Game Service Sample Configuration Guide and Documentation



By Cameron Goodwin

Xbox Advanced Technology Group

**This document is confidential** and provided to you under a Microsoft non-disclosure agreement. While we have tried to ensure the accuracy of this document, we provide no express warranties or guarantees regarding the information. The information is subject to change. Microsoft may have intellectual property rights in the subject matter of this paper. This document doesn’t grant you a license to those rights—it’s for informational purposes only.

Abstract

This guide provides step-by-step instructions to configure an App Service in Azure running the Game Service sample and allow an Xbox Live enabled title to call that endpoint. The sample provides code for handling the authorization between client and server with Xbox Secure Tokens (XSTS or X-tokens) and calling Xbox Live services from your server. Included in this guide are some additional documentation and explanations about design decisions and best practices in game service development.

Contents

[Update history 3](#_Toc32496984)

[Introduction 3](#_Toc32496985)

[Section 1 – Handling X-tokens 3](#_Toc32496986)

[Configuring the NSAL, X-token definitions, and Relying Parties in Partner Center 4](#_Toc32496987)

[Configuring your Relying Party to use an asymmetric certificate 4](#_Toc32496988)

[Creating a self-signed Relying Party certificate 4](#_Toc32496989)

[Defining your Web Service, Relying Party and X-token claims 5](#_Toc32496990)

[Defining the service endpoint in your App’s NSAL 6](#_Toc32496991)

[Building the sample and debugging locally 7](#_Toc32496992)

[Building the sample in Visual Studio 2019 7](#_Toc32496993)

[Enabling Application Settings and Certs when debugging locally 7](#_Toc32496994)

[Running and debugging the sample locally 8](#_Toc32496995)

[Using Fiddler to debug request calls locally 8](#_Toc32496996)

[Deploying to an Azure App Service and debugging remotely 9](#_Toc32496997)

[Adding certificates with private keys to your deployed service 10](#_Toc32496998)

[Configuring the App Settings in Azure 11](#_Toc32496999)

[Creating a debug deployment to Azure for debugging 11](#_Toc32497000)

[Attaching the Visual Studio Debugger to your Azure App Service 11](#_Toc32497001)

[Using the SimpleWinHttp sample to verify your service and NSAL configuration 12](#_Toc32497002)

[Viewing log output from the service 12](#_Toc32497003)

[Viewing log output when running locally through Visual Studio 2019 13](#_Toc32497004)

[Enabling service log output through Azure 13](#_Toc32497005)

[Viewing service log output real-time when running on Azure 13](#_Toc32497006)

[Viewing service log output real-time from Visual Studio 13](#_Toc32497007)

[Section 2 – Service Auth X-tokens 13](#_Toc32497008)

[Creating a Business Partner certificate 14](#_Toc32497009)

[Adding the cert’s thumbprint for initialization to the App Settings 15](#_Toc32497010)

[Verifying your service can obtain a Service Token 16](#_Toc32497011)

[Section 3 – Delegated Auth X-tokens 16](#_Toc32497012)

[Section 4 – Calling Xbox Live Services 17](#_Toc32497013)

[Section 5 – Collections and Commerce 17](#_Toc32497014)

[Configuring your products for B2B Collections in Partner Center 18](#_Toc32497015)

[Managing consumables with Transaction IDs 19](#_Toc32497016)

[Validating License Tokens for PC titles 19](#_Toc32497017)

[Section 6 – Logging & Correlation Vectors 19](#_Toc32497018)

[Appendix A: Setting up an Azure SQL Database 20](#_Toc32497019)

[Creating an Azure SQL Database 20](#_Toc32497020)

[Initializing the database with Migrations and debugging locally 21](#_Toc32497021)

[Appendix B: Running the sample on Linux 21](#_Toc32497022)

[FAQs and troubleshooting 22](#_Toc32497023)

# Update history

|  |  |  |
| --- | --- | --- |
| **Date** | **Version** | **Description** |
| January 25, 2019 | 1.3 | * Redesigned and re-organized into different sections that build on top of each other. |
| July 30, 2019 | 1.4 | * Added note about License Token functionality and updated Troubleshooting answer related to SSL cert issues |
| November 1, 2019 | 1.5 | * Renamed to “Game Service Sample” * Updated to .NET Core 3.0 and Visual Studio 2019 * Terminology and naming updated to match the Xfest 2019 talk XSTS Auth and Server to Server Made Easy * Migrated locally cached items to in-memory cache rather than in-memory database * Removed Azure Key Vault usage and migrated secrets and certs to App Settings * Reordered and condensed the sections in the configuration guide * Moved source code files to match the new sections layout |
| January 24, 2020 | 1.5 | * Text updates to Configuration Guide and Readme only |
| February 25, 2020 | 1.7 | * Added GDNP erasure list b2b endpoint * Fixed bug with S-token caching that caused errors generating signature headers |
| May 12, 2020 | 1.8 | * Removal of Symmetric token encryption handling * Added XBL signing cert caching at server startup using the target URI <https://xsts.auth.xboxlive.com/xsts/signingkeys> |

# Introduction

The Xbox Live platform rely heavily on HTTPS communication with RESTful web services. Being able to use HTTPS with RESTful services in your own Xbox Live enabled title gives you extended flexibility to develop game services that are quick and reliable. This guide describes how to set up an Azure Web Service running the Game Service sample. With this you will be able to start exploring the use of the Xbox Secure Token Service (XSTS) tokens and HTTPS for communication between your Xbox Live enabled title and your custom game services.

Although this guide uses Azure services to host the sample, you can setup the Game Service sample using other cloud hosting services including Linux based services.

