GLOBAL INTEGRATION BOOTCAMP

Labs

TOMASSO GROENENDIJK

ELDERT GROOTENBOER

STEEF-JAN WIGGERS

ROB FOX

Contents

Naming Conventions

As each resource in Azure should have an unique name, we advice to use the following naming convention.

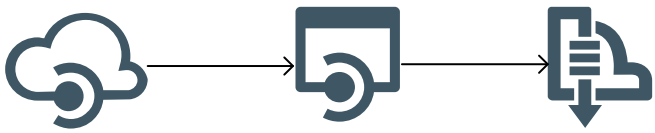
Resource	Suggested Name*	Sample
Resource Group	gib<loc>17-rgrp-<ini><##>	gibmel17-rgrp-pc01
Service Bus	gib<loc>17-sbus-<ini><##>	gibmel17-sbus-pc01
Storage Account	gib<loc>17st<ini><##>	gibmel17stpc01
API Management	gib<loc>17<ini><##>	gibmel17pc01
API App	gib<loc>17<ini><##>ordersapi	gibmel17pc01ordersapi
Function App	gib<loc>17-func-<ini><##>	gibmel17-func-pc01
Logic Apps Lab 1	gib<loc>17-logic-<ini><##>-validatemaporder	gibmel17-logic-pc01-validatemaporder
Logic Apps Lab 2	gib<loc>17-logic-<ini><##>- storeorderonprem	gibmel17-logic-pc01-storeorderonprem
Logic Apps Lab 3	gib<loc>17-logic-<ini><##>-procbusinesscustorder	gibmel17-logic-pc01-procbusinesscustorder
IoT Hub	gib<loc>17-ioth-<ini><##>	gibmel17-ioth-pc01
Document DB	gib<loc>17-docdb-<ini><##>	gibmel17-docdb-pc01

* Replace <loc> for a 3 char acronym of your location
* Replace <ini> for your 2 initials
* Replace <##> for 2 random numbers for uniqueness

Lab 1 - API Management + API Apps

Objective

In this first lab, we will be creating an API app which is protected using API Management. The API will be used by vendors to place orders, and allows for fast innovations to be rolled out. By using API Management SETR can easily and flexibly give new vendors access to their ordering system. Once the order has been received, it will be placed in a Service Bus queue, where the next system will pick it up, giving us a loosely coupled ordering service which can keep running even if any of our downstream services are down due to maintenance or upgrades.



Prerequisites

- Azure account - You can Open an Azure account for free or Activate Visual Studio subscriber benefits.
- Service Bus Explorer
- Visual Studio 2015 with the Azure SDK for .NET - The SDK installs Visual Studio 2015 automatically if you don't already have it.

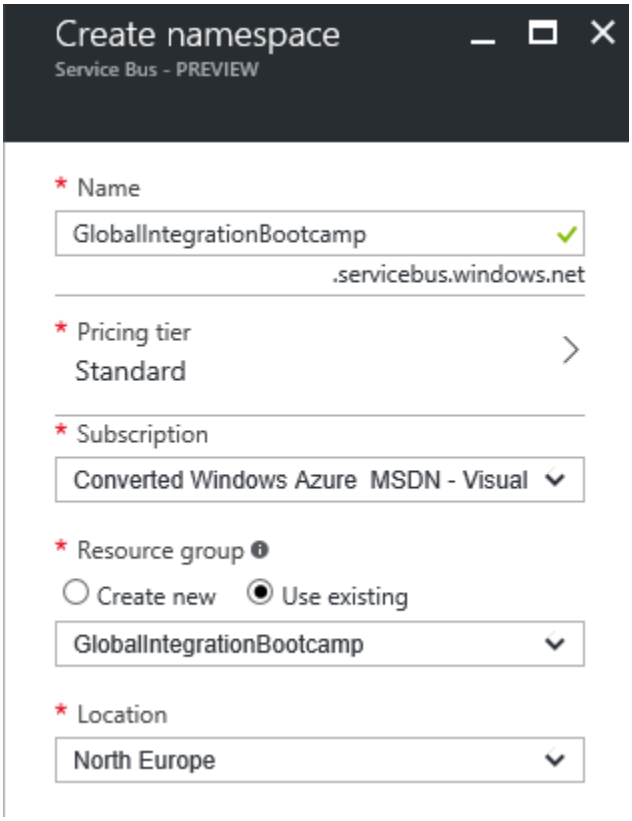
Steps

To build the solution in this lab you have to follow the steps described in this section. From a high level view the steps are:

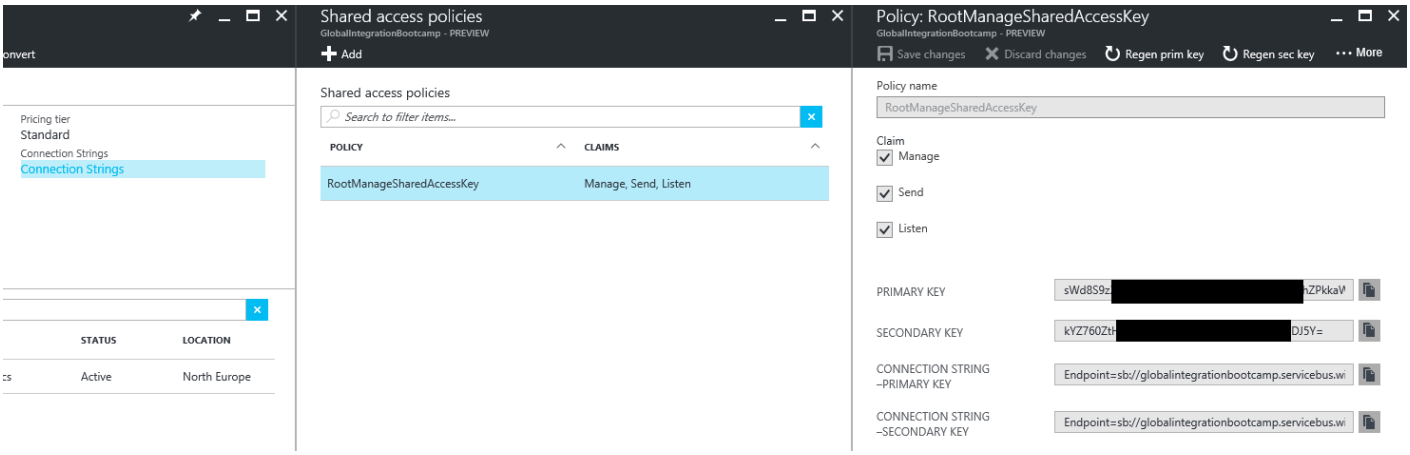
1. Create Service Bus namespace
2. Create an API Management instance
3. Create an Azure API App
4. Test your API App locally
5. Publish API App to Azure
6. Import the Order API in API Management
7. Add the Order API to the Starter product
8. Call the Order API from the Developer Portal

Create Service Bus Namespace

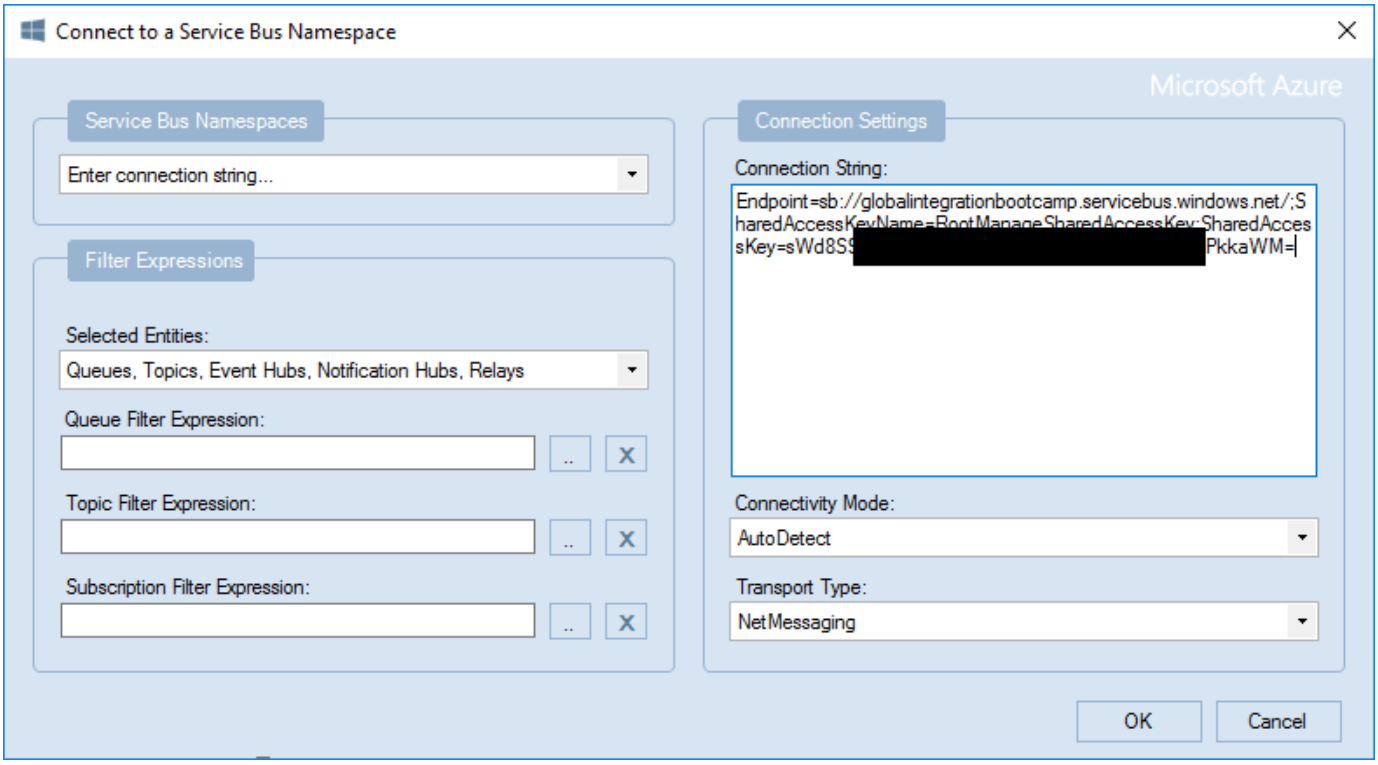
As we will be sending our outgoing messages to a Service Bus queue, we will also need to create this in Azure. Go to the Service Bus blade in the portal, and add a new namespace (this should be a unique namespace). This namespace should be created on the **Standard** tier, as this is the tier from where topics are included as well. (Topics are needed in one of the other labs)



Once the namespace has been created, we will need to add the queue. For this we will use Service Bus Explorer, a great tool when working with Azure Service Bus. Start by retrieving the connection string for the **RootManageSharedAccessKey**, which we will use to manage our namespace from Service Bus Explorer.



Now start Service Bus Explorer, and connect using the connection string we just retrieved.

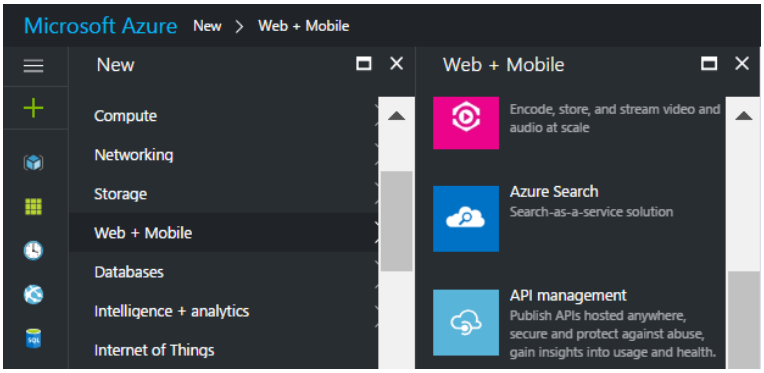


Next add a queue called **neworders**, and keep all the default settings.

Create an API Management instance

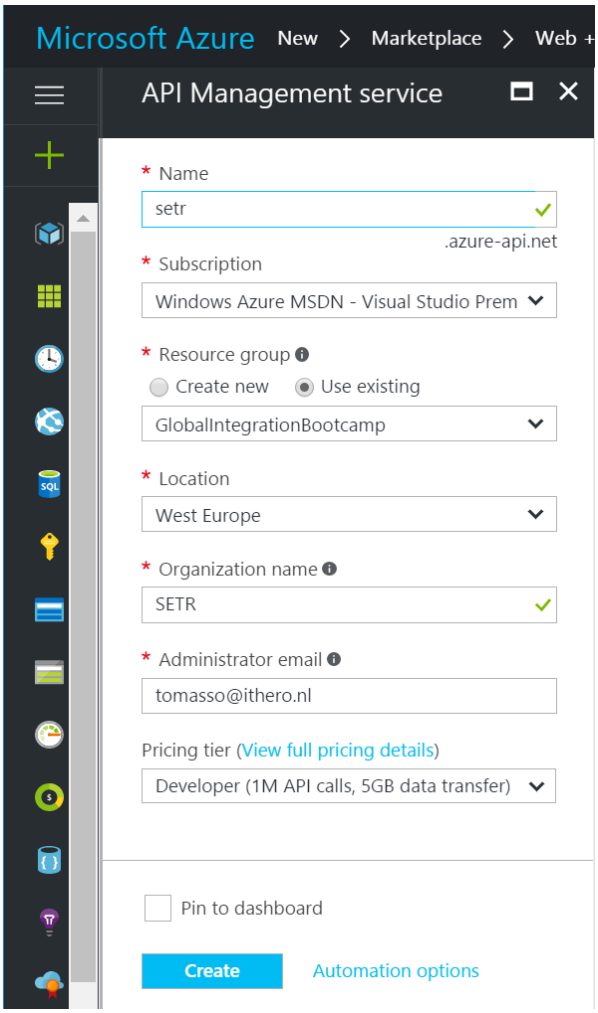
The first step in working with API Management is to create a service instance.

1. Sign in to the Azure Portal and click **New**, **Web + Mobile**, **API Management**.



2. For Name, specify a unique sub-domain name to use for the service URL.
   Choose the desired Subscription, Resource group and Location for your service instance.
   Enter SETR for the Organization Name, and enter your email address in the Administrator E-Mail field.

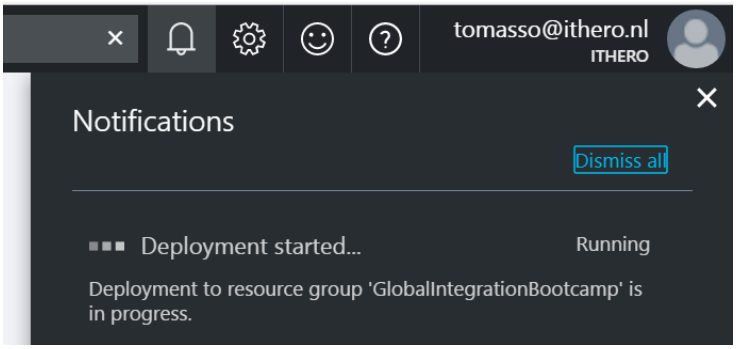
   Note
   The email address is used for notifications from the API Management system.



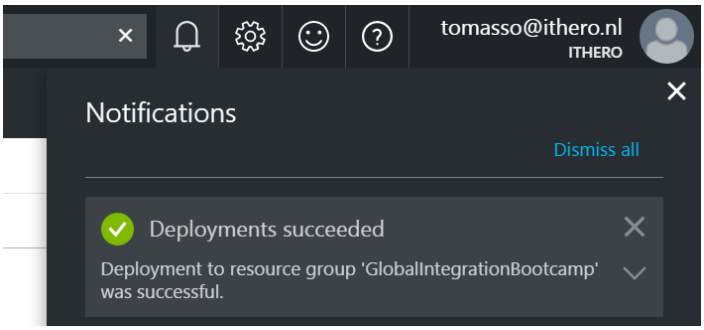
Note

API Management service instances are available in three tiers: Developer, Standard, and Premium. You can complete this lab by using the Developer tier.

3. Click **Create** to start provisioning your service instance.
4. The Deployment takes several minutes. Click in the Header toolbar on Notifications to follow the Deployment



5. Once the service instance is created, click on the notification to open API Management.

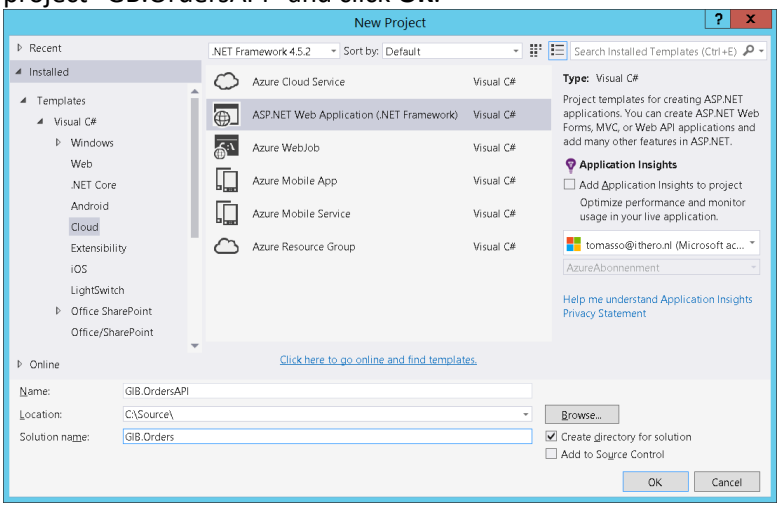


The next step is to import the Order API.

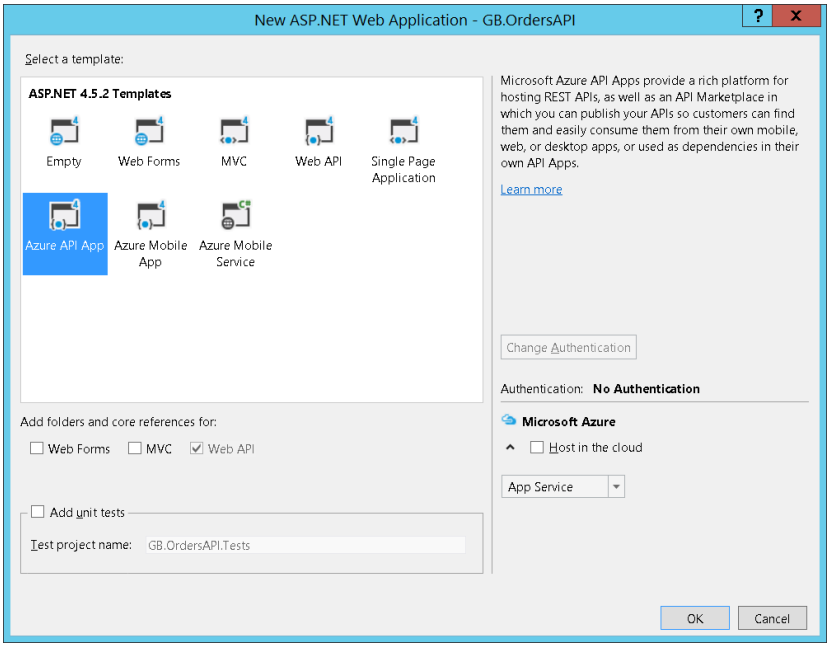
Create an Azure API App

The second step in this lab is to create an API App in Visual Studio and test it locally.

