Azure Bot Service: Authentication Preview

Technical Guide

# Overview

This guide outlines the process for how to take advantage of new bot authentication capabilities in Azure Bot Service. The preview contains capabilities to make it easy for bot developers to have the bot authenticate users to various identity providers such as AAD, GitHub, Uber, etc. This preview also takes steps towards an improved user experience by eliminating the “magic code verification” for some clients (and in a few weeks, single-sign support for some clients).

Before this preview, the Bot was required to have all of the needed OAuth controllers and login links, store the target client IDs and secrets, and perform user token management (storage and refresh). These capabilities were bundled in the BotAuth and AuthBot samples that are GitHub.

With this preview, bot developers no longer need to host OAuth controllers or manage token lifecycle as all of this is done by the Azure Bot Service.

**Please treat the content of this email as part of your NDA with Microsoft** and do not share this information with others. If you have other people who you think would like to try this, please email [jeffders@microsoft.com](mailto:jeffders@microsoft.com) and them an into email.

# What’s Included

This preview will continue to improve, and as of today includes:

* Improvements to the DirectLine, WebChat, Telegram, Facebook, SMS, Email, Kik, Slack, and GroupMe channels to support new authentication features.
* Improvements to the Azure Portal to add, delete, and configure connection settings to various OAuth identity providers
* Support for a variety of out-of-the-box identity providers including AAD v1, AAD v2, GitHub, etc.
* Updates to the C# Bot Builder SDK to be able to retrieve tokens, create OAuthCards and handle TokenResponse events
* New Webchat and DirectLineJS libraries to eliminate the need for the 6-digit magic code verification.
* Samples for how to make a bot that authenticates to AAD (v1 and v2) and GitHub
* A web client sample for how to host WebChat

You can use some or all of these pieces as part of the preview.

# What’s Coming Soon

* Improvements to the Teams, Skype, Skype for Business, and Cortana channels to support new authentication features.
* A Management API to be able to add, delete, and configure connection settings programmatically
* Updates to the Node Bot Builder SDK to be able to retrieve tokens, create OAuthCards and handle TokenResponse events
* Single-Sign On (SSO) support in Webchat
* Single-Sign On (SSO) support in Teams, Cortana, Skype, and Skype for Business
* Removing the 6-digit magic code when using Teams, Cortana, Skype, and Skype for Business
* Updates to the emulator to use sign-in cards

# How Do I Get Support

Please email Jeff Derstadt [jeffders@microsoft.com](mailto:jeffders@microsoft.com) and Ming-wei Wang [mingweiw@microsoft.com](mailto:mingweiw@microsoft.com) with any questions, issues, or points of feedback.

# New Feature Walkthrough

Below are steps on how to use these features to create a sample bot that connects to the Microsoft Graph using an AAD V2 token. In this sample, you’ll use a number of things from the GitHub repo and this section will describe how to set each one up including the bot and a React client application you can use to talk with the bot.

Prereqs: To do these steps, you’ll need Visual Studio 2017, npm, node, and git installed.

**Create the basic bot**

1. Create a bot using the Azure Portal and send the botid (the text name like ‘MyCoolBot’) to [jeffders@microsoft.com](mailto:jeffders@microsoft.com) to have it whitelisted.
2. Enable this bot to use the DirectLine channel, and copy down your DirectLine secret as you’ll need this later.
3. Remember the App Service name you used to publish this app, as you’ll be publishing one of the sample bots into this App Service later.

**Create the AAD v2 application**

(Note: These instructions are also in section 6.2)

1. You can create an AAD v2 application by going here: <https://apps.dev.microsoft.com>
2. Click on the “Add an app” button
3. Give your bot an Application Name and click “Create”
4. **Copy down the “Application Id” GUID**…this will be your Client ID later in configuration
5. Under the Application Secrets section, click on “Generate New Password”
6. **Copy the password from the pop-up**…this will be you Client Secret later in configuration
7. Under the “Platforms” section, click “Add Platform”
8. In the pop-up,In the click on “Web”
9. Leave “Allow implicit Flow” checked, and enter the Redirect URL:

**https://token-test-eastus.botframework.com/.auth/web/redirect**

The above URL is the current Azure Bot Service signin URL. This will change as we progress through the preview

1. Under Microsoft Graph Permissions, you can add additional Delegated Permissions such as “email”, “Mail.Read”, “profile”, “User.Read”, etc.
2. At the bottom of the page, click on the “Save” button.