# Section 1 – Handling X-tokens

This section will start you off by setting up an Azure instance of sample that your title will be calling with X-tokens. It will then walk you through the configuration process in Partner Center to enable your title to obtain X-tokens from the client and use them for authentication to the Game Service sample. By the end of this section you will be able to call the sample service from your title and get a reply of all the claims found within the client’s X-token used for auth.

If you are unfamiliar with X-tokens or single sign-on auth for Xbox Live enabled titles, we recommend you review the Xfest 2019 talk *XSTS Auth and Server to Server Made Easy*, as well as the documentation article *Your Title, XSTS Tokens, and Web Services.*

For a more in-depth explanation of X-tokens and the process of handling them on your service we recommend you reference the documentation article *Understanding Security Tokens for Xbox*. The sample’s source code comments should also provide helpful information as to what the code is doing and walk you through the token validation and handling process.

## Configuring the NSAL, X-token definitions, and Relying Parties in Partner Center

Before your app can talk to your web service using X-tokens, you will need to define your web service in Partner Center, create a Relying Party, and publish your title’s NSAL via the Xbox Live Config in your sandbox.

## Configuring your Relying Party to use an asymmetric certificate

With asymmetric encryption, the cert’s public key is used to encrypt the token’s content encryption key which was used to encrypt the payload. The private key (the secret) is used on your service to decrypt the content encryption key which can then be used to decrypt that specific token’s payload. But to ensure that the token came from a reliable source at Xbox Live, the token has a digital signature that is created with Xbox Live’s private key and validated with the public key of the cert. So, if your public key is compromised an attacker can encrypt and create fake X-tokens, however they would not have the Xbox Live private key to generate a proper signature.

The Game Service sample has code that specifically fetches the current and upcoming Xbox Live signing certs from <https://xsts.auth.xboxlive.com/xsts/signingkeys> and then caches them on server startup. Alternatively, the sample also provides example code on fetching the cert at runtime from the x5u value in a client token. If the X5U value does not map to an endpoint under \*.xboxlive.com, it will not trust the certificate, fail authentication of the token, and finally log a warning. When the Xbox Live Signing certificate is renewed, the x5u value on all subsequent tokens will be updated. The sample service as written will detect this change, fetch the new certificate, and use it to continue validating the tokens automatically. However, the renewed certificates wil be posted to <https://xsts.auth.xboxlive.com/xsts/signingkeys> weeks before the actuall rollover so it should be already cached when the service is rebooted before the cert rollover. There is no need to refresh or update the Xbox Live certificate on your service if using the methods shown in the sample.

### Creating a self-signed Relying Party certificate

1. On your computer, open the **Developer Command Prompt for Visual Studio**.
2. Run the following command, replacing the example name (Contoso) with your own:

makecert -sv RP\_Private\_Key.pvk -n "CN=Contoso Relying Party" RP\_Cert.cer -b 01/01/2018 -e 12/31/2199 -sky exchange -ss My -a sha256 -len 2048 -r –pe

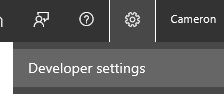
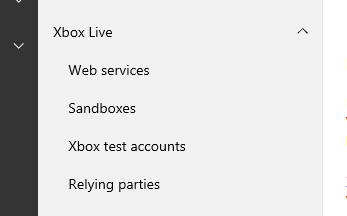
1. When prompted, enter a password for the private key.
2. Run the following command, replacing the password at the end for your own:

PVK2PFX –pvk RP\_Private\_Key.pvk –spc RP\_Cert.cer –pfx RP\_Full\_Cert.pfx -po {password you used above}

1. When prompted, enter the password you set for the private key in step 3.
2. Use the RP\_Cert.cer file to set up your token definition setup in Partner Center.
3. Hold on to the RP\_Full\_Cert.pfx file as we will upload it to the key vault shortly
4. When importing the full certificate, make sure you select the **Mark this key as exportable** option so that you can export it to other servers if needed.

If you ever need to re-export the public key .cer for this certificate, select the **Base-64 encoded X.509 (.CER)** option in the Certificate Export Wizard.

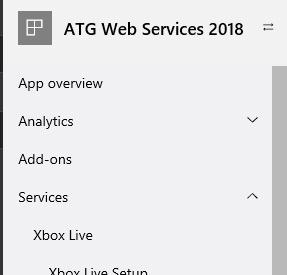
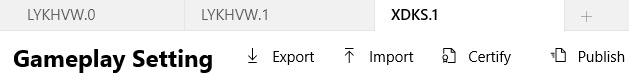
## Defining your Web Service, Relying Party and X-token claims

1. In Partner Center, Select the **Settings Icon in the upper right and then Developer Settings**
2. Next select **Xbox Live->Relying Parties**
3. On the Relying parties page, select **New relying party**
4. Provide an Audience URI\*
5. Select **Asymmetric encryption – JWE RFC 7516** for Encryption Type
6. Click browse your files and select the RP\_Cert.cer file you created on step 6 above.
7. Enter how long (in hours) you want your token’s lifetime to be (recommended 4 hours)
8. Add the **Partner Xbox User ID (ptx)** claim to the token (this will be the unique ID you will have for each user in your database).
9. Click **Save**

\* The Audience URI must be in the form of a host URI, but the name can be different from the URI of the actual service. For example, [*myservice.com*](http://myservice.com/) could be Audience URI of the Relying Party, but the service endpoint is actually https:*//game.myservice.com/action/*.

## Defining the service endpoint in your App’s NSAL

Now that you have defined your Relying Party and X-token claims you will need to define the URL your app will be calling in the Network Security Access List (NSAL) of your Xbox Live configuration. This informs the GetTokenAndSignature API on the client which X-token definition it should give when calling that URL host.

