# The authentication and authorization flow for integrated Web Apps in Microsoft Teams

netcompany

I work as a Software Consultant at Netcompany Norway - an international company with more than 2000 employees in 6 countries that provides valuable expertise and experience to help organizations with digital transformation. Currently, they are exploring new fields and ways to automate time-consuming processes using Microsoft cloud services in Microsoft Teams (MS Teams). One of them, which this article emphasizes, is to integrate a content page (web app) in MS Teams that can interact with Microsoft Graph API (MS Graph) for generating an Excel file on the user's OneDrive.

## The flexibility of Microsoft Teams and MS Graph API

Microsoft Teams is a collaboration app with 13 million active users daily. A hub for teamwork that combines chat, video meetings, calling, and files into a complete integrated app. What makes MS Teams so unique is the flexibility it provides with integrated web apps.

Organizations today can create content pages (web apps), integrate it with MS Teams, and, communicate with MS Graph API, a Restful API endpoint to interact with Office, OneDrive, SharePoint, Outlook and so on. For instance, a project manager needs to move worked hours for 100 employees from system A to B, one can imagine the amount of time and resources such work takes. The manager must create an excel file, add title and records, and store each file on OneDrive once finished for every employee. This can take several hours perhaps days to complete. With MS Graph API combined with MS Teams, we can automate such a process to take seconds. It supports various endpoints for Excel and many other Microsoft cloud services.

> If you are interested and want to try some endpoints MS Graph offers, check out Microsoft Grap Explorer.

## Outcome

The outcome of this article is to generate an Excel file and store it on the user's OneDrive, for this to happen we must follow these steps:

**Step 1** - Authenticate user **Step 2** - Ged access token **Step 3** - Access MS Graph API

> What is Azure Active Directory (AAD), and why do we need it? To secure our data from unauthorized users, we must ensure two things; they belong to the correct domain (company), and consent to provide access permissions like (User.Read, File.Read).

This article won't cover how to integrate a web application with Microsoft Teams using App Studio. Here's a nice article by Pär Joona that covers it very well.

## Silent Authentication with OAuth 2.0

In this article, we'll use Silent Authentication, an authentication flow for tabs that uses OAuth 2.0. It's recommended to have some basic understanding of OAuth 2.0, here's a good overview. In general, it reduces the number of times a user needs to enter their login credentials by silently refreshing the authentication token (hides the login popup page if the user has already signed in). This creates a pleasant user experience.

## Prerequisites

- Microsoft Teams account
- Office 365 Developer subscription (Required to create apps in MS Teams)
- Register app in App registriation to integrate it with Microsoft identity platform and call MS Graph API
- A deployed web application (We've used Netlify)

The main outcome of this article is to generate an Excel file on the user's OneDrive. For this to happen

> Did you know that App Registration in Azure AD offers developers a simple, secure, and flexible way to sign-in and access Azure resources like Graph API. Additionally, one can grant specified permissions on each user to preserve a secure system from malicious attacks.

## # Authenticate user

This part covers how to authenticate a user, what packages we need to install, the setup process, and how to acquire an id token. Id token is a part of OpenID Connect flow which the client can use to authenticate the user.

> Note: ID Tokens should be used to validate that a user is who they claim to be and get additional useful information about them - it shouldn't be used for authorization in place of an access token.

### 1. Installation

1. Microsoft Teams SDK
2. ADAL.js SDK

### 2. Install NPM packages

To display the content page and communicate with Teams context such as retrieve user details, install Microsoft Teams JavaScript client SDK. Additionally, you also need to install the Azure Active Directory Library SDK to perform authentication operations.

The framework we are using in this example is React with TypeScript to easily create and manage web components, however, if you don't want to use a framework - plain JavaScript works as well. Most examples in Microsoft docs show the authentication process with either plain JavaScript, Angular, or NodeJS.

> Be aware that the underlying authentication flow is generally the same out there, but with few differences depending on the framework.

Let us begin with installing these two packages, open your terminal and run:

```
npm install --save adal-angular
npm install --save @microsoft/teams-js
```

> Note: Currently there is one NPM package providing both the plain JS library (adal.js) and the AngularJS wrapper (adal-angular.js). In short, `adal-angular` works fine with plain JavaScript or TypeScript even though the package name has Angular.

