**Using Web Chat with Azure Bot Service’s Authentication**

*Posted on*[*September 1, 2018*](https://blog.botframework.com/2018/09/01/using-webchat-with-azure-bot-services-authentication/)*by*[*The Bot Framework Team*](https://blog.botframework.com/author/botframeworkteam/)

**[Updated on 9/23/2019]**

This blog covers how to use Web Chat with the Azure Bot Service’s built-in authentication capability to authenticate chat users with various identity providers such AAD, GitHub, Facebook, etc, including best practices on how to ensure a secure experience. This tutorial also covers where the built-in authentication features are currently supported and where they are not.

**Step 1: Registering Connections and Making a Bot**

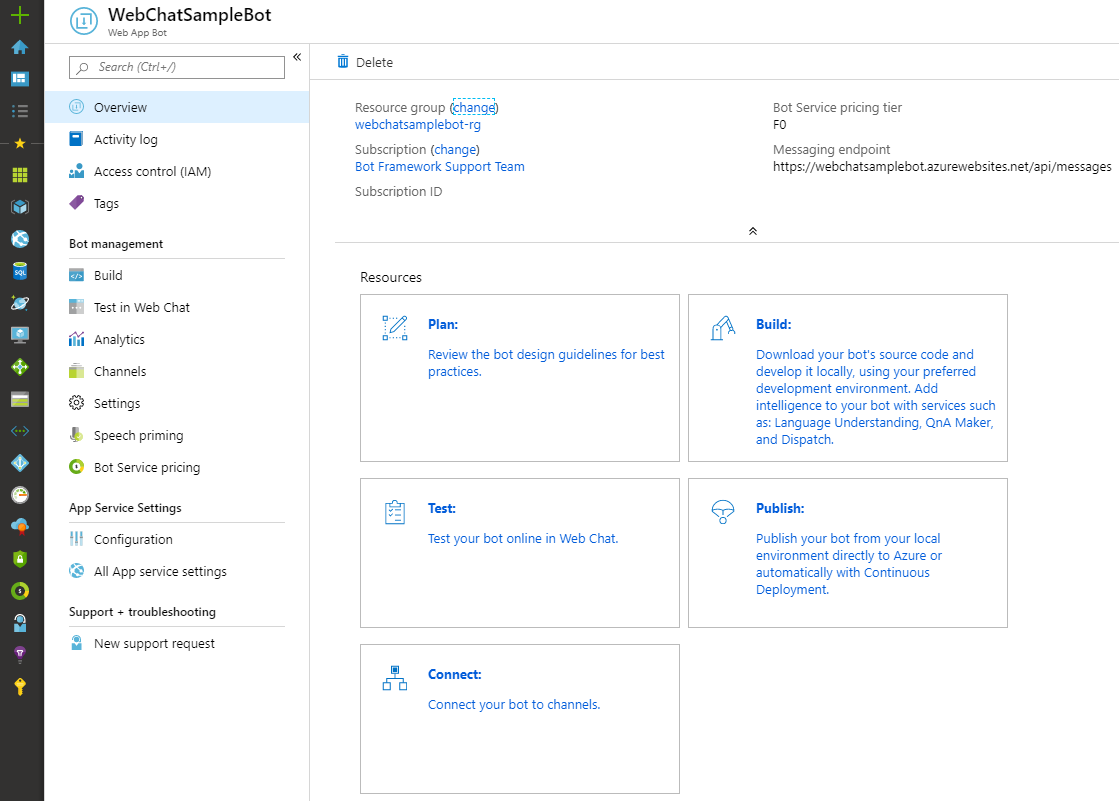
The first step is to register your connections and to create a bot that supports sending OAuthCards and receiving tokens. These steps are documented [here](https://docs.microsoft.com/en-us/azure/bot-service/bot-builder-authentication?view=azure-bot-service-4.0&tabs=csharp%2Cbot-oauth). Once you have a bot registered, a connection set, and are able to run your bot successfully using the Bot Framework Emulator to sign a user in you are ready to build a Web Chat based client.

