WPF Dev Camp - Module 5

Securing Service with Azure Active Directory

Lab version: 1.0

Last updated: 1/13/2015

Estimated demo delivery time: **10 to 15 minutes**.



Contents

[Overview 3](#_Toc408548772)

[Prerequisites 3](#_Toc408548773)

[Setup 3](#_Toc408548774)

[Exercise 1: Securing Service with Azure Active Directory 4](#_Toc408548775)

[Task 1: Creating a Directory 4](#_Toc408548776)

[Task 2: Creating a Demo User 5](#_Toc408548777)

[Task 3: Creating a Windows Azure Active Directory Service Application 8](#_Toc408548778)

[Task 4: Securing Expenses Service with Azure Active Directory 10](#_Toc408548779)

[Task 5: Securing Expenses Client with Azure Active Directory 15](#_Toc408548780)

# Overview

In this demo, we will show how to use Azure Active Directory to help secure the Expenses service.

# Prerequisites

The following are required to complete this demo:

* [Microsoft Visual Studio 2013](http://www.visualstudio.com/en-us/downloads/download-visual-studio-vs.aspx) (tested with Update 4)
* Internet connection
* [Microsoft Azure](http://azure.microsoft.com/en-us/pricing/free-trial/) subscription
* Expenses codebase

# Setup

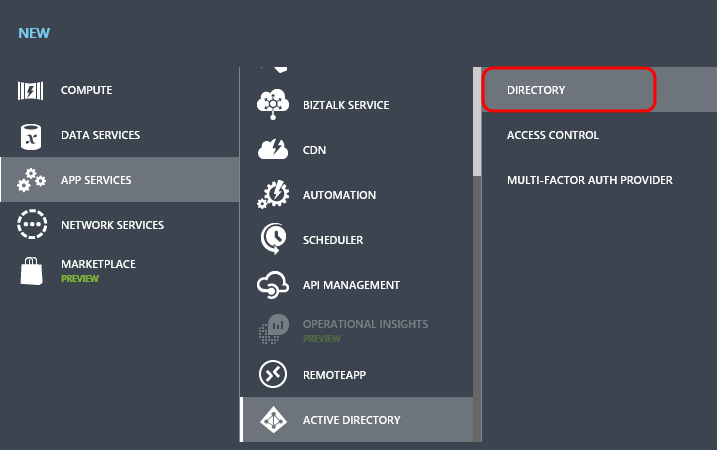
* **Use a fresh copy of the starting Expenses codebase for this demo in a NEW LOCATION** (where we run the service and database locally). Note that the fresh copy is to be put in a NEW LOCATION as to avoid overwriting the progress made during the “Moving Service to Azure” demo. Although it is certainly possible to continue where previous demos left off, this allows this demo to be delivered free of unnecessary pre-requisites and moving parts.
* Load and build the Expenses solution to ensure that it builds correctly.
* Log into the Microsoft Azure subscription that you will be using for demonstration.

# Exercise 1: Securing Service with Azure Active Directory

In this exercise, we will setup and configure Azure Active Directory for the purposes of securing the Expenses WCF service.

## Task 1: Creating a Directory

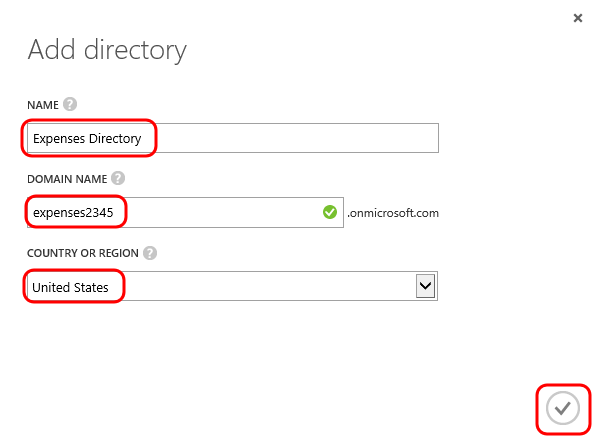
1. Log into the full Microsoft Azure portal [here](https://manage.windowsazure.com) (or if you are currently in the Preview portal at portal.azure.com you can click on your user name in the top-right corner and select the Full Azure Portal link).
2. Click New the new button in the bottom-left corner and then select App Services | Active Directory | Directory.



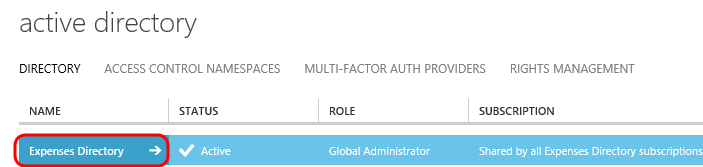
1. Click Custom Create.



1. In the Add Directory window, provide a Name, Domain Name, and Country or Region. Click the Complete button to create the directory.