If you are using TypeScript, there is a `@types` package for `adal-angular`. Having types on a package you haven't worked on before is extremely handy, especially when you want to see the abilities/limitations a package offers without needing to open the documentation for every object or method.

```
# Only if your using TypeScript
npm install --save @types/adal-angular
```

### 3. Import packages

Once you've installed the NPM packages, the next step is to import these packages in your JS/TS file. The modules we need to import is `AuthenticationContext` for handling authentication calls to Azure AD, and MS Teams to integrate and display the content page (web app) within the Teams app. Last but not least, if you are using TypeScript, you can go ahead and add the `Option` interface which shows what properties can be added to an object (the example for this will be shown later).

```
import AuthenticationContext, { Options } from 'adal-angular';
import * as microsoftTeams from '@microsoft/teams-js';
```

### 4. Initialize Microsoft Teams

Since MS Teams is showing the content page using `iframe` (a nested browsing context) we need to make sure the user is a part of the MS Teams context before running authentication. It means the authentication process runs only if the content page is opened within MS Teams. This does necessarily mean the app is fully secure but ensures that authentication flow only runs for MS Teams users.

To use MS Teams services, we must initialize it first:

```
microsoftTeams.initialize();
```

This way the content page is displayed in MS Teams through the embedded view context.

### 5. Setup Configuration

The setup configuration is pretty much straightforward. But before we set up the configuration with information present in Azure AD to authenticate a user, we must add the `redirect URI` in the list of redirect URIs for the Azure AD app. Once the user is authentication is successful, Azure AD will check if the `redirect URI` is defined in Azure AD, if it's defined it will add an access token and return it as a query string in which the developer can further use to access MS Graph API. If Azure AD doesn't find the `redirectURI`, it returns a popup page saying: `The reply URL specified in the request does not match the reply URLs configured for the application`.

But how does the authentication flow look when we try to log in a user? Here's a basic example that illustrates the authentication flow:

```
// 1. Login method is executed

// 2. A popup windows is shown: wait for user credentials
login.microsoft.com?clientId=asdfasdfasdf&redirectUri=domain.com/auth/silent-end

// 3. Login successful (only if credentials are correct)

// 3. Azured AD sees the request: if the redirect URI is specified, respond with
id token in the query string
domain.com/idToken=blablabla
```

Now that we have a basic understanding of the authentication flow, and the relevancy of `redirectURI`, the next step is to set up the `config` object. The information is required for Azure AD to authenticate the user, and redirect user in the right domain with the right token. This information such as `clientID` and `tenant` can be found in the overview page in Azure AD.

```
let config: Options = {
    tenant: 'tenant_id', // Can be found in Azure AD
    clientId: 'client_id', // Can be found in Azure AD
    redirectUri: 'URI' + '/auth/silent-end', // Important: URL must be registered
in Redirect URL otherwise it won't work.
    cacheLocation: "localStorage",
    popUp: true, // Set this to true to enable login in a pop-up window instead of
a full page redirect.
    navigateToLoginRequestUrl: false,
};
```

## 6. Create an authentication context

In order to access methods like `login`, `logout`, `getCachedUser`, `getCachedToken`, `aquireToken` and so on, we need to create an `AutenticationContext` and pass `config` object as argument. This establishes a communication bridge between the web app and Azure AD for authenticating users.

```
let authContext = new AuthenticationContext(config);
```

Now every time you use `authContext`, it will perform operations based on what is defined in `config` object. This means if your application interacts with various Azure AD domains, you can setup multiple contexts.

## 7. Login user

Once we have created an `AuthenticationContext(config)`, the next step is to check if the user is cached. Keep in mind that the whole authentication process is done through

`MicrosoftTeams.getContext({...auth process goes here...})` to ensure the auth process only runs within MS Teams.

```
microsoftTeams.getContext((context) => {
    let user = authContext.getCachedUser();
    if (user) {
        // Use is now authenticated
        // Get access_token to access MS Graph API (This part is found in **get
access token** section below)
    } else {
        // Show login popup
        authContext.login();
    }
}
```

As shown above, before we can authenticate the user, we check if the user is cached (already stored in memory). If user is not cached, we invoke the `authContext.login()` method which opens up a popup page that waits for user credentials. Remember to set the popup property to `true` in configs otherwise, the popup page won't show. If the user has already signed in MS Teams, the popup page uses that context to automatically sign in the user. It means the popup page will only be visible within 1-2 seconds.

### 8. Handle cache

Once the user has logged in, to reduce the number of authentication requests to the server we need to cache it. Working with cache, in general, is always a challenge in terms of deciding when to change the old value with the new value. However, `Adal.js` provides an easy and convenient way to handle cache with methods like `authContext.getCachedUser()`, `authContext.getCachedToken()`, and 'authContext.clearCache()'.