*Please note: AAD organizational policies, established by the AAD tenant administrator, may restrict the ability to authenticate against an AAD tenant. In these cases, it is necessary to work with your AAD tenant administrator to acquire the proper permissions. If that is not an option, then consider creating a secondary AAD directory used specifically for authentication.*

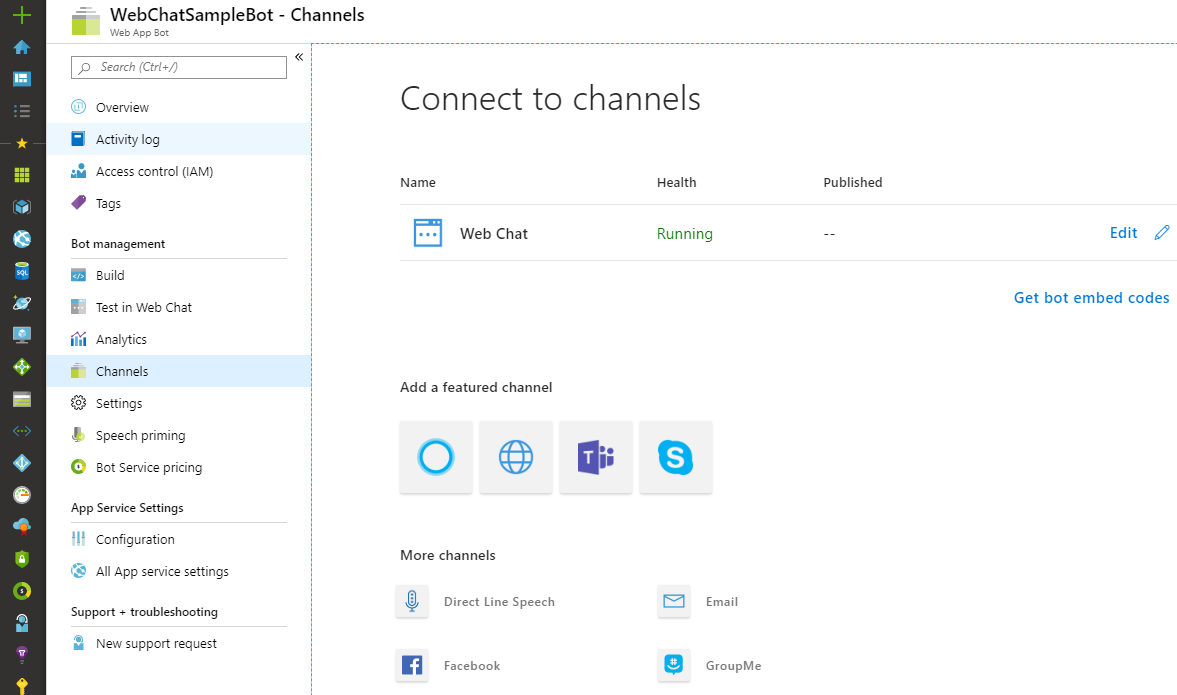
**Step 2: Enabling Direct Line**

The next thing you’ll want to do is to enable Direct Line for your bot. Also note, all bots come “pre-registered” with the Web Chat channel. You now have the choice of using either the Direct Line *or* Web Chat secret for ABS authentication. To register for the Direct Line channel:

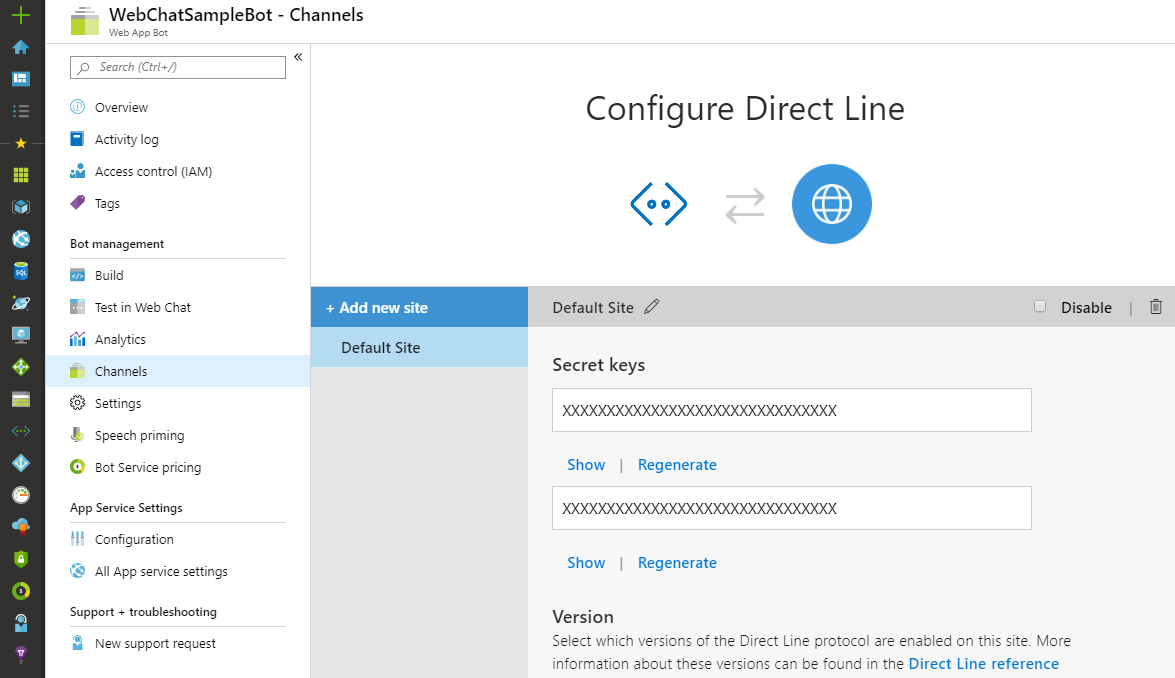
1. Go to <https://portal.azure.com/>
2. Navigate to your bot resource:



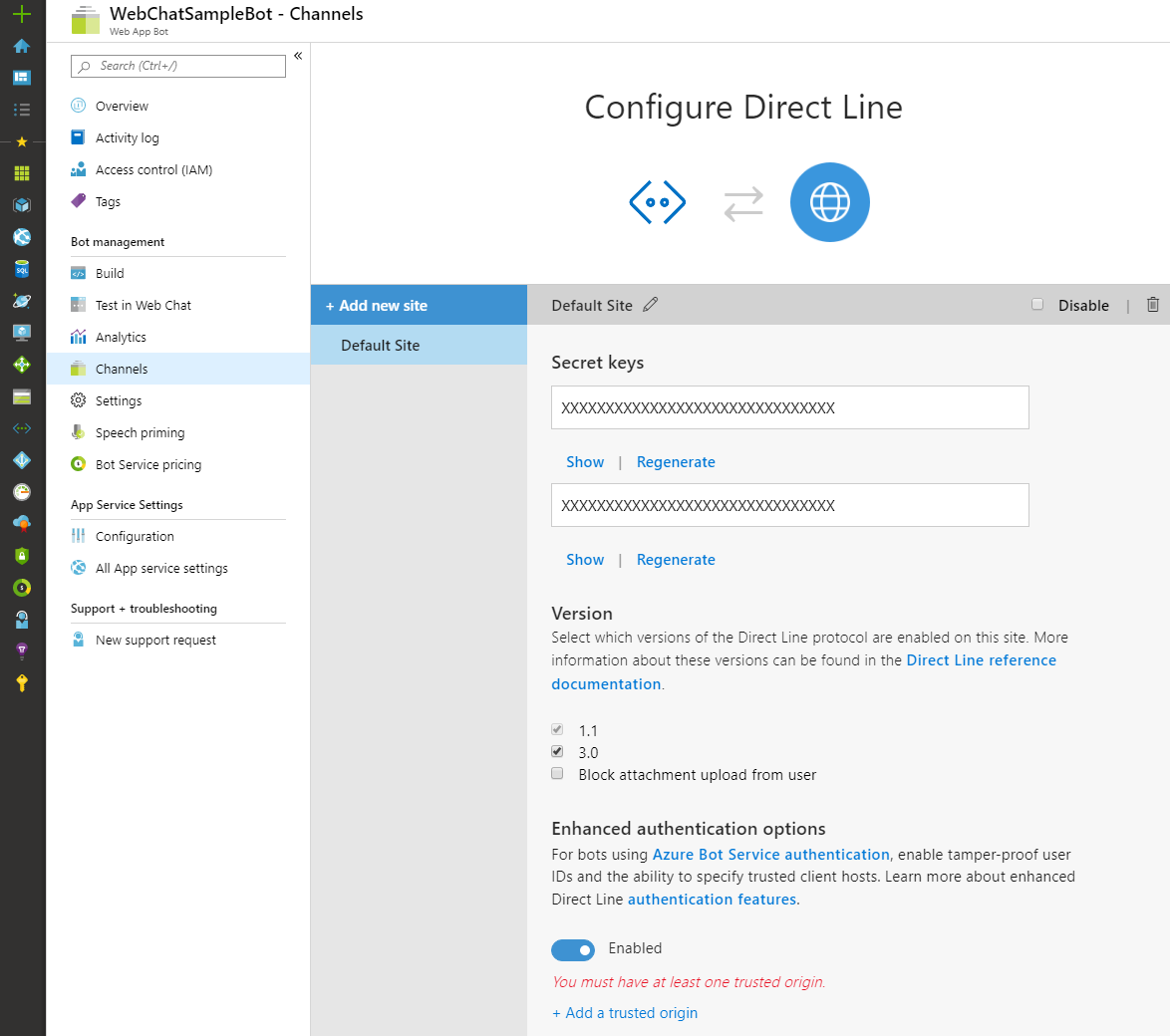
1. Click on “Channels”, then click on the Direct Line icon (it looks like a globe):

