Using Azure Service Bus to Provide Messaging Between Azure SQL MI Instances



2020/01/24

Version 1.0

Prepared by

**Mitch van Huuksloot**

Data Migration Jumpstart Team

Azure Data

Table of Contents

[Disclaimer 1](#_Toc20823378)

[Introduction 2](#_Toc20823379)

[Create the Target Service Bus Topic 3](#_Toc20823380)

[Building the CLR Trigger 6](#_Toc20823381)

[Building the CLR User Defined Function 8](#_Toc20823382)

[Using the CLR UDF with Service Broker 9](#_Toc20823383)

[Processing the Azure Service Bus Topic Message 11](#_Toc20823384)

[Testing Message Processing End-to-End 17](#_Toc20823385)

[Performance Testing 21](#_Toc20823386)

[Next Steps – Additional Design Considerations 22](#_Toc20823387)

# Disclaimer

Any Sample Code, High Level Architecture, Migration Dispositions and Guidelines in this document or included with this document were developed by Microsoft Corporation technical architects.  Because Microsoft must respond to changing market conditions, this document should not be interpreted as an invitation to contract or a commitment on the part of Microsoft.

Microsoft has provided high level guidance in this artifact with the understanding that the reader would undertake detailed design and comprehensive reviews of the overall solution before the solution would be implemented/delivered.  Microsoft is not responsible for the final implementation undertaken.

MICROSOFT MAKES NO WARRANTIES, EXPRESS OR IMPLIED WITH RESPECT TO THE INFORMATION CONTAINED HEREIN.

# Introduction

One of our customers has a complex cross-server multi-master replication framework built on Service Broker as the message transport. As of September 2019, Service Broker in Azure SQL Managed Instance is local only, so this issue is impeding their move to Azure.

The Data Migration Jumpstart (DMJ) team was asked to explore design alternatives and develop a Proof of Concept (POC) to mitigate this issue. Assuming that the customer wants to stay with SQL Server and not explore [Azure Cosmos DB Global Replication](https://docs.microsoft.com/en-us/azure/cosmos-db/distribute-data-globally) (to reduce initial re-development costs), the following document explores what can be done from SQL Server.

Azure has several managed message passing services that could be leveraged; including [Event Hubs](https://azure.microsoft.com/en-us/services/event-hubs/) and [Service Bus](https://azure.microsoft.com/en-us/services/service-bus/). Since this scenario requires the passing of [messages vs. events](https://docs.microsoft.com/en-us/azure/event-grid/compare-messaging-services) we chose to use to use a Service Bus for our proof of concept. Service Bus has the concepts of [queues and topics](https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-messaging-overview), we chose a topic because it has the ability to have multiple subscribers to a single topic, in the actual implement, the choice would depend on if you have multiple subscriber applications processing messages (topic) or a single receiver application (queue).

For a SQL Server connection to send a message to an Azure Service Bus, a Common Language Runtime (CLR) [User Defined Function (UDF)](https://docs.microsoft.com/en-us/sql/relational-databases/user-defined-functions/create-clr-functions?view=sql-server-2017) or [trigger](https://docs.microsoft.com/en-us/sql/relational-databases/triggers/create-clr-triggers?view=sql-server-2017) is required. Based on the .NET libraries available in SQL Server, the [.NET Service Bus driver](https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-dotnet-get-started-with-queues) is not a good choice, while using the less performant [REST Interface](https://docs.microsoft.com/en-us/rest/api/servicebus/service-bus-runtime-rest), which has a lot fewer dependencies, is much easier.

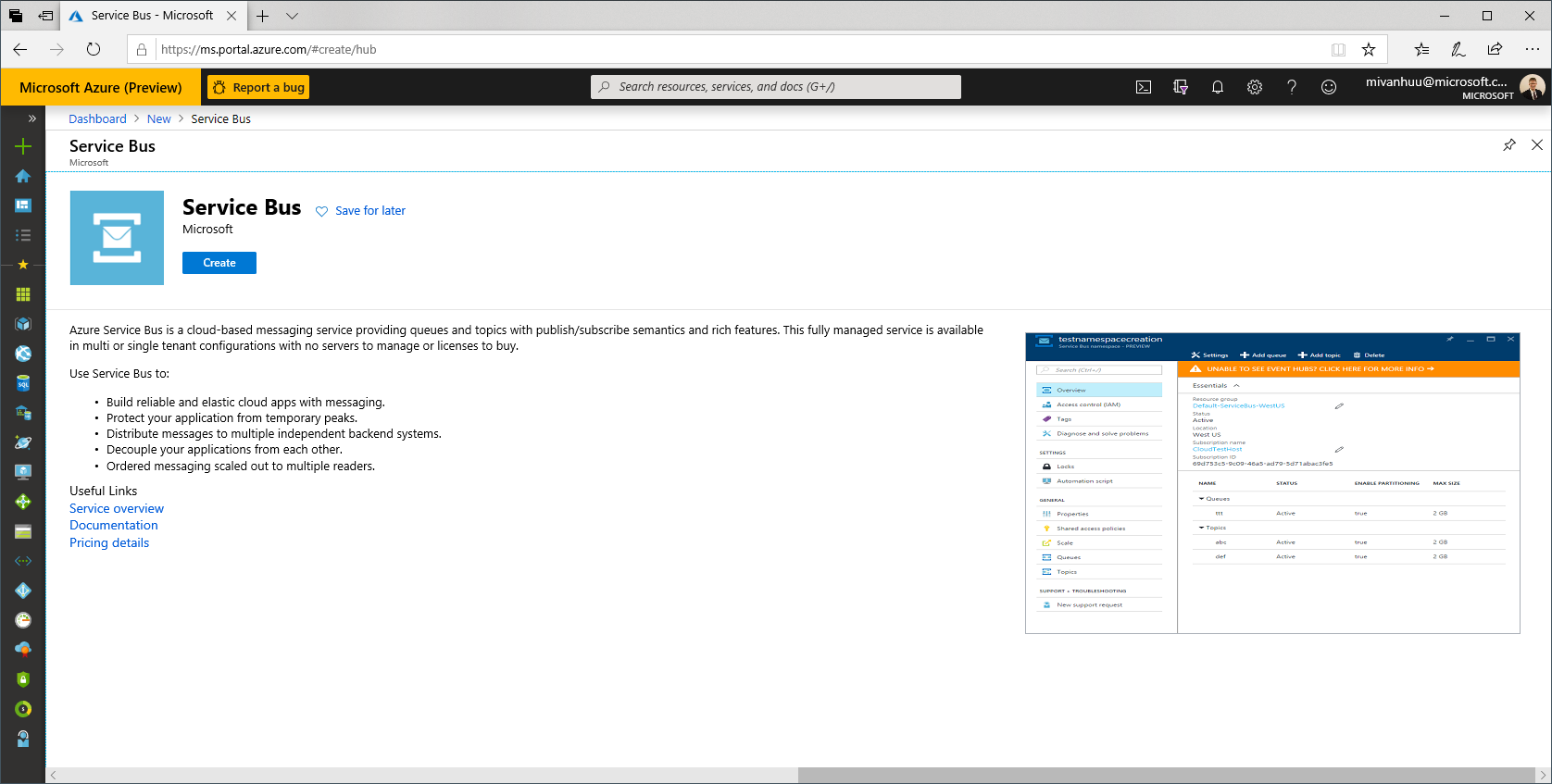
For the POC, we considered three alternative designs;

1. A CLR trigger that processes insert, update and delete actions and sends the appropriate message to the Service Bus topic.
2. A CLR UDF that is used in a TSQL DML trigger to create and send a message to the Service Bus topic
3. A DML trigger that creates and sends a message to the local Service Broker, which sends the message to Service Bus from the Activation Proc.

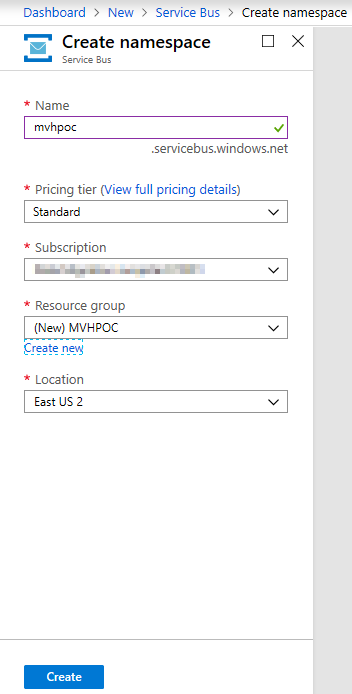
The third option was the most complex, and required the most setup, but it may be required to reduce the overhead/latency on the user making the original data change, depending on the overhead and performance of options 1 and 2.

# Create the Target Service Bus Topic

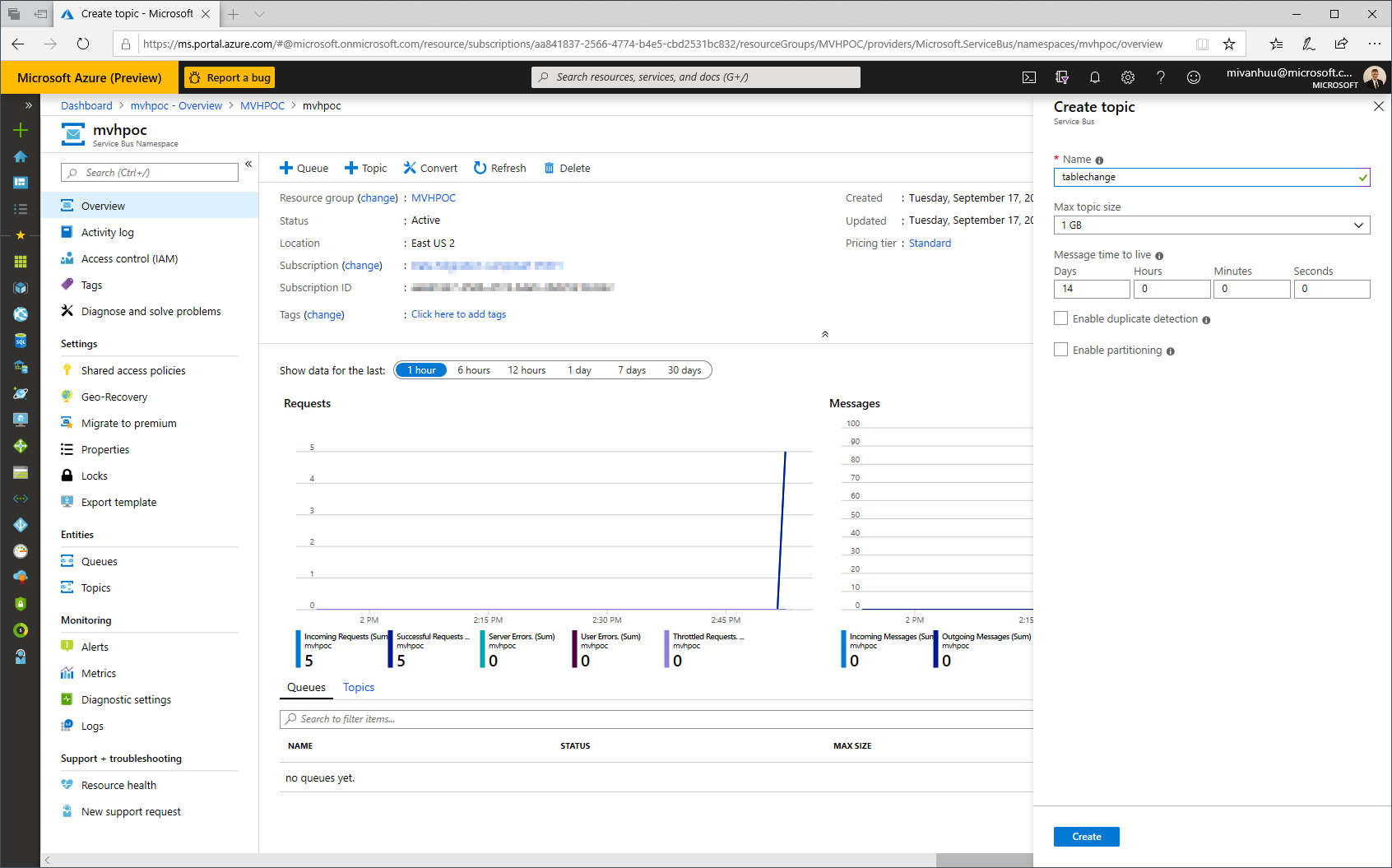
To create an Azure Service Bus Topic and configure Subscribers, open the Azure portal and click on the + to create a new resource. Search for Service Bus – which will give you the following;



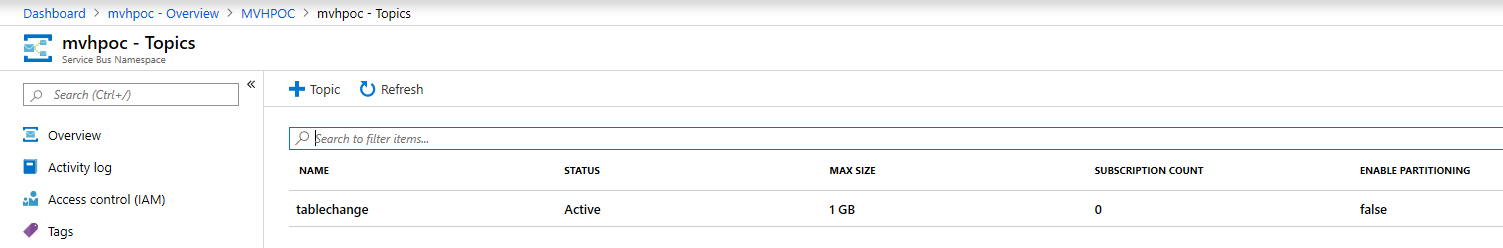
Click on Create and you will be prompted to create a Namespace – enter a unique name, choose a pricing tier, a subscription, a resource group (or create a new one) and an Azure region. Click Create.