1. Start Visual Studio and select **New Project** from the **Start** page. Or, from the **File** menu, select **New** and then **Project**.
2. In the **Templates** pane, select **Installed Templates** and expand the **Visual C#** node. Under **Visual C#**, select **Cloud**. In the list of project templates, select **ASP.NET Web Application**. Name the project "GB.OrdersAPI" and click **OK**.



3. In the **New ASP.NET Project** dialog, select the **Azure API App** template. and click **OK**.



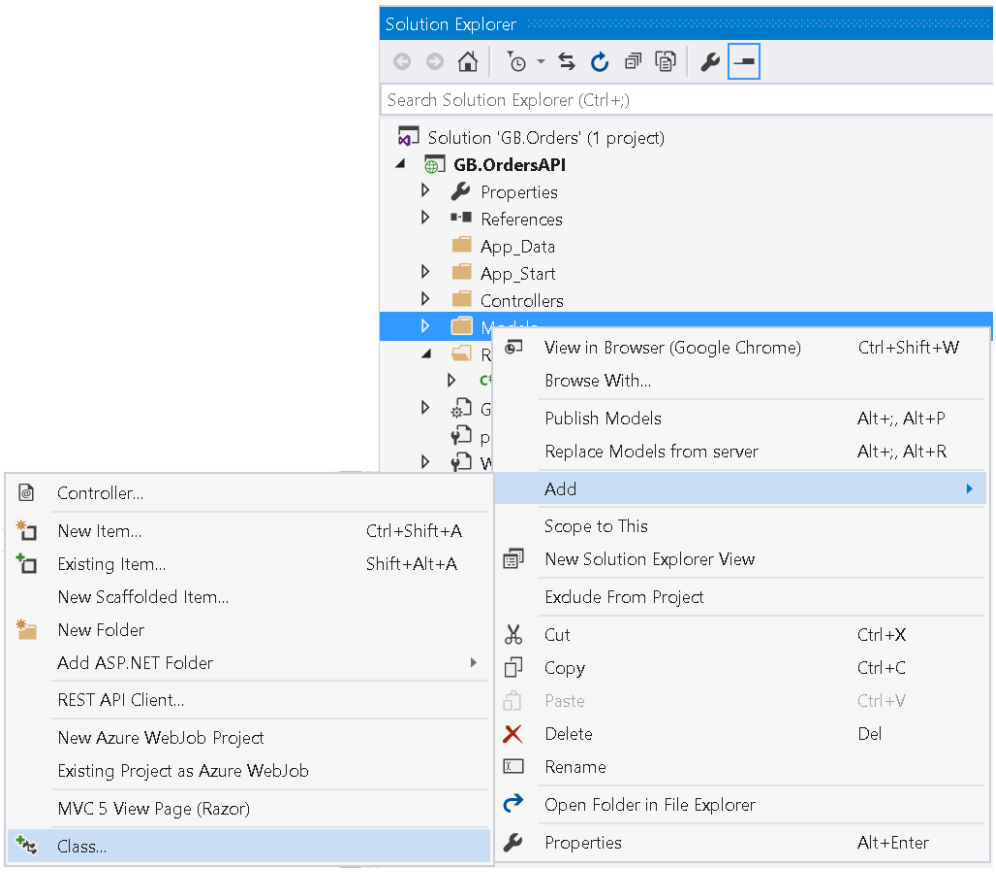
Note

Make sure that the checkbox “Host in the cloud” is unchecked. Hosting in Azure will be described in the step: Publish API App to Azure.

4. Open the web.config file and add the ServiceBus connection string to the appSettings.

```
<configuration>
  <appSettings>
    <add key="Microsoft.ServiceBus.ConnectionString"
value="Endpoint=sb://[your
namespace].servicebus.windows.net;SharedAccessKeyName=RootManageSharedAccessKey;SharedAccessKey=[your secret]" />
  </appSettings>
…
```

5. Add models for the order to the API App project. A model is an object that represents the data in your application. If Solution Explorer is not already visible, click the View menu and select Solution Explorer. In Solution Explorer, right-click the Models folder. From the context menu, select Add then select Class.



6. The following models need to be created:

```
public class Customer
{
```

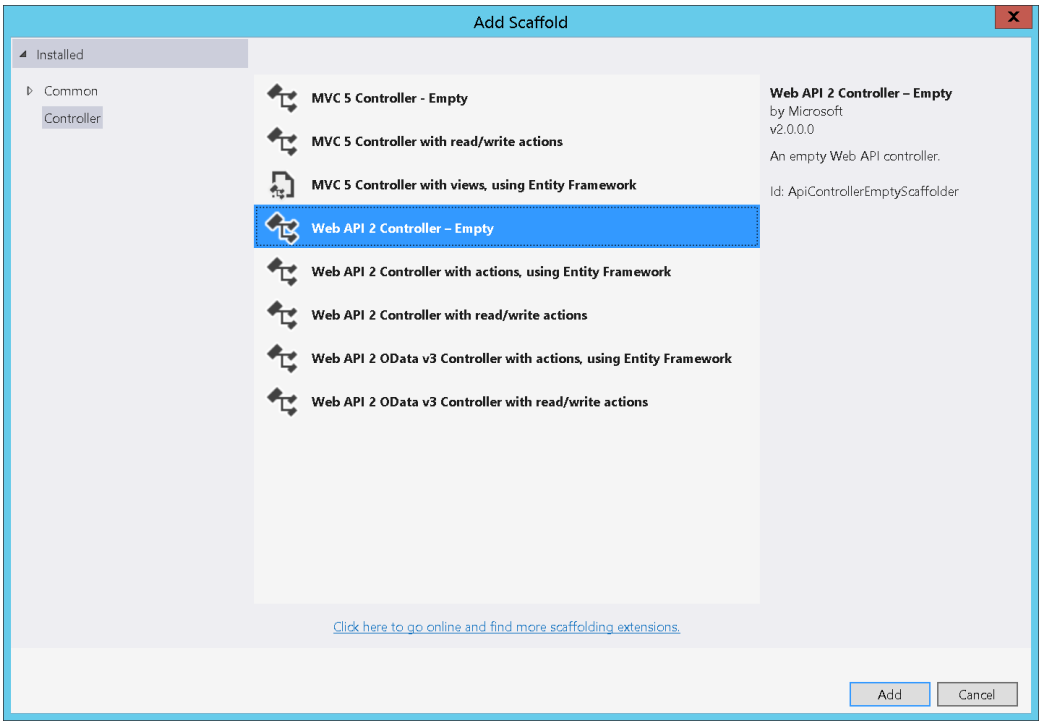
```
        public string CustomerNumber { get; set; }
}

public class Product
{
        public int ProductNumber { get; set; }
        public int Amount { get; set; }
}

public class Products
{
        public List<Product> Product { get; set; }
}

public class Order
{
        public Customer Customer { get; set; }
        public List<Product> Products { get; set; }
        public DateTime OrderedDateTime { get; set; }
}

public class CreateOrder
{
        public Order Order { get; set; }
}
```

7. In **Solution Explorer**, right-click the GB.OrdersAPI project. Select **Add** and then select **New folder**.
8. Name the folder: Repositories and press Enter.
9. Add a repository class to the project. A repository is an object that encapsulates the ServiceBus layer. In Solution Explorer, right-click the Repositories folder. From the context menu, select Add then select Class.
   Name the class ServiceBusRepository and tap Add.
10. Replace the code in this file with the following:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using Microsoft.ServiceBus.Messaging;
using System.Configuration;
using GB.OrdersAPI.Models;
using System.IO;
using System.Xml;
using System.Xml.Serialization;

namespace GB.OrdersAPI.Repositories
{
    public class ServiceBusRepository
    {

        public void Send(CreateOrder order)
        {
```

```
            string xmlObject = GetXMLFromObject(order);

            string connectionString =
ConfigurationManager.AppSettings["Microsoft.ServiceBus.ConnectionString"];
            string queueName = "NewOrders";

            var client =
QueueClient.CreateFromConnectionString(connectionString, queueName);
            var message = new BrokeredMessage(xmlObject);

            client.Send(message);
        }

        private string GetXMLFromObject(object o)
        {
            StringWriter sw = new StringWriter();
            XmlTextWriter tw = null;
            try
            {
                XmlSerializer serializer = new XmlSerializer(o.GetType());
                tw = new XmlTextWriter(sw);
                serializer.Serialize(tw, o);
            }
            catch (Exception ex)
            {
                //Handle Exception Code
            }
            finally
            {
                sw.Close();
                if (tw != null)
                {
                    tw.Close();
                }
            }
            return sw.ToString();
        }
    }
}
```

11. Add a Controller to the project. A controller is an object that handles HTTP requests. In **Solution Explorer**, right-click the Controllers folder. Select **Add** and then select **Controller**. In the **Add Scaffold** dialog, select **Web API Controller - Empty**. Click **Add**.

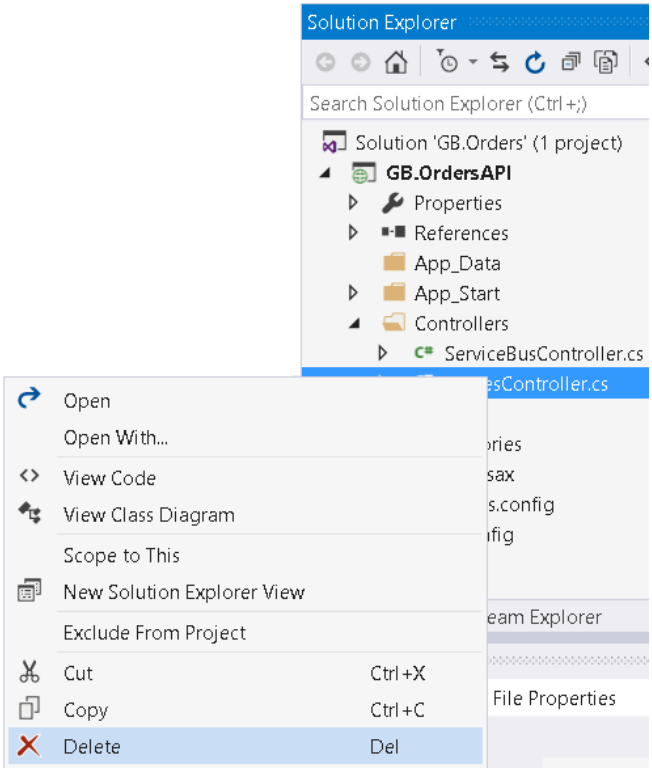


12. In the **Add Controller** dialog, name the controller "OrdersController". Click **Add**. The scaffolding creates a file named OrdersController.cs in the Controllers folder. If this file is not open already, double-click the file to open it. Replace the code in this file with the following:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;
using Swashbuckle.Swagger.Annotations;
using GB.OrdersAPI.Models;
using GB.OrdersAPI.Repositories;

namespace GB.OrdersAPI.Controllers
{
    public class OrdersController : ApiController
    {

        // POST api/values
        [SwaggerOperation("Create")]
        [SwaggerResponse(HttpStatusCode.Created)]
        public IHttpActionResult Post([FromBody]CreateOrder order)
        {
            try
            {
                ServiceBusRepository reposistory = new
ServiceBusRepository();
```

```
                repository.Send(order);

                return Ok();
            }
            catch (Exception ex)
            {
                return InternalServerError();
            }
        }
    }
}
```

13. Delete the Values Controller that is added out the box. Right click the ValuesController.cs file and select Delete.



14. In the OrderAPI project in **Solution Explorer**, open the *App_Start\SwaggerConfig.cs* file, then scroll down and uncomment the following code.

```
/*
 })
.EnableSwaggerUi(c =>
 {
*/
```

The *SwaggerConfig.cs* file is created when you install the Swashbuckle package in a project. The file provides a number of ways to configure Swashbuckle.

The code you've uncommented enables the Swagger UI that you use in the following steps. When you create a Web API project by using the API app project template, this code is commented out by default as a security measure.

Test your API App locally

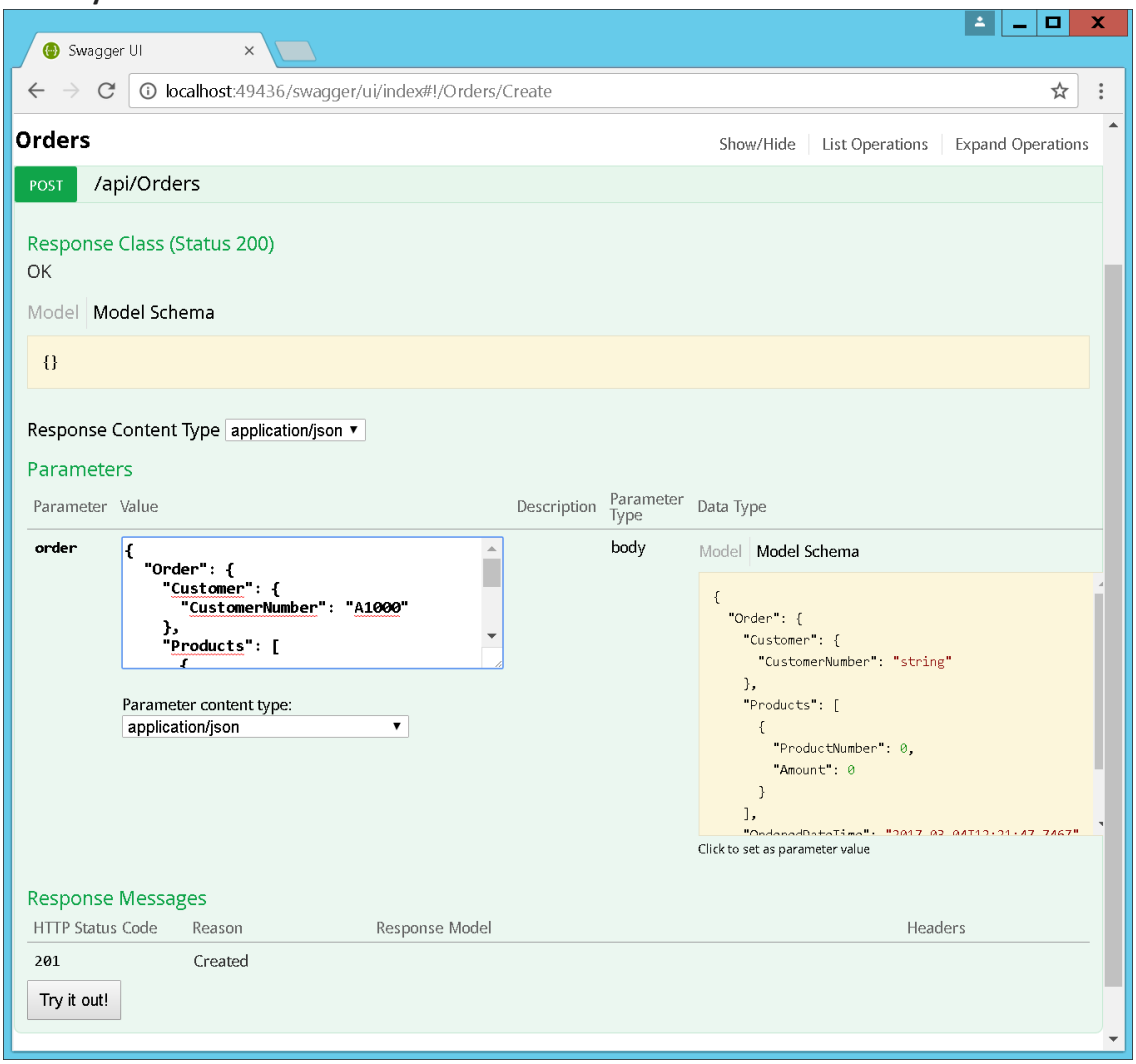
In this section we are going to test the created API locally on your Development machine

1. Press F5 or click **Debug > Start Debugging** to run the project in debug mode.
2. In your browser address bar, add `swagger` to the end of the line, and then press Return. (The URL is `http://localhost:45914/swagger`.)
3. When the Swagger UI page appears, click **Orders** to see the methods available.
4. Click **Post**, and then click the box under **Model Schema**.

   Clicking the model schema prefills the input box where you can specify the parameter value for the Post method. (If this doesn't work in Internet Explorer, use a different browser or enter the parameter value manually in the next step.)
5. Change the JSON in the `order` parameter input box so that it looks like the following example, or substitute your own description text:

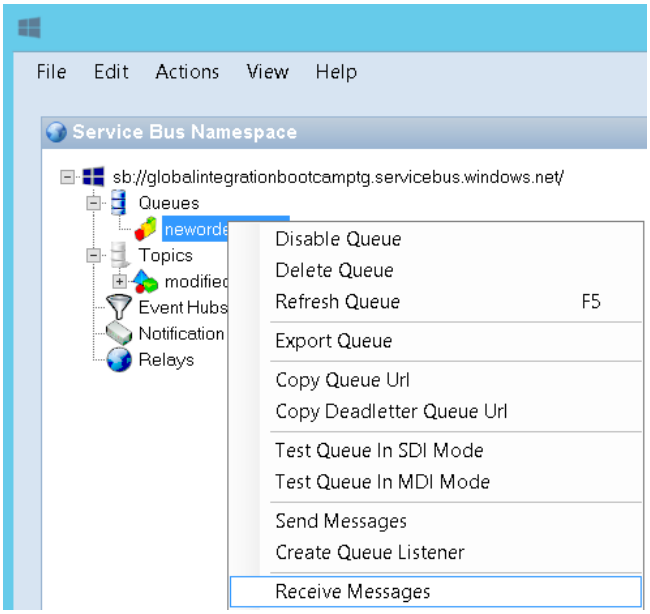
```
{
  "Order": {
    "Customer": {
      "CustomerNumber": "A1000"
    },
    "Products": [
      {
        "ProductNumber": 200,
        "Amount": 1
      }
    ],
    "OrderedDateTime": "2017-03-01T19:10:33.294Z"
  }
}
```

6. Click **Try it out**.

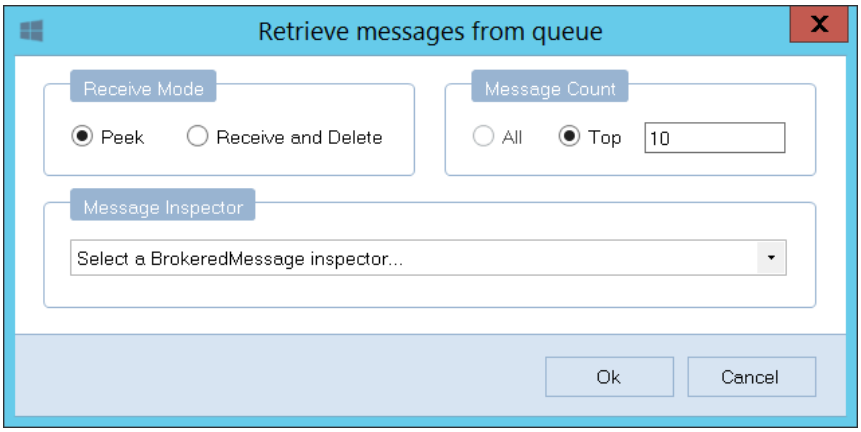


The Order API returns an HTTP 200 response code that indicates success.