1. Navigate to the new directory in the portal.

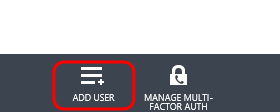


## Task 2: Creating a Demo User

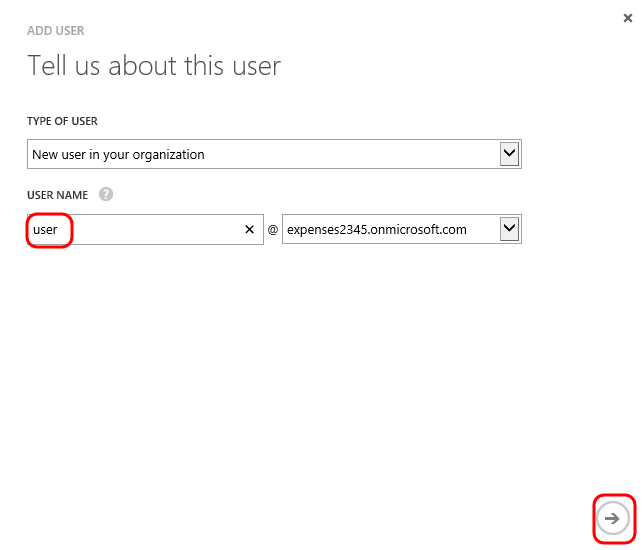
1. Select the Users tab.



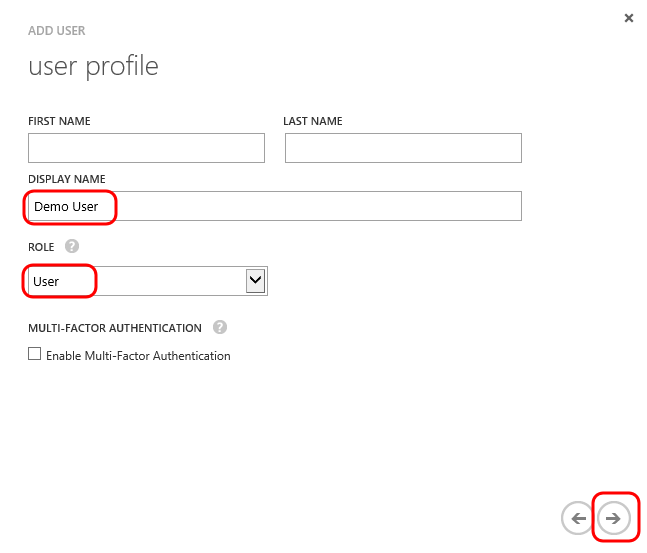
1. Click the Add User button at the bottom.



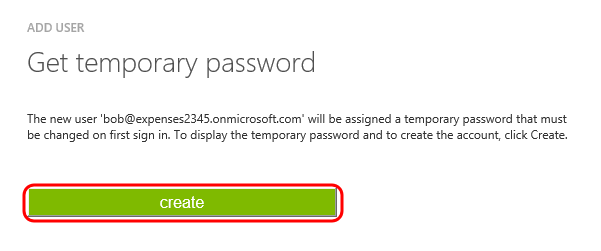
1. Use the default for the Type of User option, which is “New user in your organization”.
2. Create a user in the directory named “user” for demonstration purposes.
3. Click the Next button.



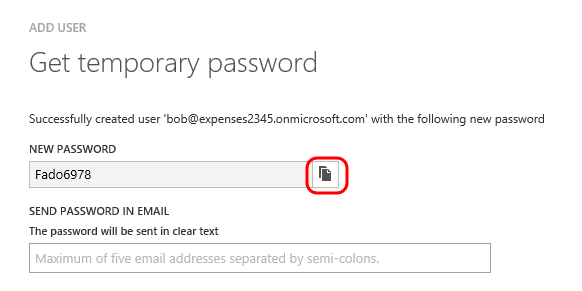
1. Provide a display name, use the default role of User, and then click the Next button.



1. Click the Create button to generate a temporary password.



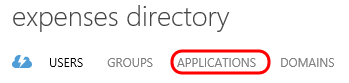
1. Copy the temporary password to Notepad or other location so that we can use it to login with.



1. Click the Complete button.

## Task 3: Creating a Windows Azure Active Directory Service Application

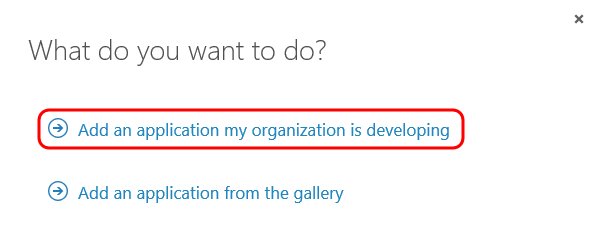
1. Click the Applications tab.



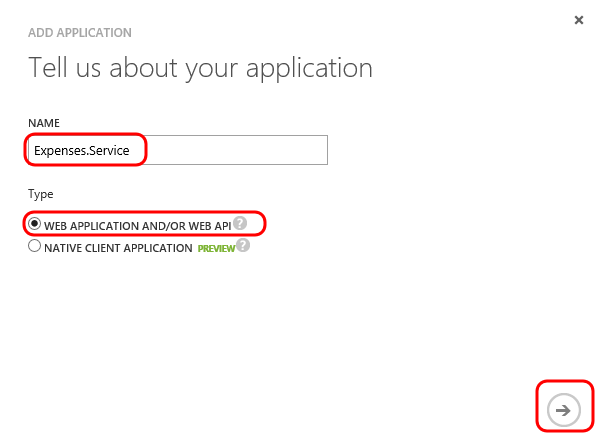
1. Click the Add button near the bottom.