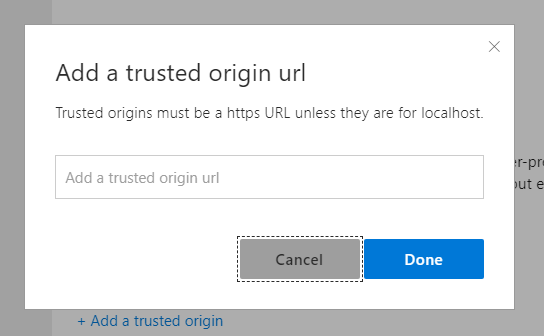


1. Copy one of the Secret Key values and store this for later. This is your ‘Direct Line Secret’ string.



1. Enable “Enhanced authentication options”. This feature provides access to your bot from external sites via CORS. Add at least one trusted origin in the form of a URI. For example, if you host Web Chat at [https://contoso.com](https://contoso.com/) then enter “https://contoso.com” as a trusted origin. If you test Web Chat on localhost, then add [http://localhost](http://localhost/) as a trusted origin. Learn more about these options [here](https://aka.ms/bots/directline-auth-enhancements).





Your bot is now enabled to use Direct Line and configured for authentication and we can proceed to the next steps.

**Step 3: Creating a Client Controller**

While there are many mechanisms to invoke Web Chat on a client HTML page, one thing that is very important to do is to **ensure you keep your Direct Line Secret a secret**! The Direct Line Secret is as important as a database connection string and these sorts of things should never show up in HTML or even in the JavaScript that is running behind an HTML page.

Instead what you need to do is to keep your Direct Line Secret hidden in a client controller. This client controller should exchange the Direct Line Secret for a Direct Line Token, which CAN be exposed in HTML and JavaScript. You can find lots of details about this [here](https://docs.microsoft.com/en-us/azure/bot-service/rest-api/bot-framework-rest-direct-line-3-0-authentication?view=azure-bot-service-3.0), but the summary is that a Direct Line Secret can connect to ANY conversation whereas a Direct Line Token is bound to a single conversation.

A good practice for securing the Direct Line Secret is to create a client controller that performs the exchange for a Direct Line Token and then passes this Direct Line Token back to the HTML page.