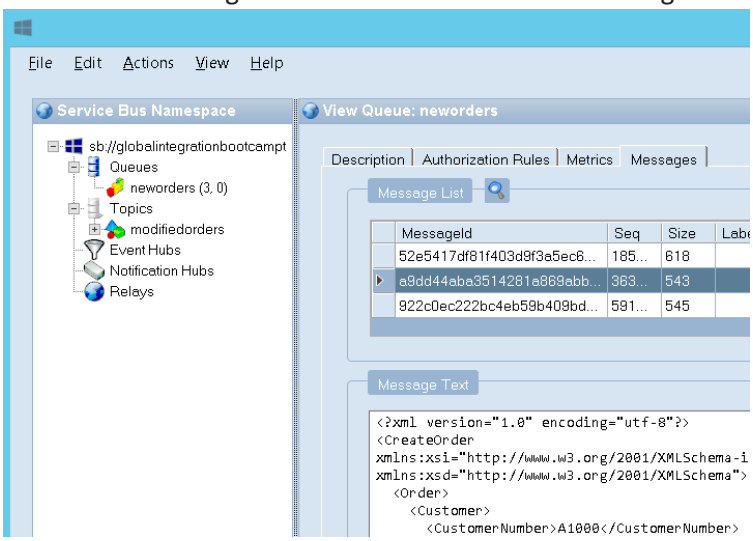
7. Check in the Service Bus Explorer tool if the order message is sent to the NewOrders queue. Open the Service Bus Explorer, right click the neworders queue and select Receive Messages.



8. Select the Peek radio button and click on Ok.



9. Click on the MessageId to see the content of the message.

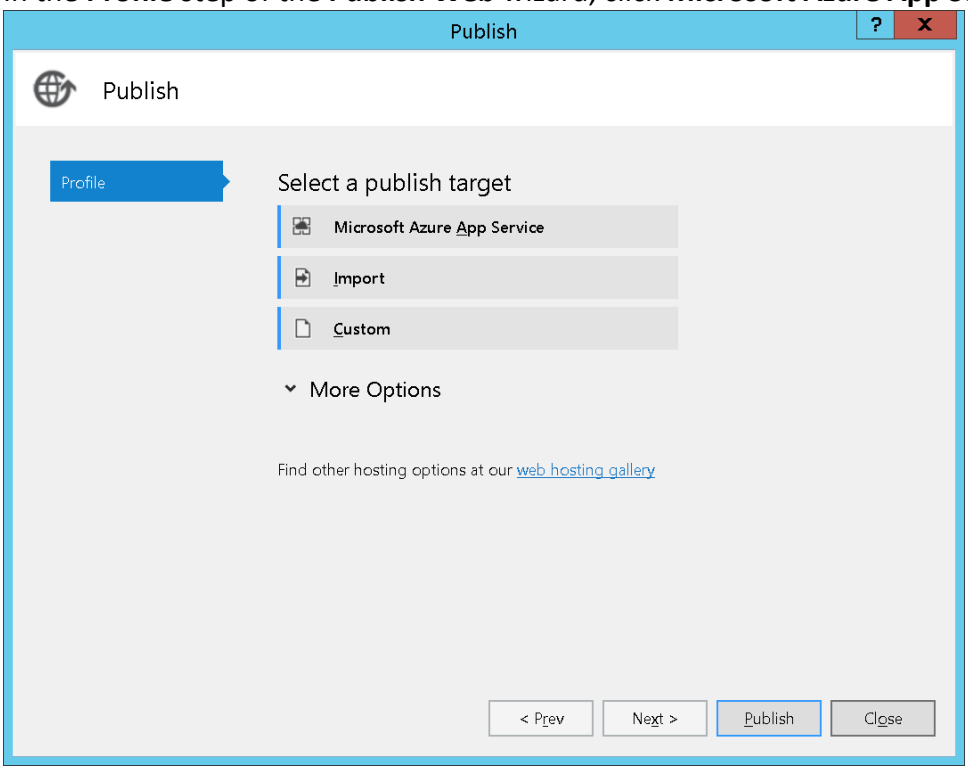


Publish API App to Azure

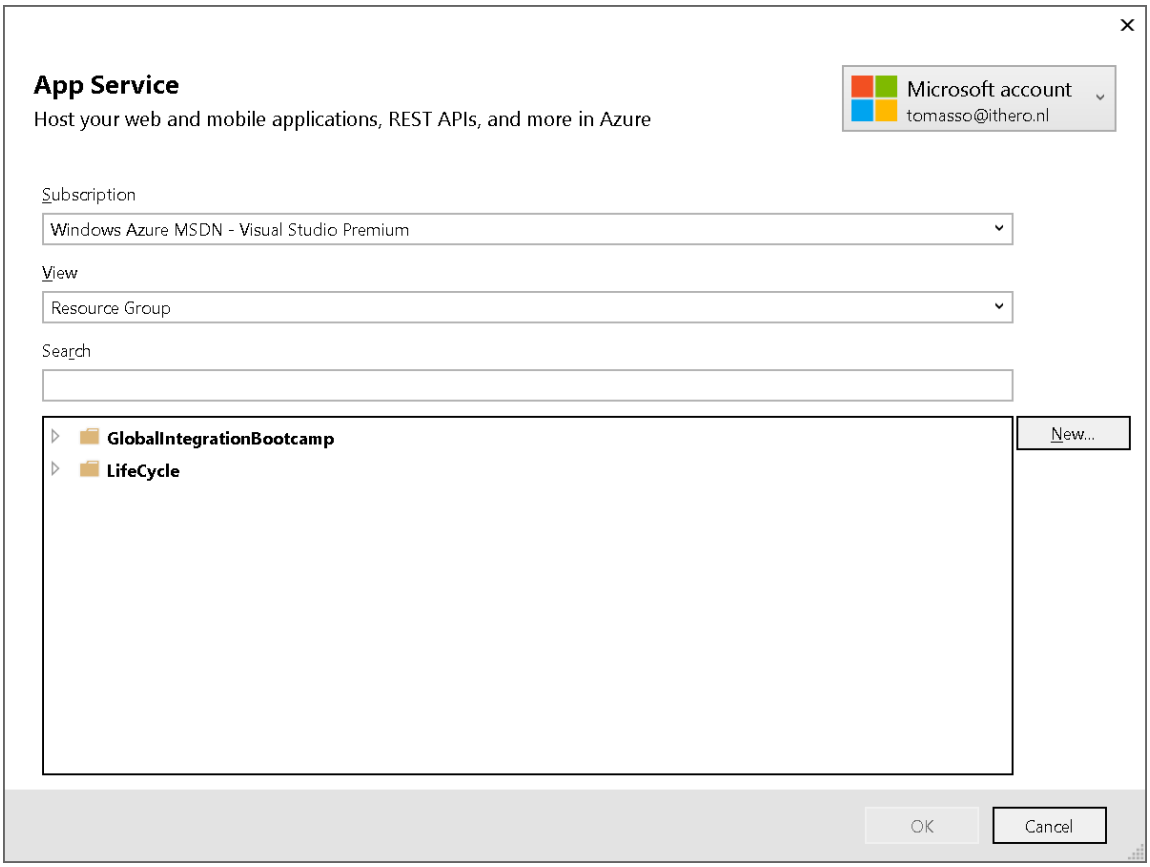
In this section, you use Azure tools that are integrated into the Visual Studio **Publish Web** wizard to create a new API app in Azure. Then you deploy the Order API project to the new API app and call the API by running the Swagger UI.

1. In **Solution Explorer**, right-click the Order API project, and then click **Publish**.

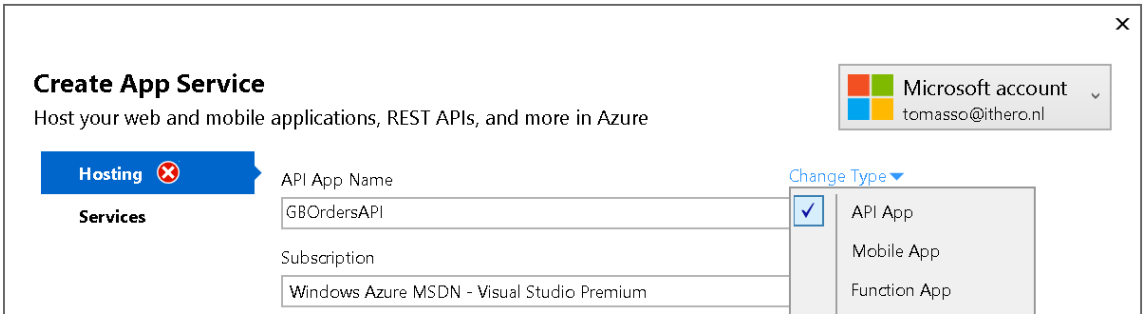
2. In the **Profile** step of the **Publish Web** wizard, click **Microsoft Azure App Service**.



3. Sign in to your Azure account if you have not already done so, or refresh your credentials if they're expired.
4. In the App Service dialog box, choose the Azure Subscription you want to use, and then click New.



The Hosting tab of the Create App Service dialog box appears.
Because you're deploying a Web API project that has Swashbuckle installed, Visual Studio assumes that you want to create an API App. This is indicated by the API App Name title and by the fact that the Change Type drop-down list is set to API App.



5. Enter an **API App Name** that is unique in the *azurewebsites.net* domain. You can accept the default name that Visual Studio proposes.

   If you enter a name that someone else has already used, you see a red exclamation mark to the right.

   The URL of the API app will be {API app name}.azurewebsites.net.

6. In the **Resource Group** drop-down, click **New**, and then enter "GlobalIntegrationBootcamp" or another name if you prefer.
   A resource group is a collection of Azure resources such as API apps, databases, VMs, and so forth. For this tutorial, it's best to create a new resource group because that makes it easy to delete in one step all the Azure resources that you create for the tutorial.

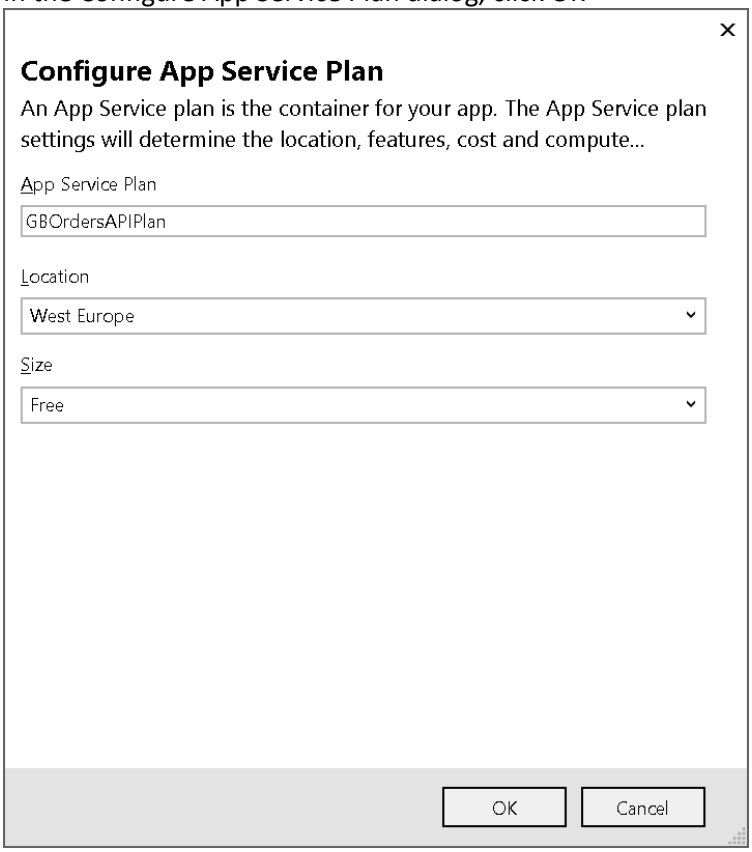
   This box lets you select an existing resource group or create a new one by typing in a name that is different from any existing resource group in your subscription.
7. Click the **New** button next to the **App Service Plan** drop-down.

   The screen shot shows sample values for **API App Name**, **Subscription**, and **Resource Group** -- your values will be different.

   In the following steps you create an App Service plan for the new resource group. An App Service plan specifies the compute resources that your API app runs on. For example, if you choose the free tier, your API app runs on shared VMs, while for some paid tiers it runs on dedicated VMs. For information about App Service plans, see App Service plans overview.
8. In the Configure App Service Plan dialog, enter "GlobalIntegrationPlan" or another name if you prefer.
9. In the Location drop-down list, choose the location that is closest to you.

   This setting specifies which Azure datacenter your app will run in. Choose a location close to you to minimize latency.
10. In the Size drop-down, click Free.
    For this tutorial, the free pricing tier will provide sufficient performance.

11. In the Configure App Service Plan dialog, click OK



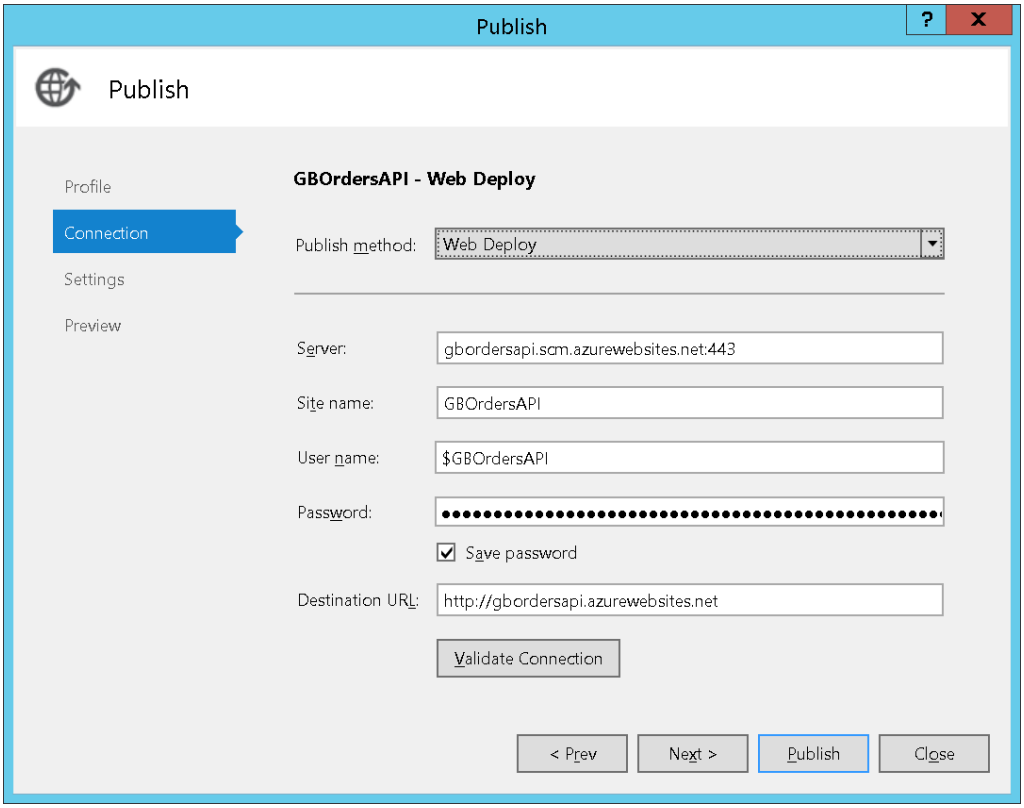
12. In the **Create App Service** dialog box, click **Create**.

    Visual Studio creates the API app and a publish profile that has all of the required settings for the API app. Then it opens the **Publish Web** wizard, which you'll use to deploy the project.

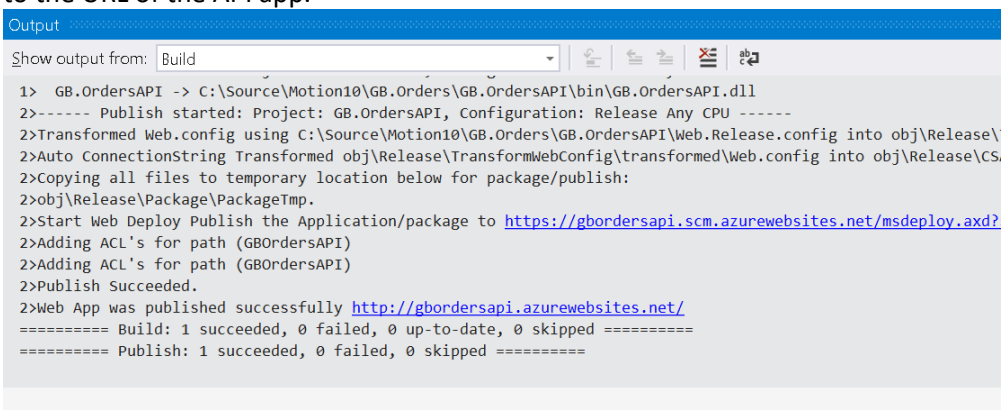
    The **Publish Web** wizard opens on the **Connection** tab (shown below).