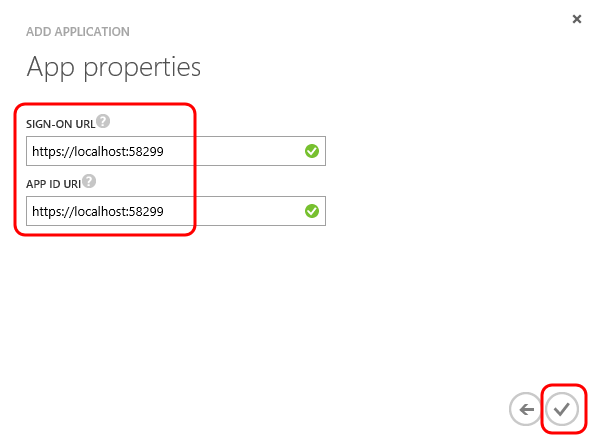
1. Click the “Add an application my organization is developing” link.



1. Enter “Expenses.Service” for the name and select “Web Application and/or Web API” for the type. Click the Next button to continue.

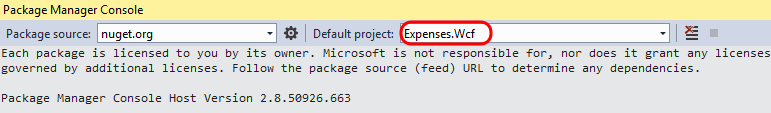


1. Since we are in the development phase, provide a Sign-On URL of <https://localhost:58299>.
2. For the App ID URI, provide a unique logical identifier. This does not need to resolve to an Internet address, the only requirement is that it is in valid URI format.
3. Click the Complete button.



## Task 4: Securing Expenses Service with Azure Active Directory

1. Open the Expenses.sln solution in Visual Studio.
2. Select Tools | NuGet Package Manager | Package Manager Console from the main menu.
3. In the Package Manager Console window, ensure that the Expenses.Wcf project is selected.



1. Use the following command to install the Active Directory Authentication Library:

Install-Package Microsoft.IdentityModel.Clients.ActiveDirectory -Version 2.12.111071459

1. Use the following command to install the needed OWIN components:

Install-Package Microsoft.Owin.Security.ActiveDirectory -Version 3.0.0

1. Use the following command to install the bits necessary to enable OWIN-based applications to run on IIS using the ASP.NET request pipeline:

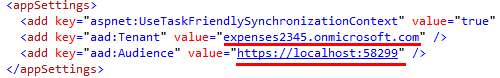
Install-Package Microsoft.Owin.Host.SystemWeb -Version 3.0.0

1. Open the Web.config file in the editor and add the following app settings keys:

<add key="aad:Tenant" value="[Enter tenant name, e.g. expenses.onmicrosoft.com]" />

<add key="aad:Audience" value="[Enter App ID URI of service]" />

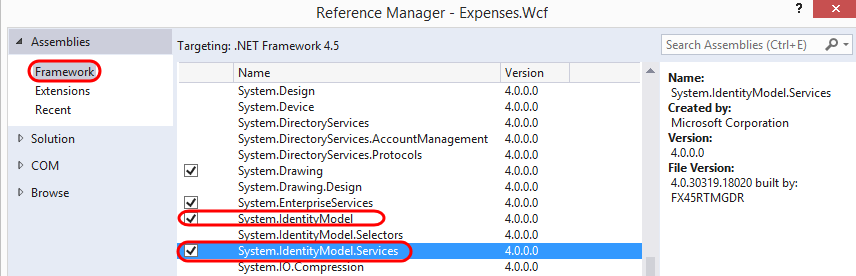
1. Replace the placeholder values with your tenant name and App ID URI value.



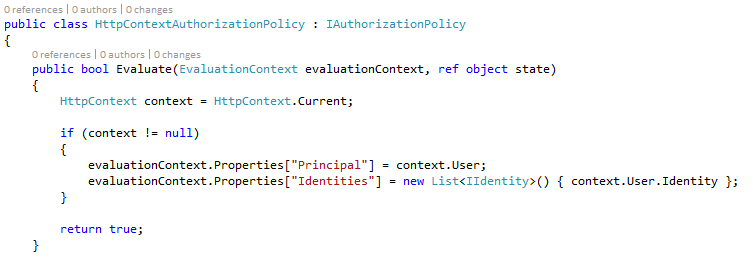
1. The next step is to configure the usage of Azure Active Directory bearer authentication. The client will send bearer tokens to the service to authenticate each request. Right-click on the Expenses.Wcf project node in Solution Explorer and select Add | Existing Item.
2. Open the Startup.cs file from the DemoFiles folder.
3. Open Startup.cs file in the code editor. This file will perform the necessary configuration to process the bearer tokens send with each request. Note that it has an assembly level attribute applied to it for OwinStartup.



