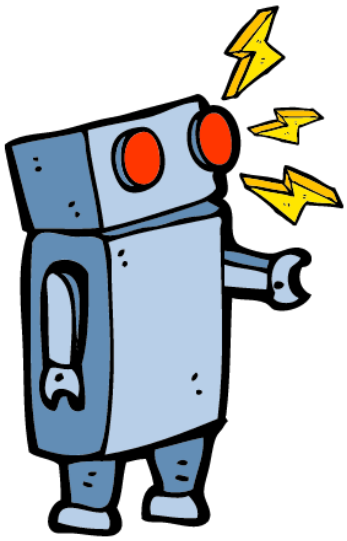


TEAMS APPS

HANDS-ON LAB

Using the Teams Platform to surface a Recruitment application in Teams

ABSTRACT

This lab document contains a step-by-step guide to create a Teams App that enables a recruitment team to manage the job positions and potential candidates who have applied for those job positions all within Teams.

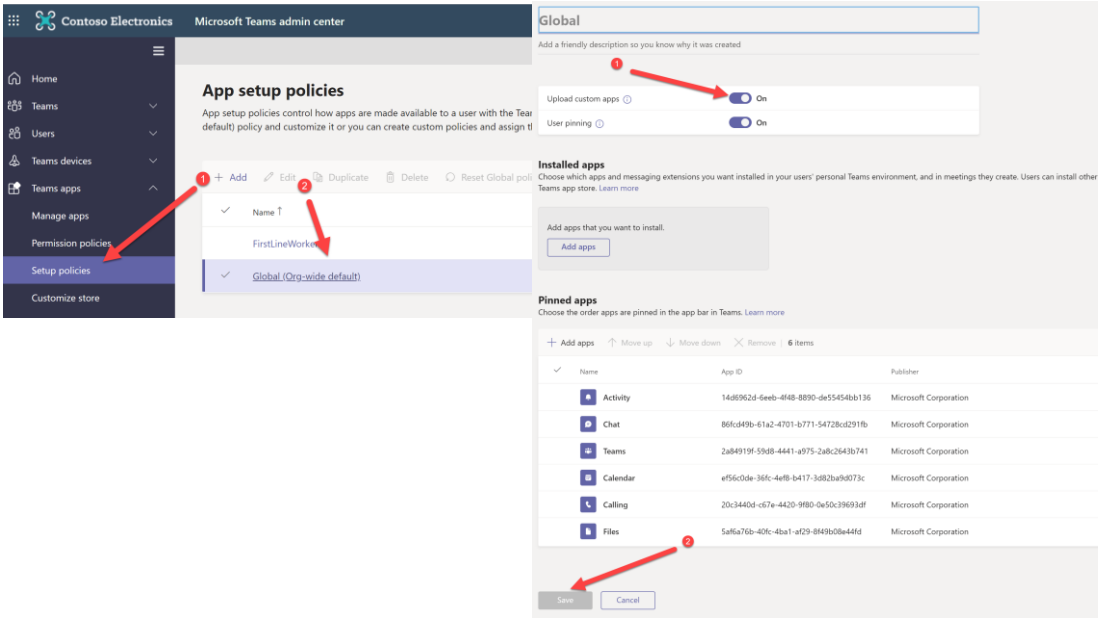
About

Lab guide v2 created by Jack Lewis and Scott Perham of Microsoft UK. Credit to the original team behind this Teams app hands on lab, found here - https://github.com/OfficeDev/msteams-sample-contoso-hr-talent-app

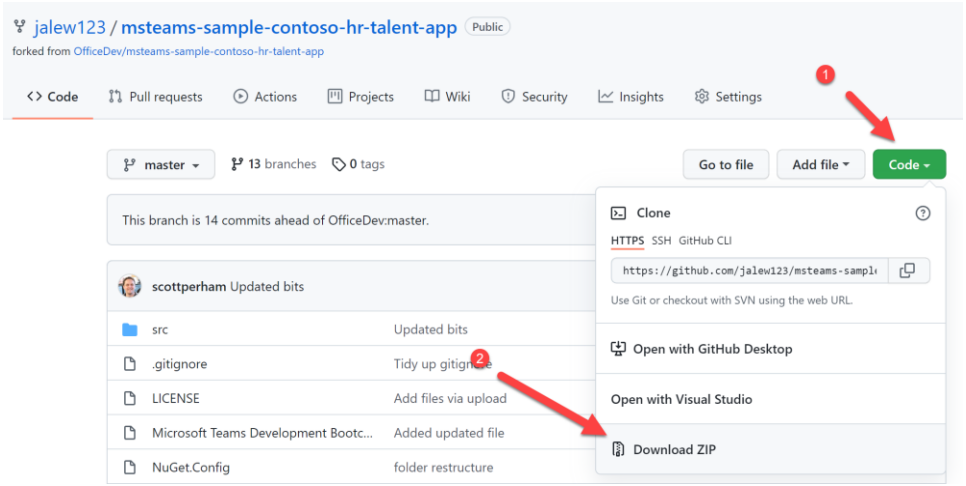
Getting Started

You will need the following to complete this lab:

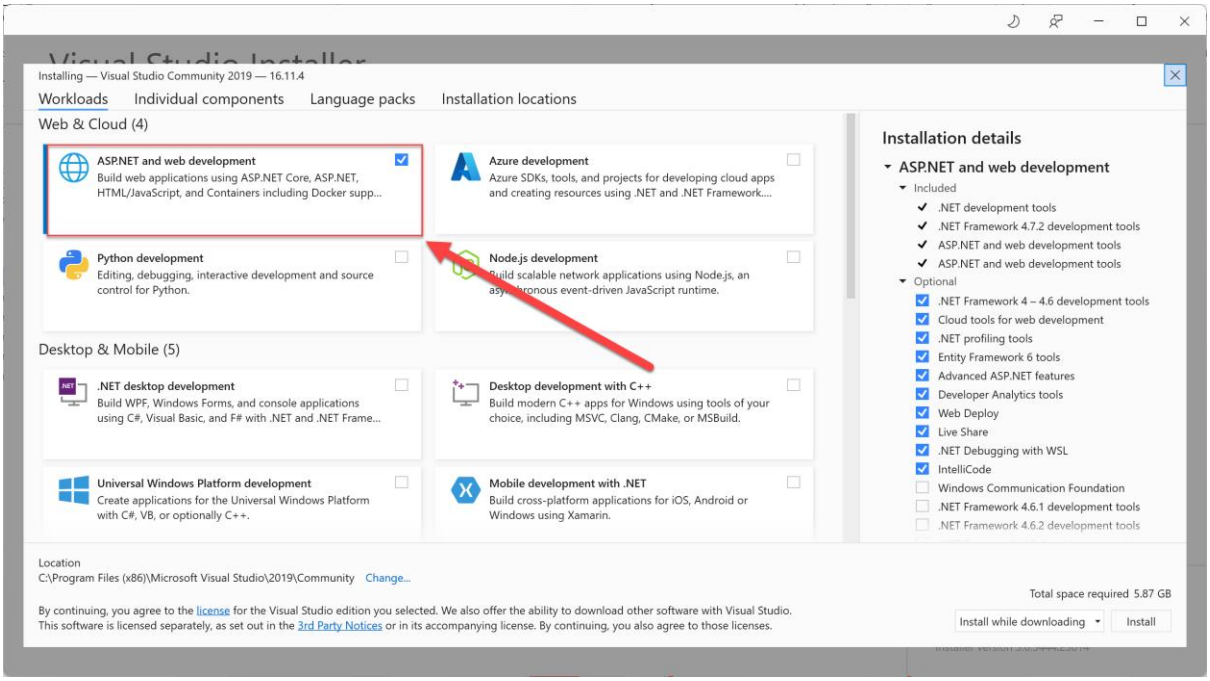
1. **A Microsoft 365 E5 Tenant**, which can be obtained from https://cdx.transform.microsoft.com/ (for Microsoft Partners) or by signing up to the Microsoft 365 Developer Program – Welcome to the Microsoft 365 Developer Program | Microsoft Docs . Alternatively, a Microsoft 365 E5 Trial will also provide the functionality you need. We do **not** recommend using your production/live Tenant for this lab. We will refer to this Tenant as the **Test/Dev Tenant** in this documentation.
2. **Enable Side-Loading in Teams**. To do this, go to https://admin.teams.microsoft.com/ and login as an administrator in your Test/Dev Tenant. Click **Setup policies** and select the **Global (Org-wide default)** policy. Ensure **Upload Custom Apps** is **On,** click **Save** if you have set this from Off to On. ***Note: if this is not enabled, it may take up to 8 hours for this policy to take effect.***



3. **The Talent Management App**, which can be downloaded from here, click **Code** and select **Download ZIP**. You'll then need to extract that ZIP to a location on your computer.



4. **Visual Studio**, installed onto your local machine. If you do not have Visual Studio, the Community edition can be downloaded here. When installing Visual Studio, make sure to select **ASP.NET and web development**.



5. **.Net 6 SDK**, which is required to run the Talent Management App, available here.
6. **NGROK,** a tunnelling tool, which will allow you to run your app within Teams, available here. You'll need to extract the NGROK ZIP to a location on your computer. *We highly recommend signing up for a free NGROK account, as without this, your tunnels will expire in 2 hours.*
7. **Postman,** a tool that allows you to send HTTP requests from your computer, which will be used during the Microsoft Graph API section of this lab, and is also a useful troubleshooting tool, available here.
8. **Azure Subscription,** required to create the Azure Bot Service instance that will be used within your Teams app. This resource does not cost anything so if you want to use an existing subscription that is fine. Alternatively, sign up for an Azure trial here.

Talent Management App Lab

Overview (Mandatory reading before you begin!)

During this lab, you will learn how to build and deploy a Microsoft Teams app in Office 365 that will be used by the Human Resources department within their Microsoft Teams clients. The app will facilitate the department's hiring of new talent into the organization, provide immediate interview feedback, schedule interview loops, and improve the overall hiring process of new employees. The lab is divided into several exercises that will help you understand how to transform the hiring and candidate management flow of new talent and make it more interactive and responsive for HR teams and interviewees.

Before you attempt to begin and complete the lab in this document, please make sure all the pre-requisite steps have been completed and your machine is ready for the labs. These pre-requisites can be found in the 'Getting Started' section at the very beginning of this document.

It is strongly recommended that you complete each section of the lab, in the order of this document. While you may be able to jump around, it is likely that you will miss key concepts/configuration if you skip steps in this lab.

If you are unfamiliar with Teams Apps and the interfaces that are available for you to expose a third-party app's capabilities in Teams, I would highly recommend reviewing the following documentation, which will introduce you to the interfaces, at a high-level, that you'll be working with throughout this lab - Build apps for the Microsoft Teams platform - Teams | Microsoft Docs .

I'd also encourage everybody to visit the app store in Teams, and install some popular apps, such as Polly, and see how they work and allow you to interact with the third-party app, all inside the Teams interface. This will allow you to understand some of what is possible via the various interfaces, and for developers who work at our partners, it will allow you to get an initial understanding of how multi-tenant SAAS apps are offered as Teams app via the Store.

A key concept to understand is how auth works in Teams apps. From our experience, working with Partners in the UK, the most difficult/time-consuming work is typically aligned to getting SSO working for your Teams app. Remember that; all users of Teams have signed into Teams using an Azure Active Directory Identity, there is no other way to get access to Teams. For this reason, it makes sense to configure your application to work with Azure AD SSO, specifically OpenID and OAuth for Teams Apps (not SAML), as this allows you to do a silent, non-interactive, retrieval of the SSO token, via Azure AD, by using the Teams JavaScript SDK and allows you to work with the Microsoft Graph API. Another key concept to understand with Teams and SSO, is that because your app is being delivered inside an iFrame, the Teams SDK must be used to control when an authentication/login screen is being displayed. From experience, we have found that some SPA apps have had to go through a decent amount of re-configuration to hand off this process, when a protected page/resource is accessed. I am making you aware of this, not to put you off developing a Teams app, but to let you know that some work is likely going to be required to make sure the authentication/access process, into your app, is as smooth as possible. Once you've got the hang of working with the Teams SDK, I am confident you will see that it adds immense value, and results in a great user experience when they are accessing your app. Finally, we work a lot with partners, when building the SSO with them, around Identity matching. This is likely something most partners will need to do, to ensure that when customers access your app in Teams, and Azure AD SSO is performed via the Teams SDK, they gain access to the same resources, if they logged into your app, using a client and a username/password that is unique for your application. This sample app does not demonstrate how to do identity matching, but Scott has built a sample, available on his GitHub,

that does demonstrate how this ***can*** work in a production environment – [scottperham/AADLab (github.com)]

Additionally, while this is a .NET/C# sample app, in the real world, your app can use whatever code/framework you want, and we have SDKs that work across multiple languages, and samples for you to review. In the UK we have worked with partners who have built Teams apps using; React, node/JS, .NET, Ruby on Rails, Angular, JAVA, Python, Blazor, PROGRESS, and many others! See the following for a list of some of the Teams apps samples that are available - [OfficeDev/Microsoft-Teams-Samples: Welcome to the Microsoft Teams samples repository. Here you will find task-focused samples in C#, JavaScript and TypeScript to help you get started with the Microsoft Teams App! (github.com)]

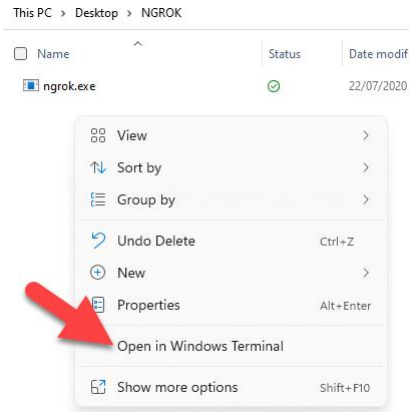
Finally, we want to say, good luck with the labs, and we hope you enjoy and learn a lot from them! It has taken a substantial amount of time for us to rework this app, get it updated; using the latest version of .NET, Teams JS SDK & Bot Framework SDK, and of course, to write this lab guide (which probably was the most time-consuming task out of it all!). Please do let us know if you have any feedback, by either logging an issue on the GitHub repo, or getting in touch with us, via Teams or email, at jack.lewis@microsoft.com & scperham@microsoft.com . Thank you for taking the time to work through these labs!

Lab Steps

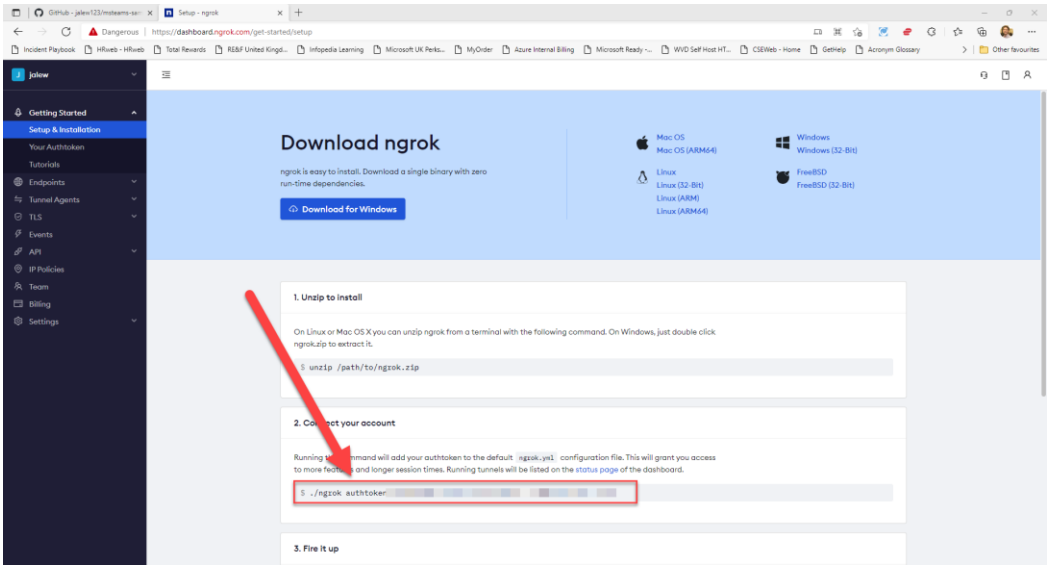
1) Setup Talent Management App

In this step we will setup the NGROK Tunnel and open and configure the Talent Management App in Visual Studio. Setup the Azure AD App Registration & Bot Service Resource in Azure. And then run (debug) the Talent Management App locally, within Visual Studio.

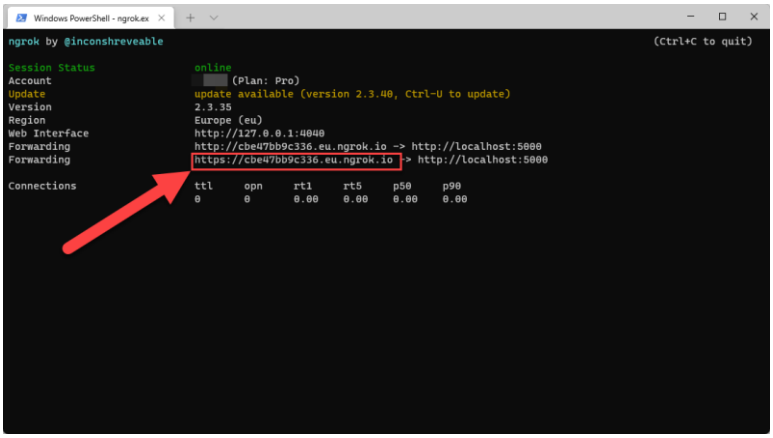
1. Open the folder where you extracted the NGROK ZIP to and start a **Command Prompt/Windows Terminal** session here.



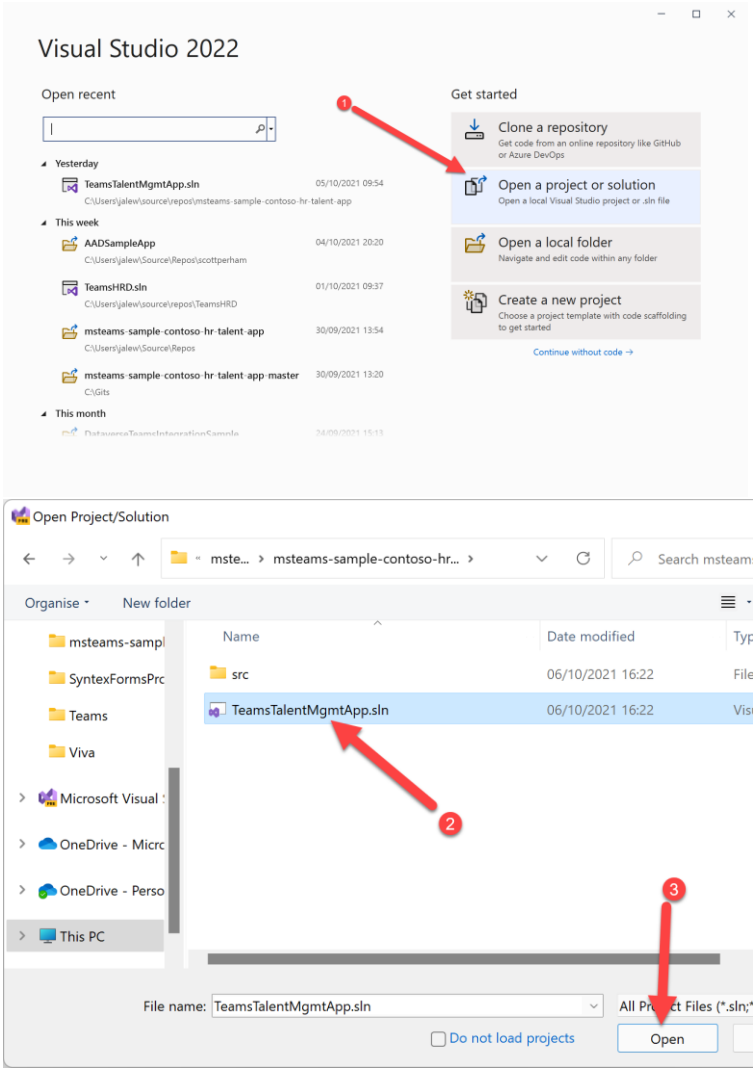
2. If you signed up for an NGROK free trial, you'll need to submit your access token to NGROK, to authenticate you with their service. To do this, enter the following in the Command Prompt/Windows Terminal session: **.\ngrok authtoken ENTERAUTHTOKENHERE**. You can get your auth token from the NGROK account dashboard, found here https://dashboard.ngrok.com/get-started/setup



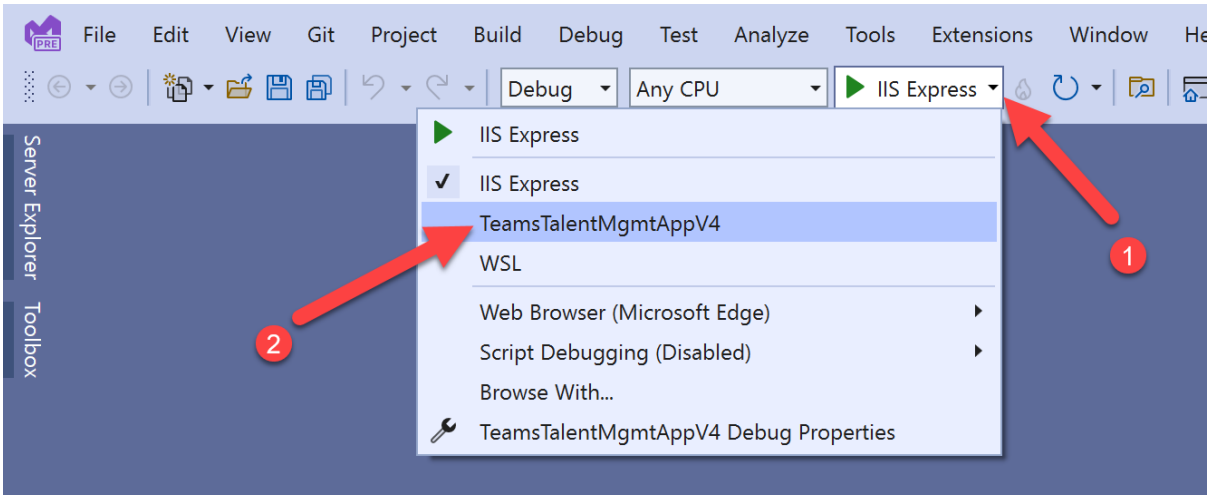
3. Enter the following in the Command Prompt/Windows Terminal session: **.\ngrok http 5000** This will open an NGROK tunnel and forward traffic destined for the forwarding URLs to localhost:5000 (where you will run the Talent Management app later in this lab).
Make note of the HTTPS forwarding URL. You will need this later when you configure the Teams app. Do not close the NGROK window. Make sure this window is left open.



4. Open Visual Studio and open the **TeamsTalentMgmtApp.sln** file found within the extracted Talent Management Repo you downloaded from GitHub.

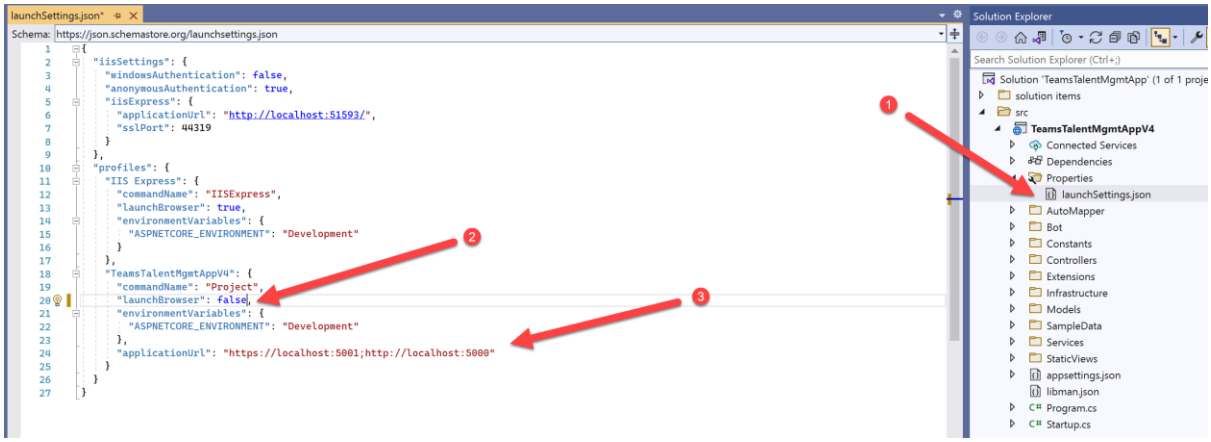


5. When Visual Studio has loaded the solution and associated projects, we need to set the debug properties correctly. In the Visual Studio toolbar, select the **IIS Express dropdown menu** and select **TeamsTalentMgmtAppV6**.

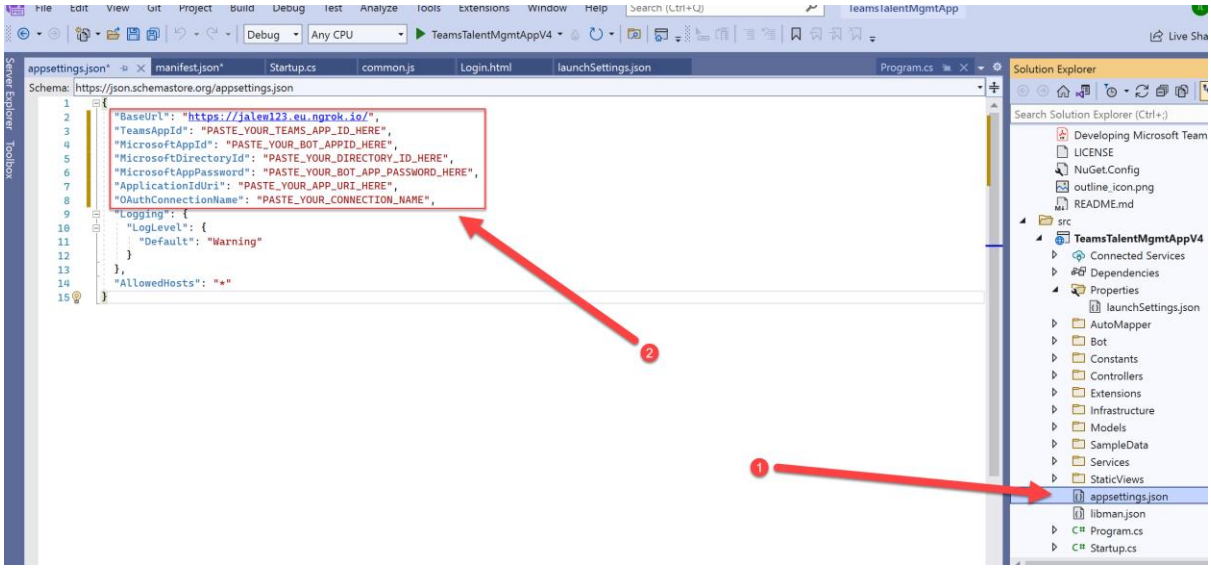


6. On the right-side of Visual Studio, within Solution Explorer, expand **src, TeamsTalentMgmtAppV6, Properties** and select **launchSettings.json.** This file contains the properties that are used when we run (debug) the Talent Management App locally, in Visual Studio. Change the value of **launchBrowser** (line 20) to **false**. Review the **applicationUrl**

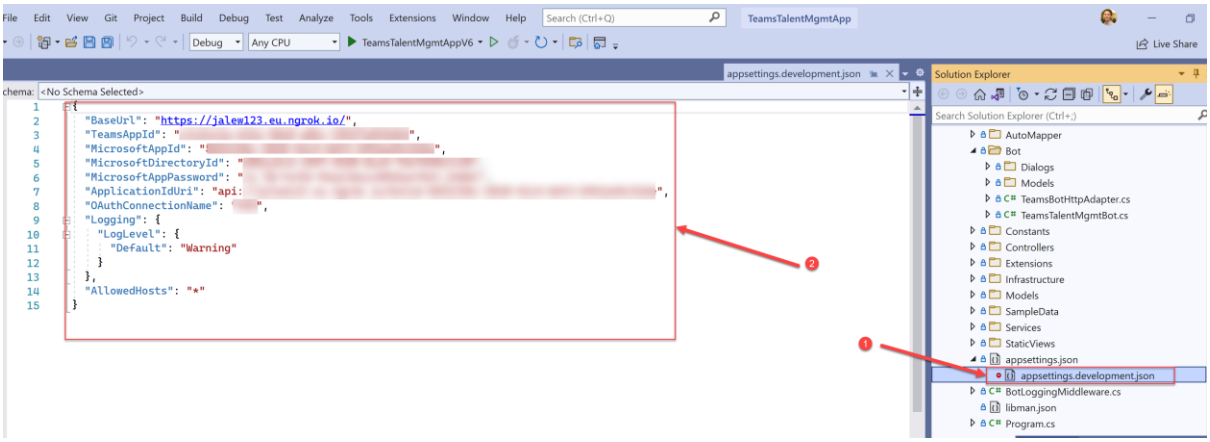
values (line 24) – these should use port 5001 and 5000 (for HTTPS and HTTP respectively). Hit **CTRL+S** to save or click the Save icon in the toolbar.



7. Select **appSettings.json**. This file contains placeholder values for **BaseUrl, TeamsAppId, MicrosoftAppId, MicrosoftDirectoryId, MicrosoftAppPassword, ApplicationIdUri & OAuthConnectionName**. Before we can run this app, we need to replace these with actual values. Currently, we only know the **BaseUrl** value - it is the NGROK URL. Replace the value for **BaseUrl** (line 2) with the **NGROK HTTPS URL Value** (gathered during step 2 of this section of the lab). Hit **CTRL+S** to save or click the Save icon in the toolbar.

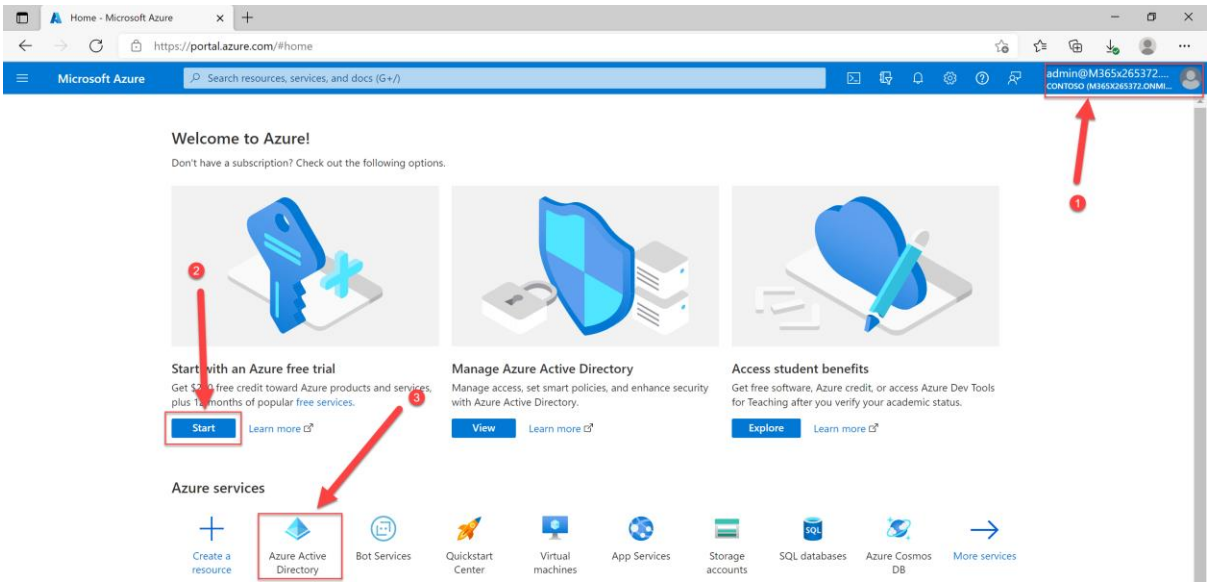


OPTIONAL: If you have cloned this this application, from a forked GitHub repo, and want to ensure that your App Settings file is not synchronised back to GitHub when changes are pushed from Visual Studio to your GitHub repo, then create a file and name it development.appsettings.json and copy the contents of appsettings.json into the development file. You will then notice that a red icon (similar to a stop sign here in the UK!) appears next to the file, this is telling you that the file will be 'ignored' when you use GitHub. If you populate development.appsettings.json with your application's values, if will not synchronise back to GitHub, therefore keeping the values that you want to protect (such as the app secret) stored locally on your machine only:

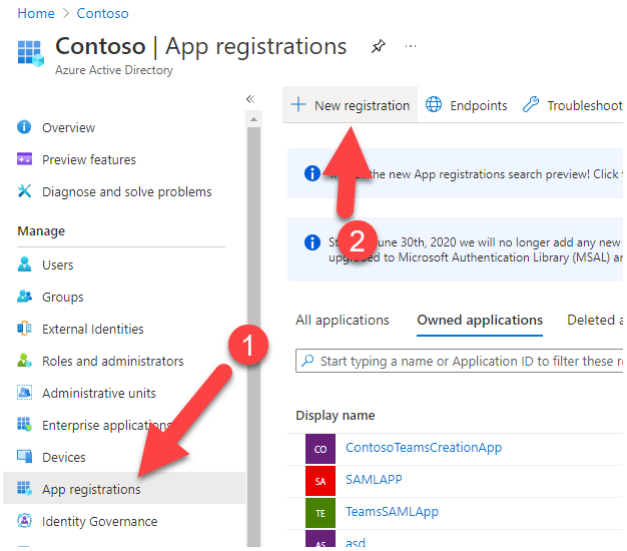


Please Note: The ServiceUrl item, in the appsettings.json file, is pre-populated with the value "https://smba.trafficmanager.net/uk" – this should work for global implementations of this Teams app, although you can change the path from uk, to one of the following, if you'd prefer: amer, apac, in, emea or jp.

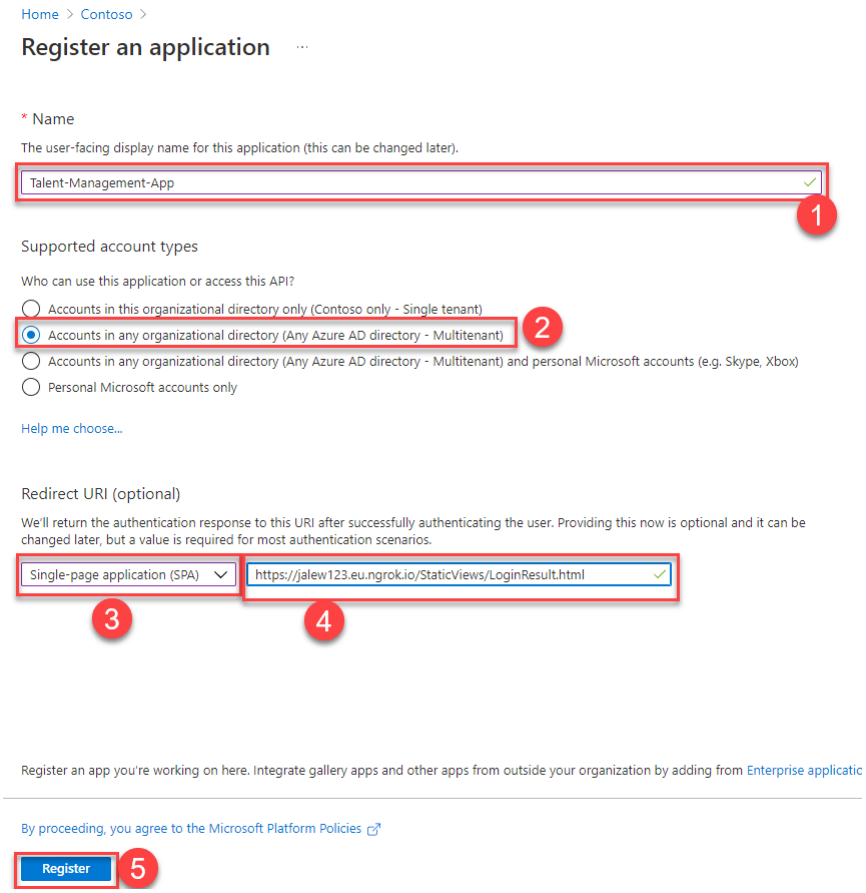
8. We now need to setup the Azure AD App Registration and Azure Bot Service resource in Azure. This will provide us with the remaining values for the appSettings.json file. Navigate to https://portal.azure.com/ and **sign in with the Administrator account, that has access to an Azure Subscription. If you do not have an Azure Subscription, sign in with your Test/Dev Tenant Administrator, and sign up for an Azure Free Trial**. Once you are signed in and have access to an Azure Subscription on that account, Select **Azure Active Directory**.



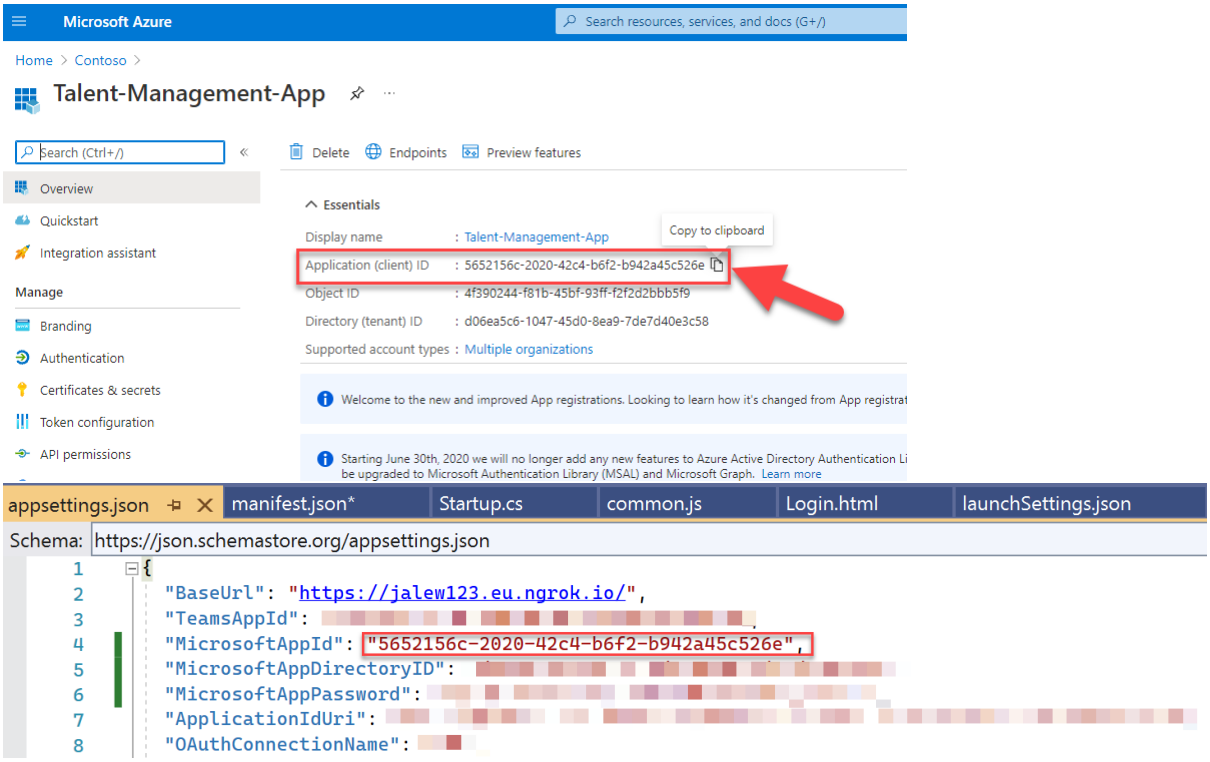
9. Select **App Registrations** and then click **New Registration**



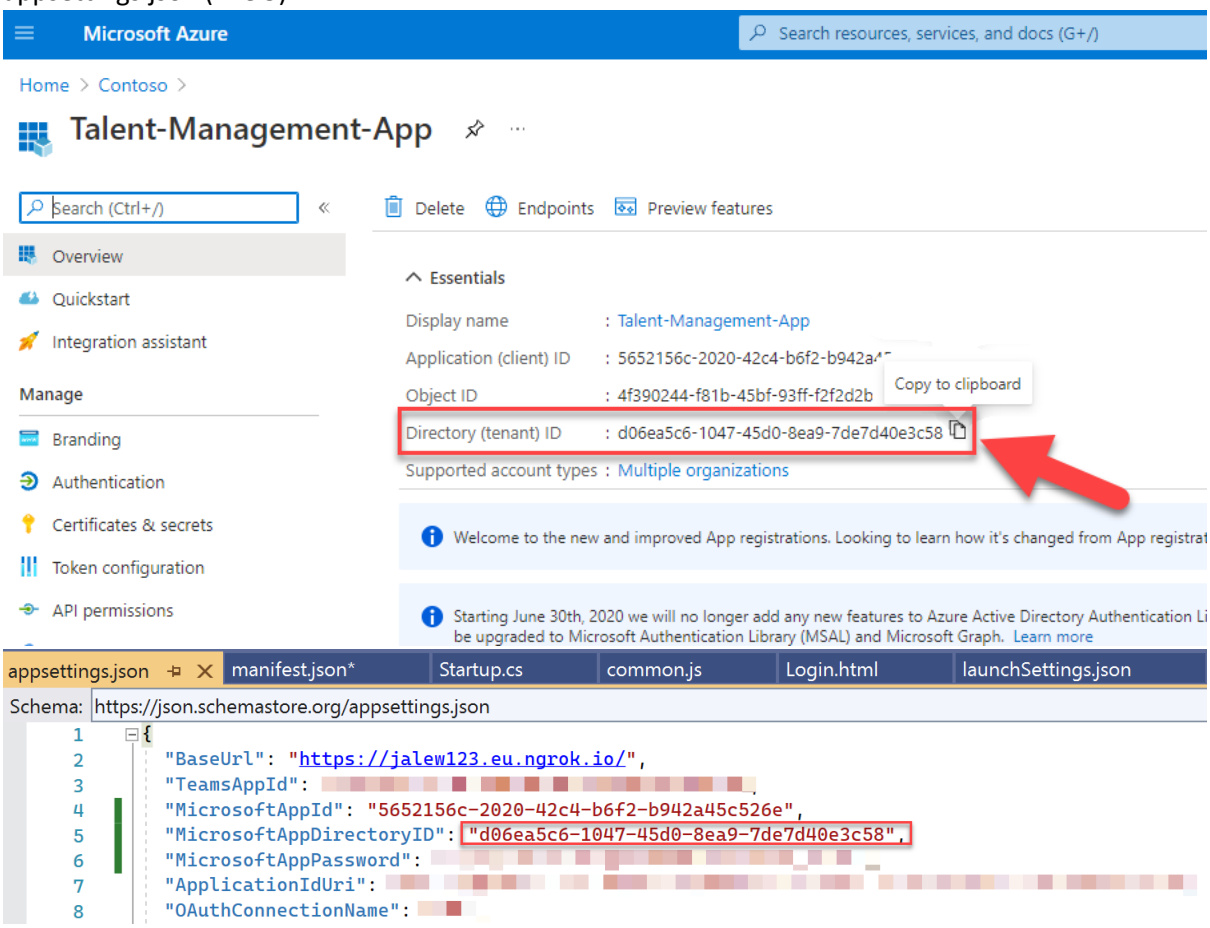
10. Enter a name for your App, such as **Talent-Management-App**, select **Accounts in any organizational directory (Any Azure AD directory – Multitenant)**, select **Single-page application (SPA)** from the dropdown menu and then enter your Redirect URI as NGROKURL/StaticViews/LoginResult.html – click **Register.**



11. On the Overview page, copy the **Application (client) ID** and enter this as the **MicrosoftAppId** value in appsettings.json (line 4).

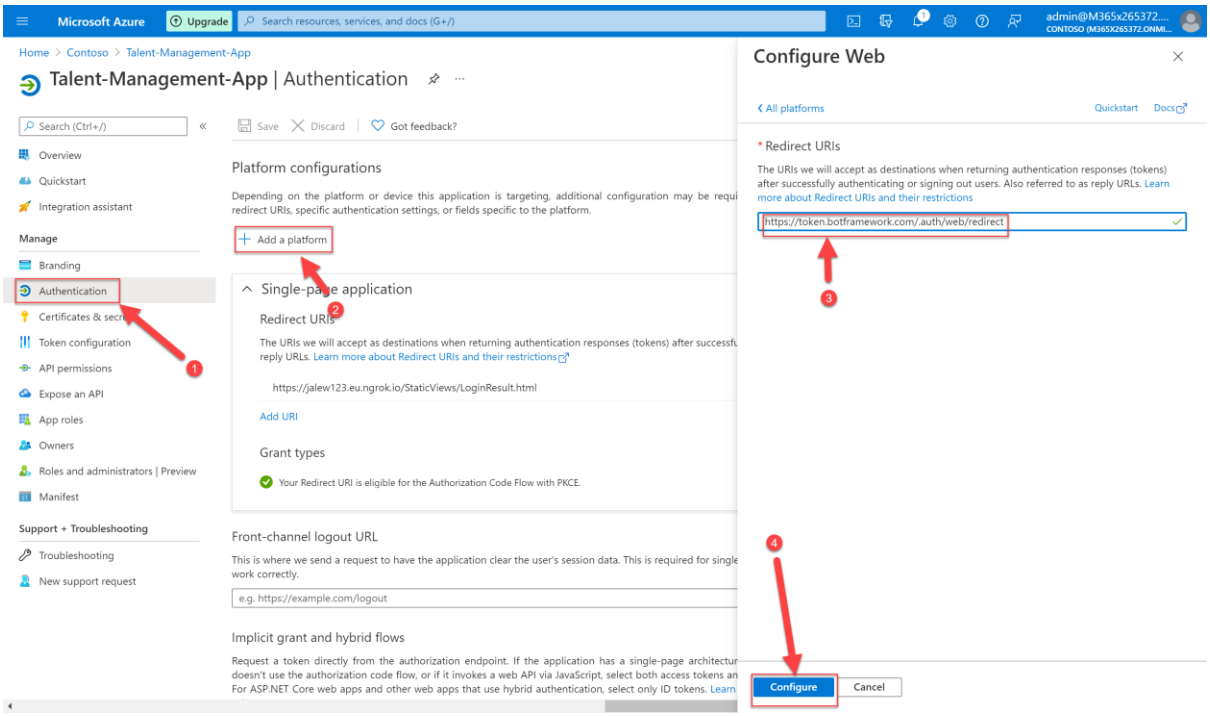


12. Copy the Directory (tenant) ID and enter this as the **MicrosoftDirectoryID** value in appsettings.json (line 5).

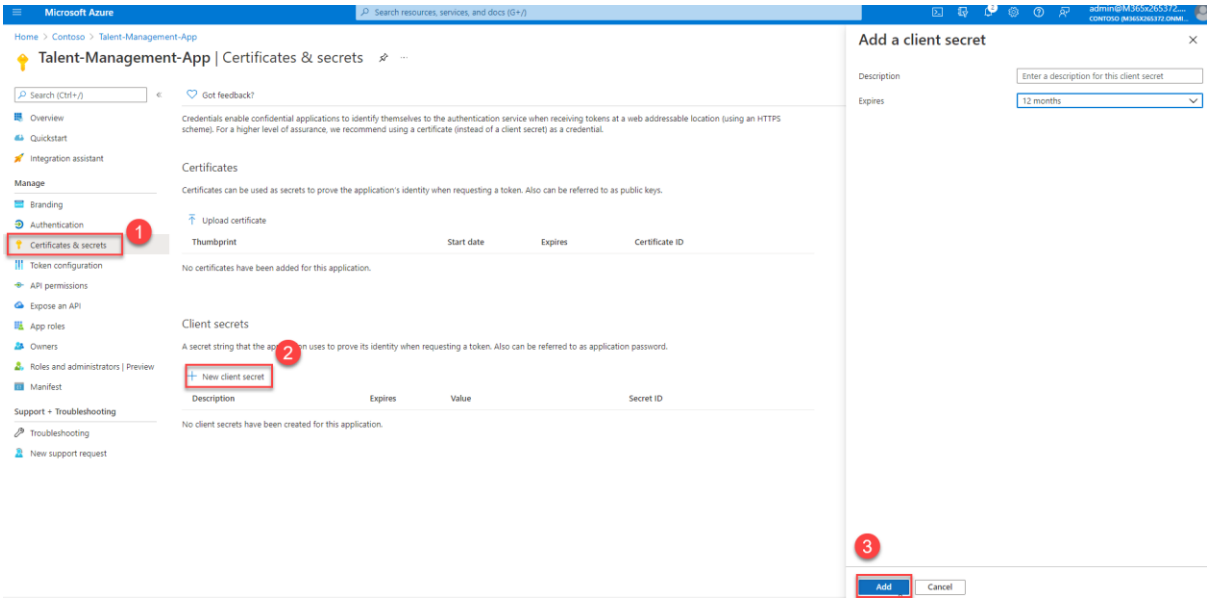


13. Select **Authentication**, click **Add a platform,** select **Web**, enter the Redirect URI of **https://token.botframework.com/auth/web/redirect** and click **Configure**. For context,

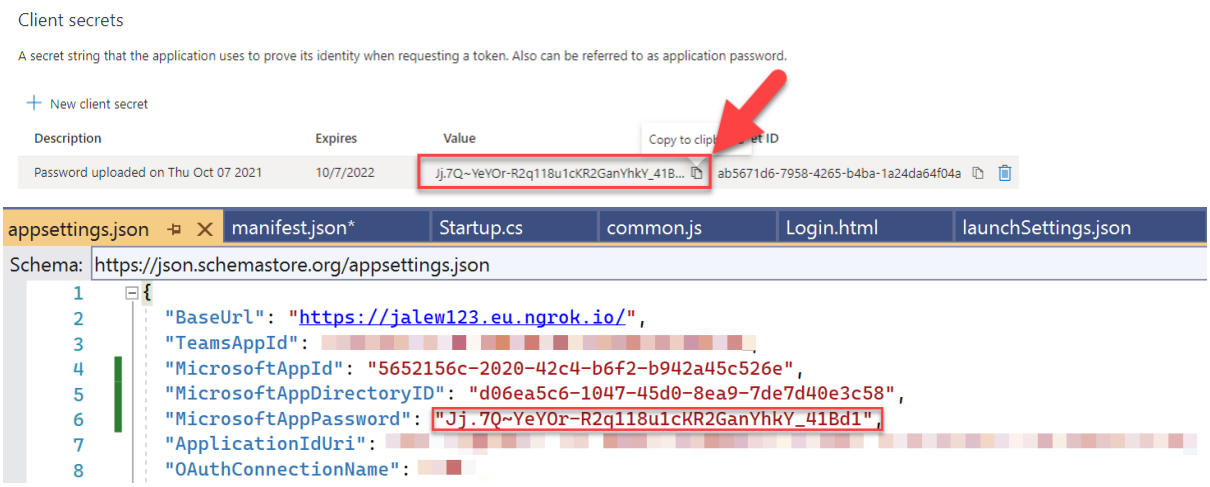
these Redirect URIs are the allowed locations that Azure AD will redirect clients to when an access token has been received via a successful authentication. We need to configure the Bot Framework URI, as Bot SSO uses this address when obtaining authorisation/consent. The SPA URI is used to redirect browsers back to you LoginRedirect.html page, which is the page used to process the Azure AD access token, and then store it in Local Storage for use when calling the Talent Management APIs. Feel free to inspect the LoginRedirect.html & Login.html pages to see what they do, and how the use the latest MSAL Browser libraries to enable SSO support for this app.



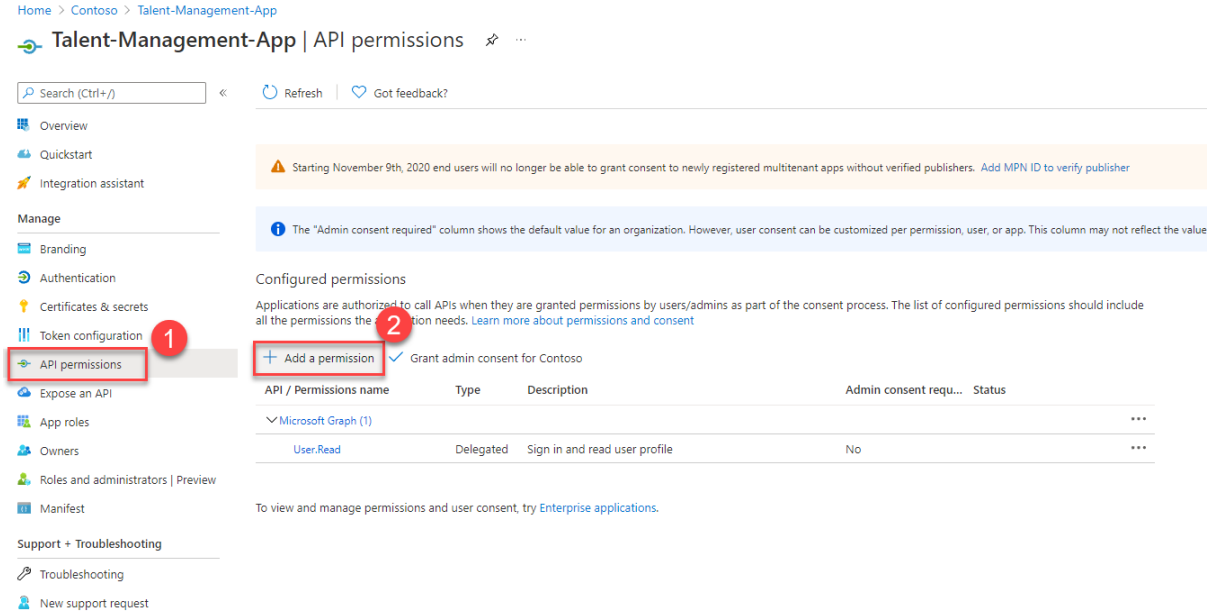
14. Select **Certificates & secrets,** select **New client secret** and click **Add**



15. Once generated, copy the **Value** of the Client Secret and enter this as the **MicrosoftAppPassword** value in appsettings.json (line 6).

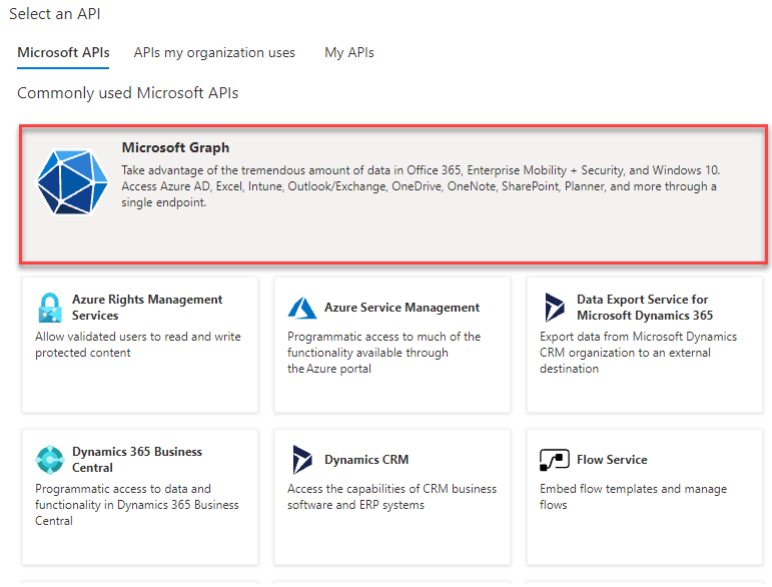


16. Select **API permission** and click **Add a permission**.

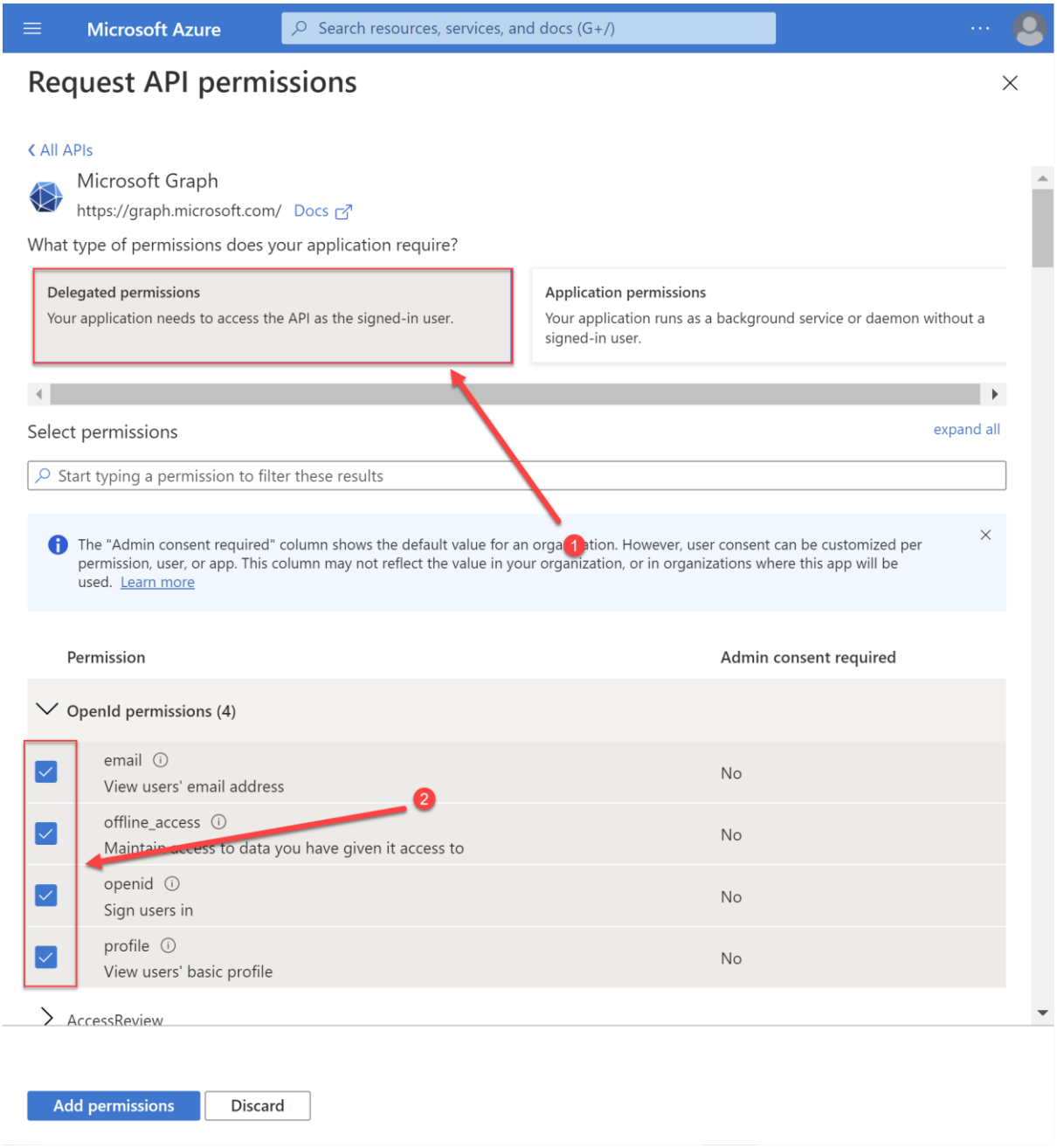


17. Select **Microsoft Graph**

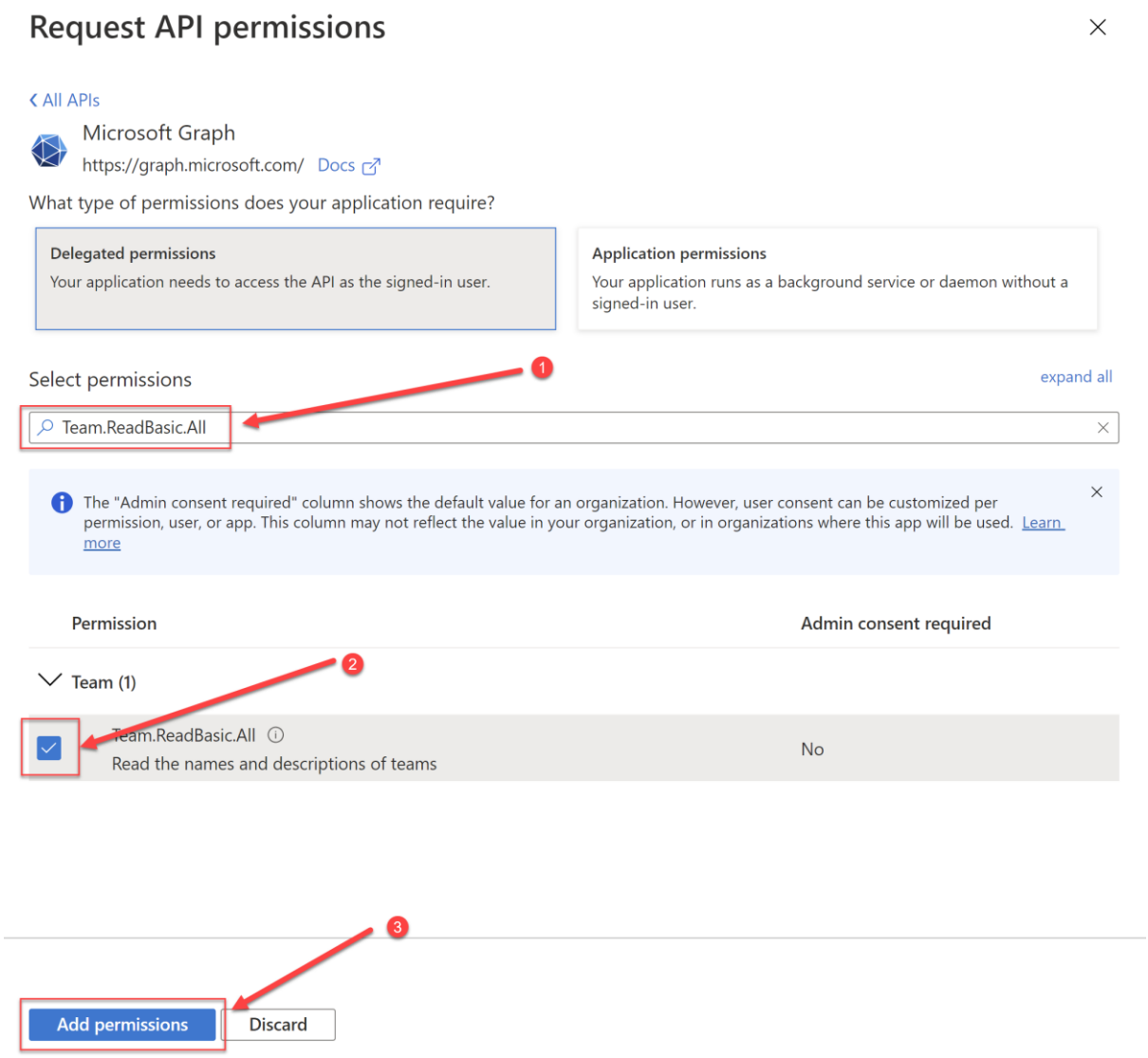
Request API permissions



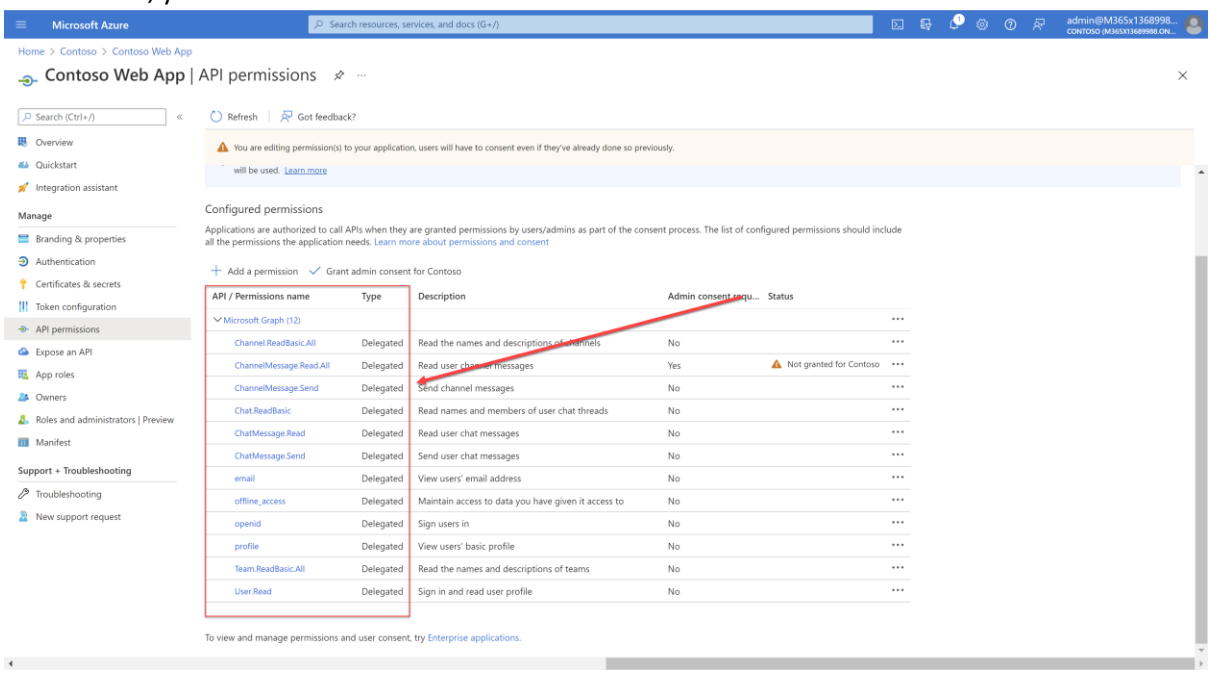
18. We'll start with the Delegated Permissions first, then we'll do the App Permissions. Select **Delegated permissions**, select **email, offline_access, openid & profile.**



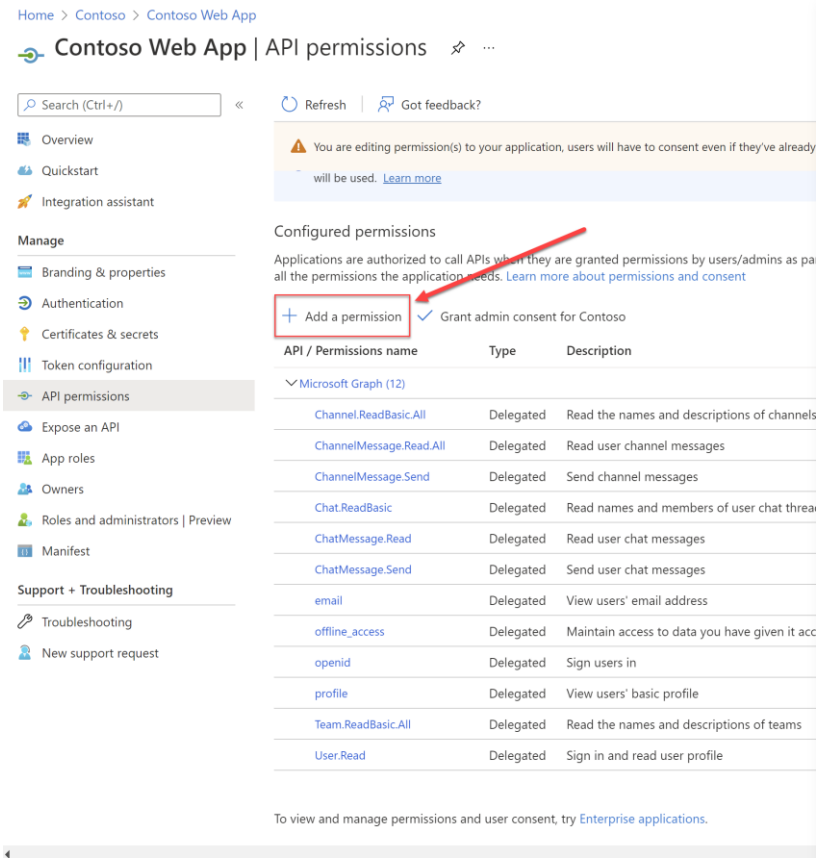
19. Then **search for and select Team.ReadBasic.All, Channel.ReadBasic.All, ChannelMessage.Send, ChannelMessage.Read.All, Chat.ReadBasic, ChatMessage.Read & ChatMessage.Send.** Select **Add permissions** once all these scopes have been selected.



