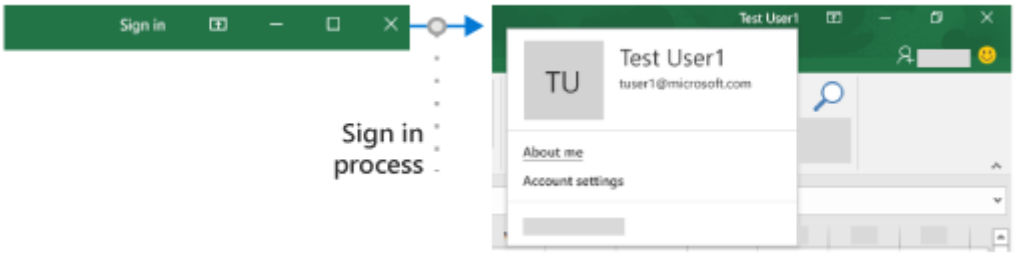
Enable single sign-on (SSO) in an Office Add-in

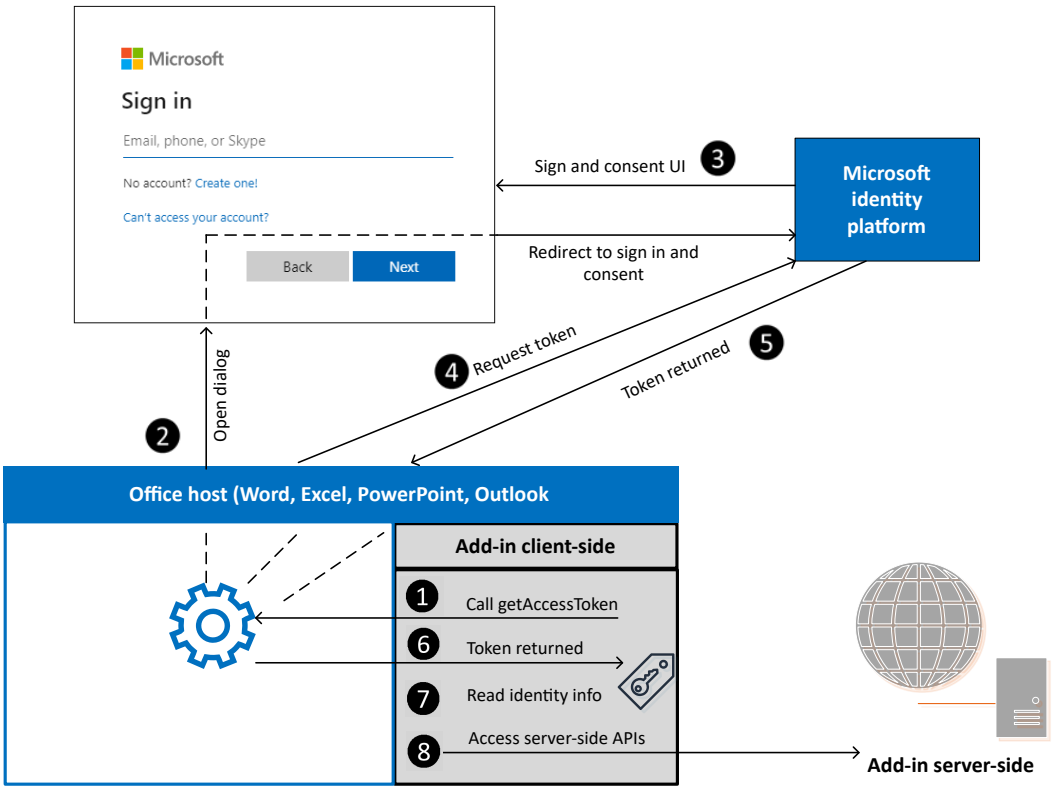
Article • 05/21/2024

Users sign in to Office using either their personal Microsoft account or their Microsoft 365 Education or work account. Take advantage of this and use single sign-on (SSO) to authenticate and authorize the user to your add-in without requiring them to sign in a second time.



How SSO works at runtime

The following diagram shows how the SSO process works. The blue elements represent Office or the Microsoft identity platform. The gray elements represent the code you write and include the client-side code (task pane) and the server-side code for your add-in.



1. In the add-in, your JavaScript code calls the Office.js API getAccessToken. If the user is already signed in to Office, the Office host will return the access token with the claims of the signed in user.
2. If the user is not signed in, the Office host application opens a dialog box for the user to sign in. Office redirects to the Microsoft identity platform to complete the sign-in process.
3. If this is the first time the current user has used your add-in, they are prompted to consent.
4. The Office host application requests the **access token** from the Microsoft identity platform for the current user.
5. The Microsoft identity platform returns the access token to Office. Office will cache the token on your behalf so that future calls to **getAccessToken** simply return the cached token.
6. The Office host application returns the **access token** to the add-in as part of the result object returned by the `getAccessToken` call.
7. The token is both an **access token** and an **identity token**. You can use it as an identity token to parse and examine claims about the user, such as the user's name and email address.
8. Optionally, the add-in can use the token as an **access token** to make authenticated HTTPS requests to APIs on the server-side. Because the access token contains

identity claims, the server can store information associated with the user's identity; such as the user's preferences.

Requirements and Best Practices

Don't cache the access token

Never cache or store the access token in your client-side code. Always call getAccessToken when you need an access token. Office will cache the access token (or request a new one if it expired.) This will help to avoid accidentally leaking the token from your add-in.

Enable modern authentication for Outlook

If you're working with an Outlook add-in, be sure to enable Modern Authentication for the Microsoft 365 tenancy. For information about how to do this, see Enable or disable modern authentication for Outlook in Exchange Online.

Implement a fallback authentication system

You should *not* rely on SSO as your add-in's only method of authentication. You should implement an alternate authentication system that your add-in can fall back to in certain error situations. For example, if your add-in is loaded on an older version of Office that does not support SSO, the `getAccessToken` call will fail.

For Excel, Word, and PowerPoint add-ins you will typically want to fall back to using the Microsoft identity platform. For more information, see Authenticate with the Microsoft identity platform.

For Outlook add-ins, there is a recommended fallback system. For more information, see Scenario: Implement single sign-on to your service in an Outlook add-in.

You can also use a system of user tables and authentication, or you can leverage one of the social login providers. For more information about how to do this with an Office Add-in, see Authorize external services in your Office Add-in.

For code samples that use the Microsoft identity platform as the fallback system, see Office Add-in NodeJS SSO and Office Add-in ASP.NET SSO.

Develop an SSO add-in

This section describes the tasks involved in creating an Office Add-in that uses SSO. These tasks are described here independently of language or framework. For step-by-step instructions, see:

- Create a Node.js Office Add-in that uses single sign-on
- Create an ASP.NET Office Add-in that uses single sign-on

Register your add-in with the Microsoft identity platform

To work with SSO you need to register your add-in with the Microsoft identity platform. This will enable the Microsoft identity platform to provide authentication and authorization services for your add-in. Creating the app registration includes the following tasks.

- Get an application (client) ID to identify your add-in to the Microsoft identity platform.
- Generate a client secret to act as a password for your add-in when requesting a token.
- Specify the permissions that your add-in requires. The Microsoft Graph "profile" and "openid" permissions are always required. You may need additional permissions depending on what your add-in needs to do.
- Grant the Office applications trust to the add-in.
- Pre-authorize the Office applications to the add-in with the default scope *access_as_user*.

For more details about this process, see Register an Office Add-in that uses SSO with the Microsoft identity platform.

Configure the add-in

Your manifest must provide Office with certain information about how the add-in is registered in Microsoft Entra ID. The configuration depends on which type of manifest the add-in uses.

Unified manifest

There should be a "webApplicationInfo" property in the root of the manifest. It has a required child "id" property which must be set to the application ID (a GUID) of the add-in in the Microsoft identity platform. For SSO, it must also have a child "resource" property that is set to the URI of the add-in. This is the same **Application ID URI** (including the `api:` protocol) that you set when you registered the add-in

with the Microsoft identity platform. The URI must end with the client ID specified in the "webApplicationInfo.id" property. The following is an example:

JSON

```
"webApplicationInfo": {
    "id": "a661fed9-f33d-4e95-b6cf-624a34a2f51d",
    "resource": "api://addin.contoso.com/a661fed9-f33d-4e95-b6cf-
624a34a2f51d"
},
```

> **Note**
>
> Experienced add-in developers should note that, there is no unified manifest property corresponding to the **<Scopes>** element in the add-in only manifest. Microsoft Graph and other API permissions are requested at runtime in your code.

Include the Identity API requirement set

To use SSO your add-in requires the Identity API 1.3 requirement set. For more information, see IdentityAPI.

Add client-side code

Add JavaScript to the add-in to:

- Call getAccessToken.
- Parse the access token or pass it to the add-in's server-side code.

The following code shows a simple example of calling `getAccessToken` and parsing the token for the user name and other credentials.

> **Note**
>
> This example handles only one kind of error explicitly. For examples of more elaborate error handling, see Office Add-in NodeJS SSO and Office Add-in ASP.NET SSO.

JavaScript

```
async function getUserData() {
    try {
        let userTokenEncoded = await OfficeRuntime.auth.getAccessToken();
        let userToken = jwt_decode(userTokenEncoded); // Using the
https://www.npmjs.com/package/jwt-decode library.
        console.log(userToken.name); // user name
        console.log(userToken.preferred_username); // email
        console.log(userToken.oid); // user id
    }
    catch (exception) {
        if (exception.code === 13003) {
            // SSO is not supported for domain user accounts, only
            // Microsoft 365 Education or work account, or a Microsoft
account.
        } else {
            // Handle error
        }
    }
}
```

When to call getAccessToken

If your add-in requires a signed in user, then you should call `getAccessToken` from inside `Office.initialize`. You should also pass `allowSignInPrompt: true` in the `options` parameter of `getAccessToken`. For example; `OfficeRuntime.auth.getAccessToken( { allowSignInPrompt: true });` This will ensure that if the user is not yet signed in, that Office prompts the user through the UI to sign in now.

If the add-in has some functionality that doesn't require a signed in user, then you can call `getAccessToken` *when the user takes an action that requires a signed in user*. There is no significant performance degradation with redundant calls of `getAccessToken` because Office caches the access token and will reuse it, until it expires, without making another call to the Microsoft identity platform whenever `getAccessToken` is called. So you can add calls of `getAccessToken` to all functions and handlers that initiate an action where the token is needed.

> **ⓘ Important**
>
> As a best security practice, always call `getAccessToken` when you need an access token. Office will cache it for you. Don't cache or store the access token using your own code.

Pass the access token to server-side code

If you need to access web APIs on your server, or additional services such as Microsoft Graph, you'll need to pass the access token to your server-side code. The access token provides access (for the authenticated user) to your web APIs. Also the server-side code can parse the token for identity information if it needs it. (See **Use the access token as an identity token** below.) There are many libraries available for different languages and platforms that can help simplify the code you write. For more information, see Overview of the Microsoft Authentication Library (MSAL).

If you need to access Microsoft Graph data, your server-side code should do the following:

- Validate the access token (see **Validate the access token** below).
- Initiate the OAuth 2.0 On-Behalf-Of flow with a call to the Microsoft identity platform that includes the access token, some metadata about the user, and the credentials of the add-in (its ID and secret). The Microsoft identity platform will return a new access token that can be used to access Microsoft Graph.
- Get data from Microsoft Graph by using the new token.
- If you need to cache the new access token for multiple calls, we recommend using Token cache serialization in MSAL.NET.

> **ⓘ Important**
>
> As a best security practice, always use the server-side code to make Microsoft Graph calls, or other calls that require passing an access token. Never return the OBO token to the client to enable the client to make direct calls to Microsoft Graph. This helps protect the token from being intercepted or leaked. For more information on the proper protocol flow, see the **OAuth 2.0 protocol diagram**

The following code shows an example of passing the access token to the server-side. The token is passed in an `Authorization` header when sending a request to a server-side web API. This example sends JSON data, so it uses the `POST` method, but `GET` is sufficient to send the access token when you are not writing to the server.

JavaScript

```
$.ajax({
    type: "POST",
    url: "/api/DoSomething",
    headers: {
        "Authorization": "Bearer " + accessToken
    },
    data: { /* some JSON payload */ },
    contentType: "application/json; charset=utf-8"
}).done(function (data) {
```

```
    // Handle success
}).fail(function (error) {
    // Handle error
}).always(function () {
    // Cleanup
});
```

For more details about getting authorized access to the user's Microsoft Graph data, see Authorize to Microsoft Graph in your Office Add-in.

Validate the access token

Web APIs on your server must validate the access token if it is sent from the client. The token is a JSON Web Token (JWT), which means that validation works just like token validation in most standard OAuth flows. There are a number of libraries available that can handle JWT validation, but the basics include:

- Checking that the token is well-formed
- Checking that the token was issued by the intended authority
- Checking that the token is targeted to the Web API

Keep in mind the following guidelines when validating the token.

- Valid SSO tokens will be issued by the Azure authority, `https://login.microsoftonline.com`. The `iss` claim in the token should start with this value.
- The token's `aud` parameter will be set to the application ID of the add-in's Azure app registration.
- The token's `scp` parameter will be set to `access_as_user`.

For more information on token validation, see Microsoft identity platform access tokens.

Use the access token as an identity token

If your add-in needs to verify the user's identity, the access token returned from `getAccessToken()` contains information that can be used to establish the identity. The following claims in the token relate to identity.

- `name` - The user's display name.
- `preferred_username` - The user's email address.
- `oid` - A GUID representing the ID of the user in the Microsoft identity system.
- `tid` - A GUID representing the tenant tha the user is signing in to.

For more details on these and other claims, see Microsoft identity platform ID tokens. If you need to construct a unique ID to represent the user in your system, refer to Using claims to reliably identify a user for more information.

Example access token

The following is a typical decoded payload of an access token. For information about the properties, see Microsoft identity platform access tokens.

JavaScript


```
{
    aud: "2c3caa80-93f9-425e-8b85-0745f50c0d24",
    iss: "https://login.microsoftonline.com/fec4f964-8bc9-4fac-b972-
1c1da35adbcd/v2.0",
    iat: 1521143967,
    nbf: 1521143967,
    exp: 1521147867,
    aio:
"ATQAy/8GAAAA0agfnU4DTJUlEqGLisMtBk5q6z+6DB+sgiRjB/Ni73q83y0B86yBHU/WFJnlMQJ
8",
    azp: "e4590ed6-62b3-5102-beff-bad2292ab01c",
    azpacr: "0",
    e_exp: 262800,
    name: "Mila Nikolova",
    oid: "6467882c-fdfd-4354-a1ed-4e13f064be25",
    preferred_username: "milan@contoso.com",
    scp: "access_as_user",
    sub: "XkjgWjdmaZ-_xDmhgN1BMP2vL2YOfeVxfPT_o8GRWaw",
    tid: "fec4f964-8bc9-4fac-b972-1c1da35adbcd",
    uti: "MICAQyhrH02ov54bCtIDAA",
    ver: "2.0"
}
```


Using SSO with an Outlook add-in

There are some small, but important differences in using SSO in an Outlook add-in from using it in an Excel, PowerPoint, or Word add-in. Be sure to read Authenticate a user with a single sign-on token in an Outlook add-in and Scenario: Implement single sign-on to your service in an Outlook add-in.

Google Chrome 3rd party cookie support

Google Chrome is phasing out 3rd party cookies in 2024 and introducing a feature named Tracking Prevention. If Tracking Prevention is enabled in the Chrome browser,

your add-in will not be able to use any 3rd party cookies. Your add-in may encounter issues when authenticating the user, such as multiple sign-on requests, or errors.

For improved authentication experiences, see Using device state for an improved SSO experience on browsers with blocked third-party cookies.

For more information about the Google Chrome rollout, see the following resources:

- The next step toward phasing out third-party cookies in Chrome.
- The Privacy Sandbox Timeline for the Web

See also

- Microsoft identity platform documentation
- Requirement sets
- IdentityAPI

Single sign-on (SSO) quick start

Article • 05/16/2025

In this article, you'll use the Yeoman generator for Office Add-ins to create an Office Add-in for Excel, Outlook, Word, or PowerPoint that uses single sign-on (SSO).

> **Note**
>
> The SSO template provided by the Yeoman generator for Office Add-ins only runs on localhost and cannot be deployed. If you're building a new Office Add-in with SSO for production purposes, follow the instructions in Create a Node.js Office Add-in that uses single sign-on.

Prerequisites

- Node.js (the latest LTS version).
- The latest version of Yeoman and the Yeoman generator for Office Add-ins. To install these tools globally, run the following command via the command prompt.

  command line

  ```
  npm install -g yo generator-office
  ```

  > **Note**
  >
  > Even if you've previously installed the Yeoman generator, we recommend you update your package to the latest version from npm.

- If you're using a Mac and don't have the Azure CLI installed on your machine, you must install Homebrew. The SSO configuration script that you'll run during this quick start will use Homebrew to install the Azure CLI, and will then use the Azure CLI to configure SSO within Azure.

Create the add-in project

> **Tip**

The Yeoman generator can create an SSO-enabled Office Add-in for Excel, Outlook, Word, or PowerPoint with script type of JavaScript or TypeScript. The following instructions specify `JavaScript` and `Excel`, but you should choose the script type and Office client application that best suits your scenario.

Run the following command to create an add-in project using the Yeoman generator. A folder that contains the project will be added to the current directory.

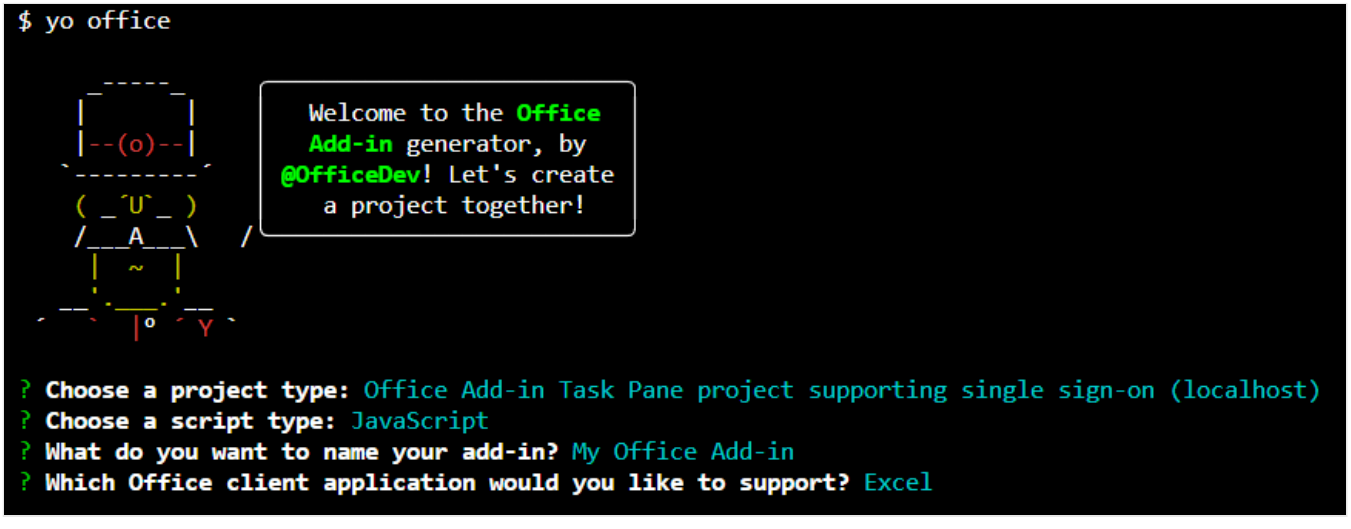
command line

```
yo office
```

> **Note**
>
> When you run the `yo office` command, you may receive prompts about the data collection policies of Yeoman and the Office Add-in CLI tools. Use the information that's provided to respond to the prompts as you see fit.

When prompted, provide the following information to create your add-in project.

- **Choose a project type:** `Office Add-in Task Pane project supporting single sign-on (localhost)`
- **Choose a script type:** `JavaScript`
- **What do you want to name your add-in?** `My Office Add-in`
- **Which Office client application would you like to support?** Choose `Excel`, `Outlook`, `Word`, or `Powerpoint`.



After you complete the wizard, the generator creates the project and installs supporting Node components.

Explore the project

The add-in project that you've created with the Yeoman generator contains code for an SSO-enabled task pane add-in.

Configuration

The following files specify configuration settings for the add-in.

- The **./manifest.xml** or **manifest.json** file in the root directory of the project defines the settings and capabilities of the add-in.
- The **./.ENV** file in the root directory of the project defines constants that are used by the add-in project.

Task pane

The following files define the add-in's task pane UI and functionality.

- The **./src/taskpane/taskpane.html** file contains the HTML markup for the task pane.
- The **./src/taskpane/taskpane.css** file contains the CSS that's applied to content in the task pane.
- The **./src/taskpane/taskpane.js** file contains code to initialize the add-in and also code that uses the Office JavaScript API library to add the data from Microsoft Graph to the Office document.

Authentication

The following files facilitate the SSO process and write data to the Office document.

- In a JavaScript project, the **./src/helpers/documentHelper.js** file contains code that encapsulates the user's profile information for insertion into the current Office document. There's no such file in a TypeScript project. Instead, the code that gathers the profile information is inline in the **./src/taskpane/taskpane.ts** file.
- The **./src/helpers/fallbackauthdialog.html** file is the UI-less page that loads the JavaScript for the fallback authentication strategy. The `<script>` tag to load the JavaScript is inserted into the file when Webpack.config.js runs.
- The **./src/helpers/fallbackauthdialog.js** file contains the JavaScript for the fallback authentication strategy that signs in the user with msal.js.

- The **./src/helpers/message-helper.js** file contains JavaScript that shows or hides error messages to the user.
- The **./src/helpers/middle-tier-calls.js** file contains the JavaScript that calls your web API for fetching data.
- The **./src/helpers/sso-helper.js** file contains the JavaScript call to the SSO API, `getAccessToken`, receives the access token, and includes it in a call to Microsoft Graph for the data. In the event of an error or in scenarios when SSO authentication isn't supported, it invokes the fallback strategy.

Configure SSO

Now that your add-in project is created and contains the code that's necessary to facilitate the SSO process, complete the following steps to configure SSO for your add-in.

1. Go to the root folder of the project.

   command line

   ```
   cd "My Office Add-in"
   ```

2. Run the following command to configure SSO for the add-in.

   command line

   ```
   npm run configure-sso
   ```

   > **⚠ Warning**
   >
   > This command will fail if your tenant is configured to require two-factor authentication. In this scenario, you'll need to manually complete the Azure app registration and SSO configuration steps by following all the steps in the Create a Node.js Office Add-in that uses single sign-on tutorial.

3. A web browser window will open and prompt you to sign in to Azure. Sign in to Azure using your Microsoft 365 administrator credentials. These credentials will be used to register a new application in Azure and configure the settings required by SSO.

   > **ⓘ Note**

If you sign in to Azure using non-administrator credentials during this step, the `configure-sso` script won't be able to provide administrator consent for the add-in to users within your organization. SSO will therefore not be available to users of the add-in and they'll be prompted to sign-in.

4. After you enter your credentials, close the browser window and return to the command prompt. As the SSO configuration process continues, you'll see status messages being written to the console. As described in the console messages, files within the add-in project that the Yeoman generator created are automatically updated with data that's required by the SSO process.

Test your add-in

If you've created an Excel, Word, or PowerPoint add-in, complete the steps in the following section to try it. If you've created an Outlook add-in, complete the steps in the Outlook section instead.

Excel, Word, and PowerPoint

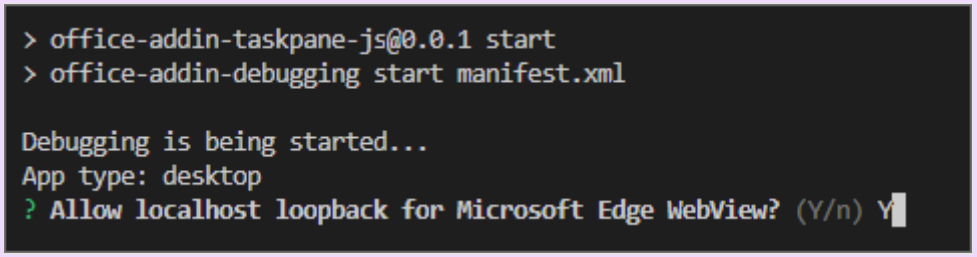
Complete the following steps to test an Excel, Word, or PowerPoint add-in.

1. When the SSO configuration process completes, run the following command to build the project, start the local web server, and sideload your add-in in the previously selected Office client application.

   ⓘ **Note**

   - Office Add-ins should use HTTPS, not HTTP, even while you're developing. If you're prompted to install a certificate after you run one of the following commands, accept the prompt to install the certificate that the Yeoman generator provides. You may also have to run your command prompt or terminal as an administrator for the changes to be made.
   - If this is your first time developing an Office Add-in on your machine, you may be prompted in the command line to grant Microsoft Edge WebView a loopback exemption ("Allow localhost loopback for Microsoft Edge WebView?"). When prompted, enter `Y` to allow the exemption. Note that you'll need administrator privileges to allow the exemption. Once allowed, you shouldn't be prompted for an exemption when you sideload Office Add-ins in

the future (unless you remove the exemption from your machine). To learn more, see "We can't open this add-in from localhost" when loading an Office Add-in or using Fiddler.



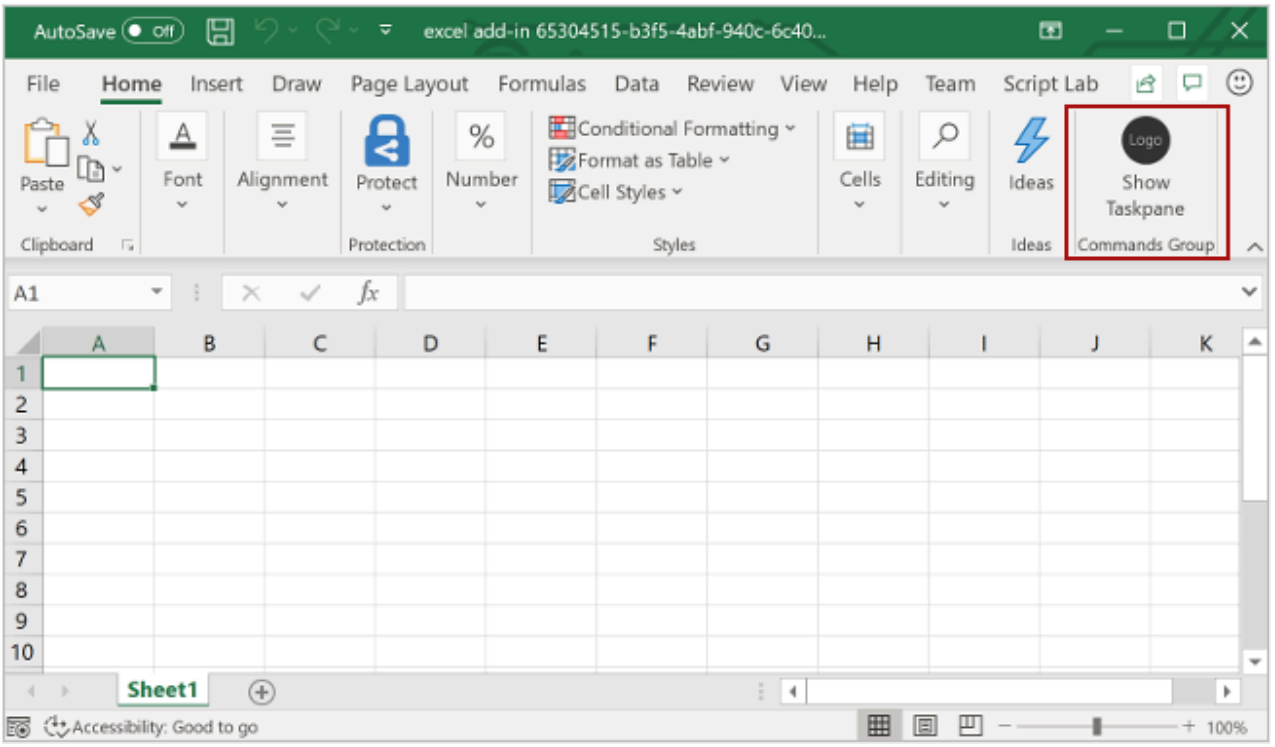
- When you first use Yeoman generator to develop an Office Add-in, your default browser opens a window where you'll be prompted to sign in to your Microsoft 365 account. If a sign-in window doesn't appear and you encounter a sideloading or login timeout error, run `atk auth login m365`.

command line

```
npm start
```

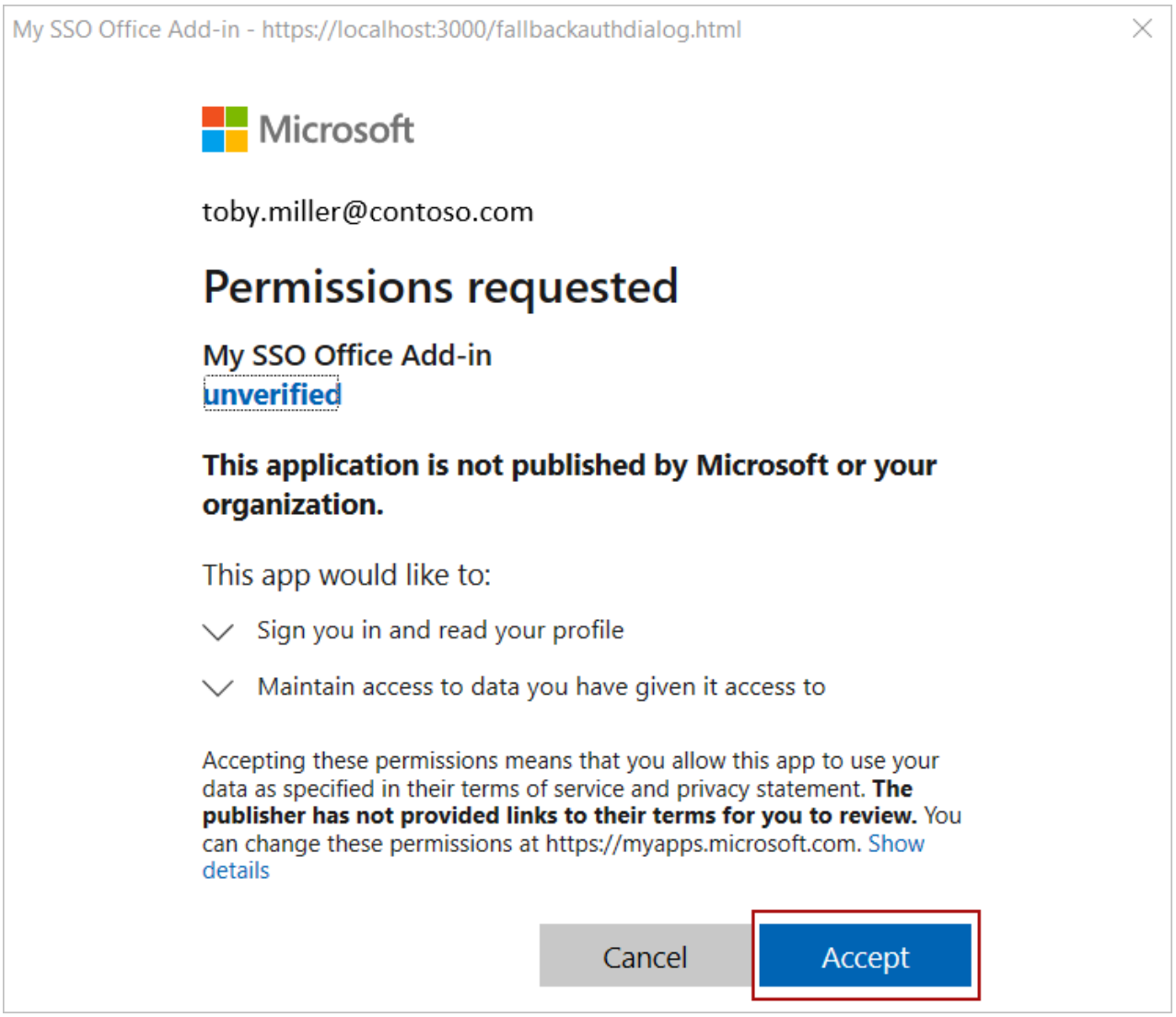
2. When Excel, Word, or PowerPoint opens when you run the previous command, make sure you're signed in with a user account that's a member of the same Microsoft 365 organization as the Microsoft 365 administrator account that you used to connect to Azure while configuring SSO in step 3 of the previous section. Doing so establishes the appropriate conditions for SSO to succeed.

3. In the Office client application, choose the **Home** tab, and then choose **Show Taskpane** to open the add-in task pane.



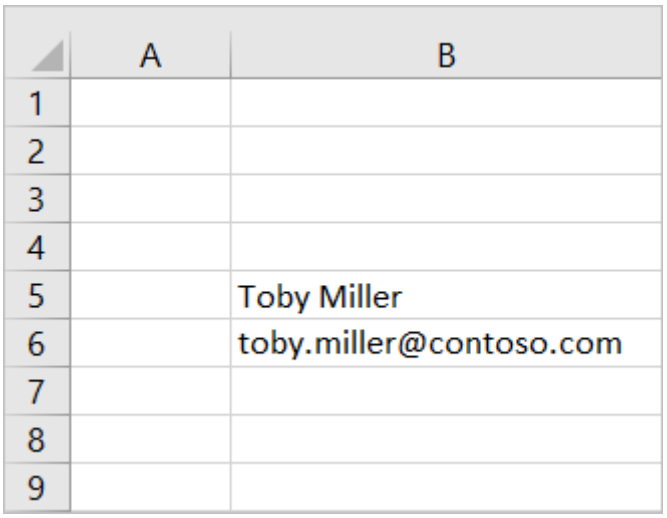
4. At the bottom of the task pane, choose the **Get My User Profile Information** button to initiate the SSO process.

5. If a dialog window appears to request permissions on behalf of the add-in, this means that SSO is not supported for your scenario and the add-in has instead fallen back to an alternate method of user authentication. This may occur when the tenant administrator hasn't granted consent for the add-in to access Microsoft Graph, or when the user isn't signed in to Office with a valid Microsoft account or Microsoft 365 Education or Work account. Choose **Accept** to continue.



> ⓘ Note
>
> After a user accepts this permissions request, they won't be prompted again in the future.

6. The add-in retrieves profile information for the signed-in user and writes it to the document. The following image shows an example of profile information written to an Excel worksheet.



7. When you want to stop the local web server and uninstall the add-in, follow the applicable instructions:

    - To stop the server, run the following command. If you used `npm start`, the following command also uninstalls the add-in.

        ```
        command line

        npm stop
        ```

    - If you manually sideloaded the add-in, see Remove a sideloaded add-in.

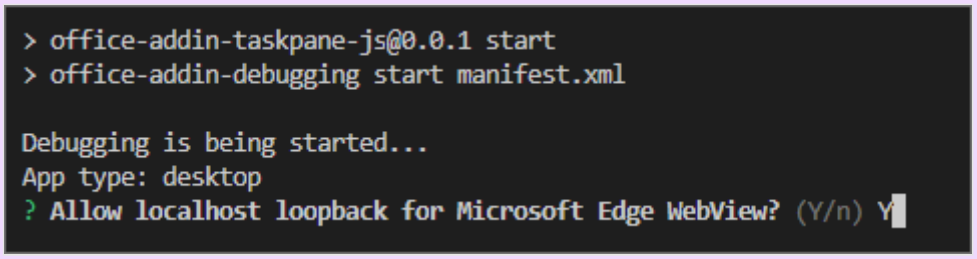
Outlook

Complete the following steps to try out an Outlook add-in.

1. When the SSO configuration process completes, run the following command to build the project and start the local web server.

    > **Note**
    >
    > - Office Add-ins should use HTTPS, not HTTP, even while you're developing. If you're prompted to install a certificate after you run one of the following commands, accept the prompt to install the certificate that the Yeoman generator provides. You may also have to run your command prompt or terminal as an administrator for the changes to be made.
    >
    > - If this is your first time developing an Office Add-in on your machine, you may be prompted in the command line to grant Microsoft Edge WebView a loopback exemption ("Allow localhost loopback for Microsoft Edge

WebView?"). When prompted, enter `Y` to allow the exemption. Note that you'll need administrator privileges to allow the exemption. Once allowed, you shouldn't be prompted for an exemption when you sideload Office Add-ins in the future (unless you remove the exemption from your machine). To learn more, see "We can't open this add-in from localhost" when loading an Office Add-in or using Fiddler.

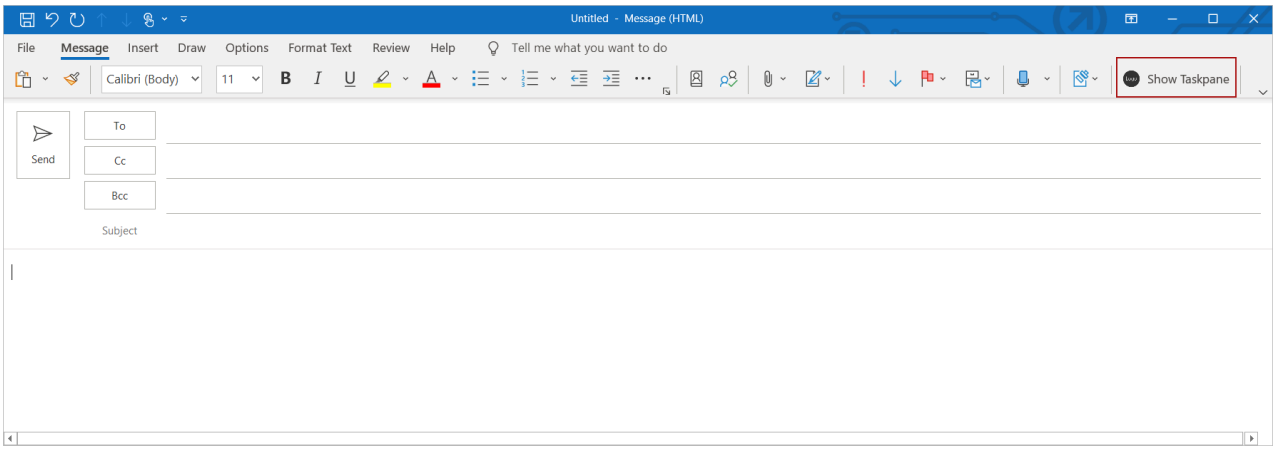


- When you first use Yeoman generator to develop an Office Add-in, your default browser opens a window where you'll be prompted to sign in to your Microsoft 365 account. If a sign-in window doesn't appear and you encounter a sideloading or login timeout error, run `atk auth login m365`.

command line

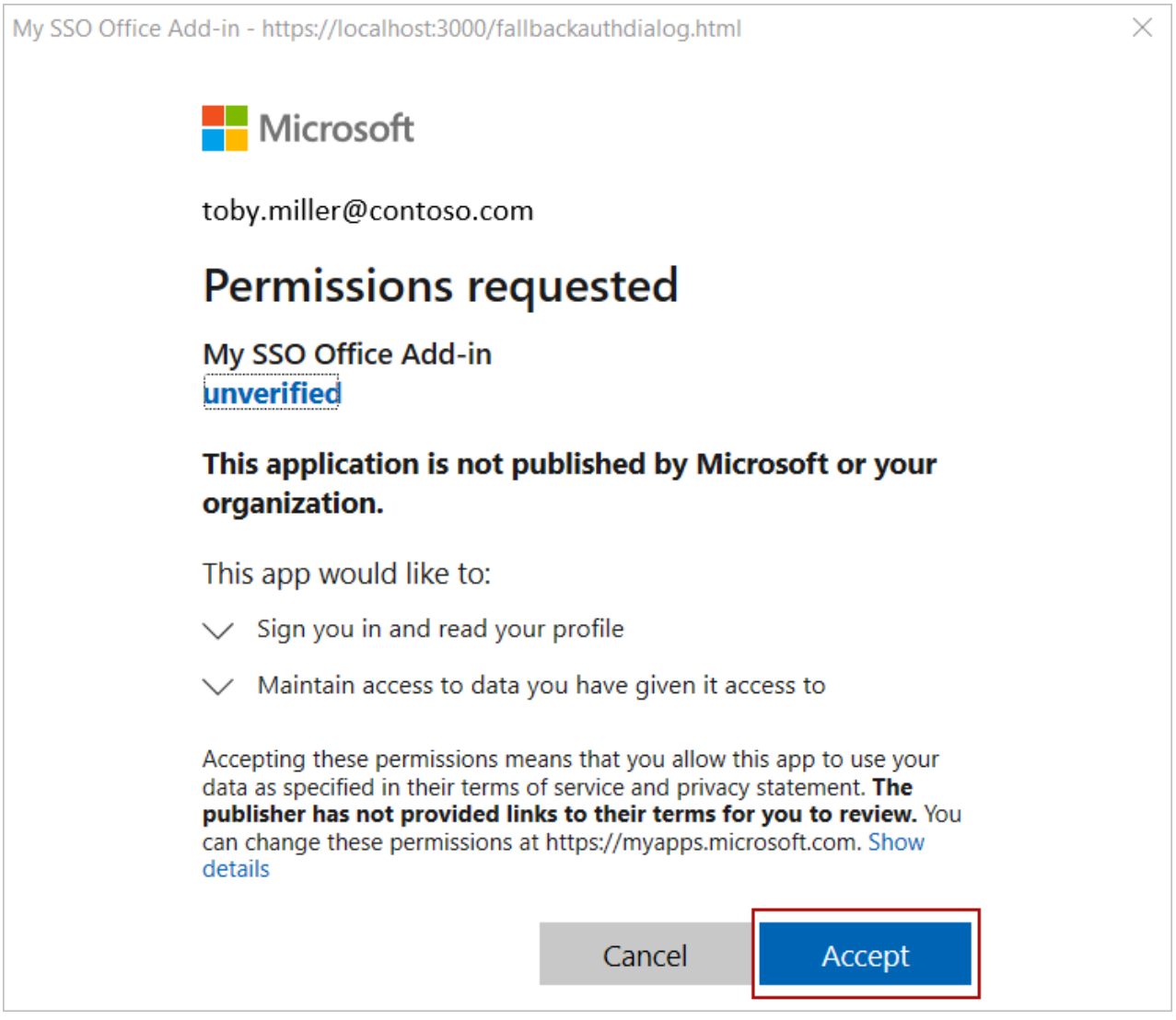
```
npm start
```

2. Outlook will start and sideload the add-in. Make sure that you're signed in to Outlook with a user that's a member of the same Microsoft 365 organization as the Microsoft 365 administrator account that you used to connect to Azure while configuring SSO in step 3 of the previous section. Doing so establishes the appropriate conditions for SSO to succeed.
3. In Outlook, compose a new message.
4. In the message compose window, choose the **Show Taskpane** button to open the add-in task pane.