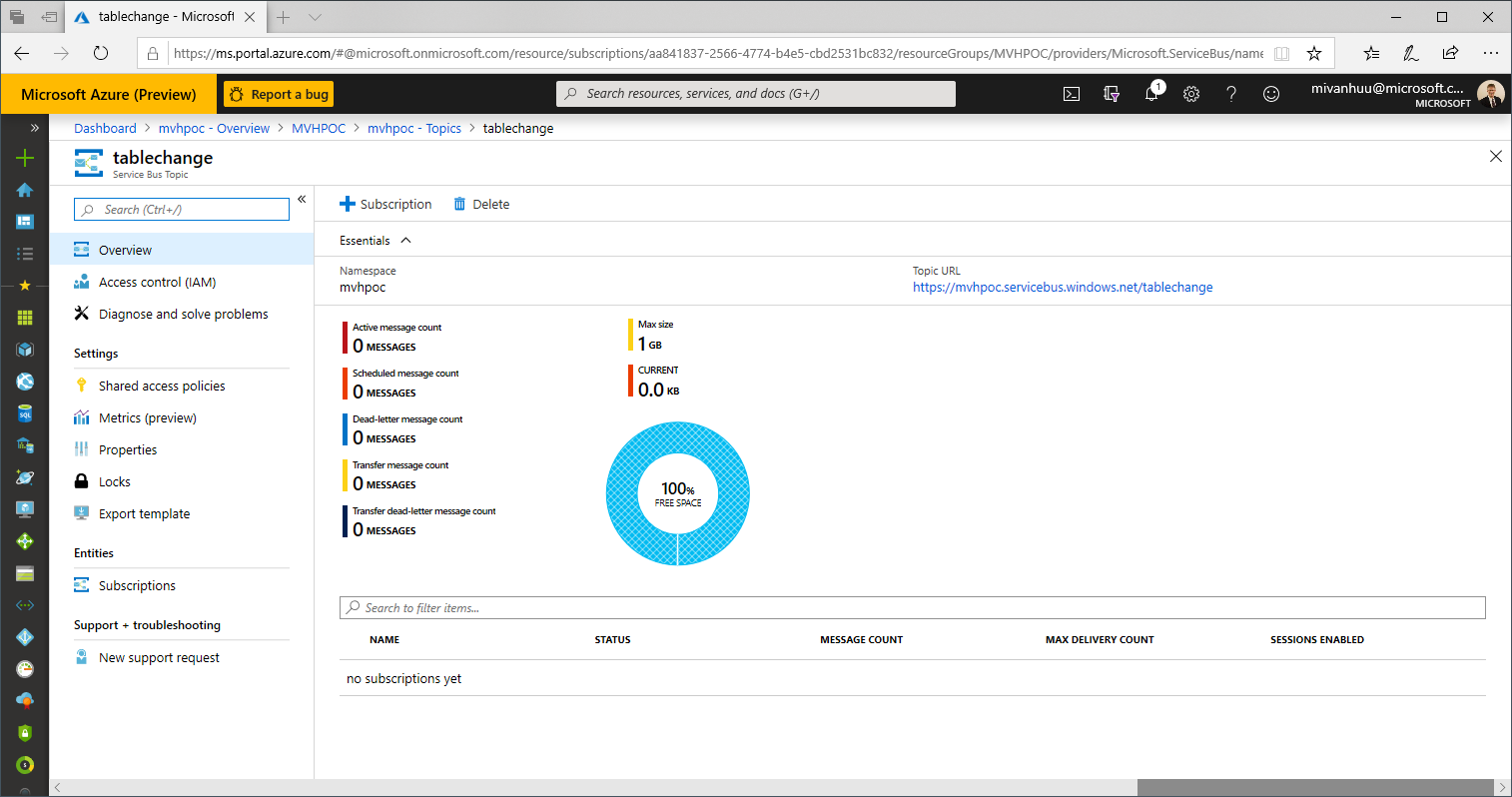
Open the new Service Bus Name Space and click on + Topic to create a new topic;



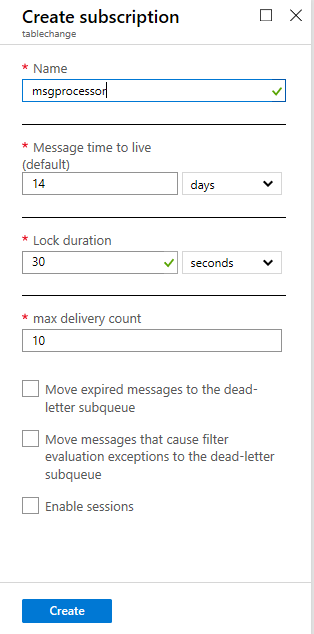
Click Create and once provisioned, select Topic under Entities on the left and the new topic will appear.



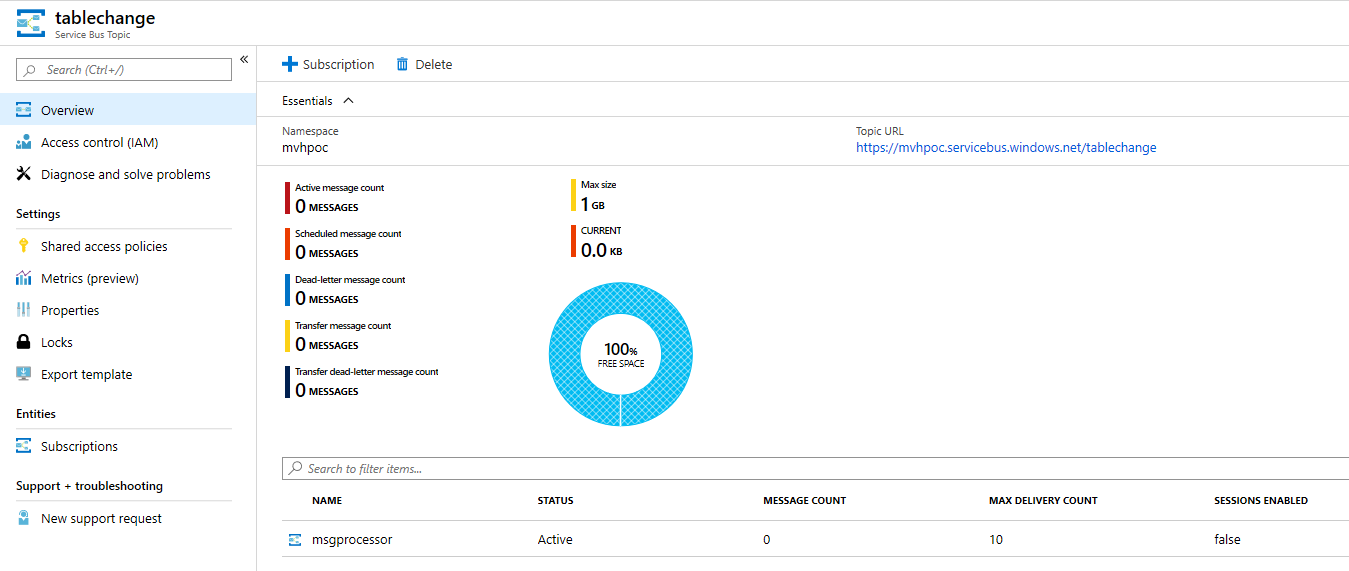
Click on the topic to configure subscriptions;



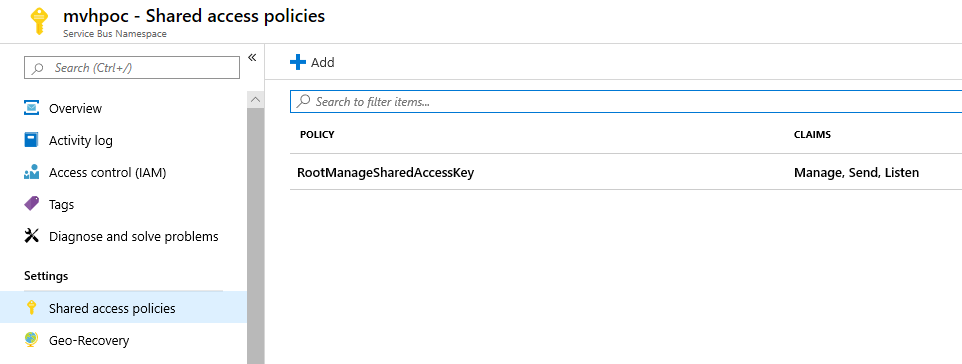
Copy and save the Topic URL. Create a subscription;



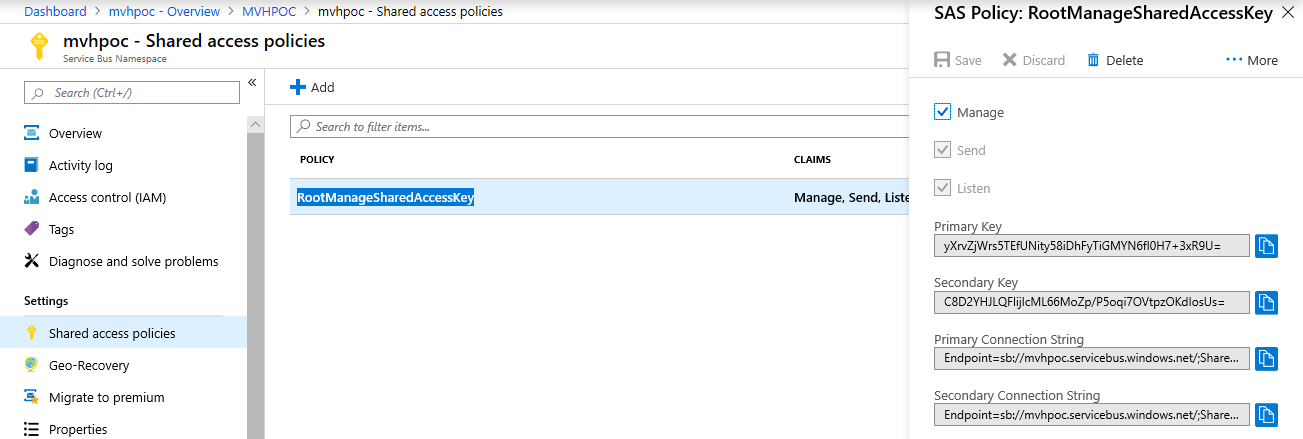
Which should give you something that looks like;



You need SAS keys to access the service. To make things easy, capture the information for the RootManageSharedAccessKey;



Click on the RootManageSharedAccessKey policy and copy the Primary Key and save it somewhere. You will need the Namespace, the SAS key policy name (keyname), the Primary Key, the Primary Connection String and the Topic URL later.



# Building the CLR Trigger

To build a CLR trigger, create a Visual Studio project that supports a .NET DLL as output. The function that SQL Server calls needs to be static with no return value or parameters. The function can be decorated with a SqlTrigger attribute, but this associates the trigger with a specific table (Target), and we want the same CLR trigger to be able to handle multiple tables. The sample project can be used “as is” or you can create a new project and copy the appropriate code from the sample. Note that you will have to update the code with your own Service Bus information.

[SqlTrigger(Name = @"UsersAudit", Target = "[dbo].[users]", Event = "FOR INSERT")]

The CLR trigger infrastructure in SQL Server, calls your .NET code with a SqlContext object which includes a reference to a SqlTriggerContext object. Unfortunately, neither of these objects has the object name or ID of the table that fired the trigger, the design appears to assume that triggers and tables have a 1:1 relationship, so the CLR trigger implicitly knows what table it is attached to. One suggested work around to get the current table name/id appears to be the somewhat questionable use of the currently locked table information. In complicated transactions, this simple table detection technique would likely fail. Instead, we came up with a slightly more expensive hash matching mechanism that uses the inserted/deleted schema to find the matching table in the database.