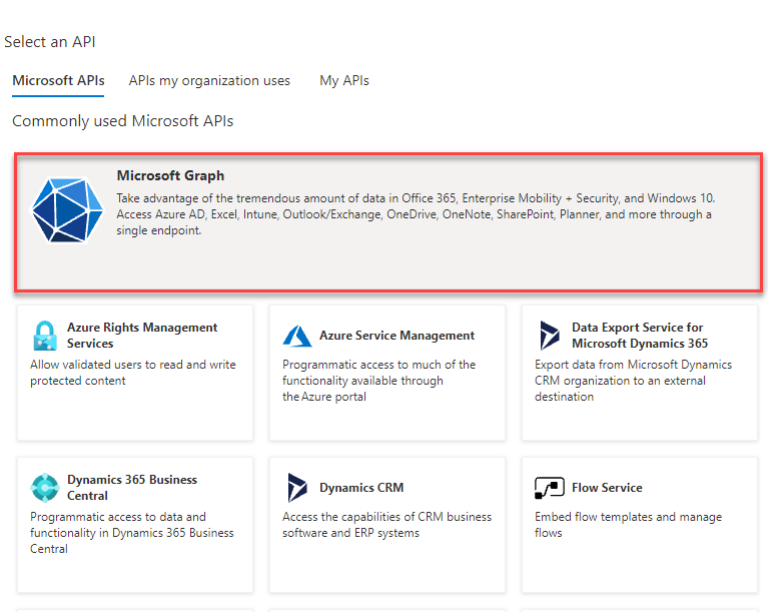
20. If successful, your API Permissions should look like the below:



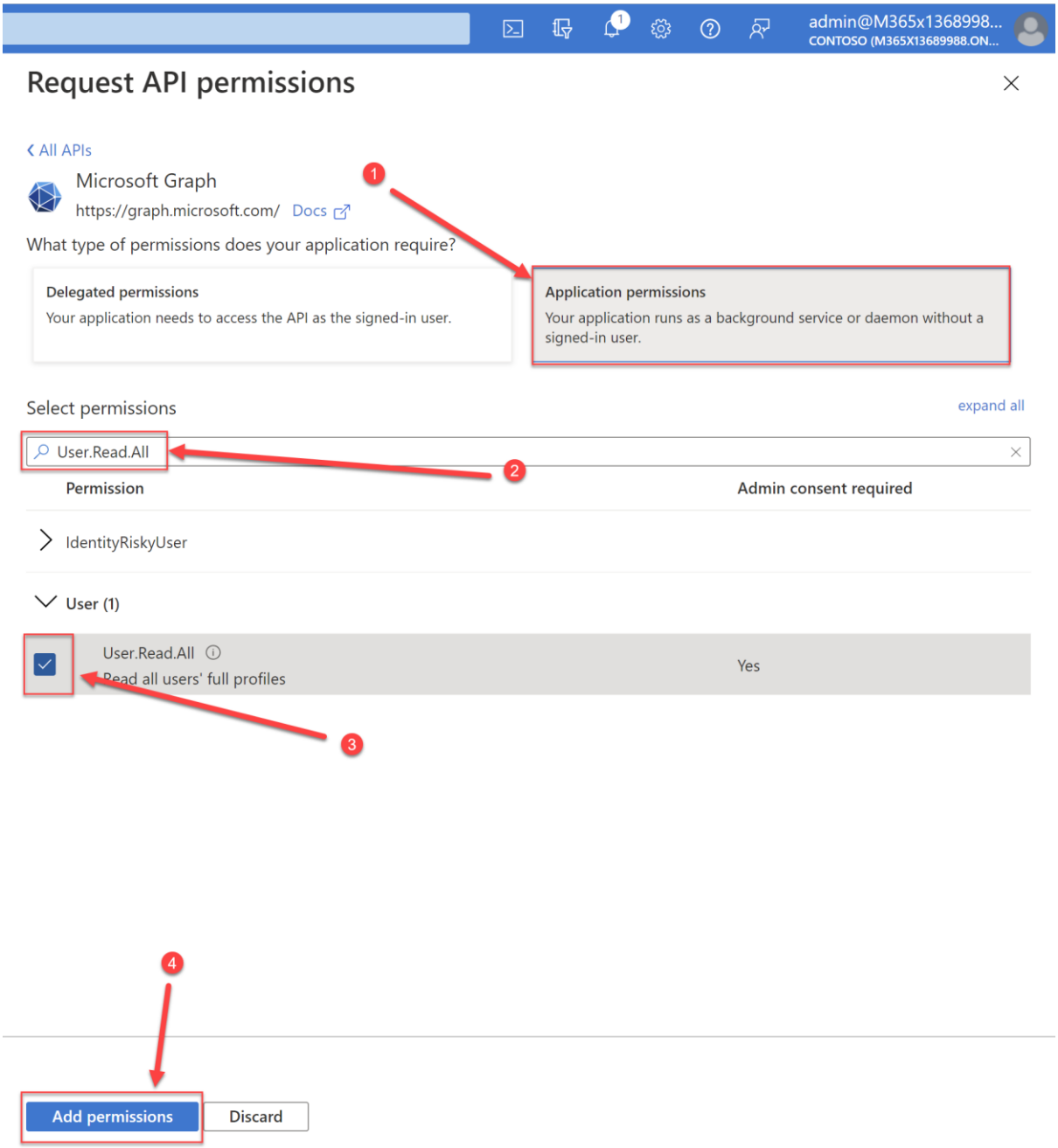
21. Now that the delegated permissions have been added, we can now add our Application Permissions. Click **Add a permission** again



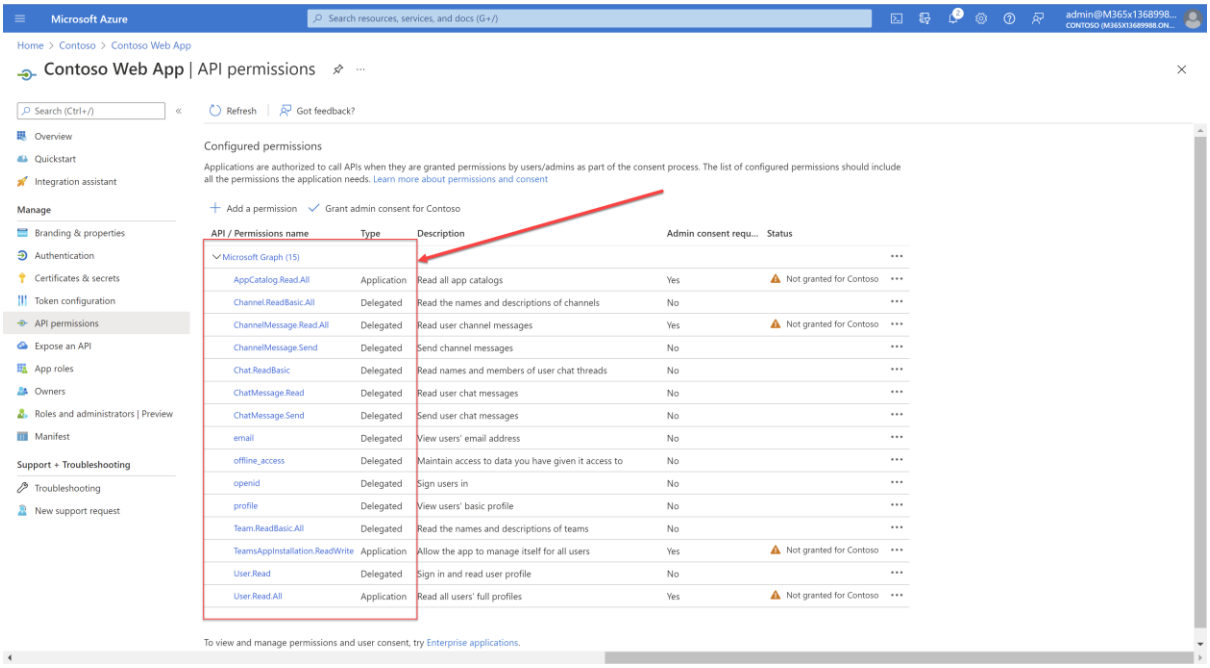
22. Select **Microsoft Graph**



23. Select **Application permission**, then **search for and select AppCatalog.Read.All, TeamsAppInstallation.ReadWriteSelfForUser.All & User.Read.All.** Select **Add permissions** once all these scopes have been selected.



24. If successful, your API permissions should now look like the below, and include both the delegated and application permission type scopes:



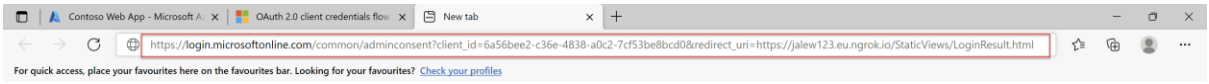
25. We now need to consent to these scopes in our application, as the Talent Management app utilises some of these scopes for proactive messaging, whilst the remaining scopes are used in section 7 of this lab guide, when you start working with Postman to interact with the Microsoft Graph API. To do this, we need to generate an Admin Consent URL, that will ask for all of the scopes you have just selected in the App Registration. In order to build your admin consent URL you will need to take the following template, and replace the values in **BOLD** with the values that are in the appsettings.json file in Visual Studio:

    https://login.microsoftonline.com/common/adminconsent?client_id=**MICROSOFTAPPID**&redirect_uri=**BASEURL**StaticViews/LoginResult.html

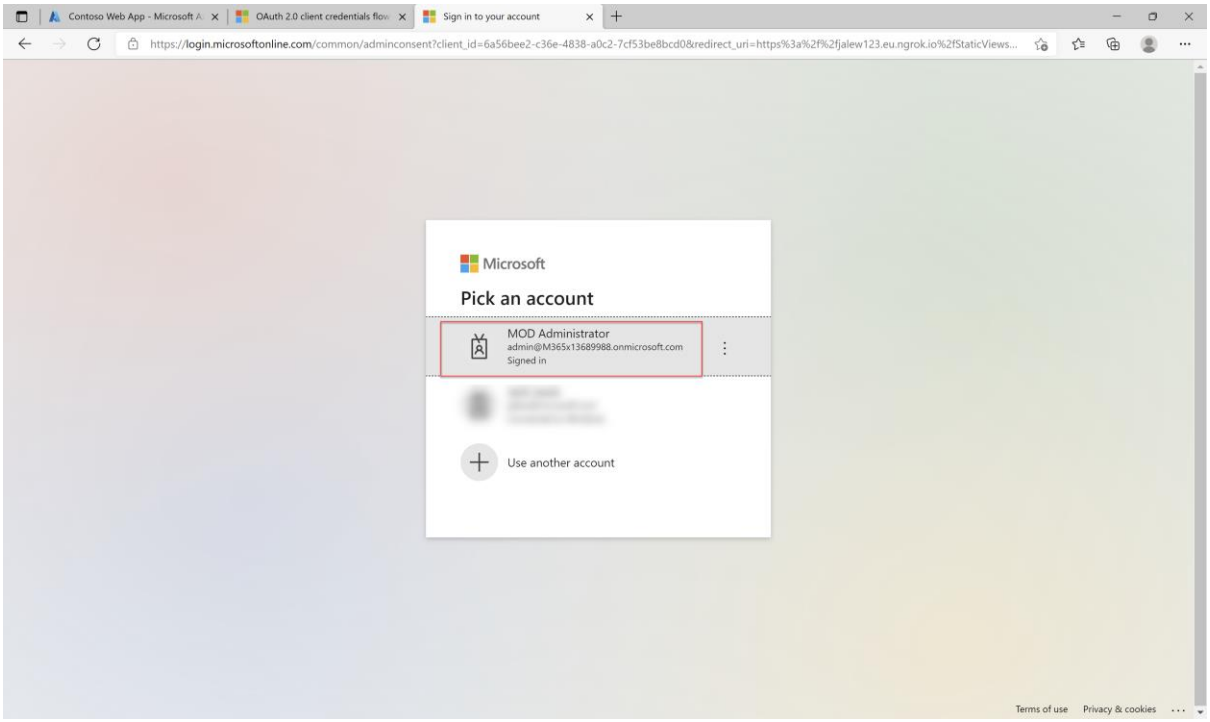
    For example, my admin consent URL looks like the following:

    https://login.microsoftonline.com/common/adminconsent?client_id=6a56bee2-c36e-4838-a0c2-7cf53be8bcd0&redirect_uri=https://jalew123.eu.ngrok.io/StaticViews/LoginResult.html

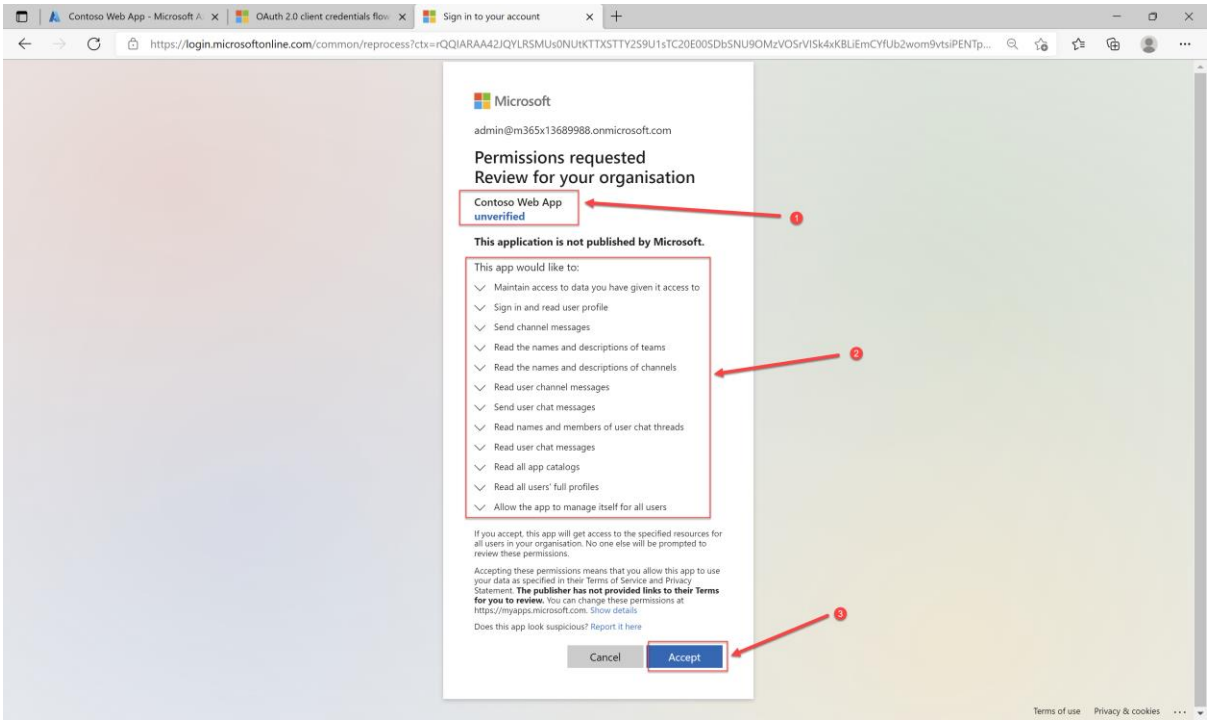
26. Once you have the admin consent URL created, open up a new tab in your preferred web browser, and go to that website.



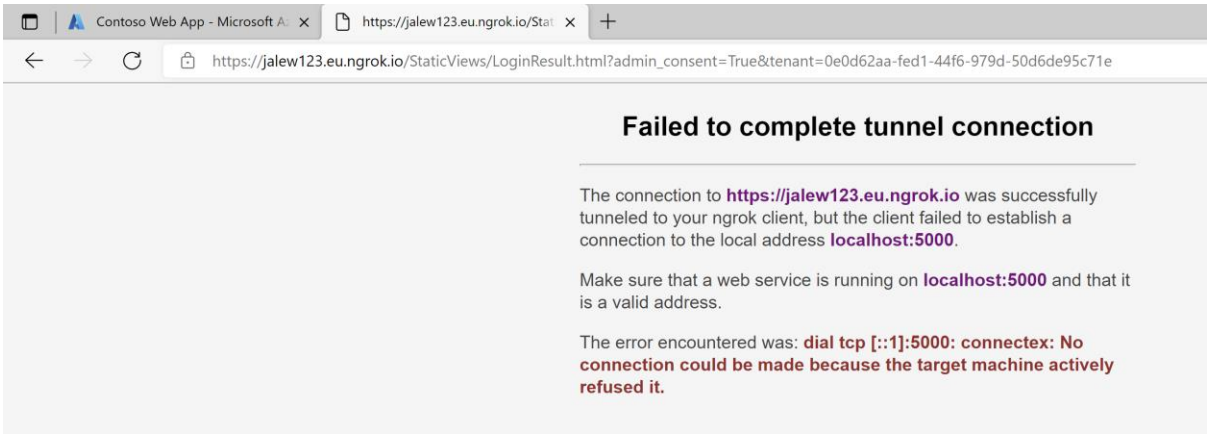
27. You will be prompted to authenticate, make sure you **sign in as a user, in your test/dev tenant who is a Global Administrator**.



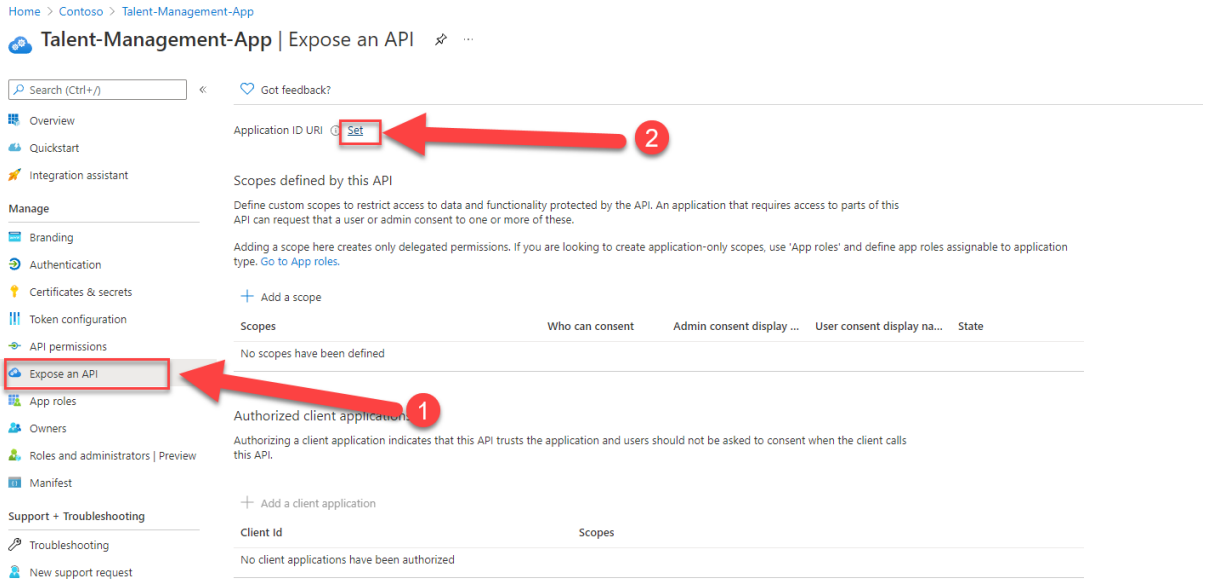
28. Review the information on the consent screen. Firstly, **make sure that your App Registration's name is correct**, and then **review the scopes you are being asked to consent to**, they should match the scopes that you selected in your app registration, a few steps previously and will include both delegated and app permission type scopes. Once you are happy with this, select **Accept**.



29. You will then be redirected back to your application, but as the app isn't running, this will fail and is expected. In a real world scenario, you would be redirected back to a web-page, that is able to process the consent being granted (or not granted) and then do something with that consent (such as start to install apps, bots, send chat messages, etc…).

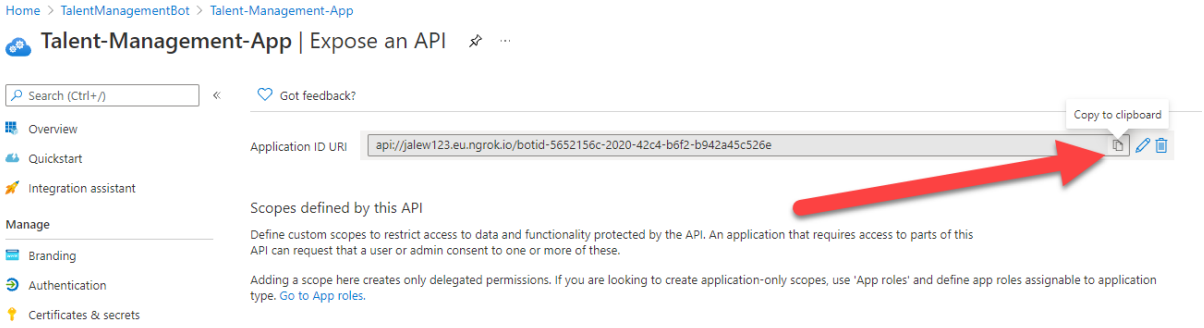


30. Now that we have consent in place, go back to the Azure Portal, and back into your App Registration. Select **Expose an API** and **Set** the Application ID URI.

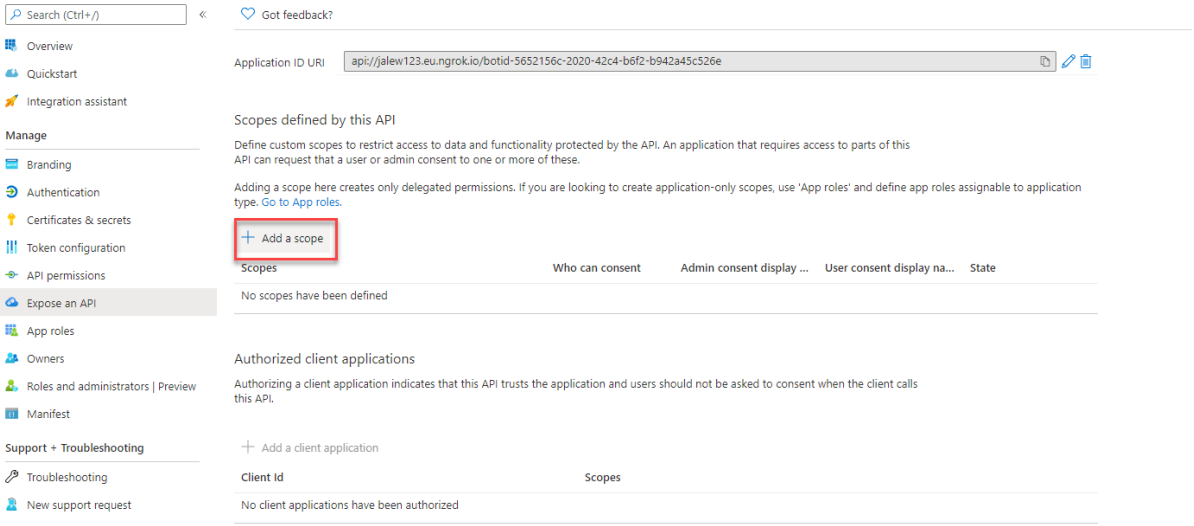


31. The Application ID URI needs to be configured in the following format: **api://NGROKURL/botid-MICROSOFTAPPID** – click **Save.**

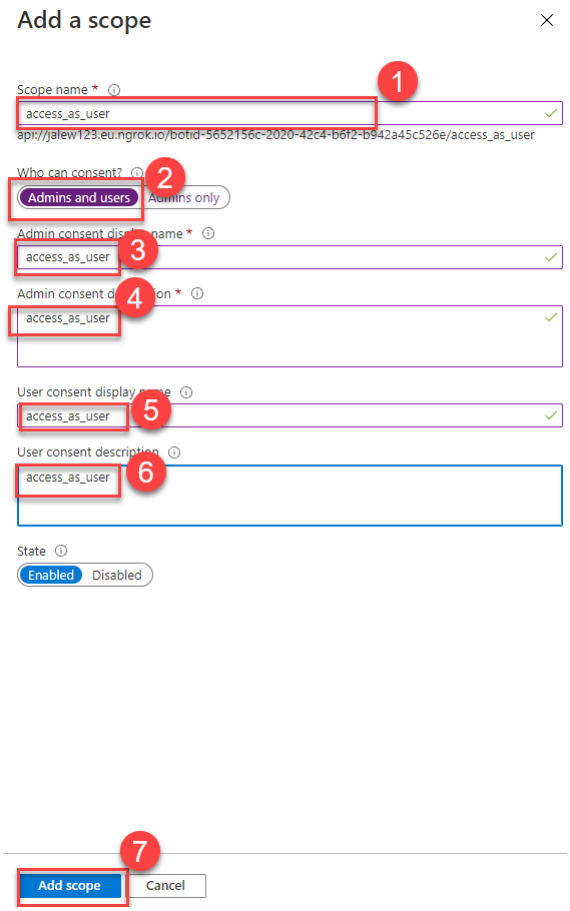
32. Once saved, copy the **Application ID URI** and enter this as the **ApplicationIdUri** value in appsettings.json (line 7).



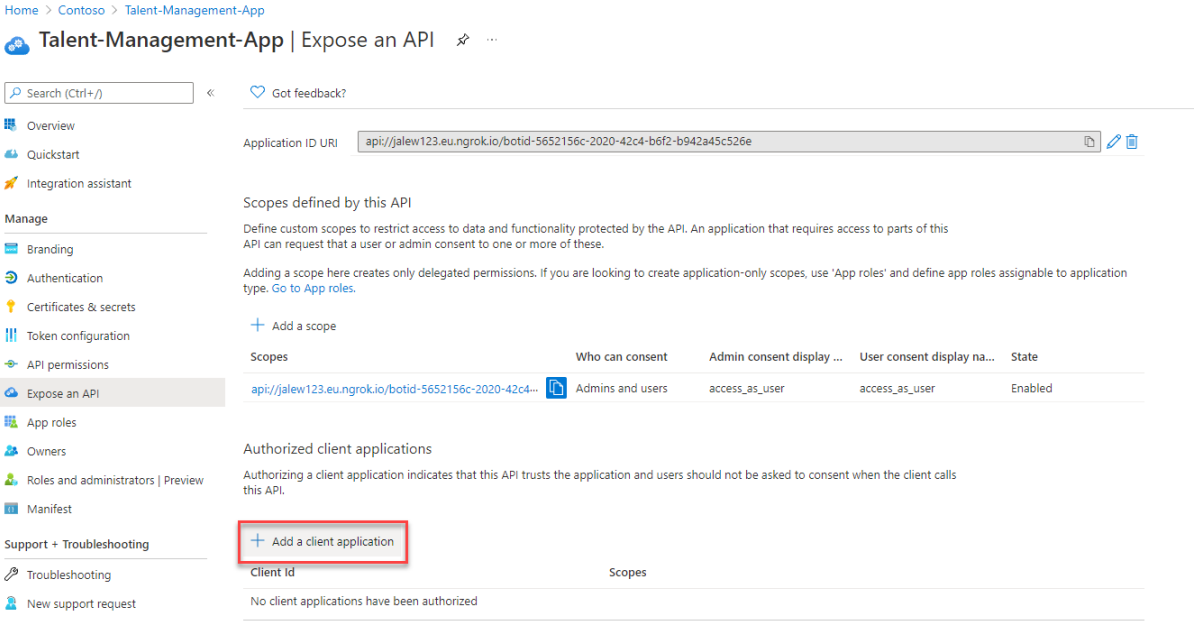
33. Select **Add a scope**



34. We now need to configure the API scope that will be used when SSO into your application has been completed both inside and outside of Microsoft Teams. Enter the Scope name **access_as_user** select **Admins and users**. For the display names and descriptions enter **access_as_user** (note: in a real world scenario you would populate these with more information). Select **Add scope**

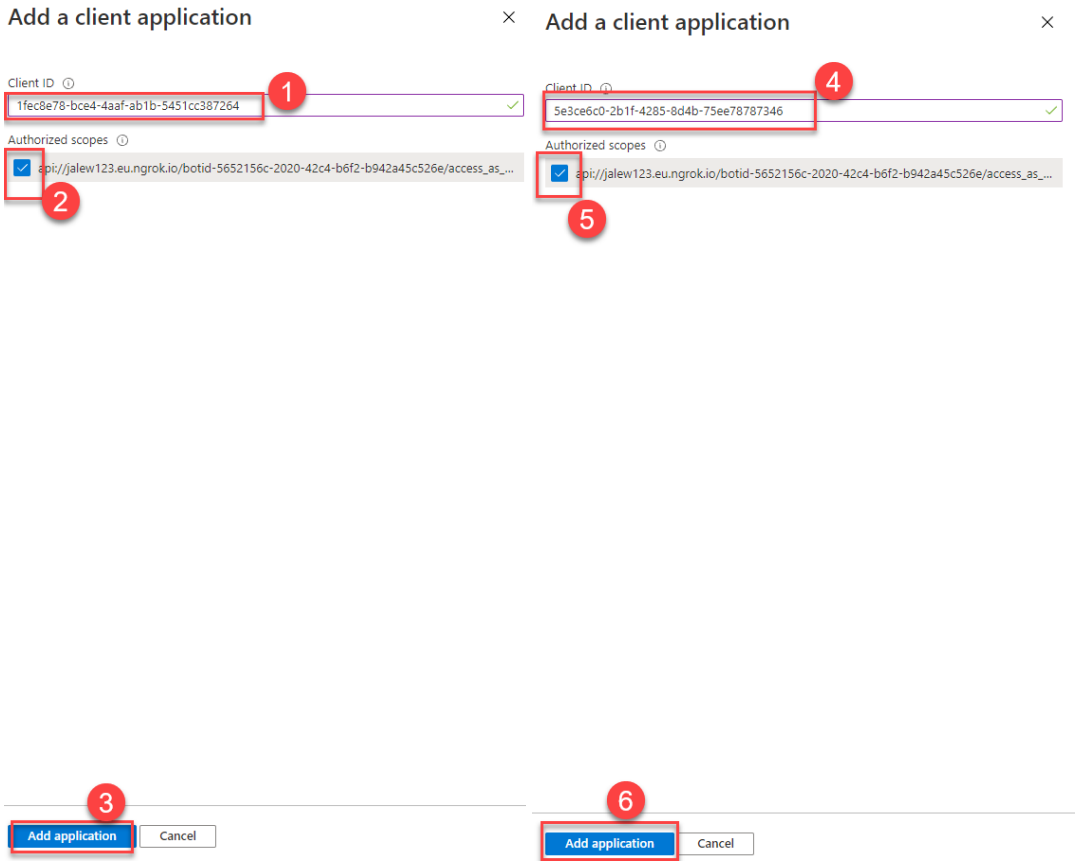


35. We now need to enable the Teams Native/Mobile and Web applications to swap their access tokens for an access token for the Talent Management Application. Select **Add a client application.**



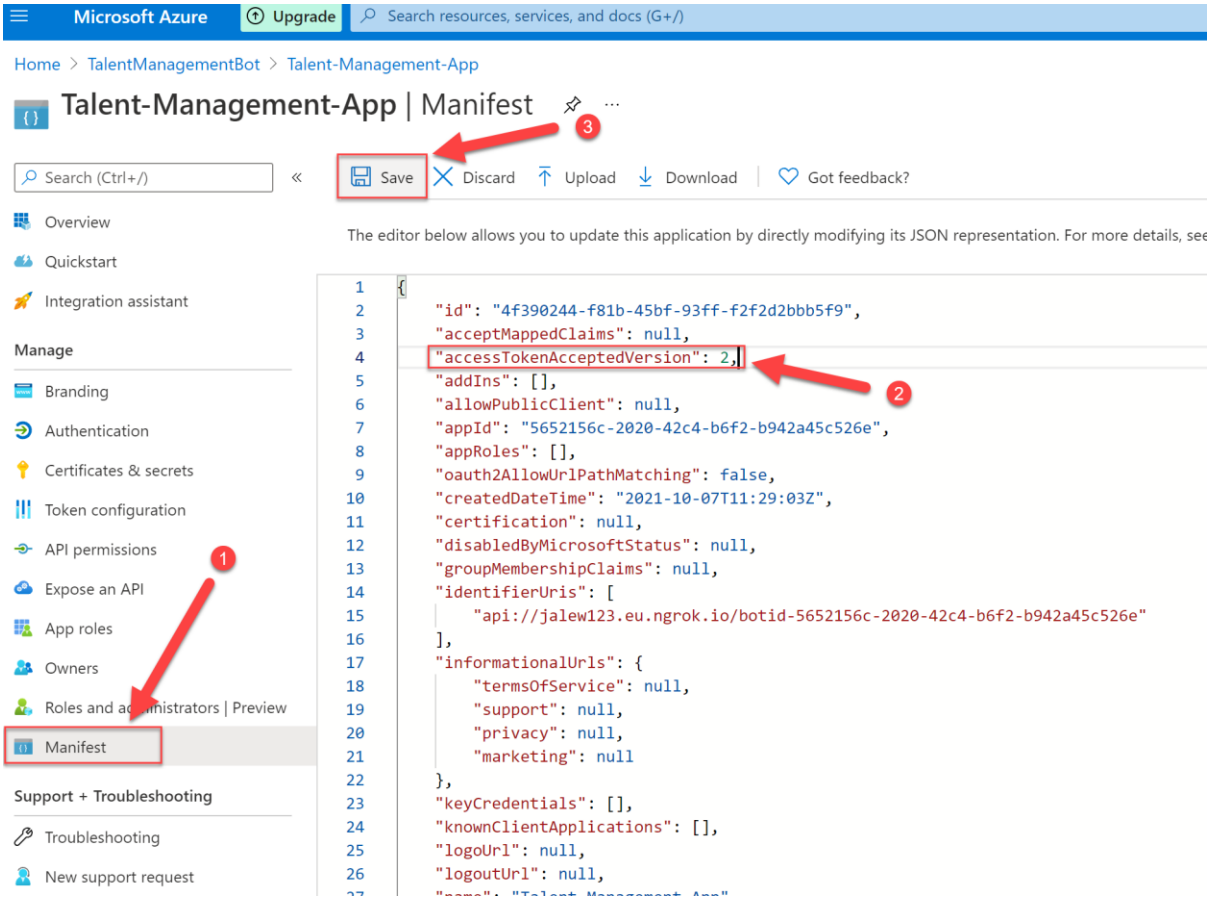
36. Teams is identified by 2 separate Client IDs, so we need to repeat this step twice. Enter **1fec8e78-bce4-4aaf-ab1b-5451cc387264** as the Client ID, select the **access_as_user** scope and click **Add Application.** Then click **Add a client application** again, enter

5e3ce6c0-2b1f-4285-8d4b-75ee78787346 as the Client ID, select the **access_as_user** scope and click **Add Application.**

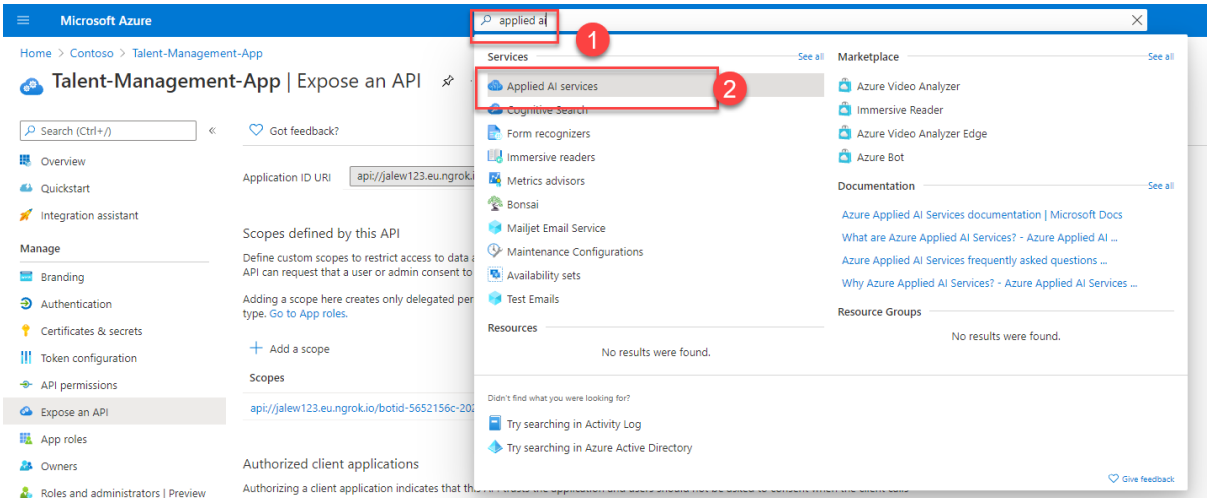


You have now authorised the Teams Apps to swap their access tokens for an access token for you application. More on this later in the lab!

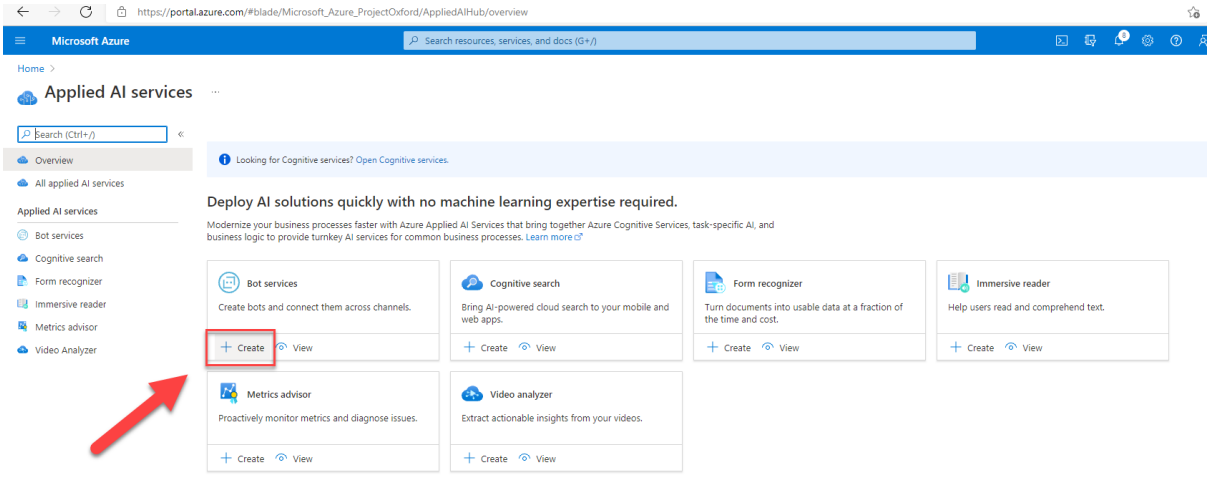
37. The final step we need to do, is to configure the App Registration manifest to only use Access Tokens v2. Select **Manifest**, change the value of **accessTokenAcceptedVersion** to **2** (line 4). Select **Save**.



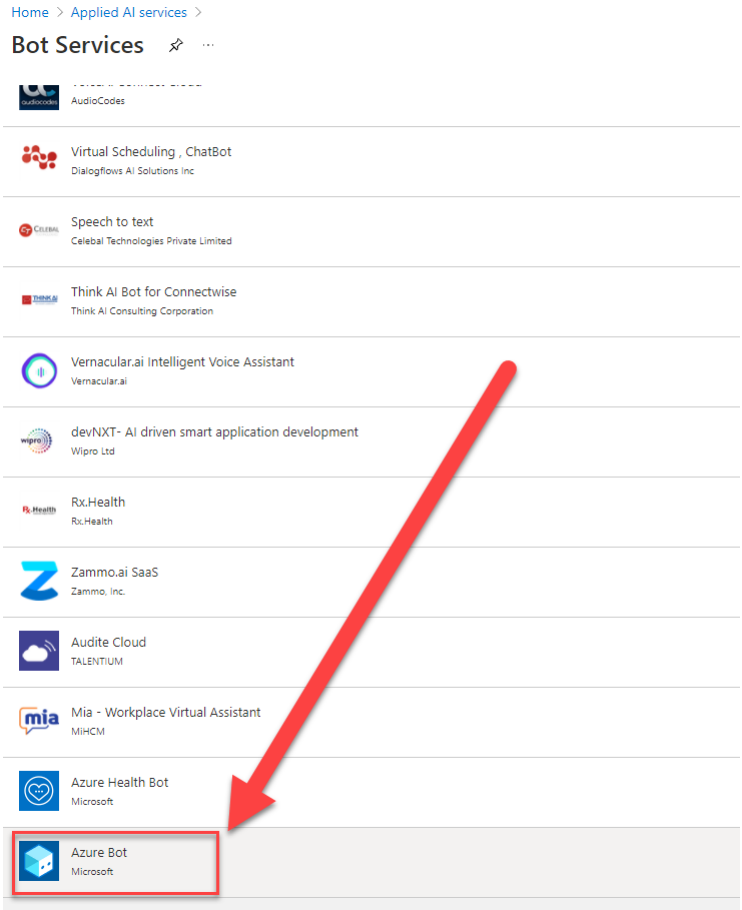
38. The Azure AD App Registration has now successfully been configured. We now need to configure the Azure Bot Services resource, that is linked to the Azure AD App Registration you have just created. In the Search bar at the top of the Azure Portal, search for and select **Applied AI services**.



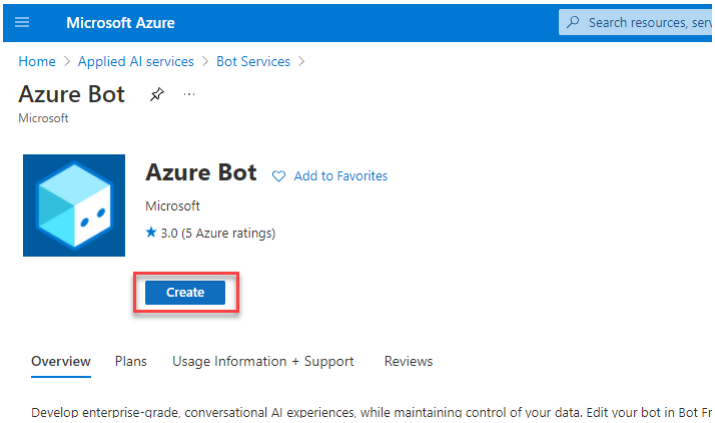
39. Find Bot Services and click **Create**.



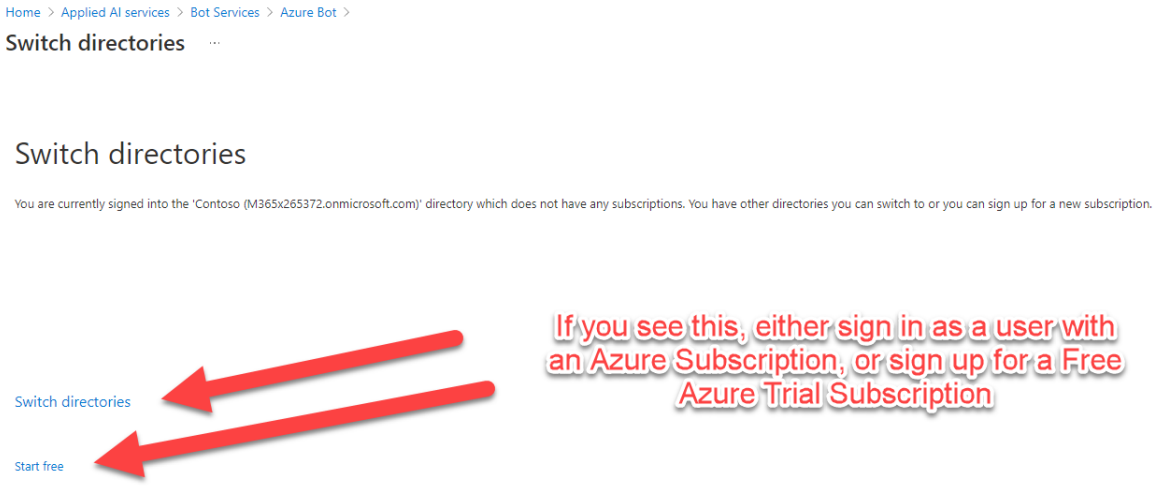
40. Scroll down and select **Azure Bot** (note: you may need to select **load more** at the bottom of the page to see Azure Bot!)



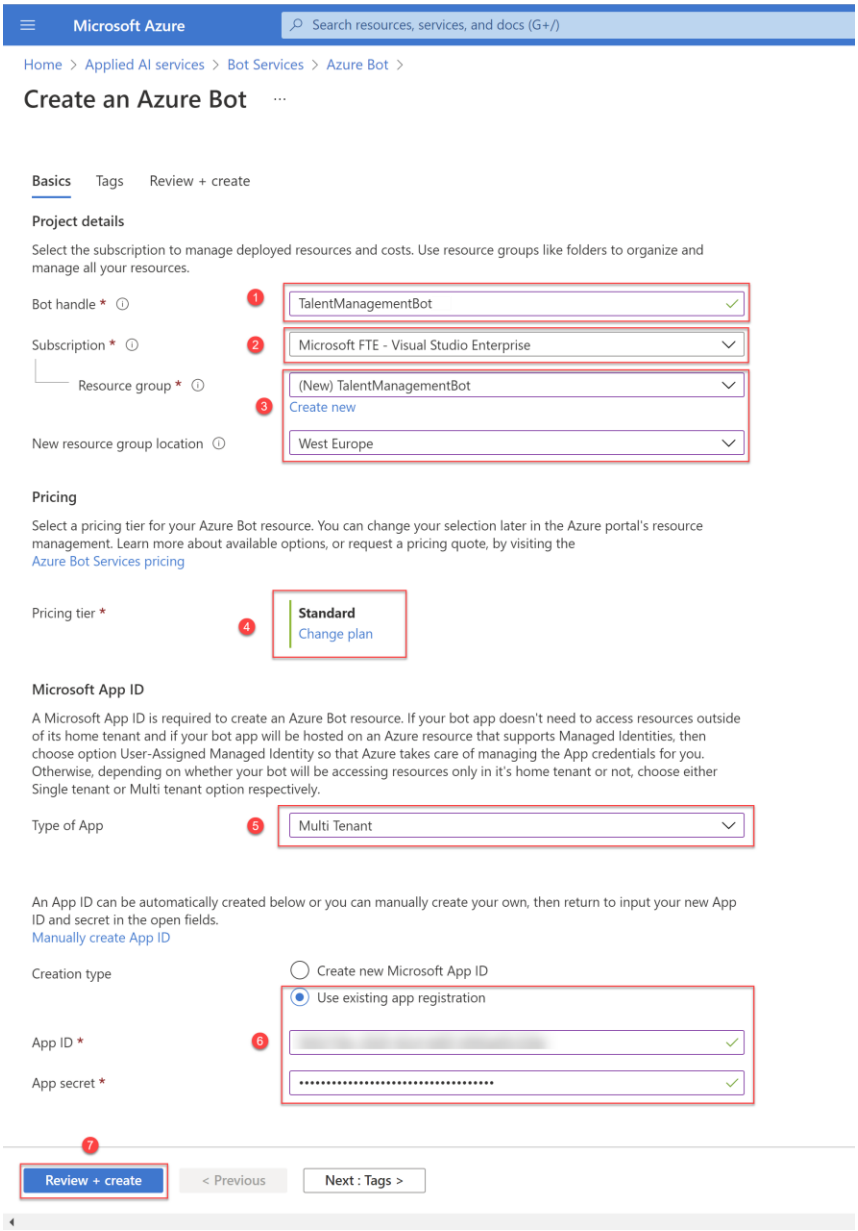
41. Select **Create**



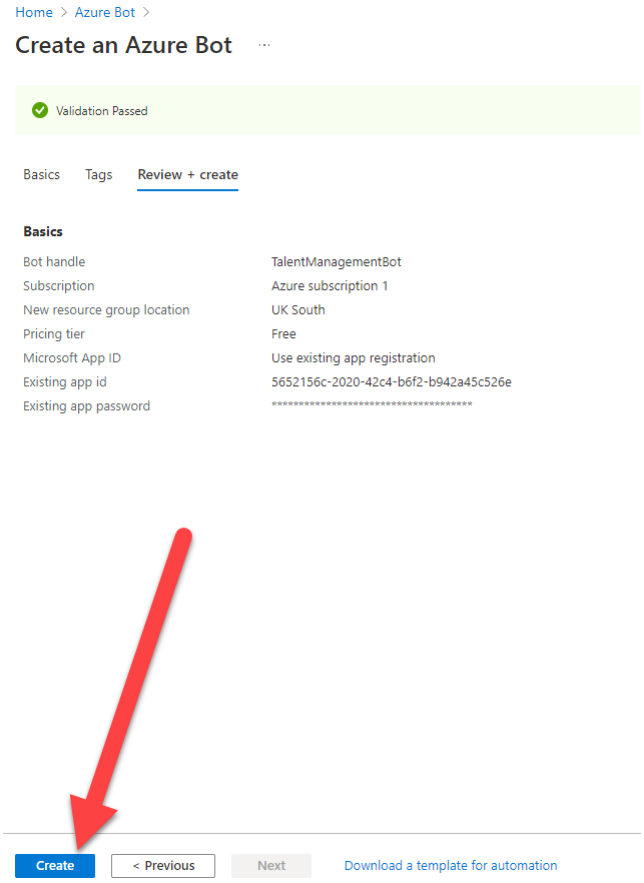
42. **At this point, you may see the following error page** – this is because the user you are logged in as has no access to an Azure Subscription with funding/credits. If you see this, either sign-in as another user with access to an Azure Subscription or sign up for an Azure Free trial!



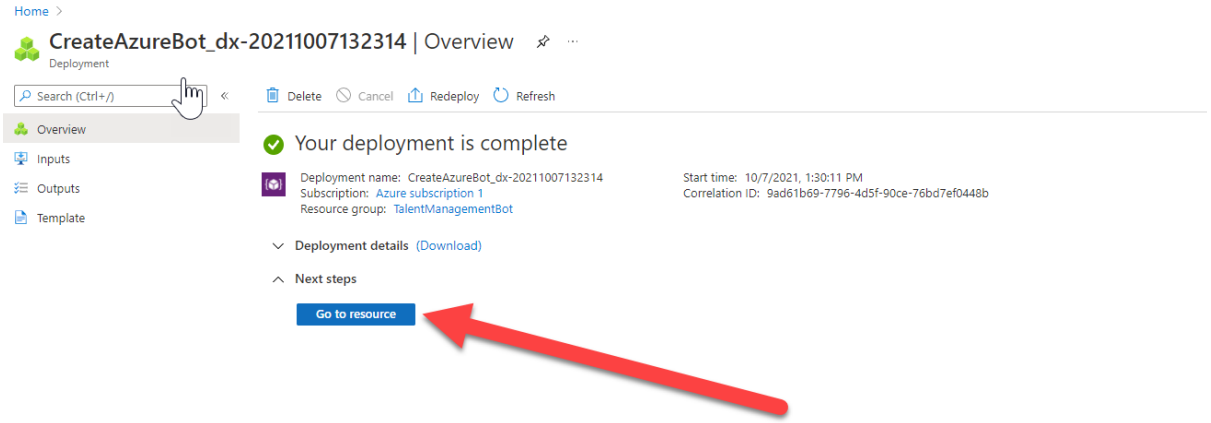
43. Assuming you have the Azure Subscription in place, you will see this interface. Enter a Bot handle name such as **TalentManagementBot** (note: these must be globally unique). Select an **Azure Subscription**, create or select an existing **Resource Group** for the Azure Bot to reside within. Select a **location** for the resource group if you are creating a new resource group. The pricing tier can remain for **standard** as the Teams channel doesn't incur any costs (only premium channels are charged). In type of app, select **Multi Tenant**. For the Microsoft App ID, it's important that you select **Use an existing app registration**, enter your **Microsoft App ID** and **App Secret** that you generated earlier in this lab. Select **Review + Create**.



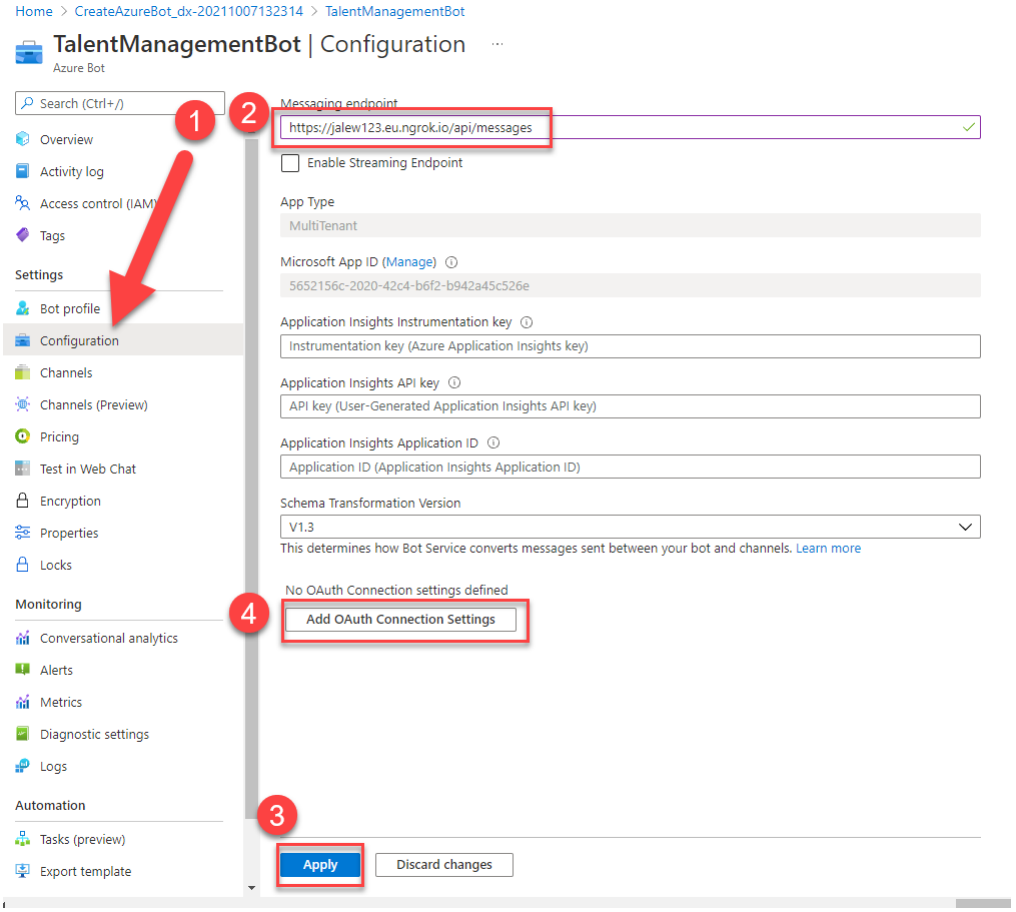
44. Once Validation has passed, select **Create**.



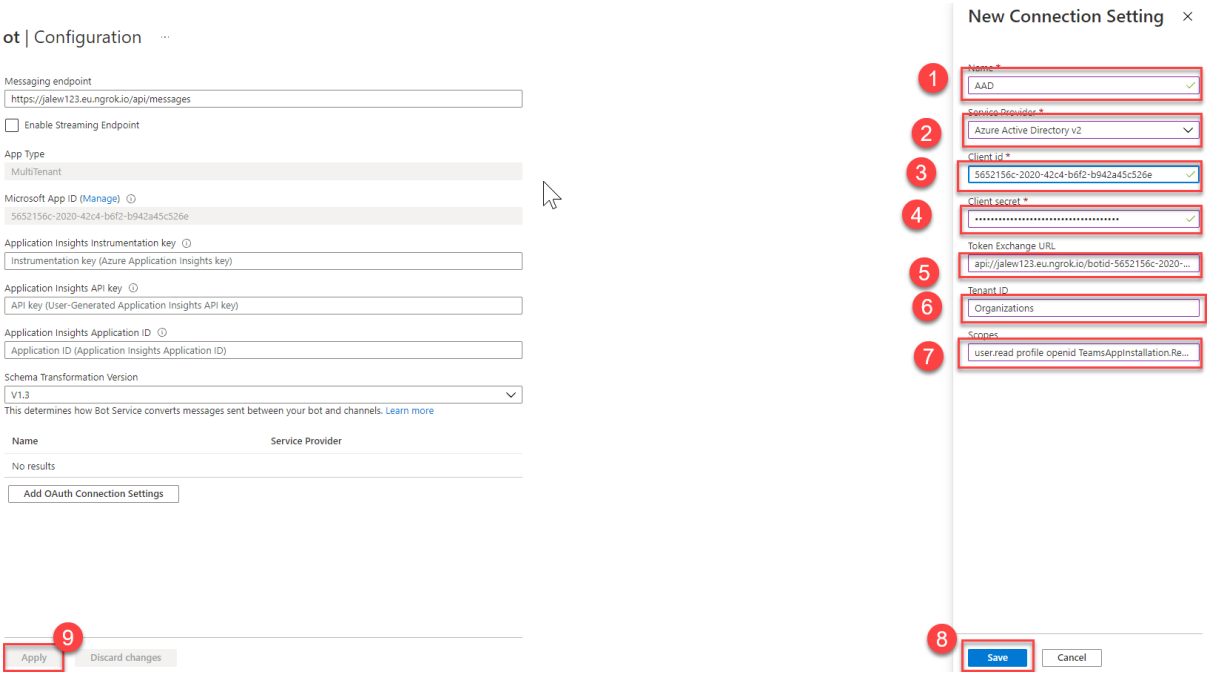
45. Select **Go to resource** once the resource has been deployed



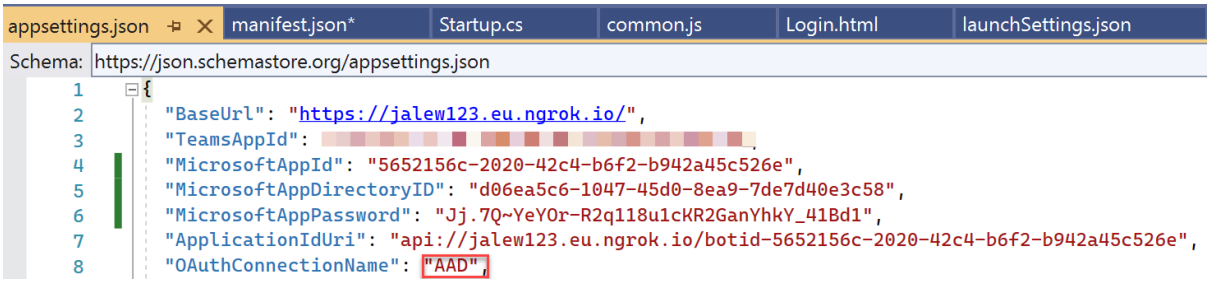
46. Select **Configuration**, enter the **Messaging endpoint** in the format of **NGROKURL/api/messages –** select **Apply** and then click **Add OAuth Connection Settings**.



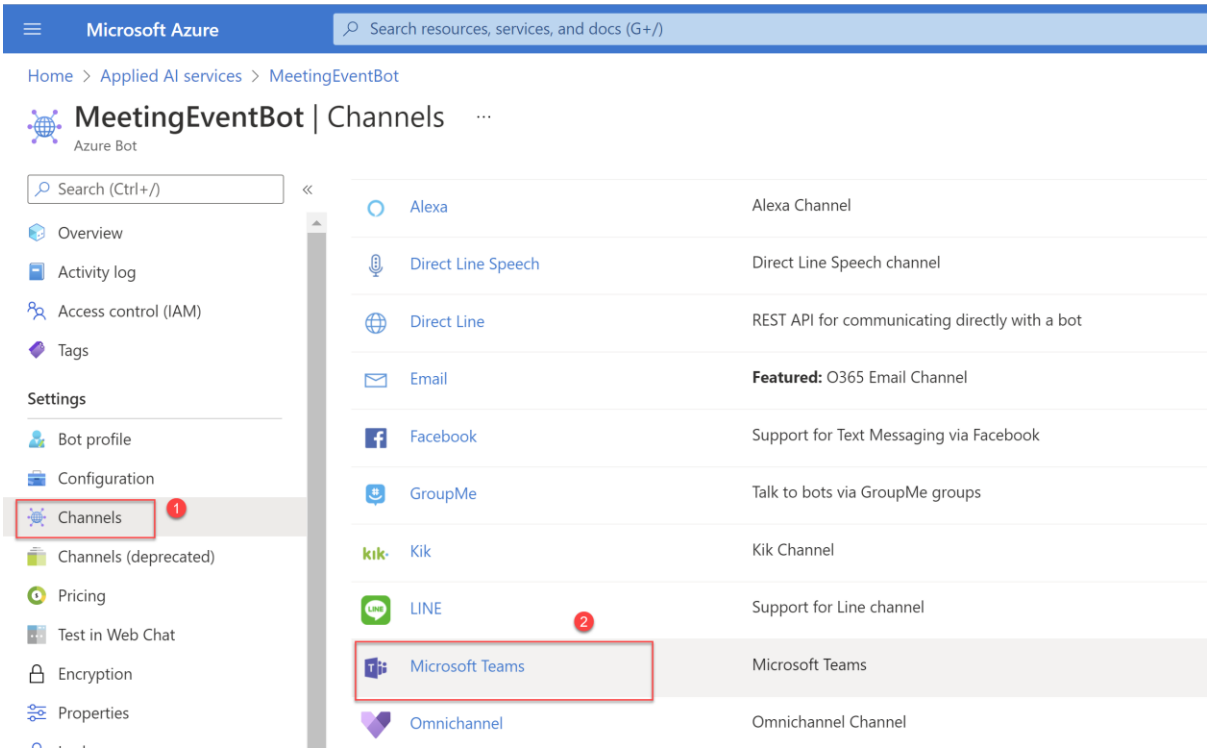
47. Enter the name as **AAD**, select **Azure Active Directory v2** as the Service Provider, the Client ID is your **MicrosoftAppID**, the Client secret is your **MicrosoftAppPassword**, the Token Exchange URL if your **ApplicationIDURI,** Tenant ID is **Organizations** and Scopes is **user.read profile openid TeamsAppInstallation.ReadForUser –** click **Save** and **Apply.**



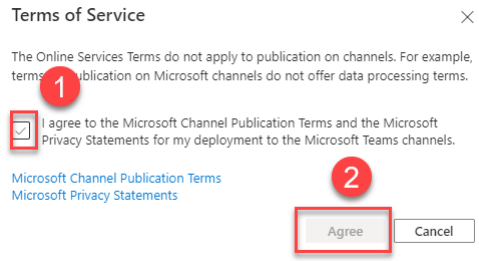
48. Back into Visual Studio, set the **OAuthConnectionName** value in appsettings.json (line 8) to **AAD –** click CTRL+S to save the appsettings file.



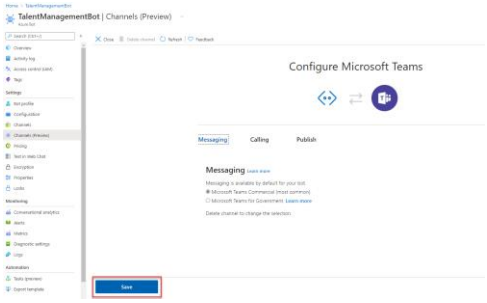
49. Now go back into the Azure Portal and configure the channels for the bot to support Teams. Select **Channels** and select **Microsoft Teams.**



50. Tick the **checkbox** to agree to the terms of service and select **Agree**.

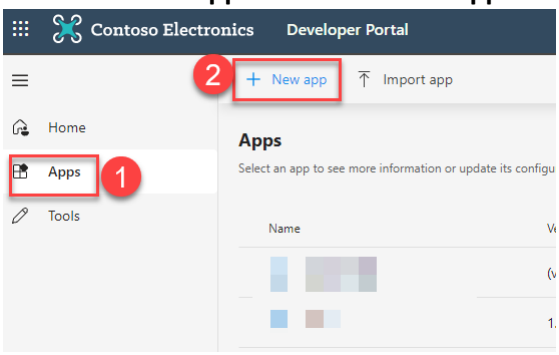


51. Click **Save**.

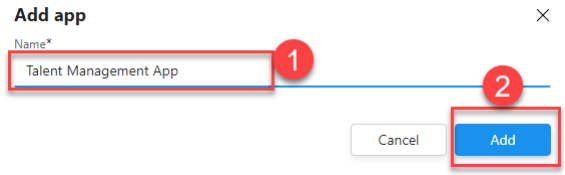


52. Your Bot is now configured to forward messages received onto your App (via the NGROK URL) and is configured to work in Teams. The final step, before running the app locally, is to

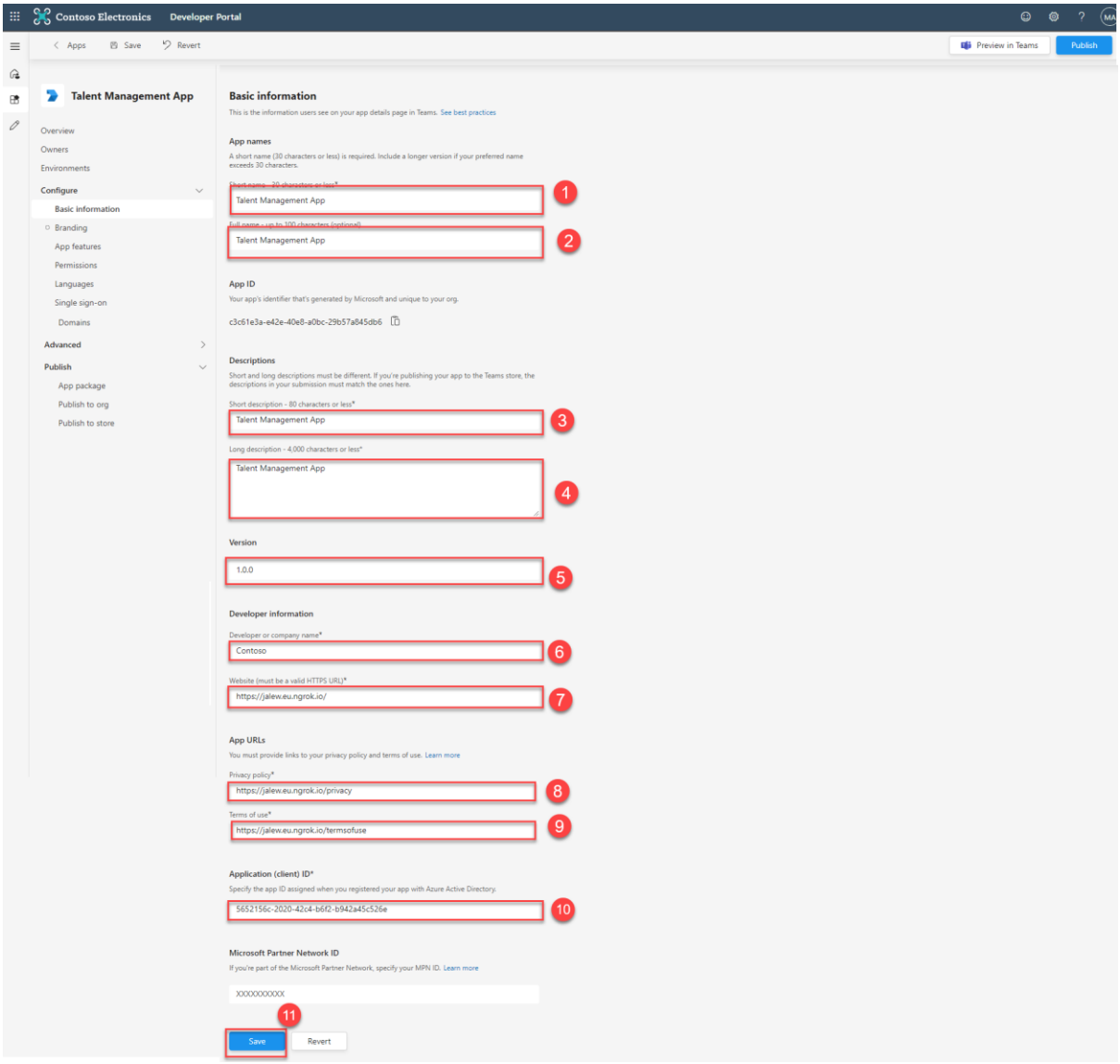
create a Teams App Manifest and populate the final value (TeamsAppID) in the appsettings.json file. To do this go to the **Teams Developer Portal** https://dev.teams.microsoft.com/ and sign in with the Test/Dev Tenant Administrator account. Select **Apps** and click **+ New App.**



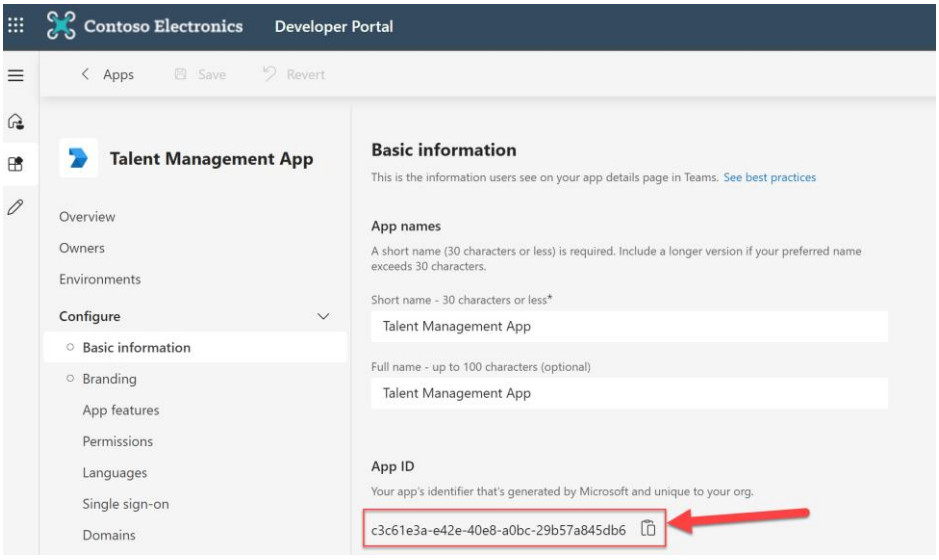
53. Enter the name **Talent Management App** and click **Add.**

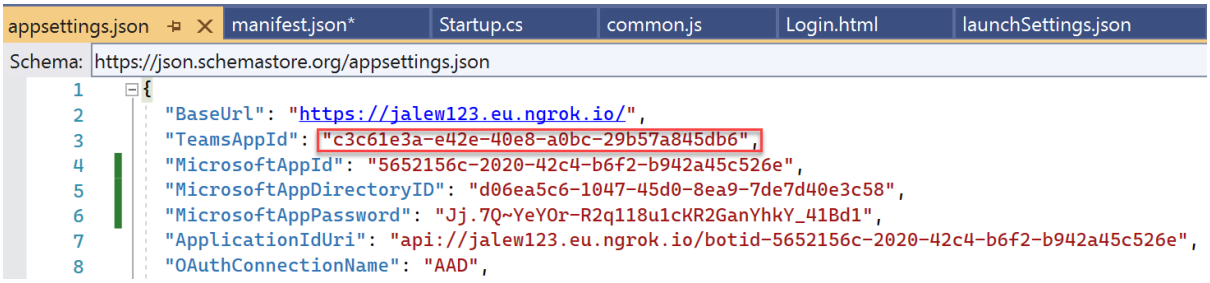


54. We now need to enter some placeholder values into the Basic Information configuration of your Teams app. Start with copying the Short Name **Talent Management App**, and paste it into the **Full Name, Short Description and Long Description** fields. Set the version as **1.0.0**. Developer to **Contoso**. Website to your **NGROKURL.** Privacy Policy URL to **NGROKURL/privacy** and Terms of Use URL to **NGROK/termsofuse.** For Application (Client) ID, enter your **MicrosoftAppID**. Click **Save**.

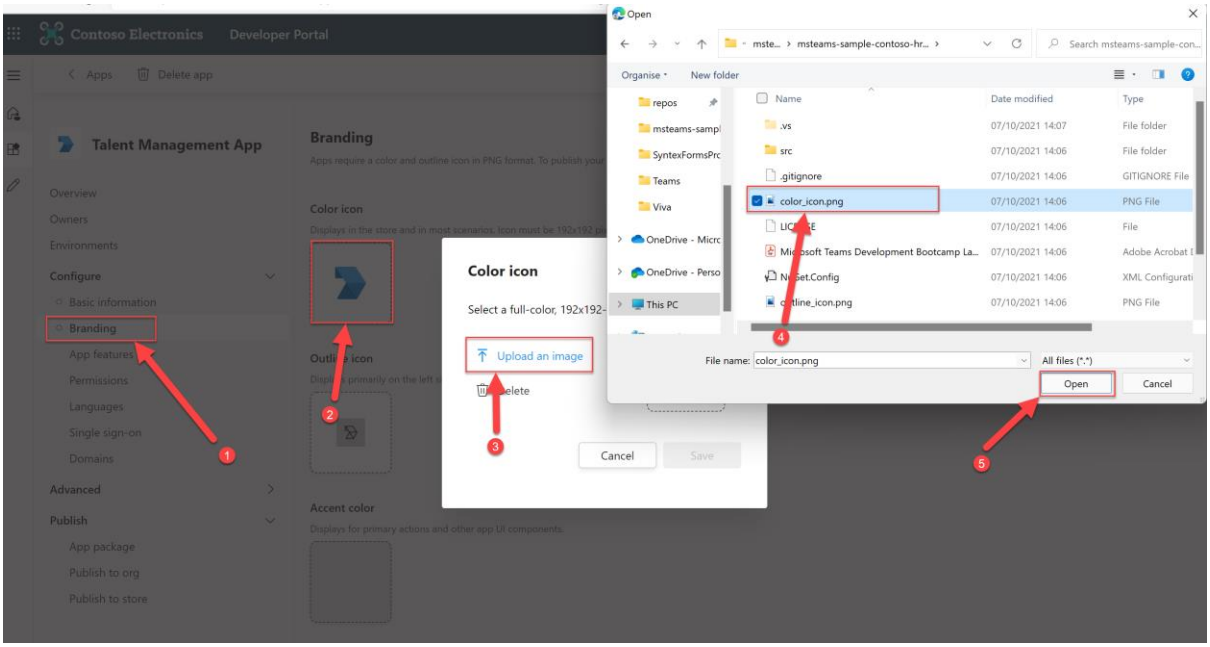


55. Copy the **App ID** and paste this into appsettings.json as the value of **TeamsAppID** (line 3). Save appsettings.json.

