# Generating a word (of PDF) document from SharePoint List data using Azure Functions.

## Overview

This post explains an azure function that can populate a word template stored in a document library with information passed into it in the post body. The azure function can save the generated document back to another SharePoint document library. The function impersonates the logged in user to ensure that the function only can write to libraries the caller has access to. It can also create a PDF version of the file using the new v2.0 SharePoint rest interface.

A sample SPFX list view command set is also included to demonstrate how to call the azure function passing in data from multiple lists in a site.

The caller of the azure function must pass in the following json structure via a POST:

public class PostData

{

public PlainTextReplacementParameters[] plainTextParameters { get; set; }

public string templateServerRelativeUrl { get; set; }

public string destinationFolderServerRelativeUrl { get; set; }

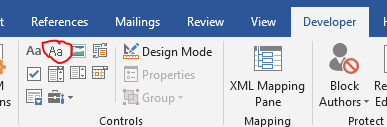
public string temporaryFolderServerRelativeUrl { get; set; }

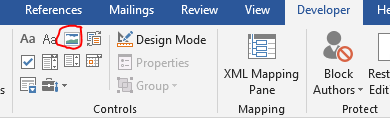
public string webServerRelativeUrl { get; set; }

public string fileName { get; set; }

public string saveAsFormat { get; set; }

}

The templateServerRelativeUrl parameter is the server relative URL of the docx file that has the word template to be populated. Word Content controls need to be put into the document wherever substitution should take place. The solution currently supports the Plain Text Content Control: 

and the Image content control: . Checkboxes can be handled by passing in the checkbox characters (☐ or ☑) as a string and placing them in a Plain Text Content Control.

The destinationFolderServerRelativeUrl parameter is the URL of the folder where the generated pdf or docx file should be placed.

The temporaryFolderServerRelativeUrl parameter is the URL of a folder used to store temporary files. If a PDF file is requested, a word document will be created and stored in the temporary folder, then a pdf will be generated from that word document and stored in the destinationFolderServerRelativeUrl.

The webServerRelativeUrl parameter is the relative URL of the web where the above three libraries are located. (they must be in the same web and the caller must have write access to them).

The filename parameter is the name the generated file will be given (without an extension).

The saveAsFormat can be either PDF or DOCX.

The PlainTextReplacementParameters[] parameter is an array of objects defined as this:

public class PlainTextReplacementParameters

{

public string token { get; set; }

public string value { get; set; }

public string replacementType { get; set; } // PlainText or Image (for now)

}

The token parameter specifies the name of a token in the template document that needs to be replaced.

The value parameter specifies what the token should be replaced with.

The replacementType parameter can be ‘PlainText’ or ‘Image’. If it’s ‘PlainText’, then the token (which must be the name of a Plain Text Content Control in the template) is replaced with the text stored in the value parameter. If the replacementType parameter is ‘Image’ then the token (which must be the name of a Picture Content Control in the template) will be replaced with the image whose URL is stored in the value parameter ().

## The Azure Function

The following settings need to be set up for the Azure Function:

"HostName": "tenant.sharepoint.com", // used to construct sharepoint urls

"ClientId": "{your client id here}",

"ClientSecret": "{your client secret here}",

"TenantId": "{your tenant id here}"

The Azure function has two endpoints—GenerateDocument and GetPDFPreviewUrl. Both return a json object defined as:

public class ResponseBody

{

public string url { get; set; }

public List<string> messages { get; set; }

}

If the client specifies PDF as the saveAsFormat for GenerateDocument, the endpoint will first create a word document in the temporaryFolderServerRelativeUrl folder, then get a pdf version of it and store that in the destinationFolderServerRelativeUrl folder. The full URL of the PDF file in the destinationFolderServerRelativeUrl is returned to the caller in the ResponseBody. If the caller specifies DOCX as the saveAsFormat for GenerateDocument, the endpoint will create the word document and store that in the destinationFolderServerRelativeUrl and return the URL to that document in the ResponseBody. The temporaryFolderServerRelativeUrl parameter can be omitted if generating DOCX files.

If the client calls the GetPDFPreviewUrl the saveAsFormat is ignored and can be omitted. If the caller specifies PDF as the saveAsFormat for GenerateDocument, the endpoint will first create a word document in the temporaryFolderServerRelativeUrl folder, the endpoint will then generate the URL to open that document as a PDF and return that in the response body. Note that this URL is only valid for a short time and is only returned so that the client can immediately display the pdf at that URL to the user.

At a high level, the code first gets an access token to talk to SharePoint on behalf of the user. It then saves that token in the Client Context (so that it can talk to SharePoint via CSOM), and in an HTTPClient (so that it can take to SharePoint via rest to get pdf versions of generated files).