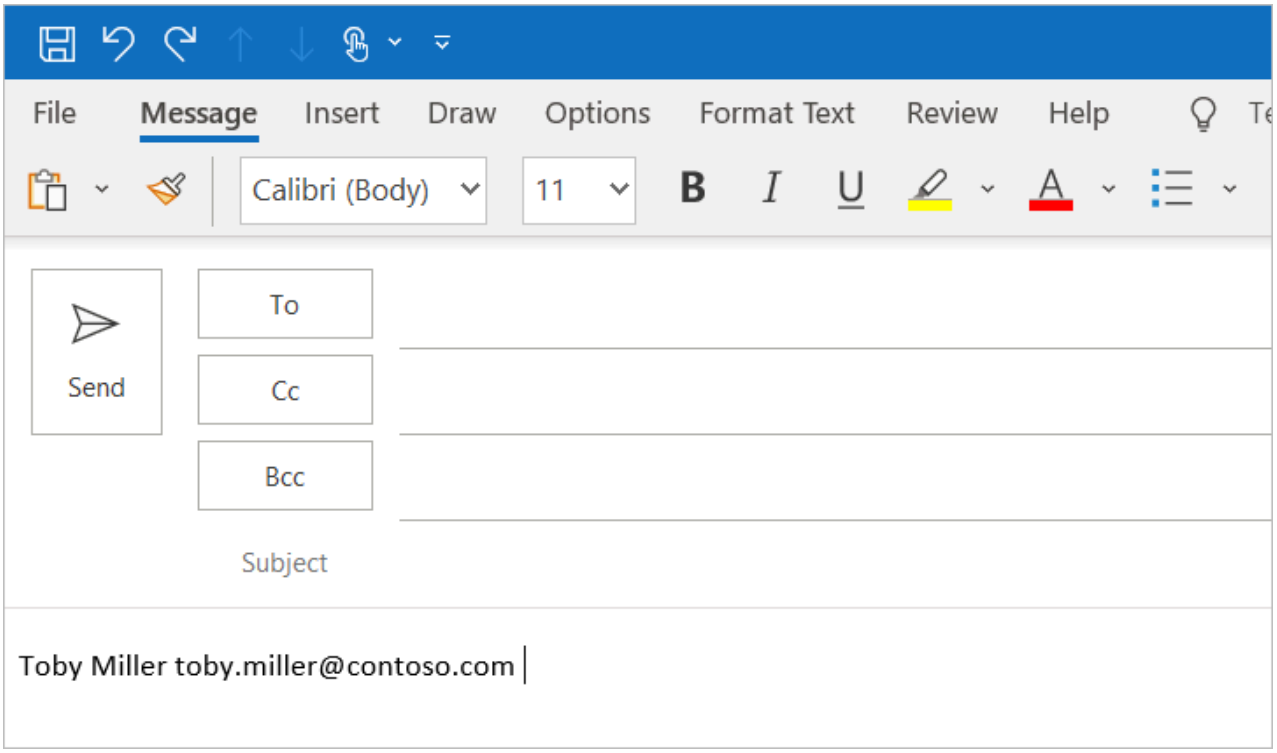
5. At the bottom of the task pane, choose the **Get My User Profile Information** button to initiate the SSO process.

6. If a dialog window appears to request permissions on behalf of the add-in, this means that SSO is not supported for your scenario and the add-in has instead fallen back to an alternate method of user authentication. This may occur when the tenant administrator hasn't granted consent for the add-in to access Microsoft Graph, or when the user isn't signed in to Office with a valid Microsoft account or Microsoft 365 Education or Work account. Choose **Accept** to continue.



> **Note**
>
> After a user accepts this permissions request, they won't be prompted again in the future.

7. The add-in retrieves profile information for the signed-in user and writes it to the body of the email message.



8. When you want to stop the local web server and uninstall the add-in, follow the applicable instructions:

   - To stop the server, run the following command. If you used `npm start`, the following command should also uninstall the add-in.

     ```
     command line
     npm stop
     ```

   - If you manually sideloaded the add-in, see Remove a sideloaded add-in.

Next steps

Congratulations, you've successfully created a task pane add-in that uses SSO when possible, and uses an alternate method of user authentication when SSO is not supported. To learn

about customizing your add-in to add new functionality that requires different permissions, see Customize your Node.js SSO-enabled add-in.

Troubleshooting

- Ensure your environment is ready for Office development by following the instructions in Set up your development environment.
- Some of the sample code uses ES6 JavaScript. This isn't compatible with older versions of Office that use the Trident (Internet Explorer 11) browser engine. For information on how to support those platforms in your add-in, see Support older Microsoft webviews and Office versions. If you don't already have a Microsoft 365 subscription to use for development, you might qualify for a Microsoft 365 E5 developer subscription through the Microsoft 365 Developer Program; for details, see the FAQ. Alternatively, you can sign up for a 1-month free trial or purchase a Microsoft 365 plan.
- The automatic `npm install` step Yo Office performs may fail. If you see errors when trying to run `npm start`, navigate to the newly created project folder in a command prompt and manually run `npm install`. For more information about Yo Office, see Create Office Add-in projects using the Yeoman Generator.

See also

- Enable single sign-on for Office Add-ins
- Customize your Node.js SSO-enabled add-in
- Create a Node.js Office Add-in that uses single sign-on
- Troubleshoot error messages for single sign-on (SSO)
- Using Visual Studio Code to publish

Register an Office Add-in that uses single sign-on (SSO) with the Microsoft identity platform

Article • 11/13/2023

This article explains how to register an Office Add-in with the Microsoft identity platform so that you can use SSO. Register the add-in when you begin developing it so that when you progress to testing or production, you can change the existing registration or create separate registrations for development, testing, and production versions of the add-in.

The following table itemizes the information that you need to carry out this procedure and the corresponding placeholders that appear in the instructions.

Information	Examples	Placeholder
A human readable name for the add-in. (Uniqueness recommended, but not required.)	`Contoso Marketing Excel Add-in (Prod)`	`<add-in-name>`
An application ID which Azure generates for you as part of the registration process.	`c6c1f32b-5e55-4997-881a-753cc1d563b7`	`<app-id>`
The fully qualified domain name (except for protocol) of the add-in. *You must use a domain that you own.* For this reason, you cannot use certain well-known domains such as `azurewebsites.net` or `cloudapp.net`. The domain must be the same, including any subdomains, as is used in the URLs in the <**Resources**> section of the add-in's manifest.	`localhost:6789`, `addins.contoso.com`	`<fully-qualified-domain-name>`
The permissions to the Microsoft identity platform and Microsoft Graph that your add-in needs. (`profile` is always required.)	`profile`, `Files.Read.All`	N/A

⊗ Caution

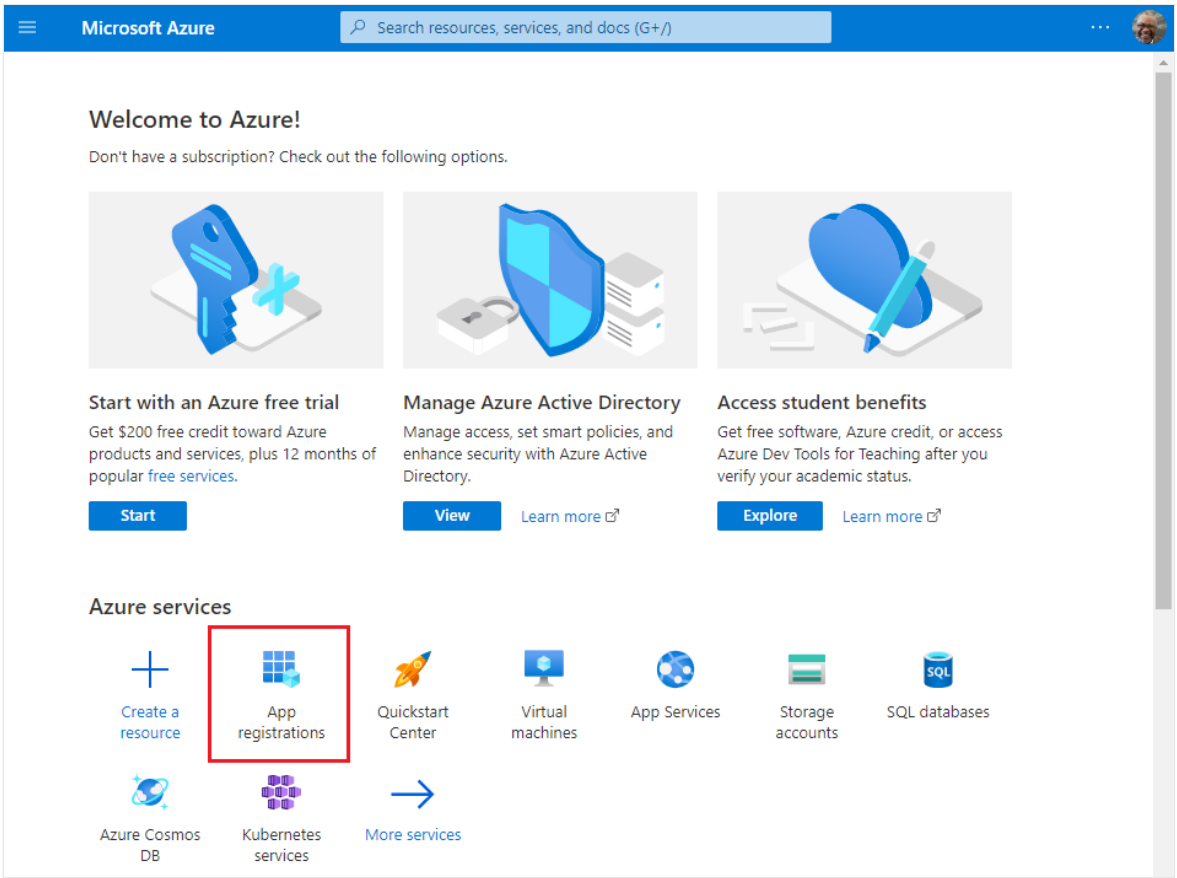
Sensitive information: The application ID URI (`<fully-qualified-domain-name>`) is logged as part of the authentication process when an add-in using SSO is activated in Office running inside of Microsoft Teams. The URI mustn't contain sensitive information.

Register the add-in with Microsoft identity platform

You need to create an app registration in Azure that represents your web server. This enables authentication support so that proper access tokens can be issued to the client code in JavaScript. This registration supports both SSO in the client, and fallback authentication using the Microsoft Authentication Library (MSAL).

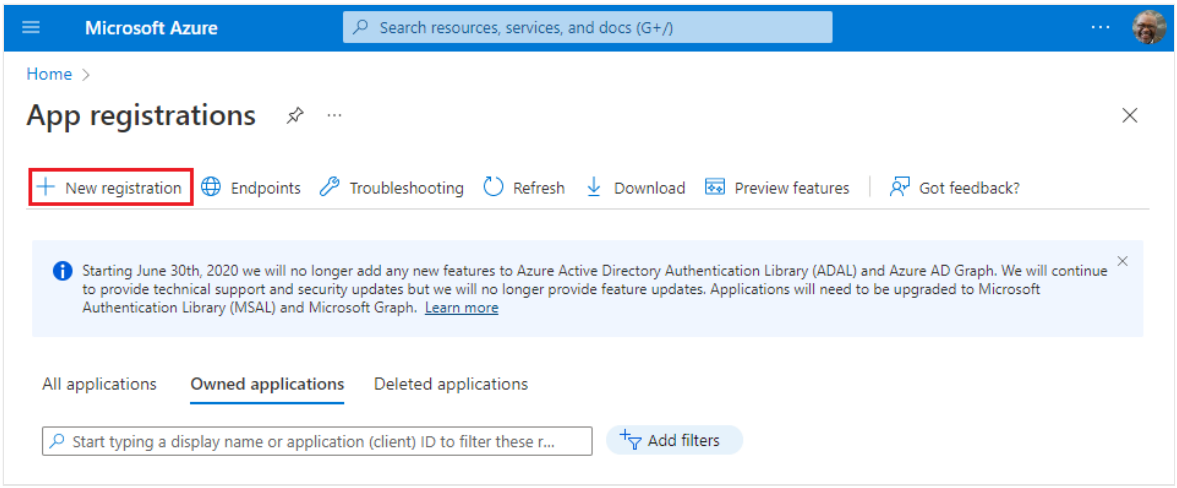
1. Sign in to the Azure portal with the ***admin*** credentials to your Microsoft 365 tenancy. For example, **MyName@contoso.onmicrosoft.com**.

2. Select **App registrations**. If you don't see the icon, search for "app registration" in the search bar.



   The **App registrations** page appears.

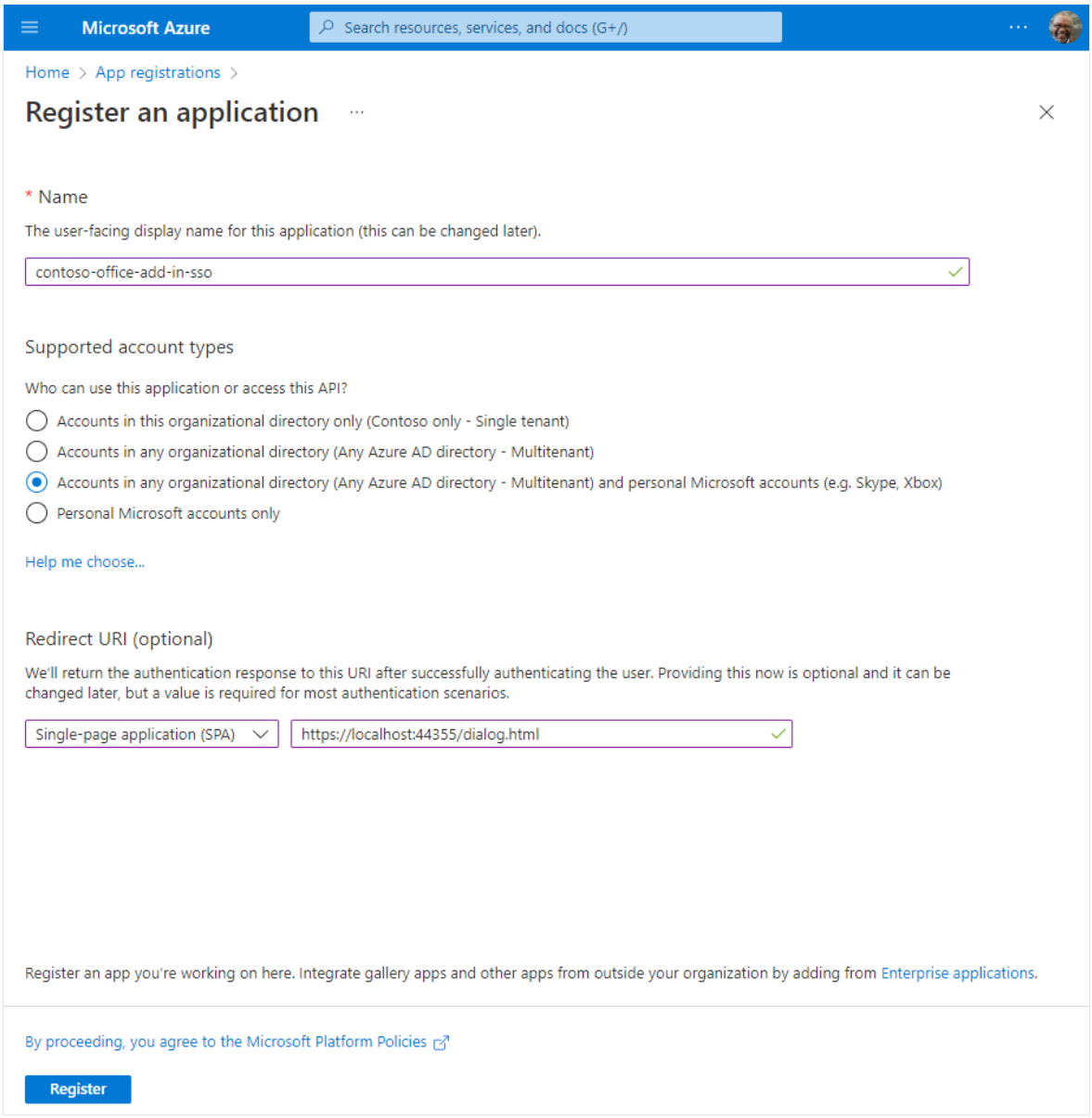
3. Select **New registration**.



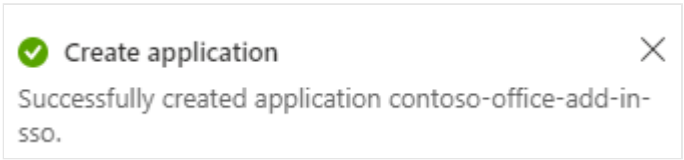
The **Register an application** page appears.

4. On the **Register an application** page, set the values as follows.

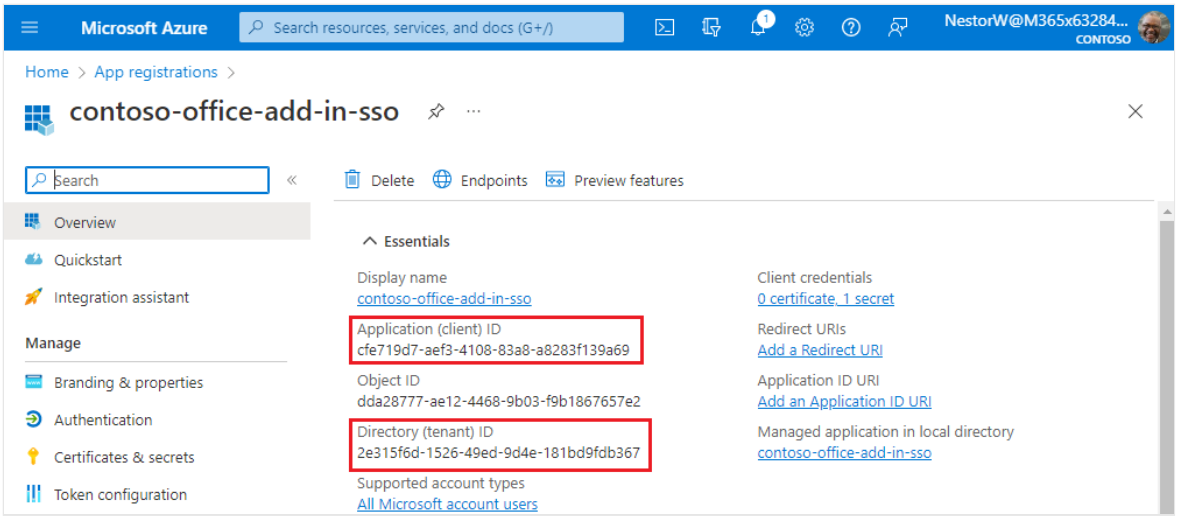
    - Set **Name** to `<add-in-name>`.
    - Set **Supported account types** to **Accounts in any organizational directory (any Azure AD directory - multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)**.
    - Set **Redirect URI** to use the platform **Single-page application (SPA)** and the URI to `https://<fully-qualified-domain-name>/dialog.html`.



5. Select **Register**. A message is displayed stating that the application registration was created.



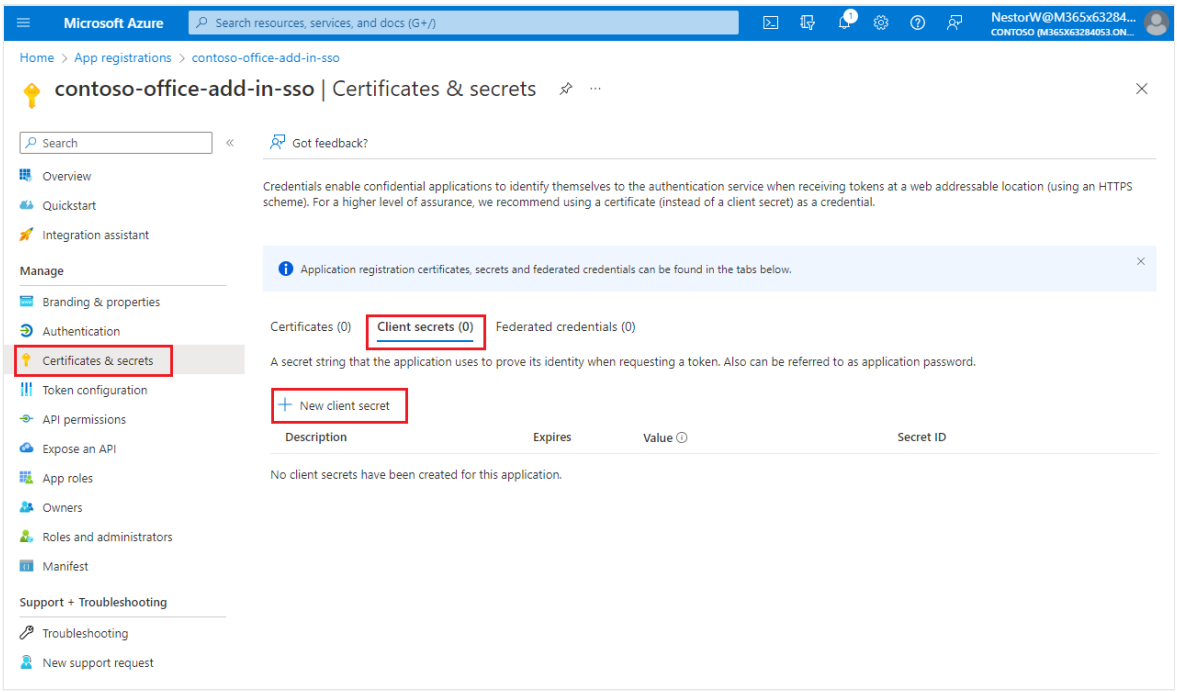
6. Copy and save the values for the **Application (client) ID** and the **Directory (tenant) ID**. You'll use both of them in later procedures.



Add a client secret

Sometimes called an *application password*, a client secret is a string value your app can use in place of a certificate to identity itself.

1. From the left pane, select **Certificates & secrets**. Then on the **Client secrets** tab, select **New client secret**.



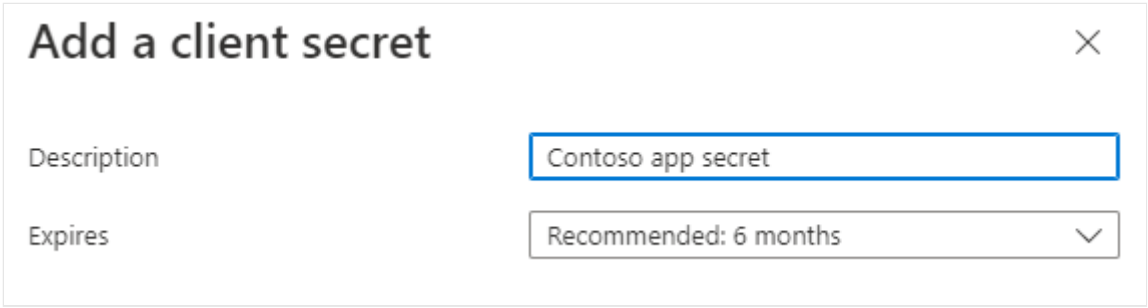
   The **Add a client secret** pane appears.

2. Add a description for your client secret.

3. Select an expiration for the secret or specify a custom lifetime.

   - Client secret lifetime is limited to two years (24 months) or less. You can't specify a custom lifetime longer than 24 months.

- Microsoft recommends that you set an expiration value of less than 12 months.



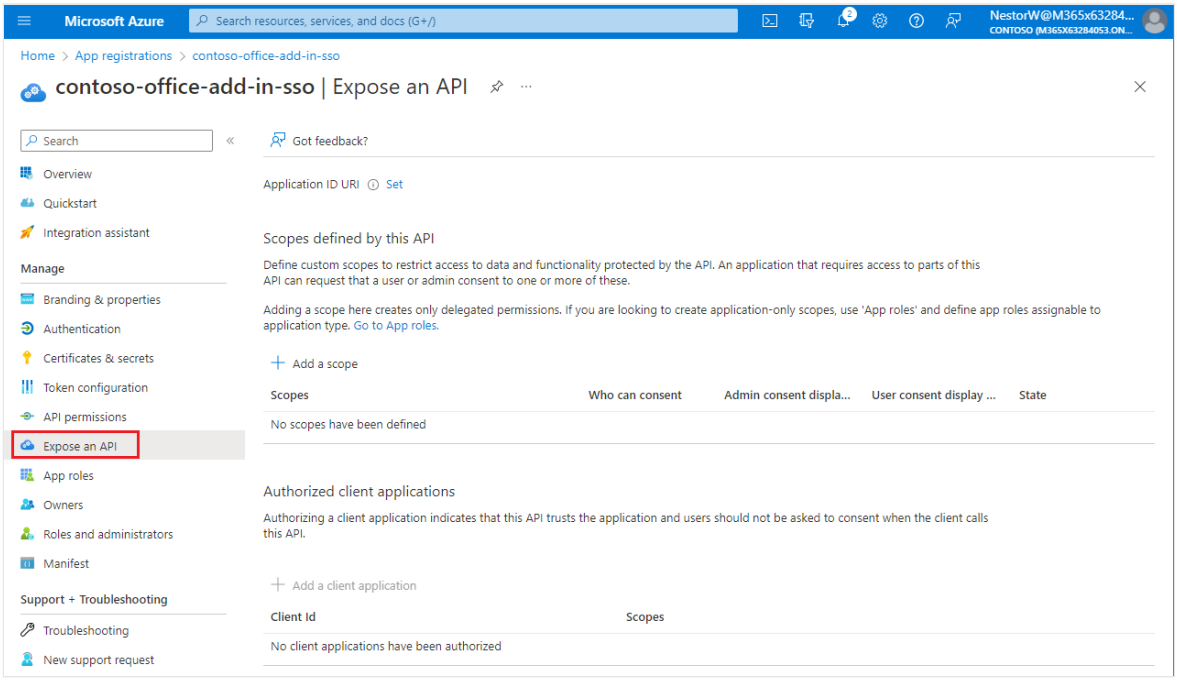
4. Select **Add**. The new secret is created and the value is temporarily displayed.

> **Important**
>
> *Record the secret's value* for use in your client application code. This secret value is *never displayed again* after you leave this pane.

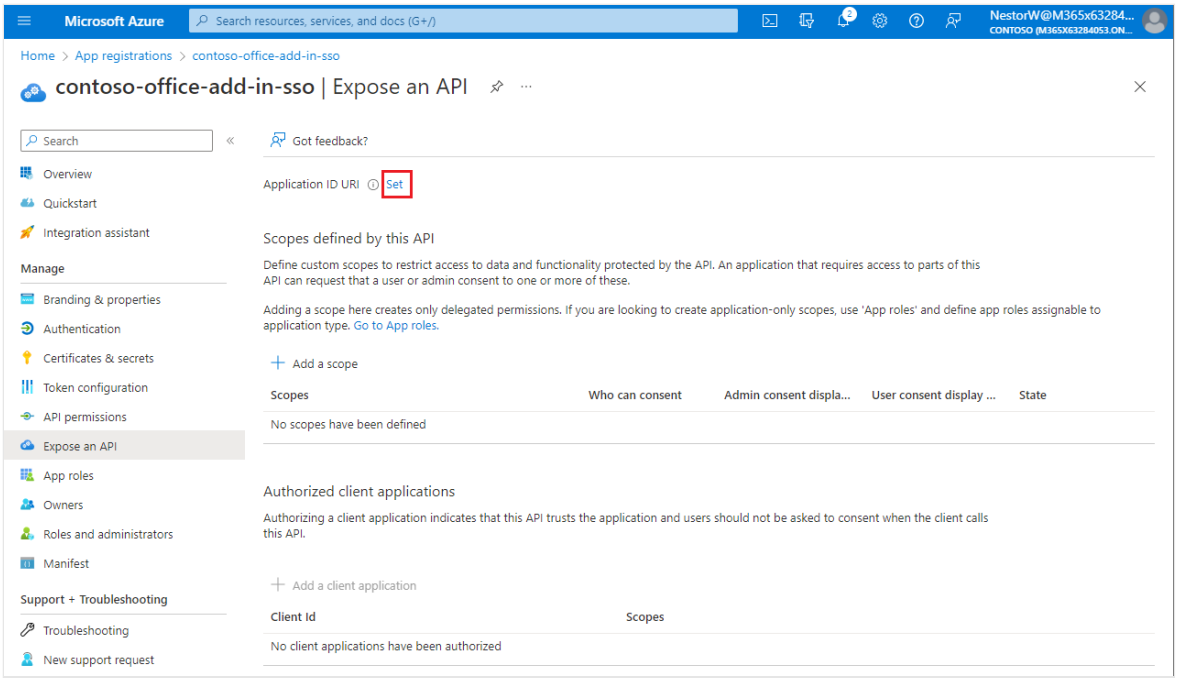
Expose a web API

1. From the left pane, select **Expose an API**.

   The **Expose an API** pane appears.

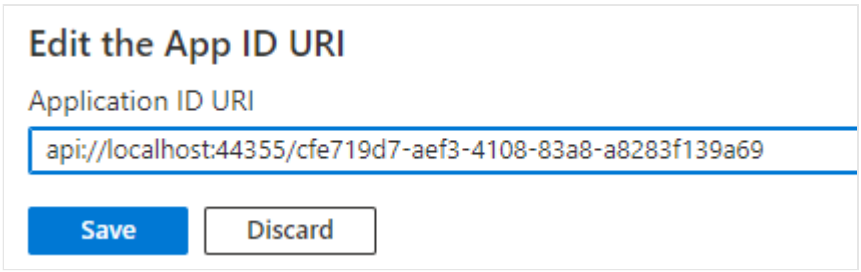


2. Select **Set** to generate an application ID URI.



The section for setting the application ID URI appears with a generated Application ID URI in the form `api://<app-id>`.

3. Update the application ID URI to `api://<fully-qualified-domain-name>/<app-id>`.



- The **Application ID URI** is pre-filled with app ID (GUID) in the format `api://<app-id>`.
- The application ID URI format should be: `api://<fully-qualified-domain-name>/<app-id>`
- Insert the `fully-qualified-domain-name` between `api://` and `<app-id>` (which is a GUID). For example, `api://contoso.com/<app-id>`.
- If you're using localhost, then the format should be `api://localhost:<port>/<app-id>`. For example, `api://localhost:3000/c6c1f32b-5e55-4997-881a-753cc1d563b7`.

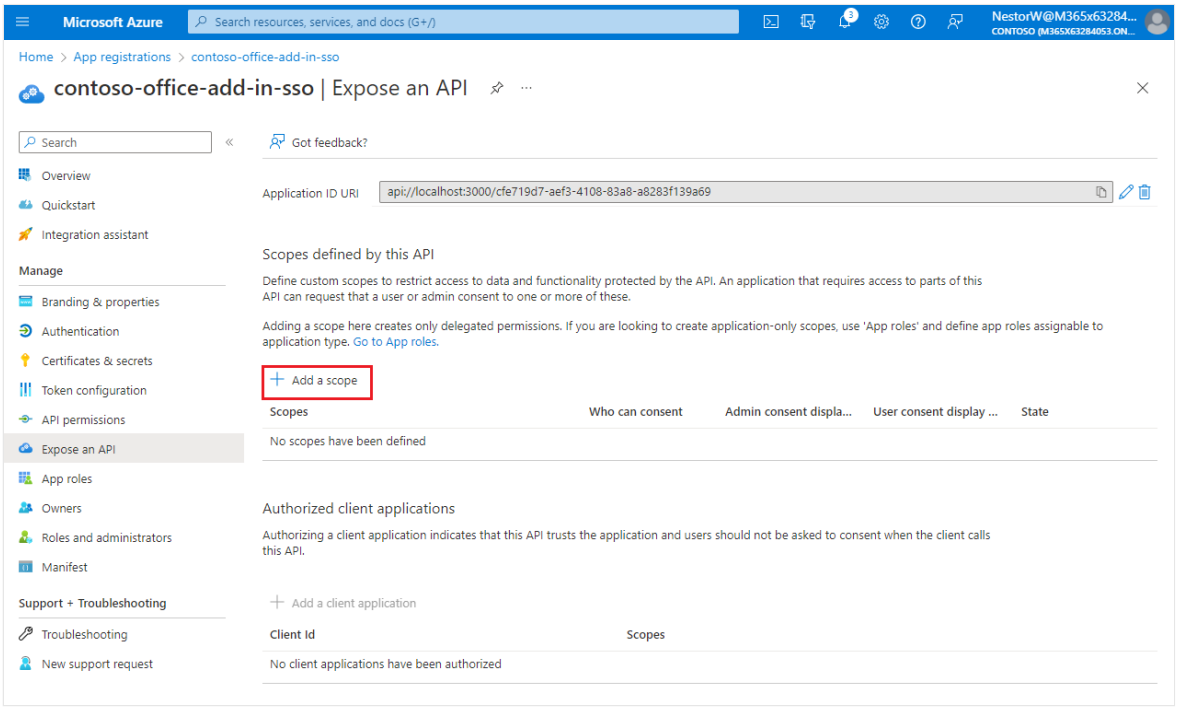
For additional application ID URI details, see Application manifest identifierUris attribute.

> **Note**

If you get an error saying that the domain is already owned but you own it, follow the procedure at **Quickstart: Add a custom domain name to Azure Active Directory** to register it, and then repeat this step. (This error can also occur if you are not signed in with credentials of an admin in the Microsoft 365 tenancy. See step 2. Sign out and sign in again with admin credentials and repeat the process from step 3.)

Add a scope

1. On the **Expose an API** page, select **Add a scope**.



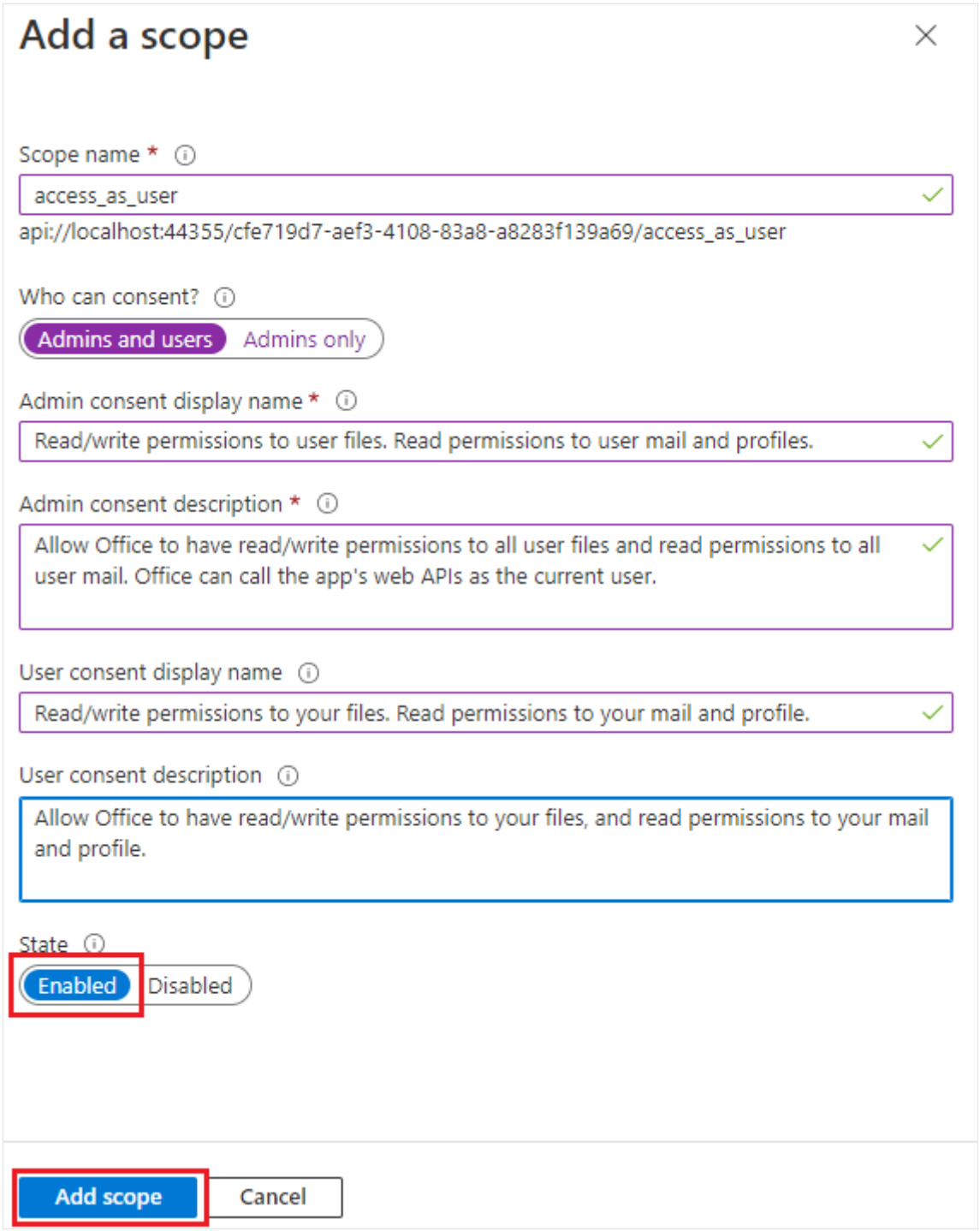
   The **Add a scope** pane opens.

2. In the **Add a scope** pane, specify the scope's attributes. The following table shows example values for and Outlook add-in requiring the `profile`, `openid`, `Files.ReadWrite`, and `Mail.Read` permissions. Modify the text to match the permissions your add-in needs.

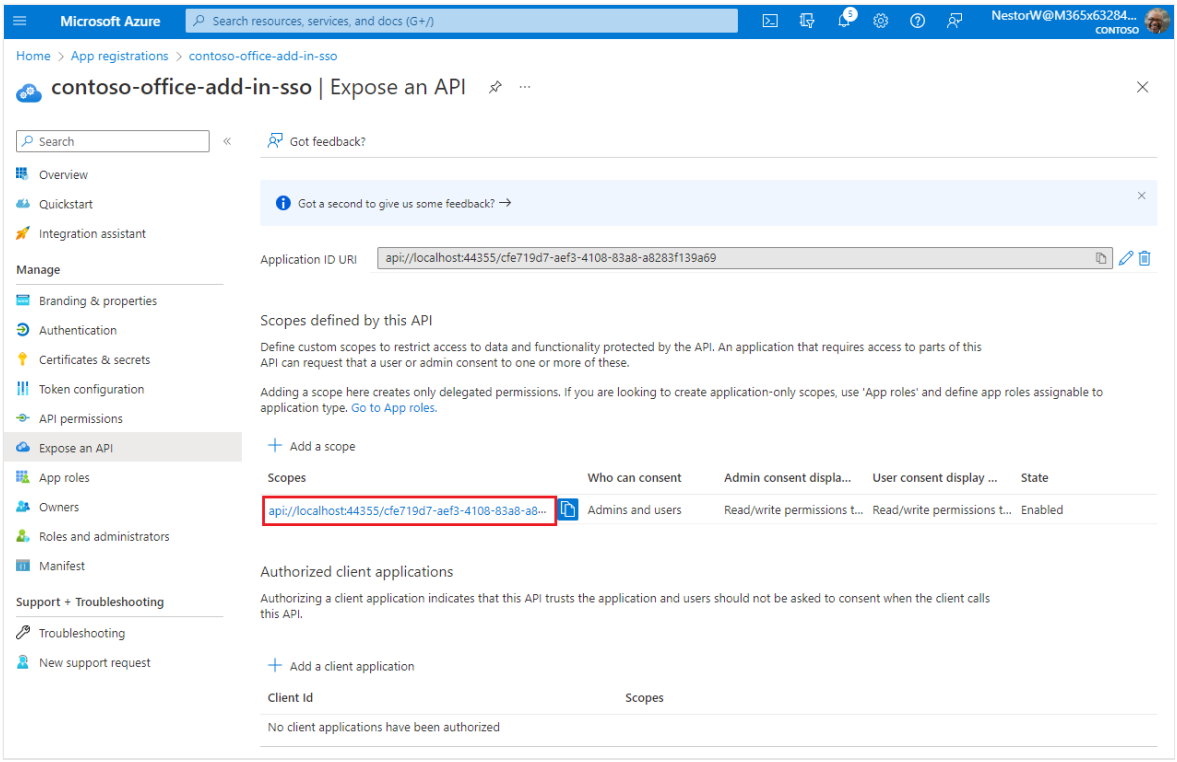
Field	Description	Values
Scope name	The name of your scope. A common scope naming convention is `resource.operation.constraint`.	For SSO this must be set to `access_as_user`.
Who can consent	Determines if admin consent is required or if users can consent without an admin approval.	For learning SSO and samples, we recommend you set this to **Admins and users**.