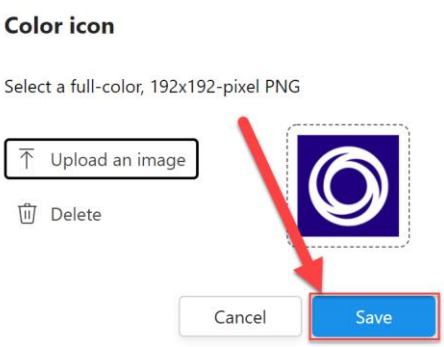




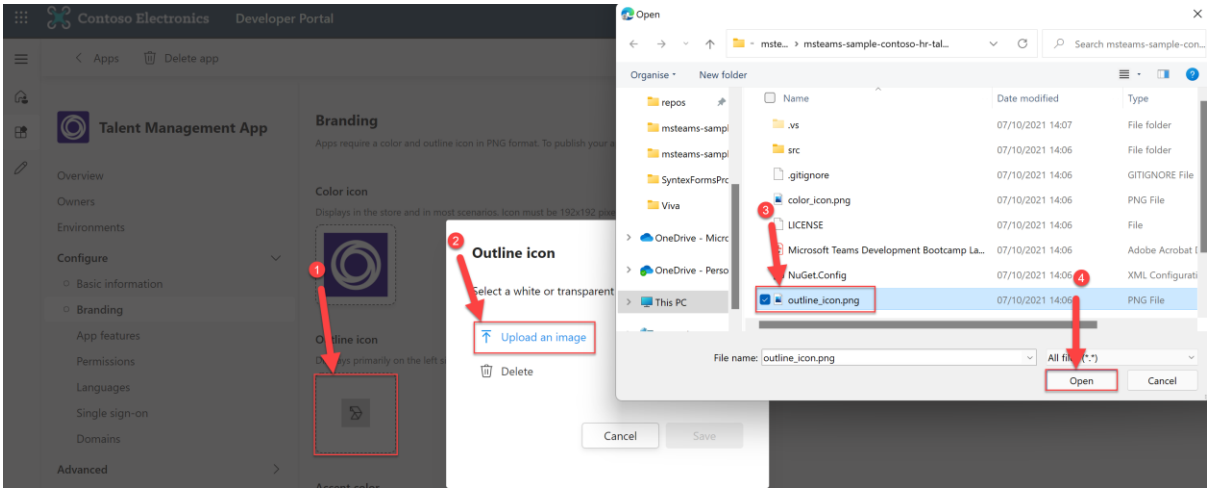
56. Go back into the Teams Developer Portal, and select **Branding,** click the **Colour icon**, select **Upload an image**, navigate to the folder that contains the application you downloaded from GitGub, select the **color_icon.png** file and click **Open**



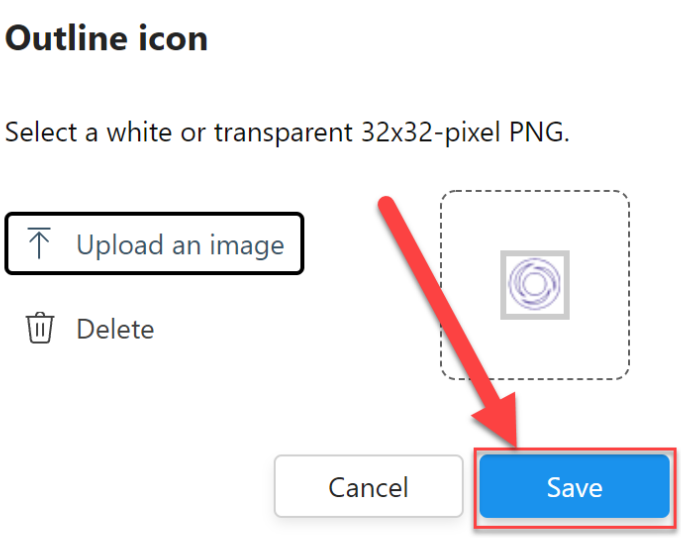
57. Click **Save**



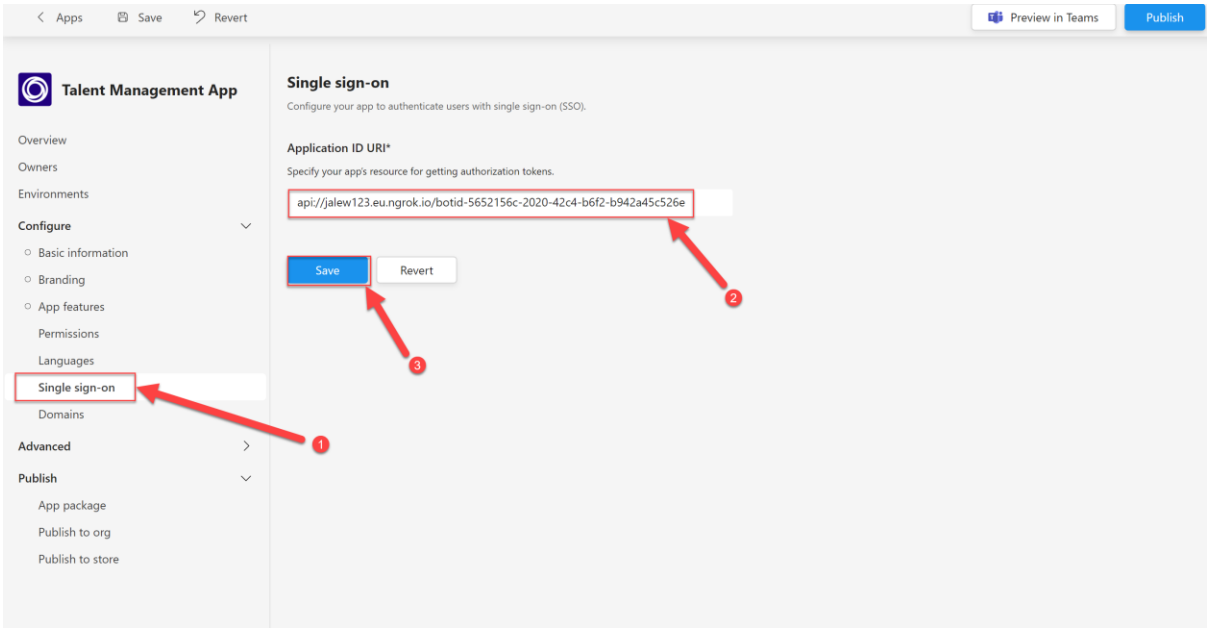
58. Click the **Outline icon,** select **Upload an image**, navigate to the folder that contains the application you downloaded from GitHub, select the **outline_icon.png** file and click **Open**



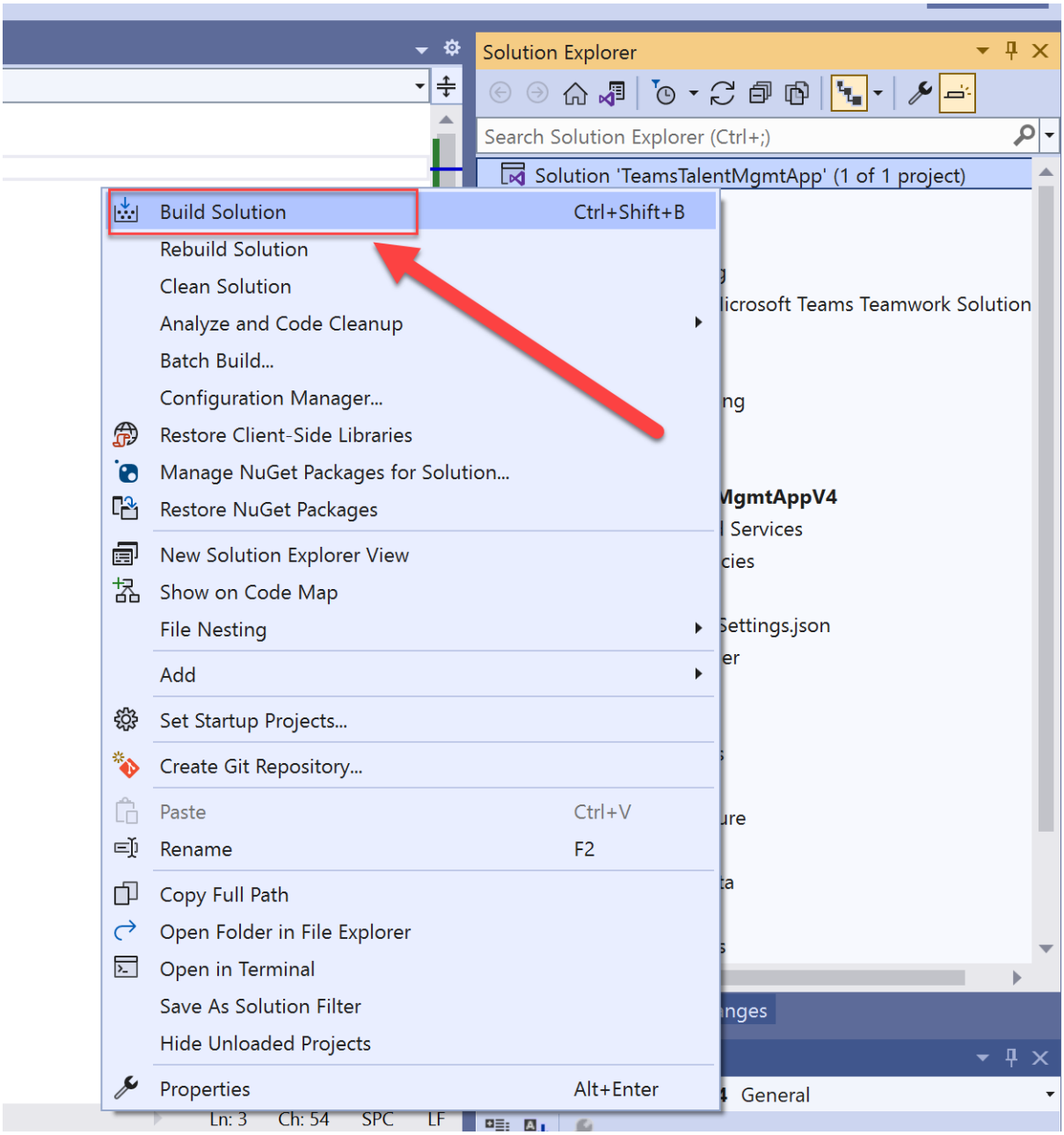
59. Click **Save**



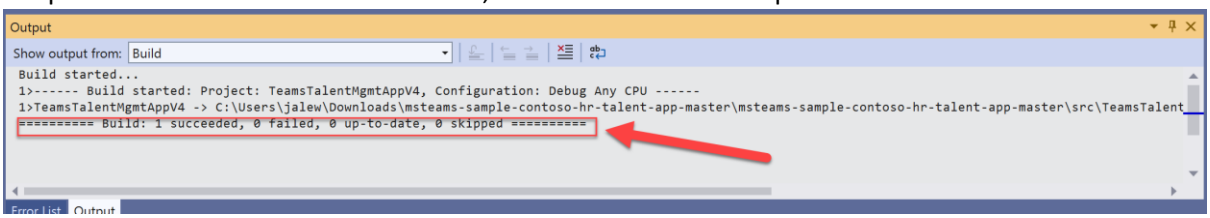
60. The final step to configure in the Manifest, during the initial setup, is the Single sign-on properties. Select **Single sign-on**, enter the **ApplicationIdURI** value into the textbox and click **Save**.



61. Now that the manifest has been configured and all 7 settings in appsettings.json have values associated to them, we are in a position to be able to build and run the app locally, using Visual Studio. **Right click the Solution** and click **Build Solution**.

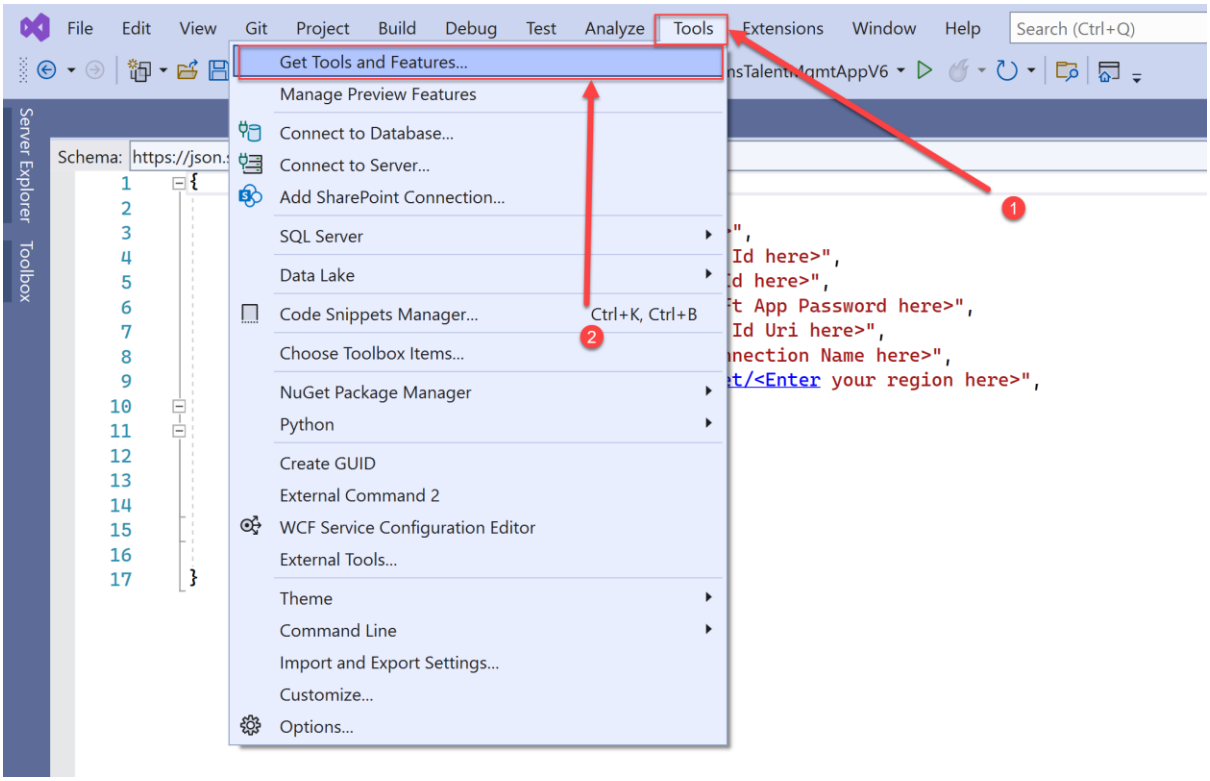


62. At the bottom of Visual Studio, the Output section will begin populating with text. Once the Output confirms that the build succeeds, move onto the next step.

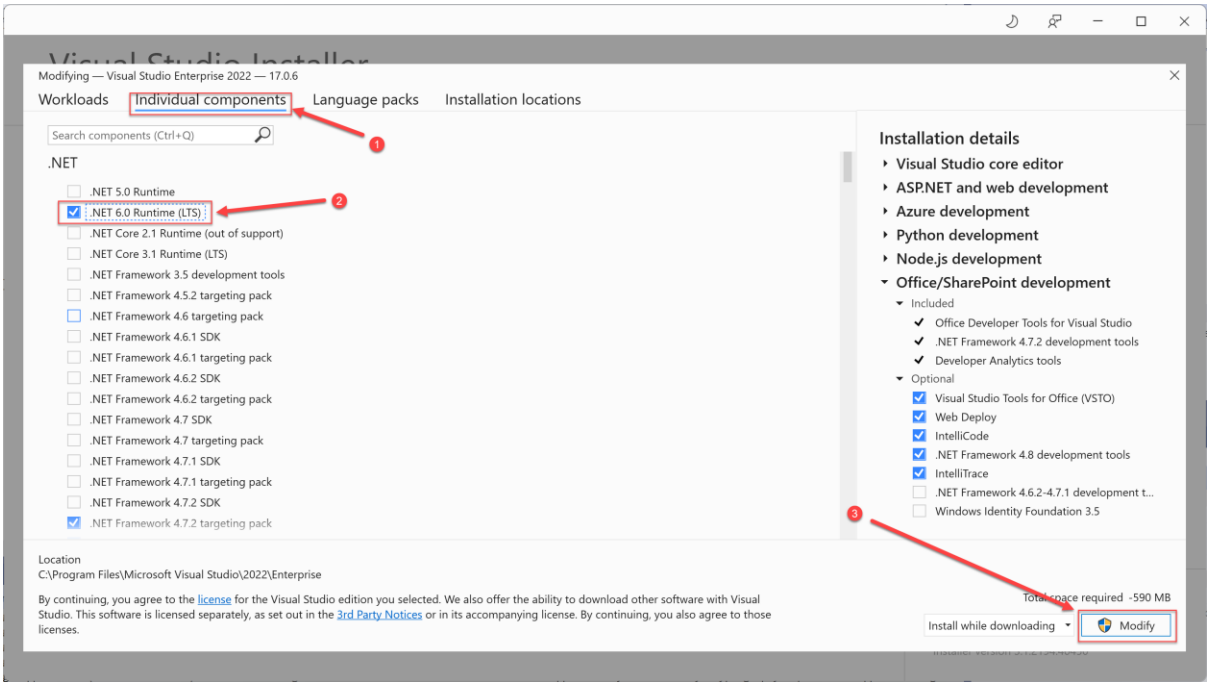


NOTE: If your build fails, it is likely caused by either; the .NET 6 Runtime not being available in Visual Studio, or the NuGet package manager settings not being setup correctly.

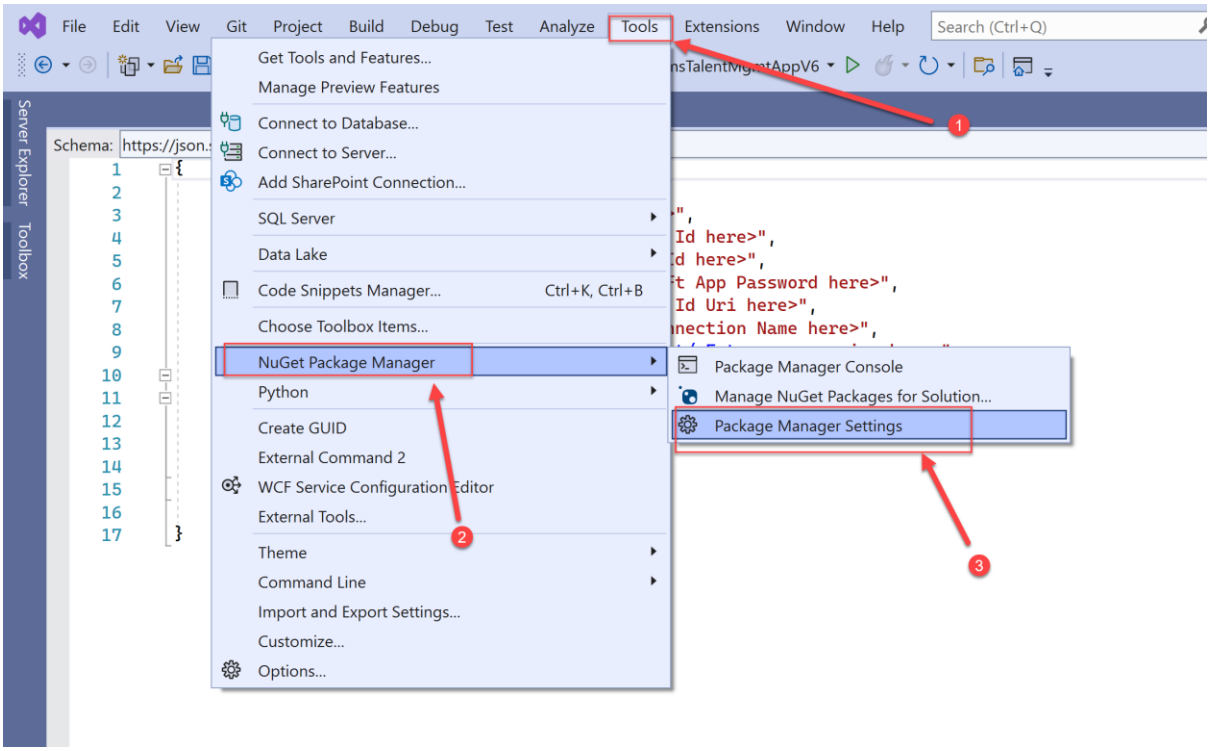
To resolve .Net Runtime issues, click Tools and select Get Tools and Features:



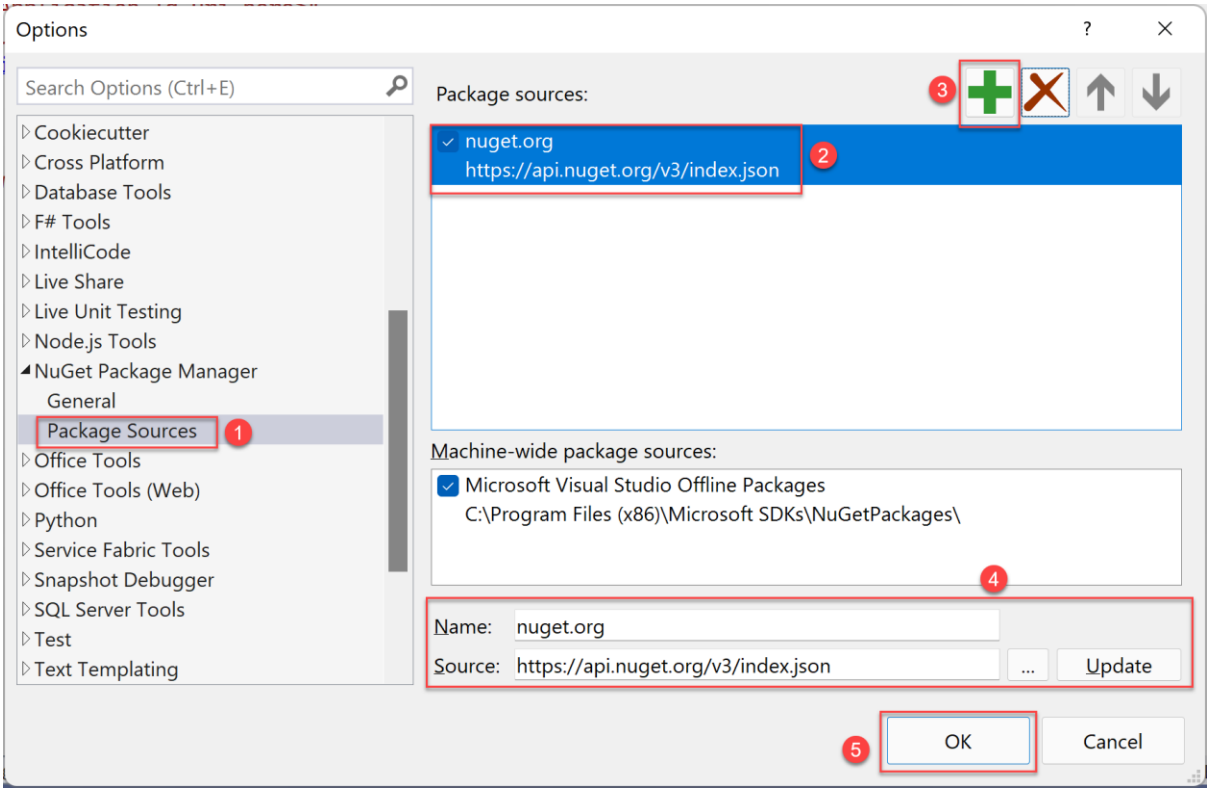
Select Individual components, click .NET 6.0 Runtime (LTS) and click Modify to install the .NET 6 runtime for use in Visual Studio.



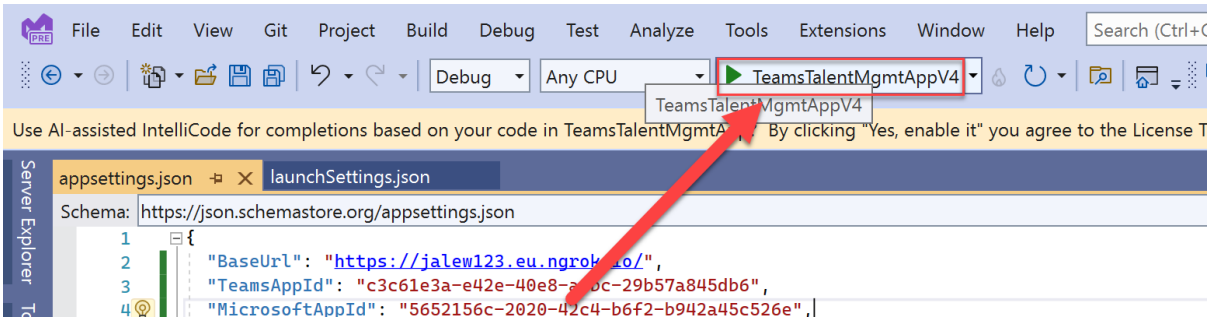
To resolve NuGet Package Manager restore issues:
Select Tools, highlight NuGet Package Manager and select Package Manager Settings



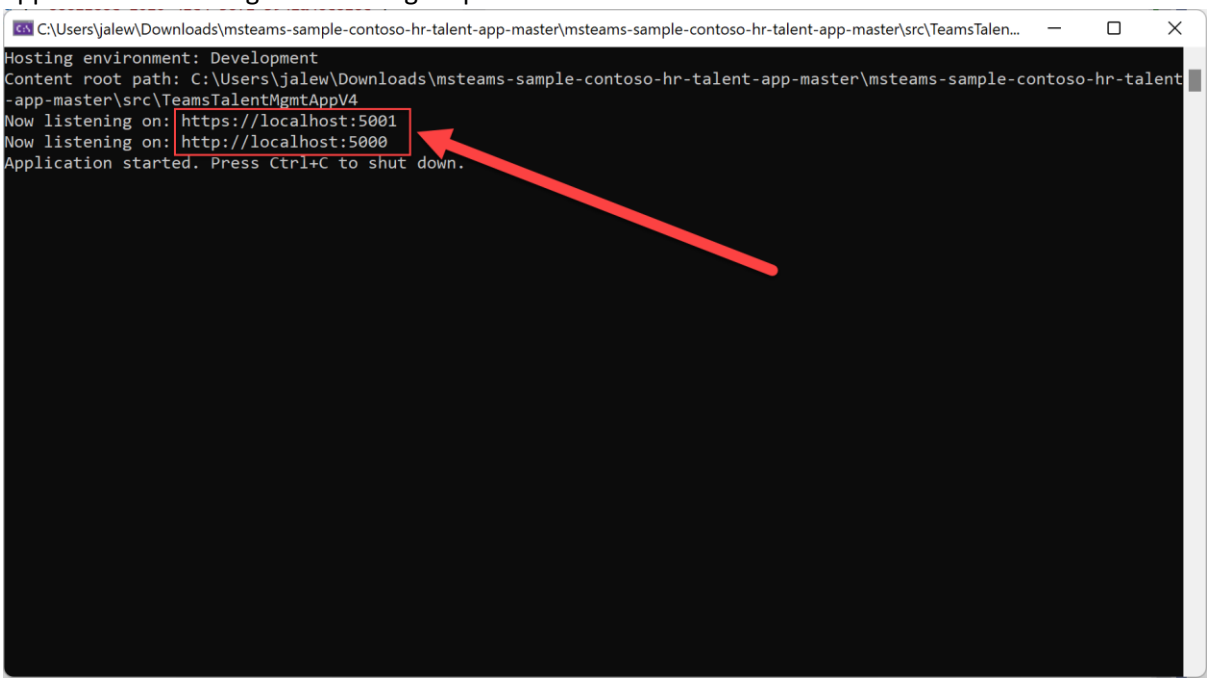
Select <u>Package Sources</u> and review the Package Sources. If nuget.org appears as it does in the below screenshot, your NuGet Package Sources are configured correctly. If not, select <u>+</u>, and enter the Name of <u>nuget.org</u> and Source of <u>https://api.nuget.org/v3/index.json</u> select <u>OK</u>.



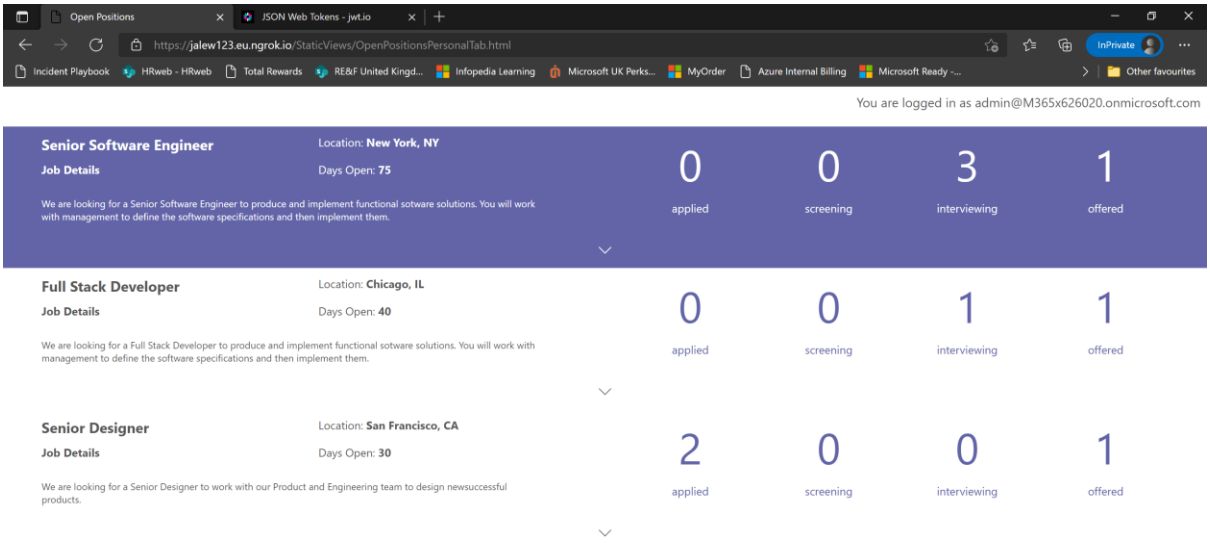
63. Click the **Play** button, to run (debug) the Talent Management App locally.



64. All being well, you should receive a message in a console window, confirming that the application is running and listening on ports 5000 and 5001.



65. The NGROK tunnel that we established earlier, should be forwarding traffic it receives on 443 (https) to localhost:5000. We can test this by going to the following URL in an InPrivate browser window: **NGROK/StaticViews/OpenPositionsPersonalTab.html** – you should then be sent through an interactive Login and Consent process via Azure AD, and presented with the following dashboard view:

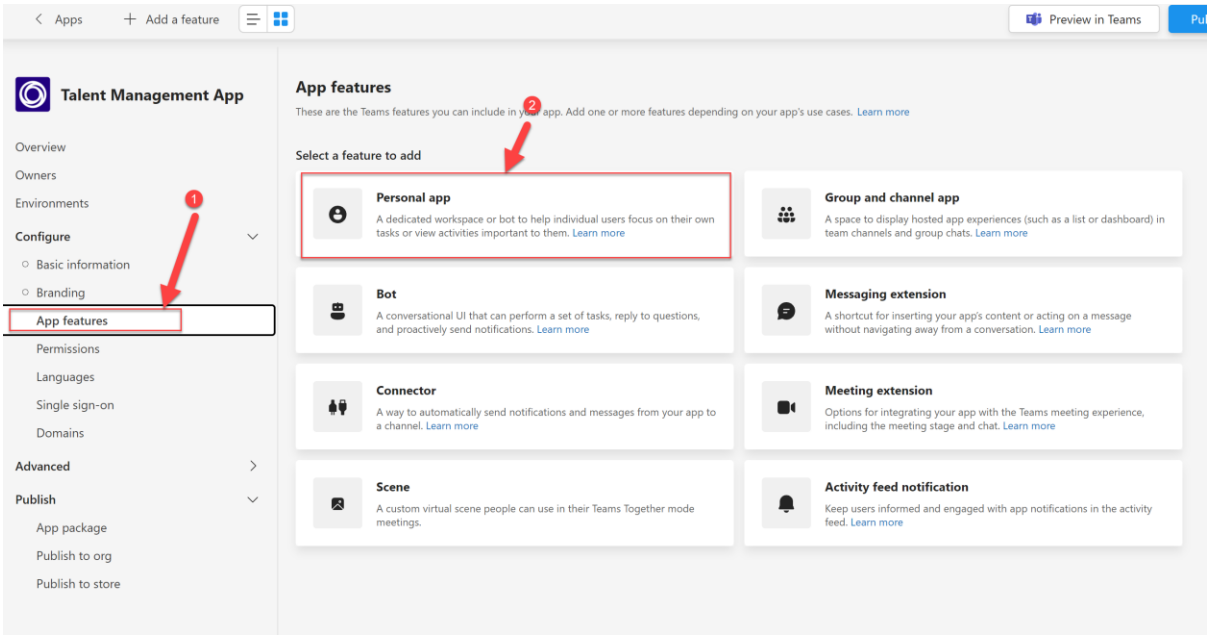


Nice job! You have setup the developer environment and Azure resources required to run this application locally. Now that the application is running and working outside of Teams. We can start to configure our Teams App Manifest to bring the apps functionality inside Teams. We'll start with this Open Positions dashboard view in the next lab!

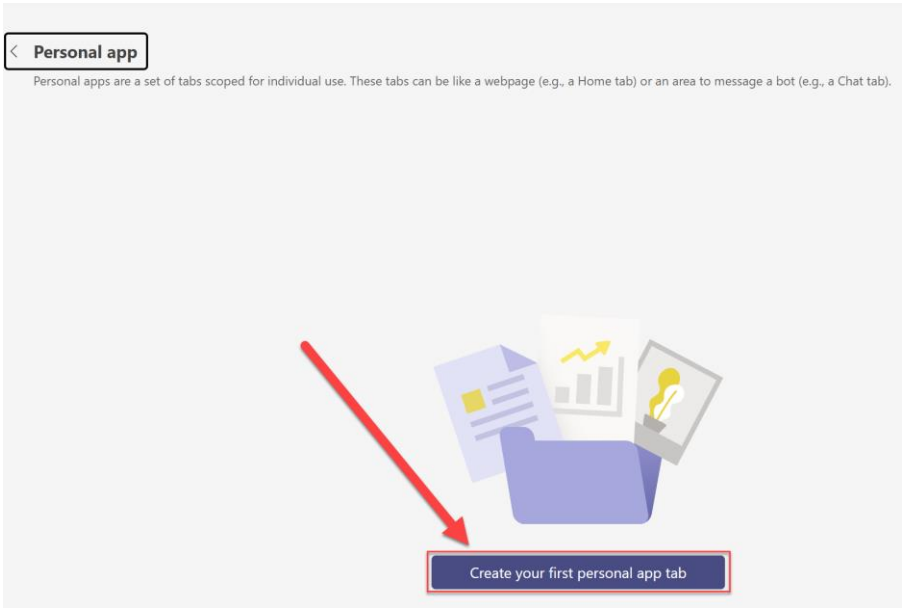
2) Implement a Tab inside a Personal App

In this step we will configure the Positions dashboard to work inside Teams as a Tab inside a Personal App. We will also see Teams SSO in action and review the code that allows this all to work!

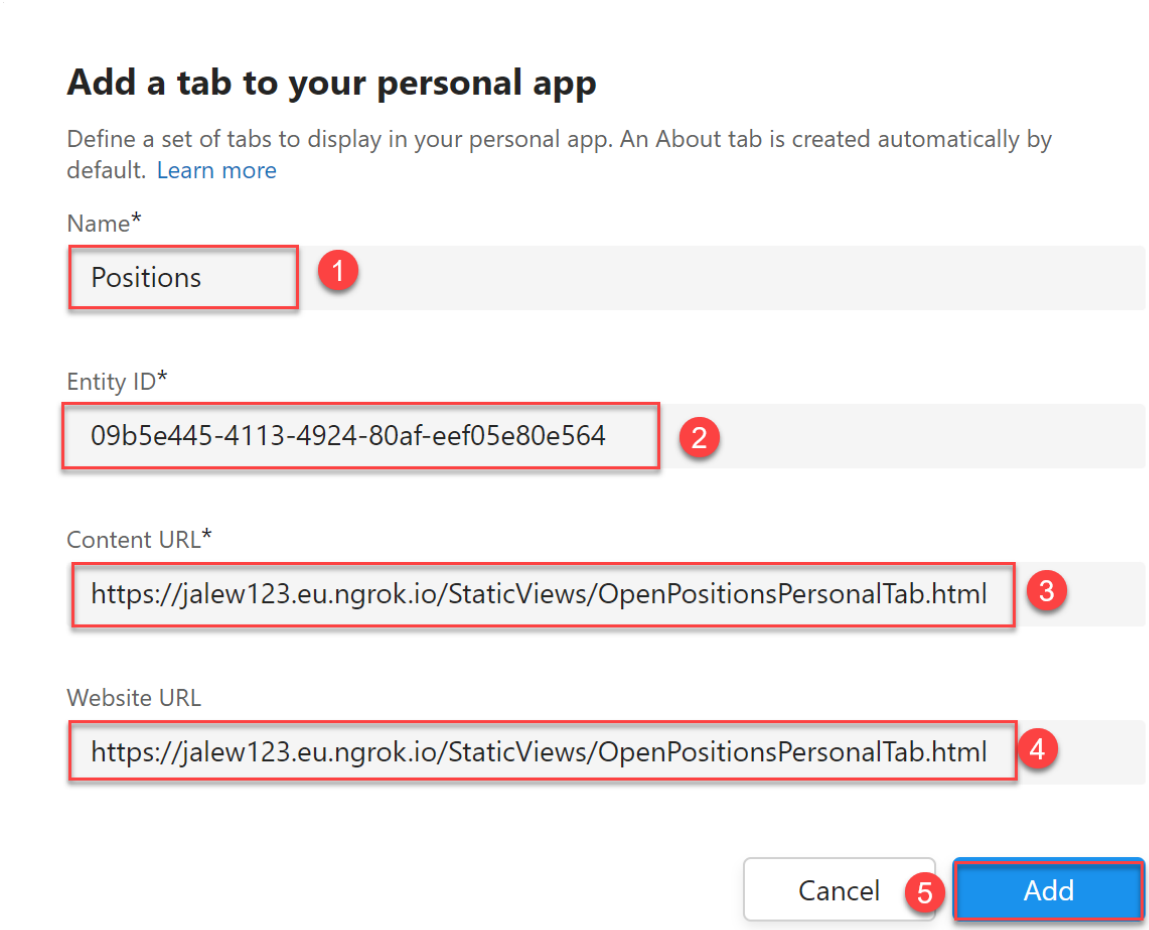
1. Go back to the Teams Developer Portal and open your Manifest. Select **App Features**, and then select **Personal app**.



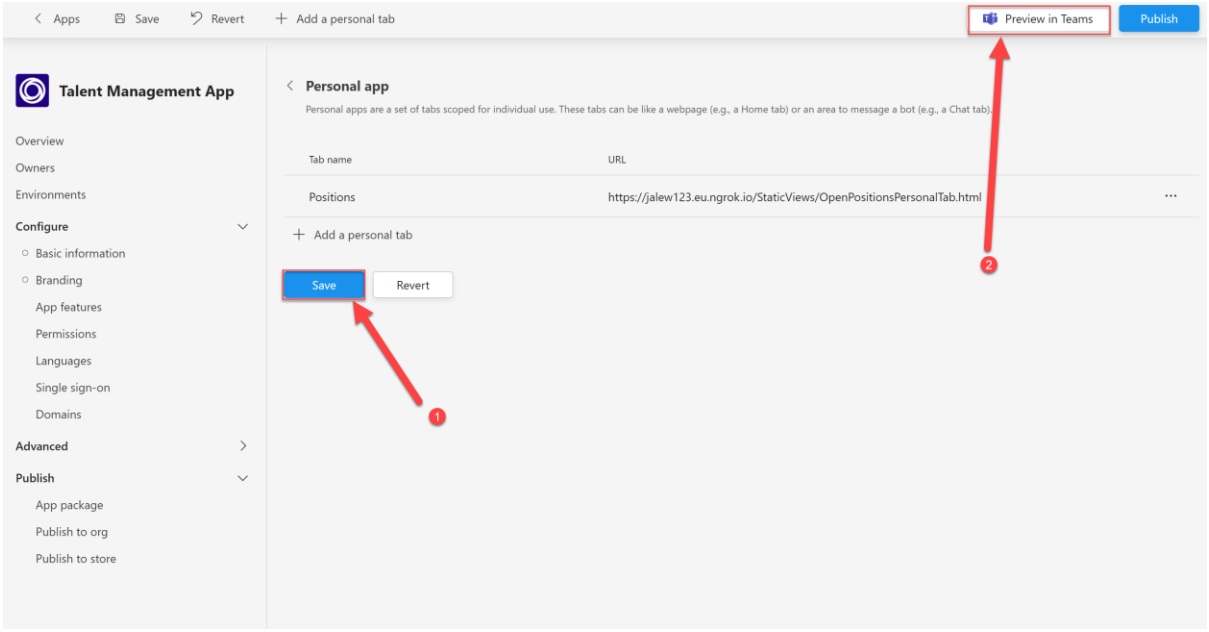
2. Select **Create your first personal app tab.**



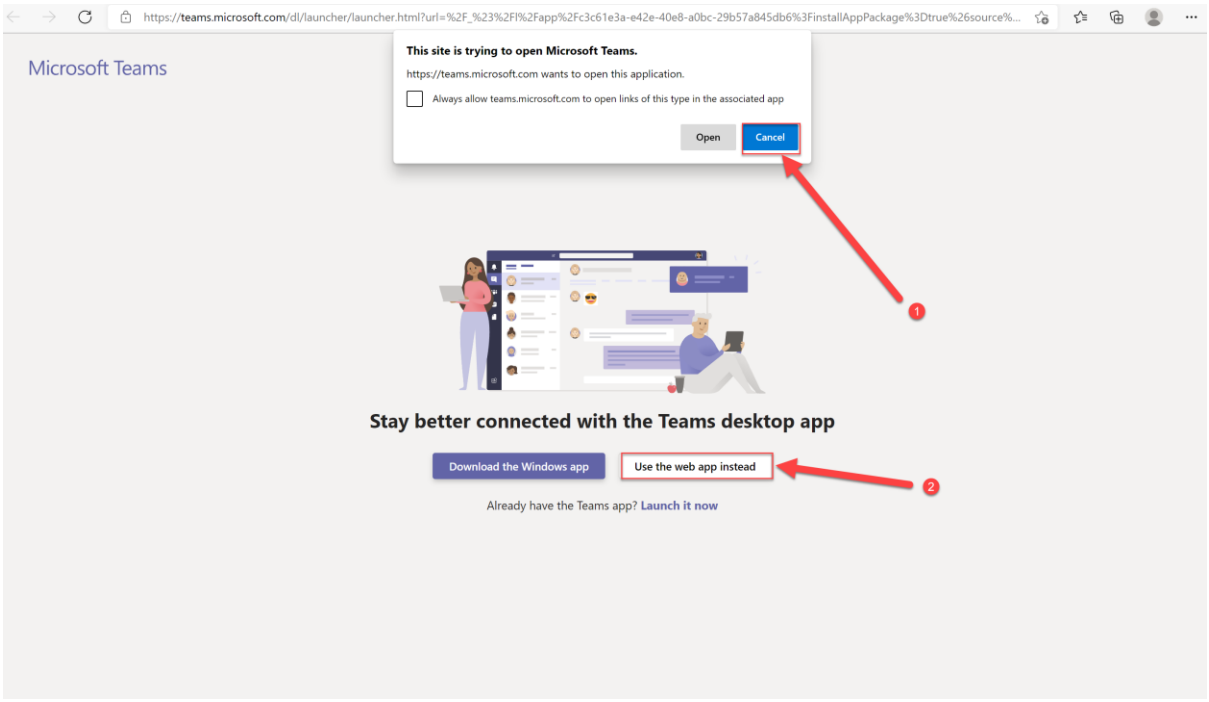
3. Enter the name **Positions**, leave the Entity ID as default, Content URL as **NGROK/StaticViews/OpenPositionsPersonalTab.html** and Website URL **NGROK/StaticViews/OpenPositionsPersonalTab.html** – The Website URL is what will be opened if the user wants to open the website outside of Teams. The Content URL is what will be used when the Tab is loaded inside Teams.



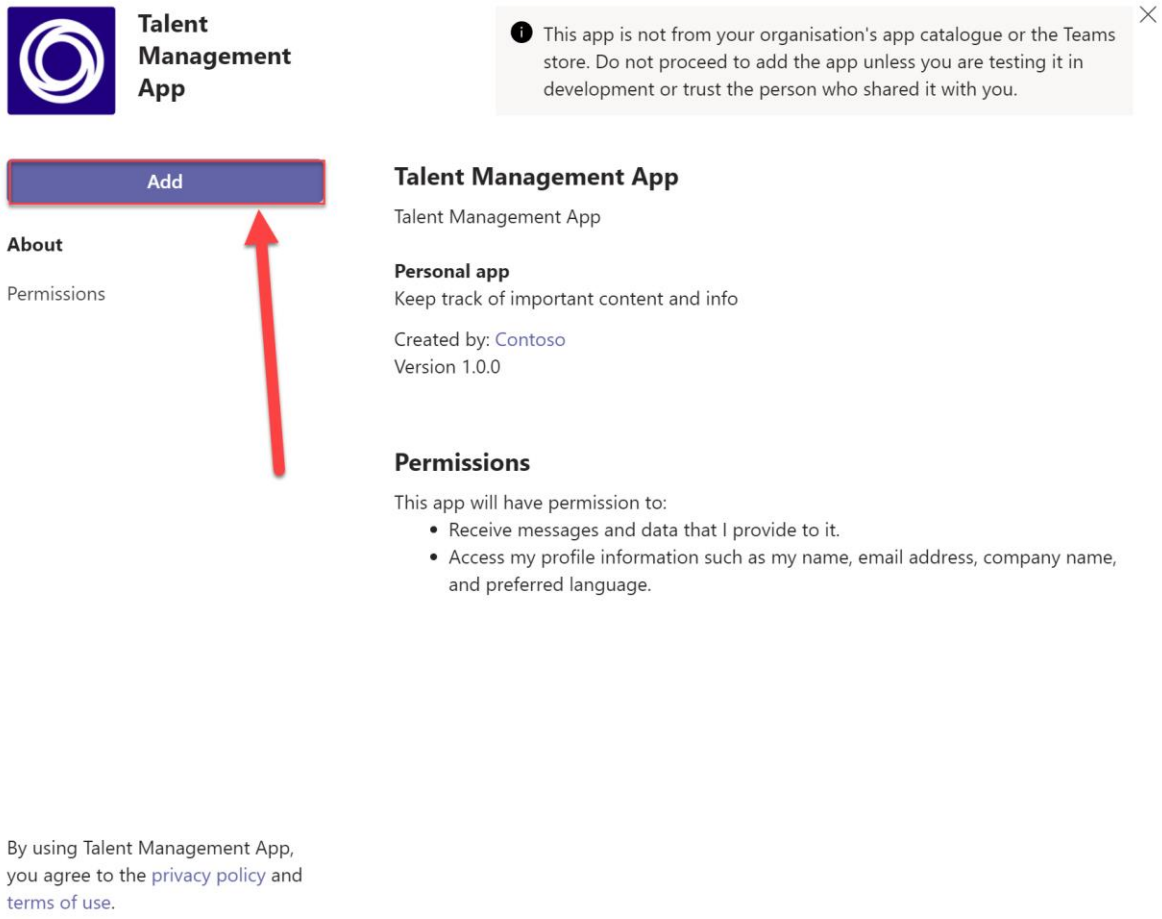
4. We are now in a place where we can deploy this application into Teams to review how the Tabbed interfaces work. There are a couple of options you can use to get the app installed into Teams, so that you can test it, but as we are only testing Tab interfaces in this initial app, we will deploy it as a sideloaded app for an individual user. When we introduce Bots into the app, in section 4, we will install the Teams app as a sideloaded app into the organisational Teams app store, to keep the Teams App IDs consistent across the organisation (more on this later, and why it is important when working with apps that contain bots!). To sideload the app for an individual Teams user click **Save** and select **Preview in Teams**



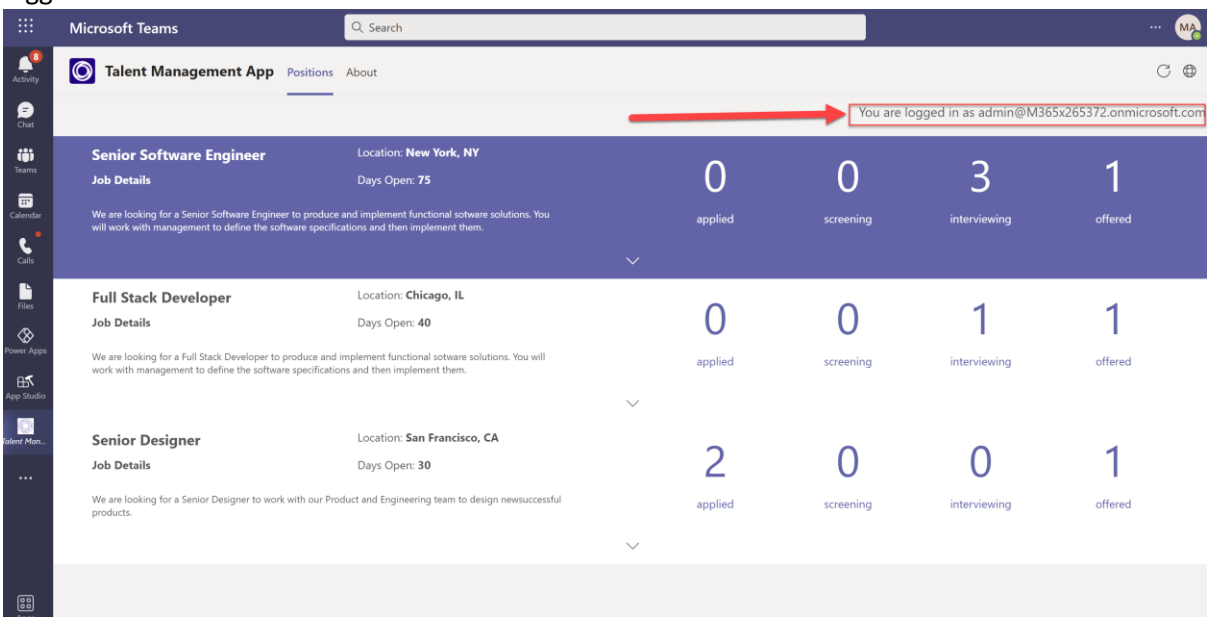
5. Select **Cancel** and click **Use the web app instead.**



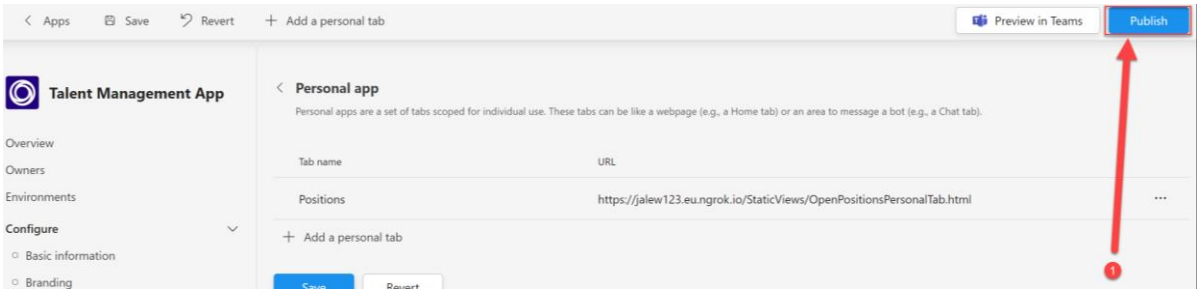
6. After a short wait, your app will appear in Teams. Select **Add.**



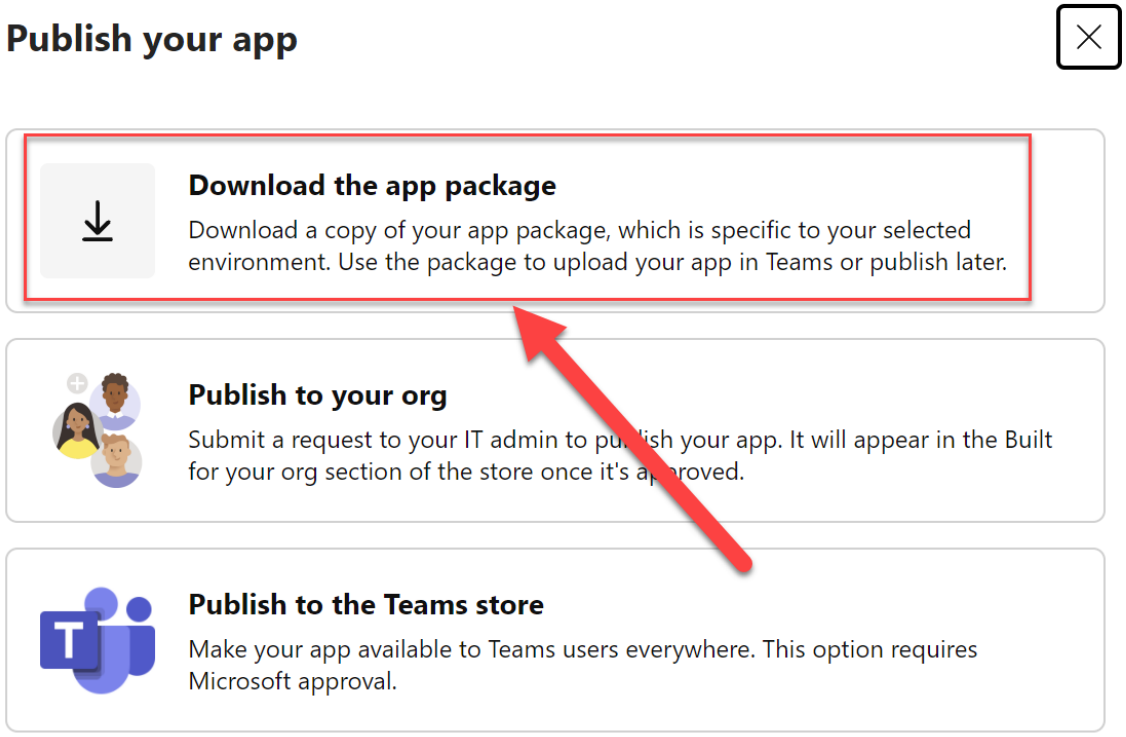
7. Once the Tab has successfully loaded, confirm that you are logged in as the user who is logged into Teams.



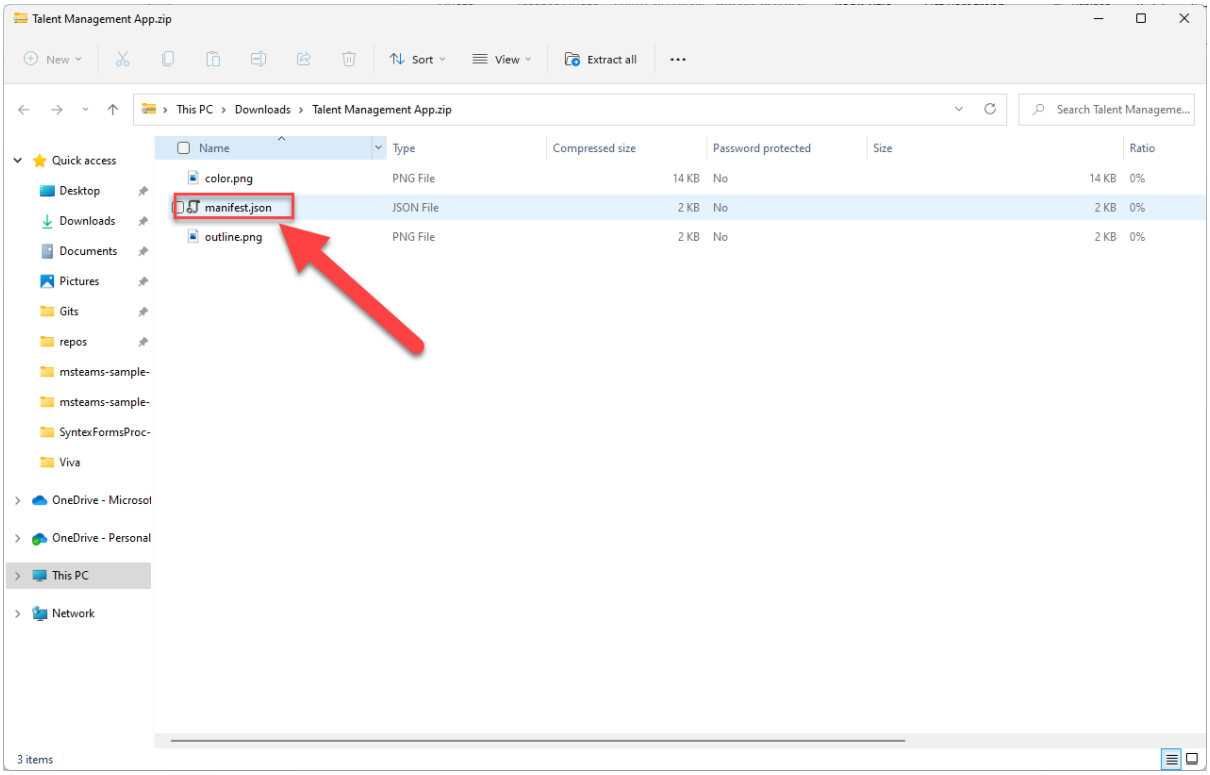
8. We will now review the App Manifest. Go back to the Teams Developer Portal, open your Manifest and select **Publish**.



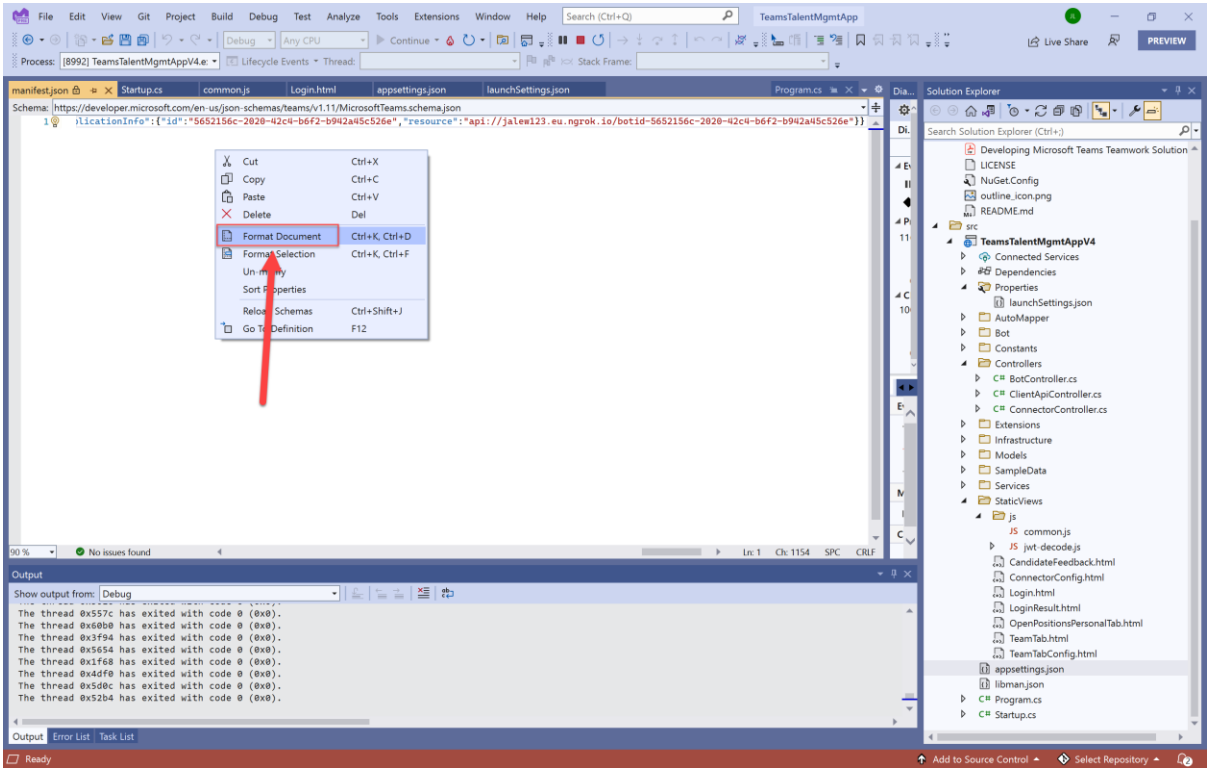
9. Click **Download the app package.**



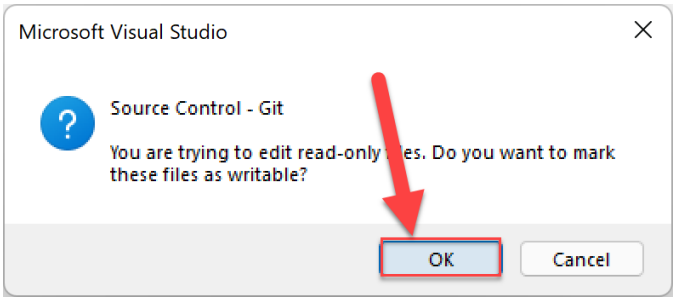
10. Open the ZIP that contains your manifest that has just been downloaded and open the **manifest.json** file in Visual Studio.



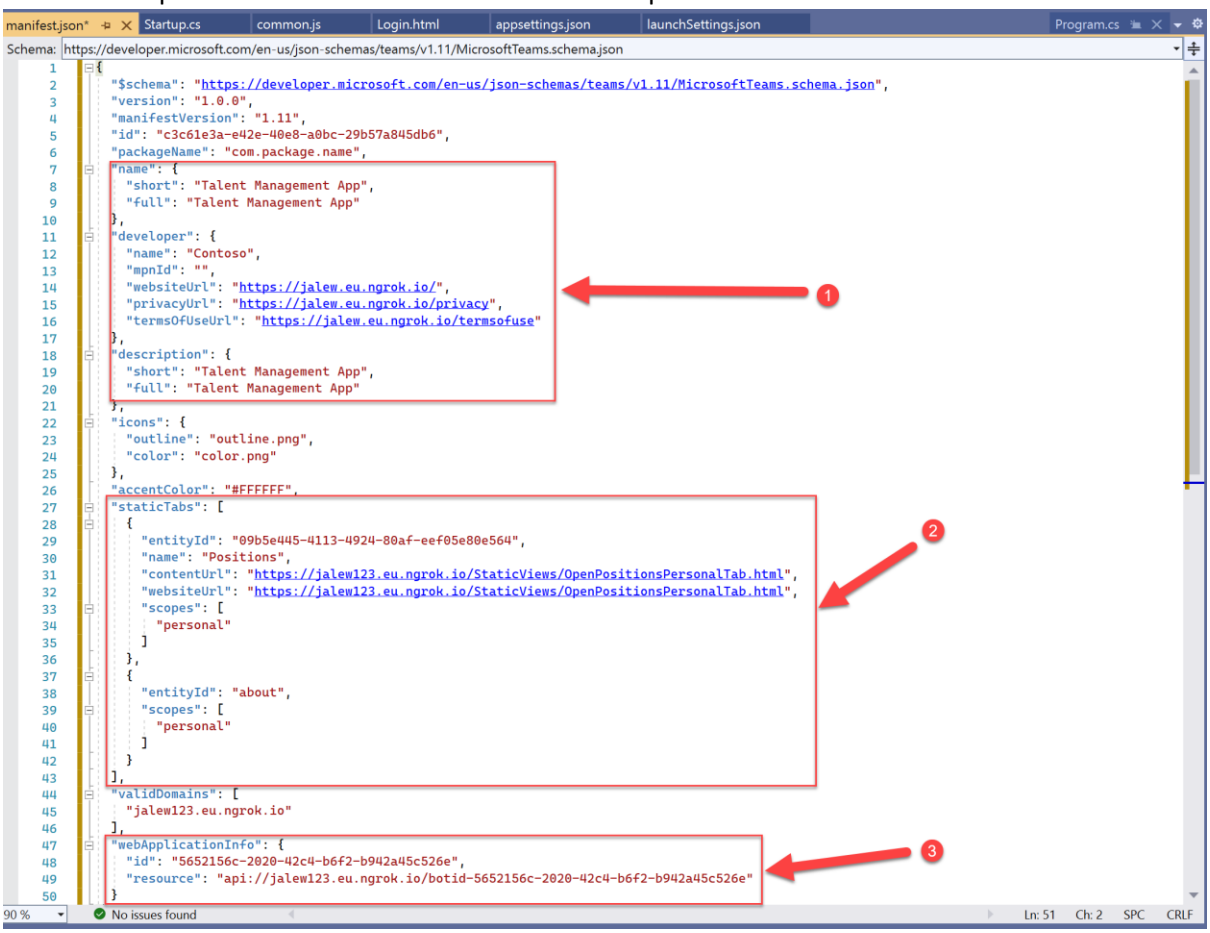
11. To improve readability, **right click the whitespace** in the manifest.json file and select **Format Document.**



12. If prompted, select **Ok.**

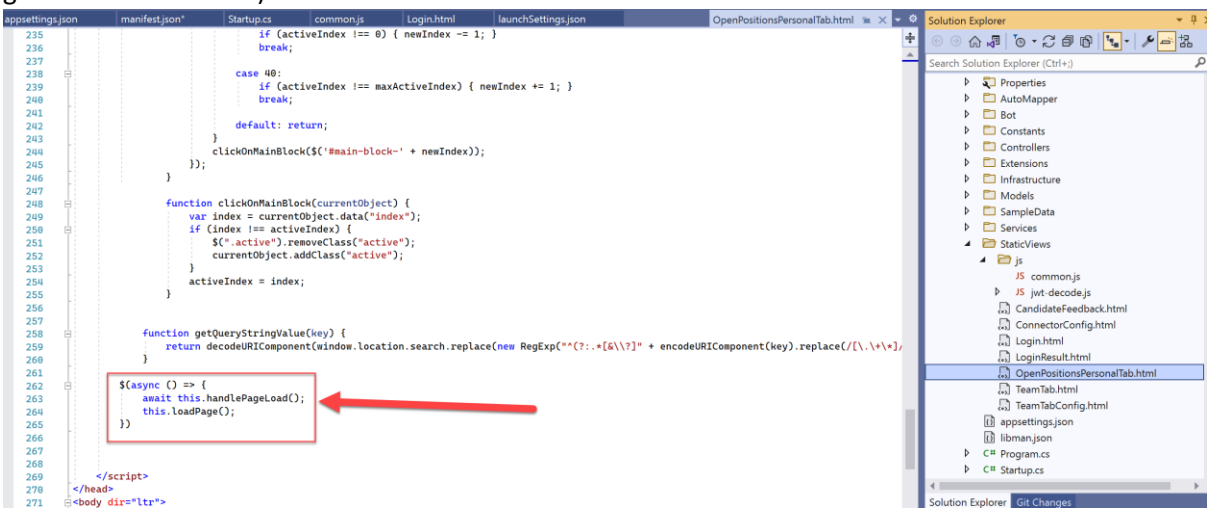


13. Review the settings and values in this manifest – the manifest is your Teams app! Firstly, see the name, developer and description settings and values. They contain the Basic Information you entered on the first page when you created your Teams app in the Teams Developer Portal. Secondly, review the Static Tabs array – this contains the contentUrl and websiteUrl settings and values. Finally, the webApplicationInfo settings and values contain your Azure AD and SSO information, that will be used when the page is loaded to perform Single Sign-On. This is explained in more detail in the next few steps.

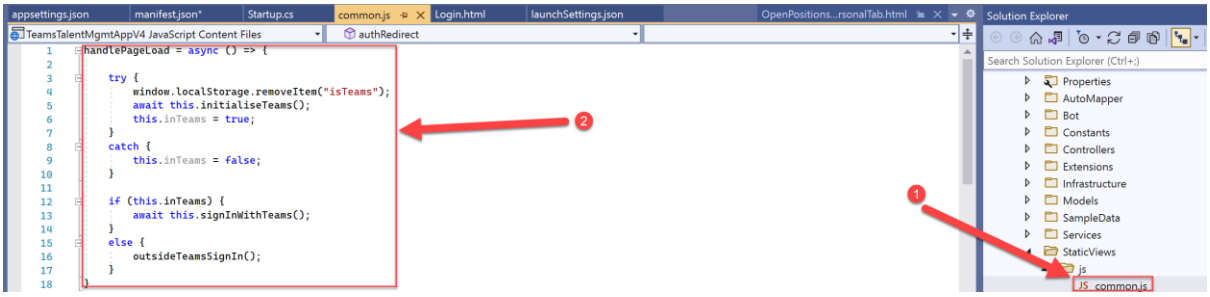


14. Go back into Visual Studio, expand the Static Views folder and select the **OpenPositionsPersonalTab.html** file. Review the scripts that are used when this page is loaded, importantly the Teams SDK (line 7) and the common.js (line 11) are used here. The Teams SDK is used in web applications to enable interoperability between the Teams client and the web-app. It can be used to pass context, perform SSO and many other things. common.js is where all the sign-in and Teams related JavaScript resides. We will review this later.

15. For this specific page, we are using jQuery to control when the JavaScript functions are called. This is detailed on line 262, where we call and await the completion of **handlePageLoad** (which is found in common.js and handles authentication both inside and outside of Teams), before we call **loadPage** (which is what builds the page and calls APIs to get data for Positions).



16. Expand the **js** folder, found inside StaticViews, and open the **common.js** file. Here you can see the **handlePageLoad** function. We are using this to call other functions that then determine if the page is being loaded inside or outside of Teams. If the page is loaded inside Teams, then the function **signInWithTeams** is called, which then attempts to do a Teams SSO. If the page is outside of Teams, the function **outsideTeamsSignIn** is called, which will redirect the user to Azure AD. It's important to note that Azure AD (and many other identity providers) do **not** support iFraming of their authentication pages. As website inside Teams are iFramed, this means that we need a custom way of handling SSO. Hence why the flow of SSO is different when the page is loaded in Teams.