Here is an example ASP.NET controller that does this and returns a Direct Line Token:

public class HomeController : Controller

{

public async Task<ActionResult> Index()

{

var secret = GetSecret();

HttpClient client = new HttpClient();

HttpRequestMessage request = new HttpRequestMessage(

HttpMethod.Post,

$" <https://directline.botframework.com/v3/directline/tokens/generate>");

request.Headers.Authorization = new AuthenticationHeaderValue("Bearer", secret);

var response = await client.SendAsync(request);

string token = String.Empty;

if (response.IsSuccessStatusCode)

{

var body = await response.Content.ReadAsStringAsync();

token = JsonConvert.DeserializeObject<DirectLineToken>(body).token;

}

var config = new ChatConfig()

{

Token = token,

};

return View(config);

}

}

public class DirectLineToken

{

public string conversationId { get; set; }

public string token { get; set; }

public int expires\_in { get; set; }

}

public class ChatConfig

{

public string Token { get; set; }

}

And here is a node.js controller that does the same:

const path = require('path');

const restify = require('restify');

const bodyParser = require('body-parser');

const request = require('request');

const corsMiddleware = require('restify-cors-middleware');

const ENV\_FILE = path.join(\_\_dirname, '.env');

require('dotenv').config({

path: ENV\_FILE || process.env.directLineSecret

});

const cors = corsMiddleware({

origins: ['\*']

});

let server = restify.createServer(); // Get an instance of the restify server

server.pre(cors.preflight);

server.use(cors.actual);

server.use(bodyParser.json({

extended: false

}));

server.listen(process.env.port || process.env.PORT || 3000, function() {

console.log(`Listening to ${ server.url }.`);

});

// Generates a Direct Line token

server.post('/directline/token', (req, res) => {

const options = {

method: 'POST',

uri: 'https://directline.botframework.com/v3/directline/tokens/generate',

headers: {

'Authorization': `Bearer ${process.env.directLineSecret}`

};

request.post(options, (error, response, body) => {

if (!error && response.statusCode < 300) {

res.send( body );

} else {

res.status(500).send('Call to retrieve token from Direct Line failed');

}

});

});

**Step 3: Trusted Origins and Tamper-proof user IDs**

**Trusted origins**

When an OAuthCard shows up in a client as a sign-in link or button, the sign-in link needs to be protected from attackers who try to steal other people’s tokens. The gist of the attack is that an attacker starts a conversation with a bot to get a sign-in link. The attacker sends the sign-in link to someone else and tells him or her to click on the link. When that person does, he or she will be asked to sign-in and the result of that flow will present the bot with a token to use for that conversation; but that conversation is the attacker’s conversation. To prevent this, we want to guarantee that the sign in link can only be used in the same browser session as the OAuthCard.

One approach is to present a final factor code at the end of sign-in that the user must type into the chat window before the bot can retrieve the user’s token. This is often referred to as the “magic code”. This however doesn’t guarantee that the sign-in link can only be used in the same browser session. The attacker can still send the sign-in link and use phishing techniques to try to get another user’s magic code.

With Web Chat, we added a mechanism that’s both more secure and users do not need to deal with any “magic code”. This mechanism ensures that a user can only sign-in through the sign-in link in the same browser session as the OAuthCard. There are two parts of this mechanism:

1. Using a list of ‘trusted origin’ hostnames that can request a Web Chat session id
2. Associate a unique signed session id cookie with the OAuthCard.

**The browser that is viewing Web Chat must have 3rd party cookies enabled for this to work**

**Tamper-proof user IDs**

The authentication capabilities in Azure Bot Service acquire user tokens for a given user using a connection on a particular bot. The way Azure Bot Service distinguishes which user it’s acquiring a token for is using the User.Id that comes through on Activities. In the case of Web Chat, this User.Id is modifiable by the client. If an attacker knows another user’s user ID, they can modify the Web Chat control in their browser to pretend to be that user. There are a couple of ways to mitigate against this.

1. Ensure the user IDs are not guessable.
2. Detect that a client has changed the user ID and reject the change.

The first mitigation is up to you. A good way is to generate these within your controller method. If your website is anonymous, then you should generate a unique value within the controller and return this to your client. If your website already is using some sort of authentication, then using a bearer token (or its hash for shorter IDs) would suffice. Do not use email addresses or regular names as these are easily guessed.

We’ve implemented platform support for #2.

With the “Enhanced authentication options” that you enabled earlier, all you need to do to leverage both the no magic code safe sign in and the tamper-proof user ID is to specify a user ID when you exchange the Direct Line Secret for a Direct Line token. Note that the user ID must begin with “dl\_”. To learn more about these options, go [here](https://aka.ms/bots/directline-auth-enhancements).