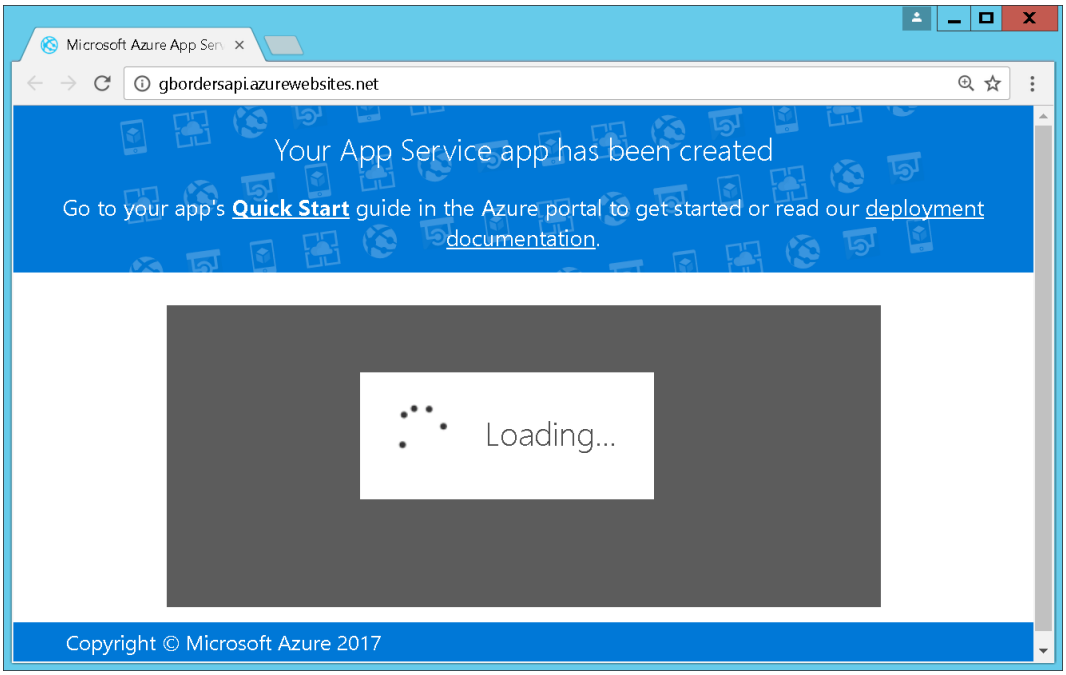
    On the **Connection** tab, the **Server** and **Site name** settings point to your API app. The **User name** and **Password** are deployment credentials that Azure creates for you. After deployment, Visual Studio opens a browser to the **Destination URL** (that's the only purpose for **Destination URL**).

13. Click **Next**.

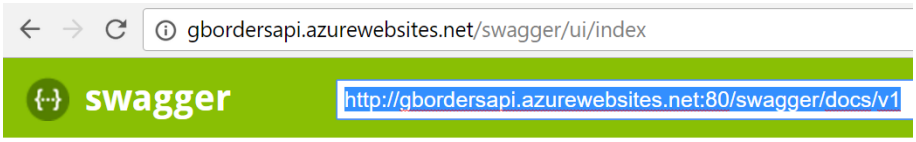


14. Click **Next**.
15. Click **Publish.**
    Visual Studio deploys the Order API project to the new API app. The **Output** window logs successful deployment, and a "successfully created" page appears in a browser window opened to the URL of the API app.





16. In your browser address bar, add swagger to the end of the line, and then press Return.
17. Copy the Swagger metadata URL (The URL is http:// {your api} /swagger/docs/v1)
    This is the default URL used by Swashbuckle to return Swagger 2.0 JSON metadata for the API



GB.OrdersAPI

Note
The Swagger metadata URL is needed in the next lab to import an API App in API Management.

Manage your API in Azure API Management

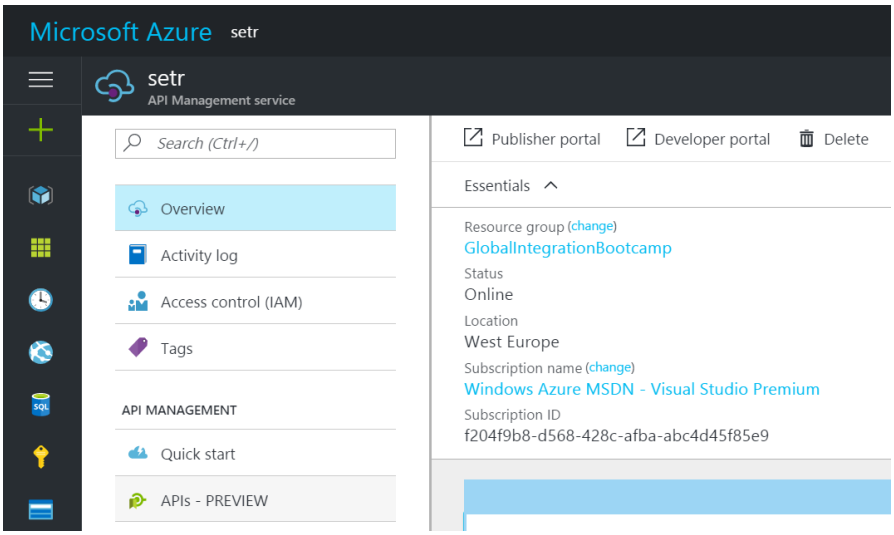
Import the Order API in API Management

APIs can be created (and operations can be added) manually, or they can be imported. In this lab, we will import the API for a sample calculator web service provided by Microsoft and hosted on Azure.

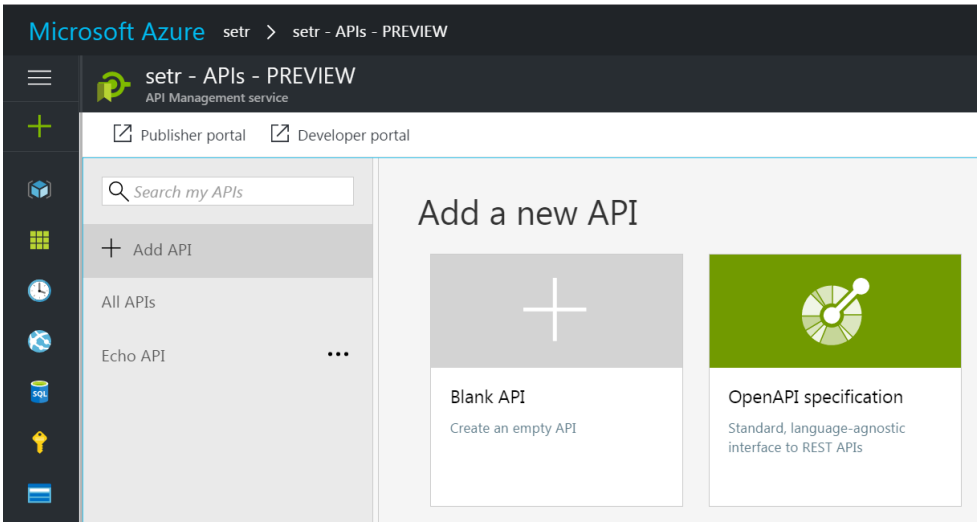
Note

Microsoft is in the process of transitioning functionality from the Publisher Portal to the Azure Portal. New functionality will first appear with a " - PREVIEW" label. When it becomes finalized the preview labels will be removed and the Publisher Portal will no longer be accessible.

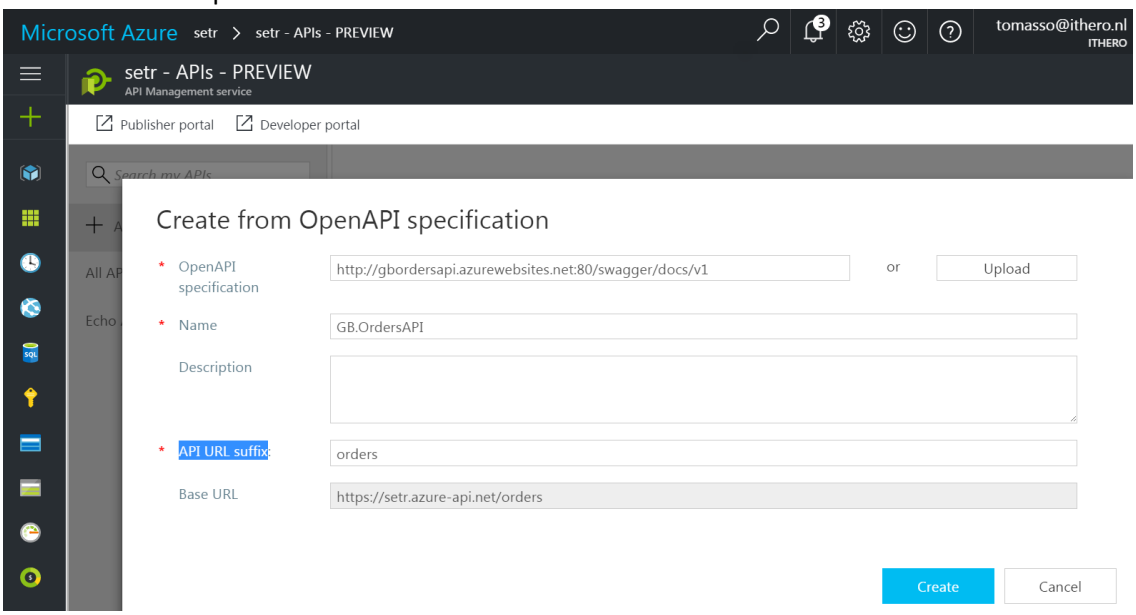
1. APIs can now also be configured from the Azure portal. To reach it, click **APIs - PREVIEW** from the toolbar.



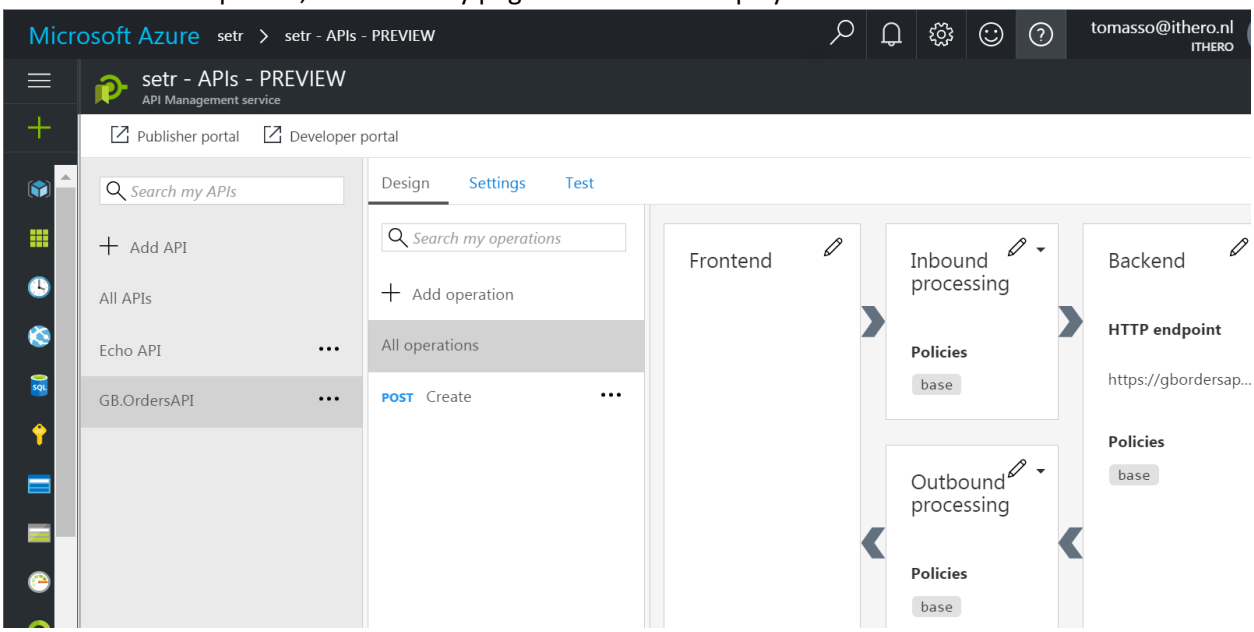
2. To import the Order API, click on **OpenAPI specification**.



3. Perform the following steps to configure the Order API:
   Enter the metadata swagger url from the Order API into the OpenAPI specification text box, and click the Tab button the select the next field.
   Type orders into the API URL suffix text box.
4. Click Create to import the API.



5. Once the API is imported, the summary page for the API is displayed.



Add the Order API to the Starter product

By default, each API Management instance comes with two sample products:

- **Starter**

- **Unlimited**

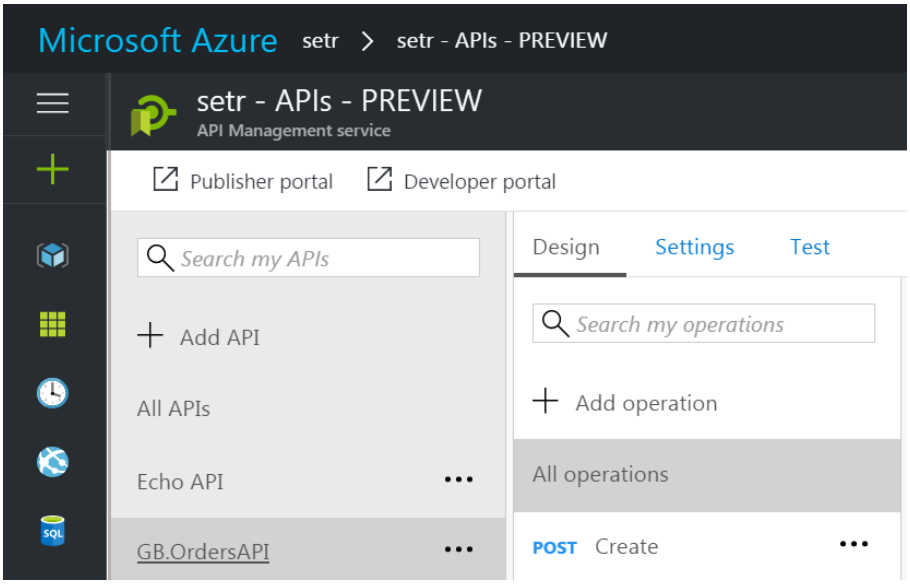
In order to make calls to an API, developers must first subscribe to a product that gives them access to it. Developers can subscribe to products in the developer portal, or administrators can subscribe developers to products in the publisher portal. You are an administrator since you created the API Management instance in the previous steps in the tutorial, so you are already subscribed to every product by default.

In this lab, we are going to add the Order API to the Starter product.

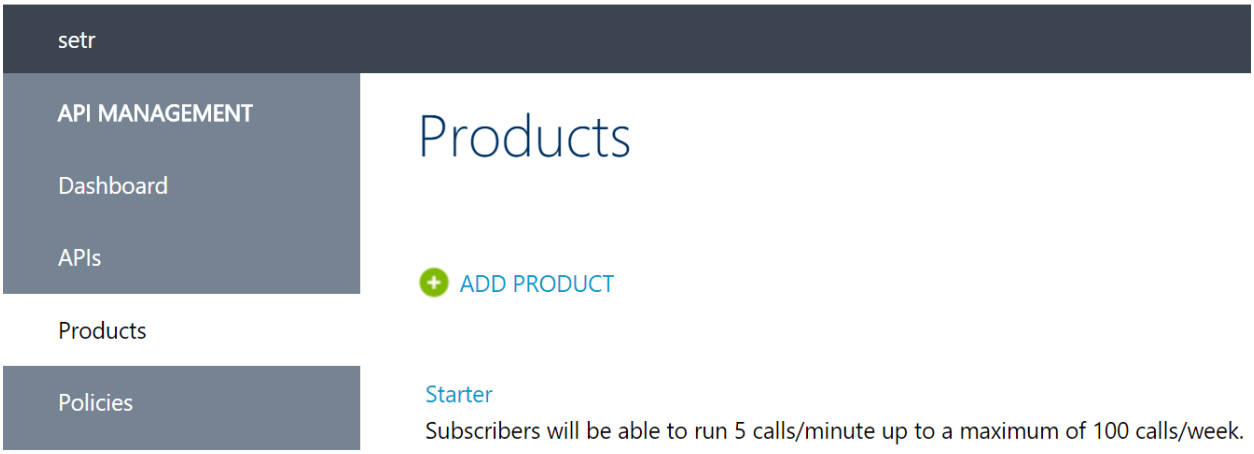
Note

Products are still being transitioned from the Publisher Portal and not yet available in the Azure Portal.

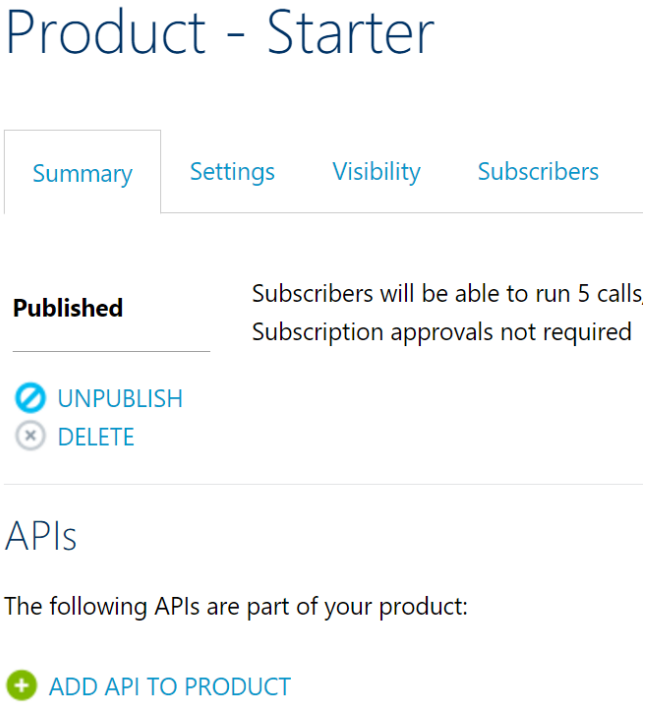
1. Click on Publisher portal link to open the API Management Publisher Portal.



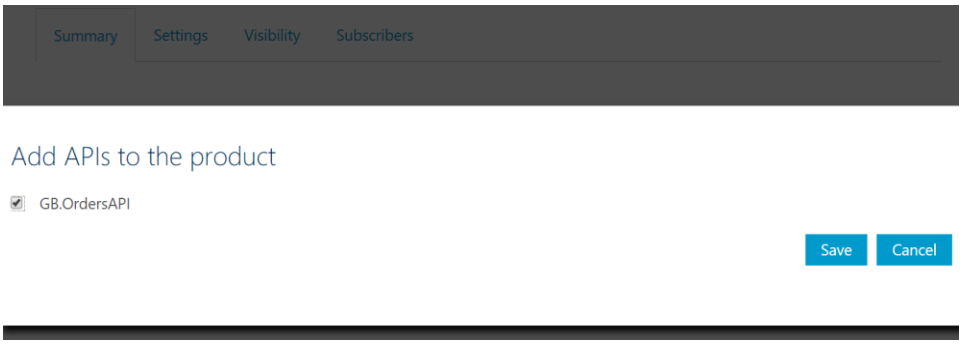
2. Click in the menu on Products and then on the Starter product name.