1. Add the Startup.cs file to the root folder of the project so that this configuration will take place on startup.
2. We would like to be able to perform some claims based authentication in our service, so we can implement IAuthorizationPolicy to help with this. All WCF requests will be evaluated by this code once we get it configured in Web.config. Add the HttpContextAuthorizationPolicy.cs file from the DemoFiles folder to the project.
3. Now we need to add some additional references. Right-click on the References node for the service project and select Add Reference. Add System.IdentityModel and System.IdentityModel.Services.



1. Open HttpContextAuthorizationPolicy.cs in the editor. Note that Evaluate method will look at the current HTTP context and add Principal and Identities properties to the current evaluation context.



1. Add the following to Web.config to configure the custom authorization policy that we just added to the project. This is a behavior that should go in the <system.ServiceModel><behaviors><serviceBehaviors><behavior> section.

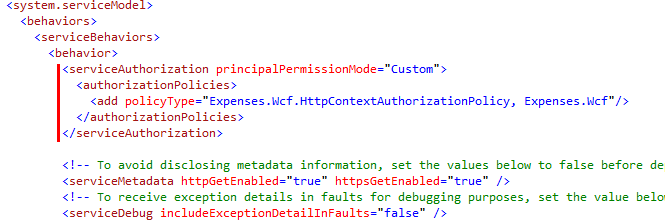
<serviceAuthorization principalPermissionMode="Custom">

<authorizationPolicies>

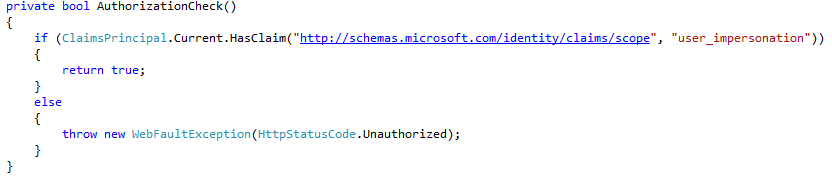
<add policyType="Expenses.Wcf.HttpContextAuthorizationPolicy, Expenses.Wcf"/>

</authorizationPolicies>

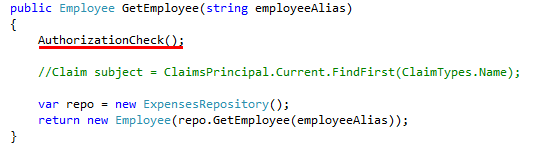
</serviceAuthorization>



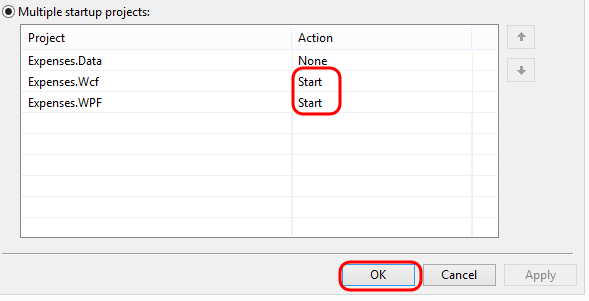
1. Now we just need to modify the service logic itself to check for authorization. Overwrite the ExpenseService.svc.cs file with the one found in the DemoFiles folder (easiest to do this using Explorer window).
2. Double-click on ExpenseService.svc in Solution Explorer to load the code in the editor.
3. The difference here is that we now have an AuthorizationCheck method in place that checks the claims of the current principal, specifically that the user has a scope value of “user\_impersonation”, or full access to the service. If the current principal does not have that claim, the proper unauthorized error code will be returned.



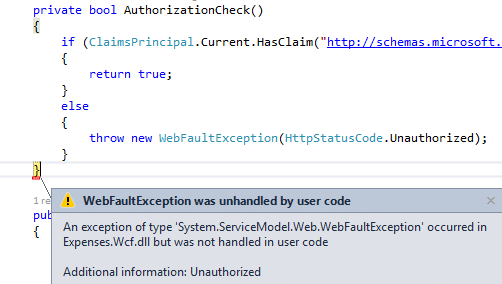
1. Take a quick look at the GetEmployee method. It now makes a call to AuthorizationCheck as its first step. Also note the line of commented code that shows how to get access to user metadata such as Name if desired. This could be used by the Expenses API in the future to perhaps modify returned results.