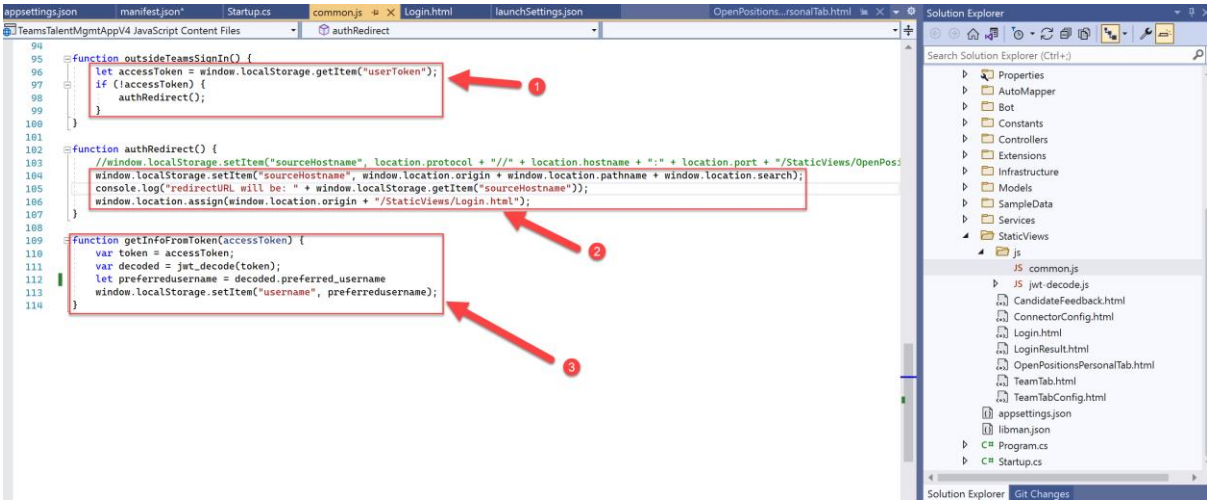


17. Scroll down and review the **getTeamsToken** function. This is the function that attempts to do a Teams SSO, by calling the **microsoftTeams.authentication.getAuthToken** function. This will take the Access Token that was used to sign-in to Teams, by the end user, and submit it

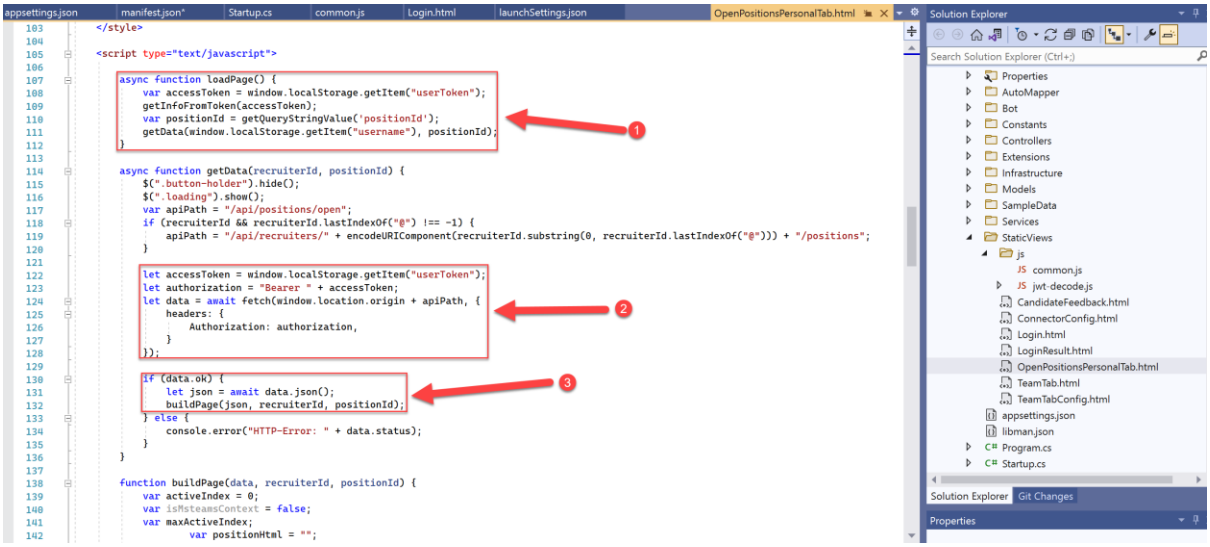
to a back-end service that is operated by Microsoft. The back-end service will then try to swap the Teams Access Token, for an access token that has an Audience of your application. This should work based on the configured that you applied during the setup of the Azure AD App Registration, and will return an access token, with your MicrosoftAppId as the Audience in the claims of the access token, with a scope of access_as_user. **If successful, it will set the access token as an item in Local Storage for the Browser**, this can then be used by the page to call protected APIs to get Position and Candidate data. We will inspect this later on in the lab. The **teamsFallbackAuth** function is called if the SSO function fails, and will allow the user to perform an interactive sign-in to either provide Consent, or get an access token for use within the App.

```
getTeamsToken = () => {
    window.localStorage.setItem("isTeams", "yes");
    return new Promise((resolve, reject) => {
        microsoftTeams.authentication.getAuthToken({
            successCallback: (token) => {
                window.localStorage.setItem("userToken", token);
                resolve(token);
            },
            failureCallback: (reason) => {
                reject(reason);
            }
        })
    });
}

function teamsFallbackAuth() {
    microsoftTeams.authentication.authenticate({
        url: window.location.origin + "/StaticViews/Login.html",
        width: 600,
        height: 535,
        successCallback: function (result) {
        },
        failureCallback: function (reason) {
            handleAuthError(reason);
        }
    });
}
```

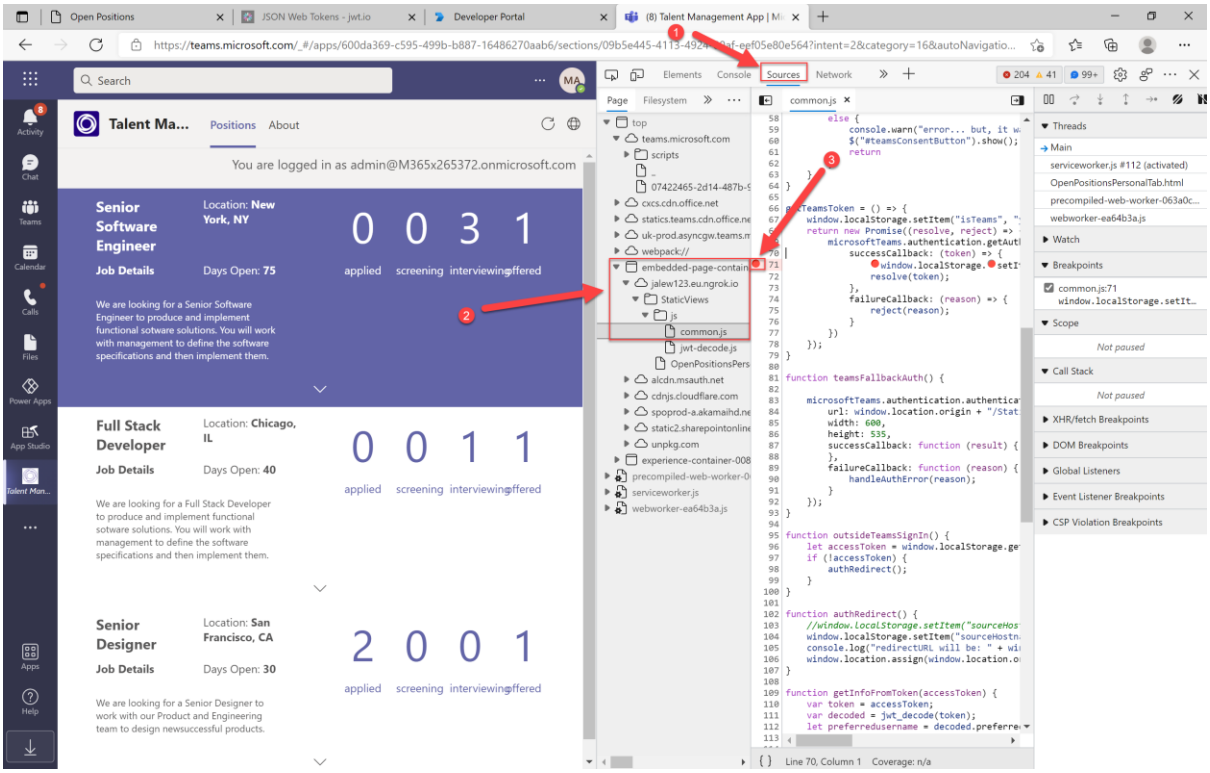
18. Scrolling to the bottom of the common.js file, you will see the **outsideTeamsSignIn** function. This checks for the existence of an access token in Local Storage, and if one doesn't exist, will call the **authRedirect** function. **authRedirect** will set some properties and then redirect the end-user to **Login.html**, which will then redirect the user to Azure AD for sign-in. Eventually, this will provide the user with an access token and they will return to the OpenPositionsPersonalTab.html page, but this time, the accessToken variable will contain an access token and the user will not be redirected the Azure AD. **getInfoFromToken** is called on the OpenPositionsPersonalTab.html page, and is used to pull out the claims (specifically the username attribute) for use when that page is loaded.



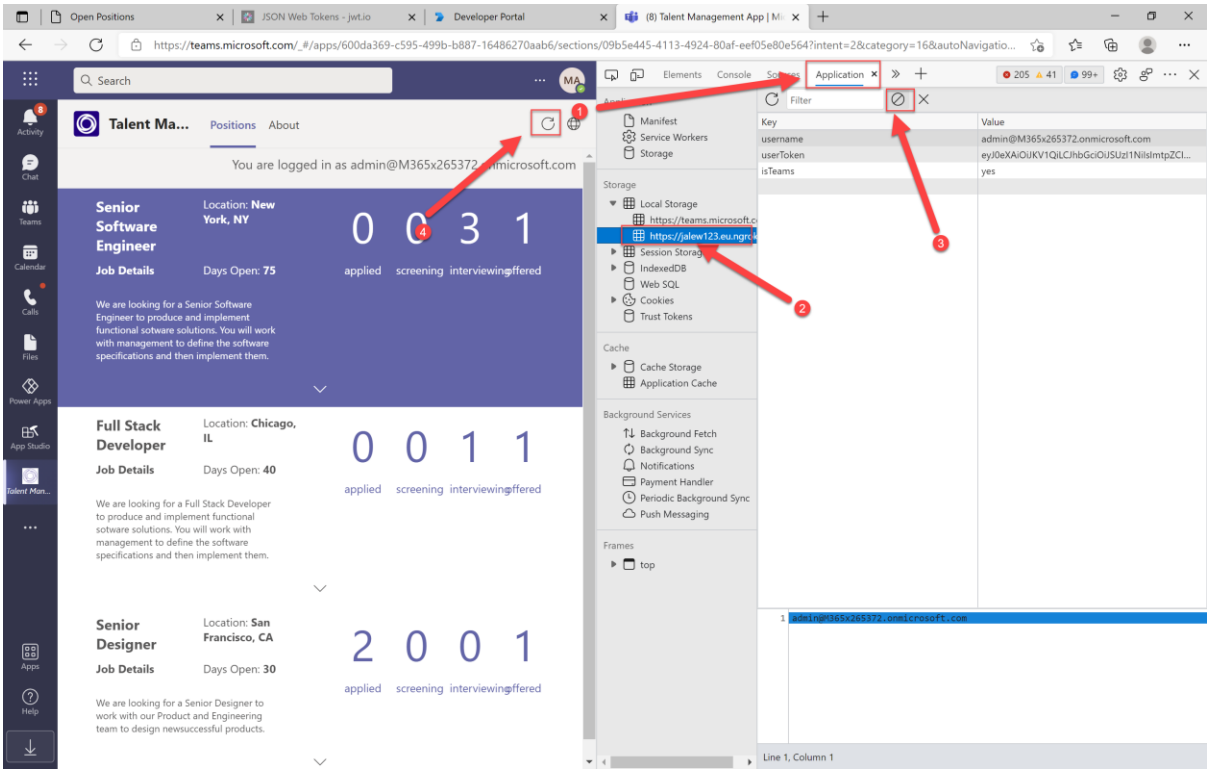
19. Go back to **OpenPositionsPersonalTab.html** and review the **loadPage** function. This will call the **getData** function. **getData** uses fetch within JavaScript to make a HTTP request, with the access token obtained from Azure AD in the header, to the apps Client APIs, this will return information about the available positions stored within the app. Once we have this information, and a 200 OK response is returned form the Client APIs, **buildPage** will be called, and the position data (stored in the **json** variable), recruiterId (which is your username) and positionId (which is obtained from the URL querystrings) is passed into this function – buildPage will then render the page as you see it!



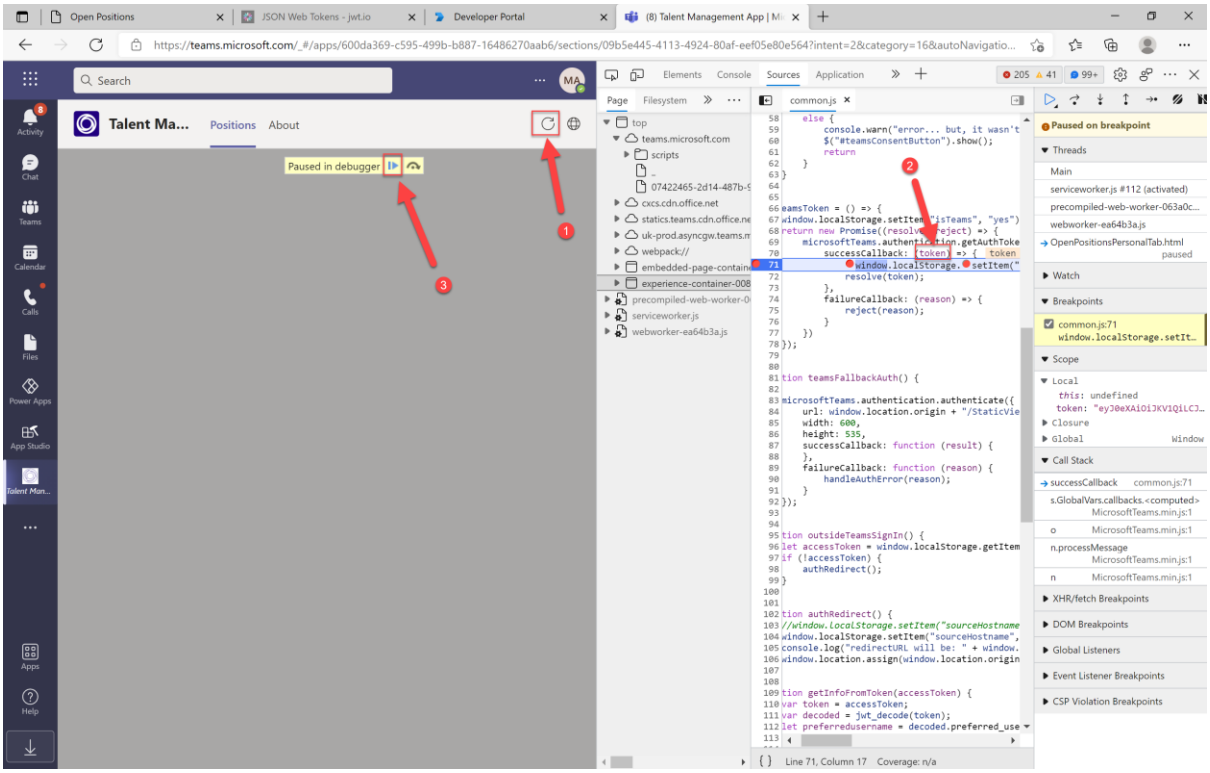
20. Let's see this in action! Go back into Teams and open the Talent Management App. Hit **F12** on your keyboard to open your browser's development tools. Select **Sources**, expand the folder hierarchy, until you find **common.js** (which is inside StaticViews/js folder, found within embedded-page/NGROK). Scroll down and click **to the left of line 71 to add a Breakpoint** (this will add a red dot to the left of the number).



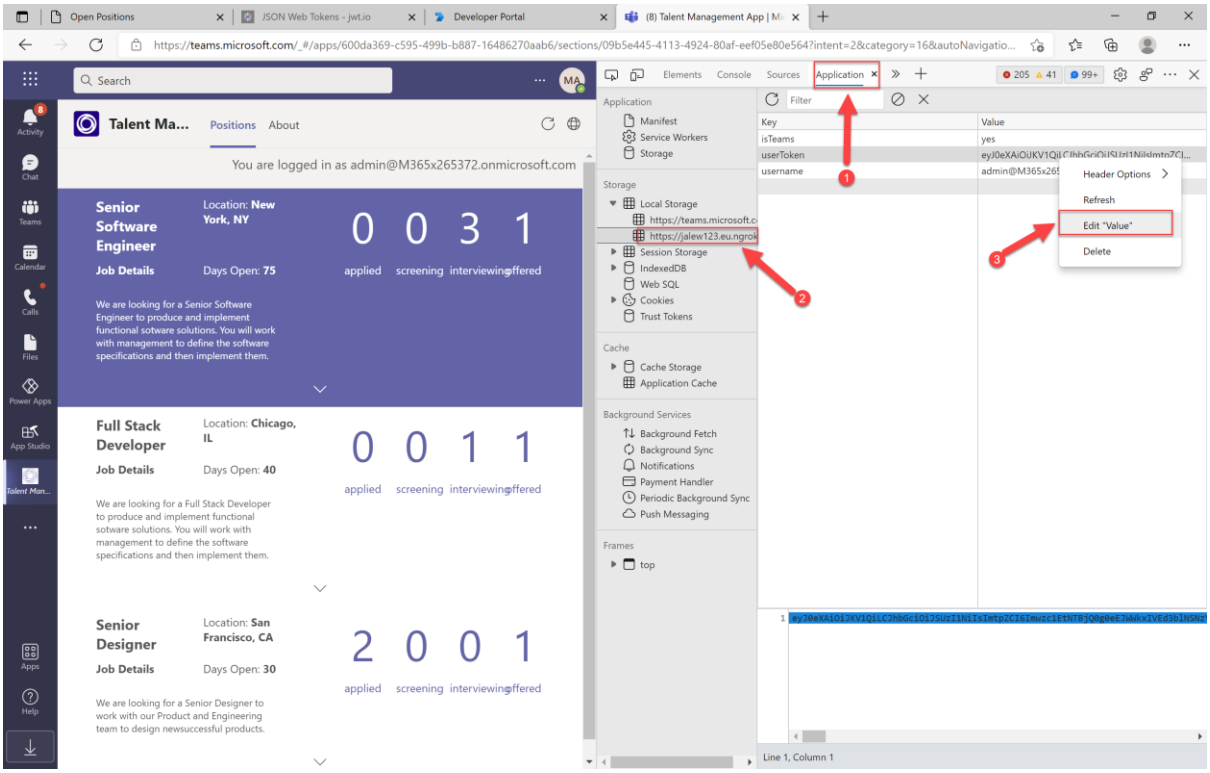
21. Before we re-load the page, it's a good idea to remove the items in Local Storage for this app. Click **Application**, expand **Local Storage** and select your **NGROK app**. Click the **clear button.**



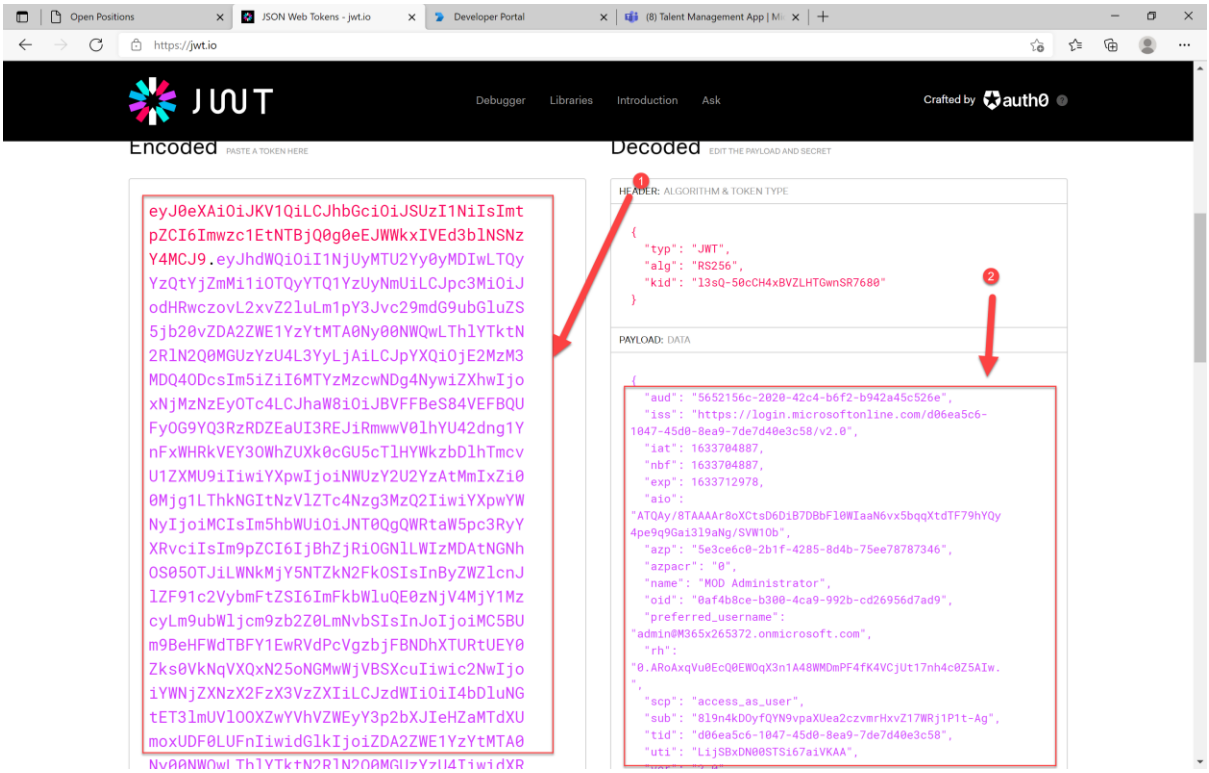
22. Hit the **Reload page** button. This will trigger the breakpoint. Mouse over the **Token** variable on line 70. This should then present you with an Access Token that has been retrieved via the Teams SSO method. Hit the **Continue** button to allow the page to complete loading.



23. Go back to the **Application** tab and select your **app** under **Local Storage.** Right click the **userToken** item, click **Edit** and **copy the value to your clipboard**.



24. Open a browser tab and go to https://jwt.io , **paste your access token** into the left input box and **review the claims within the token**. Importantly, you will see that it was Issued by Azure AD, the Audience is your MicrosoftAppId, the scope is access_as_user & the user is the current logged in Teams user.

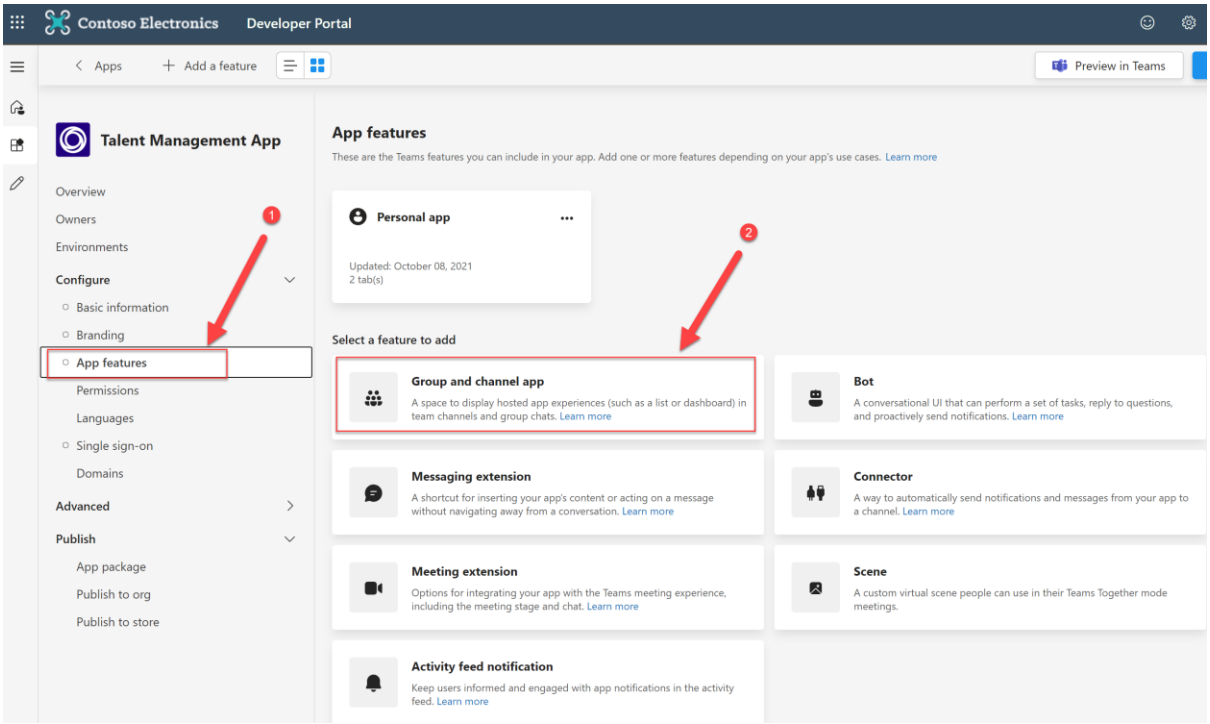


Great work! Now that the Personal Tab is working, and we have seen Teams SSO in action, let's move onto a Tab in a Team! These work slightly differently as they use a configuration page.

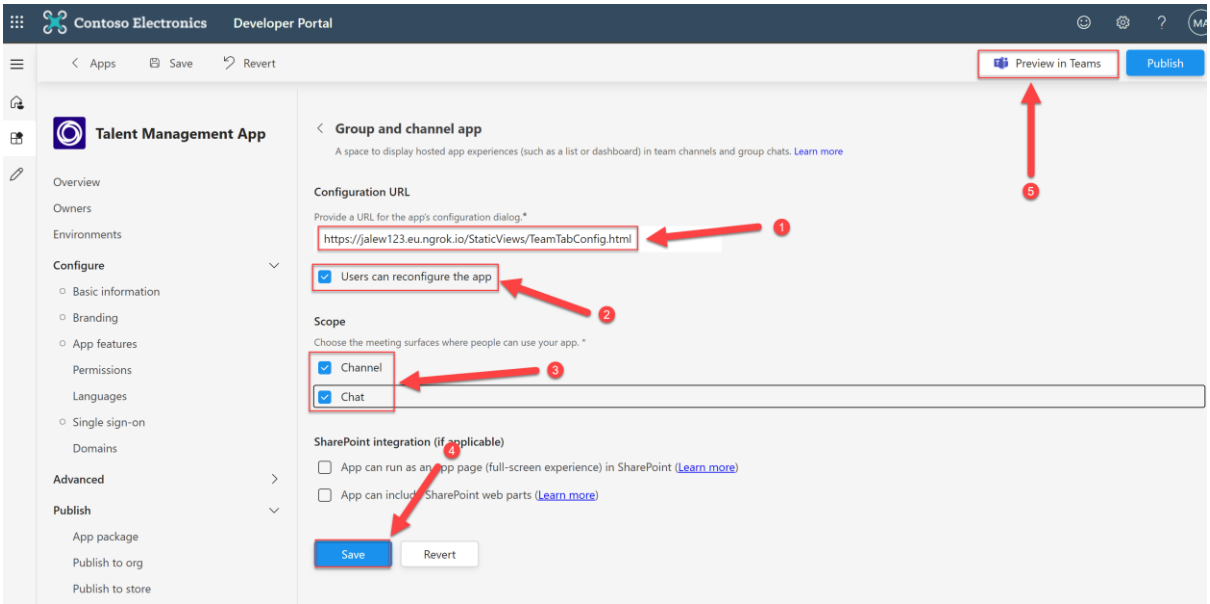
3) Implement a Tab inside a Team Channel

In this step we will implement a Team Channel Tab, which uses a special Configuration page, that we will inspect during this section of the lab. Channel Tabs are great for pinning specific information about specific objects that exist within your application and enable collaboration around that specific object. In this example, we'll be pinning a tab that contains details about an available position and the associated candidates that have applied for that position.

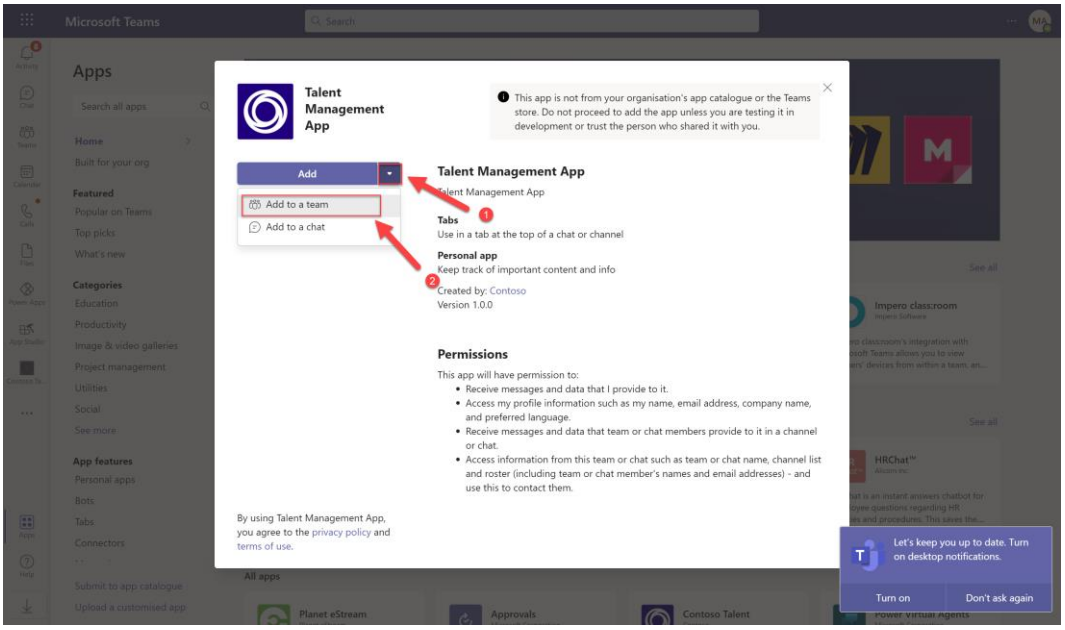
1. Go back to the Teams Developer Portal and open the Teams Talent Management App. Select **App Features** and click **Group and channel app**.



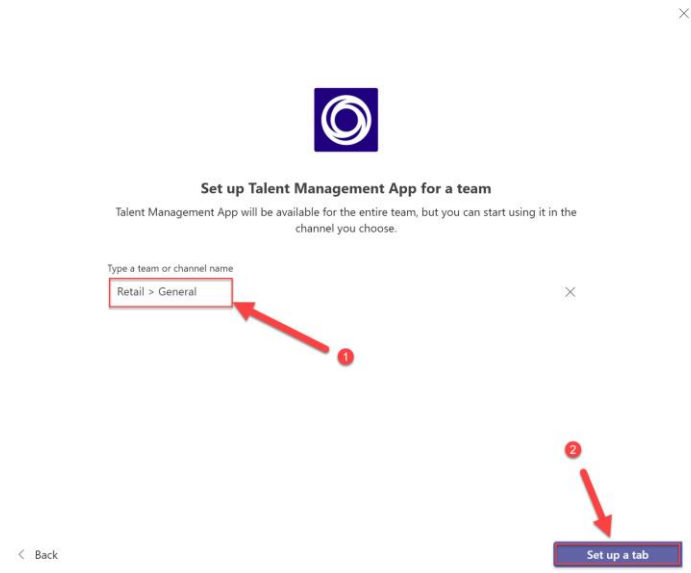
2. Enter the following into the Configuration URL **NGROK/StaticViews/TeamTabConfig.html** – select **Users can reconfigure the app** and select both **Channel and Chat** in the scopes. Click **Save.** For context, the Configuration URL is the URL that will open the Tab Configuration page, this is a special page that allows users to decide which Position you want to include in the Team Tab, we will inspect the JavaScript later in this lab that enables this. Click **Preview in Teams.**



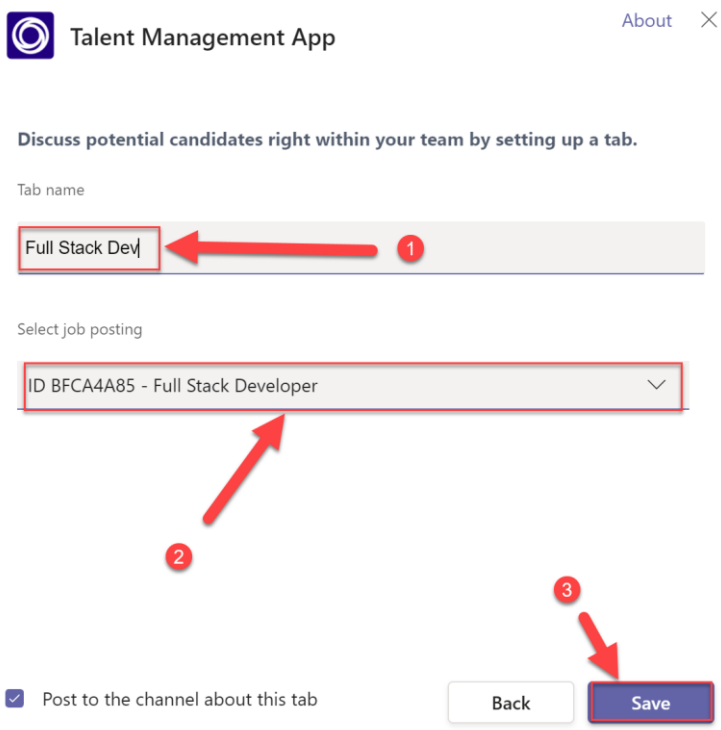
3. Select the **drop down button** and select **Add to a Team**.



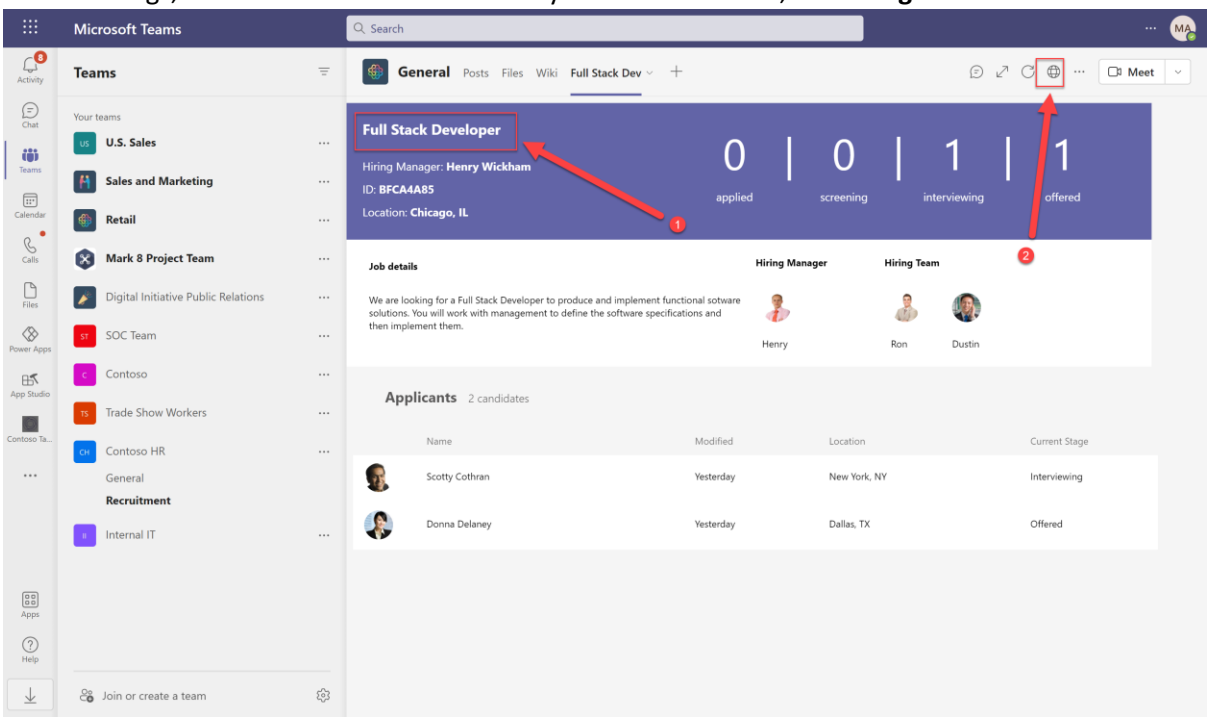
4. Select a Team and Channel, in this demo I selected **Retail -> General**, and click **Set up a tab**.



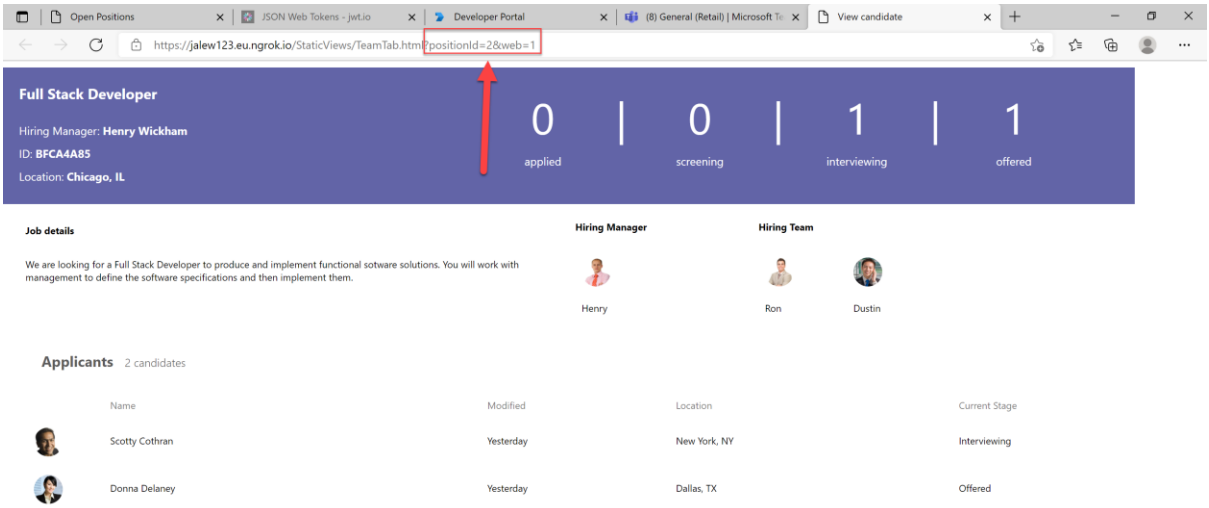
5. This will open the Tab Configuration page defined in the App Manifest. On this page you can enter a name for the Tab and select a job Position. Enter the tab name of **Full Stack Dev** and select **Full Stack Developer** in the dropdown. Click **Save.**



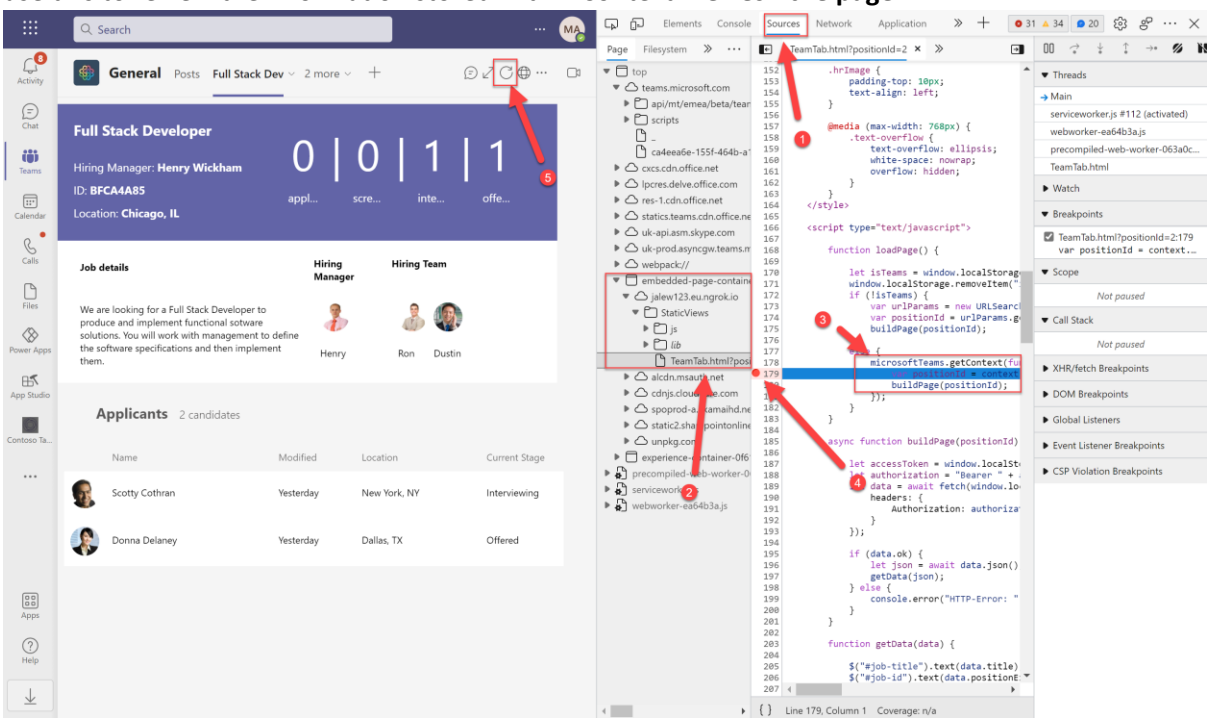
6. This should load the TeamTab.html page inside the Team Channel Tab, that contains information about the Full Stack Developer position. This is achieved by passing the Teams Tab EntityID to our web-application, via the microsoftTeams.getContext function in the Teams SDK. This page has also been configured to load outside of Teams, by using the URL Search Strings, instead of the Teams Tab EntityID. To confirm this, click the **globe** icon.



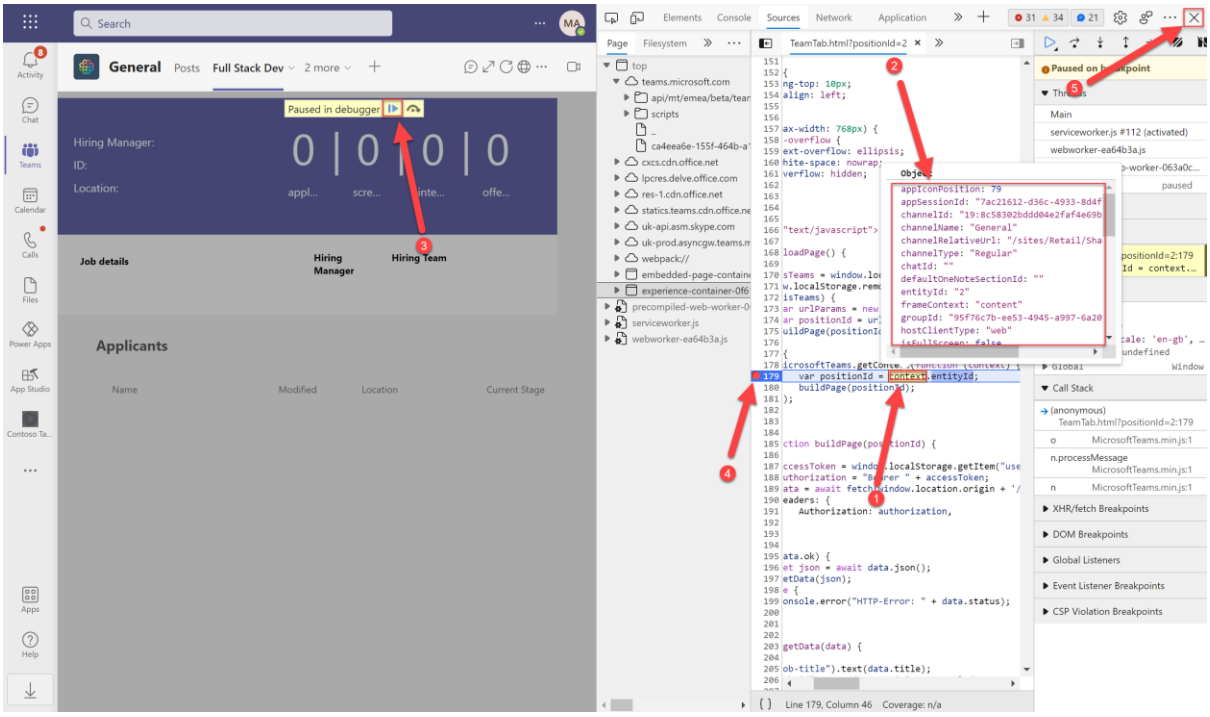
7. You will then see the page load successfully, and if you review the URL Search Strings, you will see **positionID & web** contain values.



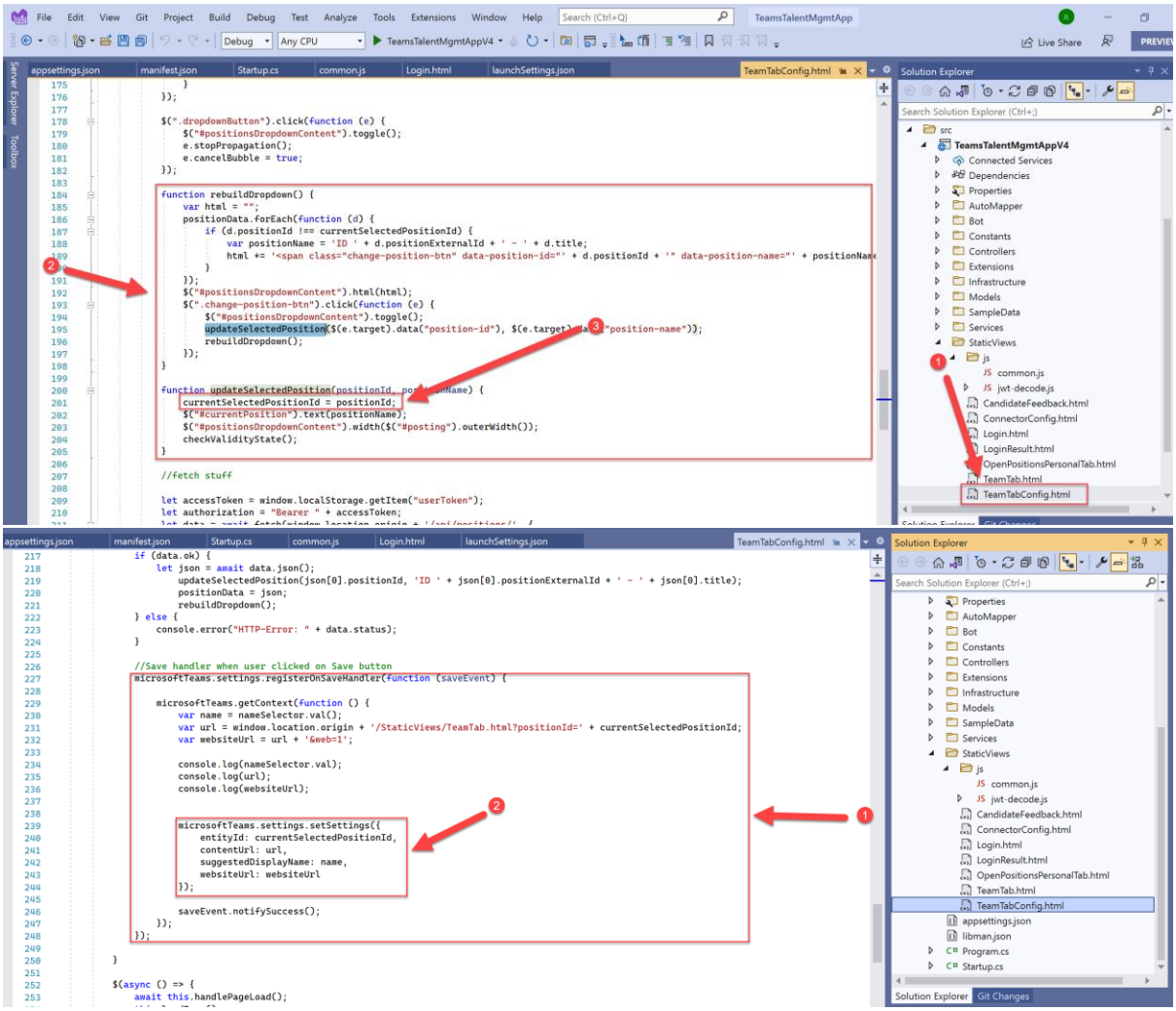
8. Let's review the code that enables this functionality. Go back into Teams, within the Channel Tab. Press **F12** to open the development tools. Select **Sources**, and expand **embedded-page, NGROK, StaticViews and select TeamTab.html**. Scroll down to line 177 in the code and review the **microsoftTeams.getContext** function, this is the capability within the Teams SDK that will pass the EntityID, and a lot of other context information through to the Talent Management Web App. We use the EntityID, stored within the Context variable, to build the page by calling **buildPage(positionID)** on line 180. **Add a breakpoint to line 179**, as we will use this to review the information stored within Context. **Refresh the page**.



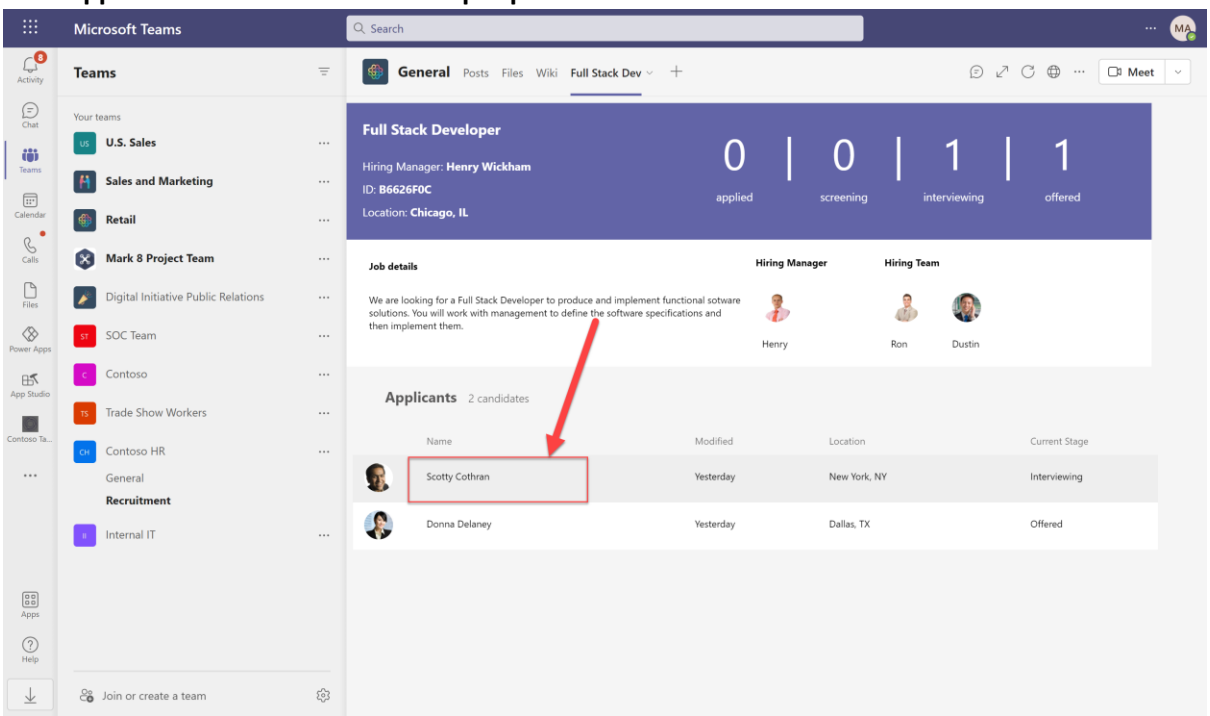
9. The breakpoint will be triggered. **Mouse over the word context** on line 179, this will display the values stored within the context variable, which are set when **microsoftTeams.getContext** is called. You can see here that alot of useful information is available to you, when you are building you web-app. In our case, we simply use **entityID** to decide which position we should show on the page, but this context can be used to discover channelName, themeSettings, and a load of other settings! Review this and consider what you could use in your application to provide a brilliant experience when your app is delivered inside Teams. Click the **continue** button to resume the debugger, remove the **breakpoint** by clicking red circle icon to the left of line 179, and **close the developer tools**. Let's now go back into visual studio and review the steps that are taken to pass the entityID from the TeamTabConfig.html page, into the TeamTab.html page.



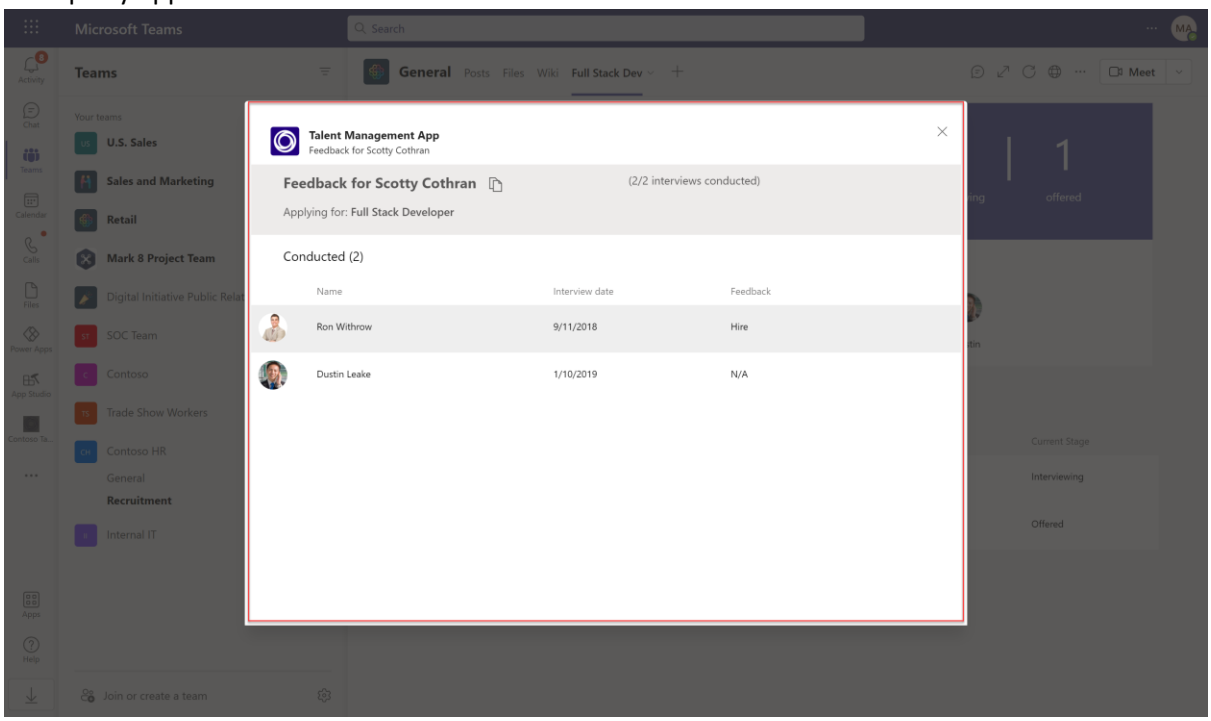
10. Open Visual Studio, and open the **TeamTabConfig.html** page, scroll down to line 184 and review the code contained in the **rebuildDropdown and updateSelectedPosition** functions. These work together to build the dropdown with available Positions, and when a position is selected, they set the **currentSelectedPositionId** variable with the ID of the selected position. Scrolling further down to line 227, review the **microsoftTeams.settings.registerOnSaveHandler** function – this uses the **currentSelectedPositionId** variable, to set the **entityId** that will be passed into the TeamTab.html page when it is loaded inside Teams.



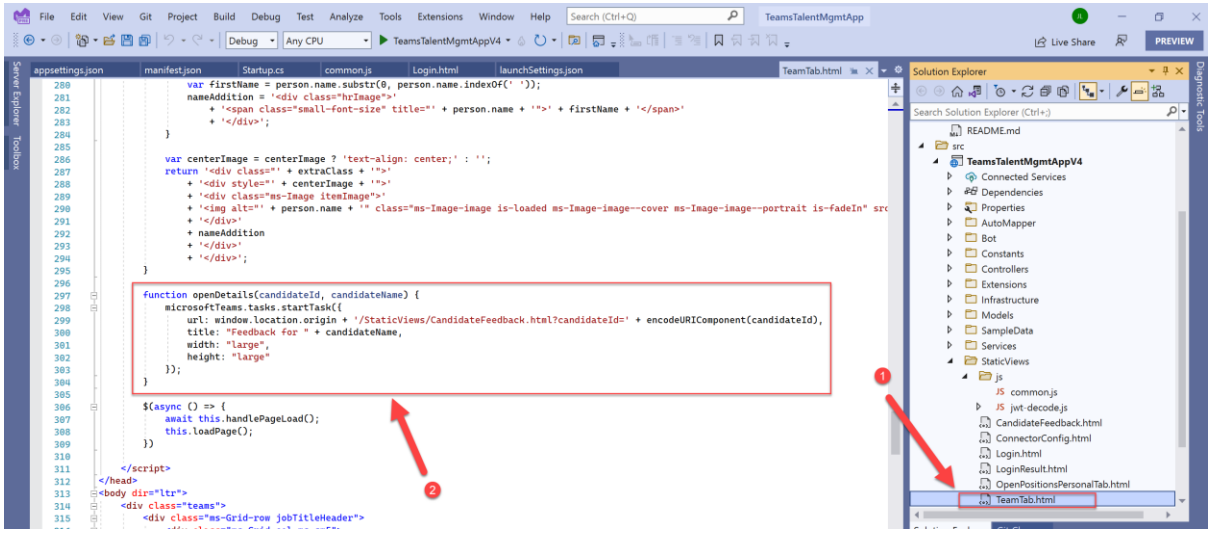
11. Go back into Teams and open the Team Channel Tab and select one of the **candidates that have applied for the Full Stack Developer position**.



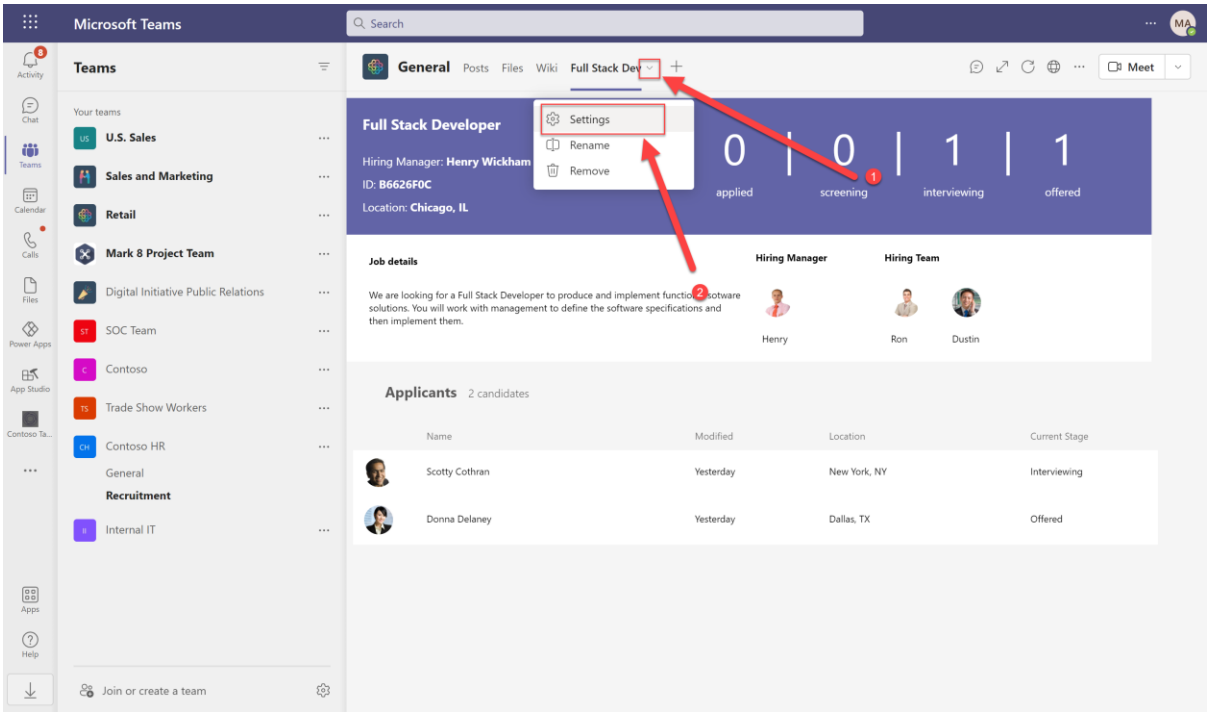
12. This will open a **Task Module**, which is a pop-up within Teams that iFrames a web-application (similar to how Tabs work!). These are great, as they allow you to stay in the context of what you are doing within Teams, while reviewing information that exists within a third party application. Let's review the code that initiated the task module…



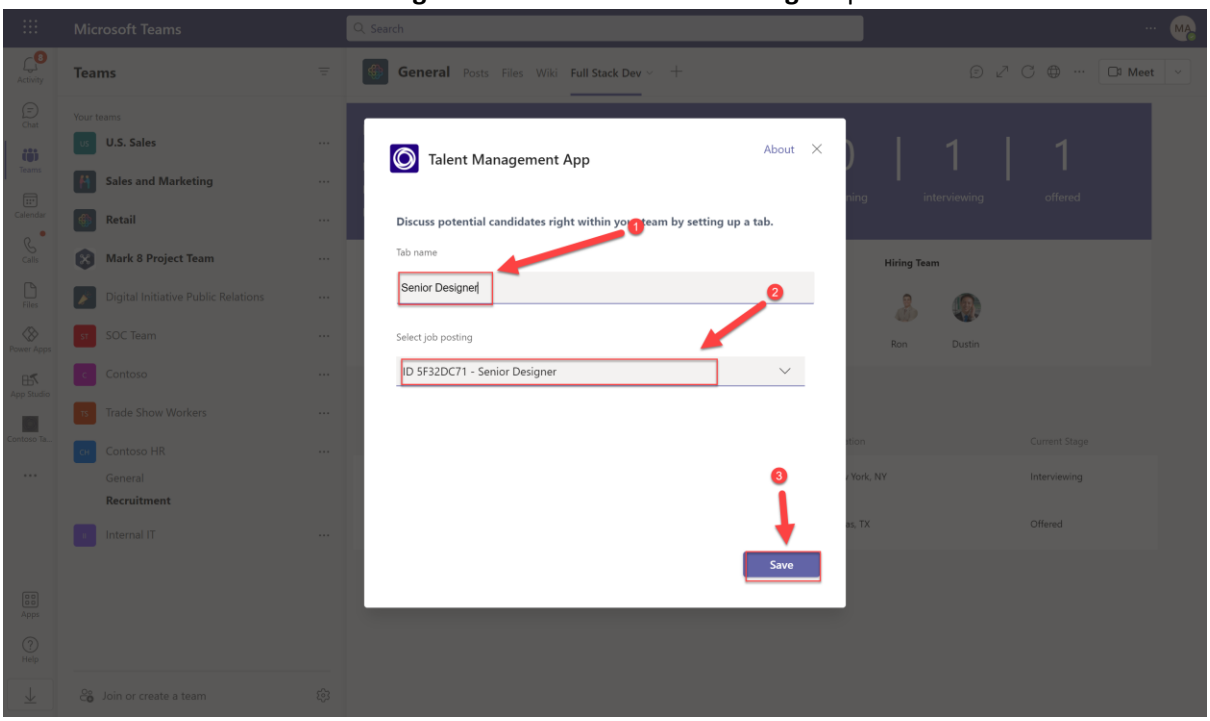
13. Head back into Visual Studio and open the **TeamTab.html** file. Scroll down to line 297 and review the **openDetails** function. This uses the **microsoftTeams.tasks.startTask** function, passing in the url, title, width and height of the task module.



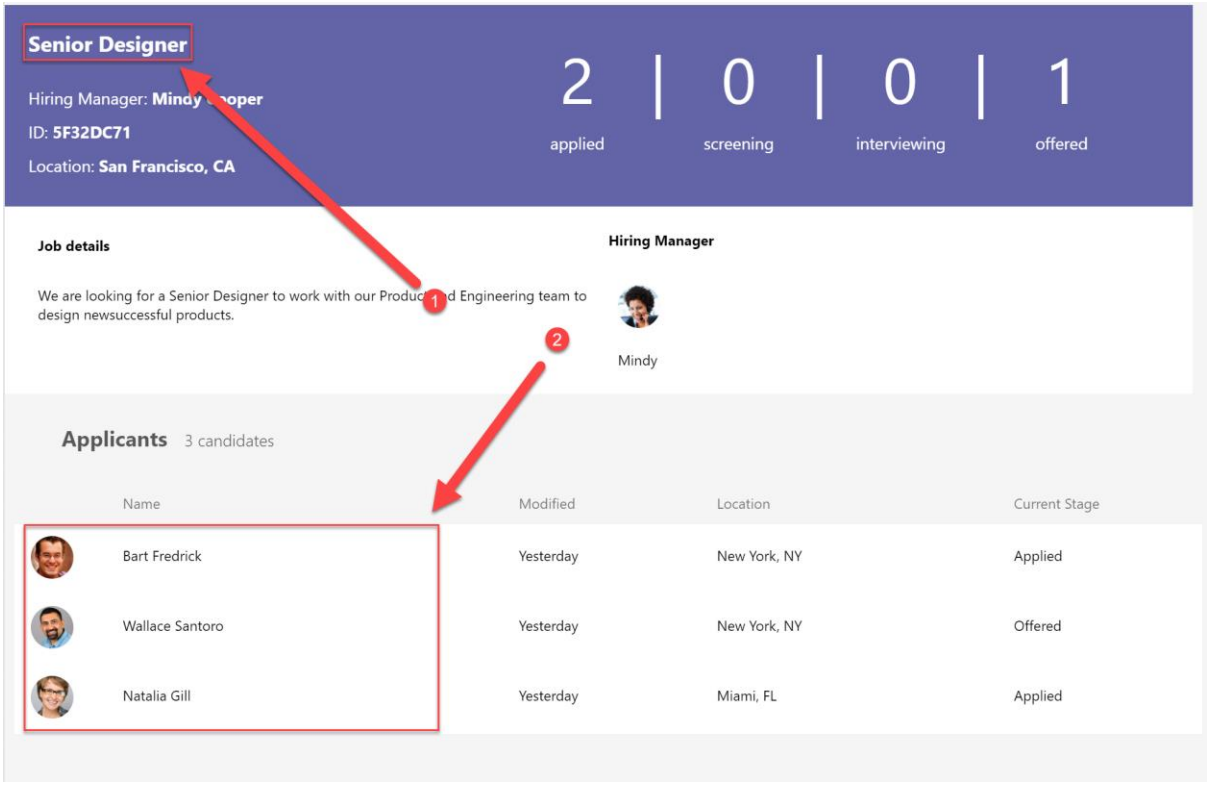
14. Head back into Teams and let's review how you can edit Team Channel Tabs, which will re-invoke the TeamTabConfig.html page. Select the **drop-down button** and click **Settings**.



15. Enter the Tab name of **Senior Designer** and select the **Senior Designer** position. Click **Save**



16. You will see now that the TeamTabConfig.html page has changed the entityId to match the positionId of the Senior Designer position. If you want to; review the context in the developer tools to confirm this.

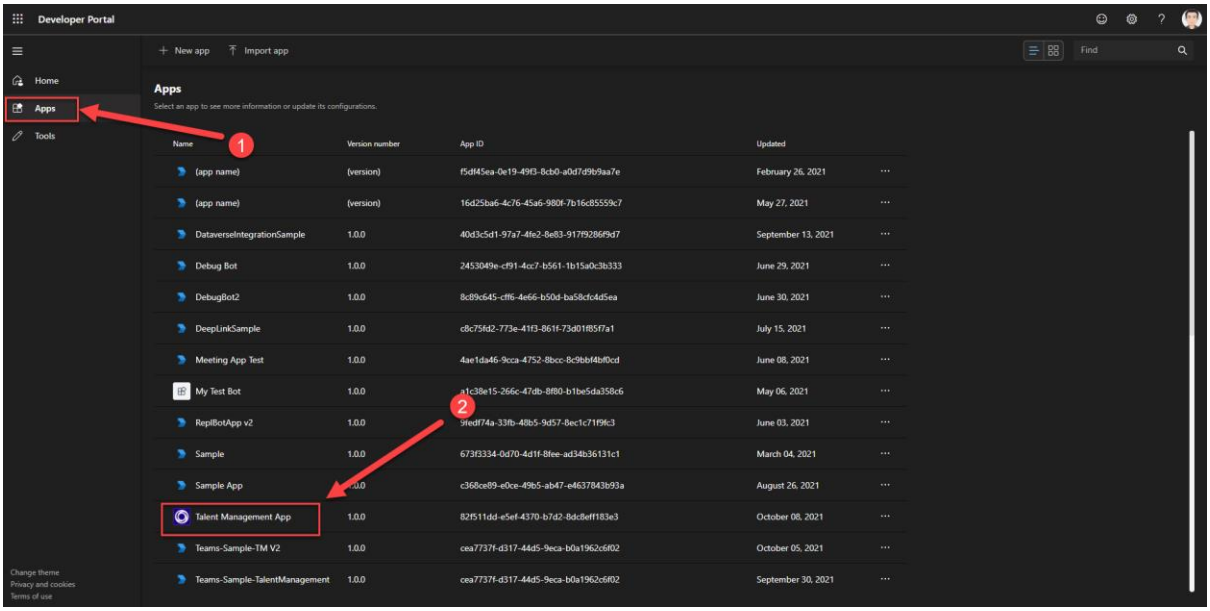


Now that the Channel Tab is working we will now move onto configuring the ChatBot. As we are now moving from HTML and JavaScript to .Net/C#, you will see the following screenshots in dark-mode - a favourite amongst back-end developers…

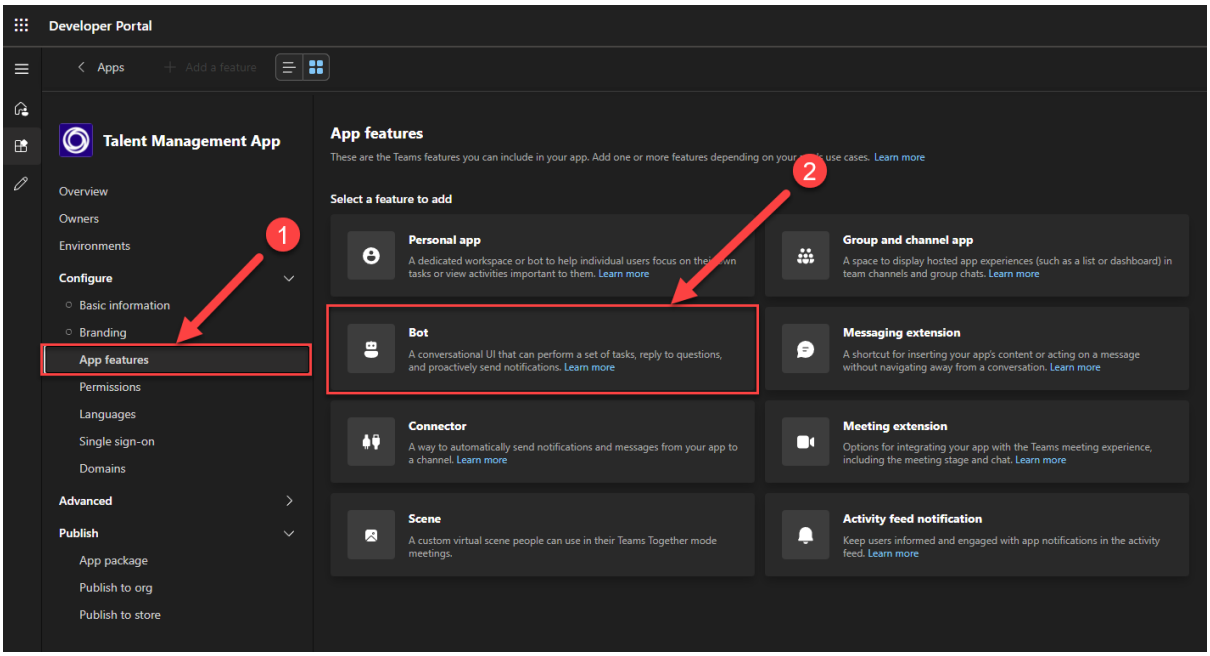
4) Implement a ChatBot that supports Personal, Group and Channel scopes

In this step we will modify the application manifest to add our bot to the app. Importantly, we need to the sideload the app, that contains the bot, into the Organisational app store, or the Teams App ID is randomised, which will then cause issues when we want to initiate a Task Module from an adaptive card (the SSO won't work properly if the Teams app ID is randomised).

