

# MS-600: Building Applications and Solutions with Microsoft 365 Core Services

- Download Latest Student Handbook and AllFiles Content  
`/home/l1/Azure_clone/Azure_new/MS-600-Building-Applications-and-Solutions-with-Microsoft-365-Core-Services/../../releases/latest`
- Are you a MCT? - Have a look at our [GitHub User Guide for MCTs](#)
- Need to manually build the lab instructions? - Instructions are available in the [MicrosoftLearning/Docker-Build](#) repository

## **What are we doing?**

- To support this course, we will need to make frequent updates to the course content to keep it current with the Microsoft 365 services used in the course. We are publishing the lab instructions and lab files on GitHub to allow for open contributions between the course authors and MCTs to keep the content current with changes in the Microsoft 365 platform.
- We hope that this brings a sense of collaboration to the labs like we've never had before - when Microsoft 365 services changes and you find it first during a live delivery, go ahead and make an enhancement right in the lab source. Help your fellow MCTs.

## **How should I use these files relative to the released MOC files?**

- The instructor handbook and PowerPoints are still going to be your primary source for teaching the course content.
- These files on GitHub are designed to be used in conjunction with the student handbook, but are in GitHub as a central repository so MCTs and course authors can have a shared source for the latest lab files.
- It will be recommended that for every delivery, trainers check GitHub for any changes that may have been made to support the latest Microsoft 365 services, and get the latest files for their delivery.

## **What about changes to the student handbook?**

- We will review the student handbook on a quarterly basis and update through the normal MOC release channels as needed.

## **How do I contribute?**

- Any MCT can submit a pull request to the code or content in the GitHub repro, Microsoft and the course author will triage and include content and lab code changes as needed.
- You can submit bugs, changes, improvement and ideas. Find a new Microsoft 365 feature before we have? Submit a new demo!

# **Notes**

## **Classroom Materials**

**It is strongly recommended that MCTs and Partners access these materials and in turn, provide them separately to students. Pointing students directly to GitHub to access Lab steps as part of an ongoing class will require them to access yet another UI as part of the course, contributing to a confusing experience for the student. An explanation to the student regarding why they are receiving separate Lab instructions can highlight the nature of an always-changing cloud-based interface and platform. Microsoft Learning support for accessing files on GitHub and support for navigation of the GitHub site is limited to MCTs teaching this course only.**

title: Online Hosted Instructions permalink: index.html layout: home

---

# **Content Directory**

Hyperlinks to each of the lab exercises and demos are listed below.

## Labs

```
{% assign labs = site.pages | where_exp:"page", "page.url contains '/Instructions/Labs'" %} |  
Module | Lab | | --- | --- | {%- for activity in labs %}{% activity.lab.module %} | {{  
activity.lab.title }}.{% if activity.lab.type %}-{{ activity.lab.type }}.{% endif %},| {%- endfor  
%}
```

## Demos

```
{% assign demos = site.pages | where_exp:"page", "page.url contains '/Instructions/Demos'" %} | Module | Demo || --- | --- | { % for activity in demos %}| {{ activity.demo.module }} | {{ activity.demo.title }}. | { % endfor %}
```

---

demo: title: 'Demo: Deploying an ARM Template' module: 'Module 1: Exploring Azure Resource Manager'

---

# Demo: Deploying an ARM Template

## Instructions

1. Quisque dictum convallis metus, vitae vestibulum turpis dapibus non.
  1. Suspendisse commodo tempor convallis.
  2. Nunc eget quam facilisis, imperdiet felis ut, blandit nibh.
  3. Phasellus pulvinar ornare sem, ut imperdiet justo volutpat et.
2. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos.
3. Vestibulum hendrerit orci urna, non aliquet eros eleifend vitae.
4. Curabitur nibh dui, vestibulum cursus neque commodo, aliquet accumsan risus.

Sed at malesuada orci, eu volutpat ex

5. In ac odio vulputate, faucibus lorem at, sagittis felis.
6. Fusce tincidunt sapien nec dolor congue facilisis lacinia quis urna.

**Note:** Ut feugiat est id ultrices gravida.

7. Phasellus urna lacus, luctus at suscipit vitae, maximus ac nisl.
  - Morbi in tortor finibus, tempus dolor a, cursus lorem.
  - Maecenas id risus pharetra, viverra elit quis, lacinia odio.
  - Etiam rutrum pretium enim.
8. Curabitur in pretium urna, nec ullamcorper diam. ♦♦♦# Student lab answer key

## Microsoft Azure user interface

Given the dynamic nature of Microsoft cloud tools, you might experience Azure user interface (UI) changes after the development of this training content. These changes might cause the lab instructions and lab steps to not match up.

The Microsoft Worldwide Learning team updates this training course as soon as the community brings needed changes to our attention. However, because cloud updates occur frequently, you might encounter UI changes before this training content is updated. **If this occurs, adapt to the changes and work through them in the labs as needed.**

# Instructions

**Note:** Lab virtual machine sign in instructions will be provided to you by your instructor.

## Review installed applications

- Observe the taskbar located at the bottom of your **Windows 10** desktop. The taskbar contains the icons for the applications you will use in this lab:
  - Microsoft Edge
  - File Explorer
  - Visual Studio Code
  - Microsoft Visual Studio 2019
  - Windows PowerShell

❖❖❖# Exercise 1: Registering an application in Azure Active Directory

## Task 1: Open the Azure portal

1. On the taskbar, select the **Microsoft Edge** icon.
2. In the open browser window, navigate to the **Azure portal ([portal.azure.com](https://portal.azure.com))**.
3. Enter the **email address** for your Microsoft account.
4. Select **Next**.
5. Enter the **password** for your Microsoft account.
6. Select **Sign in**.

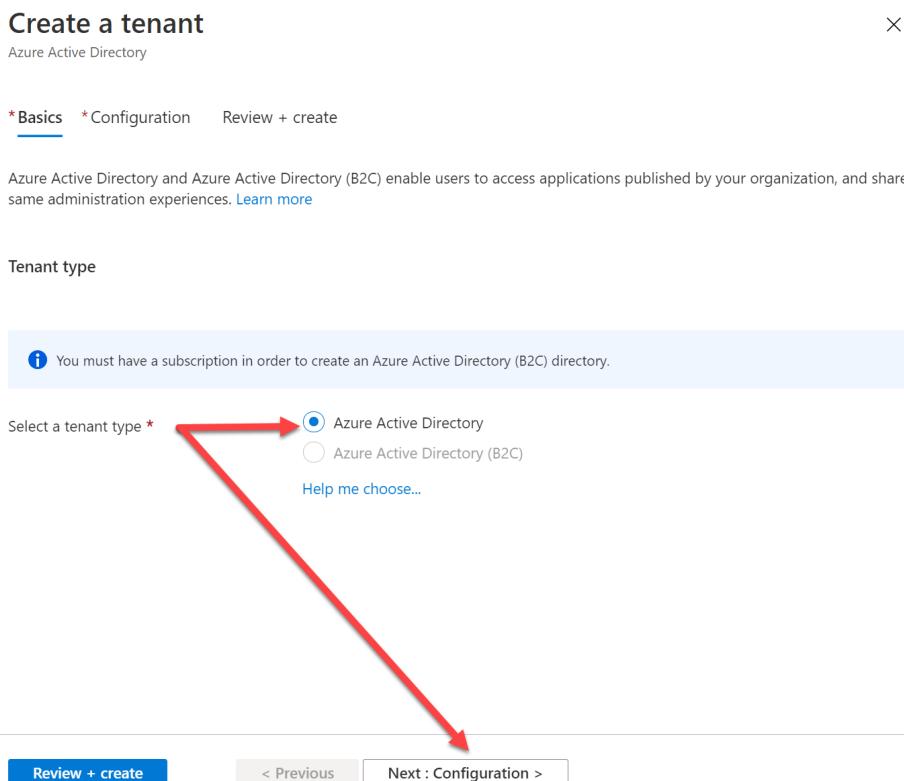
**Note:** If this is your first time signing in to the Azure Portal, a dialog box will appear offering a tour of the portal. Select Get Started to begin using the portal.

## Task 2: Create an Active Directory tenant

In this task, you will create a new Active Directory organization so you will understand the steps involved. Even though you will be creating a new Azure Active Directory organization, the remainder of the tasks will have you continue to work in the existing organization that was provided with your Azure portal.

### Create an Azure AD organization

1. Expand the left navigation pane, select **Azure Active Directory**.
2. Select **Create a tenant** located in the right side of the portal.
3. From the **Basics** tab, perform the following actions:
  1. In the **Select a tenant type** section, select **Azure Active Directory**.
  2. Select **Next: Configuration**.



4. From the **Configuration** tab, perform the following actions:
  1. **Organization name:** *Contoso Marketing Company*
  2. **Initial domain name:** *contosomarketingXXXX* where you replace XXXX with numbers or letters to make your domain name unique.
  3. Select the **Country or region**.

## Create a tenant

Azure Active Directory

X

\* Basics \* Configuration

Review + create

### Directory details

Configure your new directory

Organization name \* ⓘ

Contoso Marketing Company



Initial domain name \* ⓘ

contosomarketingms001



contosomarketingms001.onmicrosoft.com

Country/Region ⓘ

United States



Datacenter location - United States

Datacenter location is based on the country/region selected above.

Review + create

< Previous

Next : Review + create >

### 4. Select **Review + create**.

### 5. Ensure settings are correct and validation passed.

## Create a tenant

Azure Active Directory

X

Validation passed.

\* Basics \* Configuration

Review + create

### Summary

#### Basics

Tenant type

Azure Active Directory

#### Configuration

Organization name

Contoso Marketing Company

Initial domain name

contosomarketingms001.onmicrosoft.com

Country/Region

United States

Datacenter location

United States

Create

< Previous

Next >

### 6. Select **Create**.

7. A **Directory creation in progress, this will take a few minutes** message is displayed.

1. When directory creation is complete, select **Click here to navigate to your new directory**.

The screenshot shows the Azure Active Directory Overview page for the tenant "Contoso Marketing Company". The left sidebar contains navigation links for Overview, Getting started, Preview hub, Diagnose and solve problems, Manage (Users, Groups, External Identities, Roles and administrators, Administrative units, Enterprise applications, Devices, App registrations, Identity Governance, Application proxy, Licenses), and other features like Azure AD Connect, Custom domain names, and Mobility (MDM and MAM). The main content area displays "Tenant information" (Your role: Global administrator, License: Azure AD Free, Tenant ID: 111086d9-f820-4c8f-b636-8141..., Primary domain: i.onmicrosoft.com) and "Azure AD Connect" status (Status: Not enabled, Last sync: Sync has never run). Below these sections is a "Sign-ins" chart showing activity from October 4 to November, with a single bar labeled "Sign ins".

## Task 3: Create a user in new Active Directory tenant

In this task, you will create a user in the new Active Directory tenant you just created in the previous steps. This user account will be used later in an exercise used for adding guest users to a single organization. 1. From the Active Directory page (Contoso Marketing Company), select **Users**.

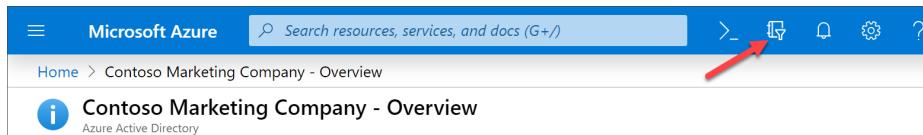
1. Select **New user**.
2. Ensure **Create user** is selected.
3. On the **New user** page, enter information for this user:
  - **User name:** testuser
  - **Name:** Test User
  - **First Name:** Test
  - **Last Name:** User
  - **Password:** Auto-generate passwordSelect **Show password**. Copy the autogenerated password provided in the Password box. You'll need to give this password to the user to sign in for the first time.
4. Select **Create**.
5. Launch a **New InPrivate Window** browser session.
6. Test new user in Azure portal and change password:
  1. Navigate to the Azure portal, <https://portal.azure.com> in the InPrivate browser session.
  2. Sign in with your new *testuser@contosomarketingxx.onmicrosoft.com* account.
  3. You will be prompted to change the password. Paste the auto-generated password you copied and paste into the **Current password** field.
  4. Set the password as desired.
  5. Save the username and password so you will remember the credentials for using it later in this lab.

## Task 4: Register an application

In this task, you will switch Active Directory tenants and then continue with registering an application into the default

### Switch Azure AD Directory

1. Select the Directory + subscription icon.

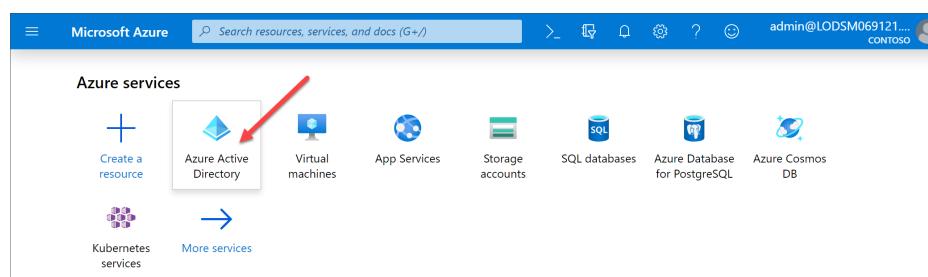


2. Select Contoso.

The screenshot shows the Azure portal interface for the 'Contoso Marketing Company - Overview' page. On the left, there's a navigation pane with options like 'Overview', 'Getting started', and 'Diagnose and solve problems'. Below that is a 'Manage' section with links for 'Users', 'Groups', 'Organizational relationships', 'Roles and administrators', 'Enterprise applications', 'Devices', 'App registrations', 'Identity Governance', and 'Application proxy'. The main area displays 'Sign-ins' data from November 3 to November 24, with a count of 100. A 'What's new in Azure AD' section at the bottom encourages users to stay up-to-date with release notes and blog posts. On the right, a 'Directory + subscription' blade is open, showing a 'Default subscription filter' message and a 'Current directory' set to 'contosomarketingms01.onmicrosoft.com'. It also lists 'Favorites' and 'All Directories' with a search bar. A red arrow points to the 'Contoso' entry in the 'All Directories' list.

3. You should be redirected to the Active Directory page for Contoso. If it redirected you to the default Azure portal landing page, then follow the steps below to navigate back to Azure Active Directory:

1. You will be redirected to the Azure portal landing page where you should now see an Azure Active Directory button available.
2. Select the **Azure Active Directory** icon. If you do not have the icon: expand the left navigation pane, select **Azure Active Directory**.



### Register a new application

1. On the left menu, navigate to **App registrations**.
2. Select **New Registration**.
3. From the **Register an application** pane, perform the following actions:
  1. In the **Name** text box, enter the value **ContosoApp**.
  2. In the **Supported account type** section, select **Accounts in this organizational directory only**.
  3. Leave the **Redirect URI (optional)** as blank.

## Register an application

\* Name

The user-facing display name for this application (this can be changed later).

### Supported account types

Who can use this application or access this API?

- Accounts in this organizational directory only (Contoso only - Single tenant)  
 Accounts in any organizational directory (Any Azure AD directory - Multitenant)  
 Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)  
 Personal Microsoft accounts only

[Help me choose...](#)

### Redirect URI (optional)

We'll return the authentication response to this URI after successfully authenticating the user. Providing this now is optional and it can be changed later, but a value is required for most authentication scenarios.

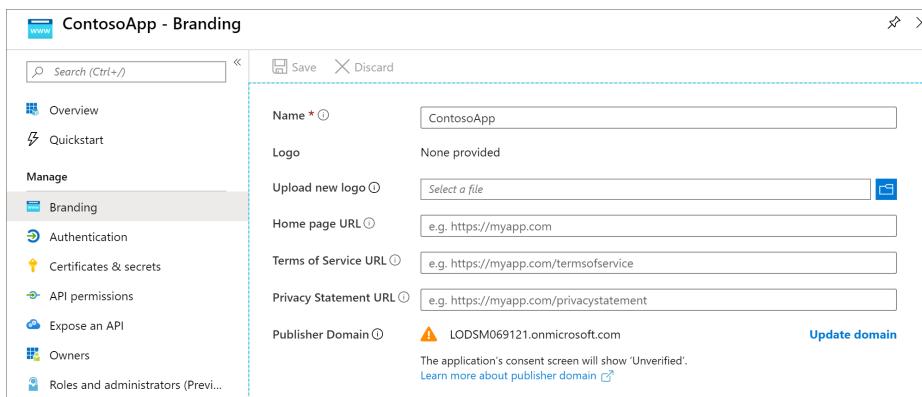
[By proceeding, you agree to the Microsoft Platform Policies](#) 

**Register**

4. Select **Register**. The application overview page is displayed.

## Task 5: Branding of the application

1. On the left menu, navigate to **Branding**.
2. Verify that you can personalize branding by providing a logo, home page URL, terms of service URL, and a privacy statement URL.



3. This exercise is now complete.

## Review

In this exercise, you learned how to create a new Active Directory organization, create a new user, switch directories, register and brand the application.

### ❖❖❖# Exercise 2: Implementing authentication

This exercise will demonstrate the different account types that are used within the Microsoft identity platform.

**Note:** This exercise demonstrates signing into a web application using two different accounts. These two accounts will come from two organizations, one of them being the organization where the Azure AD application is registered. Therefore, in order to complete the exercise, you'll need access to two user accounts in different Azure AD directories.

# Task 1: Create application that only allows single organization sign in

In this task, you will register an application in the Azure portal that allows users from the current organization to sign in.

## Register a single-tenant Azure AD application

1. From the Azure portal <https://portal.azure.com>, navigate to **Azure Active Directory**.
2. Select **Manage > App registrations** in the left-hand navigation.
3. On the **App registrations** page, select **New registration**.
4. On the **Register an application** page, set the values as follows:
  - **Name:** Hello ASPNET Core Identity 01
  - **Supported account types:** Accounts in this organizational directory only (Single tenant)

### Register an application

\* Name  
The user-facing display name for this application (this can be changed later).

Hello ASPNET Core Identity 01 ✓

Supported account types

Who can use this application or access this API?

Accounts in this organizational directory only (Contoso only - Single tenant)  
 Accounts in any organizational directory (Any Azure AD directory - Multitenant)  
 Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)  
 Personal Microsoft accounts only

[Help me choose...](#)

Redirect URI (optional)  
We'll return the authentication response to this URI after successfully authenticating the user. Providing this now is optional and it can be changed later, but a value is required for most authentication scenarios.

Web e.g. https://myapp.com/auth

By proceeding, you agree to the Microsoft Platform Policies [↗](#)

**Register**

5. Select **Register** to create the application.
6. On the **Hello ASPNET Core Identity 01** page, copy the values **Application (client) ID** and **Directory (tenant) ID**; you'll need these values later in this exercise.

The screenshot shows the Azure Active Directory admin center interface. On the left, there's a navigation sidebar with options like Overview, Quickstart, Manage, API permissions, Expose an API, Owners, Roles and administrators, Manifest, Support + Troubleshooting, Troubleshooting, and New support request. The main content area is titled "Hello ASP.NET Core Identity 01". It displays the application's details: Display name "Hello ASP.NET Core Identity 01", Application (client) ID "7ae7dadd-3744-4e57-959a-4e16fdf4d4be", Directory (tenant) ID "7ade07e0-76e4-4ef6-91d2-2aa01ba662a7", Object ID "8eb03e32-c65c-4fbf-8c87-3bc3f1d41407", Supported account types "My organization only", Redirect URLs, Application ID URI, and Managed application in local directory "Hello ASP.NET Core Identity 01". A red box highlights the Application (client) ID and Directory (tenant) ID fields.

7. On the **Hello ASPNET Core Identity 01** page, select the **Add a Redirect URI** link under the **Redirect URIs**.
8. Locate the section **Platform configurations** and select **+Add a platform**.
9. Select **Web** and add the following URLs:
  - **https://localhost:3007**
10. Select **Configure**
11. Locate the section **Redirect URIs** under **Web**, select **Add URI** and add the following URL:
  - **https://localhost:3007/signin-oidc**
12. Locate the section **Logout URL** and add the following **Logout URL**:  
**https://localhost:3007/signout-oidc**
13. Locate the section **Implicit grant** and select both **Access tokens** and **ID tokens**. This tells Azure AD to return these tokens the authenticated user if requested.
14. Select **Save** when finished setting these values.

## Hello ASP.NET Core Identity 01 | Authentication

Search (Ctrl+ /) Save Discard Got feedback?

Overview Quickstart Integration assistant | Preview

Manage

- Branding
- Authentication **Selected**
- Certificates & secrets
- Token configuration
- API permissions
- Expose an API
- Owners
- Roles and administrators | Preview
- Manifest

Support + Troubleshooting

- Troubleshooting
- New support request

**Platform configurations**

Depending on the platform or device this application is targeting, additional configuration may be required such as redirect URIs, specific authentication settings, or fields specific to the platform.

+ Add a platform

**Web** Quickstart Docs

**Redirect URLs**

The URLs we will accept as destinations when returning authentication responses (tokens) after successfully authenticating users. Also referred to as reply URLs. [Learn more about Redirect URLs and their restrictions](#)

`https://localhost:3007/signin-oidc`

`https://localhost:3007`

[Add URI](#)

**Logout URL**

This is where we send a request to have the application clear the user's session data. This is required for single sign-out to work correctly.

`https://localhost:3007/signout-oidc`

**Implicit grant**

Allows an application to request a token directly from the authorization endpoint. Checking Access tokens and ID tokens is recommended only if the application has a single-page architecture (SPA), has no back-end components, does not use the latest version of MSAL.js with auth code flow, or it invokes a web API via JavaScript. ID Token is needed for ASP.NET Core Web Apps. [Learn more about the implicit grant flow](#)

To enable the implicit grant flow, select the tokens you would like to be issued by the authorization endpoint:

Access tokens  ID tokens

## Task 2: Create a single organization ASP.NET core web application

In this first application, you'll create an ASP.NET Core web application that allows users from the current organization to sign in and display their information. 1. Open your command prompt, navigate to a directory where you want to save your work, create a new folder, and change directory into that folder. For example:

```
```powershell
Cd c:/LabFiles
md SingleOrg
cd SingleOrg
````
```

1. Execute the following command to create a new ASP.NET Core MVC web application:

```
powershell dotnet new mvc --auth SingleOrg
```

2. Open the folder in Visual Studio code by executing from the project directory: `code .`

### Configure the web application with the Azure AD application you created

1. Locate and open the `./appsettings.json` file in the ASP.NET Core project.
2. Set the **AzureAd.Domain** property to the domain of your Azure AD tenant where you created the Azure AD application (*for example: contoso.onmicrosoft.com*).
3. Set the **AzureAd.TenantId** property to the **Directory (tenant) ID** you copied when creating the Azure AD application in the previous step.
4. Set the **AzureAd.ClientId** property to the **Application (client) ID** you copied when creating the Azure AD application in the previous step.

### Update the web application's launch configuration

1. Locate and open the `./Properties/launchSettings.json` file in the ASP.NET Core project.
2. Set the **iisSettings.iisExpress.applicationUrl** property to **https://localhost:3007**.
3. Set the **iisSettings.iisExpress.sslPort** property to **3007**.

### Update the user experience

1. Finally, update the user experience of the web application to display all the claims in the OpenID Connect ID token.
2. Locate and open the `./Views/Home/Index.cshtml` file.
3. Add the following code to the end of the file:

```
powershell @if (User.Identity.IsAuthenticated) { <div> <table
cellpadding="2" cellspacing="2"> <tr> <th>Claim</th> <th>Value</th> </tr>
@foreach (var claim in User.Claims) { <tr> <td>@claim.Type</td>
<td>@claim.Value</td> </tr> } </table> </div> }
```

## Task 3: Build and test the single organization web app

1. Execute the following command in a command prompt to compile and run the application:

```
powershell dotnet build dotnet run
```

2. Open a browser and navigate to the url **https://localhost:5001**. The web application will redirect you to the Azure AD sign in page.

3. Sign in using a Work and School account from your Azure AD directory. Azure AD will redirect you back to the web application. Notice some of the details from the claims included in the ID token.

The screenshot shows the 'msidentity\_aspnet' web application's 'Welcome' page. At the top, there is a navigation bar with 'msidentity\_aspnet', 'Home', and 'Privacy' links, and a sign-in message 'Hello admin@M365x068225.onmicrosoft.com! Sign out'. Below the navigation bar is a 'Welcome' heading and a link to 'Learn about building Web apps with ASP.NET Core.' Underneath, there is a table titled 'Claims' showing the following data:

Claim	Value
aio	42VgYEil1RUsNu7XvPw3OaMl++6Wi2zbc9Vl9wqmadxfGKn8lUA
http://schemas.microsoft.com/claims/authnmethodsreferences	pwd
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/surname	Administrator
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname	MOD
name	MOD Administrator
http://schemas.microsoft.com/identity/claims/objectidentifier	b9dd14c5-1943-448d-a4bf-70bd0ccd2592
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier	vVecxWIDJLUz0tVpBSCZdCP-nkPkAdQKJfg5kahg
http://schemas.microsoft.com/identity/claims/tenantid	7a4e07e0-76e4-4ef6-91d2-2aa01ba662a7
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name	admin@M365x068225.onmicrosoft.com
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/upn	admin@M365x068225.onmicrosoft.com
uti	4lxz2p4z80imXz4F6MA0AQ

At the bottom of the page, there is a footer with the text '© 2019 - msidentity\_aspnet - [Privacy](#)'.

4. Take special note of the **tenantid** and **upn** claim. These claims indicate the ID of the Azure AD directory and ID of the user that signed in. Make a note of these values to compare them to other options in a minute.
5. Now try logging in as a user from a different organization. Select the **Sign out** link in the top left. Wait for Azure AD and the web application signs out the current user. When the web application reloads, repeat the sign in process, except this time try signing in as a user from a different organization or use a Microsoft Account.
6. Notice Azure AD will reject the user's sign in, explaining that the user's account doesn't exist in the current tenant.
7. Stop the web server by pressing **CTRL+C** in the command prompt.

## Task 4: Create application that allows any organization's users to sign in

In this task, you will register an application in the Azure portal that allows users from any organization or Microsoft Accounts to sign in.

### Register a multi-tenant Azure AD application

1. From the Azure portal <https://portal.azure.com>, navigate to **Azure Active Directory**.
2. Select **Manage > App registrations** in the left-hand navigation.
3. On the **App registrations** page, select **New registration**.
4. On the **Register an application** page, set the values as follows:
  - **Name:** Hello ASPNET Core Identity 02
  - **Supported account types:** Accounts in any organizational directory only (Any Azure AD directory - Multitenant)

#### Register an application

\* Name  
The user-facing display name for this application (this can be changed later).  
 ✓

Supported account types  
Who can use this application or access this API?  
 Accounts in this organizational directory only (Contoso only - Single tenant)  
 Accounts in any organizational directory (Any Azure AD directory - Multitenant)  
 Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)  
 Personal Microsoft accounts only  
[Help me choose...](#)

Redirect URI (optional)  
We'll return the authentication response to this URI after successfully authenticating the user. Providing this now is optional and it can be changed later, but a value is required for most authentication scenarios.

By proceeding, you agree to the Microsoft Platform Policies [\[?\]](#)

5. Select **Register** to create the application.
6. On the **Hello ASPNET Core Identity 02** page, copy the values **Application (client) ID** and **Directory (tenant) ID**; you'll need these values later in this exercise.
7. On the **Hello ASPNET Core Identity 02** page, select the **Add a Redirect URI** link under the **Redirect URIs**.
8. Locate the section **Redirect URIs** and add the following two URLs:

- <https://localhost:3007>
- <https://localhost:3007/signin-oidc>

9. Add the following **Logout URL**: <https://localhost:3007/signout-oidc>

10. Locate the section **Implicit grant** and select both **Access tokens** and **ID tokens**. This tells Azure AD to return these tokens the authenticated user if requested.

11. Select **Save** when finished setting these values.

The screenshot shows the Azure portal interface for managing an application named "Hello ASPNET Core Identity 02". The left sidebar lists various management sections like Overview, Quickstart, Integration assistant, Manage, Branding, Authentication, Certificates & secrets, Token configuration, API permissions, Expose an API, Owners, Roles and administrators, and Manifest. The Authentication section is currently selected. The main content area is titled "Platform configurations" and specifically targets "Web". It contains sections for "Redirect URLs" and "Logout URL". Under "Redirect URLs", there are two entries: "https://localhost:3007/signin-oidc" and "https://localhost:3007". Below these is a "Add URI" button. Under "Logout URL", the value "https://localhost:3007/signout-oidc" is entered. At the bottom, the "Implicit grant" section is expanded, showing options for issuing tokens. Both "Access tokens" and "ID tokens" are checked, indicating they will be issued by the authorization endpoint.

## Task 5: Create a multiple organization ASP.NET core web application

In this second application, you'll create an ASP.NET Core web application that allows users from any organization or Microsoft Accounts to sign in and display their information.

1. Open your command prompt, navigate to a directory where you want to save your work, create a new folder, and change directory into that folder. For example:

```
powershell Cd c:/LabFiles md MultiOrg cd MultiOrg
```

2. Execute the following command to create a new ASP.NET Core MVC web application:

```
powershell dotnet new mvc --auth MultiOrg
```

3. Open the folder in Visual Studio code by executing from the project directory: `code .`

### Configure the web application with the Azure AD application you created

1. Locate and open the **./appsettings.json** file in the ASP.NET Core project.
2. Set the **AzureAd.ClientId** property to the **Application (client) ID** you copied when creating the Azure AD application in the previous step.

### Update the web application's launch configuration

1. Locate and open the **./Properties/launchSettings.json** file in the ASP.NET Core project.
2. Set the **iisSettings.iisExpress.applicationUrl** property to **https://localhost:3007**.
3. Set the **iisSettings.iisExpress.sslPort** property to **3007**.

### Update the user experience

1. Finally, update the user experience of the web application to display all the claims in the OpenID Connect ID token.
2. Locate and open the **./Views/Home/Index.cshtml** file.
3. Add the following code to the end of the file:

```
powershell @if (User.Identity.IsAuthenticated) { <div> <table cellpadding="2" cellspacing="2"> <tr> <th>Claim</th> <th>Value</th> </tr> @foreach (var claim in User.Claims) { <tr> <td>@claim.Type</td> <td>@claim.Value</td> </tr> } </table> </div> }
```

4. Save the above all changes.

## Task 6: Build and test the multiple organization web app

1. Execute the following command in a command prompt to compile and run the application:

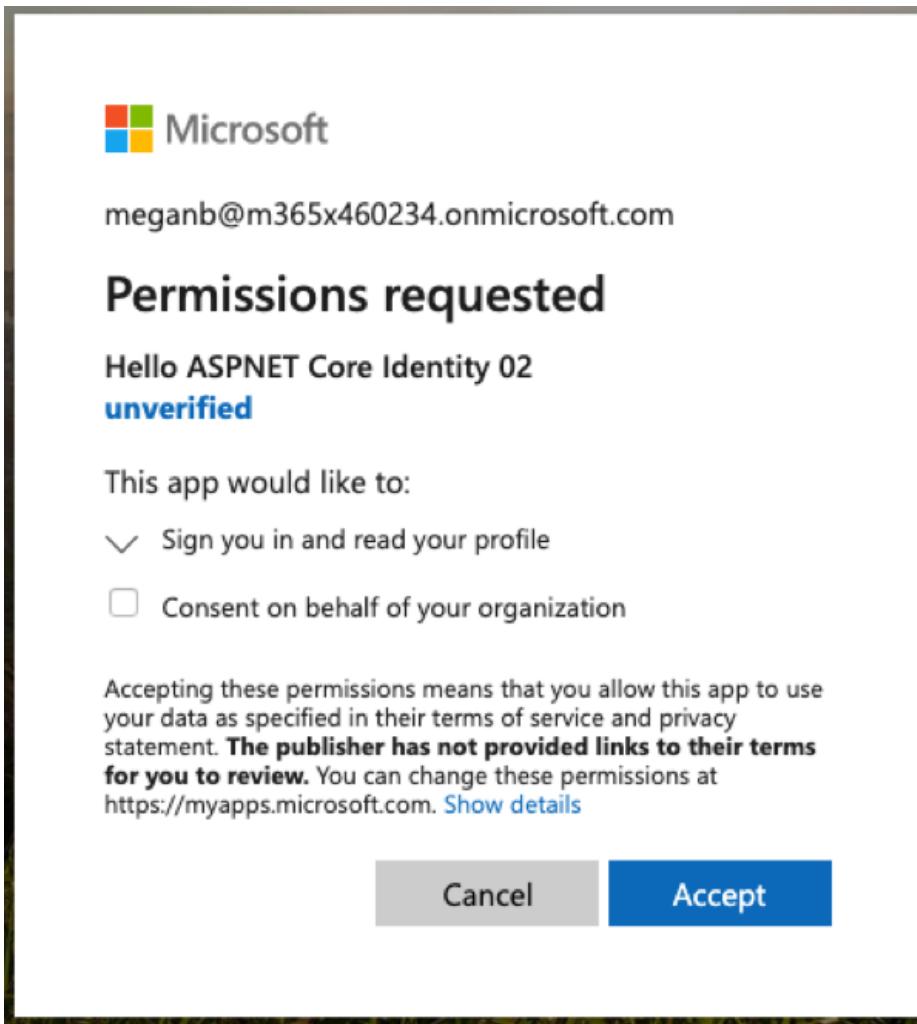
```
powershell dotnet build dotnet run
```

2. Open a browser and navigate to the url **https://localhost:5001**. The web application will redirect you to the Azure AD sign in page.

3. Sign in using a Work and School account from your Azure AD directory. Azure AD will redirect you back to the web application. Notice some of the details from the claims included in the ID token.

Claim	Value
aio	42VgYAjWVLryOnZ3uJm42RejefkH88Ua6qbKmd9KeGW09NCeL/YA
<a href="http://schemas.microsoft.com/claims/authnmethodsreference">http://schemas.microsoft.com/claims/authnmethodsreference</a>	pwd
<a href="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/surname">http://schemas.xmlsoap.org/ws/2005/05/identity/claims/surname</a>	Administrator
<a href="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname">http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname</a>	MOD
name	MOD Administrator
<a href="http://schemas.microsoft.com/identity/claims/objectidentifier">http://schemas.microsoft.com/identity/claims/objectidentifier</a>	b9dd14c5-1943-448d-a4bf-70bd0cccd2592
<a href="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier">http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier</a>	w6wbkCD8rgZXW7e56JP303z0FMhs1Ckf_rdjg3k3p_k
<a href="http://schemas.microsoft.com/identity/claims/tenantid">http://schemas.microsoft.com/identity/claims/tenantid</a>	7a4e07e0-76e4-4ef6-91d2-2aa01ba662a7
<a href="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name">http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name</a>	admin@M365x068225.onmicrosoft.com
<a href="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/upn">http://schemas.xmlsoap.org/ws/2005/05/identity/claims/upn</a>	admin@M365x068225.onmicrosoft.com
uti	NCupKOCExEWF2XarXO0YAQ

4. Take special note of the **tenantid** and **upn** claim. These indicate the ID of the Azure AD directory and ID of the user that signed in. Make a note of these values to compare them to other options in a minute.
5. Now try logging in as a user from a different organization. Select the **Sign out** link in the top left. Wait for Azure AD and the web application signs out the current user.
6. This time, the user is prompted to first trust the application:



7. Select **Accept**.

8. Notice the web application's page loads with different claims, specifically for the **tenantid** and **upn** claim. This indicates the user is not from the current directory where the Azure AD application is registered:

The screenshot shows a "Welcome" page for a web application. At the top right is the user information "Hello MeganB@M365x460234.OnMicrosoft.com! Sign out". Below it is a table of user claims:

Claim	Value
aio	ASQA2/8NAAAChNjQkG9ys/L0reUM5XBGGw7tfpe7n4xapXD3xNF7w=
http://schemas.microsoft.com/claims/authnmethodsreferences	pwd
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/surname	Bowen
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname	Megan
name	Megan Bowen
http://schemas.microsoft.com/identity/claims/objectidentifier	b8c8446f-f788-4527-8cf6-c40e65228539
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier	C2fyWGqB8gBmb0vJxSTCMaJORzef7fKW0hpBG52hDk
http://schemas.microsoft.com/identity/claims/tenantid	7adff0e7-07b3-4eb5-ba39-7ea421035371
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name	MeganB@M365x460234.OnMicrosoft.com
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/upn	MeganB@M365x460234.OnMicrosoft.com
uti	B1nQq0avIEaL40SbngADAA

9. Stop the web server by going back to the running project in Visual Studio Code. From the Terminal, press **CTRL+C** in the command prompt.

## Review

In this exercise, you learned how to create different types of Azure AD applications and use an ASP.NET Core application to support the different sign in options that support different types of accounts.

### ◆◆◆# Exercise 3: Implementing application that supports B2B

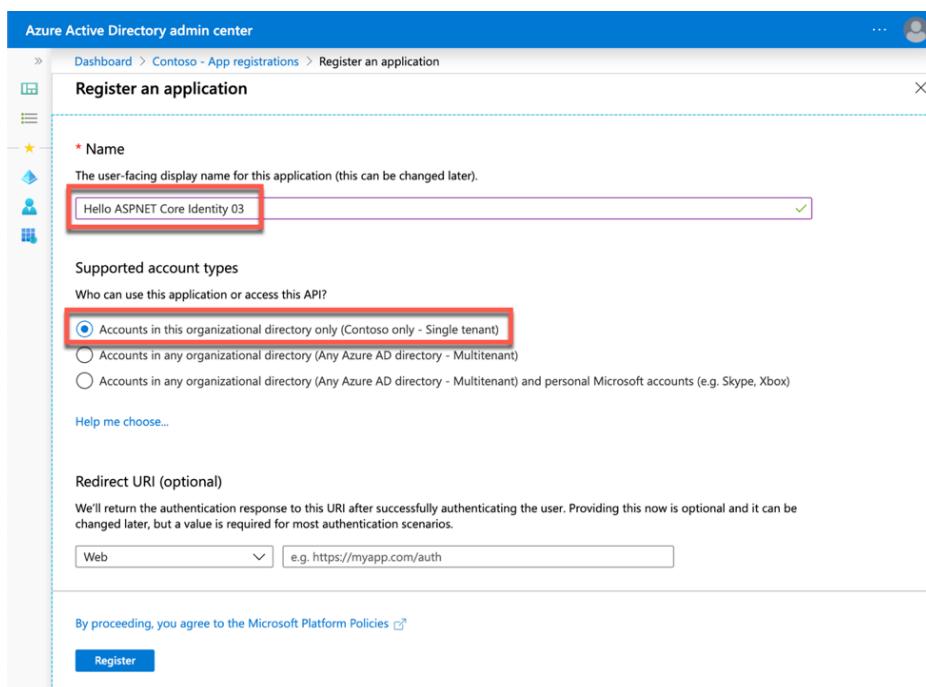
This exercise will demonstrate to the user how to configure and implement an application that supports B2B. **Note:** This exercise demonstrates signing into a web application using three different accounts. These three accounts will come from two organizations, one of them being the organization where the Azure AD application is registered. Therefore, in order to complete the exercise, you'll need access to two user accounts in different Azure AD directories.

# Task 1: Create a single-tenant Azure AD application

In this task, you'll create an Azure AD application that allows users from the current organization to sign in.

## Register a single-tenant Azure AD application

1. From the Azure portal <https://portal.azure.com>, navigate to **Azure Active Directory**.
2. Select **Manage > App registrations** in the left-hand navigation.
3. On the **App registrations** page, select **New registration**.
4. On the **Register an application** page, set the values as follows:
  - **Name:** Hello ASPNET Core Identity 03
  - **Supported account types:** Accounts in this organizational directory only (Single tenant)



5. Select **Register** to create the application.
6. On the **Hello ASPNET Core Identity 03** page, copy the values **Application (client) ID** and **Directory (tenant) ID**; you'll need these values later in this exercise.
7. On the **Hello ASPNET Core Identity 03** page, select the **Add a Redirect URI** link under the **Redirect URIs**.
8. Locate the section **Redirect URIs** and add the following two URLs:
  - **https://localhost:3007**

- <https://localhost:3007/signin-oidc>

9. Add the following **Logout URL:** <https://localhost:3007/signout-oidc>

10. Locate the section **Implicit grant** and select both **Access tokens** and **ID tokens**. This tells Azure AD to return these tokens the authenticated user if requested.

11. Select **Save** when finished setting these values.

## Task 2: Create a single organization ASP.NET core web application

In this application, you'll create an ASP.NET Core web application that allows users from the current organization to sign in and display their information.

1. Open your command prompt, navigate to a directory where you want to save your work, create a new folder, and change directory into that folder. For example:

```
powershell Cd c:/LabFiles md SingleOrgGuests cd SingleOrgGuests
```

2. Execute the following command to create a new ASP.NET Core MVC web application:

```
powershell dotnet new mvc --auth SingleOrg
```

3. Open the folder in Visual Studio code by executing from the project directory: **code .**

### Configure the web application with the Azure AD application you created

1. Locate and open the **./appsettings.json** file in the ASP.NET Core project.
2. Set the **AzureAd.Domain** property to the domain of your Azure AD tenant where you created the Azure AD application (*for example: contoso.onmicrosoft.com*).
3. Set the **AzureAd.TenantId** property to the **Directory (tenant) ID** you copied when creating the Azure AD application in the previous step.
4. Set the **AzureAd.ClientId** property to the **Application (client) ID** you copied when creating the Azure AD application in the previous step.

### Update the web application's launch configuration

1. Locate and open the **./Properties/launchSettings.json** file in the ASP.NET Core project.
2. Set the **iisSettings.iisExpress.applicationUrl** property to **https://localhost:3007**.
3. Set the **iisSettings.iisExpress.sslPort** property to **3007**.

### Update the user experience

1. Finally, update the user experience of the web application to display all the claims in the OpenID Connect ID token.
2. Locate and open the **./Views/Home/Index.cshtml** file.
3. Add the following code to the end of the file:

```
powershell @if (User.Identity.IsAuthenticated) { <div> <table cellpadding="2" cellspacing="2"> <tr> <th>Claim</th> <th>Value</th> </tr> @foreach (var claim in User.Claims) { <tr> <td>@claim.Type</td> <td>@claim.Value</td> </tr> } </table> </div> }
```

4. Save the above all changes.

## Task 3: Build and test the app

1. Execute the following command in a command prompt to compile and run the application:

```
powershell dotnet build dotnet run
```

2. Open a browser and navigate to the url **https://localhost:5001**. The web application will redirect you to the Azure AD sign in page.

3. Sign in using a Work and School account from your Azure AD directory. Azure AD will redirect you back to the web application. Notice some of the details from the claims included in the ID token.

The screenshot shows the 'msidentity\_aspnet' application's welcome screen. At the top, there is a navigation bar with 'msidentity\_aspnet', 'Home', and 'Privacy' links. To the right, it shows the user 'Hello admin@M365x068225.onmicrosoft.com! Sign out'. Below the navigation bar, the word 'Welcome' is centered. Underneath 'Welcome', there is a link 'Learn about building Web apps with ASP.NET Core.' A table titled 'Claims' is displayed, listing various claims and their values. Some specific claims are highlighted with red boxes: 'http://schemas.microsoft.com/claims/authnmethodsreferences', 'http://schemas.xmlsoap.org/ws/2005/05/identity/claims/surname', 'http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname', 'http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name', 'http://schemas.microsoft.com/identity/claims/objectidentifier', 'http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier', 'http://schemas.microsoft.com/identity/claims/tenantid', 'http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name', and 'http://schemas.xmlsoap.org/ws/2005/05/identity/claims/upn'. The 'surname' value is 'Administrator', 'givenname' is 'MOD', 'name' is 'MOD Administrator', 'objectidentifier' is 'b9dd14c5-1943-448d-a4bf-70bd0ccd2592', 'nameidentifier' is 'vVexcWIDJLUz0tVpBSCZdCP-nkPkAdQKJfg5kahg', 'tenantid' is '7a4e07e0-76e4-4ef6-91d2-2aa01ba662a7', 'name' is 'admin@M365x068225.onmicrosoft.com', and 'upn' is 'admin@M365x068225.onmicrosoft.com'. The 'utl' claim has a value of '4lxz2p4z80imXz4F6MA0AQ'.

Claim	Value
aio	42VgYEiL1RUsNu7XvPw3OaMl++6Wi2zbcj9Vl9wqmadxfGKn8lUA
http://schemas.microsoft.com/claims/authnmethodsreferences	pwd
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/surname	Administrator
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname	MOD
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name	MOD Administrator
http://schemas.microsoft.com/identity/claims/objectidentifier	b9dd14c5-1943-448d-a4bf-70bd0ccd2592
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier	vVexcWIDJLUz0tVpBSCZdCP-nkPkAdQKJfg5kahg
http://schemas.microsoft.com/identity/claims/tenantid	7a4e07e0-76e4-4ef6-91d2-2aa01ba662a7
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name	admin@M365x068225.onmicrosoft.com
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/upn	admin@M365x068225.onmicrosoft.com
utl	4lxz2p4z80imXz4F6MA0AQ

4. Take special note of the **tenantid** and **upn** claim. These claims indicate the ID of the Azure AD directory and ID of the user that signed in. Make a note of these values to compare them to other options in a minute.
5. Now try logging in as a user from a different organization. Select the **Sign out** link in the top left. Wait for Azure AD and the web application signs out the current user. When the web application reloads, repeat the sign in process, except this time try signing in as a user from a different organization or use a Microsoft Account.
6. Notice Azure AD will reject the user's sign in, explaining that the user's account doesn't exist in the current tenant.



## Sign in

Sorry, but we're having trouble signing you in.

AADSTS50020: User account 'MeganB@M365x460234.OnMicrosoft.com' from identity provider 'https://sts.windows.net/7adf0e7f-07b3-4eb5-ba39-7ea421035371/' does not exist in tenant '████████' and cannot access the application '0b841307-928d-400d-854d-37aaa2b06890'(Hello ASP.NET Core Identity 01) in that tenant. The account needs to be added as an external user in the tenant first. Sign out and sign in again with a different Azure Active Directory user account.

Before this user can access this application, they need to be added as a guest into the Azure AD directory where the application was registered. Proceed with the next task to invite a guest user to your organization.

## Task 4: Invite a guest user from another organization

In this task, you will configure your Active Directory tenant to allow external users and then will invite a guest user from another Active Directory organization.

1. From the Azure portal, navigate to **Azure Active Directory**.
2. In the left-hand navigation, select **User**.
3. Examine the external user settings for available to administrators by selecting **User Settings** and then **Manage external collaboration settings**.

The screenshot shows the Azure Active Directory admin center interface. On the left, there's a navigation sidebar with options like All users, Deleted users, Password reset, and User settings (which is highlighted with a red box). The main content area has sections for Enterprise applications, App registrations, Administration portal, LinkedIn account connections, and External users. The External users section (also highlighted with a red box) contains a link to 'Manage external collaboration settings'. At the top right, there are Save and Discard buttons.

4. Notice that administrators can configure the Azure AD directory so guest users have limited rights compared to other users, and who can invite guest users.

External collaboration settings X

Save Discard

Guest user access

Guest user access restrictions (Preview) ⓘ

[Learn more](#)

Guest users have the same access as members (most inclusive)

Guest users have limited access to properties and memberships of directory objects

Guest user access is restricted to properties and memberships of their own directory objects (most restrictive)

Guest invite settings

Admins and users in the guest inviter role can invite ⓘ

Yes  No

Members can invite ⓘ

Yes  No

Guests can invite ⓘ

Yes  No

Enable Email One-Time Passcode for guests (Preview) ⓘ

[Learn more](#)

Yes  No

Enable guest self-service sign up via user flows (Preview) ⓘ

[Learn more](#)

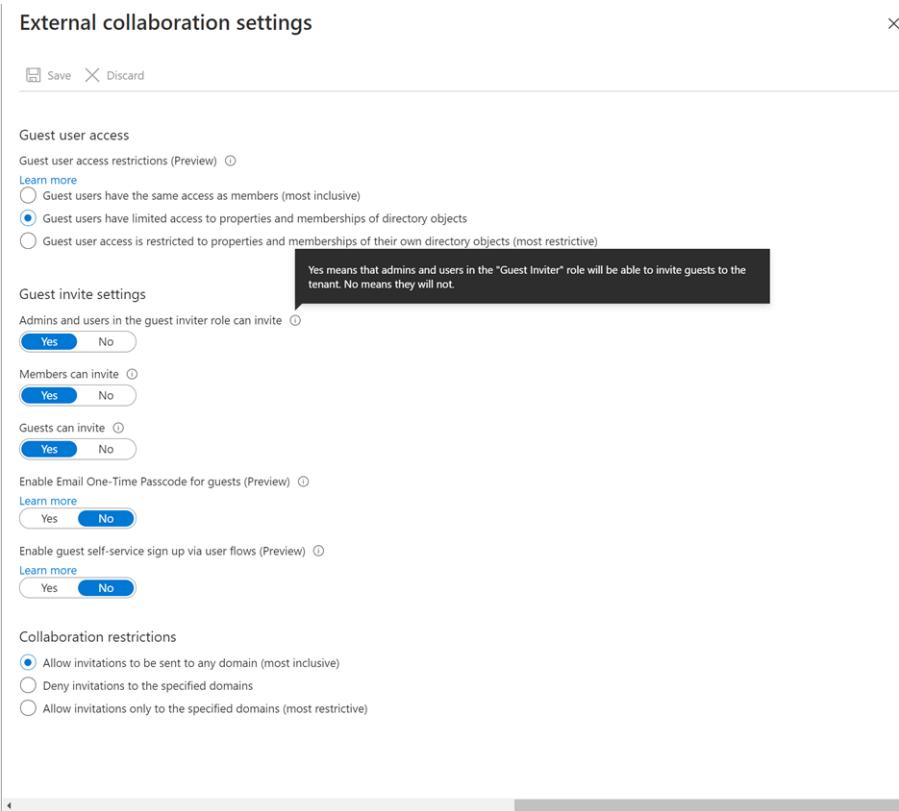
Yes  No

Collaboration restrictions

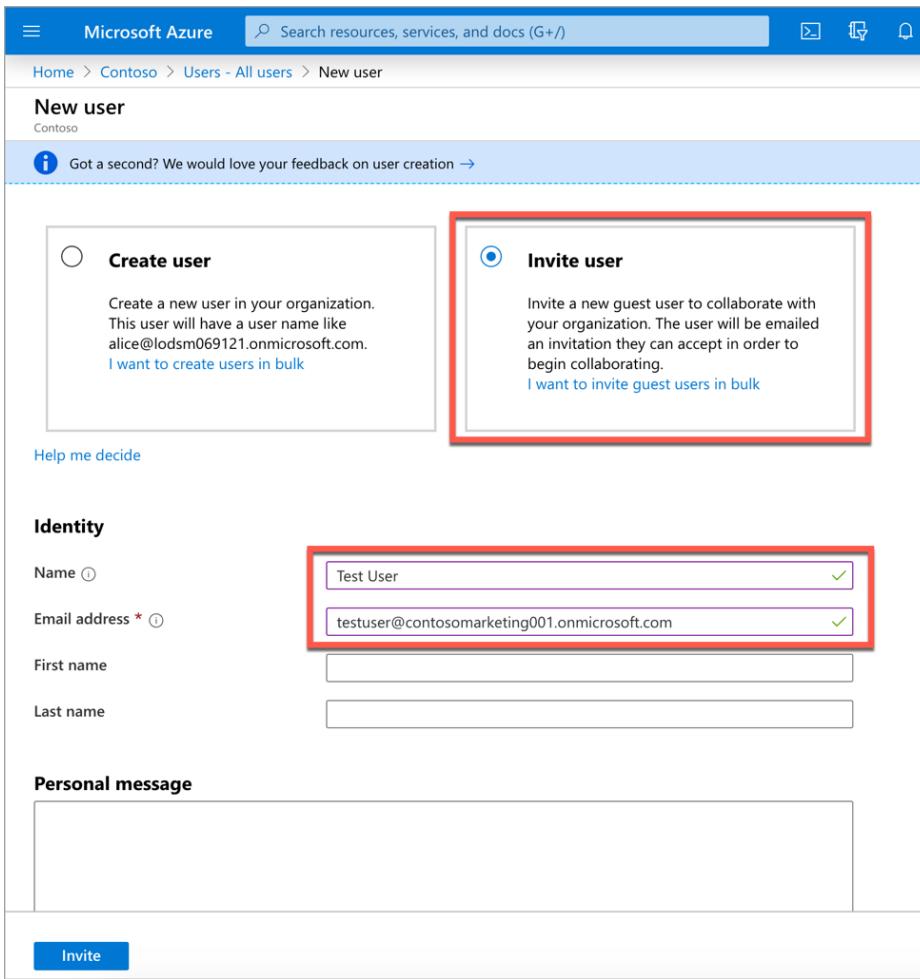
Allow invitations to be sent to any domain (most inclusive)

Deny invitations to the specified domains

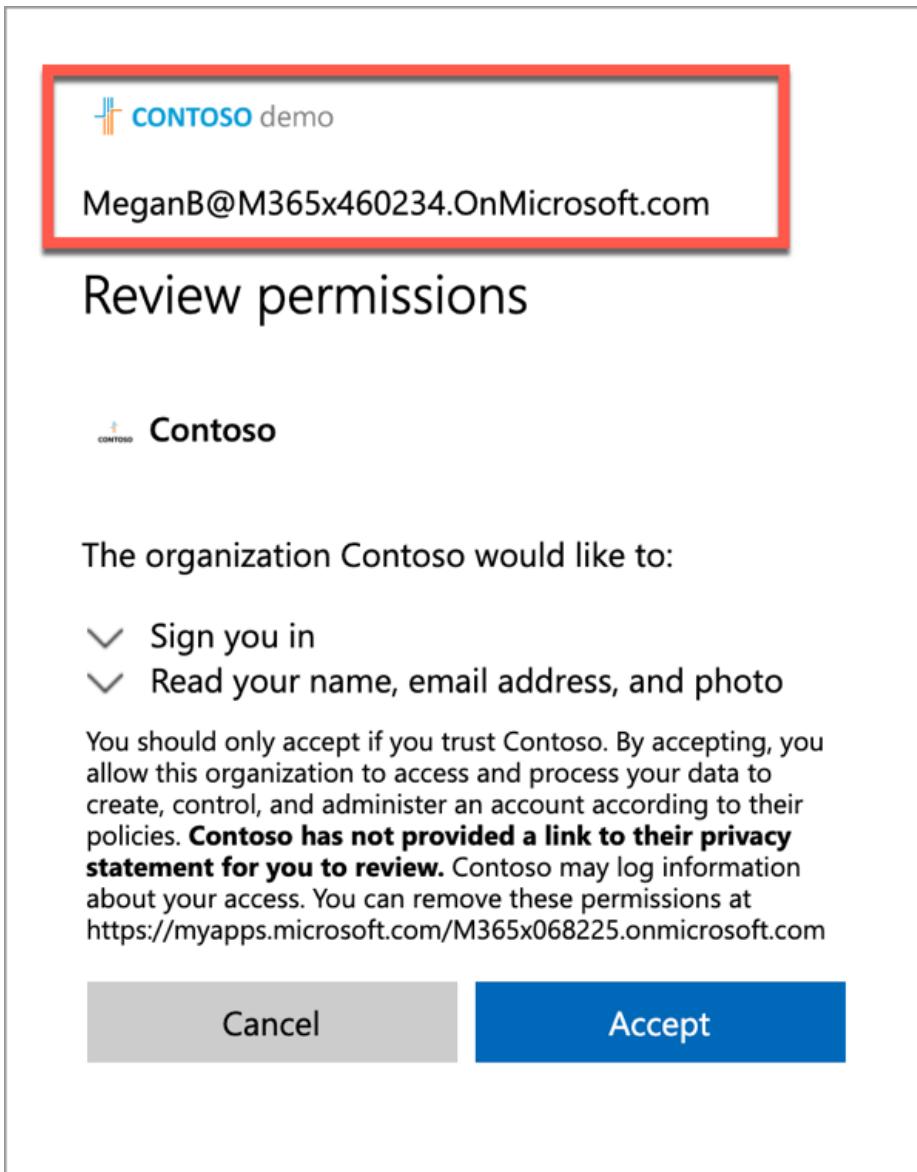
Allow invitations only to the specified domains (most restrictive)



5. Now let's invite a guest user. Select **All users** in the left-hand navigation, and then select **New guest user**:
6. On the **New user** page, select **Invite user** and then add the guest user's information:
  - **Name:** Test User
  - **Email address:** Type the email address of the new user you created in Exercise 1 Task 3. *For example: testuser@contosomarketing001.onmicrosoft.com*



7. Select **Invite** to automatically send the invitation to the guest user.
8. Now let's try to sign in with the user. In the browser, navigate to **<https://localhost:5001>**.
9. This time, after successfully logging in, the user's Azure AD directory will prompt the user to grant the application's Azure AD directory permissions to sign in as the user and obtain basic information about the user.



10. Take note of what is happening at this point. The original Azure AD directory is not signing in the user, rather the user has been redirected to sign in with their Azure AD directory. Once they sign in, their Azure AD directory will provide a token to our directory that is used to verify the user is authenticated and authorized the application to obtain their basic profile information. It then creates a new access token that can be used by our ASP.NET Core web application.
11. After selecting **Accept**, the user is taken to our ASP.NET application. Notice the difference in some of the claims.
  - The **identityprovider** claim is the ID of the Azure AD directory that authenticated the user. This claim is the user's Azure AD directory
  - The **tenantid** claim is the ID of the Azure AD directory our application is registered in. Notice this value is not the same as the **identityprovider** claim, indicating the user's identity is in one directory while they have been added as a guest user to another Azure AD directory.

# Welcome

Learn about [building Web apps with ASP.NET Core](#).

Claim	Value
aio	AUQAU/8NAAAAA3JX+tQN94Hx5t88Cj4SAwrTpWnxgDyliF/tfWqZFW+LT2
http://schemas.microsoft.com/claims/authnmethodsreferences	pwd
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/emailaddress	MeganB@M365x460234.onmicrosoft.com
http://schemas.microsoft.com/identity/claims/identityprovider	https://sts.windows.net/7adfd0e7f-07b3-4eb5-ba39-7ea421035371/
name	Megan Bowen
http://schemas.microsoft.com/identity/claims/objectidentifier	a028b9b0-1ae8-4f59-8d3b-5c07eef0e01f
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier	Lx2U6RdlH1de6a0Y8HyWA-N8oOAE7_IWKNNeTm8_4LI
http://schemas.microsoft.com/identity/claims/tenantid	7a4e07e0-76e4-4ef6-91d2-2aa01ba662a7
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name	MeganB@M365x460234.onmicrosoft.com
uti	bZluluIlyF0eHp6MV8osEAA

12. Stop the web server by going back to the running project in Visual Studio Code. From the Terminal, press **CTRL+C** in the command prompt.

## Review

In this exercise, you created an ASP.NET Core web application and Azure AD application that

allows guest users from partner Azure AD directories to sign in and access the application. You

then invited a guest user to the directory and signed into the application with this user. ♦♦♦#

Exercise 4: Configuring permissions to consume an API

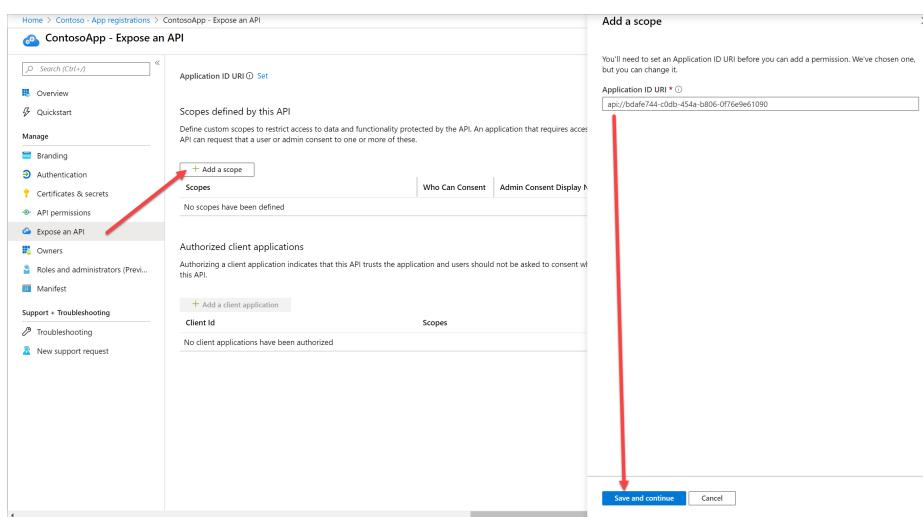
In this exercise, you'll learn how to configure an application to expose a new scope to make it available to client applications.

## **Task 1: Navigate to an application to configure**

1. In the left-hand navigation pane of the Azure portal, select the **Azure Active Directory** service and then select **App registrations**.
2. Find and select the application you want to configure such as **ContosoApp**. Once you've selected the app, you'll see the application's **Overview** or main registration page.
3. Navigate to **Expose an API**.

## Task 2: Add a scope

1. In the **Scopes defined by this API** section, select **Add a scope**.
2. If you have not set an **Application ID URI**, you will see a prompt to enter one. Enter your application ID URI or use the one provided and then select **Save and continue**.



3. When the **Add a scope** page appears, enter your scope's information:
  1. In the **Scope name** textbox, enter a meaningful name for your scope. For example, **Employees.Read.All**.
  2. For **Who can consent**, select whether this scope can be consented to by users, or if admin consent is required. Select **Admins only** for higher-privileged permissions.
  3. Provide a user-friendly **Admin consent display name** and **Admin consent description**.
  4. Provide a user-friendly **User consent display name** and **User consent description**.
  5. Set the **State** and select **Add scope** when you're done.
  6. Your Add a Scope pane should look similar to the image below.

**Add a scope**

---

Scope name \* ⓘ  
 ✓  
 api://bdafe744-c0db-454a-b806-0f76e9e61090/Employees.Read.All

Who can consent? ⓘ  
 Admins and users    Admins only

Admin consent display name \* ⓘ  
 ✓

Admin consent description \* ⓘ  
 ✓

User consent display name ⓘ  
 ✓

User consent description ⓘ

State ⓘ  
 Enabled    Disabled

## Expose a new scope or role through the application manifest

Now that you learned how to create a new scope using the UI, next you will add a new scope through the application manifest.

1. In the left navigation menu of your app, select **Manifest**.
2. A web-based manifest editor opens, allowing you to **Edit** the manifest within the portal. Optionally, you can select **Download** and edit the manifest locally, and then use **Upload** to reapply it to your application.
3. Scroll down until you find the **oauth2Permissions** collection.
4. Below is the JSON element that was added to the oauth2Permissions collection.

```
json { "adminConsentDescription": "Allow the application to have read-only access to all Employee data.", "adminConsentDisplayName": "Read-only access to Employee records", "id": "ede407eb-3c3a-4918-9e2e-803d2298e247", "isEnabled": true, "type": "User", "userConsentDescription": "Allow the application to have read-only access to your Employee data.", "userConsentDisplayName": "Read-only access to your Employee records", "value": "Employees.Read.All" }
```

**Note:** The id value must be generated programmatically or by using a GUID generation tool such as guidgen. The id represents a unique identifier for the scope as exposed by the web API. Once a client is appropriately configured with permissions to access your web API, it is issued an OAuth 2.0 access token by Azure AD. When the client calls the web

API, it presents the access token that has the scope (scp) claim set to the permissions requested in its application registration.

You can expose additional scopes later as necessary. Consider that your web API might expose multiple scopes associated with a variety of different functions. Your resource can control access to the web API at runtime by evaluating the scope (scp) claim(s) in the received OAuth 2.0 access token.

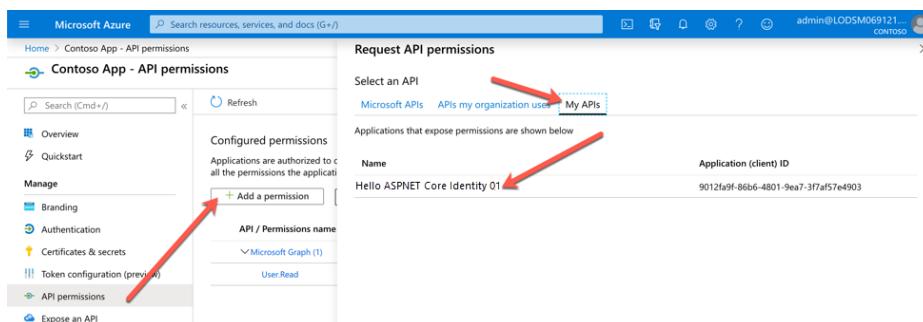
## Task 3: Add a client application and API permissions

### Add permissions scope to Hello ASPNET Core Identity 01

1. Navigate back to **App registrations**.
2. Select the ContosoApp you created earlier in this lab and copy the **Application (client) ID**.
3. Navigate back to the **Overview** page for your **Hello ASPNET Core Identity 01** app.
4. Navigate to **Expose an API**.
5. Under the **Authorized client applications** section, select **Add a client application**.
6. Paste the Application (client) ID you copied from the **ContosoApp** app into the **Client ID** text box.
7. Check the API scope presented under the **Authorize** scope.
8. Select **Add application**.

### Add permissions to access Web APIs for ContosApp

1. Navigate back to **App registrations** and select **ContosApp**.
2. Select **API permissions** and then select the **Add a permission** button.
3. From the Request API permissions dialog, select **My APIs**. You should now see your Hello ASPNET Core Identity 01 app available to select.
4. Select **Hello ASPNET Core Identity 01**.



5. Expand **Employees (1)**, select **Employees.Read.All** and then select **Add permissions**.
6. Wait for **Preparing the consent** to finish then select **Grant admin consent for Contoso**.
7. From the **Permissions requested** dialog, select **Yes**.
8. Your **ContosoApp** is now configured and authorized to use the API permissions from the **Hello ASPNET Core Identity 01** app.
9. This exercise is now complete.

## Review

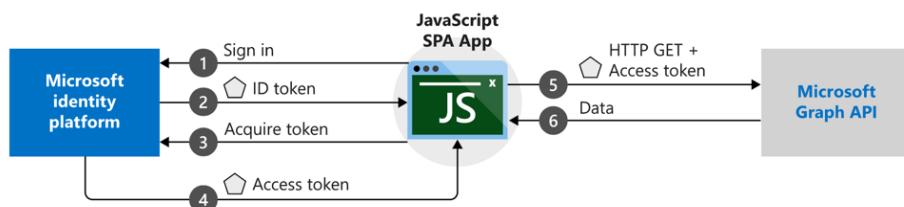
In this exercise, you learned how to define scope and how to authorize applications to make the API calls.

### ❖❖❖# Exercise 5: Implementing authorization to consume an API

This exercise demonstrates how a JavaScript single-page application (SPA) can:

- Sign in personal accounts, as well as work and school accounts.
- Acquire an access token.
- Call the Microsoft Graph API or other APIs that require access tokens from the Microsoft identity platform endpoint.

The sample application used in this exercise enables a JavaScript SPA to query the Microsoft Graph API or a web API that accepts tokens from the Microsoft identity platform endpoint. In this scenario, after a user signs in, an access token is requested and added to HTTP requests through the authorization header. Token acquisition and renewal are handled by the Microsoft Authentication Library (MSAL).



## Task 1: Download sample project

1. Download the sample project for Node.js:
  1. To run the project by using a local web server, such as Node.js, download the project files to your **C:/Labfiles** directory.  
Visit <https://github.com/Azure-Samples/active-directory-javascript-graphapi-v2/releases> and download the latest release: **Source code (zip)**.
2. Navigate to where the download zip file is and unblock file.
  1. Right-select and select **Properties**.
  2. Select the **Unblock** checkbox and select **OK**.
3. Extract the zipped file.
4. Open the project in Visual Studio Code.
  1. Launch Visual Studio Code as Administrator and browse for the **active-directory-javascript-graphapi-v2-quickstart** root directory and open.

## Task 2: Register an application

1. On the left menu, navigate to **App registrations**.
2. Select **New Registration**.
3. From the **Register an application** pane, perform the following actions:
  1. In the **Name** text box, enter a meaningful application name that will be displayed to users of the app, for example **JavaScript-SPA-App**.
  2. Under **Supported account type**, select **Accounts in any organizational directory and personal Microsoft accounts**.

### For setting a redirect URL for the Node.js project

Follow these steps if you choose to use the Node.js project. For Node.js, you can set the web server port in the server.js file. This project uses port 30662, but you can use any other available port.

1. To set up a redirect URL in the application registration information, switch back to the **Application Registration** pane, and do either of the following:
  1. Set `http://localhost:30662/` as the **Redirect URL**.
  2. If you're using a custom TCP port, use `http://localhost: [port] /` (where **[port]** is the custom TCP port number).
2. Select **Register**.
3. On the app **Overview** page, note the **Application (client) ID** value for later use.
4. This sample project requires the **Implicit grant flow** to be enabled. In the left pane of the registered application, select **Authentication**.
5. In **Advanced** settings, under **Implicit grant**, select the **ID tokens** and **Access tokens** check boxes. ID tokens and access tokens are required because this app must sign in users and call an API.
6. Select **Save**.

# Task 3: Permission and scope setup

## Update the App API permissions

1. In the left navigation, navigate to **API Permissions**.

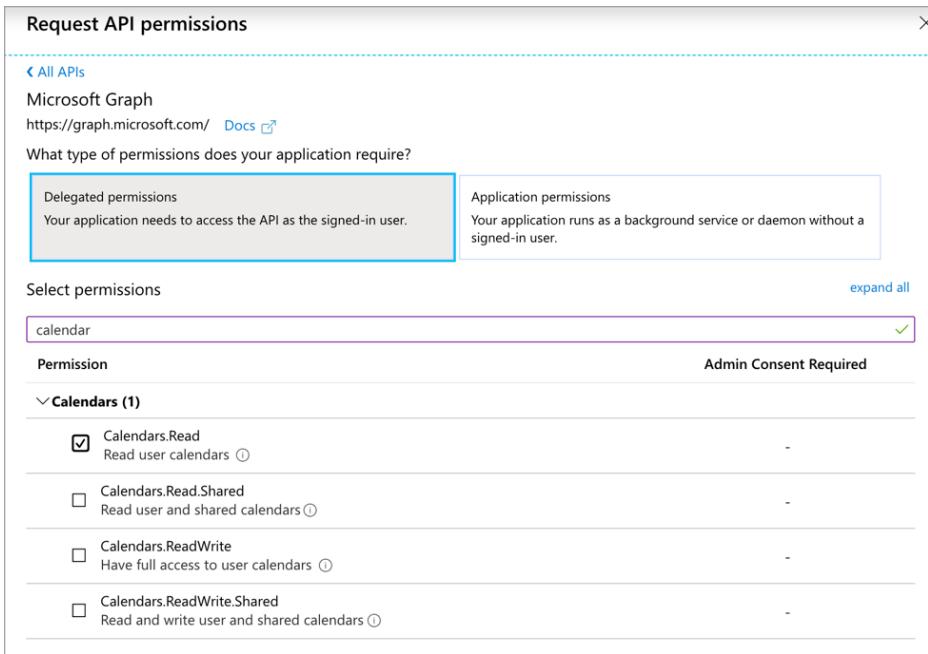
The screenshot shows the 'JavaScript-SPA-App - API permissions' page in the Azure portal. The left sidebar has sections like Overview, Quickstart, Manage (with Branding, Authentication, Certificates & secrets, API permissions selected, Expose an API, Owners, Roles and administrators, and Manifest). The main area shows 'Configured permissions' with a note about consent. A table lists one permission: Microsoft Graph (1) with User.Read scope, Type Delegated, Admin Consent... status, and Sign in and read user profile description.

2. Select **Add a permission**.

3. Next, select **Microsoft Graph** from **Microsoft APIs**.

The screenshot shows the 'Request API permissions' dialog. The 'Microsoft APIs' tab is selected. An arrow points to the 'Microsoft Graph' card, which is described as taking advantage of the tremendous amount of data in Office 365, Enterprise Mobility + Security, and Windows 10. Other cards include Azure Rights Management Services, Azure Service Management, Dynamics 365 Business Central, Flow Service, Intune, Office 365 Management APIs, OneNote, Power BI Service, SharePoint, and Skype for Business.

4. Select **Delegated Permissions** and under **Select permissions**, search for **calendars** and select **Calendars.Read**.



5. Search for people and select **People.Read** permission and provide admin consent.

6. Select **Add permissions**.

1. Wait for **Preparing for consent** to finish then select **Grant admin consent for Contoso**.
2. From the Permissions requested dialog, select **Yes**.
3. Your **JavaScript-SPA-App** app is now configured and authorized to use the Calendars.Read and People.Read permissions for Microsoft Graph.