1. Go to your App’s overview page in Dev Center and select **Services->Xbox Live**
2. Check the tab towards the top of the page for the Sandbox you will be testing in.
3. Expand the Services list in the panel on the left again and you will now see the option for **Xbox Live single sign-on,** click on that
4. Click **New endpoint**
5. Enter the URL host address\* of the service your app will be calling
6. Select the **Relying Party** from the drop-down that you previously configured above
7. Click **Save**
8. Go back to the **Xbox Live Gameplay Settings** page
9. Select the tab of the Sandbox you want to publish your updated NSAL to
10. Click **Publish** in the right corner  
    

\* For Section 1, we have not yet deployed or built the service, decide now what the name of the deployed app service in azure will be and then add “**.azurewebsites.net”** to it. For example, if our app service name will be GameServiceSample the URI would be https://GameServiceSample.azurewebsites.net. This value can also be changed at any time to reflect the endpoint of your service.

## Building the sample and debugging locally

Initially it might be easier to run and debug the sample locally from your PC before attempting to run the service through a cloud-based host. To run the sample on your development PC, follow the steps below. It is possible to debug and deploy through Azure, which will be covered later in this section.

*Note: Versions of the sample prior to 1.5 used .NET Core 2.2 and Visual Studio 2017. Version 1.5 of the sample was upgraded to use .NET Core 3.0 which requires Visual Studio 2019.*

### Building the sample in Visual Studio 2019

1. [Download and install the latest .NET Core 3.1 SDK](https://dotnet.microsoft.com/download/dotnet/current)
2. Open **GameService.sln** in Visual Studio 2019
3. The NuGet packages for Newtonsoft.Json, Jose.JWT, and others will download automatically shortly after the project loads
4. Open the **XstsConstants.cs** file and change the value of **ServiceName** to be something related to your title or service (this value is used later in logging and calling other services to identify your server).
5. Compile the solution and verify it succeeds.

### Enabling Application Settings and Certs when debugging locally

When running locally, the app settings need to be configured using the .NET secret-manager by either installing the [Azure Command Line-Interface (CLI](https://docs.microsoft.com/en-us/cli/azure/install-azure-cli?view=azure-cli-latest)) or by using Visual Studio as outlined below. For more information about user-secrets and the Azure Secret Management Tool, see [Safe storage of app secrets in development in ASP.NET Core](https://docs.microsoft.com/en-us/aspnet/core/security/app-secrets?tabs=windows&view=aspnetcore-2.2#SecretManager)

1. Right click on the **GameService** project and then select **Manage User Secrets**
2. In the secrets.json file add the following example to simulate the App Setting to cache your Relying Party cert on startup your service would read from Azure.

"RP\_CERT\_THUMBPRINT": "[thumbprint of your Relying Party Cert]"

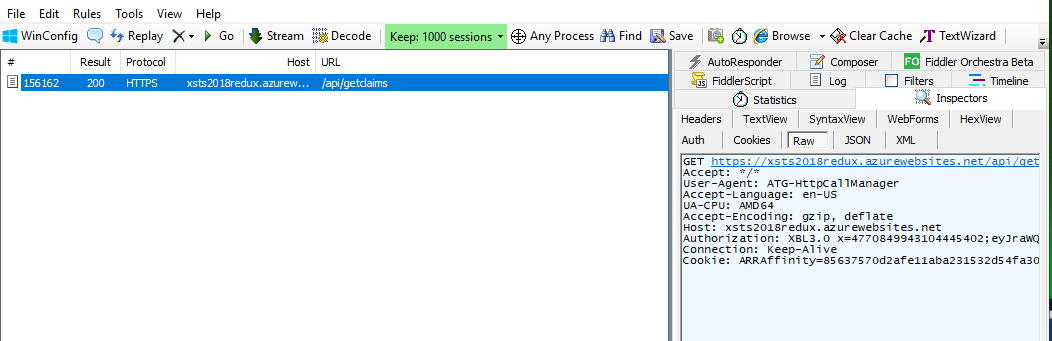
For Certificates, import the certs including private certs to your Machine / Current User cert store as that is where Azure will place the certs once configured above.

### Running and debugging the sample locally

1. Follow the steps above and ensure that the sample builds properly
2. Open **GameService.sln** in Visual Studio 2019
3. Right click on the GameService project and select **Properties**
4. Go to the **Debug** tab
5. Check **Enable SSL** and record the https:// address shown (you will need this when using Fiddler to replay calls to the service for debugging later)
6. Check **Enable Anonymous Authentication**
7. Press **F5** to compile and run the sample locally with the debugger attached

The service is now setup to run on your local machine for debugging. Using Fiddler to debug your server locally is very useful and can be configured as outlined below.

### Using Fiddler to debug request calls locally

1. Install an HTTP development tool such as [Fiddler](https://www.telerik.com/fiddler) (Postman can also be used)
2. Make sure you have configured your title’s NSAL in Partner Center as outlined previously
3. Enable Fiddler monitoring on the console as outlined in [How to use Fiddler with Xbox One](https://developer.microsoft.com/en-us/games/xbox/docs/xdk/fiddler-setup-networking)
4. Run the SimpleWinHttp sample and make a call to your Azure Service as outlined in [Using the SimpleWinHttp sample to verify your server and NSAL configuration](#_Using_the_Xbox) below
5. Look for and copy the raw request format for the call to GetClaims (Example below)  
   
6. Compile and run the server locally
7. Once the server is running and ready to receive requests go to Fiddler’s **Composer** tab and select the **Scratchpad** sub-tab.
8. Paste the raw request text you copied from the window in step 5 into the scratchpad and update the host name to be localhost and the port that your debug server is running on that you got from step 5 of [Running and debugging the sample locally](#_Running_and_debugging). Example below:

GET https://localhost:44366/api/getclaims HTTP/1.1

Accept: \*/\*

Host: localhost:44366

User-Agent: LocalDev

Authorization: XBL3.0 x=[User Hash Goes Here];[Token Goes Here]

Accept-Language: en-US

UA-CPU: AMD64

Accept-Encoding: gzip, deflate

Connection: Keep-Alive

1. Highlight the text in the Composer window and click the **Execute** button to issue the request to your local debug service.