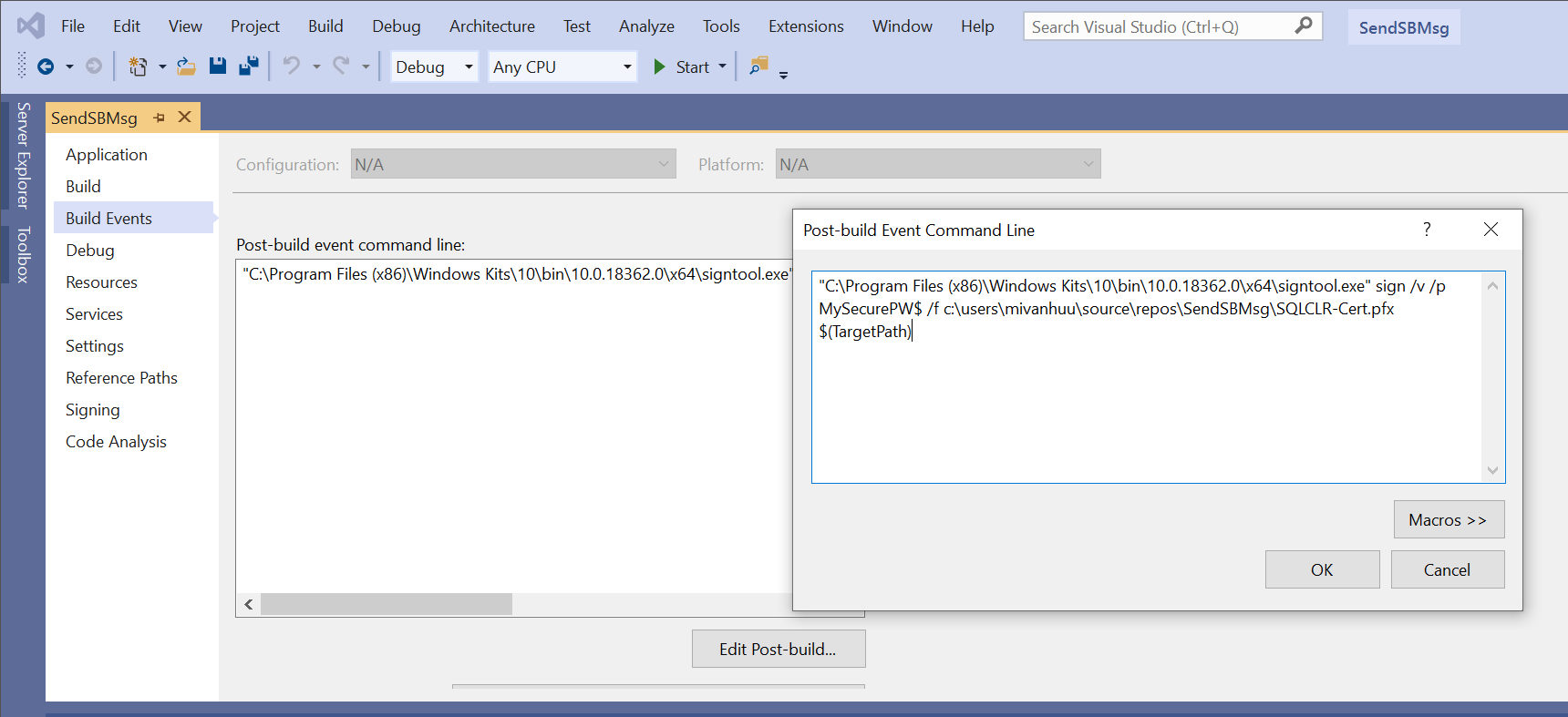
To optimize the interactions between the trigger and the database, the hashing of the column names from the inserted or deleted virtual tables is interleaved with the building of the actual payload message. It does make the code slightly less readable, but it is more efficient. The C# POC code sample represents the table changes in XML, to be consistent across the 3 different design approaches, since Service Broker doesn’t natively support validated JSON messages (at this point - indeed, we could have selected no validation and used JSON). Since the TSQL DML trigger code uses a “select” statement with output formatted by “FOR XML PATH”, this dictates the format of the XML string in the C# code.

The trigger infrastructure in SQL Server provides “INSERTED” and “DELETED” virtual tables to track any changes to the table. Insert and update actions will populate the “INSERTED” virtual table with one or more rows, while update and delete actions will populate the “DELETED” virtual table with one or more rows. The sample code assumes that the downstream processing will be doing an “upsert” (insert if the row doesn’t exist, update if it does) and therefore creates the same message for both insert and update. The delete processing passes all the column values in the message created, which could be optimized to pass only the primary key value(s), if the table has a primary key.

To pass the message to the Service Bus topic using the REST api requires a SAS token string and an HTTP POST of the UTF8 encoded message to the REST endpoint. The POC sample borrows some code from various sites on the internet, primarily from [jsomers.be](https://jsomers.be/archive/2018/12/20/sending-messages-to-azure-from-sql-server-part-1). Once the message is send, the return status is check and any error status is returned to the caller.

Installing CLR components because more complex in SQL Server 2017, with the introduction of the “CLR strict security” configuration option. This option is enabled by default and requires all assemblies to be signed and a login based on the certificate with the “UNSAFE ASSEMBLY” permission. There is a great series by Solomon Rutzky on [Sql Quantum Leap](https://sqlquantumleap.com/2017/08/07/sqlclr-vs-sql-server-2017-part-1-clr-strict-security/), that describes the required [steps](https://sqlquantumleap.com/2017/08/09/sqlclr-vs-sql-server-2017-part-2-clr-strict-security-solution-1/) to create a certificate to sign the assembly and create the associated login with the required permissions. The result of these steps were captured in the CertStuff folder, we recommend you generate your own certificates and not reuse the ones in the sample.

The sample project has a post build step to sign the output, which makes deployment easier, since you don’t have to manually sign the assembly. If you don’t already have it installed, you will need to install a Windows SDK to get the signtool.exe application.



To actually install the CLR component into SQL Server, you use the CREATE ASSEMBLY command;

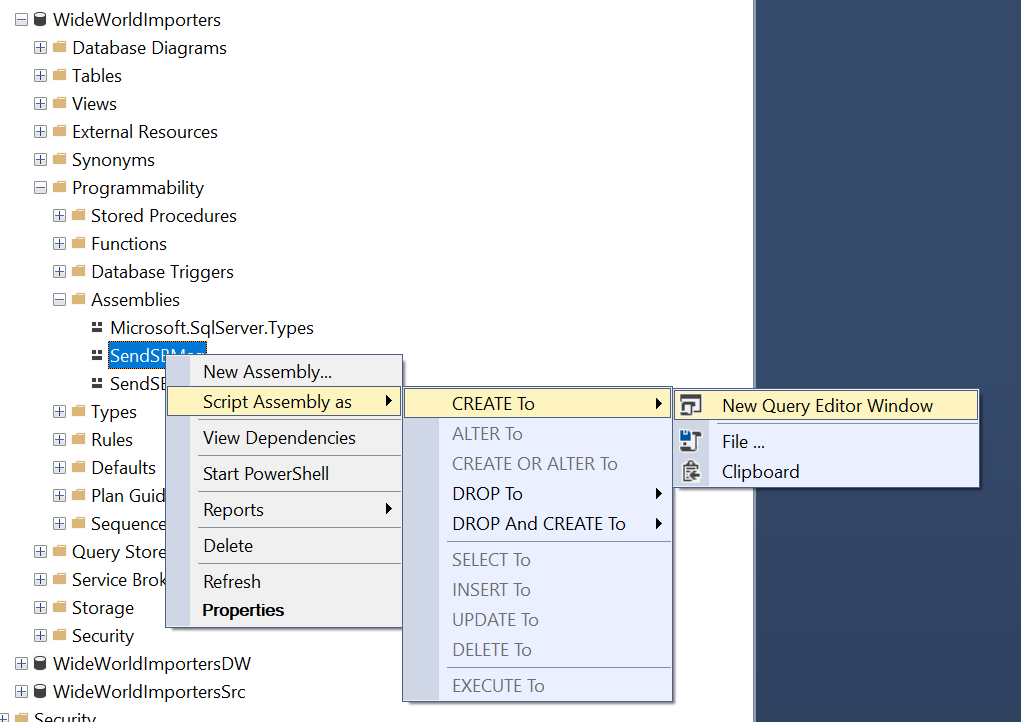
CREATE ASSEMBLY SendSBMsg FROM 'C:\repos\SendSBMsg\bin\Debug\SendSBMsg.dll' WITH PERMISSION\_SET = UNSAFE;

You can then create a trigger that calls the CLR trigger component;

CREATE TRIGGER testclr\_SO ON Sales.Orders

FOR INSERT, UPDATE, DELETE AS EXTERNAL NAME SendSBMsg.CLRTriggers.trgSendSBMsg;

To install the component in Azure SQL Managed Instance – load the component into SQL Server as above and use SSMS to create a script for the CLR assembly – you can then run the script on MI.



# Building the CLR User Defined Function

The CLR UDF is just a subset of the CLR Trigger code, it just reuses the logic to send the message to the Service Bus. Unlike the CLR trigger, the UDF has an input parameter (the message body) and a return value (the HTTP status code and description), both defined as native SQL Server data types. The SqlFunction attribute is also used to set additional properties for the UDF including the data access safety level and whether it is deterministic. You can use the Visual Studio project “as is”, but you will need to update the Service Bus information with your own.

To install the component into SQL Server, will require it to be signed in the same way that the CLR trigger was. Again, the CREATE ASSEMBY command is used to load the assembly;

CREATE ASSEMBLY SendSBMsgUDF FROM 'C:\repos\SendSBMsgUDF\bin\Debug\SendSBMsgUDF.dll' WITH PERMISSION\_SET = UNSAFE;

To be able to call the component you use the CREATE FUNCTION command;

CREATE FUNCTION SendSBMsg(@msgbody nvarchar(max))

RETURNS nvarchar(max)

The trigger to call the CLR component is more complicated, since the logic to build the message is in the trigger instead of the UDF. First it checks what change caused it to be called, by checking the number of rows in the inserted virtual table. If there are rows, it is an insert or update operation, otherwise it is a delete operation. The table columns and values are serialized with “FOR XML PATH” as noted above for ease of coding and to make the trigger as performant as possible. The last step is to send the message using the CLR UDF.

create trigger trgsenddirect\_SO on Sales.Orders

for insert, update, delete as

declare @inserted int

select @inserted=count(\*) from inserted

declare @msgbody nvarchar(max)

if (@inserted > 0)

begin

set @msgbody = (select \* from inserted for xml path)

select @msgbody = '<message><server>' + REPLACE(@@SERVERNAME, '\', '%5C') + '</server><action>insert</action><table>Sales.Orders</table><rows>' + @msgbody + '</rows></message>'

end

else

begin

set @msgbody = (select \* from deleted for xml path)

select @msgbody = '<message><server>' + REPLACE(@@SERVERNAME, '\', '%5C') + '</server><action>delete</action><table>Sales.Orders</table><rows>' + @msgbody + '</rows></message>'

end

select dbo.SendSBMsg(@msgbody)

# Using the CLR UDF with Service Broker

Sending a message to Service Broker requires some setup. Jonathan Kehayias wrote a great article on [SQLPerformance.com](https://sqlperformance.com/2014/03/sql-performance/configuring-service-broker) that simplifies the process. The POC sample leverages Jonathan’s code to create the message types, the contract, the processing and request queues and the activation procedure (in the sample as \SendSBMsgUDF\ConfigServiceBroker.sql). The POC sample takes the activation procedure and extends it to call the CLR UDF to forward the message to the Azure Service Bus.

The TSQL DML trigger on the table needs to be changed to pass the message to Service Broker;

create trigger trgSBtest\_SO on Sales.Orders

for insert, update, delete as

declare @inserted int

select @inserted=count(\*) from inserted

declare @msgbody nvarchar(max)

if (@inserted > 0)

begin

set @msgbody = (select \* from inserted for xml path)

select @msgbody = '<message><server>' + REPLACE(@@SERVERNAME, '\', '%5C') + '</server><action>insert</action><table>Sales.Orders</table><rows>' + @msgbody + '</rows></message>'

end

else

begin

set @msgbody = (select \* from deleted for xml path)

select @msgbody = '<message><server>' + REPLACE(@@SERVERNAME, '\', '%5C') + '</server><action>delete</action><table>Sales.Orders</table><rows>' + @msgbody + '</rows></message>'

end

EXECUTE dbo.SendBrokerMessage

@FromService = N'RequestService',

@ToService = N'ProcessingService',

@Contract = N'AsyncContract',

@MessageType = N'AsyncRequest',

@MessageBody = @msgbody

While the activation proc needs to forward the message to the Service Bus Topic;

CREATE PROCEDURE dbo.ProcessingQueueActivation AS

BEGIN

SET NOCOUNT ON;

DECLARE @conversation\_handle UNIQUEIDENTIFIER;

DECLARE @message\_body XML;

DECLARE @message\_type\_name sysname;

WHILE (1=1)

BEGIN

BEGIN TRANSACTION;

WAITFOR

(

RECEIVE TOP (1)

@conversation\_handle = conversation\_handle,

@message\_body = CAST(message\_body AS XML),

@message\_type\_name = message\_type\_name

FROM ProcessingQueue

), TIMEOUT 5000;

IF (@@ROWCOUNT = 0)

BEGIN

ROLLBACK TRANSACTION;

BREAK;

END

IF @message\_type\_name = N'AsyncRequest'

BEGIN

-- Handle complex long processing here

declare @resp nvarchar(max)

SELECT @resp = dbo.SendSBMsg(cast(@message\_body as nvarchar(max)))

-- Build reply message and send back

DECLARE @reply\_message\_body XML = N'<response>' + @resp + '</response>';