Field	Description	Values
		Select **Admins only** for higher-privileged permissions.
Admin consent display name	A short description of the scope's purpose visible to admins only.	`Read/write permissions to user files. Read permissions to user mail and profiles.`
Admin consent description	A more detailed description of the permission granted by the scope that only admins see.	`Allow Office to have read/write permissions to all user files and read permissions to all user mail. Office can call the app's web APIs as the current user.`
User consent display name	A short description of the scope's purpose. Shown to users only if you set **Who can consent** to **Admins and users**.	`Read/write permissions to your files. Read permissions to your mail and profile.`
User consent description	A more detailed description of the permission granted by the scope. Shown to users only if you set **Who can consent** to **Admins and users**.	`Allow Office to have read/write permissions to your files, and read permissions to your mail and profile.`

3. Set the **State** to **Enabled**, and then select **Add scope**.

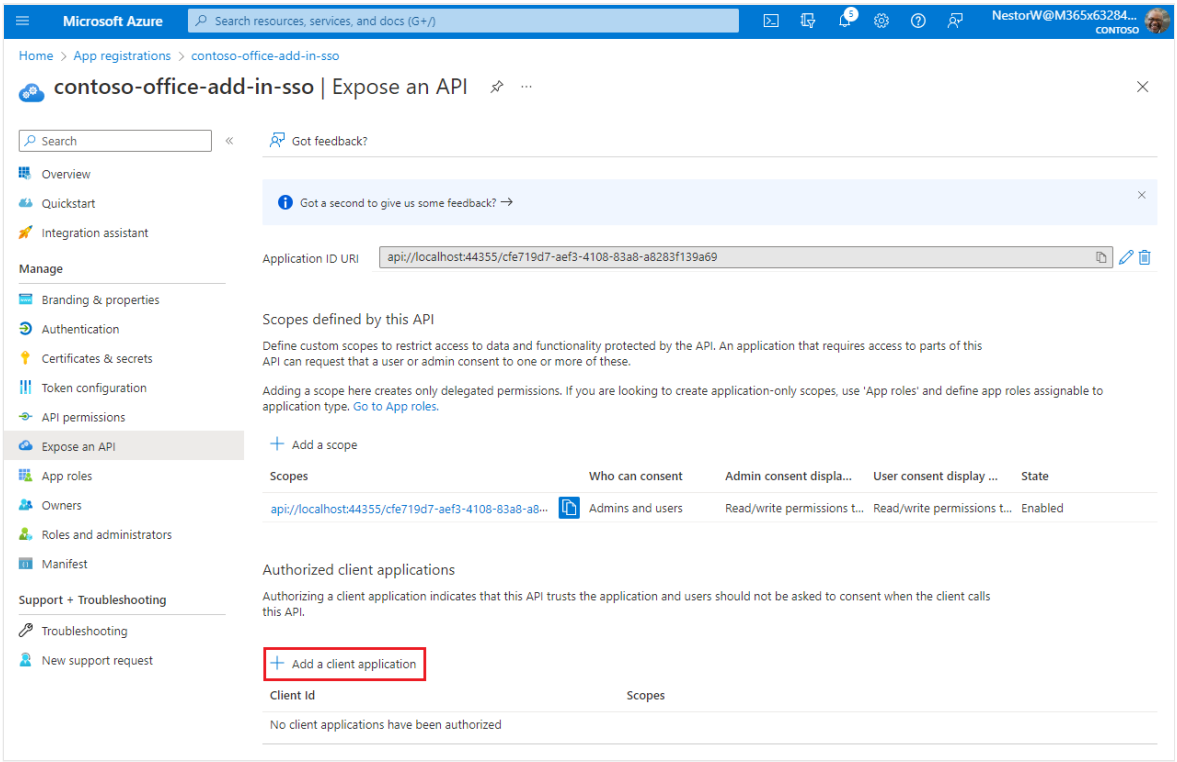


The new scope you defined displays on the pane.



> **Note**
>
> The domain part of the **Scope name** displayed just below the text field should automatically match the **Application ID URI** set in the previous step, with `/access_as_user` appended to the end; for example, `api://localhost:6789/c6c1f32b-5e55-4997-881a-753cc1d563b7/access_as_user`.

4. Select **Add a client application**.



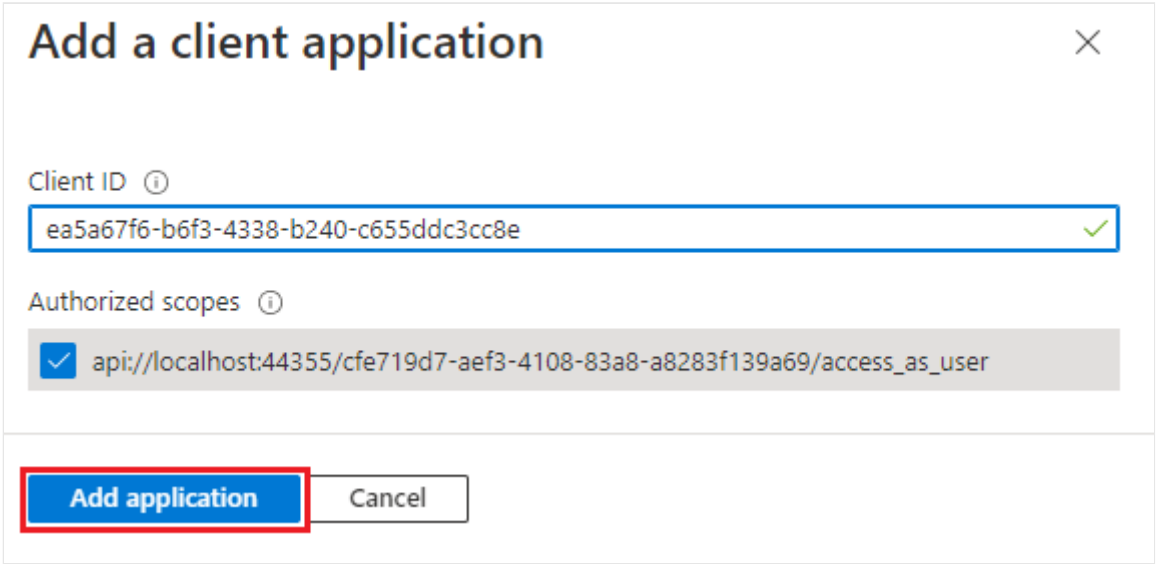
The **Add a client application** pane appears.

5. In the **Client ID** enter `ea5a67f6-b6f3-4338-b240-c655ddc3cc8e`. This value pre-authorizes all Microsoft Office application endpoints. If you also want to pre-authorize Office when used inside of Microsoft Teams, add `1fec8e78-bce4-4aaf-ab1b-5451cc387264` (Microsoft Teams desktop and Teams mobile) and `5e3ce6c0-2b1f-4285-8d4b-75ee78787346` (Teams on the web).

   > **Note**
   >
   > The `ea5a67f6-b6f3-4338-b240-c655ddc3cc8e` ID pre-authorizes Office on all the following platforms. Alternatively, you can enter a proper subset of the following IDs if, for any reason, you want to deny authorization to Office on some platforms. If you do so, leave out the IDs of the platforms from which you want to withhold authorization. Users of your add-in on those platforms will not be able to call your Web APIs, but other functionality in your add-in will still work.
   >
   > - `d3590ed6-52b3-4102-aeff-aad2292ab01c` (Microsoft Office)
   > - `93d53678-613d-4013-afc1-62e9e444a0a5` (Office on the web)
   > - `bc59ab01-8403-45c6-8796-ac3ef710b3e3` (Outlook on the web)

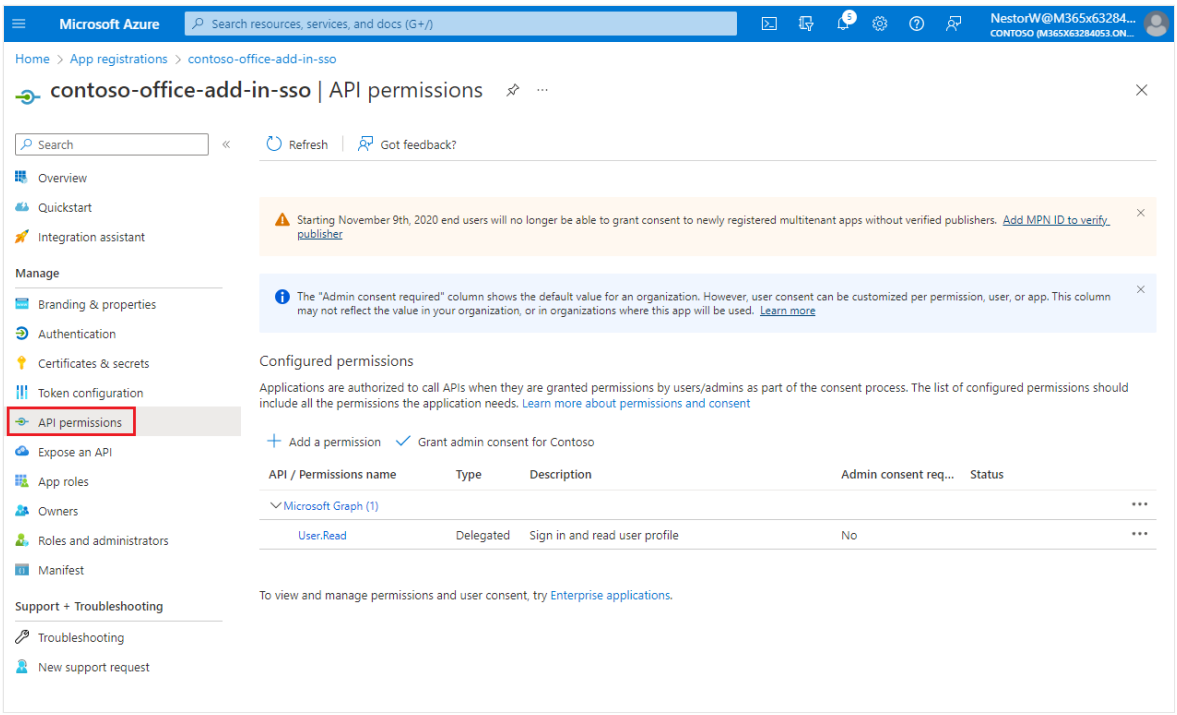
6. In **Authorized scopes**, select the `api://<fully-qualified-domain-name>/<app-id>/access_as_user` checkbox.

7. Select **Add application**.



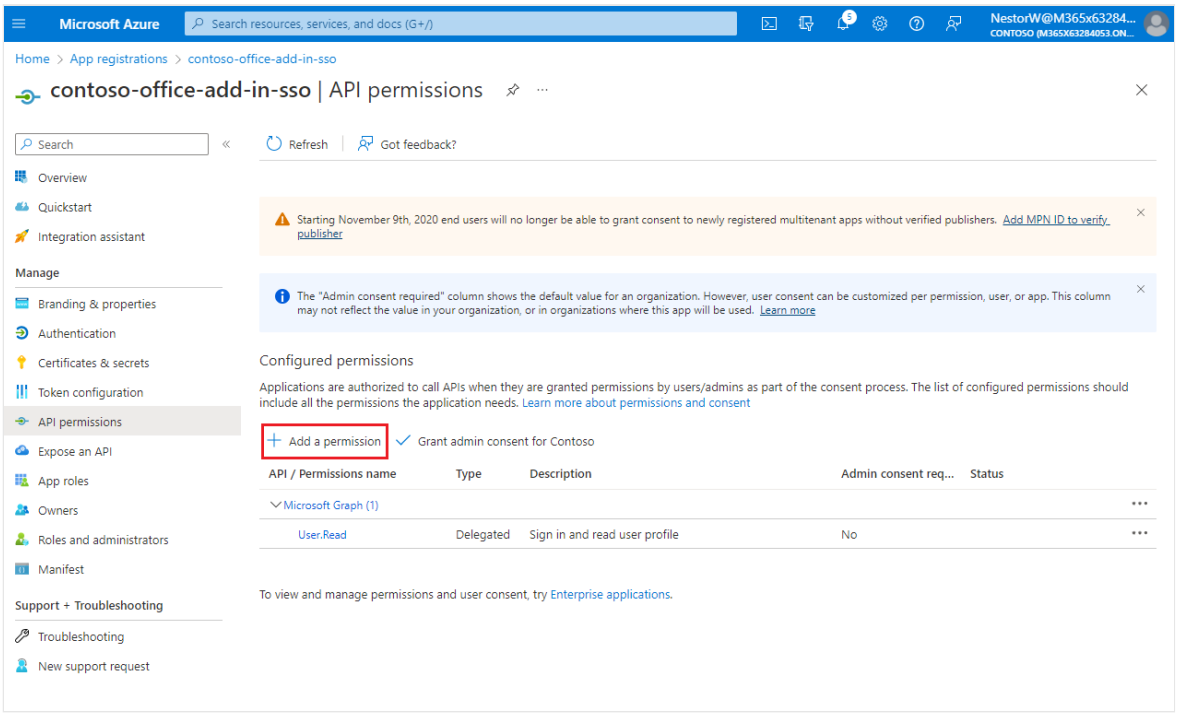
Add Microsoft Graph permissions

1. From the left pane, select **API permissions**.



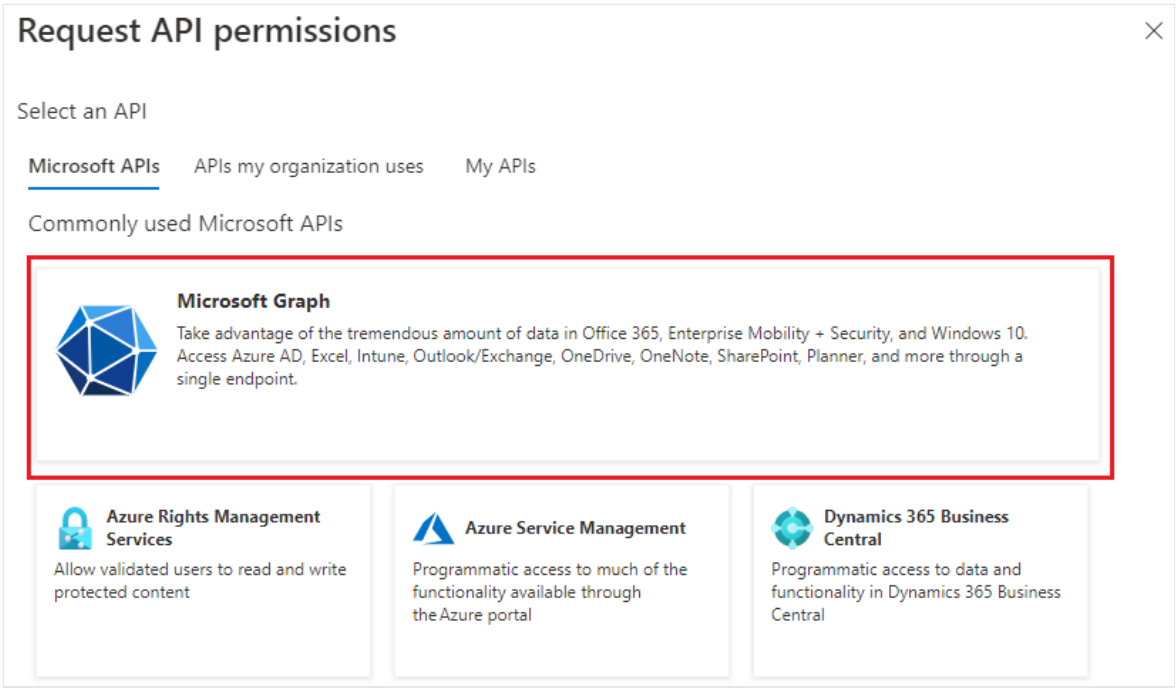
The **API permissions** pane opens.

2. Select **Add a permission**.

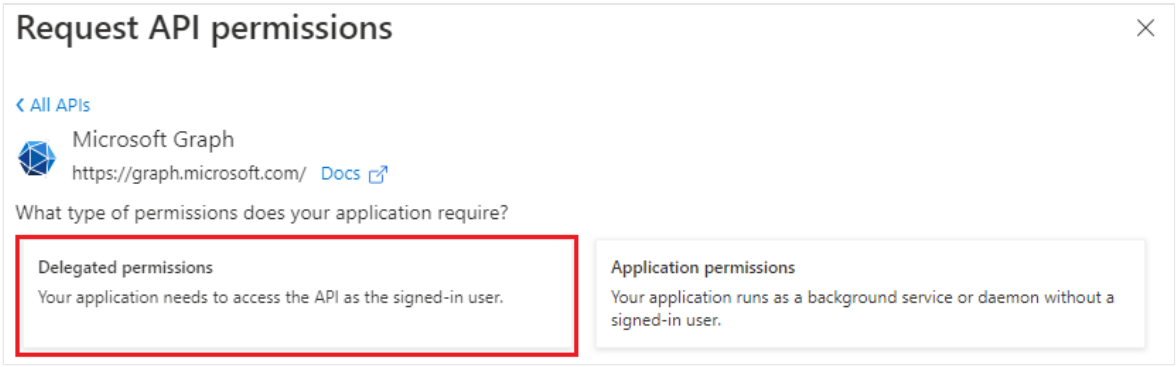


The **Request API permissions** pane opens.

3. Select **Microsoft Graph**.



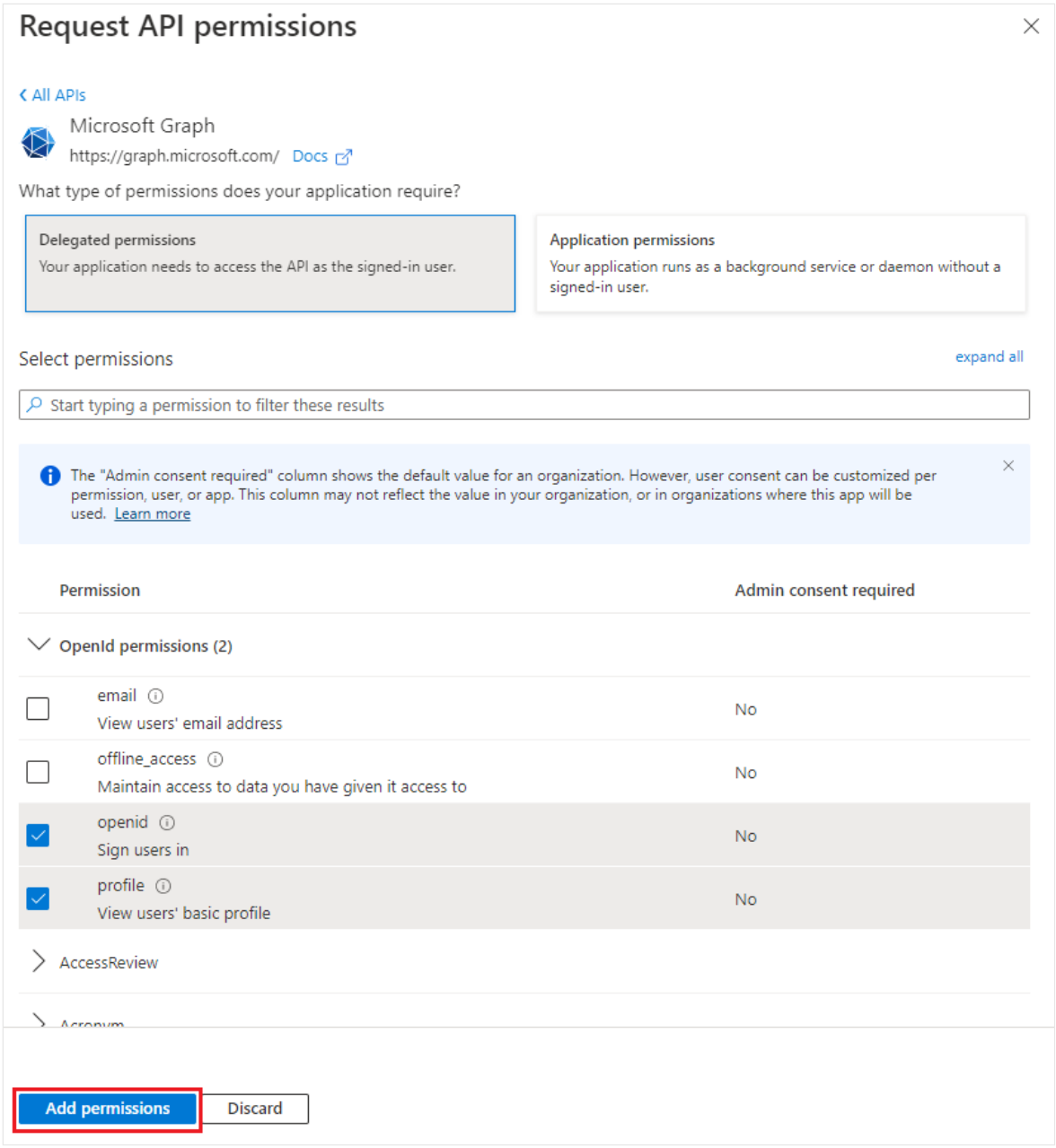
4. Select **Delegated permissions**.



5. In the **Select permissions** search box, search for the permissions your add-in needs. For example, for an Outlook add-in, you might use `profile`, `openid`, `Files.ReadWrite`, and `Mail.Read`.

> **Note**
>
> The `User.Read` permission may already be listed by default. It's a good practice to only request permissions that are needed, so we recommend that you uncheck the box for this permission if your add-in doesn't actually need it.

6. Select the checkbox for each permission as it appears. Note that the permissions will not remain visible in the list as you select each one. After selecting the permissions that your add-in needs, select **Add permissions**.



7. Select **Grant admin consent for [tenant name]**. Select **Yes** for the confirmation that appears.

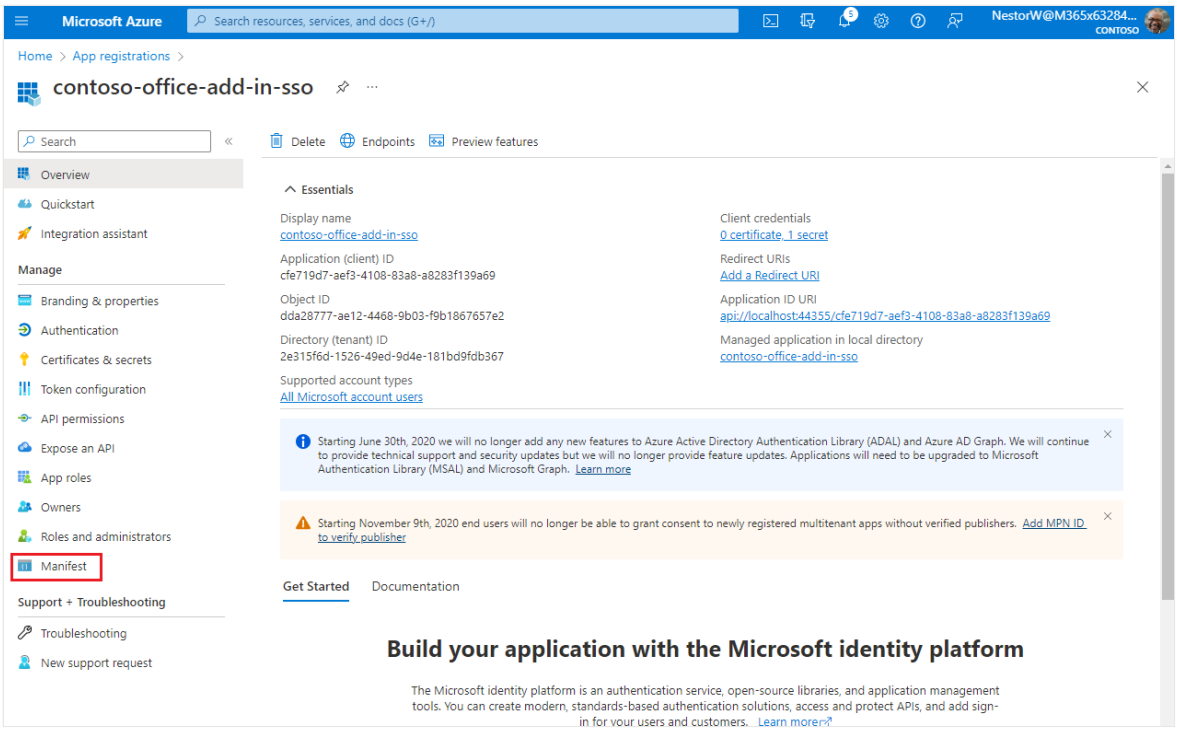
Configure access token version

You must define the access token version that is acceptable for your app. This configuration is made in the Azure Active Directory application manifest.

Define the access token version

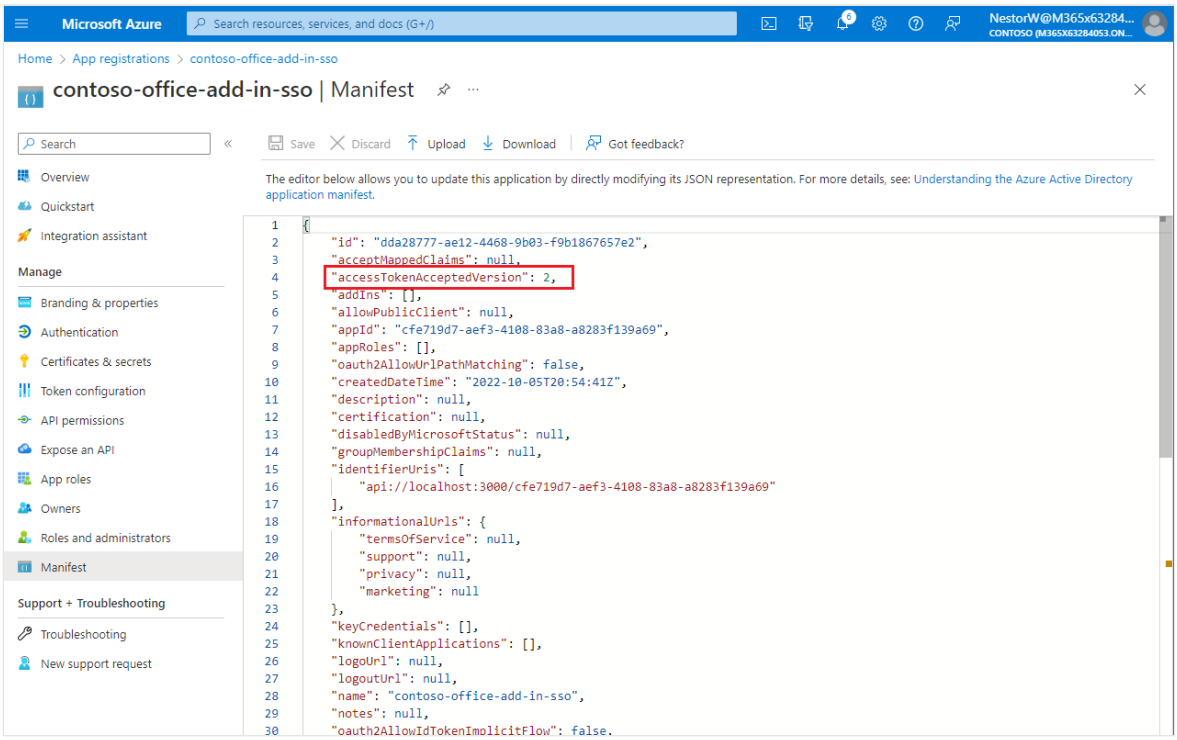
The access token version can change if you chose an account type other than **Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)**. Use the following steps to ensure the access token version is correct for Office SSO usage.

1. From the left pane, select **Manifest**.



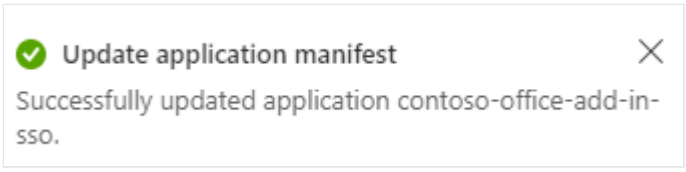
The Azure Active Directory application manifest appears.

2. Enter **2** as the value for the `accessTokenAcceptedVersion` property.



3. Select **Save**.

A message pops up on the browser stating that the manifest was updated successfully.



Congratulations! You've completed the app registration to enable SSO for your Office add-in.

Use SSO to get the identity of the signed-in user

06/24/2025

Use the `getAccessToken` API to get an access token that contains the identity for the current user signed in to Office. The access token is also an ID token because it contains identity claims about the signed-in user, such as their name and email. You can also use the ID token to identify the user when calling your own web services. To call `getAccessToken`, you must configure your Office Add-in to use SSO with Office.

In this article, you'll create an Office Add-in that gets the ID token, and displays the user's name, email, and unique ID in the task pane.

> **Note**
>
> SSO with Office and the `getAccessToken` API don't work in all scenarios. Always implement a fallback dialog to sign in the user when SSO is unavailable. For more information, see Authenticate and authorize with the Office dialog API.

Create an app registration

To use SSO with Office, you need to create an app registration in the Azure portal so the Microsoft identity platform can provide authentication and authorization services for your Office Add-in and its users.

1. To register your app, go to the Azure portal - App registrations page.
2. Sign in with the ***admin*** credentials to your Microsoft 365 tenancy. For example, MyName@contoso.onmicrosoft.com.
3. Select **New registration**. On the **Register an application** page, set the values as follows.
   - Set **Name** to `Office-Add-in-SSO`.
   - Set **Supported account types** to **Accounts in any organizational directory and personal Microsoft accounts (e.g. Skype, Xbox, Outlook.com)**.
   - Set the application type to **Web** and then set **Redirect URI** to `https://localhost:[port]/dialog.html`. Replace `[port]` with the correct port number for your web application. If you created the add-in using Yo Office, the port number is typically 3000 and found in the package.json file. If you created the add-in with Visual Studio 2019, the port is found in the **SSL URL** property of the web project.

- Choose **Register**.

4. On the **Office-Add-in-SSO** page, copy and save the values for the **Application (client) ID** and the **Directory (tenant) ID**. You'll use both of them in later procedures.

   > ⓘ **Note**
   >
   > This **Application (client) ID** is the "audience" value when other applications, such as the Office client application (e.g., PowerPoint, Word, Excel), seek authorized access to the application. It's also the "client ID" of the application when it, in turn, seeks authorized access to Microsoft Graph.

5. Select **Authentication** under **Manage**. In the **Implicit grant** section, enable the checkboxes for both **Access token** and **ID token**.

6. Select **Save** at the top of the form.

7. Select **Expose an API** under **Manage**. Select the **Set** link. This will generate the Application ID URI in the form `api://[app-id-guid]`, where `[app-id-guid]` is the **Application (client) ID**.

8. In the generated ID, insert `localhost:[port]/` (note the forward slash "/" appended to the end) between the double forward slashes and the GUID. Replace `[port]` with the correct port number for your web application. If you created the add-in using Yo Office, the port number is typically 3000 and found in the package.json file. If you created the add-in with Visual Studio 2019, the port is found in the **SSL URL** property of the web project.

   When you're finished, the entire ID should have the form `api://localhost:[port]/[app-id-guid]`; for example `api://localhost:44355/c6c1f32b-5e55-4997-881a-753cc1d563b7`.

9. Select the **Add a scope** button. In the panel that opens, enter `access_as_user` as the `<Scope>` name.

10. Set **Who can consent?** to **Admins and users**.

11. Fill in the fields for configuring the admin and user consent prompts with values that are appropriate for the `access_as_user` scope which enables the Office client application to use your add-in's web APIs with the same rights as the current user. Suggestions:

    - **Admin consent display name**: Office can act as the user.
    - **Admin consent description**: Enable Office to call the add-in's web APIs with the same rights as the current user.
    - **User consent display name**: Office can act as you.

- **User consent description**: Enable Office to call the add-in's web APIs with the same rights that you have.

12. Ensure that **State** is set to **Enabled**.

13. Select **Add scope** .

> ⓘ Note
>
> The domain part of the `<Scope>` name displayed just below the text field should automatically match the Application ID URI that you set earlier, with `/access_as_user` appended to the end; for example, `api://localhost:6789/c6c1f32b-5e55-4997-881a-753cc1d563b7/access_as_user`.

14. In the **Authorized client applications** section, enter the following ID to pre-authorize all Microsoft Office application endpoints.

    - `ea5a67f6-b6f3-4338-b240-c655ddc3cc8e` (All Microsoft Office application endpoints)

> ⓘ Note
>
> The `ea5a67f6-b6f3-4338-b240-c655ddc3cc8e` ID pre-authorizes Office on all the following platforms. Alternatively, you can enter a proper subset of the following IDs if for any reason you want to deny authorization to Office on some platforms. Just leave out the IDs of the platforms from which you want to withhold authorization. Users of your add-in on those platforms will not be able to call your Web APIs, but other functionality in your add-in will still work.
>
> - `d3590ed6-52b3-4102-aeff-aad2292ab01c` (Microsoft Office)
> - `93d53678-613d-4013-afc1-62e9e444a0a5` (Office on the web)
> - `bc59ab01-8403-45c6-8796-ac3ef710b3e3` (Outlook on the web)

15. Select the **Add a client application** button and then, in the panel that opens, set the `[app-id-guid]` to the Application (client) ID and check the box for `api://localhost:44355/[app-id-guid]/access_as_user`.

16. Select **Add application**.

17. Select **API permissions** under **Manage** and select **Add a permission**. On the panel that opens, choose **Microsoft Graph** and then choose **Delegated permissions**.

18. Use the **Select permissions** search box to search for the permissions your add-in needs. Search for and select the **profile** permission. The `profile` permission is required for the Office application to get a token to your add-in web application.

    - profile

    > ⓘ **Note**
    >
    > The `User.Read` permission may already be listed by default. It's a good practice not to ask for permissions that aren't needed, so we recommend that you uncheck the box for this permission if your add-in doesn't actually need it.

19. Select the **Add permissions** button at the bottom of the panel.

20. On the same page, choose the **Grant admin consent for <tenant-name>** button, and then select **Yes** for the confirmation that appears.

Create the Office Add-in

Visual Studio 2019

1. Start Visual Studio 2019 and choose to **Create a new project**.
2. Search for and select the **Excel Web Add-in** project template. Then choose **Next**. Note: SSO works with any Office application, but Excel is the application being used with this article.
3. Enter a project name, such as **sso-display-user-info**, and choose **Create**. You can leave the other fields at default values.
4. In the **Choose the add-in type** dialog box, select **Add new functionality to Excel**, and choose **Finish**.

The project is created and will contain two projects in the solution.

- **sso-display-user-info**: Contains the manifest and details for sideloading the add-in to Excel.
- **sso-display-user-infoWeb**: The ASP.NET project that hosts the web pages for the add-in.

Configure the manifest

Visual Studio 2019

In **Solution Explorer**, open **sso-display-user-info** > **sso-display-user-infoManifest** > **sso-display-user-info.xml**.

1. Near the bottom of the manifest is a closing `</Resources>` element. Insert the following XML just below the `</Resources>` element but before the closing `</VersionOverrides>` element. For Office applications other than Outlook, add the markup to the end of the `<VersionOverrides ... xsi:type="VersionOverridesV1_0">` section. For Outlook, add the markup to the end of the `<VersionOverrides ... xsi:type="VersionOverridesV1_1">` section.

   XML

   ```
   <WebApplicationInfo>
       <Id>[application-id]</Id>
       <Resource>api://localhost:[port]/[application-id]</Resource>
       <Scopes>
           <Scope>openid</Scope>
           <Scope>user.read</Scope>
           <Scope>profile</Scope>
       </Scopes>
   </WebApplicationInfo>
   ```

2. Replace `[port]` with the correct port number for your project. If you created the add-in using Yo Office, the port number is typically 3000 and found in the package.json file. If you created the add-in with Visual Studio 2019, the port is found in the **SSL URL** property of the web project.

3. Replace both `[application-id]` placeholders with the actual application ID from your app registration.

4. Save the file.

The XML you inserted contains the following elements and information.

- `<WebApplicationInfo>` - The parent of the following elements.
- `<Id>` - The client ID of the add-in This is an application ID that you obtain as part of registering the add-in. See Register an Office Add-in that uses single sign-on (SSO) with the Microsoft identity platform.
- `<Resource>` - The URL of the add-in. This is the same URI (including the `api:` protocol) that you used when registering the add-in in Microsoft Entra ID. The domain part of this URI must match the domain, including any subdomains, used in the URLs in the

`<Resources>` section of the add-in's manifest and the URI must end with the client ID in the `<Id>`.

- `<Scopes>` - The parent of one or more `<Scope>` elements.
- `<Scope>` - Specifies a permission that the add-in needs. The `profile` and `openID` permissions are always needed and may be the only permissions needed, if your add-in doesn't access Microsoft Graph. If it does, you also need `<Scope>` elements for the required Microsoft Graph permissions; for example, `User.Read`, `Mail.Read`. Libraries that you use in your code to access Microsoft Graph may need additional permissions. For example, Microsoft Authentication Library (MSAL) for .NET requires the `offline_access` permission. For more information, see Authorize to Microsoft Graph from an Office Add-in.

Add the jwt-decode package

You can call the `getAccessToken` API to get the ID token from Office. First, let's add the jwt-decode package to make it easier to decode and view the ID token.

Visual Studio 2019

1. Open the Visual Studio solution.
2. On the menu, choose **Tools** > **NuGet Package Manager** > **Package Manager Console**.
3. Enter the following command in the **Package Manager Console**.

   ```
   Install-Package jwt-decode -Projectname sso-display-user-infoWeb
   ```

Add UI to the task pane

Modify the task pane so that it can display the user information you'll get from the ID token.

Visual Studio 2019

1. Open the Home.html file.
2. Add the following script tag to the `<head>` section of the page. This will include the jwt-decode package was added earlier.

HTML

```
<script src="Scripts/jwt-decode-2.2.0.js" type="text/javascript">
</script>
```

3. Replace the `<body>` section with the following HTML.

HTML

```
<body>
  <h1>Welcome</h1>
  <p>
    Sign in to Office, then choose the <b>Get ID Token</b> button to see
your
    ID token information.
  </p>
  <button id="getIDToken">Get ID Token</button>
  <div>
    <span id="userInfo"></span>
  </div>
</body>
```

Call the getAccessToken API

The final step is to get the ID token by calling `getAccessToken`.

Visual Studio 2019

1. Open the **Home.js** file.

2. Replace the entire contents of the file with the following code.

JavaScript

```
(function () {
  "use strict";

  // The initialize function must be run each time a new page is loaded.
  Office.initialize = function (reason) {
    $(document).ready(function () {
      $("#getIDToken").on("click", getIDToken);
    });
  };

  async function getIDToken() {
```

```
    try {
      let userTokenEncoded = await OfficeRuntime.auth.getAccessToken({
        allowSignInPrompt: true,
      });
      let userToken = jwt_decode(userTokenEncoded);
      document.getElementById("userInfo").innerHTML =
        "name: " +
        userToken.name +
        "<br>email: " +
        userToken.preferred_username +
        "<br>id: " +
        userToken.oid;
      console.log(userToken);
    } catch (error) {
      document.getElementById("userInfo").innerHTML =
        "An error occurred. <br>Name: " +
        error.name +
        "<br>Code: " +
        error.code +
        "<br>Message: " +
        error.message;
      console.log(error);
    }
  }
})();
```

3. Save the file.

Run the add-in

Visual Studio 2019

Choose **Debug** > **Start Debugging**, or press `F5`.

1. When Excel starts, sign in to Office with the same tenant account you used to create the app registration.
2. On the **Home** ribbon, choose **Show Taskpane** to open the add-in.
3. In the add-in's task pane, choose **Get ID token**.

The add-in will display the name, email, and ID of the account you signed in with.

> **Note**
>
> If you encounter any errors, review the registration steps in this article for the app registration. Missing a detail when setting up the app registration is a common cause of

issues working with SSO. If you still can't get the add-in to run successfully, see Troubleshoot error messages for single sign-on (SSO).

Stop the add-in

Visual Studio 2019

Choose **Stop Debugging**, or press `Shift` + `F5`.

See also

Using claims to reliably identify a user (Subject and Object ID)

Authorize to Microsoft Graph with SSO

08/13/2025

Users sign in to Office using either their personal Microsoft account or their Microsoft 365 Education or work account. The best way for an Office Add-in to get authorized access to Microsoft Graph is to use the credentials from the user's Office sign on. This enables them to access their Microsoft Graph data without needing to sign in a second time.

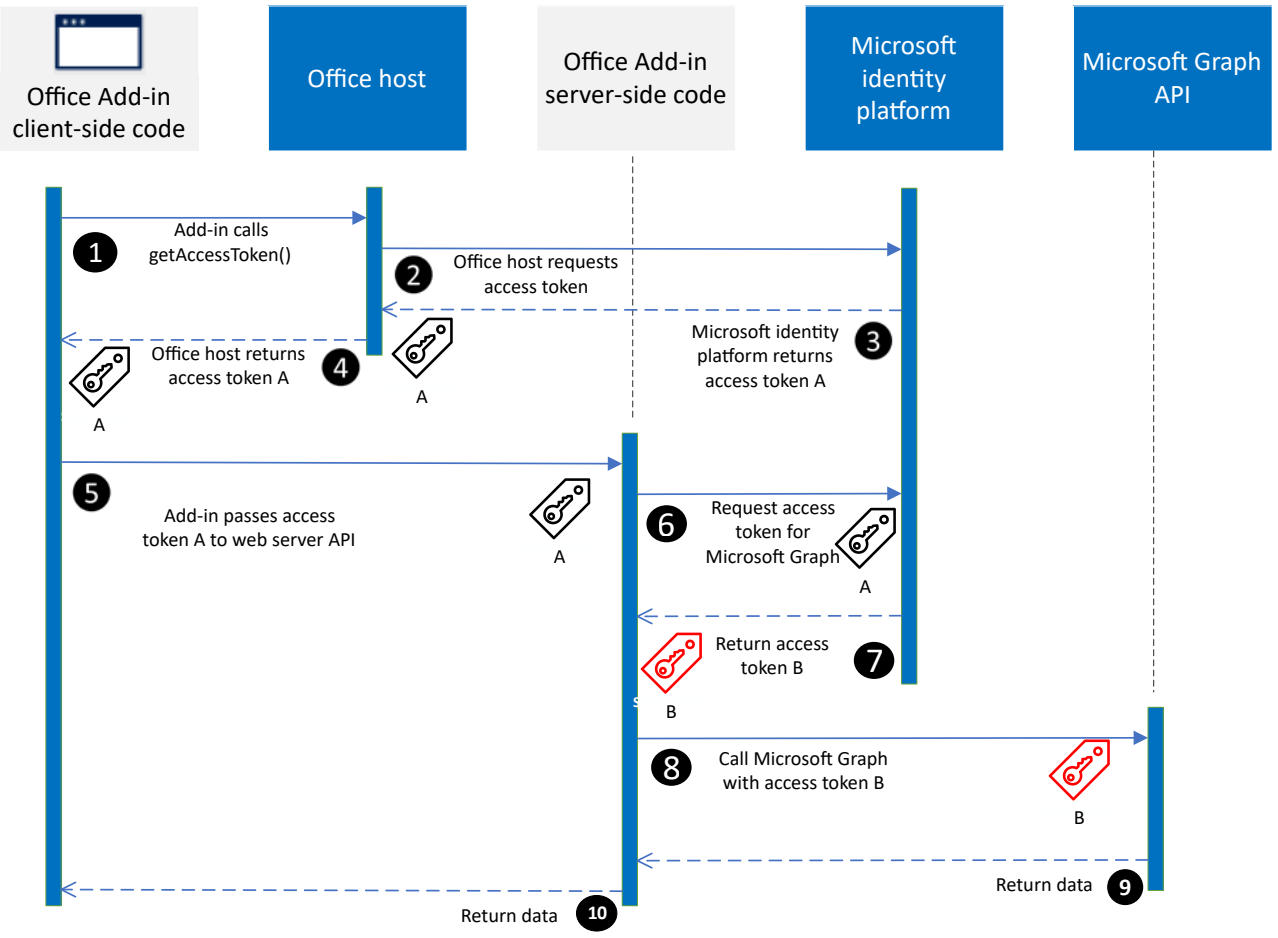
Add-in architecture for SSO and Microsoft Graph

In addition to hosting the pages and JavaScript of the web application, the add-in must also host, at the same fully qualified domain name, one or more web APIs that will get an access token to Microsoft Graph and make requests to it.