So the way we handle cache is by simply checking if the expected user (from MS Teams context) is the same as the cached user (from Azure AD):

```
// Clear cache if the expected user is not the same as the cached user
microsoftTeams.getContext((context) => {

    let user = authContext.getCachedUser();
    if (user.userName !== context.upn) { // upn stands for user principal name,
same as username (based on the Internet standard RFC 822)
        authContext.clearCache();
    }

}
```

As shown in the example above, if the expected user is not the same as the user cached, we clear the cache. Next time the user enters the content page, he must log in again. This step is necessary to keep track of the current user, otherwise, we end up with a conflict between old and new data in the authentication process.

## 9. Get `id_token`

If authentication is successful and the user is cached, Azure AD returns an id token which means the user is authenticated. So whenever we need to access MS Graph API, we check if the id token exists first to make sure the user is still logged-in, and the token hasn't expired.

To get the id token, we wait for an event to be triggered by the user like:

```
return (
    <div>
        <button onClick={ () => generateExcelFile() }>Generate Excel file</button>
    </div>
);
```

When the user clicks on the button to Generate an Excel file, the function `generateExcelFile()` is invoked. This function runs a method `authContext.aquireToken(resource: string, callback: TokenCallback)` which returns 3 arguments in the callback function (`errorDesc`, `token`, and `error`):

```
function generateExcelFile() {
    microsoftTeams.getContext((context) => {
        authContext.acquireToken(config.clientId, function (errorDesc, token,
error) {
            if (error) {
                // Show sign-in button
            }
            else {
                // Get cached access_token
                // Create Excel File
                // Access MS Graph API (to store Excel file on OneDrive)
            }
        });
    })
}
```

As shown above, we check if an `error` is returned which can be a message of type `token renewal has failed` or `token does not exist`. If the error is present, we can show a sign-in button where the user can try to login again. If the error is not present, we can go ahead and get the access token, create an Excel file, and then store it on the user's OneDrive using MS Graph API.

## # Get access token

Once the user is authenticated, the next step is to authorize the user. At first, these two words seem synonyms, but must not be mixed. Authentication means confirming the user's identity wheres authorization means being allowed to access the domain. In other words, to allow a the content page to interact with MS Graph API, for instance, generate an excel file or get user details, the user must authorize it. This is done by showing a popup page with a list of permissions the user can consent to or cancel.

**Application or Delegate**

There are two permission types (Application or Delegate), or ways to get an access token. If a daemon service wants to send emails on a weekly-basis (user authentication is not required) this is known as Application request. If actions are based on user events (user authentication is required) this is known as Delegate requests. Admin can set permissions depending if the requests come from a daemon or user in Azure AD.

```
# User based
https://login.microsoftonline.com/${context.tid}/oauth2/v2.0/authorize?
{queryParams}

# Application based
https://login.microsoftonline.com/${context.tid}/oauth2/v2.0/token?{queryParams}
```

As mentioned, to open the gateway to interact with MS Graph API we need an `access_token`. To get the access token, we must navigate to `authorizeEndpoint` with the query parameters defined in `queryParams` object. The authorization endpoint also needs a tenant id which can be found in the MS graph context or the config object defined above.

```
 let queryParams = {
    client_id: config.clientId,
    response_type: "token",
    response_mode: "fragment",
    scope: "https://graph.microsoft.com/User.Read openid",
    redirect_uri: window.location.origin + '/auth/autherization',
    prompt: 'login',
    nonce: 1234,
    state: 5687,
    login_hint: context.loginHint,
};

let authorizeEndpoint =
`https://login.microsoftonline.com/${context.tid}/oauth2/v2.0/authorize?
${toQueryString(queryParams)}`;
window.location.assign(authorizeEndpoint);
```

Once we navigate to the authorization endpoint, Azure AD will check if the query parameters are correct, and most importantly if the user is authenticated. If those criteria are true, it will return an `access_token` as a query string on the URL path. Since we are working with a single page application like React, we need a way to capture this `access_token`. The simplest way is to create a component that runs once the `authorizationEndpoint` is triggered.