\*Note, although not needed for Section 1, you will need to update Fiddler’s HTTPS settings to skip decryption for **service.auth.xboxlive.com.** You need to do this otherwise you will not be able to get a Service token when debugging locally. See section 3 for more instructions.

## Deploying to an Azure App Service and debugging remotely

Deploying the sample as a fully working service in Azure is quick and easy. This is the fastest way to get a fully functioning endpoint that your game can make calls to from the console. If you are using the sample as a starting foundation for creating your own service, you can use the Azure Web Service to expand your capacity and deployments later to meet your needs.

You will need to have the Azure SDK installed for Visual Studio 2019 to be able to deploy and create a service from Visual Studio. [You can download the needed SDK here.](https://azure.microsoft.com/en-us/downloads/)

1. Right click on the GameService project and select **Publish**
2. Select **New Profile**
3. Select **App Service** and the **Create New** option
4. Click the **Advanced…** link
5. Expand the **File Publish Options** and check **Remove additional files at destination**
6. Click **Save**
7. Click **Publish** and the **Create App Service** wizard will open
8. Enter a name for your app. Note: this name will be part of the URI used to call your service. Example: “GameServiceSample” would have an address of [https://GameServiceSample.azurewebsites.net](https://XstsServerSample.azurewebsites.net)
9. It is recommended to define a new Resource Group for your App Service with the name ending in “\_rg”
10. Create a new Hosting Plan or use an existing one
11. Click **Create**
12. Once back on the Publish window
13. Select the **Actions** dropdown and select **Rename**
14. Rename the profile and replace *Web Deploy* with **Release**

Azure will now create all the needed resources and configuration for your app service and once completed, Visual studio will deploy the sample to the service. An internet browser should open and direct you to a page that reads “Access Denied: No auth header”. This indicates the service is up and running, but our request didn’t have a valid X-token and we have not yet added the Relying Party certificate to the deployment.

### Adding certificates with private keys to your deployed service

We now need to provide our deployed service access to the private certificates for our Relying Party so that it can properly handle the X-tokens created with that cert. The easiest way to do this is to upload our certs’ .pfx files is as SSL certificates in the Azure settings. The running service can then access the certs and cache them at runtime to be used frequently in handling requests. See [Use an SSL certificate in your application code in Azure App Service](https://docs.microsoft.com/en-us/azure/app-service/app-service-web-ssl-cert-load) for more information.

To upload a private cert and have it deployed to your web service do the following:

1. Log into the Azure portal and go to your App Service’s page
2. Select **TLS/SSL settings**
3. Select **Private Key Certificates (.pfx)**
4. Click **Upload certificate**
5. Navigate to the file and enter the password for the private key
6. Click **Upload**

Your cert should now show up on this page as healthy and show the thumbprint value of the cert. But this is not enough to deploy the cert with our Service in Azure. We will need to add the cert’s thumbprint to the App Settings.

1. Copy the thumbprint of the cert
2. Go to the service’s Application Settings (as previously outlined above)
3. Add a new setting called **WEBSITE\_LOAD\_CERTIFICATES**
4. Add the thumbprint of the cert as the value of this setting. If the setting already exists, add a semicolon to the end of the existing string and then add your new cert’s thumbprint.

### Configuring the App Settings in Azure

You can update and add Azure App Settings through the Azure portal (steps below) or with Visual Studio by using the **Edit App Service Settings…** option of the Publish window outlined in [Deploying to an Azure App Service and debugging remotely](#_Deploying_to_an).

1. Log into the Azure portal and go to your App Service’s page
2. Select **Application settings**
3. Scroll down to the **Application settings** section
4. Click **Add new setting**
5. Name the app setting **RP\_CERT\_THUMBPRINT**
6. Set the value to the thumbprint of your Relying Party certificate you uploaded.
7. Click **Save**

This allows the service at startup to know which certificate it should look to cache and will be used as the service runs.

### Creating a debug deployment to Azure for debugging

To debug your service while running in Azure you will need to setup a debug publish profile and then connect to the service as outlined below.

1. Right click on the GameService project and select **Publish**
2. Select **New Profile**
3. Select **App Service** and the **Select Existing** option
4. Click the **Advanced…** link
5. Set Configuration to “Debug”
6. Expand the **File Publish Options** and check **Remove additional files at destination**
7. Click **Save**
8. Click **Publish**
9. Expand the folder icon that has the name of your Service’s Resource Group
10. Select your service from the expanded list (has a blue circle icon to next to it)
11. Click **OK** and Visual Studio will begin a compile and publish of the new profile
12. Select the **Actions** dropdown on the Publish window and select **Rename**
13. Rename the profile and replace *Web Deploy* with **Debug**

You may run into this common issue during deployment, follow the link for a fix:

[When re-deploying to Azure I get the following error: “Web deployment task failed. (Web Deploy cannot modify the file ‘Microsoft.XboxSecureTokens.dll’ on the destination because it is locked by an external process.’](#_When_re-deploying_to)

### Attaching the Visual Studio Debugger to your Azure App Service

1. Right click on the GameService project and select **Publish**
2. Select the Debug profile you created above from the profile drop-down
3. Click **Publish**
4. Once the publish has completed, open the **Cloud Explorer** window (View -> Cloud Explorer)
5. Expand your subscription list and the **App Services** list
6. **Right Click** on your app service and select **Attach Debugger**

You should now be real-time debugging the instance running in azure. If it is unable to attach the debugger, try sending a request call to one of the service endpoints to wake it up and then re-attach.