After the authentication work is done, it downloads the template file from SharePoint and saves it to a temporary directory on the azure function’s server. It opens that file and then spins through each of the replacement parameters replacing the Content Controls with the actual content. The local file is then saved and closed.

If the client requested a DOCX file to be created the code just uploads that file to the destinationFolderServerRelativeUrl and returns the URL to the client.

If the client requested a PDF file to be created the code uploads the file temporary directory on the azure function’s server to the temporaryFolderServerRelativeUrl. It then downloads that file in pdf format and then saves the pdf to a temporary directory on the azure function’s server. It then uploads that pdf file to the destinationFolderServerRelativeUrl and returns the URL to the client.

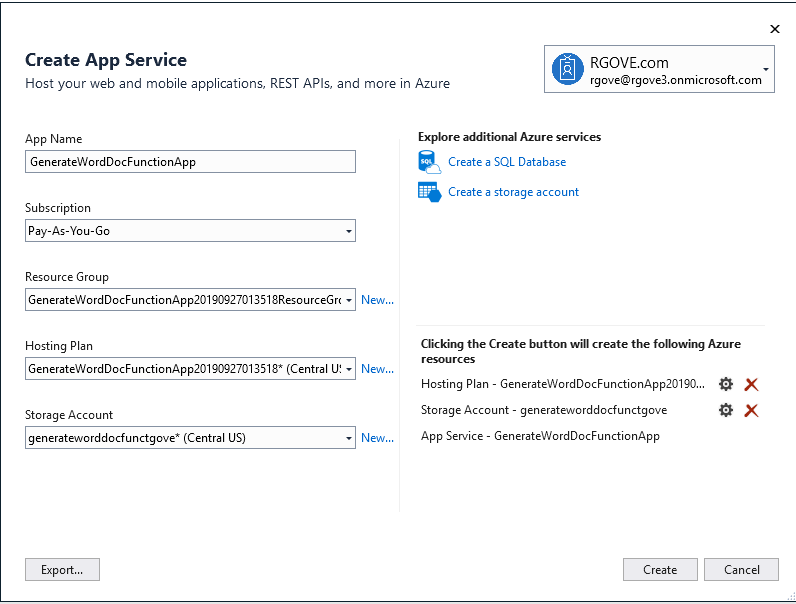
## The sample SPFX client

The above azure function can be used in many scenarios. The sample just adds three commands to a list view command set -- Preview PDF, Generate DOCX and Generate PDF.

## Installation

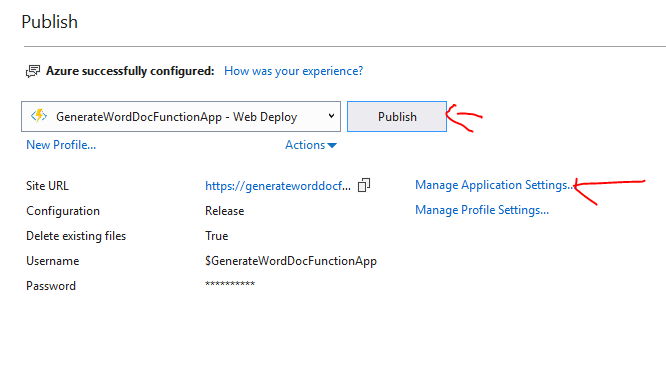
### Azure Function Installation

Publish the Azure function by right clicking on the Project and selecting Publish. First Create a Publishing Profile:

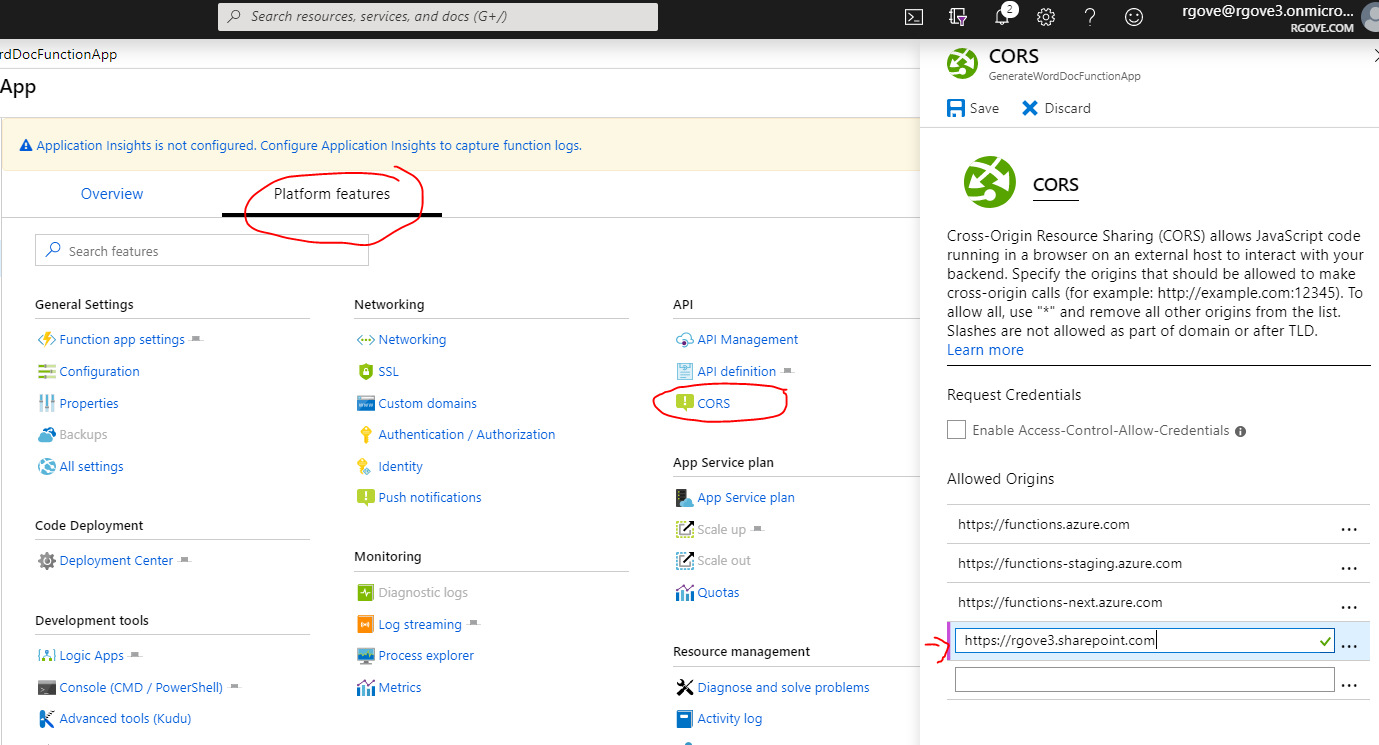


Enter values in the dialog box appropriate for your environment.

Then click Publish to actually publish the function:

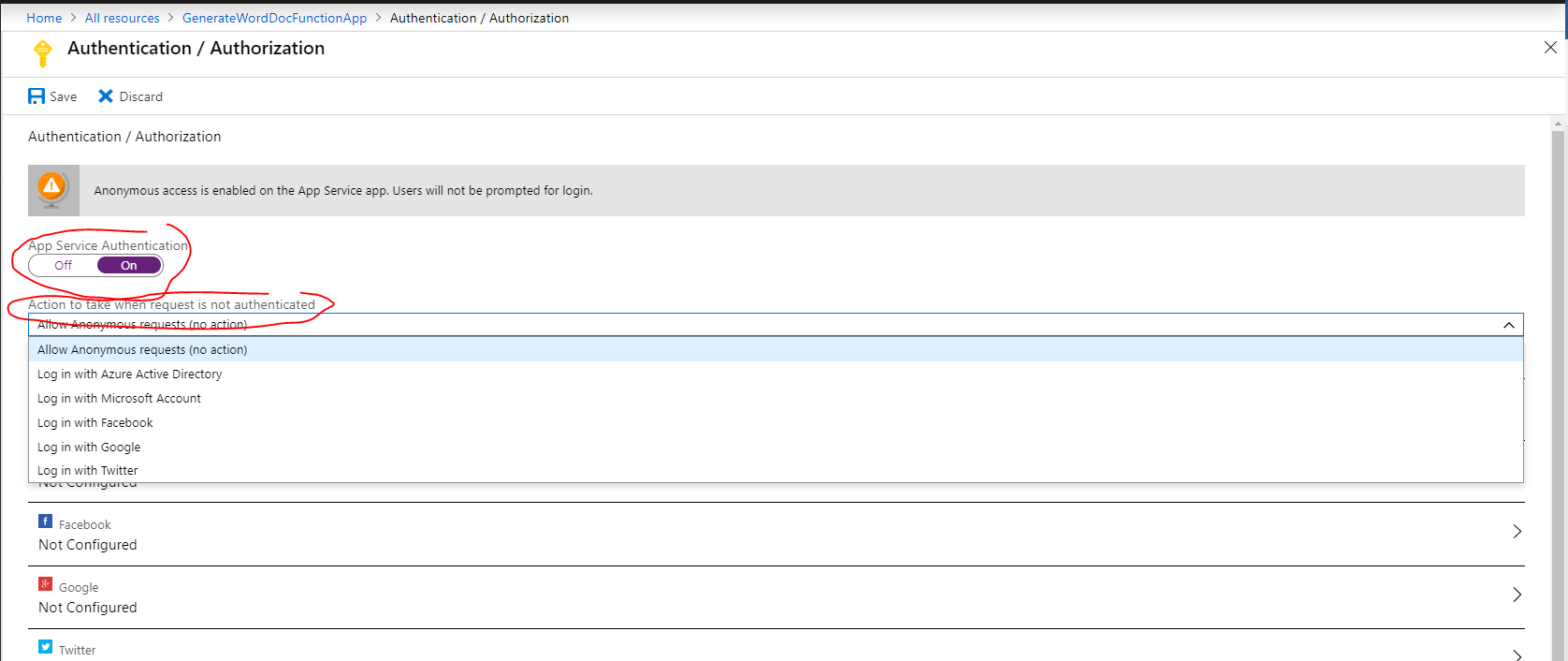


Navigate to portal.azure.com and open your function. Click on Platform Features ->CORS then enter the URL to your SharePoint tenant

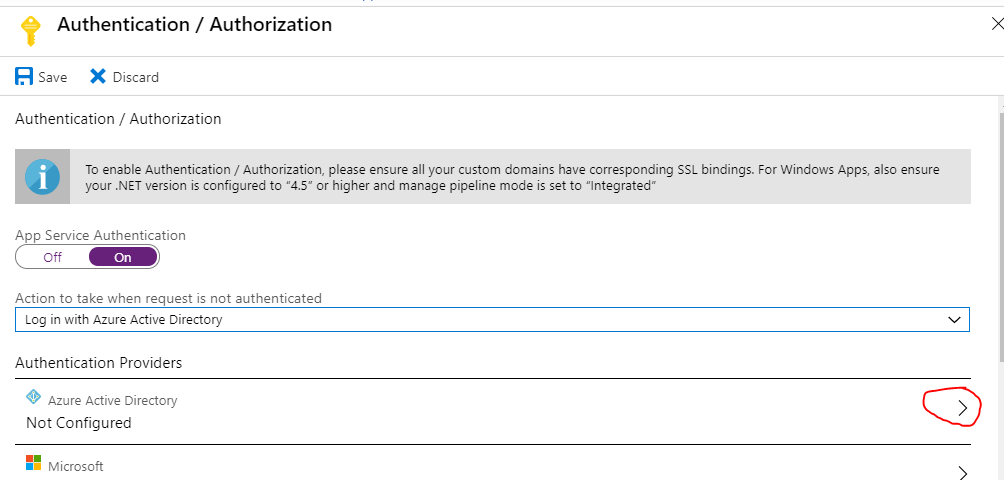


Click Save to save your CORS settings.

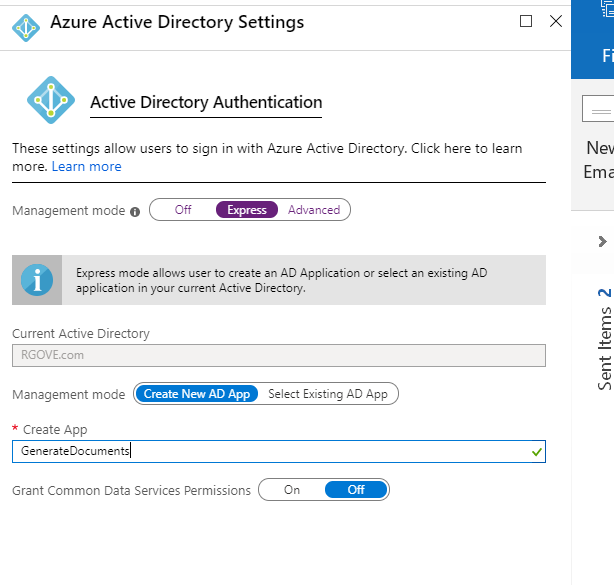
Click on Authentication/Authorization and turn on App Service Authentication. Then in the dropdown labeled ‘Action to take when request is not authenticated’ select ‘Log in with Azure Active Directory’:



Click the arrow adjacent to the Azure Active Directory Authentication provider to configure it:

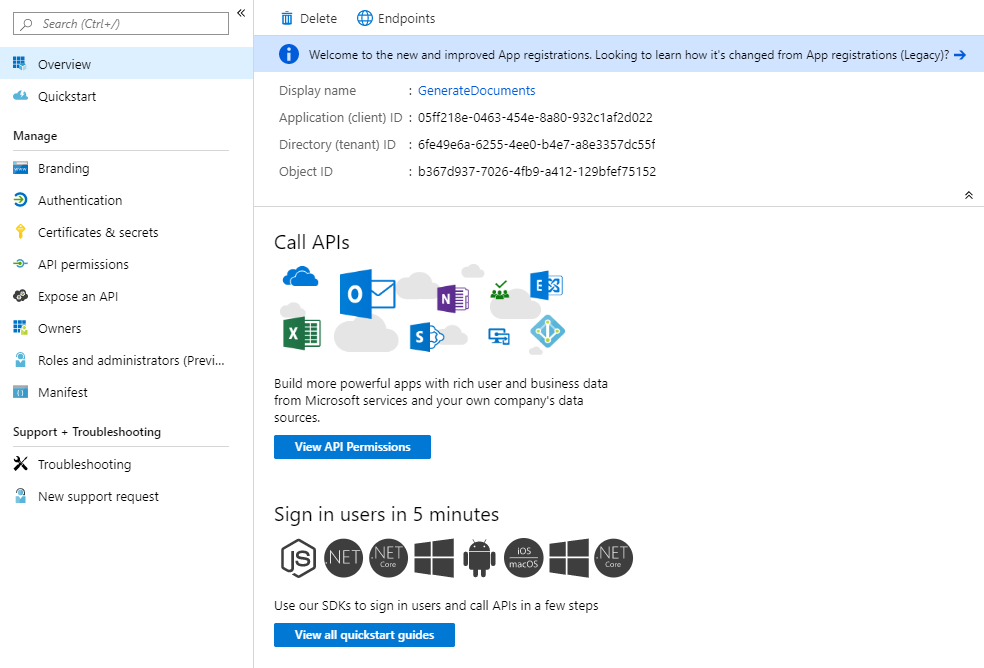


Select ‘Express’ management mode and ‘Create new AD App’. Give the app an appropriate name such as GenerateDocuments:

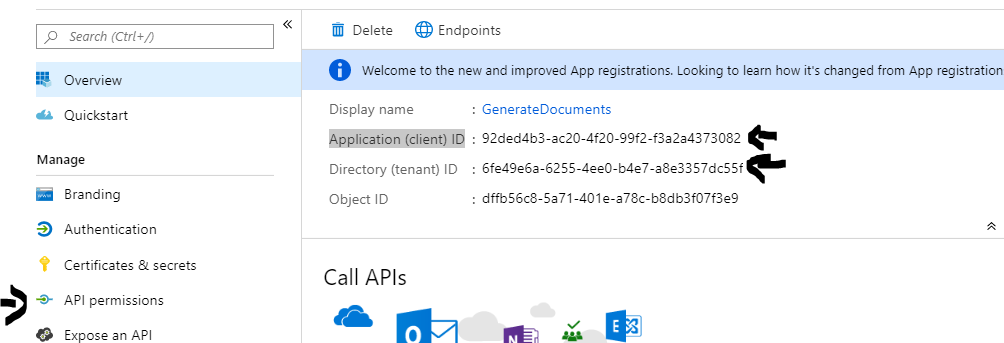


Save your changes and make a note of the Application name as we will need that later.

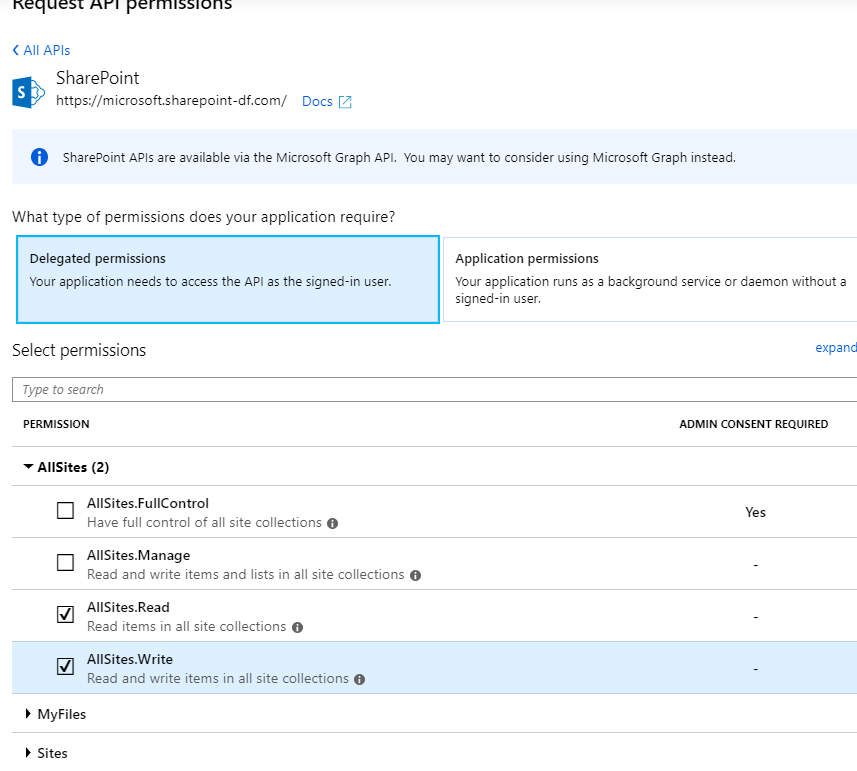
Next navigate to Azure Active Directory-> App Registrations and open the app we just created (GenerateDocuments):



