

Authenticate using Azure AD and OpenID Connect

07/21/2017 • 8 minutes to read • Contributors  all

In this article

[Register the app with Azure AD](#)

[Configure the auth middleware](#)

[Initiate the authentication flow](#)

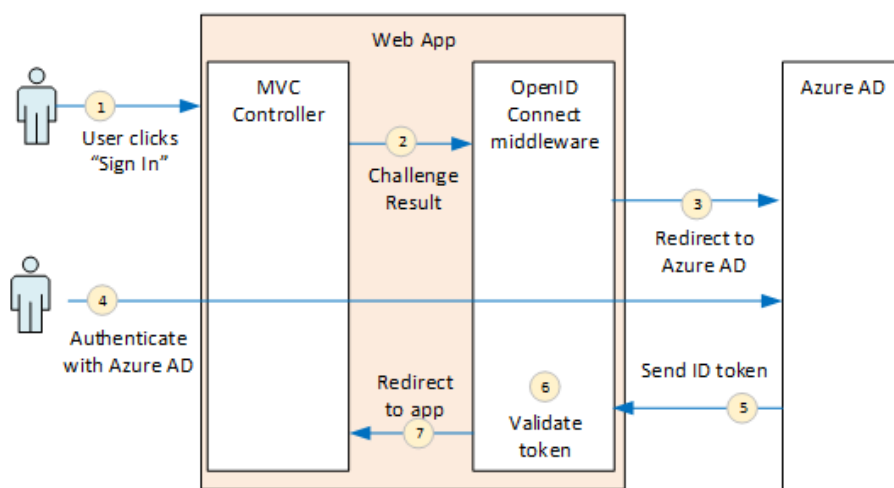
[User login sessions](#)

[About the OpenID Connect middleware](#)



[Sample code](#)

The Surveys application uses the OpenID Connect (OIDC) protocol to authenticate users with Azure Active Directory (Azure AD). The Surveys application uses ASP.NET Core, which has built-in middleware for OIDC. The following diagram shows what happens when the user signs in, at a high level.



1. The user clicks the "sign in" button in the app. This action is handled by an MVC controller.
2. The MVC controller returns a **ChallengeResult** action.
3. The middleware intercepts the **ChallengeResult** and creates a 302 response, which redirects the user to the Azure AD sign-in page.
4. The user authenticates with Azure AD.
5. Azure AD sends an ID token to the application.
6. The middleware validates the ID token. At this point, the user is now authenticated inside the application.
7. The middleware redirects the user back to application.

Register the app with Azure AD

To enable OpenID Connect, the SaaS provider registers the application inside their own Azure AD tenant.

To register the application, follow the steps in [Quickstart: Register an application with the Microsoft identity platform](#).

See [Run the Surveys application](#) for the specific steps for the Surveys application. Note the following:


- For a multitenant application, you must configure the multitenanted option explicitly. This enables other organizations to access the application.

- The reply URL is the URL where Azure AD will send OAuth 2.0 responses. When using the ASP.NET Core, this needs to match the path that you configure in the authentication middleware (see next section).

Configure the auth middleware

This section describes how to configure the authentication middleware in ASP.NET Core for multitenant authentication with OpenID Connect.


In your [startup class](#), add the OpenID Connect middleware:

C#	
<pre>app.UseOpenIdConnectAuthentication(new OpenIdConnectOptions { ClientId = configOptions.AzureAd.ClientId, ClientSecret = configOptions.AzureAd.ClientSecret, // for code flow Authority = Constants.AuthEndpointPrefix, ResponseType = OpenIdConnectResponseType.CodeIdToken, PostLogoutRedirectUri = configOptions.AzureAd.PostLogoutRedirectUri, SignInScheme = CookieAuthenticationDefaults.AuthenticationScheme, TokenValidationParameters = new TokenValidationParameters { ValidateIssuer = false }, Events = new SurveyAuthenticationEvents(configOptions.AzureAd, loggerFactory), });</pre>	

Notice that some of the settings are taken from runtime configuration options. Here's what the middleware options mean:

- **ClientId**. The application's client ID, which you got when you registered the application in Azure AD.
- **Authority**. For a multitenant application, set this to `https://login.microsoftonline.com/common/`. This is the URL for the Azure AD common endpoint, which enables users from any Azure AD tenant to sign in. For more information about the common endpoint, see [this blog post](#).
- In **TokenValidationParameters**, set **ValidateIssuer** to false. That means the app will be responsible for validating the issuer value in the ID token. (The middleware still validates the token itself.) For more information about validating the issuer, see [Issuer validation](#).
- **PostLogoutRedirectUri**. Specify a URL to redirect users after the sign out. This should be a page that allows anonymous requests — typically the home page.
- **SignInScheme**. Set this to `CookieAuthenticationDefaults.AuthenticationScheme`. This setting means that after the user is authenticated, the user claims are stored locally in a cookie. This cookie is how the user stays logged in during the browser session.
- **Events**. Event callbacks; see [Authentication events](#).