SEND ON CONVERSATION @conversation\_handle

MESSAGE TYPE [AsyncResult] (@reply\_message\_body);

END

-- If end dialog message, end the dialog

ELSE IF @message\_type\_name = N'http://schemas.microsoft.com/SQL/ServiceBroker/EndDialog'

BEGIN

END CONVERSATION @conversation\_handle;

END

-- If error message, log and end conversation

ELSE IF @message\_type\_name = N'http://schemas.microsoft.com/SQL/ServiceBroker/Error'

BEGIN

-- Log the error code and perform any required handling here

-- End the conversation for the error

END CONVERSATION @conversation\_handle;

END

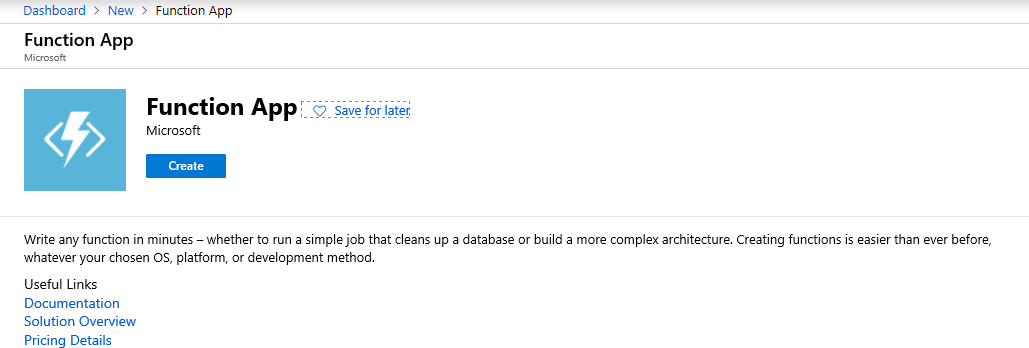
COMMIT TRANSACTION;

END

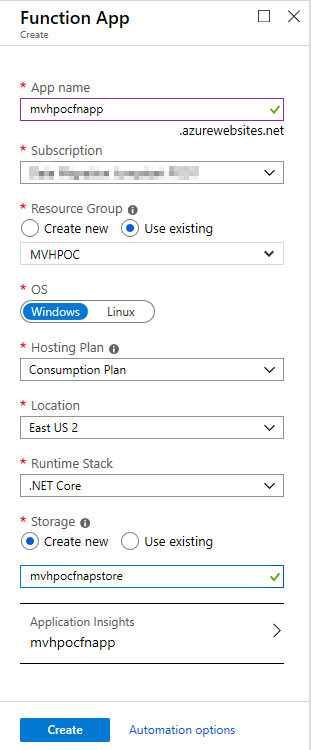
END

# Processing the Azure Service Bus Topic Message

To process the message, we will use an Azure Function App. To create one go to the Azure Portal and click the + to add a resource. Type in Function App and you should see;

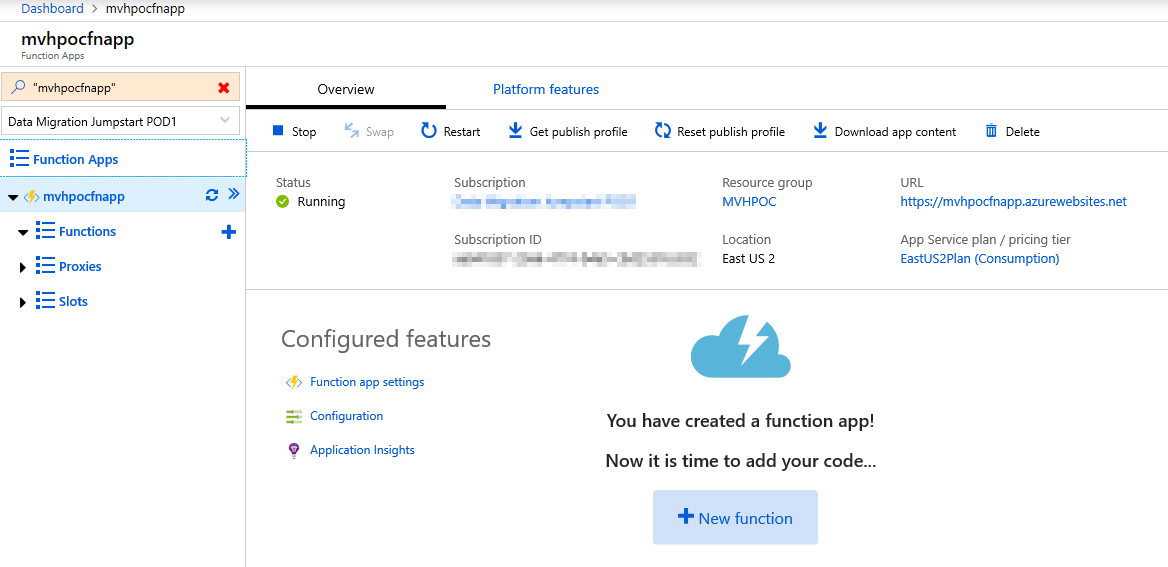


Click Create – fill in the app name and the rest of the fields, in this case we are putting the function app in the same resource group as the Service Bus.

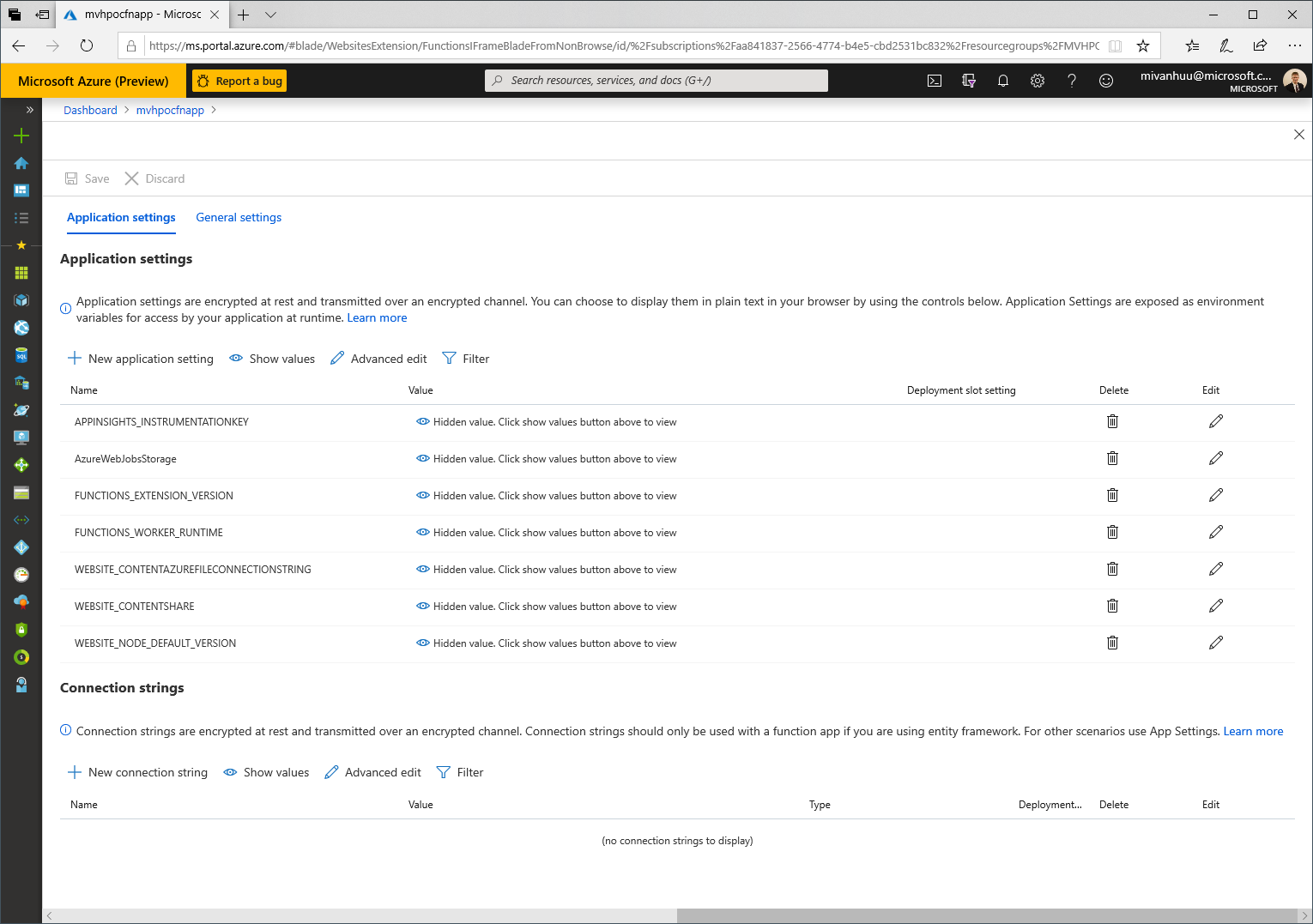


Click Create and wait for it to be provisioned.

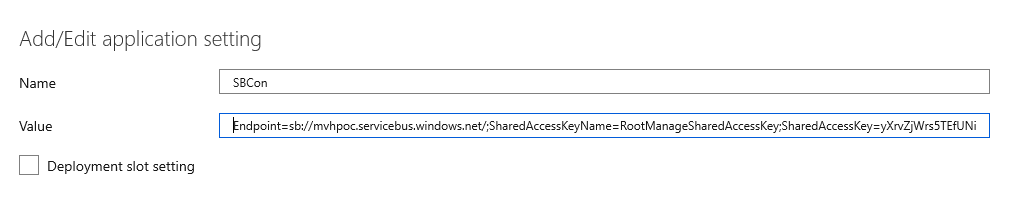
Once provisioned, open the resource and it should look like;



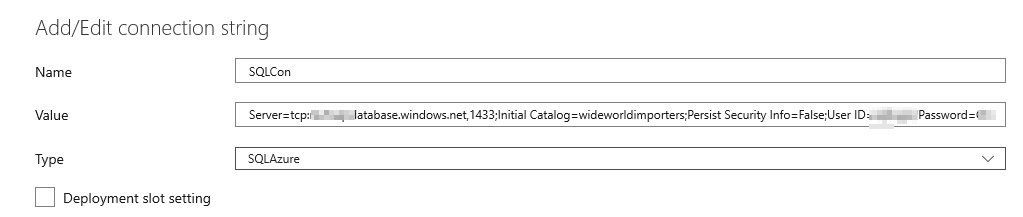
Click on Configuration to add a connection string for the Service Bus;