Here is a look at the final controller method in ASP.NET that does the Direct Line Secret exchange with a unique generated user id:

public class HomeController : Controller

{

public async Task<ActionResult> Index()

{

var secret = GetSecret();

HttpClient client = new HttpClient();

HttpRequestMessage request = new HttpRequestMessage(

HttpMethod.Post,

$" https://directline.botframework.com/v3/directline/tokens/generate");

request.Headers.Authorization = new AuthenticationHeaderValue("Bearer", secret);

var userId = $"dl\_{Guid.NewGuid()}";

request.Content = new StringContent(

JsonConvert.SerializeObject(

new { User = new { Id = userId } }),

Encoding.UTF8,

"application/json");

var response = await client.SendAsync(request);

string token = String.Empty;

if (response.IsSuccessStatusCode)

{

var body = await response.Content.ReadAsStringAsync();

token = JsonConvert.DeserializeObject<DirectLineToken>(body).token;

}

var config = new ChatConfig()

{

Token = token,

UserId = userId Guid. NewGuid(). ToString()

};

return View(config);

}

}

public class DirectLineToken

{

public string conversationId { get; set; }

public string token { get; set; }

public int expires\_in { get; set; }

}

public class ChatConfig

{

public string Token { get; set; }

public string UserId { get; set; }

}

And here is a node.js controller that does the same:

// userID must start with `dl\_`

const userId = ( req.body && req.body.id ) ? req.body.id : `dl\_${ Date.now() + Math.random().toString( 36 ) }`;

// Generates a Direct Line token

server.post('/directline/token', (req, res) => {

const options = {

method: 'POST',

uri: 'https://directline.botframework.com/v3/directline/tokens/generate',

headers: {

'Authorization': `Bearer ${process.env.directLineSecret

},

json: {

User: {

Id: userId

}

}

};

request.post(options, (error, response, body) => {

if (!error && response.statusCode < 300) {

res.send({

token: body.token

});

} else {

res.status(500).send('Call to retrieve token from DirectLine failed');

}

});

});

**Step 4: Creating the Web Chat HTML page**

There are several ways of including Web Chat within an HTML page and these are all outlined [here](https://github.com/Microsoft/BotFramework-WebChat). Not all of these are appropriate for use when building a website that performs secure logins:

* Using an IFRAME is not a supported way of driving the authentication flow using OAuthCards because you cannot set a trusted origin and you cannot set user ids
* Running Web Chat inline from the CDN or your own copy of Web Chat is supported.
* Running Web Chat as a React component is supported.

Here is an example of a “Home” view for the ASP.NET controller defined earlier:

@model ChatConfig

@{

ViewData["Title"] = "Home Page";

}

<div id="webchat" role="main" />

<script src="https://cdn.botframework.com/botframework-webchat/latest/webchat.js

"></script>

<script>

window.WebChat.renderWebChat({

directLine: window.WebChat.createDirectLine({ token: 'YOUR\_DIRECT\_LINE\_TOKEN' }),

userID: 'YOUR\_USER\_ID'

}, document.getElementById('webchat'));

</script>

Here is an example of a React instantiation of Web Chat that uses the token and userId:

import { DirectLine } from 'botframework-directlinejs';

import React from 'react';

import ReactWebChat from 'botframework-webchat';

export default class extends React.Component {

constructor(props) {

super(props);

this.directLine = new DirectLine({ token: 'YOUR\_DIRECT\_LINE\_TOKEN' });

}

render() {

return (

<ReactWebChat directLine={ this.directLine } userID="YOUR\_USER\_ID" />

element

);

}

}

**Summary**

While there are a number of steps in this tutorial to create a secure Web Chat client that allows user login, the basic steps involve:

* Performing Direct Line Secret to Token exchange on a client controller
* Passing in TrustedOrigin values when performing secret to token exchange
* Ensuring that each user has a unique, non-guessable user id value that is used with Web Chat
* Choosing the proper way to place Web Chat into your HTML page: by inlining it or using a React component but NOT using an IFRAME

Posted in [Bot Framework](https://blog.botframework.com/category/bot-framework/)