Microsoft Store Services 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 Microsoft Store Service sample. The sample provides code for handling the authorization between client and server with Microsoft Store Services through Azure Active Directory (AAD), Access Tokens, and User Store IDs. More information on the auth flow to talk to the Microsoft Store Services can be found in the GDK Commerce documentation and [Manage product entitlements from a service - UWP applications | Microsoft Docs](https://docs.microsoft.com/en-us/windows/uwp/monetize/view-and-grant-products-from-a-service).

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** |
| March 2021 | 1.0 | * Initial Release |
| April 2021 | 1.01 | * Fixed a few logic issues with Clawback * Updated sample service code to use the architecture of the public GitHub repo for Microsoft.StoreServices |

# Introduction

The Microsoft Store Services provide partner services the ability to query and manage user entitlements and products within the Xbox or Windows Store. The primary services are as follows:

* Collections – Query user owned products and manage consumable products.
* Purchase – Grant free products directly, query and manage user subscriptions, and query for refunds of consumable products already fulfilled through the Collections service.

This configuration guide will help you setup the existing Microsoft Store Service sample so that you can make test calls and see how to authorize and integrate your own services with these Microsoft Store Services using the Microsoft.StoreServices namespace. The Public GitHub repo with latest updates of this namespace can be found at <https://github.com/microsoft/Microsoft-Store-Services>.

# Section 1 – Configuring your AAD, Application, and Products

The Microsoft Store Services use a set of bearer tokens that identify your service and the user for whom your service is asking about. These are referred to as Access Tokens and User Store Ids within the sample and **Microsoft.StoreServices**. The Access Tokens are created using an Azure Active Directory (AAD) defined for your service / publisher and the User Store Ids are provided by the client to identify the user signed into the store.

Before your service will be able to make calls to the Microsoft Store Services, you will need to configure an Azure Active Directory and register an app under that AAD. Additionally, you will need to configure your products in Partner Center to allow your AAD visibility to those products when calling the Microsoft Store Services. Finally, you will need to generate the Access Tokens and User Store Ids depending on which Microsoft Store Service you are wanting to call.

The article [Manage product entitlements from a service | Microsoft Docs](https://docs.microsoft.com/en-us/windows/uwp/monetize/view-and-grant-products-from-a-service#overview) has the latest steps and info to setup your service configurations with AAD. Refer to the article’s Section 1 and 2 before deploying the sample service as the service will need your AAD Tennant Id, Client Id, and Client Secret. Additionally, you can review the Game Development Kit (GDK) documentation in the section *Manage products from your services* under *Overviews and How-tos* / *Commerce*.

## Configuring your products to be visible to your service

Once you have your AAD App registered (also referred to as the Client Id), you need to specify that Id with each of your products you want to be visible within calls to the Store Services when authenticated through this Id.

1. Navigate to the parent product (Game or App) in [Partner Center](https://partner.microsoft.com/en-us/dashboard).
2. Select the **Services -> Product collections and purchases** option on the menu.
3. On the Product collections and purchases page, enter the AAD App Id that you created previously into one of the available **Client ID** fields.
4. Click **Save.**
5. Re-publish the App or Game to your development environment.

**NOTE**: Enabling the Client ID for your service on the parent App / Game will enable all Add-ons under that product to be visible in calls to the Microsoft Store Services.

## Creating Access Tokens

Following the previously linked article, see [Step 3: Create Azzure AD Access Tokens](https://docs.microsoft.com/en-us/windows/uwp/monetize/view-and-grant-products-from-a-service#step-3-create-azure-ad-access-tokens). As the article notes, it is important to understand that there are 3 different Access Tokens used which are defined by which audience URI you request. Your service will always need a Service Access Token (audience: <https://onestore.microsoft.com>) that will be used as the Bearer token for all calls to the Microsoft Store Services. With the Client ID, Client Secret, and Tennant ID you can use Fiddler to manually make an access token request as your service will to validate that they are configured properly.