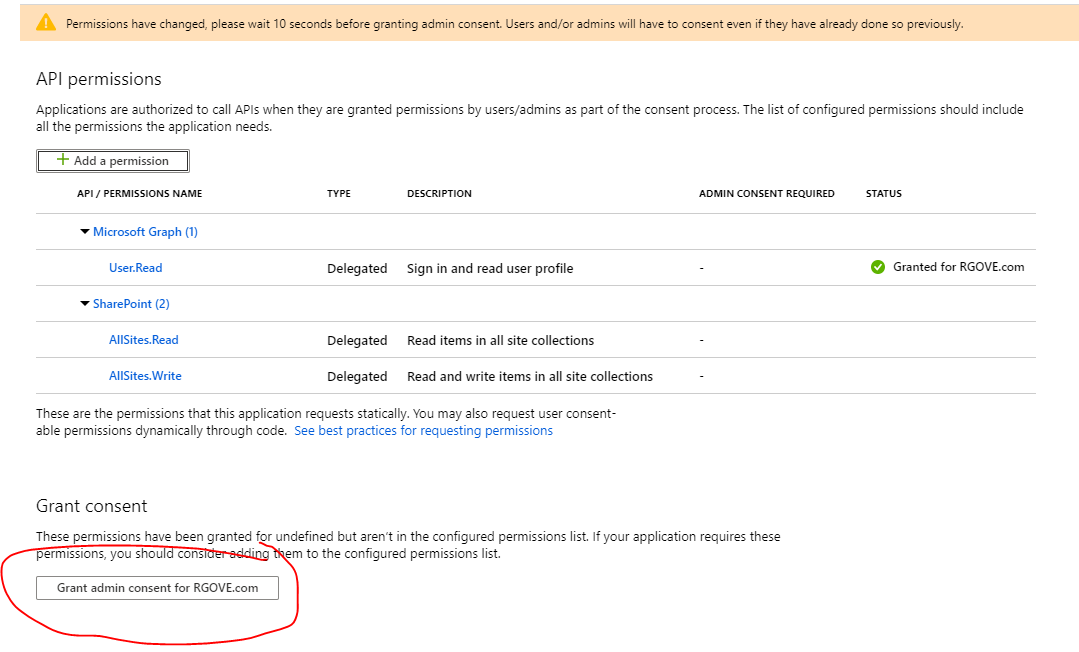
Make a note of the Application (client) ID and Directory (tenant)



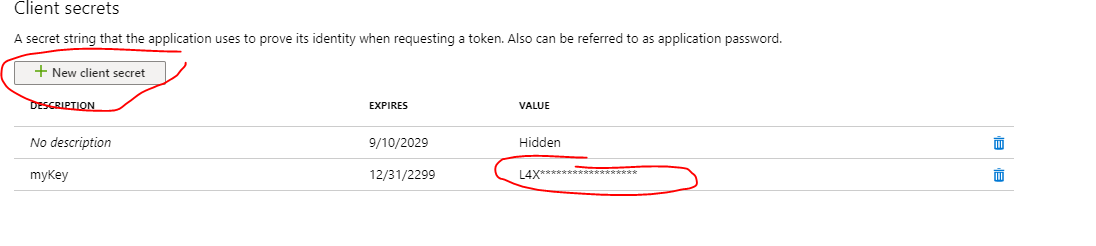
Open the API Permissions blade, and click the Add a permissions button and give the app Delegated permissions to read and write to SharePoint sites on behalf of the user:



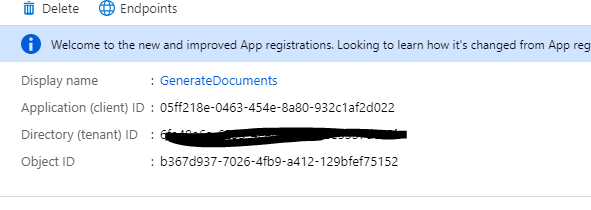
After Saving the permissions click on Grant admin consent for tenant



Open the Certificates and secrets blade generate a new Client secret and make a note of it: m+.+p74M4CGDr9\_PT0e+b]4ynQ\*pG5@G

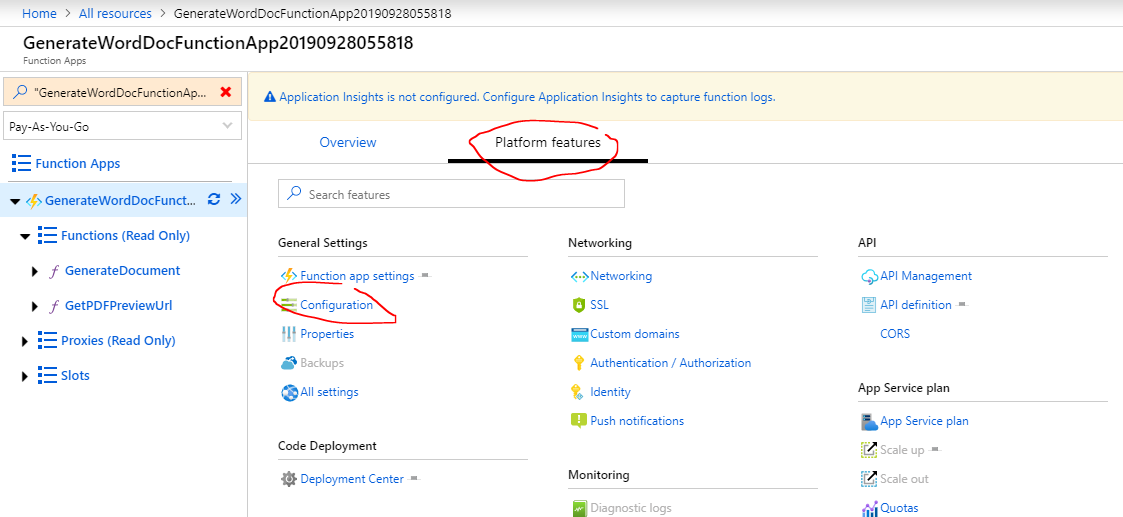


Open the Overview blade make a note of your Application (client ID) and Directory (tenant) ID:

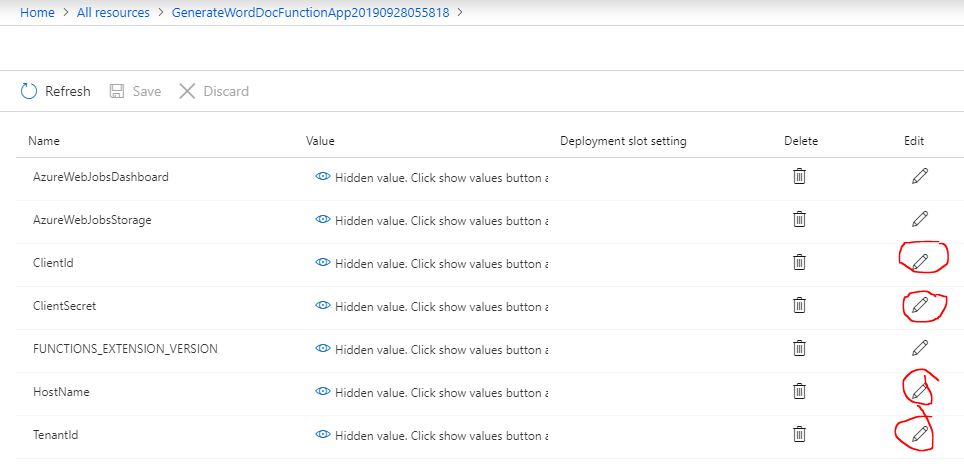


Now we need to go back to the azure function and enter these values in the function Configuration.

Navigate back to your function app and select Platform features, and then click Configuration:



Click the buttons to edit the ClientID, ClientSecret, TenantId and HostName:



Enter the values you recorded when setting up the app registration. For the hostname, just enter {tenantname}.sharepoint .com.

### List View Command Set Installation

Now that we’ve completed the installation of the Azure function, we need to install the Sample SPFX application. But before we do that , we need to crate the document library the list view command set will work with.

On the site you want to test with, create three document libraries called Templates, Temporary and Published.