## Update solution code

1. Navigate back to Visual Studio Code and open the **index.html** file.

2. Locate and select the following code:

```
html <div class="leftContainer"> <p id="WelcomeMessage">Welcome to the Microsoft Authentication Library For Javascript Quickstart</p> <button id="SignIn" onclick="signIn()">Sign In</button> </div>
```

3. Replace the selected code with the following, to add a new button to the application:

```
html <div class="leftContainer"> <p id="WelcomeMessage">Welcome to the Microsoft Authentication Library For Javascript Quickstart</p> <button id="SignIn" onclick="signIn()">Sign In</button> <button id="Share" onclick="acquireTokenPopupAndCallMSGraph()">Share</button> </div>
```

4. Locate the object **requestObj** and change the scopes to **people.read**.

```
javascript var requestObj = { scopes: ["people.read"] };
```

5. Update the graph API URL to call the people object.

```
javascript var graphConfig = { graphMeEndpoint: "https://graph.microsoft.com/v1.0/me", graphPeopleEndpoint:
```

```
"https://graph.microsoft.com/v1.0/people" };
```

6. Create a new endpoint for **people.read** and replace the method **acquireTokenPopupAndCallMSGraph** with the code below.

```
javascript function acquireTokenPopupAndCallMSGraph() { //Always start with
acquireTokenSilent to obtain a token in the signed in user from cache
myMSALObj.acquireTokenSilent(requestObj).then(function (tokenResponse) {
callMSGraph(graphConfig.graphMePeopleEndpoint, tokenResponse.accessToken,
graphAPICallback); }).catch(function (error) { console.log(error); // Upon
acquireTokenSilent failure (due to consent or interaction or login required
ONLY) // Call acquireTokenPopup (popup window) if
(requiresInteraction(error.errorCode)) {
myMSALObj.acquireTokenPopup(requestObj).then(function (tokenResponse) {
callMSGraph(graphConfig.graphPeopleEndpoint, tokenResponse.accessToken,
graphAPICallback); }).catch(function (error) { console.log(error); });
}); }
```

7. Find the **var msalConfig** code block and replace the following:

1. <Enter\_the\_Application\_Id\_here> is the **Application (client) ID** for the application you registered.
2. <Enter\_the\_Tenant\_info\_here> is set to one of the following options:
  - If your application supports **Accounts in this organizational directory**, replace this value with the **Tenant ID** or **Tenant name** (for example, **contoso.microsoft.com**).
  - If your application supports **Accounts in any organizational directory**, replace this value with **organizations**.
  - If your application supports **Accounts in any organizational directory and personal Microsoft accounts**, replace this value with **common**. To restrict support to Personal Microsoft accounts only, replace this value with **consumers**.

## Task 4: Run the application

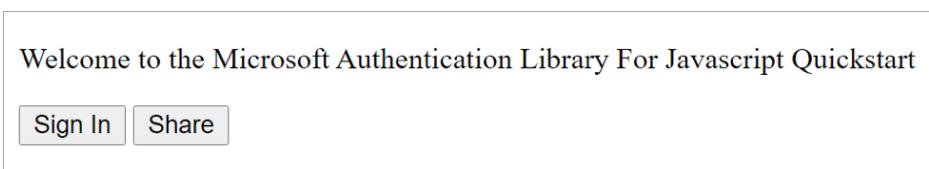
1. From **Visual Studio Code**, open **Terminal**. Type the following and hit ENTER:

```
typescript npm install
```

2. From the Terminal, type the following and hit ENTER.

```
typescript node server.js
```

3. From the browser, launch: **http://localhost:30662**



Welcome to the Microsoft Authentication Library For Javascript Quickstart

[Sign In](#) [Share](#)

A screenshot of a web browser displaying a landing page for the Microsoft Authentication Library for JavaScript. The page has a light blue header with the text "Welcome to the Microsoft Authentication Library For Javascript Quickstart". Below the header are two buttons: "Sign In" and "Share".

4. Select **Sign In**.

5. If the **Permissions requested** dialog opens, select **Accept**.

The screenshot shows a Microsoft Edge browser window with the following details:

- Title bar: Sign in to your account - MS Learning - Microsoft Edge
- Address bar: https://login.microsoftonline.com/organizations/oauth2/v2
- Content area:
  - Microsoft logo
  - Email address: admin@lodsm069121.onmicrosoft.com
  - Section title: Permissions requested
  - App details: JavaScript-SPA-App, unverified
  - Description: This app would like to:
  - Granted permissions:
    - Read your relevant people list
    - View your basic profile
    - Maintain access to data you have given it access to
  - Pending permission:
    - Consent on behalf of your organization
  - Text: Accepting these permissions means that you allow this app to use your data as specified in their terms of service and privacy statement. **The publisher has not provided links to their terms for you to review.** You can change these permissions at <https://myapps.microsoft.com>. [Show details](#)
  - Buttons: Cancel (gray), Accept (blue)

6. You should now be authenticated successfully.

Welcome admin@LODSM069121.onmicrosoft.com to Microsoft Graph API

[Sign Out](#) [Share](#)

7. Go back to the Visual Studio Code project. Stop the running project by selecting **Ctrl + C**.

8. Exit Visual Studio Code.

## Review

In this exercise, you implemented authorization and incremental consent using the Microsoft Identity. # Exercise 6: Creating a service to access Microsoft Graph

In this exercise, you'll create an Azure AD application and single page application for a user to sign in and display their information on the page. This application will use MSAL.js for the access token request.

# Task 1: Single page web application

In this application, you'll create an ASP.NET Core web application that allows users from the current organization to sign in and display their information.

## Create a Node.js web application

1. Open your command prompt, navigate to a directory where you want to save your work, create a new folder, and change directory into that folder. For example:

```
powershell cd c:/LabFiles md HelloWorldIdentity cd HelloWorldIdentity
```

2. Execute the following command to create a new Node.js application:

```
powershell npm init -y
```

3. Install the Node.js webserver **express** and HTTP request middleware **morgan** into the application:

```
powershell npm install express morgan
```

4. Open the project in Visual Studio code by executing:

```
powershell code .
```

5. Create a new file **server.js** in the root of the folder and add the following JavaScript to it. This code will start the web server:

```
javascript var express = require('express'); var app = express(); var  
morgan = require('morgan'); var path = require('path'); var port = 3007;  
app.use(morgan('dev')); // set the front-end folder to serve public assets.  
app.use(express.static('web'));// set up our one route to the index.html  
file. app.get('*', function (req, res) { res.sendFile(path.join(__dirname +  
'/index.html'))}); // Start the server. app.listen(port);  
console.log(`Listening on port ${port}...`); console.log('Press CTRL+C to  
stop the web server...');
```

## Create a web page for the user to sign in and display details

Create a new folder **web** in the current folder and add a new file **index.html** in the folder. Add the following code to the **index.html** file:

```
html <!DOCTYPE html> <html> <head> <title>Getting Started with Microsoft  
identity</title> <script  
src="https://cdnjs.cloudflare.com/ajax/libs/bluebird/3.5.5/bluebird.min.js">  
</script> <script src="https://alcdn.msftauth.net/lib/1.1.3/js/msal.min.js">  
</script> </head> <body> <div class="container"> <div> <p  
id="WelcomeMessage">Microsoft Authentication Library For Javascript (MSAL.js)  
Exercise</p> <button id="SignIn" onclick="signIn()">Sign In</button> </div>  
<div> <pre id="json"></pre> </div> <script> var msalConfig = { auth: {  
clientId: '', authority: '', redirectURI: '' }, cache: { cacheLocation:  
"localStorage", storeAuthStateInCookie: true } }; var graphConfig = {  
graphMeEndpoint: "https://graph.microsoft.com/v1.0/me", requestObj: { scopes:  
["user.read"] } }; var msalApplication = new  
Msal.UserAgentApplication(msalConfig); // init the auth handling on the page  
initPage(); // TODO: add CODE before this line // TODO: add FUNCTIONS before  
this line function initPage() { // Browser check variables var ua =  
window.navigator.userAgent; var msie = ua.indexOf('MSIE '); var msie11 =
```

```

ua.indexOf('Trident/'); var msedge = ua.indexOf('Edge/'); var isIE = msie > 0
|| msie11 > 0; var isEdge = msedge > 0; // if you support IE, recommendation:
sign in using Redirect APIs vs. popup // Browser check variables // can change
this to default an experience outside browser use var loginType = isIE ?
"REDIRECT" : "POPUP"; // runs on page load, change config to try different
login types to see what is best for your application switch (isIE) { case true:
document.getElementById("SignIn").onclick = function () {
msalApplication.loginRedirect(graphConfig.requestObj); }; // avoid duplicate
code execution on page load in case of iframe and popup window if
(msalApplication.getAccount() &&
!msalApplication.isCallback(window.location.hash)) { updateUserInterface();
acquireTokenRedirectAnd GetUser(); } break; case false: // avoid duplicate code
execution on page load in case of iframe and popup window if
(msalApplication.getAccount()) { updateUserInterface();
acquireTokenPopupAnd GetUser(); } break; default: console.error('Please set a
valid login type'); break; } } </script> </body> </html>

```

**Note:** The remainder of this exercise instructs you to add code to this **index.html** file. Pay close attention where you add the code using the using the two **TODO:** comments for placement.

1. Add the following function to the **index.html** file immediately before the **// TODO: add FUNCTIONS before this line** comment that will configure the welcome message for the page:

```

javascript function updateUserInterface() { var divWelcome =
document.getElementById('WelcomeMessage'); divWelcome.innerHTML = `Welcome
<strong>${msalApplication.getAccount().userName}</strong> to Microsoft
Graph API`; var loginbutton = document.getElementById('SignIn');
loginbutton.innerHTML = 'Sign Out'; loginbutton.setAttribute('onclick',
'signOut();'); }

```

2. Next, add the following functions to **index.html** immediately before the **// TODO: add FUNCTIONS before this line** comment.

- These functions request an access token from Microsoft identity and submit a request to Microsoft Graph for the current user's information. The function **acquireTokenPopupAnd GetUser()** uses the popup approach that works for all modern browsers while the **acquireTokenRedirectAnd GetUserFromMSGraph()** function uses the redirect approach that is suitable for Internet Explorer:

```

javascript function acquireTokenPopupAnd GetUser() {
msalApplication.acquireTokenSilent(graphConfig.requestObj)
.then(function (tokenResponse) {
getUserFromMSGraph(graphConfig.graphMeEndpoint,
tokenResponse.accessToken, graphAPICallback); }).catch(function (error)
{ console.log(error); if (requiresInteraction(error.errorCode)) {
msalApplication.acquireTokenPopup(graphConfig.requestObj).then(function
(tokenResponse) { getUserFromMSGraph(graphConfig.graphMeEndpoint,
tokenResponse.accessToken, graphAPICallback); }).catch(function (error)
{ console.log(error); }); } });
function acquireTokenRedirectAnd GetUser() {
msalApplication.acquireTokenSilent(graphConfig.requestObj).then(function
(tokenResponse) { getUserFromMSGraph(graphConfig.graphMeEndpoint,
tokenResponse.accessToken, graphAPICallback); }).catch(function (error)
{ console.log(error); if (requiresInteraction(error.errorCode)) {
msalApplication.acquireTokenRedirect(graphConfig.requestObj); } });
function requiresInteraction(errorCode) { if (!errorCode ||
!errorCode.length) { return false; } return errorCode ===
"consent_required" || errorCode === "interaction_required" || errorCode
=== "login_required"; }

```

**Note:** These functions first attempt to retrieve the access token silently from the currently signed in user. If the user needs to sign in, the functions will trigger either the popup or

redirect authentication process. The redirect approach to authenticating requires an extra step. The MSAL application on the page needs to see if the current page was requested based on a redirect from Azure AD. If so, it needs to process information in the URL request provided by Azure AD.

1. Add the following function immediately before the **// TODO: add FUNCTIONS before this line** comment:

```
javascript function authRedirectCallBack(error, response) { if (error) {  
    console.log(error); } else { if (response.tokenType === "access_token") {  
        getUserFromMSGraph(graphConfig.graphMeEndpoint, response.accessToken,  
        graphAPICallback); } else { console.log("token type is:" +  
        response.tokenType); } } }
```

3. Configure MSAL to use this function by adding the following line immediately before the **// TODO: add CODE before this line** comment:

```
javascript msalApplication.handleRedirectCallback(authRedirectCallBack);
```

1. Once the user is authenticated, the code can submit a request to Microsoft Graph for the current user's information. The two **acquireToken\***() functions pass the access token acquired from Azure AD to the function:
2. Add the following function immediately before the **// TODO: add FUNCTIONS before this line** comment:

```
javascript function getUserFromMSGraph(endpoint, accessToken, callback) {  
    var xmlhttp = new XMLHttpRequest(); xmlhttp.onreadystatechange = function()  
    { if (this.readyState == 4 && this.status == 200)  
        callback(JSON.parse(this.responseText)); } xmlhttp.open("GET", endpoint,  
        true); xmlhttp.setRequestHeader('Authorization', 'Bearer ' + accessToken);  
    xmlhttp.send(); } function graphAPICallback(data) {  
    document.getElementById("json").innerHTML = JSON.stringify(data, null, 2);  
}
```

4. Finally, add the following two functions to implement a sign in and sign out capability for the button on the page. Add the following function immediately before the **// TODO: add FUNCTIONS before this line** comment:

```
javascript function signIn() {  
    msalApplication.loginPopup(graphConfig.requestObj).then(function  
    (loginResponse) { updateUserInterface(); acquireTokenPopupAnd GetUser();  
    }).catch(function (error) { console.log(error); }); } function signOut() {  
    msalApplication.logout(); }
```

## Task 2: Register a new application

1. From the Azure portal <https://portal.azure.com>, navigate to **Azure Active Directory**.
2. Select **Manage > App registrations** in the left-hand navigation.
3. On the **App registrations** page, select **New registration**.
4. On the **Register an application** page, set the values as follows:
  - **Name:** Hello World Identity
  - **Supported account types:** Accounts in this organizational directory only (Single tenant)
  - **Redirect URI:** Web = http://localhost:3007
5. Select **Register** to create the application.
6. On the **Hello World Identity** page, copy the values **Application (client) ID** and **Directory (tenant) ID**; you'll need these values later in this exercise.
7. On the **Hello World Identity** page, select the **1 web, 0 public client** link under the **Redirect URIs**.
8. Locate the section **Implicit grant** and select both **Access tokens** and **ID tokens**.
9. Select **Save** when finished setting these values.

## Task 3: Update the web page with the Azure AD application details

The last task is to configure the web page to use the Azure AD application. 1. Go back to the project in Visual Studio Code.

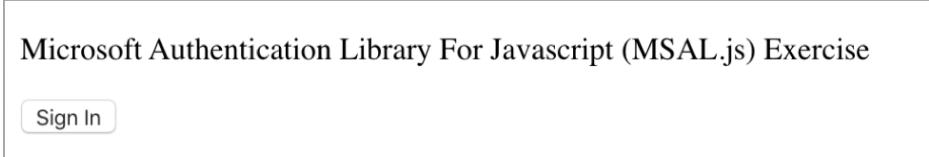
1. Open the **index.html** file and locate the `var msalConfig = {}` code. The auth object contains three properties you need to set as follows:
  - **clientId**: set to the Azure AD application's ID
  - **authority**: set to `https://login.microsoftonline.com/{{DIRECTORY_ID}}`, replacing the `{{DIRECTORY_ID}}` with the Azure AD directory ID of the Azure AD application.
  - **redirectURI**: set to the Azure AD application's redirect URI: `http://localhost:3007`

## Task 4: Test the web application

1. To test the web page, first start the local web server. In the command prompt, execute the following command from the root of the project:

```
powershell node server.js
```

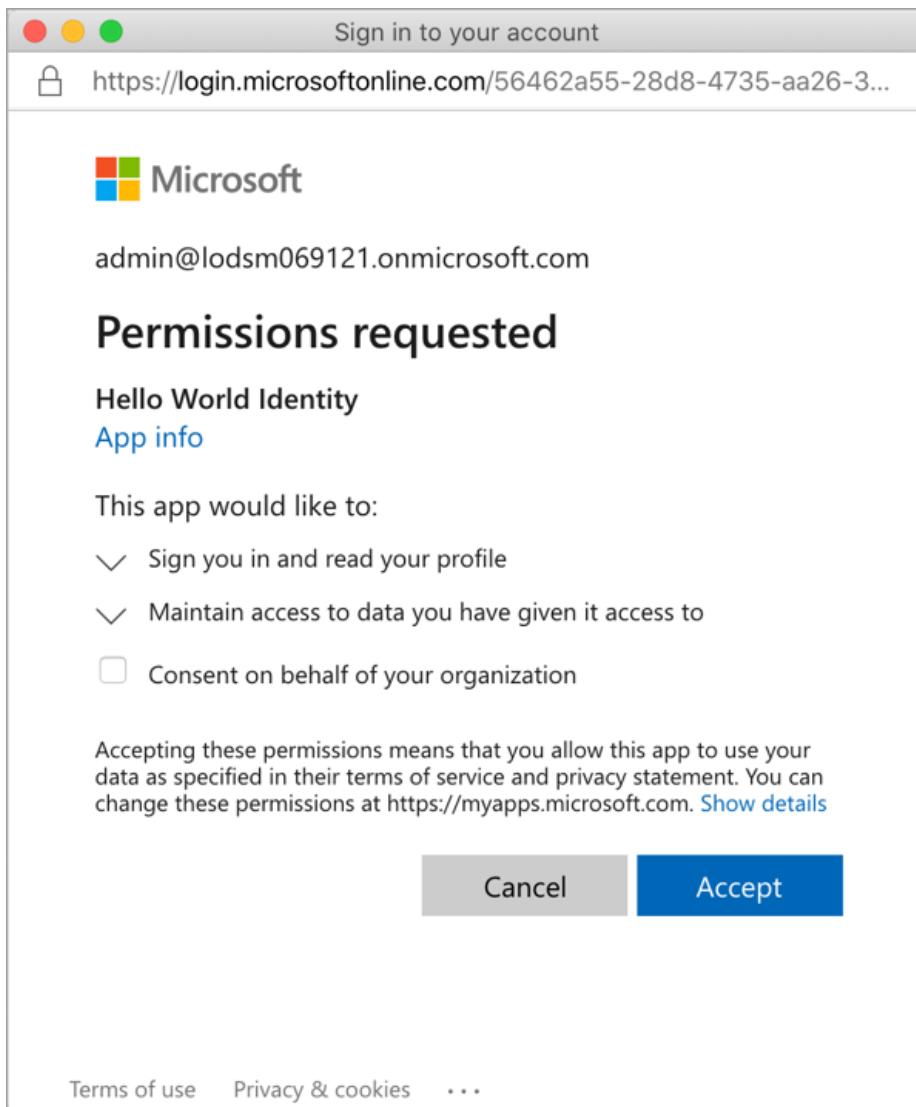
2. Next, open a browser and navigate to **http://localhost:3007**. The page initially contains a default welcome message and sign in button.



Microsoft Authentication Library For Javascript (MSAL.js) Exercise

Sign In

3. Select the **sign in** button.
4. Depending on the browser, you are using, a popup window will load or the page will redirect to the Azure AD sign in prompt.
5. Sign in using a **Work or School Account** and accept the permissions requested for the application by selecting **Accept**.



6. Depending on the browser you're using, the popup will disappear, or you will be redirected back to the web page. When the page loads, MSAL will request an access token and request your information from Microsoft Graph. After the request complete, it will display the results on the page:

Welcome **admin@LODSM069121.onmicrosoft.com** to Microsoft Graph API

[Sign Out](#)

```
{ "@odata.context": "https://graph.microsoft.com/v1.0/$metadata#users/$entity", "businessPhones": [ "8006427676" ], "displayName": "MOD Administrator", "givenName": "MOD", "jobTitle": null, "mail": "admin@LODSM069121.OnMicrosoft.com", "mobilePhone": "425-882-1032", "officeLocation": null, "preferredLanguage": "en-US", "surname": "Administrator", "userPrincipalName": "admin@LODSM069121.onmicrosoft.com", "id": "1adc26ec-31ec-412b-b9b1-75f63c0d8ff9" }
```

7. Stop the web server by going back to the running project in Visual Studio Code. From the Terminal, press **CTRL+C** in the command prompt.

## **Review**

In this exercise, you created an Azure AD application and single page application for a user to sign in and display their information on the page.

◆◆◆# Student lab manual

## Lab scenario

You are a developer in a large organization that is currently supporting applications and .NET solutions with multiple identity providers. To centralize the identity management for your organization, you decided to use Azure Active Directory to simplify the access to internal and external applications. You registered applications in Azure, implemented authentication, configured permissions for consuming the API, and created a service to access Microsoft Graph. As a result, IT can manage all applications from a centralized location to provide users a seamless sign-in experience.

## **Objectives**

After you complete this lab, you will be able to:

- Register an application in Azure AD.
- Implement authentication.
- Configure permissions to consume an API.
- Implement authorization to consume an API.
- Create a service to access Microsoft Graph.

## **Lab Setup**

Estimated duration: 180 minutes

# Instructions

**Note:** Lab virtual machine sign in instructions will be provided to you by your instructor.

## Review installed applications

Observe the taskbar located at the bottom of your Windows 10 desktop. The taskbar contains the icons for the applications you will use in this lab:

- Microsoft Edge
- File Explorer
- Visual Studio Code
- Microsoft Visual Studio 2019
- Windows PowerShell

❖❖❖# Exercise 1: Registering an application in Azure Active Directory

## Task 1: Open the Azure portal

1. On the taskbar, select the **Microsoft Edge** icon.
2. In the open browser window, navigate to the **Azure portal ([portal.azure.com](https://portal.azure.com))**.
3. Enter the **email address** for your Microsoft account.
4. Select **Next**.
5. Enter the **password** for your Microsoft account.
6. Select **Sign in**.

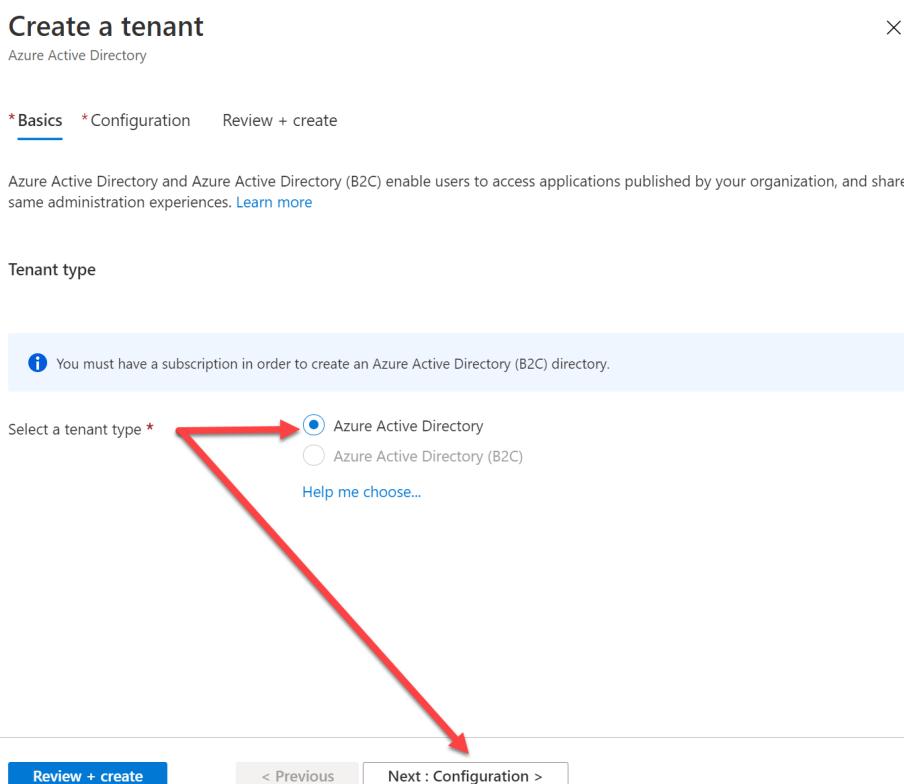
**Note:** If this is your first time signing in to the Azure Portal, a dialog box will appear offering a tour of the portal. Select Get Started to begin using the portal.

## Task 2: Create an Active Directory tenant

In this task, you will create a new Active Directory organization so you will understand the steps involved. Even though you will be creating a new Azure Active Directory organization, the remainder of the tasks will have you continue to work in the existing organization that was provided with your Azure portal.

### Create an Azure AD organization

1. Expand the left navigation pane, select **Azure Active Directory**.
2. Select **Create a tenant** located in the right side of the portal.
3. From the **Basics** tab, perform the following actions:
  1. In the **Select a tenant type** section, select **Azure Active Directory**.
  2. Select **Next: Configuration**.



4. From the **Configuration** tab, perform the following actions:
  1. **Organization name:** *Contoso Marketing Company*
  2. **Initial domain name:** *contosomarketingXXXX* where you replace XXXX with numbers or letters to make your domain name unique.
  3. Select the **Country or region**.

## Create a tenant

X

Azure Active Directory

\* Basics

\* Configuration

Review + create

### Directory details

Configure your new directory

Organization name \* ⓘ

Contoso Marketing Company



Initial domain name \* ⓘ

contosomarketingms001



contosomarketingms001.onmicrosoft.com

Country/Region ⓘ

United States



Datacenter location - United States

Datacenter location is based on the country/region selected above.

[Review + create](#)

< Previous

Next : Review + create >

### 4. Select **Review + create**.

### 5. Ensure settings are correct and validation passed.

## Create a tenant

X

Azure Active Directory

Validation passed.

\* Basics

\* Configuration

Review + create

### Summary

#### Basics

Tenant type

Azure Active Directory

#### Configuration

Organization name

Contoso Marketing Company

Initial domain name

contosomarketingms001.onmicrosoft.com

Country/Region

United States

Datacenter location

United States

[Create](#)

< Previous

Next >

### 6. Select **Create**.

7. A **Directory creation in progress, this will take a few minutes** message is displayed.

1. When directory creation is complete, select **Click here to navigate to your new directory**.

The screenshot shows the Azure Active Directory Overview page for the tenant "Contoso Marketing Company". The left sidebar contains navigation links such as Overview, Getting started, Preview hub, Diagnose and solve problems, Manage (Users, Groups, External Identities, Roles and administrators, Administrative units, Enterprise applications, Devices, App registrations, Identity Governance, Application proxy, Licenses), Azure AD Connect, Custom domain names, and Mobility (MDM and MAM). The main content area displays "Tenant information" (Your role: Global administrator, License: Azure AD Free, Tenant ID: 111086d9-f820-4c8f-b636-8141..., Primary domain: i.onmicrosoft.com) and "Azure AD Connect" status (Status: Not enabled, Last sync: Sync has never run). Below these sections is a "Sign-ins" chart showing activity from October 4 to November, with a single data point labeled "Sign ins".

## Task 3: Create a user in new Active Directory tenant

In this task, you will create a user in the new Active Directory tenant you just created in the previous steps. This user account will be used later in an exercise used for adding guest users to a single organization. 1. From the Active Directory page (Contoso Marketing Company), select **Users**.

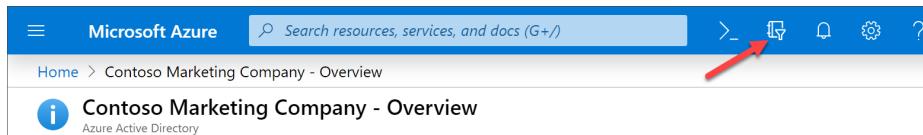
1. Select **New user**.
2. Ensure **Create user** is selected.
3. On the **New user** page, enter information for this user:
  - **User name:** testuser
  - **Name:** Test User
  - **First Name:** Test
  - **Last Name:** User
  - **Password:** Auto-generate passwordSelect **Show password**. Copy the autogenerated password provided in the Password box. You'll need to give this password to the user to sign in for the first time.
4. Select **Create**.
5. Launch a **New InPrivate Window** browser session.
6. Test new user in Azure portal and change password:
  1. Navigate to the Azure portal, <https://portal.azure.com> in the InPrivate browser session.
  2. Sign in with your new *testuser@contosomarketingxx.onmicrosoft.com* account.
  3. You will be prompted to change the password. Paste the auto-generated password you copied and paste into the **Current password** field.
  4. Set the password as desired.
  5. Save the username and password so you will remember the credentials for using it later in this lab.

## Task 4: Register an application

In this task, you will switch Active Directory tenants and then continue with registering an application into the default

### Switch Azure AD Directory

1. Select the Directory + subscription icon.

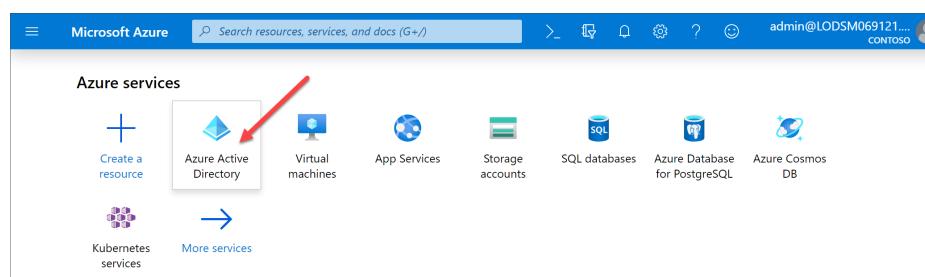


2. Select Contoso.

The screenshot shows the Azure portal interface for the 'Contoso Marketing Company - Overview' page. On the left, there's a navigation pane with options like 'Overview', 'Getting started', and 'Diagnose and solve problems'. Below that is a 'Manage' section with links for 'Users', 'Groups', 'Organizational relationships', 'Roles and administrators', 'Enterprise applications', 'Devices', 'App registrations', 'Identity Governance', and 'Application proxy'. The main area displays 'Sign-ins' data from November 3 to November 24, with a count of 100. A 'What's new in Azure AD' section at the bottom encourages users to stay up-to-date with release notes and blog posts. On the right, a 'Directory + subscription' blade is open, showing a 'Default subscription filter' message and a 'Current directory' set to 'contosomarketingms01.onmicrosoft.com'. It also lists 'Favorites' and 'All Directories' with a search bar. A red arrow points to the 'Contoso' entry in the 'All Directories' list.

3. You should be redirected to the Active Directory page for Contoso. If it redirected you to the default Azure portal landing page, then follow the steps below to navigate back to Azure Active Directory:

1. You will be redirected to the Azure portal landing page where you should now see an Azure Active Directory button available.
2. Select the **Azure Active Directory** icon. If you do not have the icon: expand the left navigation pane, select **Azure Active Directory**.



### Register a new application

1. On the left menu, navigate to **App registrations**.
2. Select **New Registration**.
3. From the **Register an application** pane, perform the following actions:
  1. In the **Name** text box, enter the value **ContosoApp**.
  2. In the **Supported account type** section, select **Accounts in this organizational directory only**.
  3. Leave the **Redirect URI (optional)** as blank.

## Register an application

\* Name

The user-facing display name for this application (this can be changed later).

### Supported account types

Who can use this application or access this API?

- Accounts in this organizational directory only (Contoso only - Single tenant)  
 Accounts in any organizational directory (Any Azure AD directory - Multitenant)  
 Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)  
 Personal Microsoft accounts only

[Help me choose...](#)

### Redirect URI (optional)

We'll return the authentication response to this URI after successfully authenticating the user. Providing this now is optional and it can be changed later, but a value is required for most authentication scenarios.

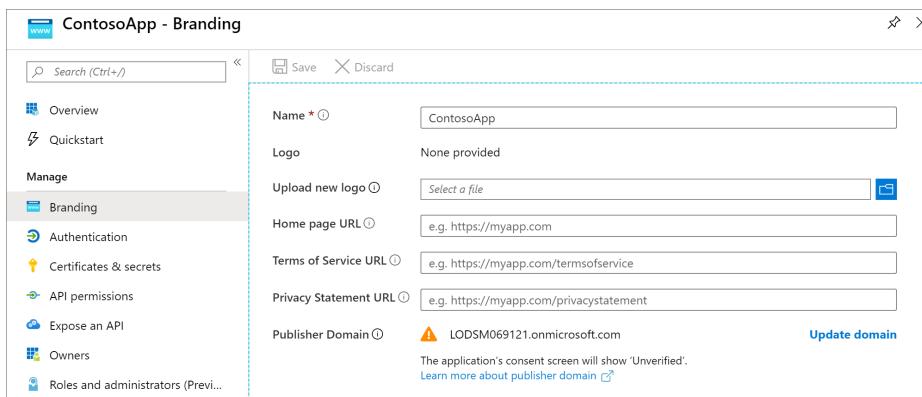
[By proceeding, you agree to the Microsoft Platform Policies](#) 

**Register**

4. Select **Register**. The application overview page is displayed.

## Task 5: Branding of the application

1. On the left menu, navigate to **Branding**.
2. Verify that you can personalize branding by providing a logo, home page URL, terms of service URL, and a privacy statement URL.



3. This exercise is now complete.

## Review

In this exercise, you learned how to create a new Active Directory organization, create a new user, switch directories, register and brand the application.

### ❖❖❖# Exercise 2: Implementing authentication

This exercise will demonstrate the different account types that are used within the Microsoft identity platform.

**Note:** This exercise demonstrates signing into a web application using two different accounts. These two accounts will come from two organizations, one of them being the organization where the Azure AD application is registered. Therefore, in order to complete the exercise, you'll need access to two user accounts in different Azure AD directories.

# Task 1: Create application that only allows single organization sign in

In this task, you will register an application in the Azure portal that allows users from the current organization to sign in.

## Register a single-tenant Azure AD application

1. From the Azure portal <https://portal.azure.com>, navigate to **Azure Active Directory**.
2. Select **Manage > App registrations** in the left-hand navigation.
3. On the **App registrations** page, select **New registration**.
4. On the **Register an application** page, set the values as follows:
  - **Name:** Hello ASPNET Core Identity 01
  - **Supported account types:** Accounts in this organizational directory only (Single tenant)

### Register an application

\* Name  
The user-facing display name for this application (this can be changed later).

Hello ASPNET Core Identity 01 ✓

Supported account types

Who can use this application or access this API?

Accounts in this organizational directory only (Contoso only - Single tenant)  
 Accounts in any organizational directory (Any Azure AD directory - Multitenant)  
 Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)  
 Personal Microsoft accounts only

[Help me choose...](#)

Redirect URI (optional)  
We'll return the authentication response to this URI after successfully authenticating the user. Providing this now is optional and it can be changed later, but a value is required for most authentication scenarios.

Web e.g. https://myapp.com/auth

By proceeding, you agree to the Microsoft Platform Policies [↗](#)

**Register**

5. Select **Register** to create the application.
6. On the **Hello ASPNET Core Identity 01** page, copy the values **Application (client) ID** and **Directory (tenant) ID**; you'll need these values later in this exercise.

The screenshot shows the Azure Active Directory admin center interface. On the left, there's a navigation sidebar with options like Overview, Quickstart, Manage, API permissions, Expose an API, Owners, Roles and administrators, Manifest, Support + Troubleshooting, Troubleshooting, and New support request. The main content area is titled "Hello ASP.NET Core Identity 01". It includes sections for Delete, Endpoints, Display name (Hello ASP.NET Core Identity 01), Application (client) ID (7ae7dadd-3744-4e57-959a-4e16fdf4d4be), Directory (tenant) ID (7ade07e0-76e4-4ef6-91d2-2aa01ba662a7), Object ID (8eb03e32-c65c-4fbf-8c87-3bc3f1d41407), Supported account types (My organization only), Redirect URLs (Add a Redirect URI), Application ID URI (Add an Application ID URI), and Managed application in local directory (Hello ASP.NET Core Identity 01). A welcome message at the bottom says "Welcome to the new and improved App registrations. Looking to learn how it's changed from App registrations (Legacy)? Learn more". Below that are sections for Call APIs (with icons for various Microsoft services) and Documentation (links to Microsoft identity platform, Authentication scenarios, Authentication libraries, Code samples, Microsoft Graph, Glossary, and Help and Support). At the bottom, there's a "View API permissions" button and a "Sign in users in 5 minutes" link.

7. On the **Hello ASPNET Core Identity 01** page, select the **Add a Redirect URI** link under the **Redirect URIs**.
8. Locate the section **Redirect URIs** and add the following two URLs:
  - **https://localhost:3007**
  - **https://localhost:3007/signin-oidc**
9. Add the following **Logout URL:** **https://localhost:3007/signout-oidc**
10. Locate the section **Implicit grant** and select both **Access tokens** and **ID tokens**. This tells Azure AD to return these tokens the authenticated user if requested.
11. Select **Save** when finished setting these values.

## Hello ASP.NET Core Identity 01 | Authentication

Search (Ctrl+ /) Save Discard Got feedback?

Overview Quickstart Integration assistant | Preview

Manage

- Branding
- Authentication **Selected**
- Certificates & secrets
- Token configuration
- API permissions
- Expose an API
- Owners
- Roles and administrators | Preview
- Manifest

Support + Troubleshooting

- Troubleshooting
- New support request

**Platform configurations**

Depending on the platform or device this application is targeting, additional configuration may be required such as redirect URIs, specific authentication settings, or fields specific to the platform.

+ Add a platform

**Web** Quickstart Docs

**Redirect URLs**

The URLs we will accept as destinations when returning authentication responses (tokens) after successfully authenticating users. Also referred to as reply URLs. [Learn more about Redirect URLs and their restrictions](#)

`https://localhost:3007/signin-oidc`

`https://localhost:3007`

[Add URI](#)

**Logout URL**

This is where we send a request to have the application clear the user's session data. This is required for single sign-out to work correctly.

`https://localhost:3007/signout-oidc`

**Implicit grant**

Allows an application to request a token directly from the authorization endpoint. Checking Access tokens and ID tokens is recommended only if the application has a single-page architecture (SPA), has no back-end components, does not use the latest version of MSAL.js with auth code flow, or it invokes a web API via JavaScript. ID Token is needed for ASP.NET Core Web Apps. [Learn more about the implicit grant flow](#)

To enable the implicit grant flow, select the tokens you would like to be issued by the authorization endpoint:

Access tokens  ID tokens

## Task 2: Create a single organization ASP.NET core web application

In this first application, you'll create an ASP.NET Core web application that allows users from the current organization to sign in and display their information. 1. Open your command prompt, navigate to a directory where you want to save your work, create a new folder, and change directory into that folder. For example:

```
```powershell
Cd c:/LabFiles
md SingleOrg
cd SingleOrg
````
```

1. Execute the following command to create a new ASP.NET Core MVC web application:

```
powerShell dotnet new mvc --auth SingleOrg
```

2. Open the folder in Visual Studio code by executing from the project directory: `code .`

### Configure the web application with the Azure AD application you created

1. Locate and open the `./appsettings.json` file in the ASP.NET Core project.
2. Set the **AzureAd.Domain** property to the domain of your Azure AD tenant where you created the Azure AD application (*for example: contoso.onmicrosoft.com*).
3. Set the **AzureAd.TenantId** property to the **Directory (tenant) ID** you copied when creating the Azure AD application in the previous step.
4. Set the **AzureAd.ClientId** property to the **Application (client) ID** you copied when creating the Azure AD application in the previous step.

### Update the web application's launch configuration

1. Locate and open the `./Properties/launchSettings.json` file in the ASP.NET Core project.
2. Set the **iisSettings.iisExpress.applicationUrl** property to **https://localhost:3007**.
3. Set the **iisSettings.iisExpress.sslPort** property to **3007**.

### Update the user experience

1. Finally, update the user experience of the web application to display all the claims in the OpenID Connect ID token.
2. Locate and open the `./Views/Home/Index.cshtml` file.
3. Add the following code to the end of the file:

```
powerShell @if (User.Identity.IsAuthenticated) { <div> <table
cellpadding="2" cellspacing="2"> <tr> <th>Claim</th> <th>Value</th> </tr>
@foreach (var claim in User.Claims) { <tr> <td>@claim.Type</td>
<td>@claim.Value</td> </tr> } </table> </div> }
```

## Task 3: Build and test the single organization web app

1. Execute the following command in a command prompt to compile and run the application:

```
powershell dotnet build dotnet run
```

2. Open a browser and navigate to the url **https://localhost:5001**. The web application will redirect you to the Azure AD sign in page.

3. Sign in using a Work and School account from your Azure AD directory. Azure AD will redirect you back to the web application. Notice some of the details from the claims included in the ID token.

The screenshot shows the 'msidentity\_aspnet' web application's 'Welcome' page. At the top, there are links for 'Home' and 'Privacy'. On the right, it shows a greeting: 'Hello admin@M365x068225.onmicrosoft.com! Sign out'. Below the greeting, the word 'Welcome' is displayed in large letters. Underneath 'Welcome', there is a link to 'Learn about building Web apps with ASP.NET Core.' The main content area displays a table of 'Claims' and their corresponding 'Values'. Some specific claims are highlighted with red boxes:

Claim	Value
aio	42VgYEil1RUsNu7XvPw3OaMl++6Wi2zbcj9Vl9wqmadxfGKn8lUA
http://schemas.microsoft.com/claims/authnmethodsreferences	pwd
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/surname	Administrator
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname	MOD
name	MOD Administrator
http://schemas.microsoft.com/identity/claims/objectidentifier	b9dd14c5-1943-448d-a4bf-70bd0ccd2592
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier	vVecxWIDJLUz0tVpBSCZdCP-nkPkAdQKJfg5kahg
http://schemas.microsoft.com/identity/claims/tenantid	7a4e07e0-76e4-4ef6-91d2-2aa01ba662a7
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name	admin@M365x068225.onmicrosoft.com
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/upn	admin@M365x068225.onmicrosoft.com
uti	4lxz2p4z80imXz4F6MA0AQ

At the bottom left, there is a copyright notice: '© 2019 - msidentity\_aspnet - [Privacy](#)'.

4. Take special note of the **tenantid** and **upn** claim. These claims indicate the ID of the Azure AD directory and ID of the user that signed in. Make a note of these values to compare them to other options in a minute.
5. Now try logging in as a user from a different organization. Select the **Sign out** link in the top left. Wait for Azure AD and the web application signs out the current user. When the web application reloads, repeat the sign in process, except this time try signing in as a user from a different organization or use a Microsoft Account.
6. Notice Azure AD will reject the user's sign in, explaining that the user's account doesn't exist in the current tenant.
7. Stop the web server by pressing **CTRL+C** in the command prompt.

## Task 4: Create application that allows any organization's users to sign in

In this task, you will register an application in the Azure portal that allows users from any organization or Microsoft Accounts to sign in.

### Register a multi-tenant Azure AD application

1. From the Azure portal <https://portal.azure.com>, navigate to **Azure Active Directory**.
2. Select **Manage > App registrations** in the left-hand navigation.
3. On the **App registrations** page, select **New registration**.
4. On the **Register an application** page, set the values as follows:
  - **Name:** Hello ASPNET Core Identity 02
  - **Supported account types:** Accounts in any organizational directory only (Any Azure AD directory - Multitenant)

#### Register an application

\* Name  
The user-facing display name for this application (this can be changed later).  
   ✓

Supported account types  
Who can use this application or access this API?  
 Accounts in this organizational directory only (Contoso only - Single tenant)  
 Accounts in any organizational directory (Any Azure AD directory - Multitenant)  
 Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)  
 Personal Microsoft accounts only  
[Help me choose...](#)

Redirect URI (optional)  
We'll return the authentication response to this URI after successfully authenticating the user. Providing this now is optional and it can be changed later, but a value is required for most authentication scenarios.

By proceeding, you agree to the Microsoft Platform Policies [\[?\]](#)

5. Select **Register** to create the application.
6. On the **Hello ASPNET Core Identity 02** page, copy the values **Application (client) ID** and **Directory (tenant) ID**; you'll need these values later in this exercise.
7. On the **Hello ASPNET Core Identity 02** page, select the **Add a Redirect URI** link under the **Redirect URIs**.
8. Locate the section **Redirect URIs** and add the following two URLs:

- <https://localhost:3007>
- <https://localhost:3007/signin-oidc>

9. Add the following **Logout URL**: <https://localhost:3007/signout-oidc>

10. Locate the section **Implicit grant** and select both **Access tokens** and **ID tokens**. This tells Azure AD to return these tokens the authenticated user if requested.

11. Select **Save** when finished setting these values.

The screenshot shows the Azure portal interface for managing an application named "Hello ASPNET Core Identity 02". The left sidebar lists various management sections like Overview, Quickstart, Integration assistant, Manage, Branding, Authentication, Certificates & secrets, Token configuration, API permissions, Expose an API, Owners, Roles and administrators, and Manifest. The Authentication section is currently selected. The main content area is titled "Platform configurations" and specifically targets a "Web" application. It contains sections for "Redirect URLs" and "Logout URL". Under "Redirect URLs", there are two entries: "https://localhost:3007/signin-oidc" and "https://localhost:3007". Below these is a "Add URI" button. Under "Logout URL", the value "https://localhost:3007/signout-oidc" is entered. At the bottom, the "Implicit grant" section is expanded, showing options for issuing tokens. Both "Access tokens" and "ID tokens" are checked, indicating they will be issued by the authorization endpoint.

## Task 5: Create a multiple organization ASP.NET core web application

In this second application, you'll create an ASP.NET Core web application that allows users from any organization or Microsoft Accounts to sign in and display their information.

1. Open your command prompt, navigate to a directory where you want to save your work, create a new folder, and change directory into that folder. For example:

```
powershell Cd c:/LabFiles md MultiOrg cd MultiOrg
```

2. Execute the following command to create a new ASP.NET Core MVC web application:

```
powershell dotnet new mvc --auth MultiOrg
```

3. Open the folder in Visual Studio code by executing from the project directory: `code .`

### Configure the web application with the Azure AD application you created

1. Locate and open the `./appsettings.json` file in the ASP.NET Core project.
2. Set the **AzureAd.ClientId** property to the **Application (client) ID** you copied when creating the Azure AD application in the previous step.

### Update the web application's launch configuration

1. Locate and open the `./Properties/launchSettings.json` file in the ASP.NET Core project.
2. Set the **iisSettings.iisExpress.applicationUrl** property to **https://localhost:3007**.
3. Set the **iisSettings.iisExpress.sslPort** property to **3007**.

### Update the user experience

1. Finally, update the user experience of the web application to display all the claims in the OpenID Connect ID token.
2. Locate and open the `./Views/Home/Index.cshtml` file.
3. Add the following code to the end of the file:

```
powershell @if (User.Identity.IsAuthenticated) { <div> <table cellpadding="2" cellspacing="2"> <tr> <th>Claim</th> <th>Value</th> </tr> @foreach (var claim in User.Claims) { <tr> <td>@claim.Type</td> <td>@claim.Value</td> </tr> } </table> </div> }
```

4. Save the above all changes.

## Task 6: Build and test the multiple organization web app

1. Execute the following command in a command prompt to compile and run the application:

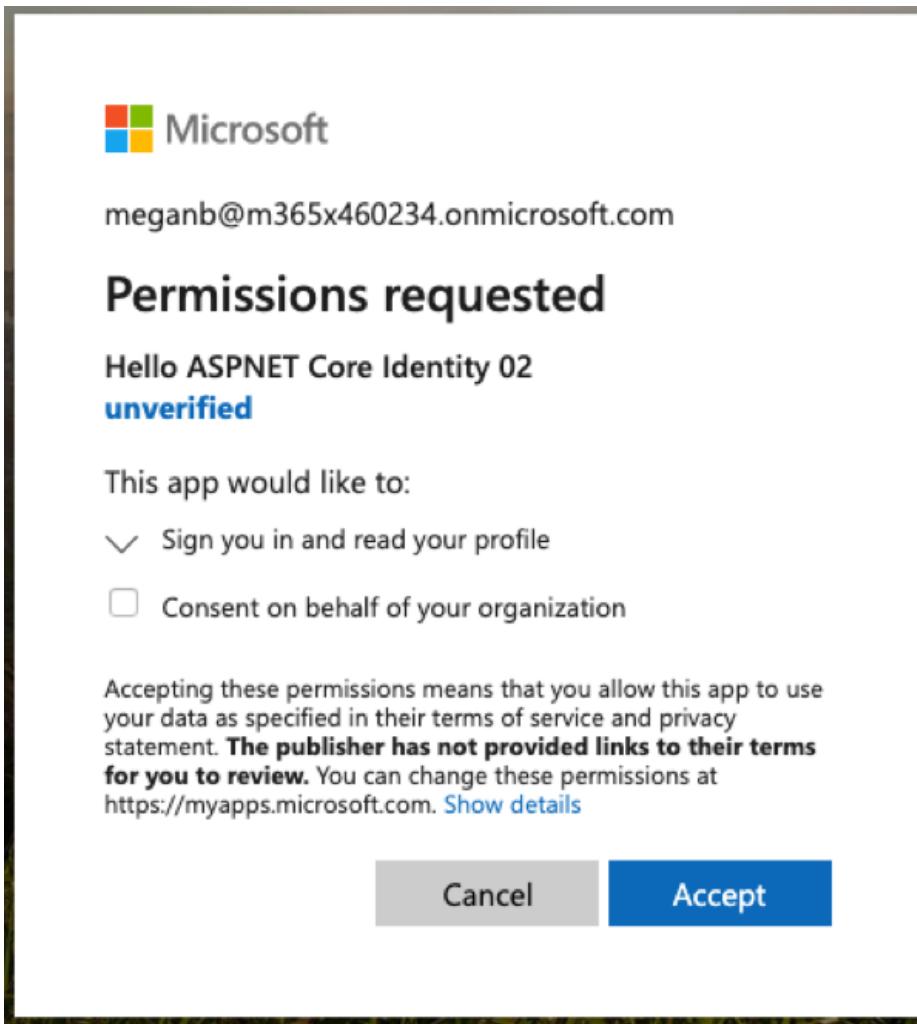
```
powershell dotnet build dotnet run
```

2. Open a browser and navigate to the url **https://localhost:5001**. The web application will redirect you to the Azure AD sign in page.

3. Sign in using a Work and School account from your Azure AD directory. Azure AD will redirect you back to the web application. Notice some of the details from the claims included in the ID token.

Claim	Value
aio	42VgYAjWVLryOnZ3uJm42RejefkH88Ua6qbKmd9KeGW09NCeL/YA
<a href="http://schemas.microsoft.com/claims/authnmethodsreference">http://schemas.microsoft.com/claims/authnmethodsreference</a>	pwd
<a href="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/surname">http://schemas.xmlsoap.org/ws/2005/05/identity/claims/surname</a>	Administrator
<a href="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname">http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname</a>	MOD
name	MOD Administrator
<a href="http://schemas.microsoft.com/identity/claims/objectidentifier">http://schemas.microsoft.com/identity/claims/objectidentifier</a>	b9dd14c5-1943-448d-a4bf-70bd0cccd2592
<a href="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier">http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier</a>	w6wbkCD8rgZXW7e56JP303z0FMhs1Ckf_rdjg3k3p_k
<a href="http://schemas.microsoft.com/identity/claims/tenantid">http://schemas.microsoft.com/identity/claims/tenantid</a>	7a4e07e0-76e4-4ef6-91d2-2aa01ba662a7
<a href="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name">http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name</a>	admin@M365x068225.onmicrosoft.com
<a href="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/upn">http://schemas.xmlsoap.org/ws/2005/05/identity/claims/upn</a>	admin@M365x068225.onmicrosoft.com
uti	NCupKOCExEWF2XarXO0YAQ

4. Take special note of the **tenantid** and **upn** claim. These indicate the ID of the Azure AD directory and ID of the user that signed in. Make a note of these values to compare them to other options in a minute.
5. Now try logging in as a user from a different organization. Select the **Sign out** link in the top left. Wait for Azure AD and the web application signs out the current user.
6. This time, the user is prompted to first trust the application:



7. Select **Accept**.

8. Notice the web application's page loads with different claims, specifically for the **tenantid** and **upn** claim. This indicates the user is not from the current directory where the Azure AD application is registered:

The screenshot shows a "Welcome" page for a web application. At the top right is the user information "Hello MeganB@M365x460234.OnMicrosoft.com! Sign out". Below it is a table of user claims:

Claim	Value
aio	ASQA2/8NAAAChNjQkG9ys/L0reUM5XBGGw7tfpe7n4xapXD3xNF7w=
http://schemas.microsoft.com/claims/authnmethodsreferences	pwd
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/surname	Bowen
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname	Megan
name	Megan Bowen
http://schemas.microsoft.com/identity/claims/objectidentifier	b8c8446f-f788-4527-8cf6-c40e65228539
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier	C2fyWGqB8gBmb0vJxSTCMaJORzef7fKW0hpBG52hDk
http://schemas.microsoft.com/identity/claims/tenantid	7adff0e7-07b3-4eb5-ba39-7ea421035371
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name	MeganB@M365x460234.OnMicrosoft.com
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/upn	MeganB@M365x460234.OnMicrosoft.com
uti	B1nQq0avIEaL40SbngADAA

9. Stop the web server by going back to the running project in Visual Studio Code. From the Terminal, press **CTRL+C** in the command prompt.

## Review

In this exercise, you learned how to create different types of Azure AD applications and use an ASP.NET Core application to support the different sign in options that support different types of accounts.

### ❖❖❖# Exercise 3: Implementing application that supports B2B

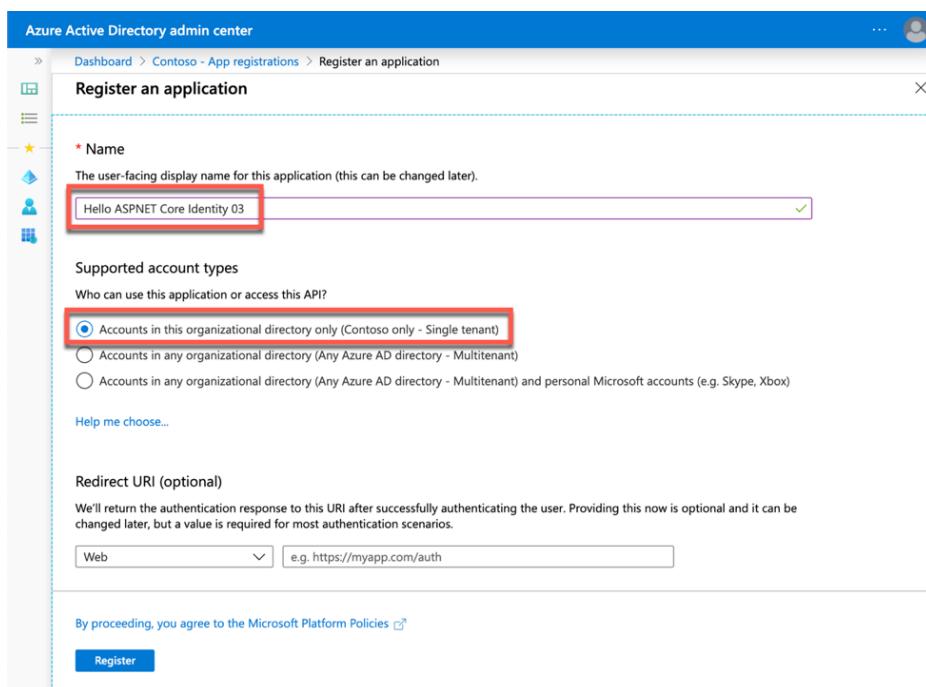
This exercise will demonstrate to the user how to configure and implement an application that supports B2B. **Note:** This exercise demonstrates signing into a web application using three different accounts. These three accounts will come from two organizations, one of them being the organization where the Azure AD application is registered. Therefore, in order to complete the exercise, you'll need access to two user accounts in different Azure AD directories.

# Task 1: Create a single-tenant Azure AD application

In this task, you'll create an Azure AD application that allows users from the current organization to sign in.

## Register a single-tenant Azure AD application

1. From the Azure portal <https://azure.portal.com>, navigate to **Azure Active Directory**.
2. Select **Manage > App registrations** in the left-hand navigation.
3. On the **App registrations** page, select **New registration**.
4. On the **Register an application** page, set the values as follows:
  - **Name:** Hello ASPNET Core Identity 03
  - **Supported account types:** Accounts in this organizational directory only (Single tenant)



5. Select **Register** to create the application.
6. On the **Hello ASPNET Core Identity 03** page, copy the values **Application (client) ID** and **Directory (tenant) ID**; you'll need these values later in this exercise.
7. On the **Hello ASPNET Core Identity 03** page, select the **Add a Redirect URI** link under the **Redirect URIs**.
8. Locate the section **Redirect URIs** and add the following two URLs:
  - **https://localhost:3007**

- <https://localhost:3007/signin-oidc>

9. Add the following **Logout URL:** <https://localhost:3007/signout-oidc>

10. Locate the section **Implicit grant** and select both **Access tokens** and **ID tokens**. This tells Azure AD to return these tokens the authenticated user if requested.

11. Select **Save** when finished setting these values.

## Task 2: Create a single organization ASP.NET core web application

In this application, you'll create an ASP.NET Core web application that allows users from the current organization to sign in and display their information.

1. Open your command prompt, navigate to a directory where you want to save your work, create a new folder, and change directory into that folder. For example:

```
powershell Cd c:/LabFiles md SingleOrgGuests cd SingleOrgGuests
```

2. Execute the following command to create a new ASP.NET Core MVC web application:

```
powershell dotnet new mvc --auth SingleOrg
```

3. Open the folder in Visual Studio code by executing from the project directory: **code .**

### Configure the web application with the Azure AD application you created

1. Locate and open the **./appsettings.json** file in the ASP.NET Core project.
2. Set the **AzureAd.Domain** property to the domain of your Azure AD tenant where you created the Azure AD application (*for example: contoso.onmicrosoft.com*).
3. Set the **AzureAd.TenantId** property to the **Directory (tenant) ID** you copied when creating the Azure AD application in the previous step.
4. Set the **AzureAd.ClientId** property to the **Application (client) ID** you copied when creating the Azure AD application in the previous step.

### Update the web application's launch configuration

1. Locate and open the **./Properties/launchSettings.json** file in the ASP.NET Core project.
2. Set the **iisSettings.iisExpress.applicationUrl** property to **https://localhost:3007**.
3. Set the **iisSettings.iisExpress.sslPort** property to **3007**.

### Update the user experience

1. Finally, update the user experience of the web application to display all the claims in the OpenID Connect ID token.
2. Locate and open the **./Views/Home/Index.cshtml** file.
3. Add the following code to the end of the file:

```
powershell @if (User.Identity.IsAuthenticated) { <div> <table cellpadding="2" cellspacing="2"> <tr> <th>Claim</th> <th>Value</th> </tr> @foreach (var claim in User.Claims) { <tr> <td>@claim.Type</td> <td>@claim.Value</td> </tr> } </table> </div> }
```

4. Save the above all changes.

## Task 3: Build and test the app

1. Execute the following command in a command prompt to compile and run the application:

```
powershell dotnet build dotnet run
```

2. Open a browser and navigate to the url **https://localhost:5001**. The web application will redirect you to the Azure AD sign in page.

3. Sign in using a Work and School account from your Azure AD directory. Azure AD will redirect you back to the web application. Notice some of the details from the claims included in the ID token.

The screenshot shows the 'msidentity\_aspnet' application's welcome screen. At the top, there is a navigation bar with 'msidentity\_aspnet', 'Home', and 'Privacy' links. To the right, it shows the user 'Hello admin@M365x068225.onmicrosoft.com!' and a 'Sign out' link. Below the navigation bar, the word 'Welcome' is centered. Underneath 'Welcome', there is a link to 'Learn about building Web apps with ASP.NET Core.' A table titled 'Claims' is displayed, listing various claims and their values. Some specific claims are highlighted with red boxes: 'http://schemas.microsoft.com/claims/authnmethodsreferences', 'http://schemas.xmlsoap.org/ws/2005/05/identity/claims/surname', 'http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname', 'http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name', 'http://schemas.microsoft.com/identity/claims/objectidentifier', 'http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier', 'http://schemas.microsoft.com/identity/claims/tenantid', 'http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name', and 'http://schemas.xmlsoap.org/ws/2005/05/identity/claims/upn'. The 'value' column for these highlighted claims includes the string '7a4e07e0-76e4-4ef6-91d2-2aa01ba662a7'.

Claim	Value
aio	42VgYEiL1RUsNu7XvPw3OaMl++6Wi2zbcj9Vl9wqmadxfGKn8lUA
http://schemas.microsoft.com/claims/authnmethodsreferences	pwd
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/surname	Administrator
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname	MOD
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name	MOD Administrator
http://schemas.microsoft.com/identity/claims/objectidentifier	b9dd14c5-1943-448d-a4bf-70bd0ccd2592
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier	vVecxWIDJLUz0tVpBSCZdCP-nkPkAdQKJfg5kahg
http://schemas.microsoft.com/identity/claims/tenantid	7a4e07e0-76e4-4ef6-91d2-2aa01ba662a7
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name	admin@M365x068225.onmicrosoft.com
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/upn	admin@M365x068225.onmicrosoft.com
uti	4lxz2p4z80imXz4F6MA0AQ

4. Take special note of the **tenantid** and **upn** claim. These claims indicate the ID of the Azure AD directory and ID of the user that signed in. Make a note of these values to compare them to other options in a minute.
5. Now try logging in as a user from a different organization. Select the **Sign out** link in the top left. Wait for Azure AD and the web application signs out the current user. When the web application reloads, repeat the sign in process, except this time try signing in as a user from a different organization or use a Microsoft Account.
6. Notice Azure AD will reject the user's sign in, explaining that the user's account doesn't exist in the current tenant.



## Sign in

Sorry, but we're having trouble signing you in.

AADSTS50020: User account 'MeganB@M365x460234.OnMicrosoft.com' from identity provider 'https://sts.windows.net/7adf0e7f-07b3-4eb5-ba39-7ea421035371/' does not exist in tenant '████████' and cannot access the application '0b841307-928d-400d-854d-37aaa2b06890'(Hello ASP.NET Core Identity 01) in that tenant. The account needs to be added as an external user in the tenant first. Sign out and sign in again with a different Azure Active Directory user account.

Before this user can access this application, they need to be added as a guest into the Azure AD directory where the application was registered. Proceed with the next task to invite a guest user to your organization.

## Task 4: Invite a guest user from another organization

In this task, you will configure your Active Directory tenant to allow external users and then will invite a guest user from another Active Directory organization.

1. From the Azure portal, navigate to **Azure Active Directory**.
2. In the left-hand navigation, select **User**.
3. Examine the external user settings for available to administrators by selecting **User Settings** and then **Manage external collaboration settings**.

The screenshot shows the Azure Active Directory admin center interface. On the left, there's a navigation sidebar with options like All users, Deleted users, Password reset, and User settings (which is highlighted with a red box). The main content area has sections for Enterprise applications, App registrations, Administration portal, LinkedIn account connections, and External users. The External users section (also highlighted with a red box) contains a link to 'Manage external collaboration settings'. At the top right, there are Save and Discard buttons.

4. Notice that administrators can configure the Azure AD directory so guest users have limited rights compared to other users, and who can invite guest users.

External collaboration settings X

Save Discard

Guest user access

Guest user access restrictions (Preview) ⓘ

[Learn more](#)

Guest users have the same access as members (most inclusive)

Guest users have limited access to properties and memberships of directory objects

Guest user access is restricted to properties and memberships of their own directory objects (most restrictive)

Guest invite settings

Admins and users in the guest inviter role can invite ⓘ

Yes  No

Members can invite ⓘ

Yes  No

Guests can invite ⓘ

Yes  No

Enable Email One-Time Passcode for guests (Preview) ⓘ

[Learn more](#)

Yes  No

Enable guest self-service sign up via user flows (Preview) ⓘ

[Learn more](#)

Yes  No

Collaboration restrictions

Allow invitations to be sent to any domain (most inclusive)

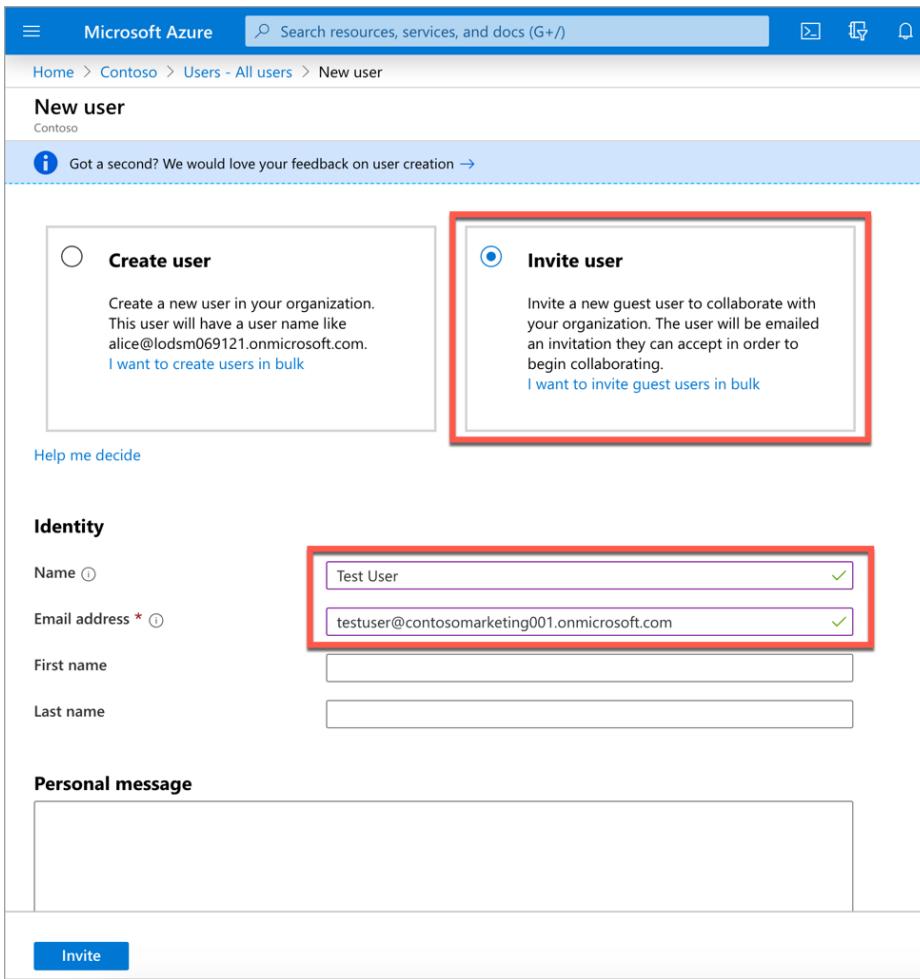
Deny invitations to the specified domains

Allow invitations only to the specified domains (most restrictive)

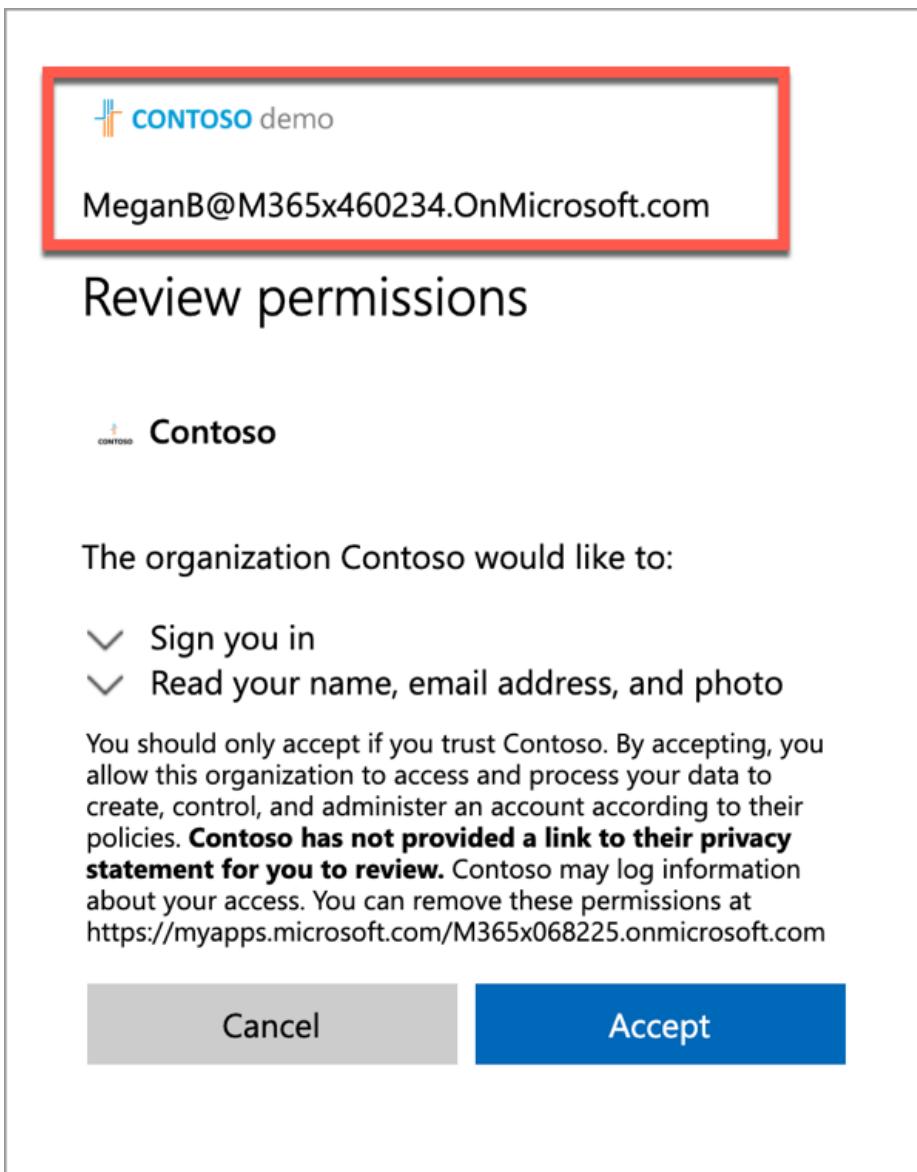
5. Now let's invite a guest user. Select **All users** in the left-hand navigation, and then select **New guest user**:

6. On the **New user** page, select **Invite user** and then add the guest user's information:

- **Name:** Test User
- **Email address:** Type the email address of the new user you created in Exercise 1 Task 3. *For example: testuser@contosomarketing001.onmicrosoft.com*



7. Select **Invite** to automatically send the invitation to the guest user.
8. Now let's try to sign in with the user. In the browser, navigate to **<https://localhost:5001>**.
9. This time, after successfully logging in, the user's Azure AD directory will prompt the user to grant the application's Azure AD directory permissions to sign in as the user and obtain basic information about the user.



10. Take note of what is happening at this point. The original Azure AD directory is not signing in the user, rather the user has been redirected to sign in with their Azure AD directory. Once they sign in, their Azure AD directory will provide a token to our directory that is used to verify the user is authenticated and authorized the application to obtain their basic profile information. It then creates a new access token that can be used by our ASP.NET Core web application.
11. After selecting **Accept**, the user is taken to our ASP.NET application. Notice the difference in some of the claims.
  - The **identityprovider** claim is the ID of the Azure AD directory that authenticated the user. This claim is the user's Azure AD directory
  - The **tenantid** claim is the ID of the Azure AD directory our application is registered in. Notice this value is not the same as the **identityprovider** claim, indicating the user's identity is in one directory while they have been added as a guest user to another Azure AD directory.

# Welcome

Learn about [building Web apps with ASP.NET Core](#).

Claim	Value
aio	AUQAU/8NAAAAA3JX+tQN94Hx5t88Cj4SAwrTpWnxgDyliF/tfWqZFW+LT2
http://schemas.microsoft.com/claims/authnmethodsreferences	pwd
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/emailaddress	MeganB@M365x460234.onmicrosoft.com
http://schemas.microsoft.com/identity/claims/identityprovider	https://sts.windows.net/7adfd0e7f-07b3-4eb5-ba39-7ea421035371/
name	Megan Bowen
http://schemas.microsoft.com/identity/claims/objectidentifier	a028b9b0-1ae8-4f59-8d3b-5c07eef0e01f
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier	Lx2U6RdlH1de6a0Y8HyWA-N8oOAE7_IWKNNeTm8_4LI
http://schemas.microsoft.com/identity/claims/tenantid	7a4e07e0-76e4-4ef6-91d2-2aa01ba662a7
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name	MeganB@M365x460234.onmicrosoft.com
uti	bZluluIlyF0eHp6MV8osEAA

12. Stop the web server by going back to the running project in Visual Studio Code. From the Terminal, press **CTRL+C** in the command prompt.

## Review

In this exercise, you created an ASP.NET Core web application and Azure AD application that

allows guest users from partner Azure AD directories to sign in and access the application. You

then invited a guest user to the directory and signed into the application with this user. ♦♦♦#

Exercise 4: Configuring permissions to consume an API

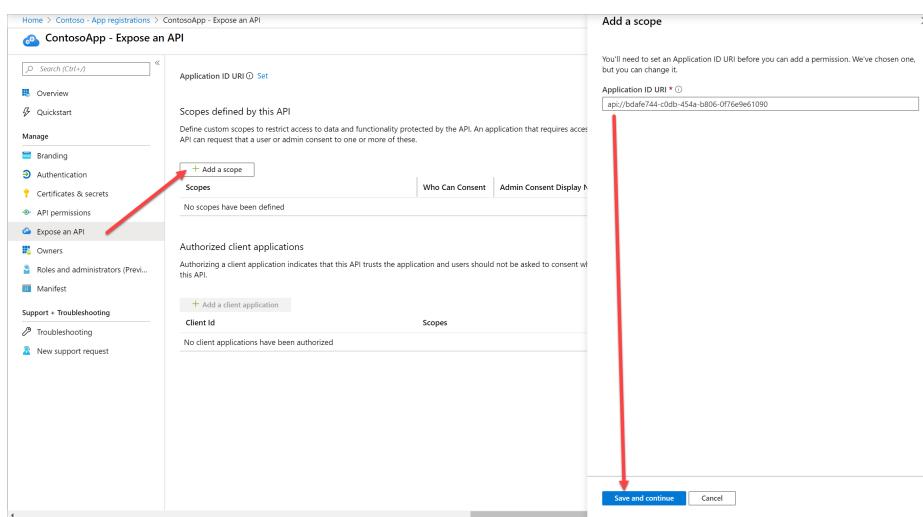
In this exercise, you'll learn how to configure an application to expose a new scope to make it available to client applications.