3. The **Starter Product** page contains four links for configuration: **Summary**, **Settings**, **Visibility**, and **Subscribers**. The **Summary** tab is where you can add anAPI.
   Click on the ADD API TO PRODUCT button



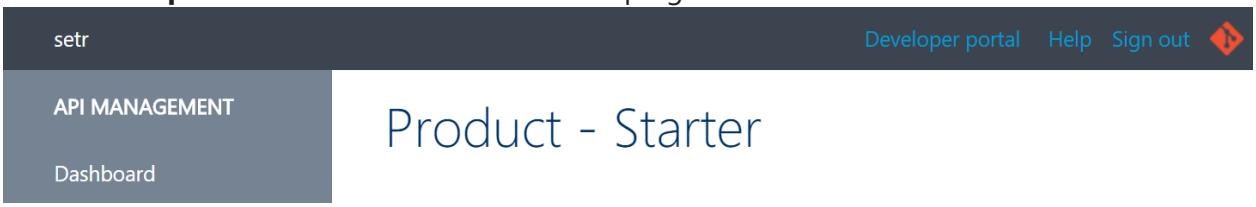
4. Select the Order API and click **Save**.



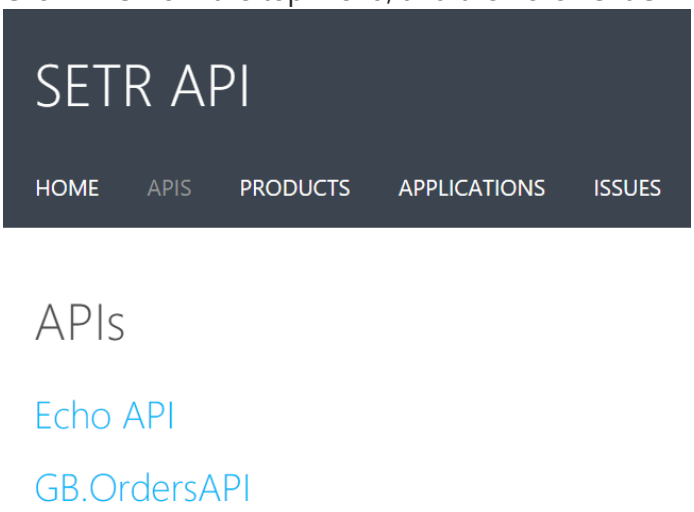
Call the Order API from the Developer Portal

Operations can be called directly from the developer portal, which provides a convenient way to view and test the operations of an API. In this step, you will call the Order API's **Create** operation.

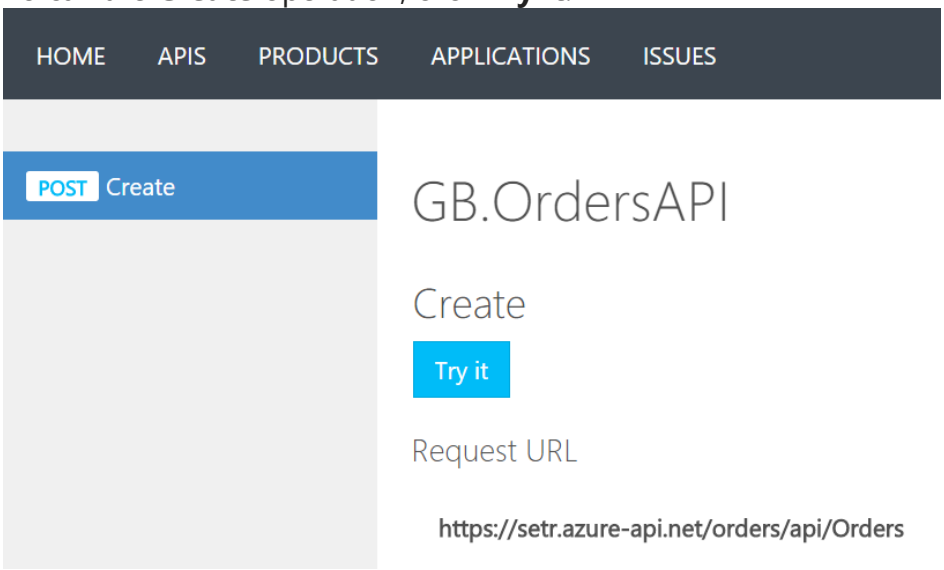
1. Click **Developer Portal** from the menu at the top right of the Publisher Portal.



2. Click **APIs** from the top menu, and then click **Order API** to see the available operations.



3. Note the sample descriptions and parameters that were imported along with the API and operations, providing documentation for the developers that will use this operation. These descriptions can also be added when operations are added manually.
To call the **Create** operation, click **Try it**.



4. You can modify the content of the body and then click **Send**.

```
  "Order": {
    "Customer": {
      "CustomerNumber": "DEV01"
    },
    "Products": [
      {
        "ProductNumber": 10000,
        "Amount": 1
      }
    ],
    "OrderedDateTime": "2017-03-07T12:03:18.605Z"
  }
}
```

Authorization

Subscription key Primary-9954...

Request URL

```
https://setr.azure-api.net/orders/api/Orders
```

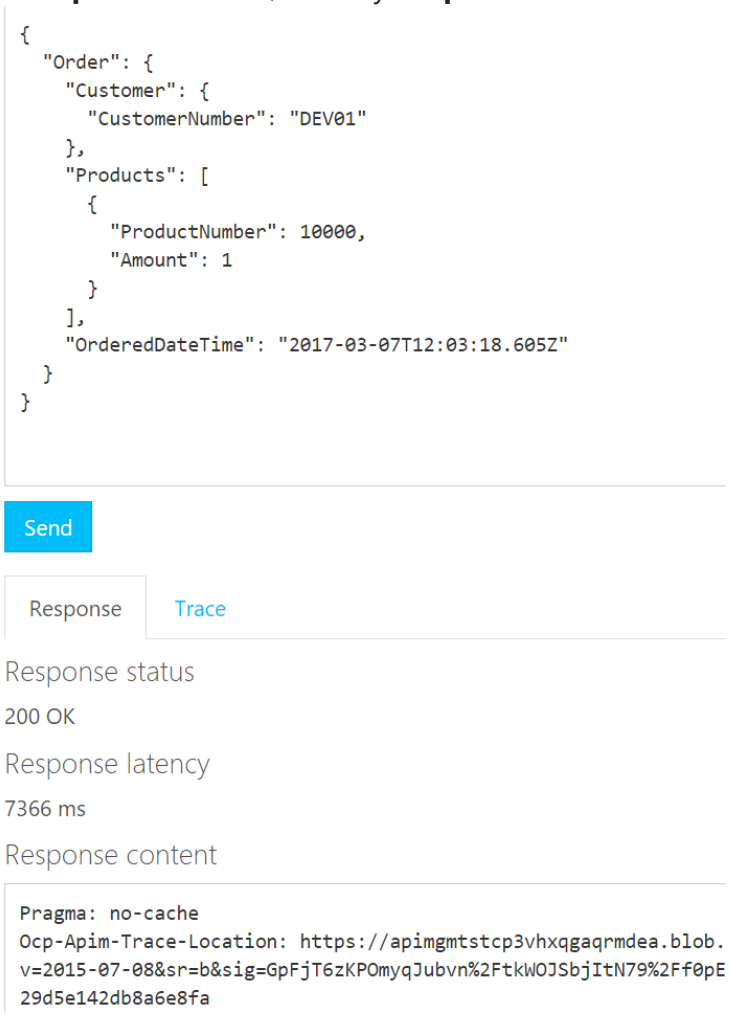
HTTP request

```
POST https://setr.azure-api.net/orders/api/Orders HTTP/1.1
Content-Type: application/json
Host: setr.azure-api.net
Ocp-Apim-Trace: true
Ocp-Apim-Subscription-Key: ••••••••••••••••••••••••••••••••
```

```
{
  "Order": {
    "Customer": {
      "CustomerNumber": "DEV01"
    },
    "Products": [
      {
        "ProductNumber": 10000,
        "Amount": 1
      }
    ],
    "OrderedDateTime": "2017-03-07T12:03:18.605Z"
  }
}
```

Send

5. After an operation is invoked, the developer portal displays the **Response status**, the **Response headers**, and any **Response content**.

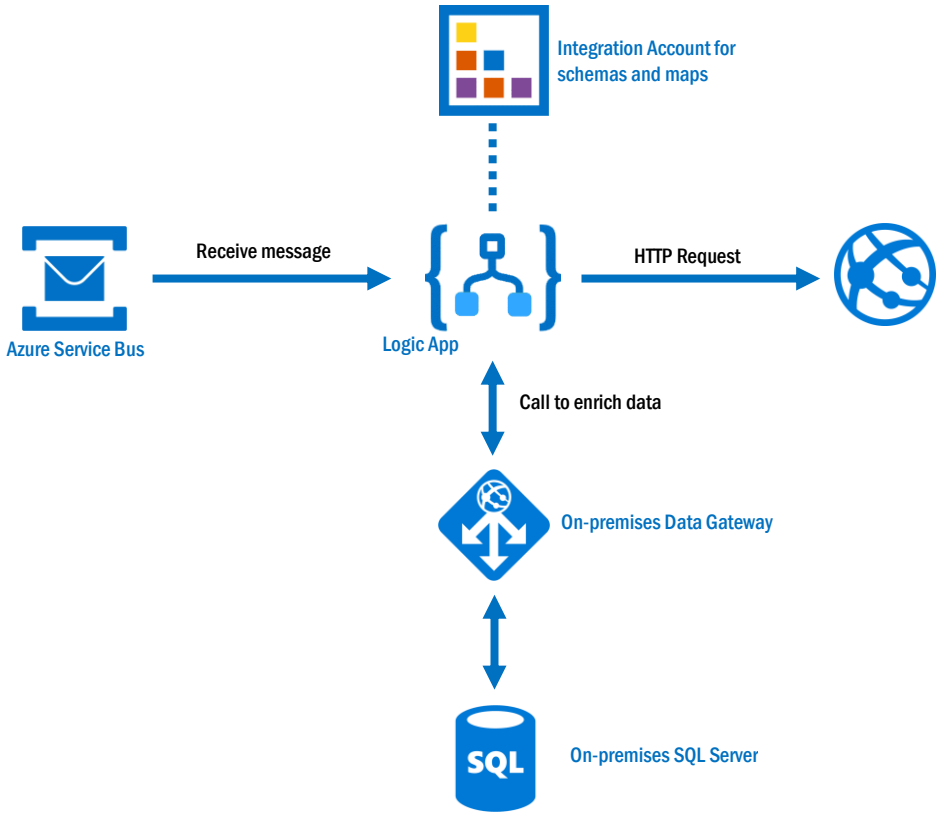


Lab 2 - Service Bus + Enterprise Integration Pack + On-premises Data Gateway

Objective

In this second lab, we will be creating a Logic App, which reads the data from the queue in our previous lab. As SETR has its customer data in an on-premises database, and does not want to move this to the cloud at this moment, we will need to retrieve the customer data from this database. For optimal security, they do not want to expose their database directly to the internet, so instead we will be using the On-Premises Data Gateway to connect to the database in an easy and secure manner. We will retrieve the data for the customer, and enrich our message with this data, after which we will call our next logic apps.

The logic app receives a message, which needs to be validated and mapped to the on-premises data format. The On-Premises Data Gateway will then be used to interact with the on-premises SQL Server to retrieve data.



Prerequisites

- Azure Subscription
- **Visual Studio 2015** IDE with **Microsoft Azure Logic Apps Enterprise Integration Tools for Visual Studio 2015 2.0** installed.
    - https://www.visualstudio.com/downloads/
    - https://www.microsoft.com/en-us/download/details.aspx?id=53016
- An on-premises machine with **SQL Server 2016** installed and configured. This on-premises machine can also be an Azure Virtual Machine with SQL Server 2016.

Steps

To build the solution in this lab you have to follow the steps described in this section. From a high level view the steps are:

1. Install and configure On-Premises Data Gateway
2. Register the On-Premises Data Gateway in Azure
3. Import the LegacyOrderSystem data in the on-premises SQL Server
4. Create a schema for validation in Visual Studio 2015
5. Create an Integration Account
6. Upload the schema to the Integration Account
7. Provision a Logic App
8. Associate the Integration Account with the Logic App
9. Build Logic App Definition
10. Test the Solution

Install and configure the On-Premises Data Gateway

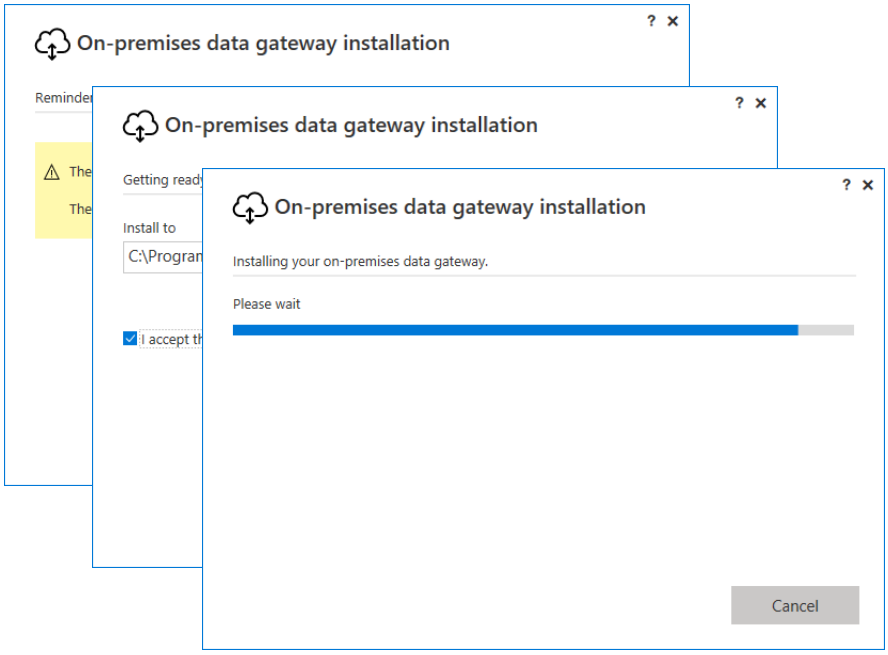
Since we are going to use the On-Premises Data Gateway, the gateway should be installed and configured first.

The On-Premises Data Gateway (OPDGW) should be installed the on-premises machine containing SQL Server.

1. Download the On-Premises Data Gateway:

   https://www.microsoft.com/en-us/download/details.aspx?id=53127

2. Install the On-Premises Data Gateway



3. Configure the On-Premises Data Gateway

   After successful installation, it should be configured to be used with Azure.

   The On-Premises Data Gateway **does not work with a Microsoft account**. You should use a work or school account.

   If you are using a Microsoft account, follow the steps below (**step I**) to add a user to the active directory and setup the gateway.

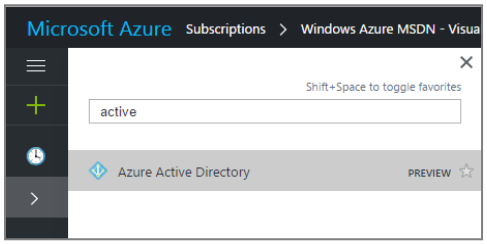
   If you are using a work or school account, you can proceed to **step II**.

   I. Adding a user to an active directory (for non-school or work account users)

The On-Premises Data Gateway needs a user to be setup in the Azure active directory to function properly.

a. Login to the Azure Portal by navigating to https://portal.azure.com.

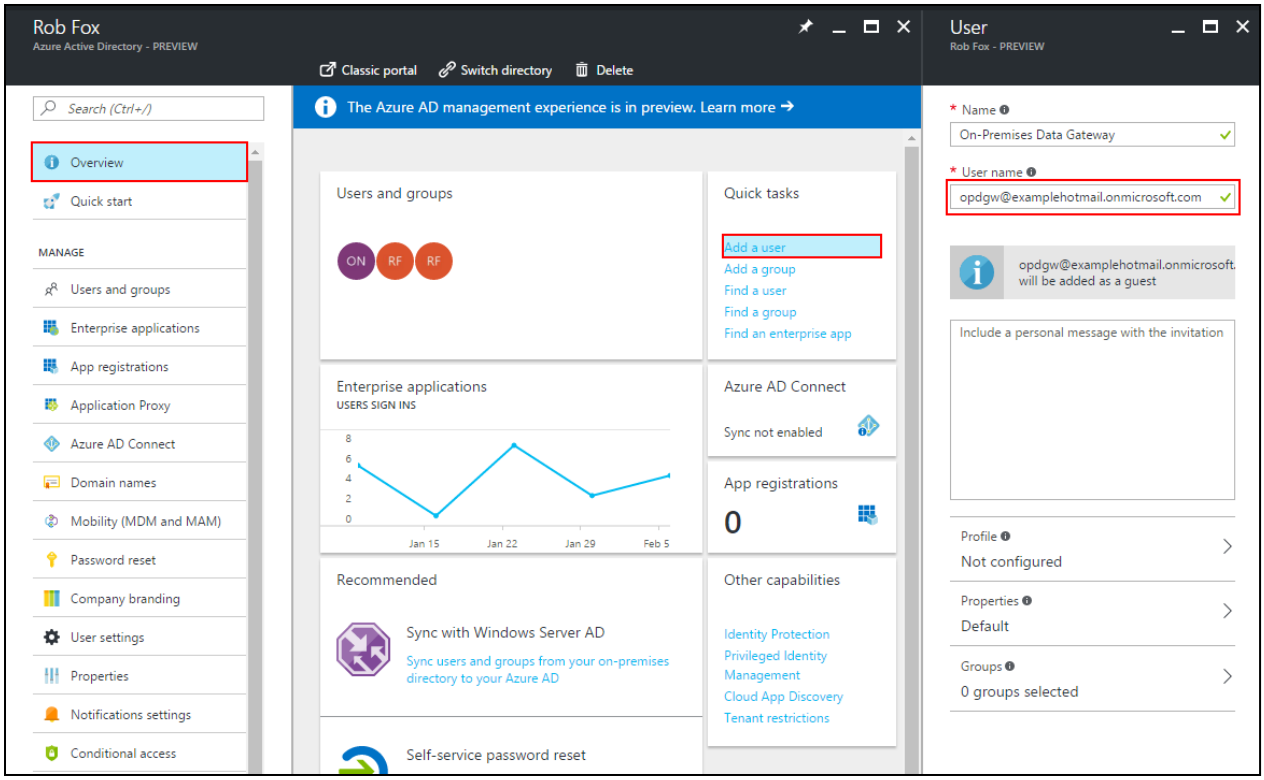
b. Navigate to the Azure Active Directory by searching for it in the resources directory. And open it.



c. In the overview click ***Add a user*** and create a new user for your On-Premises Data Gateway. This user **must have an extension of onmicosoft.com**.