The add-in manifest contains a `<WebApplicationInfo>` element that provides important Azure app registration information to Office, including the permissions to Microsoft Graph that the add-in requires.

How it works at runtime

The following diagram shows the steps involved to sign in and access Microsoft Graph. The entire process uses OAuth 2.0 and JWT access tokens.



1. The client-side code of the add-in calls the Office.js API getAccessToken. This tells the Office host to obtain an access token for the add-in.

   If the user is not signed in, the Office host in conjunction with the Microsoft identity platform provides UI for the user to sign in and consent.

2. The Office host request an access token from the Microsoft identity platform.

3. The Microsoft identity platform returns access token *A* to the Office host. Access token *A* only provides access to the add-in's own server-side APIs. It does not provide access to Microsoft Graph.

4. The Office host returns access token *A* to the add-in's client-side code. Now the client-side code can make authenticated calls to the server-side APIs.

5. The client-side code makes an HTTP request to a web API on the server-side that requires authentication. It includes access token *A* as authorization proof. Server-side code validates access token *A*.

6. The server-side code uses the OAuth 2.0 On-Behalf-Of flow (OBO) to request a new access token with permissions to Microsoft Graph.

7. The Microsoft identity platform returns the new access token *B* with permissions to Microsoft Graph (and a refresh token, if the add-in requests *offline_access* permission). The server can optionally cache access token *B*.

8. The server-side code makes a request to a Microsoft Graph API and includes access token *B* with permissions to Microsoft Graph.

9. Microsoft Graph returns data back to the server-side code.

10. The server-side code returns the data back to the client-side code.

On subsequent requests the client code will always pass access token *A* when making authenticated calls to server-side code. The server-side code can cache token *B* so that it does not need to request it again on future API calls.

Develop an SSO add-in that accesses Microsoft Graph

You develop an add-in that accesses Microsoft Graph just as you would any other application that uses SSO. For a thorough description, see Enable single sign-on for Office Add-ins. The difference is that it is mandatory that the add-in have a server-side Web API.

Depending on your language and framework, libraries might be available that will simplify the server-side code you have to write. Your code should do the following:

- Validate the access token *A* every time it is passed from the client-side code. For more information, see Validate the access token.
- Initiate the OAuth 2.0 On-Behalf-Of flow (OBO) with a call to the Microsoft identity platform that includes the access token, some metadata about the user, and the credentials of the add-in (its ID and secret). For more information about the OBO flow, see Microsoft identity platform and OAuth 2.0 On-Behalf-Of flow.
- Optionally, after the flow completes, cache the returned access token *B* with permissions to Microsoft Graph. You'll want to do this if the add-in makes more than one call to Microsoft Graph. For more information, see Acquire and cache tokens using the Microsoft Authentication Library (MSAL)
- Create one or more Web API methods that get Microsoft Graph data by passing the (possibly cached) access token *B* to Microsoft Graph.

For examples of detailed walkthroughs and scenarios, see:

- Create a Node.js Office Add-in that uses single sign-on
- Create an ASP.NET Office Add-in that uses single sign-on
- Scenario: Implement single sign-on to your service in an Outlook add-in

Distributing SSO-enabled add-ins in Microsoft AppSource

When a Microsoft 365 admin acquires an add-in from AppSource, the admin can redistribute it through the integrated apps portal and grant admin consent to the add-in to access Microsoft Graph scopes. It's also possible, however, for the end user to acquire the add-in directly from AppSource, in which case the user must grant consent to the add-in. This can create a potential performance problem for which we've provided a solution.

If your code passes the `allowConsentPrompt` option in the call of `getAccessToken`, like `OfficeRuntime.auth.getAccessToken( { allowConsentPrompt: true } );`, then Office can prompt the user for consent if the Microsoft identity platform reports to Office that consent has not yet been granted to the add-in. However, for security reasons, Office can only prompt the user to consent to the Microsoft Graph `profile` scope. *Office cannot prompt for consent to other Microsoft Graph scopes*, not even `User.Read`. This means that if the user grants consent on the prompt, Office returns an access token. But the attempt to exchange the access token for a new access token with additional Microsoft Graph scopes fails with error AADSTS65001, which means consent (to Microsoft Graph scopes) has not been granted.

> **Note**
>
> The request for consent with `{ allowConsentPrompt: true }` could still fail even for the `profile` scope if the administrator has turned off end-user consent. For more information, see Configure how end-users consent to applications.

Your code can, and should, handle this error by falling back to an alternate system of authentication, which prompts the user for consent to Microsoft Graph scopes. For code examples, see Create a Node.js Office Add-in that uses single sign-on and Create an ASP.NET Office Add-in that uses single sign-on and the samples they link to. The entire process requires multiple round trips to the Microsoft identity platform. To avoid this performance penalty, include the `forMSGraphAccess` option in the call of `getAccessToken`; for example, `OfficeRuntime.auth.getAccessToken( { forMSGraphAccess: true } )`. This signals to Office that your add-in needs Microsoft Graph scopes. Office will ask the Microsoft identity platform to verify that consent to Microsoft Graph scopes has already been granted to the add-in. If it has, the access token is returned. If it hasn't, then the call of `getAccessToken` returns error 13012. Your code can handle this error by falling back to an alternate system of authentication immediately, without making a doomed attempt to exchange tokens with the Microsoft identity platform.

As a best practice, always pass `forMSGraphAccess` to `getAccessToken` when your add-in will be distributed in AppSource and needs Microsoft Graph scopes.

Details on SSO with an Outlook add-in

If you develop an Outlook add-in that uses SSO and you sideload it for testing, Office will *always* return error 13012 when `forMSGraphAccess` is passed to `getAccessToken` even if administrator consent has been granted. For this reason, you should comment out the `forMSGraphAccess` option **when developing** an Outlook add-in. Be sure to uncomment the option when you deploy for production. The bogus 13012 only happens when you are sideloading in Outlook.

For Outlook add-ins, be sure to enable Modern Authentication for the Microsoft 365 tenancy. For information about how to do this, see Enable or disable modern authentication for Outlook in Exchange Online.

Google Chrome third-party cookie support

Google Chrome is working to give users more control of their browsing experience. Users will be able to block third-party cookies in their Chrome browser. This will prevent your add-in from using any such cookies. This may cause issues when the add-in authenticates the user, such as multiple sign-on requests or errors.

For improved authentication experiences, see Using device state for an improved SSO experience on browsers with blocked third-party cookies.

For more information about the Google Chrome rollout, see A new path for Privacy Sandbox on the web.

See also

- OAuth2 Token Exchange
- Microsoft identity platform and OAuth 2.0 On-Behalf-Of flow
- IdentityAPI requirement sets

Create an ASP.NET Office Add-in that uses single sign-on

Article • 05/20/2023

Users can sign in to Office, and your Office Web Add-in can take advantage of this sign-in process to authorize users to your add-in and to Microsoft Graph without requiring users to sign in a second time. This article walks you through the process of enabling single sign-on (SSO) in an add-in.

The sample shows you how to build the following parts:

- Client-side code that provides a task pane that loads in Microsoft Excel, Word, or PowerPoint. The client-side code calls the Office JS API `getAccessToken()` to get the SSO access token to call server-side REST APIs.
- Server-side code that uses ASP.NET Core to provide a single REST API `/api/files`. The server-side code uses the Microsoft Authentication Library for .NET (MSAL.NET) for all token handling, authentication, and authorization.

The sample uses SSO and the On-Behalf-Of (OBO) flow to obtain correct access tokens and call Microsoft Graph APIs. If you are unfamiliar with how this flow works, see How SSO works at runtime for more detail.

Prerequisites

- Visual Studio 2019 or later.
- The **Office/SharePoint development** workload when configuring Visual Studio.
- At least a few files and folders stored on OneDrive for Business in your Microsoft 365 subscription.
- A build of Microsoft 365 that supports the IdentityAPI 1.3 requirement set. You might qualify for a Microsoft 365 E5 developer subscription, which includes a developer sandbox, through the Microsoft 365 Developer Program; for details, see the FAQ. The developer sandbox includes a Microsoft Azure subscription that you can use for app registrations in later steps in this article. If you prefer, you can use a separate Microsoft Azure subscription for app registrations. Get a trial subscription at Microsoft Azure.

Set up the starter project

Clone or download the repo at Office Add-in ASPNET SSO.

> **Note**
>
> There are two versions of the sample.
>
> - The **Begin** folder is a starter project. The UI and other aspects of the add-in that are not directly connected to SSO or authorization are already done. Later sections of this article walk you through the process of completing it.
> - The **Complete** folder contains the same sample with all coding steps from this article completed. To use the completed version, just follow the instructions in this article, but replace "Begin" with "Complete" and skip the sections **Code the client side** and **Code the server side**.

Use the following values for placeholders for the subsequent app registration steps.

Expand table

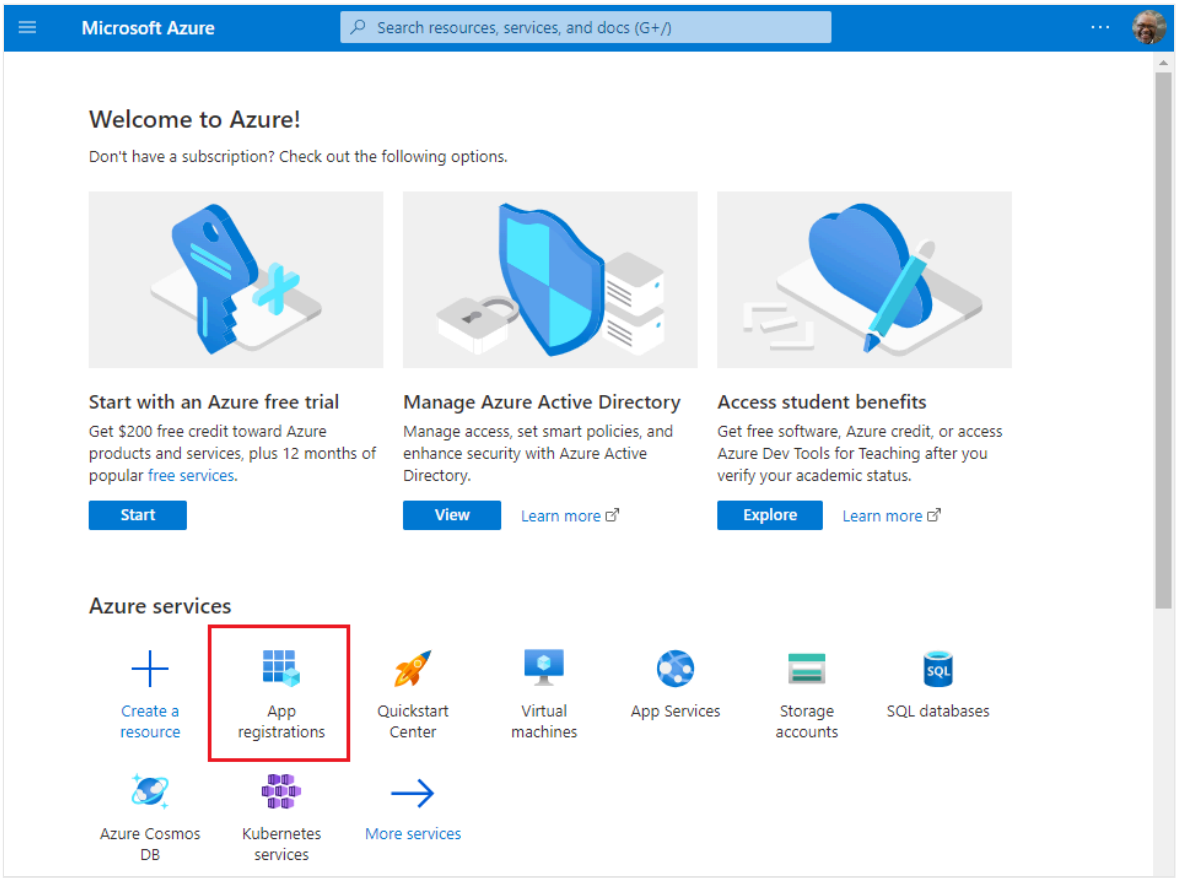
Placeholder	Value
`<add-in-name>`	**Office-Add-in-ASPNET-SSO**
`<fully-qualified-domain-name>`	`localhost:44355`
Microsoft Graph permissions	profile, openid, Files.Read

Register the add-in with Microsoft identity platform

You need to create an app registration in Azure that represents your web server. This enables authentication support so that proper access tokens can be issued to the client code in JavaScript. This registration supports both SSO in the client, and fallback authentication using the Microsoft Authentication Library (MSAL).

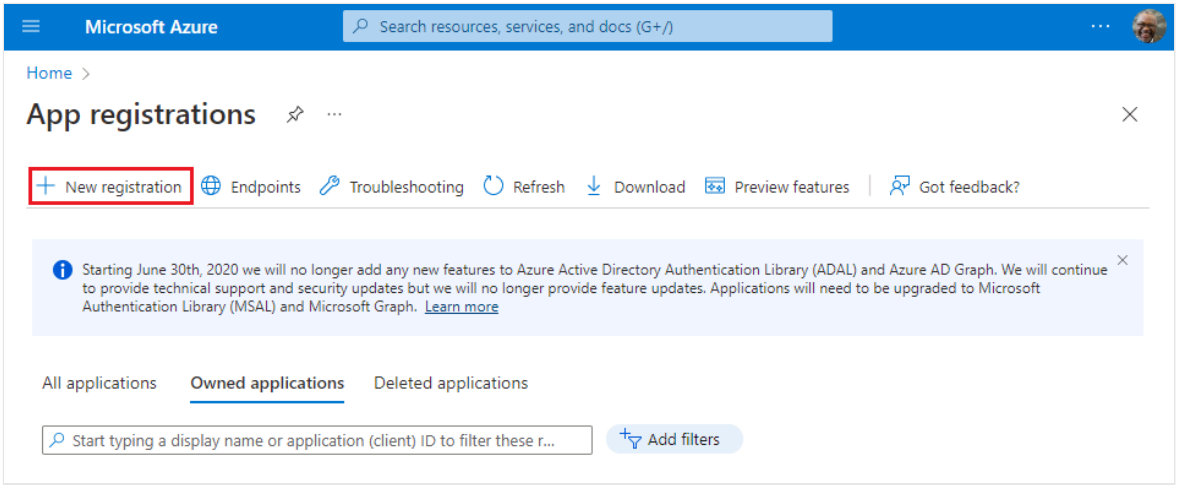
1. Sign in to the Azure portal with the ***admin*** credentials to your Microsoft 365 tenancy. For example, **MyName@contoso.onmicrosoft.com**.

2. Select **App registrations**. If you don't see the icon, search for "app registration" in the search bar.



The **App registrations** page appears.

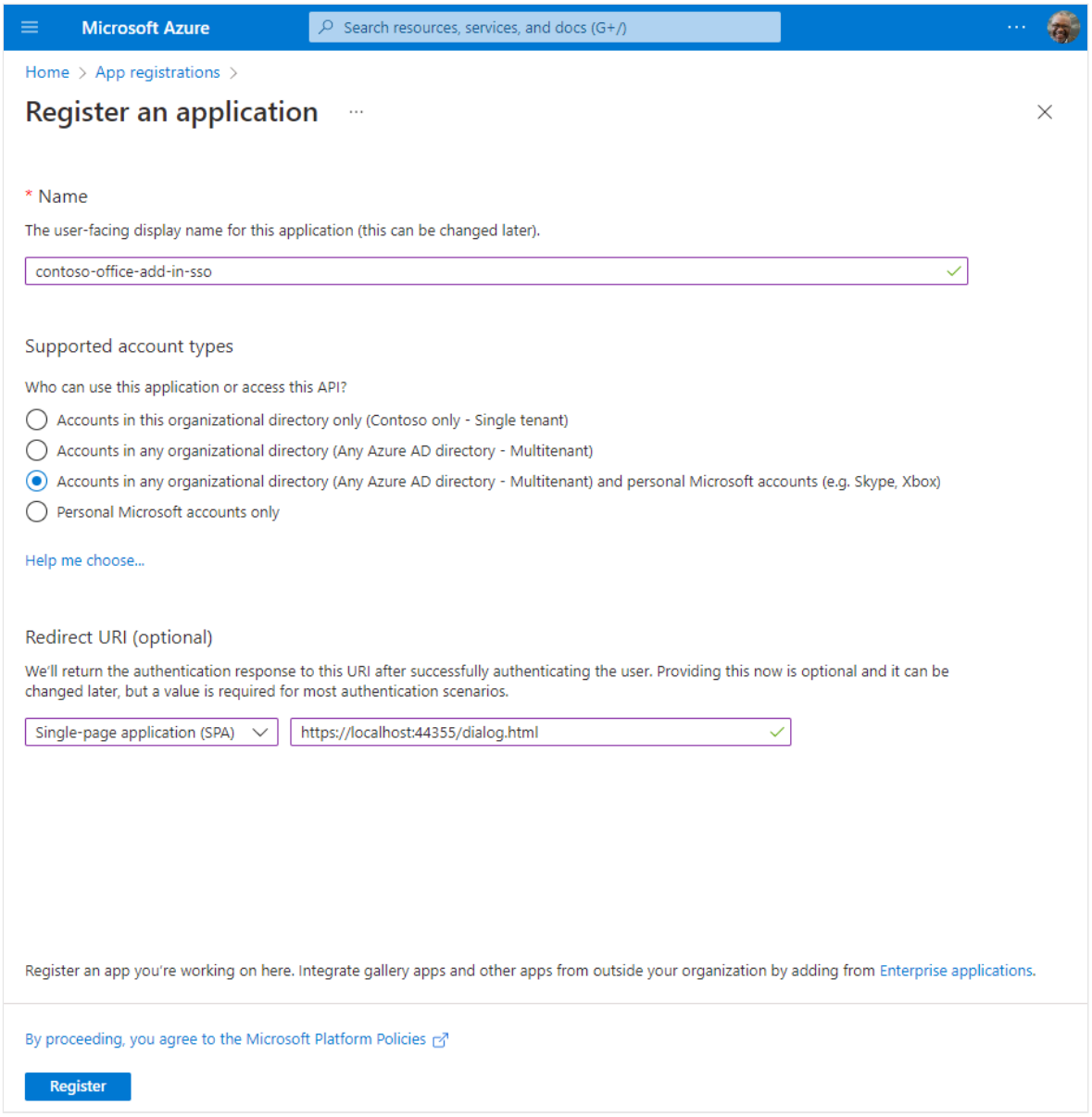
3. Select **New registration**.



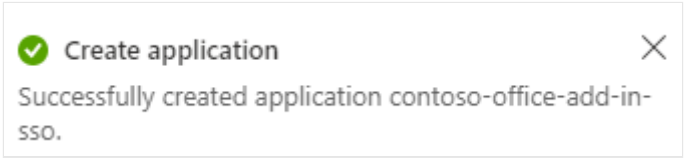
The **Register an application** page appears.

4. On the **Register an application** page, set the values as follows.

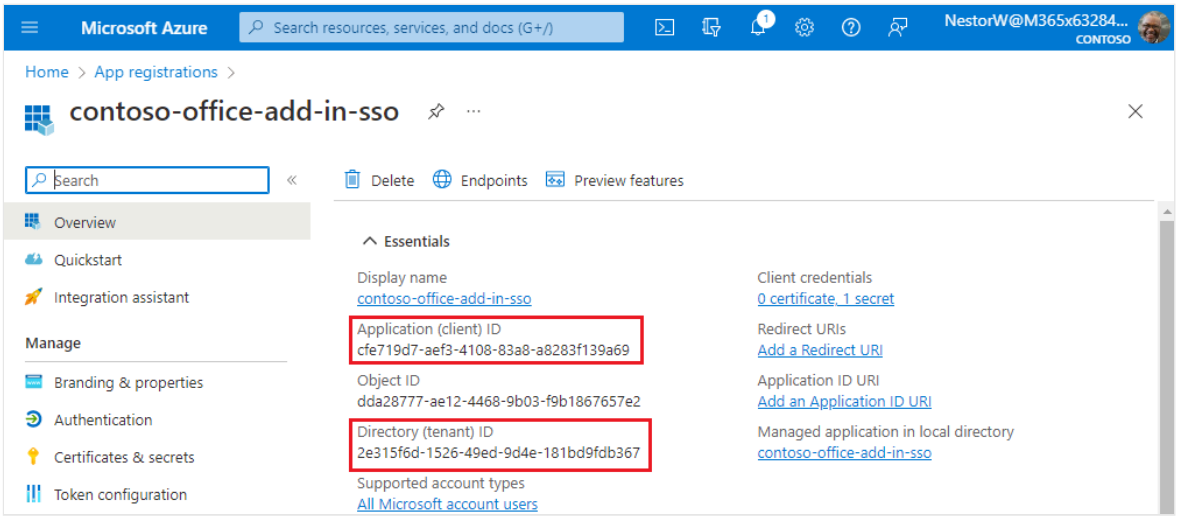
    - Set **Name** to `<add-in-name>`.
    - Set **Supported account types** to **Accounts in any organizational directory (any Azure AD directory - multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)**.
    - Set **Redirect URI** to use the platform **Single-page application (SPA)** and the URI to `https://<fully-qualified-domain-name>/dialog.html`.



5. Select **Register**. A message is displayed stating that the application registration was created.



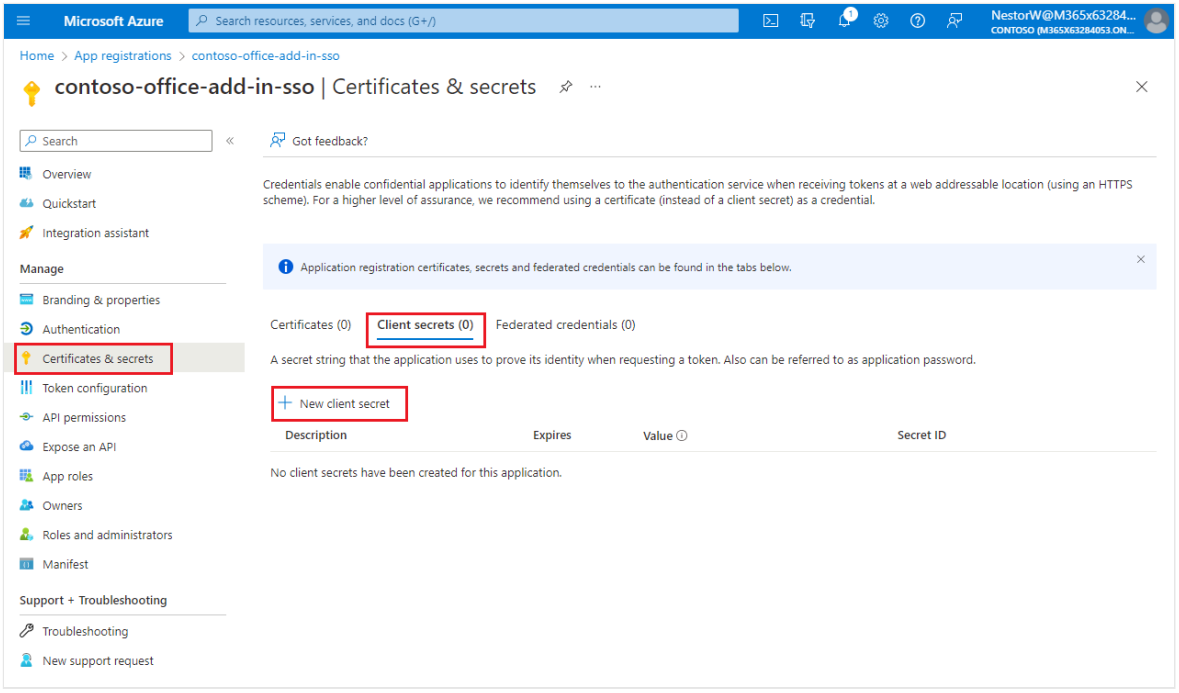
6. Copy and save the values for the **Application (client) ID** and the **Directory (tenant) ID**. You'll use both of them in later procedures.



Add a client secret

Sometimes called an *application password*, a client secret is a string value your app can use in place of a certificate to identity itself.

1. From the left pane, select **Certificates & secrets**. Then on the **Client secrets** tab, select **New client secret**.



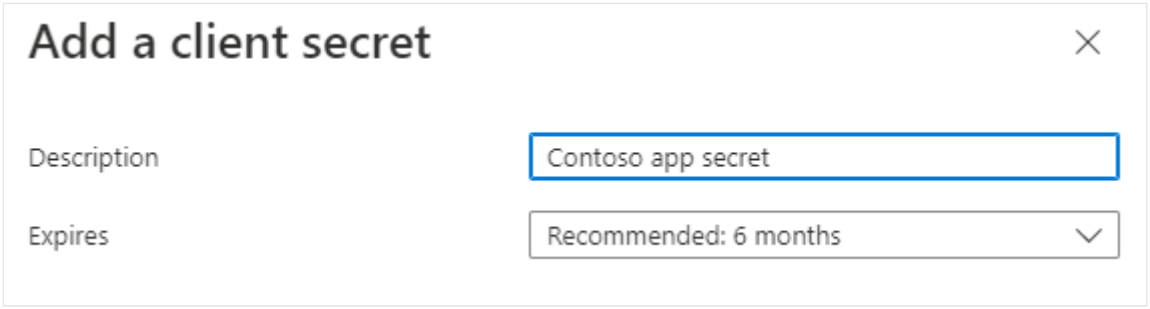
   The **Add a client secret** pane appears.

2. Add a description for your client secret.

3. Select an expiration for the secret or specify a custom lifetime.

   - Client secret lifetime is limited to two years (24 months) or less. You can't specify a custom lifetime longer than 24 months.

- Microsoft recommends that you set an expiration value of less than 12 months.



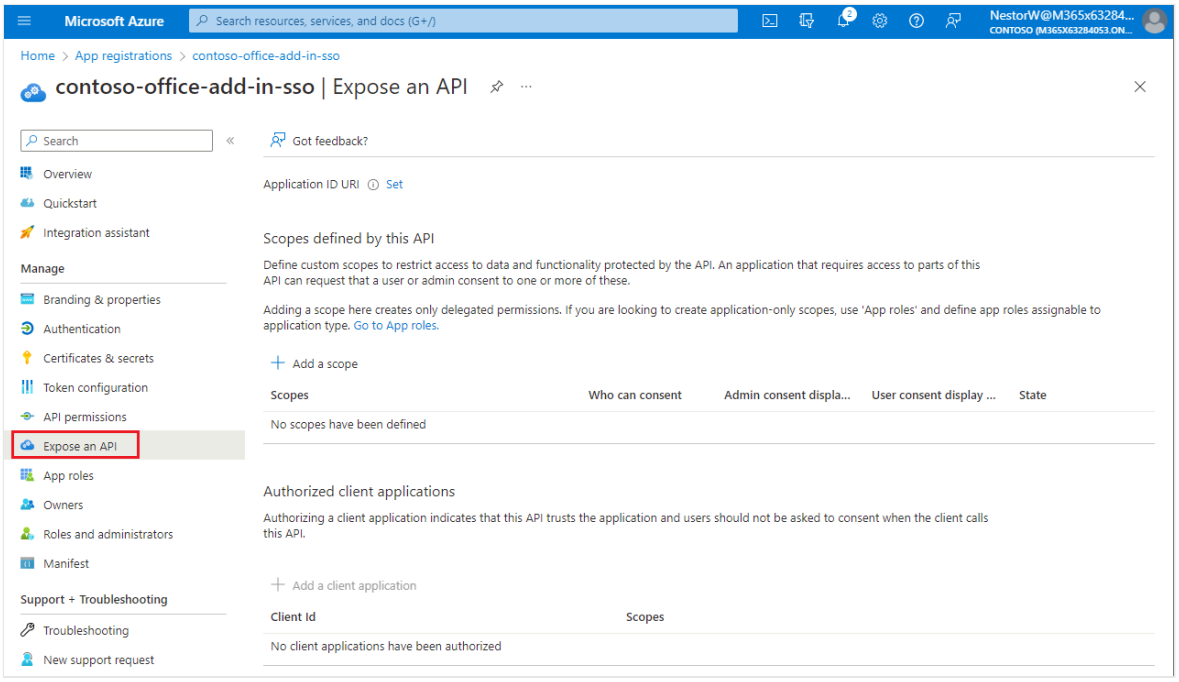
4. Select **Add**. The new secret is created and the value is temporarily displayed.

> **ⓘ Important**
>
> *Record the secret's value* for use in your client application code. This secret value is *never displayed again* after you leave this pane.

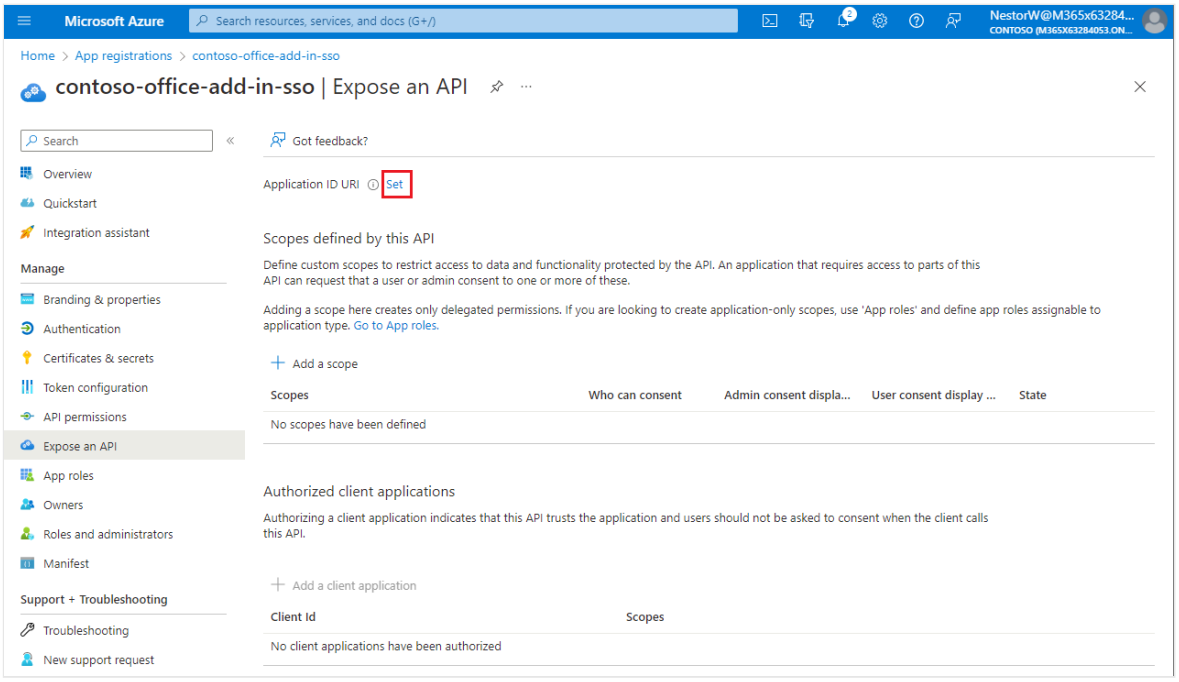
Expose a web API

1. From the left pane, select **Expose an API**.

   The **Expose an API** pane appears.

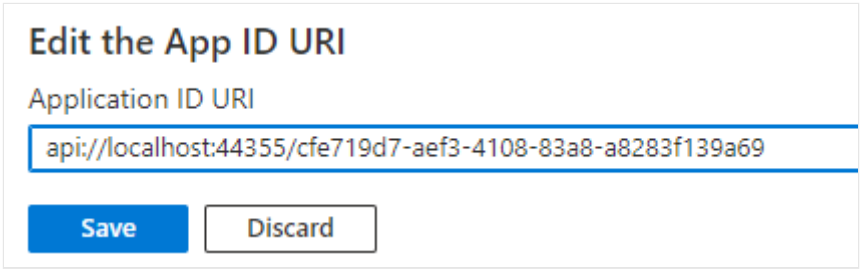


2. Select **Set** to generate an application ID URI.



The section for setting the application ID URI appears with a generated Application ID URI in the form `api://<app-id>`.

3. Update the application ID URI to `api://<fully-qualified-domain-name>/<app-id>`.



- The **Application ID URI** is pre-filled with app ID (GUID) in the format `api://<app-id>`.
- The application ID URI format should be: `api://<fully-qualified-domain-name>/<app-id>`
- Insert the `fully-qualified-domain-name` between `api://` and `<app-id>` (which is a GUID). For example, `api://contoso.com/<app-id>`.
- If you're using localhost, then the format should be `api://localhost:<port>/<app-id>`. For example, `api://localhost:3000/c6c1f32b-5e55-4997-881a-753cc1d563b7`.

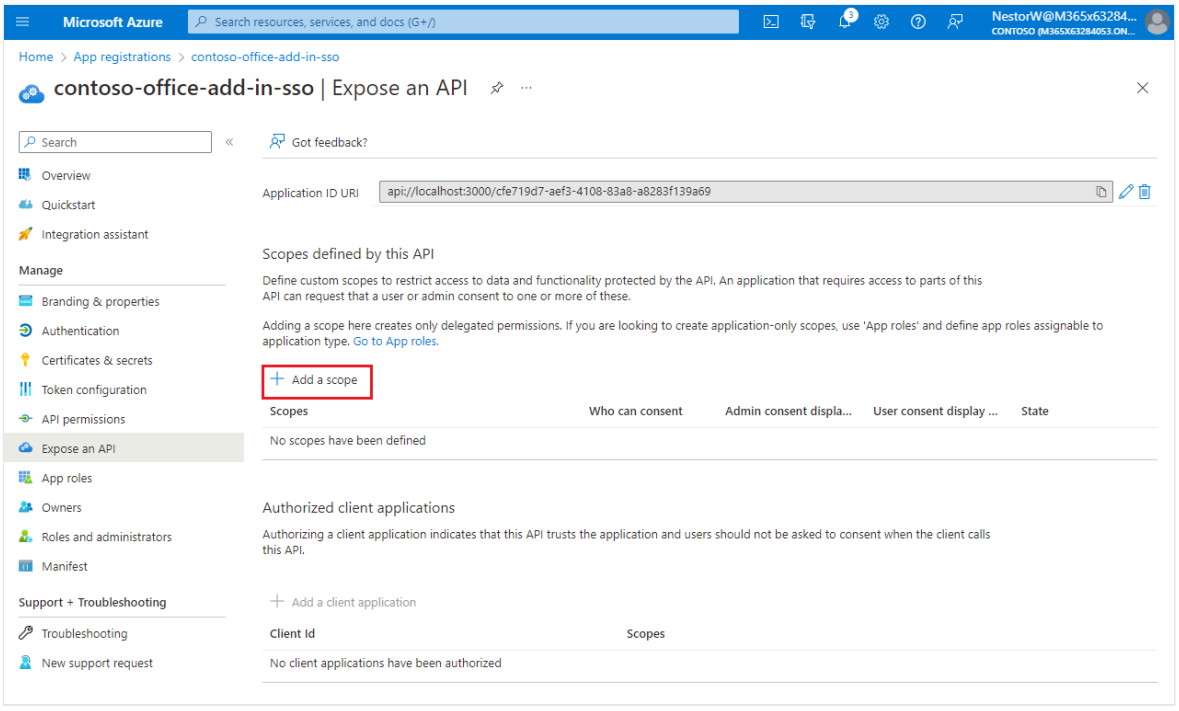
For additional application ID URI details, see Application manifest identifierUris attribute.

> **Note**

If you get an error saying that the domain is already owned but you own it, follow the procedure at Quickstart: Add a custom domain name to Azure Active Directory to register it, and then repeat this step. (This error can also occur if you are not signed in with credentials of an admin in the Microsoft 365 tenancy. See step 2. Sign out and sign in again with admin credentials and repeat the process from step 3.)

Add a scope

1. On the **Expose an API** page, select **Add a scope**.



   The **Add a scope** pane opens.

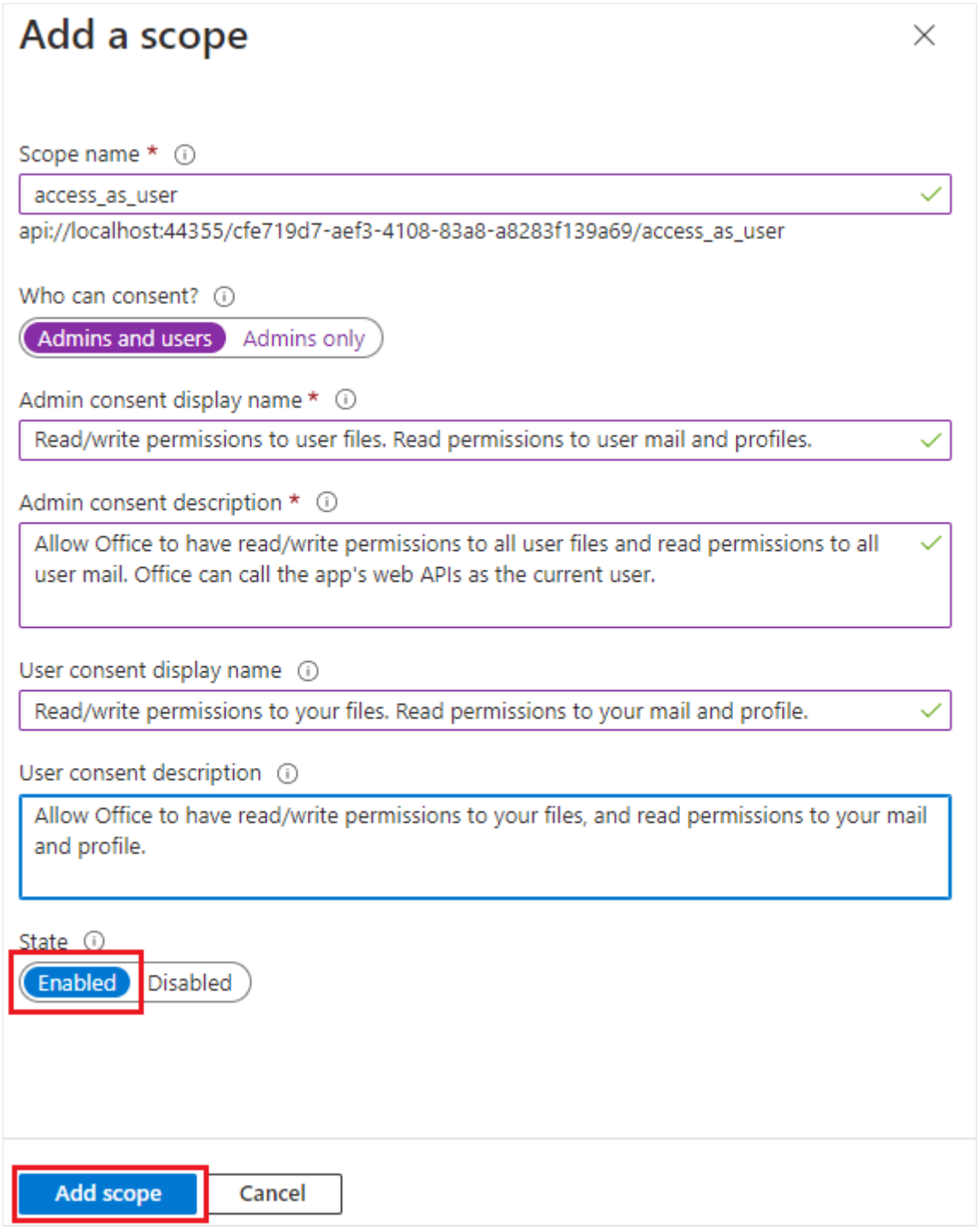
2. In the **Add a scope** pane, specify the scope's attributes. The following table shows example values for and Outlook add-in requiring the `profile`, `openid`, `Files.ReadWrite`, and `Mail.Read` permissions. Modify the text to match the permissions your add-in needs.