## Using the SimpleWinHttp sample to verify your service and NSAL configuration

This will help verify that your NSAL is set up properly and working end to end.

1. Download the SimpleWinHttp sample from the GDK [Samples](https://developer.xboxlive.com/en-us/platform/development/education/Pages/Samples.aspx) on GDN, and extract it to your development PC that has the GDK installed.
2. Open **\GDKSamples\Live\SimpleWinHttp\SimpleWinHttp.sln**.
3. Open **SimpleWinHttp.cpp**.
4. Update the definition of **\*c\_authWebAddress** to be the URI to your deployed web service. For example, https://yourserver.azurewebsites.net/api/getclaims/.
5. Open **MicrosoftGame.config.**
6. Update the value of **TitleId** to match the title ID (hex) of your title from **Partner Center -> Xbox Live -> Xbox Live Setup**
7. Update the **Identity** to also match the Name, Publisher, and Version of your title’s MicrosoftGame.config with the values found in **Partner Center - Game Setup ->Identity Details**)
8. Make sure your dev kit console is in the correct sandbox.
9. Sign in with one of your developer accounts with access to that sandbox on the console
10. Compile and run the sample solution with **F5**
11. Select to make an **Authorized Request** and verify the URI in the output window is the correct URI to your service and check the returned information.

If you get an error while trying to have the client sample talk to your server’s default page, see the [FAQ and troubleshooting](#_I’m_still_having) section later in this white paper.

## Viewing log output from the service

We will cover server logging in more depth during Section 7, but there is a way to see what your service is doing as it runs locally and in Azure. This is especially helpful for determining startup errors when running in Azure where you are unable to attach to the process before the failure.

### Viewing log output when running locally through Visual Studio 2019

1. Compile and run the sample locally as outlined under [Running and debugging the sample locally](#_Running_and_debugging)
2. Select the Output window
3. Select **Debug** from the **Show output from**: dropdown menu
4. **Right click** on this window and uncheck the following to reduce output other than the logging from the server:
   1. **Module Load Messages**
   2. **Module Unload Messages**
   3. **Thread Exit Messages** to greatly reduce the output from other processes in the window.

### Enabling service log output through Azure

1. Open the Azure Portal and go to your App Service’s page
2. Scroll down the list of options to Monitoring
3. Select **App Service logs**
4. Set **Application Logging (filesystem)** to **On**
5. Set **Level** to **Warning\***
6. Click **Save**

\*This makes it so that only Warning level logs and higher are displayed. Much of the server’s output is of type Information to let you know what it is doing, but this also will include a lot more output from other parts of .Net Core.

### Viewing service log output real-time when running on Azure

1. Open the Azure Portal and go to your App Service’s page
2. Scroll down the list of options to Monitoring
3. Select **Log stream**

### Viewing service log output real-time from Visual Studio

1. Open the Cloud Explorer window
2. Navigate to your service under App Services
3. Right click your service
4. Select View Streaming Logs

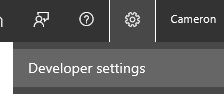
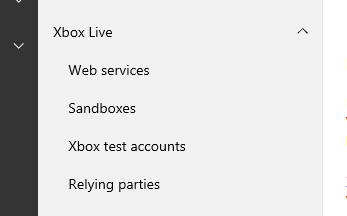
# Section 2 – Service Auth X-tokens

In this section we will introduce you to Service Tokens that are obtained by calling the XSAS service and using a Business Partner Certificate as authentication. The Service token can then be exchanged for a Service Auth X-tokens that allow you to call Xbox Live Services from your Service (backend-to-backend or b2b). When the sample service starts, it will try to cache the business partner certificate specified in the BP\_CERT\_THUMBPRINT app setting and then get and cache a Service Token from that BP Cert in the in-memory cache.

In the next section we will talk about Delegated Auth X-tokens that allow your service to call other Xbox Live services on-behalf-of the user.

For more information on Service tokens see *Obtaining a Service token* in the whitepaper *Calling Xbox Live Services from Your Title Service.*

## Creating a Business Partner certificate

1. Select the **Settings Icon in the upper right and then Developer Settings**
2. Next select **Xbox Live->Web services**
3. Select **New Web Service**
4. Select your publisher from the drop down (there is normally only one, but may be more)\*
5. Enter a name to identify your web service (this can be the host URL or any string name to help you identify where the certificates will be used and manage them)
6. Click **Save**
7. On the Web Service page, you should now see your new Web Service listed
8. Click **Generate Certificate** for your new Web Service
9. Click the **Show Options** link to see a PowerShell script to create the needed key. You can run this script on any machine and then export the certificate to other machines.
10. Run **Windows PowerShell** as an Administrator
11. Copy in the script from step 9 and press **Enter**
12. Copy the resulting string CSP blob string from PowerShell into the text box under **Download the CSP Certificate** (remove any word wrap line breaks from the console window first)
13. If you want this certificate to only work with one specific sandbox you can select one from the drop-down list. Leaving the drop-down on *Select sandbox* will result in a BP cert that can be used in any of your sandboxes and RETAIL (recommended)
14. Click **Download**
15. Save the .cer file on the PC that ran the script in step 11, rename it with the name you used in step 5 to distinguish it against the others.
16. Follow the instructions under **Binding the Certificate**
17. Make sure once imported that the certificate has the private key bound to it (indicated by the small key icon on the top left)  
    
18. Record the first section (8x4x4x4x12 GUID) of the Issued To name on the certificate. We will need this value if you are configuring your service to talk to the Collections service b2b in Section 7
19. Right click on the certificate and select All Tasks->Export.
20. Follow the export Wizard and make sure to select **Yes, export the private key**
21. Select to secure the exported cert with a password
22. Finish exporting the certificate through the wizard