**Register your AAD v2 application with your bot**

You now have an AAD v2 application configured. Next, you’ll configure your bot to use this applications by creating a connection in your bot. To do this, follow these steps:

1. Navigate to your bot’s Azure Resource page on <http://portal.azure.com/>
2. Click on “Settings”
3. Scroll to the bottom and find the section called “Service Provider Connection Settings”
4. Click “Add Setting”
5. Fill in the form as follows:
   1. **Name**: A name for your connection that you’ll use in your bot code (this is referred to as the “ConnectionName”)
   2. **Service Provider**: Azure Active Directory v2
   3. **Client id**: <The application id you copied down from your application registration>
   4. **Client secret**: <The password you copied down from your application registration>
   5. **Tenant ID**: <the tenant id for your Azure Active Directory, for example ‘microsoft.com’>
   6. **Scopes**: <the scopes you want, such as “profile Mail.Send Mail.Read User.Read openid”>

**Note**: Scopes is a space-separated list of values for AAD v2.

1. Click “Save”

**Prepare the Bot Sample Code**

1. Clone the github repository at <https://github.com/Jeffders/BotBuilder-Auth>
2. In this directory, open the solution file at: BotBuilder-Auth\Samples\CSharp\Samples.sln
3. Set the AadV2Bot as the start up project.
4. Open the web.config and modify the App Settings as follows:
   1. Set the “ConnectionName” to the value you used when you configured your bot’s Service Provider Connection Setting.
   2. Set the MicrosoftAppId value\*
   3. Set the MicosoftAppPassword value\*

<appSettings>

<add key="ConnectionName" value="YOUR\_AADV2\_CONNECTION\_NAME"/>

<add key="MicrosoftAppId" value="YOUR\_BOT\_MS\_APPID" />

<add key="MicrosoftAppPassword" value="YOUR\_BOT\_MS\_PASSWORD" />

</appSettings>

\*If you do not know how to get you MicosoftAppId and MicrosoftPassword values, look in the ApplicationSettings of the Azure App Service that was provisioned for your bot on the Azure Portal.

1. You can now publish this bot code back to your Azure subscription (right-click on the project and choose ‘Publish…’). You’ll need to set up a publishing configuration that uses the App Service that you used when configurating the bot in the Azure Portal.

**What the Bot Will Do**

At this point the sample of the bot code is ready, but we can take a closer look at what this bot is going to do. This bot can respond to a few simple tasks that can be done against AAD such as checking recent email, sending an email, or displaying who you are and who your manager is. The bot does this by using a token from an AAD v2 application against the Microsoft.Graph library.

**The Token Retrieval Flow**

When a user asks the bot to do something that requires the bot to have the user logged in, the bot can execute the following code to initiate retrieving a token for a given connection:

|  |
| --- |
| // First ask Bot Service if it already has a token for this user  var token = await context.GetUserTokenAsync(ConnectionName).ConfigureAwait(false);  if (token != null)  {  // use the token to do exciting things!  }  else  {  // If Bot Service does not have a token, send an OAuth card to sign in  await SendOAuthCardAsync(context, (Activity)context.Activity);  } |

In this code, first the bot does a quick check to determine if the Azure Bot Service already has a token for the user (which is identified by the current Activity sender) and the given ConnectionName (which is the connection name used in configuration). The bot will either already have a token cached or it will not. The call to GetUserTokenAsync peforms this ‘quick check’. If Azure Bot Service has a token and returns it, the token can immediately be used. If Azure Bot Service does not have a token, this method will return *null*. In this case, the bot can send a customized OAuthCard for the user to login:

|  |
| --- |
| private async Task SendOAuthCardAsync(IDialogContext context, Activity activity)  {  await context.PostAsync($"To do this, you'll first need to sign in.");  var reply = activity.CreateReply();  reply.Attachments = new List<Attachment>() {  new Attachment()  {  ContentType = OAuthCard.ContentType,  Content = new OAuthCard()  {  Text = $"Please sign in to proceed.",  ConnectionName = ConnectionName,  Buttons = new CardAction[]  {  new CardAction() { Title = "Sign In", Type = ActionTypes.Signin }  },  }  }  };  await context.PostAsync(reply);  context.Wait(WaitForToken);  } |

You can customize the OAuthCard with whatever text and button text you want. The important pieces are:

* That the ContentType be set to OAuthCard.ContentType
* That you specify the ConnectionName property with the name of the connection you want to use
* That there is one button with a CardAction of Type ActionTypes.Signin; note that you do not need to specify any value for the sign in link

At the end of this call, the bot needs to “wait for the token” to come back. This waiting takes place on the main Activity stream because there could be a lot the user needs to do to sign-in.