## **Task 1: Navigate to an application to configure**

1. In the left-hand navigation pane of the Azure portal, select the **Azure Active Directory** service and then select **App registrations**.
2. Find and select the application you want to configure such as **ContosoApp**. Once you've selected the app, you'll see the application's **Overview** or main registration page.
3. Navigate to **Expose an API**.

## Task 2: Add a scope

1. In the **Scopes defined by this API** section, select **Add a scope**.
2. If you have not set an **Application ID URI**, you will see a prompt to enter one. Enter your application ID URI or use the one provided and then select **Save and continue**.



3. When the **Add a scope** page appears, enter your scope's information:
  1. In the **Scope name** textbox, enter a meaningful name for your scope. For example, **Employees.Read.All**.
  2. For **Who can consent**, select whether this scope can be consented to by users, or if admin consent is required. Select **Admins only** for higher-privileged permissions.
  3. Provide a user-friendly **Admin consent display name** and **Admin consent description**.
  4. Provide a user-friendly **User consent display name** and **User consent description**.
  5. Set the **State** and select **Add scope** when you're done.
  6. Your Add a Scope pane should look similar to the image below.

**Add a scope**

---

Scope name \* ⓘ  
 ✓  
 api://bdafe744-c0db-454a-b806-0f76e9e61090/Employees.Read.All

Who can consent? ⓘ  
 Admins and users    Admins only

Admin consent display name \* ⓘ  
 ✓

Admin consent description \* ⓘ  
 ✓

User consent display name ⓘ  
 ✓

User consent description ⓘ

State ⓘ  
 Enabled    Disabled

## Expose a new scope or role through the application manifest

Now that you learned how to create a new scope using the UI, next you will add a new scope through the application manifest.

1. In the left navigation menu of your app, select **Manifest**.
2. A web-based manifest editor opens, allowing you to **Edit** the manifest within the portal. Optionally, you can select **Download** and edit the manifest locally, and then use **Upload** to reapply it to your application.
3. Scroll down until you find the **oauth2Permissions** collection.
4. Below is the JSON element that was added to the oauth2Permissions collection.

```
json { "adminConsentDescription": "Allow the application to have read-only access to all Employee data.", "adminConsentDisplayName": "Read-only access to Employee records", "id": "ede407eb-3c3a-4918-9e2e-803d2298e247", "isEnabled": true, "type": "User", "userConsentDescription": "Allow the application to have read-only access to your Employee data.", "userConsentDisplayName": "Read-only access to your Employee records", "value": "Employees.Read.All" }
```

**Note:** The id value must be generated programmatically or by using a GUID generation tool such as guidgen. The id represents a unique identifier for the scope as exposed by the web API. Once a client is appropriately configured with permissions to access your web API, it is issued an OAuth 2.0 access token by Azure AD. When the client calls the web

API, it presents the access token that has the scope (scp) claim set to the permissions requested in its application registration.

You can expose additional scopes later as necessary. Consider that your web API might expose multiple scopes associated with a variety of different functions. Your resource can control access to the web API at runtime by evaluating the scope (scp) claim(s) in the received OAuth 2.0 access token.

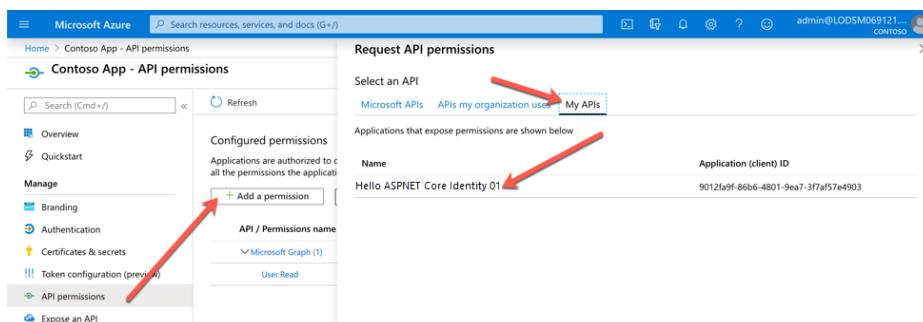
## Task 3: Add a client application and API permissions

### Add permissions scope to Hello ASPNET Core Identity 01

1. Navigate back to **App registrations**.
2. Select the ContosoApp you created earlier in this lab and copy the **Application (client) ID**.
3. Navigate back to the **Overview** page for your **Hello ASPNET Core Identity 01** app.
4. Navigate to **Expose an API**.
5. Under the **Authorized client applications** section, select **Add a client application**.
6. Paste the Application (client) ID you copied from the **ContosoApp** app into the **Client ID** text box.
7. Check the API scope presented under the **Authorize** scope.
8. Select **Add application**.

### Add permissions to access Web APIs for ContosApp

1. Navigate back to **App registrations** and select **ContosApp**.
2. Select **API permissions** and then select the **Add a permission** button.
3. From the Request API permissions dialog, select **My APIs**. You should now see your Hello ASPNET Core Identity 01 app available to select.
4. Select **Hello ASPNET Core Identity 01**.



5. Expand **Employees (1)**, select **Employees.Read.All** and then select **Add permissions**.
6. Wait for **Preparing the consent** to finish then select **Grant admin consent for Contoso**.
7. From the **Permissions requested** dialog, select **Yes**.
8. Your **ContosoApp** is now configured and authorized to use the API permissions from the **Hello ASPNET Core Identity 01** app.
9. This exercise is now complete.

## Review

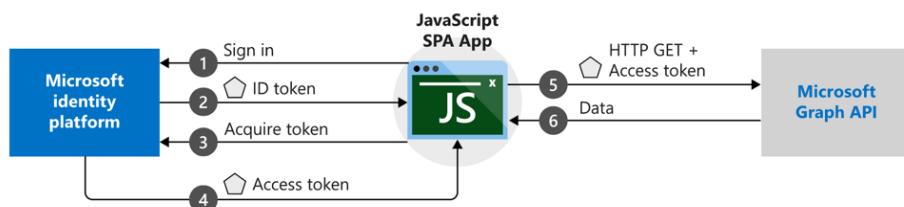
In this exercise, you learned how to define scope and how to authorize applications to make the API calls.

### ❖❖❖# Exercise 5: Implementing authorization to consume an API

This exercise demonstrates how a JavaScript single-page application (SPA) can:

- Sign in personal accounts, as well as work and school accounts.
- Acquire an access token.
- Call the Microsoft Graph API or other APIs that require access tokens from the Microsoft identity platform endpoint.

The sample application used in this exercise enables a JavaScript SPA to query the Microsoft Graph API or a web API that accepts tokens from the Microsoft identity platform endpoint. In this scenario, after a user signs in, an access token is requested and added to HTTP requests through the authorization header. Token acquisition and renewal are handled by the Microsoft Authentication Library (MSAL).



## Task 1: Download sample project

1. Download the sample project for Node.js:
  1. To run the project by using a local web server, such as Node.js, download the project files to your **C:/Labfiles** directory.  
Visit <https://github.com/Azure-Samples/active-directory-javascript-graphapi-v2/releases> and download the latest release: **Source code (zip)**.
2. Navigate to where the download zip file is and unblock file.
  1. Right-select and select **Properties**.
  2. Select the **Unblock** checkbox and select **OK**.
3. Extract the zipped file.
4. Open the project in Visual Studio Code.
  1. Launch Visual Studio Code as Administrator and browse for the **active-directory-javascript-graphapi-v2-quickstart** root directory and open.

## Task 2: Register an application

1. On the left menu, navigate to **App registrations**.
2. Select **New Registration**.
3. From the **Register an application** pane, perform the following actions:
  1. In the **Name** text box, enter a meaningful application name that will be displayed to users of the app, for example **JavaScript-SPA-App**.
  2. Under **Supported account type**, select **Accounts in any organizational directory and personal Microsoft accounts**.

### For setting a redirect URL for the Node.js project

Follow these steps if you choose to use the Node.js project. For Node.js, you can set the web server port in the server.js file. This project uses port 30662, but you can use any other available port.

1. To set up a redirect URL in the application registration information, switch back to the **Application Registration** pane, and do either of the following:
  1. Set `http://localhost:30662/` as the **Redirect URL**.
  2. If you're using a custom TCP port, use `http://localhost: [port] /` (where **[port]** is the custom TCP port number).
2. Select **Register**.
3. On the app **Overview** page, note the **Application (client) ID** value for later use.
4. This sample project requires the **Implicit grant flow** to be enabled. In the left pane of the registered application, select **Authentication**.
5. In **Advanced** settings, under **Implicit grant**, select the **ID tokens** and **Access tokens** check boxes. ID tokens and access tokens are required because this app must sign in users and call an API.
6. Select **Save**.

# Task 3: Permission and scope setup

## Update the App API permissions

1. In the left navigation, navigate to **API Permissions**.

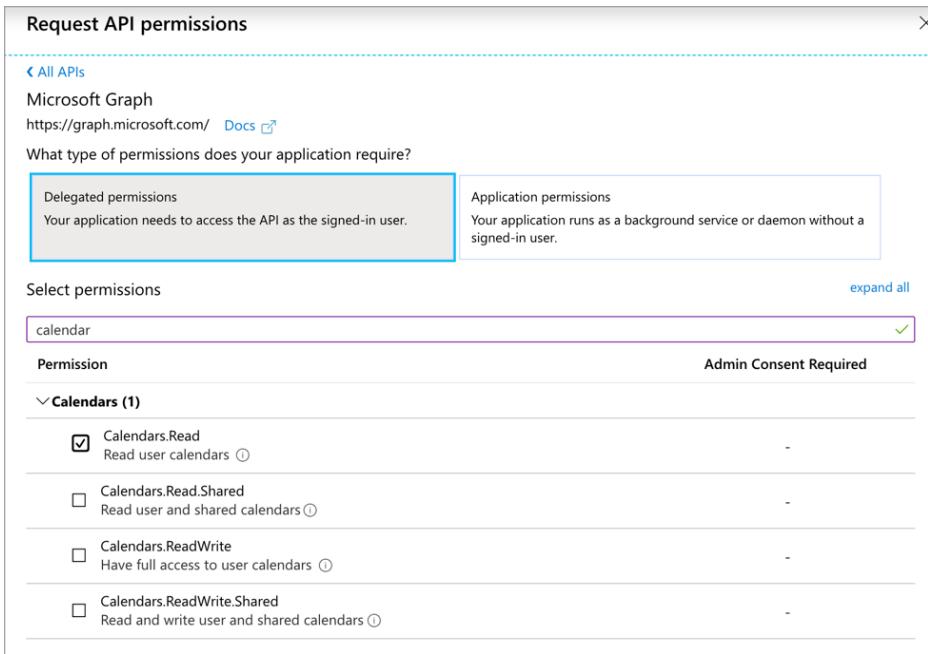
The screenshot shows the 'JavaScript-SPA-App - API permissions' page in the Azure portal. The left sidebar has sections like Overview, Quickstart, Manage (which is expanded), Branding, Authentication, Certificates & secrets, API permissions (selected), Expose an API, Owners, Roles and administrators (Preview), and Manifest. The main area shows 'Configured permissions' with a note about consent. A table lists one permission: Microsoft Graph (1) with User.Read scope, Type Delegated, Admin Consent... status, and Sign in and read user profile description.

2. Select **Add a permission**.

3. Next, select **Microsoft Graph** from **Microsoft APIs**.

The screenshot shows the 'Request API permissions' dialog. The 'Microsoft APIs' tab is selected. An arrow points to the 'Microsoft Graph' card, which is described as taking advantage of the tremendous amount of data in Office 365, Enterprise Mobility + Security, and Windows 10. Other cards include Azure Rights Management Services, Azure Service Management, Dynamics 365 Business Central, Flow Service, Intune, Office 365 Management APIs, OneNote, Power BI Service, SharePoint, and Skype for Business.

4. Select **Delegated Permissions** and under **Select permissions**, search for **calendars** and select **Calendars.Read**.



5. Search for people and select **People.Read** permission and provide admin consent.

6. Select **Add permissions**.

1. Wait for **Preparing for consent** to finish then select **Grant admin consent for Contoso**.
2. From the Permissions requested dialog, select **Yes**.
3. Your **JavaScript-SPA-App** app is now configured and authorized to use the Calendars.Read and People.Read permissions for Microsoft Graph.

## Update solution code

1. Navigate back to Visual Studio Code and open the **index.html** file.

2. Locate and select the following code:

```
html <div class="leftContainer"> <p id="WelcomeMessage">Welcome to the Microsoft Authentication Library For Javascript Quickstart</p> <button id="SignIn" onclick="signIn()">Sign In</button> </div>
```

3. Replace the selected code with the following, to add a new button to the application:

```
html <div class="leftContainer"> <p id="WelcomeMessage">Welcome to the Microsoft Authentication Library For Javascript Quickstart</p> <button id="SignIn" onclick="signIn()">Sign In</button> <button id="Share" onclick="acquireTokenPopupAndCallMSGraph()">Share</button> </div>
```

4. Locate the object **requestObj** and change the scopes to **people.read**.

```
javascript var requestObj = { scopes: ["people.read"] };
```

5. Update the graph API URL to call the people object.

```
javascript var graphConfig = { graphMeEndpoint: "https://graph.microsoft.com/v1.0/me", graphPeopleEndpoint:
```

```
"https://graph.microsoft.com/v1.0/people" };
```

6. Create a new endpoint for **people.read** and replace the method **acquireTokenPopupAndCallMSGraph** with the code below.

```
javascript function acquireTokenPopupAndCallMSGraph() { //Always start with
acquireTokenSilent to obtain a token in the signed in user from cache
myMSALObj.acquireTokenSilent(requestObj).then(function (tokenResponse) {
callMSGraph(graphConfig.graphMePeopleEndpoint, tokenResponse.accessToken,
graphAPICallback); }).catch(function (error) { console.log(error); // Upon
acquireTokenSilent failure (due to consent or interaction or login required
ONLY) // Call acquireTokenPopup (popup window) if
(requiresInteraction(error.errorCode)) {
myMSALObj.acquireTokenPopup(requestObj).then(function (tokenResponse) {
callMSGraph(graphConfig.graphPeopleEndpoint, tokenResponse.accessToken,
graphAPICallback); }).catch(function (error) { console.log(error); });
}); }
```

7. Find the **var msalConfig** code block and replace the following:

1. <Enter\_the\_Application\_Id\_here> is the **Application (client) ID** for the application you registered.
2. <Enter\_the\_Tenant\_info\_here> is set to one of the following options:
  - If your application supports **Accounts in this organizational directory**, replace this value with the **Tenant ID** or **Tenant name** (for example, **contoso.microsoft.com**).
  - If your application supports **Accounts in any organizational directory**, replace this value with **organizations**.
  - If your application supports **Accounts in any organizational directory and personal Microsoft accounts**, replace this value with **common**. To restrict support to Personal Microsoft accounts only, replace this value with **consumers**.

## Task 4: Run the application

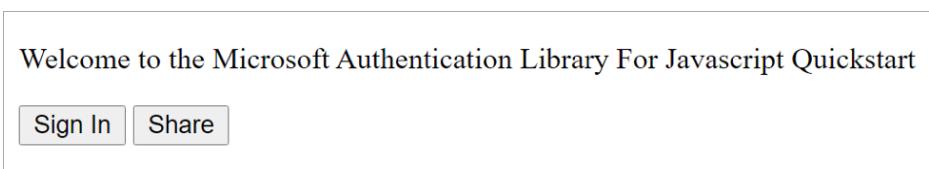
1. From **Visual Studio Code**, open **Terminal**. Type the following and hit ENTER:

```
typescript npm install
```

2. From the Terminal, type the following and hit ENTER.

```
typescript node server.js
```

3. From the browser, launch: **http://localhost:30662**



Welcome to the Microsoft Authentication Library For Javascript Quickstart

[Sign In](#) [Share](#)

A screenshot of a web browser displaying a landing page for the Microsoft Authentication Library for JavaScript. The page has a light blue header with the text "Welcome to the Microsoft Authentication Library For Javascript Quickstart". Below the header are two buttons: "Sign In" and "Share".

4. Select **Sign In**.

5. If the **Permissions requested** dialog opens, select **Accept**.

The screenshot shows a Microsoft Edge browser window with the following details:

- Title bar: Sign in to your account - MS Learning - Microsoft Edge
- Address bar: https://login.microsoftonline.com/organizations/oauth2/v2
- Content area:
  - Microsoft logo
  - Email address: admin@lodsm069121.onmicrosoft.com
  - Section title: Permissions requested
  - App details: JavaScript-SPA-App, unverified
  - Description: This app would like to:
  - Granted permissions:
    - Read your relevant people list
    - View your basic profile
    - Maintain access to data you have given it access to
  - Pending permission:
    - Consent on behalf of your organization
  - Text: Accepting these permissions means that you allow this app to use your data as specified in their terms of service and privacy statement. **The publisher has not provided links to their terms for you to review.** You can change these permissions at <https://myapps.microsoft.com>. [Show details](#)
  - Buttons: Cancel (gray), Accept (blue)

6. You should now be authenticated successfully.

Welcome admin@LODSM069121.onmicrosoft.com to Microsoft Graph API

[Sign Out](#) [Share](#)

7. Go back to the Visual Studio Code project. Stop the running project by selecting **Ctrl + C**.

8. Exit Visual Studio Code.

## Review

In this exercise, you implemented authorization and incremental consent using the Microsoft Identity. # Exercise 6: Creating a service to access Microsoft Graph

In this exercise, you'll create an Azure AD application and single page application for a user to sign in and display their information on the page. This application will use MSAL.js for the access token request.

# Task 1: Single page web application

In this application, you'll create an ASP.NET Core web application that allows users from the current organization to sign in and display their information.

## Create a Node.js web application

1. Open your command prompt, navigate to a directory where you want to save your work, create a new folder, and change directory into that folder. For example:

```
powershell cd c:/LabFiles md HelloWorldIdentity cd HelloWorldIdentity
```

2. Execute the following command to create a new Node.js application:

```
powershell npm init -y
```

3. Install the Node.js webserver **express** and HTTP request middleware **morgan** into the application:

```
powershell npm install express morgan
```

4. Open the project in Visual Studio code by executing:

```
powershell code .
```

5. Create a new file **server.js** in the root of the folder and add the following JavaScript to it. This code will start the web server:

```
javascript var express = require('express'); var app = express(); var  
morgan = require('morgan'); var path = require('path'); var port = 3007;  
app.use(morgan('dev')); // set the front-end folder to serve public assets.  
app.use(express.static('web'));// set up our one route to the index.html  
file. app.get('*', function (req, res) { res.sendFile(path.join(__dirname +  
'/index.html'))}); // Start the server. app.listen(port);  
console.log(`Listening on port ${port}...`); console.log('Press CTRL+C to  
stop the web server...');
```

## Create a web page for the user to sign in and display details

Create a new folder **web** in the current folder and add a new file **index.html** in the folder. Add the following code to the **index.html** file:

```
html <!DOCTYPE html> <html> <head> <title>Getting Started with Microsoft  
identity</title> <script  
src="https://cdnjs.cloudflare.com/ajax/libs/bluebird/3.5.5/bluebird.min.js">  
</script> <script src="https://alcdn.msftauth.net/lib/1.1.3/js/msal.min.js">  
</script> </head> <body> <div class="container"> <div> <p  
id="WelcomeMessage">Microsoft Authentication Library For Javascript (MSAL.js)  
Exercise</p> <button id="SignIn" onclick="signIn()">Sign In</button> </div>  
<div> <pre id="json"></pre> </div> <script> var msalConfig = { auth: {  
clientId: '', authority: '', redirectURI: '' }, cache: { cacheLocation:  
"localStorage", storeAuthStateInCookie: true } }; var graphConfig = {  
graphMeEndpoint: "https://graph.microsoft.com/v1.0/me", requestObj: { scopes:  
["user.read"] } }; var msalApplication = new  
Msal.UserAgentApplication(msalConfig); // init the auth handling on the page  
initPage(); // TODO: add CODE before this line // TODO: add FUNCTIONS before  
this line function initPage() { // Browser check variables var ua =  
window.navigator.userAgent; var msie = ua.indexOf('MSIE '); var msie11 =
```

```

ua.indexOf('Trident/'); var msedge = ua.indexOf('Edge/'); var isIE = msie > 0
|| msie11 > 0; var isEdge = msedge > 0; // if you support IE, recommendation:
sign in using Redirect APIs vs. popup // Browser check variables // can change
this to default an experience outside browser use var loginType = isIE ?
"REDIRECT" : "POPUP"; // runs on page load, change config to try different
login types to see what is best for your application switch (isIE) { case true:
document.getElementById("SignIn").onclick = function () {
msalApplication.loginRedirect(graphConfig.requestObj); }; // avoid duplicate
code execution on page load in case of iframe and popup window if
(msalApplication.getAccount() &&
!msalApplication.isCallback(window.location.hash)) { updateUserInterface();
acquireTokenRedirectAnd GetUser(); } break; case false: // avoid duplicate code
execution on page load in case of iframe and popup window if
(msalApplication.getAccount()) { updateUserInterface();
acquireTokenPopupAnd GetUser(); } break; default: console.error('Please set a
valid login type'); break; } } </script> </body> </html>

```

**Note:** The remainder of this exercise instructs you to add code to this **index.html** file. Pay close attention where you add the code using the using the two **TODO:** comments for placement.

1. Add the following function to the **index.html** file immediately before the **// TODO: add FUNCTIONS before this line** comment that will configure the welcome message for the page:

```

javascript function updateUserInterface() { var divWelcome =
document.getElementById('WelcomeMessage'); divWelcome.innerHTML = `Welcome
<strong>${msalApplication.getAccount().userName}</strong> to Microsoft
Graph API`; var loginbutton = document.getElementById('SignIn');
loginbutton.innerHTML = 'Sign Out'; loginbutton.setAttribute('onclick',
'signOut();'); }

```

2. Next, add the following functions to **index.html** immediately before the **// TODO: add FUNCTIONS before this line** comment.

- These functions request an access token from Microsoft identity and submit a request to Microsoft Graph for the current user's information. The function **acquireTokenPopupAnd GetUser()** uses the popup approach that works for all modern browsers while the **acquireTokenRedirectAnd GetUserFromMSGraph()** function uses the redirect approach that is suitable for Internet Explorer:

```

javascript function acquireTokenPopupAnd GetUser() {
msalApplication.acquireTokenSilent(graphConfig.requestObj)
.then(function (tokenResponse) {
getUserFromMSGraph(graphConfig.graphMeEndpoint,
tokenResponse.accessToken, graphAPICallback); }).catch(function (error)
{ console.log(error); if (requiresInteraction(error.errorCode)) {
msalApplication.acquireTokenPopup(graphConfig.requestObj).then(function
(tokenResponse) { getUserFromMSGraph(graphConfig.graphMeEndpoint,
tokenResponse.accessToken, graphAPICallback); }).catch(function (error)
{ console.log(error); }); } });
function acquireTokenRedirectAnd GetUser() {
msalApplication.acquireTokenSilent(graphConfig.requestObj).then(function
(tokenResponse) { getUserFromMSGraph(graphConfig.graphMeEndpoint,
tokenResponse.accessToken, graphAPICallback); }).catch(function (error)
{ console.log(error); if (requiresInteraction(error.errorCode)) {
msalApplication.acquireTokenRedirect(graphConfig.requestObj); } });
function requiresInteraction(errorCode) { if (!errorCode ||
!errorCode.length) { return false; } return errorCode ===
"consent_required" || errorCode === "interaction_required" || errorCode
=== "login_required"; }

```

**Note:** These functions first attempt to retrieve the access token silently from the currently signed in user. If the user needs to sign in, the functions will trigger either the popup or

redirect authentication process. The redirect approach to authenticating requires an extra step. The MSAL application on the page needs to see if the current page was requested based on a redirect from Azure AD. If so, it needs to process information in the URL request provided by Azure AD.

1. Add the following function immediately before the **// TODO: add FUNCTIONS before this line** comment:

```
javascript function authRedirectCallBack(error, response) { if (error) { console.log(error); } else { if (response.tokenType === "access_token") { getUserFromMSGraph(graphConfig.graphMeEndpoint, response.accessToken, graphAPICallback); } else { console.log("token type is:" + response.tokenType); } } }
```

3. Configure MSAL to use this function by adding the following line immediately before the **// TODO: add CODE before this line** comment:

```
javascript msalApplication.handleRedirectCallback(authRedirectCallBack);
```

1. Once the user is authenticated, the code can submit a request to Microsoft Graph for the current user's information. The two **acquireToken\***() functions pass the access token acquired from Azure AD to the function:
2. Add the following function immediately before the **// TODO: add FUNCTIONS before this line** comment:

```
javascript function getUserFromMSGraph(endpoint, accessToken, callback) { var xmlhttp = new XMLHttpRequest(); xmlhttp.onreadystatechange = function () { if (this.readyState == 4 && this.status == 200) callback(JSON.parse(this.responseText)); } xmlhttp.open("GET", endpoint, true); xmlhttp.setRequestHeader('Authorization', 'Bearer ' + accessToken); xmlhttp.send(); } function graphAPICallback(data) { document.getElementById("json").innerHTML = JSON.stringify(data, null, 2); }
```

4. Finally, add the following two functions to implement a sign in and sign out capability for the button on the page. Add the following function immediately before the **// TODO: add FUNCTIONS before this line** comment:

```
javascript function signIn() { msalApplication.loginPopup(graphConfig.requestObj).then(function (loginResponse) { updateUserInterface(); acquireTokenPopupAnd GetUser(); }).catch(function (error) { console.log(error); }); } function signOut() { msalApplication.logout(); }
```

## Task 2: Register a new application

1. From the Azure portal <https://portal.azure.com>, navigate to **Azure Active Directory**.
2. Select **Manage > App registrations** in the left-hand navigation.
3. On the **App registrations** page, select **New registration**.
4. On the **Register an application** page, set the values as follows:
  - **Name:** Hello World Identity
  - **Supported account types:** Accounts in this organizational directory only (Single tenant)
  - **Redirect URI:** Web = http://localhost:3007
5. Select **Register** to create the application.
6. On the **Hello World Identity** page, copy the values **Application (client) ID** and **Directory (tenant) ID**; you'll need these values later in this exercise.
7. On the **Hello World Identity** page, select the **1 web, 0 public client** link under the **Redirect URIs**.
8. Locate the section **Implicit grant** and select both **Access tokens** and **ID tokens**.
9. Select **Save** when finished setting these values.

## Task 3: Update the web page with the Azure AD application details

The last task is to configure the web page to use the Azure AD application. 1. Go back to the project in Visual Studio Code.

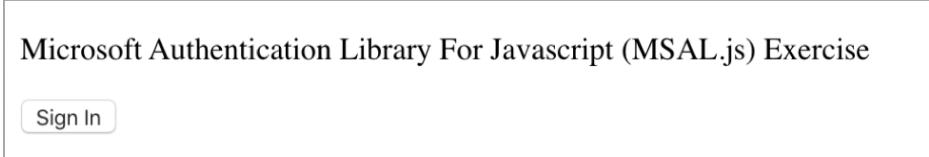
1. Open the **index.html** file and locate the `var msalConfig = {}` code. The auth object contains three properties you need to set as follows:
  - **clientId**: set to the Azure AD application's ID
  - **authority**: set to `https://login.microsoftonline.com/{{DIRECTORY_ID}}`, replacing the `{{DIRECTORY_ID}}` with the Azure AD directory ID of the Azure AD application.
  - **redirectURI**: set to the Azure AD application's redirect URI: `http://localhost:3007`

## Task 4: Test the web application

1. To test the web page, first start the local web server. In the command prompt, execute the following command from the root of the project:

```
powershell node server.js
```

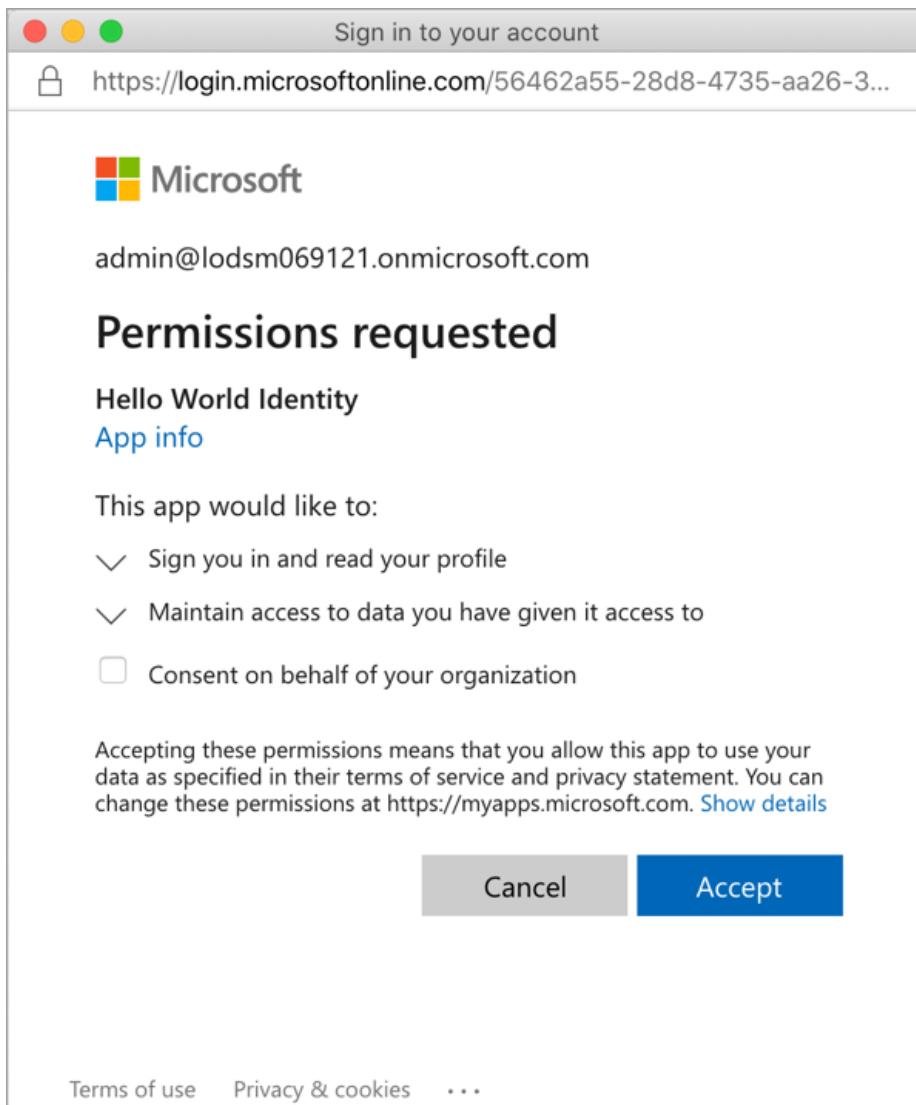
2. Next, open a browser and navigate to **http://localhost:3007**. The page initially contains a default welcome message and sign in button.



Microsoft Authentication Library For Javascript (MSAL.js) Exercise

Sign In

3. Select the **sign in** button.
4. Depending on the browser, you are using, a popup window will load or the page will redirect to the Azure AD sign in prompt.
5. Sign in using a **Work or School Account** and accept the permissions requested for the application by selecting **Accept**.



6. Depending on the browser you're using, the popup will disappear, or you will be redirected back to the web page. When the page loads, MSAL will request an access token and request your information from Microsoft Graph. After the request complete, it will display the results on the page:

Welcome **admin@LODSM069121.onmicrosoft.com** to Microsoft Graph API

[Sign Out](#)

```
{ "@odata.context": "https://graph.microsoft.com/v1.0/$metadata#users/$entity", "businessPhones": [ "8006427676" ], "displayName": "MOD Administrator", "givenName": "MOD", "jobTitle": null, "mail": "admin@LODSM069121.OnMicrosoft.com", "mobilePhone": "425-882-1032", "officeLocation": null, "preferredLanguage": "en-US", "surname": "Administrator", "userPrincipalName": "admin@LODSM069121.onmicrosoft.com", "id": "1adc26ec-31ec-412b-b9b1-75f63c0d8ff9" }
```

7. Stop the web server by going back to the running project in Visual Studio Code. From the Terminal, press **CTRL+C** in the command prompt.

## **Review**

In this exercise, you created an Azure AD application and single page application for a user to sign in and display their information on the page.

❖❖❖# Student lab answer key

## **Microsoft Azure user interface**

Given the dynamic nature of Microsoft cloud tools, you might experience Azure user interface (UI) changes after the development of this training content. These changes might cause the lab instructions and lab steps to not match up.

The Microsoft Worldwide Learning team updates this training course as soon as the community brings needed changes to our attention. However, because cloud updates occur frequently, you might encounter UI changes before this training content is updated. If this occurs, adapt to the changes and work through them in the labs as needed.

# Instructions

**Note:** Lab virtual machine sign in instructions will be provided to you by your instructor. ♦♦♦

Observe the taskbar located at the bottom of your Windows 10 desktop. The taskbar contains the icons for the applications you will use in this lab, including:

- Microsoft Edge
- File Explorer
- Visual Studio Code
- Microsoft Visual Studio 2019
- PowerShell

Also, ensure that the following necessary utilities are installed:

- [.NET Core 3.1 SDK](#)
- ngrok

♦♦♦# Exercise 1: Using query parameters when querying Microsoft Graph via HTTP

**Note:** Users may sign in with their Microsoft account or use the default sample account to complete the following tasks.

By the end of this exercise you will be able to:

- Use `$filter` query parameter.
- Use `$select` query parameter.
- Order results using `$orderby` query parameter.
- Set page size of results using `$skip` and `$top` query parameters.
- Expand and retrieve resources using `$expand` query parameter.
- Retrieve the total count of matching resources using `$count` query parameter.
- Search for resources using `$search` query parameter.

## Task 1: Go to the Graph Explorer

1. Open the Microsoft Edge browser.
2. Go to this link: <https://developer.microsoft.com/en-us/graph/graph-explorer>.

This page allows users to interact with Microsoft Graph without writing any code. Microsoft Graph Explorer provides sample data to use for read operations.

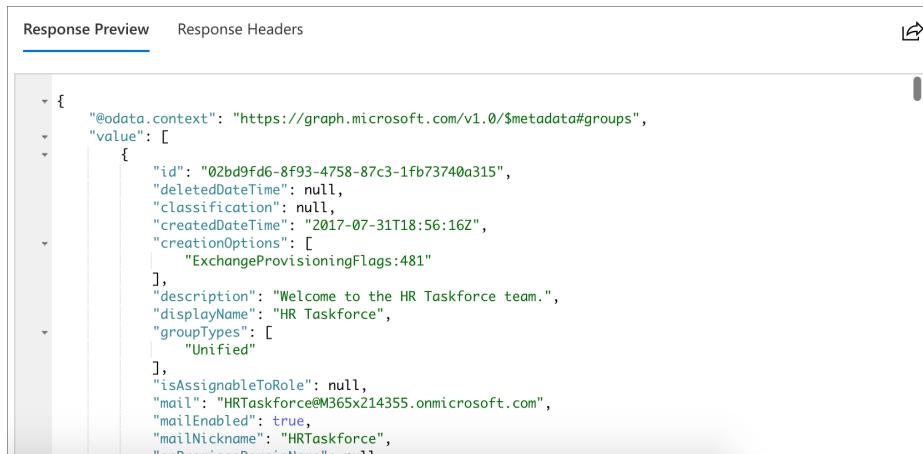
**NOTE:** Some organizations may not allow users to sign in or consent to specific scopes required for some operations.

## Task 2: Use `\$select` to retrieve only some object properties

1. In the **request URL** box amend the request to be:  
`https://graph.microsoft.com/v1.0/groups`

### 2. Select Run Query.

This lists all the groups in the tenant that the current user can see in the **Response Preview** pane at the bottom of the page. Note there are many properties returned per record.



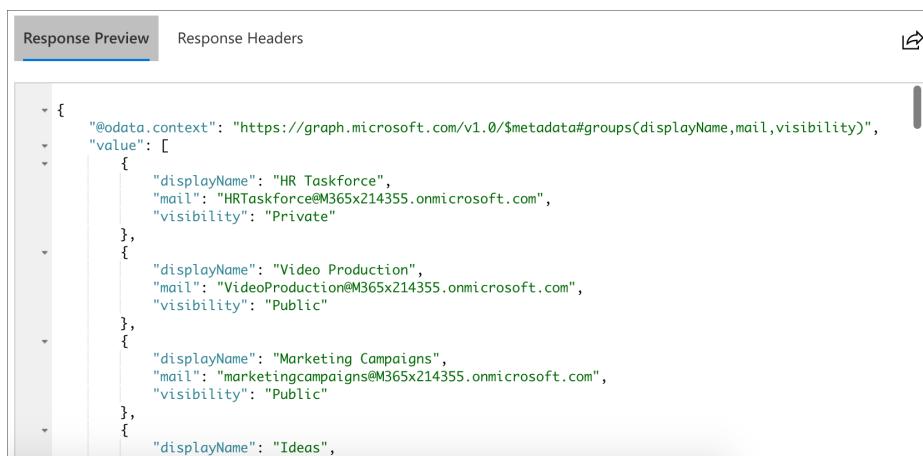
```
{  
  "@odata.context": "https://graph.microsoft.com/v1.0/$metadata#groups",  
  "value": [  
    {  
      "id": "02bd9fd6-8f93-4758-87c3-1fb73740a315",  
      "deletedDateTime": null,  
      "classification": null,  
      "createdDateTime": "2017-07-31T18:56:16Z",  
      "creationOptions": [  
        "ExchangeProvisioningFlags:481"  
      ],  
      "description": "Welcome to the HR Taskforce team.",  
      "displayName": "HR Taskforce",  
      "groupTypes": [  
        "Unified"  
      ],  
      "isAssignableToRole": null,  
      "mail": "HRTaskforce@M365x214355.onmicrosoft.com",  
      "mailEnabled": true,  
      "mailNickname": "HRTaskforce",  
      "...  
    }  
  ]  
}
```

### 3. Select the **Response Headers** tab.

Note the value of the **content-length** header. This shows how much data was returned in the response for this request.

4. To reduce the response to only include the necessary properties, you can make use of the **\$select** parameter. Update the **request URL** box to the following:

`https://graph.microsoft.com/v1.0/groups?$select=displayName,mail,visibility`



```
{  
  "@odata.context": "https://graph.microsoft.com/v1.0/$metadata#groups(displayName,mail,visibility)",  
  "value": [  
    {  
      "displayName": "HR Taskforce",  
      "mail": "HRTaskforce@M365x214355.onmicrosoft.com",  
      "visibility": "Private"  
    },  
    {  
      "displayName": "Video Production",  
      "mail": "VideoProduction@M365x214355.onmicrosoft.com",  
      "visibility": "Public"  
    },  
    {  
      "displayName": "Marketing Campaigns",  
      "mail": "marketingcampaigns@M365x214355.onmicrosoft.com",  
      "visibility": "Public"  
    },  
    {  
      "displayName": "Ideas",  
      "mail": "Ideas@M365x214355.onmicrosoft.com",  
      "visibility": "Public"  
    }  
  ]  
}
```

### 5. Select Run Query.

In the **Response Headers** tab note the new value for the **content-length** header. There is a much smaller amount of data returned now.

6. Select the **Response Preview** tab.

Note only the selected properties are returned. To allow an application to interact directly with an object in Microsoft Graph, it is best to have the unique identifier for that object.

7. With this in mind, amend your query to include the **id** property:

```
https://graph.microsoft.com/v1.0/groups?  
$select=displayName,mail,visibility,id
```

8. Select **Run Query**.

The screenshot shows the Microsoft Graph API Response Preview tool. It has two tabs at the top: "Response Preview" (which is selected) and "Response Headers". The main area displays a JSON response with three groups. A red arrow points to the "id" field of the first group, which is highlighted in yellow. The JSON data is as follows:

```
{  
  "@odata.context": "https://graph.microsoft.com/v1.0/$metadata#groups(displayName,mail,visibility,id)",  
  "value": [  
    {  
      "displayName": "HR Taskforce",  
      "mail": "HRTaskforce@M365x214355.onmicrosoft.com",  
      "visibility": "Private",  
      "id": "02bd9fd6-8f93-4758-87c3-1fb73740a315"  
    },  
    {  
      "displayName": "Video Production",  
      "mail": "VideoProduction@M365x214355.onmicrosoft.com",  
      "visibility": "Public",  
      "id": "06f62f70-9827-4e6e-93ef-8e0f2d9b7b23"  
    },  
    {  
      "displayName": "Marketing Campaigns",  
      "mail": "marketingcampaigns@M365x214355.onmicrosoft.com",  
      "visibility": "Public",  
      "id": "0a53828f-36c9-44c3-be3d-99a7fce977ac"  
    }  
  ]  
}
```

## Task 3: Use `$orderby` to sort results

When presenting data to end users it's often necessary to use a sort order other than the default provided by Microsoft Graph. This should be done using the `$orderby` parameter.

Continuing from the previous task, sort the groups by the `displayName` property.

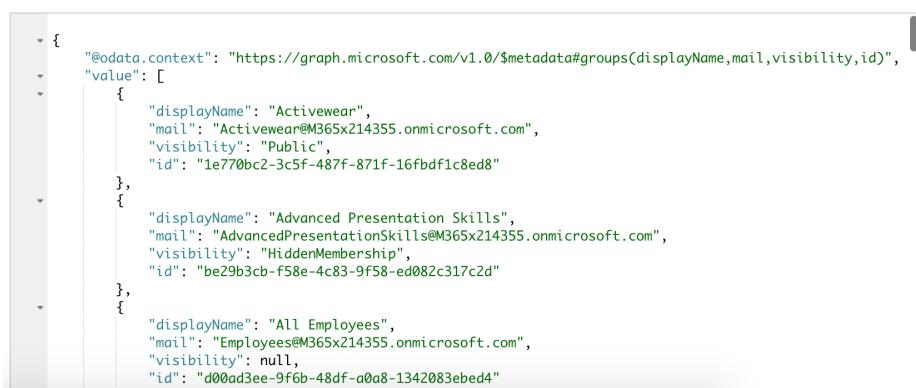
1. Change the request URL box to:

```
https://graph.microsoft.com/v1.0/groups?  
$select=displayName,mail,visibility,id&$orderBy=displayName
```

2. Select **Run Query**.

The sorted set of groups is shown in the **Response Preview** pane.

Response Preview   Response Headers 



```
[{"@odata.context": "https://graph.microsoft.com/v1.0/$metadata#groups(displayName,mail,visibility,id)", "value": [{"displayName": "Activewear", "mail": "Activewear@M365x214355.onmicrosoft.com", "visibility": "Public", "id": "1e770bc2-3c5f-487f-871f-16fbdf1c8ed8"}, {"displayName": "Advanced Presentation Skills", "mail": "AdvancedPresentationSkills@M365x214355.onmicrosoft.com", "visibility": "HiddenMembership", "id": "be29b3cb-f58e-4c83-9f58-ed082c317c2d"}, {"displayName": "All Employees", "mail": "Employees@M365x214355.onmicrosoft.com", "visibility": null, "id": "d00ad3ee-9f6b-48df-a0a8-1342083ebcd"}]}
```

**NOTE:** Many properties cannot be used for sorting. For example, if you were to use `mail` in the `$orderby` you would receive an HTTP 400 response like this:

```
http { "error": { "code": "Request_UnsupportedQuery", "message": "Unsupported sort property 'mail' for 'Group'.", "innerError": { "requestId": "582643b8-7ac2-415a-b58b-59009ec63ec1", "date": "2019-10-24T19:54:55" } } }
```

3. Set the sort direction using either **asc** or **desc** like this:

```
https://graph.microsoft.com/v1.0/groups?  
$select=displayName,mail,visibility,id&$orderBy=displayName desc
```

## Task 4: Use `$filter` to retrieve a subset of data available

1. In the **request URL** box amend the request to:

```
https://graph.microsoft.com/v1.0/me/messages
```

2. Select **Run Query**.

This shows a list of messages for the current **Response Preview** pane at the bottom of the page.

Note the response also includes an `@odata.nextLink` property. This allows developers to fetch additional pages of results and, by its presence, indicates there may be a large possible number of results. To find just the email messages that have attachments, add a filter.

The screenshot shows the Microsoft Graph Explorer interface with the 'Response Preview' tab selected. It displays a JSON response structure. A red arrow points to the '@odata.context' object, specifically to the '@odata.nextLink' property, which contains the URL `https://graph.microsoft.com/v1.0/me/messages?$skip=15`. The main body of the response shows a list of message objects, each with properties like id, subject, bodyPreview, and hasAttachments.

```
{
  "@odata.context": "https://graph.microsoft.com/v1.0/$metadata#users('48d31887-5fad-4d73-a9f5-3c356e68a038')/messages",
  "@odata.nextLink": "https://graph.microsoft.com/v1.0/me/messages?$skip=15",
  "value": [
    {
      "id": "AAMkAGVnMDEzMTM4LTZmYWUtNDdkNC1hMDZiLTU1OGY50TZhYmY40AAuAAAAAAiQ8W967B7TKBjgx9rVEURBwAiI",
      "subject": "MyAnalytics | Focus Edition",
      "bodyPreview": "MyAnalytics\nDiscover your habits. Work smarter.\nFor your eyes only",
      "hasAttachments": false,
      "categories": [],
      "lastModifiedDateTime": "2019-11-04T16:12:47Z",
      "changeKey": "CQAAABYAAAiIsqMbYjsTSe/T7KzowPTAAIgRSJV",
      "receivedDateTime": "2019-11-04T16:12:47Z",
      "sentDateTime": "2019-11-04T16:12:47Z",
      "parentFolderId": "AAMkAGVnMDEzMTM4LTZmYWUtNDdkNC1hMDZiLTU1OGY50TZhYmY40AAQAFRLF1C8ZxIlmEwGEI8Pc",
      "conversationId": "AAQkAGVnMDEzM4LTZmYWUtNDdkNC1hMDZiLTU1OGY50TZhYmY40AAQAFRLF1C8ZxIlmEwGEI8Pc"
    }
  ]
}
```

3. Add a filter so the **request URL** box contains the following:

```
https://graph.microsoft.com/v1.0/me/messages?$filter=hasAttachments eq true
```

4. Select **Run Query**.

The results are shown in the **Response Preview** pane. Note there is no `@odata.nextLink` this time. This indicates all the matching records have been returned in the request. If there is an `@odata.nextLink` it should be used to fetch additional pages of results. It's possible to compose queries that traverse object properties as well. For instance, to find mail sent from a specific email address.

5. Change the **request URL** box to:

```
https://graph.microsoft.com/v1.0/me/messages?
$filter=from/emailAddress/address eq 'no-reply@microsoft.com'
```

6. Select **Run Query**.

The results are shown in the **Response Preview** pane. Note there are multiple pages of results as indicated by the presence of `@odata.nextLink`. Some properties support the use of a start with operator.

7. Enter the following into the **request URL** box:

```
https://graph.microsoft.com/v1.0/me/messages?  
$filter=startswith(subject,'my')
```

8. Select **Run Query**.

The results are shown in the **Response Preview** pane.

9. Some collection properties can be used in **\$filter** using a Lambda expression.

For example, to find all groups that have the Unified type, enter the following query into the request URL box:

```
https://graph.microsoft.com/v1.0/groups?$filter=groupTypes/any(c: c eq  
'Unified')
```

10. Select **Run Query**.

The list of groups that have the Unified type is shown in the **Response Preview** pane.

## Task 5: Use `\$skip` and `\$top` for explicit pagination of results

1. In the request URL box change the request to:

```
https://graph.microsoft.com/v1.0/me/events
```

2. Select **Run Query**.

The results are shown in the **Response Preview** pane.

Note the `@odata.nextLink` property finishes in `$skip 10`. This is the default first page size for the `/events` resource. Different resources have different default first page sizes. The size of the first page and subsequent pages may not be the same. Edit the query to fetch an explicit number of results. `$top` is used to set maximum the number of results to be returned.

3. In the request URL box amend the request to:

```
https://graph.microsoft.com/v1.0/me/events?$top=5
```

4. Select **Run Query**.

The results are shown in the **Response Preview** pane.

Note the `@odata.nextLink` property finishes in `$skip 5`. Using the next link here would fetch the next five records in the data set. To fetch the next page set the `\$skip` value to 5:

```
https://graph.microsoft.com/v1.0/me/events?$top=5&$skip=5
```

This is the same as the `@odata.nextLink` returned for this resource. The `$skip` value tells Microsoft Graph to start returning results after the number provided here.

5. Select **Run Query**.

The results are shown in the **Response Preview** pane. The value returned contains an `@odata.nextLink` that can be followed. If a client needs to load all data from a specific resource or query it should keep following the `@odata.nextLink` until it is no longer present in the results.

**NOTE:** The formula for composing manual pagination queries is this:

```
?$top={pageSize}&$skip={pageSize}*({pageNumber} - 1)}
```

## Task 6: Expand and retrieve resources using \\$expand query parameter

In this exercise you will fetch the attachments from the mail of the current user. In a previous task you found the messages for the current user, which have attachments. In the request URL box amend the request to perform this query again.

1. In the **request URL** box amend the request to:

```
https://graph.microsoft.com/v1.0/me/messages?$filter=hasAttachments eq true
```

2. Select **Run Query**.

The results are shown in the **Response Preview** pane. Note when examining the results there is no information provided about the attachments. Amend the query to fetch information about the messages' attachments.

1. In the **request URL** box amend the request to:

```
https://graph.microsoft.com/v1.0/me/messages?$filter=hasAttachments eq true&$expand=attachments
```

2. Select **Run Query**.

The results are shown in the **Response Preview** pane.

Note the attachment objects included in the response include the **contentBytes** property. This is the actual file content and can cause the application to fetch far more data than might be desired.

3. Select the **Response Headers** tab.

Note the value for the **content-length** header.

4. Amend the query to use **\$select** on the **\$expanded** attachments collection. In the **request URL** box amend the request to:

```
https://graph.microsoft.com/v1.0/me/messages?$filter=hasAttachments eq true&$expand=attachments($select=id,name,contentType,size)
```

5. Select **Run Query**.

Note the new value for the **content-length** header is much smaller than the previous request.

6. Select **Response Preview**.

Note that for each attachment, in addition to the explicitly requested properties, there are some additional **@odata** properties.

## Task 7: Use `\$count` to discover the total number of matching resources

1. In the request URL box amend the request to:

```
https://graph.microsoft.com/v1.0/me/messages?$count=true
```

2. Select **Run Query**.

In the **Response Preview** pane there is an additional property shown `@odata.count`. This value shows the total number of resources that match the provided query. Note that as no query has been explicitly provided, the `@odata.count` value is that of all resources at this path.

3. Add a query, in this instance query to see how many unread messages the current user has. In the **request URL** box amend the request to:

```
https://graph.microsoft.com/v1.0/me/messages?$count=true&$filter=isRead eq false
```

4. Select **Run Query**.

In the **Response Preview** pane, the first page displays matching results. Note the `@odata.count` value is updated to reflect the query passed via the `$filter` parameter.

## Task 8: Use \\$search to discover the total number of matching resources

1. In the request URL box amend the request to:

```
https://graph.microsoft.com/v1.0/me/messages?$search="business"
```

2. Select **Run Query**.

In the **Response Preview** the messages that have matches for the keyword supplied are shown. Note the number of messages that match (4). The scope of the search can be adjusted specifying property names that are recognized by the KQL syntax.

See <https://docs.microsoft.com/en-us/graph/query-parameters#search-parameter> for more information.

1. Amend the **request URL** box to scope the search to only target the subject field. In the **request URL** box amend the request to:

```
https://graph.microsoft.com/v1.0/me/messages?$search="subject:business"
```

2. Select **Run Query**.

In the **Response Preview** the messages that have matches in the specified property are shown. Note there should be fewer results (3).

## Review

In this exercise, you learned how to make use of query parameters:

- Fetch resources with properties matching certain parameters by using the `$filter` query parameter.
- Fetch only the necessary data by using the `$select` query parameter.
- Set the order of results using `$orderby` query parameter.
- Set page size of results using `$skip` and `$top` query parameters.
- Expand and retrieve additional resources using `$expand` query parameter.
- Retrieve the total count of matching resources using `$count` query parameter.
- Search for resources using `$search` query parameter.

### ❖❖❖# Exercise 2: Retrieve and control information returned from Microsoft Graph

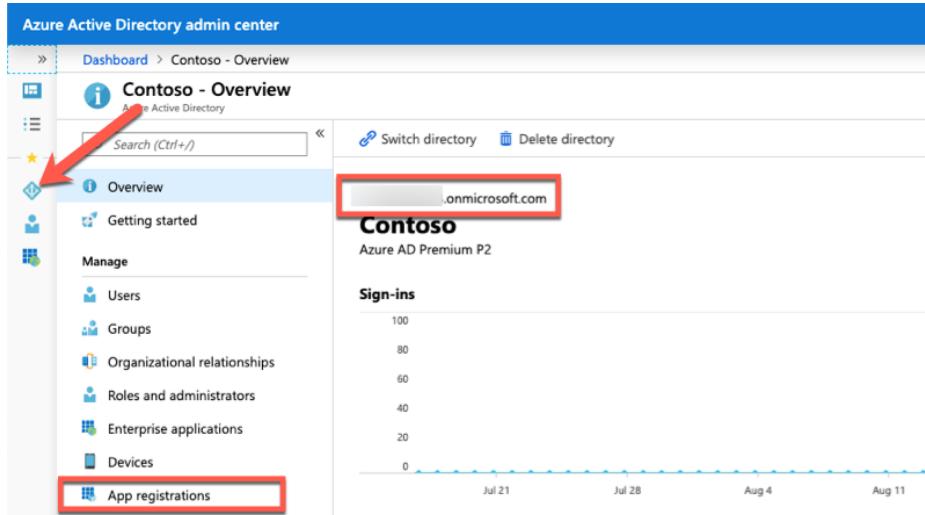
In this exercise, you will create a new Azure AD web application registration using the Azure Active Directory (Azure AD) admin center, a .NET Core console application, and query Microsoft Graph.

By the end of this exercise you will be able to use the following queries:

- `$select`
- `$top`
- `$orderby`
- `$filter`

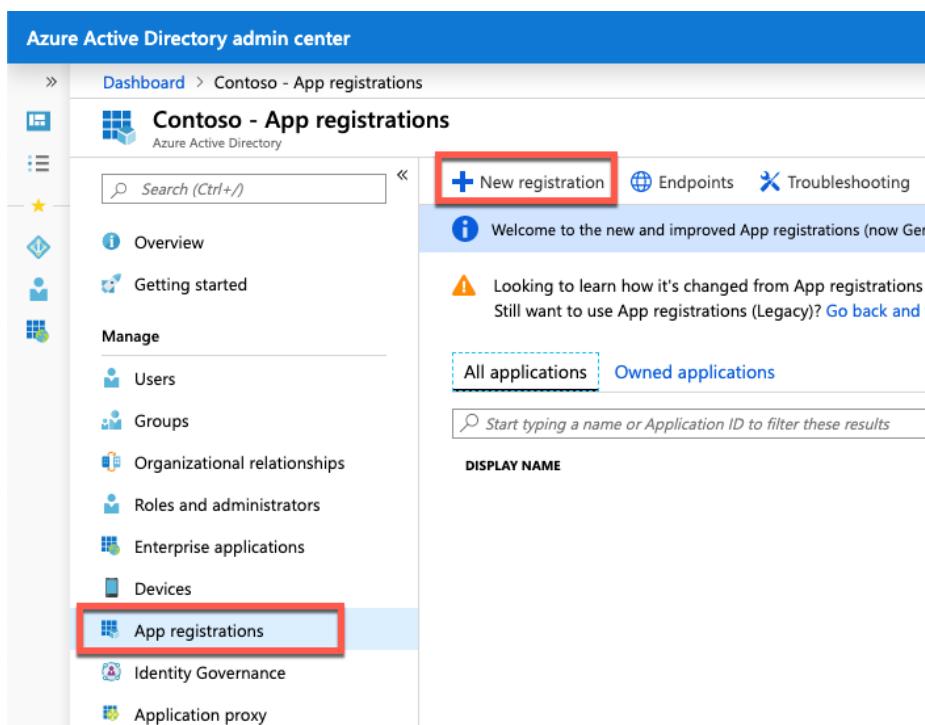
## Task 1: Create an Azure AD application

1. Open a browser and navigate to the [Azure Active Directory admin center](https://aad.portal.azure.com) (<https://aad.portal.azure.com>). Sign in using a **Work or School Account** that has global administrator rights to the tenancy.
2. Select **Azure Active Directory** in the leftmost navigation panel.



The screenshot shows the Azure Active Directory admin center interface. The left sidebar has a 'Manage' section with links for Users, Groups, Organizational relationships, Roles and administrators, Enterprise applications, and Devices. Below these, 'App registrations' is highlighted with a red box. The top navigation bar also has a 'App registrations' link highlighted with a red box. The main content area displays 'Contoso - Overview' for 'Azure AD Premium P2'. It includes a 'Sign-ins' chart showing activity from July 21 to Aug 11, with values ranging from 0 to 100.

3. Select **Manage > App registrations** in the left navigation panel.
4. On the **App registrations** page, select **New registration**.



The screenshot shows the 'Contoso - App registrations' page. The left sidebar has a 'Manage' section with links for Users, Groups, Organizational relationships, Roles and administrators, Enterprise applications, and Devices. Below these, 'App registrations' is highlighted with a red box. The top navigation bar has a 'New registration' button highlighted with a red box. The main content area includes a welcome message, a note about legacy app registrations, and tabs for 'All applications' and 'Owned applications'. There is also a search bar and a 'DISPLAY NAME' input field.

5. On the **Register an application** page, set the values as follows:
  - o **Name:** Graph Console App

- **Supported account types:** Accounts in this organizational directory only (Contoso only - Single tenant)
- **Redirect URI:** Web = <https://localhost>

\* Name  
The user-facing display name for this application (this can be changed later).

Graph Console App ✓

Supported account types  
Who can use this application or access this API?

Accounts in this organizational directory only (Contoso only - Single tenant)  
 Accounts in any organizational directory (Any Azure AD directory - Multitenant)  
 Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)  
 Personal Microsoft accounts only

[Help me choose...](#)

Redirect URI (optional)  
We'll return the authentication response to this URI after successfully authenticating the user. Providing this now is optional and it can be changed later, but a value is required for most authentication scenarios.

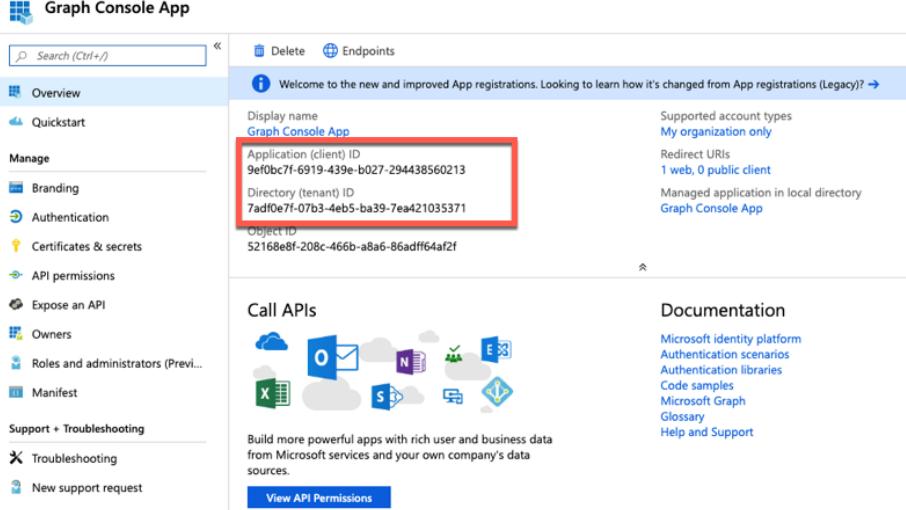
Web https://localhost ✓

By proceeding, you agree to the Microsoft Platform Policies [↗](#)

[Register](#)

## 6. Select Register.

7. On the **Graph Console App** overview page, copy the value of the **Application (client) ID** and **Directory (tenant) ID**; you will need them later in this exercise.



The screenshot shows the Microsoft Azure portal's App registrations section. A specific app named "Graph Console App" is selected. The "Overview" tab is active. Key details shown include:

- Display name:** Graph Console App
- Application (client) ID:** 9ef0bc7f-6919-439e-b027-294438560213 (highlighted with a red box)
- Directory (tenant) ID:** 7adff0e7f-07b3-4eb5-ba39-7ea421035371
- Object ID:** 52168e8f-208c-466b-a8a6-86adff64af2f
- Supported account types:** My organization only
- Redirect URIs:** 1 web, 0 public client
- Managed application in local directory:** Graph Console App

## 8. Select Manage > Certificates & secrets.

## 9. Select New client secret.

**GraphNotificationTutorial - Certificates & secrets**

Certificates enable applications to identify themselves to the authentication service when receiving tokens at a web addressable location (using an HTTPS scheme). For a higher level of assurance, we recommend using a certificate (instead of a client secret) as a credential.

**Certificates**

Certificates can be used as secrets to prove the application's identity when requesting a token. Also can be referred to as public keys.

**Client secrets**

A secret string that the application uses to prove its identity when requesting a token. Also can be referred to as application password.

DESCRIPTION	EXPIRES	VALUE
No client secrets have been created for this application.		

10. In the **Add a client secret** page, enter a value in **Description**, select one of the options for **Expires**, and select **Add**.

**Home > Contoso - App registrations (Preview) > GraphNotificationTutorial - Certificates & secrets**

**GraphNotificationTutorial - Certificates & secrets**

**Add a client secret**

Description  
Forever

Expires  
 In 1 year  
 In 2 years  
 Never

**Add** **Cancel**

No certificates have been added for this application.

**Client secrets**

A secret string that the application uses to prove its identity when requesting a token. Also can be referred to as application password.

DESCRIPTION	EXPIRES	VALUE
No client secrets have been created for this application.		

11. Copy the client secret value before you leave this page. You will need it in the next step.

**Important:** This client secret is never shown again, so make sure you copy it now.

**Client secrets**

A secret string that the application uses to prove its identity when requesting a token. Also can be referred to as application password.

+ New client secret	Expires	Value
Description Forever	Expires 12/31/2299	XXXXXXXXXXXXXX... (red box)

12. Grant Azure AD application permissions to Microsoft Graph. After creating the application, you need to grant it the necessary permissions to Microsoft Graph. Select **API Permissions** in the leftmost navigation panel.

The screenshot shows the Azure Active Directory admin center interface. In the top navigation bar, the path is 'Dashboard > Contoso - App registrations > Graph Console App'. On the left sidebar, under the 'Graph Console App' section, there are several menu items: Overview, Quickstart, Manage, Branding, Authentication, Certificates & secrets, API permissions (which is highlighted with a red box), Expose an API, Owners, Roles and administrators (Preview), and Manifest.

13. Select the **Add a permission** button.

The screenshot shows the 'API permissions' page for the 'Graph Console App'. At the top, it says 'Applications are authorized to call APIs when they are granted permissions by users/admins as part of the all the permissions the application needs.' Below this is a table with one row:

API / PERMISSIONS NAME	TYPE	DESCRIPTION
Microsoft Graph (1)	User.Read	Delegated Sign in and read user profile

Below the table, it says 'These are the permissions that this application requests statically. You may also request user consentable permissions dynamically through code. See best practices for requesting permissions'.

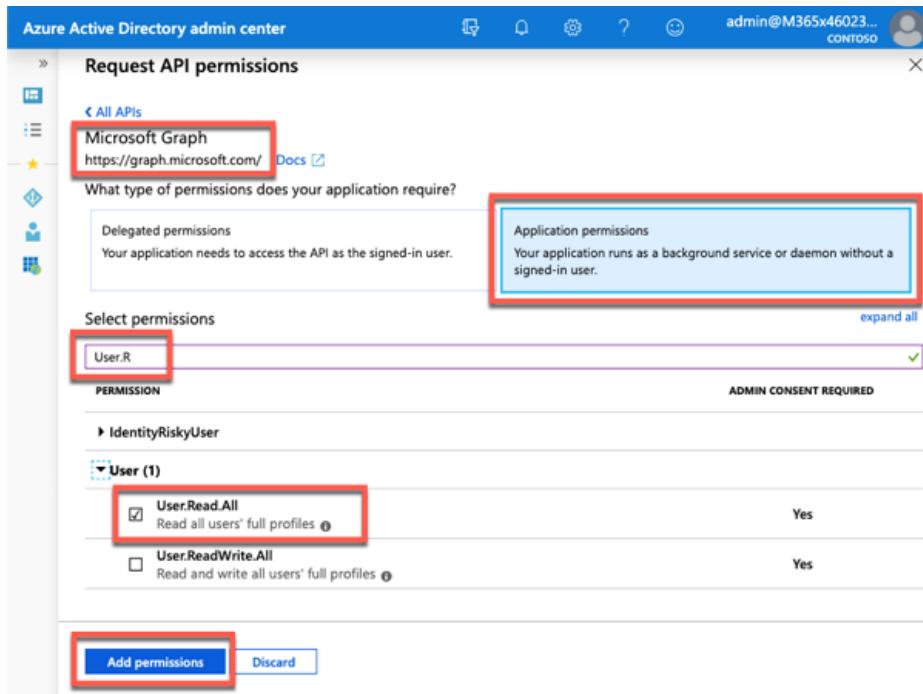
14. In the Request API permissions panel that appears, select Microsoft Graph from the Microsoft APIs tab.

The screenshot shows the 'Request API permissions' dialog box. At the top, it says 'Select an API' and has tabs for 'Microsoft APIs' (which is highlighted with a red box), 'APIs my organization uses', and 'My APIs'. Below this is a section titled 'Commonly used Microsoft APIs' with a box around the 'Microsoft Graph' section:

**Microsoft Graph**  
Take advantage of the tremendous amount of data in Office 365, Enterprise Mobility + Security, and Windows 10. Access Azure AD, Excel, Intune, Outlook/Exchange, OneDrive, OneNote, SharePoint, Planner, and more through a single endpoint.

Icons for other services are shown: Azure Rights Management Services, Azure Service Management, and Azure Storage.

15. When prompted for the type of permission, select **Application permissions**.



16. Enter **User.R** in the **Select permissions** search box and select the **User.Read.All** permission, followed by the **Add permission** button at the bottom of the panel.
17. Under the **Configured permissions** section, select the button **Grant admin consent for [tenant]**, followed by the **Yes** button to grant all users in your organization this permission.

## Task 2: Create .NET Core console application

1. Open your command prompt, navigate to a directory where you have rights to create your project, and run the following command to create a new .NET Core console application:  
`dotnet new console -o graphconsoleapp`
2. After creating the application, run the following commands to ensure your new project runs correctly.

```
dotnetcli cd graphconsoleapp dotnet add package Microsoft.Identity.Client  
dotnet add package Microsoft.Graph dotnet add package  
Microsoft.Extensions.Configuration dotnet add package  
Microsoft.Extensions.Configuration.FileExtensions dotnet add package  
Microsoft.Extensions.Configuration.Json
```

3. Open the application in Visual Studio Code using the following command: `code .`
4. If Visual Studio Code displays a dialog box asking if you want to add required assets to the project, select **Yes**.