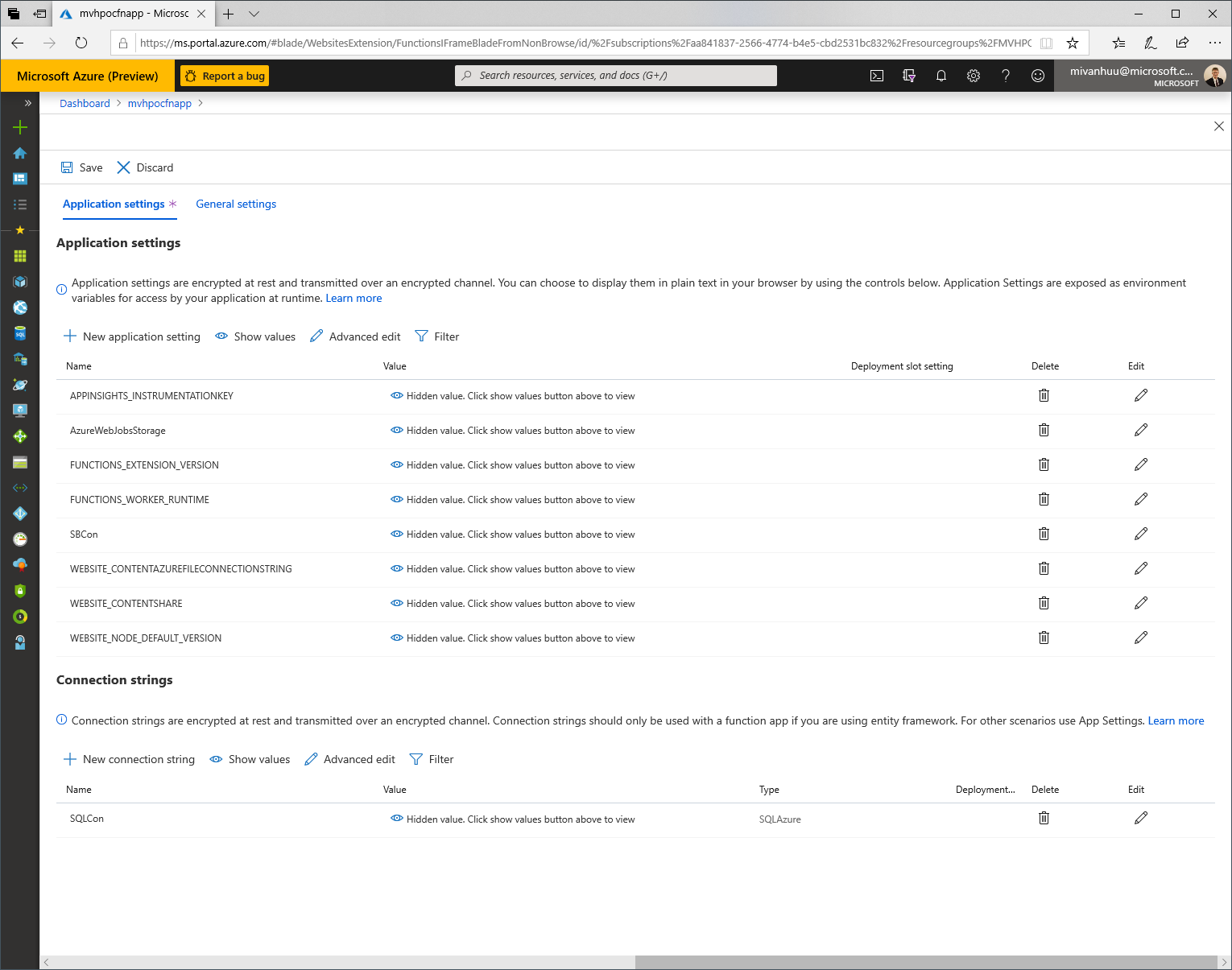
Click on + New application setting



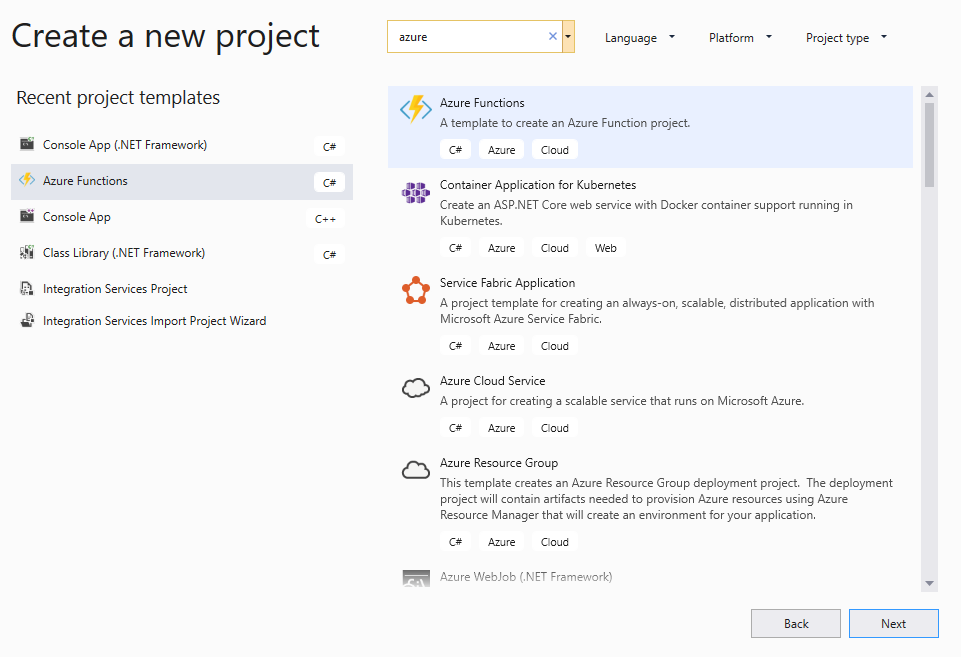
You will also need a Connection String for the SQL Server Connection you will be using. Add one by using + New connection string;



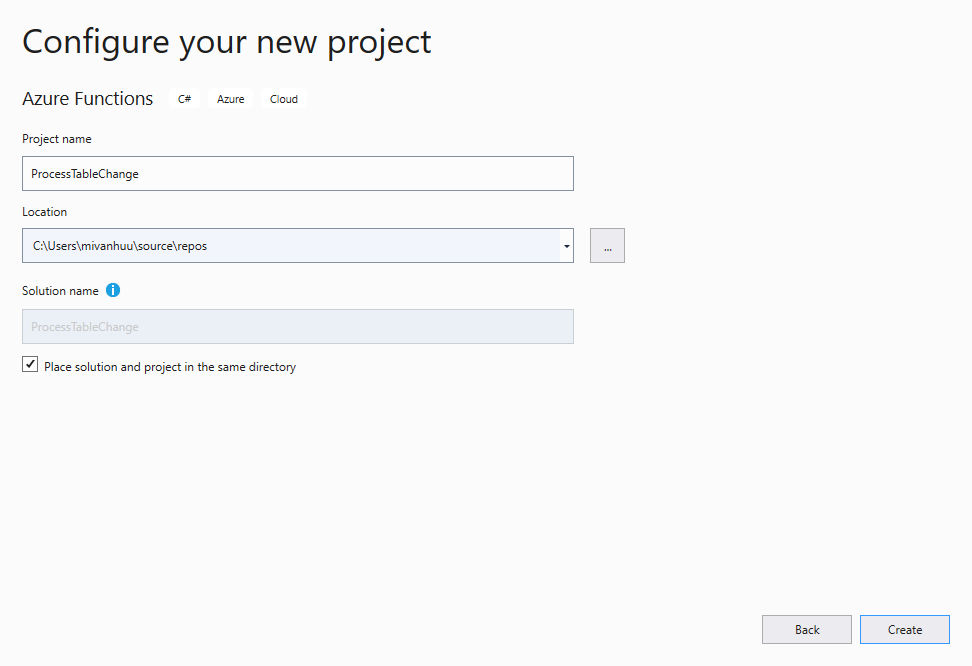
You should have something that looks like the following – ensure you hit Save;



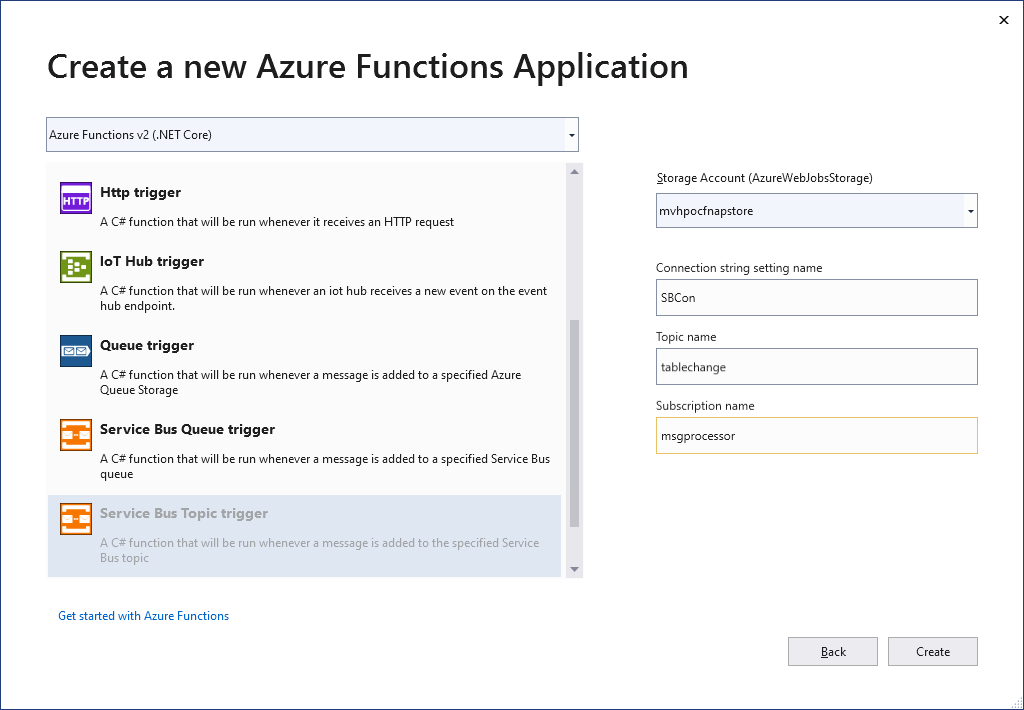
Assuming you have Visual Studio installed, ensure that you have the [latest Azure Functions tools](https://go.microsoft.com/fwlink/?linkid=2016394) installed, which will depend on your version of VS. In Visual Studio (2019 below) create a new project and type Azure into the search bar – if you don’t see Azure Functions you are missing the pre-requisites.



Walk through the project creation;



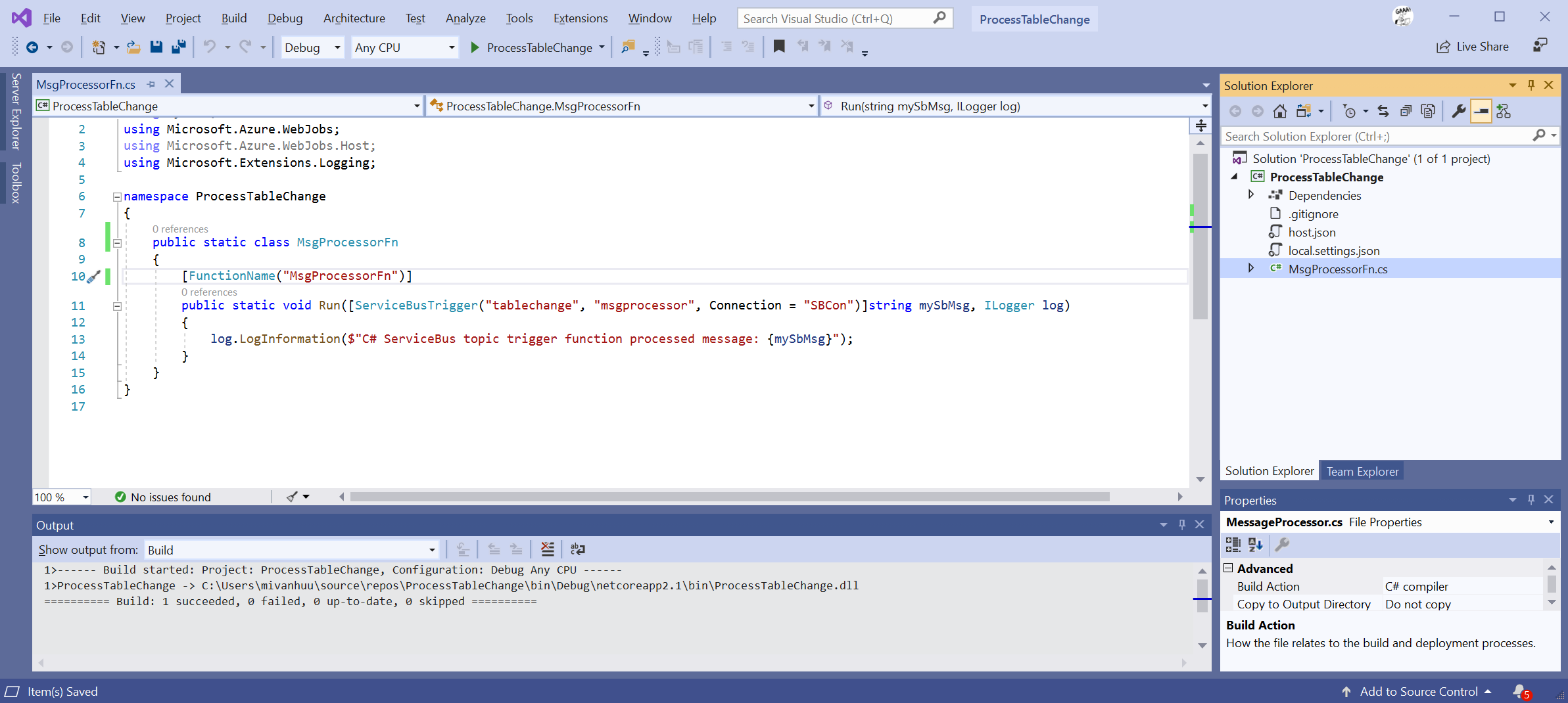
Set the Service Bus properties based on above;



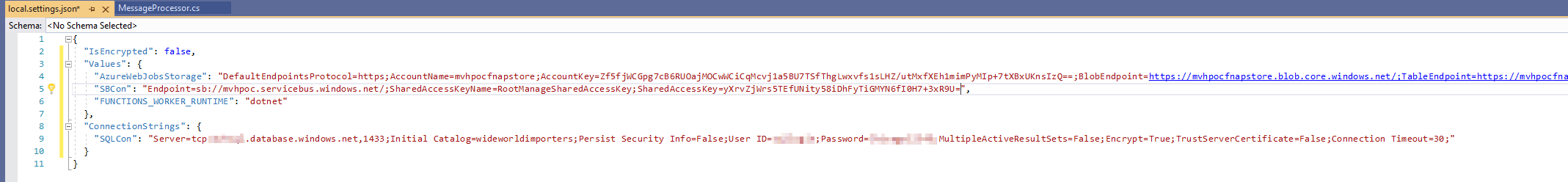
After the project is created you should have a function that looks like;



Change the name of the file and Function1 to something more meaningful.