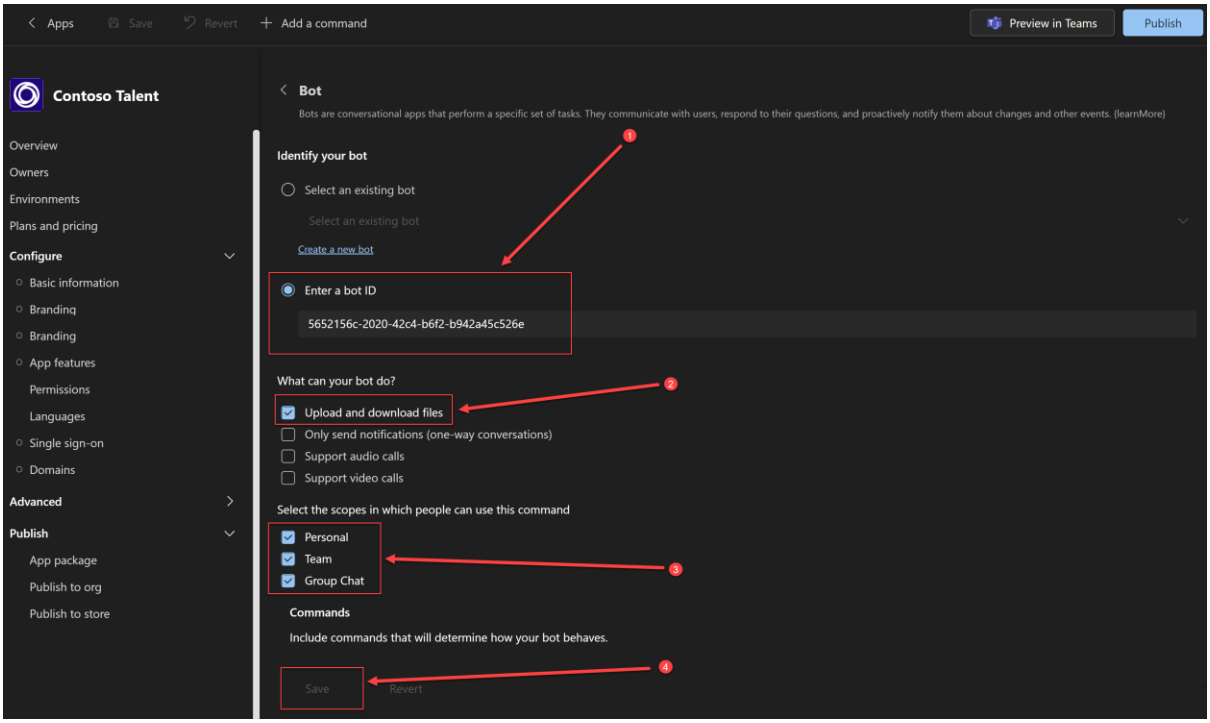
1. Open the Developer Portal by navigating to *https://dev.teams.microsoft.com*. Once there we will select **Apps** from the navigation menu and find the application we had previously created.



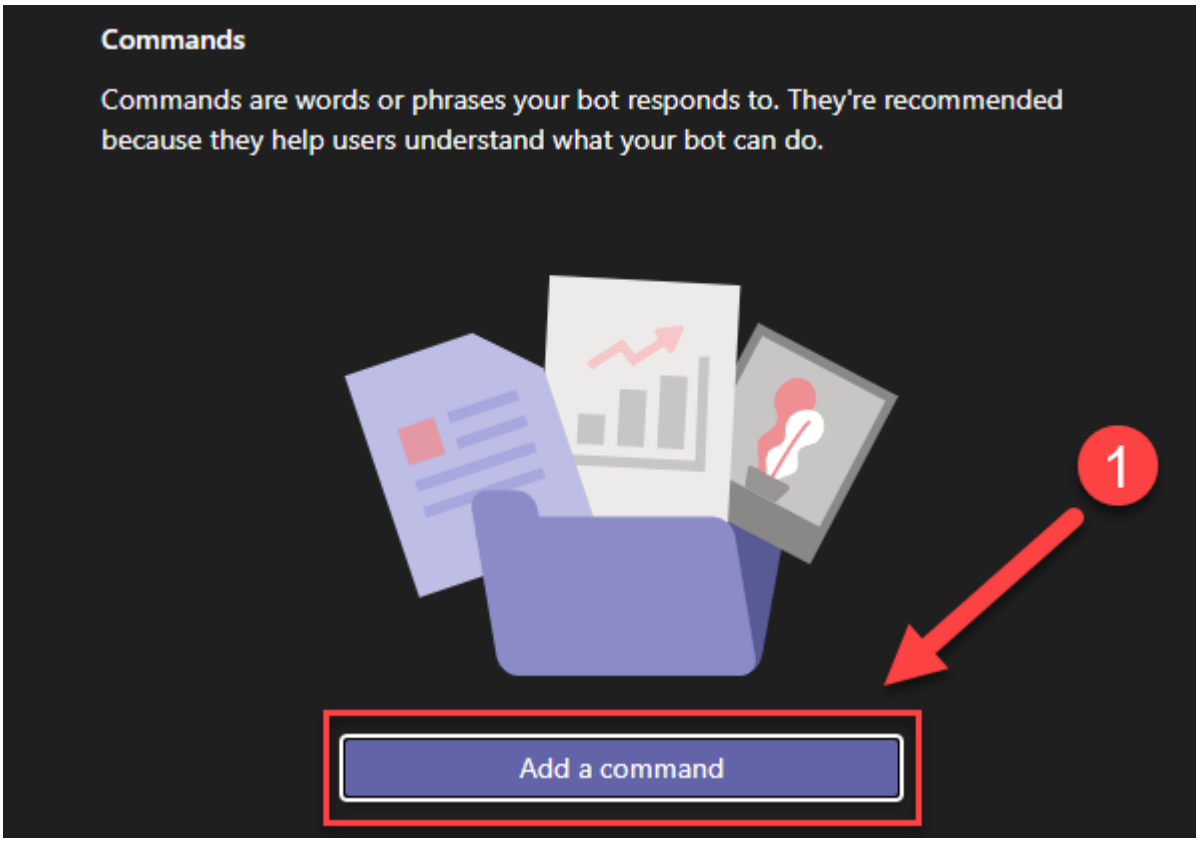
2. Once we are in the context of our application, we will add the bot. Select **App Features** from the left menu and choose **Bot** as the feature we would like to add.



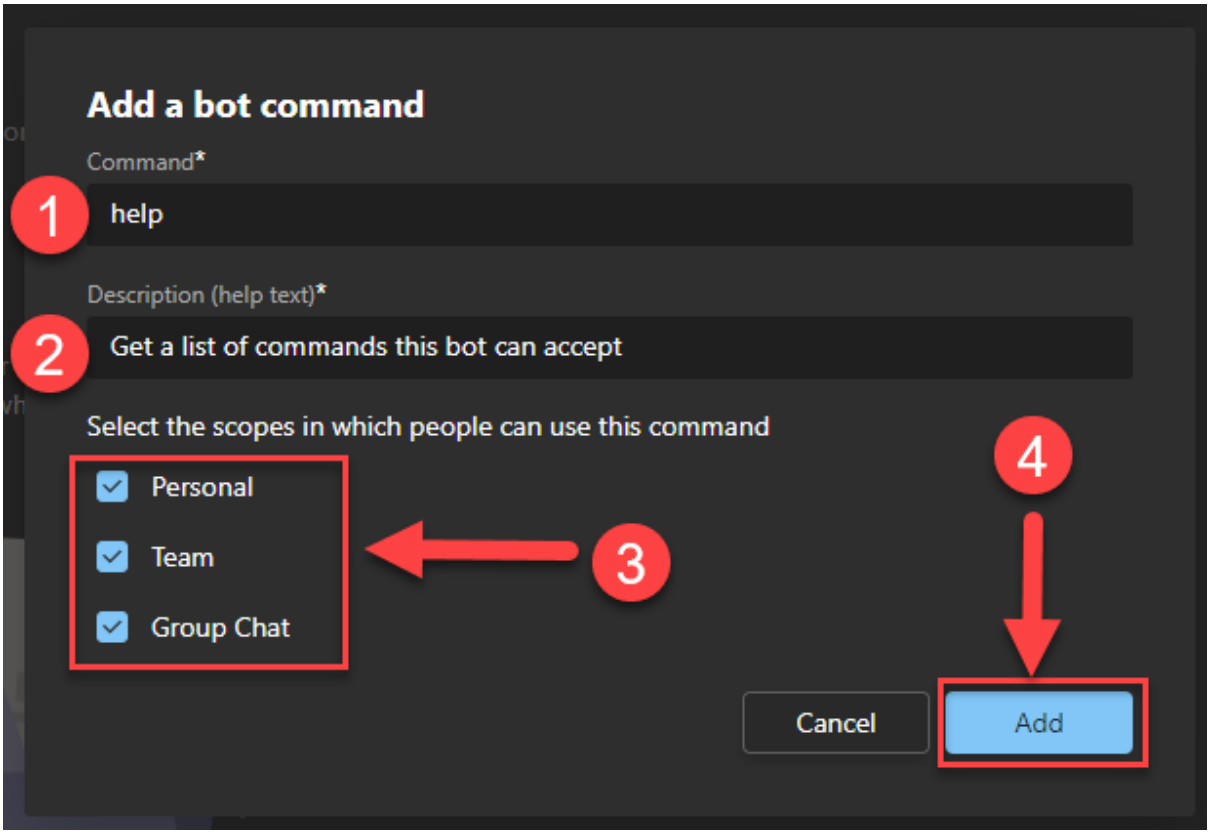
3. Here we will modify the configuration to use the bot service instance we created earlier. There is a bug in the Developer Portal here and adding an existing bot ID doesn't enable the Save button. In order to get around this issue, either **select an existing** bot first (if you have one, or **create a new bot**) Once the dropdown has a bot and the **Save** button has been enabled, click **Enter a bot ID** and copy in the **MicrosoftAppId** from the App Settings in Visual Studio. The Microsoft App Id (Client ID in Azure AD App Registration) and bot ID are the same, as there is a 1-2-1 mapping between Bots and Azure AD App Registrations. Select **Upload and download files** as this bot is able to work with files within Teams. Under scopes, select **Personal, Team and Group Chat.** Finally, click **Save** to save the manifest.



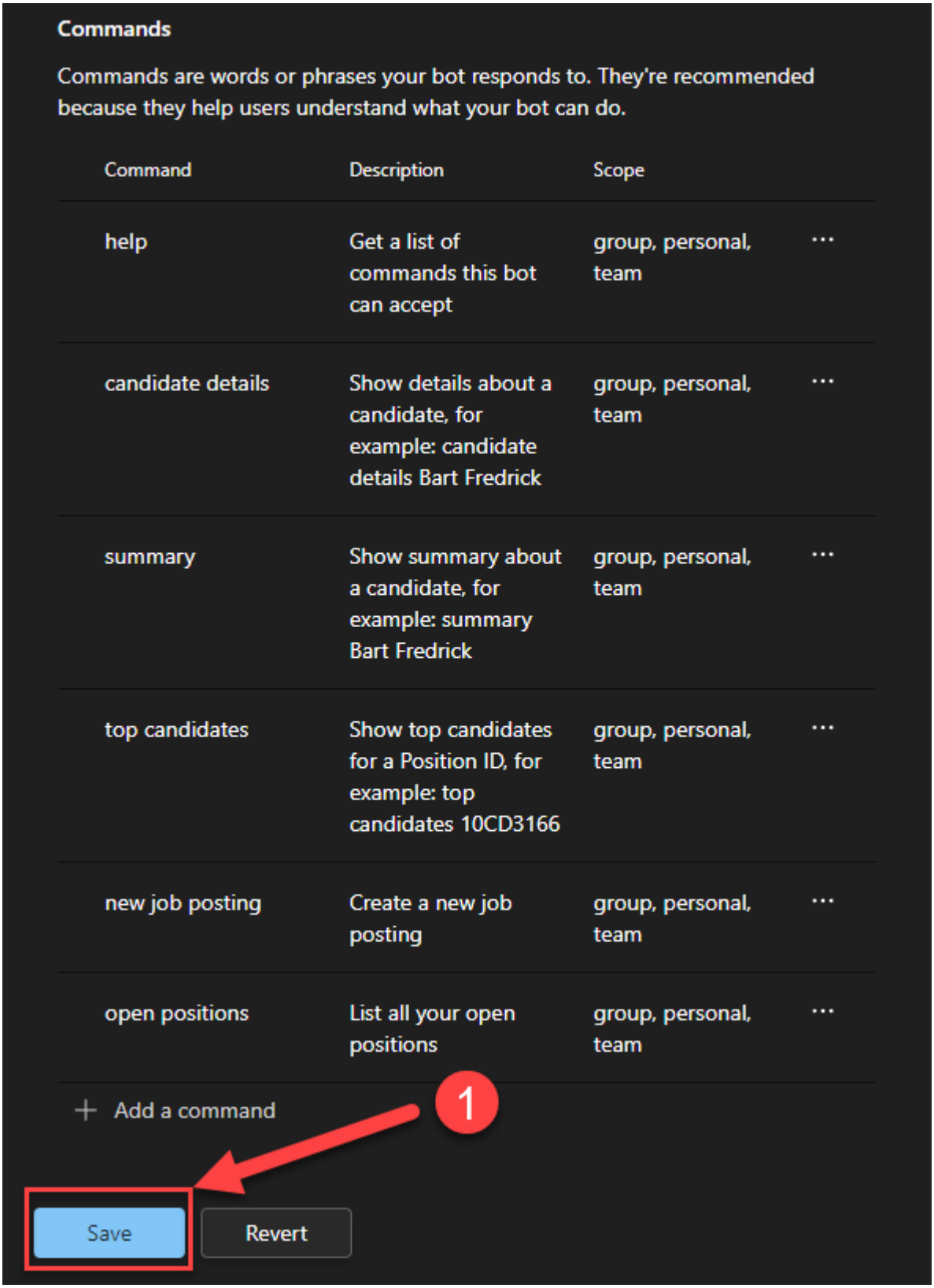
4. Now we want to add commands to the bot, a new Commands dialog should now be visible. Click the **Add a command** button.



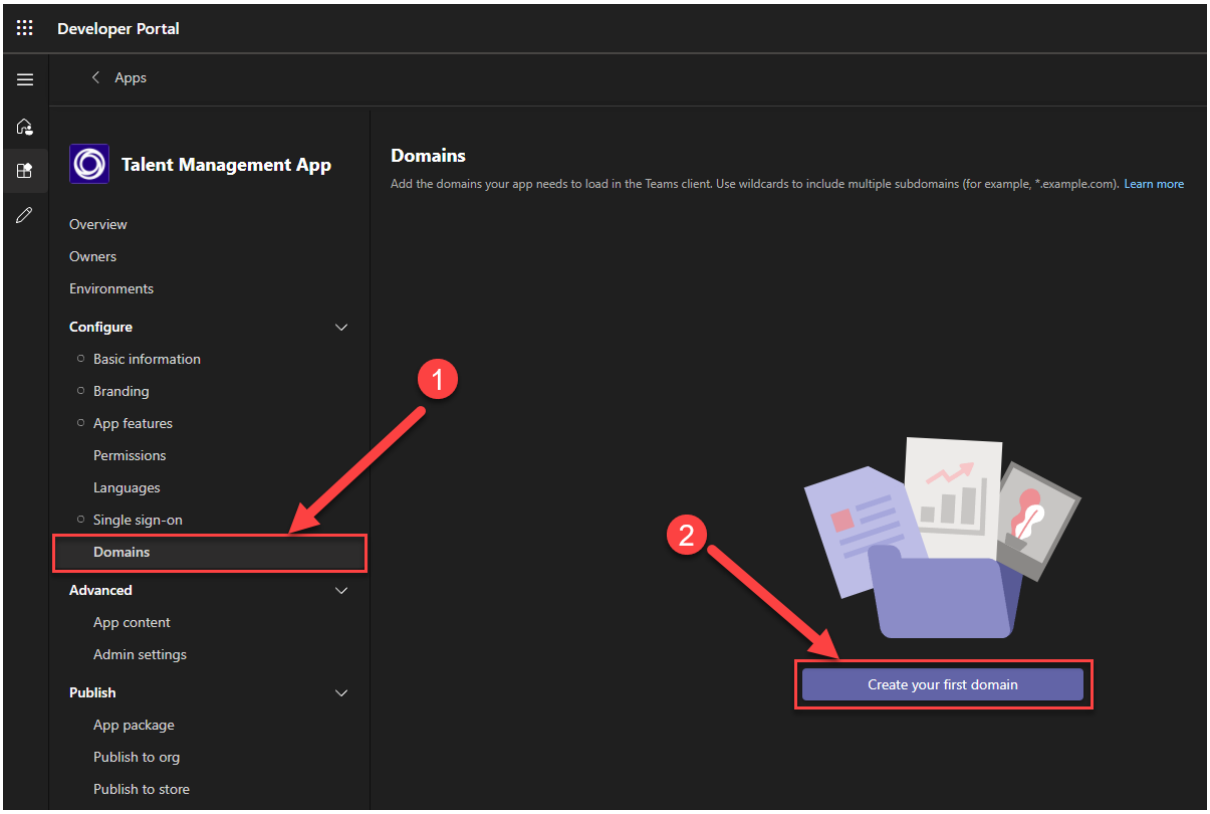
5. Now we will configure the command. Enter the details below and click **Add.**



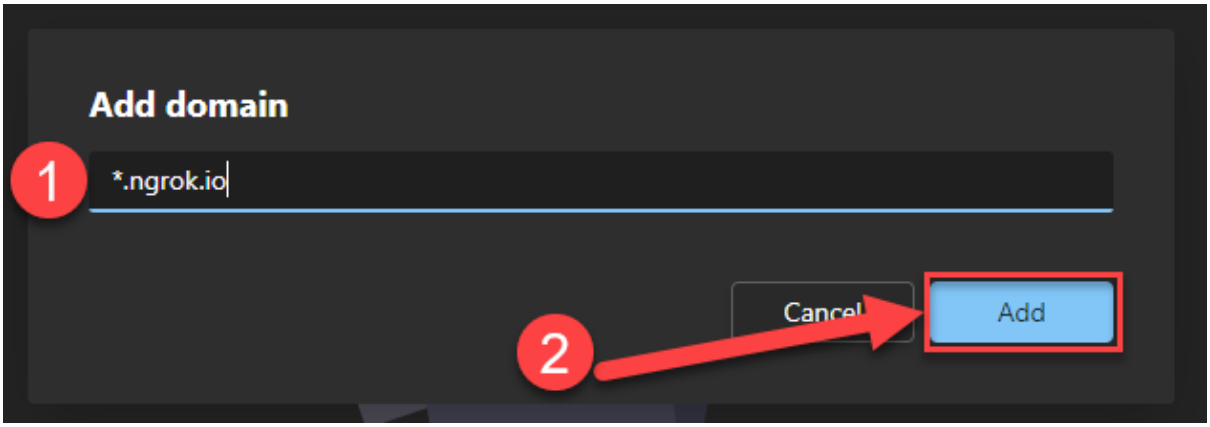
6. Repeat this process for all the commands the bot is able to perform as shown below.
NOTE: Ensure that you click **Save** after each command to ensure that you don't lose any configuration if the edit breaks!



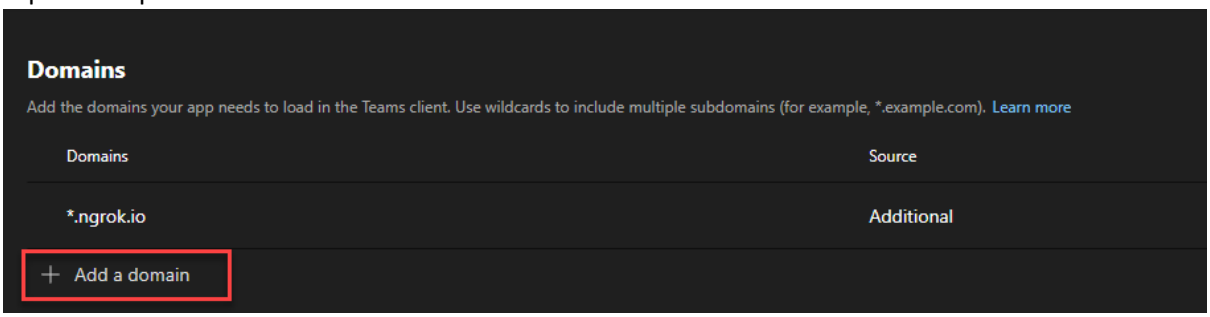
7. Next we need to define which domains are allowed to be opened within the iFrame, this needs to include Bot Service domains (for fallback authentication) and the domain your local app instance is using (the NGROK domain). Select **Domains** from the left menu and then click **Create your first domain**.



8. There are two domains we will need to add, the first will be your API service. As we are using ngrok to tunnel, add the wildcard domain ***.ngrok.io** and the click **Save**.

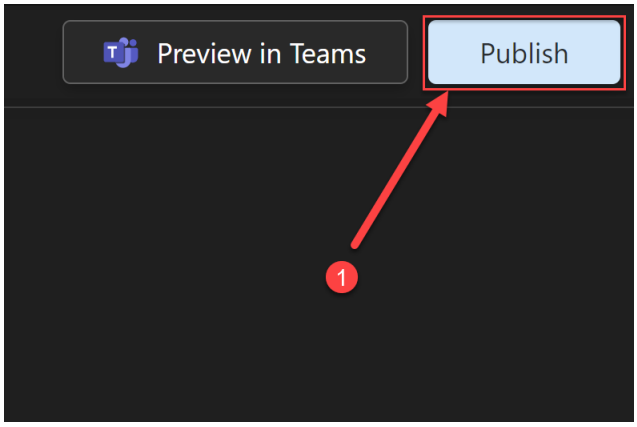


9. You will now see that your domain appears in the domains list. Click **Add a domain** and repeat this process.

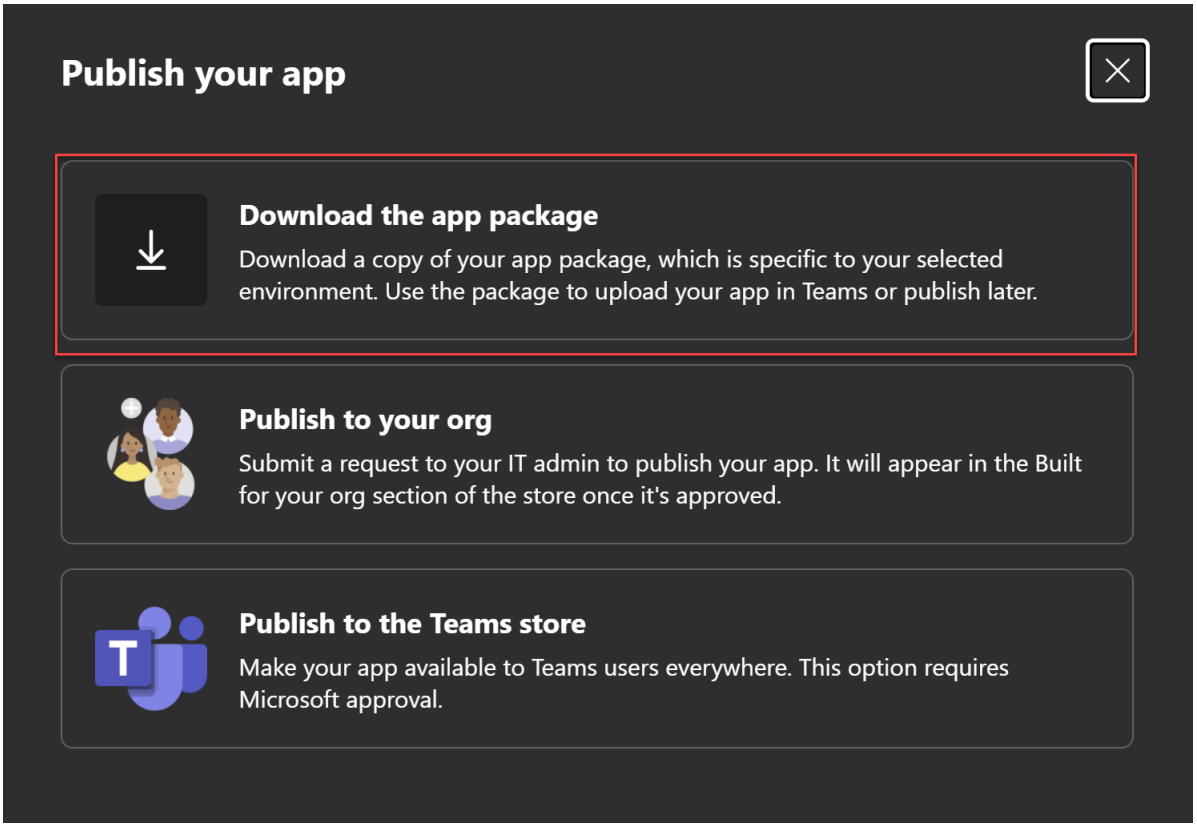


10. We will now add the domain that bot framework uses for fallback authentication, with our bot, if SSO fails. Enter **token.botframework.com** and click **Add**.

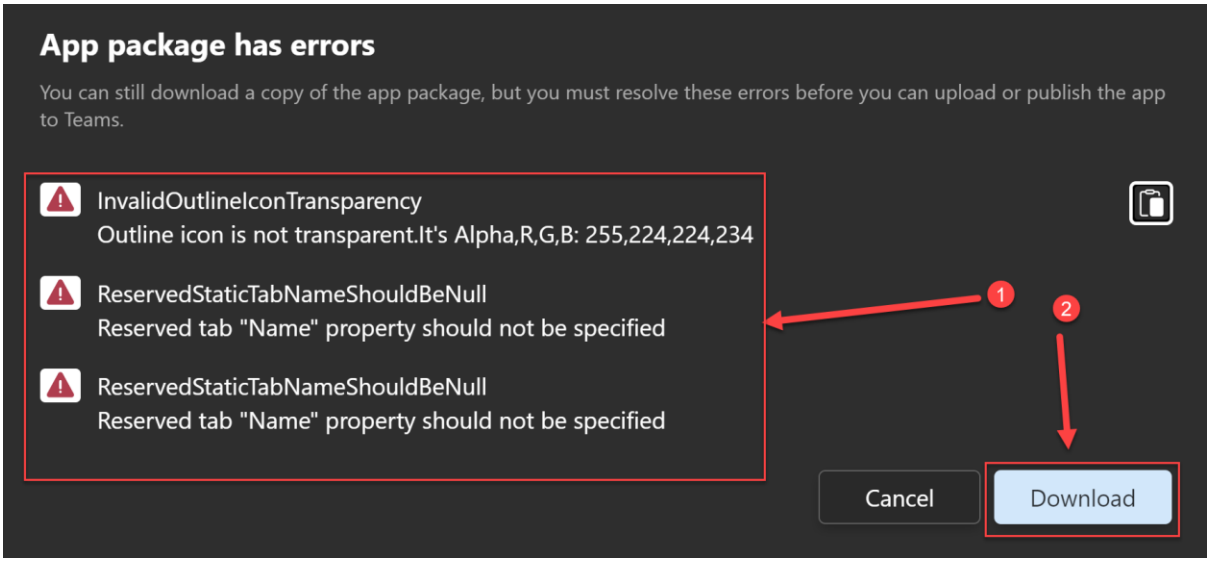
11. Now that the bot has been configured completely, let's test it out! Instead of sideloading the app for a specific user, this time we need to publish the app to the organisation, as this will ensure that the Teams App ID, defined in the manifest, does not change (sideloaded apps for specific users have random Teams App IDs). Click **Publish**.



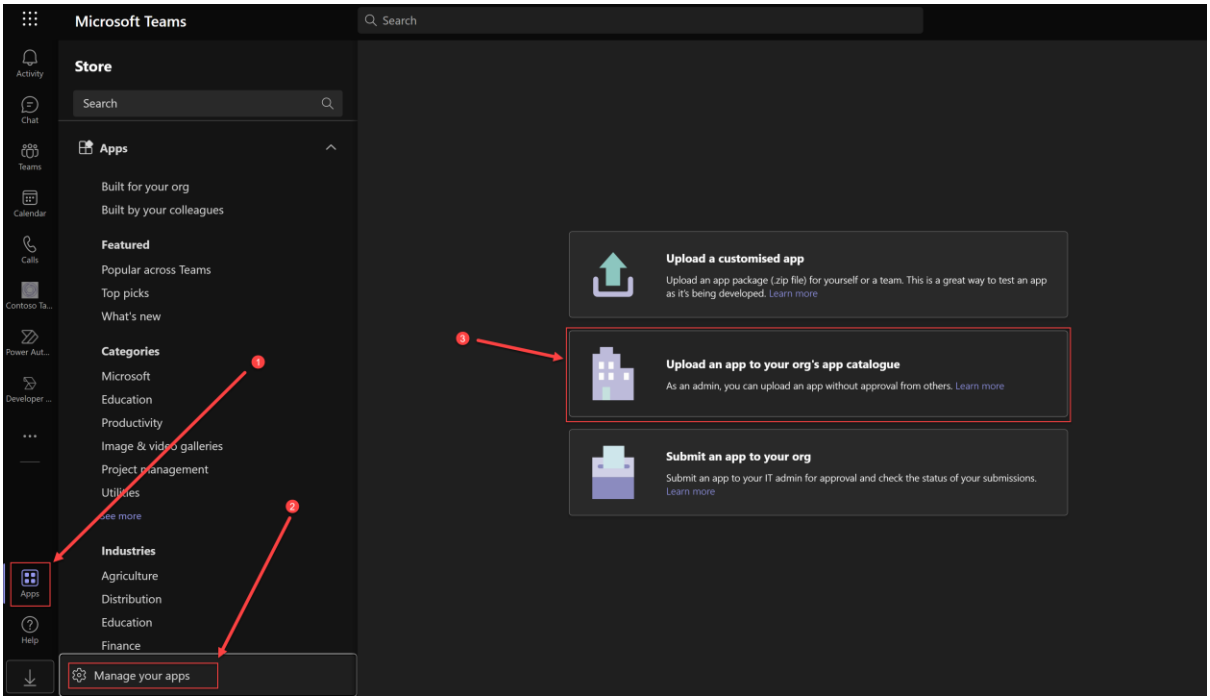
12. Click **Download the app package**, this will save the app manifest ZIP into your Downloads folder.



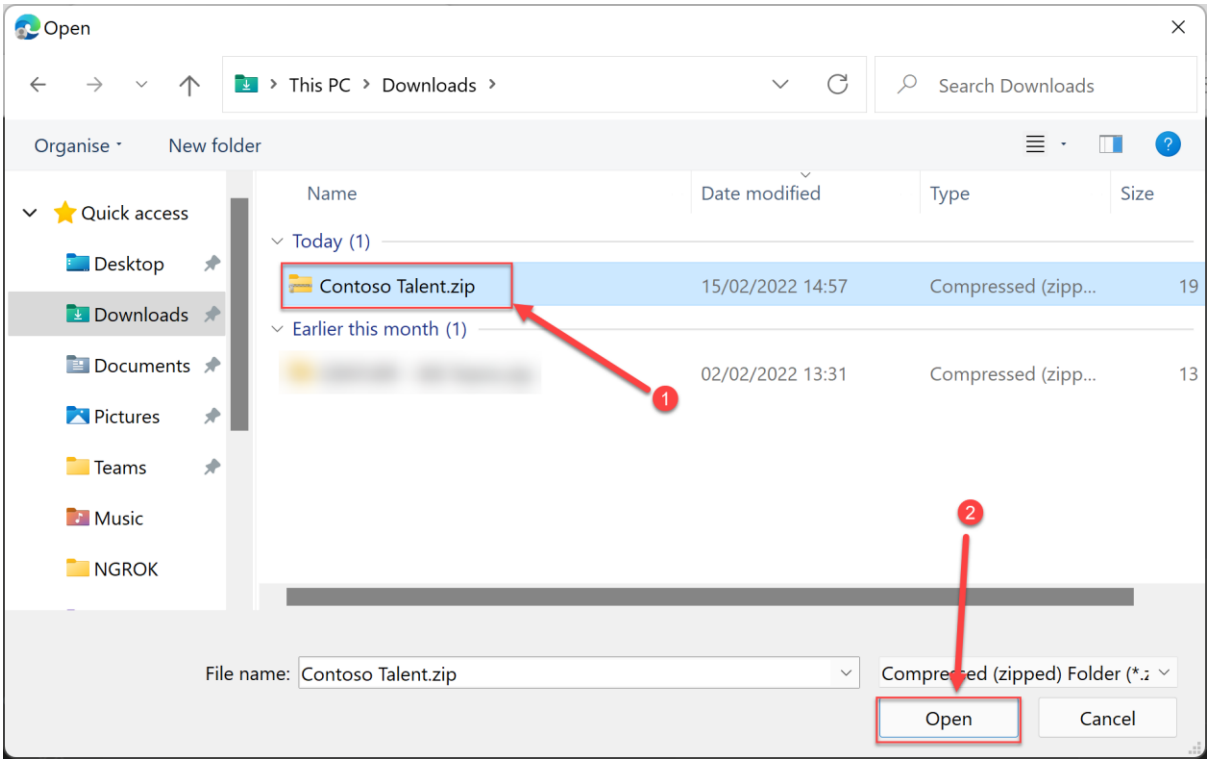
13. If the App package has errors, they will display here. Some errors must be fixed before you can install the app into Teams (note, in the screenshot below, all 3 errors can be safely ignored), **review and fix the errors**, click **Download**.



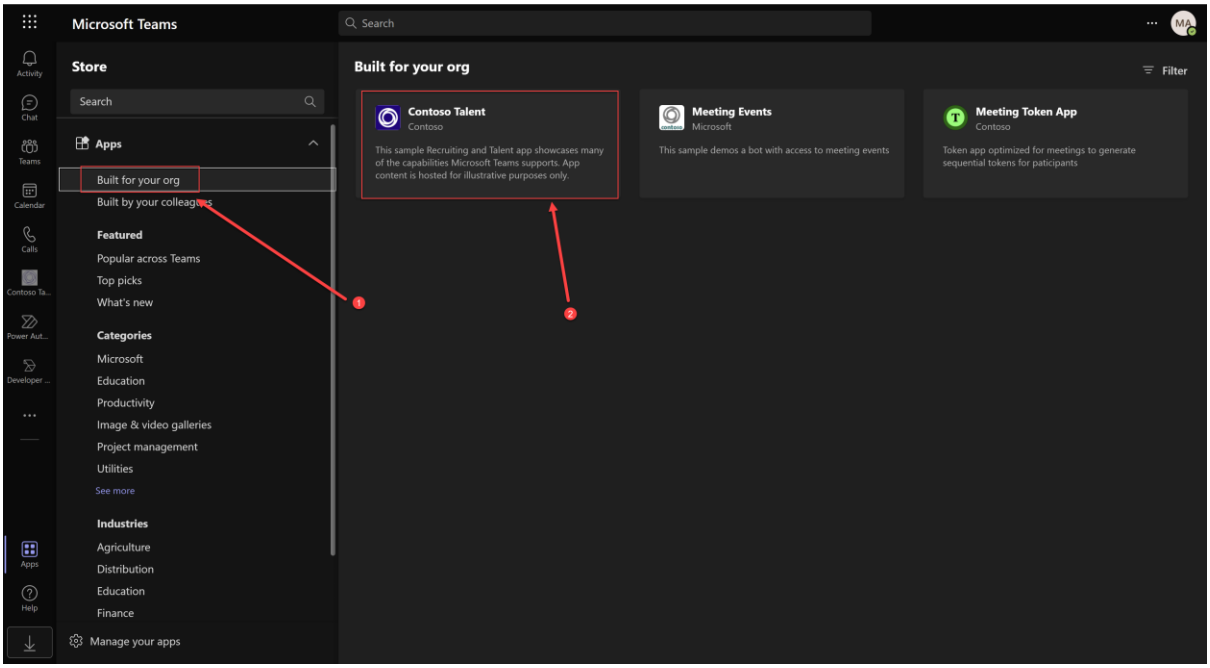
14. Go back into Teams, select **Apps**, then click **Manage your apps**, click **Upload an app to your org's app catalogue**. ***Note: this option will only exist if you are logged into Teams as a Global Administrator, if you are not, log-out and log-in as a Global Administrator.***



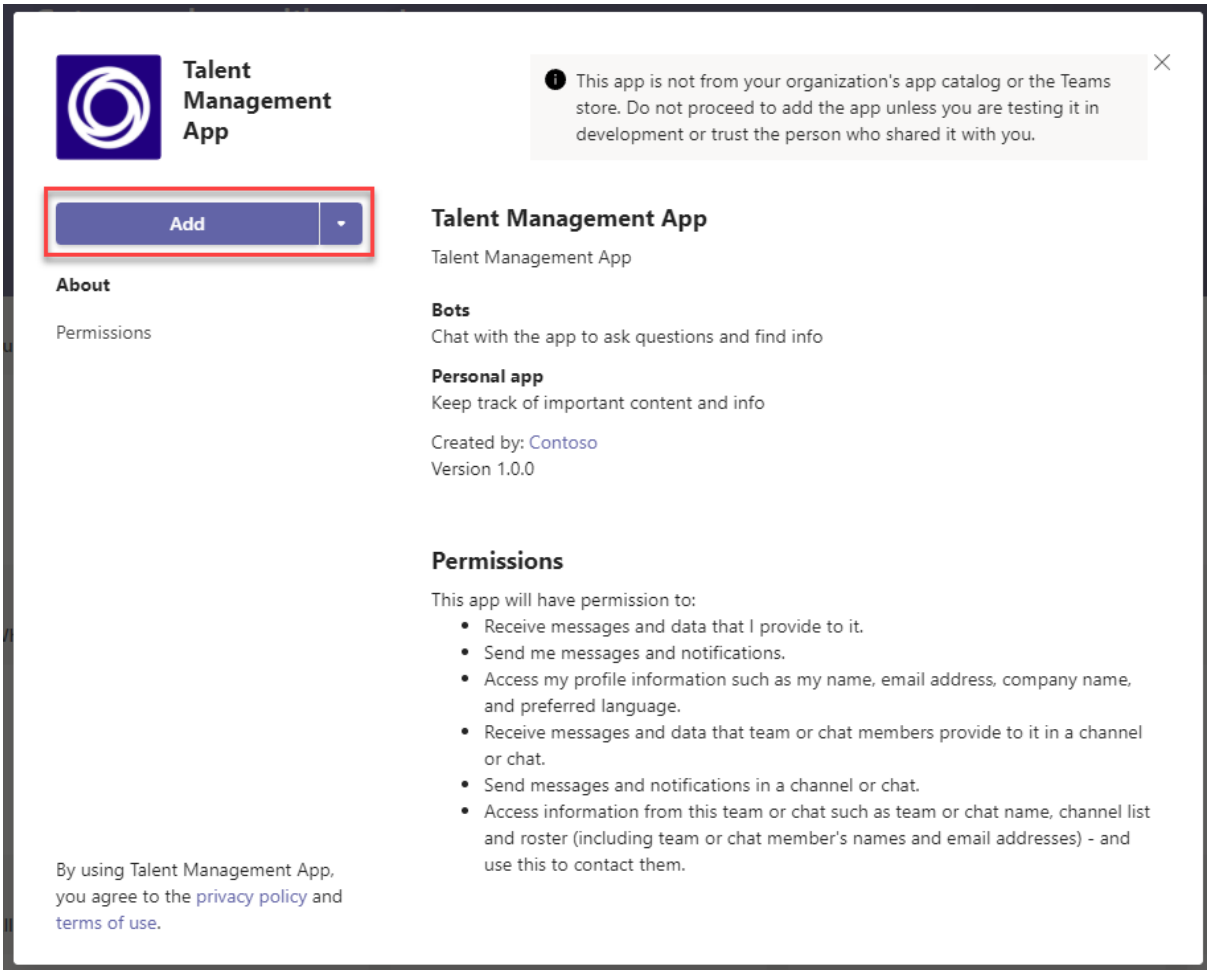
15. Navigate to the directory (folder) where the Teams App Manifest was saved (probably Downloads). **Select the App Manifest ZIP**, and click **Open**.



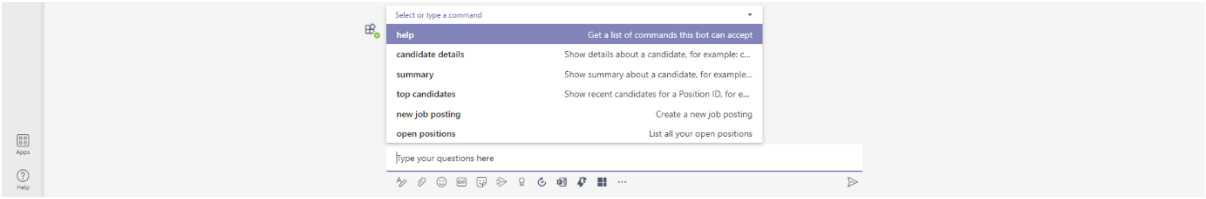
16. Click **Built for your org**, and select the **Contoso Talent** Teams app.



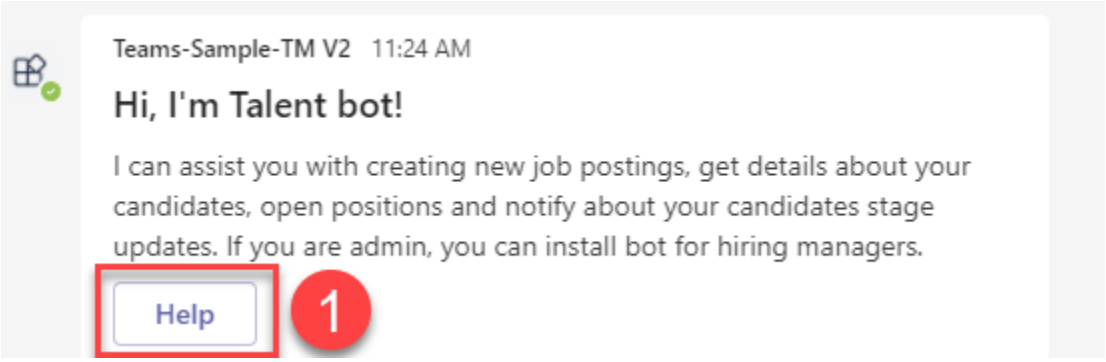
17. Click **Add** to install the latest version of this application, which includes the Bot.



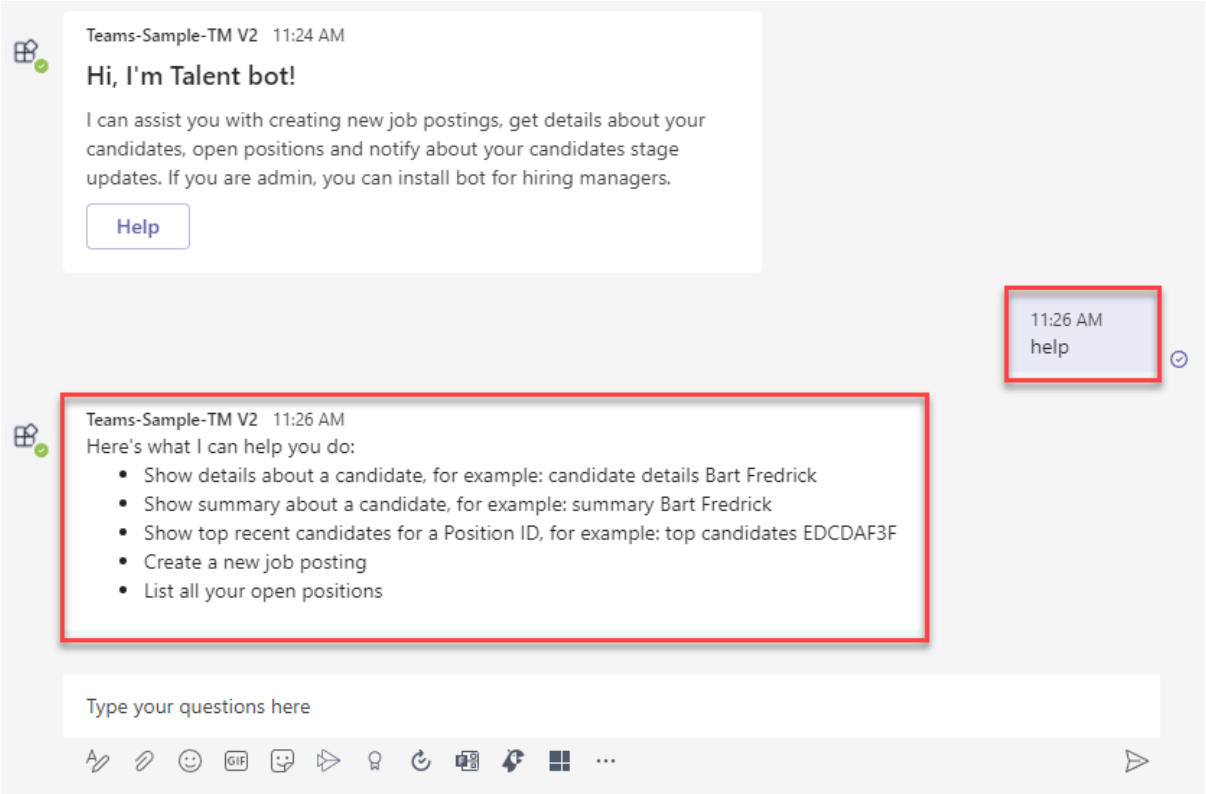
18. When you highlight the Compose box, you will now see that you are presented with a list of commands available through the bot. You can either click a command or type the name. Note: that the bot will have also added a welcome message but depending on the size of your window it might be hidden by the bot command list. Once the list is dismissed you will see the welcome message that was sent by your bot.



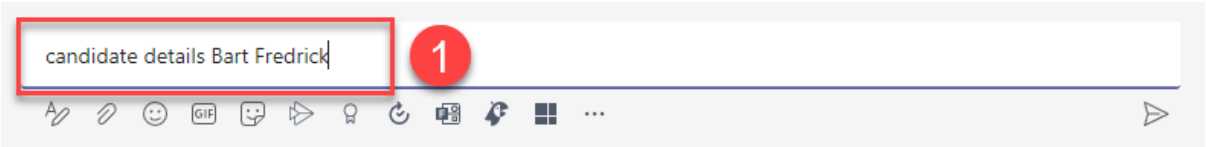
19. This message includes an action button which has been configured send the help command to the bot. Clicking **Help** will send the text "help" to the bot just as if you'd typed it yourself in the compose box. ***Note: This is using the 'imBack' (instant message back to bot) action type, from within the Adaptive Card, to send the 'help' message back to the bot when you click on the button.***



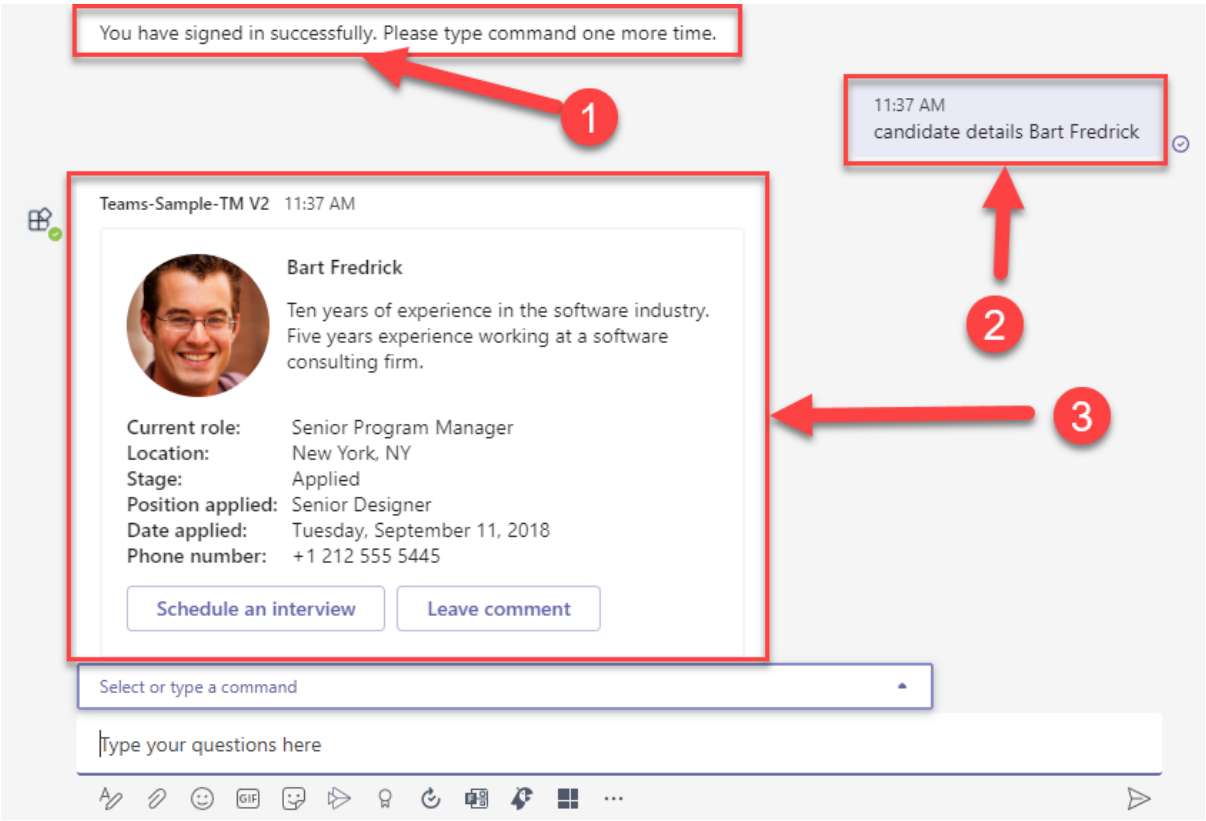
20. Once the command has been received by the bot it will respond with a friendly help message listing some of the common commands.



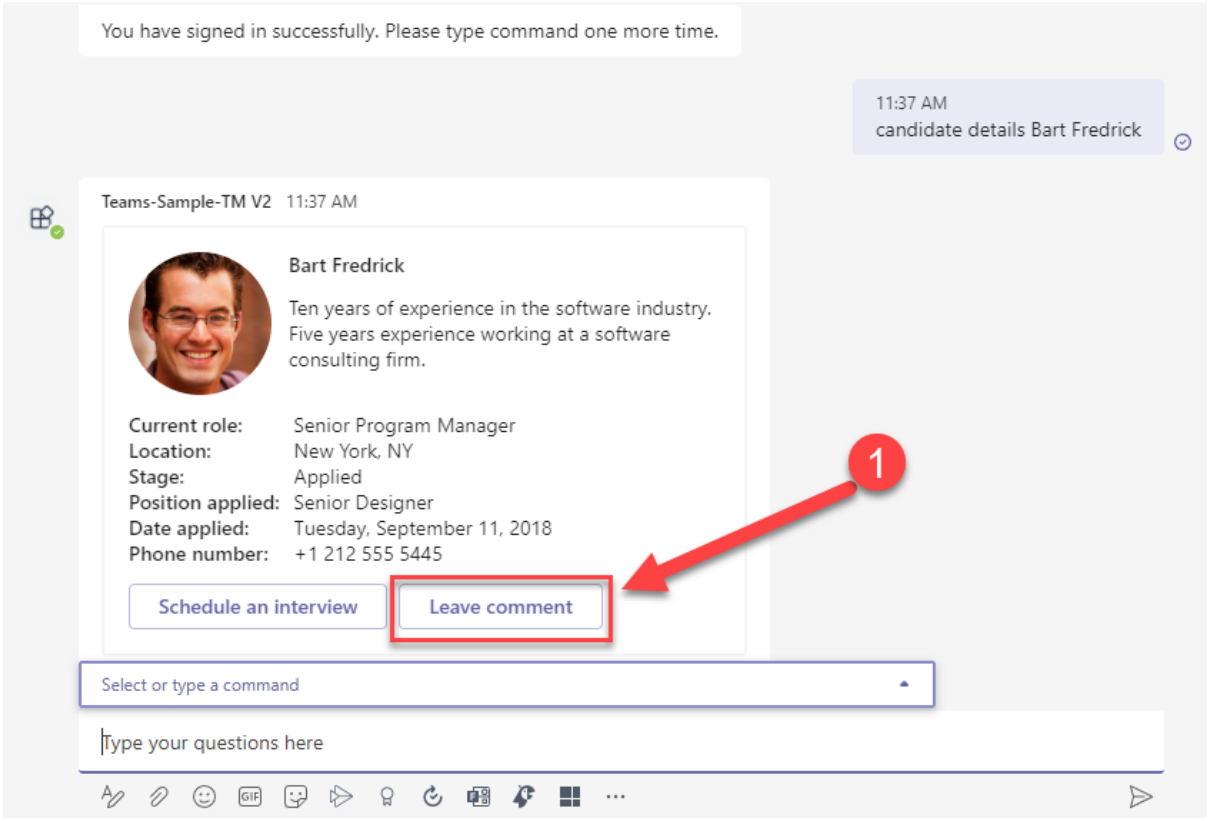
21. One such command is the candidate details command. Let's try this now… copy or type **candidate details Bart Fredrick** to see details about the candidate, and send it to the bot.



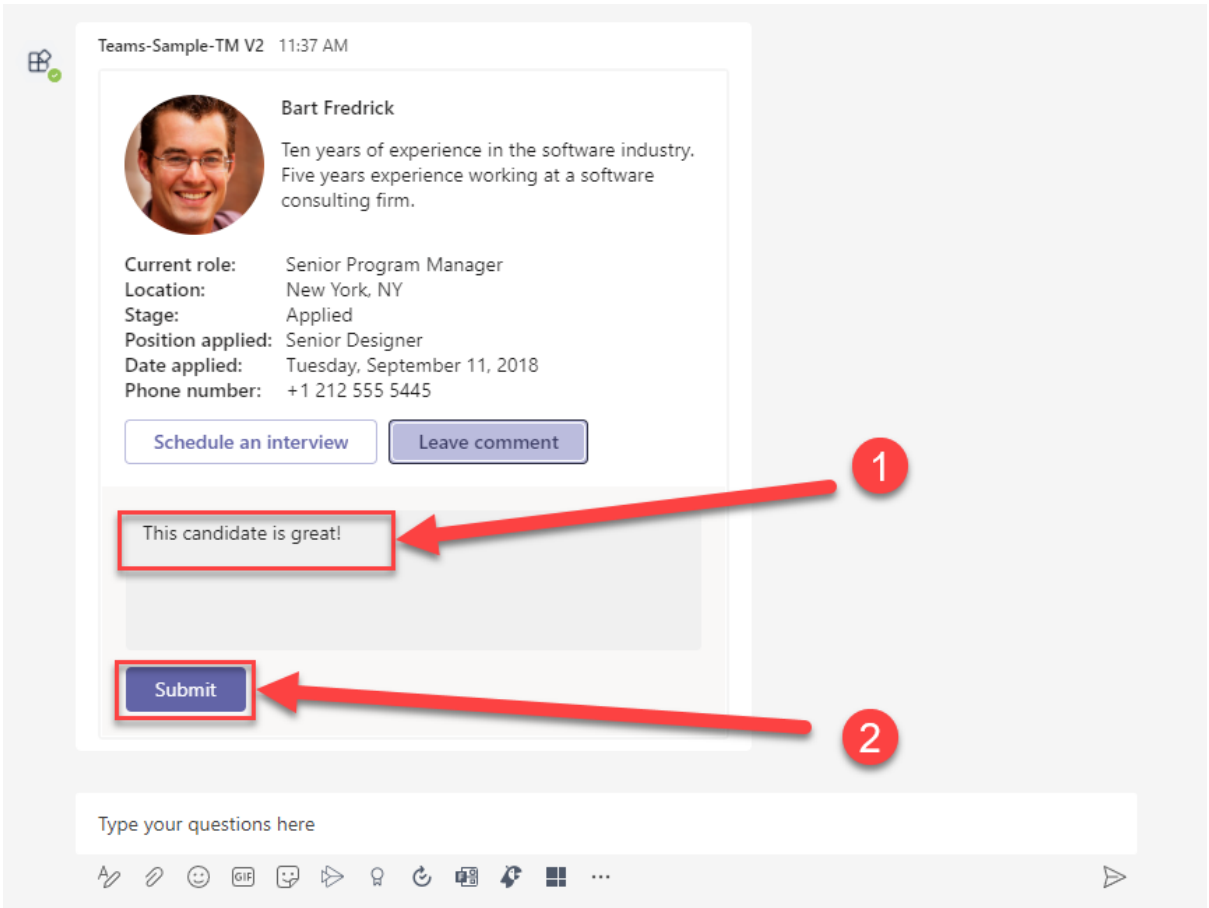
22. You will notice that before any candidate details are shown you are presented with a message saying that you have successfully been signed in. This is the single sign-on process working in the background. The message will also say that now you are signed in you must resend your command. Doing this will display an Adaptive Card displaying details about Bart Fredrick.



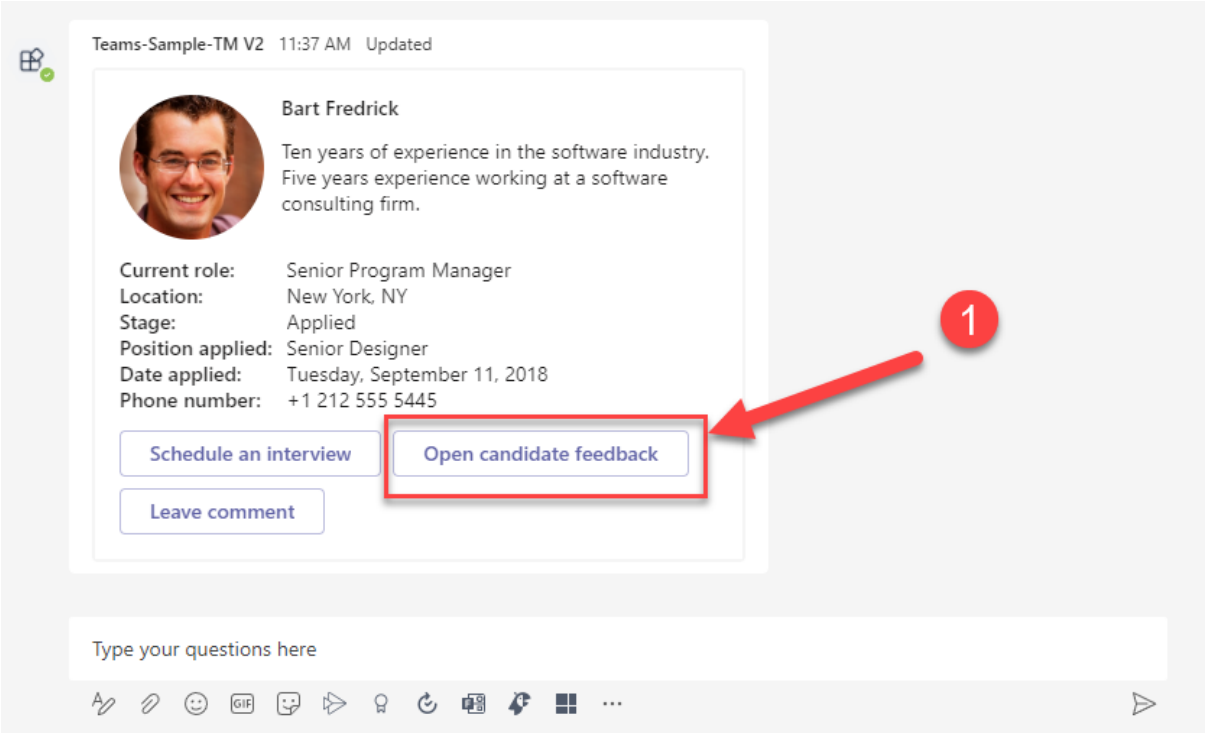
23. You will also notice that there are some buttons on the card. Clicking **Leave comment** will display a textbox giving you the ability to leave a comment about the candidate.



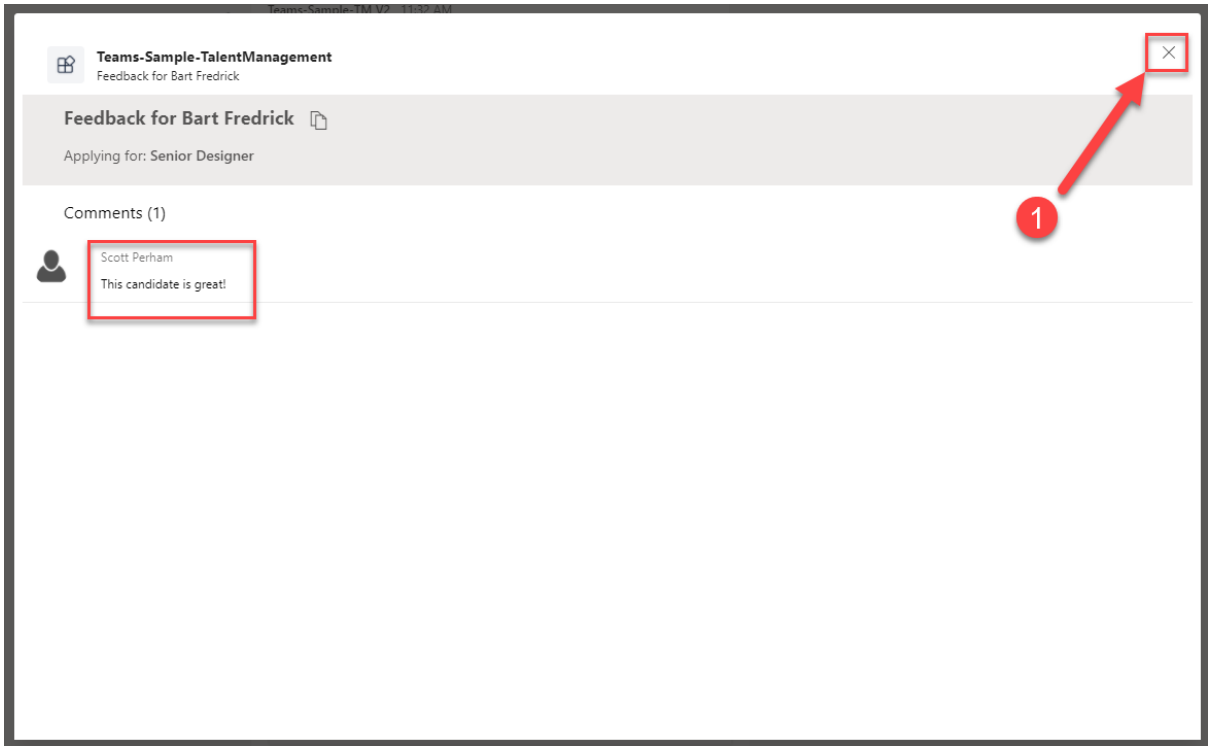
24. Enter a comment and click **Submit** to send the comment to the bot.



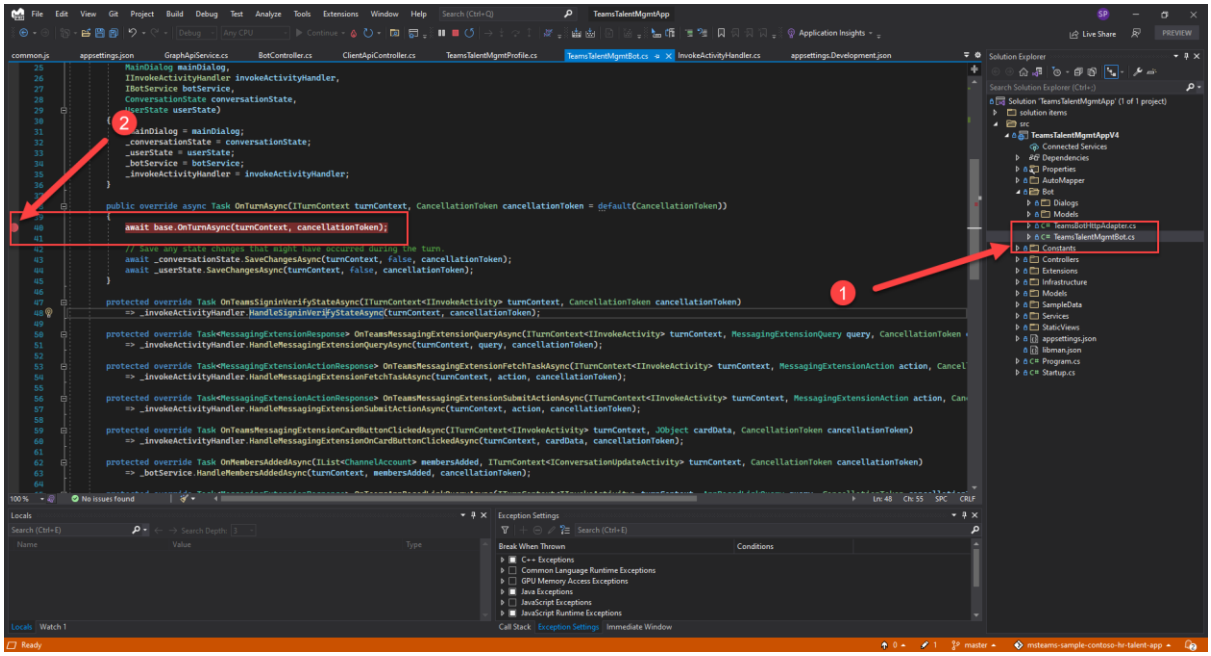
25. This will resend the candidate adaptive card to Teams and because we now have some feedback, and additional (conditional) button will be shown. Click **Open candidate feedback** to see a summary of all the feedback left for the candidate.



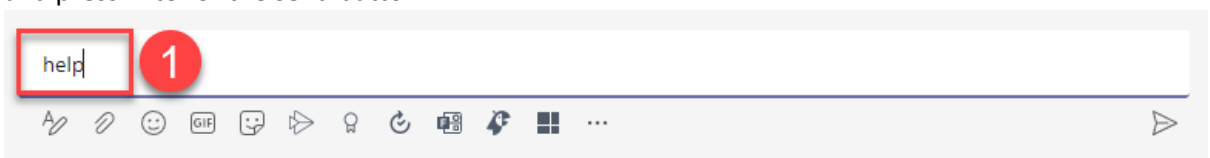
26. You will see your feedback in the list! ***NOTE: If the feedback does not appear, this is likely because the Teams SSO within the Task Module has failed. In our experience this is usually because the Teams App ID (associated with the invoke task module button) that is being passed down in the Adaptive Card (from the code), does not match the Teams App ID in your Teams App manifest. If this is the case, you will see 401 unauthorised displayed in the network tab of the browsers developer tools, when the task module is initialised and it tries to pull data from the Client APIs. In most circumstances this mismatch is because the Teams App hasn't been installed into the Organisational App Store, and instead, has been sideloaded for a specific user. When Teams apps are sideloaded for a specific user, the Teams App ID is randomised. To fix this, go back to step 11 in this section and install the app into the Teams Organisation App Store, and try again.*** Once you are done click the X in the corner to close the Task Module.



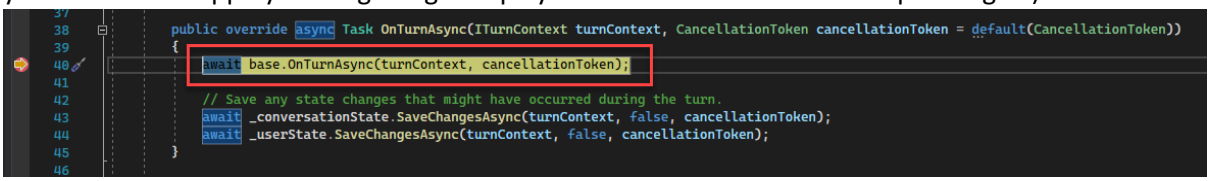
27. We will now have a look behind the scenes and see what happens when commands are sent from Teams to your bot. Open Visual Studio and navigate to the TeamsTalentMgmtBot.cs file. Then put a breakpoint on line 40. (Breakpoint can either be set by clicking the leftmost gutter on the appropriate line, or by clicking somewhere in the line and hitting F9). The line will turn red once the breakpoint has been set.



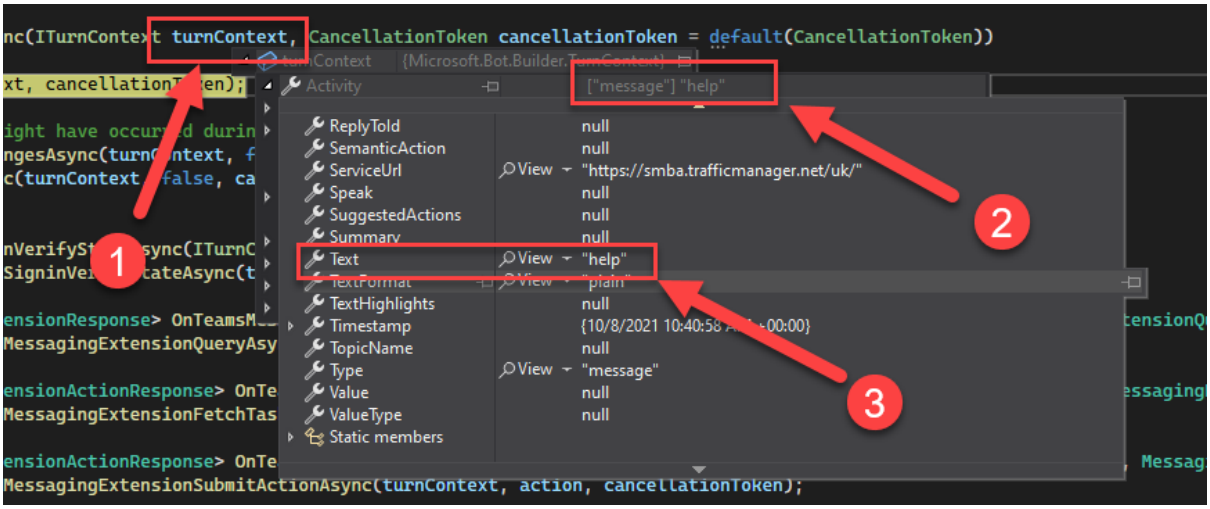
28. Now with your application running, go back to Teams, in your bot chat channel, type “help” and press *Enter* or the send button.



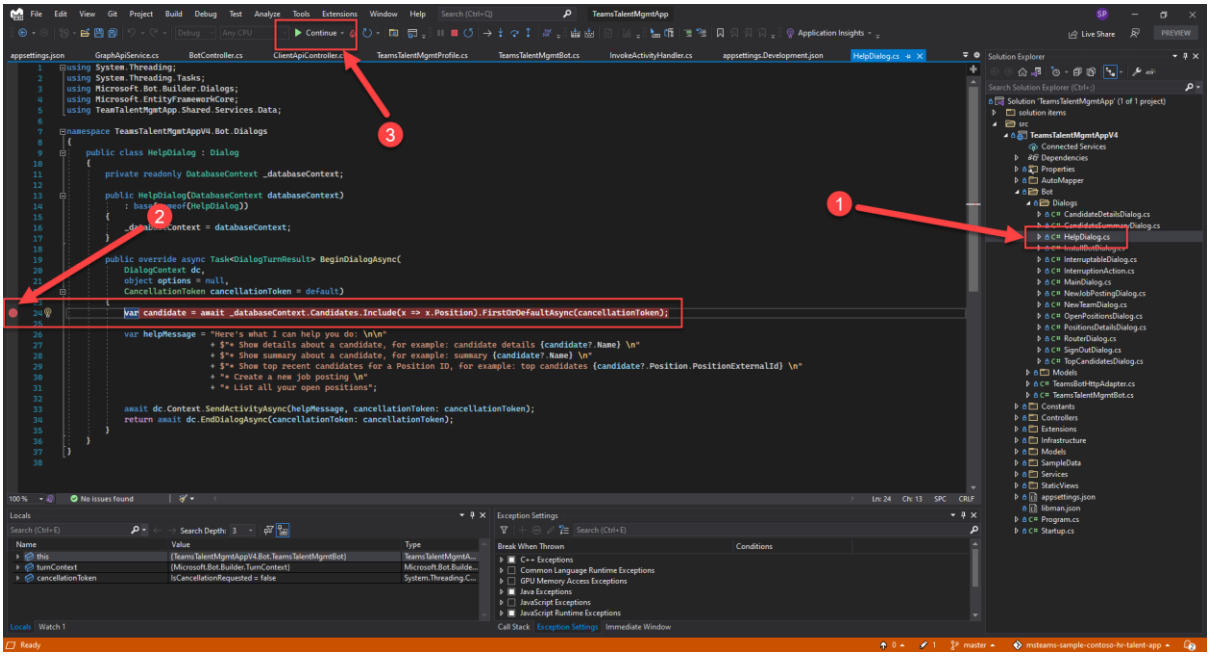
29. You should see that Visual Studio has halted execution and highlighted line 40 in yellow. This means that this is the current line of execution but the breakpoint has caused the application to pause. (Note that while you have halted execution the bot will not work until you resume the app by clicking the green play button from the toolbar - or pressing F5).