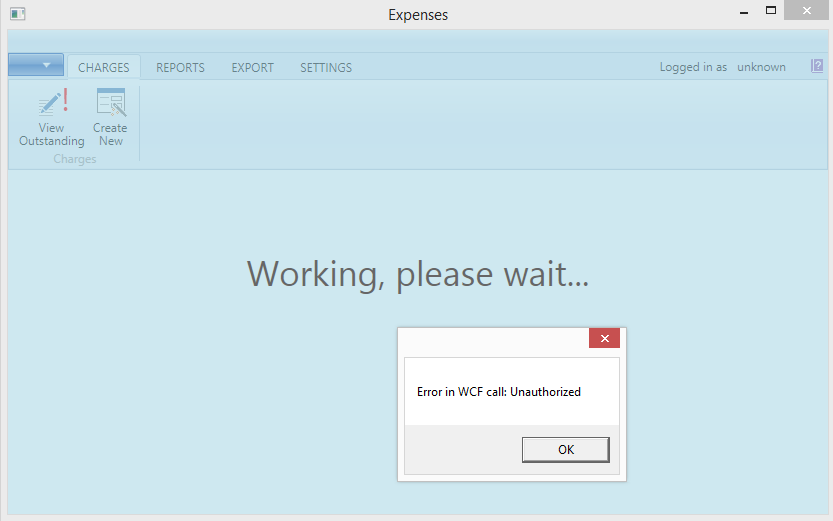
1. Now let’s test the application to ensure that the Expenses service is now enforcing authentication as expected. Right-click on the solution node in Solution Explorer and select Set StartUp Projects.
2. Ensure that “Multiple startup projects” option is selected and that Expenses.Wcf and Expenses.WPF projects are set to Start. Click OK.



1. Press F5 to launch the application. Before the Expenses client fully loads, the debugger should break within the authorization check in the WCF service where the WebFaultException is thrown. Press F5 to continue.



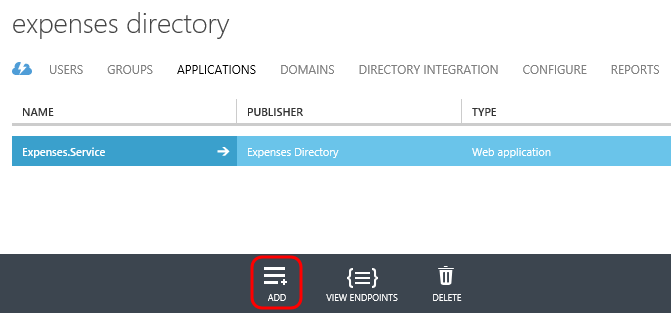
1. This error will make its way back to the client and a dialog indicating that the call was unauthorized should be shown (it may not show up on in the foreground, so check the taskbar).

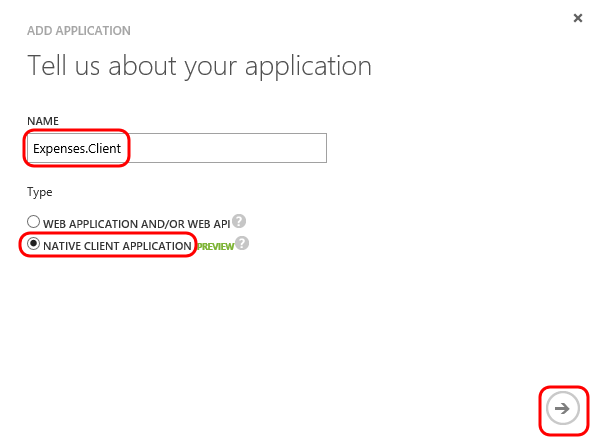


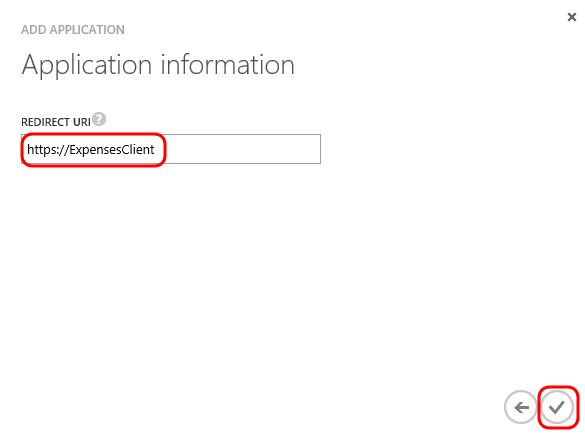
1. Stop debugging the application and service.

## Task 5: Securing Expenses Client with Azure Active Directory

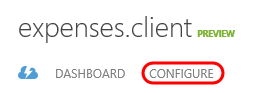
1. Now we can tackle updating the Expenses client in order to acquire and then send the needed bearer token in requests that are made to the service.
2. Return to the Applications tab for your demonstration Active Directory in the Azure portal, then click the Add button.



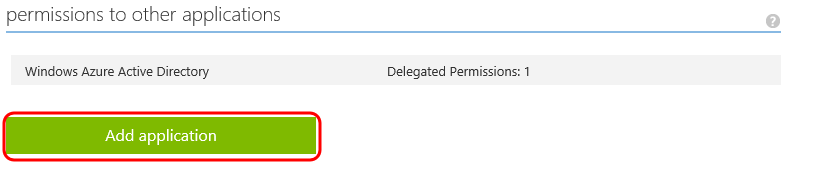
1. Click the “Add an application my organization is developing” link.
2. Enter “Expenses.Client” for the name and select “Native Client Application” for the type. Click the Next button to continue.  
   
3. Enter <https://ExpensesClient> for the Redirect URI. This does not need to be a valid endpoint, but does need to be a valid URI.
4. Click the Complete button.



1. Now we need to configure the Expenses.Client application so that it has access to the Expenses.Service application. Click the Configure tab.