Also add the Cookie Authentication middleware to the pipeline. This middleware is responsible for writing the user claims to a cookie, and then reading the cookie during subsequent page loads.

C#	
<pre>app.UseCookieAuthentication(new CookieAuthenticationOptions { AutomaticAuthenticate = true, AutomaticChallenge = true, AccessDeniedPath = "/Home/Forbidden", CookieSecure = CookieSecurePolicy.Always, // The default setting for cookie expiration is 14 days. SlidingExpiration is set to true by default ExpireTimeSpan = TimeSpan.FromHours(1), SlidingExpiration = true });</pre>	

Initiate the authentication flow

To start the authentication flow in ASP.NET MVC, return a **ChallengeResult** from the controller:

C# Copy

```
[AllowAnonymous]
public IActionResult SignIn()
{
    return new ChallengeResult(
        OpenIdConnectDefaults.AuthenticationScheme,
        new AuthenticationProperties
        {
            IsPersistent = true,
            RedirectUri = Url.Action("SignInCallback", "Account")
        });
}
```

This causes the middleware to return a 302 (Found) response that redirects to the authentication endpoint.

User login sessions

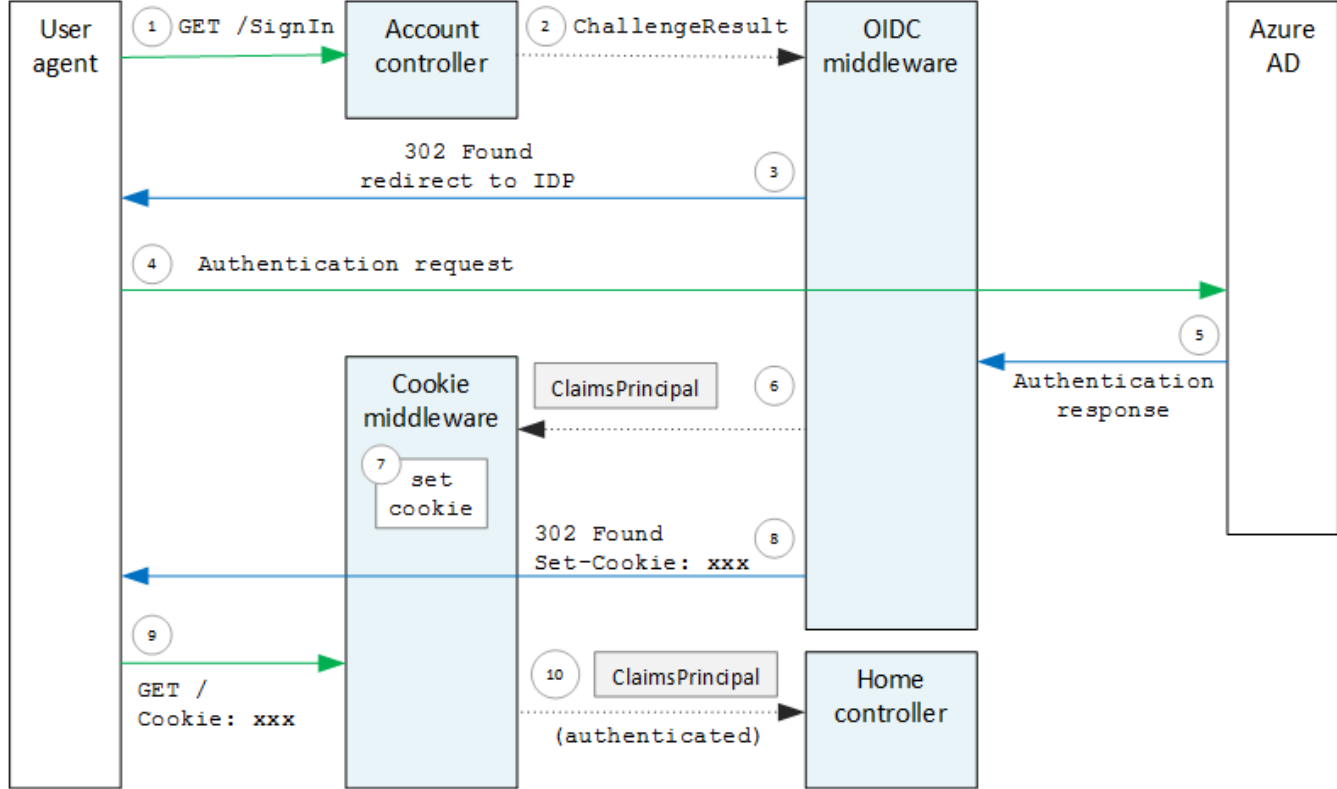
As mentioned, when the user first signs in, the Cookie Authentication middleware writes the user claims to a cookie. After that, HTTP requests are authenticated by reading the cookie.

By default, the cookie middleware writes a [session cookie](#), which gets deleted once the user closes the browser. The next time the user next visits the site, they will have to sign in again. However, if you set **IsPersistent** to true in the **ChallengeResult**, the middleware writes a persistent cookie, so the user stays logged in after closing the browser. You can configure the cookie expiration; see [Controlling cookie options](#). Persistent cookies are more convenient for the user, but may be inappropriate for some applications (say, a banking application) where you want the user to sign in every time.