The sample uses the **Microsoft.StoreServices.IAccessTokenProvider** class to generate these tokens with the provided Audience URI, Tenant Id, Application Id, and Client Secret that were obtained when configuring your service in AAD.

## Sample Implementation of an Access Token Cache

For simplicity and how a deployed game service would more optimally handle Access Tokens, the MicrosoftStoreService sample uses the **Microsoft.StoreServices.CachedAccessTokenProvider** class. This class uses an in-memory cache to manage the needed Service, Collections, and Purchase Access Tokens that will be needed for most calls. If a token is expired or not found in the cache, it creates a new **AccessToken** object for the needed audience.

This is more efficient and reduces the number of calls to generate tokens as each instance of a service can re-use the Access Tokens until their expire time specified when it is created.

# Section 2 – Building and Deploying the Sample Service

With the required AAD Ids and secret the sample can now be run to obtain the Access Tokens that will be needed to authenticate with the Microsoft Store Service.

**NOTE:** In a future update we will release the Microsoft.StoreServices library as a NuGet package. However, at this time the sample requires you to pull down the Microsoft.StoreServices github repo next to the sample’s repo. Example:

D:\Repos\Microsoft-Store-Services\  
D:\Repos\Microsoft-Store-Services-Sample\

## 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 also possible to debug and deploy through Azure, which will be covered later in this section.

*Note: This version of the sample requires VS 2019 and .NET Core 5.0*

### Building the sample in Visual Studio 2019

1. [Download and install the latest .NET Core 5.0 SDK](https://dotnet.microsoft.com/download/dotnet/current)
2. Open **MicrosoftStoreServicesSample.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 **ServiceConstants.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 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. This allows you to specify your AAD credentials and secrets without hard coding them into the service. 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 **MicrosoftStoreService** project and then select **Manage User Secrets**
2. In the **secrets.json** file add the following example to simulate the App Setting you would have with a full deployment

{

"AAD\_CLIENT\_ID": "[Your registered Client App Id]",

"AAD\_TENANT\_ID": "[Your Tenant Id (GUID)]",

"AAD\_CLIENT\_SECRET": "[Secret generated with your Client App]"

}

### Running and debugging the sample locally

1. Follow the steps above and ensure that the sample builds properly.
2. Open **MicrosoftStoreServicesSample.sln** in Visual Studio 2019
3. Right click on the **MicrosoftStoreService** 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. Compile and run the server locally.
3. Once the server is running and ready to receive requests go to Fiddler’s **Composer** tab and select the **Scratchpad** sub-tab.
4. Use the following example request to have the server give you its cached Collections and Purchase Access Tokens. Make sure to update port to be the one that your debug server is running on step 5 of [Running and debugging the sample locally](#_Running_and_debugging).

GET https://localhost:5001/collections/AccessTokens HTTP/1.1

Connection: Keep-Alive

Content-Type: application/json

Host: localhost:5001

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

# Section 3 – Generating User Store Ids

Now that the service is up and running, we can obtain the needed Access Tokens that your app running on the client will need to generate the User Store Ids. There are two specific User Store Ids you can crate depending on which Microsoft Store Service your service will be calling:

* **UserPurchaseId** – Purchase, Recurrence, Clawback
* **UserCollectionId** – Collections, Query, Consume
* NOTE: We use the term User Store ID which is the term these keys and APIs are called in the Game Development Kit APIs. In the older UWP Store APIs they are called Microsoft Store ID keys.

To generate these, you must send the corresponding Access Token the service has generated down to the client (Ex: Purchase Access Token to generate a UserPurchaseId). The client then calls the appropriate API such as XStoreGetUserPurchaseId (GDK) or StoreContext.GetCustomerPurchaseId (UWP) with a Purchase Access Token. The API will return a User Purchase Id that represents the user signed into the Microsoft Store App on the client. The client then needs to transmit this Id securely back to your service so that it can be used used to call the Microsoft Store Services on-behalf-of that user.