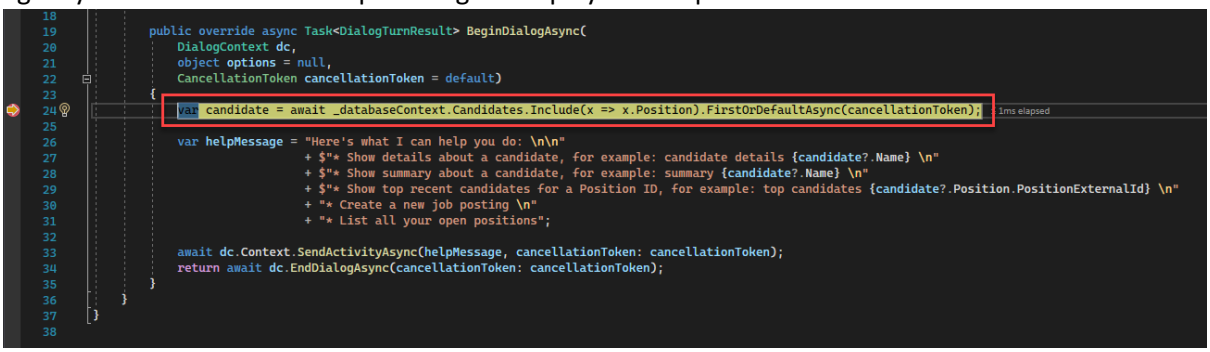
30. While we are paused Visual Studio allows us to inspect the value of any variable in the current scope. As we are in the scope of a method called **OnTurnContext**, we can inspect the parameters that were passed to it. To do this, hover your mouse over the parameter called *turnContext* then expand the various treeview items in the pop-out to see the values. In the case of the image below, we can see the text that was passed in the message.
Note: In bot framework, conversational concepts are used to describe the various parts of bot interaction. In the same way that two humans might take turns speaking in a conversation, a Turn in bot framework represents your bots chance to respond to an event.



31. Another concept in bot framework is that of Dialogs. A dialog represents a part of a conversation. This could be a single text response or a conversation that spans multiple turns. It allows us to encapsulate the functionality of a particular conversation.
One such dialog is the HelpDialog. Open the **HelpDialog.cs** file and **set a breakpoint on line 24**. Click the **green play button** to resume the bots execution.

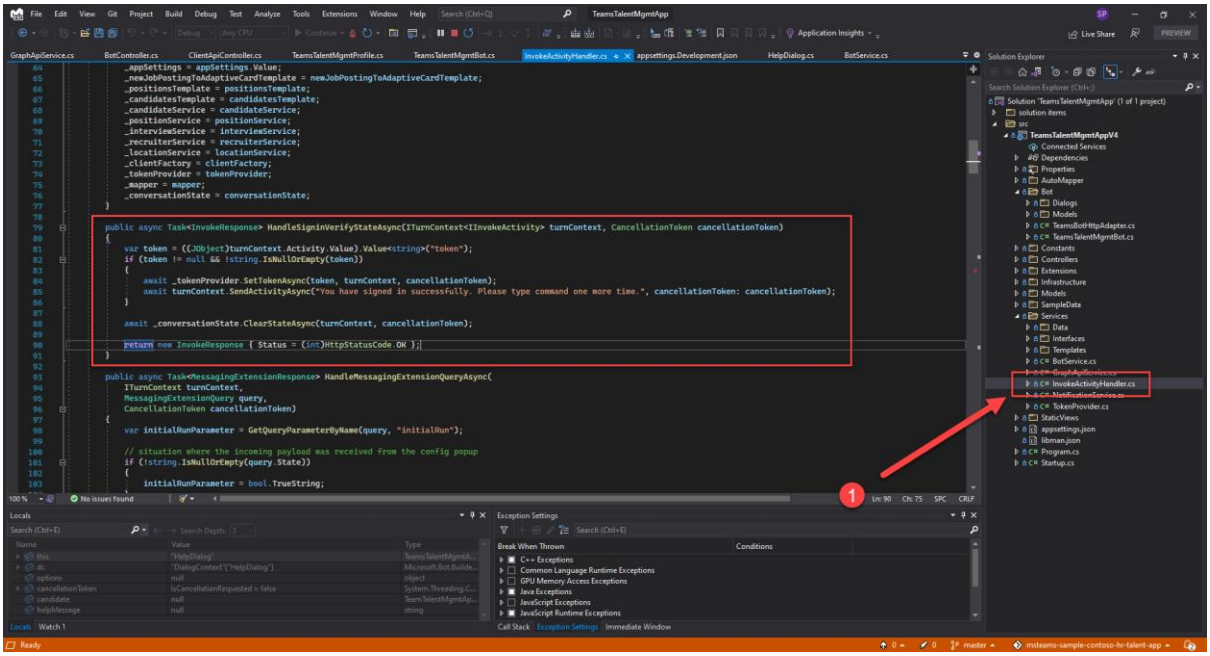


32. You will see that your breakpoint has been hit! As before you can inspect the various variable values if you wish.
    Finally, this method will send the **helpMessage** back to Teams.
    Note: If an execution takes more than a few seconds, Teams will ignore the response as it will assume that the bot has broken and is not responding. You may therefore find that when you resume execution from this point the help message may not actually appear in Teams. This is expected! If you remove both of the breakpoints and send the *help* command again you will see that the help message is displayed as expected.

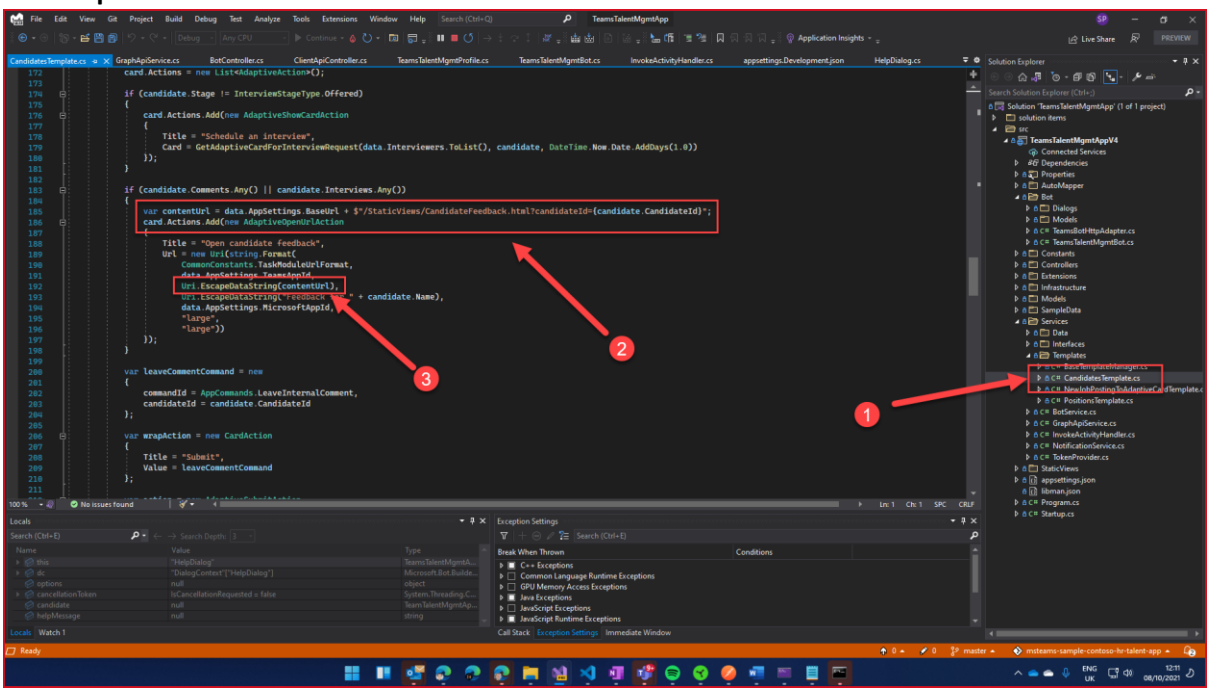


```
        public override async Task<DialogTurnResult> BeginDialogAsync(
            DialogContext dc,
            object options = null,
            CancellationToken cancellationToken = default)
        {
            var candidate = await _databaseContext.Candidates.Include(x => x.Position).FirstOrDefaultAsync(cancellationToken);

            var helpMessage = "Here's what I can help you do: \n\n"
                            + $"* Show details about a candidate, for example: candidate details {candidate?.Name} \n"
                            + $"* Show summary about a candidate, for example: summary {candidate?.Name} \n"
                            + $"* Show top recent candidates for a Position ID, for example: top candidates {candidate?.Position.PositionExternalId} \n"
                            + "* Create a new job posting \n"
                            + "* List all your open positions";

            await dc.Context.SendActivityAsync(helpMessage, cancellationToken: cancellationToken);
            return await dc.EndDialogAsync(cancellationToken: cancellationToken);
        }
    }
}
```

33. There are two more areas of interest in the code that you may be interested to see. The first is the code for SSO. Most of this is handled for us in a dialog provided by the framework, but we still have to handle the users security token once it's been returned to us. This is particularly useful if you want to cache the token for use in the future, perhaps exchange it for a graph token to be able to call graph on the users behalf?
    To see how this sample has dealt with the token, open the **InvokeActivityHandler.cs** file and inspect the **HandleSignInVerifyStateAsync** method.



34. Another area of interest is how the bot responds with adaptive cards. In the CandiateTemplate.cs file is the code used to create a relatively complex card. The image below shows how the bot is configuring the candidate feedback task module to be displayed when **Open candidate feedback** is clicked.

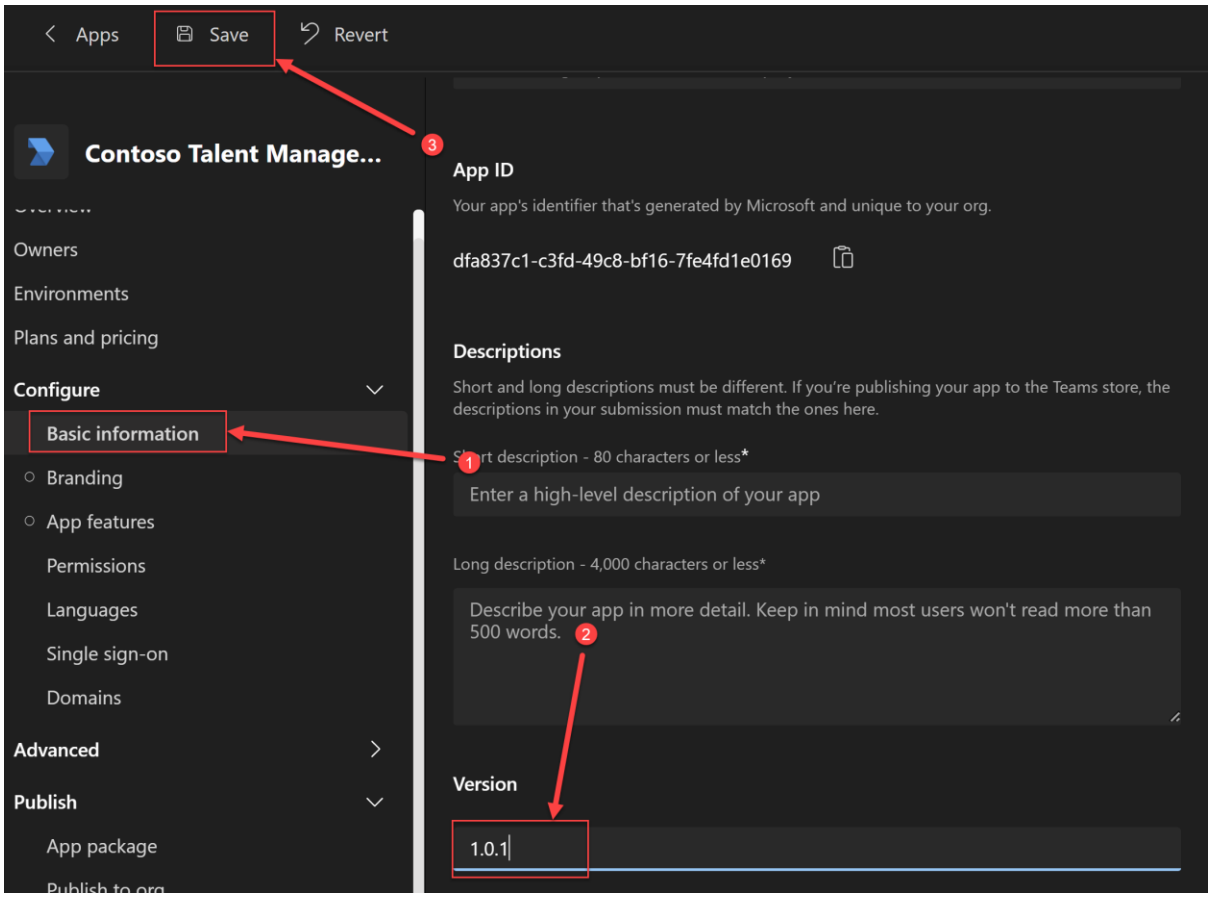


Now that the ChatBot (in a personal context) is working, we will now implement a more advanced Bot feature in Teams, known as a Messaging Extension. Messaging Extension search is where we will begin, and then we'll move onto Messaging Extension actions.

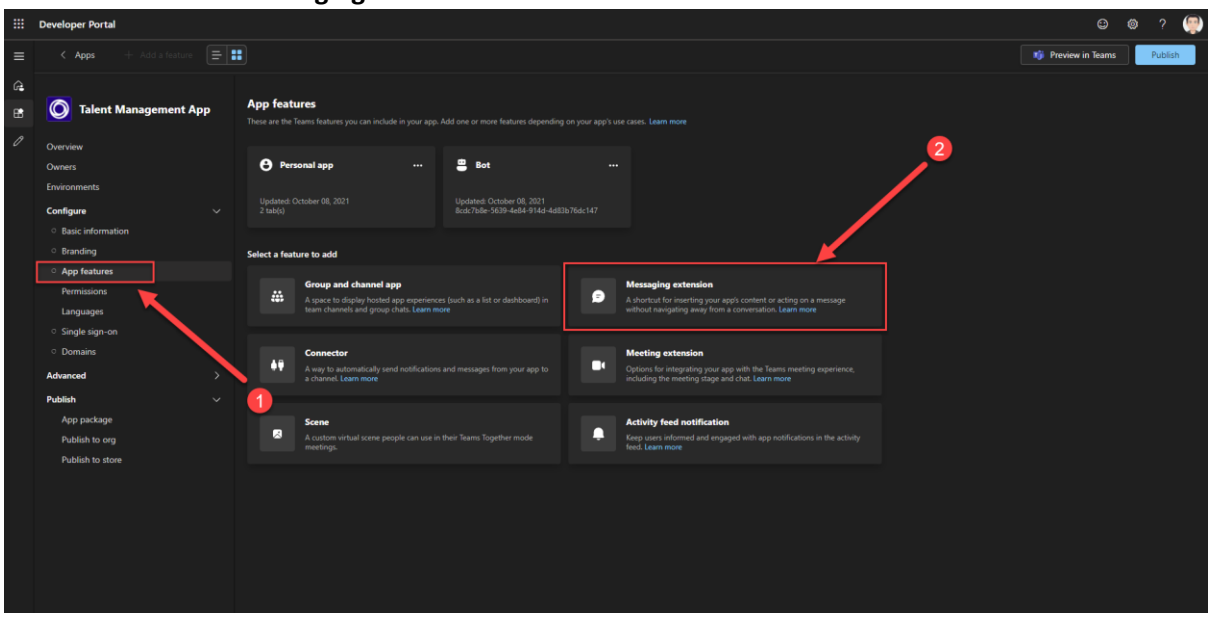
5) Implement a Message Extension Search

In this step we will extend the functionality of Teams to include a search function. Searching for candidates and positions using a message extension

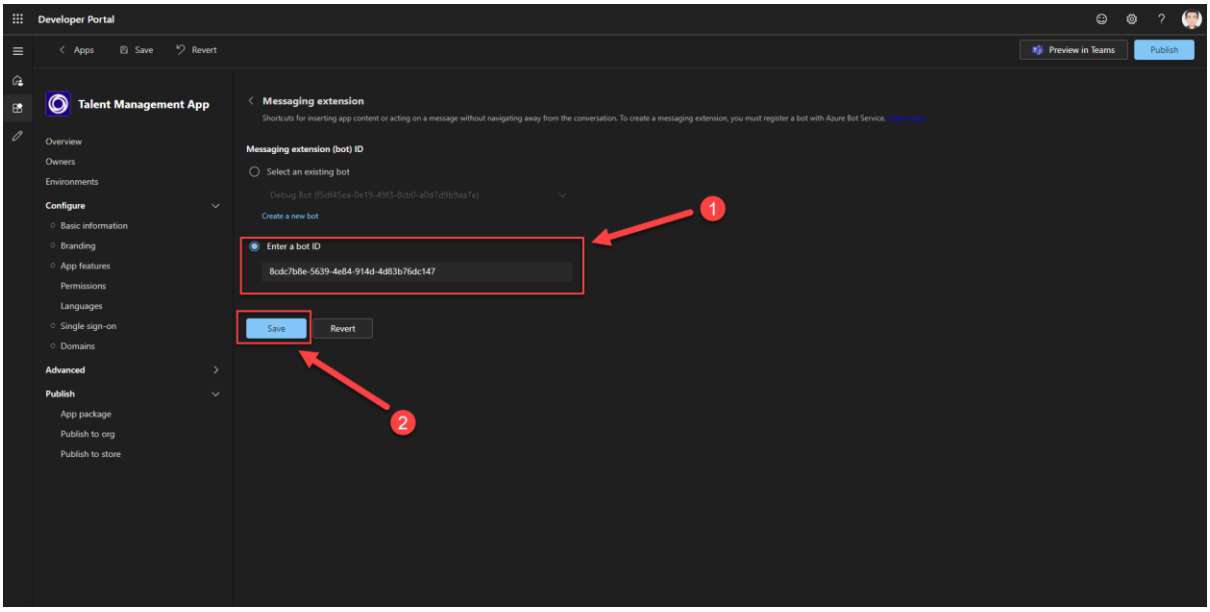
1. As with everything else in the manifest we need to go back to the Developer Portal to add a new feature. Let's start with iterating the version number, click **Basic information**. Scroll down and iterate the number from 1.0.0 to **1.0.1** – click **Save.**



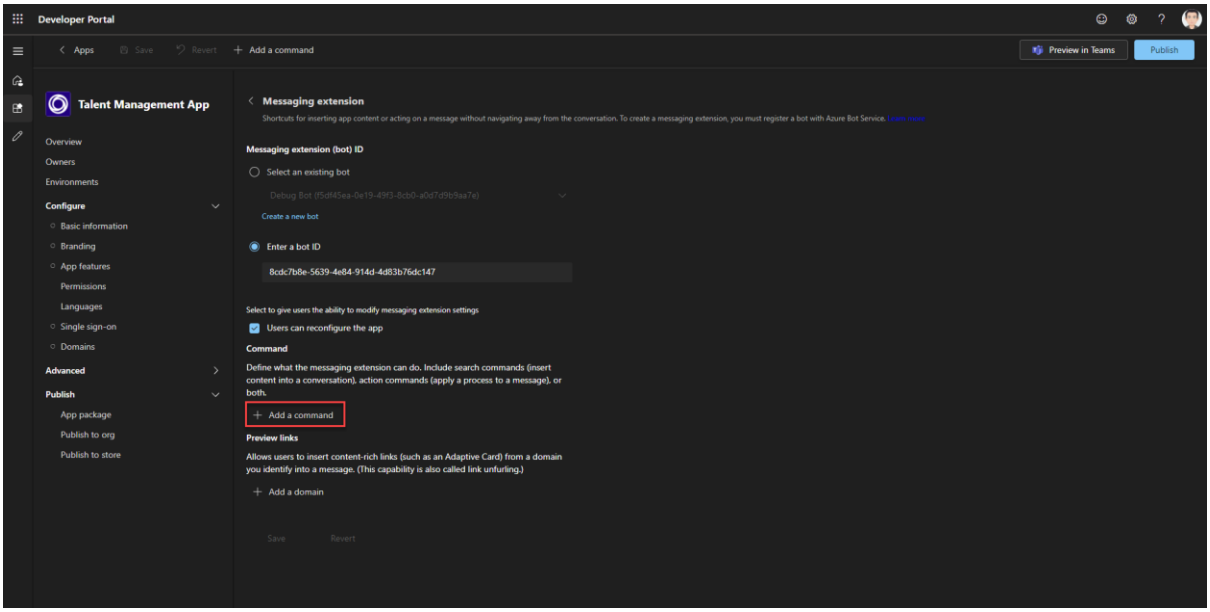
2. Now we can configure the Messaging extension configuration. Select **App Features** from the left menu and click **Messaging extensions**.



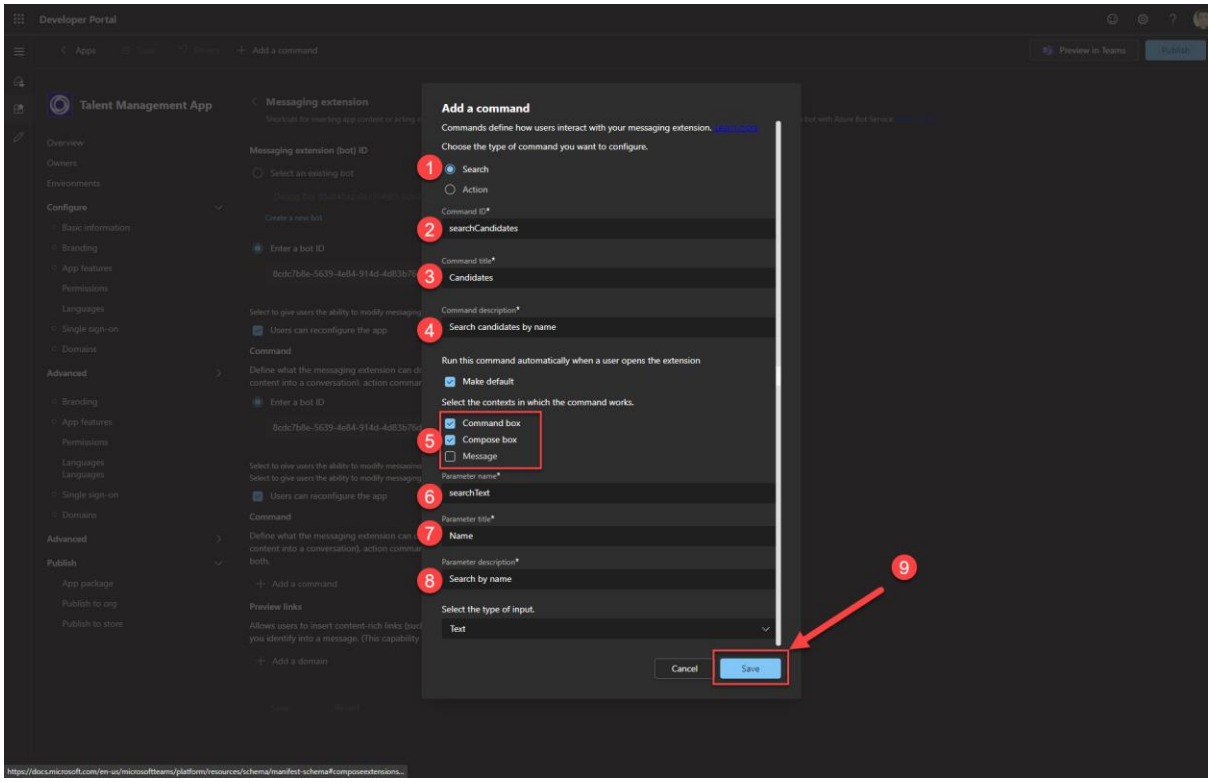
3. Select **Enter a bot id** and enter your App Registration *Client Id* into the textbox. (This is so that you can have a different bot handling your messaging extensions if you wish.) Once this is done, click **Save**.



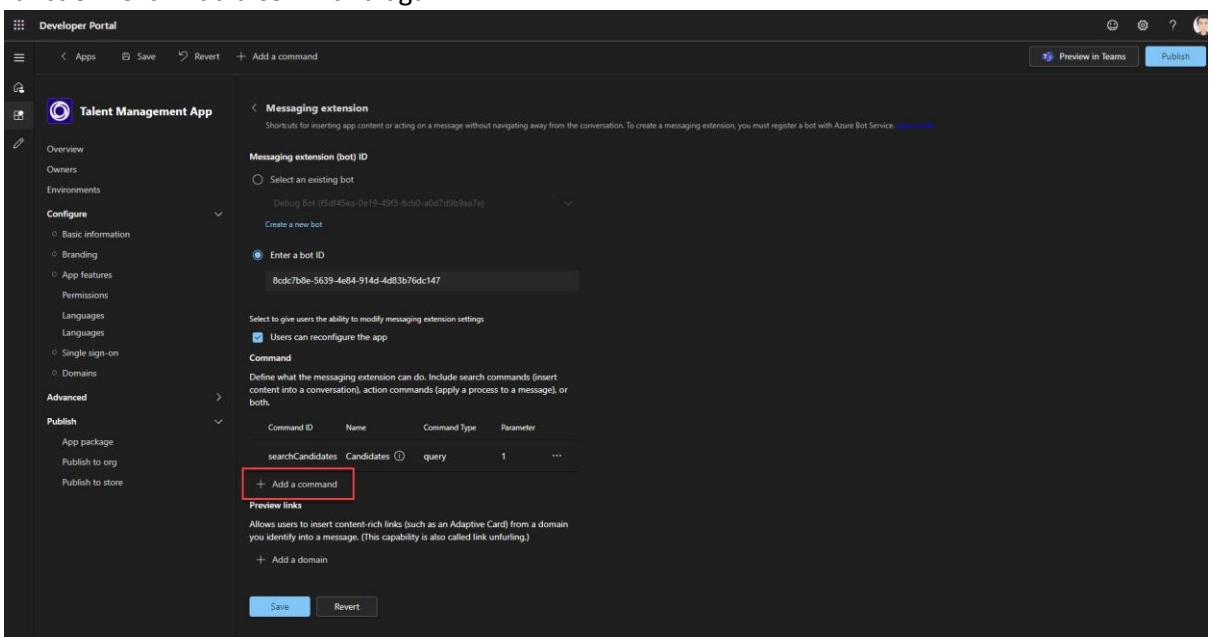
4. Now we need to add the commands. Select **Add a command**.



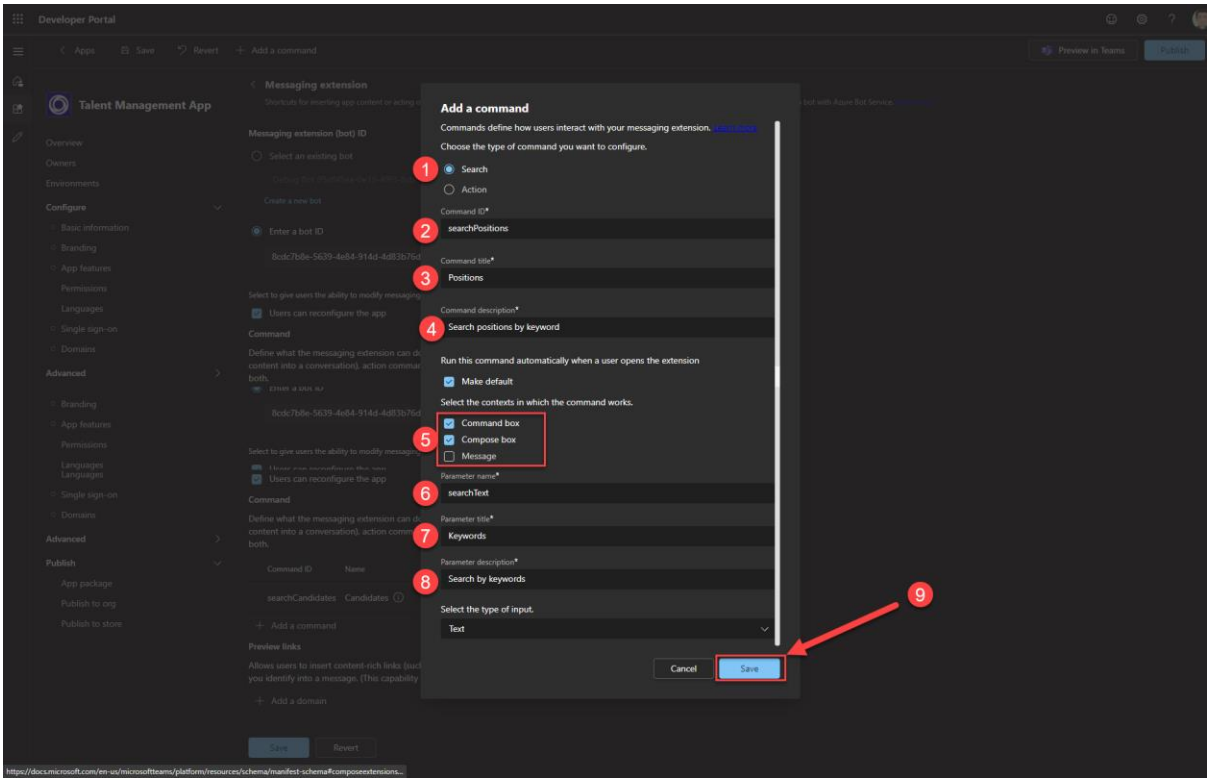
5. Enter the values below to add the search candidates search command and click **Save**.
Note: we are selecting Command box and Compose box as the areas within Teams that this function will be available. This is the large search box at the top of the Teams interface (Command box) and the area you type a message respectively.



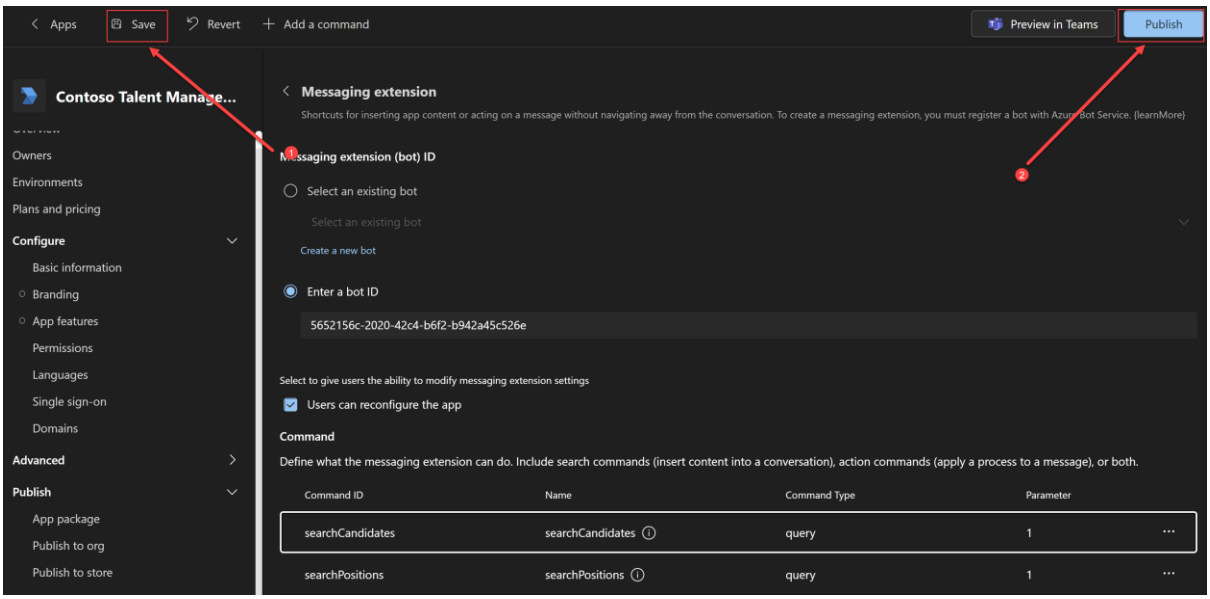
6. Once done, we will add a second search command, this one is for the position search function. Click **Add a command** again.



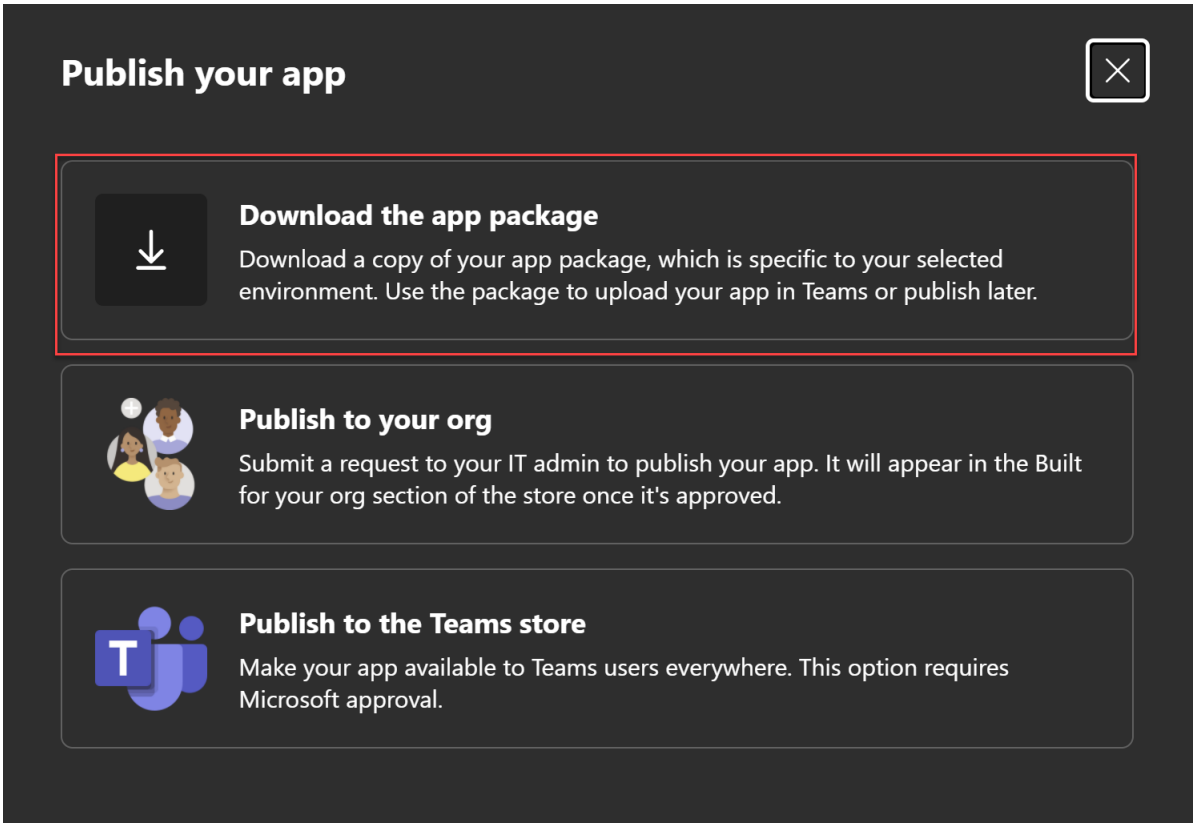
7. Enter the values below and click **Save**.



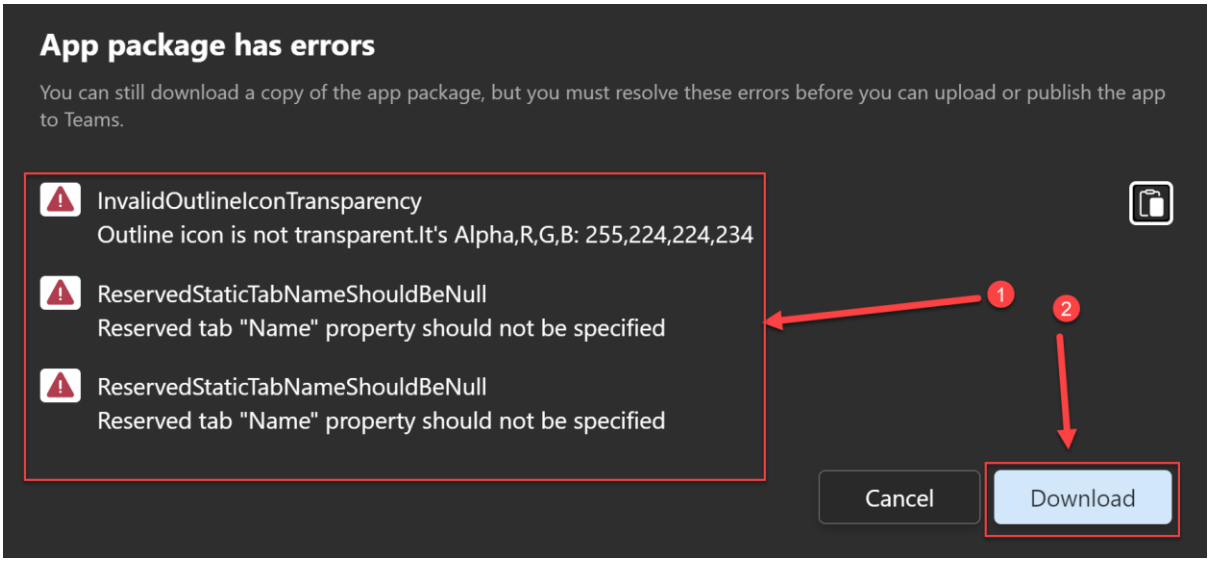
8. Now you can see the two commands have been added, click **Save** and then **Publish** to test it out.



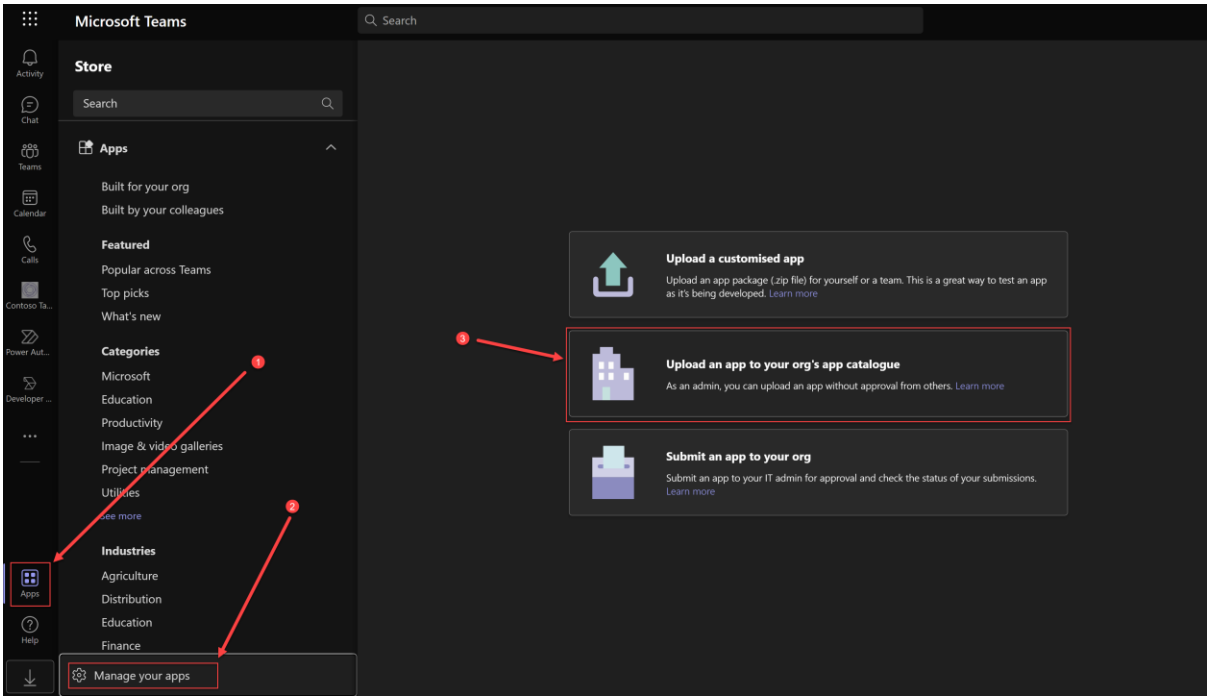
9. Click **Download the app package**, this will save the app manifest ZIP into your Downloads folder.



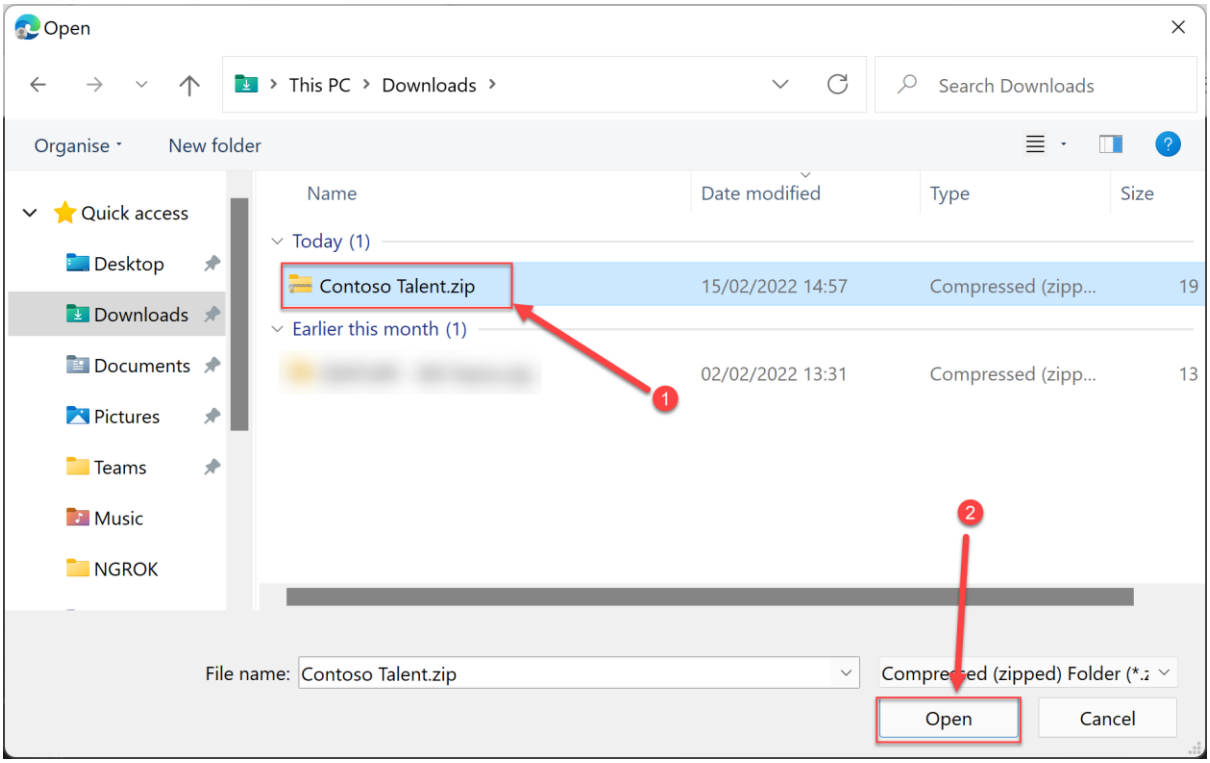
10. If the App package has errors, they will display here. Some errors must be fixed before you can install the app into Teams (note, in the screenshot below, all 3 errors can be safely ignored), **review and fix the errors**, click **Download**.



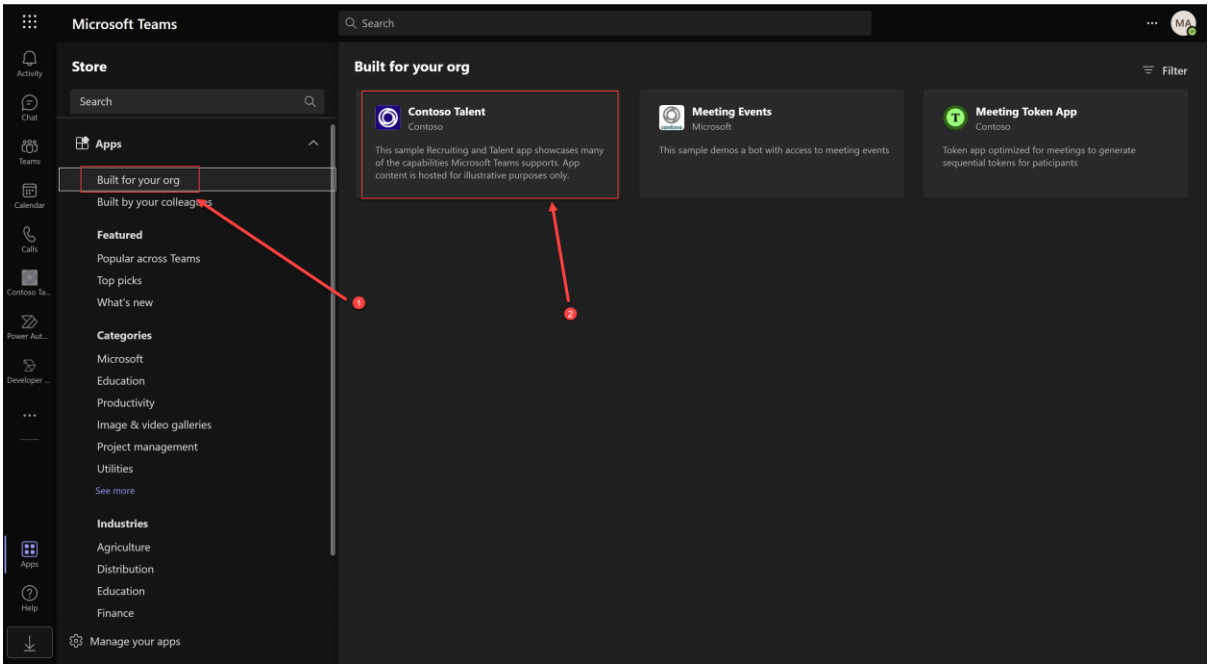
11. Go back into Teams, select **Apps**, then click **Manage your apps**, click **Upload an app to your org's app catalogue**. ***Note: this option will only exist if you are logged into Teams as a Global Administrator, if you are not, log-out and log-in as a Global Administrator.***



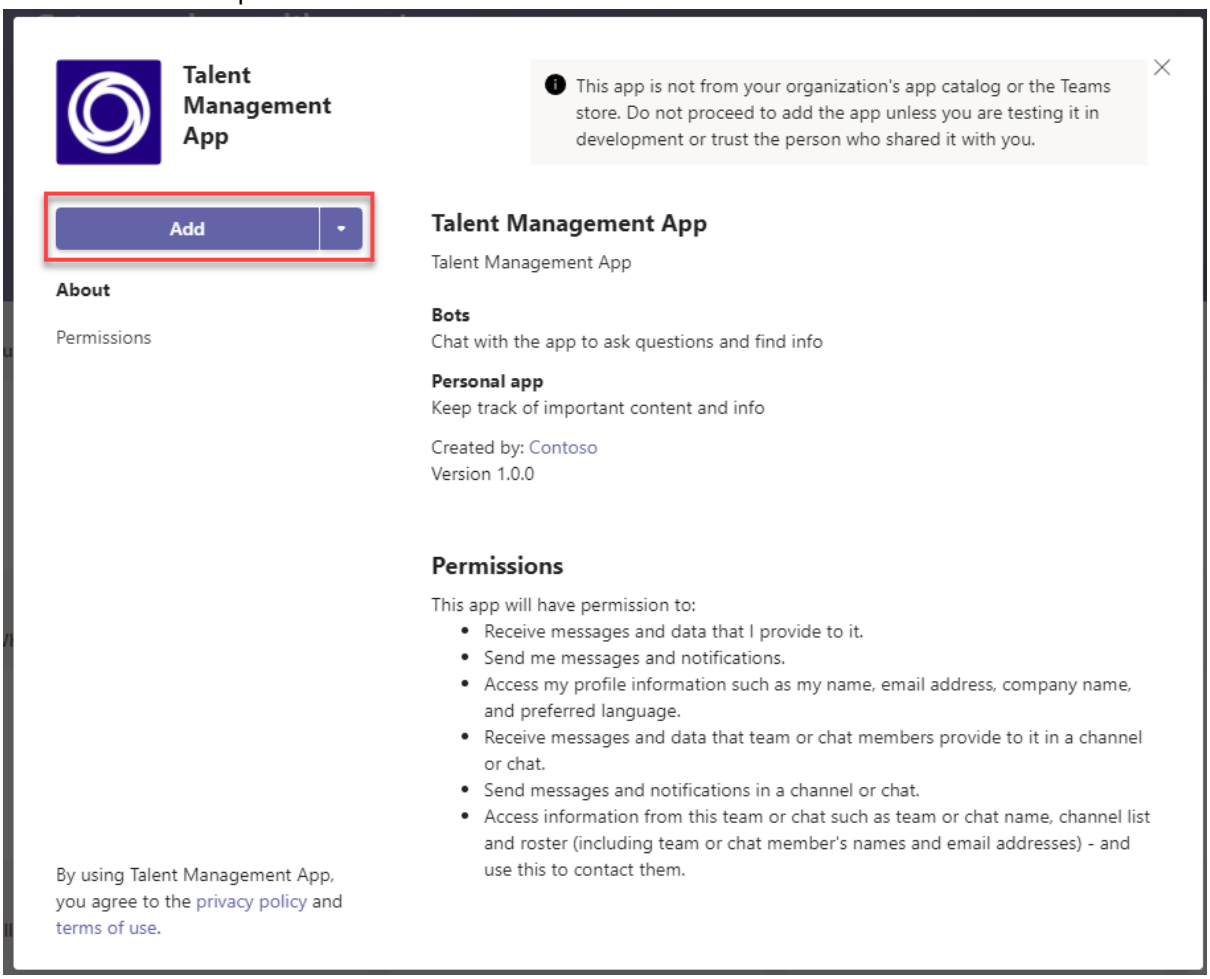
12. Navigate to the directory (folder) where the Teams App Manifest was saved (probably Downloads). **Select the App Manifest ZIP**, and click **Open**.



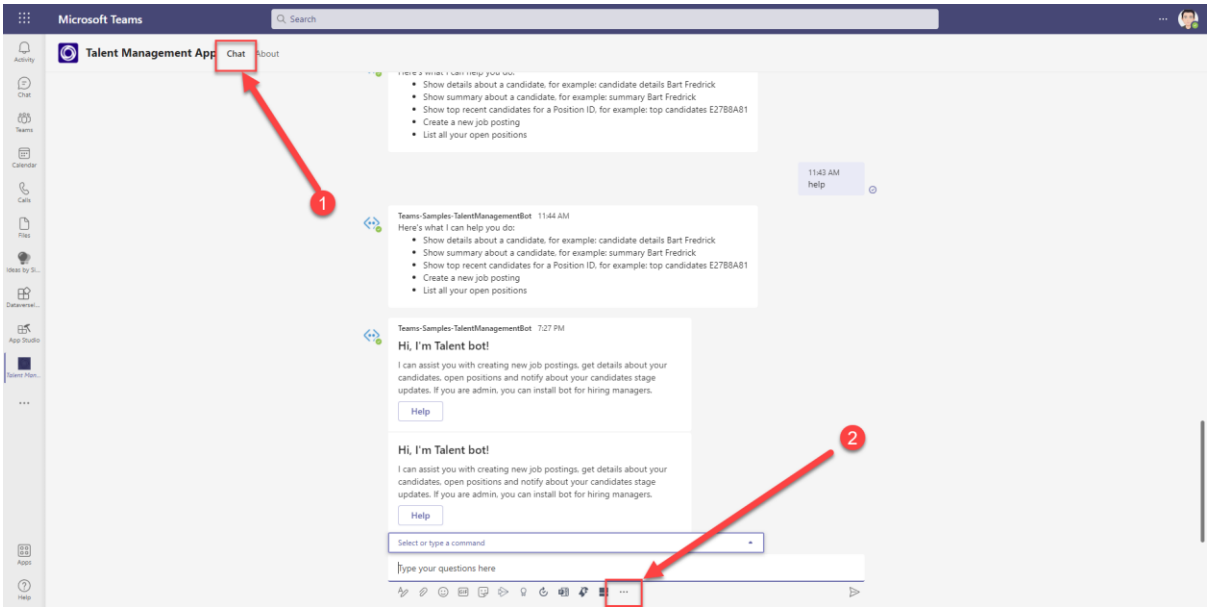
13. Click **Built for your org**, and select the **Contoso Talent** Teams app.



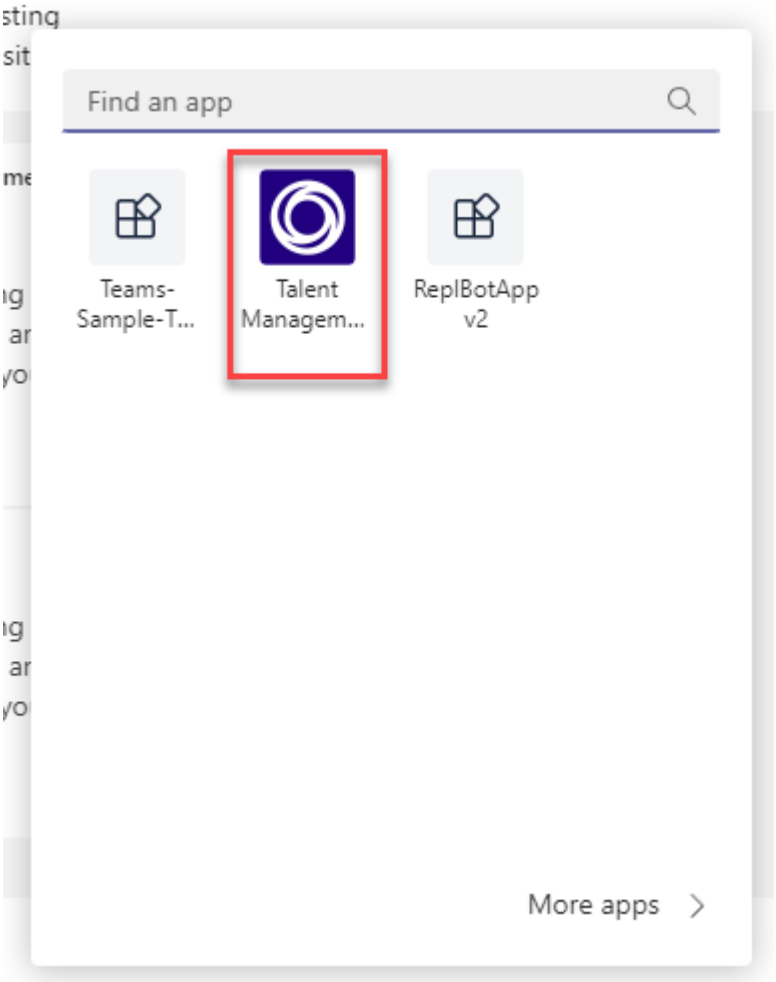
14. Click **Add** to install the latest version of this application, which includes the Messaging Extension search capabilities.



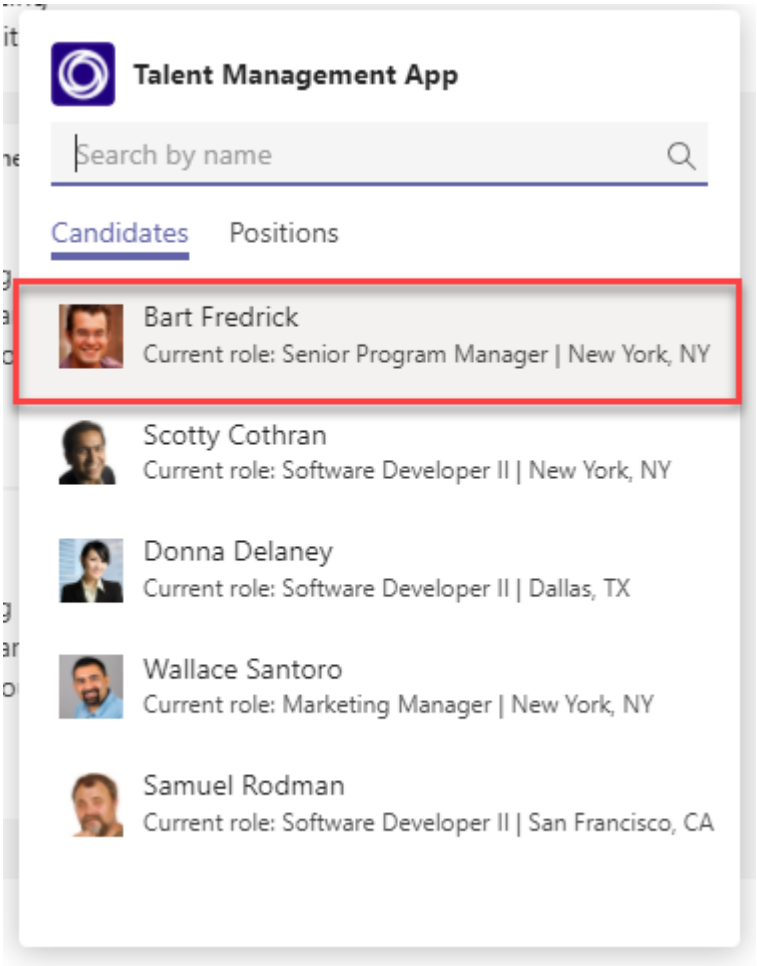
15. Ensure that you are in the Chat tab for your application. (If you have completed this document in order you will see other tabs here, but **Chat** is where we interact with our bot). Then click the ellipsis (…) next to the icons below the compose box.



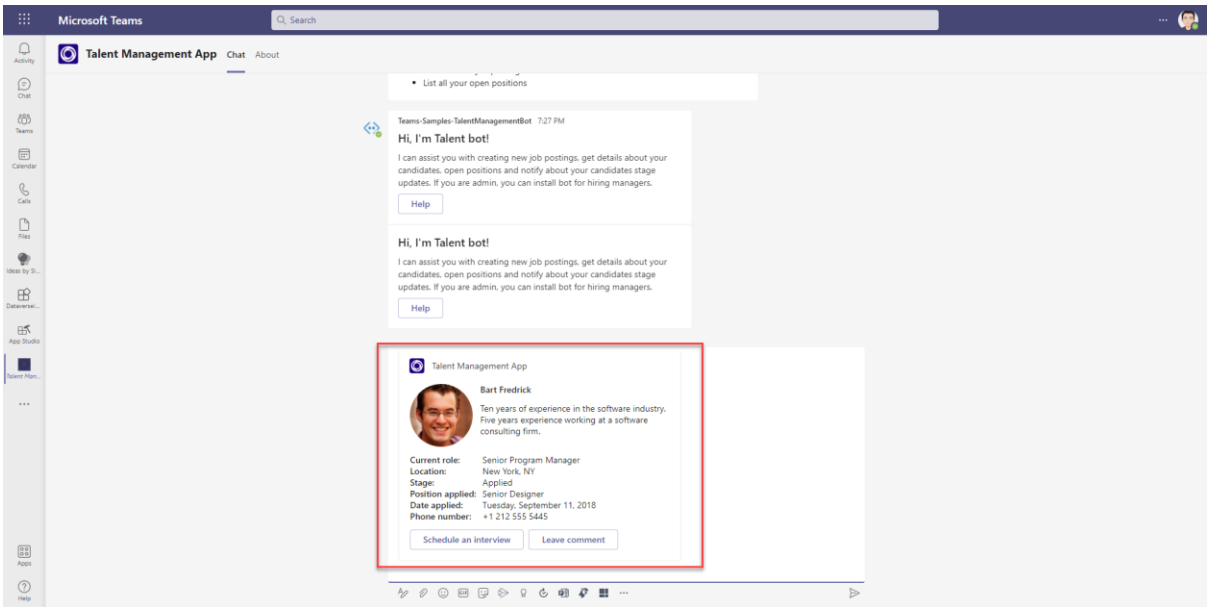
16. Within the dialog that's displayed you should see your application listed, if not you can use the **Find an app** textbox to search for it.



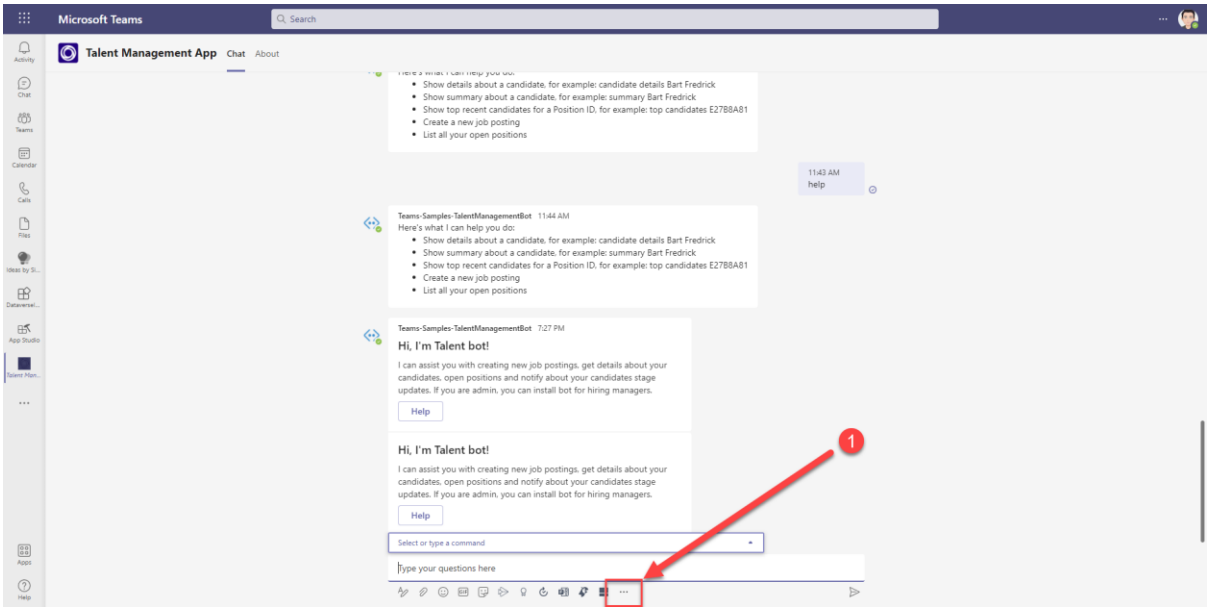
17. Selecting your application will display the following dialog. The things to note here are that there are two search functions **Candidates** and **Positions** shown as tabs below the **Search by name** textbox. Select one of the candidates.



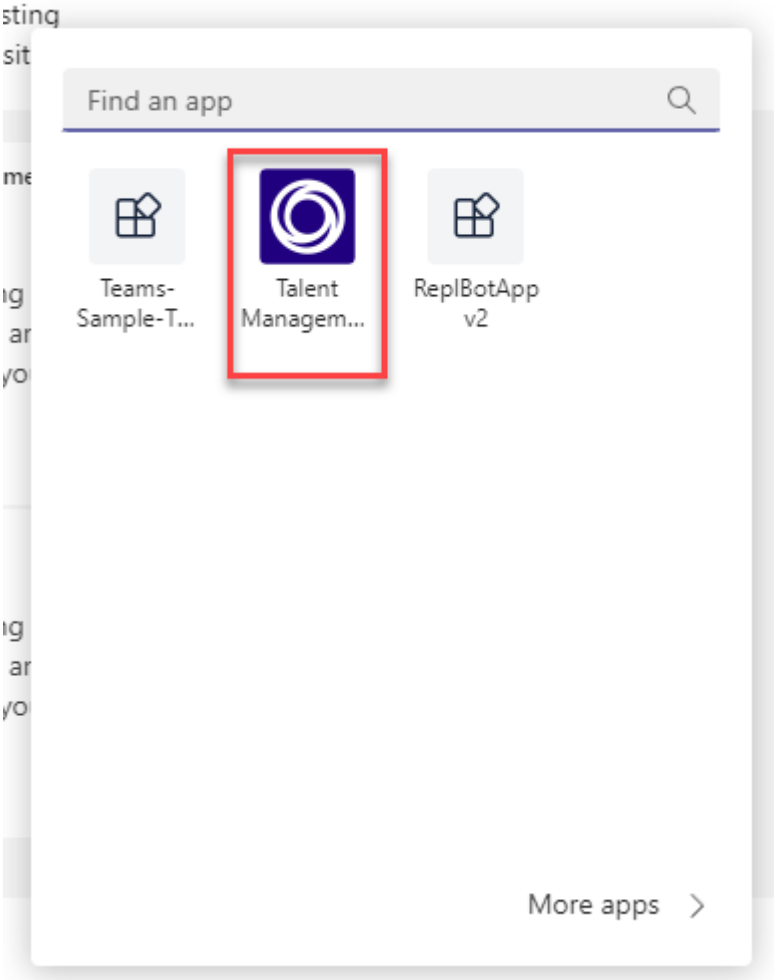
18. This will place the adaptive card we saw earlier in the compose box. It's important to note that message extensions don't post directly to the chat, instead they add their content to the compose box and allow you to include more text before choosing to send the content.



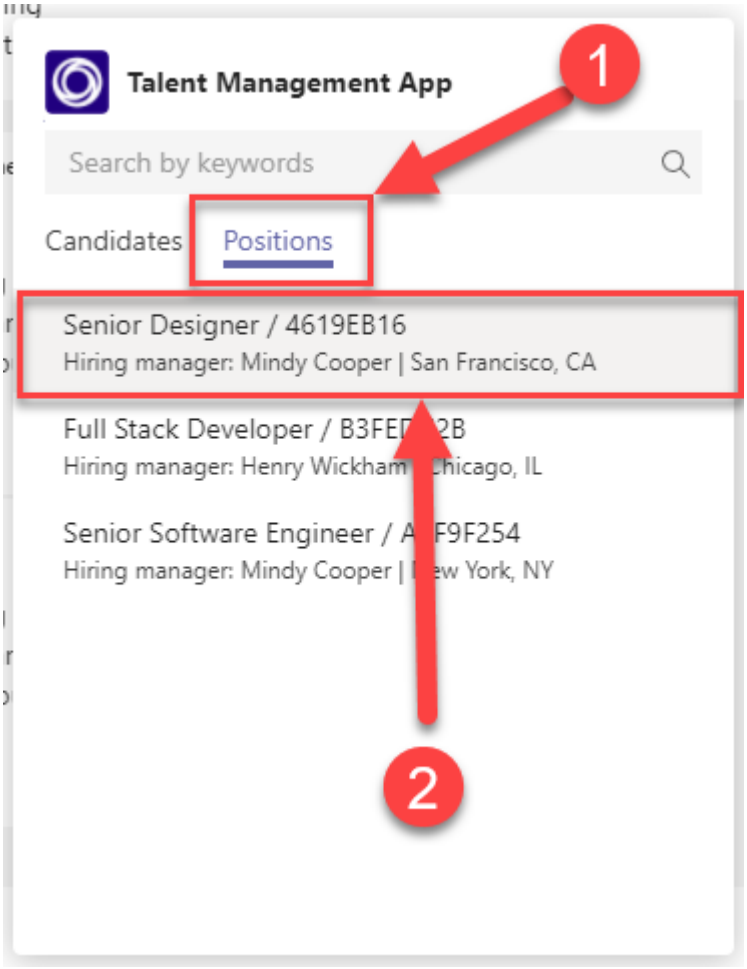
19. Repeat this process to see the positions adaptive card. Click the ellipsis again (…).



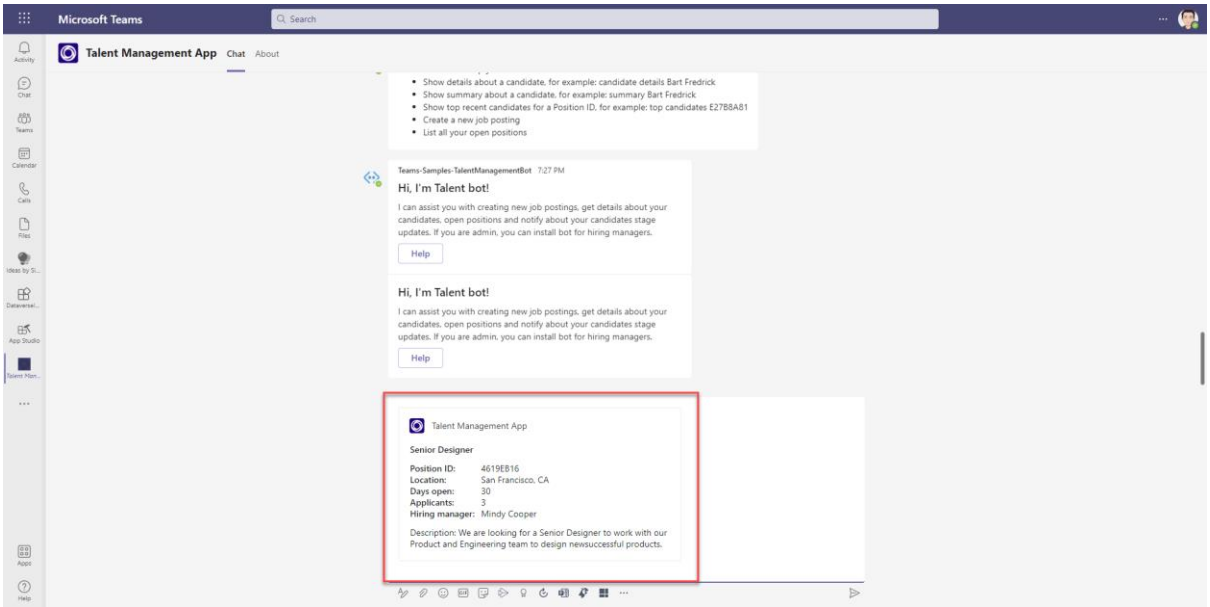
20. Find your application



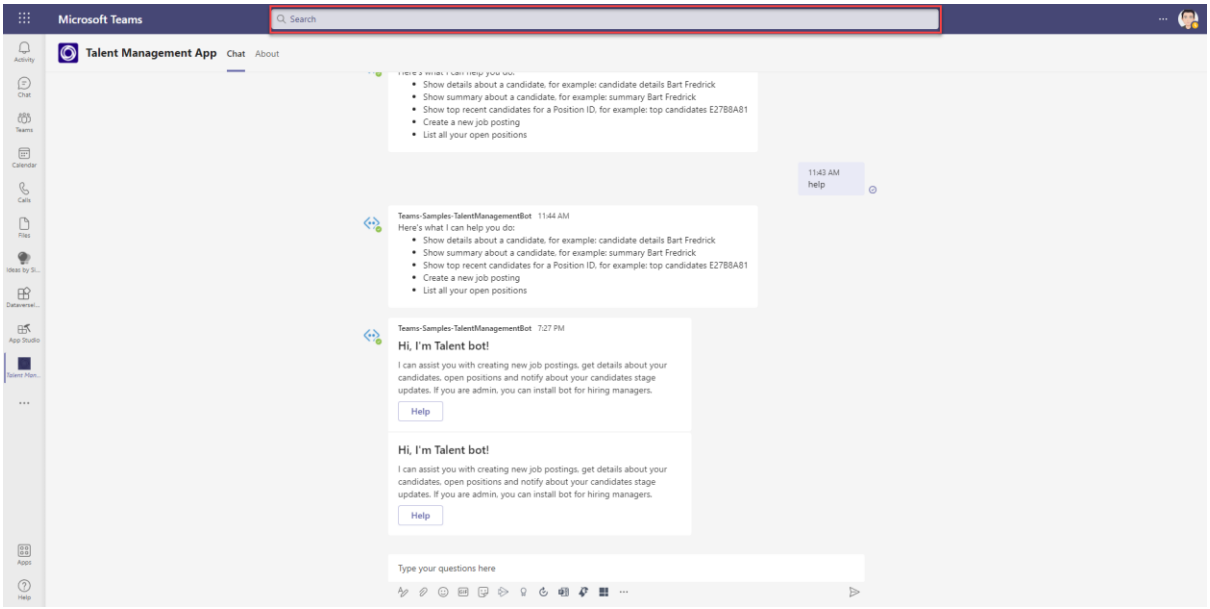
21. This time, select **Positions**. You will see a list of positions displayed. Now select one.