To add the certificate to your service deployment, follow the instructions previously outlined in Section 1 for [Adding certificates with private keys to your deployed service.](#_Adding_certificates_with_1)

### Adding the cert’s thumbprint for initialization to the App Settings

Remember to add this setting to your user-secrets if running or debugging locally as covered in [Enabling Azure App Settings when debugging locally](#_Enabling_Azure_App)

1. Log into the Azure portal and go to your App Service’s page
2. Select **Application settings**
3. Scroll down to the **Application settings** section
4. Click **Add new setting**
5. Name the app setting **BP\_CERT\_THUMBPRINT**
6. Set the value to the thumbprint of your BP cert (visible on the Private Key Certificate (.pfx) screen where you uploaded it.
7. Click **Save**

## Verifying your service can obtain a Service Token

Now that the BP cert is generated and set in our App Settings, we can test that our service is obtaining the BP cert and properly getting Service tokens. To do this you will need to setup log viewing through Azure (covered in [*Viewing log output from the service running in Azure*](#_Viewing_service_log) of Section 1) or by running the service through the debugger on your development PC.

If you are using the Azure log stream, you will want to make sure that the log filter is set to *Information* so that you can see the service tell you when it gets the BP cert and is able to get a Service token. For running locally, all output is displayed to the Output -> Show output from: ASP.NET Core Web Server.

Look for the following in the logs to indicate that the Business Partner cert was properly retrieved, and the service obtained a Service token:

[Information] GameService.Startup: {"cV":"Y96h0OLnuEua5lfp.2","info":"Initializing BP Cert..."}

[Information] GameService.Startup: {"cV":"Y96h0OLnuEua5lfp.1.3","info":"BP Cert cached with thumbprint 274C3E9F63D75A3677C7B403FE2F8B45D72B5CAA"}

[Information] GameService.Startup: {"cV":"Y96h0OLnuEua5lfp.2","info":"Initializing Service Token Cache..."}

[Information] GameService.Startup: {"cV":"Y96h0OLnuEua5lfp.3.1.1","info":"Using BP Cert: DC=BUSINESSPARTNERCERT, O=4dea8aa0-72cc-4cab-888c-cfeeea8d2ebe, OU=xcert.xboxlive.com, CN=4d6777a0-72cc-4cab-888c-cfe23a8d978e\_GameServiceBP\_a9f5e437-d8a2-4646-a914-098272947f5a\_BP"}

[Information] GameService.Startup: {"cV":"Y96h0OLnuEua5lfp.3.2","info":"GameService - Service Token cached eyJhbGciOiJSU0EtT0FF..."}

[Information] GameService.Startup: {"cV":"Y96h0OLnuEua5lfp.3","info":"Configured and ready for requests..."}

# Section 3 – Delegated Auth X-tokens

The code within this section of the sample will show you how to obtain a Delegated Auth X-token that will enable your service to call Xbox Live services on-behalf-of the user. This is valuable for getting friends lists, checking the user’s purchased items directly, and managing consumable products. We will cover calling those services in the next section.

A Delegated Auth X-token is obtained by calling the XSTS service and providing both the Service Token and the Delegation Token that is one of the claims in a user’s client token. These two tokens notify the XSTS service which partner is asking for the token and which user we are asking to make calls on-behalf-of. We also need to provide the Relying Party that maps to the endpoint we want to call with the token. To know which Relying Party to ask for, we get the Xbox Live Endpoints list. This provides us a mapping of URI hosts to which Relying Party to ask for to call that specific host or endpoint.

For more information on Delegated Auth X-tokens see *Obtaining an X-token token* in the whitepaper *Calling Xbox Live Services from Your Title Service.*

The Endpoints are generally static and won’t change, so the sample gets the endpoints data on startup and then stores them in an in-memory database cache. Delegated Auth X-tokens on the other hand are re-usable until their lifetime expires. Rather than make an external call to XSTS each time we want to call a service backend-to-backend (B2B) we will want to cache the tokens. Ideally these tokens would be stored on a distributed cache so that multiple instances of our service can use cached tokens from others. This will be more performant for our service and produce less traffic to the XSTS service.

No additional configuration or setup is required for this section.

# Section 4 – Calling Xbox Live Services

Now that your service can retrieve and cache delegated auth X-tokens, your service is able to call the Xbox Live services directly without having go through the client console. The Sample has provided the b2bFriends endpoint as an example that calls the Xbox Live People service to retrieve a player’s Friends list. You can use this as a template and make modifications to build out other endpoints or functions that would call into the various Xbox Live Service.

The Multiplayer Session Directory service, however, requires additional configuration. The MPSD service can also be called with a Service Auth X-token (no delegation token provided). This allows your service to set and create sessions for your game directly. This is most commonly used in MMO type games or games that have dedicated servers. The provided b2bMPSD endpoint is a bare minimum to check for a session’s info (provided the SCID, session name, and template name) using a Service Auth X-token. This can also be used as a starting point to build additional MPSD functionality into your service.

No additional configuration of your service is required for this section.