## Task 3: Update the console app to support Azure AD authentication

1. Create a new file named **appsettings.json** in the root of the project and add the following code to it:

```
json { "tenantId": "YOUR_TENANT_ID_HERE", "applicationId":  
"YOUR_APP_ID_HERE", "applicationSecret": "YOUR_APP_SECRET_HERE",  
"redirectUri": "YOUR_REDIRECT_URI_HERE" }
```

2. Update properties with the following values:

- **YOUR\_TENANT\_ID\_HERE**: Azure AD directory ID
- **YOUR\_APP\_ID\_HERE**: Azure AD client ID
- **YOUR\_APP\_SECRET\_HERE**: Azure AD client secret
- **YOUR\_REDIRECT\_URI\_HERE**: redirect URI you entered when creating the Azure AD app (*for example, https://localhost*)

## Task 4: Create helper classes

1. Create a new folder **Helpers** in the project.
2. Create a new file **AuthHandler.cs** in the **Helpers** folder and add the following code:

```
csharp using System.Net.Http; using System.Threading.Tasks; using Microsoft.Graph; using System.Threading; namespace Helpers { public class AuthHandler : DelegatingHandler { private IAuthenticationProvider _authenticationProvider; public AuthHandler(IAuthenticationProvider authenticationProvider, HttpMessageHandler innerHandler) { InnerHandler = innerHandler; _authenticationProvider = authenticationProvider; } protected override async Task<HttpResponseMessage> SendAsync(HttpRequestMessage request, CancellationToken cancellationToken) { await _authenticationProvider.AuthenticateRequestAsync(request); return await base.SendAsync(request, cancellationToken); } } }
```

3. Create a new file **MsalAuthenticationProvider.cs** in the **Helpers** folder and add the following code:

```
csharp using System.Net.Http; using System.Net.Http.Headers; using System.Threading.Tasks; using Microsoft.Identity.Client; using Microsoft.Graph; namespace Helpers { public class MsalAuthenticationProvider : IAuthenticationProvider { private IConfidentialClientApplication _clientApplication; private string[] _scopes; public MsalAuthenticationProvider(IConfidentialClientApplication clientApplication, string[] scopes) { _clientApplication = clientApplication; _scopes = scopes; } public async Task AuthenticateRequestAsync(HttpRequestMessage request) { var token = await GetTokenAsync(); request.Headers.Authorization = new AuthenticationHeaderValue("bearer", token); } public async Task<string> GetTokenAsync() { AuthenticationResult authResult = null; authResult = await _clientApplication.AcquireTokenForClient(_scopes).ExecuteAsync(); return authResult.AccessToken; } } }
```

## Task 5: Incorporate Microsoft Graph into the console app

1. Open the **Program.cs** file and add the following using statements to the top of the file below using System:

```
csharp using System.Collections.Generic; using Microsoft.Identity.Client;  
using Microsoft.Graph; using Microsoft.Extensions.Configuration; using  
Helpers;
```

2. Add the following static member to the Program class in the **Program.cs** file. This member will be used to instantiate the client used to call Microsoft Graph:

```
csharp private static GraphServiceClient _graphClient;
```

3. Add the following method **LoadAppSettings** to the **Program** class. This method retrieves the configuration details from the **appsettings.json** file previously created:

```
csharp private static IConfigurationRoot LoadAppSettings() { try { var  
config = new ConfigurationBuilder()  
.SetBasePath(System.IO.Directory.GetCurrentDirectory())  
.AddJsonFile("appsettings.json", false, true) .Build(); if  
(string.IsNullOrEmpty(config["applicationId"])) ||  
string.IsNullOrEmpty(config["applicationSecret"]) ||  
string.IsNullOrEmpty(config["redirectUri"]) ||  
string.IsNullOrEmpty(config["tenantId"])) { return null; } return config; }  
catch (System.IO.FileNotFoundException) { return null; } }
```

4. Add the following method **CreateAuthorizationProvider** to the **Program** class. This method will create an instance of the clients used to call Microsoft Graph:

```
csharp private static IAuthenticationProvider  
CreateAuthorizationProvider(IConfigurationRoot config) { var clientId =  
config["applicationId"]; var clientSecret = config["applicationSecret"];  
var redirectUri = config["redirectUri"]; var authority =  
$"https://login.microsoftonline.com/{config["tenantId"]}/v2.0";  
List<string> scopes = new List<string>();  
scopes.Add("https://graph.microsoft.com/.default"); var cca =  
ConfidentialClientApplicationBuilder.Create(clientId)  
.WithAuthority(authority) .WithRedirectUri(redirectUri)  
.WithClientSecret(clientSecret) .Build(); return new  
MsalAuthenticationProvider(cca, scopes.ToArray()); }
```

5. Add the following method **GetAuthenticatedGraphClient** to the **Program** class. This method creates an instance of the **GraphServiceClient** object:

```
csharp private static GraphServiceClient  
GetAuthenticatedGraphClient(IConfigurationRoot config) { var  
authenticationProvider = CreateAuthorizationProvider(config); _graphClient  
= new GraphServiceClient(authenticationProvider); return _graphClient; }
```

6. Locate the **Main** method in the **Program** class. Add the following code to the end of the **Main** method to load the configuration settings from the **appsettings.json** file:

```
csharp var config = LoadAppSettings(); if (config == null) {  
Console.WriteLine("Invalid appsettings.json file."); return; }
```

7. Add the following code to the end of the **Main** method, just after the code added in the last step. This code will obtain an authenticated instance of the **GraphServicesClient** and submit a request for the first user.

```
csharp var client = GetAuthenticatedGraphClient(config); var graphRequest =  
client.Users.Request(); var results = graphRequest.GetAsync().Result;  
foreach(var user in results) { Console.WriteLine(user.Id + ": " +  
user.DisplayName + " <" + user.Mail + ">"); } Console.WriteLine("\nGraph  
Request:");  
Console.WriteLine(graphRequest.GetHttpRequestMessage() .RequestUri);
```

## Task 6: Build and test the application

1. Run the following command in a command prompt to compile the console application:  
dotnet build

2. Run the following command to run the console application: dotnet run

3. When the application runs, you'll see a list of users displayed. The query retrieved all information about the users.

```
Hello World!
f783718b-65e1-4a10-898a-fff0a217ab41: Conf Room Adams <Adams@M365x460234.onmicrosoft.com>
7347eda9-9cd3-4e12-ab0f-a548296ec4b5: Adele Vance <AdeleV@M365x460234.OnMicrosoft.com>
cd2196f2-2643-4995-86be-337dda89371c: MOD Administrator <admin@M365x460234.OnMicrosoft.com>
68b23cce-2ca8-4aae-9816-4617b5cc4226: Alex Wilber <AlexW@M365x460234.OnMicrosoft.com>
74a31520-6eae-437a-ad0-6e7082fac0a2: Allan Deyoung <AllanD@M365x460234.OnMicrosoft.com>
4b9785a6-ab4b-4ef4-fa4fb-8db740b3722a: Conf Room Baker <Baker@M365x460234.onmicrosoft.com>
b0e4b027-0885-4ff9-ae52-6bfd12d14c6c: Bianca Pisani <>
131cad4a-b855-4089-81cb-5635dee3cca5: Brian Johnson (TAILSPIN) <BrianJ@M365x460234.onmicrosoft.com>
67fed7f6-126c-4d67-b0aa-9c5a8981e75d: Cameron White <>
cc9a10db-a351-4d96-8d7-c403b9783575: Christie Cline <ChristieC@M365x460234.OnMicrosoft.com>
8741baed-0302-421b-bf2b-1e1cd273fd61: Conf Room Crystal <Crystal@M365x460234.onmicrosoft.com>
a60f8373-c1c7-4db8-b120-8f6fa876fc2: Debra Berger <DebraB@M365x460234.OnMicrosoft.com>
0f941563-adc5-4078-951f-30b2685b726e: Delia Dennis <>
d4a42e0c-7be5-4324-8fd0-ebc63efeb3c3: Diego Siciliani <DiegoS@M365x460234.OnMicrosoft.com>
75d965c4-59b8-4e22-bc45-4d7c32e7d8fe: Emily Braun <EmilyB@M365x460234.OnMicrosoft.com>
577914b1-a163-419f-8009-0a1fa69ce0b7: Enrico Cattaneo <EnricoC@M365x460234.OnMicrosoft.com>
5964f551-5bf8-4ab7-a0a2-cbb06ca9ea9b: Gerhart Moller <>
80620a39-a095-4bd9-9e82-13972cc7d6e1: Grady Archie <GradyA@M365x460234.OnMicrosoft.com>
295047c6-af95-4bdf-a260-b769c4b116fe: Henrietta Mueller <HenriettaM@M365x460234.OnMicrosoft.com>
c66ddbec-f041-4e1f-927e-640bb62e62fe: Conf Room Hood <Hood@M365x460234.onmicrosoft.com>
56c8e653-5d9a-48a4-8fca-d8471245dfa: Irvin Sayers <IrvinS@M365x460234.OnMicrosoft.com>
ad112ce7-a8fc-4b48-bc66-9f906b2ceaf8: Isaiah Langer <IsaiahL@M365x460234.OnMicrosoft.com>
c4d0b574-580d-42cd-8f20-67a9fa0f4ba: Johanna Lorenz <JohannaL@M365x460234.OnMicrosoft.com>
0aca7432-77e8-4d5f-bae0-f2f5c24db3be: Joni Sherman <JoniS@M365x460234.OnMicrosoft.com>
c10de9f6-5e4d-4e15-ae44-cc581cebc5f2: Jordan Miller <JordanM@M365x460234.OnMicrosoft.com>
3ee7b72a-9585-4529-a3e6-85d48c7d4968: Lee Gu <LeeG@M365x460234.OnMicrosoft.com>
d2b03165-278d-4816-9dad-7340d36a0703: Lidia Holloway <LidiaH@M365x460234.OnMicrosoft.com>
53036a17-d314-4581-a880-fba271a1bbd1: Lynne Robbins <LynneR@M365x460234.OnMicrosoft.com>
bbc8446f-f788-4527-8cf6-c40e65228539: Megan Bowen <MeganB@M365x460234.OnMicrosoft.com>
526f3f98-09d2-4c98-8e64-0aa8e4128241: Miriam Graham <MiriamG@M365x460234.OnMicrosoft.com>
ea1ffc20-a151-425a-a5aa-1a639e9e1a79: Nestor Wilke <NestorW@M365x460234.OnMicrosoft.com>
12d13880-afb3-4721-a6d9-f296d1dfa1cf: Patti Fernandez <PattiF@M365x460234.OnMicrosoft.com>
5d3df140-5682-480c-98c8-fc9ec7bdbd2d: Pradeep Gupta <PradeepG@M365x460234.OnMicrosoft.com>
6589d3e8-1ebf-437a-aea3-55bcacf11d2: Conf Room Rainier <Rainier@M365x460234.onmicrosoft.com>
95c4855d-94a9-4ceb-bfd8-da93f83f63ab: Raul Razo <>
d3963c26-01b1-4cdd-af99-21ea89176353: Conf Room Stevens <Stevens@M365x460234.onmicrosoft.com>
```

**\*\*Note\*\*:**

Notice the URL written to the console. This is the entire request, including query parameters, that the Microsoft Graph SDK is generating. Take note for each query you run in this exercise.

## Task 7: Edit the application to optimize the query

The current console application isn't efficient because it retrieves all information about all users in your organization but only displays three properties. The **\$select** query parameter can limit the amount of data that is returned by Microsoft Graph, optimizing the query.

1. Update the line that starts with `var results = graphRequest` in the **Main** method with the following to limit the query to just two properties:

```
csharp var results = graphRequest .Select(u => new { u.DisplayName, u.Mail }) .GetAsync() .Result;
```

2. Rebuild and rerun the console application by executing the following commands in the command line:

```
csharp dotnet build dotnet run
```

3. Notice that the **ID** property isn't populated with data, as it wasn't included in the **\$select** query parameter.

```
Hello World!
: Conf Room Adams <Adams@M365x460234.onmicrosoft.com>
: Adele Vance <AdeleV@M365x460234.OnMicrosoft.com>
: MOD Administrator <admin@M365x460234.OnMicrosoft.com>
: Alex Wilber <AlexW@M365x460234.OnMicrosoft.com>
: Allan Deyoung <AllanD@M365x460234.OnMicrosoft.com>
: Conf Room Baker <Baker@M365x460234.onmicrosoft.com>
: Bianca Pisani <>
: Brian Johnson (TAILSPIN) <BrianJ@M365x460234.onmicrosoft.com>
: Cameron White <>
: Christie Cline <ChristieC@M365x460234.OnMicrosoft.com>
: Conf Room Crystal <Crystal@M365x460234.onmicrosoft.com>
: Debra Berger <DebraB@M365x460234.OnMicrosoft.com>
: Delia Dennis <>
: Diego Siciliani <DiegoS@M365x460234.OnMicrosoft.com>
: Emily Braun <EmilyB@M365x460234.OnMicrosoft.com>
: Enrico Cattaneo <EnricoC@M365x460234.OnMicrosoft.com>
: Gerhart Moller <>
: Grady Archie <GradyA@M365x460234.OnMicrosoft.com>
: Henrietta Mueller <HenriettaM@M365x460234.OnMicrosoft.com>
: Conf Room Hood <Hood@M365x460234.onmicrosoft.com>
: Irvin Sayers <IrvinS@M365x460234.OnMicrosoft.com>
: Isaiah Langer <IsaiahL@M365x460234.OnMicrosoft.com>
: Johanna Lorenz <JohannaL@M365x460234.OnMicrosoft.com>
: Joni Sherman <JoniS@M365x460234.OnMicrosoft.com>
: Jordan Miller <JordanM@M365x460234.OnMicrosoft.com>
: Lee Gu <LeeG@M365x460234.OnMicrosoft.com>
: Lidia Holloway <LidiaH@M365x460234.OnMicrosoft.com>
: Lynne Robbins <LynneR@M365x460234.OnMicrosoft.com>
: Megan Bowen <MeganB@M365x460234.OnMicrosoft.com>
: Miriam Graham <MiriamG@M365x460234.OnMicrosoft.com>
: Nestor Wilke <NestorW@M365x460234.OnMicrosoft.com>
: Patti Fernandez <PattiF@M365x460234.OnMicrosoft.com>
: Pradeep Gupta <PradeepG@M365x460234.OnMicrosoft.com>
: Conf Room Rainier <Rainier@M365x460234.onmicrosoft.com>
: Raul Razo <>
: Conf Room Stevens <Stevens@M365x460234.onmicrosoft.com>
```

4. Let us further limit the results to just the first 15 results. Update the line that starts with `var results = graphRequest` in the **Main** method with the following:

```
csharp var results = graphRequest .Select(u => new { u.DisplayName, u.Mail }) .Top(15) .GetAsync() .Result;
```

5. Rebuild and rerun the console application by executing the following commands in the command line:

```
csharp dotnet build dotnet run
```

6. Notice only 15 items are now returned by the query.

```
Hello World!
: Conf Room Adams <Adams@M365x460234.onmicrosoft.com>
: Adele Vance <AdeleV@M365x460234.OnMicrosoft.com>
: MOD Administrator <admin@M365x460234.OnMicrosoft.com>
: Alex Wilber <AlexW@M365x460234.OnMicrosoft.com>
: Allan Deyoung <AllanD@M365x460234.OnMicrosoft.com>
: Conf Room Baker <Baker@M365x460234.onmicrosoft.com>
: Bianca Pisani <>
: Brian Johnson (TAILSPIN) <BrianJ@M365x460234.onmicrosoft.com>
: Cameron White <>
: Christie Cline <ChristieC@M365x460234.OnMicrosoft.com>
: Conf Room Crystal <Crystal@M365x460234.onmicrosoft.com>
: Debra Berger <DebraB@M365x460234.OnMicrosoft.com>
: Delia Dennis <>
: Diego Siciliani <DiegoS@M365x460234.OnMicrosoft.com>
: Emily Braun <EmilyB@M365x460234.OnMicrosoft.com>
```

7. Sort the results in reverse alphabetic order. Update the line that starts with `var results = graphRequest` in the **Main** method with the following:

```
csharp var results = graphRequest .Select(u => new { u.DisplayName, u.Mail }) .Top(15) .OrderBy("DisplayName desc") .GetAsync() .Result;
```

8. Rebuild and rerun the console application by executing the following commands in the command line:

```
csharp dotnet build dotnet run
```

```
Hello World!
: Raul Razo <>
: Pradeep Gupta <PradeepG@M365x460234.OnMicrosoft.com>
: Patti Fernandez <PattiF@M365x460234.OnMicrosoft.com>
: Nestor Wilke <NestorW@M365x460234.OnMicrosoft.com>
: MOD Administrator <admin@M365x460234.OnMicrosoft.com>
: Miriam Graham <MiriamG@M365x460234.OnMicrosoft.com>
: Megan Bowen <MeganB@M365x460234.OnMicrosoft.com>
: Lynne Robbins <LynneR@M365x460234.OnMicrosoft.com>
: Lidia Holloway <LidiaH@M365x460234.OnMicrosoft.com>
: Lee Gu <LeeG@M365x460234.OnMicrosoft.com>
: Jordan Miller <JordanM@M365x460234.OnMicrosoft.com>
: Joni Sherman <JoniS@M365x460234.OnMicrosoft.com>
: Johanna Lorenz <JohannaL@M365x460234.OnMicrosoft.com>
: Isaiah Langer <IsaiahL@M365x460234.OnMicrosoft.com>
: Irvin Sayers <IrvinS@M365x460234.OnMicrosoft.com>
```

9. Further refine the results by selecting users whose surname starts with A, B, or C. You'll need to remove the `\$orderby` query parameter added previously as the `Users` endpoint doesn't support combining the `\$filter` and `\$orderby` parameters. Update the line that starts with `var results = graphRequest` in the **Main** method with the following:

```
csharp var results = graphRequest .Select(u => new { u.DisplayName, u.Mail }) .Top(15) // .OrderBy("DisplayName desc") .Filter("startsWith(surname,'A') or startsWith(surname,'B') or startsWith(surname,'C')") .GetAsync() .Result;
```

10. Rebuild and rerun the console application by executing the following commands in the command line:

```
powershell dotnet build dotnet run
```

```
Hello World!
: Debra Berger <DebraB@M365x460234.OnMicrosoft.com>
: Megan Bowen <MeganB@M365x460234.OnMicrosoft.com>
: Emily Braun <EmilyB@M365x460234.OnMicrosoft.com>
: MOD Administrator <admin@M365x460234.OnMicrosoft.com>
: Grady Archie <GradyA@M365x460234.OnMicrosoft.com>
: Enrico Cattaneo <EnricoC@M365x460234.OnMicrosoft.com>
: Christie Cline <ChristieC@M365x460234.OnMicrosoft.com>
```

## Review

In this exercise, you created an Azure AD application and .NET console application that retrieved user data from Microsoft Graph. You then used query parameters to limit and manipulate the data returned by Microsoft Graph to optimize the query.

### ◆◆◆# Exercise 3: Using change notifications and track changes with Microsoft Graph

This tutorial teaches you how to build a .NET Core app that uses the Microsoft Graph API to receive notifications (webhooks) when a user account changes in Azure AD and perform queries using the delta query API to receive all changes to user accounts since the last query was made.

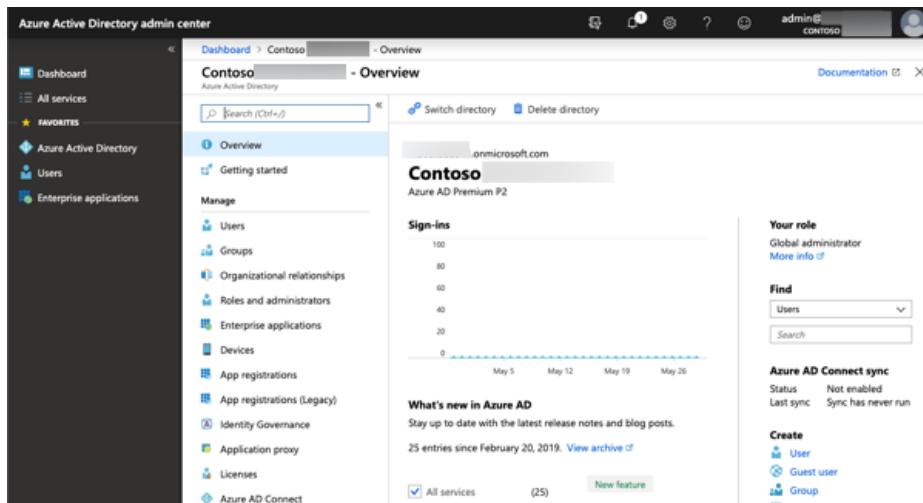
By the end of this exercise you will be able to:

- Monitor for changes using change notifications
- Get changes using a delta query

# Task 1: Create a new Azure AD web application

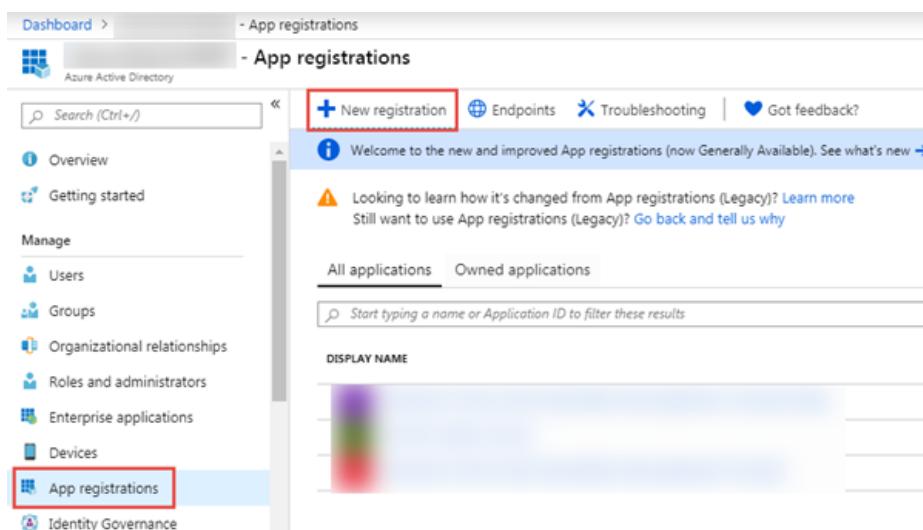
In this exercise, you will create a new Azure AD web application registration using the Azure AD admin center and grant administrator consent to the required permission scopes.

1. Open a browser and navigate to the Azure AD admin center <https://aad.portal.azure.com>. Sign in using a **Work or School Account**.
2. Select **Azure Active Directory** in the leftmost navigation panel, then select **App registrations** under **Manage**.



The screenshot shows the Azure Active Directory admin center interface. On the left, there's a navigation sidebar with links like Dashboard, All services, Favorites (Azure Active Directory, Users, Enterprise applications), and Manage (Users, Groups, Organizational relationships, Roles and administrators, Enterprise applications, Devices, App registrations, App registrations (Legacy), Identity Governance, Application proxy, Licenses, Azure AD Connect). The main content area is titled 'Contoso - Overview' and shows a chart for 'Sign-ins' from May 5 to May 26. It also includes sections for 'What's new in Azure AD' (with a link to 'View archive'), 'Azure AD Connect sync' (status: Not enabled, last sync: Sync has never run), and 'Create' (User, Guest user, Group) buttons. The 'App registrations' link in the Manage section is highlighted with a red box.

3. Select **New registration** on the Register an application page.



The screenshot shows the 'App registrations' page. The left sidebar has the 'App registrations' link highlighted with a red box. The main area has a heading 'Welcome to the new and improved App registrations (now Generally Available). See what's new →'. It includes a note about learning how it's changed from App registrations (Legacy) with a 'Learn more' link and an option to 'Go back and tell us why'. Below this are tabs for 'All applications' and 'Owned applications', a search bar 'Start typing a name or Application ID to filter these results', and a 'DISPLAY NAME' input field containing a blurred color swatch.

4. Set the values as follows:

- **Name:** Graph Notification Tutorial
- **Supported account types:** Accounts in any organizational directory and personal Microsoft accounts
- **Redirect URI:** Web > <http://localhost>

\* Name  
The user-facing display name for this application (this can be changed later).  
 ✓

#### Supported account types

Who can use this application or access this API?

- Accounts in this organizational directory only (Contoso only - Single tenant)
- Accounts in any organizational directory (Any Azure AD directory - Multitenant)
- Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)
- Personal Microsoft accounts only

[Help me choose...](#)

#### Redirect URI (optional)

We'll return the authentication response to this URI after successfully authenticating the user. Providing this now is optional and it can be changed later, but a value is required for most authentication scenarios.

✓

## 5. Select Register.

6. On the **Graph Notification Tutorial** page, copy the value of the **Application (client) ID** and **Directory (tenant) ID**; save it, you will need them later in the tutorial.

Dashboard > Contoso M365x995941 - App registrations > GraphNotificationTutorial

**GraphNotificationTutorial**

Display name : GraphNotificationTutorial

Application (client) ID : a6079933-...-bcc77ebd0770

Directory (tenant) ID : 785b4ab0-...-c6502a7da2af

Object ID : 31500560-f812-4090-b806-97c6035778b4

Supported account types : All Microsoft account users

Redirect URIs : 1 web, 0 public client

Managed application in ... : GraphNotificationTutorial

## 7. Select Manage > Certificates & secrets.

### 8. Select New client secret.

9. Enter a value in **Description**, select one of the options for **Expires**, and select **Add**.

GraphNotificationTutorial - Certificates & secrets

Overview Quickstart

Manage

- Branding
- Authentication
- Certificates & secrets**
- API permissions
- Expose an API
- Owners
- Manifest

Certificates

Certificates can be used as secrets to prove the application's identity when requesting a token. Also can be referred to as public keys.

**+ Upload certificate**

THUMPRINT	START DATE	EXPIRES
No certificates have been added for this application.		

Client secrets

A secret string that the application uses to prove its identity when requesting a token. Also can be referred to as application password.

**+ New client secret**

DESCRIPTION	EXPIRES	VALUE
No client secrets have been created for this application.		

10. Copy the client secret value before you leave this page. You will need it later in the tutorial.

**Important:** This client secret is never shown again, so make sure you copy it now.

DESCRIPTION	EXPIRES	VALUE
Forever	12/31/2299	v7NSTLG5Cb[t no_S ]l p\$

11. Select **Manage > API Permissions**.

12. Select **Add a permission** and select **Microsoft Graph**.

13. Select **Application Permission**, expand the **User** group, and select **User.Read.All** scope.

14. Select **Add permissions** to save your changes.

The screenshot shows the 'Request API permissions' page for the 'GraphNotificationTutorial' application. In the 'Select permissions' section, under the 'User' category, the 'User.Read.All' permission is checked and highlighted with a red box. Other listed permissions include 'User.Export.All', 'User.Invite.All', and 'User.ReadWrite.All'. The 'Add permissions' and 'Discard' buttons are at the bottom.

#### Configured permissions

Applications are authorized to call APIs when they are granted permissions by users/admins as part of the consent process. The list of configured permissions should include all the permissions the application needs. [Learn more about permissions and consent](#)

<a href="#">+ Add a permission</a>	<a href="#">Grant admin consent for</a>				
API / Permissions name	Type	Description	Admin consent req...	Status	...
<b>▼ Microsoft Graph (2)</b>					
User.Read	Delegated	Sign in and read user profile	-	<span style="color: green;">✓</span> Granted for [tenant]	...
User.Read.All	Application	Read all users' full profiles	Yes	<span style="color: orange;">⚠</span> Not granted for [tenant]	...

15. The application requests an application permission with the **User.Read.All** scope. This permission requires administrative consent.

16. Select **Grant admin consent for Contoso**, then select **Yes** to consent this application, and grant the application access to your tenant using the scopes you specified.

#### Configured permissions

Applications are authorized to call APIs when they are granted permissions by users/admins as part of the consent process. The list of configured permissions should include all the permissions the application needs. [Learn more about permissions and consent](#)

<a href="#">+ Add a permission</a>	<a href="#">Grant admin consent for</a>				
API / Permissions name	Type	Description	Admin consent req...	Status	...
<b>▼ Microsoft Graph (2)</b>					
User.Read	Delegated	Sign in and read user profile	-	<span style="color: green;">✓</span> Granted for [tenant]	...
User.Read.All	Application	Read all users' full profiles	Yes	<span style="color: green;">✓</span> Granted for [tenant]	...

## Task 2: Create .NET Core App

### Run ngrok

In order for the Microsoft Graph to send notifications to your application running on your development machine you need to use a tool such as ngrok to tunnel calls from the internet to your development machine. Ngrok allows calls from the internet to be directed to your application running locally without needing to create firewall rules.

**NOTE:** Graph requires using https and this lab uses ngrok free.

If you run into any issues please visit [Using ngrok to get a public HTTPS address for a local server already serving HTTPS \(for free\)](#).

1. Run ngrok by executing the following from the command line:

```
powershell ngrok http 5000
```

2. This will start ngrok and will tunnel requests from an external ngrok url to your development machine on port 5000. Copy the https forwarding address. In the example below that would be https://787b8292.ngrok.io. You will need this later.

### Create .NET Core WebApi App

1. Open your command prompt, navigate to a directory where you have rights to create your project, and run the following command to create a new .NET Core console application:  
`dotnet new webapi -o msgraphapp`

2. After creating the application, run the following commands to ensure your new project runs correctly:

```
powershell cd msgraphapp dotnet add package Microsoft.Identity.Client  
dotnet add package Microsoft.Graph dotnet run
```

3. The application will start and output the following:

```
powershell info: Microsoft.Hosting.Lifetime[0] Now listening on:  
https://localhost:5001 info: Microsoft.Hosting.Lifetime[0] Now listening  
on: http://localhost:5000 info: Microsoft.Hosting.Lifetime[0] Application  
started. Press Ctrl+C to shut down. info: Microsoft.Hosting.Lifetime[0]  
Hosting environment: Development info: Microsoft.Hosting.Lifetime[0]  
Content root path: [your file path]\msgraphapp
```

4. Stop the application running by pressing **CTRL+C**.

5. Open the application in Visual Studio Code using the following command: `code .`

6. If Visual Studio code displays a dialog box asking if you want to add required assets to the project, select Yes.

## Task 3: Code the HTTP API

Open the **Startup.cs** file and comment out the following line to disable ssl redirection.

```
csharp //app.UseHttpsRedirection();
```

### Add model classes

The application uses several new model classes for (de)serialization of messages to/from the Microsoft Graph.

1. Right-click in the project file tree and select New Folder. Name it **Models**

2. Right-click the Models folder and add three new files:

- **Notification.cs**
- **ResourceData.cs**
- **MyConfig.cs**

3. Replace the contents of **Notification.cs** with the following:

```
csharp using Newtonsoft.Json; using System; namespace msgraphapp.Models {  
    public class Notifications { [JsonProperty(PropertyName = "value")] public  
        Notification[] Items { get; set; } } // A change notification.  
    public class Notification { // The type of change. [JsonProperty(PropertyName =  
        "changeType")] public string ChangeType { get; set; } // The client state  
        used to verify that the notification is from Microsoft Graph. Compare the  
        value received with the notification to the value you sent with the  
        subscription request. [JsonProperty(PropertyName = "clientState")] public  
        string ClientState { get; set; } // The endpoint of the resource that  
        changed. For example, a message uses the format ..//Users/{user-  
        id}/Messages/{message-id} [JsonProperty(PropertyName = "resource")] public  
        string Resource { get; set; } // The UTC date and time when the webhooks  
        subscription expires. [JsonProperty(PropertyName =  
            "subscriptionExpirationDateTime")] public DateTimeOffset  
        SubscriptionExpirationDateTime { get; set; } // The unique identifier for  
        the webhooks subscription. [JsonProperty(PropertyName = "subscriptionId")]  
        public string SubscriptionId { get; set; } // Properties of the changed  
        resource. [JsonProperty(PropertyName = "resourceData")] public ResourceData  
        ResourceData { get; set; } }
```

4. Replace the contents of **ResourceData.cs** with the following:

```
csharp using Newtonsoft.Json; namespace msgraphapp.Models { public class  
    ResourceData { // The ID of the resource. [JsonProperty(PropertyName =  
        "id")] public string Id { get; set; } // The OData etag property.  
        [JsonProperty(PropertyName = "@odata.etag")] public string ODataEtag { get;  
        set; } // The OData ID of the resource. This is the same value as the  
        resource property. [JsonProperty(PropertyName = "@odata.id")] public string  
        ODataId { get; set; } // The OData type of the resource:  
        "#Microsoft.Graph.Message", "#Microsoft.Graph.Event", or  
        "#Microsoft.Graph.Contact". [JsonProperty(PropertyName = "@odata.type")]  
        public string ODataType { get; set; } }
```

5. Replace the contents of **MyConfig.cs** with the following:

```
csharp namespace msgraphapp { public class MyConfig { public string AppId { get; set; } public string AppSecret { get; set; } public string TenantId { get; set; } public string Ngrok { get; set; } } }
```

6. Open the **Startup.cs** file. Locate the method **ConfigureServices()** method and replace it with the following code:

```
csharp public void ConfigureServices(IServiceCollection services) { services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_3_0); var config = new MyConfig(); Configuration.Bind("MyConfig", config); services.AddSingleton(config); }
```

7. Open the **appsettings.json** file and replace the content with the following JSON.

```
csharp { "Logging": { "LogLevel": { "Default": "Warning" } }, "MyConfig": { "AppId": "<APP ID>", "AppSecret": "<APP SECRET>", "TenantId": "<TENANT ID>", "Ngrok": "<NGROK URL>" } }
```

8. Replace the following variables with the values you copied earlier:

- should be set to the **https ngrok url** you copied earlier.
- should be your **Office 365 tenant id** you copied earlier.
- and should be the **application id** and **secret** you copied earlier when you created the application registration.

## Add notification controller

The application requires a new controller to process the subscription and notification. 1. Right-click the **Controllers** folder, select **New File**, and name the controller **NotificationsController.cs**.

1. Replace the contents of **NotificationController.cs** with the following code:

```
csharp using System; using System.Collections.Generic; using System.IO; using System.Linq; using System.Net.Http; using System.Threading.Tasks; using Microsoft.AspNetCore.Mvc; using msgraphapp.Models; using Newtonsoft.Json; using System.Net; using System.Threading; using Microsoft.Graph; using Microsoft.Identity.Client; using System.Net.Http.Headers; namespace msgraphapp.Controllers { [Route("api/[controller]")] [ApiController] public class NotificationsController : ControllerBase { private readonly MyConfig config; public NotificationsController(MyConfig config) { this.config = config; } [HttpGet] public async Task<ActionResult<string>> Get() { var graphServiceClient = GetGraphClient(); var sub = new Microsoft.Graph.Subscription(); sub.ChangeType = "updated"; sub.NotificationUrl = config.Ngrok + "/api/notifications"; sub.Resource = "/users"; sub.ExpirationDateTime = DateTime.UtcNow.AddMinutes(5); sub.ClientState = "SecretClientState"; var newSubscription = await graphServiceClient.Subscriptions.Request().AddAsync(sub); return $"Subscribed. Id: {newSubscription.Id}, Expiration: {newSubscription.ExpirationDateTime}"; } public async Task<ActionResult<string>> Post([FromQuery]string validationToken = null) { // handle validation if(!string.IsNullOrEmpty(validationToken)) { Console.WriteLine($"Received Token: '{validationToken}'"); return Ok(validationToken); } // handle notifications using (StreamReader reader = new StreamReader(Request.Body)) { string content = await reader.ReadToEndAsync(); Console.WriteLine(content); var notifications = JsonConvert.DeserializeObject<Notifications>(content); foreach(var notification in notifications.Items) { Console.WriteLine($"Received notification: '{notification.Resource}', {notification.ResourceData?.Id}"); }}
```

```
    } } return Ok(); } private GraphServiceClient GetGraphClient() { var  
graphClient = new GraphServiceClient(new  
DelegateAuthenticationProvider((requestMessage) => { // get an access token  
for Graph var accessToken = GetAccessToken().Result; requestMessage  
.Headers .Authorization = new AuthenticationHeaderValue("bearer",  
accessToken); return Task.FromResult(0); })); return graphClient; } private  
async Task<string> GetAccessToken() { IConfidentialClientApplication app =  
ConfidentialClientApplicationBuilder.Create(config.AppId)  
.WithClientSecret(config.AppSecret)  
.WithAuthority($"https://login.microsoftonline.com/{config.TenantId}")  
.WithRedirectUri("https://daemon") .Build(); string[] scopes = new string[]  
{ "https://graph.microsoft.com/.default" }; var result = await  
app.AcquireTokenForClient(scopes).ExecuteAsync(); return  
result.AccessToken; } } }
```

2. **Save** all files.

## Task 4: Run the application

Update the Visual Studio debugger launch configuration:

**Note:** By default, the .NET Core launch configuration will open a browser and navigate to the default URL for the application when launching the debugger. For this application, we instead want to navigate to the NGrok URL. If you leave the launch configuration as is, each time you debug the application it will display a broken page. You can just change the URL, or change the launch configuration to not launch the browser:

1. In Visual Studio Code, open the file **.vscode/launch.json**.

2. Delete the following section in the default configuration:

```
json // Enable launching a web browser when ASP.NET Core starts. For more
information: https://aka.ms/VSCode-CS-LaunchJson-WebBrowser
"serverReadyAction": { "action": "openExternally", "pattern": "^\s*Now
listening on:\s+(https?://\S+)" },
```

3. **Save** your changes.

4. In **Visual Studio Code**, select **Debug > Start debugging** to run the application. VS Code will build and start the application.

5. Once you see messages in the Debug Console window that states the application started proceed with the next steps.

6. Open a browser and navigate to **http://localhost:5000/api/notifications** to subscribe to change notifications. If successful you will see output that includes a subscription id.

**Note:** Your application is now subscribed to receive notifications from the Microsoft Graph when an update is made on any user in the Office 365 tenant.

7. Trigger a notification:

1. Open a browser and navigate to the **Microsoft 365 admin center** (<https://admin.microsoft.com/AdminPortal>).

2. If prompted to login, sign-in using an admin account.

3. Select **Users > Active users**.

8. Select an active user and select **Edit** for their **Contact information**.

9. Update the **Phone number** value with a new number and select **Save**.

10. In the **Visual Studio Code Debug Console**, you will see a notification has been received. Sometimes this may take a few minutes to arrive. An example of the output is below:

```
json Received notification: 'Users/7a7fded6-0269-42c2-a0be-512d58da4463',
7a7fded6-0269-42c2-a0be-512d58da4463
```

This indicates the application successfully received the notification from the Microsoft Graph for the user specified in the output. You can then use this information to query the Microsoft Graph for the users full details if you want to synchronize their details into your application.

## Task 5: Manage notification subscriptions

Subscriptions for notifications expire and need to be renewed periodically. The following steps will demonstrate how to renew notifications:

### Update Code

1. Open **Controllers > NotificationsController.cs** file

2. Add the following two member declarations to the **NotificationsController** class:

```
csharp private static Dictionary<string, Subscription> Subscriptions = new  
Dictionary<string, Subscription>(); private static Timer subscriptionTimer  
= null;
```

3. Add the following new methods. These will implement a background timer that will run every 15 seconds to check if subscriptions have expired. If they have, they will be renewed.

```
csharp private void CheckSubscriptions(Object stateInfo) { AutoResetEvent  
autoEvent = (AutoResetEvent)stateInfo; Console.WriteLine($"Checking  
subscriptions {DateTime.Now.ToString("h:mm:ss.fff")});  
Console.WriteLine($"Current subscription count {Subscriptions.Count()});  
foreach(var subscription in Subscriptions) { // if the subscription expires  
in the next 2 min, renew it if(subscription.Value.ExpirationDateTime <  
DateTime.UtcNow.AddMinutes(2)) { RenewSubscription(subscription.Value); } }  
} private async void RenewSubscription(Subscription subscription) {  
Console.WriteLine($"Current subscription: {subscription.Id}, Expiration:  
{subscription.ExpirationDateTime}"); var graphServiceClient =  
GetGraphClient(); var newSubscription = new Subscription {  
ExpirationDateTime = DateTime.UtcNow.AddMinutes(5) }; await  
graphServiceClient .Subscriptions[subscription.Id] .Request()  
.UpdateAsync(newSubscription); subscription.ExpirationDateTime =  
newSubscription.ExpirationDateTime; Console.WriteLine($"Renewed  
subscription: {subscription.Id}, New Expiration:  
{subscription.ExpirationDateTime}"); }
```

4. The **CheckSubscriptions** method is called every 15 seconds by the timer. For production use this should be set to a more reasonable value to reduce the number of unnecessary calls to Microsoft Graph. The **RenewSubscription** method renews a subscription and is only called if a subscription is going to expire in the next two minutes.

5. Locate the method **Get()** and replace it with the following code:

```
csharp [HttpGet] public async Task<ActionResult<string>> Get() { var  
graphServiceClient = GetGraphClient(); var sub = new  
Microsoft.Graph.Subscription(); sub.ChangeType = "updated";  
sub.NotificationUrl = config.Ngrok + "/api/notifications"; sub.Resource =  
"/users"; sub.ExpirationDateTime = DateTime.UtcNow.AddMinutes(5);  
sub.ClientState = "SecretClientState"; var newSubscription = await  
graphServiceClient .Subscriptions .Request() .AddAsync(sub);  
Subscriptions[newSubscription.Id] = newSubscription; if (subscriptionTimer  
== null) { subscriptionTimer = new Timer(CheckSubscriptions, null, 5000,  
15000); } return $"Subscribed. Id: {newSubscription.Id}, Expiration:  
{newSubscription.ExpirationDateTime}"; }
```

### Test the changes

1. Within Visual Studio Code, select Debug > Start debugging to run the application.  
Navigate to the following url: **http://localhost:5000/api/notifications**. This will register a new subscription.
2. In the Visual Studio Code Debug Console window, approximately every 15 seconds, notice the timer checking the subscription for expiration:

```
powershell Checking subscriptions 12:32:51.882 Current subscription count 1
```

3. Wait a few minutes and you will see the following when the subscription needs renewing:

```
powershell Renewed subscription: 07ca62cd-1a1b-453c-be7b-4d196b3c6b5b, New  
Expiration: 3/10/2019 7:43:22 PM +00:00
```

This indicates that the subscription was renewed and shows the new expiry time.

## Task 5: Query for changes

Microsoft Graph offers the ability to query for changes to a particular resource since you last called it. Using this option, combined with Change Notifications, enables a robust pattern for ensuring you don't miss any changes to the resources. 1. Locate and open the following controller: **Controllers > NotificationsController.cs**. Add the following code to the existing **NotificationsController** class.

```
```csharp
private static object DeltaLink = null;
private static IUserDeltaCollectionPage lastPage = null;
private async Task CheckForUpdates()
{
    var graphClient = GetGraphClient();
    // get a page of users
    var users = await GetUsers(graphClient, DeltaLink);
    OutputUsers(users);
    // go through all of the pages so that we can get the delta link on the
    last page.
    while (users.NextPageRequest != null)
    {
        users = users.NextPageRequest.GetAsync().Result;
        OutputUsers(users);
    }
    object deltaLink;
    if (users.AdditionalData.TryGetValue("@odata.deltaLink", out deltaLink))
    {
        DeltaLink = deltaLink;
    }
}
private void OutputUsers(IUserDeltaCollectionPage users)
{
    foreach(var user in users)
    {
        var message = $"User: {user.Id}, {user.GivenName} {user.Surname}";
        Console.WriteLine(message);
    }
}
private async Task<IUserDeltaCollectionPage> GetUsers(GraphServiceClient
graphClient, object deltaLink)
{
    IUserDeltaCollectionPage page;
    if (lastPage == null)
    {
        page = await graphClient
            .Users
            .Delta()
            .Request()
            .GetAsync();
    }
    else
    {
        lastPage.InitializeNextPageRequest(graphClient, deltaLink.ToString());
        page = await lastPage.NextPageRequest.GetAsync();
    }
    lastPage = page;
    return page;
}
```

```

1. This code includes a new method, **CheckForUpdates()**, that will call the Microsoft Graph using the delta url and then pages through the results until it finds a new **deltalink** on the

final page of results. It stores the url in memory until the code is notified again when another notification is triggered.

2. Locate the existing **Post()** method and replace it with the following code:

```
csharp public async Task<ActionResult<string>> Post([FromQuery]string validationToken = null) { // handle validation if (!string.IsNullOrEmpty(validationToken)) { Console.WriteLine($"Received Token: '{validationToken}'"); return Ok(validationToken); } // handle notifications using (StreamReader reader = new StreamReader(Request.Body)) { string content = await reader.ReadToEndAsync(); Console.WriteLine(content); var notifications = JsonConvert.DeserializeObject<Notifications>(content); foreach (var notification in notifications.Items) { Console.WriteLine($"Received notification: '{notification.Resource}', {notification.ResourceData?.Id}"); } } // use deltaquery to query for all updates await CheckForUpdates(); return Ok(); }
```

3. The Post method will now call **CheckForUpdates** when a notification is received. **Save** all files.

## Test the changes

1. Within **Visual Studio Code**, select **Debug > Start debugging** to run the application. Navigate to the following url: <http://localhost:5000/api/notifications>. This will register a new subscription.
2. Open a browser and navigate to the **Microsoft 365 admin center** (<https://admin.microsoft.com/AdminPortal>).
3. If prompted to login, sign-in using an admin account.
  1. Select **Users > Active users**.
  2. Select an active **user** and select **Edit** for their **Contact information**.
  3. Update the **Mobile phone** value with a new number and select **Save**.

4. Wait for the notification to be received as indicated in the Visual Studio Code Debug Console:

```
powershell Received notification: 'Users/7a7fded6-0269-42c2-a0be-512d58da4463', 7a7fded6-0269-42c2-a0be-512d58da4463
```

5. The application will now initiate a delta query with the graph to get all the users and log out some of their details to the console output.

```
powershell User: 19e429d2-541a-4e0b-9873-6dff9f48fabe, Allan Deyoung User: 05501e79-f527-4913-aabf-e535646d7ffa, Christie Cline User: fecac4be-76e7-48ec-99df-df745854aa9c, Debra Berger User: 4095c5c4-b960-43b9-ba53-ef806d169f3e, Diego Siciliani User: b1246157-482f-420c-992c-fc26cbff74a5, Emily Braun User: c2b510b7-1f76-4f75-a9c1-b3176b68d7ca, Enrico Cattaneo User: 6ec9bd4b-fc6a-4653-a291-70d3809f2610, Grady Archie User: b6924afe-cb7f-45a3-a904-c9d5d56e06ea, Henrietta Mueller User: 0ee8d076-4f13-4e1a-a961-eac2b29c0ef6, Irvin Sayers User: 31f66f05-ac9b-4723-9b5d-8f381f5a6e25, Isaiah Langer User: 7ee95e20-247d-43ef-b368-d19d96550c81, Johanna Lorenz User: b2fa93ac-19a0-499b-b1b6-afa76c44a301, Joni Sherman User: 01db13c5-74fc-470a-8e45-d6d736f8a35b, Jordan Miller User: fb0b8363-4126-4c34-8185-c998ff697a60, Lee Gu User: ee75e249-a4c1-487b-a03a-5a170c2aa33f, Lidia Holloway User: 5449bd61-cc63-40b9-b0a8-e83720eeefba, Lynne Robbins User:
```

```
7ce295c3-25fa-4d79-8122-9a87d15e2438, Miriam Graham User: 737fe0a7-0b67-  
47dc-b7a6-9cf07870705, Nestor Wilke User: a1572b58-35cd-41a0-804a-  
732bd978df3e, Patti Fernandez User: 7275e1c4-5698-446c-8d1d-fa8b0503c78a,  
Pradeep Gupta User: 96ab25eb-6b69-4481-9d28-7b01cf367170, Megan Bowen User:  
846327fa-e6d6-4a82-89ad-5fd313bff0cc, Alex Wilber User: 200e4c7a-b778-436c-  
8690-7a6398e5fe6e, MOD Administrator User: 7a7fded6-0269-42c2-a0be-  
512d58da4463, Adele Vance User: 752f0102-90f2-4b8d-ae98-79dee995e35e,  
Removed?:deleted User: 4887248a-6b48-4ba5-bdd5-fed89d8ea6a0,  
Removed?:deleted User: e538b2d5-6481-4a90-a20a-21ad55ce4c1d,  
Removed?:deleted User: bc5994d9-4404-4a14-8fb0-46b8dccca0ad,  
Removed?:deleted User: d4e3a3e0-72e9-41a6-9538-c23e10a16122,  
Removed?:deleted
```

#### 6. In the Microsoft 365 Admin Portal, repeat the process of editing a user and Save again.

The application will receive another notification and will query the graph again using the last delta link it received. However, this time you will notice that only the modified user was returned in the results. powershell User: 7a7fded6-0269-42c2-a0be-512d58da4463, Adele Vance

Using this combination of notifications with delta query you can be assured you won't miss any updates to a resource. Notifications may be missed due to transient connection issues, however the next time your application gets a notification it will pick up all the changes since the last successful query.

## Review

You've now completed this exercise. In this exercise, you created a .NET Core app that used the Microsoft Graph API to receive notifications (webhooks) when a user account changes in Azure AD and perform queries using the delta query API to receive all changes to user accounts since the last query was made.

### ❖❖❖# Exercise 4: Reduce traffic with batched requests

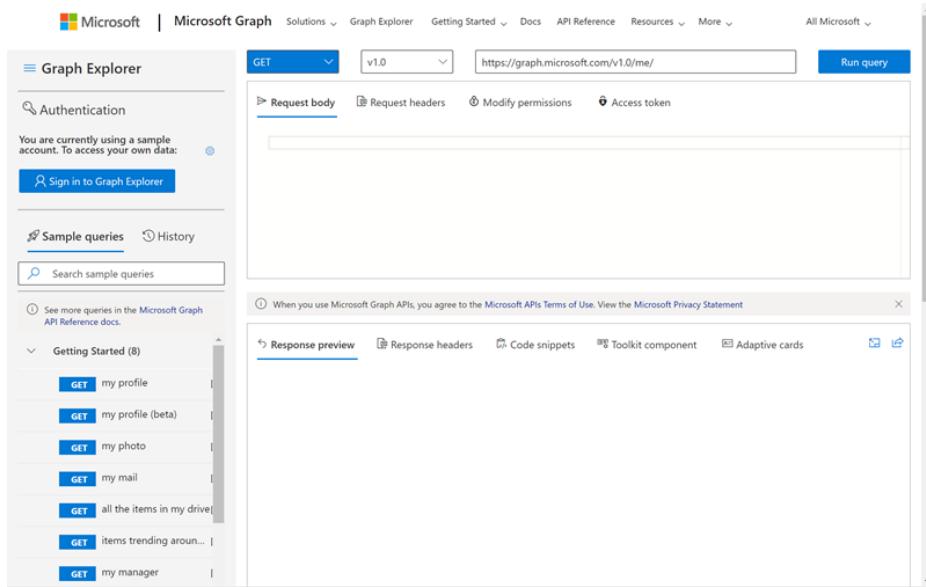
In this exercise, you'll use the Graph Explorer to create and issue a single request that contains multiple child requests. This batching of requests enables developers to submit multiple requests in a single round-trip request to Microsoft Graph, creating more optimized queries.

# Task 1: Sign in to Microsoft Graph Explorer

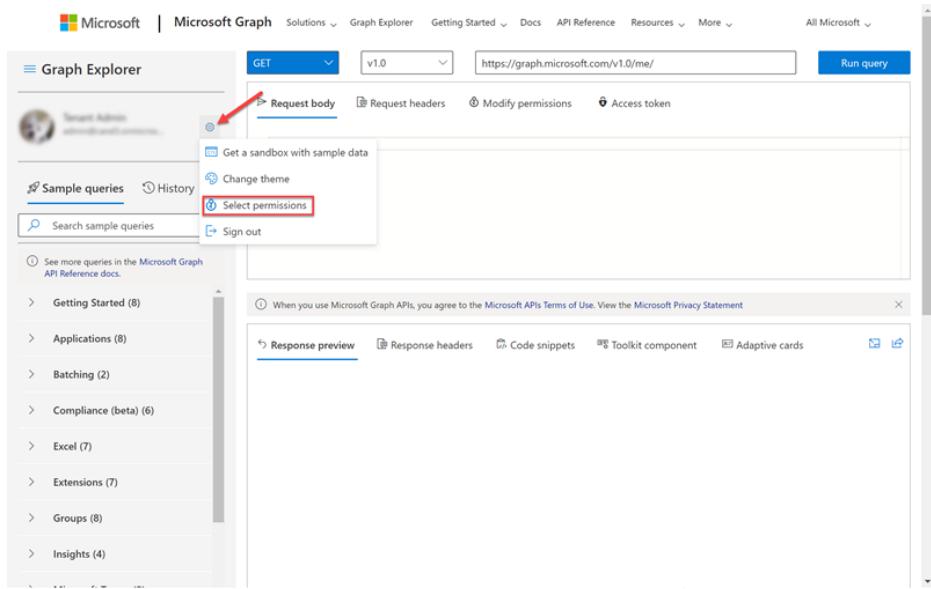
The tool Graph Explorer enables developers to create and test queries using the Microsoft Graph REST API. Previously in this module, you used the Graph Explorer as an anonymous user and executed queries using the sample data collection.

In this example, you'll sign in to Microsoft Graph with a real user.

1. Open a browser and navigate to <https://developer.microsoft.com/graph/graph-explorer>



2. Select the **Sign in to Graph Explorer** button in the leftmost panel and enter the credentials of a Work and School account.
3. After signing in, click **select permissions** and verify that the user has the permissions to submit the requests in this exercise. You must have at least these minimum permissions:
  - **Mail.Read**
  - **Calendars.Read**
  - **Files.ReadWrite**



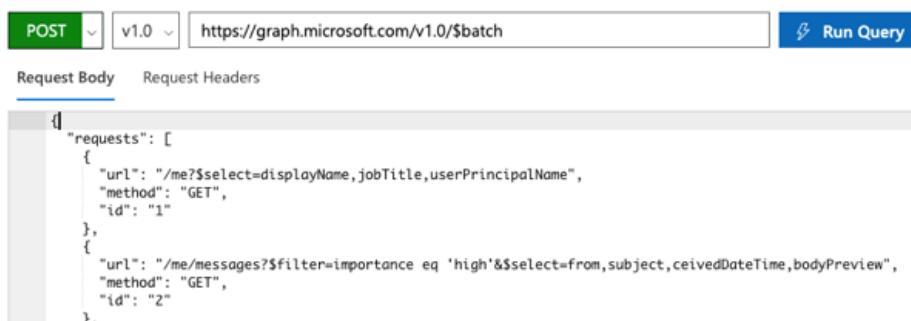
## Task 2: Submit three (3) GET requests in a single batch

All batch requests are submitted as HTTP POSTs to a specific endpoint:

[https://graph.microsoft.com/v1.0/\\$batch](https://graph.microsoft.com/v1.0/$batch). The **\$batch** query parameter is what tells Microsoft Graph to unpack the requests submitted in the body.

1. Set the request to an HTTP **POST** and the endpoint of the request to [https://graph.microsoft.com/v1.0/\\$batch](https://graph.microsoft.com/v1.0/$batch).
2. Add the following JSON code to the **Request Body** input box. This JSON code will issue three requests:
  - Request the current user's **displayName**, **jobTitle**, and **userPrincipalName** properties.
  - Request the current user's email messages that are marked with high importance.
  - Request all the current user's calendar events.

```
json { "requests": [ { "url": "/me?$select=displayName,jobTitle,userPrincipalName", "method": "GET", "id": "1" }, { "url": "/me/messages?$filter=importance eq 'high'&$select=from,subject,receivedDateTime,bodyPreview", "method": "GET", "id": "2" }, { "url": "/me/events", "method": "GET", "id": "3" } ] }
```



The screenshot shows the Microsoft Graph Explorer interface. At the top, there are dropdown menus for 'POST' and 'v1.0', and a URL input field containing 'https://graph.microsoft.com/v1.0/\$batch'. To the right is a blue 'Run Query' button. Below the URL field, there are tabs for 'Request Body' and 'Request Headers', with 'Request Body' being active. The 'Request Body' text area contains the JSON code provided in the previous step. The JSON defines three requests: one for the user's profile information, one for high-importance messages, and one for the user's calendar events.

3. Select the **Run Query** button.
4. Observe the results in the **Response Preview** box at the bottom of the page.

Success - Status Code 200, 391ms

Response Preview Response Headers

```
{ "responses": [ { }, { }, { "id": "3", "status": 200, "headers": { "Cache-Control": "private", "OData-Version": "4.0", "Content-Type": "application/json;odata.metadata=minimal;odata.streaming=true;IEEE754Compatibility=true" }, "body": { "@odata.context": "https://graph.microsoft.com/v1.0/$metadata#users('bbc8446f-f788-4527-8cff-000000000000')/events", "@odata.nextLink": "https://graph.microsoft.com/v1.0/me/events?$skip=10", "value": [ { } ] } } ] }
```

Notice the response includes three individual responses within the responses collection. Also notice for response id:3, the data that was returned, as indicated by the @odata.nextLink property, is from the **/me/events** collection. This query matches the third request in the initial request submitted.

## Task 3: Combine POST and GET requests in a single batch request

Batch requests can also include both **POST** and **GET** requests.

In this example, you'll submit a request that creates a new folder in the current user's OneDrive [for Business] and then requests the newly created folder. If the first request failed, the second request should come back empty as well.

1. Enter the following JSON code to the **Request Body** input box. This will issue three requests:

```
json { "requests": [ { "url": "/me/drive/root/children", "method": "POST", "id": "1", "body": { "name": "TestBatchingFolder", "folder": {} }, "headers": { "Content-Type": "application/json" } }, { "url": "/me/drive/root/children/TestBatchingFolder", "method": "GET", "id": "2", "DependsOn": [ "1" ] } ] }
```

2. Select the **Run Query** button.

3. Observe the results in the **Response Preview** box at the bottom of the page. Notice that this response contains two objects. The first request resulted in an HTTP 201 message that says the item, or folder, was created. The second request was also successful, and the name of the folder returned matched the folder the first request created.

Response Preview   Response Headers

```
{ "responses": [ { "id": "1", "status": 201, "headers": {}, "body": {} }, { "id": "2", "status": 200, "headers": {}, "body": { "@odata.context": "https://graph.microsoft.com/v1.0/$metadata#users('bbc8446f-f788-4527-8cff-012n5kf220plt3lt2ao2gjukgbujiokm46')/children", "createdDateTime": "2019-08-28T19:31:00Z", "eTag": "\'{B5E77A4E-40CF-4C76-9A28-C1A25105339E},1\"", "id": "012n5kf220plt3lt2ao2gjukgbujiokm46", "lastModifiedDateTime": "2019-08-28T19:31:00Z", "name": "TestBatchingFolder", "webUrl": "https://m365x460234-my.sharepoint.com/personal/meganb_m365x460234_onmicrosoft_com/TestBatchingFolder", "cTag": "\'{B5E77A4E-40CF-4C76-9A28-C1A25105339E},0\""} ] }
```

## Review

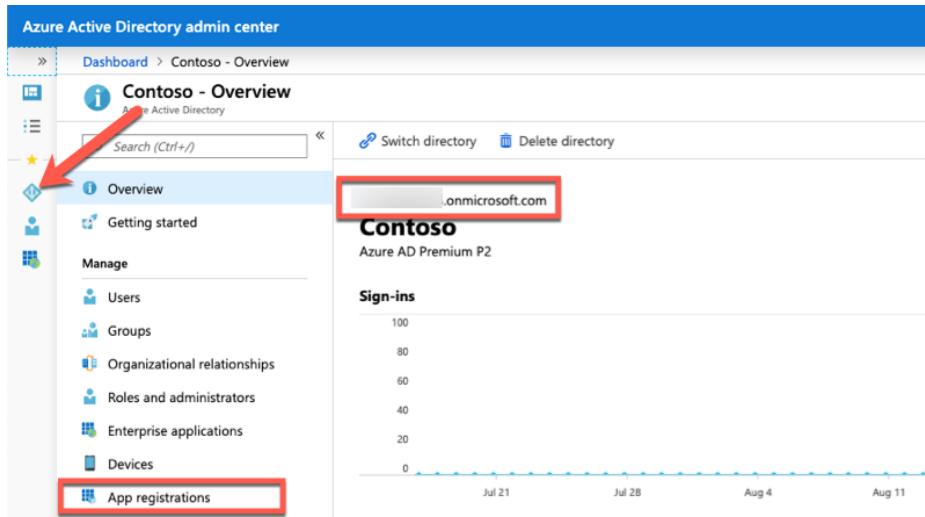
In this exercise, you used Microsoft Graph to demonstrate how you can combine multiple requests using a single request. This capability of submitting batch requests using the \$batch query parameter enables you to optimize your applications to minimize the number of requests to Microsoft Graph.

### ❖❖❖# Exercise 5: Understand throttling in Microsoft Graph

In this exercise, you will create a new Azure AD web application registration using the Azure AD admin center, a .NET Core console application, and query Microsoft Graph. You will issue many requests in parallel to trigger your requests to be throttled. This application will allow you to see the response you will receive.

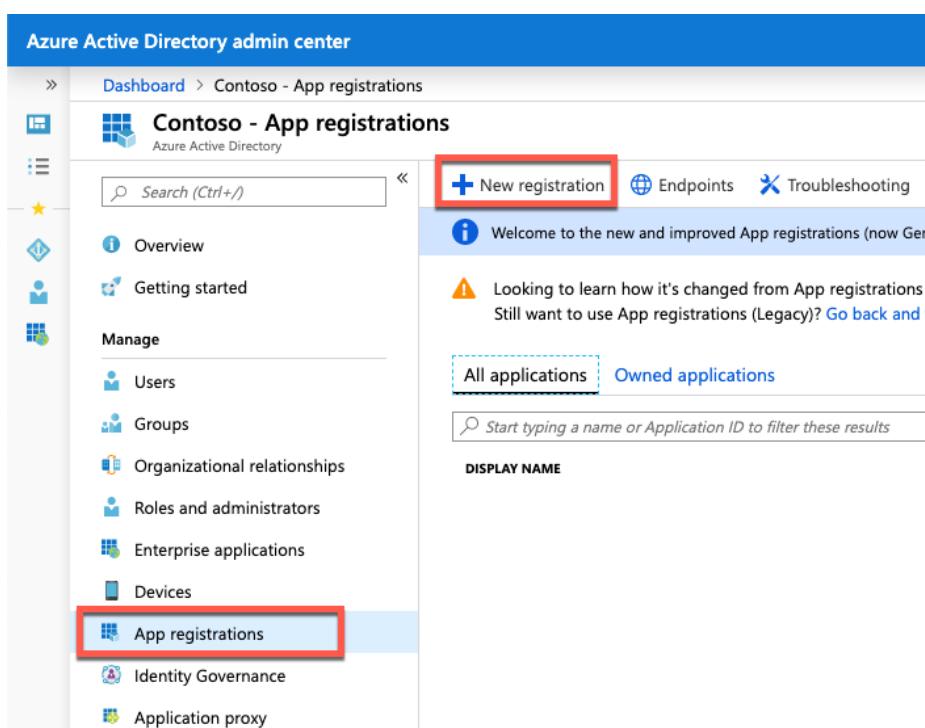
## Task 1: Create an Azure AD application

1. Open a browser and navigate to the [Azure Active Directory admin center](https://aad.portal.azure.com) (<https://aad.portal.azure.com>). Sign in using a **Work or School Account** that has global administrator rights to the tenancy.
2. Select **Azure Active Directory** in the leftmost navigation panel.



The screenshot shows the Azure Active Directory admin center interface. The left sidebar has a 'Manage' section with links for Users, Groups, Organizational relationships, Roles and administrators, Enterprise applications, and Devices. Below these, 'App registrations' is highlighted with a red box. The top navigation bar also has a 'App registrations' link highlighted with a red box. The main content area displays 'Contoso - Overview' for 'Azure AD Premium P2'. It includes a 'Sign-ins' chart showing activity from July 21 to Aug 11, with values ranging from 0 to 100.

3. Select **Manage > App registrations** in the left navigation panel.
4. On the **App registrations** page, select **New registration**.



The screenshot shows the 'Contoso - App registrations' page. The left sidebar has a 'Manage' section with links for Users, Groups, Organizational relationships, Roles and administrators, Enterprise applications, and Devices. Below these, 'App registrations' is highlighted with a red box. The top navigation bar has a 'New registration' button highlighted with a red box. The main content area includes a welcome message, a note about legacy app registrations, and tabs for 'All applications' and 'Owned applications'. There is also a search bar and a 'DISPLAY NAME' input field.

5. On the **Register an application** page, set the values as follows:
  - o **Name:** Graph Console Throttle App.

- **Supported account types:** Accounts in this organizational directory only (Contoso only - Single tenant).

\* Name

The user-facing display name for this application (this can be changed later).

Graph Console Throttle App ✓

#### Supported account types

Who can use this application or access this API?

- Accounts in this organizational directory only (Contoso only - Single tenant)
- Accounts in any organizational directory (Any Azure AD directory - Multitenant)
- Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)
- Personal Microsoft accounts only

## 6. Select Register.

7. On the **Graph Console App** page, copy the value of the **Application (client) ID** and **Directory (tenant) ID**; you will need these in the application.

Graph Console App

Search (Ctrl+ /)

Overview

Quickstart

Manage

- Branding
- Authentication
- Certificates & secrets
- API permissions
- Expose an API
- Owners
- Roles and administrators (Previous)
- Manifest

Support + Troubleshooting

- Troubleshooting
- New support request

Delete Endpoints

Welcome to the new and improved App registrations. Looking to learn how it's changed from App registrations (Legacy)? →

Display name  
Graph Console App

Application (client) ID  
9ef0bc71-6919-439e-b027-294438560213

Directory (tenant) ID  
7adf0e7f-07b3-4eb5-ba39-7ea421035371

Object ID  
52168e8f-208c-466b-a8a6-86adff64af2f

Supported account types  
My organization only

Redirect URIs  
1 web, 0 public client

Managed application in local directory  
Graph Console App

Call APIs

Documentation

Microsoft identity platform  
Authentication scenarios  
Authentication libraries  
Code samples  
Microsoft Graph  
Glossary  
Help and Support

View API Permissions

8. Select **Manage > Authentication**.

9. Under **Platform configurations**, select **Add a platform**.

10. In **Configure platforms**, select **Mobile and desktop applications**.

## Configure platforms

X

### Web applications



#### Web

Build, host, and deploy a web server application. .NET, Java, Python



#### Single-page application

Configure browser client applications and progressive web applications. Javascript.

### Mobile and desktop applications



#### iOS / macOS

Objective-C, Swift, Xamarin



#### Android

Java, Kotlin, Xamarin



#### Mobile and desktop applications

Windows, UWP, Console, IoT & Limited-entry Devices, Classic iOS + Android

11. In the section **Redirect URIs**, select the entry that begins with **msal** and enter **https://contoso** in **Custom redirect URIs** section, and then click **Configure** button.

## Configure Desktop + devices

< All platforms

Quickstart Docs

### Redirect URIs

The URIs we will accept as destinations when returning authentication responses (tokens) after successfully authenticating users. Also referred to as reply URLs. [Learn more about Redirect URIs and their restrictions](#)

- https://login.microsoftonline.com/common/oauth2/nativeclient 
- https://login.live.com/oauth20\_desktop.srf (LiveSDK) 
- msalf...://auth (MSAL only) 

### Custom redirect URIs

https://contoso



Configure

Cancel

12. Scroll down to the **Advanced settings** section and set the toggle to **Yes**.

### Advanced settings

#### Allow public client flows

Enable the following mobile and desktop flows:

Yes  No

- App collects plaintext password (Resource Owner Password Credential Flow) [Learn more](#)
- No keyboard (Device Code Flow) [Learn more](#)
- SSO for domain-joined Windows (Windows Integrated Auth Flow) [Learn more](#)

13. Select **Save** in the top menu to save your changes.

## Task 2: Grant Azure AD application permissions to Microsoft Graph

After creating the application, you need to grant it the necessary permissions to Microsoft Graph.

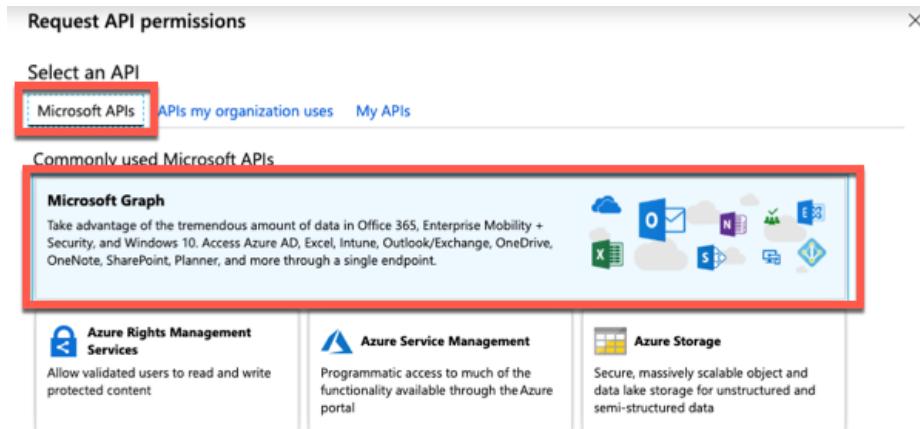
1. Select **API Permissions** in the left navigation panel.

The screenshot shows the Azure Active Directory admin center interface. In the top navigation bar, the path is "Dashboard > Contoso - App registrations > Graph Console App". On the left sidebar, under the "Graph Console App" heading, the "API permissions" option is highlighted with a red box. The main content area displays the application's details: Display name: Graph Console App, Application (client) ID: 9ef0bc7f-6919-439e-b027-294438560213, Directory (tenant) ID: 7adf0e7f-07b3-4eb5-ba39-7ea421035371, and Object ID: 52168e8f-208c-466b-a8a6-86adff64af2f. Below this, there is a section titled "Call APIs" with icons for various Microsoft services. A callout text states: "Build more powerful apps with rich user and business data from Microsoft services and your own company's data sources".

2. Select the **Add a permission** button.

The screenshot shows the "API permissions" configuration page. At the top, a note says: "Applications are authorized to call APIs when they are granted permissions by users/admins as part of the all the permissions the application needs." Below this is a table with a single row. The first column contains a blue button labeled "+ Add a permission" with a red box around it. The second column is "API / PERMISSIONS NAME" with a dropdown menu showing "Microsoft Graph (1)". The third column is "TYPE" with "Delegated" selected. The fourth column is "DESCRIPTION" with "Sign in and read user profile". A note at the bottom states: "These are the permissions that this application requests statically. You may also request user consentable permissions dynamically through code. See best practices for requesting permissions".

3. In the Request API permissions panel that appears, select **Microsoft Graph** from the **Microsoft APIs** tab.



4. When prompted for the type of permission, select **Delegated permissions**.

| PERMISSION  | ADMIN CONSENT REQUIRED |
|---|------------------------|
| <input checked="" type="checkbox"/> Mail.R  |                        |
| <b>MAIL (1)</b>   |                        |
| <input checked="" type="checkbox"/> Mail.Read<br>Read user mail                       |                        |
| <input type="checkbox"/> Mail.Read.Shared<br>Read user and shared mail                |                        |
| <input type="checkbox"/> Mail.ReadBasic<br>Read user basic mail                       |                        |
| <input type="checkbox"/> Mail.ReadWrite<br>Read and write access to user mail         |                        |
| <input type="checkbox"/> Mail.ReadWrite.Shared<br>Read and write user and shared mail |                        |

5. Enter **Mail.R** in the **Select permissions** search box and select the **Mail.Read** permission, followed by the **Add permission** button at the bottom of the panel.
6. At the bottom of the **API Permissions** panel, select the button **Grant admin consent for [tenant]**, followed by the **Yes** button, to grant all users in your organization this permission.

The option to **Grant admin consent** here in the Azure AD admin center is pre-consenting the permissions to the users in the tenant to simplify the exercise. This approach allows the console application to use the [resource owner password credential grant](#), so the user isn't prompted to grant consent to the application that simplifies the process of obtaining an OAuth access token. You could elect to implement alternative options such as the [device code flow](#) to utilize dynamic consent as another option.

## Task 3: Create .NET Core console application

1. Open your command prompt, navigate to a directory where you have rights to create your project, and run the following command to create a new .NET Core console application:  
`dotnet new console -o graphconsolethrottlepp`
2. After creating the application, run the following commands to ensure your new project runs correctly:

```
dotnetcli cd graphconsolethrottlepp dotnet add package  
Microsoft.Identity.Client dotnet add package Microsoft.Graph dotnet add  
package Microsoft.Extensions.Configuration dotnet add package  
Microsoft.Extensions.Configuration.FileExtensions dotnet add package  
Microsoft.Extensions.Configuration.Json
```

3. Open the application in Visual Studio Code using the following command: `code .`
4. If Visual Studio Code displays a dialog box asking if you want to add required assets to the project, select **Yes**.

## Task 4: Update the console app to support Azure AD authentication

1. Create a new file named **appsettings.json** in the root of the project and add the following code to it:

```
json { "tenantId": "YOUR_TENANT_ID_HERE", "applicationId":  
"YOUR_APP_ID_HERE" }
```

2. Update properties with the following values:

- **YOUR\_TENANT\_ID\_HERE**: Azure AD directory ID
- **YOUR\_APP\_ID\_HERE**: Azure AD client ID

## Task 5: Create authentication helper classes

1. Create a new folder **Helpers** in the project.
2. Create a new file **AuthHandler.cs** in the **Helpers** folder and add the following code:

```
csharp using System.Net.Http; using System.Threading; using  
System.Threading.Tasks; using Microsoft.Graph; namespace Helpers { public  
class AuthHandler : DelegatingHandler { private IAuthenticationProvider  
_authenticationProvider; public AuthHandler(IAuthenticationProvider  
authenticationProvider, HttpMessageHandler innerHandler) { InnerHandler =  
innerHandler; _authenticationProvider = authenticationProvider; } protected  
override async Task<HttpResponseMessage> SendAsync(HttpRequestMessage  
request, CancellationToken cancellationToken) { await  
_authenticationProvider.AuthenticateRequestAsync(request); return await  
base.SendAsync(request, cancellationToken); } } }
```

3. Create a new file **MsalAuthenticationProvider.cs** in the **Helpers** folder and add the following code:

```
csharp using System.Net.Http; using System.Net.Http.Headers; using  
System.Security; using System.Threading.Tasks; using  
Microsoft.Identity.Client; using Microsoft.Graph; namespace Helpers {  
public class MsalAuthenticationProvider : IAuthenticationProvider { private  
static MsalAuthenticationProvider _singleton; private  
IPublicClientApplication _clientApplication; private string[] _scopes;  
private string _username; private SecureString _password; private string  
_userId; private MsalAuthenticationProvider(IPublicClientApplication  
clientApplication, string[] scopes, string username, SecureString password)  
{ _clientApplication = clientApplication; _scopes = scopes; _username =  
username; _password = password; _userId = null; } public static  
MsalAuthenticationProvider GetInstance(IPublicClientApplication  
clientApplication, string[] scopes, string username, SecureString password)  
{ if (_singleton == null) { _singleton = new  
MsalAuthenticationProvider(clientApplication, scopes, username, password);  
} return _singleton; } public async Task  
AuthenticateRequestAsync(HttpRequestMessage request) { var accessToken =  
await GetTokenAsync(); request.Headers.Authorization = new  
AuthenticationHeaderValue("bearer", accessToken); } public async  
Task<string> GetTokenAsync() { if (!string.IsNullOrEmpty(_userId)) { try {  
var account = await _clientApplication.GetAccountAsync(_userId); if  
(account != null) { var silentResult = await  
_clientApplication.AcquireTokenSilent(_scopes, account).ExecuteAsync();  
return silentResult.AccessToken; } } catch (MsalUiRequiredException){ } }  
var result = await  
_clientApplication.AcquireTokenByUsernamePassword(_scopes, _username,  
_password).ExecuteAsync(); _userId =  
result.Account.HomeAccountId.Identifier; return result.AccessToken; } } }
```

## Task 6: Incorporate Microsoft Graph into the console app

1. Open the **Program.cs** file and add the following `using` statements to the top of the file below `using System;` line:

```
csharp using System.Collections.Generic; using System.Net; using  
System.Net.Http; using System.Net.Http.Headers; using System.Security;  
using System.Threading.Tasks; using Microsoft.Identity.Client; using  
Microsoft.Graph; using Microsoft.Extensions.Configuration; using Helpers;
```

2. Add the following method **LoadAppSettings** to the **Program** class. The method retrieves the configuration details from the **appsettings.json** file previously created:

```
csharp private static IConfigurationRoot LoadAppSettings() { try { var  
config = new ConfigurationBuilder()  
.SetBasePath(System.IO.Directory.GetCurrentDirectory())  
.AddJsonFile("appsettings.json", false, true) .Build(); if  
(string.IsNullOrEmpty(config["applicationId"])) ||  
string.IsNullOrEmpty(config["tenantId"])) { return null; } return config; }  
catch (System.IO.FileNotFoundException) { return null; } }
```

3. Add the following method **CreateAuthorizationProvider** to the **Program** class. The method will create an instance of the clients used to call Microsoft Graph.

```
csharp private static IAuthenticationProvider  
CreateAuthorizationProvider(IConfigurationRoot config, string userName,  
SecureString userPassword) { var clientId = config["applicationId"]; var  
authority = $"https://login.microsoftonline.com/{config["tenantId"]}/v2.0";  
List<string> scopes = new List<string>(); scopes.Add("User.Read");  
scopes.Add("Mail.Read"); var cca =  
PublicClientApplicationBuilder.Create(clientId) .WithAuthority(authority)  
.Build(); return MsalAuthenticationProvider.GetInstance(cca,  
scopes.ToArray(), userName, userPassword); }
```

4. Add the following method **GetAuthenticatedHttpClient** to the **Program** class. The method creates an instance of the **HttpClient** object.

```
csharp private static HttpClient  
GetAuthenticatedHttpClient(IConfigurationRoot config, string userName,  
SecureString userPassword) { var authenticationProvider =  
CreateAuthorizationProvider(config, userName, userPassword); var httpClient  
= new HttpClient(new AuthHandler(authenticationProvider, new  
HttpClientHandler())); return httpClient; }
```

5. Add the following method **ReadPassword** to the **Program** class. The method prompts the user for their password:

```
csharp private static SecureString ReadPassword() {  
Console.WriteLine("Enter your password"); SecureString password = new  
SecureString(); while (true) { Console.ReadKey(true); if  
(c.Key == ConsoleKey.Enter) { break; } password.AppendChar(c.KeyChar);  
Console.Write("*"); } Console.WriteLine(); return password; }
```

6. Add the following method **ReadUsername** to the **Program** class. The method prompts the user for their username:

```
csharp private static string ReadUsername() { string username;  
Console.WriteLine("Enter your username"); username = Console.ReadLine();  
return username; }
```

7. Locate the **Main** method in the **Program** class. Add the following code below **Console.WriteLine("Hello World!");** to load the configuration settings from the **appsettings.json** file:

```
csharp var config = LoadAppSettings(); if (config == null) {  
    Console.WriteLine("Invalid appsettings.json file."); return; }
```

8. Add the following code to the end of the **Main** method, just after the code added in the last step. This code will obtain an authenticated instance of the **HttpClient** and submit a request for the current user's email:

```
csharp var userName = ReadUsername(); var userPassword = ReadPassword();  
var client = GetAuthenticatedHTTPClient(config, userName, userPassword);
```

9. Add the following code below **var client = GetAuthenticatedHTTPClient(config, userName, userPassword);** to issue many requests to Microsoft Graph. This code will create a collection of tasks to request a specific Microsoft Graph endpoint. When a task succeeds, it will write a dot to the console, while a failed request will write an X to the console. The most recent failed request's status code and headers are saved. All tasks are then executed in parallel. At the conclusion of all requests, the results are written to the console:

```
csharp var totalRequests = 100; var successRequests = 0; var tasks = new  
List<Task>(); var failResponseCode = HttpStatusCode.OK; HttpResponseMessage  
failedHeaders = null; for (int i = 0; i < totalRequests; i++) {  
    tasks.Add(Task.Run(() => { var response =  
        client.GetAsync("https://graph.microsoft.com/v1.0/me/messages").Result;  
        Console.Write("."); if (response.StatusCode == HttpStatusCode.OK) {  
            successRequests++; } else { Console.Write('X'); failResponseCode =  
                response.StatusCode; failedHeaders = response.Headers; } })); } var allWork  
= Task.WhenAll(tasks); try { allWork.Wait(); } catch { }  
Console.WriteLine(); Console.WriteLine("{0}/{1} requests succeeded.",  
    successRequests, totalRequests); if (successRequests != totalRequests) {  
    Console.WriteLine("Failed response code: {0}",  
        failResponseCode.ToString()); Console.WriteLine("Failed response headers:  
{0}", failedHeaders); }
```

## Task 7: Build and test the application

1. Run the following command in a command prompt to compile the console application:  
dotnet build

2. Run the following command to run the console application: dotnet run

**Note:** The console app may take one or two minutes to complete the process of authenticating and obtaining an access token from Azure AD and issuing the requests to Microsoft Graph.