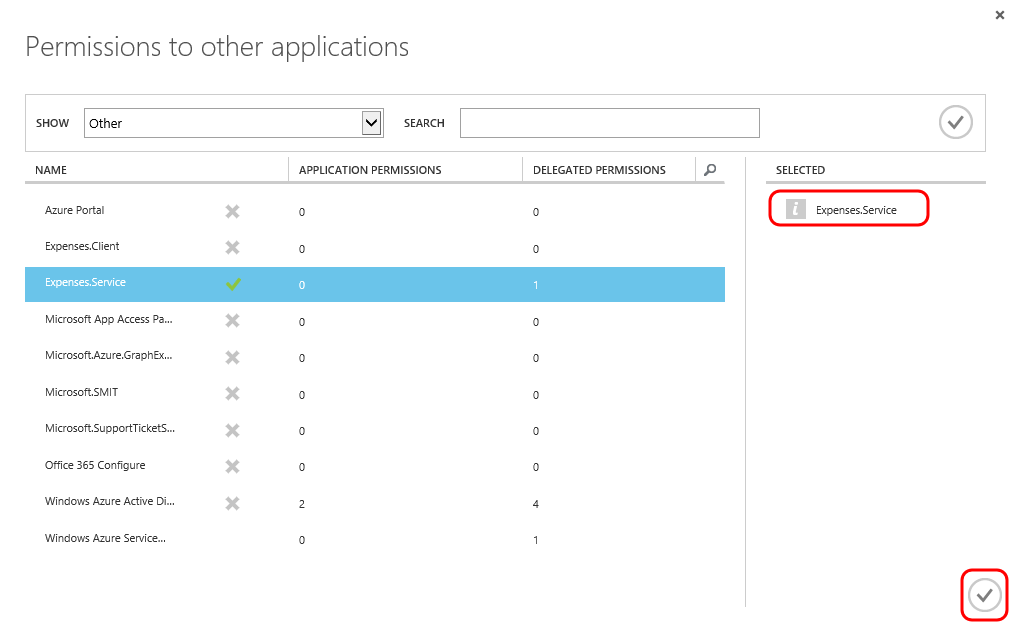
1. Scroll down to the Permissions to Other Applications section and then click Add Application.



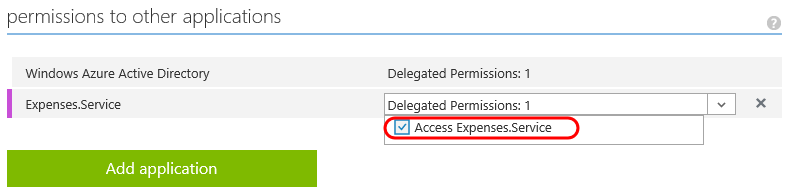
1. In the Permissions to Other Applications window, click the Show drop-down and then select Other. Click the Search button.



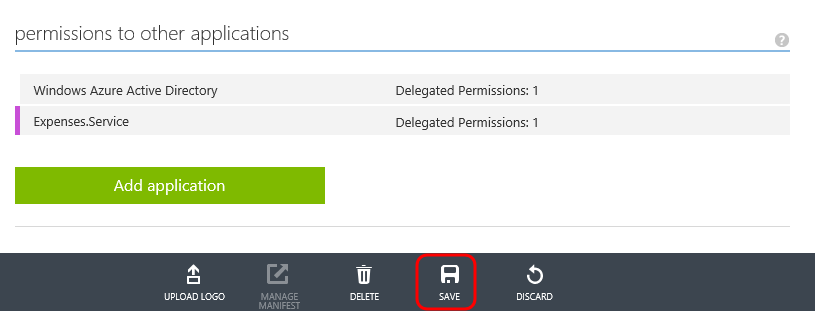
1. Click the ‘+’ button to the right of Expenses.Service to add it to the selected list and then click the Complete button.



1. Click the Delegated Permissions drop-down box and select the “Access Expenses.Service” option.



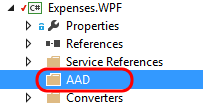
1. Click the Save button at the bottom.



1. In the Package Manager Console window, ensure that the Expenses.WPF project is selected.
2. Use the following command to install the Active Directory Authentication Library (ADAL):

Install-Package Microsoft.IdentityModel.Clients.ActiveDirectory -Version 2.12.111071459

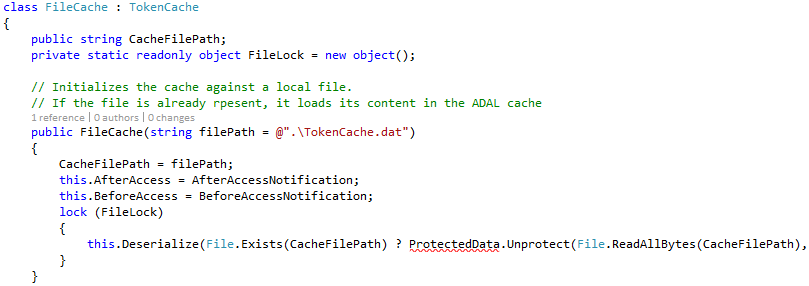
1. With the ADAL package added to the project, we can now use the library to acquire the necessary token from Azure Active Directory to be able to authenticate with the Expenses service. Start by creating a new folder in the WPF project named “AAD”.