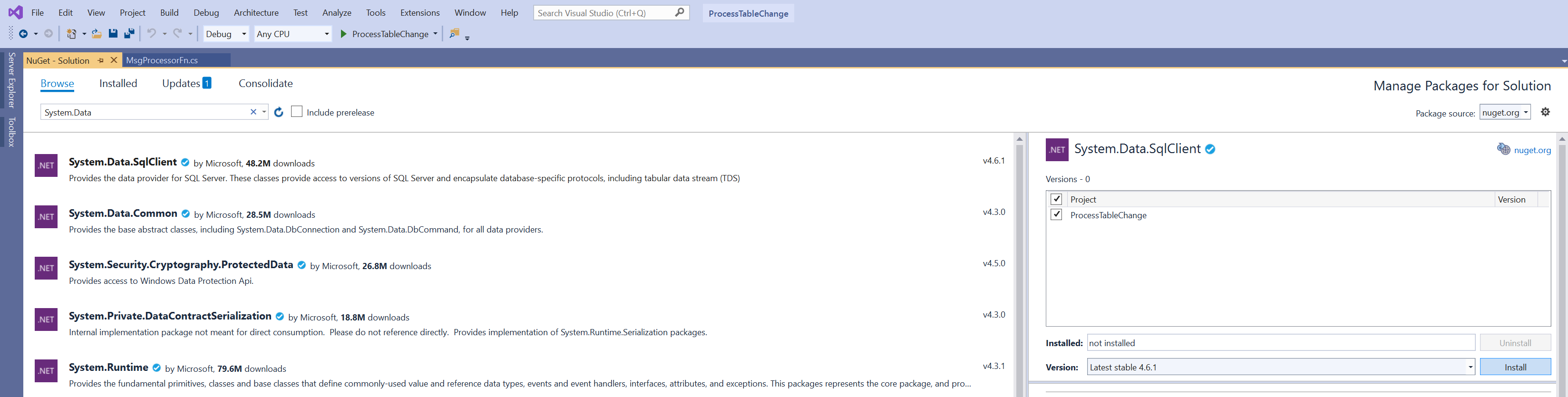


Open the local.settings.json file and add the Service Bus and SQL Connection Strings;



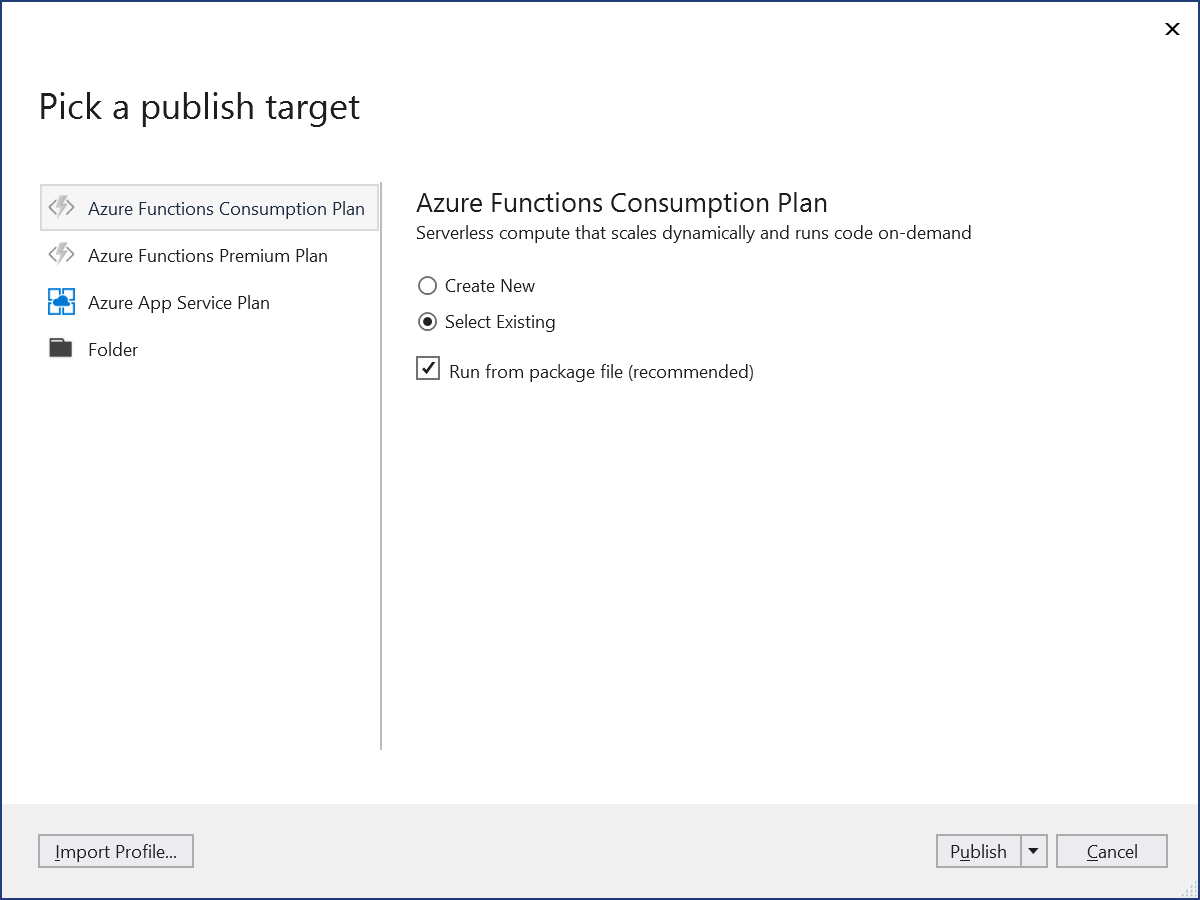
The function should build successfully.

You can then paste in the rest of the code from the sample. The sample requires a NuGet reference to System.Data.SqlClient. Open the NuGet Package Manager and add the reference to the solution;

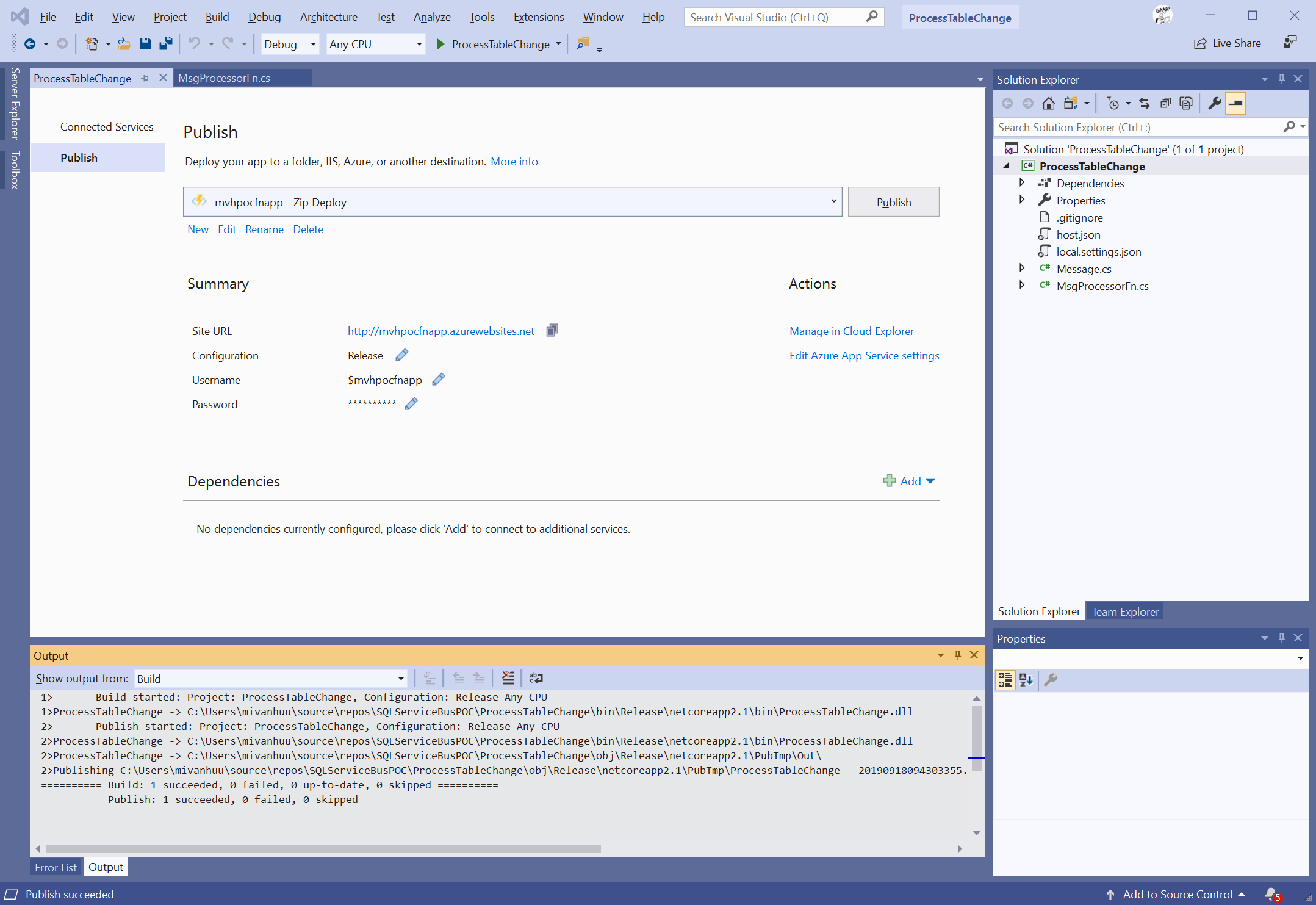


While you have the NuGet Package Manager open, update anything that has an update pending.

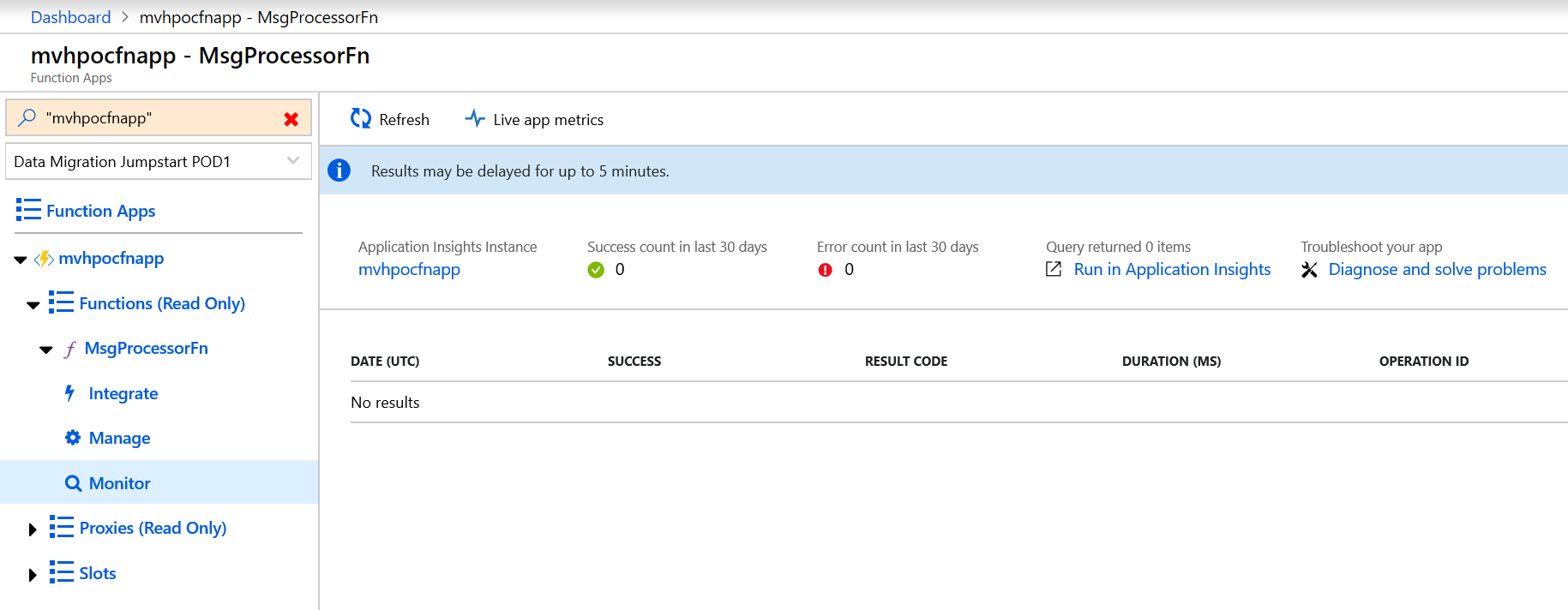
You can now build and publish the solution to an existing target Function App in Azure;



Select the existing Azure Function and publish the function;



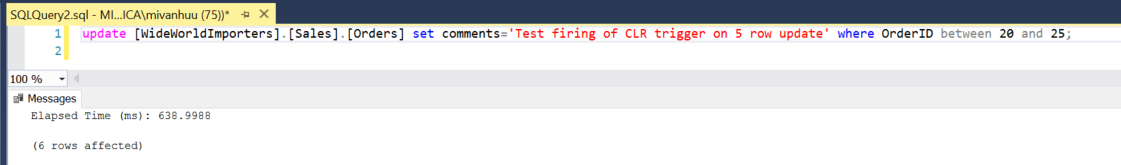
You can now see the new function in the Azure Portal;