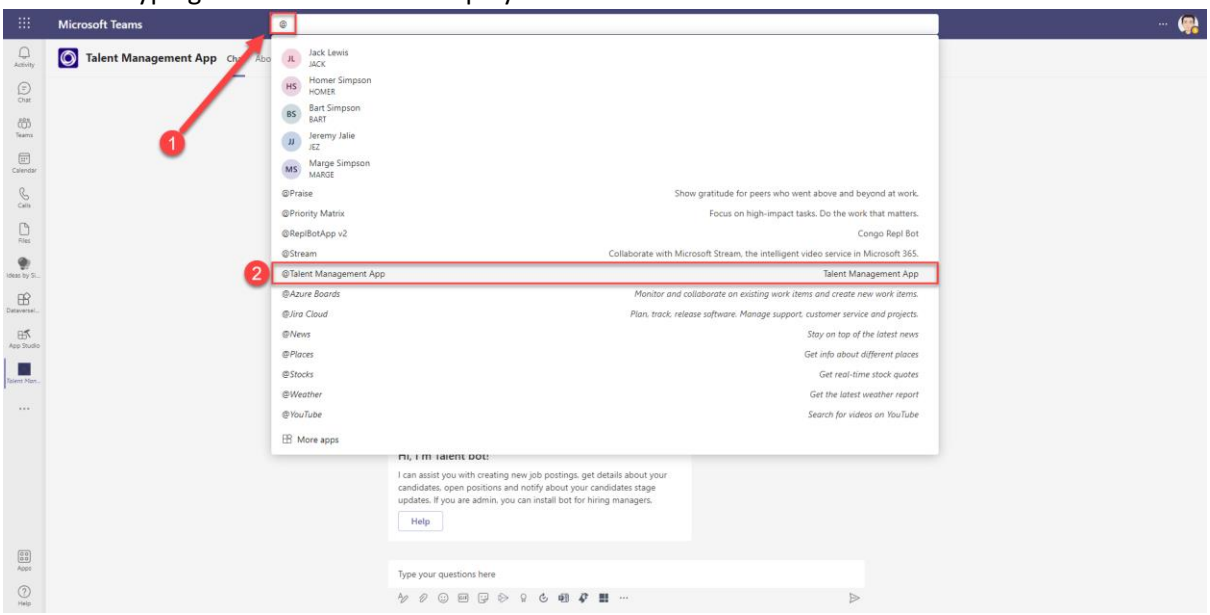
22. As before you will see that an adaptive card has been added to the compose box ready to be sent!



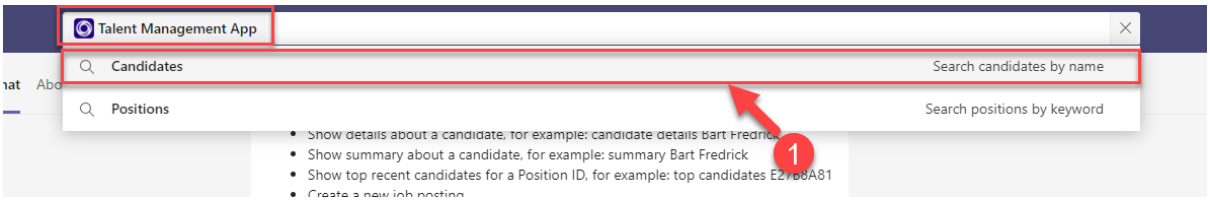
23. Another feature of a search command is that it can be made available to the global search bar (Command box) at the top of the Teams interface.



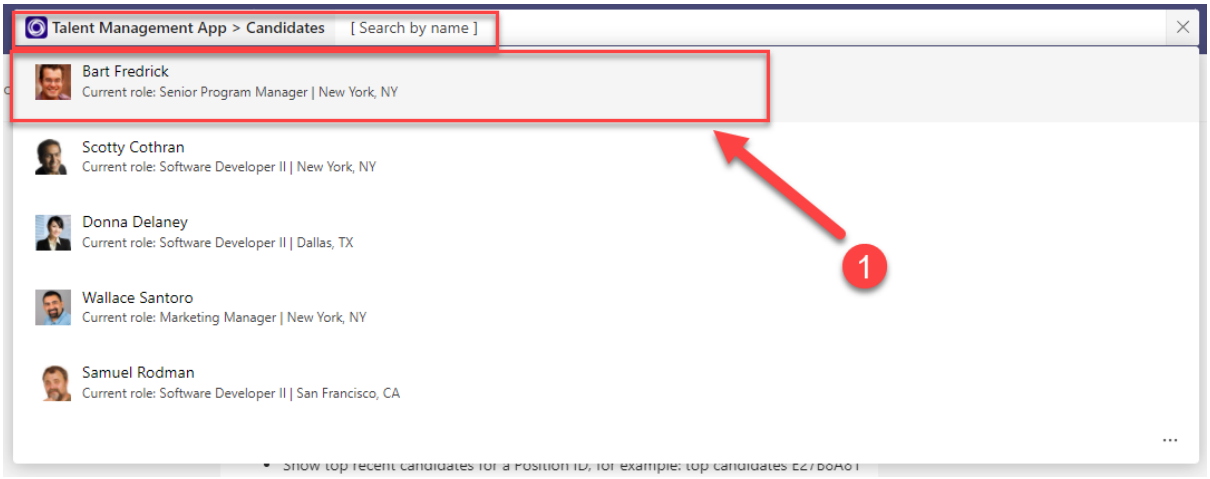
24. Typing an @ symbol will show a dropdown with a list of installed bots. If you can't see yours then start typing the name until it's displayed and then select it from the list.



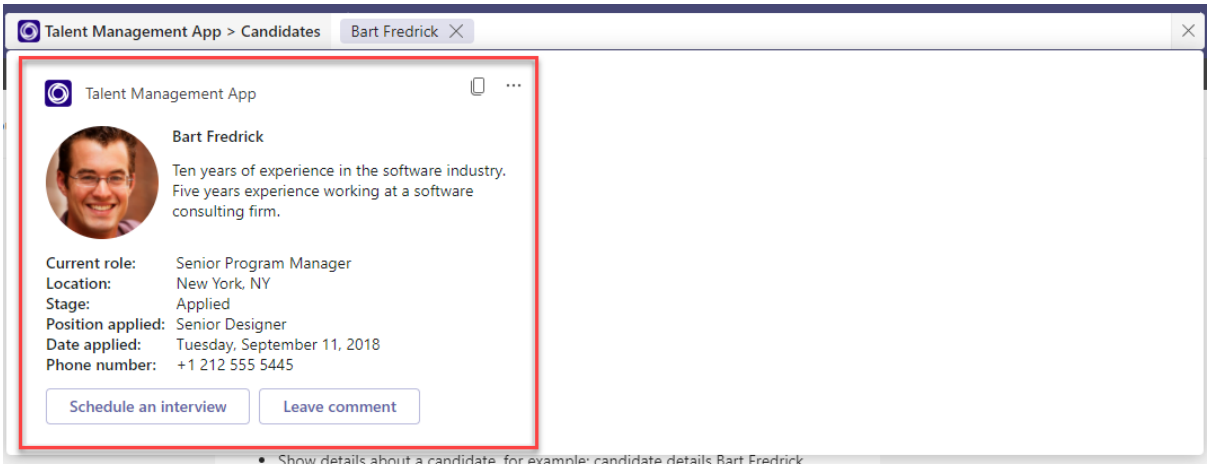
25. This will change the context of the search box to your bot only and once in that context you will be prompted with all the available search functionality exposed by our bot. In this case **Candidates** and **Positions**. Select candidates to see a list of candidates.



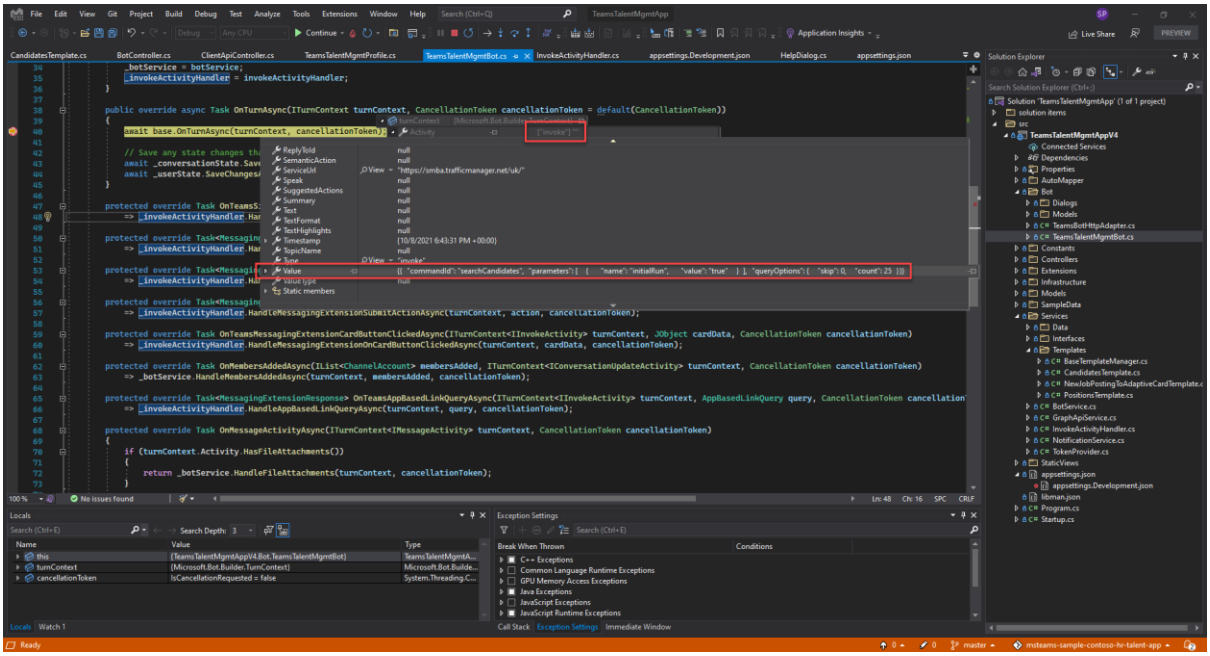
26. The search box has changed context again as is now only working within the Candidates search function. You can search for a candidate by name, or select one from the list.



27. Unlike the compose message extension, the adaptive card is displayed directly in the search box dropdown, but works like any other adaptive card.



28. To give you an idea of how this works, let's open Visual Studio and inspect some variables! Open the TeamsTalentMgmtBot.cs file and set a breakpoint on line 40. Once you invoke the search by either using the compose message extension or the command box your breakpoint should be hit and you can inspect the values that make up the search command.

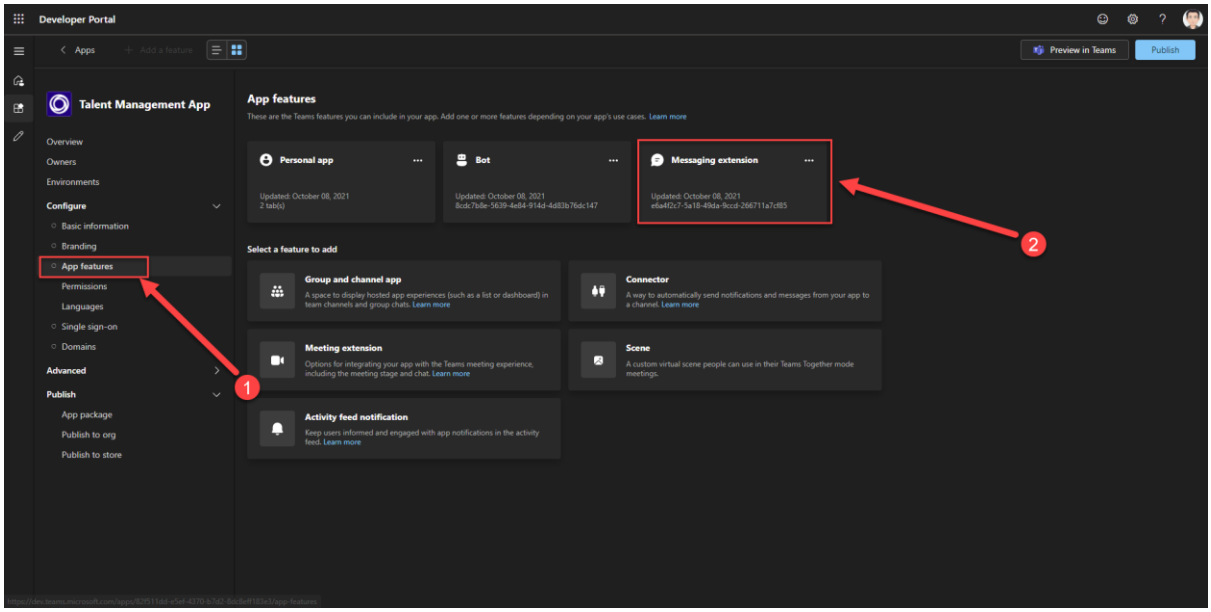


Now that we have implemented the Messaging Extension search feature, we will now move onto a Messaging Extension Action feature!

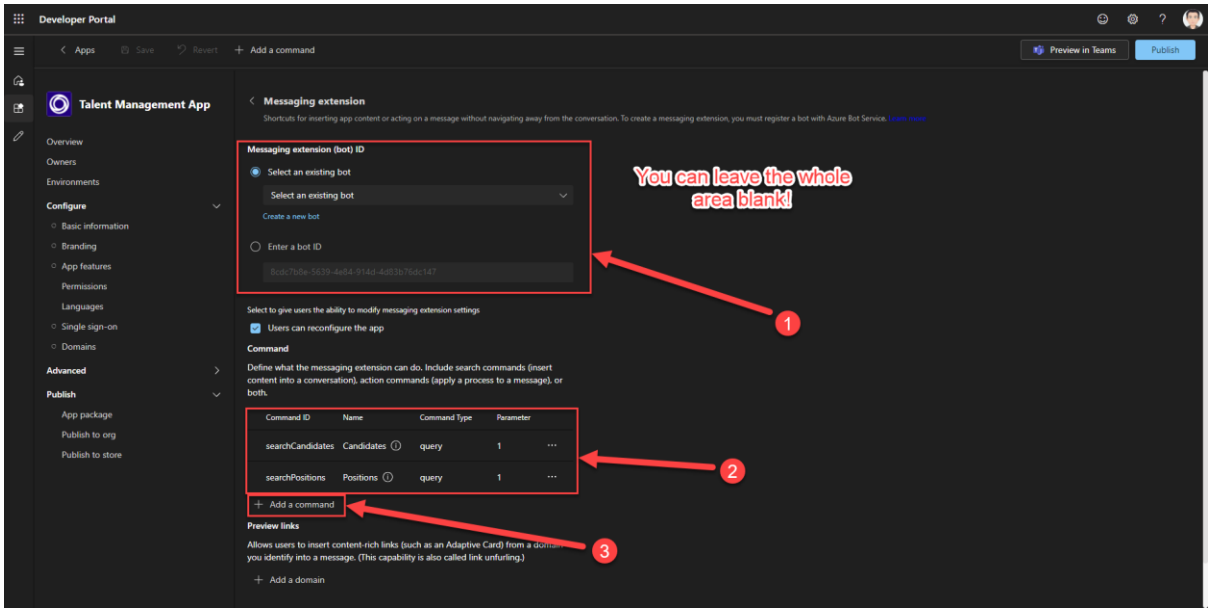
6) Implement a Message Extension Action

In this step we will add another message extension to perform an action instead of a search, this will allow us to create information (positions) in a third-party app, rather than just searching for objects/information that already exists.

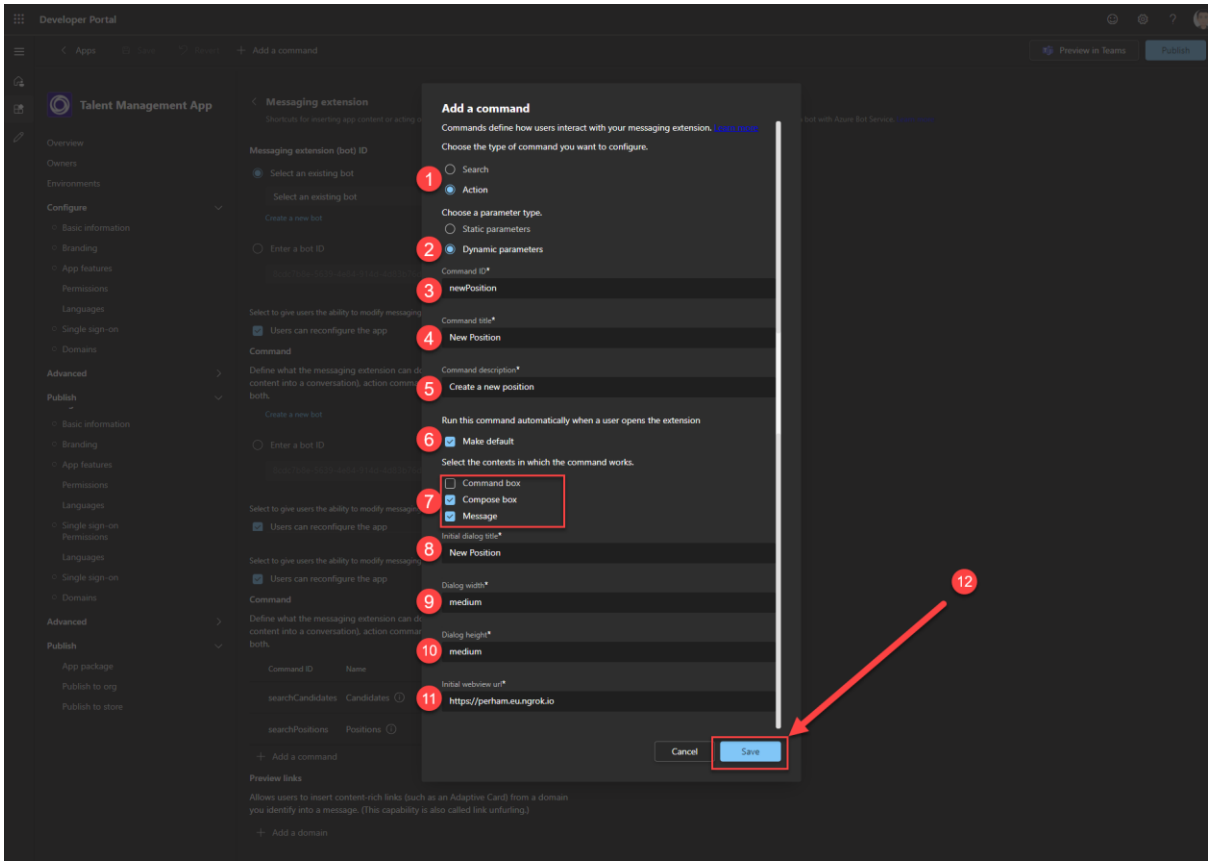
1. Open the Developer Portal, navigate to App Features and select Messaging extensions to edit the manifest.



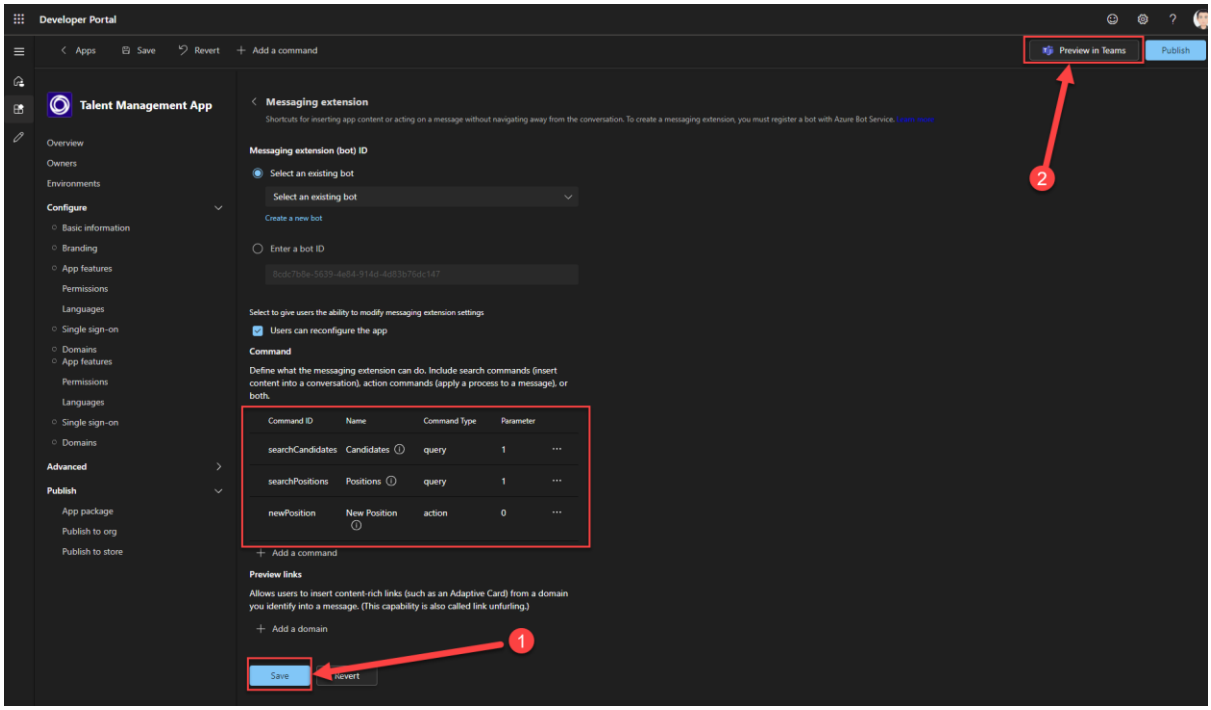
2. Here we will add a new command, if you can see the search commands listed then you are already in the context of your bot and don't need to select anything from the top section. If not you can enter your bot id as before. Then click **Add a command**.



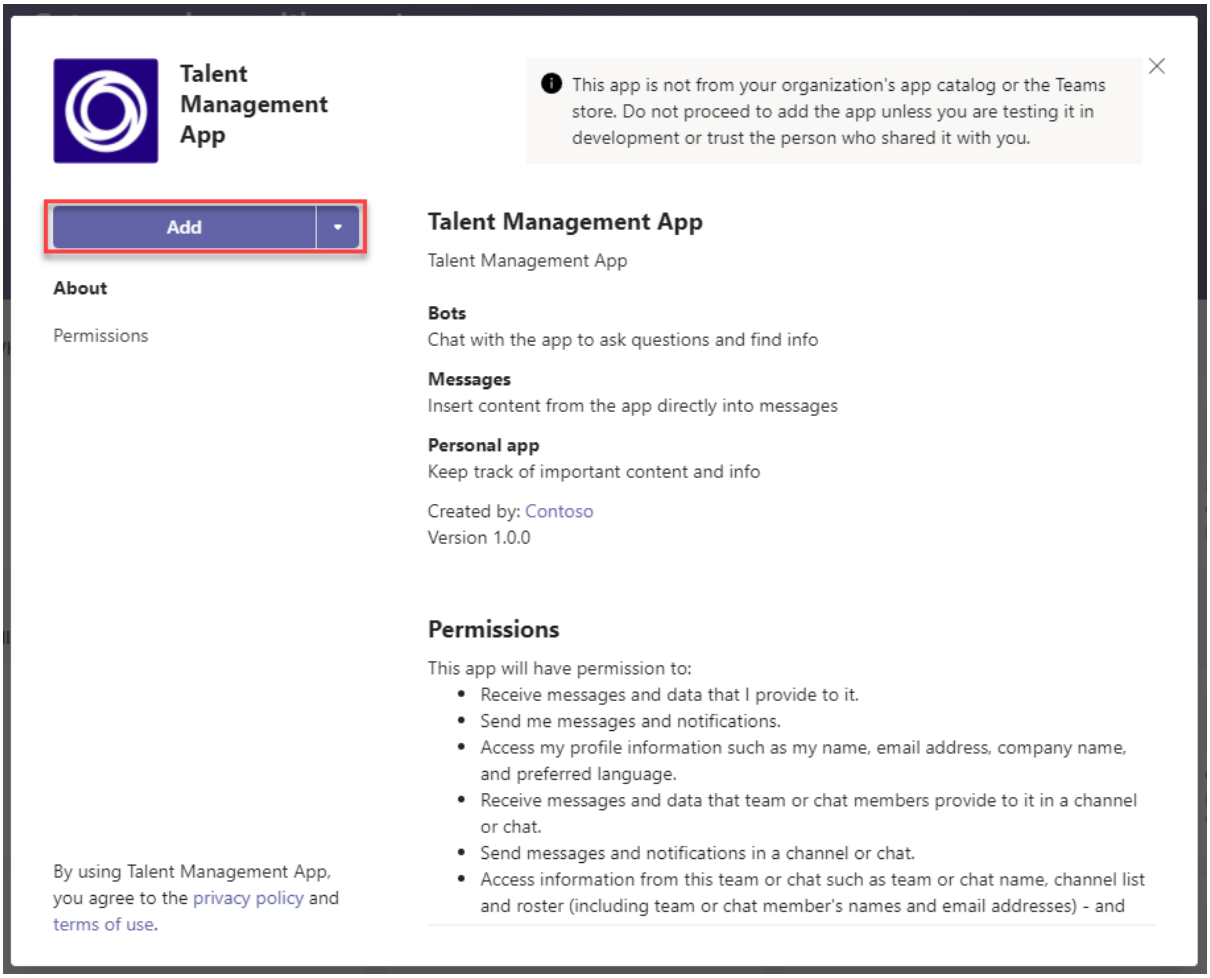
3. Complete the dialog as shown below, then click **Save**.



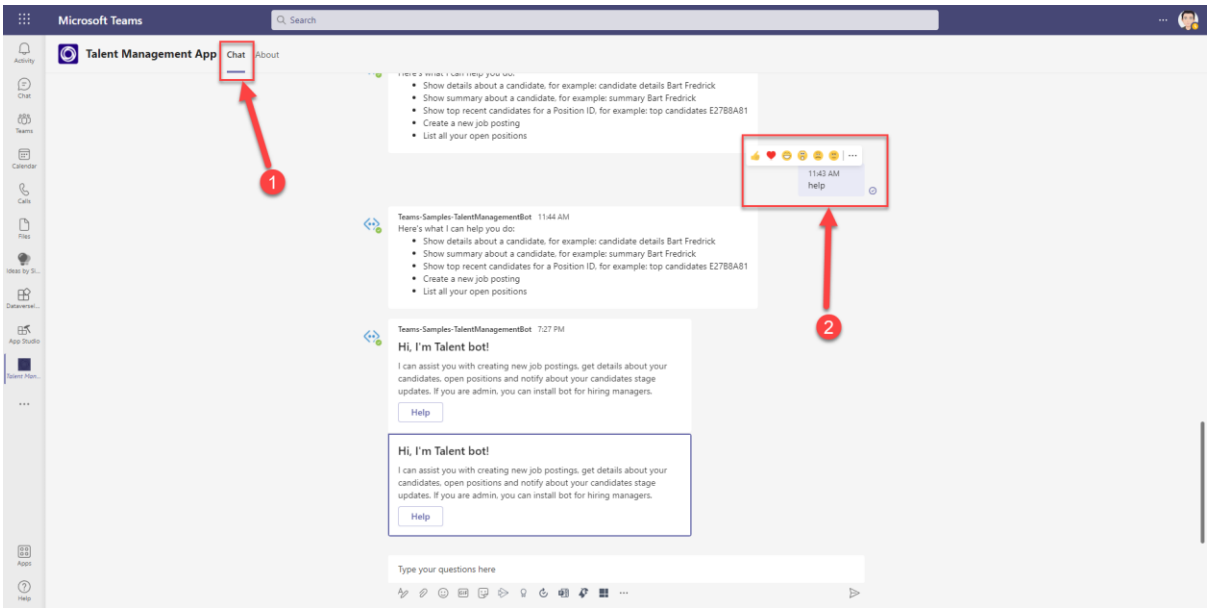
4. Ensure that you save the manifest by clicking **Save** and then we can test it. Click **Preview in Teams**.



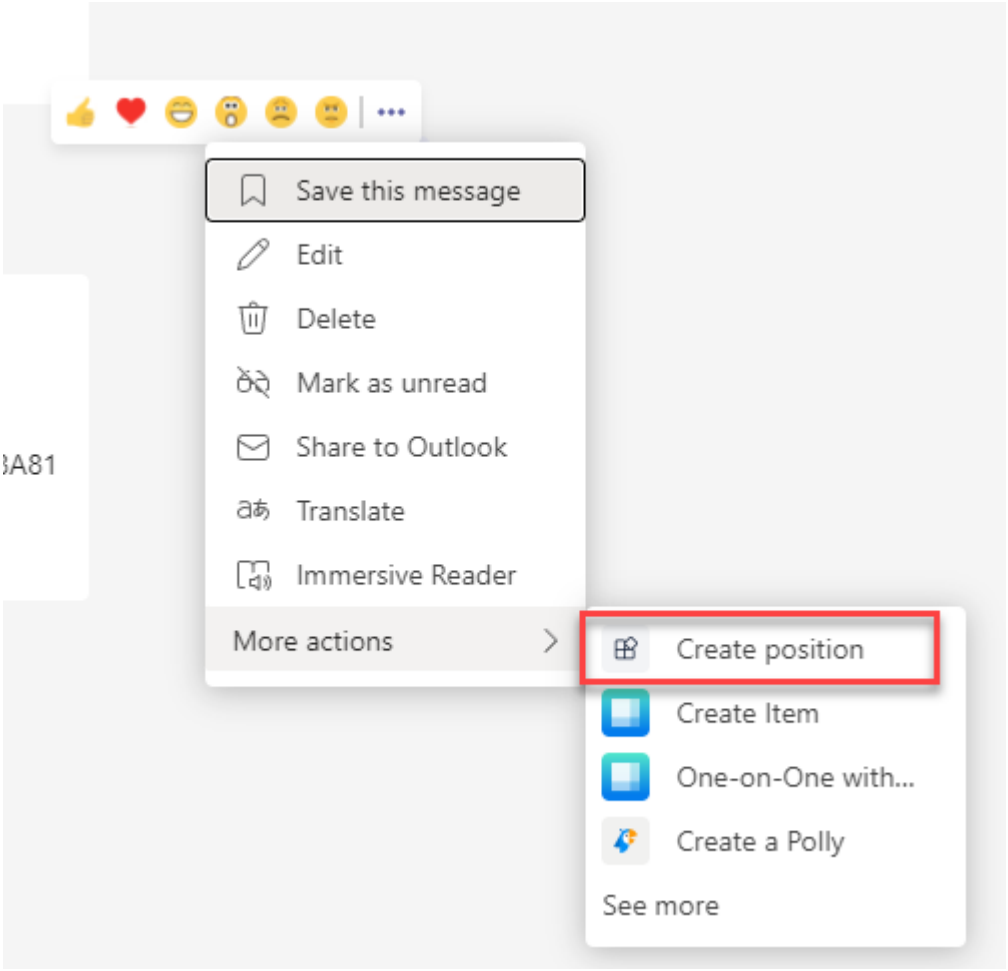
5. Again we will be prompted to add the app to Teams.



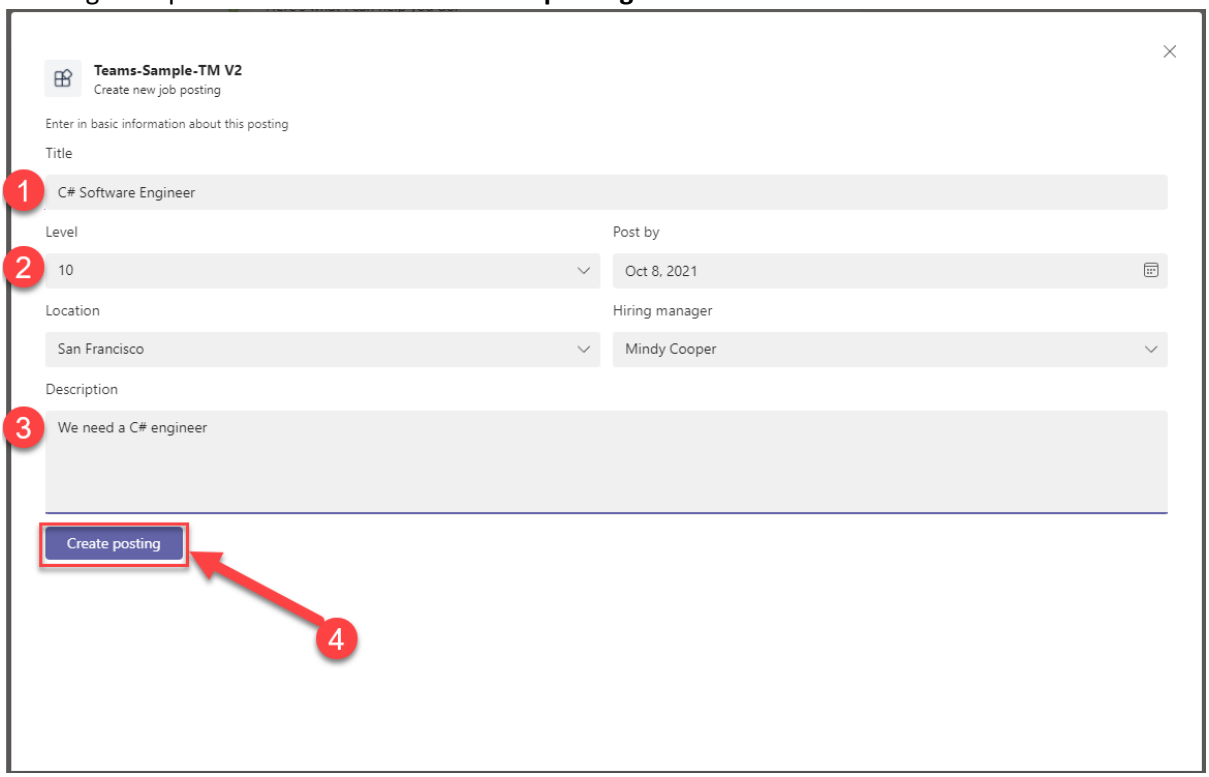
6. Again ensure that you are in the Chat tab of your application. We configured the action to work on messages so we can invoke it by hovering over an existing message, and clicking the ellipsis (…).



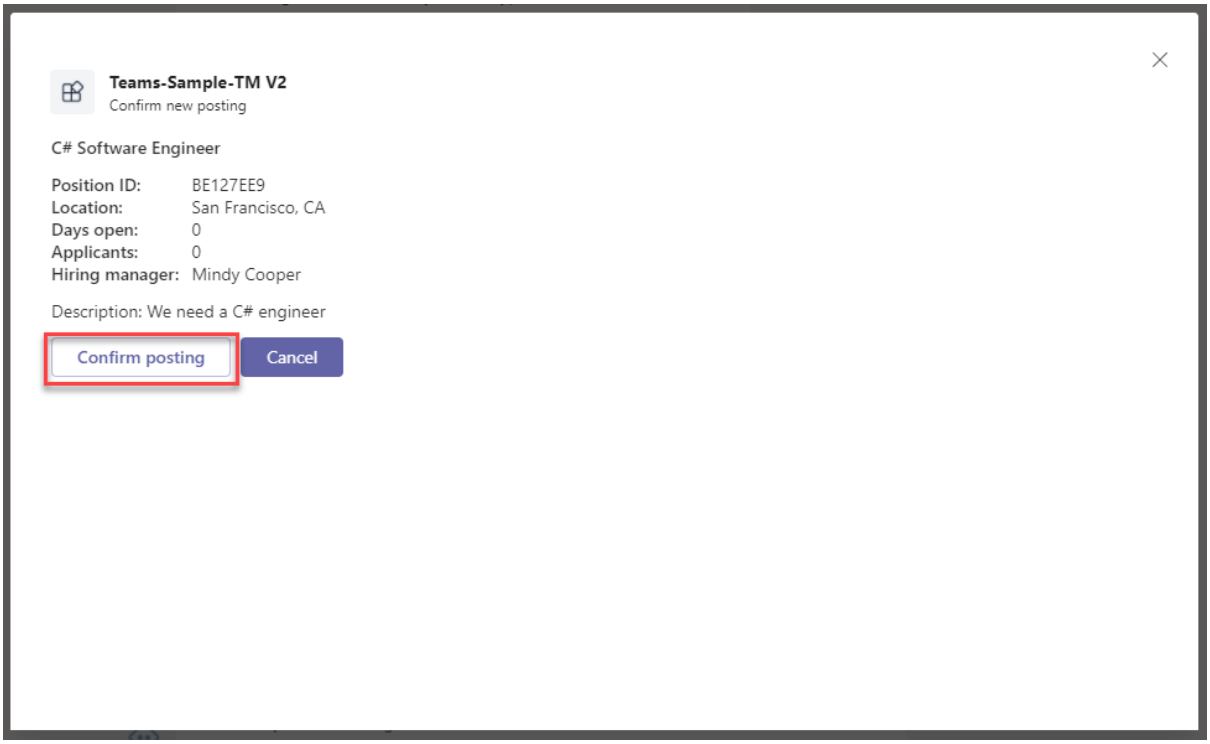
7. From the menu that's displayed, click **More actions** and then **Create position**.



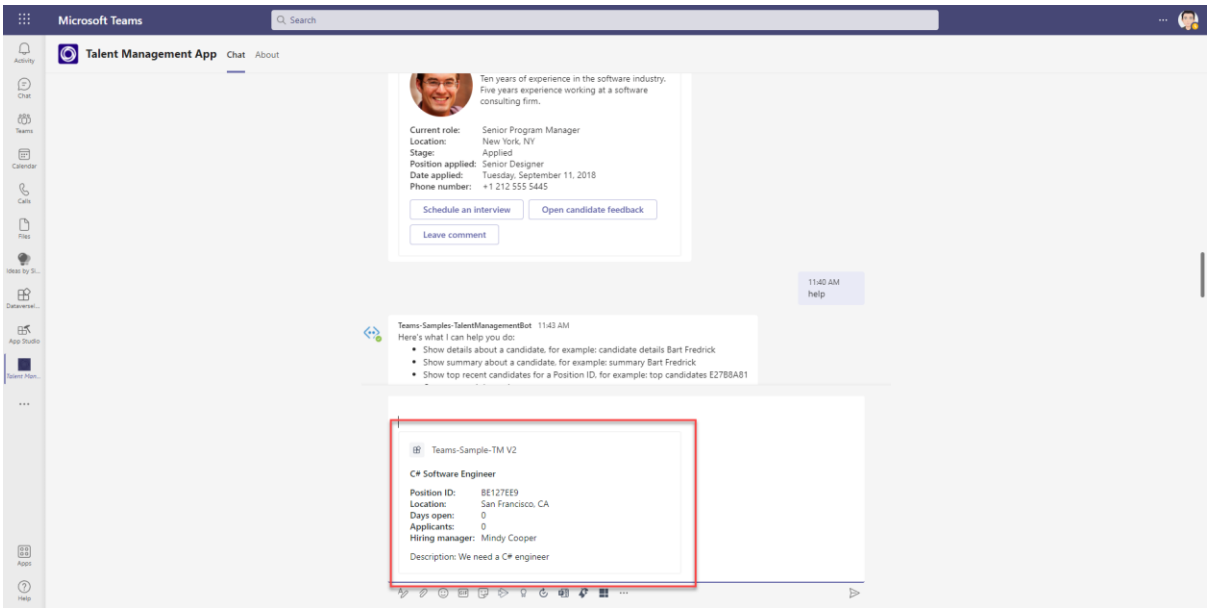
8. This will display a task module prompting for information pertaining to the position you are creating. Complete the form and click **Create posting**.



9. You will then be asked to confirm the details before clicking **Confirm posting**.



10. At this point the position has been saved and will be available through the tab or search functions shown earlier. It will also put an adaptive card in the compose box to allow you to send it to the chat if you wish!



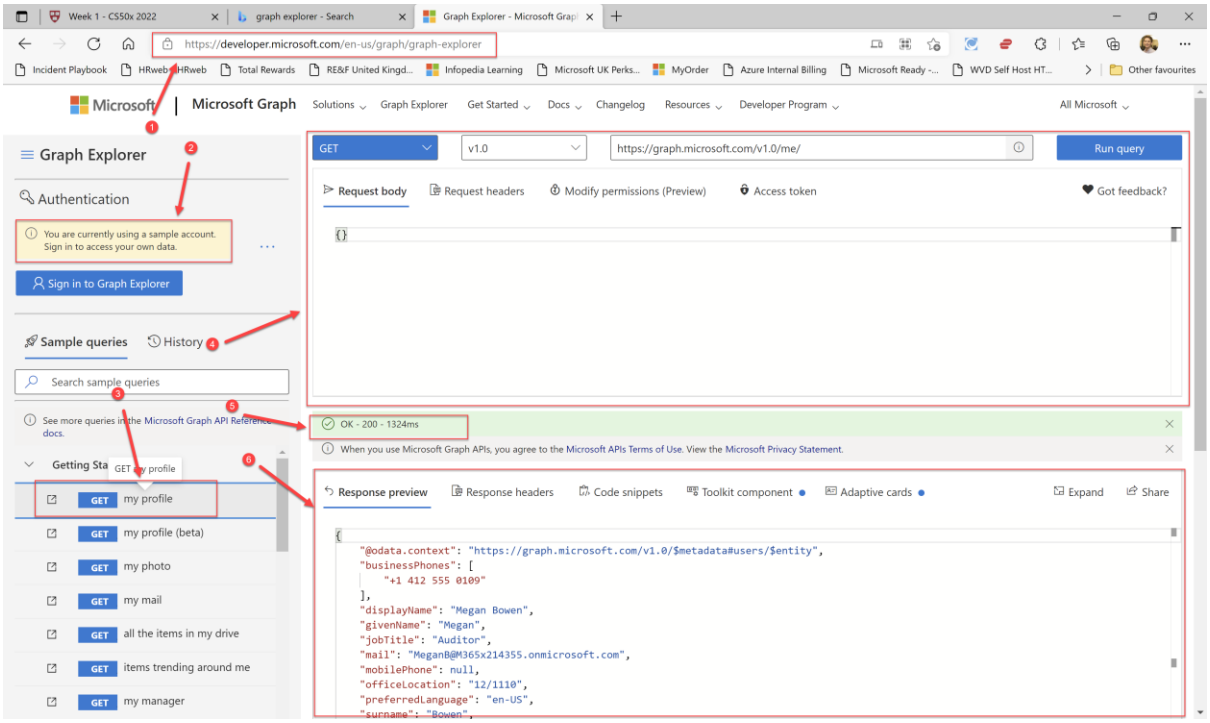
Now that the Messaging Action is working, we will now spend some time working with the Microsoft Graph API, which will allow us to perform CRUD (create, read, update & delete) operations, via a REST API, to interact with Microsoft 365 resources.

7) Working with Microsoft Graph API

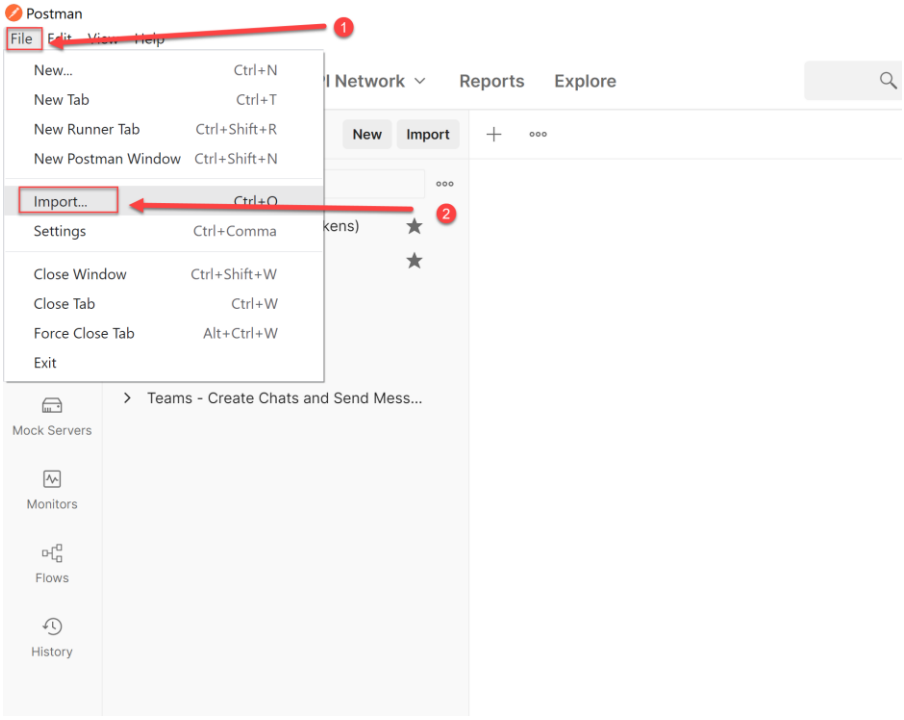
We will now move away from working with the Talent Management app for this section, to learn about the Microsoft Graph API, and how it allows you to work with Microsoft 365 data. We will begin by using the Microsoft Graph Explorer tool, a sandbox app that was created by Microsoft to introduce you to working with the Microsoft Graph API. We will then move onto crafting some custom API calls, using Postman, and will clearly define the differences between a Delegated

Access Token and an Application Access Token. Finally, we will review some of the code that is used in the Talent Management App, to proactively install the application you have been working on, as this is required to enable Bot Proactive Messaging in Teams.

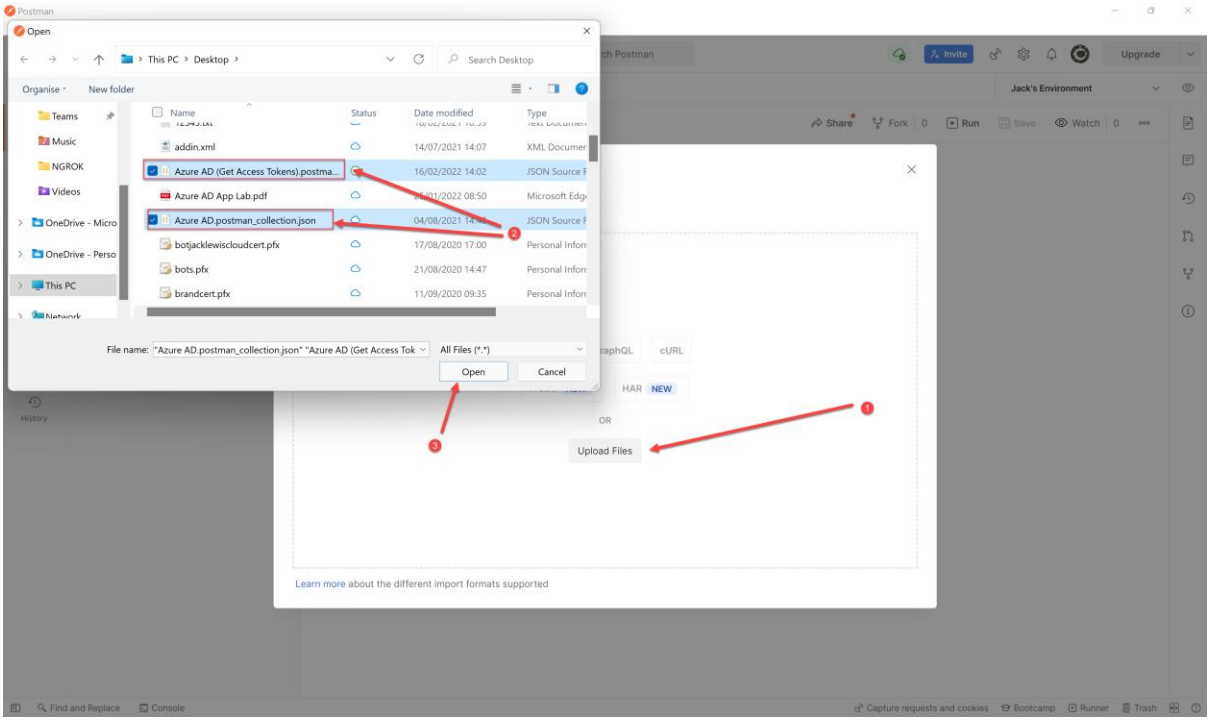
1. Microsoft 365 includes a suite of tools and technologies to enable productivity, this includes Teams, SharePoint, OneDrive, Exchange, Azure AD and many others. The unified API endpoint for working with resources that reside in Microsoft 365 is the Microsoft Graph API. It enables developers to perform CRUD (create, read, update & delete) operations, via a REST API, to interact with Microsoft 365 resources. To introduce you to these concepts, we are going to start with the Microsoft Graph Explorer sandbox. Open a web-browser and go to Graph Explorer - Microsoft Graph
2. Once you see the Graph Explorer, there are a number of features that you need to be made aware of: Firstly, you will see that you are signed in, by default, using a sample account. This is great for getting you working with the Microsoft Graph API quickly, although if you want to do more than just retrieve data, you will need to sign in, although, there is no need for you to do that yet! Looking at number 3 in the below picture, you will see some sample queries, that the Graph Explorer team have made available for you to use out of the box, if you click on the **GET my profile** sample request, you will see section 4 change and make a request, to Microsoft Graph API, to the /me resource, via the v1.0 Endpoint. In this simple first example, you will see that the request body does not contain any data, and if you check the access token, it will be for the sample account. Section 5 contains the response code, in this case it's 200, which is expected and means the request was successful, and section 6 contains the response data. If you review the response, you will see that this details the information of the sample account, allowing you to discover more information about this user, such as their email address and job title. This is an example of a request that was made using a delegated permissions access token, which means it was made in the context, and on-behalf-of the logged in user (which is the sample account). If you Sign in to Graph Explorer, using your test/dev user, and make the same request, you will be asked to consent to the Graph Explorer app being granted specific scopes, and the response will detail information about your test/dev user, as the request was made, using an access token that contains delegated permission scopes. ***Note: it is not possible to make a Graph API call, on behalf of an app, using Graph Explorer, only delegated permission scopes and access tokens on-behalf-of users can be used. We will make an API call to Microsoft Graph on behalf of an app, using Postman, in the next few steps, to allow you to explore the differences between app and delegated permissions.***



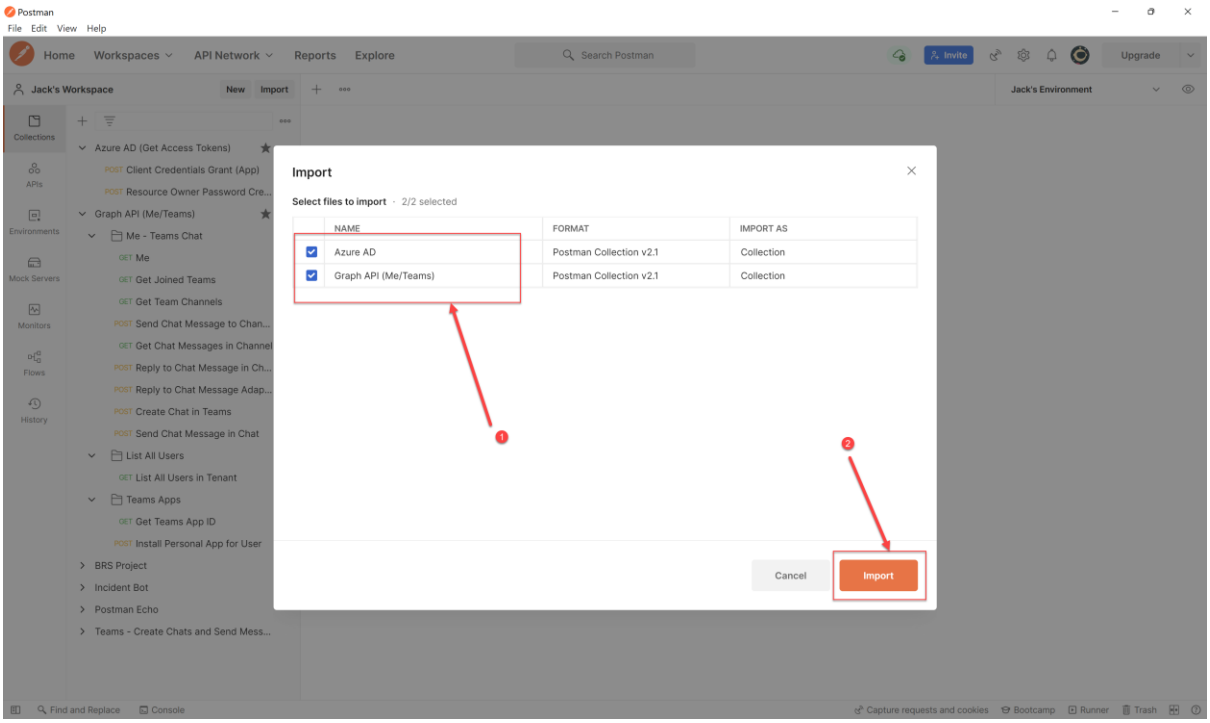
3. Continue to spend time with the Graph Explorer, using the sample queries, and when you feel comfortable with how the request/response model works for Microsoft Graph API, open Postman, which is the app you installed at the start of the labs, during the pre-reqs.
4. To import the downloaded collections into Postman, select **File**, and then click **Import…**



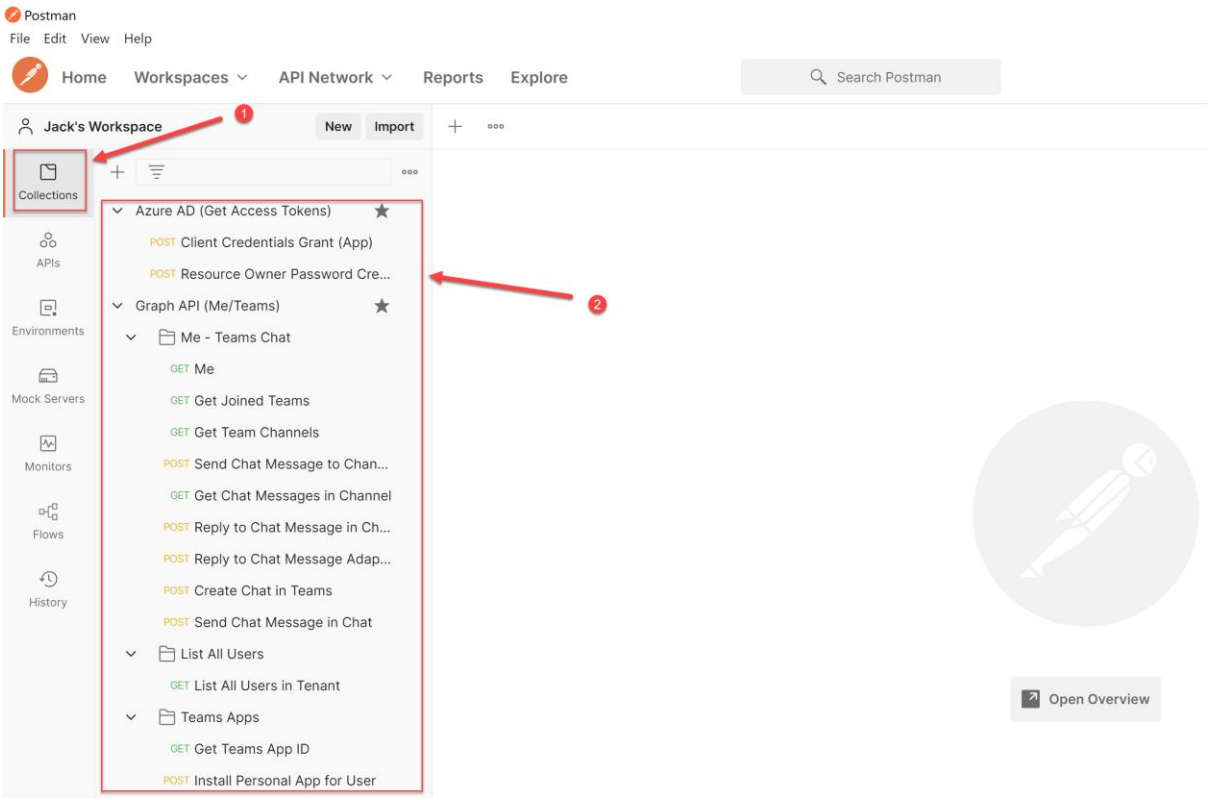
5. Select **Upload Files**, then **select the 2 collections that are contained within the Postman folder within the Repo that you downloaded to run this lab,** click **Open**



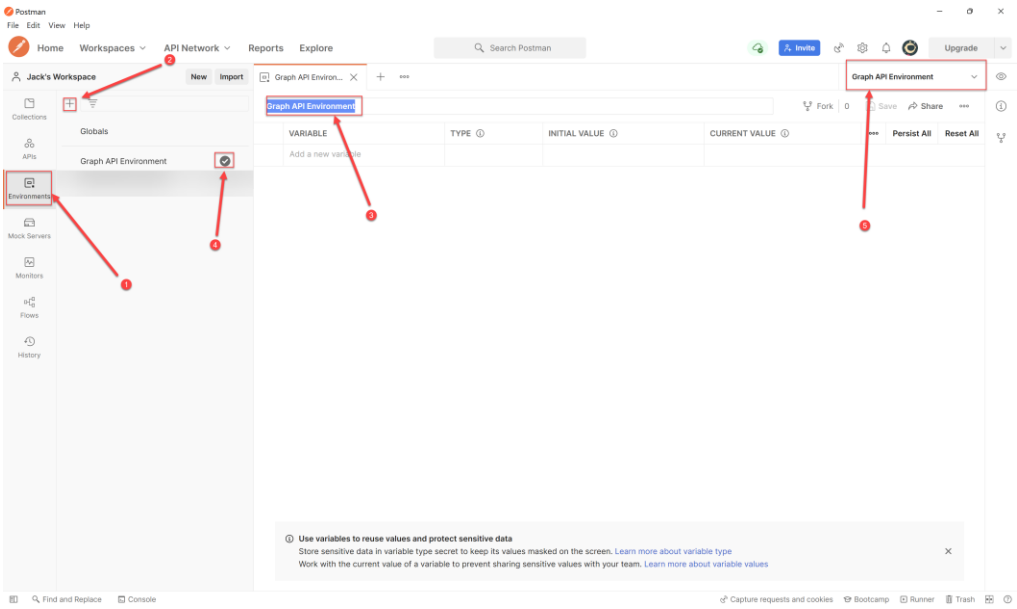
6. Review the names of the files that are being imported, there should be 2 collections **Azure AD & Graph API (Me/Teams)**, click **Import**



7. Ensure that **Collections** is selected in the left navigational menu. If so, you should now see the following Collections and Requests within those collections, once you have expanded the folder structure:



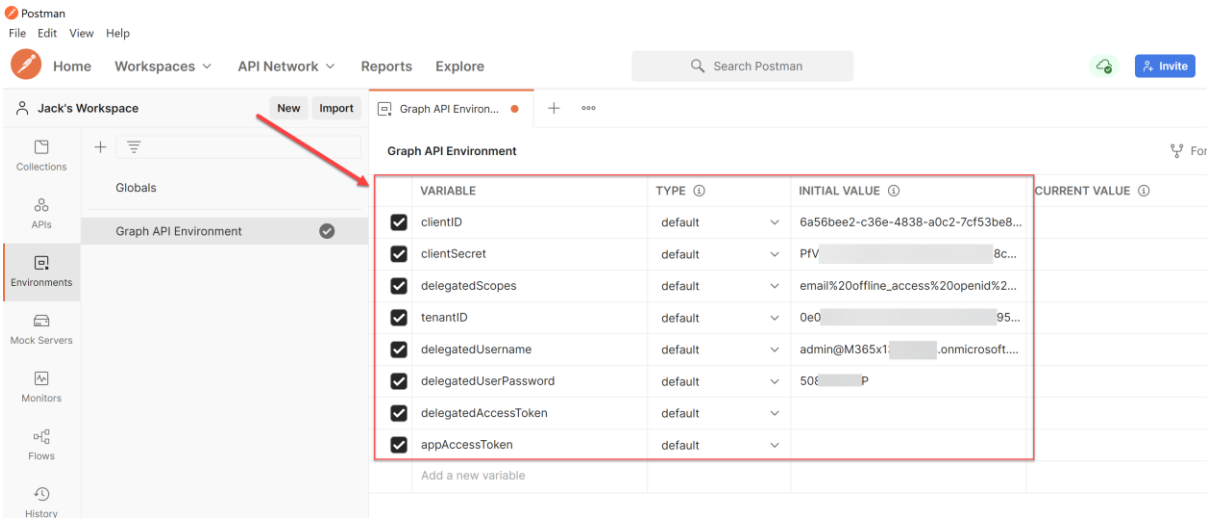
8. Similar to the Graph Explorer, the collections that you have just imported into Postman contain sample requests, that will allow you to interact with the Microsoft Graph API. The key differences here are that, when using Postman; we will need to get an Access Token manually, which are the 2 requests in the Azure AD (Get Access Tokens) folder, we can work with both App and Delegated Permission scope access tokens, and we can work with variables, by provisioning an environment and populating it with variables and values. Postman also has an abundance of other features that are useful to professional developers, such as code-snippets, mock servers, flows, and many others. I would encourage all developers who want to work with the Microsoft Graph API to get familiar with this tool.
9. Now that we have the collections imported into Postman, let's create an Environment and populate it with some variables we will need to use to make the requests function. Click **Environments**, select the **+** button and give the environment a name, such as **Graph API Environment**. Ensure that the **grey tick** is enabled to set this as the current environment. Review the environment is set to the one you just created in the top right of Postman.



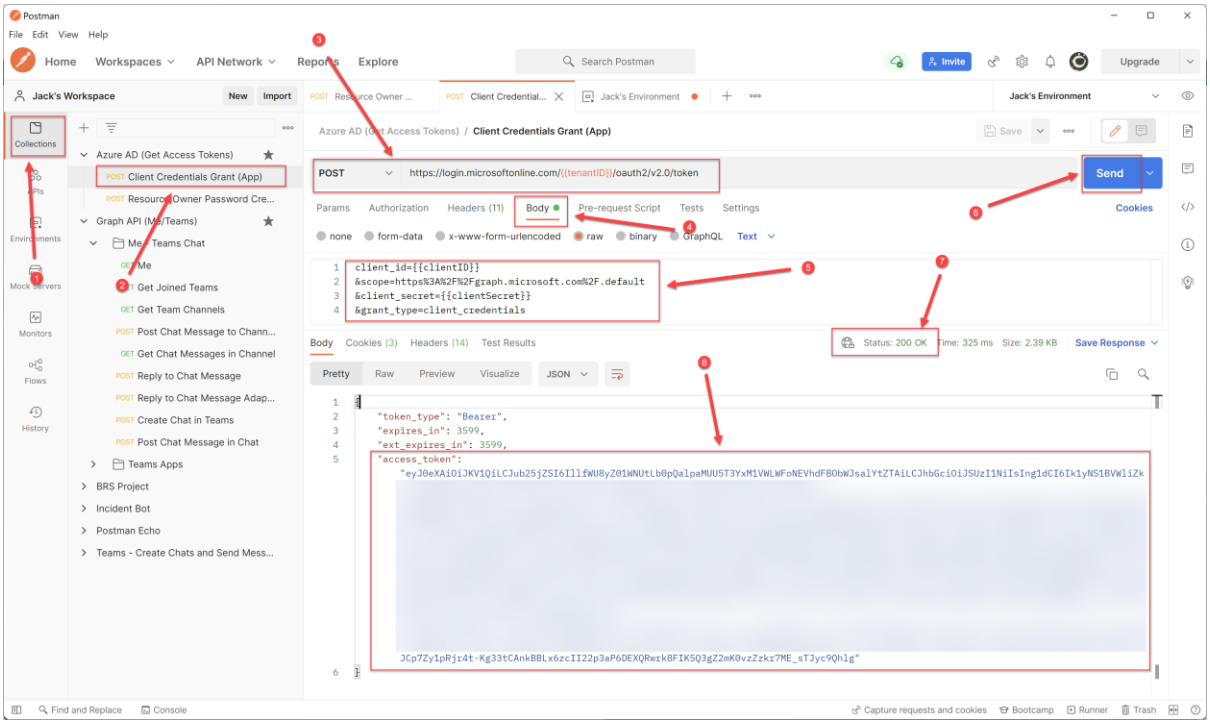
10. We now need to create the following variables, in this environment, and set the initial values, from the appsettings.json file, to the following:

Variable Name	Initial Value
clientID	***MicrosoftAppId***
clientSecret	***MicrosoftAppPassword***
delegatedScopes	**email%20offline_access%20openid%20profile%20User.Read%20Team.ReadBasic.All%20Channel.ReadBasic.All%20ChannelMessage.Send%20ChannelMessage.Read.All%20Chat.ReadBasic%20ChatMessage.Read%20ChatMessage.Send**
tenantID	***MicrosoftDirectoryId***
delegatedUsername	***admin@DOMAINNAME.onmicrosoft.com***
delegatedUserPassword	***admin@DOMAINNAME.onmicrosoft.com's password***
delegatedAccessToken	**DO NOT POPULATE WITH INITIAL VALUE (LEAVE BLANK)**
appAccessToken	**DO NOT POPULATE WITH INITIAL VALUE (LEAVE BLANK)**

Once complete, it should look something like this:

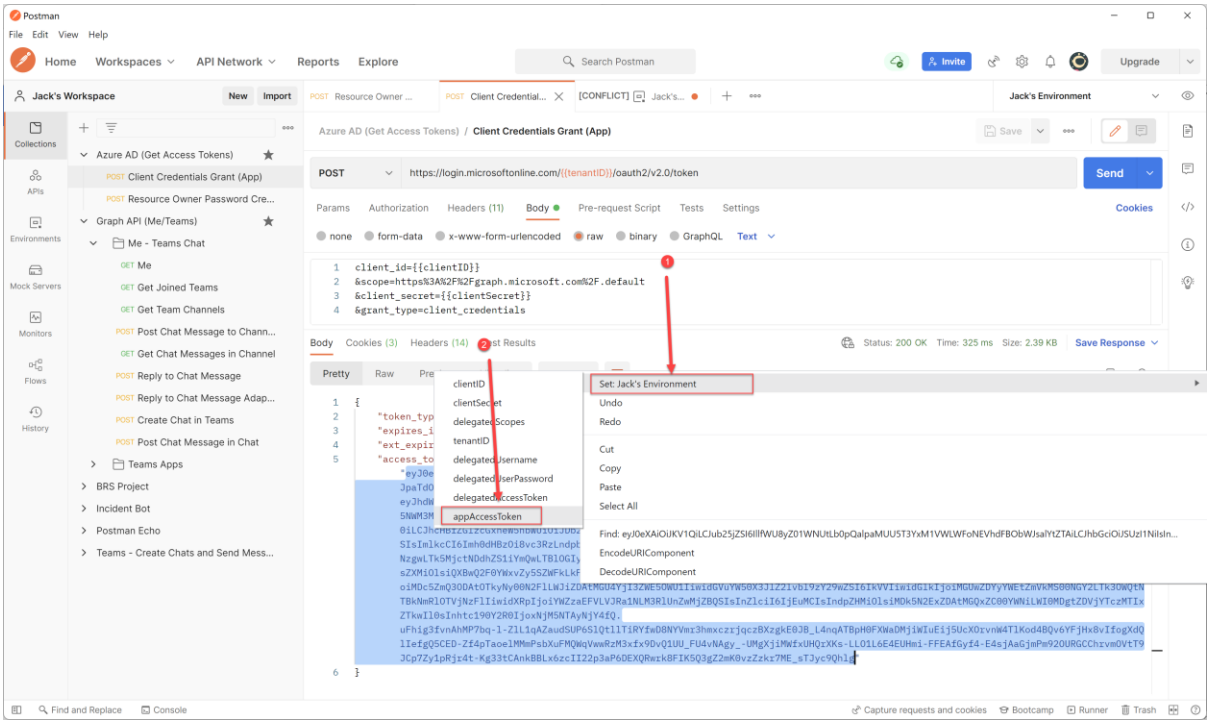


11. Once the variables have been populated, we now need to go and get an Access Token from Azure Active Directory, that will work with the Microsoft Graph API. We'll start first with getting an access token that contains App Permission scopes (roles). To get an App Permission scope access token, you need to use the Client Credentials Grant Flow with Azure AD, (more details on this grant flow can be found here - [OAuth 2.0 client credentials flow on the Microsoft identity platform | Microsoft Docs]()). Essentially, this flow requires you to send a POST HTTP request, to a specific API Endpoint, with the TenantID of the tenant you want to receive an app token for, in the URL of the request. In the body of the request, details referring to your Azure AD App Registration needs to be submitted, such as the MicrosoftAppID (client_id), the scope and the MicrosoftAppPassword (client_secret). This should then return an access token, with app permission scopes (roles), assuming the correct consent is in place. Click **Collections**, then select the **Client Credentials Grant (App)** request. Review the **HTTP Request Type of POST and the URL**, this is using the tenantID variable you provisioned in the previous step as part of the URL. Review the **Body of the request**, this should contain **client_id, scope, client_secret and grant_type**. Variables are used for the client_id & client_secret values, but for scope, we have left it hard-coded in this request, and set it to **https://graph.microsoft.com/.default**. When working with app permissions, you must use the /.default scope, which essentially asks for the scopes that are listed in your App Registration, which you configured in section 1 of this lab. Once you have reviewed these settings, hit **Send** to make the request. All being well, you should see a response status code of **200 OK**, and the response, will contain an access_token. We will now set the variable of appAccessToken to this access token, and review this access token to see what it contains.