3. After entering the username and password of a user, you will see the results written to the console.

```
Build succeeded.
  0 Warning(s)
  0 Error(s)

Time Elapsed 00:00:01.19
Hello World!
Enter your username
MeganB@M365x460234.onmicrosoft.com
Enter your password
*****
.X.X.X.X.X.X.X.X.X...XX.X.X.X.X.X.X...XX...XXX.X.X.X.X.X.X.X.X.X.X.X.
X..XX..XX.XX..XX.X.X.X.X.....X.X.X.X.X....X.....X.....X.....X.....X.
39/100 requests succeeded.
Failed response code: TooManyRequests
Failed response headers: Cache-Control: private
Transfer-Encoding: chunked
Retry-After: 1
request-id: bbef98de-74a8-4d4d-9ac3-dde92a293035
client-request-id: bbef98de-74a8-4d4d-9ac3-dde92a293035
x-ms-ags-diagnostic: {"ServerInfo":{"DataCenter":"North Central US","Slice":"SliceC","Ring":"3","ScaleUnit":"002","RoleInstance":"AGSFE_IN_18","ADSiteName":"NCU"}}
Duration: 27.2895
Strict-Transport-Security: max-age=31536000
```

There is a mix of success and failure indicators in the console. The summary states only 39% of the requests were successful.

After the results, the console has two lines that begin with **Failed response**. Notice the code states **TooManyRequests** that is the representation of the HTTP status code 429. This status code is the indication that your requests are being throttled.

Also notice within the collection of response headers, the presence of **Retry-After**. This header is the value in seconds that Microsoft Graph tells you to wait before sending your next request to avoid being further throttled.

### Add helper class to deserialize the message object returned in a REST request

It is easier to work with strongly typed objects instead of untyped JSON responses from a **REST** request. Create a helper class to simplify working with the messages objects returned from the REST request.

1. Create a new file, **Messages.cs** in the root of the project, and add the following code to it:

```
csharp using Newtonsoft.Json; using System; namespace
graphconsolethrottlepp { public class Messages { [JsonProperty(PropertyName
```

```

= "@odata.context")] public string ODataContext { get; set; }
[JsonProperty(PropertyName = "@odata.nextLink")] public string
ODataNextLink { get; set; } [JsonProperty(PropertyName = "value")] public
Message[] Items { get; set; } } public class Message {
[JsonProperty(PropertyName = "@odata.etag")] public string ETag { get; set;
} [JsonProperty(PropertyName = "id")] public string Id { get; set; }
[JsonProperty(PropertyName = "subject")] public string Subject { get; set;
} } }

```

**Note:** This class is used by the JSON deserializer to translate a JSON response into a Messages object.

## Add method to implement delayed retry strategy when requests are throttled

The application is going to be modified to first get a list of messages in the current user's mailbox, then issue a separate request for the details of each message. In most scenarios, a separate request will trigger Microsoft Graph to throttle the requests.

To address this, your code should inspect each response for situations when the request has been throttled. In those situations, the code should check for the presence of a Retry-After header in the response that specifies the number of seconds your application should wait before issuing another request. If a Retry-After header isn't present, you should have a default value to fall back on.

1. Within the **Program.cs** file, add a new method **GetMessageDetail()** and the following code to it:

```
csharp private static Message GetMessageDetail(HttpClient client, string
messageId, int defaultDelay = 2) { Message messageDetail = null; string
endpoint = "https://graph.microsoft.com/v1.0/me/messages/" + messageId; // add
code here return messageDetail; }
```

2. Add the following code before the `// add code here` comment to create a request and wait for the response from Microsoft Graph:

```
csharp // submit request to Microsoft Graph & wait to process response var
clientResponse = client.GetAsync(endpoint).Result; var httpResponseTask =
clientResponse.Content.ReadAsStringAsync(); httpResponseTask.Wait();
```

3. In the case of a successful response, return the deserialized response back to the caller to display the messages. Add the following lines to the top of the **Program.cs** file to update the **using** statements:

```
csharp using Newtonsoft.Json;
```

4. Go back to the method **GetMessageDetail()** and the following code before the `// add code here` comment:

```
csharp Console.WriteLine("...Response status code: {0} ", clientResponse.StatusCode); // IF request successful (not throttled), set message to retrieved message if (clientResponse.StatusCode == HttpStatusCode.OK) { messageDetail = JsonConvert.DeserializeObject<Message>(httpResponseTask.Result); }
```

5. In the case of a throttled response, add the following **else** statement to the if statement you just added:

```
csharp // ELSE IF request was throttled (429, aka: TooManyRequests)... else
if (clientResponse.StatusCode == HttpStatusCode.TooManyRequests) { // get
```

This code will do the following:

- Set a default delay of two seconds before the next request is made.
  - If the Retry-After header value is present and greater than zero seconds, use that value to overwrite the default delay.
  - Set the thread to sleep for the specified, or default, number of seconds.
  - Recursively call the same method to retry the request.

**Tip:** In cases where the response does not include a **Retry-After** header, it is recommended to consider implementing an exponential back-off default delay. In this code, the application will initially pause for two seconds before retrying the request. Future requests will double the delay if Microsoft Graph continues to throttle the request.

Real-world applications should have an upper limit on how long they will delay so to avoid an unreasonable delay so users are not left with an unresponsive experience.

The resulting method should look like the following:

```
csharp private static Message GetMessageDetail(HttpClient client, string  
messageId, int defaultDelay = 2) { Message messageDetail = null; string  
endpoint = "https://graph.microsoft.com/v1.0/me/messages/" + messageId; //  
submit request to Microsoft Graph & wait to process response var clientResponse  
= client.GetAsync(endpoint).Result; var httpResponseTask =  
clientResponse.Content.ReadAsStringAsync(); httpResponseTask.Wait();  
Console.WriteLine("...Response status code: {0} ", clientResponse.StatusCode);  
// IF request successful (not throttled), set message to retrieved message if  
(clientResponse.StatusCode == HttpStatusCode.OK) { messageDetail =  
JsonConvert.DeserializeObject<Message>(httpResponseTask.Result); } // ELSE IF  
request was throttled (429, aka: TooManyRequests)... else if  
(clientResponse.StatusCode == HttpStatusCode.TooManyRequests) { // get retry-  
after if provided; if not provided default to 2s int retryAfterDelay =  
defaultDelay; if (clientResponse.Headers.RetryAfter.Delta.HasValue &&  
(clientResponse.Headers.RetryAfter.Delta.Value.Seconds > 0)) { retryAfterDelay  
= clientResponse.Headers.RetryAfter.Delta.Value.Seconds; } // wait for  
specified time as instructed by Microsoft Graph's Retry-After header, // or  
fall back to default Console.WriteLine(">>>>>>>>>> sleeping for {0}  
seconds...", retryAfterDelay); System.Threading.Thread.Sleep(retryAfterDelay *  
1000); // call method again after waiting messageDetail =  
GetMessageDetail(client, messageId); } // add code here return messageDetail; }
```

### **Update application to use retry strategy**

The next step is to update the Main method to use the new method so the application will use an intelligent throttling strategy.

1. Locate the following line that obtains an instance of an authenticated **HttpClient** object in the **Main** method. Delete all code in the **Main** method after this line:

```
csharp var client = GetAuthenticatedHTTPClient(config, userName, userPassword);
```

2. Add the following code after obtaining the HttpClient object. This code will request the top 100 messages from the current user's mailbox and deserialize the response into a typed object you previously created:

```
csharp var stopwatch = new System.Diagnostics.Stopwatch();
stopwatch.Start(); var clientResponse =
client.GetAsync("https://graph.microsoft.com/v1.0/me/messages?
$select=id&$top=100").Result; // enumerate through the list of messages var
httpResponseTask = clientResponse.Content.ReadAsStringAsync();
httpResponseTask.Wait(); var graphMessages =
JsonConvert.DeserializeObject<Messages>(httpResponseTask.Result);
```

3. Add the following code to create individual requests for each message. These tasks are created as asynchronous tasks that will be executed in parallel:

```
csharp var tasks = new List<Task>(); foreach (var graphMessage in
graphMessages.Items) { tasks.Add(Task.Run(() => {
Console.WriteLine("...retrieving message: {0}", graphMessage.Id); var
messageDetail = GetMessageDetail(client, graphMessage.Id);
Console.WriteLine("SUBJECT: {0}", messageDetail.Subject); })); }
```

4. Next, add the following code to execute all tasks in parallel and wait for them to complete:

```
csharp // do all work in parallel & wait for it to complete var allWork =
Task.WhenAll(tasks); try { allWork.Wait(); } catch { }
```

5. With all work complete, write the results to the console:

```
csharp stopwatch.Stop(); Console.WriteLine(); Console.WriteLine("Elapsed
time: {0} seconds", stopwatch.Elapsed.Seconds);
```

## Build and test the updated application

1. Run the following command in a command prompt to compile the console application:  
dotnet build

2. Run the following command to run the console application: dotnet run

After entering the username and password for the current user, the application will write multiple log entries to the console, as in the following figure.



Within one or two minutes, the application will display the results of the application. Depending on the speed of your workstation and internet connection, your requests may or may not have triggered Microsoft Graph to throttle you. If not, try running the application a few more times.

If your application ran fast enough, you should see some instances where Microsoft Graph returned the HTTP status code 429, indicated by the **TooManyRequests** entries.

```
SUBJECT: Your Azure AD Identity Protection Weekly Digest
...Response status code: OK
SUBJECT: Your Azure AD Identity Protection Weekly Digest
...Response status code: TooManyRequests
...Response status code: OK
...Response status code: TooManyRequests
...Response status code: OK
SUBJECT: Questions around our marketing strategy
...Response status code: OK
...Response status code: OK
...Response status code: OK
SUBJECT: Let's reschedule 1:1
...Response status code: OK
SUBJECT: Mark 8 design
SUBJECT: Northwind Proposal
SUBJECT: You have tasks due today!
...Response status code: OK
SUBJECT: Your Azure AD Identity Protection Weekly Digest
...Response status code: TooManyRequests
SUBJECT: Your Azure AD Identity Protection Weekly Digest
>>>>>>>> sleeping for 1 seconds...
>>>>>>>> sleeping for 1 seconds...
>>>>>>>> sleeping for 1 seconds...
...Response status code: OK
...Response status code: OK
SUBJECT: View your Microsoft Workplace Analytics billing statement
...Response status code: OK
```

In this case, the **messages** endpoint returned a **Retry-After** value of one (1) because the application displays messages on the console that it slept for one second.

The important point is that the application completed successfully, retrieving all 100 messages, even when some requests were rejected due to being throttled by Microsoft Graph.

## Task 8: Implement Microsoft Graph SDK for throttling retry strategy

In the last section exercise, you modified the application to implement a strategy to determine if a request is throttled. In the case where the request was throttled, as indicated by the response to the REST endpoint request, you implemented a retry strategy using the `HttpClient`.

Let's change the application to use the Microsoft Graph SDK client, which has all the logic built in for implementing the retry strategy when a request is throttled.

### Update the `GetAuthenticatedHttpClient` method

The application will use the Microsoft Graph SDK to submit requests, not the `HttpClient`, so you need to update it.

1. Locate the method `GetAuthenticatedHttpClient` and make the following changes to it:

1. Set the **return** type from `HttpClient` to `GraphServiceClient`.
2. Rename the **method** from `GetAuthenticatedHttpClient` to `GetAuthenticatedGraphClient`.
3. Replace the last two lines in the method with the following lines to obtain and return an instance of the `GraphServiceClient`:

```
csharp var graphClient = new  
GraphServiceClient(authenticationProvider); return graphClient;
```

2. Your updated method `GetAuthenticatedGraphClient` should look similar to this:

```
csharp private static GraphServiceClient  
GetAuthenticatedGraphClient(IConfigurationRoot config, string userName,  
SecureString userPassword) { var authenticationProvider =  
CreateAuthorizationProvider(config, userName, userPassword); var  
graphClient = new GraphServiceClient(authenticationProvider); return  
graphClient; }
```

### Update the application to use the `GraphServiceClient`.

1. The next step is to update the application to use the Graph SDK that includes an intelligent throttling strategy. Locate the `Messages.cs` file in the project. Delete this file or comment all code within the file out. Otherwise, the application will get the `Message` object this file contains confused with the `Message` object in the Microsoft Graph SDK.

2. Next, within the `Main` method, locate the following line:

```
csharp var client = GetAuthenticatedHTTPClient(config, userName,  
userPassword);
```

3. Update the method called in that line to use the method you updated, `GetAuthenticatedGraphClient`:

```
csharp var client = GetAuthenticatedGraphClient(config, userName,  
userPassword);
```

4. The next few lines used the **HttpClient** to call the Microsoft Graph REST endpoint to get a list of all messages. Find these lines, as shown, and remove them:

```
csharp var clientResponse =  
client.GetAsync("https://graph.microsoft.com/v1.0/me/messages?  
$select=id&$top=100").Result; // enumerate through the list of messages var  
httpResponseTask = clientResponse.Content.ReadAsStringAsync();  
httpResponseTask.Wait(); var graphMessages =  
JsonConvert.DeserializeObject<Messages>(httpResponseTask.Result);
```

5. Replace those lines with the following code to request the same information using the Microsoft Graph SDK. The collection returned by the SDK is in a different format than what the REST API returned:

```
csharp var clientResponse = client.Me.Messages .Request() .Select(m => new  
{ m.Id }) .Top(100) .GetAsync() .Result;
```

6. Locate the **foreach** loop that enumerates through all returned messages to request each message's details. Change the collection to the following code:

```
csharp foreach (var graphMessage in clientResponse.CurrentPage)
```

## Update the **GetMessageDetail** method to return

The last step is to modify the **GetMessageDetail** method that retrieved the message details for each message. Recall from the previous section in this unit that you had to write the code to detect when requests were throttled. In the where case they were throttled, you added code to retry the request after a specified delay. Fortunately, the Microsoft Graph SDK has this logic included in it.

1. Locate the **GetMessageDetail()** method.
2. Update the signature of the method so the first parameter expects an instance of the **GraphServiceClient**, not the **HttpClient**, and remove the last parameter of a default delay. The method signature should now look like the following:

```
csharp private static Microsoft.Graph.Message  
GetMessageDetail(GraphServiceClient client, string messageId)
```

3. Next, remove all code within this method and replace it with this single line:

```
csharp // submit request to Microsoft Graph & wait to process response  
return client.Me.Messages[messageId].Request().GetAsync().Result;
```

## Task 9: Build and test the updated application

1. Run the following command in a command prompt to compile the console application:  
`dotnet build`
2. Run the following command to run the console application: `dotnet run`
3. After entering the username and password for the current user, the application will write multiple log entries to the console, as in the following image.

```
SUBJECT: Let's reschedule it!
SUBJECT: Northwind Budget
SUBJECT: EMEA Training Programs
SUBJECT: Your Office 365 E5 Demo Trial is about to expire - buy today!
SUBJECT: You have upcoming tasks due.
SUBJECT: Liquidation sale on cool cups
SUBJECT: Renew your Microsoft subscriptions
SUBJECT: Renew your Microsoft subscriptions
SUBJECT: Your Azure AD Identity Protection Weekly Digest
SUBJECT: Starts today! Seattle Restaurant Week
SUBJECT: Your Azure AD Identity Protection Weekly Digest
SUBJECT: Your Azure AD Identity Protection Weekly Digest
SUBJECT: Get started with Microsoft Workplace Analytics subscription
SUBJECT: Accepted: Tailspin Toys Proposal Review + Lunch
SUBJECT: Your Azure AD Identity Protection Weekly Digest
SUBJECT: Action Needed - Renew your Microsoft subscriptions
SUBJECT: Renew your Microsoft subscriptions
SUBJECT: Northwind Proposal
SUBJECT: Audio Conferencing for Microsoft Teams or Skype for Business Online has been turned off
SUBJECT: T-minus 50 points until lift off
SUBJECT: Attention: Your Office 365 E5 Demo Trial has expired
SUBJECT: Alex Wilber is now following you on Yammer
SUBJECT: Carve out fall travel time from $59 one way.
SUBJECT: Mark B design
SUBJECT: Your organization is out of SharePoint Online storage space
SUBJECT: View your Microsoft Workplace Analytics billing statement
SUBJECT: Your Azure AD Identity Protection Weekly Digest
SUBJECT: Potluck Party Berlin
```

The application will do the same thing as the **HttpClient** version of the application. However, one difference is that the application will not display the status code returned in the response to the requests or any of the *sleeping* log messages, because the Microsoft Graph SDK handles all the retry logic internally.

## Review

In this exercise, you used the Azure AD application and .NET console application you previously created and modified them to demonstrate two strategies to account for throttling in your application. One strategy used the **HttpClient** object but required you to implement the detect, delay, and retry logic yourself when requests were throttled. The other strategy used the Microsoft Graph SDK's included support for handling this same scenario.

### Exercise 6: Querying user data from Microsoft Graph

This exercise leads the user through a series of tasks utilizing Microsoft Graph Explorer. For this exercise you will use the default sample account and will not sign in.

**Note:** The trainer should confirm users are not signed on in a browser with a Microsoft 365 account. You may also direct them to open an InPrivate window session.

By the end of this exercise you will be able to:

- Get the signed-in user's profile.
- Get a list of users in the organization.
- Get the user's profile photo.
- Get a user object based on the user's unique identifier.
- Get the user's manager profile.

## Task 1: Go to the Graph Explorer

1. Open the Microsoft Edge browser.
2. Go to this link: <https://developer.microsoft.com/en-us/graph/graph-explorer>.

This page allows users to interact with the Microsoft Graph without needing to write any code. The Microsoft Graph Explorer provides sample data to use for read operations.

**Note:** Some organizations may not allow users to sign in or consent to specific scopes required for some operations.

## Task 2: Get the signed in user's profile

1. The page loads with a request of `/v1.0/me` entered in the **request URL** box.
2. Select **Run Query**.

The data for the current users~~?~~~~?~~~~?~~ profile is shown in the **Response Preview** pane at the bottom of the page.

## **Task 3: Get a list of users in the organization**

1. In the request URL box amend the request to be: **/v1.0/users**
2. Select **Run Query**.

This shows a list of users in the organization in the **Response Preview** pane at the bottom of the page.

## Task 4: Get the user object based on the user's unique identifier

1. In the second item shown in the results, select and copy the **ID** property. This should be the ID for the Adele Vance user.
2. In the **request URL** box change the URL to request the profile for a specific user by appending the copied ID to the request URL in the form `/users/{id}`; for example:  
`/users/87d349ed-44d7-43e1-9a83-5f2406dee5bd`
3. Select **Run Query**.

The profile information for that user is shown in the **Response Preview** pane at the bottom of the page.

## Task 5: Get the user's profile photo

1. In the request URL box append **/photo**
2. The URL should now be something like:  
**https://graph.microsoft.com/v1.0/users/87d349ed-44d7-43e1-9a83-5f2406dee5bd/photo**
3. Select **Run Query**. In the **Response Preview**, metadata about the profile picture for the user is shown.
4. In the **request URL** box, append **/\$value**
5. The URL should now be something like:  
**https://graph.microsoft.com/v1.0/users/87d349ed-44d7-43e1-9a83-5f2406dee5bd/photo/\$value**
6. Select **Run Query**. The image for the default profile picture is shown in the **Response Preview** pane at the bottom of the page.

## Task 6: Get the user's manager profile

1. In the **request URL** box, replace `/photo/$value` with `/manager`
2. The URL should now be something like:  
**`https://graph.microsoft.com/v1.0/users/87d349ed-44d7-43e1-9a83-5f2406dee5bd/manager`**
3. Select **Run Query**. The profile information for the manager of the specified user is shown in the **Response Preview** pane.
4. The URL should now be something like:  
**`https://graph.microsoft.com/v1.0/users/87d349ed-44d7-43e1-9a83-5f2406dee5bd/photo`**
5. Select **Run Query**.

## Review

In this exercise, you learned how to:

- Get the signed-in user’s profile.
- Get a list of users in the organization.
- Get the user’s profile photo.
- Get a user object based on the user’s unique identifier.
- Get the user’s manager profile.

# Student lab manual

## Lab scenario

As a developer at a large enterprise that needs a governance plan and approval process around the creation of Office 365 Teams and Groups, you will support the plan to meet requirements in the most efficient way. As you set out to meet the objectives, you need to collect data from many separate Microsoft 365 services. Focusing on simplifying programming and authentication, Microsoft Graph provides a single point of access to a wide range of Microsoft resources, supporting you with the range of data needed to complete requirements for the plan.

## **Objectives**

After you complete this lab, you will be able to:

- Optimize data usage with query parameters on Microsoft Graph.
- Optimize network traffic using Microsoft Graph.
- Access user data with Microsoft Graph.
- Access files with Microsoft Graph.
- Manage a group lifecycle on Microsoft Graph.

## **Lab setup**

Estimated duration: 240 minutes

## Instructions

Observe the taskbar located at the bottom of your Windows 10 desktop. The taskbar contains the icons for the applications you will use in this lab, including:

- Microsoft Edge
- File Explorer
- Visual Studio Code
- Microsoft Visual Studio 2019
- PowerShell

Also, ensure that the following necessary utilities are installed:

- [.NET Core 3.1 SDK](#)
- ngrok

◆◆◆# Exercise 1: Using query parameters when querying Microsoft Graph via HTTP

**NOTE:** Users may sign in with their Microsoft account or use the default sample account to complete the following tasks.

By the end of this exercise you will be able to:

- Use `$filter` query parameter.
- Use `$select` query parameter.
- Order results using `$orderby` query parameter.
- Set page size of results using `$skip` and `$top` query parameters.
- Expand and retrieve resources using `$expand` query parameter.
- Retrieve the total count of matching resources using `$count` query parameter.
- Search for resources using `$search` query parameter.

## Task 1: Go to the Graph Explorer

1. Open the Microsoft Edge browser.
2. Go to this link: <https://developer.microsoft.com/en-us/graph/graph-explorer>.

This page allows users to interact with Microsoft Graph without writing any code. Microsoft Graph Explorer provides sample data to use for read operations.

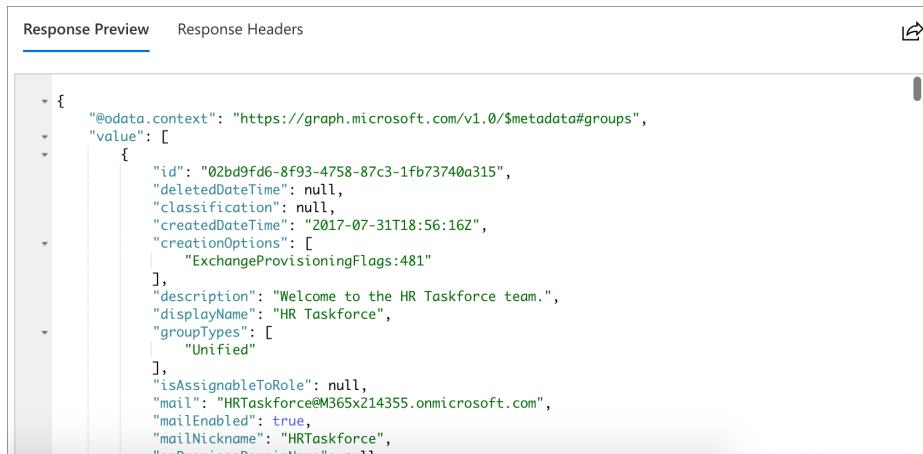
**NOTE:** Some organizations may not allow users to sign in or consent to specific scopes required for some operations.

## Task 2: Use `\$select` to retrieve only some object properties

1. In the **request URL** box amend the request to be:  
`https://graph.microsoft.com/v1.0/groups`

### 2. Select Run Query.

This lists all the groups in the tenant that the current user can see in the **Response Preview** pane at the bottom of the page. Note there are many properties returned per record.



The screenshot shows the Microsoft Graph Explorer interface with the 'Response Preview' tab selected. It displays a JSON array of group objects. Each group has properties like id, deletedDateTime, classification, createdDateTime, creationOptions, description, displayName, groupTypes, isAssignableToRole, mail, mailEnabled, and mailNickname. The JSON structure is collapsed, showing only the top-level array and some inner object details.

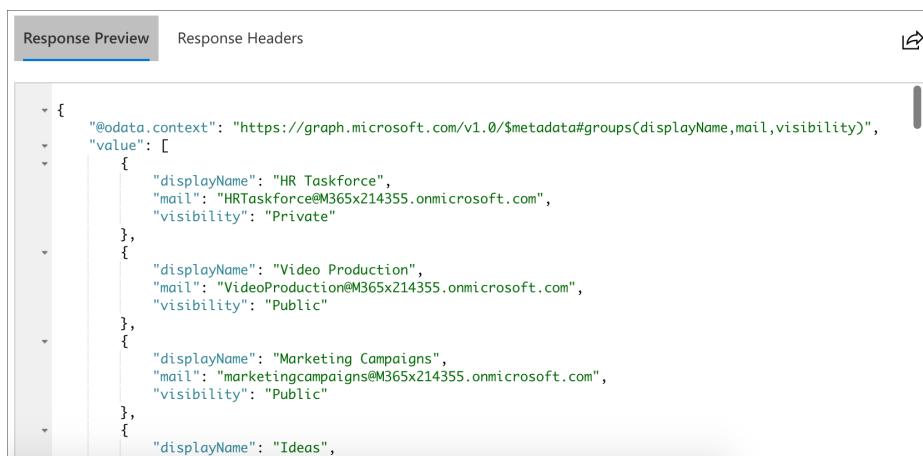
```
[{"id": "02bd9fd6-8f93-4758-87c3-1fb73740a315", "deletedDateTime": null, "classification": null, "createdDateTime": "2017-07-31T18:56:16Z", "creationOptions": [{"ExchangeProvisioningFlags:481"}], "description": "Welcome to the HR Taskforce team.", "displayName": "HR Taskforce", "groupTypes": [{"Unified"}], "isAssignableToRole": null, "mail": "HRTaskforce@M365x214355.onmicrosoft.com", "mailEnabled": true, "mailNickname": "HRTaskforce", "visibility": "Private"}, {"id": "12345678-9abc-1234-5678-987654321098", "deletedDateTime": null, "classification": null, "createdDateTime": "2018-01-01T14:45:00Z", "creationOptions": [{"ExchangeProvisioningFlags:481"}], "description": "Video Production Team", "displayName": "Video Production", "groupTypes": [{"Unified"}], "isAssignableToRole": null, "mail": "VideoProduction@M365x214355.onmicrosoft.com", "mailEnabled": true, "mailNickname": "VideoProduction", "visibility": "Public"}, {"id": "98765432-1234-5678-9abc-123456789876", "deletedDateTime": null, "classification": null, "createdDateTime": "2018-01-01T14:45:00Z", "creationOptions": [{"ExchangeProvisioningFlags:481"}], "description": "Marketing Campaigns", "displayName": "Marketing Campaigns", "groupTypes": [{"Unified"}], "isAssignableToRole": null, "mail": "marketingcampaigns@M365x214355.onmicrosoft.com", "mailEnabled": true, "mailNickname": "marketingcampaigns", "visibility": "Public"}, {"id": "87654321-1234-5678-9abc-123456789876", "deletedDateTime": null, "classification": null, "createdDateTime": "2018-01-01T14:45:00Z", "creationOptions": [{"ExchangeProvisioningFlags:481"}], "description": "Idea Generation", "displayName": "Ideas", "groupTypes": [{"Unified"}], "isAssignableToRole": null, "mail": "ideas@M365x214355.onmicrosoft.com", "mailEnabled": true, "mailNickname": "ideas", "visibility": "Public"}]
```

### 3. Select the **Response Headers** tab.

Note the value of the **content-length** header. This shows how much data was returned in the response for this request.

4. To reduce the response to only include the necessary properties, you can make use of the `$select` parameter. Update the **request URL** box to the following:

`https://graph.microsoft.com/v1.0/groups?$select=displayName,mail,visibility`



The screenshot shows the Microsoft Graph Explorer interface with the 'Response Preview' tab selected. It displays a JSON array of group objects, but only three properties are shown for each group: displayName, mail, and visibility. The JSON structure is collapsed, showing only the top-level array and some inner object details.

```
[{"displayName": "HR Taskforce", "mail": "HRTaskforce@M365x214355.onmicrosoft.com", "visibility": "Private"}, {"displayName": "Video Production", "mail": "VideoProduction@M365x214355.onmicrosoft.com", "visibility": "Public"}, {"displayName": "Marketing Campaigns", "mail": "marketingcampaigns@M365x214355.onmicrosoft.com", "visibility": "Public"}, {"displayName": "Ideas", "mail": "ideas@M365x214355.onmicrosoft.com", "visibility": "Public"}]
```

### 5. Select Run Query.

In the **Response Headers** tab note the new value for the **content-length** header. There is a much smaller amount of data returned now.

6. Select the **Response Preview** tab.

Note only the selected properties are returned. To allow an application to interact directly with an object in Microsoft Graph, it is best to have the unique identifier for that object.

7. With this in mind, amend your query to include the **id** property:

```
https://graph.microsoft.com/v1.0/groups?  
$select=displayName,mail,visibility,id
```

8. Select **Run Query**.

The screenshot shows the Microsoft Graph API Response Preview tool. It has two tabs at the top: "Response Preview" (which is selected) and "Response Headers". The main area displays a JSON response with three groups. A red arrow points to the "id" field of the first group, which is "02bd9fd6-8f93-4758-87c3-1fb73740a315".

```
{  
  "@odata.context": "https://graph.microsoft.com/v1.0/$metadata#groups(displayName,mail,visibility,id)",  
  "value": [  
    {  
      "displayName": "HR Taskforce",  
      "mail": "HRTaskforce@M365x214355.onmicrosoft.com",  
      "visibility": "Private",  
      "id": "02bd9fd6-8f93-4758-87c3-1fb73740a315"  
    },  
    {  
      "displayName": "Video Production",  
      "mail": "VideoProduction@M365x214355.onmicrosoft.com",  
      "visibility": "Public",  
      "id": "06f62f70-9827-4e6e-93ef-8e0f2d9b7b23"  
    },  
    {  
      "displayName": "Marketing Campaigns",  
      "mail": "marketingcampaigns@M365x214355.onmicrosoft.com",  
      "visibility": "Public",  
      "id": "0a53828f-36c9-44c3-be3d-99a7fce977ac"  
    }  
  ]  
}
```

## Task 3: Use `$orderby` to sort results

When presenting data to end users it's often necessary to use a sort order other than the default provided by Microsoft Graph. This should be done using the `$orderby` parameter.

Continuing from the previous task, sort the groups by the `displayName` property.

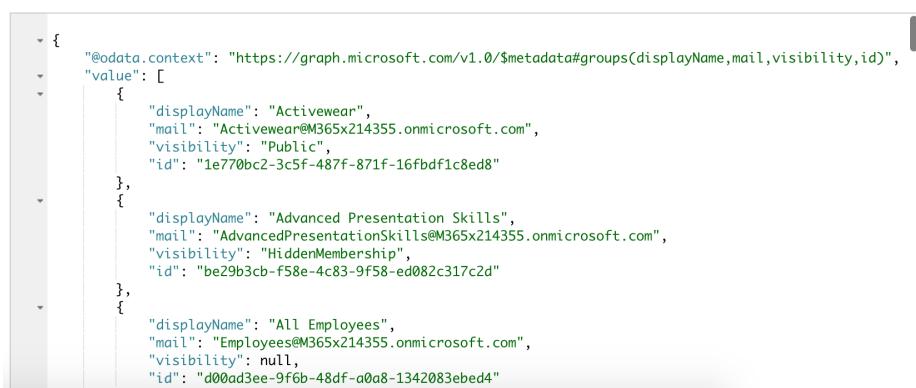
1. Change the request URL box to:

```
https://graph.microsoft.com/v1.0/groups?  
$select=displayName,mail,visibility,id&$orderBy=displayName
```

2. Select **Run Query**.

The sorted set of groups is shown in the **Response Preview** pane.

Response Preview   Response Headers 



```
{  
  "@odata.context": "https://graph.microsoft.com/v1.0/$metadata#groups(displayName,mail,visibility,id)",  
  "value": [  
    {  
      "displayName": "Activewear",  
      "mail": "Activewear@M365x214355.onmicrosoft.com",  
      "visibility": "Public",  
      "id": "1e770bc2-3c5f-487f-871f-16fbdf1c8ed8"  
    },  
    {  
      "displayName": "Advanced Presentation Skills",  
      "mail": "AdvancedPresentationSkills@M365x214355.onmicrosoft.com",  
      "visibility": "HiddenMembership",  
      "id": "be29b3cb-f58e-4c83-9f58-ed082c317c2d"  
    },  
    {  
      "displayName": "All Employees",  
      "mail": "Employees@M365x214355.onmicrosoft.com",  
      "visibility": null,  
      "id": "d00ad3ee-9f6b-48df-a0a8-1342083ebcd4"  
    }  
  ]  
}
```

**NOTE:** Many properties cannot be used for sorting. For example, if you were to use `mail` in the `$orderby` you would receive an HTTP 400 response like this:

```
http { "error": { "code": "Request_UnsupportedQuery", "message":  
  "Unsupported sort property 'mail' for 'Group'.", "innerError": { "request-  
  id": "582643b8-7ac2-415a-b58b-59009ec63ec1", "date": "2019-10-24T19:54:55"  
  } } }
```

3. Set the sort direction using either **asc** or **desc** like this:

```
https://graph.microsoft.com/v1.0/groups?  
$select=displayName,mail,visibility,id&$orderBy=displayName desc
```

## Task 4: Use \$filter to retrieve a subset of data available

1. In the **request URL** box amend the request to:

```
https://graph.microsoft.com/v1.0/me/messages
```

2. Select **Run Query**.

This shows a list of messages for the current **Response Preview** pane at the bottom of the page.

Note the response also includes an `@odata.nextLink` property. This allows developers to fetch additional pages of results and, by its presence, indicates there may be a large possible number of results. To find just the email messages that have attachments, add a filter.

The screenshot shows the Microsoft Graph Explorer interface with the 'Response Preview' tab selected. It displays a JSON response structure. A red arrow points to the '@odata.context' object, specifically to the '@odata.nextLink' property, which contains the URL `https://graph.microsoft.com/v1.0/me/messages?$skip=15`. The main body of the response shows a list of message objects, each with properties like id, subject, bodyPreview, and attachments.

```
{
  "@odata.context": "https://graph.microsoft.com/v1.0/$metadata#users('48d31887-5fad-4d73-a9f5-3c356e68a038')/messages",
  "@odata.nextLink": "https://graph.microsoft.com/v1.0/me/messages?$skip=15",
  "value": [
    {
      "id": "AAMkAGVnMDEzMTM4LTZmYWUtNDDkNC1hMDZiLTU1OGY50TZhYmY40AAUAAAAAAiQ8W967B7TKBjgx9rVEURBwAiI",
      "subject": "MyAnalytics | Focus Edition",
      "bodyPreview": "MyAnalytics\nDiscover your habits. Work smarter.\nFor your eyes only",
      "importance": "normal",
      "categories": [],
      "receivedDateTime": "2019-11-04T16:12:47Z",
      "sentDateTime": "2019-11-04T16:12:47Z",
      "hasAttachments": false,
      "internetMessageId": "<BYAPR15MB24233E7FD916D83C860BA3B0CD7F0@BYAPR15MB2423.namprd15.prod.outlook.com>",
      "parentFolderId": "AAMkAGVnMDEzMTM4LTZmYWUtNDDkNC1hMDZiLTU1OGY50TZhYmY40AAUAAAAAAiQ8W967B7TKBjgx9rVEURBwAiI",
      "conversationId": "AAQkAGVnMDEzM4LTZmYWUtNDDkNC1hMDZiLTU1OGY50TZhYmY40AAQAFRLF1C8ZxIlmEwGEI8Pc"
    }
  ]
}
```

3. Add a filter so the **request URL** box contains the following:

```
https://graph.microsoft.com/v1.0/me/messages?$filter=hasAttachments eq true
```

4. Select **Run Query**.

The results are shown in the **Response Preview** pane. Note there is no `@odata.nextLink` this time. This indicates all the matching records have been returned in the request. If there is an `@odata.nextLink` it should be used to fetch additional pages of results. It's possible to compose queries that traverse object properties as well. For instance, to find mail sent from a specific email address.

5. Change the **request URL** box to:

```
https://graph.microsoft.com/v1.0/me/messages?
$filter=from/emailAddress/address eq 'no-reply@microsoft.com'
```

6. Select **Run Query**.

The results are shown in the **Response Preview** pane. Note there are multiple pages of results as indicated by the presence of `@odata.nextLink`. Some properties support the use of a start with operator.

7. Enter the following into the **request URL** box:

```
https://graph.microsoft.com/v1.0/me/messages?  
$filter=startswith(subject,'my')
```

8. Select **Run Query**.

The results are shown in the **Response Preview** pane.

9. Some collection properties can be used in **\$filter** using a Lambda expression.

For example, to find all groups that have the Unified type, enter the following query into the request URL box:

```
https://graph.microsoft.com/v1.0/groups?$filter=groupTypes/any(c: c eq  
'Unified')
```

10. Select **Run Query**.

The list of groups that have the Unified type is shown in the **Response Preview** pane.

## Task 5: Use `\$skip` and `\$top` for explicit pagination of results

1. In the request URL box change the request to:

```
https://graph.microsoft.com/v1.0/me/events
```

2. Select **Run Query**.

The results are shown in the **Response Preview** pane.

Note the `@odata.nextLink` property finishes in `$skip 10`. This is the default first page size for the `/events` resource. Different resources have different default first page sizes. The size of the first page and subsequent pages may not be the same. Edit the query to fetch an explicit number of results. `$top` is used to set maximum the number of results to be returned.

3. In the request URL box amend the request to:

```
https://graph.microsoft.com/v1.0/me/events?$top=5
```

4. Select **Run Query**.

The results are shown in the **Response Preview** pane.

Note the `@odata.nextLink` property finishes in `$skip 5`. Using the next link here would fetch the next five records in the data set. To fetch the next page set the `\$skip` value to 5:

```
https://graph.microsoft.com/v1.0/me/events?$top=5&$skip=5
```

This is the same as the `@odata.nextLink` returned for this resource. The `$skip` value tells Microsoft Graph to start returning results after the number provided here.

5. Select **Run Query**.

The results are shown in the **Response Preview** pane. The value returned contains an `@odata.nextLink` that can be followed. If a client needs to load all data from a specific resource or query it should keep following the `@odata.nextLink` until it is no longer present in the results.

**NOTE:** The formula for composing manual pagination queries is this:

```
?$top={pageSize}&$skip={pageSize}*({pageNumber} - 1)}
```

## Task 6: Expand and retrieve resources using \\$expand query parameter

In this exercise you will fetch the attachments from the mail of the current user. In a previous task you found the messages for the current user, which have attachments. In the request URL box amend the request to perform this query again.

1. In the **request URL** box amend the request to:

```
https://graph.microsoft.com/v1.0/me/messages?$filter=hasAttachments eq true
```

2. Select **Run Query**.

The results are shown in the **Response Preview** pane. Note when examining the results there is no information provided about the attachments. Amend the query to fetch information about the messages' attachments.

1. In the **request URL** box amend the request to:

```
https://graph.microsoft.com/v1.0/me/messages?$filter=hasAttachments eq true&$expand=attachments
```

2. Select **Run Query**.

The results are shown in the **Response Preview** pane.

Note the attachment objects included in the response include the **contentBytes** property. This is the actual file content and can cause the application to fetch far more data than might be desired.

3. Select the **Response Headers** tab.

Note the value for the **content-length** header.

4. Amend the query to use **\$select** on the **\$expanded** attachments collection. In the **request URL** box amend the request to:

```
https://graph.microsoft.com/v1.0/me/messages?$filter=hasAttachments eq true&$expand=attachments($select=id,name,contentType,size)
```

5. Select **Run Query**.

Note the new value for the **content-length** header is much smaller than the previous request.

6. Select **Response Preview**.

Note that for each attachment, in addition to the explicitly requested properties, there are some additional **@odata** properties.

## Task 7: Use `\$count` to discover the total number of matching resources

1. In the request URL box amend the request to:

```
https://graph.microsoft.com/v1.0/me/messages?$count=true
```

2. Select **Run Query**.

In the **Response Preview** pane there is an additional property shown `@odata.count`. This value shows the total number of resources that match the provided query. Note that as no query has been explicitly provided, the `@odata.count` value is that of all resources at this path.

3. Add a query, in this instance query to see how many unread messages the current user has. In the **request URL** box amend the request to:

```
https://graph.microsoft.com/v1.0/me/messages?$count=true&$filter=isRead eq false
```

4. Select **Run Query**.

In the **Response Preview** pane, the first page displays matching results. Note the `@odata.count` value is updated to reflect the query passed via the `$filter` parameter.

## Task 8: Use \\$search to discover the total number of matching resources

1. In the request URL box amend the request to:

```
https://graph.microsoft.com/v1.0/me/messages?$search="business"
```

2. Select **Run Query**.

In the **Response Preview** the messages that have matches for the keyword supplied are shown. Note the number of messages that match (4). The scope of the search can be adjusted specifying property names that are recognized by the KQL syntax.

See <https://docs.microsoft.com/en-us/graph/query-parameters#search-parameter> for more information.

1. Amend the **request URL** box to scope the search to only target the subject field. In the **request URL** box amend the request to:

```
https://graph.microsoft.com/v1.0/me/messages?$search="subject:business"
```

2. Select **Run Query**.

In the **Response Preview** the messages that have matches in the specified property are shown. Note there should be fewer results (3).

## Review

In this exercise, you learned how to make use of query parameters:

- Fetch resources with properties matching certain parameters by using the `$filter` query parameter.
- Fetch only the necessary data by using the `$select` query parameter.
- Set the order of results using `$orderby` query parameter.
- Set page size of results using `$skip` and `$top` query parameters.
- Expand and retrieve additional resources using `$expand` query parameter.
- Retrieve the total count of matching resources using `$count` query parameter.
- Search for resources using `$search` query parameter.

### ❖❖❖# Exercise 2: Retrieve and control information returned from Microsoft Graph

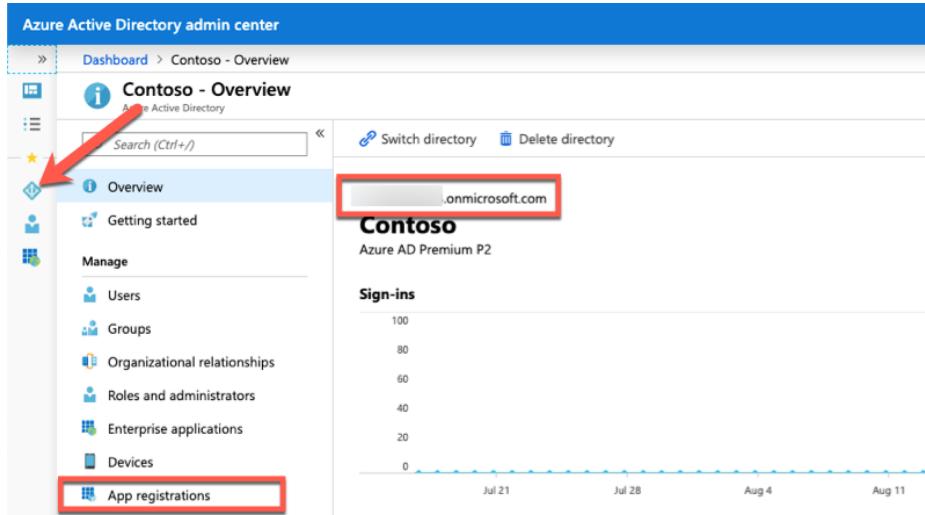
In this exercise, you will create a new Azure AD web application registration using the Azure Active Directory (Azure AD) admin center, a .NET Core console application, and query Microsoft Graph.

By the end of this exercise you will be able to use the following queries:

- `$select`
- `$top`
- `$orderby`
- `$filter`

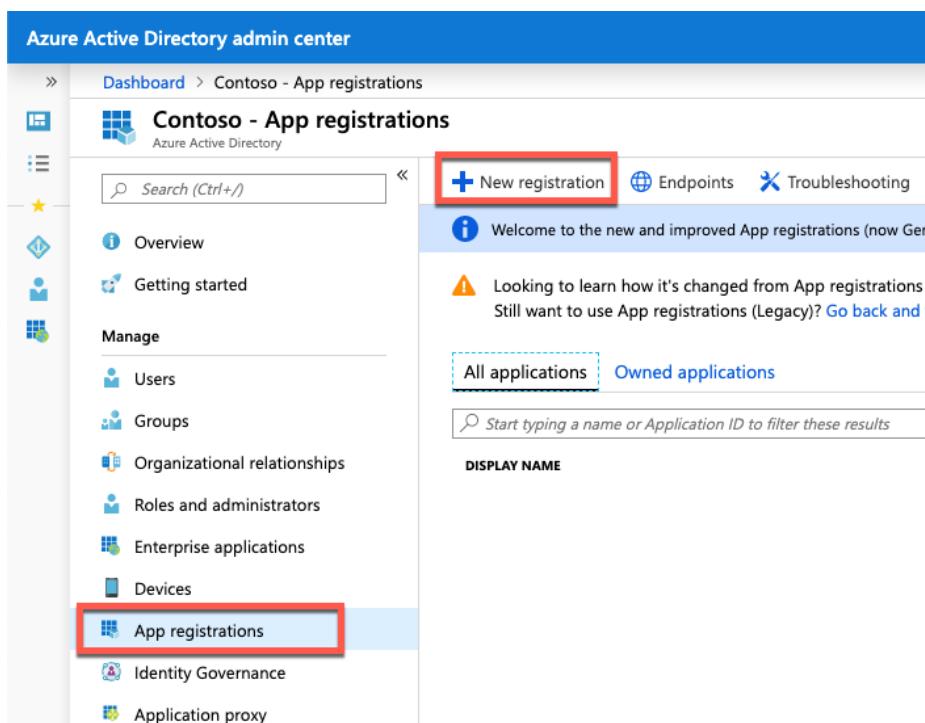
## Task 1: Create an Azure AD application

1. Open a browser and navigate to the [Azure Active Directory admin center](https://aad.portal.azure.com) (<https://aad.portal.azure.com>). Sign in using a **Work or School Account** that has global administrator rights to the tenancy.
2. Select **Azure Active Directory** in the leftmost navigation panel.



The screenshot shows the Azure Active Directory admin center interface. The left sidebar has a 'Manage' section with several options: Users, Groups, Organizational relationships, Roles and administrators, Enterprise applications, Devices, and App registrations. The 'App registrations' option is highlighted with a red box. The main content area displays 'Contoso - Overview' for 'Azure AD Premium P2'. It includes a search bar, a 'Sign-ins' chart showing activity from July 21 to Aug 11, and a 'Contoso' section with a .onmicrosoft.com URL.

3. Select **Manage > App registrations** in the left navigation panel.
4. On the **App registrations** page, select **New registration**.



The screenshot shows the 'Contoso - App registrations' page. The left sidebar has a 'Manage' section with options: Users, Groups, Organizational relationships, Roles and administrators, Enterprise applications, Devices, and App registrations. The 'App registrations' option is highlighted with a red box. The main content area features a 'New registration' button highlighted with a red box, a welcome message, a note about legacy app registrations, and tabs for 'All applications' and 'Owned applications'. A search bar and a 'DISPLAY NAME' input field are also present.

5. On the **Register an application** page, set the values as follows:
  - o **Name:** Graph Console App

- **Supported account types:** Accounts in this organizational directory only (Contoso only - Single tenant)
- **Redirect URI:** Web = <https://localhost>

\* Name

The user-facing display name for this application (this can be changed later).

Graph Console App ✓

#### Supported account types

Who can use this application or access this API?

- Accounts in this organizational directory only (Contoso only - Single tenant)
- Accounts in any organizational directory (Any Azure AD directory - Multitenant)
- Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)
- Personal Microsoft accounts only

[Help me choose...](#)

#### Redirect URI (optional)

We'll return the authentication response to this URI after successfully authenticating the user. Providing this now is optional and it can be changed later, but a value is required for most authentication scenarios.

Web ✓ https://localhost ✓

By proceeding, you agree to the Microsoft Platform Policies [↗](#)

[Register](#)

## 6. Select Register.

## 7. On the **Graph Console App** overview page, copy the value of the **Application (client) ID** and **Directory (tenant) ID**; you will need them later in this exercise.

The screenshot shows the Microsoft Azure portal's App registrations section. On the left, there's a sidebar with links like Overview, Quickstart, Manage, Branding, Authentication, Certificates & secrets, API permissions, Expose an API, Owners, Roles and administrators, Manifest, Support + Troubleshooting, Troubleshooting, and New support request. The main area is titled "Graph Console App". It has tabs for Delete and Endpoints. A message says "Welcome to the new and improved App registrations. Looking to learn how it's changed from App registrations (Legacy)?". Below this, there's a "Display name" field with "Graph Console App". To the right, under "Supported account types", it says "My organization only". Under "Redirect URIs", it lists "1 web, 0 public client" with "https://localhost" as the value. Under "Managed application in local directory", it lists "Graph Console App". A red box highlights the "Object ID" field, which contains "52168e8f-208c-466b-a8a6-86adff64af2f". At the bottom, there are sections for "Call APIs" (with icons for various Microsoft services) and "Documentation" (links to Microsoft identity platform, Authentication scenarios, Authentication libraries, Code samples, Microsoft Graph, Glossary, and Help and Support). A "View API Permissions" button is also present.

## 8. Select Manage > Certificates & secrets.

## 9. Select New client secret.

**GraphNotificationTutorial - Certificates & secrets**

Certificates enable applications to identify themselves to the authentication service when receiving tokens at a web addressable location (using an HTTPS scheme). For a higher level of assurance, we recommend using a certificate (instead of a client secret) as a credential.

**Certificates**

Certificates can be used as secrets to prove the application's identity when requesting a token. Also can be referred to as public keys.

**Client secrets**

A secret string that the application uses to prove its identity when requesting a token. Also can be referred to as application password.

| DESCRIPTION   | EXPIRES | VALUE |
|---|---------|-------|
| No client secrets have been created for this application. |         |       |

10. In the **Add a client secret** page, enter a value in **Description**, select one of the options for **Expires**, and select **Add**.

**Home > Contoso - App registrations (Preview) > GraphNotificationTutorial - Certificates & secrets**

**GraphNotificationTutorial - Certificates & secrets**

**Add a client secret**

Description  
Forever

Expires  
 In 1 year  
 In 2 years  
 Never

**Add** **Cancel**

No certificates have been added for this application.

**Client secrets**

A secret string that the application uses to prove its identity when requesting a token. Also can be referred to as application password.

| DESCRIPTION   | EXPIRES | VALUE |
|---|---------|-------|
| No client secrets have been created for this application. |         |       |

11. Copy the client secret value before you leave this page. You will need it in the next step.

**Important:** This client secret is never shown again, so make sure you copy it now.

| DESCRIPTION | EXPIRES    | VALUE                                |
|-------------|------------|--------------------------------------|
| Forever     | 12/31/2299 | 00000000-0000-0000-0000-000000000000 |

12. Grant Azure AD application permissions to Microsoft Graph. After creating the application, you need to grant it the necessary permissions to Microsoft Graph. Select **API Permissions** in the leftmost navigation panel.

The screenshot shows the Azure Active Directory admin center interface. The top navigation bar includes 'Dashboard', 'Contoso - App registrations', and 'Graph Console App'. The left sidebar has sections like 'Overview', 'Quickstart', 'Manage', 'Branding', 'Authentication', 'Certificates & secrets', and 'API permissions'. The 'API permissions' link is highlighted with a red box. The main content area displays the app's details: 'Display name: Graph Console App', 'Application (client) ID: 9ef0bc7f-6919-439e-b027-294438560213', 'Directory (tenant) ID: 7adff0e7f-07b3-4eb5-ba39-7ea421035371', and 'Object ID: 52168e8f-208c-466b-a8a6-86adff64af2f'. Below this is a section titled 'Call APIs' with various Microsoft service icons. A callout text states: 'Build more powerful apps with rich user and business data from Microsoft services and your own company's data sources.'

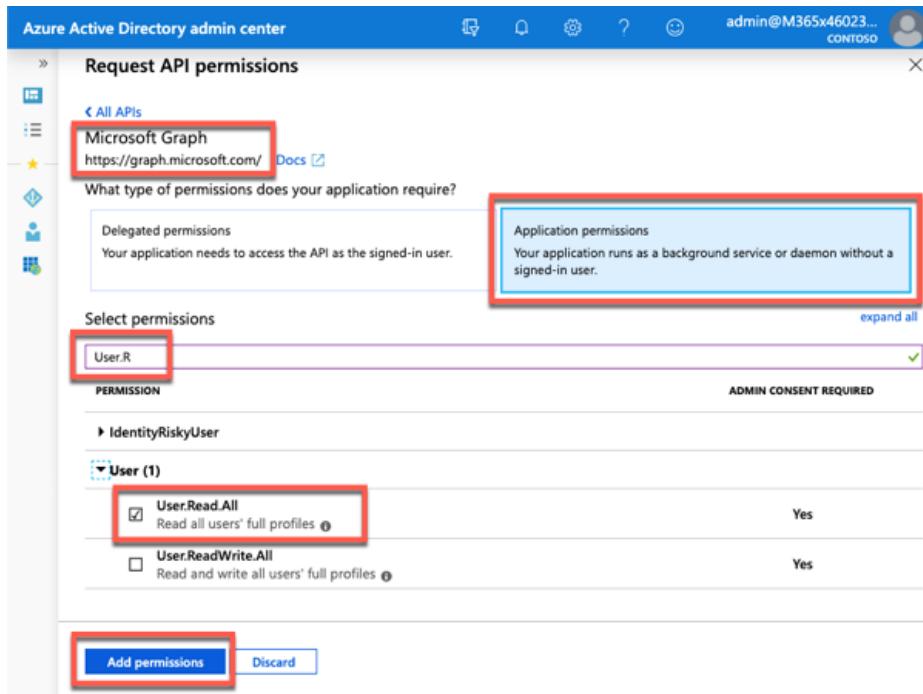
13. Select the **Add a permission** button.

The screenshot shows the 'API permissions' page for the 'Graph Console App'. It includes a note about applications being authorized to call APIs when granted permissions. A red box highlights the '+ Add a permission' button. Below it is a table with one entry: 'Microsoft Graph (1)' under 'API / PERMISSIONS NAME', 'Delegated' under 'TYPE', and 'Sign in and read user profile' under 'DESCRIPTION'. A note at the bottom says: 'These are the permissions that this application requests statically. You may also request user consentable permissions dynamically through code. See best practices for requesting permissions'.

14. In the Request API permissions panel that appears, select **Microsoft Graph** from the **Microsoft APIs** tab.

The screenshot shows the 'Request API permissions' dialog box. At the top, there's a 'Select an API' dropdown with 'Microsoft APIs' selected, which is highlighted with a red box. Below it are tabs for 'APIs my organization uses' and 'My APIs'. The main area shows 'Commonly used Microsoft APIs' with a section for 'Microsoft Graph'. This section includes a description: 'Take advantage of the tremendous amount of data in Office 365, Enterprise Mobility + Security, and Windows 10. Access Azure AD, Excel, Intune, Outlook/Exchange, OneDrive, OneNote, SharePoint, Planner, and more through a single endpoint.' To the right is a grid of icons representing various Microsoft services. Below this are three cards: 'Azure Rights Management Services', 'Azure Service Management', and 'Azure Storage'.

15. When prompted for the type of permission, select **Application permissions**.



16. Enter **User.R** in the **Select permissions** search box and select the **User.Read.All** permission, followed by the **Add permission** button at the bottom of the panel.
17. Under the **Configured permissions** section, select the button **Grant admin consent for [tenant]**, followed by the **Yes** button to grant all users in your organization this permission.

## Task 2: Create .NET Core console application

1. Open your command prompt, navigate to a directory where you have rights to create your project, and run the following command to create a new .NET Core console application:  
`dotnet new console -o graphconsoleapp`
2. After creating the application, run the following commands to ensure your new project runs correctly.

```
dotnetcli cd graphconsoleapp dotnet add package Microsoft.Identity.Client  
dotnet add package Microsoft.Graph dotnet add package  
Microsoft.Extensions.Configuration dotnet add package  
Microsoft.Extensions.Configuration.FileExtensions dotnet add package  
Microsoft.Extensions.Configuration.Json
```

3. Open the application in Visual Studio Code using the following command: `code .`
4. If Visual Studio Code displays a dialog box asking if you want to add required assets to the project, select **Yes**.

## Task 3: Update the console app to support Azure AD authentication

1. Create a new file named **appsettings.json** in the root of the project and add the following code to it:

```
json { "tenantId": "YOUR_TENANT_ID_HERE", "applicationId":  
"YOUR_APP_ID_HERE", "applicationSecret": "YOUR_APP_SECRET_HERE",  
"redirectUri": "YOUR_REDIRECT_URI_HERE" }
```

2. Update properties with the following values:

- **YOUR\_TENANT\_ID\_HERE**: Azure AD directory ID
- **YOUR\_APP\_ID\_HERE**: Azure AD client ID
- **YOUR\_APP\_SECRET\_HERE**: Azure AD client secret
- **YOUR\_REDIRECT\_URI\_HERE**: redirect URI you entered when creating the Azure AD app (*for example, https://localhost*)

## Task 4: Create helper classes

1. Create a new folder **Helpers** in the project.
2. Create a new file **AuthHandler.cs** in the **Helpers** folder and add the following code:

```
csharp using System.Net.Http; using System.Threading.Tasks; using Microsoft.Graph; using System.Threading; namespace Helpers { public class AuthHandler : DelegatingHandler { private IAuthenticationProvider _authenticationProvider; public AuthHandler(IAuthenticationProvider authenticationProvider, HttpMessageHandler innerHandler) { InnerHandler = innerHandler; _authenticationProvider = authenticationProvider; } protected override async Task<HttpResponseMessage> SendAsync(HttpRequestMessage request, CancellationToken cancellationToken) { await _authenticationProvider.AuthenticateRequestAsync(request); return await base.SendAsync(request, cancellationToken); } } }
```

3. Create a new file **MsalAuthenticationProvider.cs** in the **Helpers** folder and add the following code:

```
csharp using System.Net.Http; using System.Net.Http.Headers; using System.Threading.Tasks; using Microsoft.Identity.Client; using Microsoft.Graph; namespace Helpers { public class MsalAuthenticationProvider : IAuthenticationProvider { private IConfidentialClientApplication _clientApplication; private string[] _scopes; public MsalAuthenticationProvider(IConfidentialClientApplication clientApplication, string[] scopes) { _clientApplication = clientApplication; _scopes = scopes; } public async Task AuthenticateRequestAsync(HttpRequestMessage request) { var token = await GetTokenAsync(); request.Headers.Authorization = new AuthenticationHeaderValue("bearer", token); } public async Task<string> GetTokenAsync() { AuthenticationResult authResult = null; authResult = await _clientApplication.AcquireTokenForClient(_scopes).ExecuteAsync(); return authResult.AccessToken; } } }
```

## Task 5: Incorporate Microsoft Graph into the console app

1. Open the **Program.cs** file and add the following using statements to the top of the file below using System:

```
csharp using System.Collections.Generic; using Microsoft.Identity.Client;  
using Microsoft.Graph; using Microsoft.Extensions.Configuration; using  
Helpers;
```

2. Add the following static member to the Program class in the **Program.cs** file. This member will be used to instantiate the client used to call Microsoft Graph:

```
csharp private static GraphServiceClient _graphClient;
```

3. Add the following method **LoadAppSettings** to the **Program** class. This method retrieves the configuration details from the **appsettings.json** file previously created:

```
csharp private static IConfigurationRoot LoadAppSettings() { try { var  
config = new ConfigurationBuilder()  
.SetBasePath(System.IO.Directory.GetCurrentDirectory())  
.AddJsonFile("appsettings.json", false, true) .Build(); if  
(string.IsNullOrEmpty(config["applicationId"])) ||  
string.IsNullOrEmpty(config["applicationSecret"]) ||  
string.IsNullOrEmpty(config["redirectUri"]) ||  
string.IsNullOrEmpty(config["tenantId"])) { return null; } return config; }  
catch (System.IO.FileNotFoundException) { return null; } }
```

4. Add the following method **CreateAuthorizationProvider** to the **Program** class. This method will create an instance of the clients used to call Microsoft Graph:

```
csharp private static IAuthenticationProvider  
CreateAuthorizationProvider(IConfigurationRoot config) { var clientId =  
config["applicationId"]; var clientSecret = config["applicationSecret"];  
var redirectUri = config["redirectUri"]; var authority =  
$"https://login.microsoftonline.com/{config["tenantId"]}/v2.0";  
List<string> scopes = new List<string>();  
scopes.Add("https://graph.microsoft.com/.default"); var cca =  
ConfidentialClientApplicationBuilder.Create(clientId)  
.WithAuthority(authority) .WithRedirectUri(redirectUri)  
.WithClientSecret(clientSecret) .Build(); return new  
MsalAuthenticationProvider(cca, scopes.ToArray()); }
```

5. Add the following method **GetAuthenticatedGraphClient** to the **Program** class. This method creates an instance of the **GraphServiceClient** object:

```
csharp private static GraphServiceClient  
GetAuthenticatedGraphClient(IConfigurationRoot config) { var  
authenticationProvider = CreateAuthorizationProvider(config); _graphClient  
= new GraphServiceClient(authenticationProvider); return _graphClient; }
```

6. Locate the **Main** method in the **Program** class. Add the following code to the end of the **Main** method to load the configuration settings from the **appsettings.json** file:

```
csharp var config = LoadAppSettings(); if (config == null) {  
Console.WriteLine("Invalid appsettings.json file."); return; }
```

7. Add the following code to the end of the **Main** method, just after the code added in the last step. This code will obtain an authenticated instance of the **GraphServicesClient** and submit a request for the first user.

```
csharp var client = GetAuthenticatedGraphClient(config); var graphRequest =  
client.Users.Request(); var results = graphRequest.GetAsync().Result;  
foreach(var user in results) { Console.WriteLine(user.Id + ": " +  
user.DisplayName + " <" + user.Mail + ">"); } Console.WriteLine("\nGraph  
Request:");  
Console.WriteLine(graphRequest.GetHttpRequestMessage() .RequestUri);
```

## Task 6: Build and test the application

1. Run the following command in a command prompt to compile the console application:  
dotnet build

2. Run the following command to run the console application: dotnet run

3. When the application runs, you'll see a list of users displayed. The query retrieved all information about the users.

```
Hello World!
f783718b-65e1-4a10-898a-fff0a217ab41: Conf Room Adams <Adams@M365x460234.onmicrosoft.com>
7347eda0-9cd3-4e12-ab0f-a548296ec4b5: Adele Vance <AdeleV@M365x460234.OnMicrosoft.com>
cd2196f2-2643-4995-86be-337dda89371c: MOD Administrator <admin@M365x460234.OnMicrosoft.com>
68b23cce-2ca8-4aae-9816-4617b5cc4226: Alex Wilber <AlexW@M365x460234.OnMicrosoft.com>
74a31520-6eae-437a-ade0-6e7082fac0a2: Allan Deyoung <AllanD@M365x460234.OnMicrosoft.com>
4b9785a6-ab4b-4e4f-a4fb-8db740b3722a: Conf Room Baker <Baker@M365x460234.onmicrosoft.com>
b0e4b027-0885-4ff9-ae52-6bf1d2d14c6c: Bianca Pisani <>
131cad4a-b855-4089-81cb-5635dee3c5a5: Brian Johnson (TAILSPIN) <BrianJ@M365x460234.onmicrosoft.com>
67fed7f6-126c-4d67-b0aa-9c5a8981e75d: Cameron White <>
cc9a10db-a351-4d96-8db7-c403b9783575: Christie Cline <ChristieC@M365x460234.OnMicrosoft.com>
8741baed-0302-421b-bf2b-1e1cd273fd61: Conf Room Crystal <Crystal@M365x460234.onmicrosoft.com>
a60f8373-c1c7-4db8-b120-8f6faa876fc2: Debra Berger <DebraB@M365x460234.OnMicrosoft.com>
0f941563-adc5-4078-951f-30b2685b726e: Delia Dennis <>
d4442e0c-7be5-4324-8fd0-ebc63efeb3c3: Diego Siciliani <DiegoS@M365x460234.OnMicrosoft.com>
75d965c4-59b8-4e22-bc45-4d7c32e7d8fe: Emily Braun <EmilyB@M365x460234.OnMicrosoft.com>
577914b1-a163-419f-80b9-0a1fa69ce0b7: Enrico Cattaneo <EnricoC@M365x460234.OnMicrosoft.com>
5964f551-5b78-4ab7-a0a2-cbb06ca9eab: Gerhart Moller <>
80620a39-a095-4bd9-9e82-13972cc7d6e1: Grady Archie <GradyA@M365x460234.OnMicrosoft.com>
295047c6-af95-4bdf-a260-b769c4b116fe: Henrietta Mueller <HenriettaM@M365x460234.OnMicrosoft.com>
c66ddbec-f041-4e1f-927e-640bb62e62fe: Conf Room Hood <Hood@M365x460234.onmicrosoft.com>
56c8e653-5d9a-48a4-8fca-d84712455dfa: Irvin Sayers <IrvinS@M365x460234.OnMicrosoft.com>
ad112ce7-a8fc-4b48-bc66-9f906b2ceaf8: Isaiah Langer <IsaiahL@M365x460234.OnMicrosoft.com>
cd40b574-588d-42cd-8f20-67a9faf0f4ba: Johanna Lorenz <JohannaL@M365x460234.OnMicrosoft.com>
0aca7432-77e8-4d5f-bae0-f2f5c24db3be: Joni Sherman <JoniS@M365x460234.OnMicrosoft.com>
c10de9f6-5e4d-4e15-ae44-cc581ceb5f2: Jordan Miller <JordanM@M365x460234.OnMicrosoft.com>
3ee7b72a-9585-4529-a3e6-85d48c7d4968: Lee Gu <LeeG@M365x460234.OnMicrosoft.com>
d2b03165-278d-4816-9dad-7340d36a0703: Lidia Holloway <LidiaH@M365x460234.OnMicrosoft.com>
53036a17-d314-4581-a880-fba271a1bbd1: Lynne Robbins <LynneR@M365x460234.OnMicrosoft.com>
bbc8446f-f788-4527-8cf6-c40e65228539: Megan Bowen <MeganB@M365x460234.OnMicrosoft.com>
526f3f98-09d2-4c98-8e64-0aa8e4128241: Miriam Graham <MiriamG@M365x460234.OnMicrosoft.com>
ea1ffc20-a151-425a-a5aa-1a639e9ela79: Nestor Wilke <NestorW@M365x460234.OnMicrosoft.com>
12d13880-afb3-4721-a6d9-f296d1dfaf1fc: Patti Fernandez <PattiF@M365x460234.OnMicrosoft.com>
5d3df140-5682-480c-98c8-fc9ec7bdb2d: Pradeep Gupta <PradeepG@M365x460234.OnMicrosoft.com>
6589d3e8-1ebf-437a-aea3-55bcacfc11d2: Conf Room Rainier <Rainier@M365x460234.onmicrosoft.com>
95c4855d-94a9-4ceb-bfd8-da93f83f63ab: Raul Razo <>
d3963c26-01b1-4cdd-af99-21ea89176353: Conf Room Stevens <Stevens@M365x460234.onmicrosoft.com>
```

**Note:** Notice the URL written to the console. This is the entire request, including query parameters, that the Microsoft Graph SDK is generating. Take note for each query you run in this exercise.

## Task 7: Edit the application to optimize the query

The current console application isn't efficient because it retrieves all information about all users in your organization but only displays three properties. The **\$select** query parameter can limit the amount of data that is returned by Microsoft Graph, optimizing the query.