For more information on this and especially calling the MPSD service see *Calling Xbox Live Services* in the whitepaper *Calling Xbox Live Services from Your Title Service.*

# Section 5 – Collections and Commerce

The Collections service provides the information about what the user owns or has an entitlement to. This is useful for validating purchased content and managing consumable products directly from your service instead of relying on the information coming from the client. The CollectionsController class provides some basic functionality for querying what a user owns and consuming a specific quantity of a consumable product from the user’s balance. However, your products must be configured to map to your Business Partner certificate in Partner Center before this service will work properly. If not configured when you call the Collections service, you will not get any results back for valid products as outlined in the following forum post: [B2B call to Collections LicensePreview service returns empty results (Partner Center configured game)](https://forums.xboxlive.com/questions/78683/b2b-call-to-collections-licensepreview-service-ret.html)

## Configuring your products for B2B Collections in Partner Center

1. Make sure you have a a Business Partner Certificate and check its "Issued to:" field in the cert info. Note the first set of numbers and letters in GUID format. This is the PublisherID being stamped to your BP Certs, you will need to set your Studio ID to this value in order for b2b Collections calls to return the right results.
2. From Partner Center’s **Overview** page select **Create a new -> Product Group** (if this has not been done already)
3. Assign the product group a Dev Studio. If you have not yet created a Dev Studio in Partner Center you can create one as seen in the dropdown. Use the GUID from step 1 as the Studio ID.   
   A screenshot of a cell phone

   Description automatically generated

A screenshot of a social media post

Description automatically generated

1. Add your game and all its products into the Product Group using the **Add** button

A screenshot of a cell phone

Description automatically generated

1. Click **Save** to finish creating the product group
2. Once that has been done, republish each product in the product group into the sandbox.

## Managing consumables with Transaction IDs

As a part of consuming a quantity of consumables, the sample also provides an example of how to implement a best practice to ensure that the consume transactions are resilient to outages or network issues. When consuming a product, you provide a GUID to track that transaction with the Collections service. If the consume request never returns or is otherwise lost in transit, you can replay the same call with the same GUID to validate if that action whet through or not. If the Collections service has already taken action for this request it will return as successful with the same response as if it had consumed the item, but will not deduct that amount from the user’s balance a second time. The sample is configured to be able to use a persistent SQL database if provided to record and track all consume requests that are in a pending state. If there was a power outage or issue, those requests could then be validated and re-run later to ensure proper recording and results without effect on the user. If an SQL connection string is not provided in the **GameServbicePersistentDB** Application Settings connection string, then the sample will fall back to use an in-memory database and log a warning that this should be on a more persistent database.

To configure an SQL database, see the optional [Appendix A: Setting up an Azure SQL Database](#_Appendix_A:_Setting)

## Validating License Tokens for PC titles

Recently the handling of License Tokens was also added to the Commerce section of the sample. License Tokens are primarily used to validate the client is licensed on a Windows PC device. More information about License Tokens will be available in documentation in a future release.

# Section 6 – Logging & Correlation Vectors

As you may have noticed and was mentioned earlier, the sample service has built-in logging following the guidance of [Logging in ASP.NET Core](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging/?view=aspnetcore-2.2). (See [Viewing log output from the service](#_Viewing_log_output) for instructions of seeing the logs from your deployed or locally running service)The log messages are formatted to be JSON compatible to be more easily searchable by log retrieval tools. The sample also utilizes Correlation Vectors (cV) to help keep track of all the logs related to a specific request or flow of a request through the server (more info below).

Following the guidance found in the article [High-performance logging with LoggerMessage in ASP.NET Core](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging/loggermessage?view=aspnetcore-2.2), the sample is structured so that all of the log message formatting is handled in a centralized location (XstsLoggerExtensions.cs). To fire off a pre-formatted log we simply call the corresponding log API and pass in the needed data. This helps so that when we change the formatting of one log message it is changed for all times the service would log that info. Although the performance gain of using LoggerMessage is small, when doing something as frequent as logging in a cloud environment it adds up to a large amount of saved compute time and cost. Adding additional logging and messages can be done using the existing ones as a template. Note however, that there is a maximum limit of how many variables you can pass to a LoggerMessage before you will get errors.

Correlation Vectors are an open source protocol for tracing a correlation of events through a distributed system based on a lightweight vector clock. By adding this to all your logging messages it is easier to retrieve and search for the logs related to one of the requests made to your service and Microsoft services. The Xbox services used in this sample accept a cV header in the request (ms-cv) and will use an extended version of that cV in all logging related to that request. All responses from these services will also include an MS-CV value in the return headers that is the ending cV for that process on the server. This is especially helpful when debugging or investigating issues that require looking at logs in both yours and Xbox’s logging. If you have an issue, the cV value from the logs on your side will directly correspond to the cV values of the logs needed on the Xbox service side.

For more information on [Correlation Vectors see the Github repository here.](https://github.com/Microsoft/CorrelationVector)

# Appendix A: Setting up an Azure SQL Database

The following information can help you get started on setting up a persistent database to store values and data that you want to retain beyond server instances such as TransactionIDs for consumable validation. The sample has an example in-memory database that is turned on by default but can be migrated to a persistent service based database to live beyond the instance of the running service.

Details about database configuration and accessing the connection string at runtime through the Azure application settings see [Azure .NET Core Application Settings](https://blogs.msdn.microsoft.com/jpsanders/2017/05/16/azure-net-core-application-settings/).

## Creating an Azure SQL Database

Follow the instructions within the following article:  
[Quickstart: Create an Azure SQL database in the Azure portal](https://docs.microsoft.com/en-us/azure/sql-database/sql-database-get-started-portal)

You can also create the database under the same Resource Group as your App Service. For running the sample and testing a Basic pricing tier SQL database should be enough. If you are building your own service and plan roll it out as a production service, you will want to adjust the database capabilities and settings in Azure.