About the OpenID Connect middleware

The OpenID Connect middleware in ASP.NET hides most of the protocol details. This section contains some notes about the implementation, that may be useful for understanding the protocol flow.

First, let's examine the authentication flow in terms of ASP.NET (ignoring the details of the OIDC protocol flow between the app and Azure AD). The following diagram shows the process.



In this diagram, there are two MVC controllers. The Account controller handles sign-in requests, and the Home controller serves up the home page.

Here is the authentication process:

1. The user clicks the "Sign in" button, and the browser sends a GET request. For example: GET /Account/SignIn/.
2. The account controller returns a `ChallengeResult`.
3. The OIDC middleware returns an HTTP 302 response, redirecting to Azure AD.
4. The browser sends the authentication request to Azure AD
5. The user signs in to Azure AD, and Azure AD sends back an authentication response.
6. The OIDC middleware creates a claims principal and passes it to the Cookie Authentication middleware.
7. The cookie middleware serializes the claims principal and sets a cookie.
8. The OIDC middleware redirects to the application's callback URL.
9. The browser follows the redirect, sending the cookie in the request.
10. The cookie middleware deserializes the cookie to a claims principal and sets `HttpContext.User` equal to the claims principal. The request is routed to an MVC controller.

Authentication ticket

If authentication succeeds, the OIDC middleware creates an authentication ticket, which contains a claims principal that holds the user's claims. You can access the ticket inside the **AuthenticationValidated** or **TicketReceived** event.

ⓘ Note

Until the entire authentication flow is completed, `HttpContext.User` still holds an anonymous principal, **not** the authenticated user. The anonymous principal has an empty claims collection. After authentication completes and the app redirects, the cookie middleware deserializes the authentication cookie and sets `HttpContext.User` to a claims principal that represents the authenticated user.

Authentication events

During the authentication process, the OpenID Connect middleware raises a series of events:

- **RedirectToIdentityProvider**. Called right before the middleware redirects to the authentication endpoint. You can use this event to modify the redirect URL; for example, to add request parameters. See [Adding the admin consent prompt](#) for an example.
- **AuthorizationCodeReceived**. Called with the authorization code.
- **TokenResponseReceived**. Called after the middleware gets an access token from the IDP, but before it is validated. Applies only to authorization code flow.
- **TokenValidated**. Called after the middleware validates the ID token. At this point, the application has a set of validated claims about the user. You can use this event to perform additional validation on the claims, or to transform claims. See [Working with claims](#).
- **UserInformationReceived**. Called if the middleware gets the user profile from the user info endpoint. Applies only to authorization code flow, and only when `GetClaimsFromUserInfoEndpoint = true` in the middleware options.
- **TicketReceived**. Called when authentication is completed. This is the last event, assuming that authentication succeeds. After this event is handled, the user is signed into the app.
- **AuthenticationFailed**. Called if authentication fails. Use this event to handle authentication failures — for example, by redirecting to an error page.

To provide callbacks for these events, set the **Events** option on the middleware. There are two different ways to declare the event handlers: Inline with lambdas, or in a class that derives from **OpenIdConnectEvents**. The second approach is recommended if your event callbacks have any substantial logic, so they don't clutter your startup class. Our reference implementation uses this approach.

OpenID Connect endpoints

Azure AD supports [OpenID Connect Discovery](#), wherein the identity provider (IDP) returns a JSON metadata document from a [well-known endpoint](#). The metadata document contains information such as:

- The URL of the authorization endpoint. This is where the app redirects to authenticate the user.
- The URL of the "end session" endpoint, where the app goes to log out the user.
- The URL to get the signing keys, which the client uses to validate the OIDC tokens that it gets from the IDP.

By default, the OIDC middleware knows how to fetch this metadata. Set the **Authority** option in the middleware, and the middleware constructs the URL for the metadata. (You can override the metadata URL by setting the **MetadataAddress** option.)

OpenID Connect flows

By default, the OIDC middleware uses hybrid flow with form post response mode.

- *Hybrid flow* means the client can get an ID token and an authorization code in the same round-trip to the authorization server.
- *Form post response mode* means the authorization server uses an HTTP POST request to send the ID token and authorization code to the app. The values are form-urlencoded (content type = "application/x-www-form-urlencoded").

When the OIDC middleware redirects to the authorization endpoint, the redirect URL includes all of the query string parameters needed by OIDC. For hybrid flow:

- `client_id`. This value is set in the **ClientId** option
- `scope = "openid profile"`, which means it's an OIDC request and we want the user's profile.
- `response_type = "code id_token"`. This specifies hybrid flow.
- `response_mode = "form_post"`. This specifies form post response.

To specify a different flow, set the **ResponseType** property on the options. For example:

```
app.UseOpenIdConnectAuthentication(options =>
{
    options.ResponseType = "code"; // Authorization code flow

    // Other options
})
```

[Next](#)