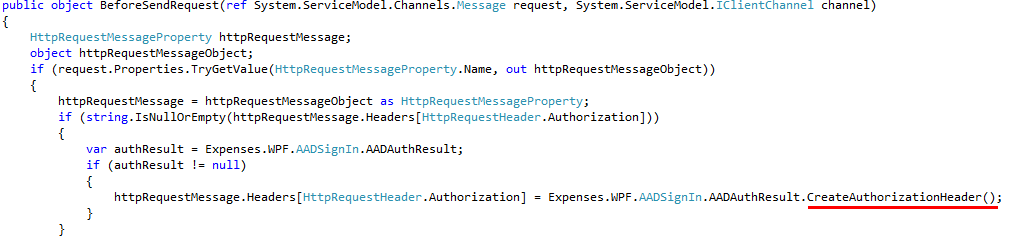
1. Add the following files from the DemoFiles folder to this new AAD folder:
   * AADSignIn.cs
   * ExpensesClientMessageInspector.cs
   * FileCache.cs
2. Open AADSignIn.cs in the editor. This class was created to perform the sign in process using ADAL. It first creates an AuthenticationContext and then it calls the AcquireToken method with the appropriate settings to request a token from your tenant settings. Once the authentication result is determined, it is then stored for later used when making calls to the Expenses WCF service.



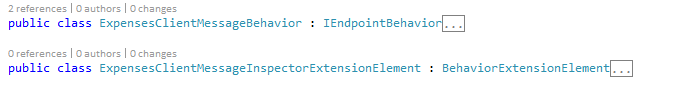
1. Note that ConfigurationManger is reported to not exist in the current context. Add a project reference to System.Configuration to fix that.
2. Open FileCache.cs in the editor. This class inherits from the TokenCache class from the ADAL and is responsible for persisting tokens in a local file protected using the Data Protection API (DPAPI). This functionality is used by the AuthenticationContext that we just looked at in the AADSignIn class.



1. Note that ProtectedData is reported to not exist in the current context. Add a project reference to System.Security to fix that.
2. Open ExpensesClientMessageInspector.cs in the editor. This class implements a WCF message inspector (by implementing IClientMessageInspector) for the purposes of adding in the appropriate bearer authorization header to outgoing service requests. The ADAL is utilized once again here in order to create the appropriate authorization header with a call to CreateAuthorizationHeader.



1. There are a couple of more classes in the ExpensesClientMessageInspector.cs file that allow this message inspector to be specified in configuration.



1. Now we need to make a call to the SignIn method that we added when the WPF application starts up. Open App.xaml.cs in the editor and add the following line of code to the start of the OnStartup method.

AADSignIn.SignIn();

1. The final step is to add the expected configuration to App.config. Start by adding in the following application settings section with the expected keys:

<appSettings>

<add key="aad:Tenant" value="[Enter tenant name, e.g. contoso.onmicrosoft.com]" />

<add key="aad:ClientId" value="[Enter client ID as obtained from Azure Portal, e.g. 82692da5-a86f-44c9-9d53-2f88d52b478b]" />

<add key="aad:RedirectUri" value="[Enter redirect URI as entered in Azure Portal, e.g. https://ContosoClient]" />

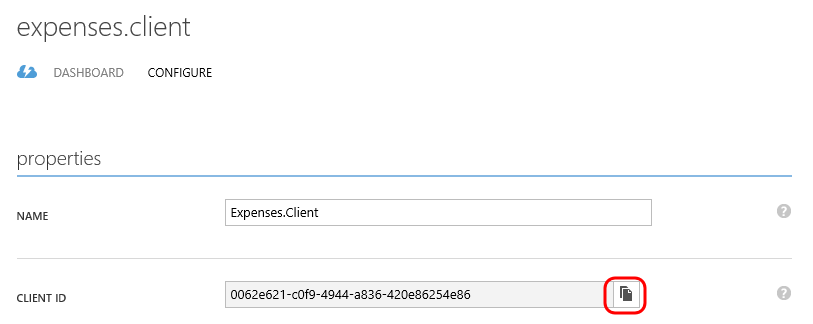
<add key="aad:ResourceId" value="[Enter App ID URI of service, e.g. https://contoso.onmicrosoft.com/ContosoService]" />

<!-- AADInstance: leave default in place (https://login.windows.net/{0}) -->

<add key="aad:AADInstance" value="https://login.windows.net/{0}" />

</appSettings>

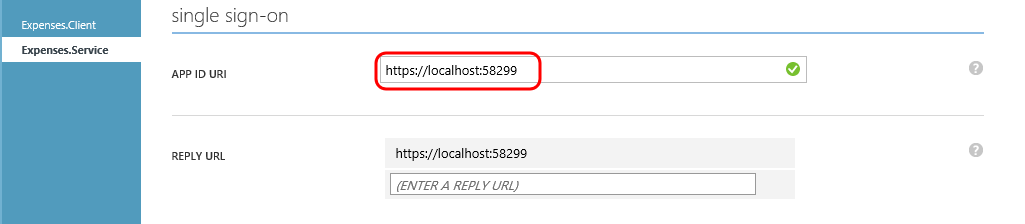
1. Replace the value for aad:Tenant with your tenant name. This will look something like contoso.onmicrosoft.com.
2. Replace the value for aad:ClientId with the value found on the Configure tab in the portal.