## Initializing the database with Migrations and debugging locally

Now that the database is created, we need to initialize it to have the tables and data that our server will store in the database. This is done by running the app locally and creating migration files that we can then use to initialize the tables in the database. As a part of this we need to add the connection string to our user-secrets file (see [Enabling Azure App Settings when debugging locally](#_Enabling_Azure_App)). This will also allow us to debug with the database on our development PC. For more information on migrations and what to do, please see the following:

[Getting Started with EF Core on ASP.NET Core with a New database](https://docs.microsoft.com/en-us/ef/core/get-started/aspnetcore/new-db?view=aspnetcore-2.1&tabs=visual-studio)

1. Open the Sample solution in Visual Studio
2. Go to **Tools > NuGet Package Manager > Package Manager Console**
3. Navigate in the Package Manager Console to the local directory where GameService.csproj is found
4. In the Package Manager Console’s Default Project dropdown menu select **Microsoft.XboxSecureTokens** (the project the db context we are trying to create lives)
5. Run the following commands to create the needed table structures for the GameServicePersistentDBContext used in the sample.

Add-Migration InitialCreate -context "GameServicePersistentDBContext"

This will start your sample running locally on your PC to build the migration.

1. After the migration has been completed, run the next command in the Package Manager Console to push the migration to your Azure SQL database.

Update-Database -context "GameServicePersistentDBContext"

If you get a connection error stating that your client IP is not allowed to access the database, log into the Azure Portal and navigate to the database. Select **Set Firewall**, add your dev PC’s IP to the allow list, save and try again to complete the database update.

1. Re-deploy your web service from Visual Studio

# Appendix B: Running the sample on Linux

The sample was designed and written in ASP.NET Core so that it would be compatible on both Windows and Linux servers. However, ongoing support for this sample will be targeting Windows on Azure. If you want to run the sample on Linux, you can follow the following guides and articles that we found helpful when setting it up ourselves:

[Quickstart: Create a Linux virtual machine in the Azure portal](https://docs.microsoft.com/en-us/azure/virtual-machines/linux/quick-create-portal)

[Install .NET Core Runtime on Linux Ubuntu 16.04 x64](https://www.microsoft.com/net/download/linux-package-manager/ubuntu16-04/runtime-current)

[Host ASP.NET Core on Linux with Nginx](https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/linux-nginx?view=aspnetcore-2.1&tabs=aspnetcore2x)

During initial development (October 2018) we confirmed the sample worked on both an Azure Linux Web Service and an Azure Linux VM running Ubuntu 16.04. However, we have not re-verified the changes and additions in the latest versions of the sample. During the validation we noted that .NET Core behaves differently with a few API’s on Linux than it does on Windows. These are noted with comments in the code and relevant links for more information on those.

There are some configuration differences in the Setup.cs file for Linux that is activated by defining “LINUX” in the Conditional symbols of the project. These are not required but are commonly added due to how Linux servers generally use a reverse proxy.

# FAQs and troubleshooting

#### When re-deploying to Azure I get the following error: “Web deployment task failed. (Web Deploy cannot modify the file ‘Microsoft.XboxSecureTokens.dll’ on the destination because it is locked by an external process.’

This usually happens when you are going between a Retail and a Debug deployment as the XboxSecureTokens .dll will be updated. To get around this error you will need to stop your service by following the instructions below and then re-deploy.

1. open the **Cloud Explorer** window
2. Expand your subscription list and the **App Services** list
3. **Right Click** on your app service and service and select **Stop**
4. Wait a moment and then try the re-publish again
5. In the **Cloud Explorer** right click on **App Services** and select **Refresh**
6. Now right click on your app service and select **Start**

#### When the client-side sample tries to call our service, it is getting errors from GetTokenAndSignatureAsync(). How can I resolve these errors?

Most of the NSAL or token-related errors from the console start with 0x87DD\*. [Error Codes 0x87DD\* when calling GetTokenAndSignatureAsync or Xbox.Services APIs](https://forums.xboxlive.com/questions/2936/error-codes-0x87dd-when-calling-gettokenandsignatu.html), a post in the Xbox Developer Forums, provides a list of the most common errors and how to resolve them. (Note: if when clicking this link, you go to a general forum page, try pasting the link into a browser address bar to get directly to the page.)

#### HRESULT error 0x800c0019 when trying to call my web service over HTTPS from the console.

This error means that the SSL certificate that you used for HTTPS traffic is not trusted by the console, or the certificate has been misconfigured on the server.

To correct this, first make sure that the certificate you created and used for SSL on your server uses the full domain name of your server. To check, double-click the exported .cer file and look at the **Issued to** value (for example, “Issued to: Server.Contoso.com”)**.**

Although you can add an SSL cert in Partner Center now on the Endpoint definition page, it is recommended that you always use fully trusted SSL certs. These can now be obtained free of charge through entities such as <http://letsencrypt.org>.

#### I’m getting 403 errors when my console tries to talk to my service,

This probably means that the certificates or secrets for your relying party are not configured properly right or you may have entered the wrong certificate or secret key into the Azure Key Vault. You may want to use Fiddler to check your title’s NSAL and debug the service as it attempts to validate the X-token. To check your title’s NSAL do reboot the device to clear the NSAL cache and follow the links below based on the device your title runs on:

[For Xbox Console](https://forums.xboxlive.com/questions/516/is-there-a-way-to-verify-the-nsal-configuration-fr.html)

[For Windows PC](https://forums.xboxlive.com/questions/54883/is-there-a-way-to-verify-the-nsal-configuration-of.html)

#### When trying to obtain a Service token from the XSAS server with my Business Partner certificate I am getting an exception that “The underlying connection was closed"

This exception is probably due to an expired Business Partner certificate. Check the expiration date of your certificate to make sure that the date has not passed. Relying Party certificates will still work with the XSTS system even if they expire, but your Business Partner certificate must be in a good and valid state to be accepted by the Xbox Server Authentication Service (XSAS). XSAS does not give you a 403 when presented with an invalid or expired certificate; instead, it simply drops the connection, which results in an exception and the messaging about a closed connection during debugging. To correct this, simply create a new Business Partner certificate.