Example

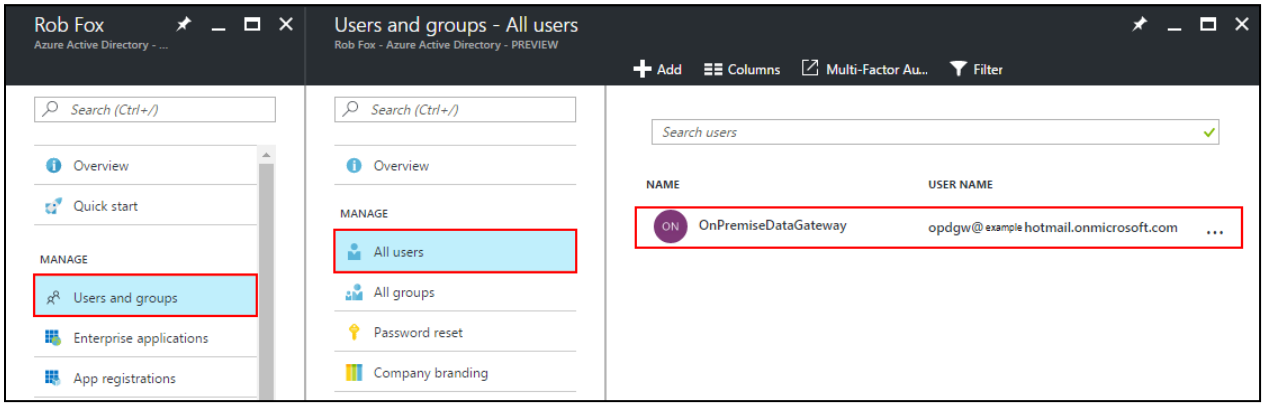
- If your Microsoft account is example@hotmail.com, the OPDGW user should be named <anything>@examplehotmail.onmicrosoft.com.
- If your Microsoft account is anotherexample@outlook.com, the OPDGW user should be name <anything>@anotherexample.onmicrosoft.com.

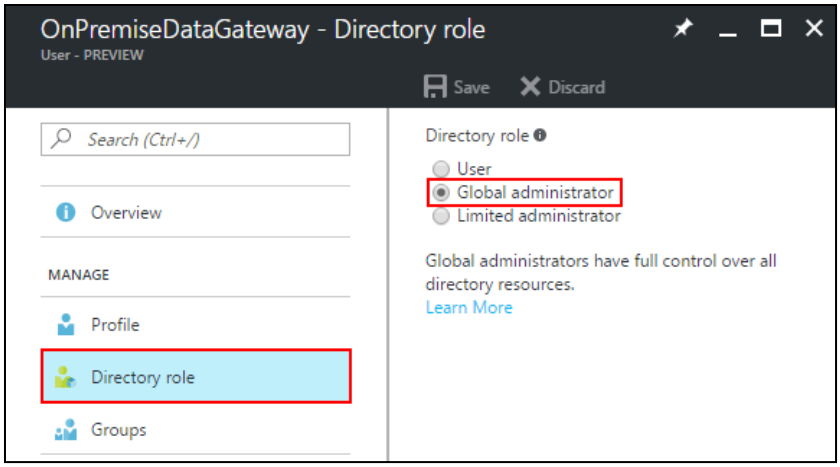


d. Set the appropriate rights for the user.

Select ***Users and groups*** and navigate to the user you just created. Now select the user and navigate to ***Directory Role***.

Make sure the user is set to ***Global Administrator*** and click Save (if applicable).

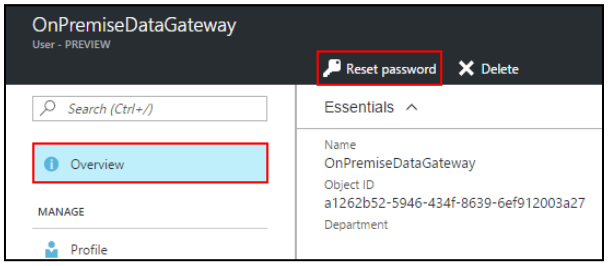




e. Reset the password of the user.

And now that we are here, the user's password should be reset. In ***Overview*** click ***Reset password***. This action will create a temporary password for the user.

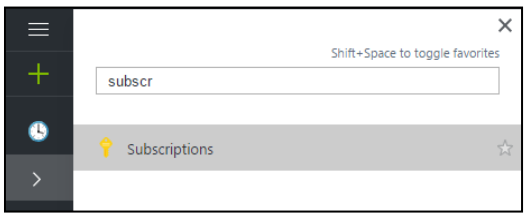
Make sure you copy it or save it for later!

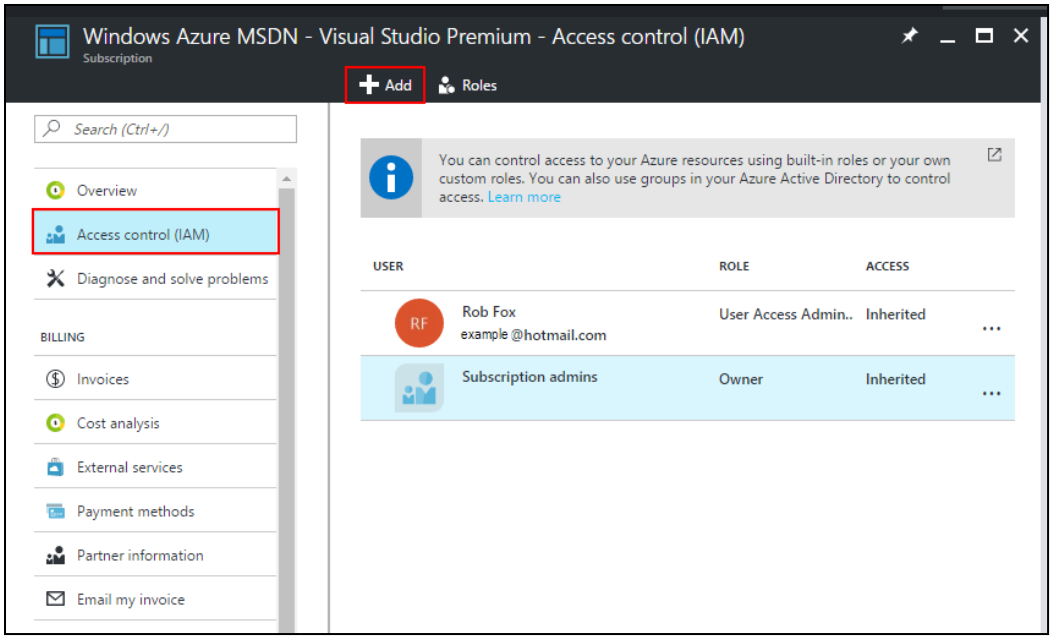


f. Make sure the new user has access to your subscription.

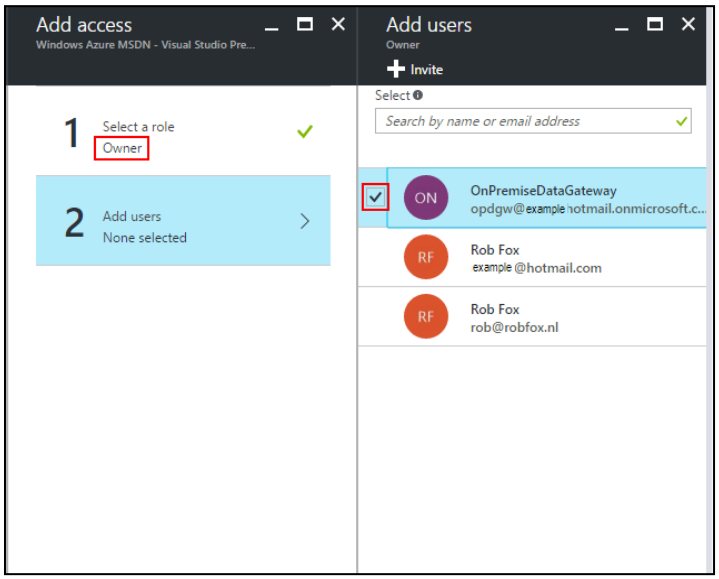
By enabling access, the user doesn't need his own subscription. Everything the user does, is part of your subscription. This way the data gateway can be added under your subscription and not under a new subscription of the user you just created.

First go to your ***Subscriptions*** and select the appropriate subscription. Now under ***Access Control*** select ***Add*** to add users to your subscription.





After clicking Add you'll end up at the following screen. First select ***Owner*** as the **role.** The second step adds the users. Select the user you just created for the On-Premises Data Gateway and apply by clicking ***Select***.



g. Login with the newly created user and reset the password

The password that was given after doing the password reset, was just a temporary password. This password must be reset by the user itself. The user will be asked to update the password on their next login.

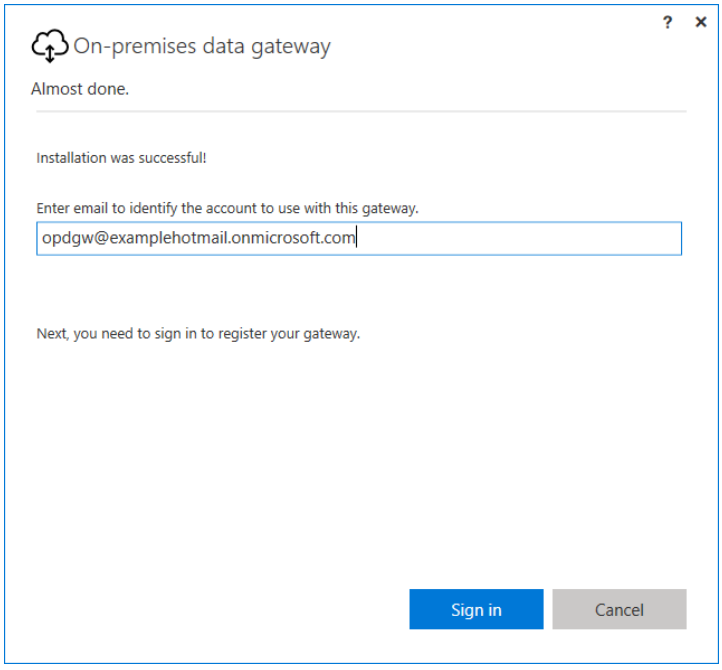
To change the password, open an in-cognito (or in private) browser window, or you can choose to sign out of the portal. After opening the new in-cognito browser window, or after signing out, navigate to https://portal.azure.com and login with the On-Premises Data Gateway user by supplying the e-mail address and the temporary password created during the reset a few steps back and set your own password.

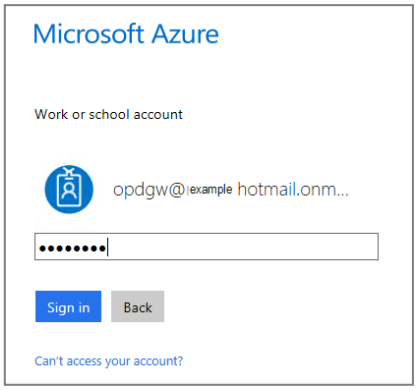
So, now we should finally be set to configure the On-Premises Data Gateway.

II. Configure the gateway

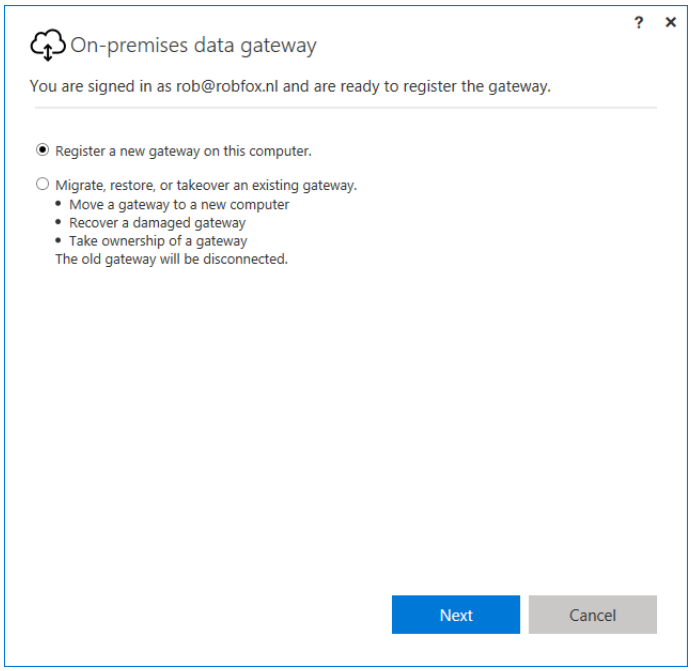
a. On the on-premises machine containing SQL server, start the On-Premises Data Gateway configuration. This should start automatically after installation.

Login with your work or school account.



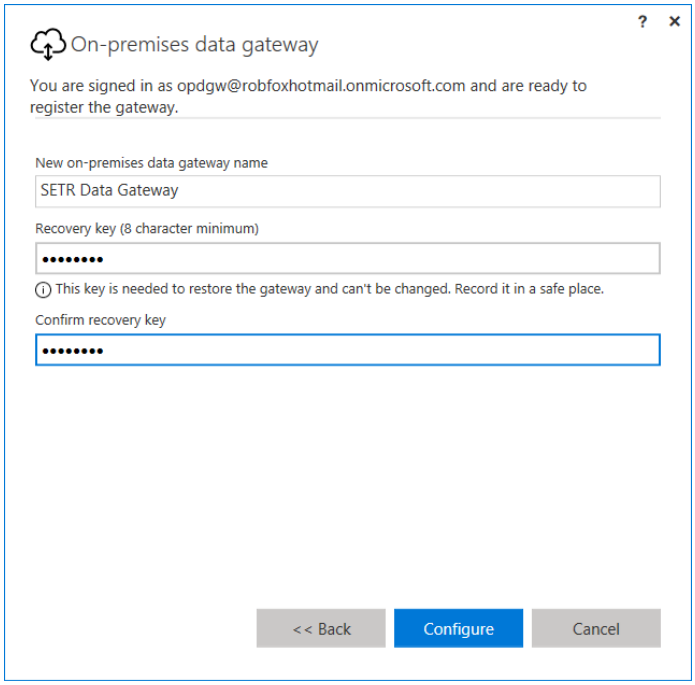


b. On the second screen choose ***Register a new Gateway on this computer***.

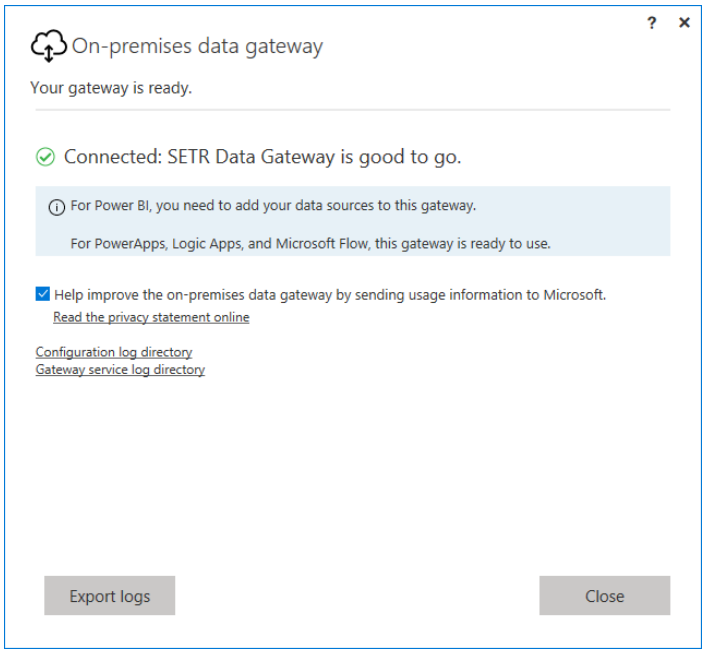


c. In the next screen enter the ***name of your gateway*** and enter a ***recovery key***. These values are totally up to you.

This step may take some time and does not always end successfully. If registration was not successful, repeat this step, until it works.



d. If registration has been successful, you will see the following screen. If registration was not successful, you'll need to repeat **step d** until it works.

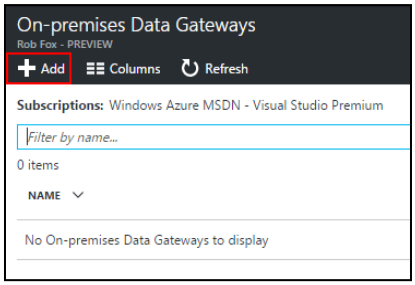


Register the gateway in Azure

1. Go to the Azure portal and ***sign in*** with the **account you used to register the On-Premises Data Gateway with**. The example in this lab was opdgw@examplehotmail.onmicrosoft.com.

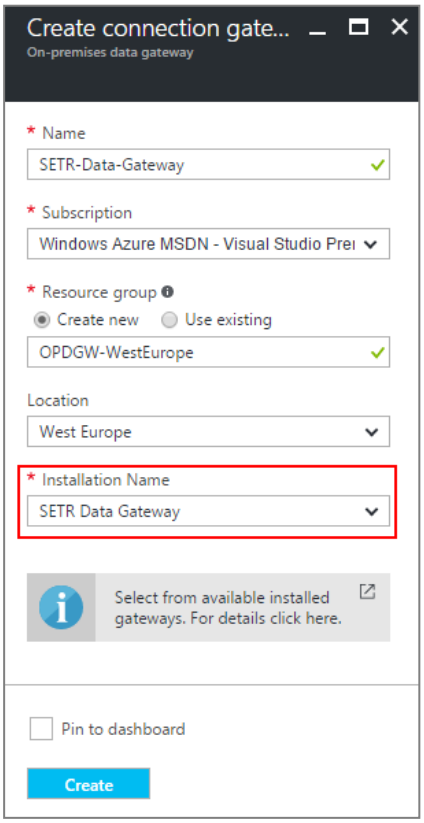
   https://portal.azure.com

2. After logging in, ***navigate to the On-Premises Data Gateway blade*** and click ***Add*** to register the gateway.



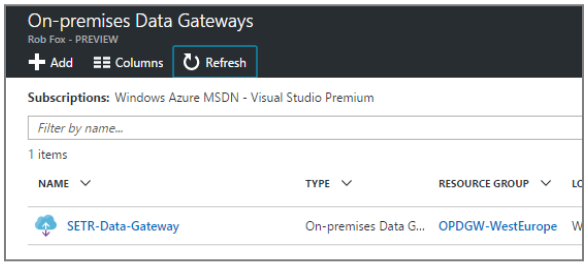
3. Now give your On-Premises Data Gateway a suitable ***name*** to identify it in Azure. Select your ***subscription*** and a ***Resource group*** (you can add a new one if you want). Select the desired ***Location***. I.e. don't select West-Europe if you are in Australia ☺. Please note that the Gateway should be in the same location as the Logic App and the Integration Account which will be created later on.

   Lastly ***select the On-Premises Data Gateway*** you just installed on your local machine. This should be visible in the dropdown box. Click ***Create*** to finish registration.



4. After successful registration, you should be able to see it in your list of gateways, which basically means we're done here.