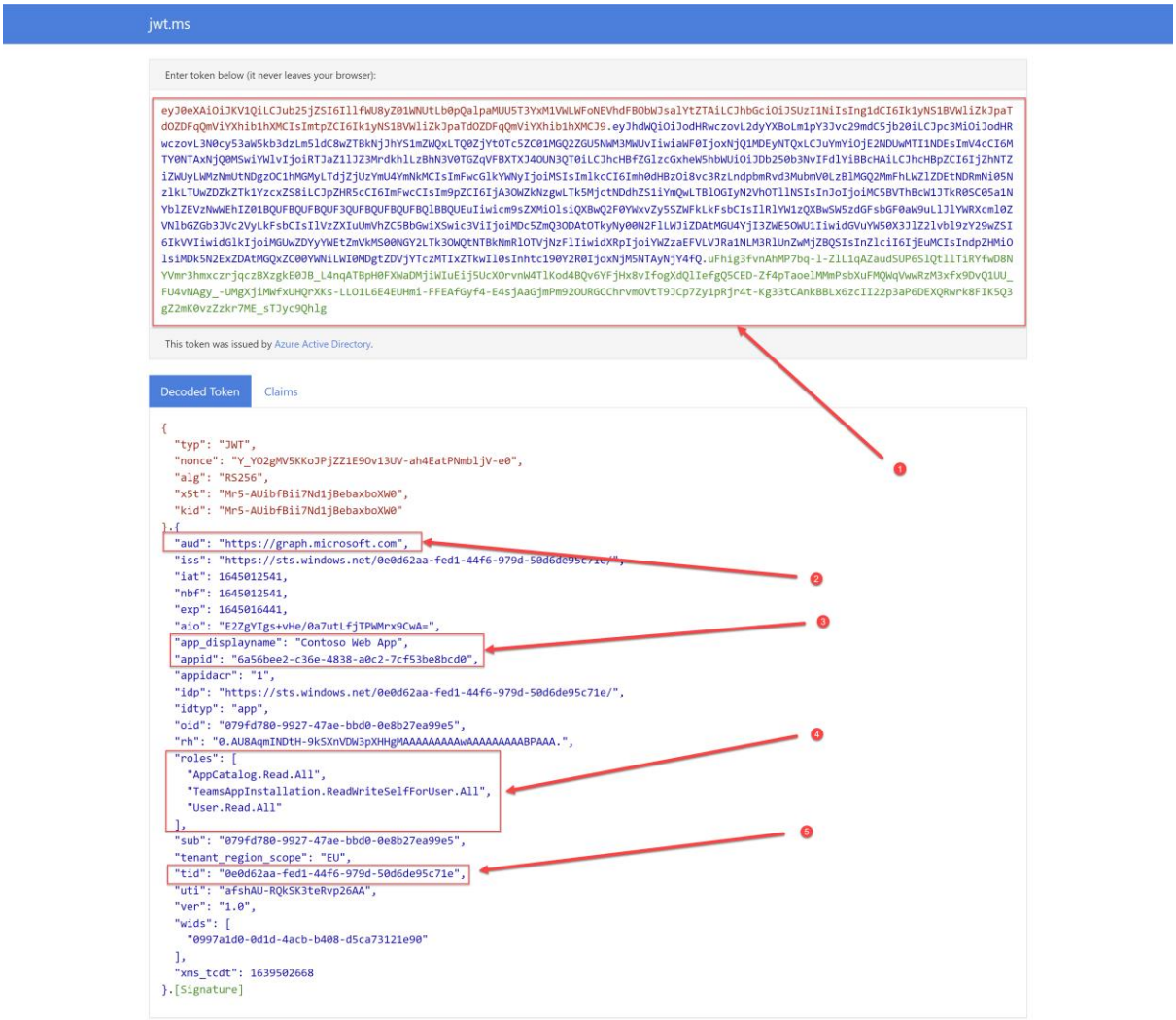
NOTE: If you receive an error when making the request of 'unauthorized_client', this means that consent for the app permission scopes, and the creation of the Enterprise app in the tenant you are requesting the token for, has not been completed. Go back to step 25, in section 1, of this lab document to go through the consent process, and solve this issue. Once the consent has been granted, try sending the request again in Postman and you should receive an access token in the response.

12. **Select all of the value of the access_token** in the response (everything in between the quotation marks) and **right click,** highlight the option to **Set: Graph API Environment** and select **appAccessToken**. This will set the variable value to this access token.



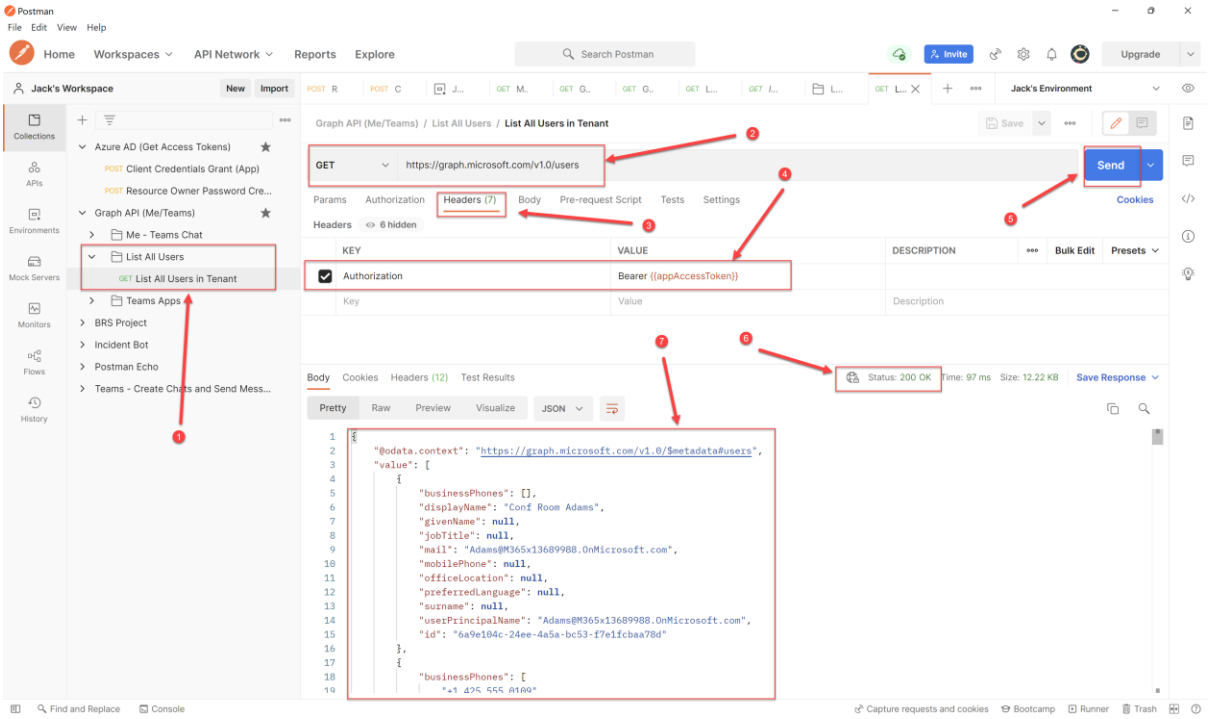
13. **Copy the value of the access_token, in the response, to your clipboard,** and open https://jwt.ms in your preferred web browser client. **Paste the value of the access token**

into the form. You can now review the claims that are embedded within the decoded access token, the most important are:
Aud: This should be The Microsoft Graph API
App_displayname & appid: Should match the name and Client ID of your App Registration
Roles: These are the scopes that are included in the access token, that your customer has consented to. Roles signify that these are application permission type scopes (delegated scopes are included in the 'scp' claim)
Tid: The tenant ID that this token is allowed to be used within



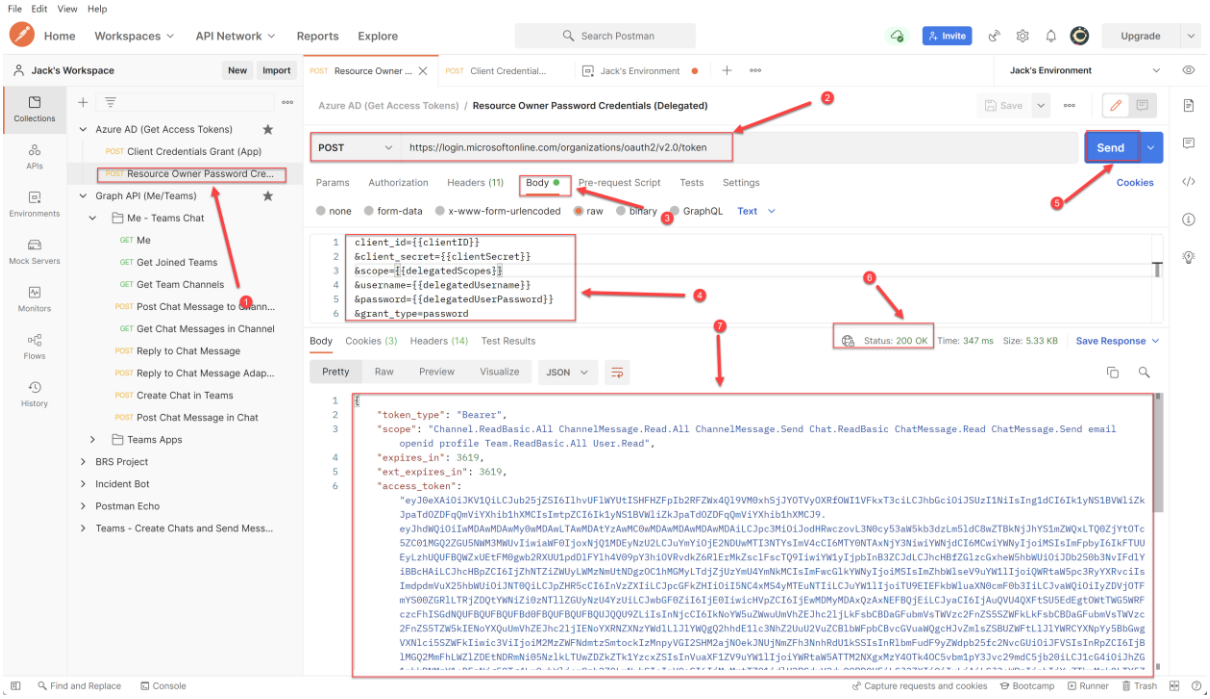
14. Now that we have reviewed what's included in the decoded application permissions access token, we can use it to make requests to the Microsoft Graph API. Firstly though it's worth noting that application permission type scopes are usually highly privileged, and are often not scoped to subsets of resources (some exceptions to this rule do exist, such as app policies that can restrict access to resources in Exchange for example). For that reason, we need to be wary that it is, in most scenarios, preferable to use delegated access tokens where possible, as this will require less privileges to perform actions. There are scenarios though, where it doesn't make sense to use delegated, and where application access tokens must be used, such as if users never log into your application, or if you need organisation/tenant-wide permissions to perform actions. **Go back into Postman,** click on the **List All Users in Tenant** request under the List All Users directory. Review the URL and HTTP Request Type. Select **Headers**, and **review the Authorization header,** this is using the

appAccessToken variable as the Bearer token for authentication with the Microsoft Graph API. Hit **Send**, and if successful, you will receive a **200 OK response code**. If you review the Body of the response, you will see that all of the users of the tenant have been sent back in the response. This is a useful request when working with Teams proactive notification bots (which we will in the next section) as we need to discover the users in a tenant, and then see if they have our Bot installed, to allow our app to send them a proactive message for notification purposes. We will review this code in Visual Studio later.

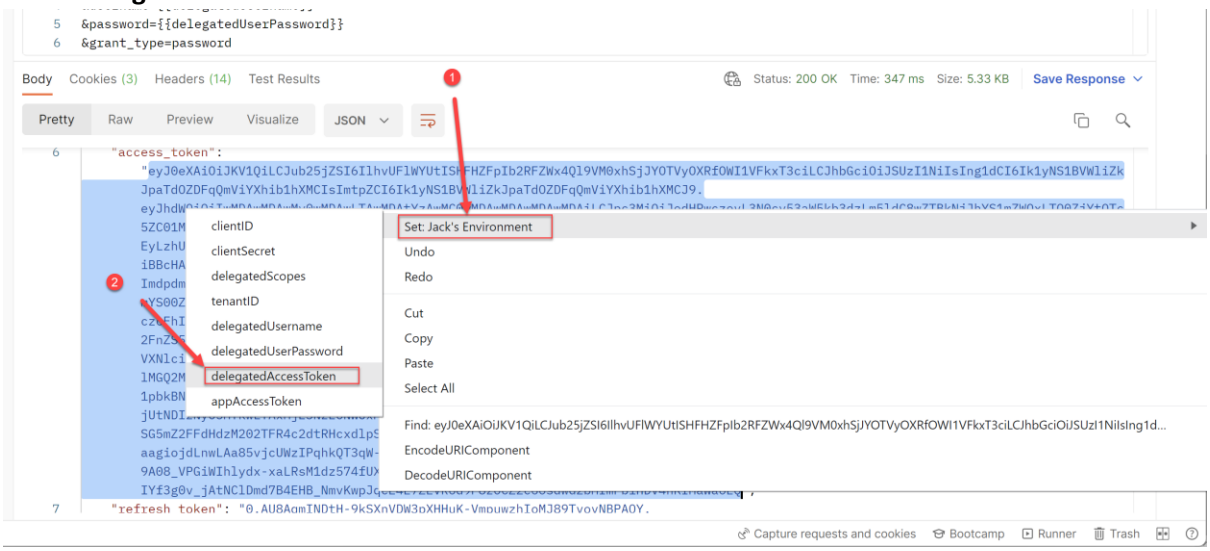


15. In the collection you downloaded and imported, there are a selection of sample requests that are designed to work with delegated permission type access tokens. In order to retrieve a delegated permission type access token, with specific scopes, we can use the Resource Owner Password Credentials Grant (more information on this here - [Sign in with resource owner password credentials grant - Microsoft identity platform | Microsoft Docs]()) for test/dev purposes. In production scenarios, this grant flow type should ***never*** be used, as more secure alternatives exist. This grant type allows you to pass the username and the password of a user to Azure AD, using the Token Endpoint APIs, to retrieve an access token for that specific user, with delegated permission scopes in the access token. Select the **Resource Owner Password Credentials (Delegated)** request, review the HTTP Request Type of **POST** and the API endpoint URL, in this scenario we do not need to specific the TenantID, instead we can use the **organizations** endpoint, as the App Registration we created is multi-tenant, and the tenantId can be retrieved from the user, when the request is made to Azure AD. Review the **Body**, you will see that the following: client_id is the MicrosoftAppID, client_secret is the MicrosoftAppPassword, scopes contains a list of the scopes we want within the delegated permissions type access token, this is different to working with app permission type access tokens, as you can define what scopes you want in the request, not just in the app registration, although it makes sense to keep the scopes in the app reg and the request consistent in most scenarios (best practices state that the app registration should contain a superset of all the permissions you app will ever request, but in each request you can ask for a subset of those permissions). In addition to this, we are also passing the username and password of the user we are authenticating as ***(note: this grant***

type does not work for users who have MFA enabled!) and finally, the grant_type of password is defined. Once you have reviewed the body, hit **Send**, if successful you will see a status response code of **200 OK**. Review the body of the response, this should contain the **scopes** and the **access_token** that we can use to make calls to Graph API using delegated permission types, on behalf of a user, rather than an app.



16. **Select all of the value of the access_token** in the response (everything in between the quotation marks) and **right click,** highlight the option to **Set: Graph API Environment** and select **delegatedAccessToken**. This will set the variable value to this access token.



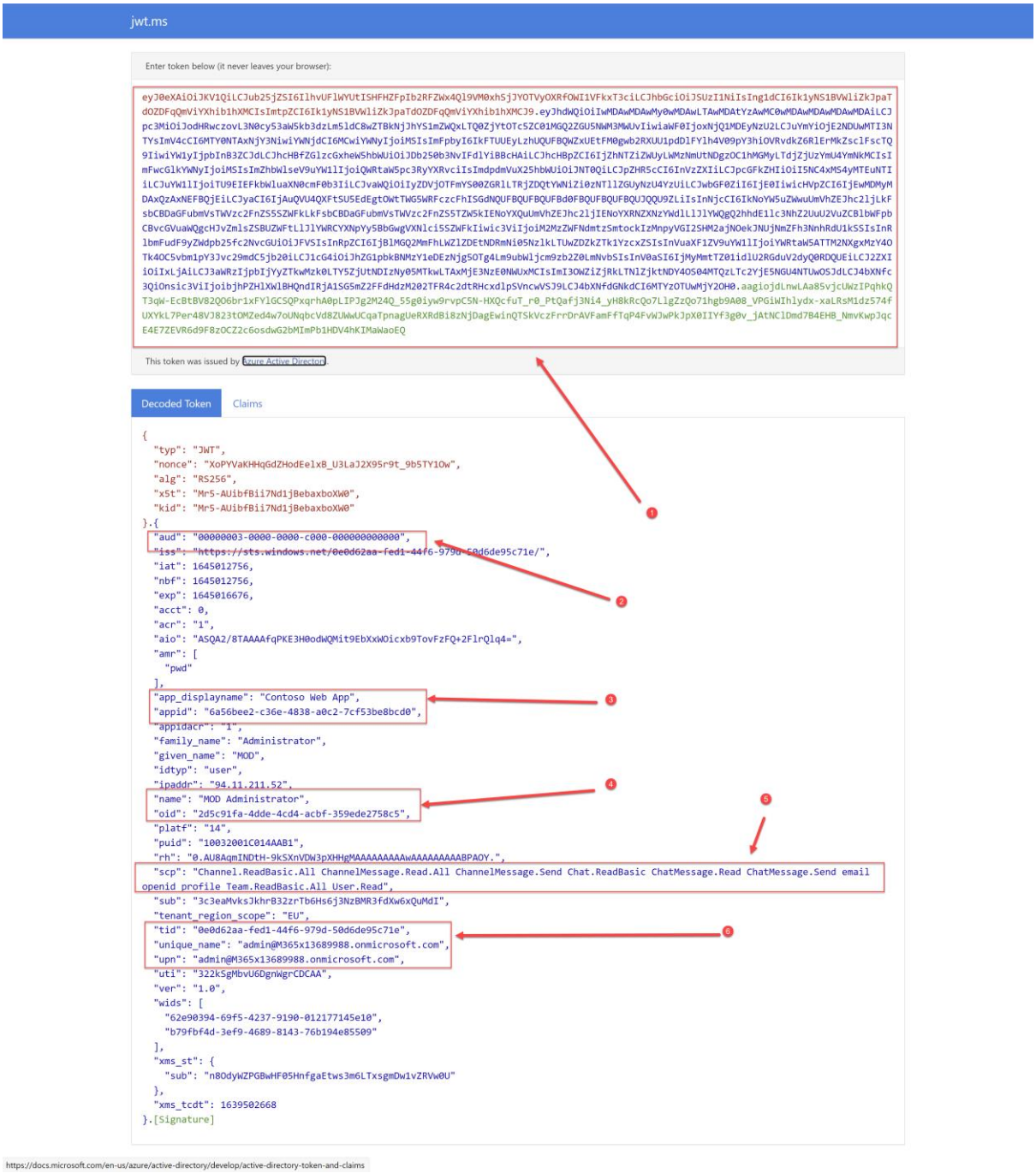
17. **Copy the value of the access_token, in the response, to your clipboard,** and open https://jwt.ms in your preferred web browser client. **Paste the value of the access token into the form.** You can now review the claims that are embedded within the decoded access token, the most important are:
Aud: This GUID is a well-known GUID and represents the Microsoft Graph API
App_displayname & appid: Should match the name and Client ID of your App Registration
Name & oid: This is the name of the user and the unique (globally) ID of the user. Most third-party applications will scope the OID of the user, when they do identity matching, as

this is guaranteed to be unique across Azure AD, and will remain consistent, regardless of email changes, surname changes, etc…

Scp: These are the scopes that are included in the access token, that your customer has consented to. Scp signifies that these are delegation permission type scopes (application scopes are included in the 'roles' claim)

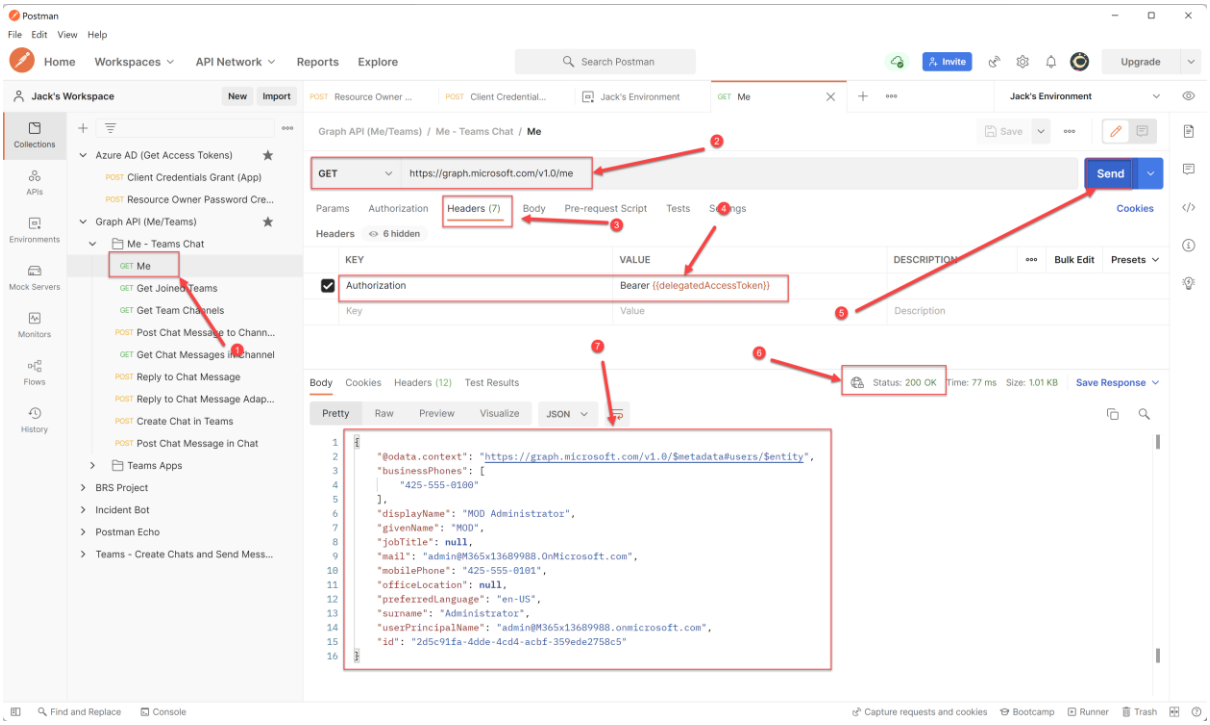
Tid: The tenant ID that this token is allowed to be used within

Unique_name and UPN: UPN is often used to match an Azure AD user, to an existing user in a third-party app, during the first authentication the user performs through Azure AD to login to the third-party app. After this initial identity match has been performed, it is recommended that OID and TID are used to match the user for subsequent authentications.



18. Now, let's head back to Postman, and make a simple request to the ME resource in Graph API, using a delegated type access token for authentication. Open the **Me** request, found

within the 'Graph API (Me/Teams)\Me – Teams Chat' directory. Review the HTTP Request type of **GET** and the URL of the API endpoint **v1.0/me**. As we are using a delegated access token, no further information is required in the body of the request, but we do need to authenticate, which is passing a Bearer Token in the headers, click **Headers** and review the **Authorization** header to see how this uses the {{delegatedAccessToken}} variable to authenticate with Graph API. Once you are happy that the request is ready, hit **Send**. If the request is successful, you will receive a **200 OK** status response code, and information about your user will be displayed in the **response body**.



19. Now that you have been introduced to these Graph API and Azure AD concepts, continue to work through the 'Graph API (Me/Teams)\Me – Teams Chat' directory, to get the Teams the user is a member of, discover the channels of a specific Team, and send a message into the conversation associated with that channel. You can then also continue to work with the requests in that directory to start a 1-2-1 or group chat in Teams, and then send messages, as that user, into those chats.

Nice one! Now that you are familiar with the key concepts associated with the Microsoft Graph API, we will now continue working on the Talent Management App, and review how the app uses the Graph API to install the App, that contains the bot, programmatically, retrieve chat/conversationIDs between the Bot and the Users, and then use this information to proactively message users with an adaptive card.

8) Proactive Notifications in Teams (via the Teams Bot)

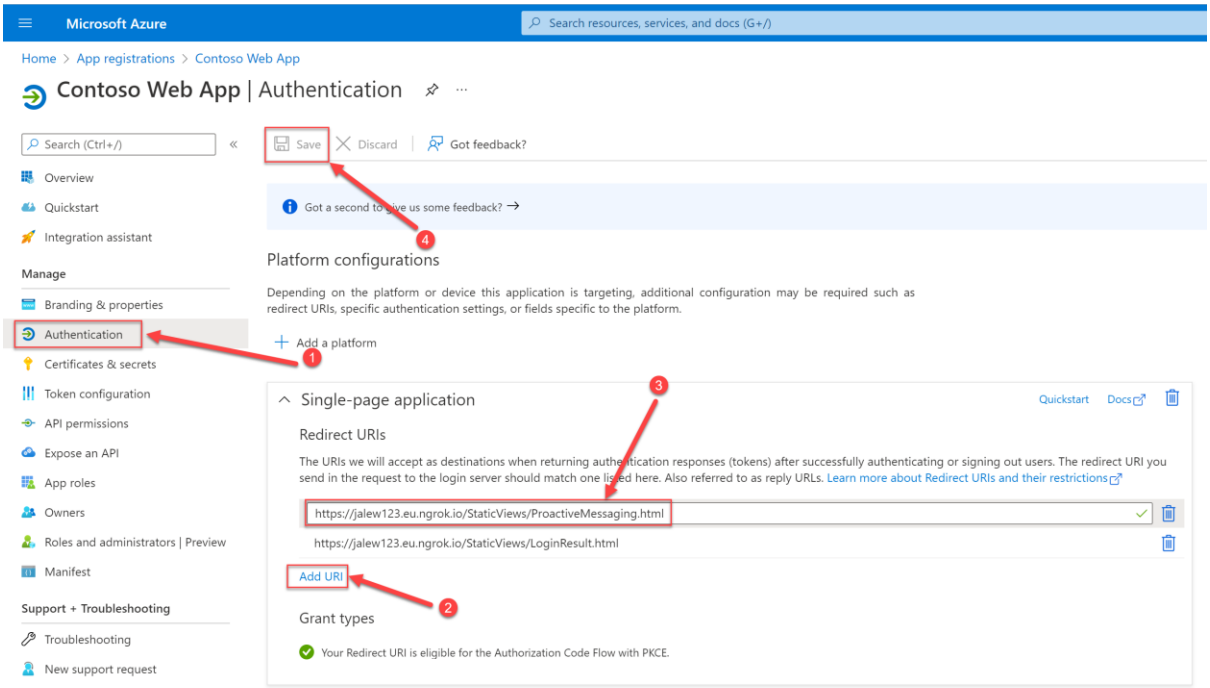
One of the most popular use cases for apps in Teams is the ability for you to notify an individual in Teams when an event happens in your third-party app. To enable this, Teams supports Proactive Notifications/Messaging. Proactive Messaging in Teams, can be done in a number of different ways; Graph API, Connectors & Bots. There are pros and cons to each approach, but in our experience, Bots offer the most functionality/flexibility, as they support 2 way communications and actionable notifications, but the trade off is they require the most code/configuration to get them to work, and also require that your Teams app is installed into the place where they want to send that notification (for example, for a bot to send a message to a specific user, the app that

contains the bot, must be installed for that user – or if the bot wants to message a Teams channel, the app that contains the bot, must be installed into that Team, before it can message the channel).

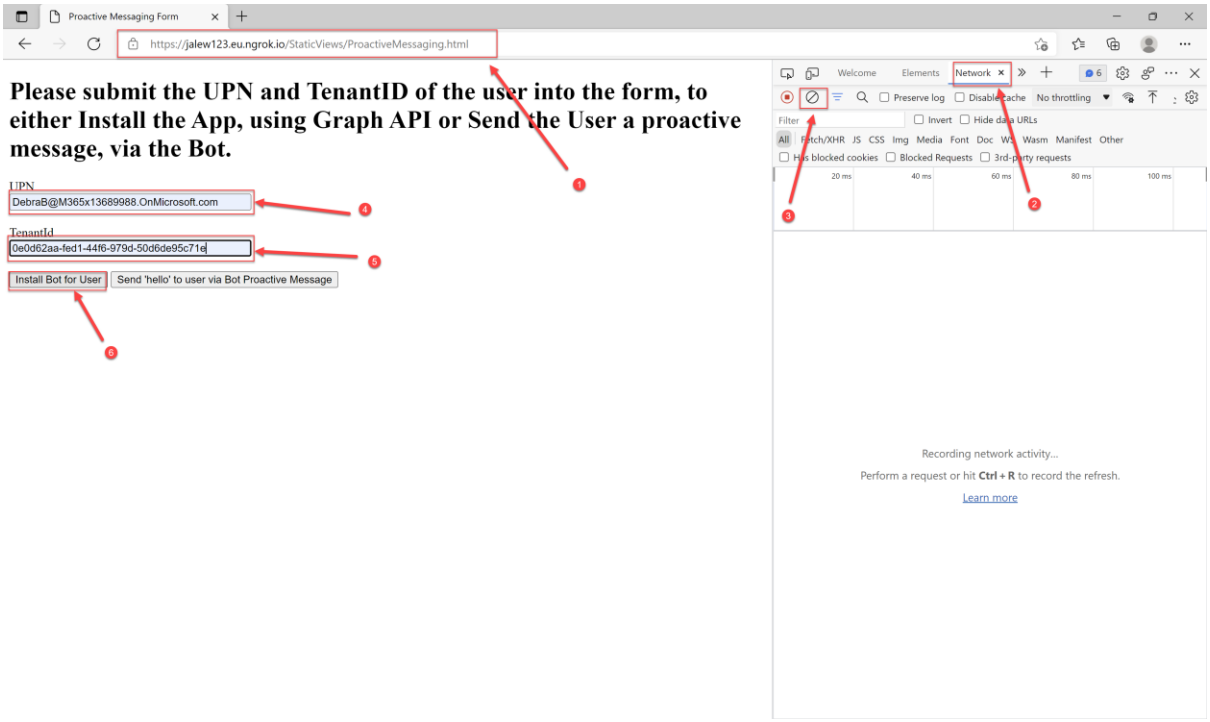
The other 2 options (Graph API and Connectors) can be used but lack the ability for you to add actionable buttons onto the adaptive cards that you send, as such, we define them as 'one way notifications'. The trade of is that they are easier and quicker to get up and running.

As we want to demonstrate the preferred way of doing this, we are going to demonstrate how you can use bots, to proactively message individuals, which also requires some Graph API calls to be made, to discover if the app is installed in the scope you want it to message, and if it isn't, install it via Graph API (meaning you are going to build on the Graph API knowledge you gained in the previous step)! Once all that is in place, it will be the Bot that sends the message/adaptive card to the user.

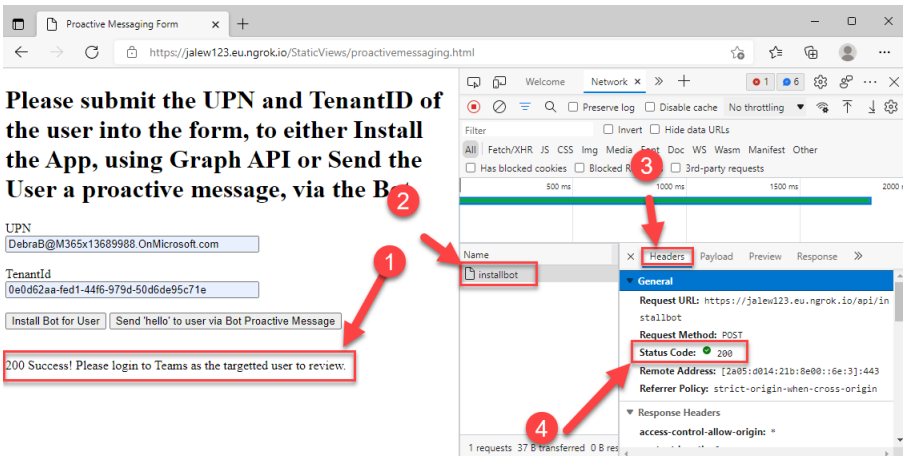
1. Before we begin with the proactive messaging, we need to add another Redirect URI to our Azure AD App Registration. Go to **https://portal.azure.com** and **navigate to your App Registration**. Select **Authentication**, then click **Add URI** and enter your URI as: **https://NGROKURL/StaticViews/ProactiveMessaging.html** (be careful, as this is case sensitive), and then click **Save**.



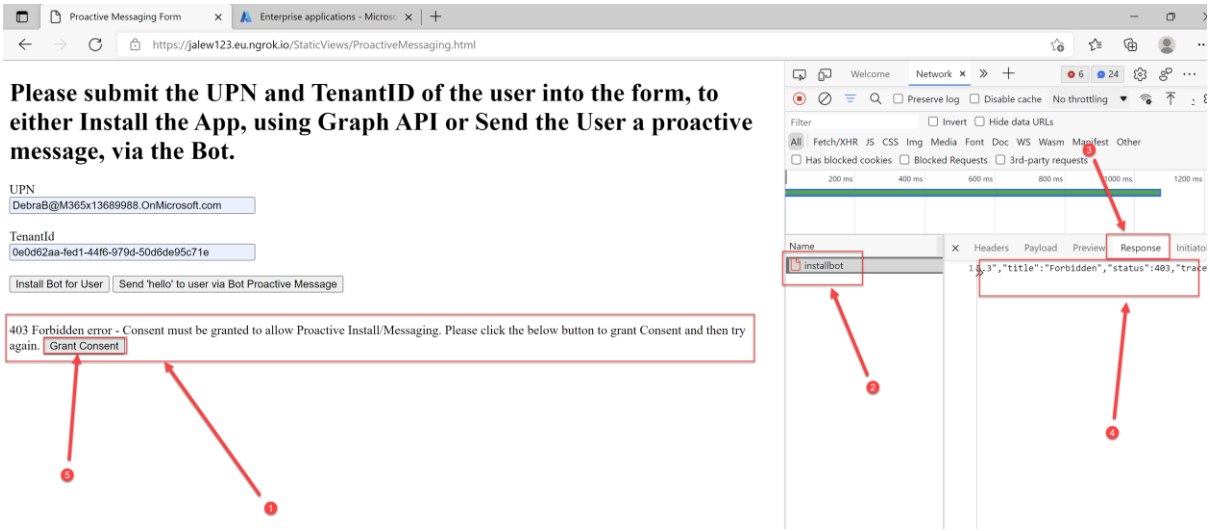
2. Then go to **https://NGROKURL/StaticViews/ProactiveMessaging.html** in a new browser tab. This is a simple page, that allows you to submit the UPN (or Alias) of a user, and their Tenant ID, to the app's APIs, to get your app to proactively install your Teams app for that user, or proactively message that user, via the Bot, with a simple 'hello' message. Press **F12** to open the developer tools, in the browser, and then select the **Network** tools, hit the **Clear Network Log** button. Enter the **UPN of a user who does not yet have your app installed (for example DebraB@YOURTENANTDOMAIN.ONMICROSOFT.COM)** and enter the **TenantID** where that user resides. Click **Install Bot for User**.



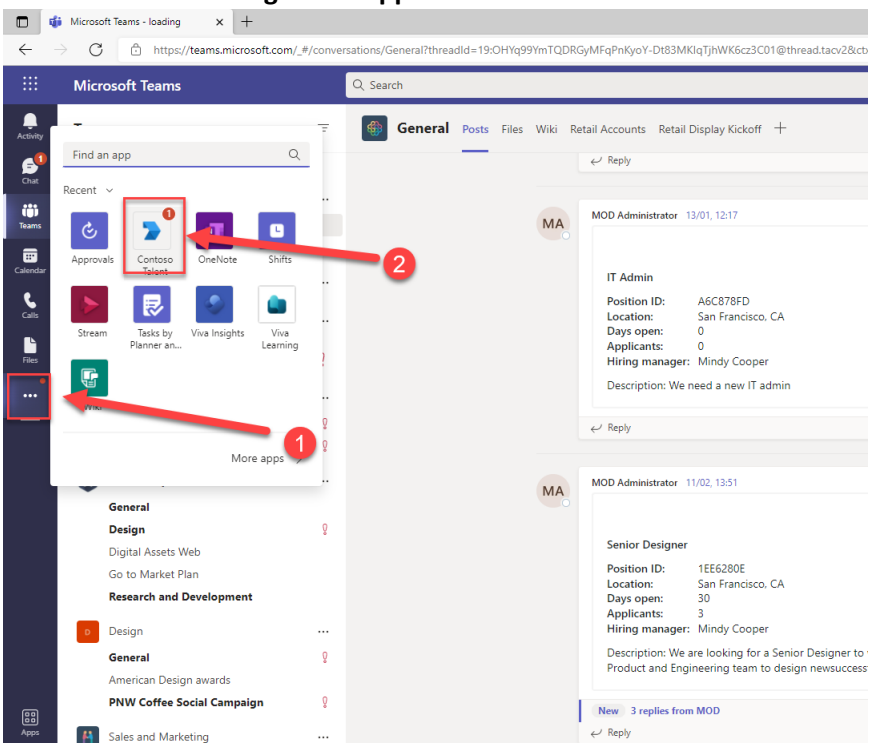
3. A request will now be made to the appropriate API endpoint controller (in this case /api/installbot) with the UPN and TenantID submitted to that API endpoint in the body of the request. There are now a number of different things that could happen within the app, and you should receive an appropriate response to notify you of if the request was successful or not. **If the request isn't successful, follow the guidance that is displayed on the screen, under the button you clicked to Install the Bot for the User, and use the network developer tools to help troubleshoot the issue.** The reasons for failure vary from not having the correct consent, to your request being incorrect (maybe you sent the wrong UPN or TenantID), although you should be able to troubleshoot this by reviewing the response information, associated with the request that was made to the endpoint controller, from within the Network developer tools in the browser. **If the Install is successful, you will receive a 200 Success message, and can now move onto the next step in this section of the lab.** Here is an example of a successful request:



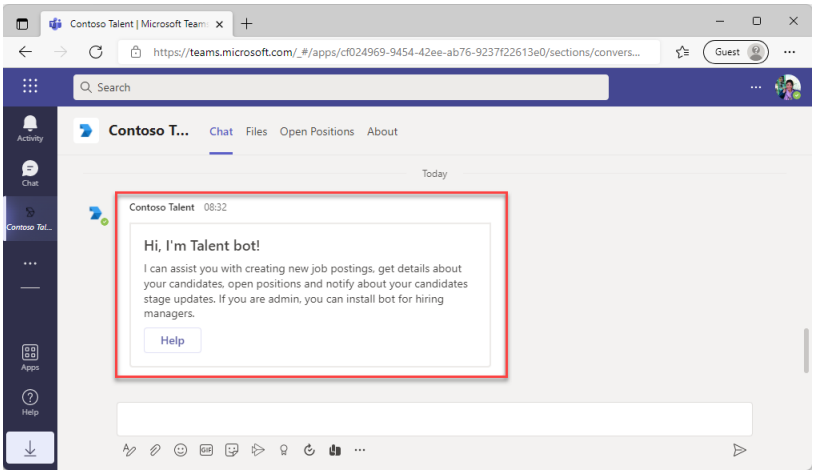
Here is an example screen of a failed request that I am troubleshooting: In this scenario, the issue is an easy fix, as you can just click the button to grant Admin Consent, so that your App has the right authorisation to install the bot for that user:



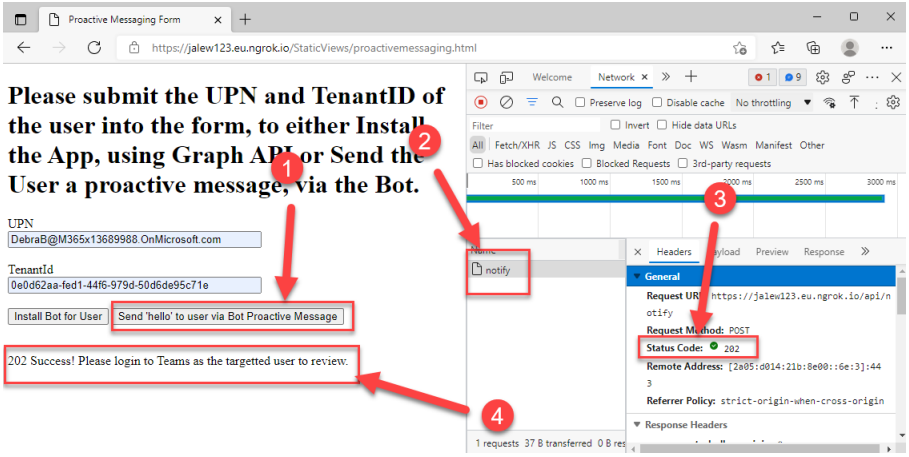
4. If you received the 200 Success message response, you should now open a InPrivate/Incognito web browser session, and go to https://teams.microsoft.com and log in as the user who you targeted the bot install to.
5. Let's now go back to the Proactive Messaging HTML page, and click the **Send 'hello' to user via Bot Proactive Message.** You can use the same methods as described in step 3 of this section of the lab to troubleshoot, but assuming you receive a **200 Success message**, you can **Go back into Teams as the targeted user**, click the **ellipsis …** and you should see the **Contoso Talent Management App** installed for the user, with a notification. Click on the **Contoso Talent Management app.**



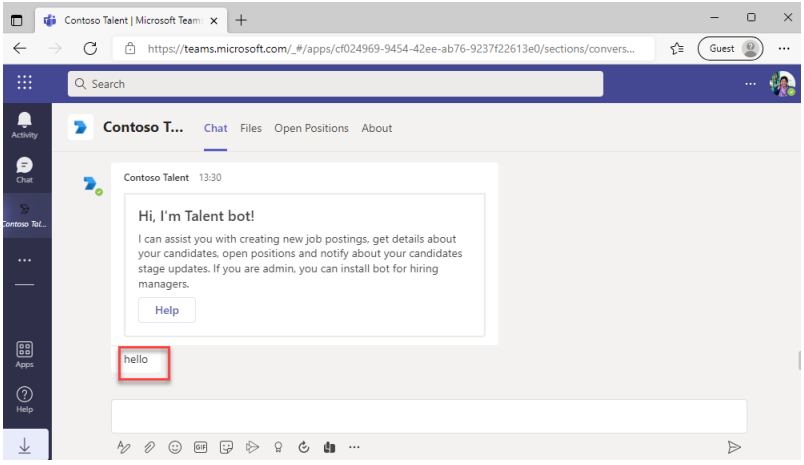
6. You should see a **Welcome Message from the Bot.**



7. Navigate back to the **ProactiveMessaging.html** page and click the **Send 'hello' to user via Bot Proactive Message.** If you review the **Network developer tools**, you will see a request being sent to the **notify** bot controller API. If the request is successful, you will receive a **202 Status** in the response to the request.



8. Navigate back into Teams, as the targeted User, and you should now see a message saying '**Hello**' from the Contoso Talent Bot.



9. Let's review some of the code in Visual Studio, to see what's happening within the API Controller, and what happens once the API controller receives a request, with the expected information it requires to perform a Bot Install and to send a Proactive Message to a User in Teams. Open **Visual Studio** and navigate to the **BotController.cs** file. You will notice there are two endpoints that handle the installation and notification requests named **InstallAsync** and **NotifyAsync** respectively.

```
[HttpPost]
[Route("api/installbot")]
public async Task<IActionResult> InstallAsync(
    [FromBody] UserTenantMessageRequest request,
    CancellationToken cancellationToken)...

[HttpPost]
[Route("api/notify")]
public async Task<IActionResult> NotifyAsync(
    [FromBody] NotifyRequest request,
    CancellationToken cancellationToken)...
```

10. Let's first look at **InstallAsync**. This method will attempt to install the bot for the user and then return the installation result in the response.
11. Installation is handled by **GraphAPI** and is defined in the **GraphApiService.cs** file in a method called **InstallBotForUser**. This method goes through the following steps to install the bot for the specified user:

```
public async Task<InstallResult> InstallBotForUser(
    string aliasUpnOrOid,
    string tenantId,
    CancellationToken cancellationToken)
{
1   var token = await GetTokenForApp(tenantId);
    var upn = await GetUpnFromAlias(token, aliasUpnOrOid, cancellationToken);

    if (upn == null)
    {
        return InstallResult.AliasNotFound;
    }

    var graphClient = GetGraphServiceClient(token);

2   var teamsApps = await graphClient.AppCatalogs.TeamsApps
        .Request()
        .Filter($"distributionMethod eq 'organization' and externalId eq '{_appSettings.TeamsAppId}'")
        .GetAsync(cancellationToken);

    var teamApp = teamsApps.FirstOrDefault();
    var success = false;
    if (!string.IsNullOrEmpty(teamApp?.Id))
    {
        try
        {
3           var installBotRequest = new BaseRequest($"https://graph.microsoft.com/v1.0/users/{upn}/teamwork/installedApps", graphClient)
            {
                Method = HttpMethods.POST,
                ContentType = MediaTypeNames.Application.Json
            };

            await installBotRequest.SendAsync(
                new TeamsAppInstallation
                {
                    AdditionalData = new Dictionary<string, object>
                    {
                        { "teamsApp@odata.bind", $"https://graph.microsoft.com/v1.0/appCatalogs/teamsApps/{teamApp.Id}" }
                    }
                }, cancellationToken);


            // Getting the chat id will confirm the installation was successful and send the welcome message
4           await GetProactiveChatIdForUserInternal(token, aliasUpnOrOid, tenantId, cancellationToken);

            success = true;
```

1. Gets an application token containing the application scopes defined in our App Registration created earlier. If an alias was passed in the request rather than a full UPN, it then attempts to lookup the UPN based on the alias.
2. Assuming this was successful, we then must find the Teams app installation ID for your application. (***Note:*** *This is not the same as the Teams App ID that's contained in the manifest, each tenant will have its own identifier for your application stored against the Teams App ID*). We look this up using the [appCatalogs/teamsApps]

GraphAPI endpoint and the ODATA filter: **distributionMethod eq 'organization' and externalId eq 'Your Teams App ID'**.

3. Once this is found, we then install the application using the [teamwork/installedApps](#) GraphAPI endpoint.
4. Finally, we request the chat ID for conversation between the user and the bot to confirm the app was installed and to prompt Teams to send the user a welcome message if this was the first time the app was installed.

12. Let's now look at **NotifyAsync** in **BotController.cs** and see how proactive notification is handled by the bot. Like with the installation process, we call off to another service to handle the actual notification logic and the rest of the controller action is handling the request and response.
13. Open **NotificationService.cs** and find the **SendProactiveNotification** method. This is where the "magic" to find the conversation between the bot and the user exists before actually sending the notification message. The goal of this process is to get a **TurnContext** that represents the conversation between the bot and the user and is reasonably involved!

```
public async Task<NotificationResult> SendProactiveNotification(string aliasUpnOrOid, string tenantId, IActivity activity, CancellationToken cancellationToken)
{
    var (upn, chatId) = await _graphApiService.GetProactiveChatIdForUser(aliasUpnOrOid, tenantId, cancellationToken);

    if (upn == null)
    {
        return NotificationResult.AliasNotFound;
    }

    if (chatId == null)
    {
        return NotificationResult.BotNotInstalled;
    }

    var credentials = new MicrosoftAppCredentials(_appSettings.MicrosoftAppId, _appSettings.MicrosoftAppPassword);

    var connectorClient = new ConnectorClient(new Uri(_appSettings.ServiceUrl), credentials);

    var members = await connectorClient.Conversations.GetConversationMembersAsync(chatId);

    var conversationParameters = new ConversationParameters
    {
        IsGroup = false,
        Bot = new ChannelAccount
        {
            Id = "28:" + credentials.MicrosoftAppId
        },
        Members = new ChannelAccount[] { members[0] },
        TenantId = tenantId,
    };

    await ((CloudAdapter)_adapter).CreateConversationAsync(credentials.MicrosoftAppId, null, _appSettings.ServiceUrl, credentials.OAuthScope, conversationParame
    {
        var conversationReference = t1.Activity.GetConversationReference();
        await ((CloudAdapter)_adapter).ContinueConversationAsync(credentials.MicrosoftAppId, conversationReference, async (t2, c2) =>
        {
            await t2.SendActivityAsync(activity, c2);
        }, cancellationToken);
    }, cancellationToken);

    return NotificationResult.Success;
}
```

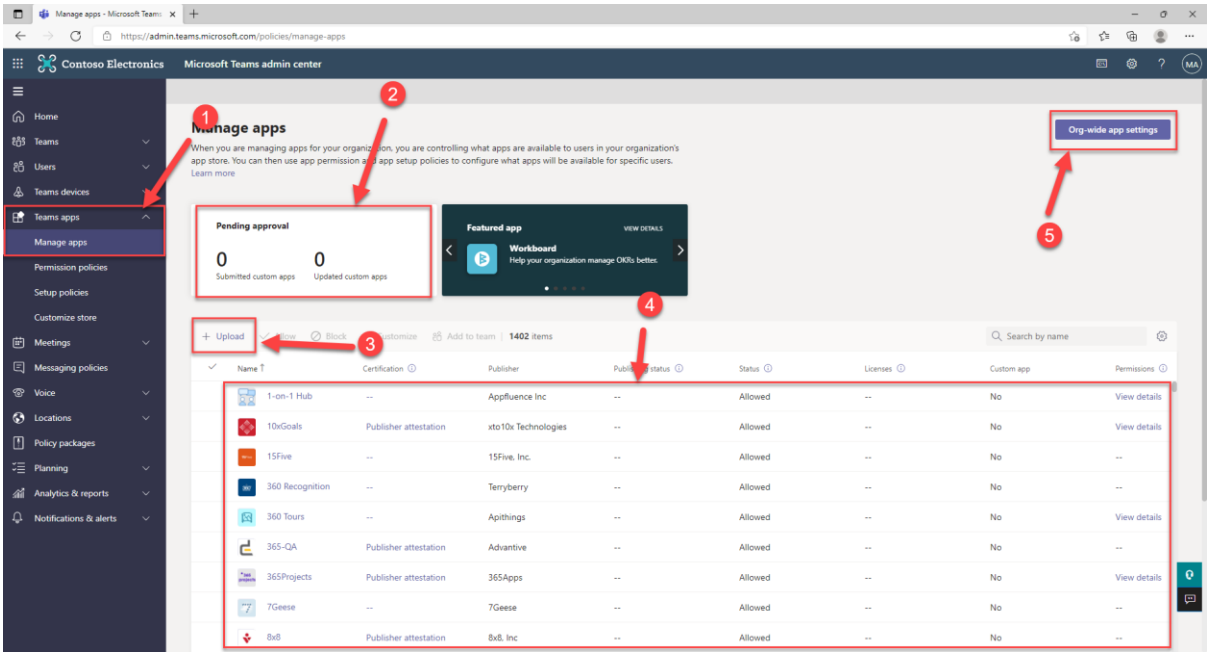
1. First it gets the chat ID for the conversation between the bot and the user. This will be used to get the internal ID for the chat members, in our case this will be the user we want to send the message to.
2. Then it creates a **ConnectorClient** and **ConversationParameters** to tell the bot service who the message is to (using the internal id we found above) and from (our bot). ***Note:*** *By convention, bot service internal identifiers are usually a number followed by and encoded value. In the case of a bot id, it's the number 28 followed by our Microsoft App Id*.
3. Finally, we find the **ConversationReference** by asking the bot service to create a new conversation between the bot and the user. This is delivered to us via a callback method which then allows us to get access to a turn context that represents the *continuation of the conversation between the bot and the user*.

Now that the Proactive Notifications are working

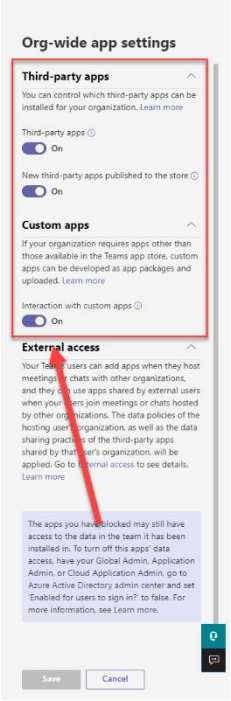
9) Teams Apps Management (via the Teams Administrator Portal)

In this step we will introduce you to the tools that Microsoft Teams Administrators have access to, to enable them to manage, install and pin third-party applications in Teams for the Tenants that they manage. This is an important concept to understand, as by working with the IT departments, and the global administrators, within a Tenant, you may be able to make accessing your application, from within Teams, easier for the users within that Tenant.

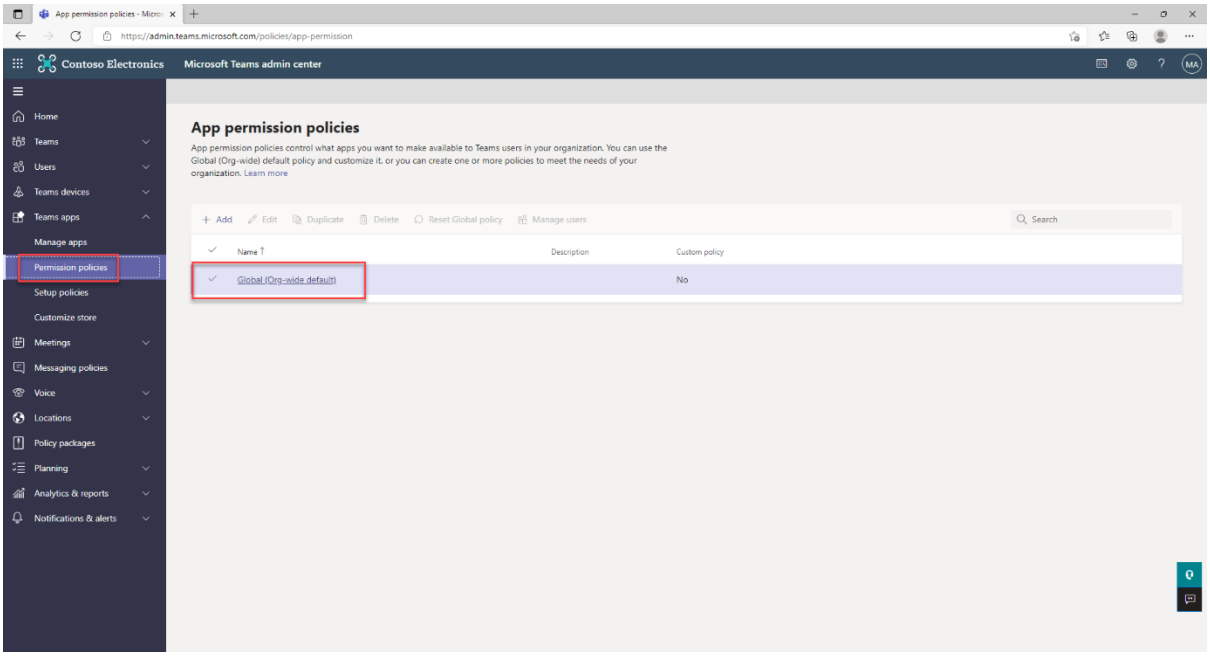
1. Navigate to https://admin.teams.microsoft.com/ and log in as the Tenant Global Administrator. Expand **Teams apps** and select **Manage apps**. This allows administrators to manage the apps that are available throughout their organisation. Here the administrators can **manage pending app approvals, that have been submitted by non-admin users**, **Upload** new apps, in the screenshot below section **4** allows administrators to manage apps at the individual level. Finally click **org-wide app settings** to view the options.



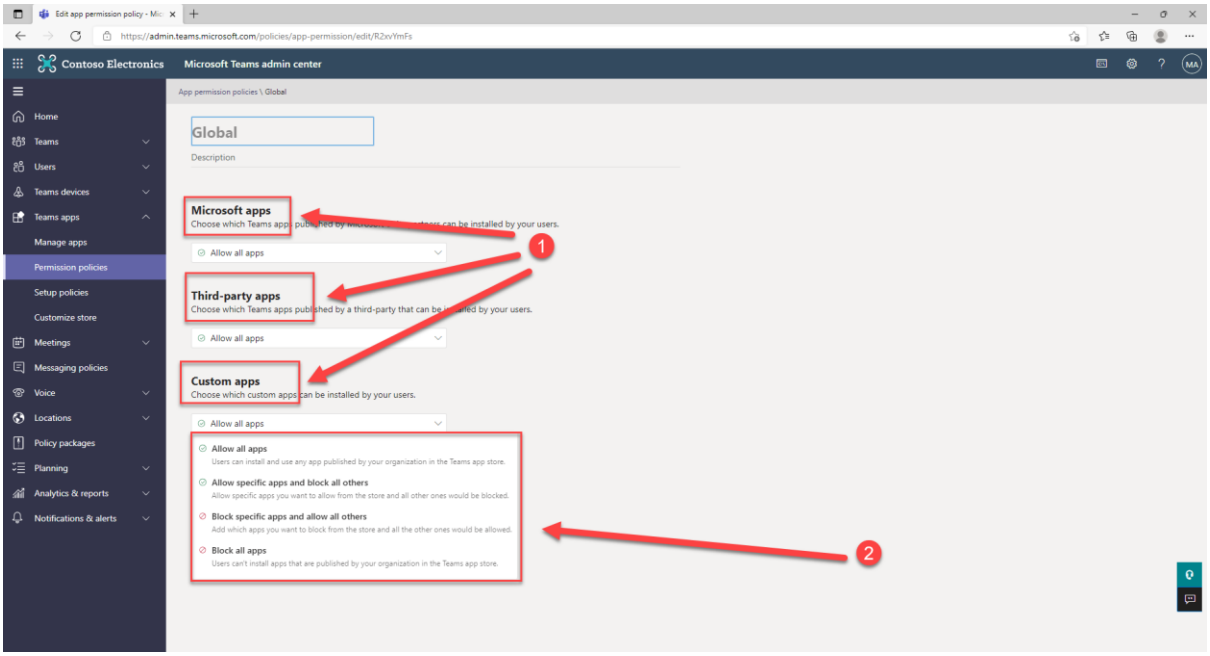
2. In org-wide app settings, administrators can enable of disable **Third party apps** in Teams, which would prevent the use of third-party apps if disabled. Here administrators can also enable or disable **custom apps**. Click **Cancel** once you've finished reviewing these settings.



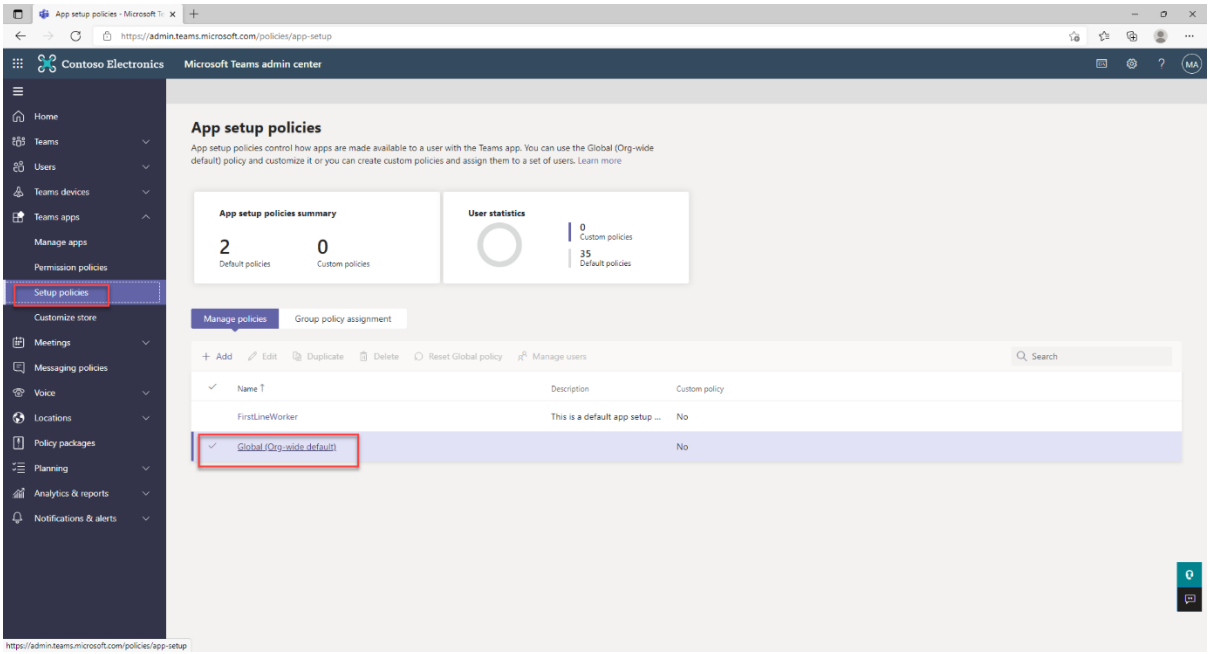
3. Now let's review the Permission Policies for Teams Apps. Select **Permission policies** and select the **Global (Org-wide default)**



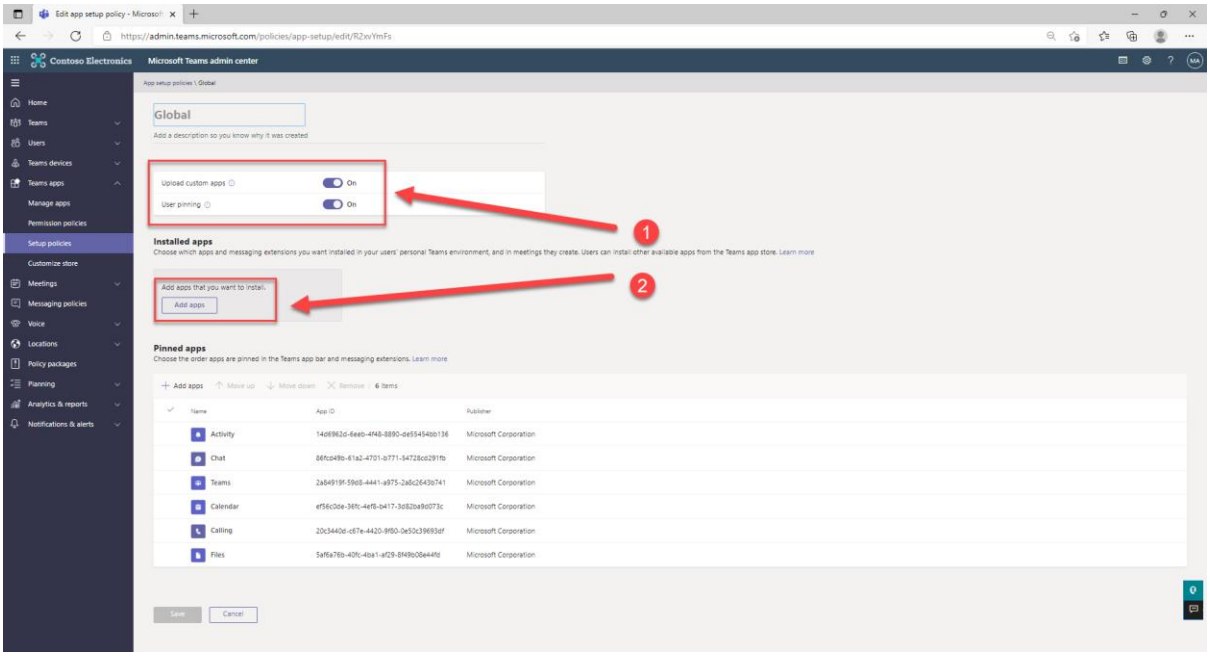
4. Permission policies enable administrators to be more specific on which apps are, or aren't, allowed to be used within Teams. Reviewing the options, you can see that administrators can **Allow all apps**, **Allow specific apps and block all others, Block specific apps and allow all others and Block all apps** for each type of app (**Microsoft apps (1st party), Third-Party apps and custom apps).**
   Administrators can create multiple policies and target them throughout their organisation accordingly, for example; an app permission policies could be targeted at Frontline Workers that enables access only to very specific Third-party apps, but blocks access to Custom apps, whilst the global policy, targeted at everybody else, enables access to all Third-party apps.



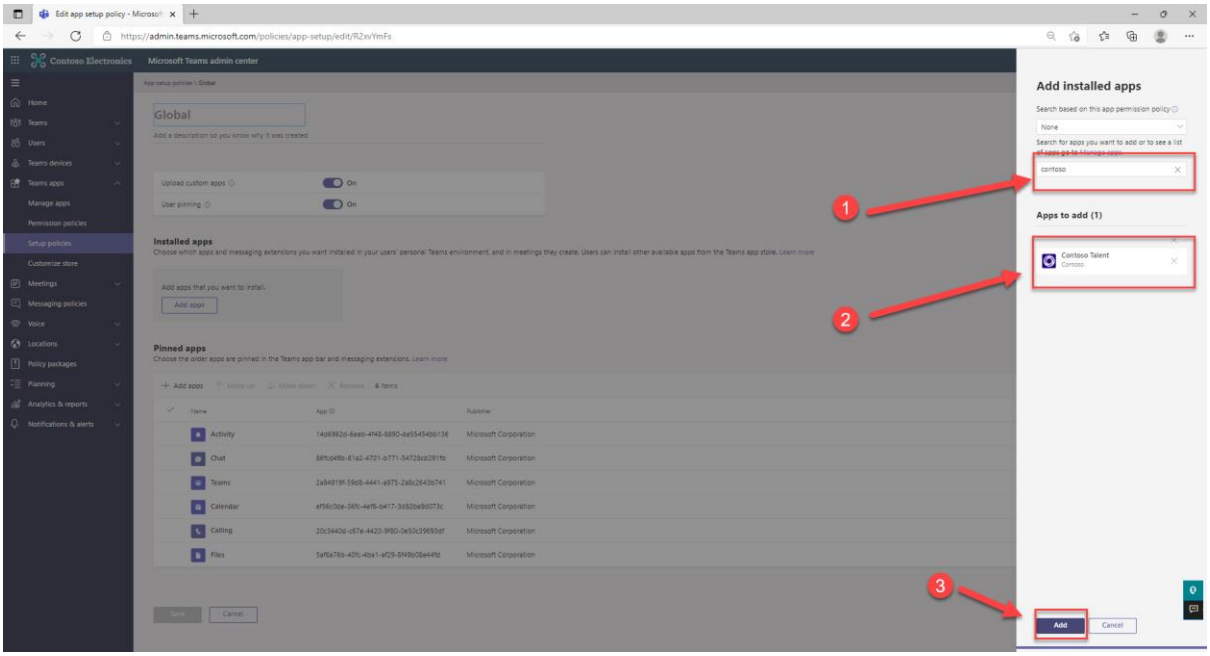
5. Now let's review the App Setup Policies, select **Setup policies** and then select the **Global (Org-wide default)** policy.



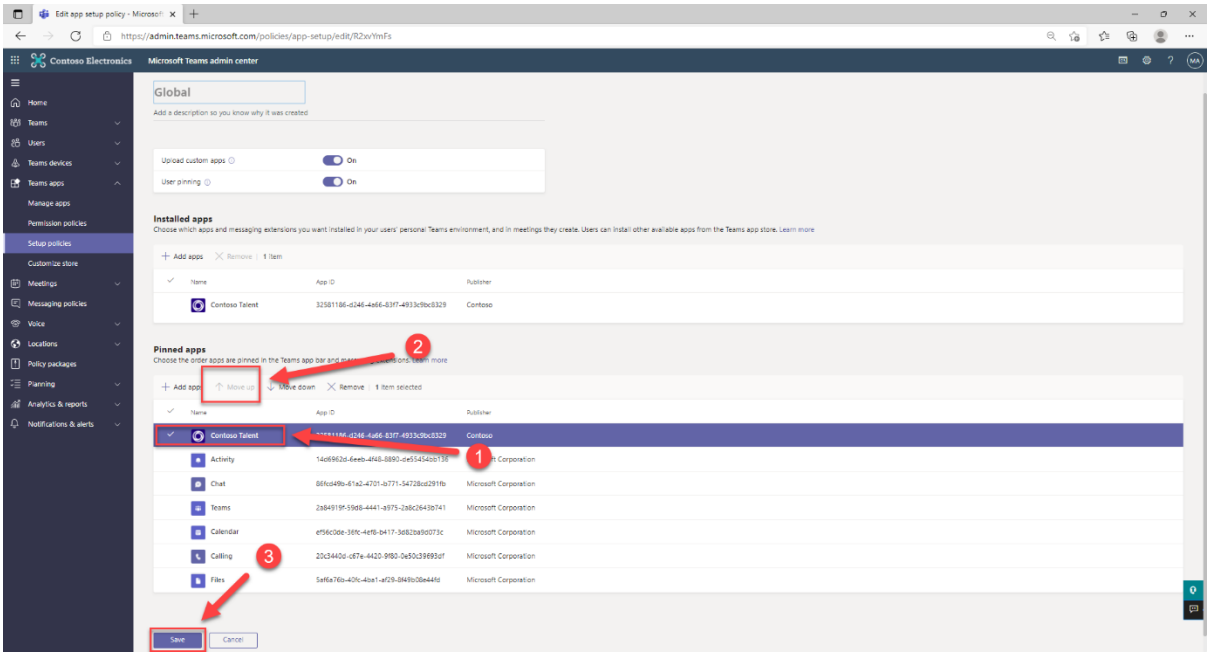
6. Setup policies are very interesting, as they allow administrators to define if the users that are in scope of these policies can **Upload custom apps and pin their own apps**, they allow administrators to **Install apps** for all users in scope of the policy, and perhaps most interestingly, **change the order of the apps that are pinned in the Teams app bar.**
NOTE: Changes to these settings can take anything upto 24hrs to take effect, due to the globally distributed services that enable Teams to function.
Within the Install apps section, select **Add apps**, and let's Install the Contoso Talent management app to all users in this tenant.



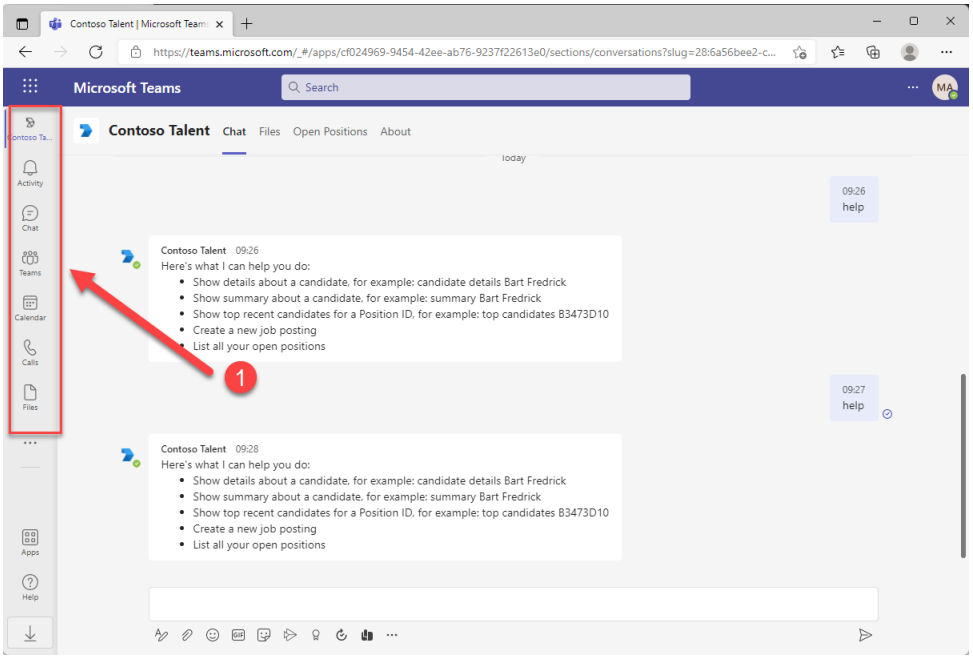
7. Search for **Contoso** and select the **Contoso Talent Management** app. Select **Add.**



8. Now that the app is scheduled to be installed for all users in this tenant, let's make it easier for them to access the Personal App in Teams. Within the **Pinned apps** section, select **Add apps**. Search for and select the **Contoso Talent Management** app, click **Add.**
9. Select the **Contoso Talent Management** app and click **Move up** until it is above the Activity app and is at the top of the list. That will make it easy for users to find the app, and will help drive usage of this app within Teams, therefore making the time spent developing this app worth the effort! Click **Save**!



10. As mentioned previously, changes to the App policies, in the Teams Admin portal, can take an extended amount of time to take effect. Once they have been applied though (try in about 4 hours and it may have been applied), if you log into Teams as a user that was in scope of the App Setup policy, you should see that the **Contoso Talent App is installed and Pinning in the Teams app bar**, above the default first party apps!



Great job! Hopefully this short section of the lab has helped to demystify how administrators can build policies to manage, install, deploy and secure Teams apps throughout their Microsoft 365 Tenants.

10) Universal Actions Adaptive Cards (delivered to both Teams and Outlook simultaneously) – Coming soon!

This section is coming soon! We are working on the code, and documentation steps, to enable this app to send proactive notifications with Adaptive Cards into both Teams and Outlook simultaneously and will push an update into the GitHub repo when it's ready.

Congratulations! You have now successfully completed this lab. We hope that you found this lab, and the associated lab materials useful. We look forward to seeing what Teams Apps you build as a result of attending this lab!

Lab Complete.