1. Update the line that starts with `var results = graphRequest` in the **Main** method with the following to limit the query to just two properties:

```
csharp var results = graphRequest .Select(u => new { u.DisplayName, u.Mail }) .GetAsync() .Result;
```

2. Rebuild and rerun the console application by executing the following commands in the command line:

```
csharp dotnet build dotnet run
```

3. Notice that the **ID** property isn't populated with data, as it wasn't included in the **\$select** query parameter.

```
Hello World!
: Conf Room Adams <Adams@M365x460234.onmicrosoft.com>
: Adele Vance <AdeleV@M365x460234.OnMicrosoft.com>
: MOD Administrator <admin@M365x460234.OnMicrosoft.com>
: Alex Wilber <AlexW@M365x460234.OnMicrosoft.com>
: Allan Deyoung <AllanD@M365x460234.OnMicrosoft.com>
: Conf Room Baker <Baker@M365x460234.onmicrosoft.com>
: Bianca Pisani <>
: Brian Johnson (TAILSPIN) <BrianJ@M365x460234.onmicrosoft.com>
: Cameron White <>
: Christie Cline <ChristieC@M365x460234.OnMicrosoft.com>
: Conf Room Crystal <Crystal@M365x460234.onmicrosoft.com>
: Debra Berger <DebraB@M365x460234.OnMicrosoft.com>
: Delia Dennis <>
: Diego Siciliani <DiegoS@M365x460234.OnMicrosoft.com>
: Emily Braun <EmilyB@M365x460234.OnMicrosoft.com>
: Enrico Cattaneo <EnricoC@M365x460234.OnMicrosoft.com>
: Gerhart Moller <>
: Grady Archie <GradyA@M365x460234.OnMicrosoft.com>
: Henrietta Mueller <HenriettaM@M365x460234.OnMicrosoft.com>
: Conf Room Hood <Hood@M365x460234.onmicrosoft.com>
: Irvin Sayers <IrvinS@M365x460234.OnMicrosoft.com>
: Isaiah Langer <IsaiahL@M365x460234.OnMicrosoft.com>
: Johanna Lorenz <JohannaL@M365x460234.OnMicrosoft.com>
: Joni Sherman <JoniS@M365x460234.OnMicrosoft.com>
: Jordan Miller <JordanM@M365x460234.OnMicrosoft.com>
: Lee Gu <LeeG@M365x460234.OnMicrosoft.com>
: Lidia Holloway <LidiaH@M365x460234.OnMicrosoft.com>
: Lynne Robbins <LynneR@M365x460234.OnMicrosoft.com>
: Megan Bowen <MeganB@M365x460234.OnMicrosoft.com>
: Miriam Graham <MiriamG@M365x460234.OnMicrosoft.com>
: Nestor Wilke <NestorW@M365x460234.OnMicrosoft.com>
: Patti Fernandez <PattiF@M365x460234.OnMicrosoft.com>
: Pradeep Gupta <PradeepG@M365x460234.OnMicrosoft.com>
: Conf Room Rainier <Rainier@M365x460234.onmicrosoft.com>
: Raul Razo <>
: Conf Room Stevens <Stevens@M365x460234.onmicrosoft.com>
```

4. Let us further limit the results to just the first 15 results. Update the line that starts with `var results = graphRequest` in the **Main** method with the following:

```
csharp var results = graphRequest .Select(u => new { u.DisplayName, u.Mail }) .Top(15) .GetAsync() .Result;
```

5. Rebuild and rerun the console application by executing the following commands in the command line:

```
csharp dotnet build dotnet run
```

6. Notice only 15 items are now returned by the query.

```
Hello World!
: Conf Room Adams <Adams@M365x460234.onmicrosoft.com>
: Adele Vance <AdeleV@M365x460234.OnMicrosoft.com>
: MOD Administrator <admin@M365x460234.OnMicrosoft.com>
: Alex Wilber <AlexW@M365x460234.OnMicrosoft.com>
: Allan Deyoung <AllanD@M365x460234.OnMicrosoft.com>
: Conf Room Baker <Baker@M365x460234.onmicrosoft.com>
: Bianca Pisani <>
: Brian Johnson (TAILSPIN) <BrianJ@M365x460234.onmicrosoft.com>
: Cameron White <>
: Christie Cline <ChristieC@M365x460234.OnMicrosoft.com>
: Conf Room Crystal <Crystal@M365x460234.onmicrosoft.com>
: Debra Berger <DebraB@M365x460234.OnMicrosoft.com>
: Delia Dennis <>
: Diego Siciliani <DiegoS@M365x460234.OnMicrosoft.com>
: Emily Braun <EmilyB@M365x460234.OnMicrosoft.com>
```

7. Sort the results in reverse alphabetic order. Update the line that starts with `var results = graphRequest` in the **Main** method with the following:

```
csharp var results = graphRequest .Select(u => new { u.DisplayName, u.Mail }) .Top(15) .OrderBy("DisplayName desc") .GetAsync() .Result;
```

8. Rebuild and rerun the console application by executing the following commands in the command line:

```
csharp dotnet build dotnet run
```

```
Hello World!
: Raul Razo <>
: Pradeep Gupta <PradeepG@M365x460234.OnMicrosoft.com>
: Patti Fernandez <PattiF@M365x460234.OnMicrosoft.com>
: Nestor Wilke <NestorW@M365x460234.OnMicrosoft.com>
: MOD Administrator <admin@M365x460234.OnMicrosoft.com>
: Miriam Graham <MiriamG@M365x460234.OnMicrosoft.com>
: Megan Bowen <MeganB@M365x460234.OnMicrosoft.com>
: Lynne Robbins <LynneR@M365x460234.OnMicrosoft.com>
: Lidia Holloway <LidiaH@M365x460234.OnMicrosoft.com>
: Lee Gu <LeeG@M365x460234.OnMicrosoft.com>
: Jordan Miller <JordanM@M365x460234.OnMicrosoft.com>
: Joni Sherman <JoniS@M365x460234.OnMicrosoft.com>
: Johanna Lorenz <JohannaL@M365x460234.OnMicrosoft.com>
: Isaiah Langer <IsaiahL@M365x460234.OnMicrosoft.com>
: Irvin Sayers <IrvinS@M365x460234.OnMicrosoft.com>
```

9. Further refine the results by selecting users whose surname starts with A, B, or C. You'll need to remove the `\$orderby` query parameter added previously as the `Users` endpoint doesn't support combining the `\$filter` and `\$orderby` parameters. Update the line that starts with `var results = graphRequest` in the **Main** method with the following:

```
csharp var results = graphRequest .Select(u => new { u.DisplayName, u.Mail }) .Top(15) // .OrderBy("DisplayName desc") .Filter("startsWith(surname,'A') or startsWith(surname,'B') or startsWith(surname,'C')") .GetAsync() .Result;
```

10. Rebuild and rerun the console application by executing the following commands in the command line:

```
powershell dotnet build dotnet run
```

```
Hello World!
: Debra Berger <DebraB@M365x460234.OnMicrosoft.com>
: Megan Bowen <MeganB@M365x460234.OnMicrosoft.com>
: Emily Braun <EmilyB@M365x460234.OnMicrosoft.com>
: MOD Administrator <admin@M365x460234.OnMicrosoft.com>
: Grady Archie <GradyA@M365x460234.OnMicrosoft.com>
: Enrico Cattaneo <EnricoC@M365x460234.OnMicrosoft.com>
: Christie Cline <ChristieC@M365x460234.OnMicrosoft.com>
```

## Review

In this exercise, you created an Azure AD application and .NET console application that retrieved user data from Microsoft Graph. You then used query parameters to limit and manipulate the data returned by Microsoft Graph to optimize the query.

### ◆◆◆# Exercise 3: Using change notifications and track changes with Microsoft Graph

This tutorial teaches you how to build a .NET Core app that uses the Microsoft Graph API to receive notifications (webhooks) when a user account changes in Azure AD and perform queries using the delta query API to receive all changes to user accounts since the last query was made.

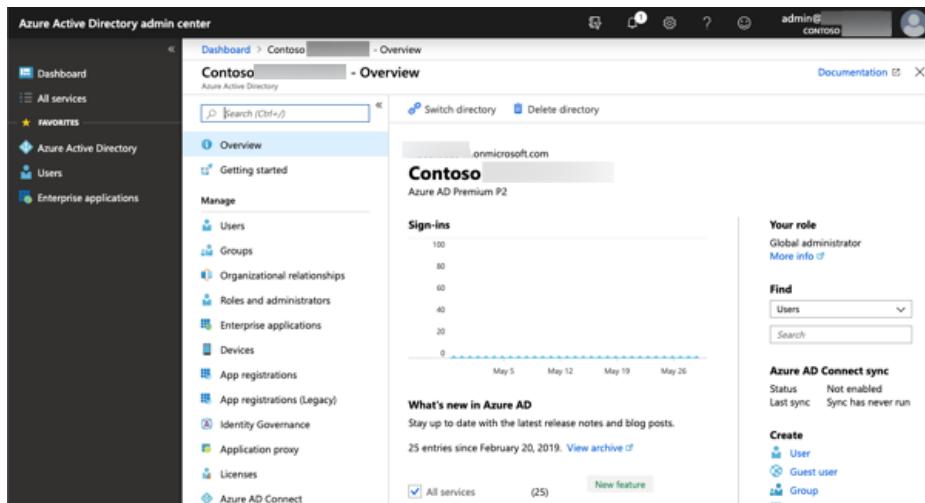
By the end of this exercise you will be able to:

- Monitor for changes using change notifications
- Get changes using a delta query

# Task 1: Create a new Azure AD web application

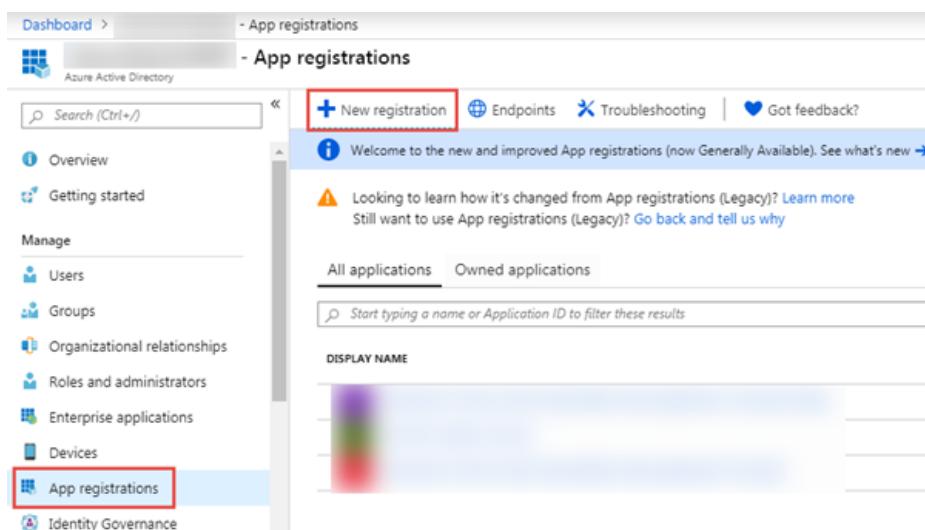
In this exercise, you will create a new Azure AD web application registration using the Azure AD admin center and grant administrator consent to the required permission scopes.

1. Open a browser and navigate to the Azure AD admin center <https://aad.portal.azure.com>. Sign in using a **Work or School Account**.
2. Select **Azure Active Directory** in the leftmost navigation panel, then select **App registrations** under **Manage**.



The screenshot shows the Azure Active Directory admin center interface. On the left, there's a navigation sidebar with links like Dashboard, All services, Favorites (Azure Active Directory, Users, Enterprise applications), and Manage (Users, Groups, Organizational relationships, Roles and administrators, Enterprise applications, Devices, App registrations, App registrations (Legacy), Identity Governance, Application proxy, Licenses, Azure AD Connect). The main content area is titled 'Contoso - Overview' and shows a chart for 'Sign-ins' from May 5 to May 26. It also includes sections for 'What's new in Azure AD', 'Azure AD Connect sync' status (Not enabled), and a 'Create' section with options for User, Guest user, and Group. The 'App registrations' link in the Manage section is highlighted with a red box.

3. Select **New registration** on the Register an application page.



The screenshot shows the 'App registrations' page. The left sidebar has a link for 'App registrations' which is highlighted with a red box. The main area has a 'New registration' button with a plus sign, which is also highlighted with a red box. There are tabs for 'Endpoints', 'Troubleshooting', and 'Got feedback?'. A welcome message says 'Welcome to the new and improved App registrations (now Generally Available). See what's new →'. A note says 'Looking to learn how it's changed from App registrations (Legacy)? Learn more' and 'Still want to use App registrations (Legacy)? Go back and tell us why'. Below this are tabs for 'All applications' and 'Owned applications', and a search bar. The 'DISPLAY NAME' section shows a blurred placeholder image.

4. Set the values as follows:

- **Name:** Graph Notification Tutorial
- **Supported account types:** Accounts in any organizational directory and personal Microsoft accounts
- **Redirect URI:** Web > <http://localhost>

\* Name  
The user-facing display name for this application (this can be changed later).

#### Supported account types

Who can use this application or access this API?

- Accounts in this organizational directory only (Contoso only - Single tenant)
- Accounts in any organizational directory (Any Azure AD directory - Multitenant)
- Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)
- Personal Microsoft accounts only

[Help me choose...](#)

#### Redirect URI (optional)

We'll return the authentication response to this URI after successfully authenticating the user. Providing this now is optional and it can be changed later, but a value is required for most authentication scenarios.

## 5. Select Register.

6. On the **Graph Notification Tutorial** page, copy the value of the **Application (client) ID** and **Directory (tenant) ID**; save it, you will need them later in the tutorial.

Dashboard > Contoso M365x995941 - App registrations > GraphNotificationTutorial

**GraphNotificationTutorial**

Display name : GraphNotificationTutorial

Application (client) ID : a6079933-  
-bcc77ebd0770

Directory (tenant) ID : 785b4ab0-  
-c6502a7da2af

Object ID : 31500560-f812-4090-b806-97c6836778b4

Supported account types : All Microsoft account users

Redirect URIs : 1 web, 0 public client

Managed application in ... : GraphNotificationTutorial

## 7. Select Manage > Certificates & secrets.

### 8. Select New client secret.

9. Enter a value in **Description**, select one of the options for **Expires**, and select **Add**.

GraphNotificationTutorial - Certificates & secrets

Certificates enable applications to identify themselves to the authentication service when receiving tokens at a web addressable location (using an HTTPS scheme). For a higher level of assurance, we recommend using a certificate (instead of a client secret) as a credential.

**Certificates**

Certificates can be used as secrets to prove the application's identity when requesting a token. Also can be referred to as public keys.

**Client secrets**

A secret string that the application uses to prove its identity when requesting a token. Also can be referred to as application password.

| New client secret |         |       |
|-------------------|---------|-------|
| DESCRIPTION       | EXPIRES | VALUE |
|                   |         |       |

The screenshot shows the 'Certificates & secrets' blade in the Azure portal. On the left, the 'Certificates & secrets' option is selected under the 'Manage' section. A modal window is open to add a new client secret, with the 'Description' field set to 'Forever' and the 'Expires' dropdown set to 'Never'. Below the modal, it says 'No certificates have been added for this application.' In the 'Client secrets' section, there is a note about what a client secret is and a button to 'New client secret'. The table below is empty, stating 'No client secrets have been created for this application.'

10. Copy the client secret value before you leave this page. You will need it later in the tutorial.

**Important:** This client secret is never shown again, so make sure you copy it now.

This screenshot shows the 'Client secrets' blade. It contains a single row for a secret with the description 'Forever', an expiration date of '12/31/2299', and a value starting with 'v7NSTLG5Cb[t|no\_S|]l|p\$'. There is a trash icon next to the value. At the top of the blade is a 'New client secret' button.

11. Select **Manage > API Permissions**.

12. Select **Add a permission** and select **Microsoft Graph**.

This screenshot shows the 'Request API permissions' blade. Under 'Select an API', 'Microsoft APIs' is selected. In the 'Commonly used Microsoft APIs' section, the 'Microsoft Graph' item is highlighted with a red box. It describes Microsoft Graph as a way to access Office 365, Enterprise Mobility + Security, and Windows 10. Below it are other options like 'Azure Data Catalog', 'Azure Data Lake', and 'Azure Rights Management Services'.

13. Select **Application Permission**, expand the **User** group, and select **User.Read.All** scope.

14. Select **Add permissions** to save your changes.

Screenshot of the Microsoft Azure portal showing the 'Request API permissions' page for the 'GraphNotificationTutorial' application.

The left sidebar shows the application's navigation menu with 'API permissions' selected. The main area displays the 'Request API permissions' interface.

**Delegated permissions**: Your application needs to access the API as the signed-in user.

**Application permissions**: Your application runs as a background service or daemon without a signed-in user. (This section is highlighted with a blue border.)

**Select permissions**: User (1 permission selected)

| PERMISSION           | ADMIN CONSENT REQUIRED |
|----------------------|------------------------|
| User.Export.All      | Yes                    |
| User.Invite.All      | Yes                    |
| <b>User.Read.All</b> | Yes                    |
| User.ReadWrite.All   | Yes                    |

**Add permissions** | **Discard**

#### Configured permissions

Applications are authorized to call APIs when they are granted permissions by users/admins as part of the consent process. The list of configured permissions should include all the permissions the application needs. [Learn more about permissions and consent](#)

| + Add a permission           | Grant admin consent for |                               |                      |   |     |
|------------------------------|-------------------------|-------------------------------|----------------------|---|-----|
| API / Permissions name       | Type                    | Description                   | Admin consent req... | Status  | ... |
| <b>▼ Microsoft Graph (2)</b> |                         |                               |                      |   |     |
| User.Read                    | Delegated               | Sign in and read user profile | -                    | ...   | ... |
| <b>User.Read.All</b>         | Application             | Read all users' full profiles | Yes                  | <span style="color: red;">⚠ Not granted for [tenant]</span> | ... |

15. The application requests an application permission with the **User.Read.All** scope. This permission requires administrative consent.

16. Select **Grant admin consent for Contoso**, then select **Yes** to consent this application, and grant the application access to your tenant using the scopes you specified.

#### Configured permissions

Applications are authorized to call APIs when they are granted permissions by users/admins as part of the consent process. The list of configured permissions should include all the permissions the application needs. [Learn more about permissions and consent](#)

| + Add a permission           | Grant admin consent for |                               |                      |   |     |
|------------------------------|-------------------------|-------------------------------|----------------------|---|-----|
| API / Permissions name       | Type                    | Description                   | Admin consent req... | Status  | ... |
| <b>▼ Microsoft Graph (2)</b> |                         |                               |                      |   |     |
| User.Read                    | Delegated               | Sign in and read user profile | -                    | <span style="color: green;">✓ Granted for [tenant]</span> | ... |
| <b>User.Read.All</b>         | Application             | Read all users' full profiles | Yes                  | <span style="color: green;">✓ Granted for [tenant]</span> | ... |

## Task 2: Create .NET Core App

### Run ngrok

In order for the Microsoft Graph to send notifications to your application running on your development machine you need to use a tool such as ngrok to tunnel calls from the internet to your development machine. Ngrok allows calls from the internet to be directed to your application running locally without needing to create firewall rules.

**NOTE:** Graph requires using https and this lab uses ngrok free.

If you run into any issues please visit [Using ngrok to get a public HTTPS address for a local server already serving HTTPS \(for free\)](#).

1. Run ngrok by executing the following from the command line:

```
powershell ngrok http 5000
```

2. This will start ngrok and will tunnel requests from an external ngrok url to your development machine on port 5000. Copy the https forwarding address. In the example below that would be https://787b8292.ngrok.io. You will need this later.

### Create .NET Core WebApi App

1. Open your command prompt, navigate to a directory where you have rights to create your project, and run the following command to create a new .NET Core console application:  
`dotnet new webapi -o msgraphapp`

2. After creating the application, run the following commands to ensure your new project runs correctly:

```
powershell cd msgraphapp dotnet add package Microsoft.Identity.Client  
dotnet add package Microsoft.Graph dotnet run
```

3. The application will start and output the following:

```
powershell info: Microsoft.Hosting.Lifetime[0] Now listening on:  
https://localhost:5001 info: Microsoft.Hosting.Lifetime[0] Now listening  
on: http://localhost:5000 info: Microsoft.Hosting.Lifetime[0] Application  
started. Press Ctrl+C to shut down. info: Microsoft.Hosting.Lifetime[0]  
Hosting environment: Development info: Microsoft.Hosting.Lifetime[0]  
Content root path: [your file path]\msgraphapp
```

4. Stop the application running by pressing **CTRL+C**.

5. Open the application in Visual Studio Code using the following command: `code .`

6. If Visual Studio code displays a dialog box asking if you want to add required assets to the project, select Yes.

## Task 3: Code the HTTP API

Open the **Startup.cs** file and comment out the following line to disable ssl redirection.

```
csharp //app.UseHttpsRedirection();
```

### Add model classes

The application uses several new model classes for (de)serialization of messages to/from the Microsoft Graph.

1. Right-click in the project file tree and select New Folder. Name it **Models**

2. Right-click the Models folder and add three new files:

- **Notification.cs**
- **ResourceData.cs**
- **MyConfig.cs**

3. Replace the contents of **Notification.cs** with the following:

```
csharp using Newtonsoft.Json; using System; namespace msgraphapp.Models {  
    public class Notifications { [JsonProperty(PropertyName = "value")] public  
        Notification[] Items { get; set; } } // A change notification.  
    public class Notification { // The type of change. [JsonProperty(PropertyName =  
        "changeType")] public string ChangeType { get; set; } // The client state  
        used to verify that the notification is from Microsoft Graph. Compare the  
        value received with the notification to the value you sent with the  
        subscription request. [JsonProperty(PropertyName = "clientState")] public  
        string ClientState { get; set; } // The endpoint of the resource that  
        changed. For example, a message uses the format ..//Users/{user-  
        id}/Messages/{message-id} [JsonProperty(PropertyName = "resource")] public  
        string Resource { get; set; } // The UTC date and time when the webhooks  
        subscription expires. [JsonProperty(PropertyName =  
            "subscriptionExpirationDateTime")] public DateTimeOffset  
        SubscriptionExpirationDateTime { get; set; } // The unique identifier for  
        the webhooks subscription. [JsonProperty(PropertyName = "subscriptionId")]  
        public string SubscriptionId { get; set; } // Properties of the changed  
        resource. [JsonProperty(PropertyName = "resourceData")] public ResourceData  
        ResourceData { get; set; } }
```

4. Replace the contents of **ResourceData.cs** with the following:

```
csharp using Newtonsoft.Json; namespace msgraphapp.Models { public class  
    ResourceData { // The ID of the resource. [JsonProperty(PropertyName =  
        "id")] public string Id { get; set; } // The OData etag property.  
        [JsonProperty(PropertyName = "@odata.etag")] public string ODataEtag { get;  
        set; } // The OData ID of the resource. This is the same value as the  
        resource property. [JsonProperty(PropertyName = "@odata.id")] public string  
        ODataId { get; set; } // The OData type of the resource:  
        "#Microsoft.Graph.Message", "#Microsoft.Graph.Event", or  
        "#Microsoft.Graph.Contact". [JsonProperty(PropertyName = "@odata.type")]  
        public string ODataType { get; set; } }
```

5. Replace the contents of **MyConfig.cs** with the following:

```
csharp namespace msgraphapp { public class MyConfig { public string AppId { get; set; } public string AppSecret { get; set; } public string TenantId { get; set; } public string Ngrok { get; set; } } }
```

6. Open the **Startup.cs** file. Locate the method **ConfigureServices()** method and replace it with the following code:

```
csharp public void ConfigureServices(IServiceCollection services) {
    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_3_0);
    var config = new MyConfig(); Configuration.Bind("MyConfig", config);
    services.AddSingleton(config); }
```

7. Open the **appsettings.json** file and replace the content with the following JSON.

```
csharp { "Logging": { "LogLevel": { "Default": "Warning" } }, "MyConfig": {
    "AppId": "<APP ID>", "AppSecret": "<APP SECRET>", "TenantId": "<TENANT
    ID>", "Ngrok": "<NGROK URL>" } }
```

8. Replace the following variables with the values you copied earlier:

- should be set to the **https ngrok url** you copied earlier.
- should be your **Office 365 tenant id** you copied earlier.
- and should be the **application id** and **secret** you copied earlier when you created the application registration.

## Add notification controller

The application requires a new controller to process the subscription and notification. 1. Right-click the **Controllers** folder, select **New File**, and name the controller **NotificationsController.cs**.

1. Replace the contents of **NotificationController.cs** with the following code:

```
csharp using System; using System.Collections.Generic; using System.IO;
using System.Linq; using System.Net.Http; using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc; using msgraphapp.Models; using
Newtonsoft.Json; using System.Net; using System.Threading; using
Microsoft.Graph; using Microsoft.Identity.Client; using
System.Net.Http.Headers; namespace msgraphapp.Controllers {
    [Route("api/[controller]")] [ApiController] public class
    NotificationsController : ControllerBase { private readonly MyConfig
    config; public NotificationsController(MyConfig config) { this.config =
    config; } [HttpGet] public async Task<ActionResult<string>> Get() { var
    graphServiceClient = GetGraphClient(); var sub = new
    Microsoft.Graph.Subscription(); sub.ChangeType = "updated";
    sub.NotificationUrl = config.Ngrok + "/api/notifications"; sub.Resource =
    "/users"; sub.ExpirationDateTime = DateTime.UtcNow.AddMinutes(5);
    sub.ClientState = "SecretClientState"; var newSubscription = await
    graphServiceClient .Subscriptions .Request() .AddAsync(sub); return
    $"Subscribed. Id: {newSubscription.Id}, Expiration:
    {newSubscription.ExpirationDateTime}"; } public async
    Task<ActionResult<string>> Post([FromQuery]string validationToken = null) {
    // handle validation if(!string.IsNullOrEmpty(validationToken)) {
    Console.WriteLine($"Received Token: '{validationToken}'"); return
    Ok(validationToken); } // handle notifications using (StreamReader reader =
    new StreamReader(Request.Body)) { string content = await
    reader.ReadToEndAsync(); Console.WriteLine(content); var notifications =
    JsonConvert.DeserializeObject<Notifications>(content); foreach(var
    notification in notifications.Items) { Console.WriteLine($"Received
    notification: '{notification.Resource}', {notification.ResourceData?.Id}"); }}
```

```
    } } return Ok(); } private GraphServiceClient GetGraphClient() { var  
graphClient = new GraphServiceClient(new  
DelegateAuthenticationProvider((requestMessage) => { // get an access token  
for Graph var accessToken = GetAccessToken().Result; requestMessage  
.Headers .Authorization = new AuthenticationHeaderValue("bearer",  
accessToken); return Task.FromResult(0); })); return graphClient; } private  
async Task<string> GetAccessToken() { IConfidentialClientApplication app =  
ConfidentialClientApplicationBuilder.Create(config.AppId)  
.WithClientSecret(config.AppSecret)  
.WithAuthority($"https://login.microsoftonline.com/{config.TenantId}")  
.WithRedirectUri("https://daemon") .Build(); string[] scopes = new string[]  
{ "https://graph.microsoft.com/.default" }; var result = await  
app.AcquireTokenForClient(scopes).ExecuteAsync(); return  
result.AccessToken; } } }
```

**2. Save all files.**

## Task 4: Run the application

Update the Visual Studio debugger launch configuration:

**Note:** By default, the .NET Core launch configuration will open a browser and navigate to the default URL for the application when launching the debugger. For this application, we instead want to navigate to the NGrok URL. If you leave the launch configuration as is, each time you debug the application it will display a broken page. You can just change the URL, or change the launch configuration to not launch the browser:

1. In Visual Studio Code, open the file **.vscode/launch.json**.

2. Delete the following section in the default configuration:

```
json // Enable launching a web browser when ASP.NET Core starts. For more
information: https://aka.ms/VSCode-CS-LaunchJson-WebBrowser
"serverReadyAction": { "action": "openExternally", "pattern": "^\s*Now
listening on:\s+(https?://\S+)" },
```

3. **Save** your changes.

4. In **Visual Studio Code**, select **Debug > Start debugging** to run the application. VS Code will build and start the application.

5. Once you see messages in the Debug Console window that states the application started proceed with the next steps.

6. Open a browser and navigate to **http://localhost:5000/api/notifications** to subscribe to change notifications. If successful you will see output that includes a subscription id.

**Note:** Your application is now subscribed to receive notifications from the Microsoft Graph when an update is made on any user in the Office 365 tenant.

7. Trigger a notification:

1. Open a browser and navigate to the **Microsoft 365 admin center** (<https://admin.microsoft.com/AdminPortal>).

2. If prompted to login, sign-in using an admin account.

3. Select **Users > Active users**.

8. Select an active user and select **Edit** for their **Contact information**.

9. Update the **Phone number** value with a new number and select **Save**.

10. In the **Visual Studio Code Debug Console**, you will see a notification has been received. Sometimes this may take a few minutes to arrive. An example of the output is below:

```
json Received notification: 'Users/7a7fded6-0269-42c2-a0be-512d58da4463',
7a7fded6-0269-42c2-a0be-512d58da4463
```

This indicates the application successfully received the notification from the Microsoft Graph for the user specified in the output. You can then use this information to query the Microsoft Graph for the users full details if you want to synchronize their details into your application.

## Task 5: Manage notification subscriptions

Subscriptions for notifications expire and need to be renewed periodically. The following steps will demonstrate how to renew notifications:

### Update Code

1. Open **Controllers > NotificationsController.cs** file

2. Add the following two member declarations to the **NotificationsController** class:

```
csharp private static Dictionary<string, Subscription> Subscriptions = new  
Dictionary<string, Subscription>(); private static Timer subscriptionTimer  
= null;
```

3. Add the following new methods. These will implement a background timer that will run every 15 seconds to check if subscriptions have expired. If they have, they will be renewed.

```
csharp private void CheckSubscriptions(Object stateInfo) { AutoResetEvent  
autoEvent = (AutoResetEvent)stateInfo; Console.WriteLine($"Checking  
subscriptions {DateTime.Now.ToString("h:mm:ss.fff")});  
Console.WriteLine($"Current subscription count {Subscriptions.Count()});  
foreach(var subscription in Subscriptions) { // if the subscription expires  
in the next 2 min, renew it if(subscription.Value.ExpirationDateTime <  
DateTime.UtcNow.AddMinutes(2)) { RenewSubscription(subscription.Value); } }  
} private async void RenewSubscription(Subscription subscription) {  
Console.WriteLine($"Current subscription: {subscription.Id}, Expiration:  
{subscription.ExpirationDateTime}"); var graphServiceClient =  
GetGraphClient(); var newSubscription = new Subscription {  
ExpirationDateTime = DateTime.UtcNow.AddMinutes(5) }; await  
graphServiceClient .Subscriptions[subscription.Id] .Request()  
.UpdateAsync(newSubscription); subscription.ExpirationDateTime =  
newSubscription.ExpirationDateTime; Console.WriteLine($"Renewed  
subscription: {subscription.Id}, New Expiration:  
{subscription.ExpirationDateTime}"); }
```

4. The **CheckSubscriptions** method is called every 15 seconds by the timer. For production use this should be set to a more reasonable value to reduce the number of unnecessary calls to Microsoft Graph. The **RenewSubscription** method renews a subscription and is only called if a subscription is going to expire in the next two minutes.

5. Locate the method **Get()** and replace it with the following code:

```
csharp [HttpGet] public async Task<ActionResult<string>> Get() { var  
graphServiceClient = GetGraphClient(); var sub = new  
Microsoft.Graph.Subscription(); sub.ChangeType = "updated";  
sub.NotificationUrl = config.Ngrok + "/api/notifications"; sub.Resource =  
"/users"; sub.ExpirationDateTime = DateTime.UtcNow.AddMinutes(5);  
sub.ClientState = "SecretClientState"; var newSubscription = await  
graphServiceClient .Subscriptions .Request() .AddAsync(sub);  
Subscriptions[newSubscription.Id] = newSubscription; if (subscriptionTimer  
== null) { subscriptionTimer = new Timer(CheckSubscriptions, null, 5000,  
15000); } return $"Subscribed. Id: {newSubscription.Id}, Expiration:  
{newSubscription.ExpirationDateTime}"; }
```

### Test the changes

1. Within Visual Studio Code, select Debug > Start debugging to run the application.  
Navigate to the following url: **http://localhost:5000/api/notifications**. This will register a new subscription.
2. In the Visual Studio Code Debug Console window, approximately every 15 seconds, notice the timer checking the subscription for expiration:

```
powershell Checking subscriptions 12:32:51.882 Current subscription count 1
```

3. Wait a few minutes and you will see the following when the subscription needs renewing:

```
powershell Renewed subscription: 07ca62cd-1a1b-453c-be7b-4d196b3c6b5b, New  
Expiration: 3/10/2019 7:43:22 PM +00:00
```

This indicates that the subscription was renewed and shows the new expiry time.

## Task 5: Query for changes

Microsoft Graph offers the ability to query for changes to a particular resource since you last called it. Using this option, combined with Change Notifications, enables a robust pattern for ensuring you don't miss any changes to the resources. 1. Locate and open the following controller: **Controllers > NotificationsController.cs**. Add the following code to the existing **NotificationsController** class.

```
```csharp
private static object DeltaLink = null;
private static IUserDeltaCollectionPage lastPage = null;
private async Task CheckForUpdates()
{
    var graphClient = GetGraphClient();
    // get a page of users
    var users = await GetUsers(graphClient, DeltaLink);
    OutputUsers(users);
    // go through all of the pages so that we can get the delta link on the
    last page.
    while (users.NextPageRequest != null)
    {
        users = users.NextPageRequest.GetAsync().Result;
        OutputUsers(users);
    }
    object deltaLink;
    if (users.AdditionalData.TryGetValue("@odata.deltaLink", out deltaLink))
    {
        DeltaLink = deltaLink;
    }
}
private void OutputUsers(IUserDeltaCollectionPage users)
{
    foreach(var user in users)
    {
        var message = $"User: {user.Id}, {user.GivenName} {user.Surname}";
        Console.WriteLine(message);
    }
}
private async Task<IUserDeltaCollectionPage> GetUsers(GraphServiceClient
graphClient, object deltaLink)
{
    IUserDeltaCollectionPage page;
    if (lastPage == null)
    {
        page = await graphClient
            .Users
            .Delta()
            .Request()
            .GetAsync();
    }
    else
    {
        lastPage.InitializeNextPageRequest(graphClient, deltaLink.ToString());
        page = await lastPage.NextPageRequest.GetAsync();
    }
    lastPage = page;
    return page;
}
```

```

1. This code includes a new method, **CheckForUpdates()**, that will call the Microsoft Graph using the delta url and then pages through the results until it finds a new **deltalink** on the

final page of results. It stores the url in memory until the code is notified again when another notification is triggered.

2. Locate the existing **Post()** method and replace it with the following code:

```
csharp public async Task<ActionResult<string>> Post([FromQuery]string validationToken = null) { // handle validation if (!string.IsNullOrEmpty(validationToken)) { Console.WriteLine($"Received Token: '{validationToken}'"); return Ok(validationToken); } // handle notifications using (StreamReader reader = new StreamReader(Request.Body)) { string content = await reader.ReadToEndAsync(); Console.WriteLine(content); var notifications = JsonConvert.DeserializeObject<Notifications>(content); foreach (var notification in notifications.Items) { Console.WriteLine($"Received notification: '{notification.Resource}', {notification.ResourceData?.Id}"); } } // use deltaquery to query for all updates await CheckForUpdates(); return Ok(); }
```

3. The Post method will now call **CheckForUpdates** when a notification is received. **Save** all files.

## Test the changes

1. Within **Visual Studio Code**, select **Debug > Start debugging** to run the application. Navigate to the following url: <http://localhost:5000/api/notifications>. This will register a new subscription.
2. Open a browser and navigate to the **Microsoft 365 admin center** (<https://admin.microsoft.com/AdminPortal>).
3. If prompted to login, sign-in using an admin account.
  1. Select **Users > Active users**.
  2. Select an active **user** and select **Edit** for their **Contact information**.
  3. Update the **Mobile phone** value with a new number and select **Save**.

4. Wait for the notification to be received as indicated in the Visual Studio Code Debug Console:

```
powershell Received notification: 'Users/7a7fded6-0269-42c2-a0be-512d58da4463', 7a7fded6-0269-42c2-a0be-512d58da4463
```

5. The application will now initiate a delta query with the graph to get all the users and log out some of their details to the console output.

```
powershell User: 19e429d2-541a-4e0b-9873-6dff9f48fabe, Allan Deyoung User: 05501e79-f527-4913-aabf-e535646d7ffa, Christie Cline User: fecac4be-76e7-48ec-99df-df745854aa9c, Debra Berger User: 4095c5c4-b960-43b9-ba53-ef806d169f3e, Diego Siciliani User: b1246157-482f-420c-992c-fc26cbff74a5, Emily Braun User: c2b510b7-1f76-4f75-a9c1-b3176b68d7ca, Enrico Cattaneo User: 6ec9bd4b-fc6a-4653-a291-70d3809f2610, Grady Archie User: b6924afe-cb7f-45a3-a904-c9d5d56e06ea, Henrietta Mueller User: 0ee8d076-4f13-4e1a-a961-eac2b29c0ef6, Irvin Sayers User: 31f66f05-ac9b-4723-9b5d-8f381f5a6e25, Isaiah Langer User: 7ee95e20-247d-43ef-b368-d19d96550c81, Johanna Lorenz User: b2fa93ac-19a0-499b-b1b6-afa76c44a301, Joni Sherman User: 01db13c5-74fc-470a-8e45-d6d736f8a35b, Jordan Miller User: fb0b8363-4126-4c34-8185-c998ff697a60, Lee Gu User: ee75e249-a4c1-487b-a03a-5a170c2aa33f, Lidia Holloway User: 5449bd61-cc63-40b9-b0a8-e83720eeefba, Lynne Robbins User:
```

```
7ce295c3-25fa-4d79-8122-9a87d15e2438, Miriam Graham User: 737fe0a7-0b67-  
47dc-b7a6-9cf07870705, Nestor Wilke User: a1572b58-35cd-41a0-804a-  
732bd978df3e, Patti Fernandez User: 7275e1c4-5698-446c-8d1d-fa8b0503c78a,  
Pradeep Gupta User: 96ab25eb-6b69-4481-9d28-7b01cf367170, Megan Bowen User:  
846327fa-e6d6-4a82-89ad-5fd313bff0cc, Alex Wilber User: 200e4c7a-b778-436c-  
8690-7a6398e5fe6e, MOD Administrator User: 7a7fded6-0269-42c2-a0be-  
512d58da4463, Adele Vance User: 752f0102-90f2-4b8d-ae98-79dee995e35e,  
Removed?:deleted User: 4887248a-6b48-4ba5-bdd5-fed89d8ea6a0,  
Removed?:deleted User: e538b2d5-6481-4a90-a20a-21ad55ce4c1d,  
Removed?:deleted User: bc5994d9-4404-4a14-8fb0-46b8dccca0ad,  
Removed?:deleted User: d4e3a3e0-72e9-41a6-9538-c23e10a16122,  
Removed?:deleted
```

## 6. In the Microsoft 365 Admin Portal, repeat the process of editing a user and Save again.

The application will receive another notification and will query the graph again using the last delta link it received. However, this time you will notice that only the modified user was returned in the results. powershell User: 7a7fded6-0269-42c2-a0be-512d58da4463, Adele Vance

Using this combination of notifications with delta query you can be assured you won't miss any updates to a resource. Notifications may be missed due to transient connection issues, however the next time your application gets a notification it will pick up all the changes since the last successful query.

## Review

You've now completed this exercise. In this exercise, you created a .NET Core app that used the Microsoft Graph API to receive notifications (webhooks) when a user account changes in Azure AD and perform queries using the delta query API to receive all changes to user accounts since the last query was made.

### ❖❖❖# Exercise 4: Reduce traffic with batched requests

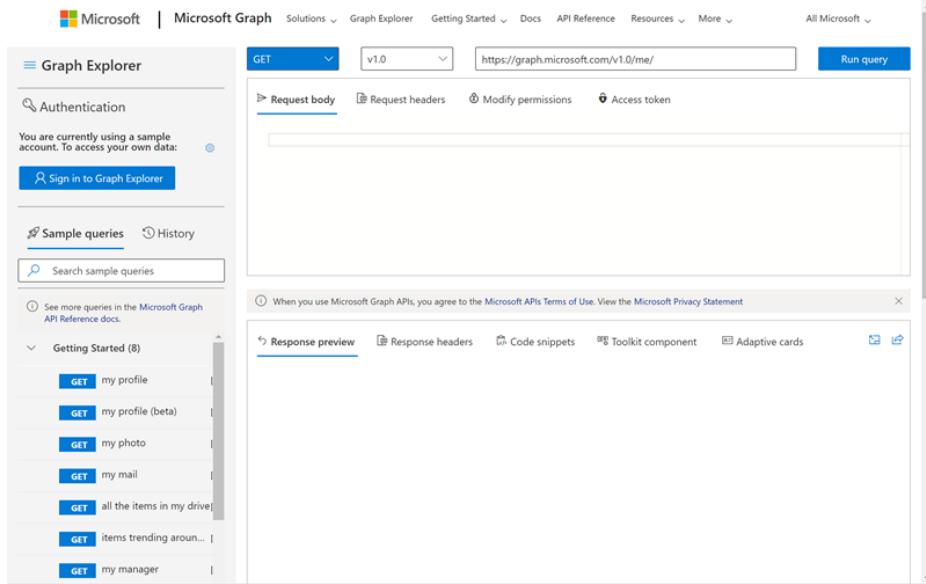
In this exercise, you'll use the Graph Explorer to create and issue a single request that contains multiple child requests. This batching of requests enables developers to submit multiple requests in a single round-trip request to Microsoft Graph, creating more optimized queries.

# Task 1: Sign in to Microsoft Graph Explorer

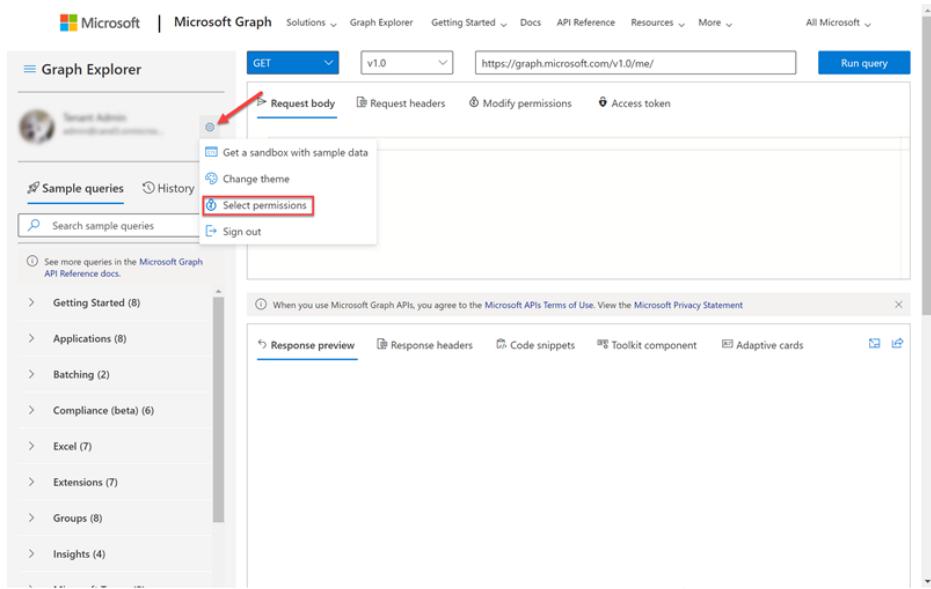
The tool Graph Explorer enables developers to create and test queries using the Microsoft Graph REST API. Previously in this module, you used the Graph Explorer as an anonymous user and executed queries using the sample data collection.

In this example, you'll sign in to Microsoft Graph with a real user.

1. Open a browser and navigate to <https://developer.microsoft.com/graph/graph-explorer>



2. Select the **Sign in to Graph Explorer** button in the leftmost panel and enter the credentials of a Work and School account.
3. After signing in, click **select permissions** and verify that the user has the permissions to submit the requests in this exercise. You must have at least these minimum permissions:
  - **Mail.Read**
  - **Calendars.Read**
  - **Files.ReadWrite**



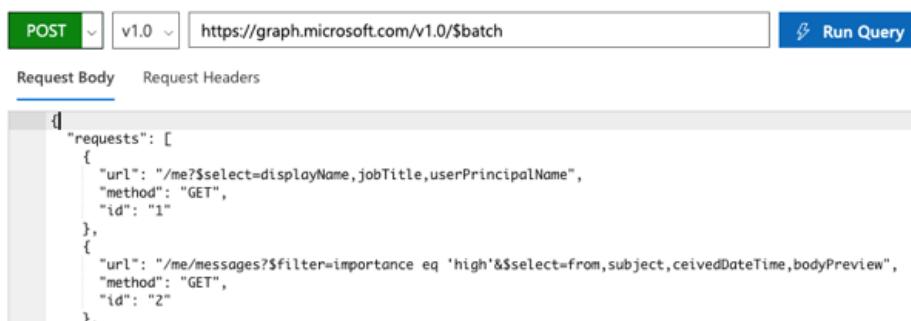
## Task 2: Submit three (3) GET requests in a single batch

All batch requests are submitted as HTTP POSTs to a specific endpoint:

[https://graph.microsoft.com/v1.0/\\$batch](https://graph.microsoft.com/v1.0/$batch). The **\$batch** query parameter is what tells Microsoft Graph to unpack the requests submitted in the body.

1. Set the request to an HTTP **POST** and the endpoint of the request to [https://graph.microsoft.com/v1.0/\\$batch](https://graph.microsoft.com/v1.0/$batch).
2. Add the following JSON code to the **Request Body** input box. This JSON code will issue three requests:
  - Request the current user's **displayName**, **jobTitle**, and **userPrincipalName** properties.
  - Request the current user's email messages that are marked with high importance.
  - Request all the current user's calendar events.

```
json { "requests": [ { "url": "/me?$select=displayName,jobTitle,userPrincipalName", "method": "GET", "id": "1" }, { "url": "/me/messages?$filter=importance eq 'high'&$select=from,subject,receivedDateTime,bodyPreview", "method": "GET", "id": "2" }, { "url": "/me/events", "method": "GET", "id": "3" } ] }
```



The screenshot shows the Microsoft Graph Explorer interface. At the top, there are dropdown menus for 'POST' and 'v1.0', and a URL input field containing 'https://graph.microsoft.com/v1.0/\$batch'. To the right is a blue 'Run Query' button. Below the URL field, there are tabs for 'Request Body' and 'Request Headers', with 'Request Body' being active. The 'Request Body' text area contains the JSON code provided in the previous step. The JSON defines three requests: one for the user's profile information, one for high-importance messages, and one for the user's calendar events.

3. Select the **Run Query** button.
4. Observe the results in the **Response Preview** box at the bottom of the page.

Success - Status Code 200, 391ms

Response Preview Response Headers

```
{ "responses": [ { }, { }, { "id": "3", "status": 200, "headers": { "Cache-Control": "private", "OData-Version": "4.0", "Content-Type": "application/json;odata.metadata=minimal;odata.streaming=true;IEEE754Compatibility=true" }, "body": { "@odata.context": "https://graph.microsoft.com/v1.0/$metadata#users('bbc8446f-f788-4527-8cff-000000000000')/events", "@odata.nextLink": "https://graph.microsoft.com/v1.0/me/events?$skip=10", "value": [ { } ] } } ] }
```

Notice the response includes three individual responses within the responses collection. Also notice for response id:3, the data that was returned, as indicated by the @odata.nextLink property, is from the **/me/events** collection. This query matches the third request in the initial request submitted.

## Task 3: Combine POST and GET requests in a single batch request

Batch requests can also include both **POST** and **GET** requests.

In this example, you'll submit a request that creates a new folder in the current user's OneDrive [for Business] and then requests the newly created folder. If the first request failed, the second request should come back empty as well.

1. Enter the following JSON code to the **Request Body** input box. This will issue three requests:

```
json { "requests": [ { "url": "/me/drive/root/children", "method": "POST", "id": "1", "body": { "name": "TestBatchingFolder", "folder": {} }, "headers": { "Content-Type": "application/json" } }, { "url": "/me/drive/root/children/TestBatchingFolder", "method": "GET", "id": "2", "DependsOn": [ "1" ] } ] }
```

2. Select the **Run Query** button.
3. Observe the results in the **Response Preview** box at the bottom of the page. Notice that this response contains two objects. The first request resulted in an HTTP 201 message that says the item, or folder, was created. The second request was also successful, and the name of the folder returned matched the folder the first request created.

Response Preview   Response Headers

```
{ "responses": [ { "id": "1", "status": 201, "headers": {}, "body": {} }, { "id": "2", "status": 200, "headers": {}, "body": { "@odata.context": "https://graph.microsoft.com/v1.0/$metadata#users('bbc8446f-f788-4527-8cff-012n5kf220plt3lt2a02gjukgbujiokm46')/children", "createdDateTime": "2019-08-28T19:31:00Z", "eTag": "\'{B5E77A4E-40CF-4C76-9A28-C1A25105339E},1\"", "id": "012n5kf220plt3lt2a02gjukgbujiokm46", "lastModifiedDateTime": "2019-08-28T19:31:00Z", "name": "TestBatchingFolder", "webUrl": "https://m365x460234-my.sharepoint.com/personal/meganb_m365x460234_onmicrosoft_com/cTag": "\'{c:{B5E77A4E-40CF-4C76-9A28-C1A25105339E},0\""}, }}
```

## Review

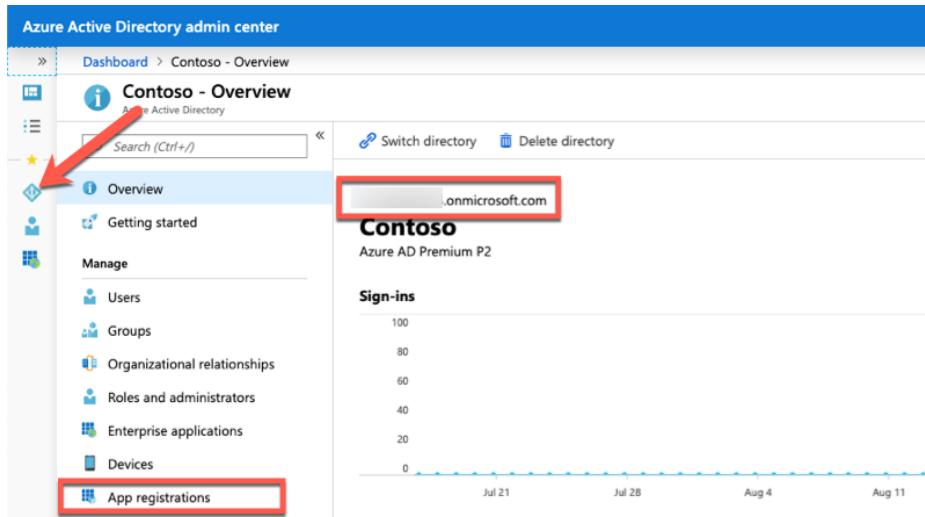
In this exercise, you used Microsoft Graph to demonstrate how you can combine multiple requests using a single request. This capability of submitting batch requests using the \$batch query parameter enables you to optimize your applications to minimize the number of requests to Microsoft Graph.

### ❖❖❖# Exercise 5: Understand throttling in Microsoft Graph

In this exercise, you will create a new Azure AD web application registration using the Azure AD admin center, a .NET Core console application, and query Microsoft Graph. You will issue many requests in parallel to trigger your requests to be throttled. This application will allow you to see the response you will receive.

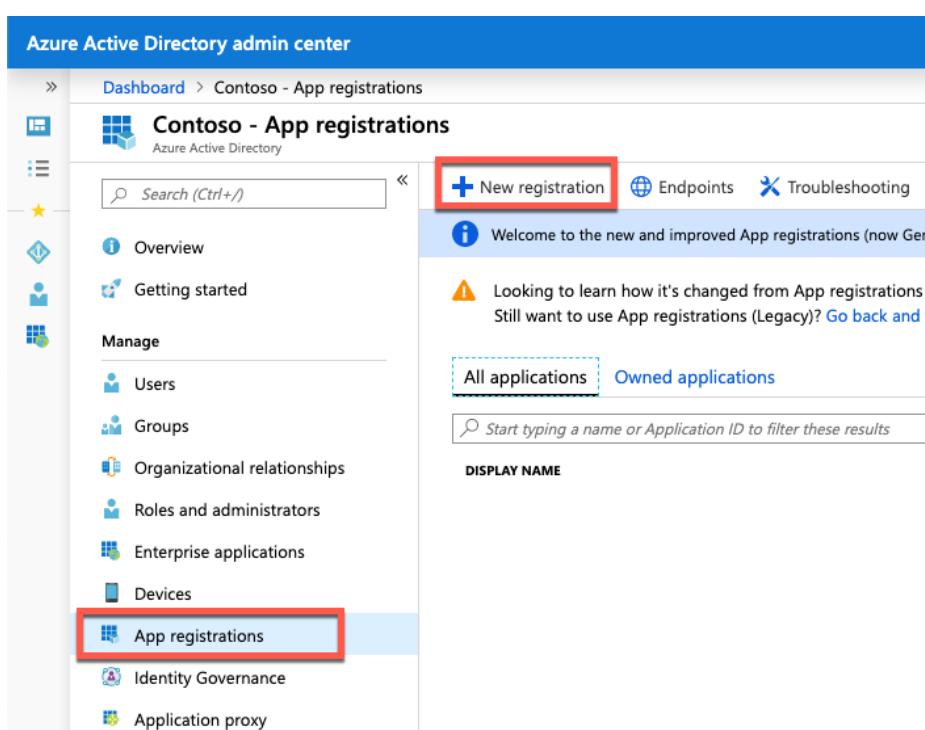
## Task 1: Create an Azure AD application

1. Open a browser and navigate to the [Azure Active Directory admin center](https://aad.portal.azure.com) (<https://aad.portal.azure.com>). Sign in using a **Work or School Account** that has global administrator rights to the tenancy.
2. Select **Azure Active Directory** in the leftmost navigation panel.



The screenshot shows the Azure Active Directory admin center interface. The left sidebar has a 'Manage' section with links for Users, Groups, Organizational relationships, Roles and administrators, Enterprise applications, and Devices. Below these, 'App registrations' is highlighted with a red box. The top navigation bar also has a 'App registrations' link highlighted with a red box. The main content area displays 'Contoso - Overview' for 'Azure AD Premium P2'. It includes a 'Sign-ins' chart showing activity from July 21 to Aug 11, with values ranging from 0 to 100.

3. Select **Manage > App registrations** in the left navigation panel.
4. On the **App registrations** page, select **New registration**.



The screenshot shows the 'Contoso - App registrations' page. The left sidebar has a 'Manage' section with links for Users, Groups, Organizational relationships, Roles and administrators, Enterprise applications, and Devices. Below these, 'App registrations' is highlighted with a red box. The top navigation bar has a 'New registration' button highlighted with a red box. The main content area includes a welcome message, a note about legacy app registrations, and tabs for 'All applications' (which is selected) and 'Owned applications'. There is also a search bar and a 'DISPLAY NAME' input field.

5. On the **Register an application** page, set the values as follows:
  - **Name:** Graph Console Throttle App.

- **Supported account types:** Accounts in this organizational directory only (Contoso only - Single tenant).

\* Name

The user-facing display name for this application (this can be changed later).

Graph Console Throttle App ✓

#### Supported account types

Who can use this application or access this API?

- Accounts in this organizational directory only (Contoso only - Single tenant)
- Accounts in any organizational directory (Any Azure AD directory - Multitenant)
- Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)
- Personal Microsoft accounts only

## 6. Select Register.

7. On the **Graph Console App** page, copy the value of the **Application (client) ID** and **Directory (tenant) ID**; you will need these in the application.

The screenshot shows the 'Graph Console App' registration page in the Azure portal. The left sidebar has sections like Overview, Quickstart, Manage, and Support + Troubleshooting. The main area shows the app's details: Display name 'Graph Console App', Supported account types 'My organization only', Application (client) ID '9ef0bc71-6919-439e-b027-294438560213', and Directory (tenant) ID '7adf0e7f-07b3-4eb5-ba39-7ea421035371'. Below these, there are sections for Call APIs (with icons for various Microsoft services) and Documentation (links to Microsoft identity platform, authentication scenarios, etc.). A 'View API Permissions' button is also visible.

8. Select **Manage > Authentication**.

9. Under **Platform configurations**, select **Add a platform**.

10. In Configure platforms, select **Mobile and desktop applications**.

## Configure platforms

X

### Web applications



#### Web

Build, host, and deploy a web server application. .NET, Java, Python



#### Single-page application

Configure browser client applications and progressive web applications. Javascript.

### Mobile and desktop applications



#### iOS / macOS

Objective-C, Swift, Xamarin



#### Android

Java, Kotlin, Xamarin



#### Mobile and desktop applications

Windows, UWP, Console, IoT & Limited-entry Devices, Classic iOS + Android

11. In the section **Redirect URIs**, select the entry that begins with **msal** and enter **https://contoso** in **Custom redirect URIs** section, and then click **Configure** button.

## Configure Desktop + devices

< All platforms

Quickstart Docs

### Redirect URIs

The URIs we will accept as destinations when returning authentication responses (tokens) after successfully authenticating users. Also referred to as reply URLs. [Learn more about Redirect URIs and their restrictions](#)

- https://login.microsoftonline.com/common/oauth2/nativeclient 
- https://login.live.com/oauth20\_desktop.srf (LiveSDK) 
- msalf...://auth (MSAL only) 

### Custom redirect URIs

https://contoso



Configure

Cancel

12. Scroll down to the **Advanced settings** section and set the toggle to **Yes**.

### Advanced settings

#### Allow public client flows

Enable the following mobile and desktop flows:

Yes  No

- App collects plaintext password (Resource Owner Password Credential Flow) [Learn more](#)
- No keyboard (Device Code Flow) [Learn more](#)
- SSO for domain-joined Windows (Windows Integrated Auth Flow) [Learn more](#)

13. Select **Save** in the top menu to save your changes.

## Task 2: Grant Azure AD application permissions to Microsoft Graph

After creating the application, you need to grant it the necessary permissions to Microsoft Graph.

1. Select **API Permissions** in the left navigation panel.

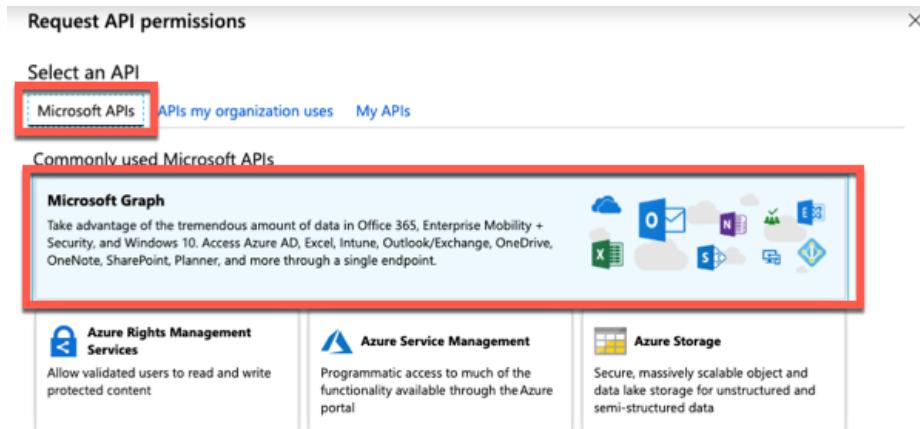
The screenshot shows the Azure Active Directory admin center interface. In the top navigation bar, the path is: Dashboard > Contoso - App registrations > Graph Console App. On the left sidebar, under the 'Graph Console App' heading, the 'API permissions' option is highlighted with a red box. The main content area displays the app's details: Display name: Graph Console App, Application (client) ID: 9ef0bc7f-6919-439e-b027-294438560213, Directory (tenant) ID: 7adff0e7f-07b3-4eb5-ba39-7ea421035371, and Object ID: 52168e8f-208c-466b-a8a6-86adff64af2f. Below this, there is a section titled 'Call APIs' with various Microsoft service icons. A callout text states: 'Build more powerful apps with rich user and business data from Microsoft services and your own company's data sources.'

2. Select the **Add a permission** button.

The screenshot shows the 'API permissions' configuration page. At the top, a note says: 'Applications are authorized to call APIs when they are granted permissions by users/admins as part of the all the permissions the application needs.' Below this is a table with a single row. The first column contains a blue button labeled '+ Add a permission' with a red box around it. The second column is 'API / PERMISSIONS NAME' with a dropdown menu showing 'Microsoft Graph (1)'. The third column is 'TYPE' with 'Delegated' selected. The fourth column is 'DESCRIPTION' with 'Sign in and read user profile' listed. A note at the bottom states: 'These are the permissions that this application requests statically. You may also request user consentable permissions dynamically through code. See best practices for requesting permissions'.

| API / PERMISSIONS NAME | TYPE      | DESCRIPTION                   |
|------------------------|-----------|-------------------------------|
| ▼ Microsoft Graph (1)  | Delegated | Sign in and read user profile |

3. In the Request API permissions panel that appears, select **Microsoft Graph** from the **Microsoft APIs** tab.



4. When prompted for the type of permission, select **Delegated permissions**.

| PERMISSION                                     | ADMIN CONSENT REQUIRED |
|--|------------------------|
| <input checked="" type="checkbox"/> Mail.R     |                        |
| <input type="checkbox"/> Mail.Read             |                        |
| <input type="checkbox"/> Mail.Read.Shared      |                        |
| <input type="checkbox"/> Mail.ReadBasic        |                        |
| <input type="checkbox"/> Mail.ReadWrite        |                        |
| <input type="checkbox"/> Mail.ReadWrite.Shared |                        |

5. Enter **Mail.R** in the **Select permissions** search box and select the **Mail.Read** permission, followed by the **Add permission** button at the bottom of the panel.

6. At the bottom of the **API Permissions** panel, select the button **Grant admin consent for [tenant]**, followed by the **Yes** button, to grant all users in your organization this permission.

The option to **Grant admin consent** here in the Azure AD admin center is pre-consenting the permissions to the users in the tenant to simplify the exercise. This approach allows the console application to use the [resource owner password credential grant](#), so the user isn't prompted to grant consent to the application that simplifies the process of obtaining an OAuth access token. You could elect to implement alternative options such as the [device code flow](#) to utilize dynamic consent as another option.

## Task 3: Create .NET Core console application

1. Open your command prompt, navigate to a directory where you have rights to create your project, and run the following command to create a new .NET Core console application:  
`dotnet new console -o graphconsolethrottlepp`
2. After creating the application, run the following commands to ensure your new project runs correctly:

```
dotnetcli cd graphconsolethrottlepp dotnet add package  
Microsoft.Identity.Client dotnet add package Microsoft.Graph dotnet add  
package Microsoft.Extensions.Configuration dotnet add package  
Microsoft.Extensions.Configuration.FileExtensions dotnet add package  
Microsoft.Extensions.Configuration.Json
```

3. Open the application in Visual Studio Code using the following command: `code .`
4. If Visual Studio Code displays a dialog box asking if you want to add required assets to the project, select **Yes**.

## Task 4: Update the console app to support Azure AD authentication

1. Create a new file named **appsettings.json** in the root of the project and add the following code to it:

```
json { "tenantId": "YOUR_TENANT_ID_HERE", "applicationId":  
"YOUR_APP_ID_HERE" }
```

2. Update properties with the following values:

- **YOUR\_TENANT\_ID\_HERE**: Azure AD directory ID
- **YOUR\_APP\_ID\_HERE**: Azure AD client ID

## Task 5: Create authentication helper classes

1. Create a new folder **Helpers** in the project.
2. Create a new file **AuthHandler.cs** in the **Helpers** folder and add the following code:

```
csharp using System.Net.Http; using System.Threading; using  
System.Threading.Tasks; using Microsoft.Graph; namespace Helpers { public  
class AuthHandler : DelegatingHandler { private IAuthenticationProvider  
_authenticationProvider; public AuthHandler(IAuthenticationProvider  
authenticationProvider, HttpMessageHandler innerHandler) { InnerHandler =  
innerHandler; _authenticationProvider = authenticationProvider; } protected  
override async Task<HttpResponseMessage> SendAsync(HttpRequestMessage  
request, CancellationToken cancellationToken) { await  
_authenticationProvider.AuthenticateRequestAsync(request); return await  
base.SendAsync(request, cancellationToken); } } }
```

3. Create a new file **MsalAuthenticationProvider.cs** in the **Helpers** folder and add the following code:

```
csharp using System.Net.Http; using System.Net.Http.Headers; using  
System.Security; using System.Threading.Tasks; using  
Microsoft.Identity.Client; using Microsoft.Graph; namespace Helpers {  
public class MsalAuthenticationProvider : IAuthenticationProvider { private  
static MsalAuthenticationProvider _singleton; private  
IPublicClientApplication _clientApplication; private string[] _scopes;  
private string _username; private SecureString _password; private string  
_userId; private MsalAuthenticationProvider(IPublicClientApplication  
clientApplication, string[] scopes, string username, SecureString password)  
{ _clientApplication = clientApplication; _scopes = scopes; _username =  
username; _password = password; _userId = null; } public static  
MsalAuthenticationProvider GetInstance(IPublicClientApplication  
clientApplication, string[] scopes, string username, SecureString password)  
{ if (_singleton == null) { _singleton = new  
MsalAuthenticationProvider(clientApplication, scopes, username, password);  
} return _singleton; } public async Task  
AuthenticateRequestAsync(HttpRequestMessage request) { var accessToken =  
await GetTokenAsync(); request.Headers.Authorization = new  
AuthenticationHeaderValue("bearer", accessToken); } public async  
Task<string> GetTokenAsync() { if (!string.IsNullOrEmpty(_userId)) { try {  
var account = await _clientApplication.GetAccountAsync(_userId); if  
(account != null) { var silentResult = await  
_clientApplication.AcquireTokenSilent(_scopes, account).ExecuteAsync();  
return silentResult.AccessToken; } } catch (MsalUiRequiredException){ } }  
var result = await  
_clientApplication.AcquireTokenByUsernamePassword(_scopes, _username,  
_password).ExecuteAsync(); _userId =  
result.Account.HomeAccountId.Identifier; return result.AccessToken; } } }
```

## Task 6: Incorporate Microsoft Graph into the console app

1. Open the **Program.cs** file and add the following `using` statements to the top of the file below `using System;` line:

```
csharp using System.Collections.Generic; using System.Net; using  
System.Net.Http; using System.Net.Http.Headers; using System.Security;  
using System.Threading.Tasks; using Microsoft.Identity.Client; using  
Microsoft.Graph; using Microsoft.Extensions.Configuration; using Helpers;
```

2. Add the following method **LoadAppSettings** to the **Program** class. The method retrieves the configuration details from the **appsettings.json** file previously created:

```
csharp private static IConfigurationRoot LoadAppSettings() { try { var  
config = new ConfigurationBuilder()  
.SetBasePath(System.IO.Directory.GetCurrentDirectory())  
.AddJsonFile("appsettings.json", false, true) .Build(); if  
(string.IsNullOrEmpty(config["applicationId"])) ||  
string.IsNullOrEmpty(config["tenantId"])) { return null; } return config; }  
catch (System.IO.FileNotFoundException) { return null; } }
```

3. Add the following method **CreateAuthorizationProvider** to the **Program** class. The method will create an instance of the clients used to call Microsoft Graph.

```
csharp private static IAuthenticationProvider  
CreateAuthorizationProvider(IConfigurationRoot config, string userName,  
SecureString userPassword) { var clientId = config["applicationId"]; var  
authority = $"https://login.microsoftonline.com/{config["tenantId"]}/v2.0";  
List<string> scopes = new List<string>(); scopes.Add("User.Read");  
scopes.Add("Mail.Read"); var cca =  
PublicClientApplicationBuilder.Create(clientId) .WithAuthority(authority)  
.Build(); return MsalAuthenticationProvider.GetInstance(cca,  
scopes.ToArray(), userName, userPassword); }
```

4. Add the following method **GetAuthenticatedHttpClient** to the **Program** class. The method creates an instance of the **HttpClient** object.

```
csharp private static HttpClient  
GetAuthenticatedHttpClient(IConfigurationRoot config, string userName,  
SecureString userPassword) { var authenticationProvider =  
CreateAuthorizationProvider(config, userName, userPassword); var httpClient  
= new HttpClient(new AuthHandler(authenticationProvider, new  
HttpClientHandler())); return httpClient; }
```

5. Add the following method **ReadPassword** to the **Program** class. The method prompts the user for their password:

```
csharp private static SecureString ReadPassword() {  
Console.WriteLine("Enter your password"); SecureString password = new  
SecureString(); while (true) { Console.ReadKey(true); if  
(c.Key == ConsoleKey.Enter) { break; } password.AppendChar(c.KeyChar);  
Console.Write("*"); } Console.WriteLine(); return password; }
```

6. Add the following method **ReadUsername** to the **Program** class. The method prompts the user for their username:

```
csharp private static string ReadUsername() { string username;  
Console.WriteLine("Enter your username"); username = Console.ReadLine();  
return username; }
```

7. Locate the **Main** method in the **Program** class. Add the following code below **Console.WriteLine("Hello World!");** to load the configuration settings from the **appsettings.json** file:

```
csharp var config = LoadAppSettings(); if (config == null) {  
    Console.WriteLine("Invalid appsettings.json file."); return; }
```

8. Add the following code to the end of the **Main** method, just after the code added in the last step. This code will obtain an authenticated instance of the **HttpClient** and submit a request for the current user's email:

```
csharp var userName = ReadUsername(); var userPassword = ReadPassword();  
var client = GetAuthenticatedHTTPClient(config, userName, userPassword);
```

9. Add the following code below **var client = GetAuthenticatedHTTPClient(config, userName, userPassword);** to issue many requests to Microsoft Graph. This code will create a collection of tasks to request a specific Microsoft Graph endpoint. When a task succeeds, it will write a dot to the console, while a failed request will write an X to the console. The most recent failed request's status code and headers are saved. All tasks are then executed in parallel. At the conclusion of all requests, the results are written to the console:

```
csharp var totalRequests = 100; var successRequests = 0; var tasks = new  
List<Task>(); var failResponseCode = HttpStatusCode.OK; HttpResponseMessage  
failedHeaders = null; for (int i = 0; i < totalRequests; i++) {  
    tasks.Add(Task.Run(() => { var response =  
        client.GetAsync("https://graph.microsoft.com/v1.0/me/messages").Result;  
        Console.Write("."); if (response.StatusCode == HttpStatusCode.OK) {  
            successRequests++; } else { Console.Write('X'); failResponseCode =  
                response.StatusCode; failedHeaders = response.Headers; } })); } var allWork  
= Task.WhenAll(tasks); try { allWork.Wait(); } catch { }  
Console.WriteLine(); Console.WriteLine("{0}/{1} requests succeeded.",  
    successRequests, totalRequests); if (successRequests != totalRequests) {  
    Console.WriteLine("Failed response code: {0}",  
        failResponseCode.ToString()); Console.WriteLine("Failed response headers:  
    {0}", failedHeaders); }
```

## Task 7: Build and test the application

1. Run the following command in a command prompt to compile the console application:  
dotnet build

2. Run the following command to run the console application: dotnet run

**Note:** The console app may take one or two minutes to complete the process of authenticating and obtaining an access token from Azure AD and issuing the requests to Microsoft Graph.

3. After entering the username and password of a user, you will see the results written to the console.

```
Build succeeded.
  0 Warning(s)
  0 Error(s)

Time Elapsed 00:00:01.19
Hello World!
Enter your username
MeganB@M365x460234.onmicrosoft.com
Enter your password
*****
.X.X.X.X.X.X.X.X.X...XX.X.X.X.X.X.X...XX...XXX.X.X.X.X.X.X.X.X.X.X.X.
X..XX..XX.XX..XX.X.X.X.X.....X.X.X.X.....X.....
39/100 requests succeeded.
Failed response code: TooManyRequests
Failed response headers: Cache-Control: private
Transfer-Encoding: chunked
Retry-After: 1
request-id: bbef98de-74a8-4d4d-9ac3-dde92a293035
client-request-id: bbef98de-74a8-4d4d-9ac3-dde92a293035
x-ms-ags-diagnostic: {"ServerInfo":{"DataCenter":"North Central US","Slice":"SliceC","Ring":"3","ScaleUnit":"002","RoleInstance":"AGSFE_IN_18","ADSiteName":"NCU"}}
Duration: 27.2895
Strict-Transport-Security: max-age=31536000
```

There is a mix of success and failure indicators in the console. The summary states only 39% of the requests were successful.

After the results, the console has two lines that begin with **Failed response**. Notice the code states **TooManyRequests** that is the representation of the HTTP status code 429. This status code is the indication that your requests are being throttled.

Also notice within the collection of response headers, the presence of **Retry-After**. This header is the value in seconds that Microsoft Graph tells you to wait before sending your next request to avoid being further throttled.

### Add helper class to deserialize the message object returned in a REST request

It is easier to work with strongly typed objects instead of untyped JSON responses from a **REST** request. Create a helper class to simplify working with the messages objects returned from the REST request.

1. Create a new file, **Messages.cs** in the root of the project, and add the following code to it:

```
csharp using Newtonsoft.Json; using System; namespace
graphconsolethrottlepp { public class Messages { [JsonProperty(PropertyName
```

```

= "@odata.context")] public string ODataContext { get; set; }
[JsonProperty(PropertyName = "@odata.nextLink")] public string
ODataNextLink { get; set; } [JsonProperty(PropertyName = "value")] public
Message[] Items { get; set; } } public class Message {
[JsonProperty(PropertyName = "@odata.etag")] public string ETag { get; set;
} [JsonProperty(PropertyName = "id")] public string Id { get; set; }
[JsonProperty(PropertyName = "subject")] public string Subject { get; set;
} }

```

**Note:** This class is used by the JSON deserializer to translate a JSON response into a Messages object.

## Add method to implement delayed retry strategy when requests are throttled

The application is going to be modified to first get a list of messages in the current user's mailbox, then issue a separate request for the details of each message. In most scenarios, a separate request will trigger Microsoft Graph to throttle the requests.

To address this, your code should inspect each response for situations when the request has been throttled. In those situations, the code should check for the presence of a Retry-After header in the response that specifies the number of seconds your application should wait before issuing another request. If a Retry-After header isn't present, you should have a default value to fall back on.