Expand table

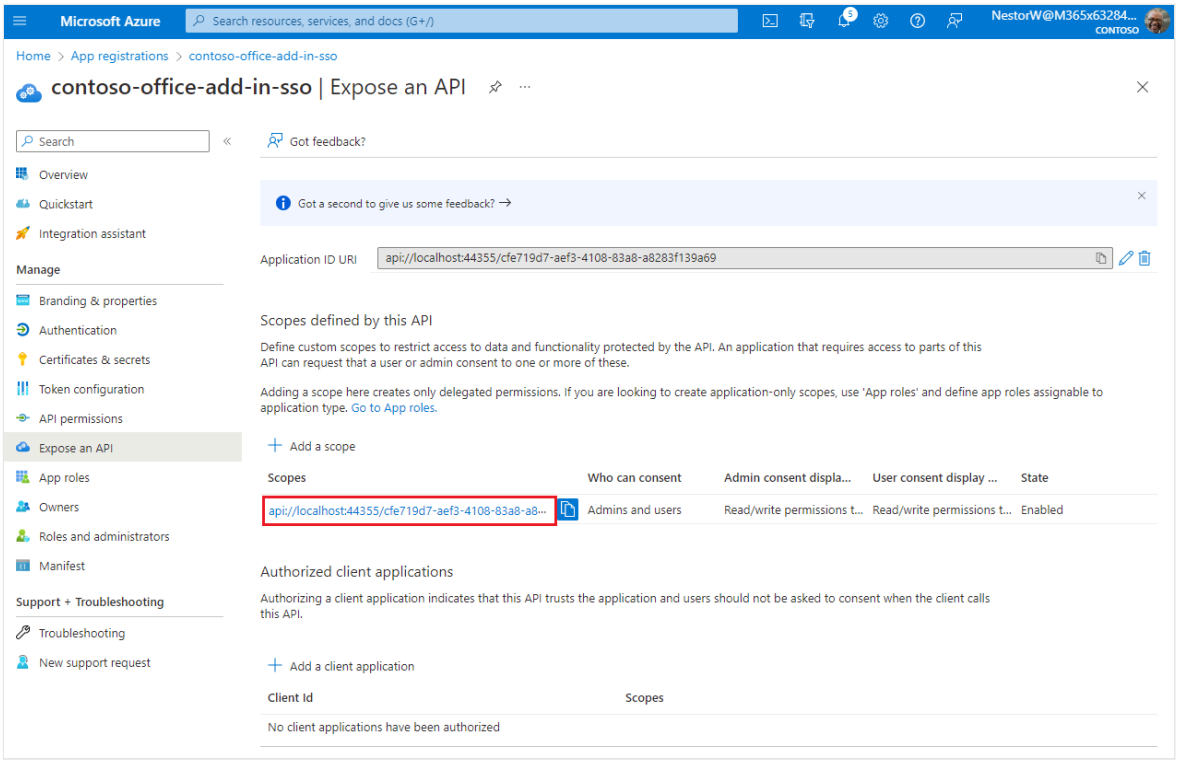
Field	Description	Values
Scope name	The name of your scope. A common scope naming convention is `resource.operation.constraint`.	For SSO this must be set to `access_as_user`.

Field	Description	Values
Who can consent	Determines if admin consent is required or if users can consent without an admin approval.	For learning SSO and samples, we recommend you set this to **Admins and users**. Select **Admins only** for higher-privileged permissions.
Admin consent display name	A short description of the scope's purpose visible to admins only.	`Read/write permissions to user files. Read permissions to user mail and profiles.`
Admin consent description	A more detailed description of the permission granted by the scope that only admins see.	`Allow Office to have read/write permissions to all user files and read permissions to all user mail. Office can call the app's web APIs as the current user.`
User consent display name	A short description of the scope's purpose. Shown to users only if you set **Who can consent** to **Admins and users**.	`Read/write permissions to your files. Read permissions to your mail and profile.`
User consent description	A more detailed description of the permission granted by the scope. Shown to users only if you set **Who can consent** to **Admins and users**.	`Allow Office to have read/write permissions to your files, and read permissions to your mail and profile.`

3. Set the **State** to **Enabled**, and then select **Add scope**.

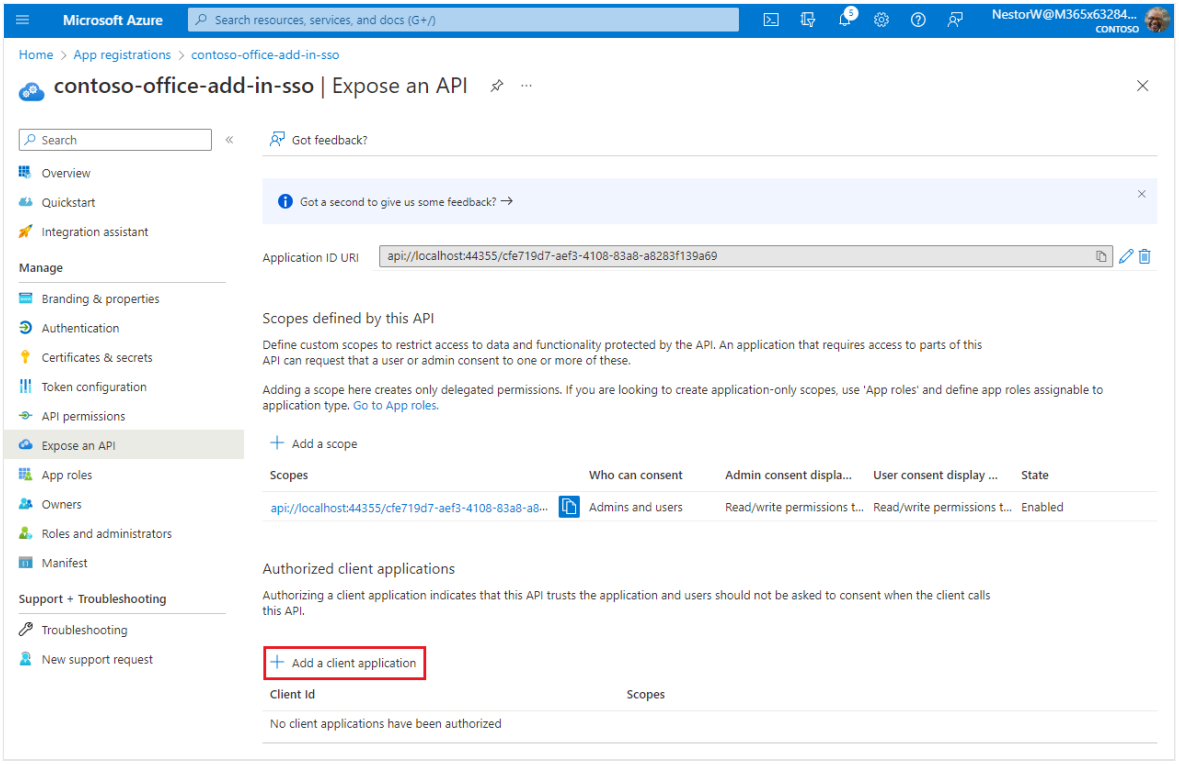


The new scope you defined displays on the pane.



> **Note**
>
> The domain part of the **Scope name** displayed just below the text field should automatically match the **Application ID URI** set in the previous step, with `/access_as_user` appended to the end; for example, `api://localhost:6789/c6c1f32b-5e55-4997-881a-753cc1d563b7/access_as_user`.

4. Select **Add a client application**.



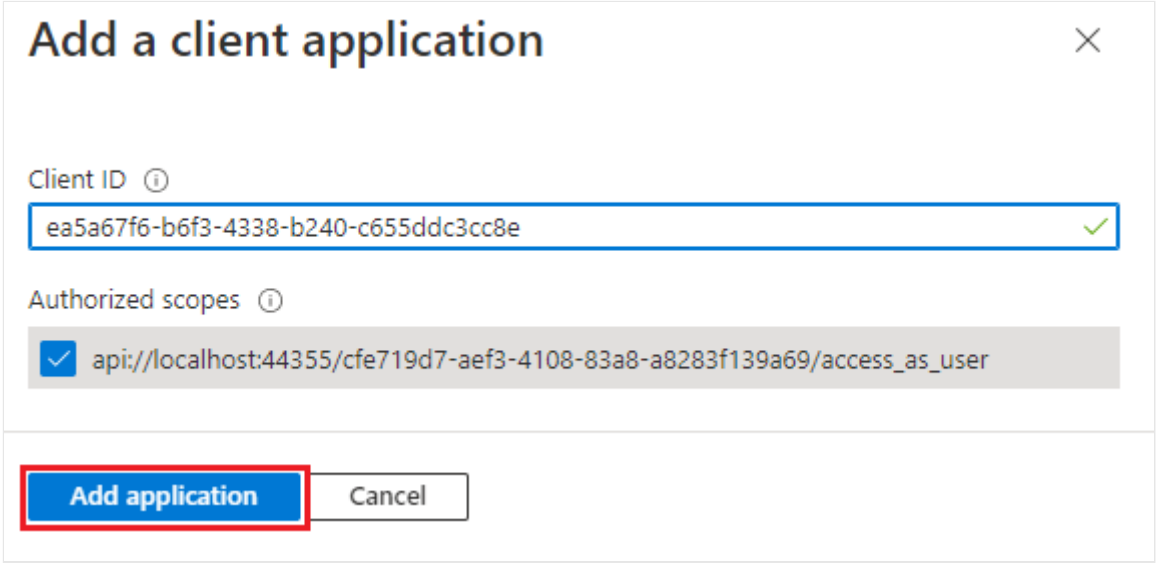
The **Add a client application** pane appears.

5. In the **Client ID** enter `ea5a67f6-b6f3-4338-b240-c655ddc3cc8e`. This value pre-authorizes all Microsoft Office application endpoints. If you also want to pre-authorize Office when used inside of Microsoft Teams, add `1fec8e78-bce4-4aaf-ab1b-5451cc387264` (Microsoft Teams desktop and Teams mobile) and `5e3ce6c0-2b1f-4285-8d4b-75ee78787346` (Teams on the web).

   > **Note**
   >
   > The `ea5a67f6-b6f3-4338-b240-c655ddc3cc8e` ID pre-authorizes Office on all the following platforms. Alternatively, you can enter a proper subset of the following IDs if, for any reason, you want to deny authorization to Office on some platforms. If you do so, leave out the IDs of the platforms from which you want to withhold authorization. Users of your add-in on those platforms will not be able to call your Web APIs, but other functionality in your add-in will still work.
   >
   > - `d3590ed6-52b3-4102-aeff-aad2292ab01c` (Microsoft Office)
   > - `93d53678-613d-4013-afc1-62e9e444a0a5` (Office on the web)
   > - `bc59ab01-8403-45c6-8796-ac3ef710b3e3` (Outlook on the web)

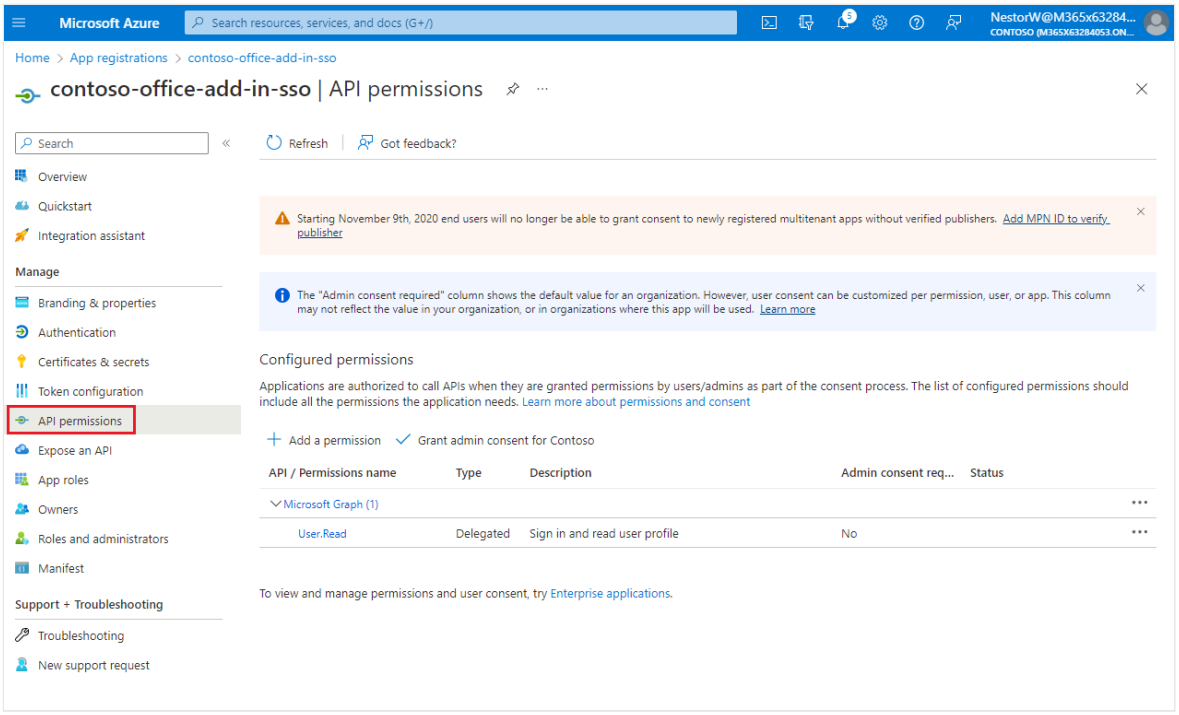
6. In **Authorized scopes**, select the `api://<fully-qualified-domain-name>/<app-id>/access_as_user` checkbox.

7. Select **Add application**.



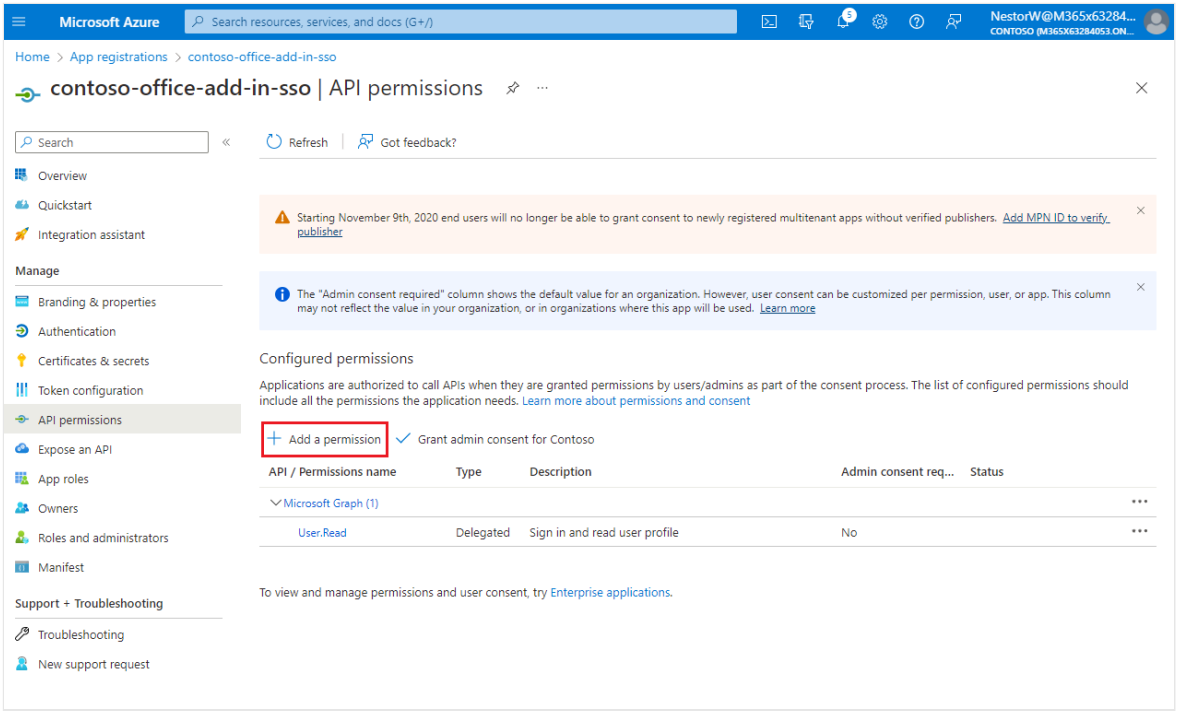
Add Microsoft Graph permissions

1. From the left pane, select **API permissions**.



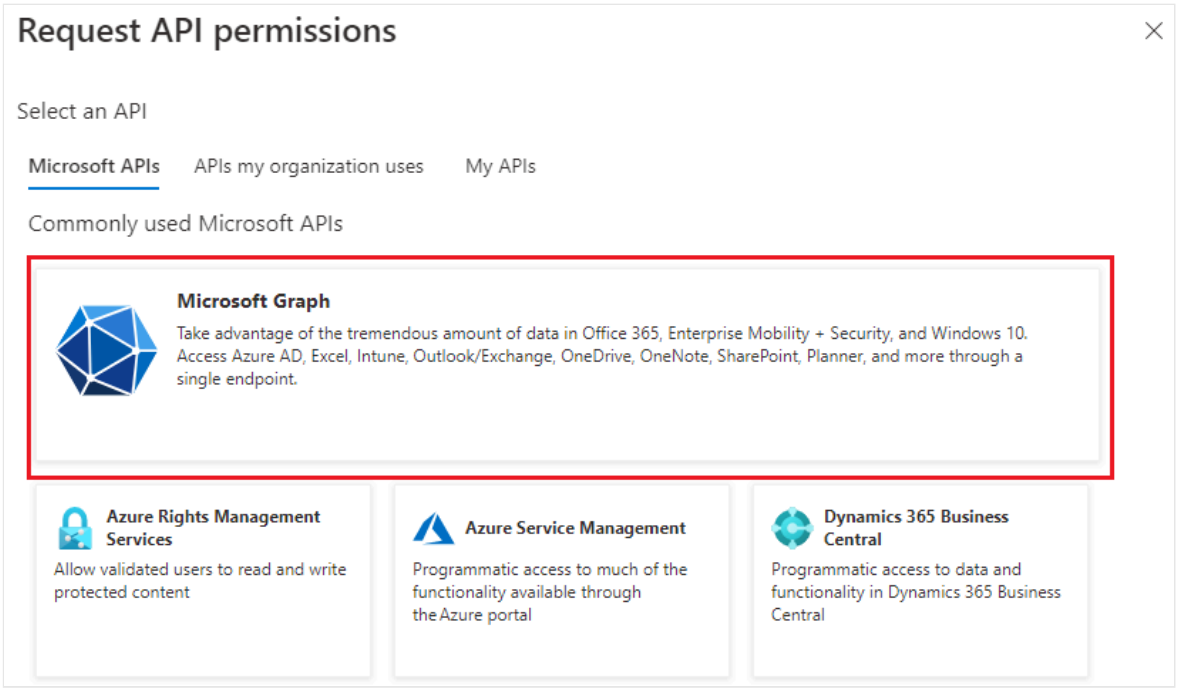
The **API permissions** pane opens.

2. Select **Add a permission**.

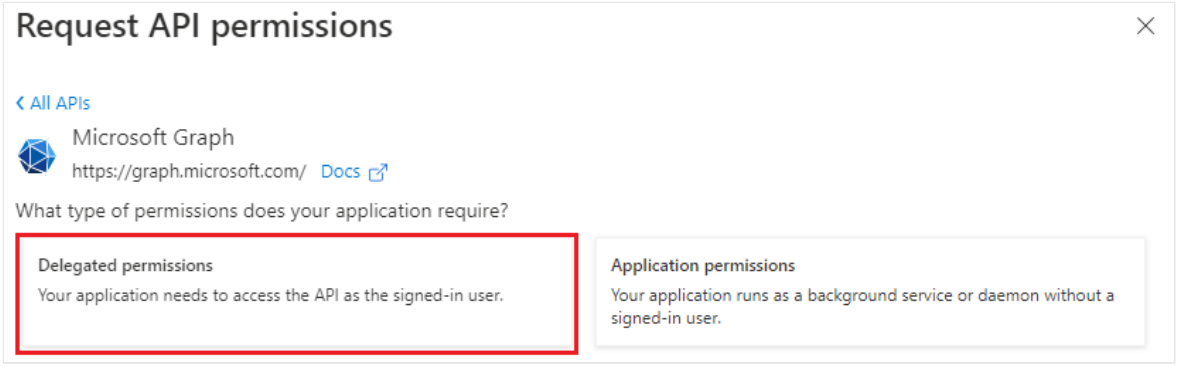


The **Request API permissions** pane opens.

3. Select **Microsoft Graph**.



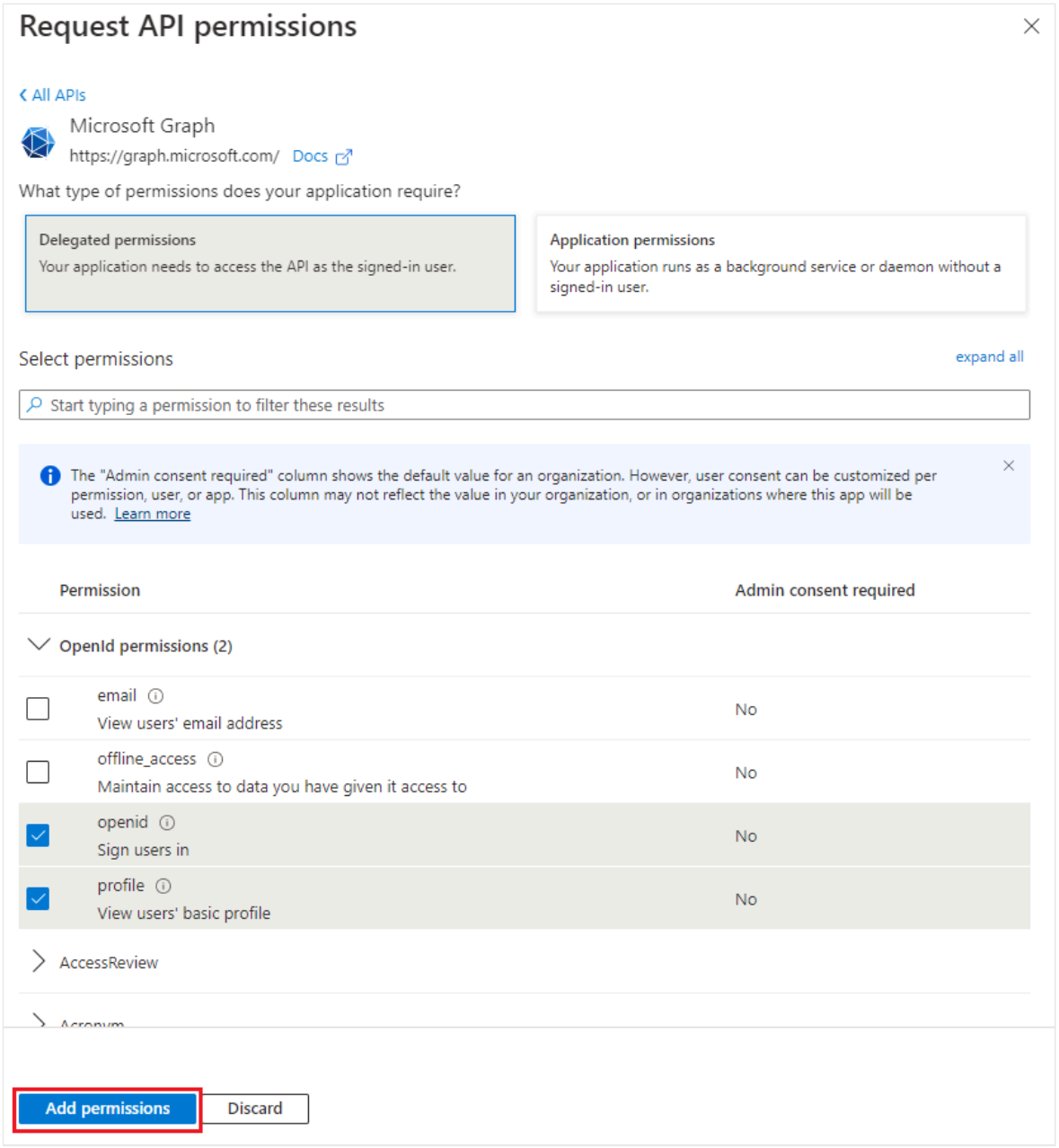
4. Select **Delegated permissions**.



5. In the **Select permissions** search box, search for the permissions your add-in needs. For example, for an Outlook add-in, you might use `profile`, `openid`, `Files.ReadWrite`, and `Mail.Read`.

> **Note**
>
> The `User.Read` permission may already be listed by default. It's a good practice to only request permissions that are needed, so we recommend that you uncheck the box for this permission if your add-in doesn't actually need it.

6. Select the checkbox for each permission as it appears. Note that the permissions will not remain visible in the list as you select each one. After selecting the permissions that your add-in needs, select **Add permissions**.



7. Select **Grant admin consent for [tenant name]**. Select **Yes** for the confirmation that appears.

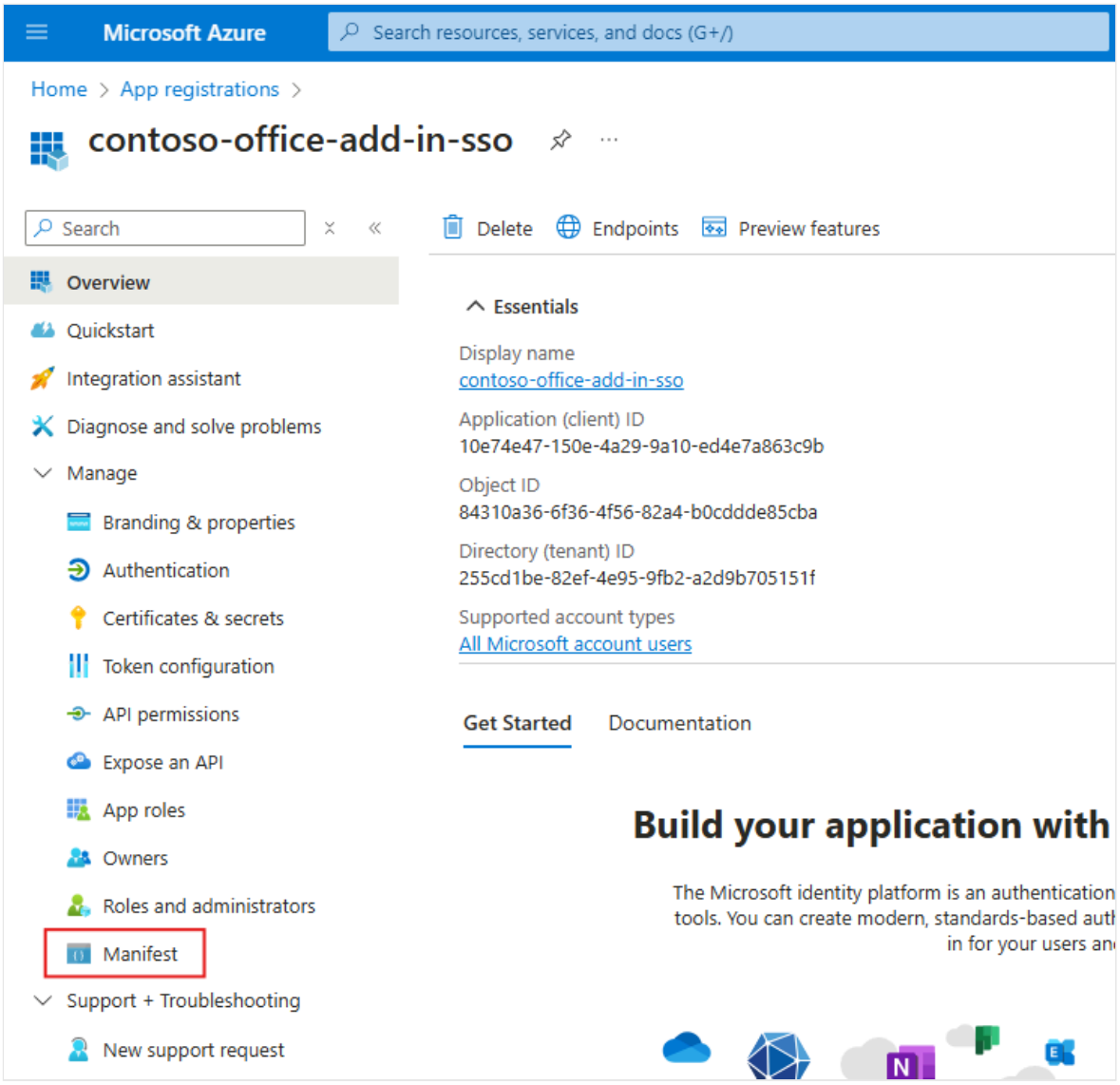
Configure access token version

You must define the access token version that is acceptable for your app. This configuration is made in the Azure Active Directory application manifest.

Define the access token version

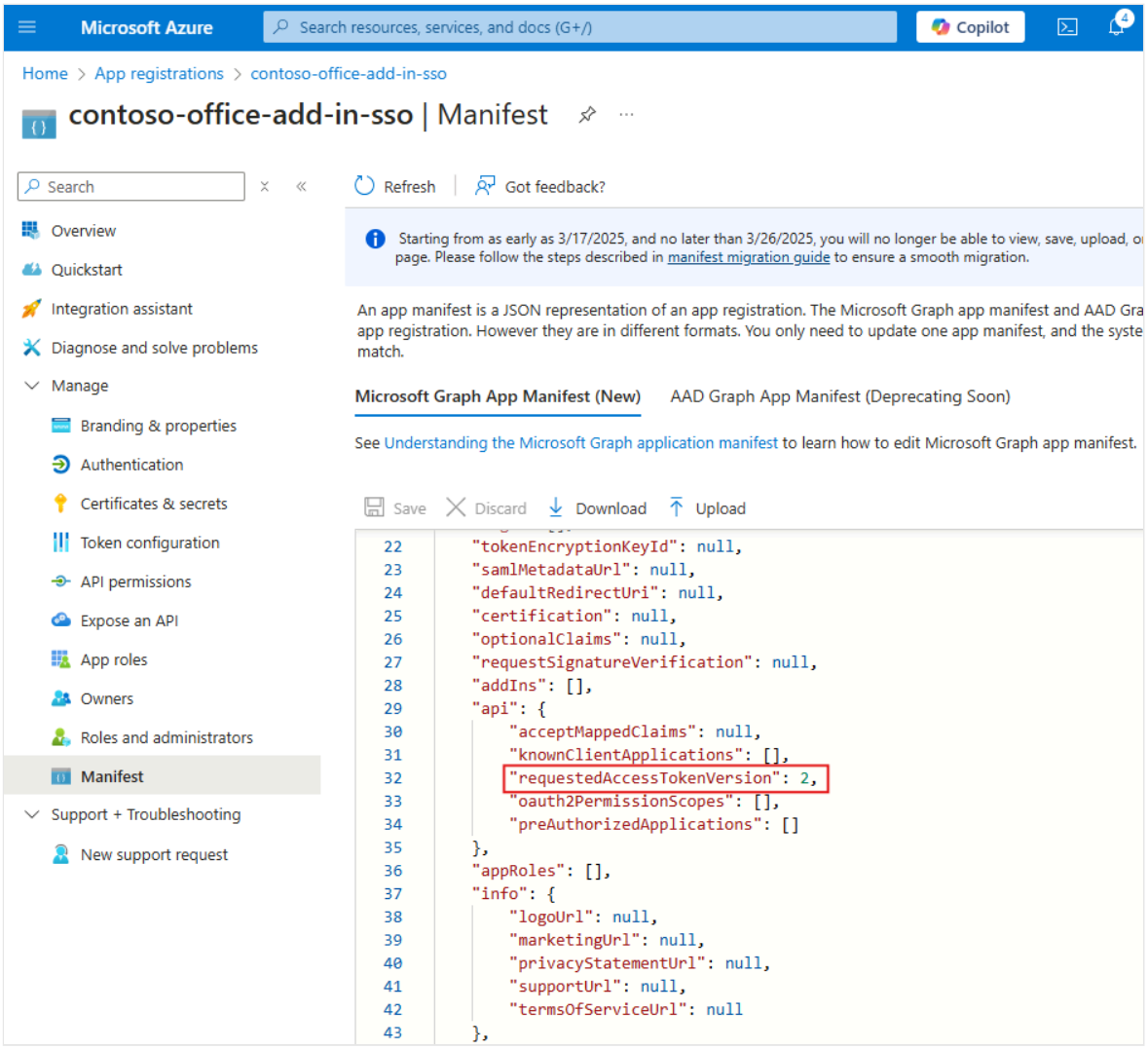
The access token version can change if you chose an account type other than **Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)**. Use the following steps to ensure the access token version is correct for Office SSO usage.

1. From the left pane, select **Manifest**.



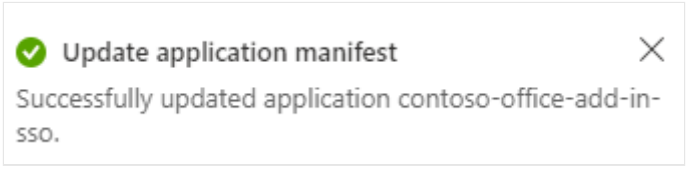
The Azure Active Directory application manifest appears.

2. Enter **2** as the value for the `requestedAccessTokenVersion` property (in the `api` object).



3. Select **Save**.

   A message pops up on the browser stating that the manifest was updated successfully.



Congratulations! You've completed the app registration to enable SSO for your Office add-in.

Configure the solution

1. In the root of the **Begin** folder, open the solution (.sln) file in **Visual Studio**. Right-click (or select and hold) the top node in **Solution Explorer** (the Solution node, not either of the project nodes), and then select **Set startup projects**.

2. Under **Common Properties**, select **Startup Project**, and then **Multiple startup projects**. Ensure that the **Action** for both projects is set to **Start**, and that the **Office-Add-in-ASPNETCoreWebAPI** project is listed first. Close the dialog.

3. In **Solution Explorer**, choose the **Office-Add-in-ASPNET-SSO-manifest** project and open the add-in manifest file "Office-Add-in-ASPNET-SSO.xml" and then scroll to the bottom of the file. Just above the end `</VersionOverrides>` tag, you'll find the following markup.

```xml
<WebApplicationInfo>
     <Id>Enter_client_ID_here</Id>
    <Resource>api://localhost:44355/Enter_client_ID_here</Resource>
    <Scopes>
        <Scope>Files.Read</Scope>
        <Scope>profile</Scope>
        <Scope>openid</Scope>
    </Scopes>
 </WebApplicationInfo>
```

4. Replace the placeholder "Enter_client_ID_here" *in both places* in the markup with the **Application ID** that you copied when you created the **Office-Add-in-ASPNET-SSO** app registration. This is the same ID you used for the application ID in the appsettings.json file.

> ⓘ Note
>
> The **<Resource>** value is the **Application ID URI** you set when you registered the add-in. The **<Scopes>** section is used only to generate a consent dialog box if the add-in is sold through AppSource.

5. Save and close the manifest file.

6. In **Solution Explorer**, choose the **Office-Add-in-ASPNET-SSO-web** project and open the **appsettings.json** file.

7. Replace the placeholder **Enter_client_id_here** with the **Application (client) ID** value you saved previously.

8. Replace the placeholder **Enter_client_secret_here** with the client secret value you saved previously.

> ⓘ Note

You must also change the **TenantId** to support single-tenant if you configured your app registration for single-tenant. Replace the **Common** value with the **Application (client) ID** for single-tenant support.

9. Save and close the appsettings.json file.

Code the client side

Get the access token and call the application server REST API

1. In the **Office-Add-in-ASPNETCore-WebAPI** project, open the **wwwroot\js\HomeES6.js** file. It already has code that ensures that Promises are supported, even in the Trident (Internet Explorer 11) webview control, and an `Office.onReady` call to assign a handler to the add-in's only button.

   > **Note**
   >
   > As the name suggests, the HomeES6.js uses JavaScript ES6 syntax because using `async` and `await` best shows the essential simplicity of the SSO API. When the localhost server is started, this file is transpiled to ES5 syntax so that the sample will support Trident.

2. In the `getUserFileNames` function, replace `TODO 1` with the following code. About this code, note:

   - It calls `Office.auth.getAccessToken` to get the access token from Office using SSO. This token will contain the user's identity as well as access permission to the application server.
   - The access token is passed to `callRESTApi` which makes the actual call to the application server. The application server then uses the OBO flow to call Microsoft Graph.
   - Any errors from calling `getAccessToken` will be handled by `handleClientSideErrors`.

   JavaScript

   ```
      let fileNameList = null;
   try {
       let accessToken = await Office.auth.getAccessToken(options);
   ```

```
        fileNameList = await callRESTApi("/api/files", accessToken);
    }
    catch (exception) {
        if (exception.code) {
            handleClientSideErrors(exception);
        }
        else {
            showMessage("EXCEPTION: " + exception);
        }
    }
```

3. In the `getUserFileNames` function, replace `TODO 2` with the following code. This will write the list of file names to the document.

JavaScript

```
try {
    await writeFileNamesToOfficeDocument(fileNameList);
    showMessage("Your data has been added to the document.");
} catch (error) {
    // The error from writeFileNamesToOfficeDocument will begin
    // "Unable to add filenames to document."
    showMessage(error);
}
```

4. In the `callRESTApi` function, replace `TODO 3` with the following code. About this code, note:

    - It constructs an authorization header containing the access token. This confirms to the application server that this client code has access permissions to the REST APIs.
    - It request JSON return types, so that all return values are handled in JSON.
    - Any errors are passed to `handleServerSideErrors` for processing.

JavaScript

```
try {
    let result = await $.ajax({
        url: relativeUrl,
        headers: { "Authorization": "Bearer " + accessToken },
        type: "GET",
        dataType: "json",
        contentType: "application/json; charset=utf-8"
    });
    return result;
} catch (error) {
```

```
        handleServerSideErrors(error);
    }
```

Handle SSO errors and application REST API errors

1. In the `handleSSOErrors` function, replace `TODO 4` with the following code. For more information about these errors, see Troubleshoot SSO in Office Add-ins.

JavaScript

```
    switch (error.code) {
        case 13001:
            // No one is signed into Office. If the add-in cannot be
effectively used when no one
            // is logged into Office, then the first call of
getAccessToken should pass the
            // `allowSignInPrompt: true` option.
            showMessage("No one is signed into Office. But you can use
many of the add-ins functions anyway. If you want to log in, press the
Get OneDrive File Names button again.");
            break;
        case 13002:
            // The user aborted the consent prompt. If the add-in cannot
be effectively used when consent
            // has not been granted, then the first call of getAccessToken
should pass the `allowConsentPrompt: true` option.
            showMessage("You can use many of the add-ins functions even
though you have not granted consent. If you want to grant consent,
press the Get OneDrive File Names button again.");
            break;
        case 13006:
            // Only seen in Office on the web.
            showMessage("Office on the web is experiencing a problem.
Please sign out of Office, close the browser, and then start again.");
            break;
        case 13008:
            // Only seen in Office on the web.
            showMessage("Office is still working on the last operation.
When it completes, try this operation again.");
            break;
        case 13010:
            // Only seen in Office on the web.
            showMessage("Follow the instructions to change your browser's
zone configuration.");
            break;
        default:
            // For all other errors, including 13000, 13003, 13005, 13007,
13012, and 50001, fall back
            // to non-SSO sign-in by using MSAL authentication.
            showMessage("SSO failed. In these cases you should implement a
falback to MSAL authentication.");
```

```
            break;
    }
```

2. In the `handleServerSideErrors` function, replace `TODO 5` with the following code.

```JavaScript
// Check headers to see if admin has not consented.
const header = errorResponse.getResponseHeader('WWW-Authenticate');
if (header !== null && header.includes('proposedAction=\"consent\"')) {
    showMessage("MSAL ERROR: " + "Admin consent required. Be sure admin
consent is granted on all scopes in the Azure app registration.");
    return;
}
```

3. In the `handleServerSideErrors` function, replace `TODO 6` with the following code. About this code, note:

    - In some cases, additional consent is required, such as 2FA. Microsoft identity returns the additional claims that are required to complete consent. This code adds the `authChallenge` property with the additional claims and calls `getUserfileNames` again. When `getAccessToken` is called again with the additional claims, the user gets a prompt for all required forms of authentication.

```JavaScript
// Check if Microsoft Graph requires an additional form of
authentication. Have the Office host
// get a new token using the Claims string, which tells Microsoft
identity to prompt the user for all
// required forms of authentication.
const errorDetails =
JSON.parse(errorResponse.responseJSON.value.details);
if (errorDetails) {
    if (errorDetails.error.message.includes("AADSTS50076")) {
        const claims = errorDetails.message.Claims;
        const claimsAsString = JSON.stringify(claims);
        getUserFileNames({ authChallenge: claimsAsString });
        return;
    }
}
```

4. In the `handleServerSideErrors` function, replace `TODO 7` with the following code. About this code, note:

- In the rare case the original SSO token is expired, it will detect this error condition and call `getUserFilenames` again. This results in another call to `getAccessToken` which returns a refreshed access token. The `retryGetAccessToken` variable counts the retries and is currently configured to only retry once.
- Finally, if an error cannot be handled, the default is to display the error in the task pane.

JavaScript

```
// Results from other errors (other than AADSTS50076) will have an
ExceptionMessage property.
const exceptionMessage =
JSON.parse(errorResponse.responseText).ExceptionMessage;
if (exceptionMessage) {
    // On rare occasions the access token is unexpired when Office
validates it,
    // but expires by the time it is sent to Microsoft identity in the
OBO flow. Microsoft identity will respond
    // with "The provided value for the 'assertion' is not valid. The
assertion has expired."
    // Retry the call of getAccessToken (no more than once). This time
Office will return a
    // new unexpired access token.
    if ((exceptionMessage.includes("AADSTS500133"))
        && (retryGetAccessToken <= 0)) {
        retryGetAccessToken++;
        getUserFileNames();
        return;
    }
    else {
        showMessage("MSAL error from application server: " +
JSON.stringify(exceptionMessage));
        return;
    }
}
// Default error handling if previous checks didn't apply.
showMessage(errorResponse.responseJSON.value);
```

5. Save the file.

Code the server side

The server-side code is an ASP.NET Core server that provides REST APIs for the client to call. For example, the REST API `/api/files` gets a list of filenames from the user's OneDrive folder. Each REST API call requires an access token by the client to ensure the correct client is accessing their data. The access token is exchanged for a Microsoft

Graph token through the On-Behalf-Of flow (OBO). The new Microsoft Graph token is cached by the MSAL.NET library for subsequent API calls. It's never sent outside of the server-side code. Microsoft identity documentation refers to this server as the middle-tier server because it is in the middle of the flow from client-side code to Microsoft services. For more information, see Middle-tier access token request

Configure Microsoft Graph and OBO flow

1. Open the `Program.cs` file and replace `TODO 8` with the following code. About this code, note:

    - It adds required services to handle token validation that is required for the REST APIs.
    - It adds Microsoft Graph and OBO flow support in the call to `EnableTokenAcquisitionToCallDownstreamApi().AddMicrosoftGraph(...)`. The OBO flow is handled automatically for you, and the Microsoft Graph SDK is provided to your REST API controllers.
    - The **DownstreamApi** configuration is specified in the **appsettings.json** file.