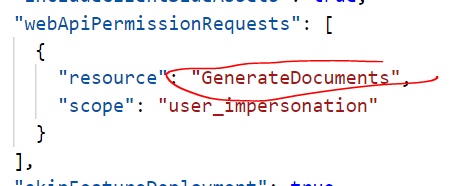
Upload the file testTemplate.docx (included in the SPFX project) to the templates library.

Also create a custom (generic) List called Tasks, and Add a new column called IsComplete (A choice column with two Choices ‘Yes’, and ‘No’). be sure the list allows attachments. Add an item or two to the list and attach a .jpg file to each.

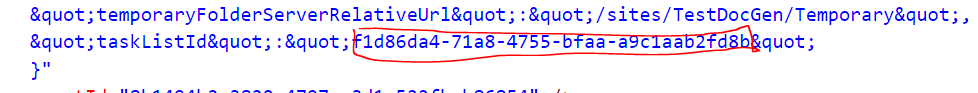
Next get the list Id of the task list, as we will need this to configure the List view command set.

Now open the react-command-generate-documents project in VS Code.

If you changed the name of the App Registration to something other than GenerateDocuments you will need to edit the webApiPermissionRequests in the package-solution.json file to reflect the name you used:



Next, in both the ClientSideInstance.xml and the elements.xml files in the SharePoint/Assets folder replace the list Id with the list Id of the tasks list you created above:



Also change the azureFunctionBaseUrl to the URL of your azure function. and change the templateServerrelativeUrl, webServerrelativeUrl and temporaryFolderServerRelativeUrl to point to your site.

After that build the solution and Upload it to the site collection app catalog.

After the solution is installed, go to site contents and click the link to Add an App. Select react-command-generate-documents-client-side-solution to install an instance of the app.

Note: as an alternative to adding the app in the previous step, the following PowerShell can be used:

connect-pnponline -Url https://tenant.sharepoint.com/sites/yoursite

Get-PnPCustomAction

Add-PnPCustomAction -Name "CreateDocument" `

-ClientSideComponentId "8b1494b2-3820-4797-a3d1-522fbab86854" `

-ClientSideComponentProperties `

"{`"azureFunctionBaseUrl`":`"https://generateworddocfunctionapp20190928055818.azurewebsites.net`",

`"azureFunctionGenerateDocumentMethod`":`"GenerateDocument`",

`"azureFunctionGetPDFPreviewUrlMethod`":`"GetPDFPreviewUrl`",

`"templateServerRelativeUrl`":`"/sites/TestDocGen/Templates/testTemplate.docx`",

`"destinationFolderServerRelativeUrl`":`"/sites/TestDocGen/Published`",

`"webServerRelativeUrl`":`"/sites/TestDocGen`",

`"saveAsFormat`":`"PDF`",

`"temporaryFolderServerRelativeUrl`":`"/sites/TestDocGen/Temporary`",

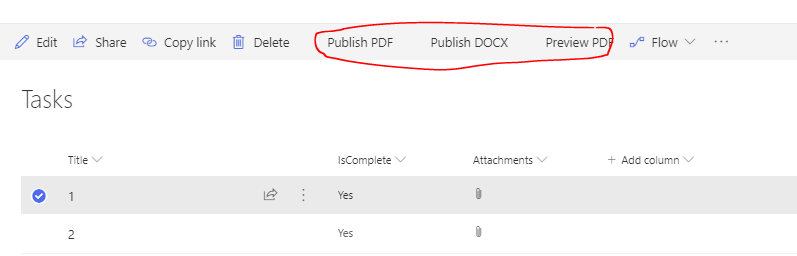
`"taskListId`":`"f1d86da4-71a8-4755-bfaa-a9c1aab2fd8b`"}" -Title "GenerateDocuments" -Location "ClientSideExtension.ListViewCommandSet.CommandBar" -RegistrationId 100 -RegistrationType List -Scope Web

Be sure to change the taslListId to the ID of the task list in your site and change all the URLs to the URL of your site and the azureFunctionBaseUrl to the URL of your azure function.

Last step is to go to your tenant-admin.sharepoint.com site switch to the modern UI, click the API Management link in the left nav and approve and approve the request.

## Testing

With the installation complete, you can now test the app. Navigate to the ‘tasks’ list in your site and select an item or two. The ribbon should show the commands as shown below:



## Notes

1. The replacement parameters in this sample came from a single list. They can come from many different lists (or anywhere you want to get them from in your SPFX solution)
2. The libraries that hold the published documents can be in a completely different site than the lists containing the data.
3. The list view command set could used different templates or even send the published documents to different sites, depending on the data.