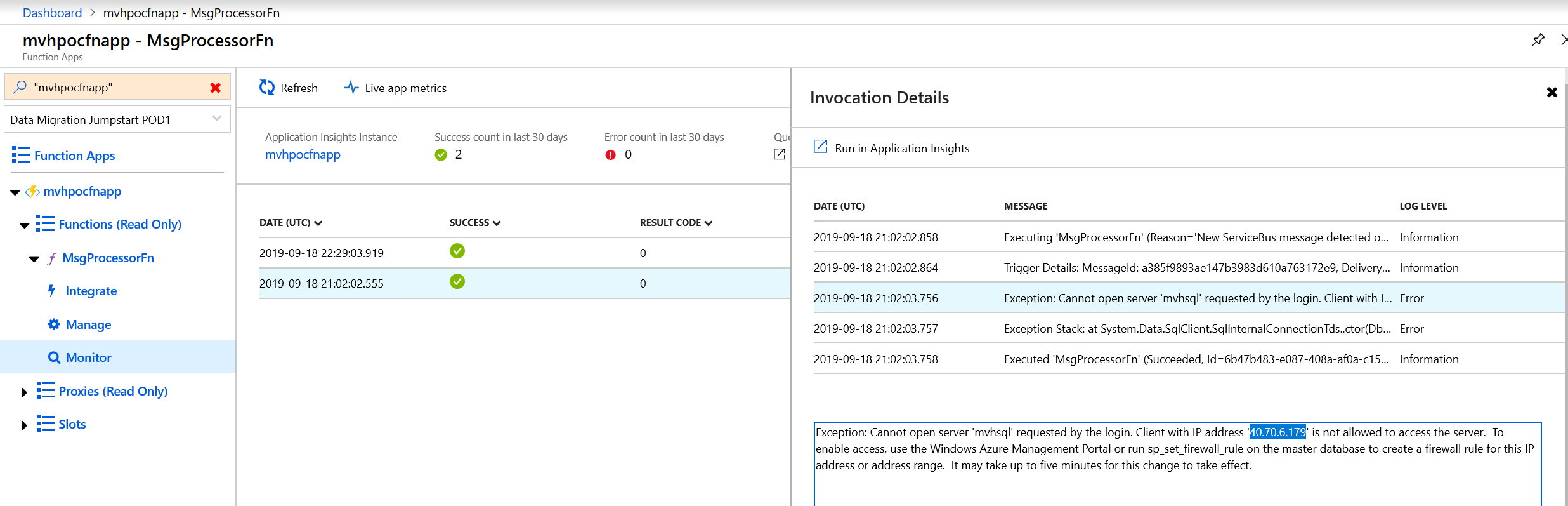
# Testing Message Processing End-to-End

To test the message generation and passing end-to-end, we need a source and target database (in the sample we use the WideWorldImports sample database) with one of the types of triggers created on the source table(s) and then we make can make a change in the table and watch the messages flow.

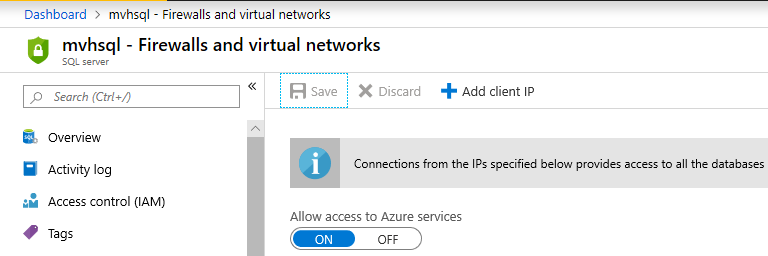
We added the CLR trigger to the Sales.Orders table and updated a group of rows (6 not 5 😊)



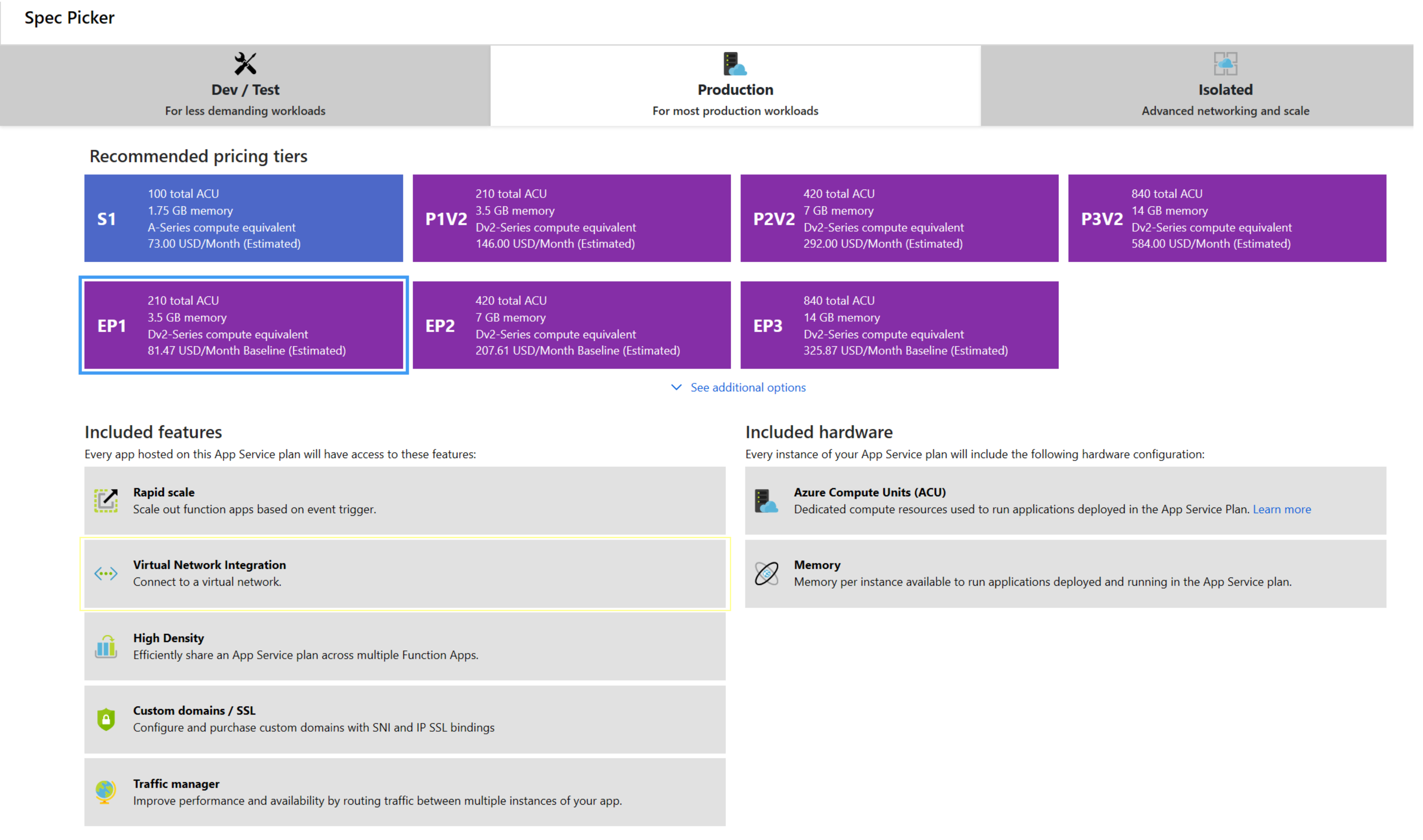
The Azure Function Fires immediately, but the logging can lag up to 5 minutes. On checking the target database, nothing changed. Once the log entries showed up it became clear why not. By default Azure SQL DB has a firewall that blocks all traffic (secure by default). The Azure Function App connection was therefore blocked;



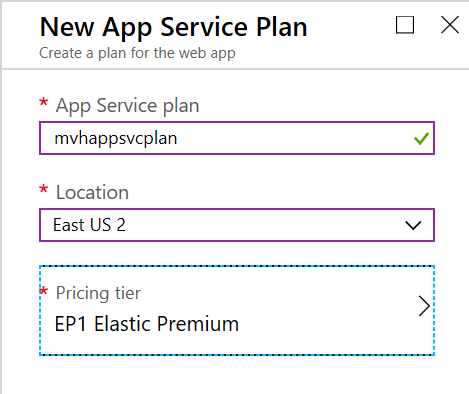
Adding the IP for the Azure Function App is a bit tricky since it can change. In theory it could come from any IP within the documented range of IPs at the Azure Data Center and indeed in every invocation the connection came from a different IP. In this case, for POC purposes, I am just going to give all Azure Services access (which is not a good practice);



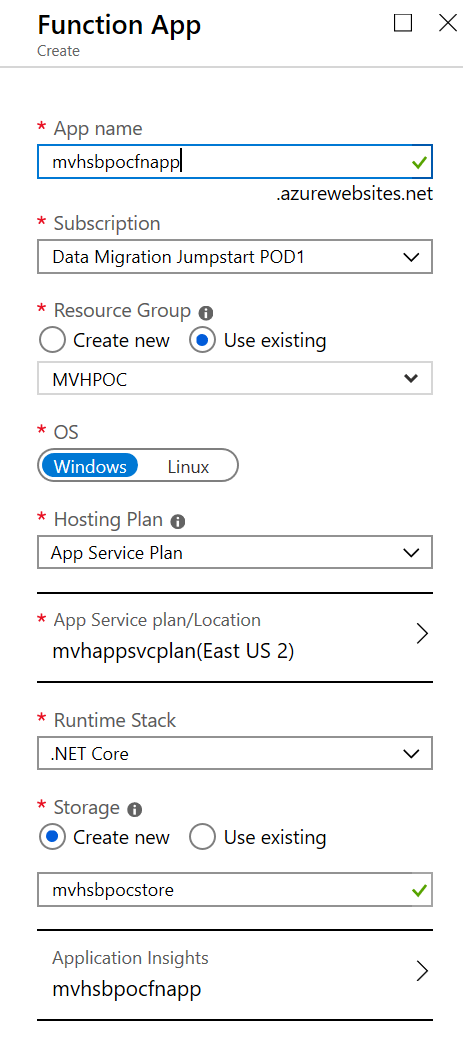
The EP1 tier of an App Service Plan allows integration with VNETs (this seems to be changing dynamically, since subsequently, I have done a test with S1 and it now has VNET integration);



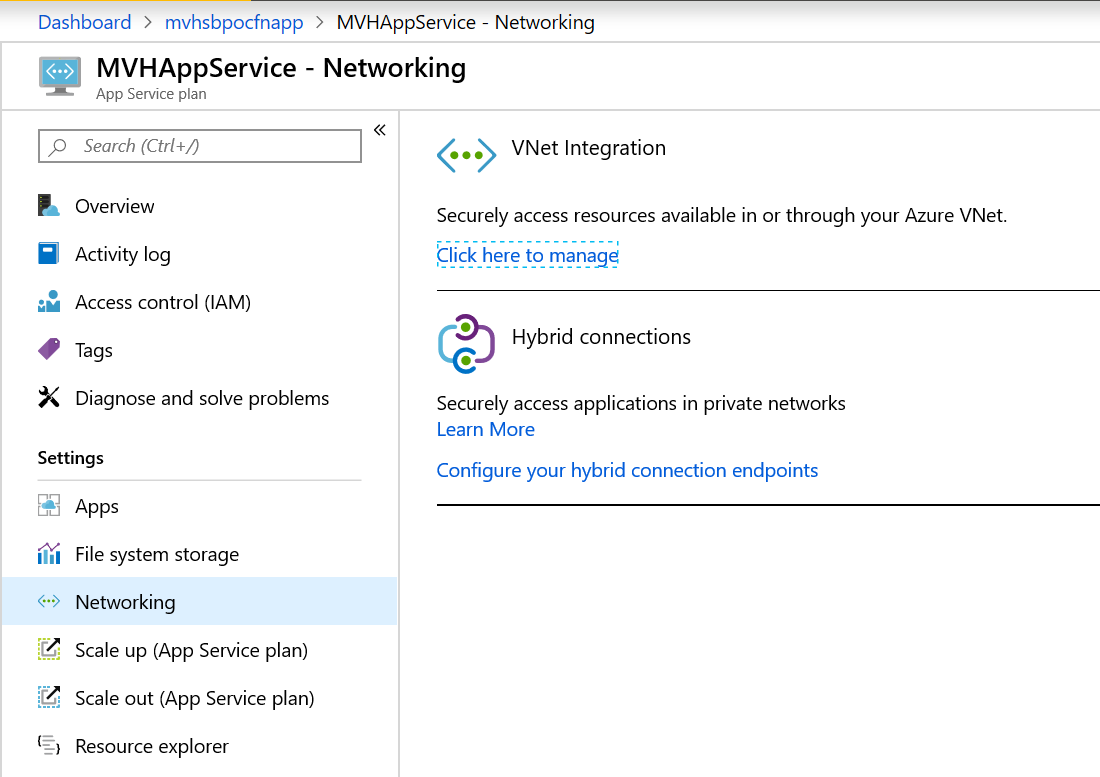
Select an EP1 plan in the same region as your MI instance;