    C#

    ```
    // Add services to the container.
    builder.Services.AddMicrosoftIdentityWebApiAuthentication(builder.Confi
    guration)
                    .EnableTokenAcquisitionToCallDownstreamApi()

    .AddMicrosoftGraph(builder.Configuration.GetSection("DownstreamApi"))
                        .AddInMemoryTokenCaches();
    ```

Create the /api/filenames REST API

1. In the **Controllers** folder, open the **FilesController.cs** file. replace `TODO 9` with the following code. About this code, note:

    - It specifies the `[Authorize]` attribute to ensure the access token is validated for each call to any REST APIs in the `FilesController` class. For more information, see Validating tokens.
    - It specifies the `[RequiredScope("access_as_user")]` attribute to ensure the client has the correct **access_as_user** scope in the access token.
    - The constructor initializes the `_graphServiceClient` object to make calling Microsoft Graph REST APIs easier.

C#

```
[Authorize]
[Route("api/[controller]")]
[RequiredScope("access_as_user")]
public class FilesController : Controller
{
    public FilesController(ITokenAcquisition tokenAcquisition,
GraphServiceClient graphServiceClient, IOptions<MicrosoftGraphOptions>
graphOptions)
    {
        _tokenAcquisition = tokenAcquisition;
        _graphServiceClient = graphServiceClient;
        _graphOptions = graphOptions;

    }

    private readonly ITokenAcquisition _tokenAcquisition;
    private readonly GraphServiceClient _graphServiceClient;
    private readonly IOptions<MicrosoftGraphOptions> _graphOptions;

    // TODO 10: Add the REST API to get filenames.

}
```

2. Replace `TODO 10` with the following code. About this code, note:

   - It creates the `/api/files` REST API.
   - It handles exceptions from MSAL through `MsalException` class.
   - It handles exceptions from Microsoft Graph API calls through the `ServiceException` class.

C#

```
 // GET api/files
    [HttpGet]
    [Produces("application/json")]
    public async Task<IActionResult> Get()
    {
        List<DriveItem> result = new List<DriveItem>();
        try
        {
            var files = await
_graphServiceClient.Me.Drive.Root.Children.Request()
                .Top(10)
                .Select(m => new { m.Name })
                .GetAsync();

            result = files.ToList();
        }
        catch (MsalException ex)
```

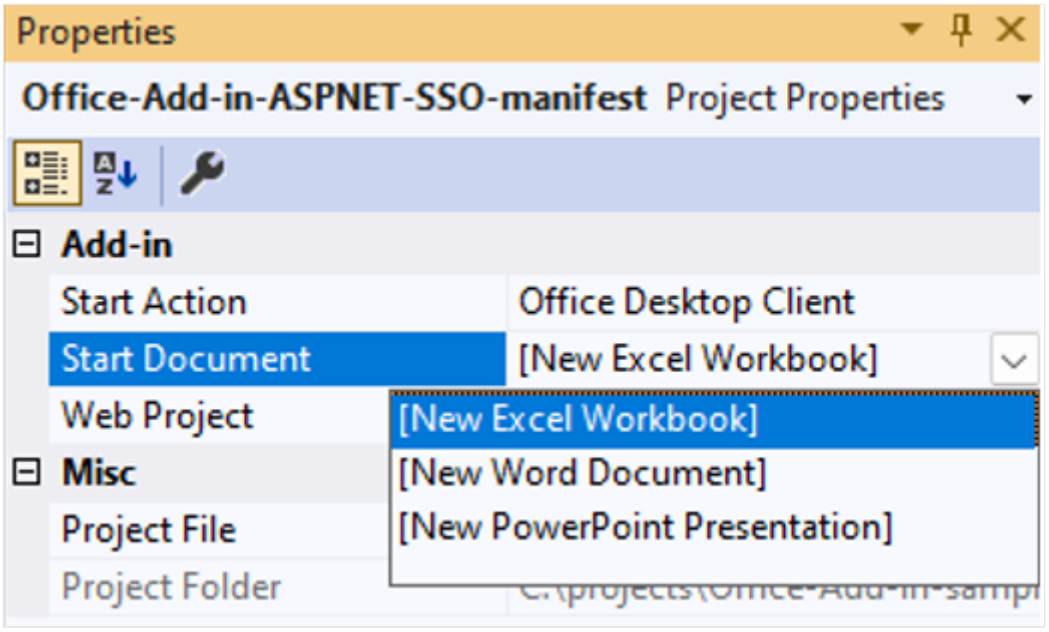
```
        {
            var errorResponse = new
            {
                message = "An authentication error occurred while
acquiring a token for downstream API",
                details = ex.Message
            };

            return StatusCode((int)HttpStatusCode.Unauthorized,
Json(errorResponse));
        }
        catch (ServiceException ex)
        {
            if (ex.InnerException is
MicrosoftIdentityWebChallengeUserException challengeException)
            {

_tokenAcquisition.ReplyForbiddenWithWwwAuthenticateHeader(_graphOptions
.Value.Scopes.Split(' '),
                    challengeException.MsalUiRequiredException);
            }
            else
            {
                var errorResponse = new
                {
                    message = "An error occurred calling Microsoft
Graph",
                    details = ex.RawResponseBody
                };
                return StatusCode((int)HttpStatusCode.BadRequest,
Json(errorResponse));
            }
        }
        catch (Exception ex)
        {
            var errorResponse = new
            {
                message = "An error occurred while calling the
downstream API",
                details = ex.Message
            };
            return StatusCode((int)HttpStatusCode.BadRequest,
Json(errorResponse));

        }
        return Json(result);
    }
```

Run the solution

1. In Visual Studio, on the **Build** menu, select **Clean Solution**. When it finishes, open the **Build** menu again and select **Build Solution**.

2. In **Solution Explorer**, select the **Office-Add-in-ASPNET-SSO-manifest** project node.

3. In the **Properties** pane, open the **Start Document** drop down and choose one of the three options (Excel, Word, or PowerPoint).



4. Press `F5`. Or select **Debug** > **Start Debugging**.

5. In the Office application, select the **Show Add-in** in the **SSO ASP.NET** group to open the task pane add-in.

6. Select **Get OneDrive File Names**. If you're logged into Office with either a Microsoft 365 Education or work account, or a Microsoft account, and SSO is working as expected, the first 10 file and folder names in your OneDrive for Business are displayed on the task pane. If you are not logged in, or you are in a scenario that does not support SSO, or SSO is not working for any reason, you will be prompted to sign in. After you sign in, the file and folder names appear.

Deploy the add-in

When you're ready to deploy to a staging or production server, be sure to update the following areas in the project solution.

- In the **appsettings.json** file, change the **domain** to your staging or production domain name.
- Update any references to `localhost:7080` throughout your project to use your staging or production URL.
- Update any references to `localhost:7080` in your Azure App registration, or create a new registration for use in staging or production.

For more information, see Host and deploy ASP.NET Core.

See also

- Create a Node.js Office Add-in that uses single sign-on.
- Authorize to Microsoft Graph with SSO.

Create a Node.js Office Add-in that uses single sign-on

Article • 05/20/2023

Users can sign in to Office, and your Office Web Add-in can take advantage of this sign-in process to authorize users to your add-in and to Microsoft Graph without requiring users to sign in a second time. For an overview, see Enable SSO in an Office Add-in.

This article walks you through the process of enabling single sign-on (SSO) in an add-in. The sample add-in you create has two parts; a task pane that loads in Microsoft Excel, and a middle-tier server that handles calls to Microsoft Graph for the task pane. The middle-tier server is built with Node.js and Express and exposes a single REST API, `/getuserfilenames`, that returns a list of the first 10 file names in the user's OneDrive folder. The task pane uses the `getAccessToken()` method to get an access token for the signed in user to the middle-tier server. The middle-tier server uses the On-Behalf-Of flow (OBO) to exchange the access token for a new one with access to Microsoft Graph. You can extend this pattern to access any Microsoft Graph data. The task pane always calls a middle-tier REST API (passing the access token) when it needs Microsoft Graph services. The middle-tier uses the token obtained via OBO to call Microsoft Graph services and return the results to the task pane.

This article works with an add-in that uses Node.js and Express. For a similar article about an ASP.NET-based add-in, see Create an ASP.NET Office Add-in that uses single sign-on.

Prerequisites

- Node.js (the latest LTS version)
- Git Bash (or another git client)
- A code editor - we recommend Visual Studio Code
- At least a few files and folders stored on OneDrive for Business in your Microsoft 365 subscription
- A build of Microsoft 365 that supports the IdentityAPI 1.3 requirement set. You might qualify for a Microsoft 365 E5 developer subscription, which includes a developer sandbox, through the Microsoft 365 Developer Program; for details, see the FAQ. The developer sandbox includes a Microsoft Azure subscription that you can use for app registrations in later steps in this article. If you prefer, you can

use a separate Microsoft Azure subscription for app registrations. Get a trial subscription at Microsoft Azure.

Set up the starter project

1. Clone or download the repo at Office Add-in NodeJS SSO.

   > **Note**
   >
   > There are two versions of the sample:
   >
   > - The **Begin** folder is a starter project. The UI and other aspects of the add-in that are not directly connected to SSO or authorization are already done. Later sections of this article walk you through the process of completing it.
   > - The **Complete** folder contains the same sample with all coding steps from this article completed. To use the completed version, just follow the instructions in this article, but replace "Begin" with "Complete" and skip the sections **Code the client side** and **Code the middle-tier server** side.

2. Open a command prompt in the **Begin** folder.
3. Enter `npm install` in the console to install all of the dependencies itemized in the package.json file.
4. Run the command `npm run install-dev-certs`. Select **Yes** to the prompt to install the certificate.

Use the following values for placeholders for the subsequent app registration steps.

Expand table

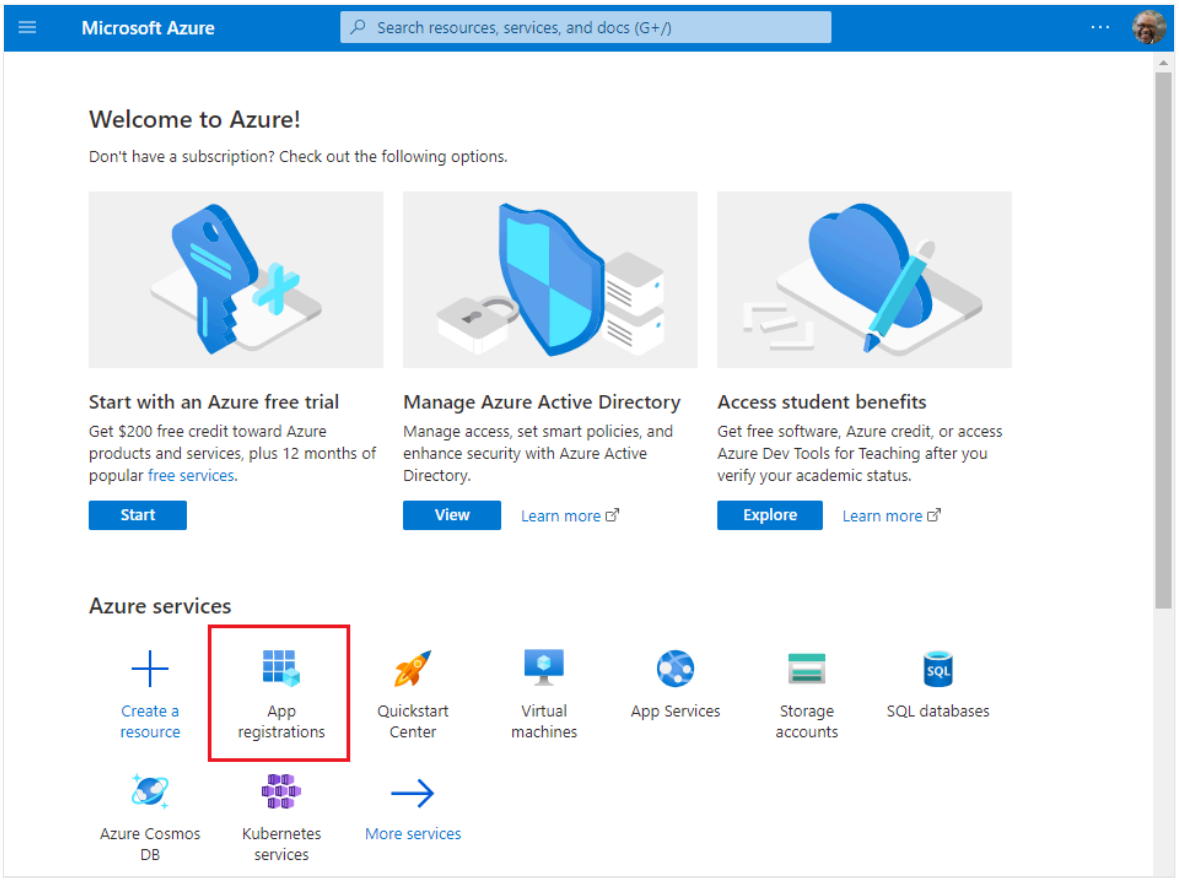
Placeholder	Value
`<add-in-name>`	Office-Add-in-NodeJS-SSO
`<fully-qualified-domain-name>`	`localhost:3000`
Microsoft Graph permissions	profile, openid, Files.Read

Register the add-in with Microsoft identity platform

You need to create an app registration in Azure that represents your web server. This enables authentication support so that proper access tokens can be issued to the client code in JavaScript. This registration supports both SSO in the client, and fallback authentication using the Microsoft Authentication Library (MSAL).

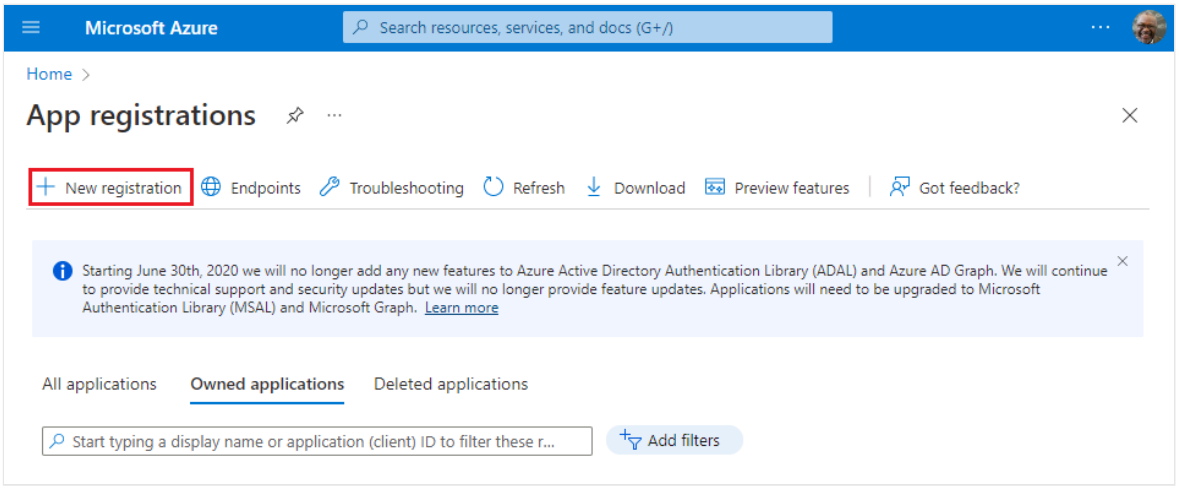
1. Sign in to the Azure portal with the ***admin*** credentials to your Microsoft 365 tenancy. For example, **MyName@contoso.onmicrosoft.com**.

2. Select **App registrations**. If you don't see the icon, search for "app registration" in the search bar.



   The **App registrations** page appears.

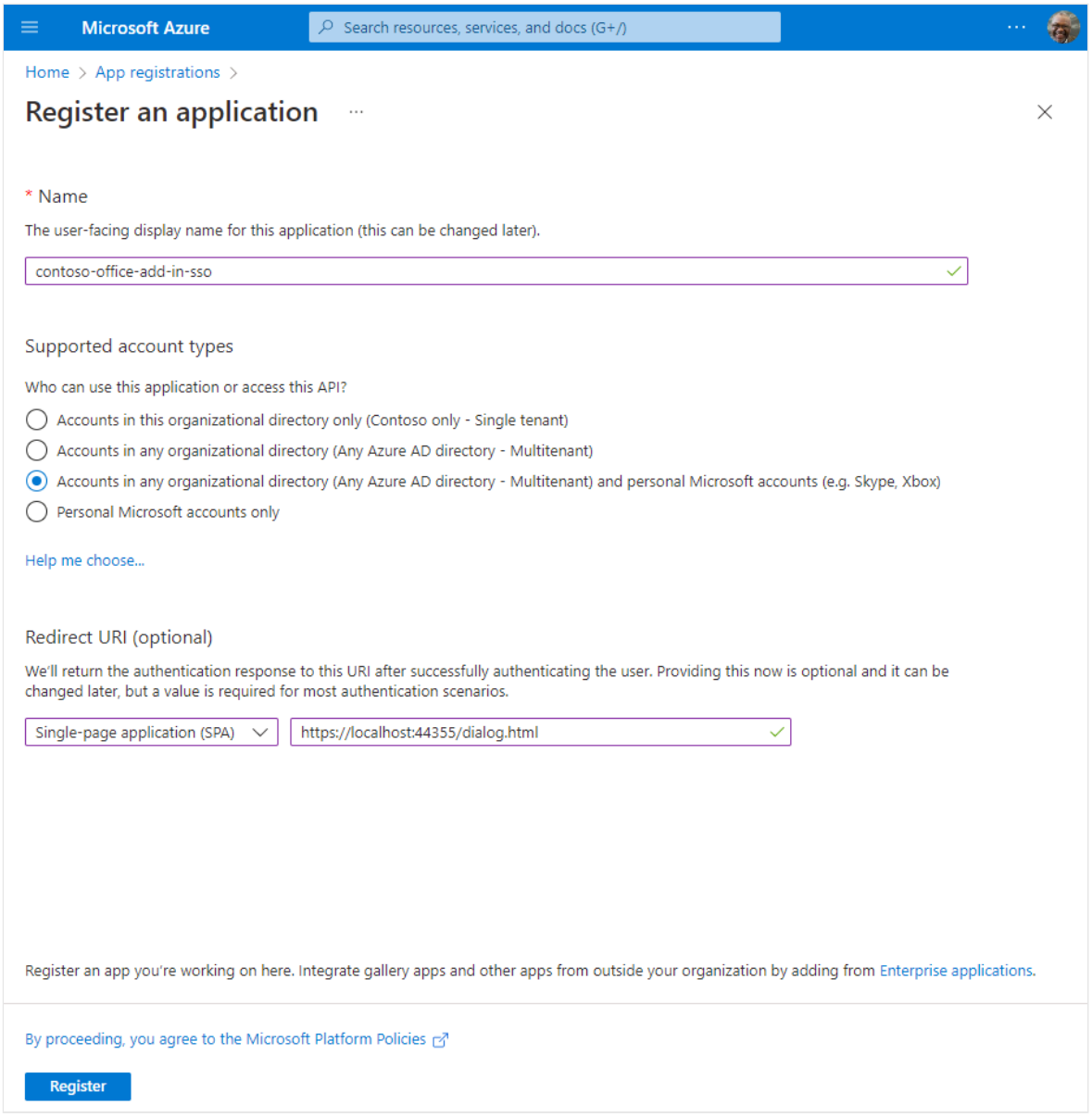
3. Select **New registration**.



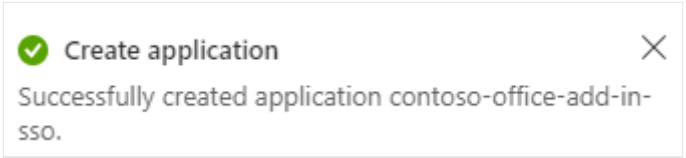
The **Register an application** page appears.

4. On the **Register an application** page, set the values as follows.

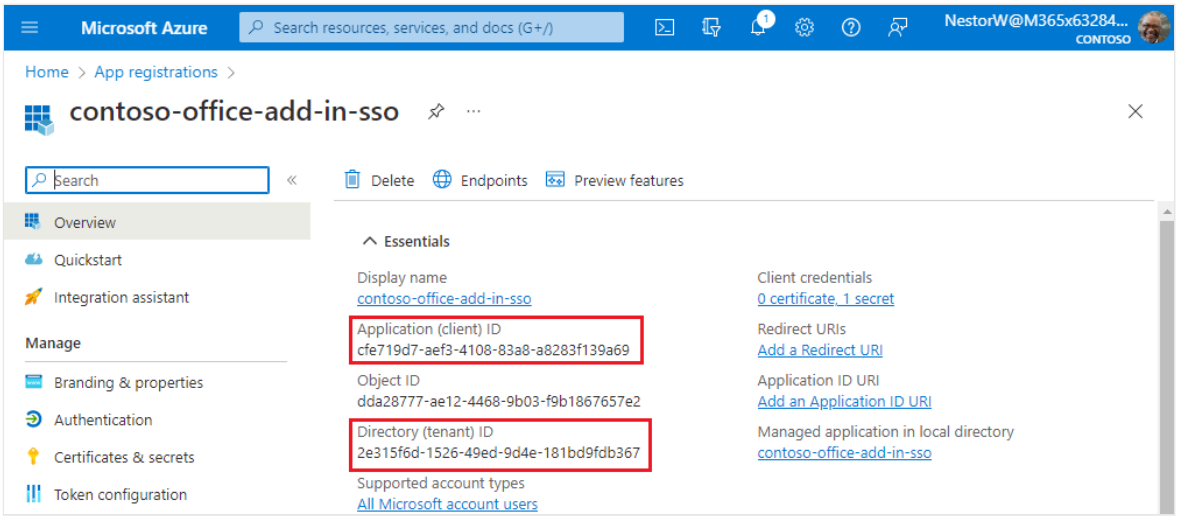
    - Set **Name** to `<add-in-name>`.
    - Set **Supported account types** to **Accounts in any organizational directory (any Azure AD directory - multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)**.
    - Set **Redirect URI** to use the platform **Single-page application (SPA)** and the URI to `https://<fully-qualified-domain-name>/dialog.html`.



5. Select **Register**. A message is displayed stating that the application registration was created.



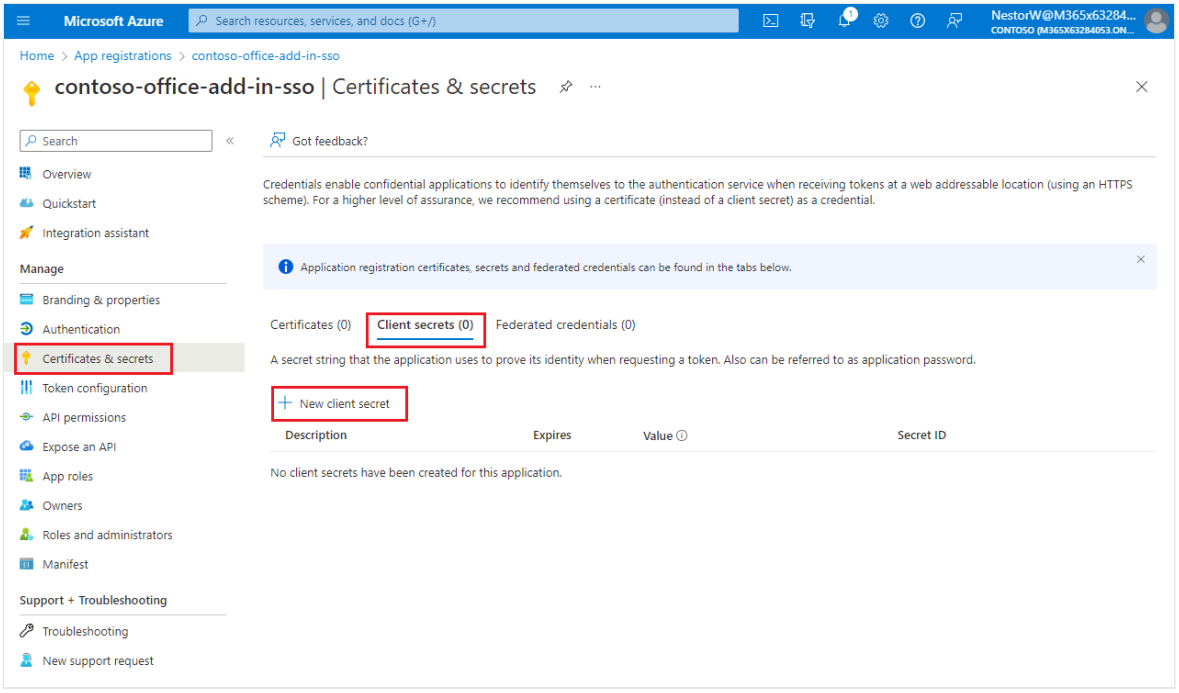
6. Copy and save the values for the **Application (client) ID** and the **Directory (tenant) ID**. You'll use both of them in later procedures.



Add a client secret

Sometimes called an *application password*, a client secret is a string value your app can use in place of a certificate to identity itself.

1. From the left pane, select **Certificates & secrets**. Then on the **Client secrets** tab, select **New client secret**.



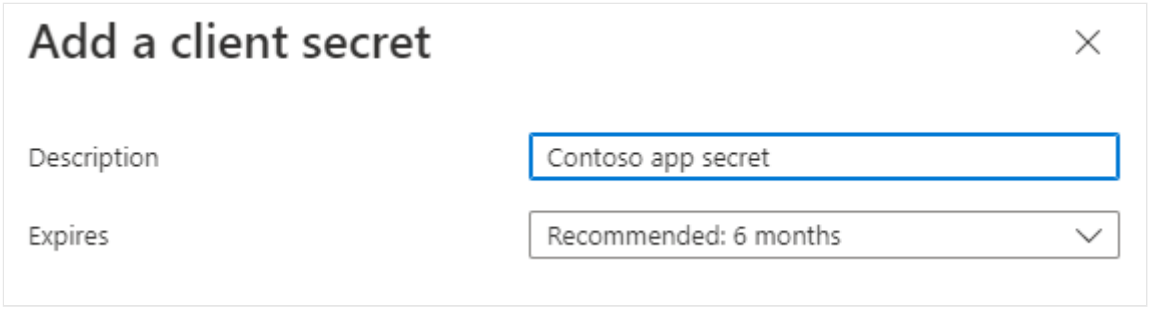
   The **Add a client secret** pane appears.

2. Add a description for your client secret.

3. Select an expiration for the secret or specify a custom lifetime.

   - Client secret lifetime is limited to two years (24 months) or less. You can't specify a custom lifetime longer than 24 months.

- Microsoft recommends that you set an expiration value of less than 12 months.



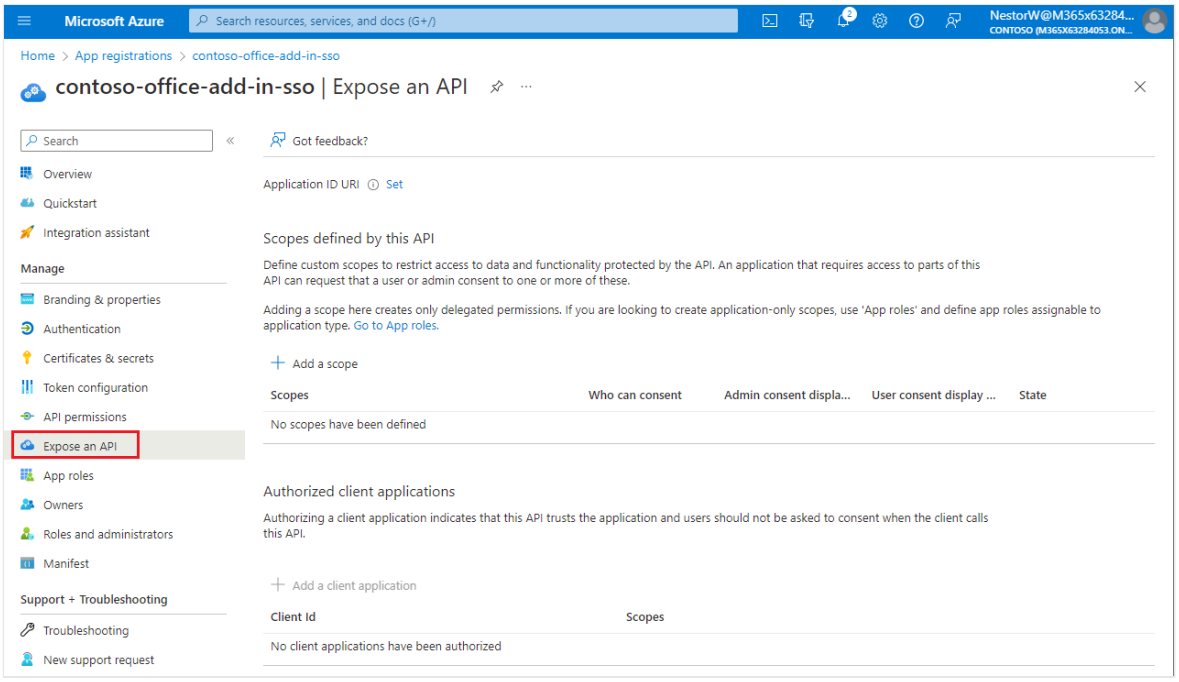
4. Select **Add**. The new secret is created and the value is temporarily displayed.

> **ⓘ Important**
>
> *Record the secret's value* for use in your client application code. This secret value is *never displayed again* after you leave this pane.

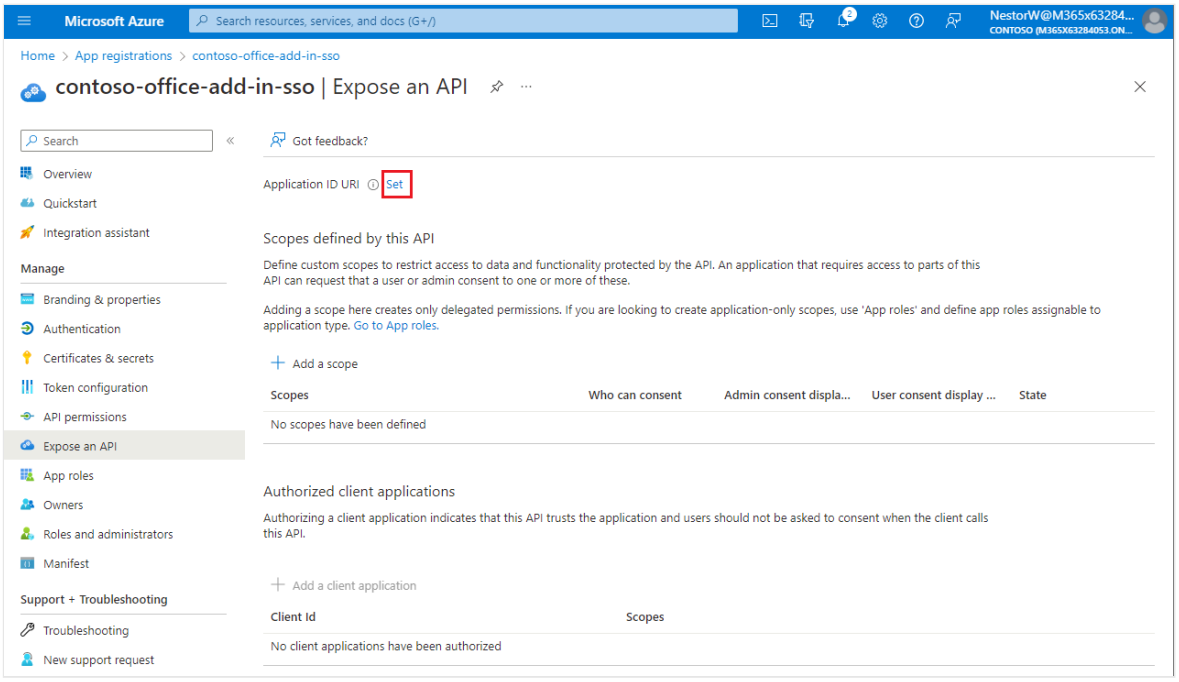
Expose a web API

1. From the left pane, select **Expose an API**.

   The **Expose an API** pane appears.

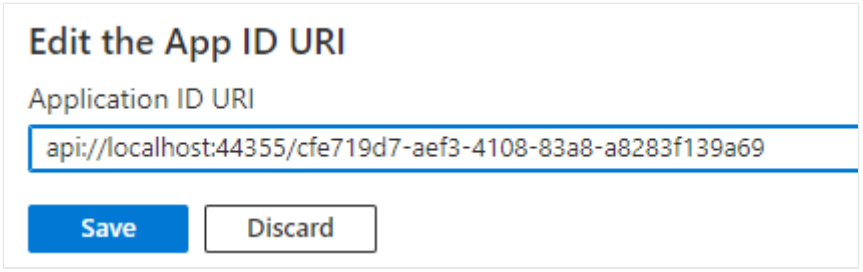


2. Select **Set** to generate an application ID URI.



The section for setting the application ID URI appears with a generated Application ID URI in the form `api://<app-id>`.

3. Update the application ID URI to `api://<fully-qualified-domain-name>/<app-id>`.



- The **Application ID URI** is pre-filled with app ID (GUID) in the format `api://<app-id>`.
- The application ID URI format should be: `api://<fully-qualified-domain-name>/<app-id>`
- Insert the `fully-qualified-domain-name` between `api://` and `<app-id>` (which is a GUID). For example, `api://contoso.com/<app-id>`.
- If you're using localhost, then the format should be `api://localhost:<port>/<app-id>`. For example, `api://localhost:3000/c6c1f32b-5e55-4997-881a-753cc1d563b7`.

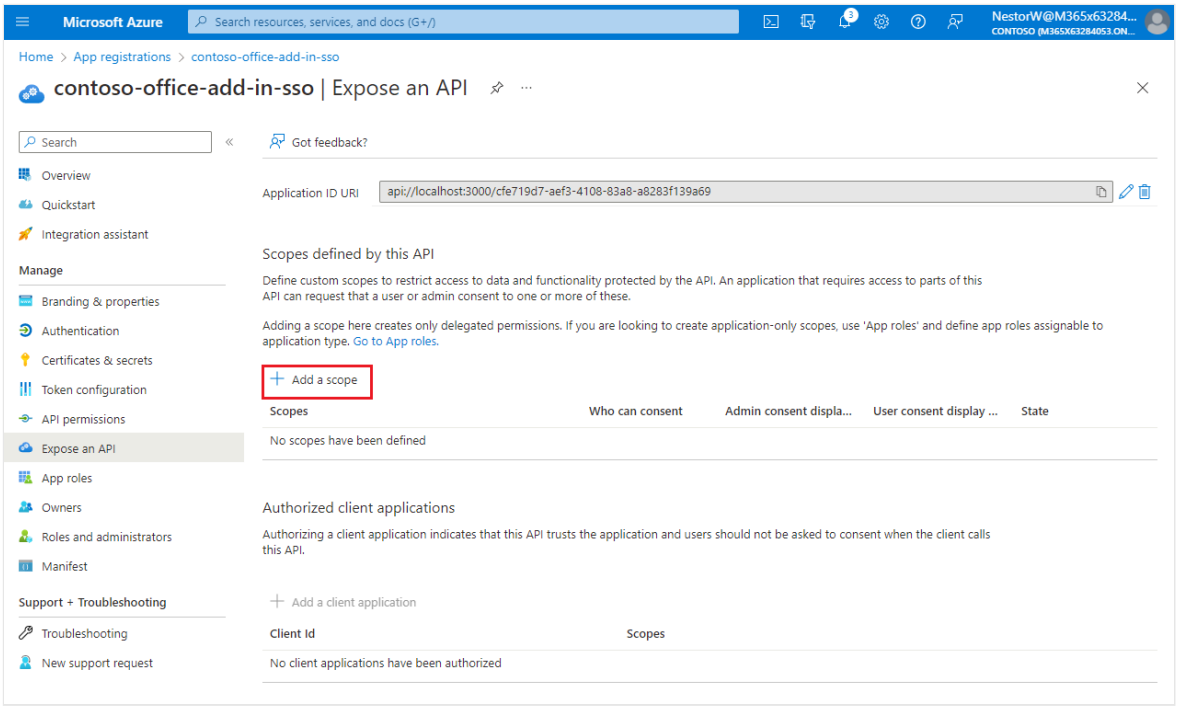
For additional application ID URI details, see Application manifest identifierUris attribute.

> **Note**

If you get an error saying that the domain is already owned but you own it, follow the procedure at Quickstart: Add a custom domain name to Azure Active Directory to register it, and then repeat this step. (This error can also occur if you are not signed in with credentials of an admin in the Microsoft 365 tenancy. See step 2. Sign out and sign in again with admin credentials and repeat the process from step 3.)

Add a scope

1. On the **Expose an API** page, select **Add a scope**.



   The **Add a scope** pane opens.

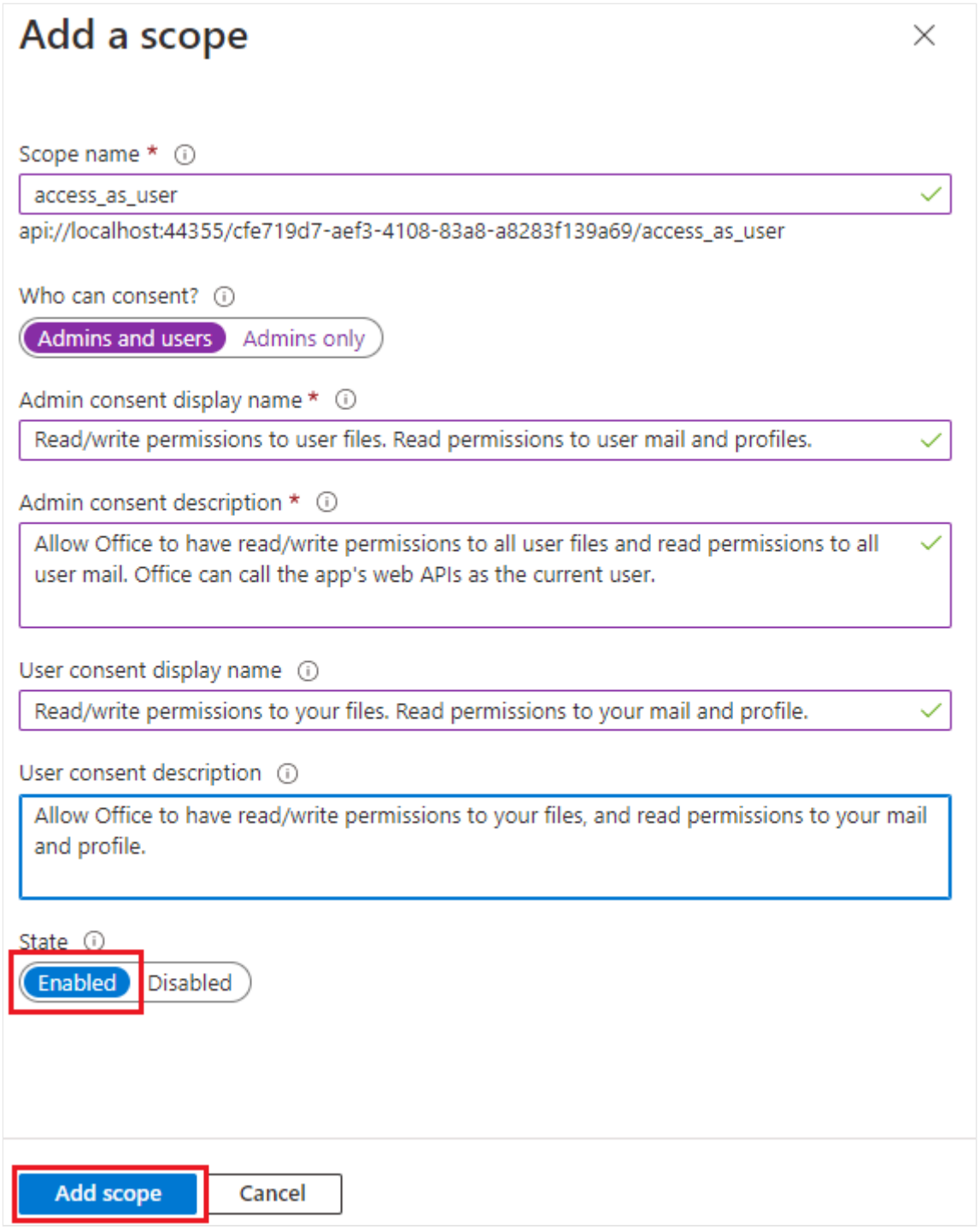
2. In the **Add a scope** pane, specify the scope's attributes. The following table shows example values for and Outlook add-in requiring the `profile`, `openid`, `Files.ReadWrite`, and `Mail.Read` permissions. Modify the text to match the permissions your add-in needs.

