                if (info.Principal.HasClaim(c => c.Type ==
"urn:google:picture"))
                {
                    await _userManager.AddClaimAsync(user,

info.Principal.FindFirst("urn:google:picture"));
                }

                // Include the access token in the properties
                var props = new AuthenticationProperties();
                props.StoreTokens(info.AuthenticationTokens);
                props.IsPersistent = true;

                await _signInManager.SignInAsync(user, props);

                _logger.LogInformation(
                    "User created an account using {Name} provider.",
                    info.LoginProvider);

                return LocalRedirect(returnUrl);
```

```
            }
        }

        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(string.Empty, error.Description);
        }
    }

    LoginProvider = info.LoginProvider;
    ReturnUrl = returnUrl;
    return Page();
}
```

How to add additional custom tokens

To demonstrate how to add a custom token, which is stored as part of `SaveTokens`, the sample app adds an AuthenticationToken with the current DateTime for an AuthenticationToken.Name of `TicketCreated`:

C# Copy

```
services.AddAuthentication().AddGoogle(options =>
{
    // Provide the Google Client ID
    options.ClientId = "XXXXXXXXXXXXXXX.apps.googleusercontent.com";
    // Register with User Secrets using:
    // dotnet user-secrets set "Authentication:Google:ClientId" "
{Client ID}"

    // Provide the Google Client Secret
    options.ClientSecret = "{Client Secret}";
    // Register with User Secrets using:
    // dotnet user-secrets set "Authentication:Google:ClientSecret" "
{Client Secret}"

    options.ClaimActions.MapJsonKey("urn:google:picture", "picture",
"url");
    options.ClaimActions.MapJsonKey("urn:google:locale", "locale",
"string");
    options.SaveTokens = true;
```

```
    options.Events.OnCreatingTicket = ctx =>
    {
        List<AuthenticationToken> tokens =
ctx.Properties.GetTokens().ToList();

        tokens.Add(new AuthenticationToken()
        {
            Name = "TicketCreated",
            Value = DateTime.UtcNow.ToString()
        });

        ctx.Properties.StoreTokens(tokens);

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

Creating and adding claims

The framework provides common actions and extension methods for creating and adding claims to the collection. For more information, see the ClaimActionCollectionMapExtensions and ClaimActionCollectionUniqueExtensions.

Users can define custom actions by deriving from ClaimAction and implementing the abstract Run method.

For more information, see Microsoft.AspNetCore.Authentication.OAuth.Claims.

Removal of claim actions and claims

ClaimActionCollection.Remove(String) removes all claim actions for the given ClaimType from the collection. ClaimActionCollectionMapExtensions.DeleteClaim(ClaimActionCollection, String) deletes a claim of the given ClaimType from the identity. DeleteClaim is primarily used with OpenID Connect (OIDC) to remove protocol-generated claims.

Sample app output

Copy

```
User Claims
```

```
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier
    9b342344f-7aab-43c2-1ac1-ba75912ca999
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name
    someone@gmail.com
AspNet.Identity.SecurityStamp
    7D4312MOWRYYBFI1KXRPHGOSTBVWSFDE
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname
    Judy
urn:google:locale
    en
urn:google:picture
    https://lh4.googleusercontent.com/-
XXXXXX/XXXXXX/XXXXXX/XXXXXX/photo.jpg

Authentication Properties

.Token.access_token
    yc23.AlvoZqz56...1lxltXV7D-ZWP9
.Token.token_type
    Bearer
.Token.expires_at
    2019-04-11T22:14:51.0000000+00:00
.Token.TicketCreated
    4/11/2019 9:14:52 PM
.TokenNames
    access_token;token_type;expires_at;TicketCreated
.persistent
.issued
    Thu, 11 Apr 2019 20:51:06 GMT
.expires
    Thu, 25 Apr 2019 20:51:06 GMT
```

Forward request information with a proxy or load balancer

If the app is deployed behind a proxy server or load balancer, some of the original request information might be forwarded to the app in request headers. This information usually includes the secure request scheme (`https`), host, and client IP address. Apps don't automatically read these request headers to discover and use the original request information.

The scheme is used in link generation that affects the authentication flow with external providers. Losing the secure scheme (`https`) results in the app generating incorrect insecure redirect URLs.

Use Forwarded Headers Middleware to make the original request information available to the app for request processing.

For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Additional resources

- dotnet/AspNetCore engineering SocialSample app: The linked sample app is on the dotnet/AspNetCore GitHub repo's `master` engineering branch. The `master` branch contains code under active development for the next release of ASP.NET Core. To see a version of the sample app for a released version of ASP.NET Core, use the **Branch** drop down list to select a release branch (for example `release/{X.Y}`).