**Setup routing for the authorization endpoint**

To set up routing for the authorization endpoint, we use the `react-router-dom` library. Then we define an exact path `/auth/authorization` which redirects to the `Authorization` component.

```
import React from 'react';
import Home from './Home';
import { BrowserRouter as Router, Route } from 'react-router-dom';
import Authorization from './Authorization';

const Navigation = () => {
    return (
        <Router>
            <div>
                <Route path="/" exact component={Home}></Route>
                <Route path="/auth/autherization" exact component={Authorization}>
</Route>
            </div>
        </Router>
    );
}

export default Navigation;
```

**Capture access token and redirect back to the homepage**

Once the redirect path is triggered, we do two things: cache the `access_token`, and then redirect back to the home page.

```
import React from 'react';
import { useHistory } from 'react-router-dom'
import qs from 'querystring';

const Authorization = (props: any) => {
    let history = useHistory();
    let querystring = qs.parse(props.location.hash);
    let access_token = querystring['#access_token'].toString();
    window.localStorage.setItem('access_token', access_token);
    history.push('/'); // redirect back to home page
    return (
        <div>
            <h1>This component is only used to cache the access token, and then
redirect back to home page</h1>
        </div>
    );
}

export default Authorization;
```

> Note: If the authorization process fails, you can show an error message here. But make sure you put the redirect code inside a conditional statement.

# Acess MS Graph API

With successful user authentication and an access token, this leads us to the final step where we can use MS Graph API to access data in Azure AD, Office 365 services, Office 365, Enterprise Mobility, Security services, Windows 10 services, and more. For this example, we'll generate an excel file, and store it on the user's OneDrive.

The library we use to create an excel file is Sheetjs, install it by following these steps here.

```
function generateExcelFile() {
    microsoftTeams.getContext((context) => {
        authContext.acquireToken(config.clientId, function (errorDesc, token,
error) {
            if (error) {
                //  Show a sign in button
            }
            else {
                // Get cached access_token
                let access_token = window.localStorage.getItem('access_token');

                if (access_token) {
                    // Basic fetch api request
                    const headers = new Headers();
                    headers.append('Authorization', `Bearer ${access_token}`);
                    headers.append('Content-Type',
'application/vnd.openxmlformats-officedocument.spreadsheetml.sheet');

                    var graphEndpoint =
"https://graph.microsoft.com/v1.0/drive/root:/mellomprosjekt/file1.xlsx:/content";

                    // Create excel file
                    const book = XLSX.utils.book_new();
                    const sheet = XLSX.utils.aoa_to_sheet(['data1', 'data2']);
                    XLSX.utils.book_append_sheet(book, sheet, 'sheet1');
                    let workBook = XLSX.write(book, { bookType: 'xlsx', type:
'array' }); // type must be array otherwise it will be corrupt in OneDrive

                    console.log('workBook', workBook);
                    let init: RequestInit = {
                        method: 'PUT',
                        headers: headers,
                        body: workBook,
                    };

                    // Run request
                    fetch(graphEndpoint, init)
                        .then(resp => resp.json())
                        .then(data => console.log(data))
                }
            }
        });
    })
}
```

###Side note According to the documentation and examples, it seems that it's possible to get access token just by using the Adal.js SDK library. That would be the easiest and most practical way of doing so by utilizing the library. For my example, I did not manage to get the access token but instead got the id token. The issue with the id token is the `aud` property (identifies the intended recipient) is set to client id, but MS Graph API expects it to be `graph.microsoft.com` which the access token satisfies.

If you look at the access token retrieved from the authorization request the `aud` property is `graph.microsoft.com` which is correct. This ways of getting first the id token and then the access token feels a bit hacky as shown in the Get access token section. However, I'm sure there are better ways of doing so.

Here's an example which uses the convenient approach (it did not work for me).

```
authContext.aquireToken('https://graph.microsoft.com', (errorDesc, token, error)
=> {})
```

I have tried many ways but could not get the access token and instead got id token. To access MS Graph API, an access token is required. After a couple of trials, I went with an approach that works, but I'm sure there are better ways of doing it. The best ways are of course to the access token when running `aquireToken` method.

# Summary

We have now covered how to authenticate a user, authorize the user to get the access token, use access token to access MS Graph API, and last but not least, generate an Excel file on user's OneDrive. To make the authentication and authorization flow work, remember to wrap the flow inside the MS Teams context otherwise the content page won't be displayed. If we put the code outside the MS Teams context, the code will still work, but not shown in MS Teams.

Here's a high-level overview of what you need to make the content page (web app) work in MS Teams.

```
microsoftTeams.initialize();
microsoftTeams.getContext(context => {
    // This is where we perform the authentication and authorization flow
});
```

Authentication in Microsoft or any other languages is always a challenge thus there are many ways to do it and concerns to be aware of in terms of security. In either way, it's recommended to perform security tests and perhaps contact those that are have done it before. In general, authentication can never be 100 % secure, but we can try our best to make it as secure as possible following good guidelines provided by Microsoft.

## Resources

- Register an application in Azure AD
- Create content page in MS Teams
- Authentication Flow for Tabs
- Silent Authentication

- Microsoft Teams Sample Auth Node