Expand table

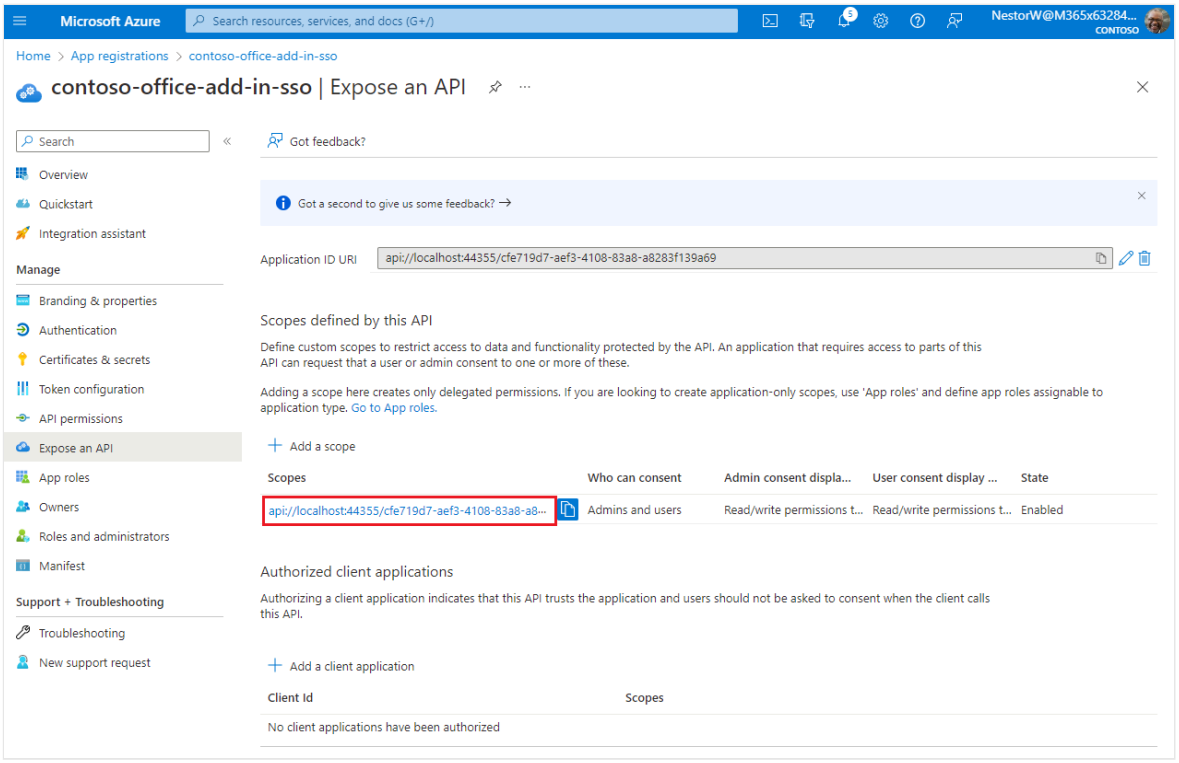
Field	Description	Values
Scope name	The name of your scope. A common scope naming convention is `resource.operation.constraint`.	For SSO this must be set to `access_as_user`.

Field	Description	Values
Who can consent	Determines if admin consent is required or if users can consent without an admin approval.	For learning SSO and samples, we recommend you set this to **Admins and users**. Select **Admins only** for higher-privileged permissions.
Admin consent display name	A short description of the scope's purpose visible to admins only.	`Read/write permissions to user files. Read permissions to user mail and profiles.`
Admin consent description	A more detailed description of the permission granted by the scope that only admins see.	`Allow Office to have read/write permissions to all user files and read permissions to all user mail. Office can call the app's web APIs as the current user.`
User consent display name	A short description of the scope's purpose. Shown to users only if you set **Who can consent** to **Admins and users**.	`Read/write permissions to your files. Read permissions to your mail and profile.`
User consent description	A more detailed description of the permission granted by the scope. Shown to users only if you set **Who can consent** to **Admins and users**.	`Allow Office to have read/write permissions to your files, and read permissions to your mail and profile.`

3. Set the **State** to **Enabled**, and then select **Add scope**.

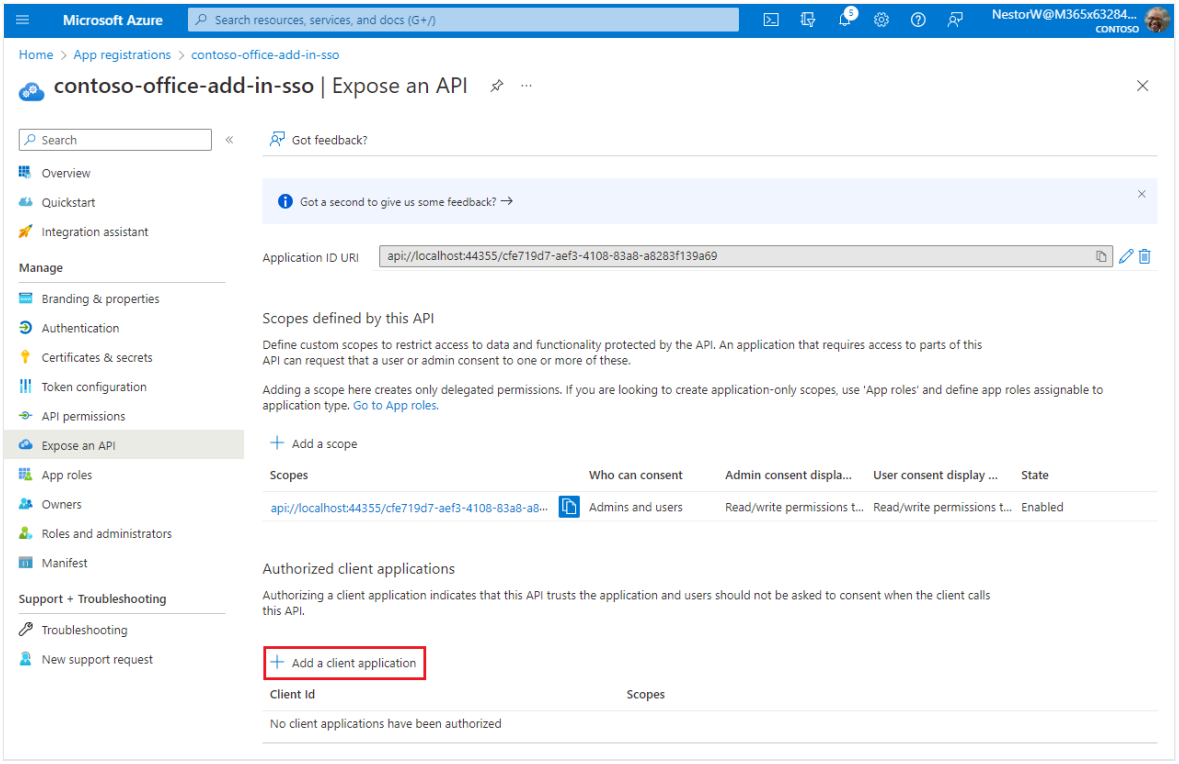


The new scope you defined displays on the pane.



> **Note**
>
> The domain part of the **Scope name** displayed just below the text field should automatically match the **Application ID URI** set in the previous step, with `/access_as_user` appended to the end; for example, `api://localhost:6789/c6c1f32b-5e55-4997-881a-753cc1d563b7/access_as_user`.

4. Select **Add a client application**.



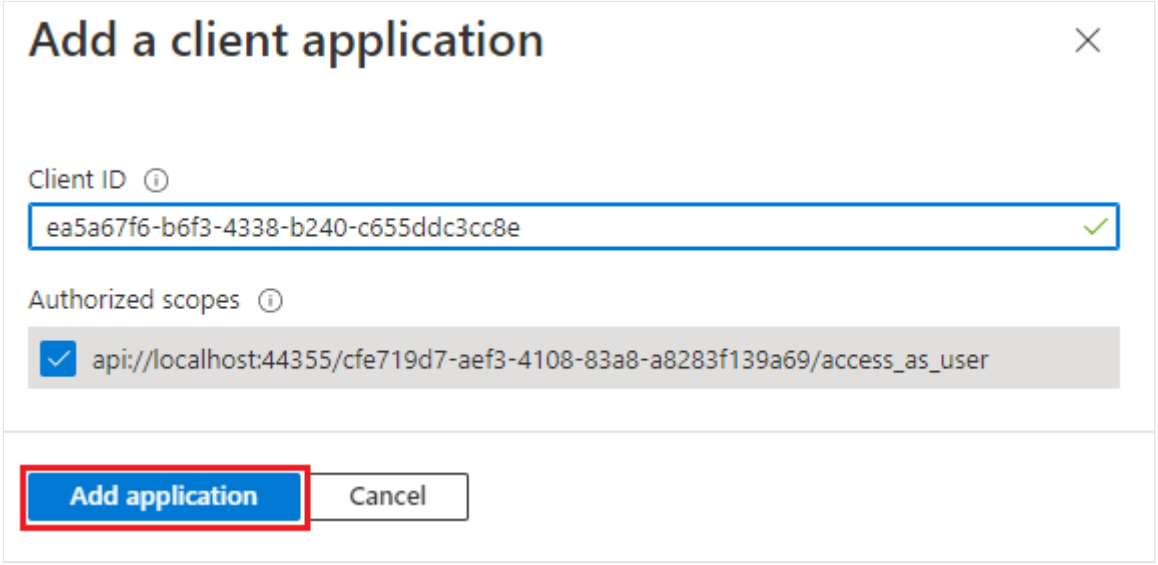
The **Add a client application** pane appears.

5. In the **Client ID** enter `ea5a67f6-b6f3-4338-b240-c655ddc3cc8e`. This value pre-authorizes all Microsoft Office application endpoints. If you also want to pre-authorize Office when used inside of Microsoft Teams, add `1fec8e78-bce4-4aaf-ab1b-5451cc387264` (Microsoft Teams desktop and Teams mobile) and `5e3ce6c0-2b1f-4285-8d4b-75ee78787346` (Teams on the web).

   > **Note**
   >
   > The `ea5a67f6-b6f3-4338-b240-c655ddc3cc8e` ID pre-authorizes Office on all the following platforms. Alternatively, you can enter a proper subset of the following IDs if, for any reason, you want to deny authorization to Office on some platforms. If you do so, leave out the IDs of the platforms from which you want to withhold authorization. Users of your add-in on those platforms will not be able to call your Web APIs, but other functionality in your add-in will still work.
   >
   > - `d3590ed6-52b3-4102-aeff-aad2292ab01c` (Microsoft Office)
   > - `93d53678-613d-4013-afc1-62e9e444a0a5` (Office on the web)
   > - `bc59ab01-8403-45c6-8796-ac3ef710b3e3` (Outlook on the web)

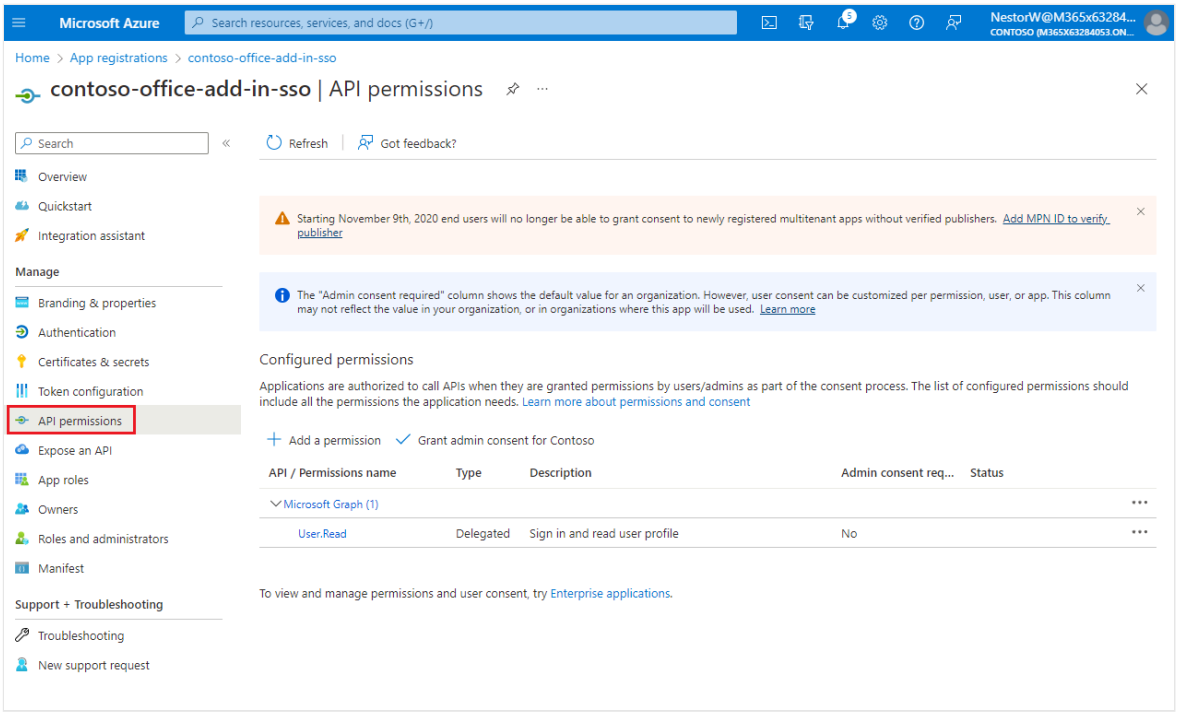
6. In **Authorized scopes**, select the `api://<fully-qualified-domain-name>/<app-id>/access_as_user` checkbox.

7. Select **Add application**.



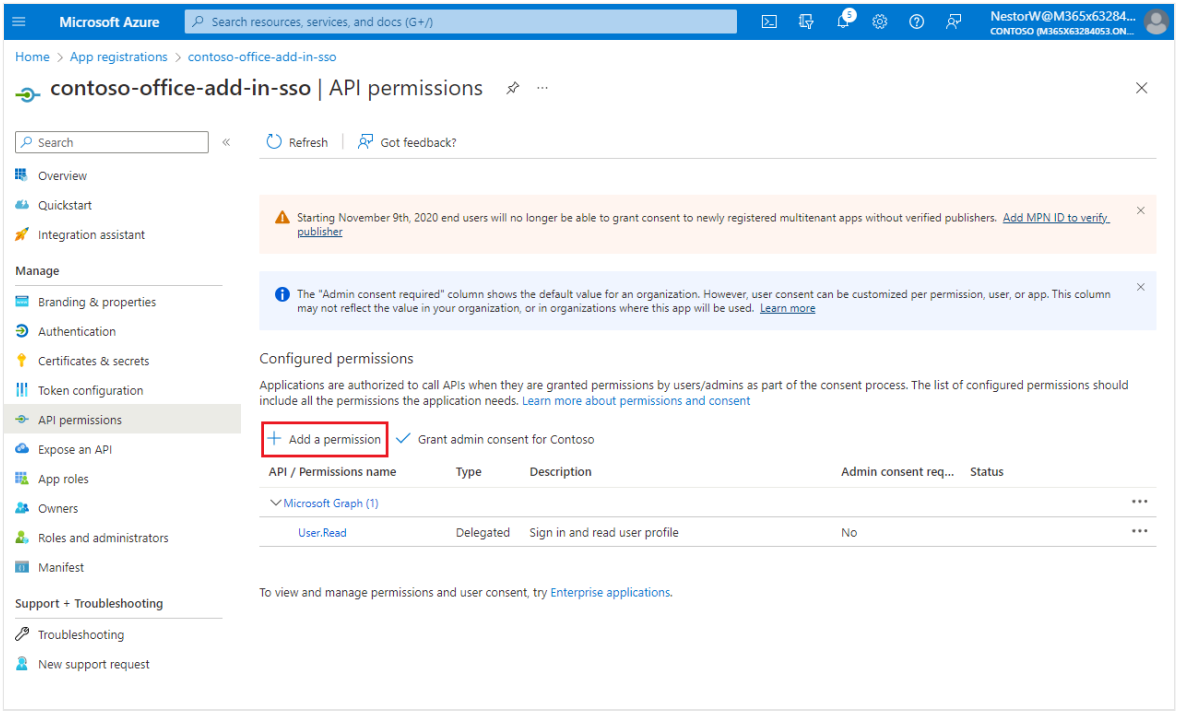
Add Microsoft Graph permissions

1. From the left pane, select **API permissions**.



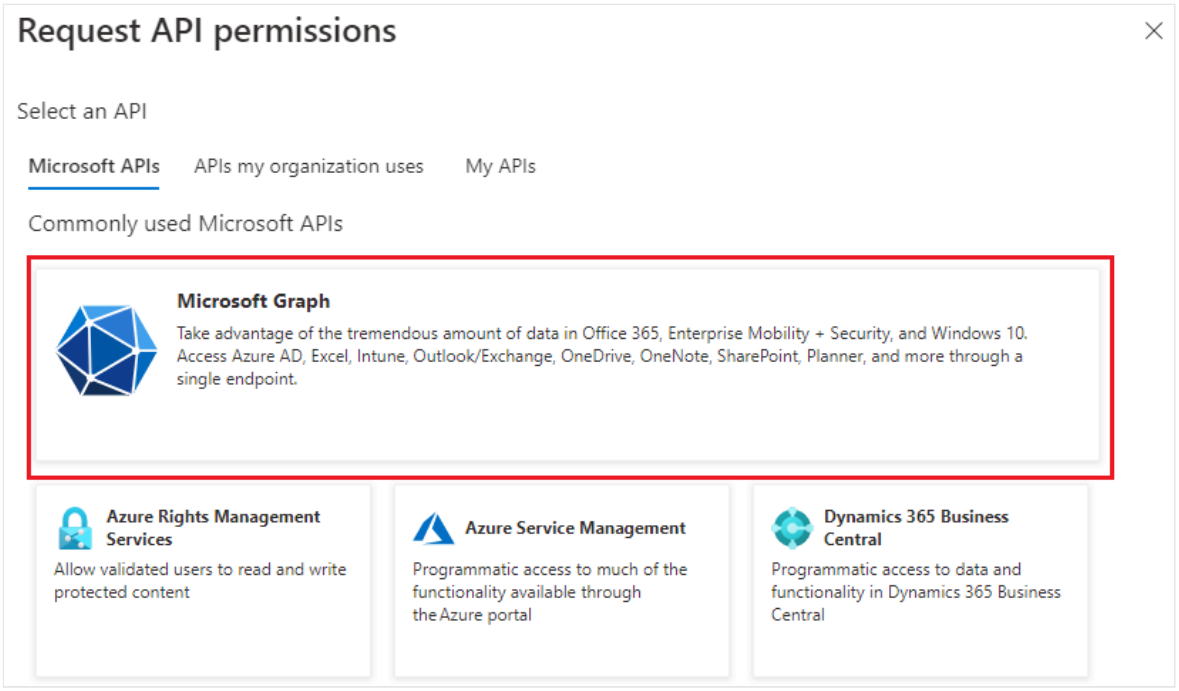
The **API permissions** pane opens.

2. Select **Add a permission**.

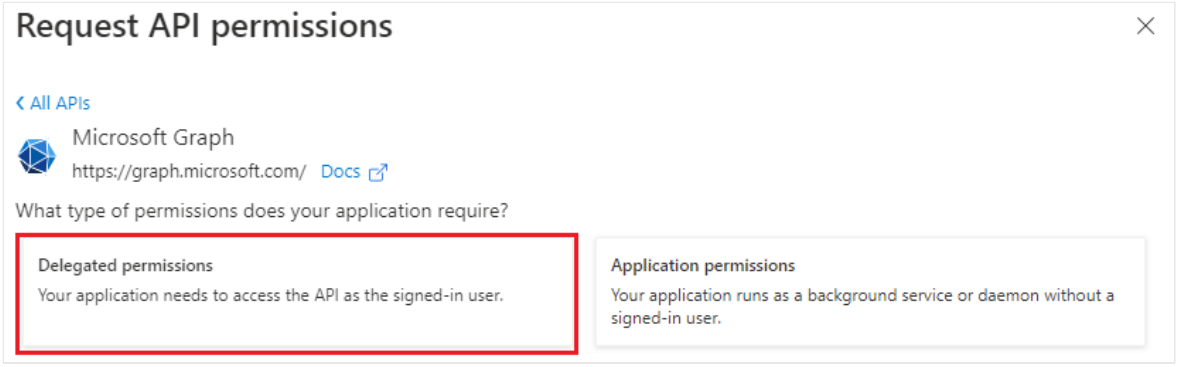


The **Request API permissions** pane opens.

3. Select **Microsoft Graph**.



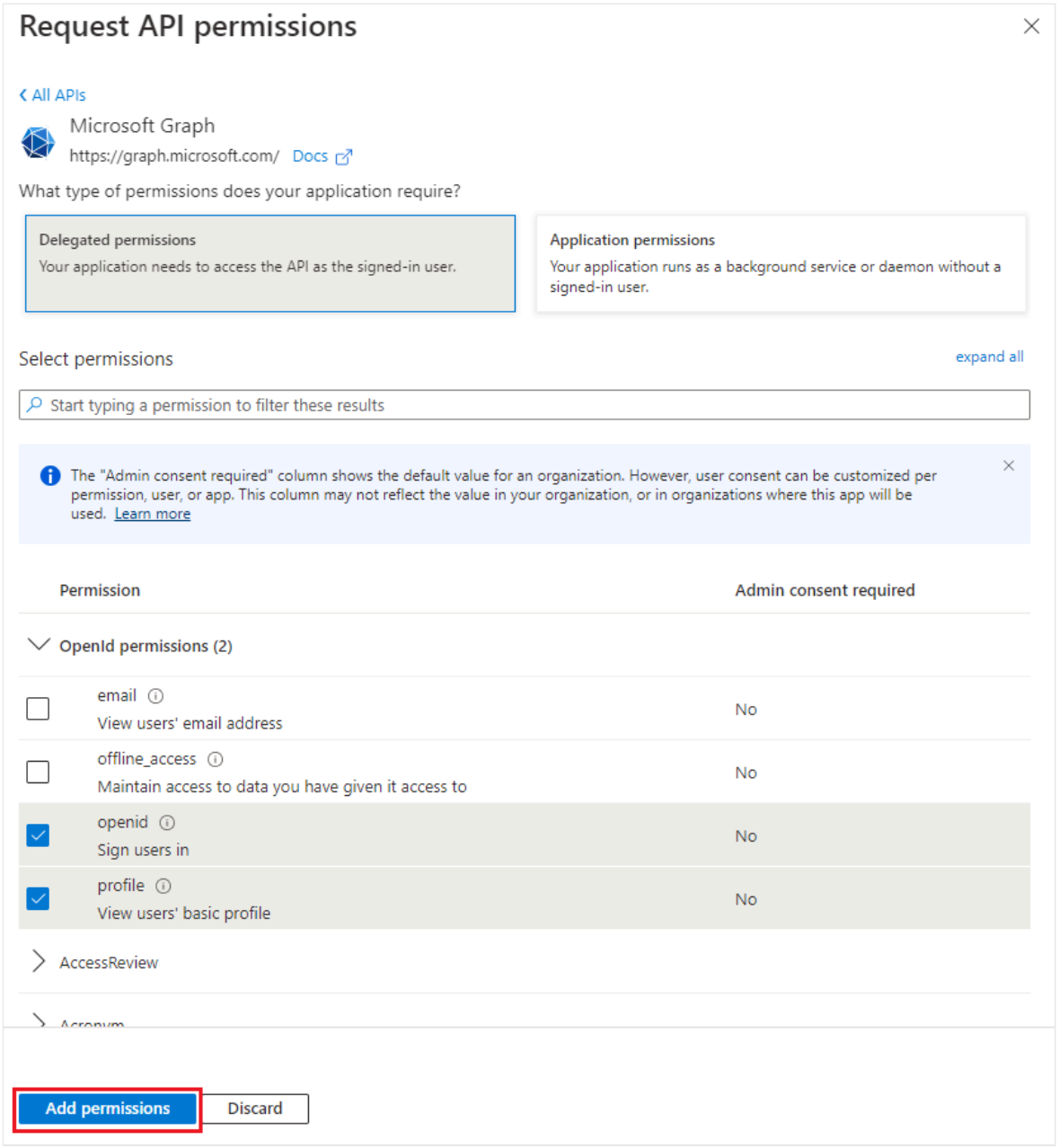
4. Select **Delegated permissions**.



5. In the **Select permissions** search box, search for the permissions your add-in needs. For example, for an Outlook add-in, you might use `profile`, `openid`, `Files.ReadWrite`, and `Mail.Read`.

> **Note**
>
> The `User.Read` permission may already be listed by default. It's a good practice to only request permissions that are needed, so we recommend that you uncheck the box for this permission if your add-in doesn't actually need it.

6. Select the checkbox for each permission as it appears. Note that the permissions will not remain visible in the list as you select each one. After selecting the permissions that your add-in needs, select **Add permissions**.



7. Select **Grant admin consent for [tenant name]**. Select **Yes** for the confirmation that appears.

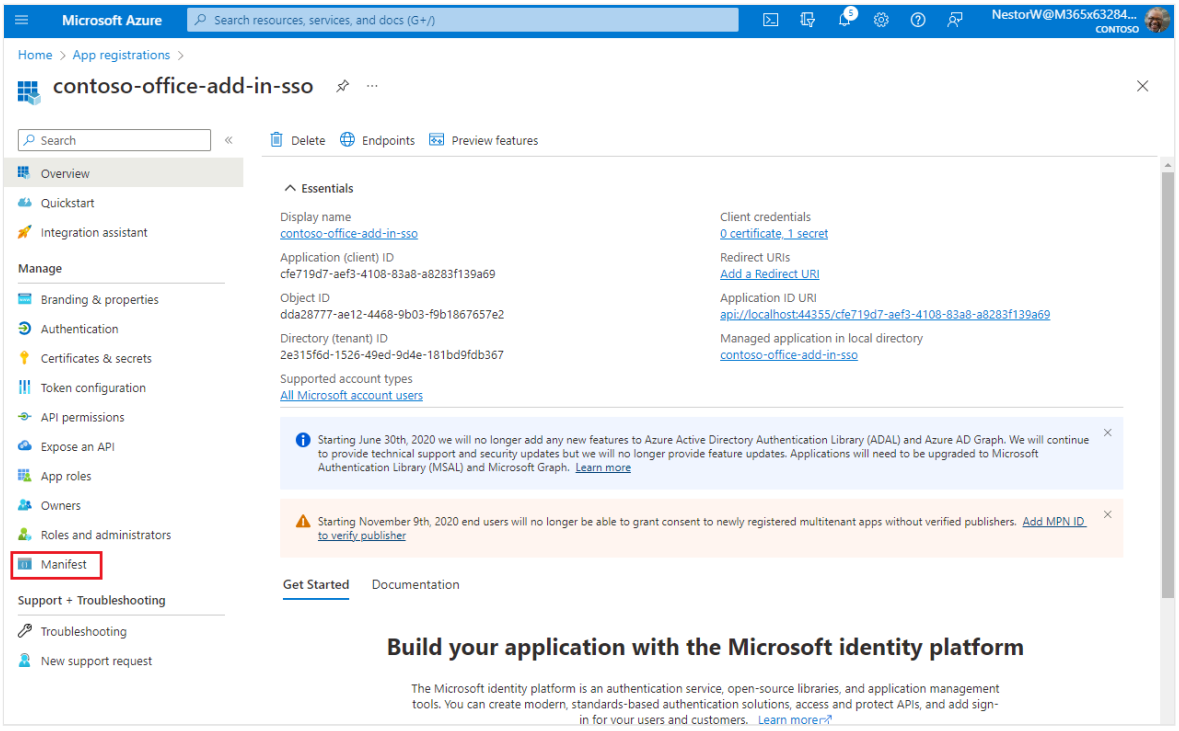
Configure access token version

You must define the access token version that is acceptable for your app. This configuration is made in the Azure Active Directory application manifest.

Define the access token version

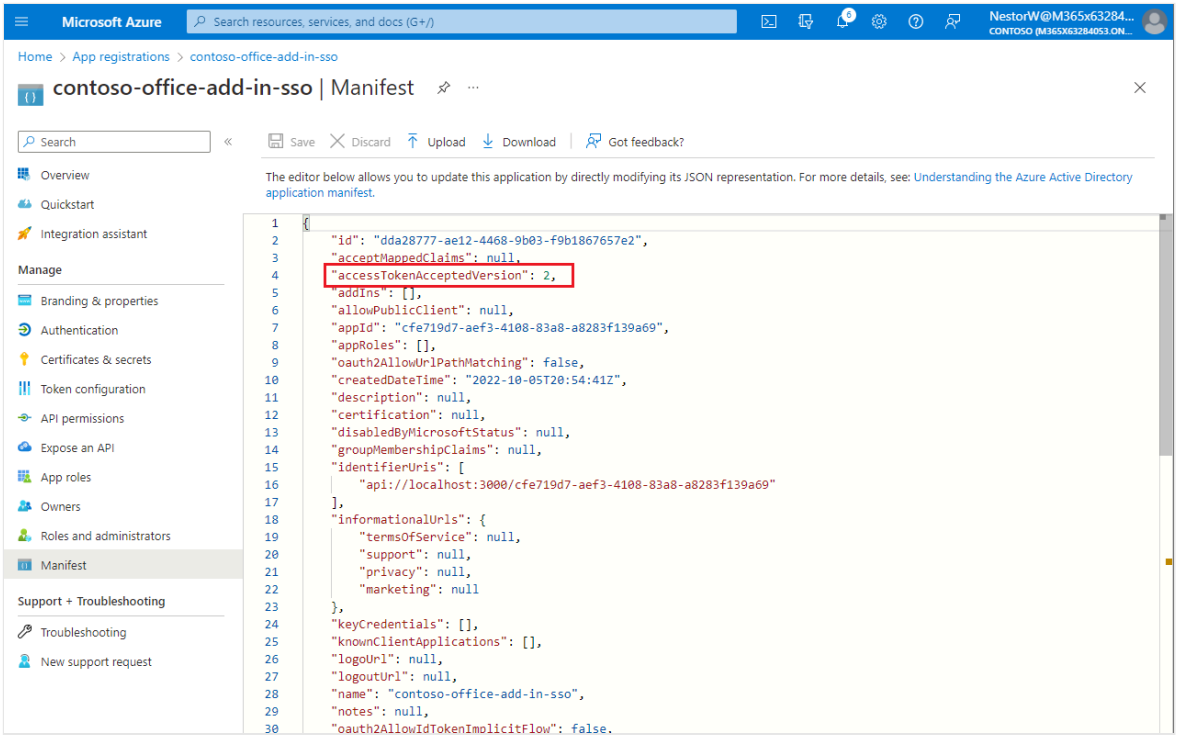
The access token version can change if you chose an account type other than **Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)**. Use the following steps to ensure the access token version is correct for Office SSO usage.

1. From the left pane, select **Manifest**.



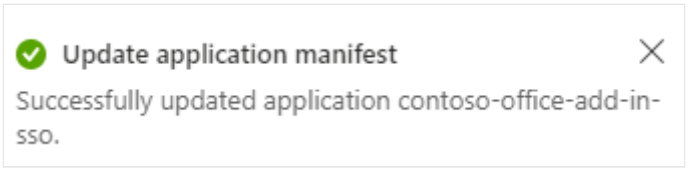
The Azure Active Directory application manifest appears.

2. Enter **2** as the value for the `accessTokenAcceptedVersion` property.



3. Select **Save**.

A message pops up on the browser stating that the manifest was updated successfully.



Congratulations! You've completed the app registration to enable SSO for your Office add-in.

Configure the add-in

1. Open the `\Begin` folder in the cloned project in your code editor.
2. Open the `.ENV` file and use the values that you copied earlier from the **Office-Add-in-NodeJS-SSO** app registration. Set the values as follows:

   Expand table

Name	Value
CLIENT_ID	**Application (client) ID** from app registration overview page.
CLIENT_SECRET	**Client secret** saved from **Certificates & Secrets** page.

   The values should **not** be in quotation marks. When you are done, the file should be similar to the following:

   JavaScript

   ```
   CLIENT_ID=8791c036-c035-45eb-8b0b-265f43cc4824
   CLIENT_SECRET=X7szTuPwKNts41:-/fa3p.p@l6zsyI/p
   NODE_ENV=development
   SERVER_SOURCE=<https://localhost:3000>
   ```

3. Open the add-in manifest file "manifest\manifest_local.xml" and then scroll to the bottom of the file. Just above the `</VersionOverrides>` end tag, you'll find the following markup.

   XML

   ```
   <WebApplicationInfo>
     <Id>$app-id-guid$</Id>
     <Resource>api://localhost:3000/$app-id-guid$</Resource>
     <Scopes>
         <Scope>Files.Read</Scope>
         <Scope>profile</Scope>
         <Scope>openid</Scope>
   ```

```
  </Scopes>
</WebApplicationInfo>
```

4. Replace the placeholder "$app-id-guid$" *in both places* in the markup with the **Application ID** that you copied when you created the **Office-Add-in-NodeJS-SSO** app registration. The "$" symbols are not part of the ID, so don't include them. This is the same ID you used for the CLIENT_ID in the .ENV file.

   > ⓘ **Note**
   >
   > The **<Resource>** value is the **Application ID URI** you set when you registered the add-in. The **<Scopes>** section is used only to generate a consent dialog box if the add-in is sold through AppSource.

5. Open the `\public\javascripts\fallback-msal\authConfig.js` file. Replace the placeholder "$app-id-guid$" with the application ID that you saved from the **Office-Add-in-NodeJS-SSO** app registration you created previously.

6. Save the changes to the file.

Code the client-side

Call our web server REST API

1. In your code editor, open the file `public\javascripts\ssoAuthES6.js`. It already has code that ensures that Promises are supported, even in the Trident (Internet Explorer 11) webview control, and an `Office.onReady` call to assign a handler to the add-in's only button.

   > ⓘ **Note**
   >
   > As the name suggests, the ssoAuthES6.js uses JavaScript ES6 syntax because using `async` and `await` best shows the essential simplicity of the SSO API. When the localhost server is started, this file is transpiled to ES5 syntax so that the sample will support Trident.

2. In the `getFileNameList` function, replace `TODO 1` with the following code. About this code, note:

- The function `getFileNameList` is called when the user chooses the **Get OneDrive File Names** button on the task pane.
- It calls the `callWebServerAPI` function specifying which REST API to call. This returns JSON containing a list of file names from the user's OneDrive.
- The JSON is passed to the `writeFileNamesToOfficeDocument` function to list the file names in the document.

JavaScript

```
try {
    const jsonResponse = await callWebServerAPI('GET',
'/getuserfilenames');
    if (jsonResponse === null) {
        // Null is returned when a message was displayed to the user
        // regarding an authentication error that cannot be resolved.
        return;
    }
    await writeFileNamesToOfficeDocument(jsonResponse);
    showMessage('Your OneDrive filenames are added to the document.');
} catch (error) {
    console.log(error.message);
    showMessage(error.message);
}
```

3. In the `callWebServerAPI` function, replace `TODO 2` with the following code. About this code, note:

   - The function calls `getAccessToken` which is our own function that encapsulates using Office SSO or MSAL fallback as necessary to get the token. If it returns a null token, a message was shown for an auth error condition that cannot be resolved, so the function also returns null.
   - The function uses the `fetch` API to call the web server and if successful, returns the JSON body.

   JavaScript

   ```
   const accessToken = await getAccessToken(authSSO);
   if (accessToken === null) {
       return null;
   }
   const response = await fetch(path, {
       method: method,
       headers: {
           'Content-Type': 'application/json',
           'Authorization': 'Bearer ' + accessToken,
       },
   });
   ```

```
// Check for success condition: HTTP status code 2xx.
if (response.ok) {
    return response.json();
}
```

4. In the `callWebServerAPI` function, replace `TODO 3` with the following code. About this code, note:

    - This code handles the scenario where the SSO token expired. If so we need to call `Office.auth.getAccessToken` to get a refreshed token. The simplest way is to make a recursive call which results in a new call to `Office.auth.getAccessToken`. The `retryRequest` parameter ensures the recursive call is only attempted once.
    - The `TokenExpiredError` string is set by our web server whenever it detects an expired token.

    JavaScript

    ```
     // Check for fail condition: Is SSO token expired? If so, retry the
    call which will get a refreshed token.
    const jsonBody = await response.json();
    if (
        authSSO === true &&
        jsonBody != null &&
        jsonBody.type === 'TokenExpiredError'
    ) {
        if (!retryRequest) {
            return callWebServerAPI(method, path, true); // Try the call
    again. The underlying call to Office JS getAccessToken will refresh the
    token.
        } else {
            // Indicates a second call to retry and refresh the token
    failed.
            authSSO = false;
            return callWebServerAPI(method, path, true); // Try the call
    again, but now using MSAL fallback auth.
        }
    }
    ```

5. In the `callWebServerAPI` function, replace `TODO 4` with the following code. About this code, note:

    - The `Microsoft Graph` string is set by our web server whenever a Microsoft Graph call fails.

    JavaScript

```
// Check for fail condition: Did we get a Microsoft Graph API error,
which is returned as bad request (403)?
if (response.status === 403 && jsonBody.type === 'Microsoft Graph') {
    throw new Error('Microsoft Graph error: ' + jsonBody.errorDetails);
}
```

6. In the `callWebServerAPI` function, replace `TODO 5` with the following code.

JavaScript

```
// Handle other errors.
throw new Error(
    'Unknown error from web server: ' + JSON.stringify(jsonBody)
);
```

7. In the `getAccessToken` function, replace `TODO 6` with the following code. About this code, note:

   - `authSSO` tracks if we are using SSO, or using MSAL fallback. If SSO is used, the function calls `Office.auth.getAccessToken` and returns the token.
   - Errors are handled by the `handleSSOErrors` function which will return a token if it switches to fallback MSAL authentication.
   - Fallback authentication uses the MSAL library to sign in the user. The add-in itself is an SPA, and uses an SPA app registration to access the web server.

JavaScript

```
if (authSSO) {
    try {
        // Get the access token from Office host using SSO.
        // Note that Office.auth.getAccessToken modifies the options
parameter. Create a copy of the object
        // to avoid modifying the original object.
        const options = JSON.parse(JSON.stringify(ssoOptions));
        const token = await Office.auth.getAccessToken(options);
        return token;
    } catch (error) {
        console.log(error.message);
        return handleSSOErrors(error);
    }
} else {
    // Get access token through MSAL fallback.
    try {
        const accessToken = await getAccessTokenMSAL();
        return accessToken;
    } catch (error) {
        console.log(error);
        throw new Error(
```

```
            'Cannot get access token. Both SSO and fallback auth
failed. ' +
                error
        );
    }
}
```

8. In the `handleSSOErrors` function, replace `TODO 7` with the following code. For more information about these errors, see Troubleshoot SSO in Office Add-ins.

JavaScript

```
switch (error.code) {
    case 13001:
        // No one is signed into Office. If the add-in cannot be
effectively used when no one
        // is logged into Office, then the first call of getAccessToken
should pass the
        // `allowSignInPrompt: true` option. Since this sample does
that, you should not see
        // this error.
        showMessage(
            'No one is signed into Office. But you can use many of the
add-ins functions anyway. If you want to log in, press the Get OneDrive
File Names button again.'
        );
        break;
    case 13002:
        // The user aborted the consent prompt. If the add-in cannot be
effectively used when consent
        // has not been granted, then the first call of getAccessToken
should pass the `allowConsentPrompt: true` option.
        showMessage(
            'You can use many of the add-ins functions even though you
have not granted consent. If you want to grant consent, press the Get
OneDrive File Names button again.'
        );
        break;
    case 13006:
        // Only seen in Office on the web.
        showMessage(
            'Office on the web is experiencing a problem. Please sign
out of Office, close the browser, and then start again.'
        );
        break;
    case 13008:
        // Only seen in Office on the web.
        showMessage(
            'Office is still working on the last operation. When it
completes, try this operation again.'
        );
        break;
    case 13010:
```

```
        // Only seen in Office on the web.
        showMessage(
            "Follow the instructions to change your browser's zone
configuration."
        );
        break;
```

9. Replace `TODO 8` with the following code. For any errors that can't be handled the code switches to fallback authentication using MSAL.

JavaScript

```
default: //recursive call.
        // For all other errors, including 13000, 13003, 13005, 13007,
13012, and 50001, fall back
        // to MSAL sign-in.
        showMessage('SSO failed. Trying fallback auth.');
        authSSO = false;
        return getAccessToken(false);
}
return null; // Return null for errors that show a message to the user.
```

Code the web server REST API

The web server provides REST APIs for the client to call. For example, the REST API `/getuserfilenames` gets a list of filenames from the user's OneDrive folder. Each REST API call requires an access token by the client to ensure the correct client is accessing their data. The access token is exchanged for a Microsoft Graph token through the On-Behalf-Of flow (OBO). The new Microsoft Graph token is cached by the MSAL library for subsequent API calls. It's never sent outside of the web server. For more information, see Middle-tier access token request

Create the route and implement On-Behalf-Of flow

1. Open the file `routes\getFilesRoute.js` and replace `TODO 9` with the following code. About this code, note:

    - It calls `authHelper.validateJwt`. This ensures the access token is valid and hasn't been tampered with.
    - For more information, see Validating tokens.

JavaScript

```
router.get(
 "/getuserfilenames",
 authHelper.validateJwt,
 async function (req, res) {
   // TODO 10: Exchange the access token for a Microsoft Graph token
   //          by using the OBO flow.
 }
);
```

2. Replace `TODO 10` with the following code. About this code, note:

   - It only requests the minimum scopes it needs, such as `files.read`.
   - It uses the MSAL `authHelper` to perform the OBO flow in the call to `acquireTokenOnBehalfOf`.

JavaScript

```
try {
  const authHeader = req.headers.authorization;
  let oboRequest = {
    oboAssertion: authHeader.split(' ')[1],
    scopes: ["files.read"],
  };

  // The Scope claim tells you what permissions the client application
has in the service.
  // In this case we look for a scope value of access_as_user, or full
access to the service as the user.
  const tokenScopes = jwt.decode(oboRequest.oboAssertion).scp.split('
');
  const accessAsUserScope = tokenScopes.find(
    (scope) => scope === 'access_as_user'
  );
  if (!accessAsUserScope) {
    res.status(401).send({ type: "Missing access_as_user" });
    return;
  }
  const cca = authHelper.getConfidentialClientApplication();
  const response = await cca.acquireTokenOnBehalfOf(oboRequest);
  // TODO 11: Call Microsoft Graph to get list of filenames.
} catch (err) {
  // TODO 12: Handle any errors.
}
```

3. Replace `TODO 11` with the following code. About this code, note:

   - It constructs the URL for the Microsoft Graph API call and then makes the call via the `getGraphData` function.
   - It returns errors by sending an HTTP 500 response along with details.

- On success it returns the JSON with the filename list to the client.