1. Within the **Program.cs** file, add a new method **GetMessageDetail()** and the following code to it:

```
csharp private static Message GetMessageDetail(HttpClient client, string
messageId, int defaultDelay = 2) { Message messageDetail = null; string
endpoint = "https://graph.microsoft.com/v1.0/me/messages/" + messageId; // add
code here return messageDetail; }
```

2. Add the following code before the `// add code here` comment to create a request and wait for the response from Microsoft Graph:

```
csharp // submit request to Microsoft Graph & wait to process response var
clientResponse = client.GetAsync(endpoint).Result; var httpResponseTask =
clientResponse.Content.ReadAsStringAsync(); httpResponseTask.Wait();
```

3. In the case of a successful response, return the deserialized response back to the caller to display the messages. Add the following lines to the top of the **Program.cs** file to update the **using** statements:

```
csharp using Newtonsoft.Json;
```

4. Go back to the method **GetMessageDetail()** and the following code before the `// add code here` comment:

```
csharp Console.WriteLine("...Response status code: {0} ", clientResponse.StatusCode); // IF request successful (not throttled), set message to retrieved message if (clientResponse.StatusCode == HttpStatusCode.OK) { messageDetail = JsonConvert.DeserializeObject<Message>(httpResponseTask.Result); }
```

5. In the case of a throttled response, add the following **else** statement to the if statement you just added:

```
csharp // ELSE IF request was throttled (429, aka: TooManyRequests)... else
if (clientResponse.StatusCode == HttpStatusCode.TooManyRequests) { // get
```

This code will do the following:

- Set a default delay of two seconds before the next request is made.
  - If the Retry-After header value is present and greater than zero seconds, use that value to overwrite the default delay.
  - Set the thread to sleep for the specified, or default, number of seconds.
  - Recursively call the same method to retry the request.

**Tip:** In cases where the response does not include a **Retry-After** header, it is recommended to consider implementing an exponential back-off default delay. In this code, the application will initially pause for two seconds before retrying the request. Future requests will double the delay if Microsoft Graph continues to throttle the request.

Real-world applications should have an upper limit on how long they will delay so to avoid an unreasonable delay so users are not left with an unresponsive experience.

The resulting method should look like the following:

```
csharp private static Message GetMessageDetail(HttpClient client, string  
messageId, int defaultDelay = 2) { Message messageDetail = null; string  
endpoint = "https://graph.microsoft.com/v1.0/me/messages/" + messageId; //  
submit request to Microsoft Graph & wait to process response var clientResponse  
= client.GetAsync(endpoint).Result; var httpResponseTask =  
clientResponse.Content.ReadAsStringAsync(); httpResponseTask.Wait();  
Console.WriteLine("...Response status code: {0}", clientResponse.StatusCode);  
// IF request successful (not throttled), set message to retrieved message if  
(clientResponse.StatusCode == HttpStatusCode.OK) { messageDetail =  
JsonConvert.DeserializeObject<Message>(httpResponseTask.Result); } // ELSE IF  
request was throttled (429, aka: TooManyRequests)... else if  
(clientResponse.StatusCode == HttpStatusCode.TooManyRequests) { // get retry-  
after if provided; if not provided default to 2s int retryAfterDelay =  
defaultDelay; if (clientResponse.Headers.RetryAfter.Delta.HasValue &&  
(clientResponse.Headers.RetryAfter.Delta.Value.Seconds > 0)) { retryAfterDelay  
= clientResponse.Headers.RetryAfter.Delta.Value.Seconds; } // wait for  
specified time as instructed by Microsoft Graph's Retry-After header, // or  
fall back to default Console.WriteLine(">>>>>>>>>> sleeping for {0}  
seconds...", retryAfterDelay); System.Threading.Thread.Sleep(retryAfterDelay *  
1000); // call method again after waiting messageDetail =  
GetMessageDetail(client, messageId); } // add code here return messageDetail; }
```

### **Update application to use retry strategy**

The next step is to update the Main method to use the new method so the application will use an intelligent throttling strategy.

1. Locate the following line that obtains an instance of an authenticated **HttpClient** object in the **Main** method. Delete all code in the **Main** method after this line:

```
csharp var client = GetAuthenticatedHTTPClient(config, userName, userPassword);
```

2. Add the following code after obtaining the HttpClient object. This code will request the top 100 messages from the current user's mailbox and deserialize the response into a typed object you previously created:

```
csharp var stopwatch = new System.Diagnostics.Stopwatch();
stopwatch.Start(); var clientResponse =
client.GetAsync("https://graph.microsoft.com/v1.0/me/messages?
$select=id&$top=100").Result; // enumerate through the list of messages var
httpResponseTask = clientResponse.Content.ReadAsStringAsync();
httpResponseTask.Wait(); var graphMessages =
JsonConvert.DeserializeObject<Messages>(httpResponseTask.Result);
```

3. Add the following code to create individual requests for each message. These tasks are created as asynchronous tasks that will be executed in parallel:

```
csharp var tasks = new List<Task>(); foreach (var graphMessage in
graphMessages.Items) { tasks.Add(Task.Run(() => {
Console.WriteLine("...retrieving message: {0}", graphMessage.Id); var
messageDetail = GetMessageDetail(client, graphMessage.Id);
Console.WriteLine("SUBJECT: {0}", messageDetail.Subject); })); }
```

4. Next, add the following code to execute all tasks in parallel and wait for them to complete:

```
csharp // do all work in parallel & wait for it to complete var allWork =
Task.WhenAll(tasks); try { allWork.Wait(); } catch { }
```

5. With all work complete, write the results to the console:

```
csharp stopwatch.Stop(); Console.WriteLine(); Console.WriteLine("Elapsed
time: {0} seconds", stopwatch.Elapsed.Seconds);
```

## Build and test the updated application

1. Run the following command in a command prompt to compile the console application:  
dotnet build
2. Run the following command to run the console application: dotnet run

After entering the username and password for the current user, the application will write multiple log entries to the console, as in the following figure.



```
Bye\nm.0khAAAAAAEAAAABye...msaVWSbYvwm.0khAAAAAAZAA=...
...retrieving message: AAMkAGJ...HmFkOTH...LTHM...GtNGESY...05Y...A3LT...3N...Y0NDF...Y...N1Mg...GAAAAAA...Bq...t...rPG...P...R...100e4b1...n...q...h...w...d...r...v...y...m...s...a...W...S...
Bye\nm.0khAAAAAAEAAAABye...msaVWSbYvwm.0khAAAAAA...BAA=...
...retrieving message: AAMkAGJ...HmFkOTH...LTHM...GtNGESY...05Y...A3LT...3N...Y0NDF...Y...N1Mg...GAAAAAA...Bq...t...rPG...P...R...100e4b1...n...q...h...w...d...r...v...y...m...s...a...W...S...
```

Within one or two minutes, the application will display the results of the application. Depending on the speed of your workstation and internet connection, your requests may or may not have triggered Microsoft Graph to throttle you. If not, try running the application a few more times.

If your application ran fast enough, you should see some instances where Microsoft Graph returned the HTTP status code 429, indicated by the **TooManyRequests** entries.

```
SUBJECT: Your Azure AD Identity Protection Weekly Digest
...Response status code: OK
SUBJECT: Your Azure AD Identity Protection Weekly Digest
...Response status code: TooManyRequests
...Response status code: OK
...Response status code: TooManyRequests
...Response status code: OK
SUBJECT: Questions around our marketing strategy
...Response status code: OK
...Response status code: OK
...Response status code: OK
SUBJECT: Let's reschedule 1:1
...Response status code: OK
SUBJECT: Mark 8 design
SUBJECT: Northwind Proposal
SUBJECT: You have tasks due today!
...Response status code: OK
SUBJECT: Your Azure AD Identity Protection Weekly Digest
...Response status code: TooManyRequests
SUBJECT: Your Azure AD Identity Protection Weekly Digest
>>>>>>>> sleeping for 1 seconds...
>>>>>>>> sleeping for 1 seconds...
>>>>>>>> sleeping for 1 seconds...
...Response status code: OK
...Response status code: OK
SUBJECT: View your Microsoft Workplace Analytics billing statement
...Response status code: OK
```

In this case, the **messages** endpoint returned a **Retry-After** value of one (1) because the application displays messages on the console that it slept for one second.

The important point is that the application completed successfully, retrieving all 100 messages, even when some requests were rejected due to being throttled by Microsoft Graph.

## Task 8: Implement Microsoft Graph SDK for throttling retry strategy

In the last section exercise, you modified the application to implement a strategy to determine if a request is throttled. In the case where the request was throttled, as indicated by the response to the REST endpoint request, you implemented a retry strategy using the `HttpClient`.

Let's change the application to use the Microsoft Graph SDK client, which has all the logic built in for implementing the retry strategy when a request is throttled.

### Update the `GetAuthenticatedHttpClient` method

The application will use the Microsoft Graph SDK to submit requests, not the `HttpClient`, so you need to update it.

1. Locate the method `GetAuthenticatedHttpClient` and make the following changes to it:

1. Set the **return** type from `HttpClient` to `GraphServiceClient`.
2. Rename the **method** from `GetAuthenticatedHttpClient` to `GetAuthenticatedGraphClient`.
3. Replace the last two lines in the method with the following lines to obtain and return an instance of the `GraphServiceClient`:

```
csharp var graphClient = new  
GraphServiceClient(authenticationProvider); return graphClient;
```

2. Your updated method `GetAuthenticatedGraphClient` should look similar to this:

```
csharp private static GraphServiceClient  
GetAuthenticatedGraphClient(IConfigurationRoot config, string userName,  
SecureString userPassword) { var authenticationProvider =  
CreateAuthorizationProvider(config, userName, userPassword); var  
graphClient = new GraphServiceClient(authenticationProvider); return  
graphClient; }
```

### Update the application to use the `GraphServiceClient`.

1. The next step is to update the application to use the Graph SDK that includes an intelligent throttling strategy. Locate the `Messages.cs` file in the project. Delete this file or comment all code within the file out. Otherwise, the application will get the `Message` object this file contains confused with the `Message` object in the Microsoft Graph SDK.

2. Next, within the `Main` method, locate the following line:

```
csharp var client = GetAuthenticatedHTTPClient(config, userName,  
userPassword);
```

3. Update the method called in that line to use the method you updated, `GetAuthenticatedGraphClient`:

```
csharp var client = GetAuthenticatedGraphClient(config, userName,  
userPassword);
```

4. The next few lines used the **HttpClient** to call the Microsoft Graph REST endpoint to get a list of all messages. Find these lines, as shown, and remove them:

```
csharp var clientResponse =  
client.GetAsync("https://graph.microsoft.com/v1.0/me/messages?  
$select=id&$top=100").Result; // enumerate through the list of messages var  
httpResponseTask = clientResponse.Content.ReadAsStringAsync();  
httpResponseTask.Wait(); var graphMessages =  
JsonConvert.DeserializeObject<Messages>(httpResponseTask.Result);
```

5. Replace those lines with the following code to request the same information using the Microsoft Graph SDK. The collection returned by the SDK is in a different format than what the REST API returned:

```
csharp var clientResponse = client.Me.Messages .Request() .Select(m => new  
{ m.Id }) .Top(100) .GetAsync() .Result;
```

6. Locate the **foreach** loop that enumerates through all returned messages to request each message's details. Change the collection to the following code:

```
csharp foreach (var graphMessage in clientResponse.CurrentPage)
```

## Update the **GetMessageDetail** method to return

The last step is to modify the **GetMessageDetail** method that retrieved the message details for each message. Recall from the previous section in this unit that you had to write the code to detect when requests were throttled. In the where case they were throttled, you added code to retry the request after a specified delay. Fortunately, the Microsoft Graph SDK has this logic included in it.

1. Locate the **GetMessageDetail()** method.
2. Update the signature of the method so the first parameter expects an instance of the **GraphServiceClient**, not the **HttpClient**, and remove the last parameter of a default delay. The method signature should now look like the following:

```
csharp private static Microsoft.Graph.Message  
GetMessageDetail(GraphServiceClient client, string messageId)
```

3. Next, remove all code within this method and replace it with this single line:

```
csharp // submit request to Microsoft Graph & wait to process response  
return client.Me.Messages[messageId].Request().GetAsync().Result;
```

## Task 9: Build and test the updated application

1. Run the following command in a command prompt to compile the console application:  
`dotnet build`
2. Run the following command to run the console application: `dotnet run`
3. After entering the username and password for the current user, the application will write multiple log entries to the console, as in the following image.

```
SUBJECT: Let's reschedule it!
SUBJECT: Northwind Budget
SUBJECT: EMEA Training Programs
SUBJECT: Your Office 365 E5 Demo Trial is about to expire - buy today!
SUBJECT: You have upcoming tasks due.
SUBJECT: Liquidation sale on cool cups
SUBJECT: Renew your Microsoft subscriptions
SUBJECT: Renew your Microsoft subscriptions
SUBJECT: Your Azure AD Identity Protection Weekly Digest
SUBJECT: Starts today! Seattle Restaurant Week
SUBJECT: Your Azure AD Identity Protection Weekly Digest
SUBJECT: Your Azure AD Identity Protection Weekly Digest
SUBJECT: Get started with Microsoft Workplace Analytics subscription
SUBJECT: Accepted: Tailspin Toys Proposal Review + Lunch
SUBJECT: Your Azure AD Identity Protection Weekly Digest
SUBJECT: Action Needed - Renew your Microsoft subscriptions
SUBJECT: Renew your Microsoft subscriptions
SUBJECT: Northwind Proposal
SUBJECT: Audio Conferencing for Microsoft Teams or Skype for Business Online has been turned off
SUBJECT: T-minus 50 points until lift off
SUBJECT: Attention: Your Office 365 E5 Demo Trial has expired
SUBJECT: Alex Wilber is now following you on Yammer
SUBJECT: Carve out fall travel time from $59 one way.
SUBJECT: Mark B design
SUBJECT: Your organization is out of SharePoint Online storage space
SUBJECT: View your Microsoft Workplace Analytics billing statement
SUBJECT: Your Azure AD Identity Protection Weekly Digest
SUBJECT: Potluck Party Berlin
```

The application will do the same thing as the **HttpClient** version of the application. However, one difference is that the application will not display the status code returned in the response to the requests or any of the *sleeping* log messages, because the Microsoft Graph SDK handles all the retry logic internally.

## Review

In this exercise, you used the Azure AD application and .NET console application you previously created and modified them to demonstrate two strategies to account for throttling in your application. One strategy used the **HttpClient** object but required you to implement the detect, delay, and retry logic yourself when requests were throttled. The other strategy used the Microsoft Graph SDK's included support for handling this same scenario.

### Exercise 6: Querying user data from Microsoft Graph

This exercise leads the user through a series of tasks utilizing Microsoft Graph Explorer. For this exercise you will use the default sample account and will not sign in.

**Note:** The trainer should confirm users are not signed on in a browser with a Microsoft 365 account. You may also direct them to open an InPrivate window session.

By the end of this exercise you will be able to:

- Get the signed-in user's profile.
- Get a list of users in the organization.
- Get the user's profile photo.
- Get a user object based on the user's unique identifier.
- Get the user's manager profile.

## Task 1: Go to the Graph Explorer

1. Open the Microsoft Edge browser.
2. Go to this link: <https://developer.microsoft.com/en-us/graph/graph-explorer>.

This page allows users to interact with the Microsoft Graph without needing to write any code. The Microsoft Graph Explorer provides sample data to use for read operations.

**Note:** Some organizations may not allow users to sign in or consent to specific scopes required for some operations.

## Task 2: Get the signed in user's profile

1. The page loads with a request of `/v1.0/me` entered in the **request URL** box.
2. Select **Run Query**.

The data for the current users~~?~~~~?~~~~?~~ profile is shown in the **Response Preview** pane at the bottom of the page.

## **Task 3: Get a list of users in the organization**

1. In the request URL box amend the request to be: **/v1.0/users**
2. Select **Run Query**.

This shows a list of users in the organization in the **Response Preview** pane at the bottom of the page.

## Task 4: Get the user object based on the user's unique identifier

1. In the second item shown in the results, select and copy the **ID** property. This should be the ID for the Adele Vance user.
2. In the **request URL** box change the URL to request the profile for a specific user by appending the copied ID to the request URL in the form `/users/{id}`; for example:  
`/users/87d349ed-44d7-43e1-9a83-5f2406dee5bd`
3. Select **Run Query**.

The profile information for that user is shown in the **Response Preview** pane at the bottom of the page.

## Task 5: Get the user's profile photo

1. In the request URL box append **/photo**
2. The URL should now be something like:  
**https://graph.microsoft.com/v1.0/users/87d349ed-44d7-43e1-9a83-5f2406dee5bd/photo**
3. Select **Run Query**. In the **Response Preview**, metadata about the profile picture for the user is shown.
4. In the **request URL** box, append **/\$value**
5. The URL should now be something like:  
**https://graph.microsoft.com/v1.0/users/87d349ed-44d7-43e1-9a83-5f2406dee5bd/photo/\$value**
6. Select **Run Query**. The image for the default profile picture is shown in the **Response Preview** pane at the bottom of the page.

## Task 6: Get the user's manager profile

1. In the **request URL** box, replace `/photo/$value` with `/manager`
2. The URL should now be something like:  
**`https://graph.microsoft.com/v1.0/users/87d349ed-44d7-43e1-9a83-5f2406dee5bd/manager`**
3. Select **Run Query**. The profile information for the manager of the specified user is shown in the **Response Preview** pane.
4. The URL should now be something like:  
**`https://graph.microsoft.com/v1.0/users/87d349ed-44d7-43e1-9a83-5f2406dee5bd/photo`**
5. Select **Run Query**.

## Review

In this exercise, you learned how to:

- Get the signed-in user’s profile.
- Get a list of users in the organization.
- Get the user’s profile photo.
- Get a user object based on the user’s unique identifier.
- Get the user’s manager profile.

# Student lab answer key

## **Microsoft Azure user interface**

Given the dynamic nature of Microsoft cloud tools, you might experience user interface (UI) changes after the development of this training content. These changes might cause the lab instructions and steps to not match up.

The Microsoft Worldwide Learning team updates this training course as soon as the community brings needed changes to our attention. However, because cloud updates occur frequently, you might encounter UI changes before this training content is updated. If this occurs, adapt to the changes and work through them in the labs as needed.

## **Instructions**

### **Sign in to the lab virtual machine**

Sign in to the Windows 10 virtual machine using credentials.

**Note:** Lab virtual machine sign-in instructions will be provided to students by the instructor.

## Review installed applications

Observe the taskbar located at the bottom of your Windows 10 desktop. The taskbar contains the icons for the applications you will use in this lab:

- Microsoft Edge
- File Explorer
- Visual Studio Code
- Microsoft Teams

Also, ensure that the following necessary utilities are installed:

- [.NET Core 3.1 SDK](#)
- Ngrok
- NodeJS v10.x

❖❖❖# Exercise 1: Introduction to SharePoint Framework (SPFx)

# **Task 1: Set up your SharePoint Framework development environment**

## **Ensure Node.js is installed and is the correct supported version**

1. Open the PowerShell command prompt execute the following command: `node -v`
2. Ensure the version is Node.js **v10.x** and is NOT 9.x nor 11.x (these two are not supported with the SharePoint Framework).

## **Install Yeoman and gulp**

1. From the command prompt execute the following command: `npm install -g yo gulp`
2. From the command prompt execute the following command: `npm install -g @microsoft/generator-sharepoint`

## Task 2: Creating a SharePoint Framework client-side web part

1. From the PowerShell command prompt, change to the C:/LabFiles directory by executing the following command: `cd c:/LabFiles`
2. Make a new directory for your SharePoint project files by executing the following command: `md SharePoint`
3. Navigate to the newly created SharePoint directory by executing the following command: `cd SharePoint`
4. Make a new directory for your SharePoint project files by executing the following command: `md HelloWorld`
5. Navigate to the newly created SharePoint directory by executing the following command: `cd HelloWorld`
6. Run the SharePoint Yeoman generator by executing the following command: `yo @microsoft/sharepoint`
7. Use the following to complete the prompt that is displayed:
  - **What is your solution name?**: HelloWorld
  - **Which baseline packages do you want to target for your component(s)?**: SharePoint Online only (latest)
  - **Where do you want to place the files?**: Use the current folder
  - **Do you want to allow the tenant admin the choice of being able to deploy the solution to all sites immediately without running any feature deployment or adding apps in sites?**: No
  - **Will the components in the solution require permissions to access web APIs that are unique and not shared with other components in the tenant?**: No
  - **Which type of client-side component to create?**: WebPart
  - **What is your Web part name?**: HelloWorld
  - **What is your Web part description?**: HelloWorld description
  - **Which framework would you like to use?**: No JavaScript framework

### Trusting the self-signed developer certificate

1. From the PowerShell command prompt execute the following command: `gulp trust-dev-cert`
2. After provisioning the folders required for the project, the generator will install all the dependency packages using NPM for Node.js. When NPM completes downloading all dependencies, run the project by executing the following command: `gulp serve`

3. The SharePoint Framework's gulp serve task will build the project, start a local web server, and launch a browser open to the SharePoint Workbench.
4. Select the + web part icon button to open the list of available web parts.
5. Select the **HelloWorld** web part.
6. Edit the web part's properties by selecting the pencil (edit) icon in the toolbar to the left of the web part.
7. In the property pane that opens, change the value of the **Description** field. Notice how the web part updates as you make changes to the text.
8. Go back to PowerShell and stop the running process by executing **Ctrl + C**
9. When prompted to terminate type Y and hit the enter key.

## Task 3: Test with the local and hosted SharePoint Workbench

In this exercise you will work with two different versions of the SharePoint Workbench, the local and hosted workbench, as well as the different modes of the built-in gulp serve task.

1. Open the project in Visual Studio Code by executing the following command: `code .`
2. From the Visual Studio Code ribbon, select **Terminal > New Terminal**.
3. Run the project by executing the following command: `gulp serve`
4. Select the + web part icon button to open the list of available web parts.
5. Select the **HelloWorld** web part.
6. Leave the project running. Go back to Visual Studio Code.
7. Edit the HTML in the web part's **render()** method, located in the `src/webparts/helloWorld/HelloWorldWebPart.ts` file.
  1. Locate the style with the class `styles.title` and update change the title to **◆◆◆Welcome to Microsoft Learning for SharePoint!◆◆◆**
  2. Save the project and then navigate back to the browser and notice your web part title value changes.
8. Next, in the browser navigate to one of your SharePoint Online sites and append the following to the end of the root site's URL: `/layouts/workbench.aspx`. This is the SharePoint Online hosted workbench.
9. Notice that when you add a web part, many more web parts will appear beyond the one you created; and was the only one showing in the toolbox on the local workbench. This is because you are now testing the web part in a working SharePoint site.

**Note:** The difference between the local and hosted workbench is significant. Consider that a local workbench is not a working version of SharePoint; rather, it's just a single page that can let you test web parts. This means you won't have access to a real SharePoint context, lists, libraries, or real users when you are testing in the local workbench.

10. Let's see another difference with the local versus hosted workbench. Go back to the web part and make a change to the HTML.
11. Notice that, after saving the file, while the console displays a lot of commands, the browser that is displaying the hosted workbench does not automatically reload. This is expected. You can still refresh the page to see the updated web part, but the local web server cannot cause the hosted workbench to refresh. Close both the local and hosted workbench and stop the local web server by pressing **CTRL+C** in the command prompt.

## Review

In this exercise, you learned how to create a SharePoint Framework web part and test the web part using the SharePoint Framework Workbench and SharePoint site.

◆◆◆# Exercise 2: Working with the web part property pane

## Task 1: Create a new SPFx solution and web part

1. From the PowerShell command prompt, change to the **C:/LabFiles/SharePoint** directory by executing the following command: `cd c:/LabFiles/SharePoint`
2. Make a new directory for your SharePoint project files by executing the following command: `md HelloPropertyPane`
3. Navigate to the newly created SharePoint directory by executing the following command: `cd HelloPropertyPane`
4. Run the SharePoint Yeoman generator by executing the following command: `yo @microsoft/sharepoint`
5. Use the following to complete the prompt that is displayed:
  - **What is your solution name?**: HelloPropertyPane
  - **Which baseline packages do you want to target for your component(s)?**: SharePoint Online only (latest)
  - **Where do you want to place the files?**: Use the current folder
  - **Do you want to allow the tenant admin the choice of being able to deploy the solution to all sites immediately without running any feature deployment or adding apps in sites?**: No
  - **Will the components in the solution require permissions to access web APIs that are unique and not shared with other components in the tenant?**: No
  - **Which type of client-side component to create?**: WebPart
  - **What is your Web part name?**: HelloPropertyPane
  - **What is your Web part description?**: HelloPropertyPane description
  - **Which framework would you like to use?**: No JavaScript framework
6. Open the project in Visual Studio Code by executing the following command: `code .`
7. From the Visual Studio Code ribbon, select **Terminal > New Terminal**.
8. Verify that everything is working. Execute the following command to build, start the local web server, and test the web part in the local workbench: `gulp serve`
9. When the browser loads the local workbench, select the **Add a New web part** control.
10. Select the **HelloPropertyPane** web part to add the web part to the page.
11. Select the **edit web part** control to the side of the web part to display the property pane.

## Task 2: Add new properties to the web part

1. Create two new properties that will be used in the web part and property pane.
  1. From Visual Studio Code, open the file  
**src\webparts\helloPropertyPane\HelloPropertyPaneWebPart.ts**
  2. Locate the interface **IHelloPropertyPaneWebPartProps** after the import statements.  
Add the following two properties to the interface:
    - **myContinent: string;**
    - **numContinentsVisited: number;**
2. Update the web part rendering to display the values of these two properties:
  1. Within the **HelloPropertyPaneWebPart** class, locate the **render()** method.
  2. Within the **render()** method, locate the following line in the HTML output:

```
html <p class="${ styles.description }">${escape(this.properties.description)}</p>
```
  3. Add the following two lines after the line you just located:

```
html <p class="${ styles.description }">Continent where I reside: ${escape(this.properties.myContinent)}</p><p class="${ styles.description }">Number of continents I've visited: ${this.properties.numContinentsVisited}</p>
```
  4. At the moment the web part will render a blank string and undefined for these two fields as nothing is present in their values. This can be addressed by setting the default values of properties when a web part is added to the page.
3. Set the default property values:
  1. Open the file  
**src\webparts\helloPropertyPane\HelloPropertyPaneWebPart.manifest.json**
  2. Locate the following section in the file:  
`preconfiguredEntries[0].properties.description`
  3. Add a comma after the **description** property's value.
  4. Add the following two lines after the description property:  

```
json "myContinent": "North America", "numContinentsVisited": 4
```
4. Updates to the web part's manifest file will not be picked up until you restart the local web server.
  1. In the command prompt, press **CTRL+C** to stop the local web server.
  2. Rebuild and restart the local web server by executing the command `gulp serve`.

3. When the SharePoint Workbench loads, add the web part back to the page to see the properties.

## Task 3: Extend the property pane

Now that the web part has two new custom properties, the next step is to extend the property pane to allow users to edit the values.

1. Add a new text control to the property pane, connected to the **myContinent** property:

1. Open the file **src\webparts\helloPropertyPane\HelloPropertyPaneWebPart.ts**
2. Locate the method **getPropertyPaneConfiguration** and within it, locate the **groupFields** array.
3. Add a comma after the existing **PropertyPaneTextField()** call.
4. Add the following code after the comma:

```
json PropertyPaneTextField('myContinent', { label: 'Continent where I  
currently reside' })
```

2. If the local web server is not running, start it by executing `gulp serve`
3. Once the SharePoint Workbench is running again, add the web part to the page and open the property pane.
4. Notice that you can see the property and text box in the property pane. Any edits to the field will automatically update the web part.
5. Now, add a slider control to the property pane, connected to the **numContinentsVisited** property:

1. In the **HelloPropertyPaneWebPart.ts**, at the top of the file, add a **PropertyPaneSlider** reference to the existing import statement for the **@microsoft/sp-webpart-base** package.

```
typescript Import { PropertyPaneSlider } from '@microsoft/sp-webpart-  
base'
```

2. Scroll down to the method **getPropertyPaneConfiguration** and within it, locate the **groupFields** array.
3. Add a comma after the last **PropertyPaneTextField()** call, and add the following code:  

```
typescript PropertyPaneSlider('numContinentsVisited', { label: 'Number  
of continents I\'ve visited', min: 1, max: 7, showValue: true, })
```
4. Go back to the browser. Notice the property pane now has a slider control to control the number of continents you have visited.
6. In the command prompt, press **CTRL+C** to stop the local web server.
7. Close Visual Studio Code.

## Review

In this exercise, you learned how to change custom properties that exist in the web part manifest and how to add new properties.

◆◆◆# Exercise 3: Creating SharePoint Framework Extensions In this exercise, you will create a SPFx Application Customizer that will launch a Hello dialog on the load of the SharePoint modern page.

## Task 1: Create your project

1. From the PowerShell command prompt, change to the C:/LabFiles/SharePoint directory by executing the following command: `cd c:/LabFiles/SharePoint`
  2. Make a new directory for your SharePoint project files by executing the following command: `md SPFxAppCustomizer`
  3. Navigate to the newly created SharePoint directory by executing the following command: `cd SPFxAppCustomizer`
  4. Run the SharePoint Yeoman generator by executing the following command: `yo @microsoft/sharepoint`
  5. Use the following to complete the prompt that is displayed:
    - **What is your solution name?:** SPFxAppCustomizer
    - **Which baseline packages do you want to target for your component(s)?:** SharePoint Online only (latest)
    - **Where do you want to place the files?:** Use the current folder
    - **Do you want to allow the tenant admin the choice of being able to deploy the solution to all sites immediately without running any feature deployment or adding apps in sites?:** No
    - **Will the components in the solution require permissions to access web APIs that are unique and not shared with other components in the tenant?:** No
    - **Which type of client-side component to create?:** Extension
    - **What type of client-side extension to create?:** Application Customizer
    - **What is your Application Customizer name?:** HelloAppCustomizer
    - **What is your Application Customizer description?:** HelloAppCustomizer description
  6. After provisioning the folders required for the project, the generator will install all the dependency packages using NPM.
  7. Open the project in Visual Studio Code by executing the following command: `code .`
  8. From the Visual Studio Code **Terminal** prompt, execute the following command: `gulp trust-dev-cert`
- NOTE:** Extensions must be tested in a modern SharePoint page unlike web parts, which can be tested in the local workbench. In addition, extensions also require special URL parameters when requesting the page to load the extension from the local development web server.
9. Obtain the URL of a modern SharePoint page.
  10. Open the `./config/serve.json` file.

11. Copy the URL of your modern SharePoint page into the **serveConfigurations.default.pageUrl** property.
12. The SPFx build process **gulp serve** task will launch a browser and navigate to this URL, appending the necessary URL parameters to the end of the URL to load the SPFx extension from your local development web server.
  1. Run the project by executing the following command: `gulp serve`
13. When the SharePoint page loads, SharePoint will prompt you to load the debug scripts. This is a confirmation check to ensure you really want to load scripts from an untrusted source. In this case, that is your local development web server on `https://localhost`, which you can trust.
  1. Select the button **Load debug scripts**.
  2. Once the scripts load, a SharePoint dialog alert box will be shown: This alert box is shown by the application customizer. Open the application customizer file located at `/src/extensions/helloAppCustomizer/HelloAppCustomizerApplicationCustomizer.ts` and find the **OnInit()** method. Notice the following line in the method that is triggering the dialog to appear:

```
Dialog.alert(`Hello from ${strings.Title}:\n\n${message}`);
```
14. Stop the local web server by pressing **CTRL+C** in the console/terminal window.

## Task 2: Test your extension

1. Run the project by executing the following command: `gulp serve`
2. When prompted, select the **Load debug scripts** button.
3. Notice when the page loads, a Hello (your name) dialog should appear.
4. Stop the local web server by pressing CTRL+C in the console/terminal window.

## Task 3: Deploying your extension to all sites in the SharePoint Online tenant

1. Locate and open the `./config/package-solution.json` file.
  1. Ensure the solution object has a property named **skipFeatureDeployment** and ensure that the value of this property is set to **true**.
2. Locate and open the `./sharepoint/assets/ClientSideInstance.xml` file. This file contains the values that will be automatically set on the Tenant Wide Extensions list in your SharePoint Online tenant's App Catalog site when the package is deployed.
3. Build and package the solution by running the following commands, one at a time:

```
powershell gulp build gulp bundle --ship gulp package-solution --ship  
NOTE: If this error The build failed because a task wrote output to stderr. is displayed on the command console, please ignore it. The reason is that the build output contain a warning.
```

4. In the browser, navigate to your SharePoint Online tenant App Catalog site.
  5. Select the Apps for SharePoint list in the left navigation pane.
  6. Drag the generated `./sharepoint/solution/*.sppkg` file into the Apps for SharePoint list.
  7. In the **Do you trust spfx-app-customizer-client-side-solution?** dialog box:
    1. Select the **Make this solution available to all sites in the organization** check box.
    2. Notice the message **This package contains an extension which will be automatically enabled across sites....**
    3. Select **Deploy**.
  8. Select **Site contents** in the left navigation pane.
  9. Select **Tenant Wide Extensions**. Depending on when your tenant was created the Tenant Wide Extensions list may be hidden. If you do not see the list in the Site Contents then you will have to navigate to it manually. Do this by appending `/Lists/TenantWideExtensions/AllItems.aspx` to the URL of the app catalog site.
  10. In a separate browser window, navigate to any modern page in any modern site within your SharePoint Online tenant. If this extension has been deployed successfully you should see a Hello (your name) dialog prompt on the load of the page.
- NOTE:** It may take up to 20 minutes for a Tenant Wide Extension to get deployed across the SharePoint Online tenant so you may need to wait to fully test whether your deployment was successful.
11. Stop the local web server by pressing CTRL+C in the console/terminal window.

## **Review**

In this exercise, you learned how to create and deploy a custom header and footer extension that is applied to all sites in a SharePoint Online tenant.  # Exercise 4: Creating a command set extension

## Task 1: Create your project

1. From the PowerShell command prompt, change to the C:/LabFiles/SharePoint directory by executing the following command: `cd c:/LabFiles/SharePoint`
2. Make a new directory for your SharePoint project files by executing the following command: `md SPFxCommandSet`
3. Navigate to the newly created SharePoint directory by executing the following command: `cd SPFxCommandSet`
4. Run the SharePoint Yeoman generator by executing the following command: `yo @microsoft/sharepoint`
5. Use the following to complete the prompt that is displayed:
  - **What is your solution name?:** SPFxCommandSet
  - **Which baseline packages do you want to target for your component(s)?:** SharePoint Online only (latest)
  - **Where do you want to place the files?:** Use the current folder
  - **Do you want to allow the tenant admin the choice of being able to deploy the solution to all sites immediately without running any feature deployment or adding apps in sites?:** No
  - **Will the components in the solution require permissions to access web APIs that are unique and not shared with other components in the tenant?:** No
  - **Which type of client-side component to create?:** Extension
  - **What type of client-side extension to create?:** ListView Command Set
  - **What is your Command Set name?:** CommandSetDemo
  - **What is your Command Set description?:** CommandSetDemo description
6. After provisioning the folders required for the project, the generator will install all the dependency packages using NPM.
7. From the PowerShell command prompt execute the following command: `gulp trust-dev-cert`
8. Open the project in Visual Studio Code by executing the following command: `code .`

## Task 2: Observe the Code of your ListView Command Set

1. From Visual Studio Code, open the **CommandSetDemoCommandSet.ts** file in the **src\extensions\commandSetDemo** folder.
2. Notice that the base class for the ListView Command Set is imported from the **sp-listview-extensibility** package, which contains SharePoint Framework code required by the ListView Command Set.

```
typescript import { override } from '@microsoft/decorators'; import { Log }  
from '@microsoft/sp-core-library'; import { BaseListViewCommandSet,  
Command, IListViewCommandSetListViewUpdatedParameters,  
IListViewCommandSetExecuteEventParameters } from '@microsoft/sp-listview-  
extensibility'; import { Dialog } from '@microsoft/sp-dialog';
```

The behavior for your custom buttons is contained in the **onListViewUpdated()** and **OnExecute()** methods.

The **onListViewUpdated()** event occurs separately for each command (for example, a menu item) whenever a change happens in the ListView, and the UI needs to be re-rendered. The event function parameter represents information about the command being rendered. The handler can use this information to customize the title or adjust the visibility, for example, if a command should only be shown when a certain number of items are selected in the list view. This is the default implementation. When using the method **tryGetCommand**, you get a Command object, which is a representation of the command that shows in the UI. You can modify its values, such as **title**, or **visible**, to modify the UI element. SPFx uses this information when re-rendering the commands. These objects keep the state from the last render, so if a command is set to **visible = false**, it remains invisible until it is set back to **visible = true**.

```
typescript @override public onListViewUpdated(event:  
IListViewCommandSetListViewUpdatedParameters): void { const compareOneCommand:  
Command = this.tryGetCommand('COMMAND_1'); if (compareOneCommand) { // This  
command should be hidden unless exactly one row is selected.  
compareOneCommand.visible = event.selectedRows.length === 1; } }
```

The **OnExecute()** method defines what happens when a command is executed (for example, the menu item is selected). In the default implementation, different messages are shown based on which button was selected.

```
typescript @override public onExecute(event:  
IListViewCommandSetExecuteEventParameters): void { switch (event.itemId) { case  
'COMMAND_1': Dialog.alert(` ${this.properties.sampleTextOne}`); break; case  
'COMMAND_2': Dialog.alert(` ${this.properties.sampleTextTwo}`); break; default:  
throw new Error('Unknown command'); } }
```

## Task 3: Test your extension

You cannot currently use the local Workbench to test SharePoint Framework Extensions. You'll need to test and develop them directly against a live SharePoint Online site. You don't have to deploy your customization to the app catalog to do this, which makes the debugging experience simple and efficient. Go to any SharePoint list in your SharePoint Online site by using the modern experience or create a new list. Copy the URL off the list to clipboard. Because our ListView Command Set is hosted from localhost and is running, you can use specific debug query parameters to execute the code in the list view. Open **serve.json** file from **config** folder. Update the **pageUrl** attributes to match a URL of the list where you want to test the solution. After edits your serve.json should look somewhat like:

```
powershell { "$schema": "https://developer.microsoft.com/json-schemas/core-build/serve.schema.json", "port": 4321, "https": true, "serveConfigurations": { "default": { "pageUrl": "https://contoso.sharepoint.com/sites/Group/Lists/Orders/AllItems.aspx", "customActions": { "bf232d1d-279c-465e-a6e4-359cb4957377": { "location": "ClientSideExtension.ListViewCommandSet.CommandBar", "properties": { "sampleTextOne": "One item is selected in the list", "sampleTextTwo": "This command is always visible." } } }, "helloWorld": { "pageUrl": "https://sppnp.sharepoint.com/sites/Group/Lists/Orders/AllItems.aspx", "customActions": { "bf232d1d-279c-465e-a6e4-359cb4957377": { "location": "ClientSideExtension.ListViewCommandSet.CommandBar", "properties": { "sampleTextOne": "One item is selected in the list", "sampleTextTwo": "This command is always visible." } } } } }
```

1. From the Visual Studio Code ribbon, select **Terminal > New Terminal**.
2. Run the project by executing the following command: `gulp serve`
3. When prompted, select the **Load debug scripts** button.
4. Stop the local web server by pressing **CTRL+C** in the console/terminal window.

## Review

In this exercise, you learned how to create and deploy a command set extension. Command set extensions are used for adding additional commands into the command bar at the top of a page.

◆◆◆# Exercise 5: Creating a field customizer extension

## Task 1: Create your project

1. From the PowerShell command prompt, change to the C:/LabFiles/SharePoint directory by executing the following command: `cd c:/LabFiles/SharePoint`
2. Make a new directory for your SharePoint project files by executing the following command: `md SPFxFieldCustomizer`
3. Navigate to the newly created SharePoint directory by executing the following command: `cd SPFxFieldCustomizer`
4. Run the SharePoint Yeoman generator by executing the following command: `yo @microsoft/sharepoint`
5. Use the following to complete the prompt that is displayed:
  - **What is your solution name?:** SPFxFieldCustomizer
  - **Which baseline packages do you want to target for your component(s)?:** SharePoint Online only (latest)
  - **Where do you want to place the files?:** Use the current folder
  - **Do you want to allow the tenant admin the choice of being able to deploy the solution to all sites immediately without running any feature deployment or adding apps in sites?:** No
  - **Will the components in the solution require permissions to access web APIs that are unique and not shared with other components in the tenant?:** No
  - **Which type of client-side component to create?:** Extension
  - **What type of client-side extension to create?:** Field Customizer
  - **What is your Field Customizer name?:** HelloFieldCustomizer
  - **What is your Field Customizer description?:** HelloFieldCustomizer description
  - **Which framework would you like to use?:** No JavaScript Framework
6. After provisioning the folders required for the project, the generator will install all the dependency packages using NPM.

## Task 2: Updating the SCSS styles for the field customizer

1. Locate and open the  
**./src/extensions/helloFieldCustomizer>HelloFieldCustomizerFieldCustomizer.module.scss** file.
2. Replace the contents of the file with the following styles:

```
html .HelloFieldCustomizer { .cell { display: 'inline-block'; }  
.filledBackground { background-color: #cccccc; width: 100px; } }
```

## Task 3: Update the code for the field customizer

1. Locate and open the `./src/extensions/helloFieldCustomizer/HelloFieldCustomizerFieldCustomizer.ts` file.
2. Locate the interface **IHelloFieldCustomizerFieldCustomizerProperties** and update its properties to the following: `greenMinLimit?: string; yellowMinLimit?: string;`
3. Locate the method **onRenderCell()** and update the contents to match the following code. This code looks at the existing value in the field and builds the relevant colored bars based on the value entered in the completed percentage value.

```
typescript event.documentElement.classList.add(styles.cell);
event.documentElement.innerHTML = ` <div class='${styles.HelloFieldCustomizer}'>
<div class='${styles.filledBackground}'> <div style='width:
${event.fieldValue}px; background:#0094ff; color:#c0c0c0'>
${event.fieldValue} </div> </div> </div>`;
```

## Task 4: Update the deployment code for the field customizer

Field customizers, when deployed to production, are implemented by creating a new site column who's rendering is tied to the custom script defined in the field customizer's bundle file.

1. Locate and open the **./sharepoint/assets/elements.xml** file.
2. Add the following property to the element, setting the values on the public properties on the field customizer:

```
xml ClientSideComponentProperties=""
{"greenMinLimit":"85","yellowMinLimit":"70"}"
```

## Task 5: Testing your field customizer

1. In a browser, navigate to a SharePoint Online modern site collection where you want to test the field customizer.
2. Select the **Site contents** link in the left navigation pane.
3. Create a new SharePoint list:
  1. Select **New > List** in the toolbar.
  2. Set the list name to **Work Status** and select **Create**.
4. When the list loads, select the **Add column > Number** to create a new column.
5. When prompted for the name of the column, enter **PercentComplete**.
6. Add a few items to the list, with different numbers in the percent field.
7. Update the properties for the serve configuration used to test and debug the extension:
  1. Locate and open the `./config/serve.json` file.
  2. Copy in the full URL (including `AllItems.aspx`) of the list you just created into the `serveConfigurations.default.pageUrl` property.
  3. Locate the `serveConfigurations.default.properties` object.
  4. Change the name of the property `serveConfigurations.default.fieldCustomizers.InternalFieldName` to `serveConfigurations.default.fieldCustomizers.PercentComplete`. This tells the SharePoint Framework which existing field to associate the field customizer with.
  5. Change the value of the properties object to the following: `"properties": { "greenMinLimit": "85", "yellowMinLimit": "70" }`
  6. The JSON code for the default serve configuration should look something like the following:

```
json "default": { "pageUrl": "https://contoso.sharepoint.com/sites/mySite/Lists/Work%20Status/AllItems.aspx", "fieldCustomizers": { "PercentComplete": { "id": "6a1b8997-00d5-4bc7-a472-41d6ac27cd83", "properties": { "greenMinLimit": "85", "yellowMinLimit": "70" } } } }
```
  8. Run the project by executing the following command: `gulp serve`
  9. When prompted, select the **Load debug scripts** button.
  10. Stop the local web server by pressing **CTRL+C** in the console/terminal window.

## Review

In this exercise, you learned how to create and deploy a command set extension. Command set extensions are used for adding additional commands into the command bar at the top of a page.

◆◆◆# Exercise 6: Deploying a SharePoint Framework solution

## Task 1: Create your project

1. From the PowerShell command prompt, change to the C:/LabFiles/SharePoint directory by executing the following command: `cd c:/LabFiles/SharePoint`
2. Make a new directory for your SharePoint project files by executing the following command: `md DeploymentDemo`
3. Navigate to the newly created SharePoint directory by executing the following command: `cd DeploymentDemo`
4. Run the SharePoint Yeoman generator by executing the following command: `yo @microsoft/sharepoint`
5. Use the following to complete the prompt that is displayed:
  - **What is your solution name?:** DeploymentDemo
  - **Which baseline packages do you want to target for your component(s)?:** SharePoint Online only (latest)
  - **Where do you want to place the files?:** Use the current folder
  - **Do you want to allow the tenant admin the choice of being able to deploy the solution to all sites immediately without running any feature deployment or adding apps in sites?:** No
  - **Will the components in the solution require permissions to access web APIs that are unique and not shared with other components in the tenant?:** No
  - **Which type of client-side component to create?:** WebPart
  - **What is your Web part name?:** Deployment Demo
  - **What is your Web part description?:** Deployment Demo description
  - **Which framework would you like to use?:** No JavaScript framework
6. After provisioning the folders required for the project, the generator will install all the dependency packages using NPM.

## **Task 2: Creating a deployment package for the project**

1. When NPM completes downloading all dependencies, build the project by running the following command on the command line from the root of the project: `gulp build`
2. Create a production bundle of the project by running the following command on the command line from the root of the project: `gulp bundle --ship`
3. Create a deployment package of the project by running the following command on the command line from the root of the project: `gulp package-solution --ship`

## **Task 3: Deploying the package to a SharePoint site**

1. In a browser, navigate to your SharePoint tenant's App Catalog site.
2. Select the **Apps for SharePoint** link in the left navigation pane.
3. Drag the package created in the previous steps, located in the project's **./sharepoint/solution/deployment-demo.sppkg**, into the **Apps for SharePoint** library.
4. SharePoint will launch a dialog box asking if you want to trust the package.
5. Select **Deploy**.

## **Task 4: Installing the SharePoint package in a site collection**

1. Navigate to an existing site collection, or create a new one.
2. Select **Site Contents** from the left navigation pane.
3. From the **New** menu, select **App**.
4. Locate the solution you previously deployed and select it.

SharePoint will start to install the application. At first it will appear dimmed, but after a few moments you should see it listed.

## **Task 5: Adding the web part to a page**

1. Navigate to a SharePoint page.
2. Put it in edit mode by selecting the **Edit** button in the upper-right portion of the content area on the page.
3. Select the web part icon button to open the list of available web parts.
4. Select the expand icon, a diagonal line with two arrows in the upper-right corner, to expand the web part toolbox.
5. Scroll to the bottom; locate and select the **Deployment Demo** web part.

## **Task 6: Examining the deployed web part files**

1. Once the page loads, open the browser's developer tools and navigate to the **Sources** tab.
2. Refresh the page and examine where the JavaScript bundle is being hosted.

**Note:** If you have not enabled the Office 365 CDN then the bundle will be hosted from a document library named ClientSideAssets in the app catalog site. If you have enabled the Office 365 CDN then the bundle will be automatically hosted from the CDN.

## **Review**

In this exercise, you learned how to package and deploy your SharePoint Framework solutions.

◆◆◆# Exercise 7: Deploying SPFx solutions to Microsoft Teams

## Task 1: Create your project

1. From the PowerShell command prompt, change to the C:/LabFiles/SharePoint directory by executing the following command: `cd c:/LabFiles/SharePoint`
2. Make a new directory for your SharePoint project files by executing the following command: `md SPFxTeamsTab`
3. Navigate to the newly created SharePoint directory by executing the following command: `cd SPFxTeamsTab`
4. Run the SharePoint Yeoman generator by executing the following command: `yo @microsoft/sharepoint`
5. Use the following to complete the prompt that is displayed:
  - **What is your solution name?:** SPFxTeamsTab
  - **Which baseline packages do you want to target for your component(s)?:** SharePoint Online only (latest)
  - **Where do you want to place the files?:** Use the current folder
  - **Do you want to allow the tenant admin the choice of being able to deploy the solution to all sites immediately without running any feature deployment or adding apps in sites?:** Yes
  - **Will the components in the solution require permissions to access web APIs that are unique and not shared with other components in the tenant?:** No
  - **Which type of client-side component to create?:** WebPart
  - **What is your Web Part name?:** SPFx Teams Together
  - **What is your Web Part description?:** SPFx Teams Together description
  - **Which framework would you like to use?:** No JavaScript framework
6. After provisioning the folders required for the project, the generator will install all the dependency packages using NPM.
7. Open the project in an editor such as Visual Studio Code.
8. Enable the web part to be used in Microsoft Teams:
  1. Locate and open the file  
`/src/webparts/spFxTeamsTogether/SpFxTeamsTogetherWebPart.manifest.json`  
.
  2. Within the web part manifest file, locate the property **supportedHosts**:  
`"supportedHosts": ["SharePointWebPart"],`
  3. Add another option to enable this web part to be used as a tab in a Microsoft Teams team: `"supportedHosts": ["SharePointWebPart", "TeamsTab"],`

## Task 2: Creating and deploying the Microsoft Teams app package

Notice the project contains a folder **teams** that contains two images. These are used in Microsoft Teams to display the custom tab.

You may notice there is no **manifest.json** file present. The manifest file can be generated automatically by SharePoint from the App Catalog site. However, at this time, this functionality is not fully operational.

Alternatively, you can manually create the manifest file if you want to have more control over the default values that SharePoint will create.

The automatic generation and deployment of the Microsoft Teams manifest is not currently operational; you will manually create the Microsoft Teams manifest and app package:

1. Create a **manifest.json** file in **.teams** folder, use the following manifest file as a template:

```
json { "$schema": "https://developer.microsoft.com/json-schemas/teams/v1.5/MicrosoftTeams.schema.json", "manifestVersion": "1.5", "packageName": "{{SPFX_COMPONENT_ALIAS}}", "id": "{{SPFX_COMPONENT_ID}}", "version": "0.1", "developer": { "name": "Parker Porcupine", "websiteUrl": "https://contoso.com", "privacyUrl": "https://contoso.com/privacystatement", "termsOfUseUrl": "https://contoso.com/servicesagreement" }, "name": { "short": "{{SPFX_COMPONENT_NAME}}", "description": { "short": "{{SPFX_COMPONENT_SHORT_DESCRIPTION}}", "full": "{{SPFX_COMPONENT_LONG_DESCRIPTION}}" } }, "icons": { "outline": "{{SPFX_COMPONENT_ID}}_outline.png", "color": "{{SPFX_COMPONENT_ID}}_color.png", "accentColor": "#004578", "staticTabs": [ { "entityId": "com.contoso.personaltab.spfx", "name": "My SPFx Personal Tab", "contentUrl": "https://{teamSiteDomain}/_layouts/15/TeamsLogon.aspx?SPFx=true&dest=/_layouts/15/teamshostedapp.aspx%3Fteams%26personal%26componentId={{SPFX_COMPONENT_ID}}%26forceLocale={locale}", "scopes": [ "personal" ] }, "configurableTabs": [ { "configurationUrl": "https://{teamSiteDomain}{teamSitePath}/_layouts/15/TeamsLogon.aspx?SPFx=true&dest={teamSitePath}/_layouts/15/teamshostedapp.aspx%3FopenPropertyPane=true%26team%26componentId={{SPFX_COMPONENT_ID}}%26forceLocale={locale}" }, "canUpdateConfiguration": true, "scopes": [ "team" ] ], "validDomains": [ "*.login.microsoftonline.com", "*.sharepoint.com", "spoprod-a.akamaihd.net", "resourceseng.blob.core.windows.net" ], "webApplicationInfo": { "resource": "https://{teamSiteDomain}", "id": "00000003-0000-0000-0000-000000000000" } }
```

2. Open the **manifest.json** file. This file contains multiple strings that need to be updated to match the SPFx component. Use the following table to determine the values that should be replaced. The SPFx component properties are found in the web part manifest file:

<b>/src/webparts/spFxTeamsTogether/SpFxTeamsTogetherWebPart.manifest.json   manifest.json string  Property in SPFx component manifest  :---   :---   </b>
{ {{SPFX_COMPONENT_ALIAS}}  alias  {{SPFX_COMPONENT_NAME}}  preconfiguredEntries[0].title  {{SPFX_COMPONENT_SHORT_DESCRIPTION}}  preconfiguredEntries[0].description  {{SPFX_COMPONENT_LONG_DESCRIPTION}}  preconfiguredEntries[0].description  {{SPFX_COMPONENT_ID}}  id

**Note:** Don't miss replacing **{{SPFX\_COMPONENT\_ID}}** in **configurableTabs[0].configurationUrl**. You will likely have to scroll your editor to the

right to see it. The tokens surrounded by single curly braces (for example, **{teamSiteDomain}**) do not need to be replaced.

3. Create a Microsoft Teams app package by zipping the contents of the **./teams** folder. Make sure to zip just the contents and not the folder itself. This ZIP archive should contain three files at the root: two images and the **manifest.json**.

## Task 3: Create and deploy the SharePoint package

In order to test the web part in SharePoint and Microsoft Teams, it must first be deployed to an app catalog.

1. Open a browser and navigate to your SharePoint Online Tenant-SScoped App Catalog site.
2. Select the menu item **Apps for SharePoint** from the left navigation menu.
3. Build the project by opening a command prompt and changing to the root folder of the project. Then execute the following command: `gulp build`
4. Next, create a production bundle of the project by running the following command on the command line from the root of the project: `gulp bundle --ship`
5. Create a deployment package of the project by running the following command on the command line from the root of the project: `gulp package-solution --ship`
6. Locate the file created by the gulp task, found in the **./sharepoint/solution** folder with the name \*.sppkg.
7. Drag this file into **the Apps for SharePoint** library in the browser.
8. In the **Do you trust...?** dialog box, select the check box **Make this solution available to all sites in the organization** and then select **Deploy**.
9. This will make the SPFx web part available to all site collections in the tenant, including those that are behind a Microsoft Teams team.

## **Task 4: Test the web part in SharePoint and Microsoft Teams**

Test the SPFx web part in SharePoint:

1. In the browser, navigate to a modern SharePoint page.
2. Select the **Edit** button in the upper-right portion of the page.
3. Select the (+) icon to open the SharePoint web part toolbox and locate the web part SPFx Teams Together:
4. The SharePoint Framework web part will be displayed on the page.

## Task 5: Testing the SPFx web part in Microsoft Teams

1. Create a new Microsoft Teams team.
  1. Using the same browser where you are signed in to SharePoint Online, navigate to <https://teams.microsoft.com>. When prompted, load the web client.
  2. If you do not have any teams in your tenant, you will be presented with the following dialog. Otherwise, select **Join or create a team** at the bottom of the list of teams.
  3. On the **Create your team** dialog box, select **Build a team from scratch**.
  4. On the **What kind of team will this be?** dialog box, select **Public**.
  5. When prompted, use the name **My First Team**.
2. Install the Microsoft Teams application as a new tab that will expose the SharePoint Framework web part in Microsoft Teams:
  1. Select the **My First Team** team previously created.
  2. Select the **General** channel.
3. Add a custom tab to the team using the SPFx web part:
  1. At the top of the page, select the + icon in the horizontal navigation.
  2. In the **Add a tab** dialog box, select **More Apps**.
  3. Select the **Upload a custom app > Upload for ...** from the list of app categories.
  4. Select the Microsoft Teams application ZIP file previously created. This is the file that contains the **manifest.json** and two image files.

**Note:** After a moment, the application will appear next to your tenant name. You may need to refresh the page for the app to appear if you are using the browser Microsoft Teams client.
4. Select the **SPFx Teams Together** app.
5. In the **SPFx Teams Together** dialog box, select **Add to a team**.
6. In the **Select a channel to start using SPFx Teams Together** dialog box, make sure that the **General** channel is selected and select **Set up a tab**.
7. The next dialog box will confirm the installation of the app. Select **Save**.
8. The application should now load in Microsoft Teams within the **General** channel under the tab **SPFx Teams Together**.

## Review

In this exercise, you learned how to create a SharePoint Framework (SPFx) solution that will work in both SharePoint and as a tab in Microsoft Teams.

◆◆◆# Student lab manual

## Lab scenario

You are a developer at a growing organization that has recently migrated from SharePoint on-premises to SharePoint Online. You are being asked implement features into SharePoint Online that are not available out-of-the-box. You decided to extend the functionality and user interface of SharePoint Online to provide the functionality that the organization needs. As a result, user adoption will increase.

## **Objectives**

After you complete this lab, you will be able to:

- Describe the components of a SharePoint Framework (SPFx) web part.
- Describe SPFx extensions.
- Describe the process to package and deploy a SPFx solution.
- Describe the consumption of Microsoft Graph.
- Describe the consumption of third-party APIs secured with Azure AD from within SPFx.
- Describe web parts as Teams tabs.
- Describe branding and theming in SharePoint Online.

## **Lab setup**

**Estimated Time:** 240 minutes

## **Instructions**

Sign in to the Windows 10 virtual machine using credentials.

**Note:** Lab virtual machine sign-in instructions will be provided to students by the instructor.

## Review installed applications

Observe the taskbar located at the bottom of your Windows 10 desktop. The taskbar contains the icons for the applications you will use in this lab:

- Microsoft Edge
- File Explorer
- Visual Studio Code
- Microsoft Teams
- File Explorer

Also, ensure that the following necessary utilities are installed:

- [.NET Core 3.1 SDK](#)
- Ngrok
- NodeJS v10.x

❖❖❖# Exercise 1: Introduction to SharePoint Framework (SPFx)

# **Task 1: Set up your SharePoint Framework development environment**

## **Ensure Node.js is installed and is the correct supported version**

1. Open the PowerShell command prompt execute the following command: `node -v`
2. Ensure the version is Node.js **v10.x** and is NOT 9.x nor 11.x (these two are not supported with the SharePoint Framework).

## **Install Yeoman and gulp**

1. From the command prompt execute the following command: `npm install -g yo gulp`
2. From the command prompt execute the following command: `npm install -g @microsoft/generator-sharepoint`

## Task 2: Creating a SharePoint Framework client-side web part

1. From the PowerShell command prompt, change to the C:/LabFiles directory by executing the following command: `cd c:/LabFiles`
2. Make a new directory for your SharePoint project files by executing the following command: `md SharePoint`
3. Navigate to the newly created SharePoint directory by executing the following command: `cd SharePoint`
4. Make a new directory for your SharePoint project files by executing the following command: `md HelloWorld`
5. Navigate to the newly created SharePoint directory by executing the following command: `cd HelloWorld`
6. Run the SharePoint Yeoman generator by executing the following command: `yo @microsoft/sharepoint`
7. Use the following to complete the prompt that is displayed:
  - **What is your solution name?**: HelloWorld
  - **Which baseline packages do you want to target for your component(s)?**: SharePoint Online only (latest)
  - **Where do you want to place the files?**: Use the current folder
  - **Do you want to allow the tenant admin the choice of being able to deploy the solution to all sites immediately without running any feature deployment or adding apps in sites?**: No
  - **Will the components in the solution require permissions to access web APIs that are unique and not shared with other components in the tenant?**: No
  - **Which type of client-side component to create?**: WebPart
  - **What is your Web part name?**: HelloWorld
  - **What is your Web part description?**: HelloWorld description
  - **Which framework would you like to use?**: No JavaScript framework

### Trusting the self-signed developer certificate

1. From the PowerShell command prompt execute the following command: `gulp trust-dev-cert`
2. After provisioning the folders required for the project, the generator will install all the dependency packages using NPM for Node.js. When NPM completes downloading all dependencies, run the project by executing the following command: `gulp serve`

3. The SharePoint Framework's gulp serve task will build the project, start a local web server, and launch a browser open to the SharePoint Workbench.
4. Select the + web part icon button to open the list of available web parts.
5. Select the **HelloWorld** web part.
6. Edit the web part's properties by selecting the pencil (edit) icon in the toolbar to the left of the web part.
7. In the property pane that opens, change the value of the **Description** field. Notice how the web part updates as you make changes to the text.
8. Go back to PowerShell and stop the running process by executing **Ctrl + C**
9. When prompted to terminate type Y and hit the enter key.

## Task 3: Test with the local and hosted SharePoint Workbench

In this exercise you will work with two different versions of the SharePoint Workbench, the local and hosted workbench, as well as the different modes of the built-in gulp serve task.

1. Open the project in Visual Studio Code by executing the following command: `code .`
2. From the Visual Studio Code ribbon, select **Terminal > New Terminal**.
3. Run the project by executing the following command: `gulp serve`
4. Select the + web part icon button to open the list of available web parts.
5. Select the **HelloWorld** web part.
6. Leave the project running. Go back to Visual Studio Code.
7. Edit the HTML in the web part's **render()** method, located in the `src/webparts/helloWorld/HelloWorldWebPart.ts` file.
  1. Locate the style with the class `styles.title` and update change the title to **◆◆◆Welcome to Microsoft Learning for SharePoint!◆◆◆**
  2. Save the project and then navigate back to the browser and notice your web part title value changes.
8. Next, in the browser navigate to one of your SharePoint Online sites and append the following to the end of the root site's URL: `/layouts/workbench.aspx`. This is the SharePoint Online hosted workbench.
9. Notice that when you add a web part, many more web parts will appear beyond the one you created; and was the only one showing in the toolbox on the local workbench. This is because you are now testing the web part in a working SharePoint site.

**Note:** The difference between the local and hosted workbench is significant. Consider that a local workbench is not a working version of SharePoint; rather, it's just a single page that can let you test web parts. This means you won't have access to a real SharePoint context, lists, libraries, or real users when you are testing in the local workbench.

10. Let's see another difference with the local versus hosted workbench. Go back to the web part and make a change to the HTML.
11. Notice that, after saving the file, while the console displays a lot of commands, the browser that is displaying the hosted workbench does not automatically reload. This is expected. You can still refresh the page to see the updated web part, but the local web server cannot cause the hosted workbench to refresh. Close both the local and hosted workbench and stop the local web server by pressing **CTRL+C** in the command prompt.

## Review

In this exercise, you learned how to create a SharePoint Framework web part and test the web part using the SharePoint Framework Workbench and SharePoint site.

◆◆◆# Exercise 2: Working with the web part property pane

## Task 1: Create a new SPFx solution and web part

1. From the PowerShell command prompt, change to the **C:/LabFiles/SharePoint** directory by executing the following command: `cd c:/LabFiles/SharePoint`
2. Make a new directory for your SharePoint project files by executing the following command: `md HelloPropertyPane`
3. Navigate to the newly created SharePoint directory by executing the following command: `cd HelloPropertyPane`
4. Run the SharePoint Yeoman generator by executing the following command: `yo @microsoft/sharepoint`
5. Use the following to complete the prompt that is displayed:
  - **What is your solution name?**: HelloPropertyPane
  - **Which baseline packages do you want to target for your component(s)?**: SharePoint Online only (latest)
  - **Where do you want to place the files?**: Use the current folder
  - **Do you want to allow the tenant admin the choice of being able to deploy the solution to all sites immediately without running any feature deployment or adding apps in sites?**: No
  - **Will the components in the solution require permissions to access web APIs that are unique and not shared with other components in the tenant?**: No
  - **Which type of client-side component to create?**: WebPart
  - **What is your Web part name?**: HelloPropertyPane
  - **What is your Web part description?**: HelloPropertyPane description
  - **Which framework would you like to use?**: No JavaScript framework
6. Open the project in Visual Studio Code by executing the following command: `code .`
7. From the Visual Studio Code ribbon, select **Terminal > New Terminal**.
8. Verify that everything is working. Execute the following command to build, start the local web server, and test the web part in the local workbench: `gulp serve`
9. When the browser loads the local workbench, select the **Add a New web part** control.
10. Select the **HelloPropertyPane** web part to add the web part to the page.
11. Select the **edit web part** control to the side of the web part to display the property pane.

## Task 2: Add new properties to the web part

1. Create two new properties that will be used in the web part and property pane.
  1. From Visual Studio Code, open the file  
**src\webparts\helloPropertyPane\HelloPropertyPaneWebPart.ts**
  2. Locate the interface **IHelloPropertyPaneWebPartProps** after the import statements.  
Add the following two properties to the interface:
    - **myContinent: string;**
    - **numContinentsVisited: number;**
2. Update the web part rendering to display the values of these two properties:
  1. Within the **HelloPropertyPaneWebPart** class, locate the **render()** method.
  2. Within the **render()** method, locate the following line in the HTML output:

```
html <p class="${ styles.description }">${escape(this.properties.description)}</p>
```
  3. Add the following two lines after the line you just located:

```
html <p class="${ styles.description }">Continent where I reside: ${escape(this.properties.myContinent)}</p><p class="${ styles.description }">Number of continents I've visited: ${this.properties.numContinentsVisited}</p>
```
  4. At the moment the web part will render a blank string and undefined for these two fields as nothing is present in their values. This can be addressed by setting the default values of properties when a web part is added to the page.
3. Set the default property values:
  1. Open the file  
**src\webparts\helloPropertyPane\HelloPropertyPaneWebPart.manifest.json**
  2. Locate the following section in the file:  
`preconfiguredEntries[0].properties.description`
  3. Add a comma after the **description** property's value.
  4. Add the following two lines after the description property:  

```
json "myContinent": "North America", "numContinentsVisited": 4
```
4. Updates to the web part's manifest file will not be picked up until you restart the local web server.
  1. In the command prompt, press **CTRL+C** to stop the local web server.
  2. Rebuild and restart the local web server by executing the command `gulp serve`.

3. When the SharePoint Workbench loads, add the web part back to the page to see the properties.

## Task 3: Extend the property pane

Now that the web part has two new custom properties, the next step is to extend the property pane to allow users to edit the values.

1. Add a new text control to the property pane, connected to the **myContinent** property:

1. Open the file **src\webparts\helloPropertyPane\HelloPropertyPaneWebPart.ts**
2. Locate the method **getPropertyPaneConfiguration** and within it, locate the **groupFields** array.
3. Add a comma after the existing **PropertyPaneTextField()** call.
4. Add the following code after the comma:

```
typescript PropertyPaneTextField('myContinent', { label: 'Continent where I currently reside' })
```

2. If the local web server is not running, start it by executing `gulp serve`
3. Once the SharePoint Workbench is running again, add the web part to the page and open the property pane.
4. Notice that you can see the property and text box in the property pane. Any edits to the field will automatically update the web part.
5. Now, add a slider control to the property pane, connected to the **numContinentsVisited** property:

1. In the **HelloPropertyPaneWebPart.ts**, at the top of the file, add a **PropertyPaneSlider** reference to the existing import statement for the **@microsoft/sp-webpart-base** package.

```
typescript Import { PropertyPaneSlider } from '@microsoft/sp-webpart-base'
```

2. Scroll down to the method **getPropertyPaneConfiguration** and within it, locate the **groupFields** array.
3. Add a comma after the last **PropertyPaneTextField()** call, and add the following code:  

```
typescript PropertyPaneSlider('numContinentsVisited', { label: 'Number of continents I\'ve visited', min: 1, max: 7, showValue: true, })
```
4. Go back to the browser. Notice the property pane now has a slider control to control the number of continents you have visited.
6. In the command prompt, press **CTRL+C** to stop the local web server.
7. Close Visual Studio Code.

## Review

In this exercise, you learned how to change custom properties that exist in the web part manifest and how to add new properties.

◆◆◆# Exercise 3: Creating SharePoint Framework Extensions In this exercise, you will create a SPFx Application Customizer that will launch a Hello dialog on the load of the SharePoint modern page.

## Task 1: Create your project

1. From the PowerShell command prompt, change to the C:/LabFiles/SharePoint directory by executing the following command: `cd c:/LabFiles/SharePoint`
  2. Make a new directory for your SharePoint project files by executing the following command: `md SPFxAppCustomizer`
  3. Navigate to the newly created SharePoint directory by executing the following command: `cd SPFxAppCustomizer`
  4. Run the SharePoint Yeoman generator by executing the following command: `yo @microsoft/sharepoint`
  5. Use the following to complete the prompt that is displayed:
    - **What is your solution name?:** SPFxAppCustomizer
    - **Which baseline packages do you want to target for your component(s)?:** SharePoint Online only (latest)
    - **Where do you want to place the files?:** Use the current folder
    - **Do you want to allow the tenant admin the choice of being able to deploy the solution to all sites immediately without running any feature deployment or adding apps in sites?:** No
    - **Will the components in the solution require permissions to access web APIs that are unique and not shared with other components in the tenant?:** No
    - **Which type of client-side component to create?:** Extension
    - **What type of client-side extension to create?:** Application Customizer
    - **What is your Application Customizer name?:** HelloAppCustomizer
    - **What is your Application Customizer description?:** HelloAppCustomizer description
  6. After provisioning the folders required for the project, the generator will install all the dependency packages using NPM.
  7. Open the project in Visual Studio Code by executing the following command: `code .`
  8. From the Visual Studio Code **Terminal** prompt, execute the following command: `gulp trust-dev-cert`
- NOTE:** Extensions must be tested in a modern SharePoint page unlike web parts, which can be tested in the local workbench. In addition, extensions also require special URL parameters when requesting the page to load the extension from the local development web server.
9. Obtain the URL of a modern SharePoint page.
  10. Open the `./config/serve.json` file.

11. Copy the URL of your modern SharePoint page into the **serveConfigurations.default.pageUrl** property.
12. The SPFx build process **gulp serve** task will launch a browser and navigate to this URL, appending the necessary URL parameters to the end of the URL to load the SPFx extension from your local development web server.
  1. Run the project by executing the following command: `gulp serve`
13. When the SharePoint page loads, SharePoint will prompt you to load the debug scripts. This is a confirmation check to ensure you really want to load scripts from an untrusted source. In this case, that is your local development web server on `https://localhost`, which you can trust.
14. Select the button **Load debug scripts**.
15. Once the scripts load, a SharePoint dialog alert box will be shown: This alert box is shown by the application customizer. Open the application customizer file located at `/src/extensions/helloAppCustomizer/HelloAppCustomizerApplicationCustomizer.ts` and find the **OnInit()** method. Notice the following line in the method that is triggering the dialog to appear:

```
Dialog.alert(`Hello from  
${strings.Title}:\n\n${message}`);
```
16. Stop the local web server by pressing **CTRL+C** in the console/terminal window.

## Task 2: Test your extension

1. Run the project by executing the following command: `gulp serve`
2. When prompted, select the **Load debug scripts** button.
3. Notice when the page loads, a Hello (your name) dialog should appear.
4. Stop the local web server by pressing CTRL+C in the console/terminal window.

## Task 3: Deploying your extension to all sites in the SharePoint Online tenant

1. Locate and open the `./config/package-solution.json` file.
  1. Ensure the solution object has a property named **skipFeatureDeployment** and ensure that the value of this property is set to **true**.
2. Locate and open the `./sharepoint/assets/ClientSideInstance.xml` file. This file contains the values that will be automatically set on the Tenant Wide Extensions list in your SharePoint Online tenant's App Catalog site when the package is deployed.
3. Build and package the solution by running the following commands, one at a time:

```
powershell gulp build gulp bundle --ship gulp package-solution --ship  
NOTE: If this error The build failed because a task wrote output to stderr. is displayed on the command console, please ignore it. The reason is that the build output contain a warning.
```

4. In the browser, navigate to your SharePoint Online tenant App Catalog site.
  5. Select the Apps for SharePoint list in the left navigation pane.
  6. Drag the generated `./sharepoint/solution/*.sppkg` file into the Apps for SharePoint list.
  7. In the **Do you trust spfx-app-customizer-client-side-solution?** dialog box:
    1. Select the **Make this solution available to all sites in the organization** check box.
    2. Notice the message **This package contains an extension which will be automatically enabled across sites....**
    3. Select **Deploy**.
  8. Select **Site contents** in the left navigation pane.
  9. Select **Tenant Wide Extensions**. Depending on when your tenant was created the Tenant Wide Extensions list may be hidden. If you do not see the list in the Site Contents then you will have to navigate to it manually. Do this by appending `/Lists/TenantWideExtensions/AllItems.aspx` to the URL of the app catalog site.
  10. In a separate browser window, navigate to any modern page in any modern site within your SharePoint Online tenant. If this extension has been deployed successfully you should see a Hello (your name) dialog prompt on the load of the page.
- NOTE:** It may take up to 20 minutes for a Tenant Wide Extension to get deployed across the SharePoint Online tenant so you may need to wait to fully test whether your deployment was successful.
11. Stop the local web server by pressing CTRL+C in the console/terminal window.

## **Review**

In this exercise, you learned how to create and deploy a custom header and footer extension that is applied to all sites in a SharePoint Online tenant.