   For everyone using a Microsoft account, you can logout with the Gateway account and login again with your Microsoft account.



Import the LegacyOrderSystem data in the on-premises SQL Server

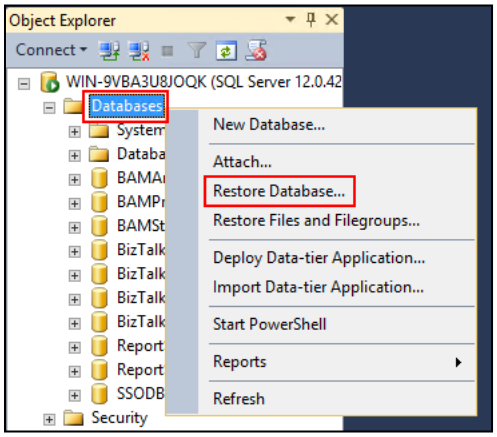
The On-Premises Data Gateway will be used to enrich the message we receive in the cloud with on-premises data.

1. Download the database backup.
   The data used for this lab can be downloaded from here.
   After downloading, unpack the zip file on the machine with SQL Server installed.

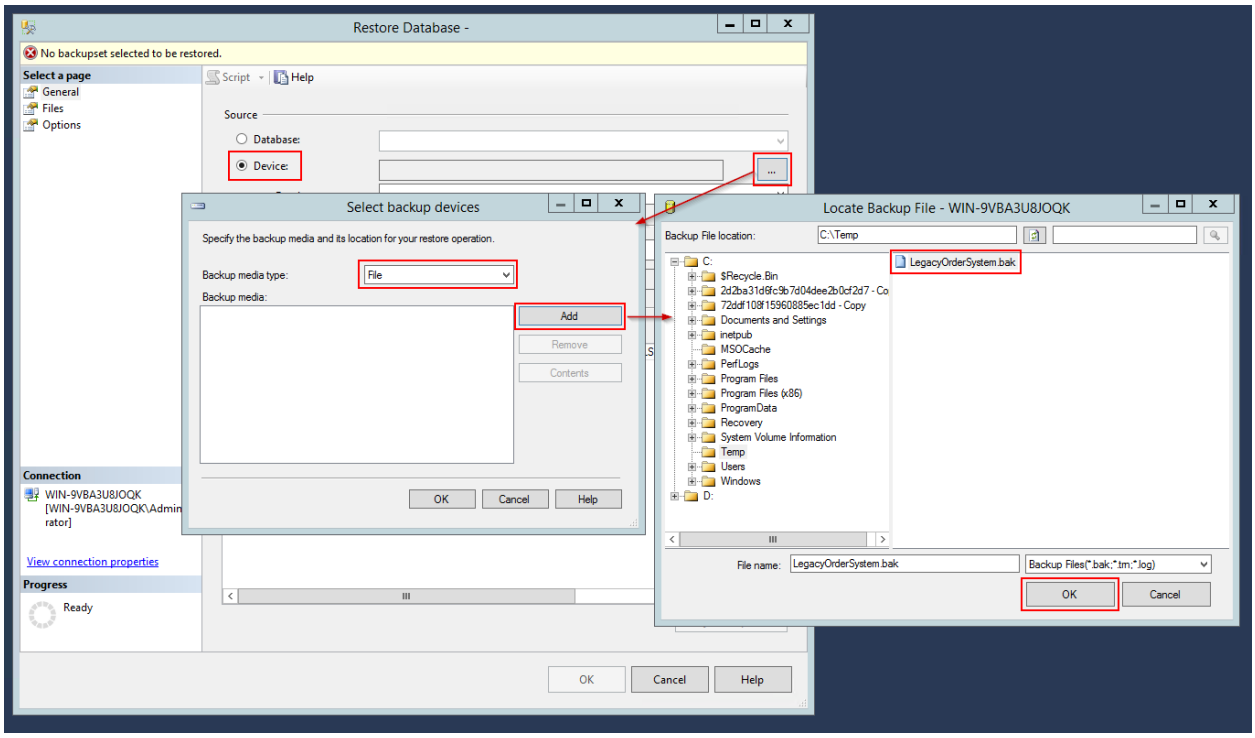
2. Restore the backup

   a. Open **SQL Server Management studio** on the local machine and connect to your local instance.

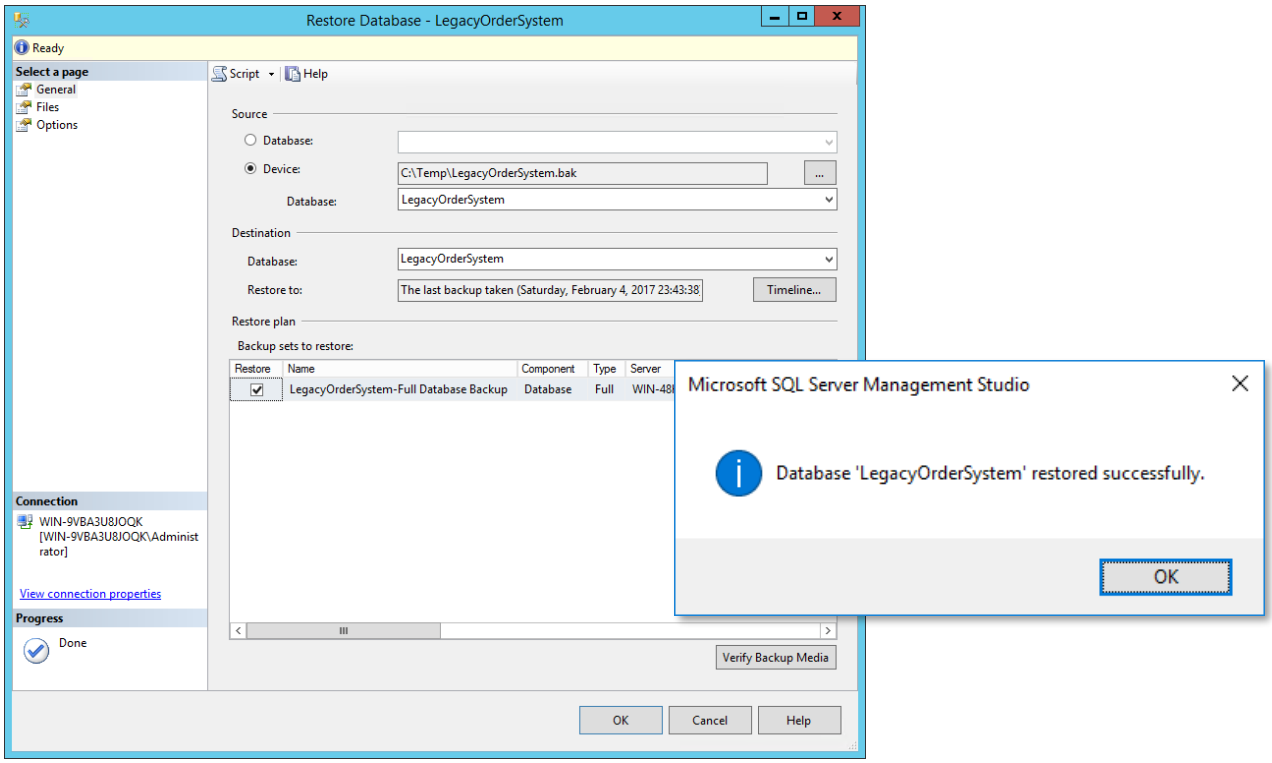
   b. In Object Explorer, ***click right on Databases*** and select restore



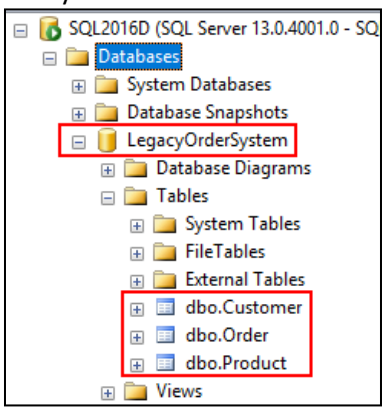
   c. Now, in the General page, select ***Device*** and ***click the elipses (…)***. Select ***File*** as Backup media type and click ***Add***. Now navigate to the **LegacyOrderSystem.bak** file and ***click OK***. Click **OK** once again to also close the other dialog.



d. The result should look like the window below. Now **click OK** to restore the backup.



e. Verify the database has been restored correctly, by navigating to it in Object Explorer.



f. Finally run the following script to import the data for this database.

```
USE [LegacyOrderSystem]
GO

DECLARE @RC int
DECLARE @CustomerNumber nvarchar(50)
DECLARE @Placed datetime
DECLARE @TotalAmount decimal(10,2)
DECLARE @Company nvarchar(50)
DECLARE @Email nvarchar(50)
DECLARE @Street nvarchar(50)
DECLARE @City nvarchar(50)
DECLARE @PostalCode nvarchar(10)
DECLARE @Country nvarchar(50)
DECLARE @Products [dbo].[ProductList]

DECLARE @ProductsToLoad TABLE (custNum int, prodNum int, amt int, prc
decimal(10,2))
DECLARE @OrdersToLoad TABLE (RowID int not null primary key identity (1,1),
custNum nvarchar(50), datePlaced datetime, comp nvarchar(50), emailAddress
nvarchar(50), st nvarchar(50), cty nvarchar(50), pcode nvarchar(10), cntry
nvarchar(50))
DECLARE @RowsToProcess int
DECLARE @CurrentRow int

-- DECLARE DATA TO LOAD

INSERT INTO @ProductsToLoad
      SELECT 1, 15267,  1,    2499  UNION ALL
      SELECT 1, 26453,  2,    499         UNION ALL
      SELECT 2, 15326,  1,    3495  UNION ALL
      SELECT 3, 10023,  20, 9.99    UNION ALL
      SELECT 3, 47423,  1,    79          UNION ALL
      SELECT 4, 35437,  1,    299         UNION ALL
      SELECT 4, 10023,  30, 9.99    UNION ALL
      SELECT 5, 20034,  1,    5           UNION ALL
      SELECT 5, 33658,  1,    1299  UNION ALL
      SELECT 5, 90573,  2,    29.99 UNION ALL
```

```
      SELECT 6, 30004,  6,    879         UNION ALL
      SELECT 6, 28854,  3,    3495

INSERT INTO @OrdersToLoad
      SELECT 'C0001', GetDate()-20, 'Mexia',
      'dan@mexia.com.au',                '33 Longland Street',   'Newstead',
                  '4016',           'Australia'       UNION ALL
      SELECT 'C0002', GetDate()-18, 'Mexia',
      'paco@mexia.com.au',          '385 Bourke Street',    'Melbourne',
            '3000',           'Australia'       UNION ALL
      SELECT 'C0003', GetDate()-17, 'Motion10',
      'eldert@motion10.com',        'Wilhelminakade 175',   'Rotterdam',
            '3009',           'Netherlands'     UNION ALL
      SELECT 'C0004', GetDate()-12, 'SixPivot',
      'bill@sixpivot.com.au',       '5699 Queen Street',    'Brisbane',
            '4000',           'Australia'       UNION ALL
      SELECT 'C0005', GetDate()-7,  'Satalyst Pty Ltd',
      'martin@satalyst.com.au',     '13/979 Albany Hwy',    'East Victoria
Park',      '6101',           'Australia'       UNION ALL
      SELECT 'C0006', GetDate()-2,  'Microsoft',
      'jon@microsoft.com',          '1 Microsoft Way',           'Redmond',
                  '98502',    'USA'

SET @RowsToProcess = @@ROWCOUNT
SET @CurrentRow = 0

-- LOOP THROUGH RECORDS TO POPULATE TABLES

WHILE @CurrentRow < @RowsToProcess
BEGIN
      SET @CurrentRow = @CurrentRow + 1
      DELETE FROM @Products
      INSERT INTO @Products SELECT prodNum, amt, prc FROM @ProductsToLoad
WHERE custNum = @CurrentRow
      SELECT @TotalAmount = SUM(Amount * Price) FROM @Products

      SELECT
            @CustomerNumber= custNum,
            @Placed          = datePlaced,
            @Company         = comp,
            @Email                 = emailAddress,
            @Street          = st,
            @City                  = cty,
            @PostalCode      = pcode,
            @Country         = cntry
      FROM @OrdersToLoad
      WHERE RowID = @CurrentRow


      EXECUTE @RC = [dbo].[InsertOrder]
         @CustomerNumber
        ,@Placed
        ,@TotalAmount
        ,@Company
        ,@Email
```

```
        ,@Street
        ,@City
        ,@PostalCode
        ,@Country
        ,@Products

END
```

Create a schema for validation in Visual Studio 2015

Creating artifacts like schemas and maps for use within Azure requires Visual Studio 2015. This can also be the community edition of Visual Studio 2015. You also need to install the Microsoft Azure Logic Apps Enterprise Integration Tools for Visual Studio 2015 2.0.

To download Visual Studio 2015, go here:

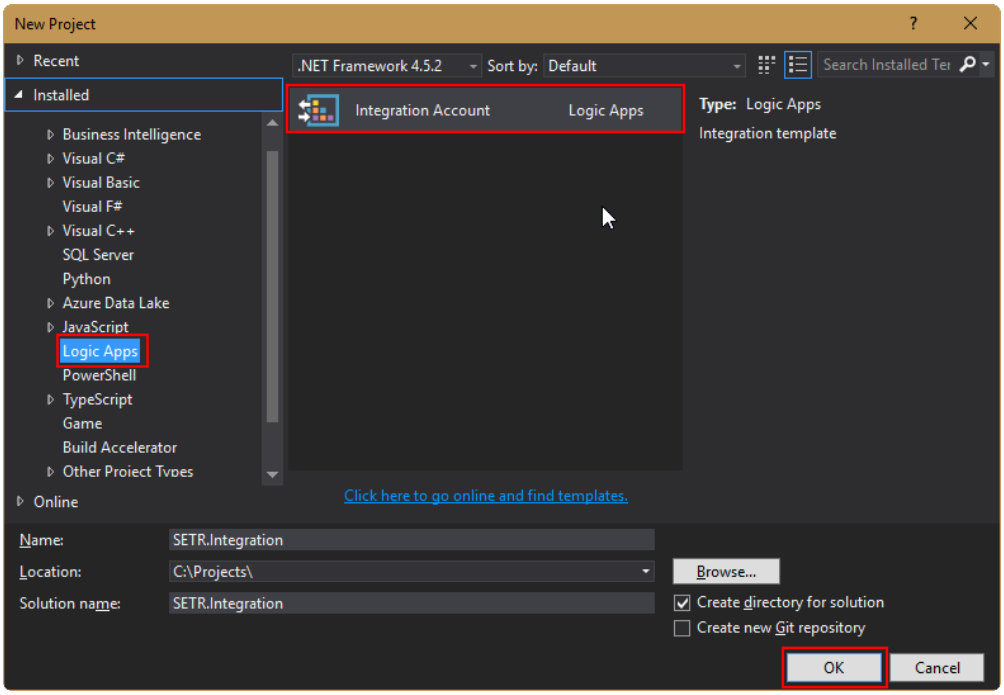
- https://www.visualstudio.com/downloads/

To download the Enterprise Integration tools, go here:

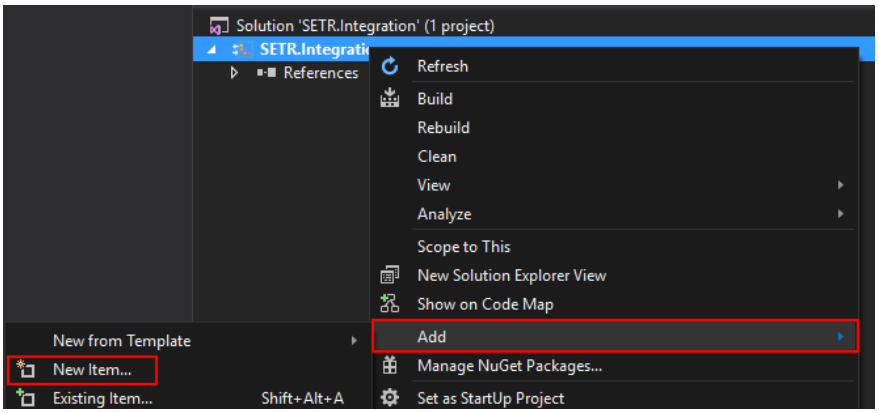
- https://www.microsoft.com/en-us/download/details.aspx?id=53016

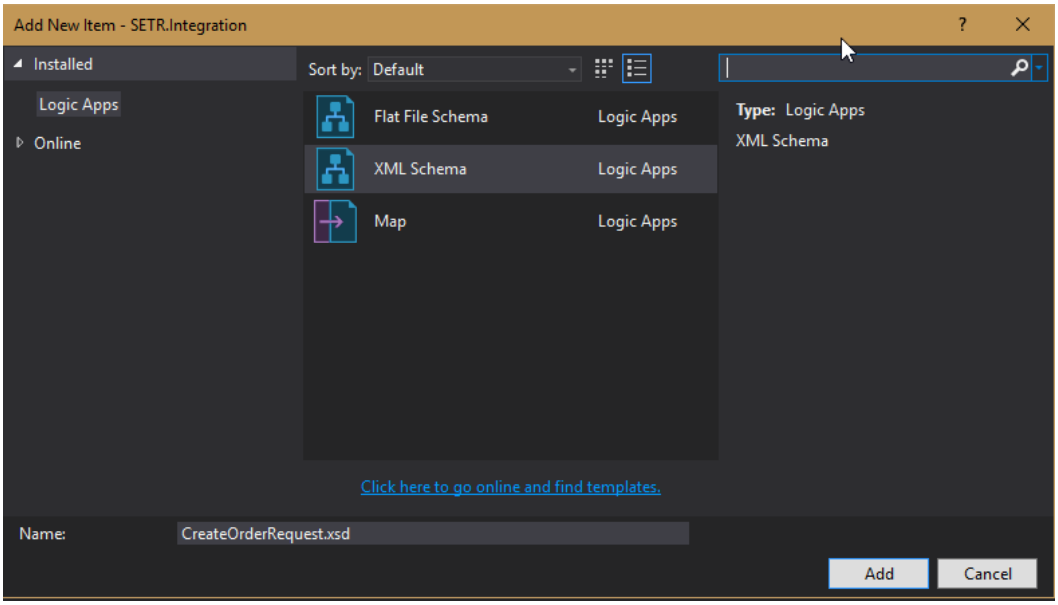
In this lab, it's only intended to get familiar with the Integration accounts. A schema will be created to validate the message that's being received from the service bus queue.

1. Open Visual Studio and create a new Integration Account project.



2. ***Right click the project*** and choose ***Add > New Item*** and choose ***XML Schema***.