JavaScript

```
// Minimize the data that must come from MS Graph by specifying only
the property we need ("name")
// and only the top 10 folder or file names.
const rootUrl = '/me/drive/root/children';

// Note that the last parameter, for queryParamsSegment, is hardcoded.
If you reuse this code in
// a production add-in and any part of queryParamsSegment comes from
user input, be sure that it is
// sanitized so that it cannot be used in a Response header injection
attack.
const params = '?$select=name&$top=10';

const graphData = await getGraphData(
  response.accessToken,
  rootUrl,
  params
);

// If Microsoft Graph returns an error, such as invalid or expired
token,
// there will be a code property in the returned object set to a HTTP
status (e.g. 401).
// Return it to the client. On client side it will get handled in the
fail callback of `makeWebServerApiCall`.
if (graphData.code) {
  res
    .status(403)
    .send({
      type: "Microsoft Graph",
      errorDetails:
        "An error occurred while calling the Microsoft Graph API.\n" +
        graphData,
    });
} else {
  // MS Graph data includes OData metadata and eTags that we don't
need.
  // Send only what is actually needed to the client: the item names.
  const itemNames = [];
  const oneDriveItems = graphData["value"];
  for (let item of oneDriveItems) {
    itemNames.push(item["name"]);
  }

  res.status(200).send(itemNames);
}
// TODO 12: Check for expired token.
```

4. Replace `TODO 12` with the following code. This code specifically checks if the token expired because the client can request a new token and call again.

```JavaScript
} catch (err) {
   // On rare occasions the SSO access token is unexpired when Office validates it,
   // but expires by the time it is used in the OBO flow. Microsoft identity platform will respond
   // with "The provided value for the 'assertion' is not valid. The assertion has expired."
   // Construct an error message to return to the client so it can refresh the SSO token.
   if (err.errorMessage.indexOf('AADSTS500133') !== -1) {
     res.status(401).send({ type: "TokenExpiredError", errorDetails: err });
   } else {
     res.status(403).send({ type: "Unknown", errorDetails: err });
   }
}
```

The sample must handle both fallback authentication through MSAL and SSO authentication through Office. The sample will try SSO first, and the `authSSO` boolean at the top of the file tracks if the sample is using SSO or has switched to fallback auth.

Run the project

1. Ensure that you have some files in your OneDrive so that you can verify the results.
2. Open a command prompt in the root of the `\Begin` folder.
3. Run the command `npm install` to install all package dependencies.
4. Run the command `npm start` to start the middle-tier server.
5. You need to sideload the add-in into an Office application (Excel, Word, or PowerPoint) to test it. The instructions depend on your platform. There are links to instructions at Sideload an Office Add-in for Testing.
6. In the Office application, on the **Home** ribbon, select the **Show Add-in** button in the **SSO Node.js** group to open the task pane add-in.
7. Click the **Get OneDrive File Names** button. If you're logged into Office with either a Microsoft 365 Education or work account, or a Microsoft account, and SSO is working as expected the first 10 file and folder names in your OneDrive for

Business are inserted into the document. (It may take as much as 15 seconds the first time.) If you're not logged in, or you're in a scenario that doesn't support SSO, or SSO isn't working for any reason, you'll be prompted to sign in. After you sign in, the file and folder names appear.

> **Note**
>
> If you were previously signed into Office with a different ID, and some Office applications that were open at the time are still open, Office may not reliably change your ID even if it appears to have done so. If this happens, the call to Microsoft Graph may fail or data from the previous ID may be returned. To prevent this, be sure to *close all other Office applications* before you press **Get OneDrive File Names**.

Stop running the project

When you're ready to stop the middle-tier server and uninstall the add-in, follow these instructions:

1. Run the following command to stop the middle-tier server.

   ```
   command line
   npm stop
   ```

2. To uninstall or remove the add-in, see the specific sideload article you used for details.

Security notes

- The `/getuserfilenames` route in `getFilesroute.js` uses a literal string to compose the call for Microsoft Graph. If you change the call so that any part of the string comes from user input, sanitize the input so that it cannot be used in a Response header injection attack.

- In `app.js` the following content security policy is in place for scripts. You may want to specify additional restrictions depending on your add-in security needs.

  `"Content-Security-Policy": "script-src https://appsforoffice.microsoft.com https://ajax.aspnetcdn.com https://alcdn.msauth.net " +`

```
process.env.SERVER_SOURCE,
```

Always follow security best practices in the Microsoft identity platform documentation.

Troubleshoot error messages for single sign-on (SSO)

07/30/2025

This article provides some guidance about how to troubleshoot problems with single sign-on (SSO) in Office Add-ins, and how to make your SSO-enabled add-in robustly handle special conditions or errors.

> **Note**
>
> The Single Sign-on API is currently supported for Word, Excel, Outlook, and PowerPoint. For more information about where the Single Sign-on API is currently supported, see IdentityAPI requirement sets. If you're working with an Outlook add-in, be sure to enable Modern Authentication for the Microsoft 365 tenancy. For information about how to do this, see Enable or disable modern authentication for Outlook in Exchange Online.

Debugging tools

We strongly recommend that you use a tool that can intercept and display the HTTP Requests from, and Responses to, your add-in's web service when you are developing. Some of the most popular are:

- Fiddler: Free for 10 days (Documentation)
- Charles: Free for 30 days. (Documentation)
- Requestly: Free basic service. (Documentation)

Causes and handling of errors from getAccessToken

For examples of the error handling described in this section, see:

- HomeES6.js in Office-Add-in-ASPNET-SSO
- ssoAuthES6.js in Office-Add-in-NodeJS-SSO

13000

The getAccessToken API isn't supported by the add-in or the Office version.

- The version of Office does not support SSO. The required version is Microsoft 365 subscription, in any monthly channel.
- The add-in manifest is missing the proper WebApplicationInfo section.

Your add-in should respond to this error by falling back to an alternate system of user authentication. For more information, see Requirements and Best Practices.

13001

The user isn't signed into Office. In most scenarios, you should prevent this error from ever being seen by passing the option `allowSignInPrompt: true` in the `AuthOptions` parameter.

But there may be exceptions. For example, you want the add-in to open with features that require a logged in user; but *only if* the user is *already* logged into Office. If the user isn't logged in, you want the add-in to open with an alternate set of features that do not require that the user is signed in. In this case, logic which runs when the add-in launches calls `getAccessToken` without `allowSignInPrompt: true`. Use the 13001 error as the flag to tell the add-in to present the alternate set of features.

Another option is to respond to 13001 by falling back to an alternate system of user authentication. This will sign the user into Microsoft Entra ID, but not sign the user into Office.

This error doesn't typically occur in Office on the web. If the user's cookie expires, Office on the web returns error 13006. However, if a user accesses Outlook on the web from Firefox with Enhanced Tracking Protection turned on, they'll encounter error 13001.

13002

The user aborted sign in or consent; for example, by choosing **Cancel** on the consent dialog.

- If your add-in provides functions that don't require the user to be signed in (or to have granted consent), then your code should catch this error and allow the add-in to stay running.
- If the add-in requires a signed-in user who has granted consent, your code should have a sign-in button appear.

13003

User Type not supported. The user isn't signed into Office with a valid Microsoft account or Microsoft 365 Education or work account. This may happen if Office runs with an on-premises domain account, for example. Your code should fall back to an alternate system of user authentication. In Outlook, this error may also occur if modern authentication is disabled for

the user's tenant in Exchange Online. For more information, see Requirements and Best Practices.

13004

Invalid Resource. (This error should only be seen in development.) The add-in manifest hasn't been configured correctly. Update the manifest. For more information, see Validate an Office Add-in's manifest. The most common problem is that the `<Resource>` element (in the `<WebApplicationInfo>` element) has a domain that does not match the domain of the add-in. Although the protocol part of the Resource value should be "api" not "https"; all other parts of the domain name (including port, if any) should be the same as for the add-in.

13005

Invalid Grant. This usually means that Office has not been pre-authorized to the add-in's web service. For more information, see Create the service application and Register an Office Add-in that uses single sign-on (SSO) with the Microsoft identity platform. This also may happen if the user has not granted your service application permissions to their `profile`, or has revoked consent. Your code should fall back to an alternate system of user authentication.

Another possible cause, during development, is that your add-in using Internet Explorer, and you are using a self-signed certificate. (To determine which browser or webview is being used by the add-in, see Browsers and webview controls used by Office Add-ins.)

13006

Client Error. This error is only seen in **Office on the web**. Your code should suggest that the user sign out and then restart the Office browser session.

13007

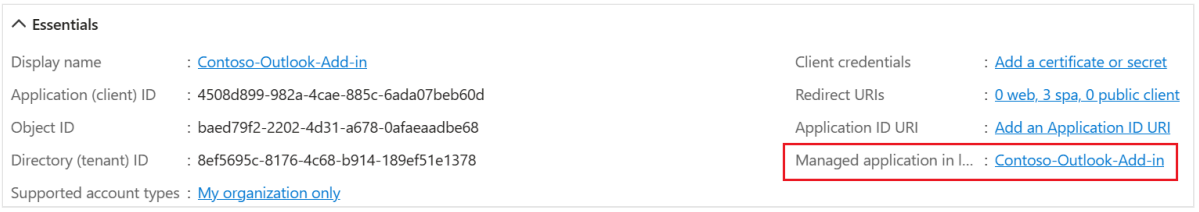
The Office application was unable to get an access token to the add-in's web service.

- If this error occurs during development, be sure that your add-in registration and add-in manifest specify the `profile` permission (and the `openid` permission, if you are using MSAL.NET). For more information, see Register an Office Add-in that uses single sign-on (SSO) with the Microsoft identity platform.
- In production, an account mismatch could cause this error. For example, if the user attempts to sign in with a personal Microsoft account (MSA) when a Work or school account was expected. For these cases, your code should fall back to an alternate system

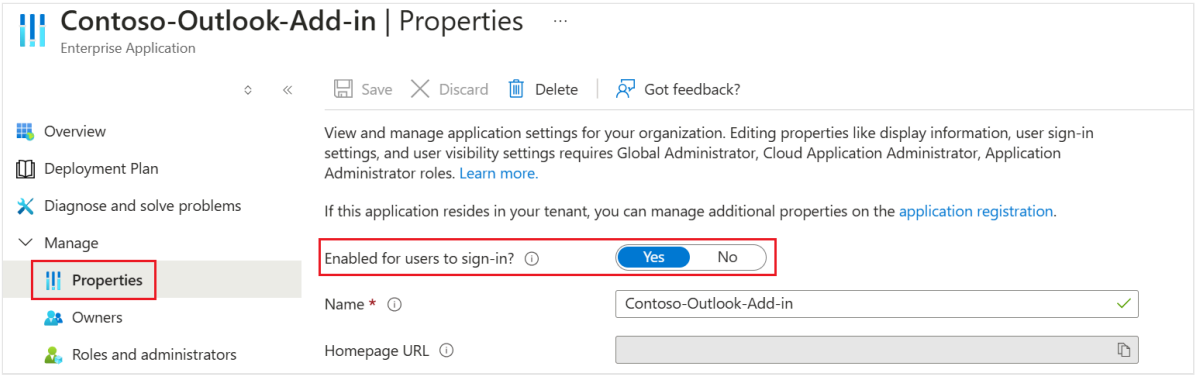
of user authentication. For more information on account types, see Identity and account types for single- and multi-tenant apps.

- Make sure your application is enabled for users to sign-in for your organization.

    1. Sign in to the Microsoft Azure portal.
    2. Go to your add-in's app registration.
    3. On the **Overview** page, select **Managed application in local directory**.



    4. Select **Manage** > **Properties**, and ensure that the value of **Enabled for users to sign-in?** is **Yes**.



13008

The user triggered an operation that calls `getAccessToken` before a previous call of `getAccessToken` completed. This error is only seen on **Office on the web**. Your code should ask the user to repeat the operation after the previous operation has completed.

13010

The user is running the add-in in Office on Microsoft Edge. The user's Microsoft 365 domain, and the `login.microsoftonline.com` domain, are in a different security zones in the browser settings. This error is only seen on **Office on the web**. If this error is returned, the user will have already seen an error explaining this and linking to a page about how to change the zone configuration. If your add-in provides functions that don't require the user to be signed in, then your code should catch this error and allow the add-in to stay running.

13012

There are several possible causes.

- The add-in is running on a platform that does not support the `getAccessToken` API. For example, it isn't supported on iPad. See also Identity API requirement sets.
- The Office document was opened from the **Files** tab of a Teams channel using the **Edit in Teams** option on the **Open** dropdown menu. The `getAccessToken` API isn't supported in this scenario.
- The `forMSGraphAccess` option was passed in the call to `getAccessToken` and the user obtained the add-in from AppSource. In this scenario, the tenant admin has not granted consent to the add-in for the Microsoft Graph scopes (permissions) that it needs. Recalling `getAccessToken` with the `allowConsentPrompt` will not solve the problem because Office is allowed to prompt the user for consent to only the Microsoft Entra ID `profile` scope.

Your code should fall back to an alternate system of user authentication.

In development, the add-in is sideloaded in Outlook and the `forMSGraphAccess` option was passed in the call to `getAccessToken`.

13013

The `getAccessToken` was called too many times in a short amount of time, so Office throttled the most recent call. This is usually caused by an infinite loop of calls to the method. There are scenarios when recalling the method is advisable. However, your code should use a counter or flag variable to ensure that the method isn't recalled repeatedly. If the same "retry" code path is running again, the code should fall back to an alternate system of user authentication. For a code example, see how the `retryGetAccessToken` variable is used in HomeES6.js or ssoAuthES6.js.

50001

This error (which isn't specific to `getAccessToken`) may indicate that the browser has cached an old copy of the office.js files. When you're developing, clear the browser's cache. Another possibility is that the version of Office isn't recent enough to support SSO. On Windows, the minimum version is Version 1911 (Build 12215.20006). On Mac, it's Version 16.32 (19102902).

In a production add-in, the add-in should respond to this error by falling back to an alternate system of user authentication. For more information, see Requirements and Best Practices.

Errors on the server-side from Microsoft Entra ID

For samples of the error-handling described in this section, see:

- Office-Add-in-ASPNET-SSO
- Office-Add-in-NodeJS-SSO

Conditional access / Multifactor authentication errors

In certain configurations of identity in Microsoft Entra ID and Microsoft 365, it is possible for some resources that are accessible with Microsoft Graph to require multifactor authentication (MFA), even when the user's Microsoft 365 tenancy does not. When Microsoft Entra ID receives a request for a token to the MFA-protected resource, via the on-behalf-of flow, it returns to your add-in's web service a JSON message that contains a `claims` property. The claims property has information about what further authentication factors are needed.

Your code should test for this `claims` property. Depending on your add-in's architecture, you may test for it on the client-side, or you may test for it on the server-side and relay it to the client. You need this information in the client because Office handles authentication for SSO add-ins. If you relay it from the server-side, the message to the client can be either an error (such as `500 Server Error` or `401 Unauthorized`) or in the body of a success response (such as `200 OK`). In either case, the (failure or success) callback of your code's client-side AJAX call to your add-in's web API should test for this response.

Regardless of your architecture, if the claims value has been sent from Microsoft Entra ID, your code should recall `getAccessToken` and pass the option `authChallenge: CLAIMS-STRING-HERE` in the `options` parameter. When Microsoft Entra ID sees this string, it prompts the user for the additional factors and then returns a new access token which will be accepted in the on-behalf-of flow.

Consent missing errors

If Microsoft Entra ID has no record that consent (to the Microsoft Graph resource) was granted to the add-in by the user (or tenant administrator), Microsoft Entra ID will send an error message to your web service. Your code must tell the client (in the body of a `403 Forbidden` response, for example).

If the add-in needs Microsoft Graph scopes that can only be consented to by an admin, your code should throw an error. If the only scopes that are needed can be consented to by the user, then your code should fall back to an alternate system of user authentication.

Invalid or missing scope (permission) errors

This kind of error should only be seen in development.

- Your server-side code should send a `403 Forbidden` response to the client which should log the error to the console or record it in a log.
- Be sure your add-in manifest Scopes section specifies all needed permissions. And be sure your registration of the add-in's web service specifies the same permissions. Check for spelling mistakes too. For more information, see Register an Office Add-in that uses single sign-on (SSO) with the Microsoft identity platform.

Invalid audience error in the access token for Microsoft Graph

Your server-side code should send a `403 Forbidden` response to the client which should present a friendly message to the user and possibly also log the error to the console or record it in a log.

Enable single sign-on in an Office Add-in with nested app authentication

08/06/2025

You can use the MSAL.js library with nested app authentication to use single sign-on (SSO) from your Office Add-in. Using nested app authentication (NAA) offers several advantages over the On-Behalf-Of (OBO) flow.

- You only need to use the MSAL.js library and don't need the `getAccessToken` function in Office.js.
- You can call services such as Microsoft Graph with an access token from your client code as an SPA. There's no need for a middle-tier server.
- You can use incremental and dynamic consent for scopes.
- You don't need to preauthorize your hosts (for example, Teams, Office) to call your endpoints.

NAA supported accounts and hosts

NAA supports both Microsoft Accounts and Microsoft Entra ID (work/school) identities. It doesn't support Azure Active Directory B2C for business-to-consumer identity management scenarios. The following table explains the current support by platform. Platforms listed as generally available (GA) are ready for production usage in your add-in.

Expand table

Application	Web	Windows	Mac	iOS/iPad	Android
Excel	In preview	In preview	In preview	In preview on iPad	Not applicable
Outlook	GA	GA	GA	GA (iOS)	GA
PowerPoint	In preview	In preview	In preview	In preview on iPad	Not applicable
Word	In preview	In preview	In preview	In preview on iPad	Not applicable

> **Important**
>
> To use NAA on platforms that are still in preview (Word, Excel, and PowerPoint), join the Microsoft 365 Insider Program and choose **Current Channel (Preview)**. Don't use NAA in production add-ins for any preview platforms. We invite you to try out NAA in test or

development environments and welcome feedback on your experience through GitHub (see the **Feedback** section at the end of this page).

For information on using NAA in Microsoft Teams, see Nested app authentication in Microsoft Teams.

Register your single-page application

You'll need to create a Microsoft Azure App registration for your add-in on the Azure portal. The app registration must have at minimum:

- A name
- A supported account type
- An SPA redirect

If your add-in requires additional app registration beyond NAA and SSO, see Single-page application: App registration.

Add a trusted broker through SPA redirect

To enable NAA, your app registration must include a specific redirect URI to indicate to the Microsoft identity platform that your add-in allows itself to be brokered by supported hosts. The redirect URI of the application must be of type **Single Page Application** and conform to the following scheme.

```
brk-multihub://your-add-in-domain
```

Your domain must include only the origin and not its subpaths. For example:

✔ brk-multihub://localhost:3000
✔ brk-multihub://www.contoso.com
✘ brk-multihub://www.contoso.com/go

Trusted broker groups are dynamic by design and can be updated in the future to include additional hosts where your add-in may use NAA flows. Currently the brk-multihub group includes Office Word, Excel, PowerPoint, Outlook, and Teams (for when Office is activated inside).

Configure MSAL config to use NAA

Configure your add-in to use NAA by calling the `createNestablePublicClientApplication` function in MSAL. MSAL returns a public client application that can be nested in a native application host (for example, Outlook) to acquire tokens for your application.

The following steps show how to enable NAA in the `taskpane.js` or `taskpane.ts` file in a project built with `yo office` (**Office Add-in Task Pane** project).

1. Add the `@azure/msal-browser` package to the `dependencies` section of the `package.json` file for your project. For more information on this package, see Microsoft Authentication Library for JavaScript (MSAL.js) for Browser-Based Single-Page Applications. We recommend using the latest version of the package (at time of the last article update it was 3.26.0).

   JSON

   ```
   "dependencies": {
       "@azure/msal-browser": "^3.27.0",
       ...
   ```

2. Save and run `npm install` to install `@azure/msal-browser`.

3. Add the following code to the top of the `taskpane.js` or `taskpane.ts` file. This will import the MSAL browser library.

   JavaScript

   ```
   import { createNestablePublicClientApplication } from "@azure/msal-browser";
   ```

Initialize the public client application

Next, you need to initialize MSAL and get an instance of the public client application. This is used to get access tokens when needed. We recommend that you put the code that creates the public client application in the `Office.onReady` method.

- In your `Office.onReady` function, add a call to `createNestablePublicClientApplication` as shown below. Replace the `Enter_the_Application_Id_Here` placeholder with the Azure app ID you saved previously.

  JavaScript

  ```
  let pca = undefined;
  Office.onReady(async (info) => {
  ```

```
  if (info.host) {
    document.getElementById("sideload-msg").style.display = "none";
    document.getElementById("app-body").style.display = "flex";
    document.getElementById("run").onclick = run;

    // Initialize the public client application
    pca = await createNestablePublicClientApplication({
      auth: {
        clientId: "Enter_the_Application_Id_Here",
        authority: "https://login.microsoftonline.com/common"
      },
    });
  }
});
```

> **Note**
>
> The previous code sample sets the **authority** to **common**, which supports work and school accounts or personal Microsoft accounts. If you want to configure a single tenant or other account types, see Application configuration options for additional authority options.

Acquire your first token

The tokens acquired by MSAL.js via NAA will be issued for your Azure app registration ID. In this code sample, you acquire a token for the Microsoft Graph API. If the user has an active session with Microsoft Entra ID the token is acquired silently. If not, the library prompts the user to sign in interactively. The token is then used to call the Microsoft Graph API.

The following steps show the pattern to use for acquiring a token.

1. Specify your scopes. NAA supports incremental and dynamic consent so always request the minimum scopes needed for your code to complete its task.
2. Call `acquireTokenSilent`. This will get the token without requiring user interaction.
3. If `acquireTokenSilent` fails, call `acquireTokenPopup` to display an interactive dialog for the user. `acquireTokenSilent` can fail if the token expired, or the user has not yet consented to all of the requested scopes.

The following code shows how to implement this authentication pattern in your own project.

1. Replace the `run` function in `taskpane.js` or `taskpane.ts` with the following code. The code specifies the minimum scopes needed to read the user's files.

```
JavaScript
```

```
async function run() {
// Specify minimum scopes needed for the access token.
const tokenRequest = {
  scopes: ["Files.Read", "User.Read", "openid", "profile"],
};
let accessToken = null;

// TODO 1: Call acquireTokenSilent.

// TODO 2: Call acquireTokenPopup.

// TODO 3: Log error if token still null.

// TODO 4: Call the Microsoft Graph API.

}
```

> **ⓘ Important**
>
> The token request must include scopes other than just `offline_access`, `openid`, `profile`, or `email`. You can use any combination of the previous scopes, but you must include at least one additional scope. If not, the token request can fail.

2. Replace `TODO 1` with the following code. This code calls `acquireTokenSilent` to get the access token.

JavaScript

```
try {
  console.log("Trying to acquire token silently...");
  const userAccount = await pca.acquireTokenSilent(tokenRequest);
  console.log("Acquired token silently.");
  accessToken = userAccount.accessToken;
} catch (error) {
  console.log(`Unable to acquire token silently: ${error}`);
}
```

3. Replace `TODO 2` with the following code. This code checks if the access token is acquired. If not it attempts to interactively get the access token by calling `acquireTokenPopup`.

JavaScript

```
if (accessToken === null) {
  // Acquire token silent failure. Send an interactive request via popup.
  try {
    console.log("Trying to acquire token interactively...");
    const userAccount = await pca.acquireTokenPopup(tokenRequest);
```

```
    console.log("Acquired token interactively.");
    accessToken = userAccount.accessToken;
  } catch (popupError) {
    // Acquire token interactive failure.
    console.log(`Unable to acquire token interactively: ${popupError}`);
  }
}
```

4. Replace `TODO 3` with the following code. If both silent and interactive sign-in failed, log the error and return.

JavaScript

```
// Log error if both silent and popup requests failed.
if (accessToken === null) {
  console.error(`Unable to acquire access token.`);
  return;
}
```

Call an API

After acquiring the token, use it to call an API. The following example shows how to call the Microsoft Graph API by calling `fetch` with the token attached in the *Authorization* header.

- Replace `TODO 4` with the following code.

JavaScript

```
// Call the Microsoft Graph API with the access token.
const response = await fetch(
  `https://graph.microsoft.com/v1.0/me/drive/root/children?
$select=name&$top=10`,
  {
    headers: { Authorization: accessToken },
  }
);

if (response.ok) {
  // Write file names to the console.
  const data = await response.json();
  const names = data.value.map((item) => item.name);

  // Be sure the taskpane.html has an element with Id = item-subject.
  const label = document.getElementById("item-subject");

  // Write file names to task pane and the console.
  const nameText = names.join(", ");
  if (label) label.textContent = nameText;
  console.log(nameText);
```

```
} else {
  const errorText = await response.text();
  console.error("Microsoft Graph call failed - error text: " + errorText);
}
```

Once all the previous code is added to the `run` function, be sure a button on the task pane calls the `run` function. Then you can sideload the add-in and try out the code.

What is nested app authentication

Nested app authentication enables SSO for applications that are nested inside of supported Microsoft applications. For example, Excel on Windows runs your add-in inside a webview. In this scenario, your add-in is a nested application running inside Excel, which is the host. NAA also supports nested apps in Teams. For example, if a Teams tab is hosting Excel, and your add-in is loaded, it is nested inside Excel, which is also nested inside Teams. Again, NAA supports this nested scenario and you can access SSO to get user identity and access tokens of the signed in user.

Best practices

We recommend the following best practices when using MSAL.js with NAA.

Use silent authentication whenever possible

MSAL.js provides the `acquireTokenSilent` method that handles token renewal by making silent token requests without prompting the user. The method first looks for a valid cached token. If it doesn't find one, the library makes the silent request to Microsoft Entra ID and if there's an active user session, a fresh token is returned.

In certain cases, the `acquireTokenSilent` method's attempt to get the token fails. Some examples of this are when there's an expired user session with Microsoft Entra ID or a password change by the user, which requires user interaction. When the acquireTokenSilent fails, you need to call the interactive `acquireTokenPopup` token method.

Have a fallback when NAA isn't supported

While we strive to provide a high-degree of compatibility with these flows across the Microsoft ecosystem, your add-in may be loaded in an older Office host that does not support NAA. In these cases, your add-in won't support seamless SSO and you may need to fall back to an

alternate method of authenticating the user. In general you'll want to use the MSAL SPA authentication pattern with the Office JS dialog API.

Use the following code to check if NAA is supported when your add-in loads.

```
JavaScript

Office.context.requirements.isSetSupported("NestedAppAuth", "1.1");
```

For more information, see the following resources.

- Outlook sample: How to fall back and support Internet Explorer 11
- Authenticate and authorize with the Office dialog API.
- Microsoft identity sample for SPA and JavaScript
- Microsoft identity samples for various app types and frameworks

MSAL.js APIs supported by NAA

The following table shows which APIs are supported when NAA is enabled in the MSAL config.

Expand table

Method	Supported by NAA
`acquireTokenByCode`	No (throws exception)
`acquireTokenPopup`	Yes
`acquireTokenRedirect`	No (throws exception)
`acquireTokenSilent`	Yes
`addEventCallback`	Yes
`addPerformanceCallback`	No (throws exception)
`disableAccountStorageEvents`	No (throws exception)
`enableAccountStorageEvents`	No (throws exception)
`getAccountByHomeId`	Yes
`getAccountByLocalId`	Yes
`getAccountByUsername`	Yes
`getActiveAccount`	Yes

Method	Supported by NAA
`getAllAccounts`	Yes
`getConfiguration`	Yes
`getLogger`	Yes
`getTokenCache`	No (throws exception)
`handleRedirectPromise`	No
`initialize`	Yes
`initializeWrapperLibrary`	Yes
`loginPopup`	Yes
`loginRedirect`	No (throws exception)
`logout`	No (throws exception)
`logoutPopup`	No (throws exception)
`logoutRedirect`	No (throws exception)
`removeEventCallback`	Yes
`removePerformanceCallback`	No (throws exception)
`setActiveAccount`	No
`setLogger`	Yes
`ssoSilent`	Yes

Security reporting

If you find a security issue with our libraries or services, report the issue to secure@microsoft.com with as much detail as you can provide. Your submission may be eligible for a bounty through the Microsoft Bounty program. Don't post security issues to GitHub or any other public site. We'll contact you shortly after receiving your issue report. We encourage you to get new security incident notifications by visiting Microsoft technical security notifications to subscribe to Security Advisory Alerts.

Code samples

Expand table

Sample name	Description
Office Add-in with SSO using nested app authentication	Shows how to use NAA in an Office Add-in to access Microsoft Graph APIs for the signed-in user.
Outlook add-in with SSO using nested app authentication	Shows how to use NAA in an Outlook Add-in to access Microsoft Graph APIs for the signed-in user.
Implement SSO in events in an Outlook add-in using nested app authentication	Shows how to use NAA and SSO in Outlook add-in events.
Send identity claims to resources using nested app authentication (NAA) and SSO	Shows how to send the signed-in user's identity claims (such as name, email, or a unique ID) to a resource such as a database. This sample replaces an obsolete pattern for legacy Exchange Online tokens.
Outlook add-in with SSO using nested app authentication including Internet Explorer fallback	Shows how to implement a fallback authentication strategy when NAA isn't available and the add-in needs to support Outlook versions that still use Internet Explorer 11.

See also

- NAA FAQ
- Nested app authentication in Microsoft Teams.

Authenticate and authorize with the Office dialog API

Article • 12/01/2023

Always use the Office dialog API to authenticate and authorize users with your Office Add-in. You must also use the Office dialog API if you're implementing fallback authentication when single sign-on (SSO) can't be used.

Office Add-ins run in an iframe when opened in Office on the web. Many identity authorities, also called Secure Token Services (STS), prevent their sign-in page from opening in an iframe. These include Google, Facebook, and services protected by the Microsoft identity platform (formerly Azure AD V 2.0) such as a Microsoft account, a Microsoft 365 Education or work account, or other common account. Also security features implemented in the webview when Office Add-ins run in Office on Windows, or Office on Mac can prevent sign-in pages from working correctly.

For authorization to work correctly, the sign-in page must open in a separate browser or webview control instance. This is why Office provides the Office dialog API, specifically the displayDialogAsync method.

> **Note**
>
> - This article assumes that you're familiar with **Use the Office dialog API in your Office Add-ins**.
> - For brevity hereafter, this article uses "browser instance" to mean "browser or webview instance".

The dialog opened with this API has the following characteristics.

- It is nonmodal.
- It is a completely separate browser instance from the task pane, meaning:
  - It has its own runtime environment and window object and global variables.
  - There is no shared execution environment with the task pane.
  - It doesn't share the same session storage (the Window.sessionStorage property) as the task pane.
- The first page opened in the dialog must be hosted in the same domain as the task pane, including protocol, subdomains, and port, if any.
- The dialog can send information back to the task pane by using the messageParent method. We recommend that this method be called only from a

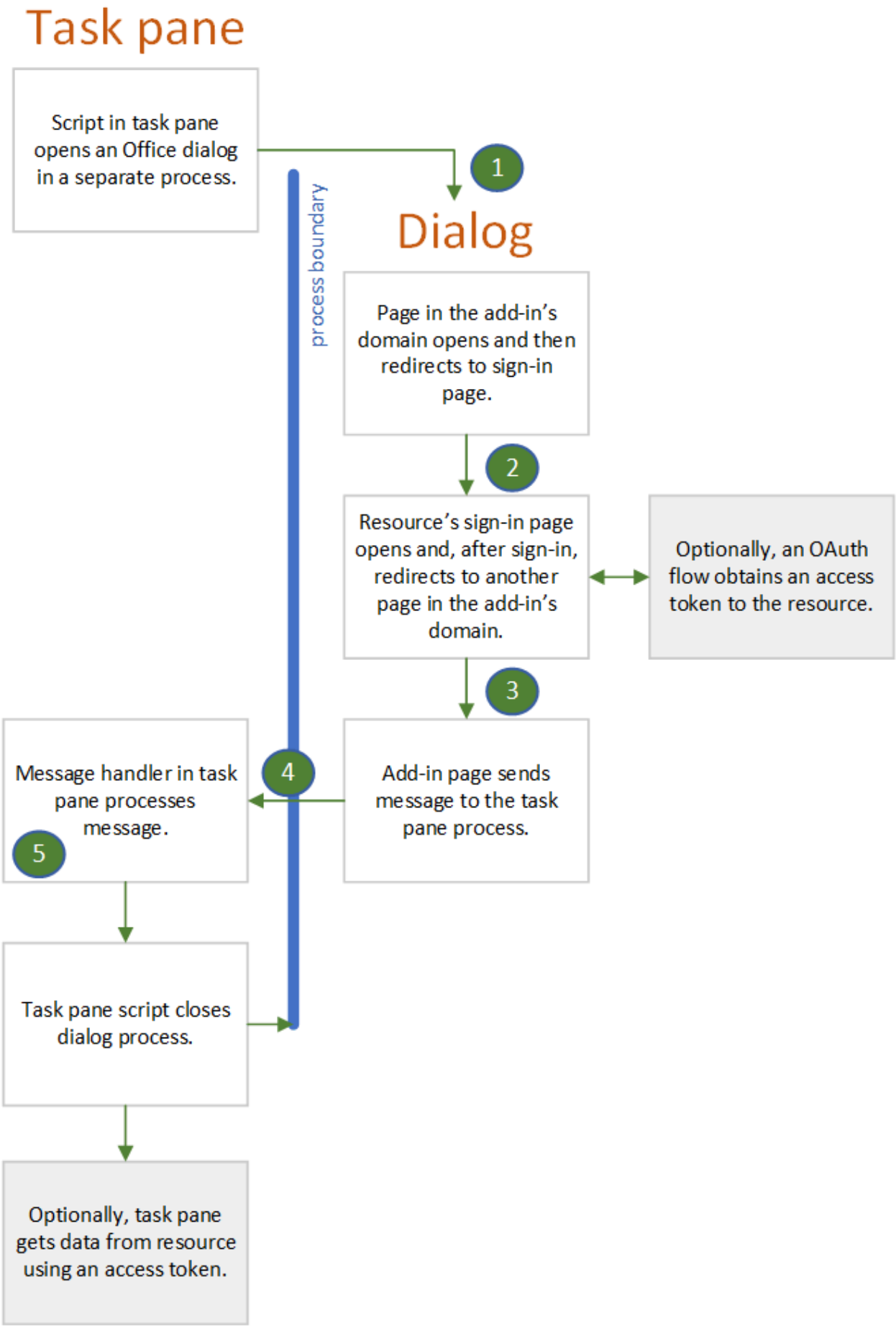
page that is hosted in the same domain as the task pane, including protocol, subdomains, and port. Otherwise, there are complications in how you call the method and process the message. For more information, see Cross-domain messaging to the host runtime.

By default, the dialog opens in a new web view control, not in an iframe. This ensures that it can open the sign-in page of an identity provider. As you'll see later in this article, the characteristics of the Office dialog have implications for how you use authentication or authorization libraries such as Microsoft Authentication Library (MSAL) and Passport.

> **Note**
>
> To configure the dialog to open in a floating iframe, pass the `displayInIframe: true` option in the call to `displayDialogAsync`. Do *not* do this when you're using the Office dialog API for sign in.

Authentication flow with the Office dialog

The following is a typical authentication flow.



1. The first page that opens in the dialog is a page (or other resource) that is hosted in the add-in's domain; that is, the same domain as the task pane window. This page can have a UI that only says "Please wait, we are redirecting you to the page

where you can sign in to *NAME-OF-PROVIDER*." The code in this page constructs the URL of the identity provider's sign-in page with information that is either passed to the dialog as described in Pass information to the dialog or is hard-coded into a configuration file of the add-in, such as a web.config file.

2. The dialog window then redirects to the sign-in page. The URL includes a query parameter that tells the identity provider to redirect the dialog window to a specific page after the user signs in. We'll call this page **redirectPage.html** in this article. The results of the sign-in attempt can be passed to the task pane with a call of `messageParent` on this page. *We recommend that this be a page in the same domain as the host window*.
3. The identity provider's service processes the incoming GET request from the dialog window. If the user is already signed in, it immediately redirects the window to **redirectPage.html** and includes user data as a query parameter. If the user isn't already signed in, the provider's sign-in page appears in the window, and the user signs in. For most providers, if the user can't sign in successfully, the provider shows an error page in the dialog window and does not redirect to **redirectPage.html**. The user must close the window by selecting the **X** in the corner. If the user successfully signs in, the dialog window is redirected to **redirectPage.html** and user data is included as a query parameter.
4. When the **redirectPage.html** page opens, it calls `messageParent` to report the success or failure to the task pane page and optionally also report user data or error data. Other possible messages include passing an access token or telling the task pane that the token is in storage.
5. The `DialogMessageReceived` event fires in the task pane page and its handler closes the dialog window and may further process of the message.

Support multiple identity providers

If your add-in gives the user a choice of providers, such as a Microsoft account, Google, or Facebook, you need a local first page (see preceding section) that provides a UI for the user to select a provider. Selection triggers the construction of the sign-in URL and redirection to it.

Authorization of the add-in to an external resource

In the modern web, users and web applications are security principals. The application has its own identity and permissions to an online resource such as Microsoft 365, Google Plus, Facebook, or LinkedIn. The application is registered with the resource provider before it is deployed. The registration includes:

- A list of the permissions that the application needs.
- A URL to which the resource service should return an access token when the application accesses the service.

When a user invokes a function in the application that accesses the user's data in the resource service, they are prompted to sign in to the service and then prompted to grant the application the permissions it needs to the user's resources. The service then redirects the sign-in window to the previously registered URL and passes the access token. The application uses the access token to access the user's resources.

You can use the Office dialog API to manage this process by using a flow that is similar to the one described for users to sign in. The only differences are:

- If the user hasn't previously granted the application the permissions it needs, the user is prompted to do so in the dialog after signing in.
- Your code in the dialog window sends the access token to the host window either by using `messageParent` to send the stringified access token or by storing the access token where the host window can retrieve it (and using `messageParent` to tell the host window that the token is available). The token has a time limit, but while it lasts, the host window can use it to directly access the user's resources without any further prompting.

Some authentication sample add-ins that use the Office dialog API for this purpose are listed in Samples.

Use authentication libraries with the dialog

Because the Office dialog and the task pane run in different browser runtime instances, you must use authentication/authorization libraries differently from how they are used when authentication and authorization take place in the same window. The following sections describe the ways that you can and can't use these libraries.

You usually can't use the library's internal cache to store tokens

Typically, auth-related libraries provide an in-memory cache to store the access token. If subsequent calls to the resource provider (such as Google, Microsoft Graph, Facebook, etc.) are made, the library will first check to see if the token in its cache is expired. If it is unexpired, the library returns the cached token rather than making another round-trip to the STS for a new token. But this pattern isn't usable in Office Add-ins. Since the sign-

in process occurs in the Office dialog's browser instance, the token cache is in that instance.

Closely related to this is the fact that a library will typically provide both interactive and "silent" methods for getting a token. When you can do both the authentication and the data calls to the resource in the same browser instance, your code calls the silent method to obtain a token just before your code adds the token to the data call. The silent method checks for an unexpired token in the cache and returns it, if there is one. Otherwise, the silent method calls the interactive method which redirects to the STS's sign-in. After sign-in completes, the interactive method returns the token, but also caches it in memory. But when the Office dialog API is being used, the data calls to the resource, which would call the silent method, are in the task pane's browser instance. The library's token cache does not exist in that instance.

As an alternative, your add-in's dialog browser instance can directly call the library's interactive method. When that method returns a token, your code must explicitly store the token someplace where the task pane's browser instance can retrieve it, such as local storage or a server-side database.

> ⓘ **Note**
>
> Changes to browser security will affect your strategy for token handling.
>
> - If your add-in runs in **Office on the web** in the Microsoft Edge Legacy (non-Chromium) or Safari browser, the dialog and task pane don't share the same **local storage**, so it can't be used to communicate between them.
> - Starting in Version 115 of Chromium-based browsers, such as Chrome and Edge, **storage partitioning** is being tested to prevent specific side-channel cross-site tracking (see also **Microsoft Edge browser policies**). This means that data stored by storage APIs, such as local storage, are only available to contexts with the same origin and the same top-level site. To work around this, in your browser, go to **chrome://flags** or **edge://flags**, then set the **Experimental third-party storage partitioning (#third-party-storage-partitioning)** flag to **Disabled**. Where possible, we recommend to pass data between the dialog and task pane using the **messageParent** and **messageChild** methods as described in **Use the Office dialog API in your Office Add-ins**.

Another option is to pass the token to the task pane with the `messageParent` method. This alternative is only possible if the interactive method stores the access token in a

place where your code can read it. Sometimes a library's interactive method is designed to store the token in a private property of an object that is inaccessible to your code.

You usually can't use the library's "auth context" object

Often, an auth-related library has a method that both obtains a token interactively and also creates an "auth-context" object which the method returns. The token is a property of the object (possibly private and inaccessible directly from your code). That object has the methods that get data from the resource. These methods include the token in the HTTP Requests that they make to the resource provider (such as Google, Microsoft Graph, Facebook, etc.).

These auth-context objects, and the methods that create them, are not usable in Office Add-ins. Since the sign-in occurs in the Office dialog's browser instance, the object would have to be created there. But the data calls to the resource are in the task pane browser instance and there is no way to get the object from one instance to another. For example, you can't pass the object with `messageParent` because `messageParent` can only pass string values. A JavaScript object with methods can't be reliably stringified.

How you can use libraries with the Office dialog API

In addition to, or instead of, monolithic "auth context" objects, most libraries provide APIs at a lower level of abstraction that enable your code to create less monolithic helper objects. For example, MSAL.NET v. 3.x.x has an API to construct a sign-in URL, and another API that constructs an AuthResult object that contains an access token in a property that is accessible to your code. For examples of MSAL.NET in an Office Add-in see: Office Add-in Microsoft Graph ASP.NET and Outlook Add-in Microsoft Graph ASP.NET. For an example of using msal.js in an add-in, see Office Add-in Microsoft Graph React.

For more information about authentication and authorization libraries, see Microsoft Graph: Recommended libraries and Other external services: Libraries.

Samples

- Office Add-in Microsoft Graph ASP.NET: An ASP.NET based add-in (Excel, Word, or PowerPoint) that uses the MSAL.NET library and the Authorization Code Flow to sign in and get an access token for Microsoft Graph data.
- Outlook Add-in Microsoft Graph ASP.NET: Just like the one above, but the Office application is Outlook.

- Office Add-in Microsoft Graph React : A NodeJS based add-in (Excel, Word, or PowerPoint) that uses the msal.js library and the Implicit Flow to sign in and get an access token for Microsoft Graph data.

See also

- Authorize external services in your Office Add-in
- Use the Office dialog API in your Office Add-ins