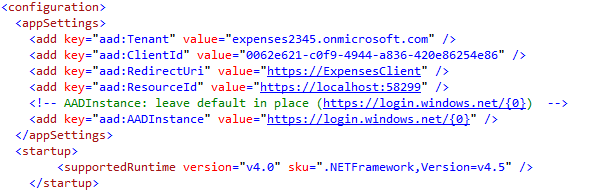
1. Replace the value for aad:RedirectUri with the value you used when creating the Expenses.Client application in your directory. You can also find this on the Configure tab in the portal.



1. Replace the value for aad:ResourceId with the App ID URI of the Expenses.Service application.



1. Leave the default value for the aad:AADInstance setting in place. Your application settings should now look similar to the following screenshot (although your specifics may vary):



1. Now we need to configure the WCF behavior extension to load our ExpensesClientMessageInspector so that outgoing requests will include the necessary bearer token. Copy the following configuration and add it to the <system.serviceModel> section.

<extensions>

<behaviorExtensions>

<add name="bearerTokenRequired" type="Expenses.WPF.ExpensesClientMessageInspectorExtensionElement, Expenses.WPF, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null"/>

</behaviorExtensions>

</extensions>

<behaviors>

<endpointBehaviors>

<behavior name="bearerTokenRequiredBehavior">

<bearerTokenRequired />

</behavior>

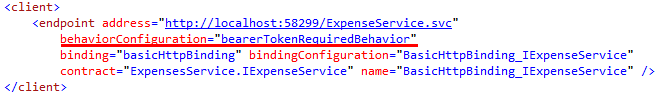
</endpointBehaviors>

</behaviors>

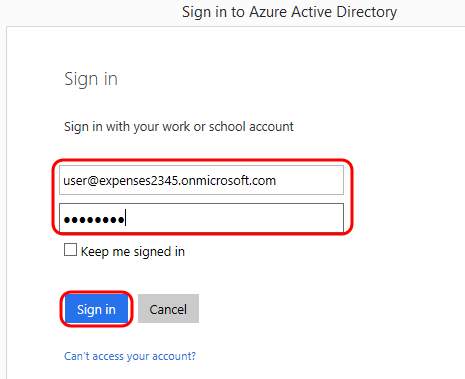
1. Finally add the following property to the client endpoint definition in the configuration file so that the endpoint uses the bearer token behavior:

behaviorConfiguration="bearerTokenRequiredBehavior"

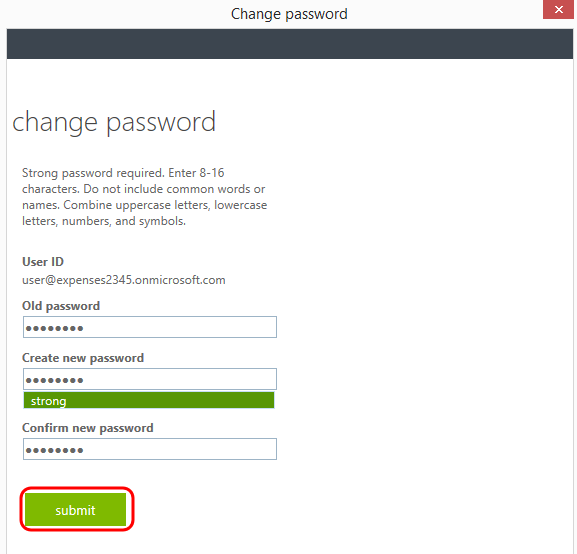
1. The client endpoint should now look like the following:



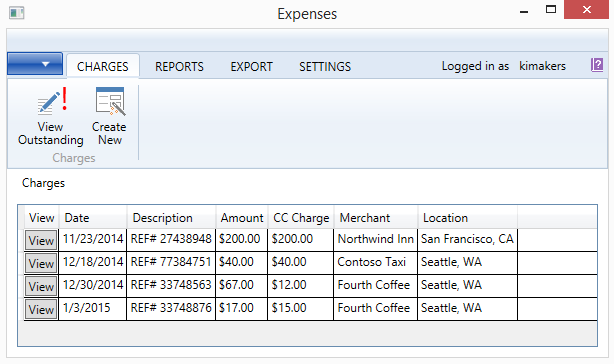
1. Press F5 to launch the Expenses service and client application.
2. When the WPF application loads, a “Sign in to Azure Active Directory” window should be shown. Sign in with the demo user you created earlier.



1. Since this is your first log in, you will be required to change the password to one of your choosing. Go ahead and do this and then click Submit.



1. Note that we are now signed in using AAD and can use the Expenses application as expected.



1. Stop debugging all applications.
2. Press F5 to launch the applications one more time.
3. Note that you are not prompted to sign in this time, as the FileCache is being used to retrieve the previously retrieved token.
4. Stop debugging all applications.