Continuing in the article linked above, see [Step 4: Create a Microsoft Store ID Key](https://docs.microsoft.com/en-us/windows/uwp/monetize/view-and-grant-products-from-a-service#step-4-create-a-microsoft-store-id-key) and specifically the Diagram located in the article to better understand how to obtain the required Microsoft Store ID key to call the Microsoft Store Services.

# Section 4 – Calling the Microsoft Store Services

We now have all the needed Access Tokens and User Store Ids to properly authenticate with the Microsoft Store Services and make calls to them.

The sample is structured so that the caller sends up the needed User Store Ids. In your deployed service, this handshake would only need to happen on initial login and before a consume fulfillment request (to ensure we have the proper Ids for Clawback reconciliation later).

You can use Fiddler to simulate calls to the sample service with the User Store Ids generated on your client code. Here are a few example ways to call the service. See the CollectionsController.cs and PurchaseController.cs files for the latest list of endpoints and required POST body to call them:

**NOTE**: you must provide some sort of unique identifier for the user in the Authorization header of the call. In your own service, you should properly be using authentication tokens and the user id would be a claim or part of that secure method. The sample is only using this for simplicity.

POST [https://localhost:5001/collections/query HTTP/1.1](https://localhost:5001/collections/query)

Connection: Keep-Alive

Authorization: UniqueIdforUser

Content-Type: application/json

Content-Length: 1670

Host: localhost:5001

{

"UserCollectionsId": "eyJ0eXAiOiJ..."

}

POST [https://localhost:5001/collections/consume HTTP/1.1](https://localhost:5001/collections/consume)

Connection: Keep-Alive

Authorization: UniqueIdforUser

Content-Type: application/json

Content-Length: 1670

Host: localhost:5001

{

"ProductId": "9PFL4RQTB1P6",

"Quantity": 1,

"UserCollectionsId": "eyJ0eXAiOiJKV1Qi...",

"UserId": "UniqueIdforUser",

"UserPurchaseId": "eyJ0eXAiOiJKV1QiLCJhbGc..."

}

If the sample service is working and your products configured properly, the Query endpoint should return the items that the user who’s User Collections Id was supplied should be returned in a formatted list by the Sample:  
  
 Game 9NBLGGH4W2V9 acquired by purchase

Durable 9MZ0MGGFPLTP acquired by subscription

Consumable 9MT5TGW893HV acquired by purchase with a balance remaining of 10

Consumable 9PFL4RQTB1P6 acquired by purchase with a balance remaining of 1

Game 9NTL0QDWZ4FS acquired by purchase

# Section 4 – Deploying the sample to Azure

**NOTE**: This is a sample, and although it can be used as an example and some code used to build a running service, the sample is not hardened, secure, or validated to run as a large, distributed service in a production environment. You should always do your own review and rely on your own security and performance reviews that would meet the need of your services.

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 for an end-to-end example of what your own services can do.

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 **MicrosoftStoreServicesSample** 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: “StoreServiceSample” would have an address of [https://StoreServiceSample.azurewebsites.net](https://storeservicesample.azurewebsites.net/)
9. Create a new Hosting Plan or use an existing one.
10. Click **Create**
11. Once back on the Publish window, select the **Actions** dropdown and select **Rename**
12. 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.

### 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 **AAD\_CLIENT\_ID.**
6. Set the value to your registered App Id.
7. Click **Save**
8. Repeat steps 4-7 to add settings for **AAD\_TENANT\_ID** and **AAD\_CLIENT\_SECRET**

This allows the service at startup with the needed values and secrets to get the Access Tokens.

### 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 **MicrosoftStoreService** 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 **MicrosoftStoreService** 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.

## Viewing log output from the service

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 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 (LoggerExtensions.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 pending consume transactions, consume history for Clawback reconciliation, etc. 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 MicrosoftStoreServicesSample.csproj is found
4. In the Package Manager Console’s Default Project dropdown menu select **MicrosoftStoreServicesSample** (the project the db context we are trying to create lives)
5. Run the following commands to create the needed table structures for the ServerDBContext used in the sample.

Add-Migration InitialCreate -context "ServerDBContext"

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 "ServerDBContext"

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.

# 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 .dll files 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.**