◆◆◆# Exercise 4: Creating a command set extension

## Task 1: Create your project

1. From the PowerShell command prompt, change to the C:/LabFiles/SharePoint directory by executing the following command: `cd c:/LabFiles/SharePoint`
2. Make a new directory for your SharePoint project files by executing the following command: `md SPFxCommandSet`
3. Navigate to the newly created SharePoint directory by executing the following command: `cd SPFxCommandSet`
4. Run the SharePoint Yeoman generator by executing the following command: `yo @microsoft/sharepoint`
5. Use the following to complete the prompt that is displayed:
  - **What is your solution name?:** SPFxCommandSet
  - **Which baseline packages do you want to target for your component(s)?:** SharePoint Online only (latest)
  - **Where do you want to place the files?:** Use the current folder
  - **Do you want to allow the tenant admin the choice of being able to deploy the solution to all sites immediately without running any feature deployment or adding apps in sites?:** No
  - **Will the components in the solution require permissions to access web APIs that are unique and not shared with other components in the tenant?:** No
  - **Which type of client-side component to create?:** Extension
  - **What type of client-side extension to create?:** ListView Command Set
  - **What is your Command Set name?:** CommandSetDemo
  - **What is your Command Set description?:** CommandSetDemo description
6. After provisioning the folders required for the project, the generator will install all the dependency packages using NPM.
7. From the PowerShell command prompt execute the following command: `gulp trust-dev-cert`
8. Open the project in Visual Studio Code by executing the following command: `code .`

## Task 2: Observe the Code of your ListView Command Set

1. From Visual Studio Code, open the **CommandSetDemoCommandSet.ts** file in the **src\extensions\commandSetDemo** folder.
2. Notice that the base class for the ListView Command Set is imported from the **sp-listview-extensibility** package, which contains SharePoint Framework code required by the ListView Command Set.

```
typescript import { override } from '@microsoft/decorators'; import { Log }  
from '@microsoft/sp-core-library'; import { BaseListViewCommandSet,  
Command, IListViewCommandSetListViewUpdatedParameters,  
IListViewCommandSetExecuteEventParameters } from '@microsoft/sp-listview-  
extensibility'; import { Dialog } from '@microsoft/sp-dialog';
```

The behavior for your custom buttons is contained in the **onListViewUpdated()** and **OnExecute()** methods.

The **onListViewUpdated()** event occurs separately for each command (for example, a menu item) whenever a change happens in the ListView, and the UI needs to be re-rendered. The event function parameter represents information about the command being rendered. The handler can use this information to customize the title or adjust the visibility, for example, if a command should only be shown when a certain number of items are selected in the list view. This is the default implementation. When using the method **tryGetCommand**, you get a Command object, which is a representation of the command that shows in the UI. You can modify its values, such as **title**, or **visible**, to modify the UI element. SPFx uses this information when re-rendering the commands. These objects keep the state from the last render, so if a command is set to **visible = false**, it remains invisible until it is set back to **visible = true**.

```
typescript @override public onListViewUpdated(event:  
IListViewCommandSetListViewUpdatedParameters): void { const compareOneCommand:  
Command = this.tryGetCommand('COMMAND_1'); if (compareOneCommand) { // This  
command should be hidden unless exactly one row is selected.  
compareOneCommand.visible = event.selectedRows.length === 1; } }
```

The **OnExecute()** method defines what happens when a command is executed (for example, the menu item is selected). In the default implementation, different messages are shown based on which button was selected.

```
typescript @override public onExecute(event:  
IListViewCommandSetExecuteEventParameters): void { switch (event.itemId) { case  
'COMMAND_1': Dialog.alert(` ${this.properties.sampleTextOne}`); break; case  
'COMMAND_2': Dialog.alert(` ${this.properties.sampleTextTwo}`); break; default:  
throw new Error('Unknown command'); } }
```

## Task 3: Test your extension

You cannot currently use the local Workbench to test SharePoint Framework Extensions. You'll need to test and develop them directly against a live SharePoint Online site. You don't have to deploy your customization to the app catalog to do this, which makes the debugging experience simple and efficient. Go to any SharePoint list in your SharePoint Online site by using the modern experience or create a new list. Copy the URL off the list to clipboard. Because our ListView Command Set is hosted from localhost and is running, you can use specific debug query parameters to execute the code in the list view. Open **serve.json** file from **config** folder. Update the **pageUrl** attributes to match a URL of the list where you want to test the solution. After edits your serve.json should look somewhat like:

```
powershell { "$schema": "https://developer.microsoft.com/json-schemas/core-build/serve.schema.json", "port": 4321, "https": true, "serveConfigurations": { "default": { "pageUrl": "https://contoso.sharepoint.com/sites/Group/Lists/Orders/AllItems.aspx", "customActions": { "bf232d1d-279c-465e-a6e4-359cb4957377": { "location": "ClientSideExtension.ListViewCommandSet.CommandBar", "properties": { "sampleTextOne": "One item is selected in the list", "sampleTextTwo": "This command is always visible." } } }, "helloWorld": { "pageUrl": "https://sppnp.sharepoint.com/sites/Group/Lists/Orders/AllItems.aspx", "customActions": { "bf232d1d-279c-465e-a6e4-359cb4957377": { "location": "ClientSideExtension.ListViewCommandSet.CommandBar", "properties": { "sampleTextOne": "One item is selected in the list", "sampleTextTwo": "This command is always visible." } } } } }
```

1. From the Visual Studio Code ribbon, select **Terminal > New Terminal**.
2. Run the project by executing the following command: `gulp serve`
3. When prompted, select the **Load debug scripts** button.
4. Stop the local web server by pressing **CTRL+C** in the console/terminal window.

## Review

In this exercise, you learned how to create and deploy a command set extension. Command set extensions are used for adding additional commands into the command bar at the top of a page.

◆◆◆# Exercise 5: Creating a field customizer extension

## Task 1: Create your project

1. From the PowerShell command prompt, change to the C:/LabFiles/SharePoint directory by executing the following command: `cd c:/LabFiles/SharePoint`
2. Make a new directory for your SharePoint project files by executing the following command: `md SPFxFieldCustomizer`
3. Navigate to the newly created SharePoint directory by executing the following command: `cd SPFxFieldCustomizer`
4. Run the SharePoint Yeoman generator by executing the following command: `yo @microsoft/sharepoint`
5. Use the following to complete the prompt that is displayed:
  - **What is your solution name?:** SPFxFieldCustomizer
  - **Which baseline packages do you want to target for your component(s)?:** SharePoint Online only (latest)
  - **Where do you want to place the files?:** Use the current folder
  - **Do you want to allow the tenant admin the choice of being able to deploy the solution to all sites immediately without running any feature deployment or adding apps in sites?:** No
  - **Will the components in the solution require permissions to access web APIs that are unique and not shared with other components in the tenant?:** No
  - **Which type of client-side component to create?:** Extension
  - **What type of client-side extension to create?:** Field Customizer
  - **What is your Field Customizer name?:** HelloFieldCustomizer
  - **What is your Field Customizer description?:** HelloFieldCustomizer description
  - **Which framework would you like to use?:** No JavaScript Framework
6. After provisioning the folders required for the project, the generator will install all the dependency packages using NPM.

## Task 2: Updating the SCSS styles for the field customizer

1. Locate and open the  
**./src/extensions/helloFieldCustomizer>HelloFieldCustomizerFieldCustomizer.module.scss** file.
2. Replace the contents of the file with the following styles:

```
html .HelloFieldCustomizer { .cell { display: 'inline-block'; }  
.filledBackground { background-color: #cccccc; width: 100px; } }
```

## Task 3: Update the code for the field customizer

1. Locate and open the **./src/extensions/helloFieldCustomizer/HelloFieldCustomizerFieldCustomizer.ts** file.
2. Locate the interface **IHelloFieldCustomizerFieldCustomizerProperties** and update its properties to the following: `greenMinLimit?: string; yellowMinLimit?: string;`
3. Locate the method **onRenderCell()** and update the contents to match the following code. This code looks at the existing value in the field and builds the relevant colored bars based on the value entered in the completed percentage value.

```
typescript event.documentElement.classList.add(styles.cell);
event.documentElement.innerHTML = ` <div class='${styles.HelloFieldCustomizer}'>
<div class='${styles.filledBackground}'> <div style='width:
${event.fieldValue}px; background:#0094ff; color:#c0c0c0'>
${event.fieldValue} </div> </div> </div>`;
```

## Task 4: Update the deployment code for the field customizer

Field customizers, when deployed to production, are implemented by creating a new site column who's rendering is tied to the custom script defined in the field customizer's bundle file.

1. Locate and open the **./sharepoint/assets/elements.xml** file.
2. Add the following property to the element, setting the values on the public properties on the field customizer:

```
xml ClientSideComponentProperties=""
{"greenMinLimit":"85","yellowMinLimit":"70"}"
```

## Task 5: Testing your field customizer

1. In a browser, navigate to a SharePoint Online modern site collection where you want to test the field customizer.
2. Select the **Site contents** link in the left navigation pane.
3. Create a new SharePoint list:
  1. Select **New > List** in the toolbar.
  2. Set the list name to **Work Status** and select **Create**.
4. When the list loads, select the **Add column > Number** to create a new column.
5. When prompted for the name of the column, enter **PercentComplete**.
6. Add a few items to the list, with different numbers in the percent field.
7. Update the properties for the serve configuration used to test and debug the extension:
  1. Locate and open the **./config/serve.json** file.
  2. Copy in the full URL (including **AllItems.aspx**) of the list you just created into the **serveConfigurations.default.pageUrl** property.
  3. Locate the **serveConfigurations.default.properties** object.
  4. Change the name of the property **serveConfigurations.default.fieldCustomizers.InternalFieldName** to **serveConfigurations.default.fieldCustomizers.PercentComplete**. This tells the SharePoint Framework which existing field to associate the field customizer with.
  5. Change the value of the properties object to the following: "properties": { "greenMinLimit": "85", "yellowMinLimit": "70" }
  6. The JSON code for the default serve configuration should look something like the following:

```
json "default": { "pageUrl": "https://contoso.sharepoint.com/sites/mySite/Lists/Work%20Status/AllItems.aspx", "fieldCustomizers": { "PercentComplete": { "id": "6a1b8997-00d5-4bc7-a472-41d6ac27cd83", "properties": { "greenMinLimit": "85", "yellowMinLimit": "70" } } } }
```

1. Run the project by executing the following command: `gulp serve`
  8. When prompted, select the **Load debug scripts** button.
  9. Stop the local web server by pressing **CTRL+C** in the console/terminal window.

## Review

In this exercise, you learned how to create and deploy a command set extension. Command set extensions are used for adding additional commands into the command bar at the top of a page.

◆◆◆# Exercise 6: Deploying a SharePoint Framework solution

## Task 1: Create your project

1. From the PowerShell command prompt, change to the C:/LabFiles/SharePoint directory by executing the following command: `cd c:/LabFiles/SharePoint`
2. Make a new directory for your SharePoint project files by executing the following command: `md DeploymentDemo`
3. Navigate to the newly created SharePoint directory by executing the following command: `cd DeploymentDemo`
4. Run the SharePoint Yeoman generator by executing the following command: `yo @microsoft/sharepoint`
5. Use the following to complete the prompt that is displayed:
  - **What is your solution name?:** DeploymentDemo
  - **Which baseline packages do you want to target for your component(s)?:** SharePoint Online only (latest)
  - **Where do you want to place the files?:** Use the current folder
  - **Do you want to allow the tenant admin the choice of being able to deploy the solution to all sites immediately without running any feature deployment or adding apps in sites?:** No
  - **Will the components in the solution require permissions to access web APIs that are unique and not shared with other components in the tenant?:** No
  - **Which type of client-side component to create?:** WebPart
  - **What is your Web part name?:** Deployment Demo
  - **What is your Web part description?:** Deployment Demo description
  - **Which framework would you like to use?:** No JavaScript framework
6. After provisioning the folders required for the project, the generator will install all the dependency packages using NPM.

## **Task 2: Creating a deployment package for the project**

1. When NPM completes downloading all dependencies, build the project by running the following command on the command line from the root of the project: `gulp build`
2. Create a production bundle of the project by running the following command on the command line from the root of the project: `gulp bundle --ship`
3. Create a deployment package of the project by running the following command on the command line from the root of the project: `gulp package-solution --ship`

## Task 3: Deploying the package to a SharePoint site

1. In a browser, navigate to your SharePoint tenant's App Catalog site.
2. Select the **Apps for SharePoint** link in the left navigation pane.
3. Drag the package created in the previous steps, located in the project's **./sharepoint/solution/deployment-demo.sppkg**, into the **Apps for SharePoint** library.
4. SharePoint will launch a dialog box asking if you want to trust the package.
5. Select **Deploy**.

## **Task 4: Installing the SharePoint package in a site collection**

1. Navigate to an existing site collection, or create a new one.
2. Select **Site Contents** from the left navigation pane.
3. From the **New** menu, select **App**.
4. Locate the solution you previously deployed and select it.

SharePoint will start to install the application. At first it will appear dimmed, but after a few moments you should see it listed.

## **Task 5: Adding the web part to a page**

1. Navigate to a SharePoint page.
2. Put it in edit mode by selecting the **Edit** button in the upper-right portion of the content area on the page.
3. Select the web part icon button to open the list of available web parts.
4. Select the expand icon, a diagonal line with two arrows in the upper-right corner, to expand the web part toolbox.
5. Scroll to the bottom; locate and select the **Deployment Demo** web part.

## **Task 6: Examining the deployed web part files**

1. Once the page loads, open the browser's developer tools and navigate to the **Sources** tab.
2. Refresh the page and examine where the JavaScript bundle is being hosted.

**Note:** If you have not enabled the Office 365 CDN then the bundle will be hosted from a document library named ClientSideAssets in the app catalog site. If you have enabled the Office 365 CDN then the bundle will be automatically hosted from the CDN.

## **Review**

In this exercise, you learned how to package and deploy your SharePoint Framework solutions.

◆◆◆# Exercise 7: Deploying SPFx solutions to Microsoft Teams

## Task 1: Create your project

1. From the PowerShell command prompt, change to the C:/LabFiles/SharePoint directory by executing the following command: `cd c:/LabFiles/SharePoint`
2. Make a new directory for your SharePoint project files by executing the following command: `md SPFxTeamsTab`
3. Navigate to the newly created SharePoint directory by executing the following command: `cd SPFxTeamsTab`
4. Run the SharePoint Yeoman generator by executing the following command: `yo @microsoft/sharepoint`
5. Use the following to complete the prompt that is displayed:
  - **What is your solution name?:** SPFxTeamsTab
  - **Which baseline packages do you want to target for your component(s)?:** SharePoint Online only (latest)
  - **Where do you want to place the files?:** Use the current folder
  - **Do you want to allow the tenant admin the choice of being able to deploy the solution to all sites immediately without running any feature deployment or adding apps in sites?:** Yes
  - **Will the components in the solution require permissions to access web APIs that are unique and not shared with other components in the tenant?:** No
  - **Which type of client-side component to create?:** WebPart
  - **What is your Web Part name?:** SPFx Teams Together
  - **What is your Web Part description?:** SPFx Teams Together description
  - **Which framework would you like to use?:** No JavaScript framework
6. After provisioning the folders required for the project, the generator will install all the dependency packages using NPM.
7. Open the project in an editor such as Visual Studio Code.
8. Enable the web part to be used in Microsoft Teams:
  1. Locate and open the file  
`/src/webparts/spFxTeamsTogether/SpFxTeamsTogetherWebPart.manifest.json`  
.
  2. Within the web part manifest file, locate the property **supportedHosts**:  
`"supportedHosts": ["SharePointWebPart"],`
  3. Add another option to enable this web part to be used as a tab in a Microsoft Teams team: `"supportedHosts": ["SharePointWebPart", "TeamsTab"],`

## Task 2: Creating and deploying the Microsoft Teams app package

Notice the project contains a folder **teams** that contains two images. These are used in Microsoft Teams to display the custom tab.

You may notice there is no **manifest.json** file present. The manifest file can be generated automatically by SharePoint from the App Catalog site. However, at this time, this functionality is not fully operational.

Alternatively, you can manually create the manifest file if you want to have more control over the default values that SharePoint will create.

The automatic generation and deployment of the Microsoft Teams manifest is not currently operational; you will manually create the Microsoft Teams manifest and app package:

1. Create a **manifest.json** file in **./teams** folder, use the following manifest file as a template:

```
json { "$schema": "https://developer.microsoft.com/json-schemas/teams/v1.5/MicrosoftTeams.schema.json", "manifestVersion": "1.5", "packageName": "{{SPFX_COMPONENT_ALIAS}}", "id": "{{SPFX_COMPONENT_ID}}", "version": "0.1", "developer": { "name": "Parker Porcupine", "websiteUrl": "https://contoso.com", "privacyUrl": "https://contoso.com/privacystatement", "termsOfUseUrl": "https://contoso.com/servicesagreement" }, "name": { "short": "{{SPFX_COMPONENT_NAME}}", "description": { "short": "{{SPFX_COMPONENT_SHORT_DESCRIPTION}}", "full": "{{SPFX_COMPONENT_LONG_DESCRIPTION}}" } }, "icons": { "outline": "{{SPFX_COMPONENT_ID}}_outline.png", "color": "{{SPFX_COMPONENT_ID}}_color.png", "accentColor": "#004578", "staticTabs": [ { "entityId": "com.contoso.personaltab.spfx", "name": "My SPFx Personal Tab", "contentUrl": "https://{teamSiteDomain}/_layouts/15/TeamsLogon.aspx?SPFx=true&dest=/_layouts/15/teamshostedapp.aspx%3Fteams%26personal%26componentId={{SPFX_COMPONENT_ID}}%26forceLocale={locale}", "scopes": [ "personal" ] }, "configurableTabs": [ { "configurationUrl": "https://{teamSiteDomain}{teamSitePath}/_layouts/15/TeamsLogon.aspx?SPFx=true&dest={teamSitePath}/_layouts/15/teamshostedapp.aspx%3FopenPropertyPane=true%26team%26componentId={{SPFX_COMPONENT_ID}}%26forceLocale={locale}" }, "canUpdateConfiguration": true, "scopes": [ "team" ] ], "validDomains": [ "*.login.microsoftonline.com", "*.sharepoint.com", "spoprod-a.akamaihd.net", "resourceseng.blob.core.windows.net" ], "webApplicationInfo": { "resource": "https://{teamSiteDomain}", "id": "00000003-0000-0000-0000-000000000000" } }
```

2. Open the **manifest.json** file. This file contains multiple strings that need to be updated to match the SPFx component. Use the following table to determine the values that should be replaced. The SPFx component properties are found in the web part manifest file:

**./src/webparts/spFxTeamsTogether/SpFxTeamsTogetherWebPart.manifest.json**

### manifest.json string

```
{ {{SPFX_COMPONENT_ALIAS}} }  
{ {{SPFX_COMPONENT_NAME}} }  
{ {{SPFX_COMPONENT_SHORT_DESCRIPTION}} }  
{ {{SPFX_COMPONENT_LONG_DESCRIPTION}} }  
{ {{SPFX_COMPONENT_ID}} }
```

### Property in SPFx component manifest

alias	alias
preconfiguredEntries[0].title	preconfiguredEntries[0].title
preconfiguredEntries[0].description	preconfiguredEntries[0].description
id	id

**Note:** Don't miss replacing `{{SPFX_COMPONENT_ID}}` in `configurableTabs[0].configurationUrl`. You will likely have to scroll your editor to the right to see it. The tokens surrounded by single curly braces (for example, `{teamSiteDomain}`) do not need to be replaced.

3. Create a Microsoft Teams app package by zipping the contents of the `./teams` folder. Make sure to zip just the contents and not the folder itself. This ZIP archive should contain three files at the root: two images and the **manifest.json**.

## Task 3: Create and deploy the SharePoint package

In order to test the web part in SharePoint and Microsoft Teams, it must first be deployed to an app catalog.

1. Open a browser and navigate to your SharePoint Online Tenant-SScoped App Catalog site.
2. Select the menu item **Apps for SharePoint** from the left navigation menu.
3. Build the project by opening a command prompt and changing to the root folder of the project. Then execute the following command: `gulp build`
4. Next, create a production bundle of the project by running the following command on the command line from the root of the project: `gulp bundle --ship`
5. Create a deployment package of the project by running the following command on the command line from the root of the project: `gulp package-solution --ship`
6. Locate the file created by the gulp task, found in the **./sharepoint/solution** folder with the name \*.sppkg.
7. Drag this file into **the Apps for SharePoint** library in the browser.
8. In the **Do you trust...?** dialog box, select the check box **Make this solution available to all sites in the organization** and then select **Deploy**.
9. This will make the SPFx web part available to all site collections in the tenant, including those that are behind a Microsoft Teams team.

## **Task 4: Test the web part in SharePoint and Microsoft Teams**

Test the SPFx web part in SharePoint:

1. In the browser, navigate to a modern SharePoint page.
2. Select the **Edit** button in the upper-right portion of the page.
3. Select the (+) icon to open the SharePoint web part toolbox and locate the web part SPFx Teams Together:
4. The SharePoint Framework web part will be displayed on the page.

## Task 5: Testing the SPFx web part in Microsoft Teams

1. Create a new Microsoft Teams team.
  1. Using the same browser where you are signed in to SharePoint Online, navigate to <https://teams.microsoft.com>. When prompted, load the web client.
  2. If you do not have any teams in your tenant, you will be presented with the following dialog. Otherwise, select **Join or create a team** at the bottom of the list of teams.
  3. On the **Create your team** dialog box, select **Build a team from scratch**.
  4. On the **What kind of team will this be?** dialog box, select **Public**.
  5. When prompted, use the name **My First Team**.
2. Install the Microsoft Teams application as a new tab that will expose the SharePoint Framework web part in Microsoft Teams:
  1. Select the **My First Team** team previously created.
  2. Select the **General** channel.
3. Add a custom tab to the team using the SPFx web part:
  1. At the top of the page, select the + icon in the horizontal navigation.
  2. In the **Add a tab** dialog box, select **More Apps**.
  3. Select the **Upload a custom app > Upload for ...** from the list of app categories.
  4. Select the Microsoft Teams application ZIP file previously created. This is the file that contains the **manifest.json** and two image files.

**Note:** After a moment, the application will appear next to your tenant name. You may need to refresh the page for the app to appear if you are using the browser Microsoft Teams client.
4. Select the **SPFx Teams Together** app.
5. In the **SPFx Teams Together** dialog box, select the **My First Team** in the **Add to a team** drop-down control and select **Install**.
6. In the **SPFx Teams Together is now available for My First Team** dialog box, select the **General** channel and select **Set up**.
7. The next dialog box will confirm the installation of the app. Select **Save**.
8. The application should now load in Microsoft Teams within the **General** channel under the tab **SPFx Teams Together**.

## **Review**

In this exercise, you learned how to create a SharePoint Framework (SPFx) solution that will work in both SharePoint and as a tab in Microsoft Teams.

❖❖❖# Student lab answer key

## Microsoft Azure user interface

Given the dynamic nature of Microsoft cloud tools, you might experience Azure user interface (UI) changes after the development of this training content. These changes might cause the lab instructions and lab steps to not match up.

The Microsoft Worldwide Learning team updates this training course as soon as the community brings needed changes to our attention. However, because cloud updates occur frequently, you might encounter UI changes before this training content is updated. **If this occurs, adapt to the changes and work through them in the labs as needed.**

## **Instructions**

### **Sign in to the lab virtual machine**

Sign in to the Windows 10 virtual machine using credentials.

**Note:** Lab virtual machine sign-in instructions will be provided to students by the instructor.

## Review installed applications

Observe the taskbar located at the bottom of your Windows 10 desktop. The taskbar contains the icons for the applications you will use in this lab:

- Microsoft Edge
- File Explorer
- Visual Studio Code
- Microsoft Teams

Also, ensure that the following necessary utilities are installed:

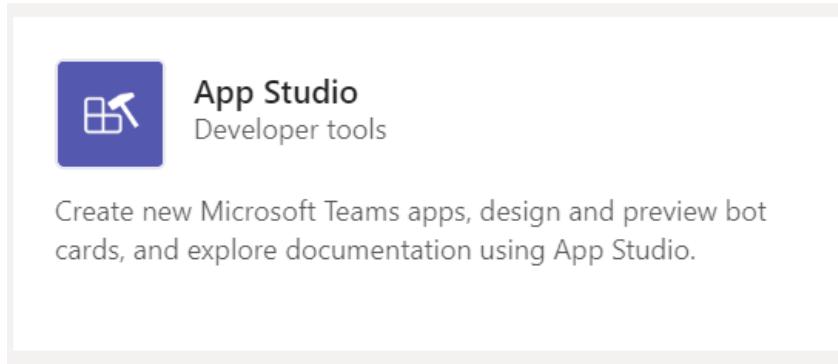
- [.NET Core 3.1 SDK](#)
- Ngrok
- NodeJS v10.x

# Exercise 1: Creating an app manifest in Microsoft Teams

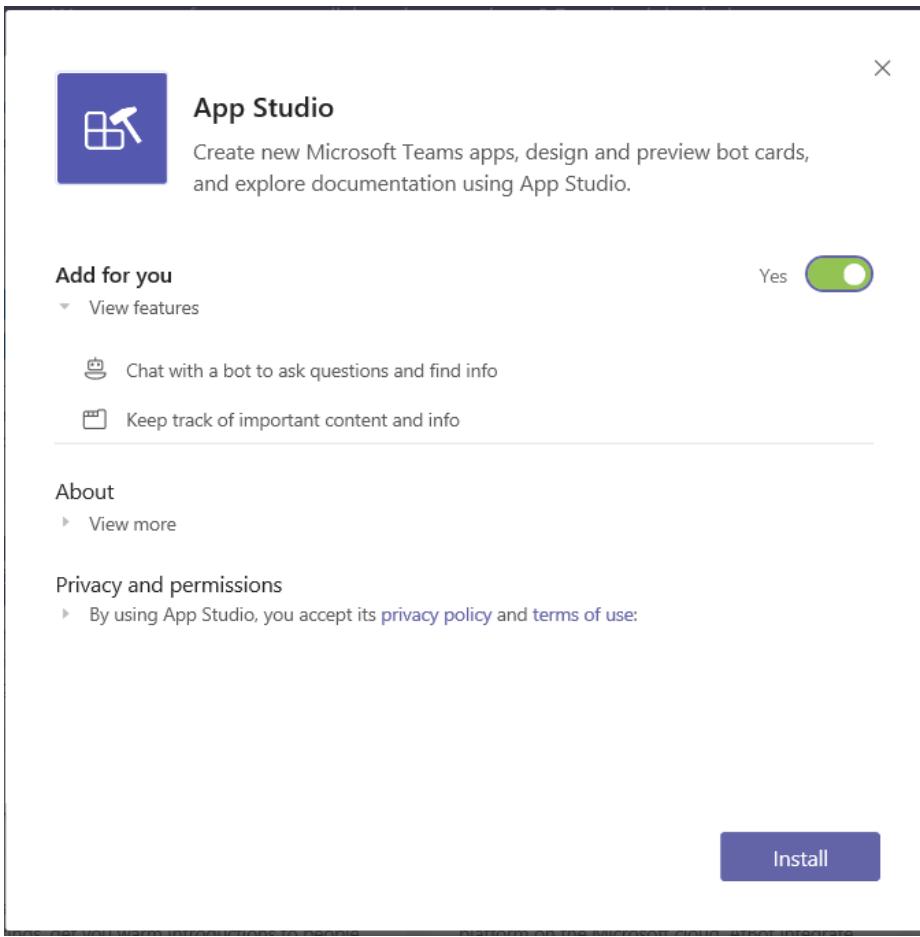
## Task 1: Install App Studio

App Studio is a Teams app which can be found in the Teams store. Follow this link for direct download: [App Studio](#) (you can also find the app in the app store).

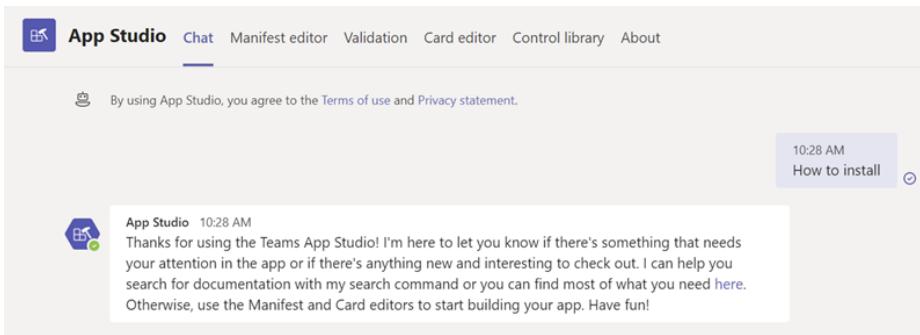
1. Open a browser and navigate to the [Teams web client](#). Sign in using a **Work or School Account** that has global administrator rights to the tenancy.
2. Select **Apps** at the bottom of the left hand bar.
3. In the store, search for **App Studio**.



4. Select the **App Studio** tile to open the app install page:



## 5. Select install.



Once you are in **App Studio**, select the Manifest editor tab where you can either import an existing app or create a new app.

## Task 2: Configure an app manifest using App Studio

### Get sample project files

1. Open PowerShell and navigate to **C:/LabFiles/Teams**
2. To clone the sample project repository, execute the following command:

```
powershell git clone https://github.com/OfficeDev/msteams-samples-hello-world-csharp.git
```

### Build a Teams App Package

1. From the PowerShell command prompt, change to the **C:/LabFiles/Teams/msteams-samples-hello-world-csharp** directory by executing the following command:  

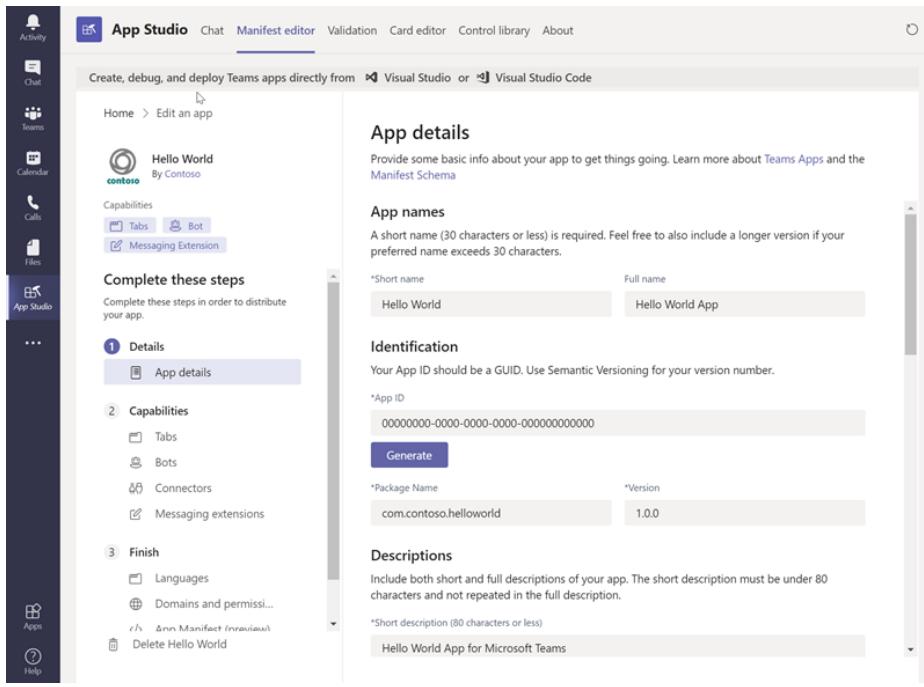
```
powershell cd c:/LabFiles/Teams/msteams-samples-hello-world-csharp
```
2. Open the folder in Visual Studio Code by executing the following from the project directory: `code .`
3. If Visual Studio code displays a dialog box asking if you want to add required assets to the project, select  **Yes**.
4. From the Visual Studio Code ribbon, select **Terminal > New Terminal**.
5. In the terminal, Run the following command to compile the application: `dotnet build`
6. After successfully compiling the solution, the **helloworldapp.zip** Teams App package will be generated in the `.\bin\Debug\netcoreapp3.1\` folder.

### Import the existing package

1. Open a browser and navigate to the [Teams web client](#).
2. In the app bar, select ... **More added apps** and then select **App Studio**.
3. In App Studio, select the **Manifest editor** tab, then select **Import an existing app**. Open the project's `.\bin\Debug\netcoreapp3.1\` folder, and then open the **helloworldapp.zip** file.
4. You should now see your imported app which you can now select to open.

### Update app manifest

Now you can modify your app manifest here in App Studio.



1. On the **App details** page update the following under the **App names** section:
  - o **Short name:** Change to **First App**
  - o **Full name:** Change to **Learn Microsoft Teams App**
2. On the **App details** page, scroll down to the **Descriptions** section and enter the following values:
  - o **Short description:** Enter **My first custom Teams app**
  - o **Long description:** Enter a longer description of your choice.\*\*
3. Change the **Version** to **1.0.1**.
4. Now update the tab name by following the steps below:
  1. In the **App Studio** navigation pane, select **Capabilities > Tabs**.
  2. Locate the only personal tab in the project, and select ... > **Edit** on that tab.
  3. Change the tab name to **My First Tab**.
  4. Select **Save** to save your changes.

## Download and Install app manifest

Once you have the app manifest ready, you will need to download it and then manually upload it to Teams.

1. Download the modified app manifest.

**NOTE:** Any changes you make in App Studio aren't saved to your project. To update the project, download the app package from App Studio. To download the project, in the App

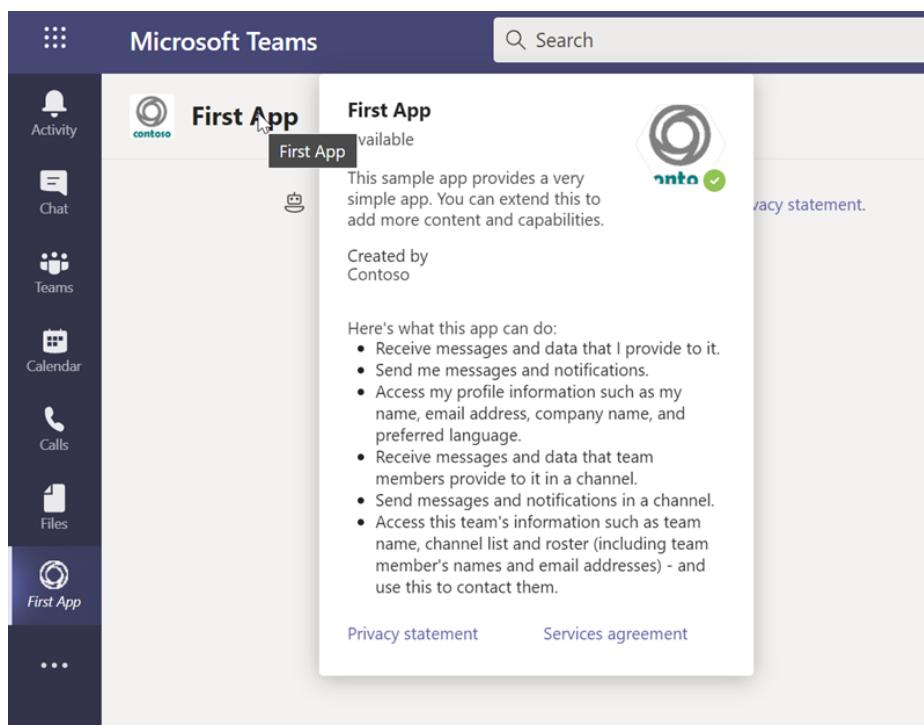
Studio navigation pane select **Finish > Test and distribute**, and then select **Download**.

**WARNING:** Be careful if you choose to update the **manifest.json** file in your project with the one in the package downloaded from App Studio. The manifest file in your project contains placeholder strings that are updated by the build and debugging process.

For instance, the **{{HOSTNAME}}** placeholder is replaced with the app's hosting URL each time the package is recreated. Because of this, it's not recommended to replace the existing **manifest.json** file with the file generated by App Studio.

1. Now upload the file you downloaded to Teams:

1. From the Teams client, select **Apps > Upload a custom app**.
2. Browse and upload the updated zip package. Open the app and notice it's now reflecting the updates you made.



# Task 3: Deploying SPFx web part to Microsoft Teams

## Create SPFx web part

1. From the PowerShell command prompt, change to the C:/LabFiles/SharePoint directory by executing the following command:

```
cd c:/LabFiles/SharePoint
```

2. Make a new directory for your SharePoint project files by executing the following command:

```
md SPFxTeamsTab
```

3. Navigate to the newly created SharePoint directory by executing the following command:

```
cd SPFxTeamsTab
```

4. Run the SharePoint Yeoman generator by executing the following command:

```
yo @microsoft/sharepoint
```

5. Use the following below to complete the prompts that are displayed:

- **What is your solution name?:** SPFxTeamsTab
- **Which baseline packages do you want to target for your component(s)?:** SharePoint Online only (latest)
- **Where do you want to place the files?:** Use the current folder
- **Do you want to allow the tenant admin the choice of being able to deploy the solution to all sites immediately without running any feature deployment or adding apps in sites?:** Yes
- **Will the components in the solution require permissions to access web APIs that are unique and not shared with other components in the tenant?:** No
- **Which type of client-side component to create?:** WebPart
- **What is your Web Part name?:** SPFx Teams Together
- **What is your Web Part description?:** SPFx Teams Together description
- **Which framework would you like to use?:** No JavaScript framework

6. After provisioning the folders required for the project, the generator will install all the dependency packages using NPM.

7. Open the project in Visual Studio Code.

8. Enable the web part to be used in Microsoft Teams:

1. Locate and open the file  
`/src/webparts/spFxTeamsTogether/SpFxTeamsTogetherWebPart.manifest.json`

- Within the web part manifest file, locate the property **supportedHosts**:

```
"supportedHosts": ["SharePointWebPart"],
```

- Add another option to enable this web part to be used as a tab in a Microsoft Teams team:

```
"supportedHosts": ["SharePointWebPart", "TeamsTab"],
```

## Creating and deploying the Microsoft Teams app package

- Create a **manifest.json** file in **./teams** folder and use the following below as the content for the file:

```
json { "$schema": "https://developer.microsoft.com/json-schemas/teams/v1.5/MicrosoftTeams.schema.json", "manifestVersion": "1.5", "packageName": "{{SPFX_COMPONENT_ALIAS}}", "id": "{{SPFX_COMPONENT_ID}}", "version": "0.1", "developer": { "name": "Parker Porcupine", "websiteUrl": "https://contoso.com", "privacyUrl": "https://contoso.com/privacystatement", "termsOfUseUrl": "https://contoso.com/servicesagreement" }, "name": { "short": "{{SPFX_COMPONENT_NAME}}", "description": { "short": "{{SPFX_COMPONENT_SHORT_DESCRIPTION}}", "full": "{{SPFX_COMPONENT_LONG_DESCRIPTION}}" }, "icons": { "outline": "{{SPFX_COMPONENT_ID}}_outline.png", "color": "{{SPFX_COMPONENT_ID}}_color.png", "accentColor": "#004578", "staticTabs": [ { "entityId": "com.contoso.personaltab.spfx", "name": "My SPFx Personal Tab", "contentUrl": "https://{teamSiteDomain}/_layouts/15/TeamsLogon.aspx?SPFx=true&dest=/_layouts/15/teamshostedapp.aspx%3Fteams%26personal%26componentId={{SPFX_COMPONENT_ID}}%26forceLocale={locale}", "scopes": [ "personal" ] }, "configurableTabs": [ { "configurationUrl": "https://{teamSiteDomain}{teamSitePath}/_layouts/15/TeamsLogon.aspx?SPFx=true&dest={teamSitePath}/_layouts/15/teamshostedapp.aspx%3FopenPropertyPane=true%26teams%26componentId={{SPFX_COMPONENT_ID}}%26forceLocale={locale}" }, "canUpdateConfiguration": true, "scopes": [ "team" ] ], "validDomains": [ "*.login.microsoftonline.com", "*.sharepoint.com", "spoprod-a.akamaihd.net", "resourceseng.blob.core.windows.net" ], "webApplicationInfo": { "resource": "https://{teamSiteDomain}", "id": "00000003-0000-0000-0000-000000000000" } }
```

- Open the **manifest.json** file.

This file contains multiple strings that need to be updated to match the SPFx component.

The SPFx component properties are found in the web part manifest file:

**./src/webparts/spFxTeamsTogether/SpFxTeamsTogetherWebPart.manifest.json**

Use the following table below to determine the values that should be replaced:

manifest.json string	Property in SPFx component manifest
{{SPFX_COMPONENT_ALIAS}}	alias
{{SPFX_COMPONENT_NAME}}	preconfiguredEntries[0].title
{{SPFX_COMPONENT_SHORT_DESCRIPTION}}	preconfiguredEntries[0].description
{{SPFX_COMPONENT_LONG_DESCRIPTION}}	preconfiguredEntries[0].description

<b>manifest.json</b> string	<b>Property in SPFx component manifest</b>
{ <b>SPFX_COMPONENT_ID</b> }	<b>id</b>

**NOTE:** Make sure you update {{**SPFX\_COMPONENT\_ID**}} in **configurableTabs[0].configurationUrl**. You will likely have to scroll your editor to the right to see it. The tokens surrounded by single curly braces (for example, **teamSiteDomain**) do not need to be replaced.

3. Create a Microsoft Teams app package by zipping the contents of the **./teams** folder.

Make sure to zip just the contents and not the folder itself. This ZIP archive should contain 3 files at the root (2 images and the **manifest.json**)

## Create and deploy the SharePoint package

1. Open the browser and navigate to your SharePoint Online Tenant-Scoped App Catalog site.
2. Select the menu item **Apps for SharePoint** from the left navigation menu.
3. Build the project by opening a command prompt and changing to the root folder of the project. Then execute the following command:

```
gulp build
```

4. Next, create a production bundle of the project by running the following command on the command line from the root of the project:

```
gulp bundle --ship
```

5. Create a deployment package of the project by running the following command on the command line from the root of the project:

```
gulp package-solution --ship
```

6. Locate the file created by the gulp task, found in the **./sharepoint/solution** folder with the name \*.sppkg.

1. Drag this file into **the Apps for SharePoint** library in the browser.

2. In the **Do you trust...?** dialog box, select the check box **Make this solution available to all sites in the organization** and then select **Deploy**.

This will make the SPFx web part available to all site collections in the tenant, including those that are behind a Microsoft Teams team.

## Testing the SPFx web part in Microsoft Teams

1. Create a new Microsoft Teams team.
1. Using the same browser where you are signed in to SharePoint Online, navigate to <https://teams.microsoft.com>. When prompted, load the web client.

2. If you do not have any teams in your tenant, you will be presented with a dialog to create a team. If you are not prompted with a dialog, select **Join or create a team** at the bottom of the list of teams.
  1. On the **Create your team** dialog box, select **Build a team from scratch**.
  2. On the **What kind of team will this be?** dialog box, select **Public**.
  3. When prompted, use the name **My First Team**.
2. Install the Microsoft Teams application as a new tab that will expose the SharePoint Framework web part in Microsoft Teams:
  1. Select the **My First Team** team previously created.
  2. Select the **General** channel.
3. Add a custom tab to the team using the SPFx web part:
  1. At the top of the page, select the + icon in the horizontal navigation.
  2. In the **Add a tab** dialog box, select **More Apps**.
  3. Select the **Upload a custom app > Upload for ...** from the list of app categories.
  4. Select the Microsoft Teams application ZIP file previously created. This is the file that contains the **manifest.json** and two image files.
- NOTE:** After a moment, the application will appear next to your tenant name. You may need to refresh the page for the app to appear if you are using the browser version of Microsoft Teams.
4. Select the **SPFx Teams Together** app.
5. In the **SPFx Teams Together** dialog box, select **Add to a team**.
6. In the **Select a channel to start using SPFx Teams Together** dialog box, make sure that the **General** channel is selected and select **Set up a tab**.
7. The next dialog box will confirm the installation of the app. Select **Save**.
8. The application should now load in Microsoft Teams within the **General** channel under the tab **SPFx Teams Together**.

## **Review**

In this exercise, you:

- Installed App Studio app in Teams.
- Configured and updated app manifest.
- Deployed a SPFx web part to Microsoft Teams.

# Exercise 2: Deploying a Microsoft Teams app

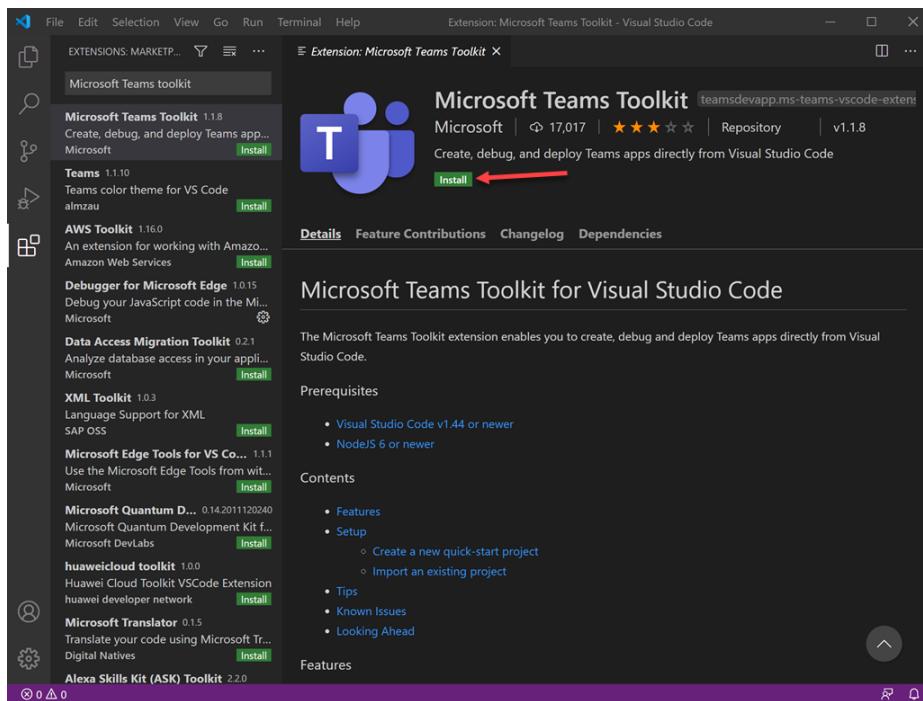
## Task 1: Setup Environment for Teams development

### Install your development tools

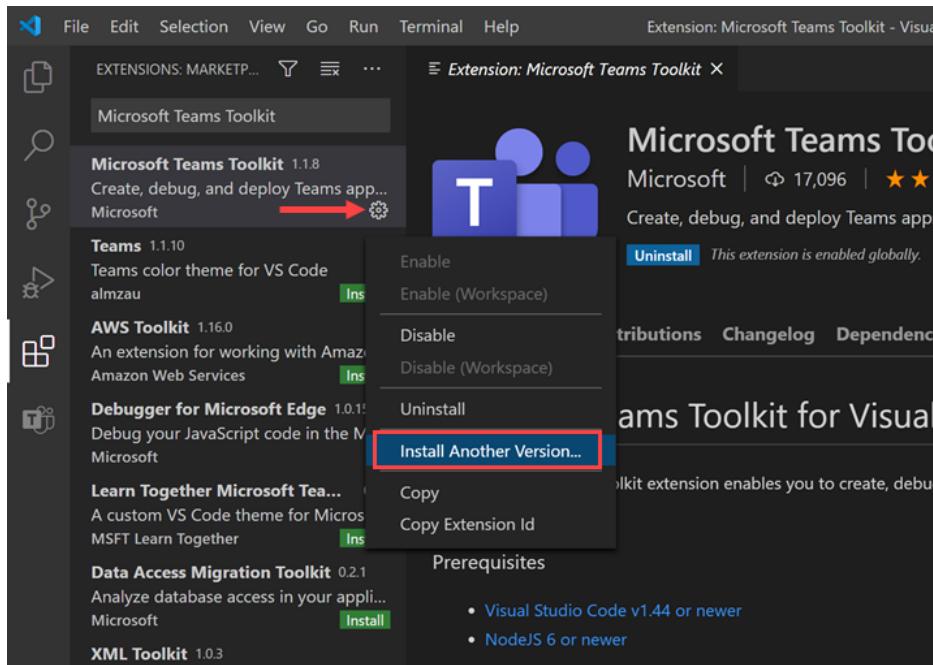
These lessons show how you can get started quickly with the Microsoft Teams Toolkit for Visual Studio Code.

**NOTE:** Due to a bug in the latest Microsoft Teams Toolkit for bot development, you will need to install version 1.1.2.

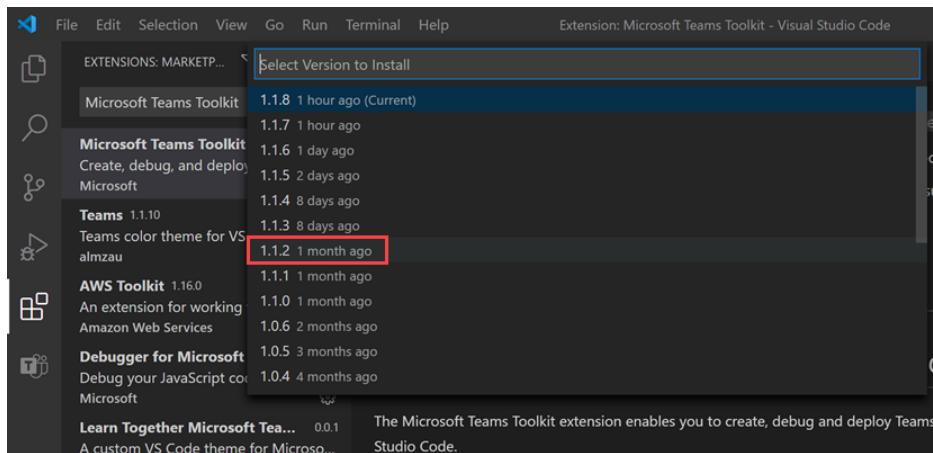
1. Open Visual Studio Code and select **Extensions** on the left Activity Bar.
2. Search **Microsoft Teams Toolkit** on the extensions panel and install the **Microsoft Teams Toolkit**.



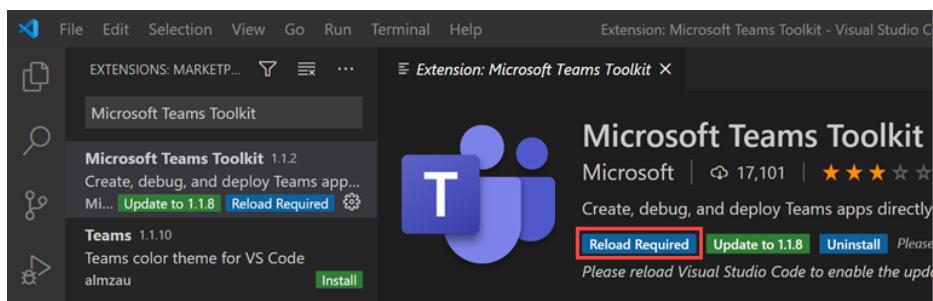
3. After the latest Microsoft Teams Toolkit is installed, click on the gear icon and choose **Install Another Version**.



#### 4. Select 1.1.2.

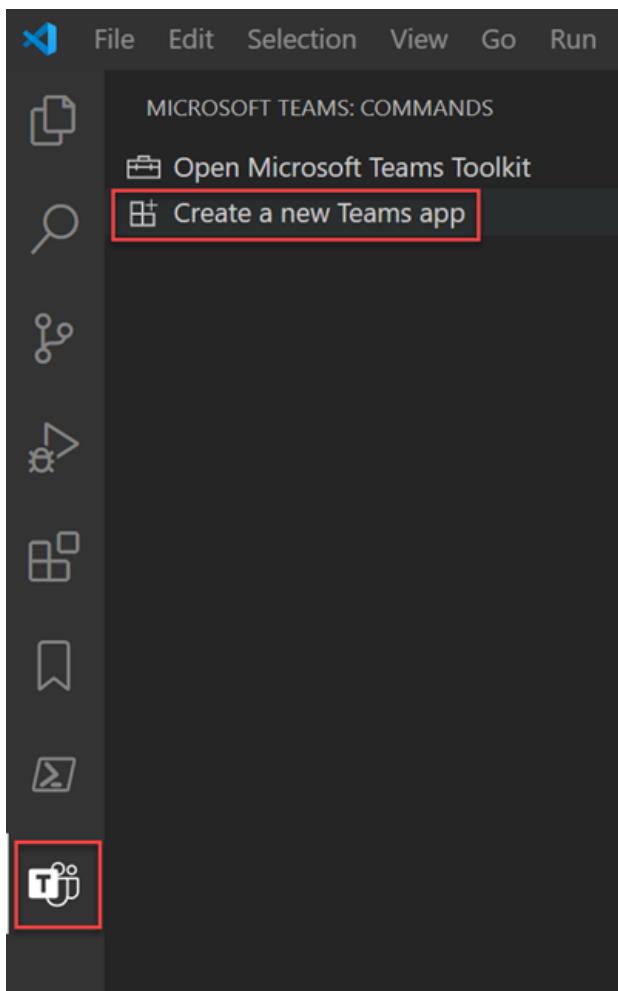


#### 5. Click on Reload Required to reload Visual Studio Code.

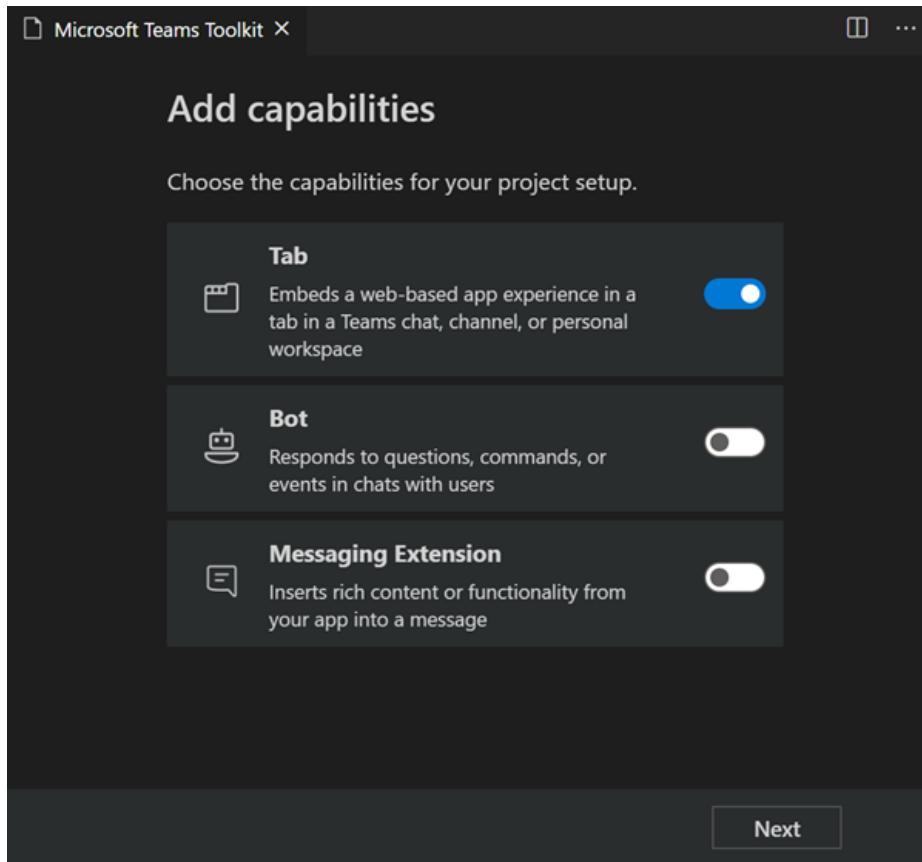


## Task 2: Use the Microsoft Teams Toolkit in Visual Studio Code to set up your first app project

1. In Visual Studio Code, select **Microsoft Teams** on the left Activity Bar and choose **Create a new Teams app**.



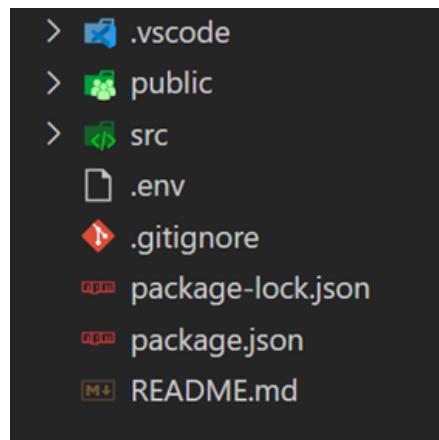
2. When prompted, sign in with your Microsoft 365 development account.
3. On the **Add capabilities** screen, select **Tab** then **Next**.



4. Enter a name for your Teams app. (This is the default name for your app and also the name of the app project directory on your local machine.)
5. Check only the **Personal tab** option and select **Finish** at the bottom of the screen to configure your project.

## Review the generated solution

Once the toolkit configures your project, you will have the components to build a basic personal tab for Teams. The project directories and files display in the Explorer area of Visual Studio Code.



**NOTE:** The toolkit automatically creates scaffolding for you in the src directory based on the capabilities you added during setup.

For example, if you create a tab during setup the **App.js** file in the **src/components** directory is important because it handles the initialization and routing of your app. It calls the [Microsoft Teams SDK](#) to establish communication between your app and Teams.

## App ID

Your Teams app ID is needed to configure your app with App Studio.

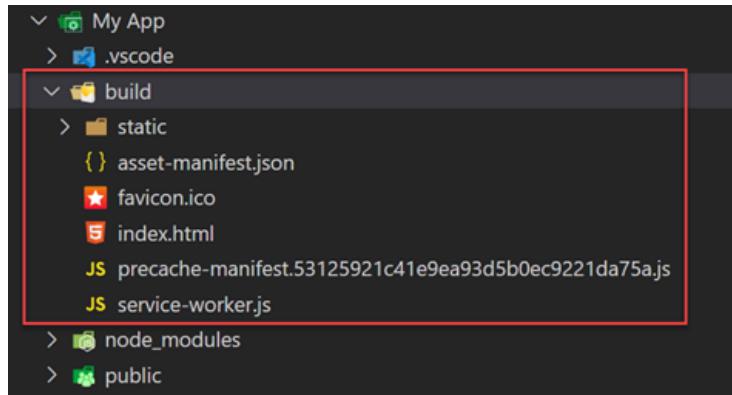
- You can find the ID in the **teamsAppId** object, which is located in your project's **package.json** file.

## Build and run your app

Your Tab will be located in the **./src/components/Tab.js** file. This is the TypeScript React based class for your Tab. - Locate the **render()** method and observe the code inside the **<div>** tag.

```
typescript <div> <h3>Hello World!</h3> <h1>Congratulations {userName}!</h1>
<h3>This is the tab you made :-)</h3> </div>
```

1. Open Terminal in Visual Studio Code. From the Visual Studio Code ribbon select **Terminal > New Terminal**.
2. Go to the root directory of your app project and run: `npm install`.
3. To build your solution run the `npm run build` command.
  - This will transpile your solution into the **./build** folder.



## Run your app

To run your app you use the `npm start` command.

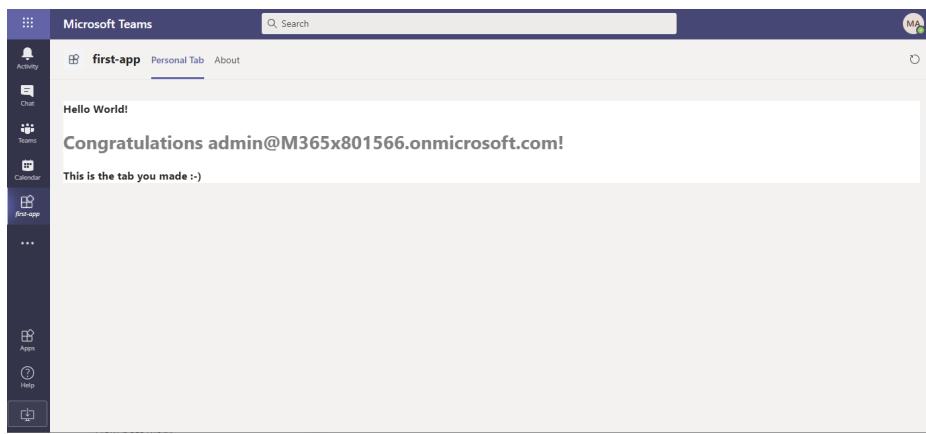
This will build and start a local web server for you to test your app. The command will also rebuild the application whenever you save a file in your project.

Once complete, there's a Compiled successfully! message in the terminal. Your app should now be running on <https://localhost:3000>.

## Task 3: Sideload an app in Microsoft Teams

Now that you've tested your tab, it's time to run your app inside Microsoft Teams.

1. In Visual Studio Code, press the **F5** key to launch a Teams web client.
2. To display your app content in Teams, specify that where your app is running (localhost) is trustworthy:
3. Open a new tab in the same browser window (Google Chrome by default) which opened after pressing **F5**.
4. Go to <https://localhost:3000/tab> and proceed to the page.
5. Go back to Teams. In the dialog, select **Add for me** to install your app.



## **Review**

In this exercise, you:

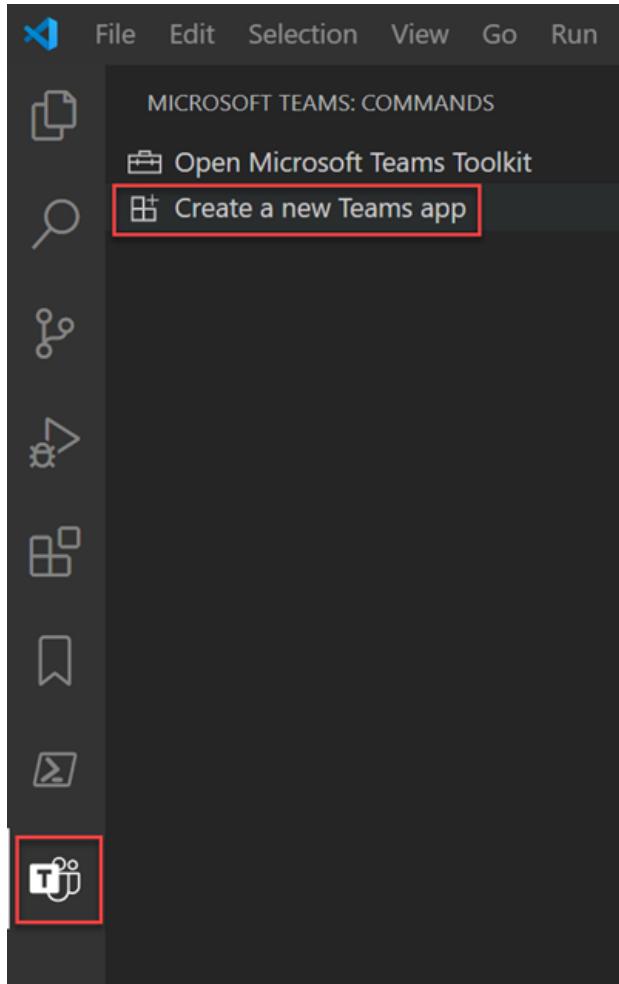
- Utilized Microsoft Teams Toolkit to creates scaffolding and reviewed the result.
- Built and run Teams Tab app in Teams.

# Exercise 3: Creating and using task modules in Microsoft Teams

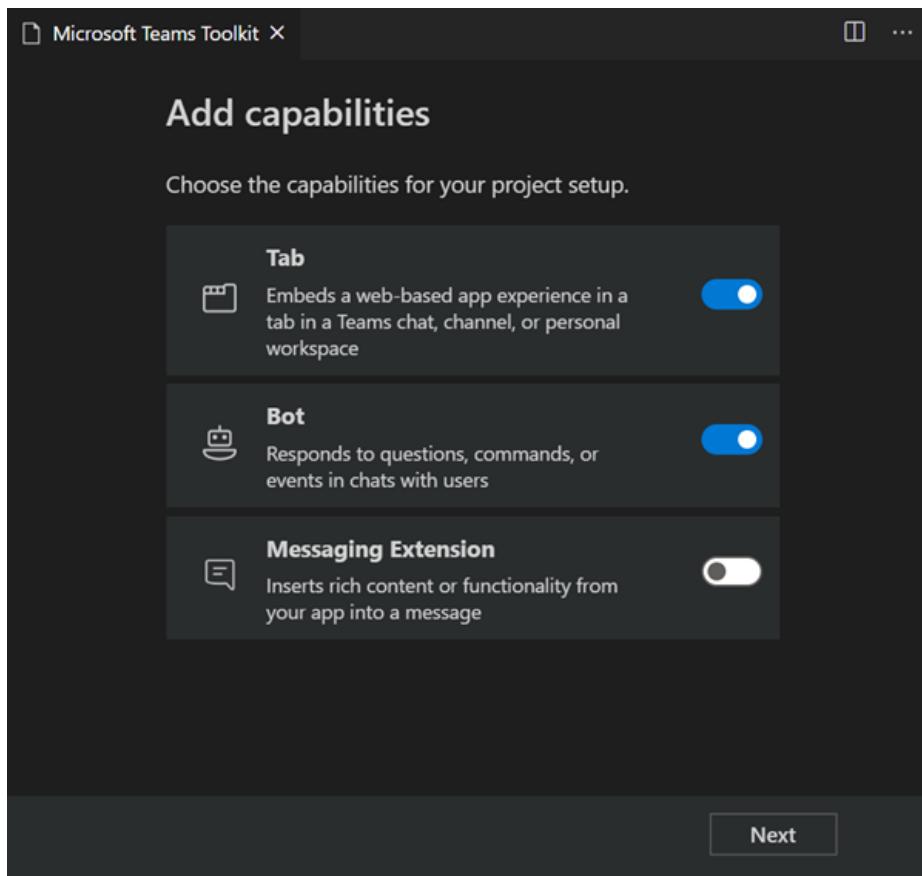
Task modules allow you to create modal popup experiences in your Teams application. Inside the popup you can run your own custom HTML/JavaScript code, show an <iframe>-based widget such as a YouTube or Microsoft Stream video or display an [Adaptive card](#).

To create a task module, you will need to create a Teams application for displaying Task modules.

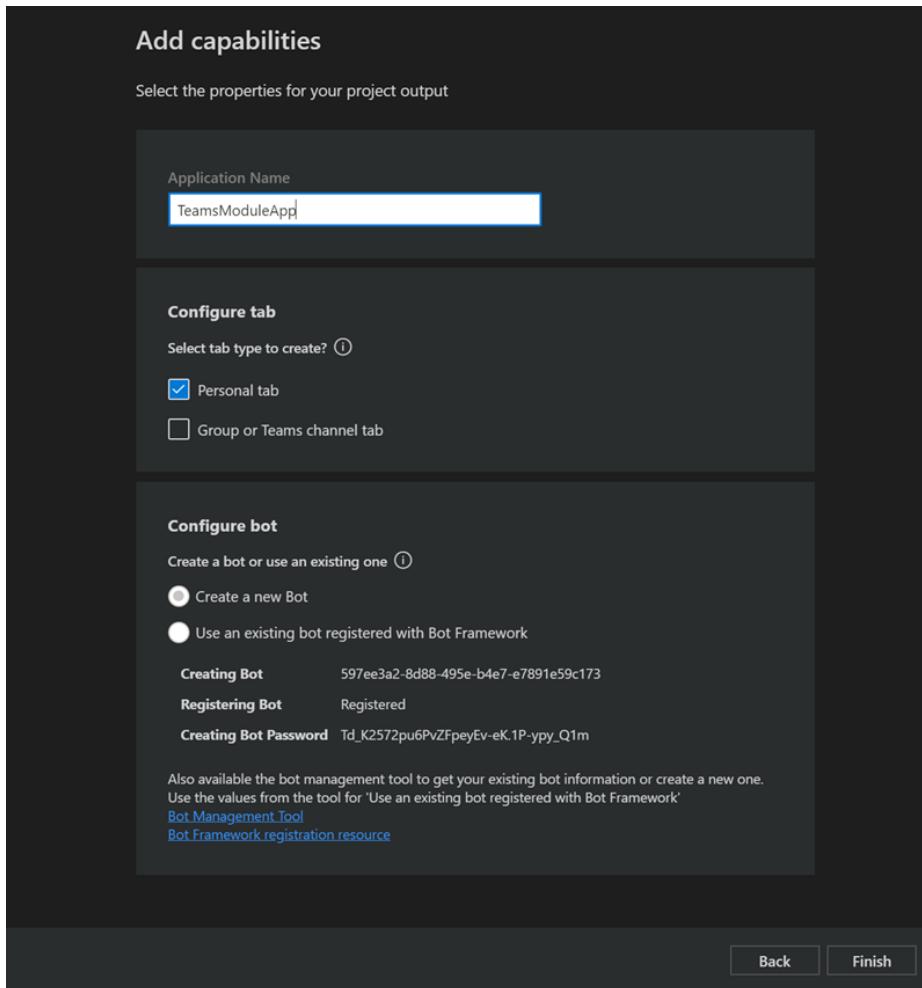
1. Open Visual Studio Code and select **Extensions** on the left Activity Bar.
2. Select **Microsoft Teams** and choose **Create a new Teams app**.



3. When prompted, sign in with your Microsoft 365 developer account.
4. On the **Add capabilities** screen, select **Tab** and **Bot** then Next.



5. Enter a name for your Teams app. (This is the default name for your app and also the name of the app project directory on your local machine.)
6. Check the **Personal tab** option in **Configure tab** section.
7. Select the **Create Bot Registration** button to register a bot in **Configure bot** section.



8. Select **Finish** at the bottom of the screen to configure your project.

## Task 1: Create a card-based task module

Cards are actionable snippets of content that you can add to a conversation through a bot, a connector, or app. Using text, graphics, and buttons, cards allow you to communicate with an audience. we will create a simple Adaptive Card to Teams Tab app project.

1. From Visual Studio Code, open the **Tab.js** file in the `..\tabs\src\components\` folder.
2. Locate the method **render()** add the following functions before it.

```
```typescript
showCardBasedTaskModule() { let cardJson = { contentType: "application/vnd.microsoft.card.adaptive", content: { type: "AdaptiveCard", body: [ { type: "TextBlock", text: "Here is a ninja cat:" }, { type: "Image", url: "https://adaptivecards.io/content/cats/1.png", size: "Medium" } ], version: "1.0" } };

let taskInfo = {
  title: null,
  height: 510,
  width: 430,
  url: null,
  card: cardJson,
  fallbackUrl: null,
  completionBotId: null,
};

microsoftTeams.tasks.startTask(taskInfo, (err, result) => {
  console.log(`Submit handler - err: ${err}`);
});
}

} ```
```

3. Continue to locate the method **render()** and update the contents to match the following code.

```
typescript render() { return ( <div> <button onClick={this.showCardBasedTaskModule}>ShowCard</button> </div> ); }
```

Now, a card-based task module has been added to the Teams Tab app.

## Task 2: Create an iframe-based task module

1. From Visual Studio Code, create a folder named **models** in the **service** folder.

2. Follow the steps below to create 4 script files in the **models** folder.

1. Create a file named **taskmoduleids.js** and copy the following code to the file.

```
```typescript const TaskModuleIds = { YouTube: 'YouTube', AdaptiveCard: 'AdaptiveCard' };

module.exports.TaskModuleIds = TaskModuleIds;````
```

2. Create a file named **taskmoduleresponsefactory.js** and copy the following code to the file.

```
```typescript class TaskModuleResponseFactory { static
createResponse(taskModuleInfoOrString) { if (typeof taskModuleInfoOrString ===
'string') { return { task: { type: 'message', value: taskModuleInfoOrString } }; }

return {
    task: {
        type: 'continue',
        value: taskModuleInfoOrString
    }
};

static toTaskModuleResponse(taskInfo) {
    return TaskModuleResponseFactory.createResponse(taskInfo);
}

}

module.exports.TaskModuleResponseFactory = TaskModuleResponseFactory;````
```

3. Create a file named **taskmoduleuiconstants.js** and copy the following code to the file.

```
```typescript const { UISettings } = require('./uisettings'); const { TaskModuleIds } =
require('./taskmoduleids');

const TaskModuleUIConstants = { YouTube: new UISettings(1000, 700, 'YouTube
Video', TaskModuleIds.YouTube, 'YouTube'), AdaptiveCard: new UISettings(400,
200, 'Adaptive Card: Inputs', TaskModuleIds.AdaptiveCard, 'Adaptive Card') };

module.exports.TaskModuleUIConstants = TaskModuleUIConstants;````
```

4. Create a file named **uisettings.js** and copy the following code to the file.

```
```typescript class UISettings { constructor(width, height, title, id, buttonTitle) {
this.width = width; this.height = height; this.title = title; this.id = id; this.buttonTitle =
buttonTitle; } }

module.exports.UISettings = UISettings;````
```

3. Create a folder named **pages** in the **service** folder.
4. Now you need to create the following HTML file in the **pages** folder.
  1. Create a file named **youtube.html** and copy the following code to the file.

```
```html <!DOCTYPE html>
```

# Table of Contents

It is strongly recommended that MCTs and Partners access these materials and in turn, provide them separately to students. Pointing students directly to GitHub to access Lab steps as part of an ongoing class will require them to access yet another UI as part of the course, contributing to a confusing experience for the student. An explanation to the student regarding why they are receiving separate Lab instructions can highlight the nature of an always-changing cloud-based interface and platform. Microsoft Learning support for accessing files on GitHub and support for navigation of the GitHub site is limited to MCTs teaching this course only.	7
Task 5: Adding the web part to a page	312
Task 6: Examining the deployed web part files	313
Task 4: Test the web part in SharePoint and Microsoft Teams	319
Task 5: Testing the SPFx web part in Microsoft Teams	320
Task 5: Adding the web part to a page	356
Task 6: Examining the deployed web part files	357
Task 4: Test the web part in SharePoint and Microsoft Teams	363
Task 5: Testing the SPFx web part in Microsoft Teams	364
Task 3: Deploying SPFx web part to Microsoft Teams	374