The Bot’s WaitForToken code looks like this:

|  |
| --- |
| private async Task WaitForToken(IDialogContext context, IAwaitable<object> result)  {  var activity = await result as Activity;  var tokenResponse = activity.ReadTokenResponseContent();  if (tokenResponse != null)  {  // Use the token to do exciting things!  }  else  {  if (!string.IsNullOrEmpty(activity.Text))  {  tokenResponse = await context.GetUserTokenAsync(ConnectionName,  activity.Text);  if (tokenResponse != null)  {  // Use the token to do exciting things!  return;  }  }  await context.PostAsync($"Hmm. Something went wrong. Let's try again.");  await SendOAuthCardAsync(context, activity);  }  } |

In this code the Bot’s dialog class is waiting for a “TokenResponseEvent” (more about how this is routed to the Dialog stack is below). The WaitForToken method first determines if this event was sent. If it was sent, it can be used by the bot. If it was not, the WaitForToken method takes whatever text was sent to the bot and passes it to GetUserTokenAsync. The reason for this is that some clients (like WebChat) do not need the Magic Code verification code and can directly send the Token in the TokenResponseEvent. Other clients still require the magic code (like Facebook or Slack). The Azure Bot Service will present these clients with a six digit magic code and ask the user to type this into the chat window. While not ideal, this is the ‘fall back’ behavior and so if WaitForToken receives a code, the bot can send this code to the AzureBotService and get a token back. If this call also fails, then you can decide to report an error, or do something else. In most cases though, the bot will now have a user token.

If you look in the MessageController.cs file, you’ll see that Event activities of this type are also routed to the dialog stack:

|  |
| --- |
| else if(message.Type == ActivityTypes.Event)  {  if(message.IsTokenResponseEvent())  {  await Conversation.SendAsync(message, () => new Dialogs.RootDialog());  }  } |

On subsequent calls to the bot, notice that the token is never cached by this sample bot. This is because the bot can always ask the Azure Bot Service for the token. This avoids the bot needing to manage the token lifecycle, refresh the token, etc, as Azure Bot Service does all of this for you.

**Using WebChat & Running the Sample Client**

The preview comes with a sample WebChat client that you can use to play with your bot. The sample WebChat client includes a few things:

* A node web service that exchanges the DirectLine secret for a DirectLine token. This is a very important piece of securing the conversations your users have in WebChat…you should always exchange the DirectLine secret for a DirectLine token from a service that is not exposed in your client. See <https://docs.microsoft.com/en-us/azure/bot-service/rest-api/bot-framework-rest-direct-line-3-0-authentication> for details. This service is in the *WebChatClient* directory.
* A React web client that takes the DirectLine token and creates a page with an embedded WebChat control in it. This is in the *WebChatClient/client* directory

The preview comes with newly built node\_module packages for botbuilder\_webchat and botbuilder\_diectlinejs. **If you ae building your own client, you’ll need to replace the versions downloaded from npm with the versions in the BotBuilder-Auth\WebchatPackages directory (they will be hosted soon).** Just delete the versions downloaded from npm in the node\_modules directory and copy these two packages into the node\_modules directoy. They contain the new code to eliminate the 6-digit magic code.

To get use the sample client, you need to:

1. From a command prompt, go to the *BotBuilder-Auth\Samples\WebChatClient\client* directory:
   1. Run “npm install”
   2. Run “setup.cmd”

This command just erases the node\_modules\botframework\_webchat and node\_modules\botframework\_directlinejs packages and replaces them with the preview ones in the *BotBuilder-Auth\WebChatPackages* directory

1. Navigate to the *BotBuilder-Auth\Samples\WebChatClient* directory:
   1. Run “npm install”
   2. Open the server.js file and add your DirectLine secret:

// Configuration

var secret = '<YOUR DIRECTLINE SECRET>';

var domain = 'https://directline.botframework.com/v3/directline';

Now the client hosting service will exchange your DirectLine secret for a token that WebChat in the client UI can use to talk with DirectLine.

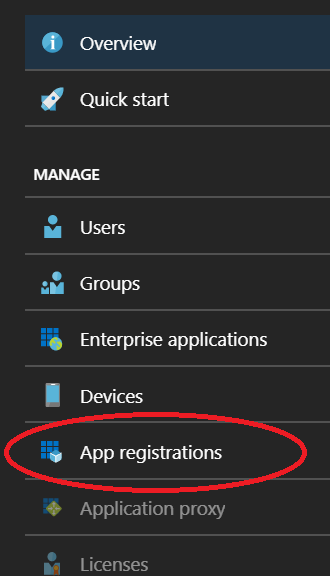
1. To run the full client (the service + client web page with WebChat) at the same time, from the *BotBuilder-Auth\Samples\WebChatClient* directory:
   1. Run: “npm start”
2. You can open a browser and go to <http://localhost:3002/> to see your web client and talk to your published bot.

# Connecting to AAD using v1 and v2 Tokens

This section describes how to create an AAD application so that your bot can connect to the O365 Graph API, you own AAD resources, etc. If you want to learn more about the difference between AAD v1 and V2, you can start [here](https://docs.microsoft.com/en-us/azure/active-directory/develop/active-directory-v2-compare) and [here](https://docs.microsoft.com/en-us/azure/active-directory/develop/active-directory-appmodel-v2-overview).