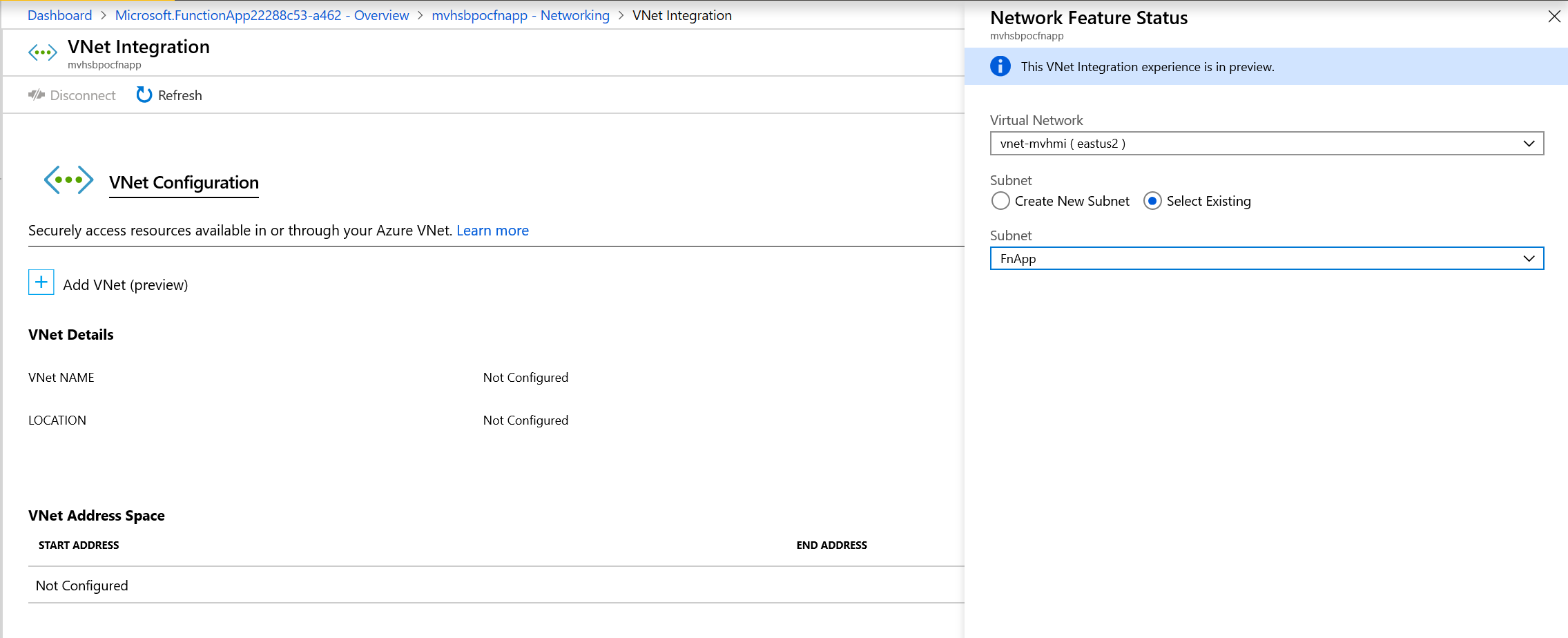
Which should give you an Function App configuration that looks like;



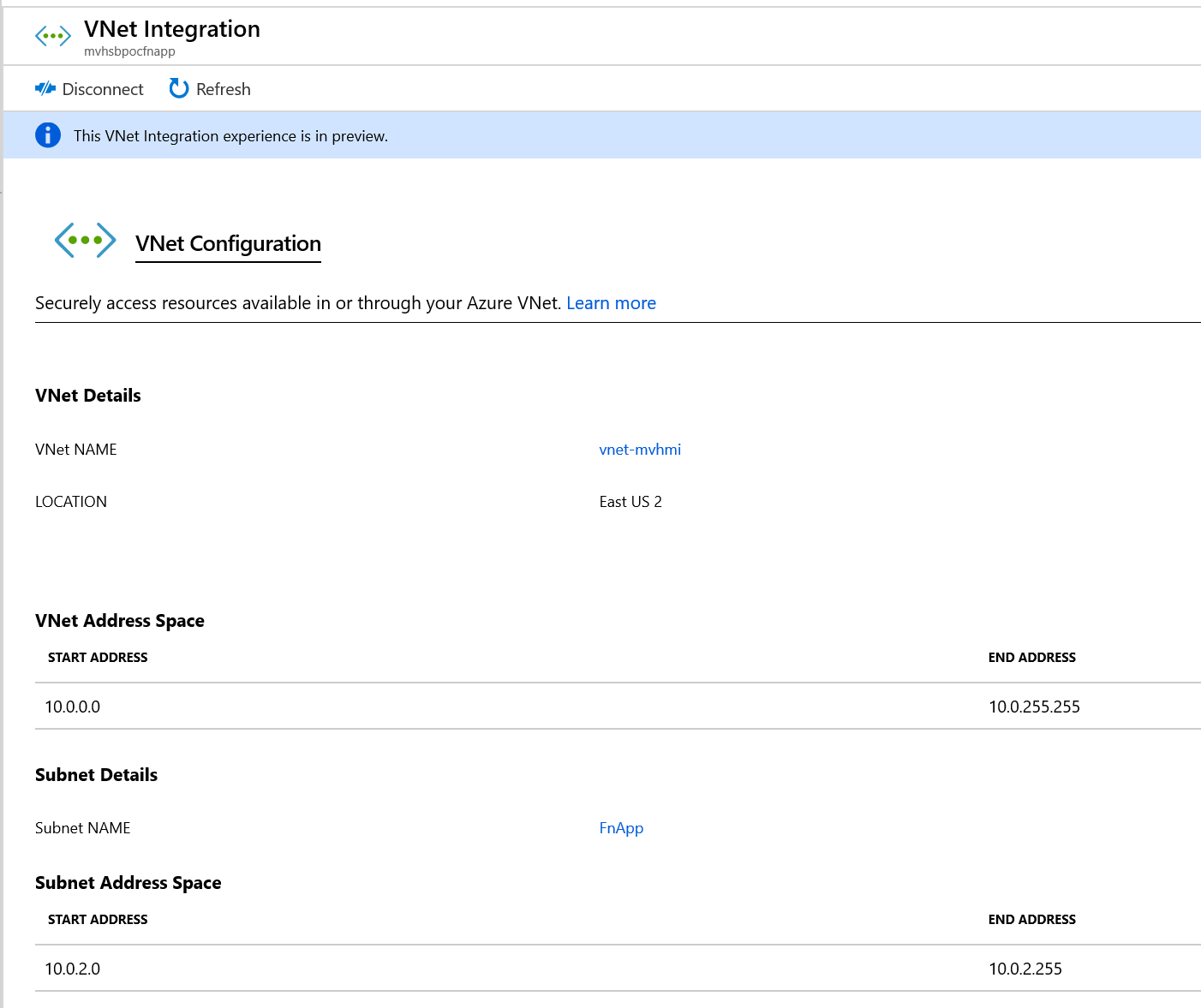
Once provisioned you will see “VNet Integration” under Networking, that is not available under consumption plans;



You can then connect your Function App to a new or existing subnet on your MI VNET.

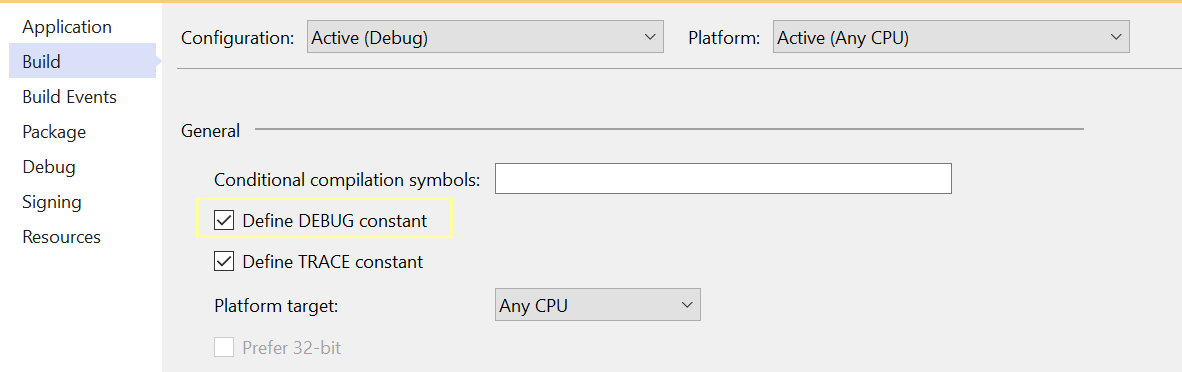


Once the change is complete, the network configuration should look something like;

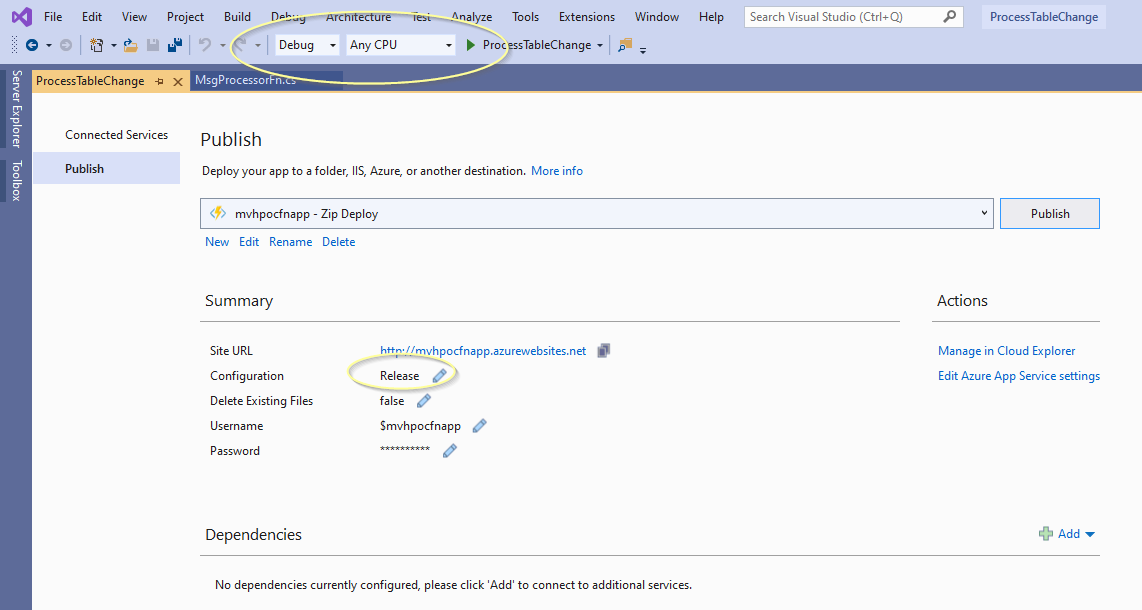


Taking this one step further, the Isolated tier of an Application Service Plan there is an advanced networking feature that allows your function app to be directly configured on a subnet of one of your VNETs. The purpose of this integration is to allow your function app access to VNET resources without having to configure complex firewall rules (for dynamic IP addresses).

Even after turning this on, there was an issue with the update statement, but the logging (surrounded by #if DEBUG / #endif statements) was not showing up in the log. The Debug constant definition needs to be turned on in the project Build settings.



By default, although VS is set to debug, the Publish process is set to Release – so the logging in the app is not enabled;



# Performance Testing

Some basic latency testing was performed to see what the end-to-end performance looked like for each option. The testing was done on a 4 core General Purpose MI instance. SQL updates were made in one database table and the change was made by the Azure Function App to a table in another database on the same MI instance – that way the system times are comparable. The application service tier was Isolated with the Function App sitting on a VNET of the MI instance. The Service Bus was in premium tier.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **CLR Trigger** | **ms** |  |  |  |
| Min | 187 |  |  |  |
| Max | 516 |  |  |  |
| Avg | 319.2 |  |  |  |
|  |  |  |  |  |
| **TSQL Trigger/CLR UDF** | **ms** |  |  |  |
| Min | 140 |  |  |  |
| Max | 1188 |  | 984.3726 | ms in trigger (otherwise max is 206 ms) |
| Avg | 384.8 |  |  |  |
|  |  |  |  |  |
| **Service Broker/CLR UDF** | **ms** |  |  |  |
| Min | 174 |  |  |  |
| Max | 421 |  |  |  |
| Avg | 278.4 |  |  |  |

All three options yielded good performance at 300-400 ms end-to-end.

Since each customer latency will depend on many factors including number and size of columns, number of rows changed per insert/update/delete operation, service tiers of each component etc. - not a lot of time was spent on this activity.

# Next Steps – Additional Design Considerations

* Scalable design – there are scalability limits in this solution. Depending on your scaling requirements, using a batching mechanism may provide better throughput at the cost of individual message latency.
* App Function Tuning - Use of partitioning and slots
* Multiple targets – use the function app as the distributor to use metadata to push the changes to one or more targets (and not back to itself in multi-master scenarios).
* Error Handling, logging, operations etc.
* Target Server Offline Handling
  + Initial thoughts here are to use the Function App as a distributor to push messages to queues for each target machine.
  + Question – does another function app dequeue and apply or do we pull/poll somehow from SQL Agent or Service Broker or…?
  + If a machine is down – how long can an Azure Function hang waiting until it is killed?