## Using an AAD v1 Token

1. You can create an AAD v1 application by going here: <https://portal.azure.com/#blade/Microsoft_AAD_IAM/ActiveDirectoryMenuBlade/Overview>
2. Click on “App Registrations”



1. Click on “New application registration”
2. Fill in the form:
   1. Name: **The name of your application**
   2. Application type: Pick “**Web app / API**”
   3. Sign-on URL: **https://token-test-eastus.botframework.com/.auth/web/redirect**

The above URL is the current Azure Bot Service signin URL. This will change as we progress through the preview

* Then press the “Create” button

1. When the blade for your application opens, **copy down the “Application ID” GUID**. This is your “Client ID” value.
2. Next click on “Settings” near the top to configure your application.
3. Click on “Keys” and under “Passwords” type in:
   1. Type “BotLogin” for the Key description.
   2. Set the Expires to “Never” (or whatever you want here)
   3. Click “Save”
   4. **Copy down the password value**. This is your “Client Secret” value
   5. Close the Keys panel.
4. Next in Settings click on “Required permissions”
   1. Click “Add”
   2. Click on “Select an API” and pick “Microsoft Graph”
   3. Click on “Select permissions” and pick the Application Permissions your application will use. Anything with “Requires Admin” will require both a user and a tenant admin to login, so for your bot tend to stay away from these. Fun ones for the Microsoft Graph include:
      1. Sign in and read user profile
      2. Read user mail
      3. Send mail as a user
      4. View users’ email address
      5. View users’ basic profile
   4. Click “Save”

You now have an AAD v1 application configured. Next, you’ll configure your bot to use this applications by creating a connection in your bot. To do this, follow these steps:

1. Navigate to your bot’s Azure Resource page on <http://portal.azure.com/>
2. Click on “Settings”
3. Scroll to the bottom and find the section called “Service Provider Connection Settings”
4. Click “Add Setting”
5. Fill in the form as follows:
   1. **Name**: A name for your connection that you’ll use in your bot code
   2. **Service Provider**: Azure Active Directory
   3. **Client id**: <The application id you copied down from your application registration>
   4. **Client secret**: <The password you copied down from your application registration>
   5. **Grant Type**: authorization\_code
   6. **Login URL**: <https://login.microsoftonline.com>
   7. **Tenant ID**: <the tenant id for your Azure Active Directory, for example ‘microsoft.com’>
   8. **Resource URL**: <https://graph.microsoft.com/>

Note: You are welcome to configure this however you want, but these values enable your application to use the Graph API in O365 as an example.

* 1. **Scopes**: leave blank

1. Click “Save”

You can now use this connection name in your bot code to retrieve user tokens.

## Using an AAD v2 Token

1. You can create an AAD v2 application by going here: <https://apps.dev.microsoft.com>
2. Click on the “Add an app” button
3. Give your bot an Application Name and click “Create”
4. **Copy down the “Application Id” GUID**…this will be your Client ID later in configuration
5. Under the Application Secrets section, click on “Generate New Password”
6. **Copy the password from the pop-up**…this will be you Client Secret later in configuration
7. Under the “Platforms” section, click “Add Platform”
8. In the pop-up,In the click on “Web”
9. Leave “Allow implicit Flow” checked, and enter the Redirect URL:

**https://token-test-eastus.botframework.com/.auth/web/redirect**

The above URL is the current Azure Bot Service signin URL. This will change as we progress through the preview

1. Under Microsoft Graph Permissions, you can add additional Delegated Permissions such as “email”, “Mail.Read”, “profile”, “User.Read”, etc.
2. At the bottom of the page, click on the “Save” button.

You now have an AAD v2 application configured. Next, you’ll configure your bot to use this applications by creating a connection in your bot. To do this, follow these steps:

1. Navigate to your bot’s Azure Resource page on <http://portal.azure.com/>
2. Click on “Settings”
3. Scroll to the bottom and find the section called “Service Provider Connection Settings”
4. Click “Add Setting”
5. Fill in the form as follows:
   1. **Name**: A name for your connection that you’ll use in your bot code
   2. **Service Provider**: Azure Active Directory v2
   3. **Client id**: <The application id you copied down from your application registration>
   4. **Client secret**: <The password you copied down from your application registration>
   5. **Tenant ID**: <the tenant id for your Azure Active Directory, for example ‘microsoft.com’>
   6. **Scopes**: <the scopes you want, such as “profile Mail.Send Mail.Read User.Read openid”>

**Note**: Scopes is a space-separated list of values for AAD v2.

1. Click “Save”

You can now use this connection name in your bot code to retrieve user tokens.