3. ***Develop a schema*** that resembles the request message below. The schema doesn't hold a target namespace.

```
<CreateOrder>
      <Order>
            <!-- Mandatory -->
            <Customer>
                  <!-- Mandatory -->
                  <CustomerNumber>String</CustomerNumber>
            </Customer>
            <!-- Mandatory -->
            <Products>
                  <!-- Mandatory -->
                  <!-- 1..* -->
                  <Product>
                        <!-- Mandatory -->
                        <ProductNumber>Integer</ProductNumber>
                        <!-- Mandatory -->
                        <Amount>Integer</Amount>
```

```
                </Product>
                <Product>
                      <ProductNumber>Integer</ProductNumber>
                      <Amount>2</Amount>
                </Product>
            </Products>
            <!-- Mandatory -->
            <OrderedDateTime>DateTime</OrderedDateTime>
      </Order>
</CreateOrder>
```

If you don't know how to develop the XML schema, you can use the XSD code below. Copy the code into the artifact in Visual Studio 2015, or just save it as CreateOrderRequest.xsd somewhere on your disk.

```
<xs:schema xmlns:b="http://schemas.microsoft.com/BizTalk/2003"
attributeFormDefault="unqualified" elementFormDefault="unqualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="CreateOrder">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="1" maxOccurs="1" name="Order">
          <xs:complexType>
            <xs:sequence>
              <xs:element minOccurs="1" maxOccurs="1" name="Customer">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element minOccurs="1" maxOccurs="1"
name="CustomerNumber" type="xs:string" />
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element minOccurs="1" maxOccurs="1" name="Products">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element minOccurs="1" maxOccurs="unbounded"
name="Product">
                      <xs:complexType>
                        <xs:sequence>
                          <xs:element name="ProductNumber" type="xs:integer"
/>
                          <xs:element name="Amount" type="xs:integer" />
                        </xs:sequence>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element minOccurs="1" maxOccurs="1" name="OrderedDateTime"
type="xs:dateTime" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
```

```
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

4. Save the project (or the XSD). There's no need to build the project, the XSD file is the only thing we need to upload to Azure.

   If maps were going to be used, a build was needed. The build only generates an XSLT file you can then upload to Azure.

Create Integration Account

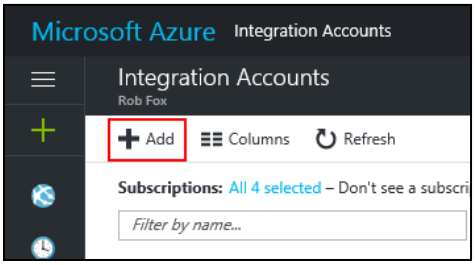
To hold the schema, an Integration Account should be created in Azure. This is a quick and straightforward process.

1. Go to the Azure Portal

   https://portal.azure.com

2. Navigate to the Integration Accounts blade

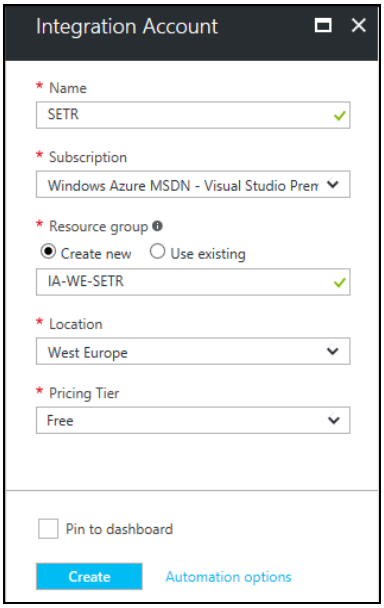
3. Click Add to add a new Integration Account



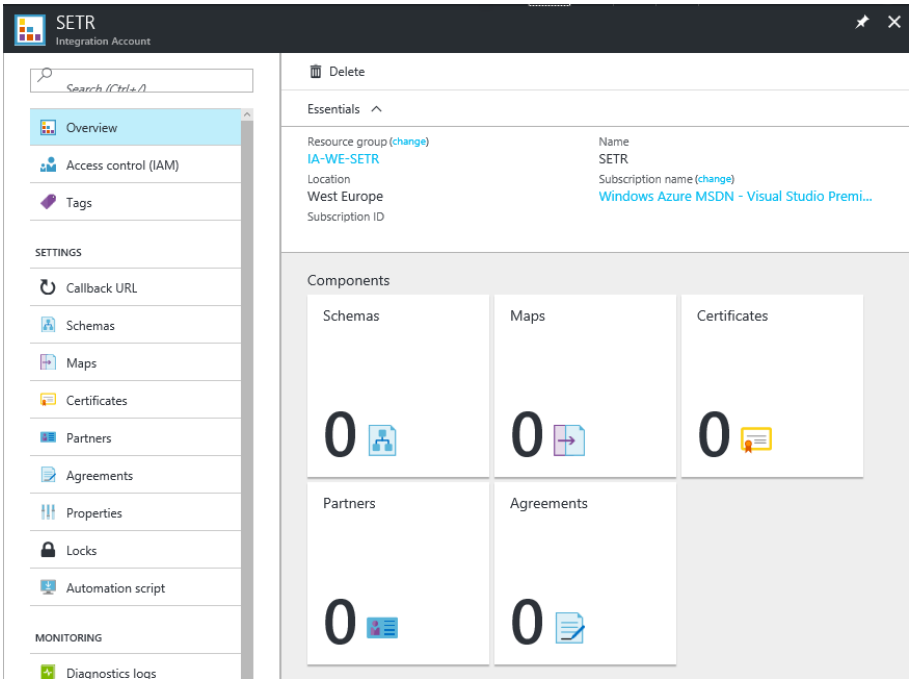
4. Fill out the form to create a new Integration Account

   Give it a logical ***name***, select your ***subscription***, a ***resource group*** (or create a new one) and choose a ***location near you***. Also select the ***free pricing tier***. This is enough for now. Click ***Create***. Deployment shouldn't take too long.

   Each region can only have 1 free Integration Account. If you already have a free pricing tier, deployment will fail. You can always use your existing one or choose another region.



5. Navigate to the Integration Account to verify the deployment



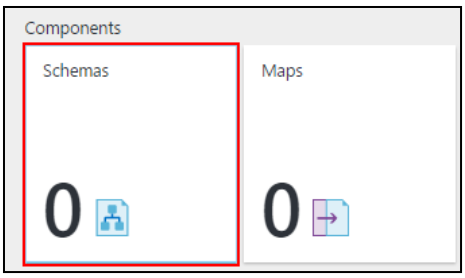
Upload schema to the Integration Account

Now that we have created the Integration Account, artifacts can be uploaded to it. After uploading the artifacts, they will become available in Logic Apps linked to the Integration Account.

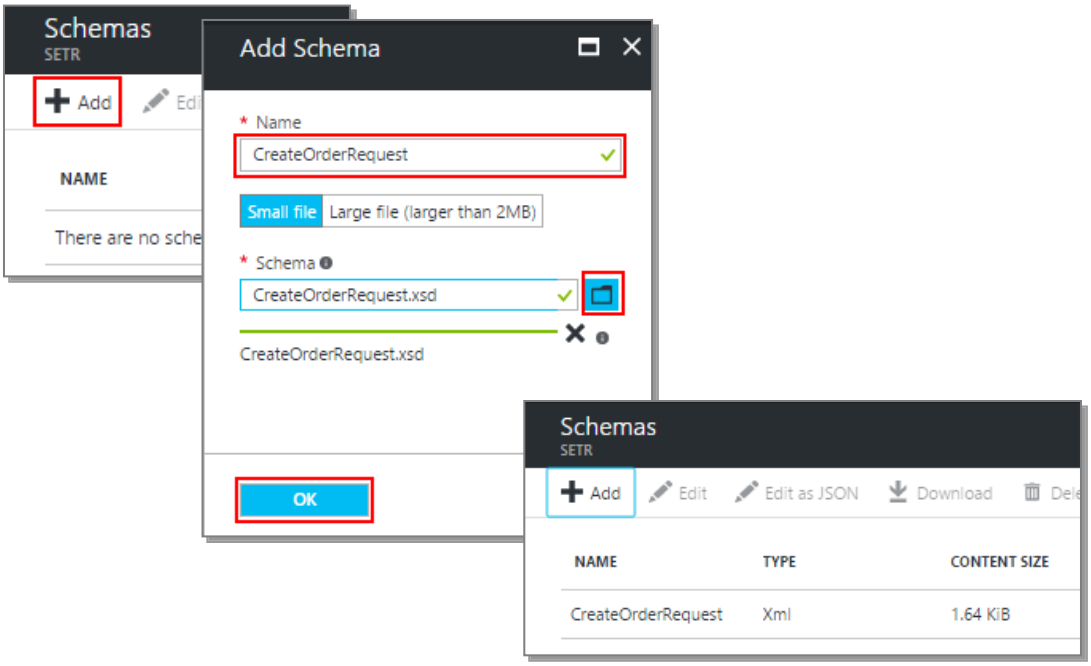
The only artifact we've created is the XSD that is going to be used for validation of the data received.

1. Navigate to Schemas

   With the SETR Integration Account selected, open ***Schemas***.



2. Click ***Add***, give your schema a unique ***name*** and ***choose the file just created*** to be uploaded. ***Click OK*** once done. After successful upload, your schema should be visible in the list of schemas.



Provision a Logic App

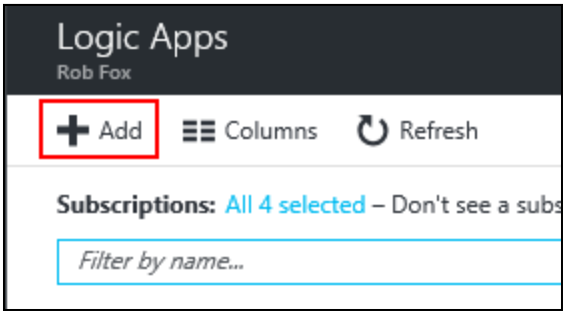
The artifacts in the Integration Account can be used in a logic app. Follow the instructions to create the logic app and associate it with the Integration Account.

1. Go to the Azure Portal

   https://portal.azure.com

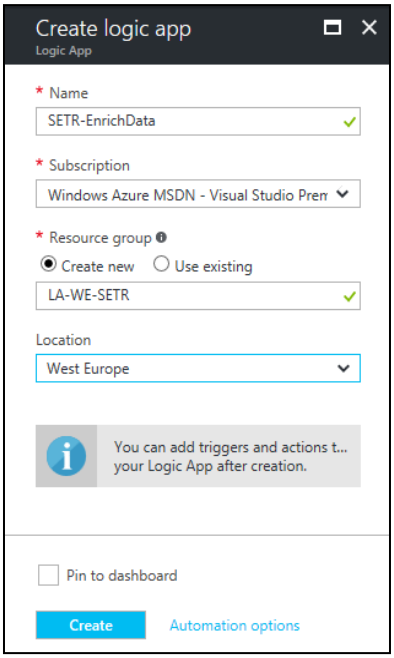
2. Navigate to the Logic Apps blade

3. Click Add to create a new Logic App



4. Create Logic App

   Give your logic app a ***name*** and assign it to a ***resource group***. Click ***Create***.

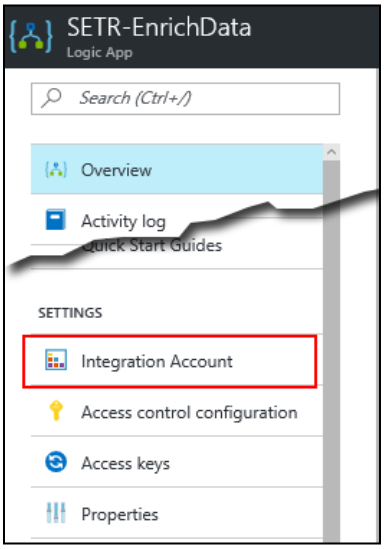


Associate the Integration Account with the Logic App

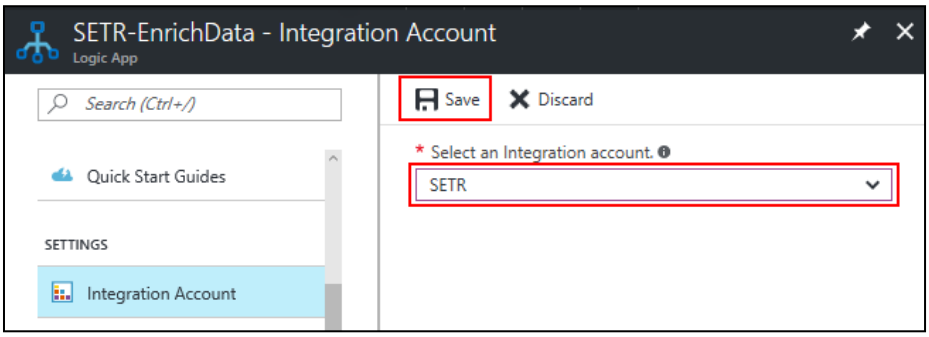
1. Navigate to the Logic App you've just created

   If you are in the Logic Apps blade and don't see the logic app, click ***refresh*** to refresh the list. If it still doesn't show, deployment might still be ongoing.

2. In the ***properties*** of your logic app, under settings, click ***Integration Account***.



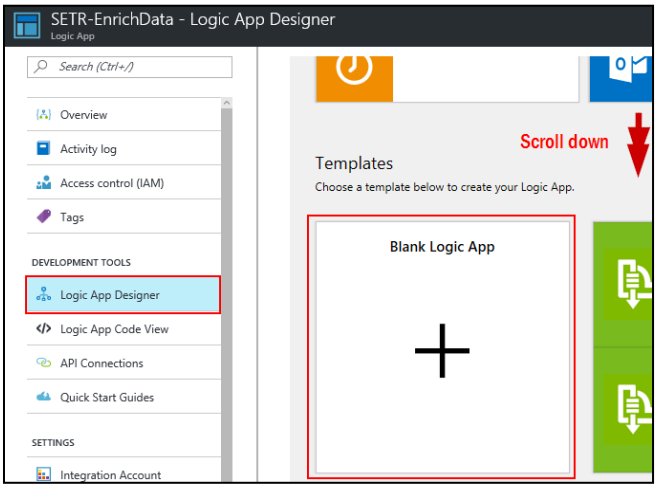
3. All your Integration Accounts should be visible in the dropdown box on the right. ***Select the correct Integration Account*** for this exercise and click ***Save***.



Build Logic App Definition

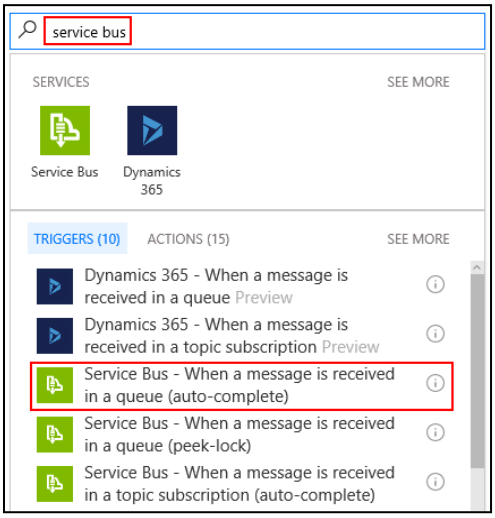
And now for the fun part: building the logic app definition.

1. In the properties, select ***Logic App Designer***. You will end up on a page with instructions and templates. **Scroll down**, until you find the Templates. Select ***Blank Logic App***. The designer will be opened.

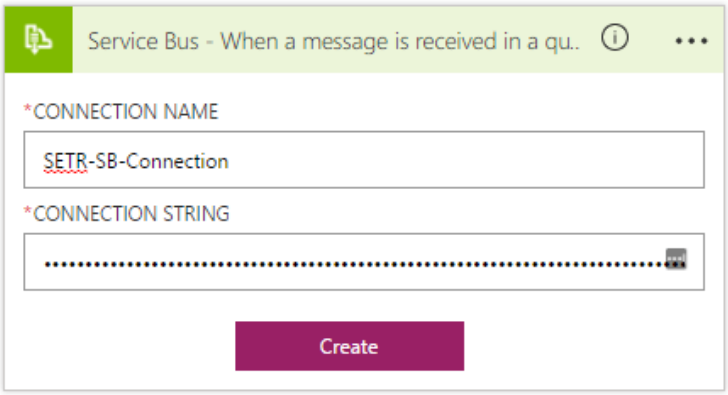


2. In the Logic App designer, you'll notice a search box. In this search box, triggers and actions can be searched and added to the definition. We need to add a trigger that polls for messages on a queue.

   Searching for ***service bus*** will filter the triggers and actions to those who are related to service bus. Now select ***Service Bus – When a message is received in a queue (auto-complete)***.



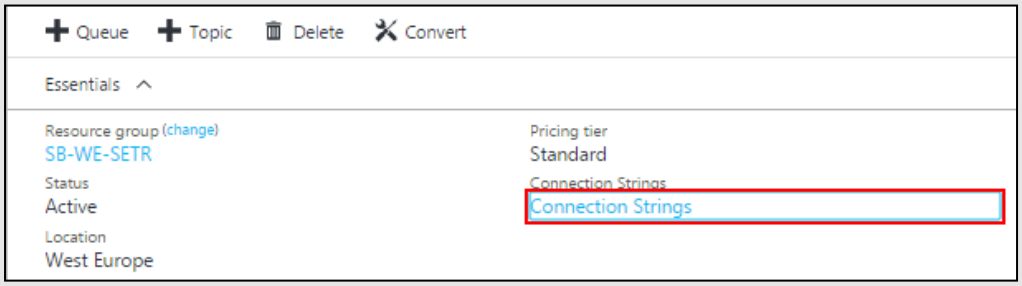
   Now fill out a ***connection name*** and the ***connection string***. This is the connection **used in the 1st lab**. Click ***Create***.



If you don't know your connection string, go through the following steps to copy it.

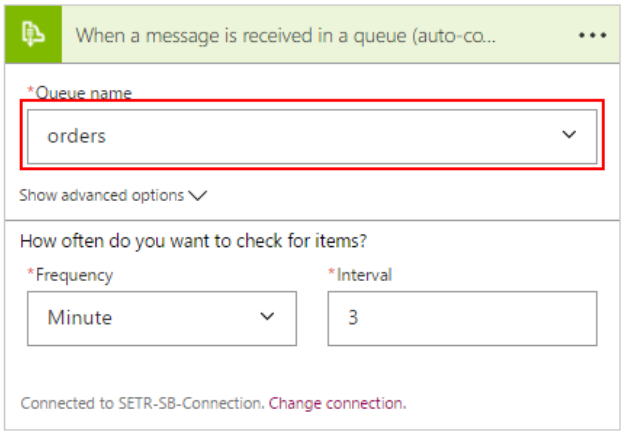
FINIDING THE SERVICE BUS CONNECTION STRING

a) Navigate to the **Service Bus blade**
b) **Open the service bus namespace** used for these labs
c) Click on **Connection Strings**



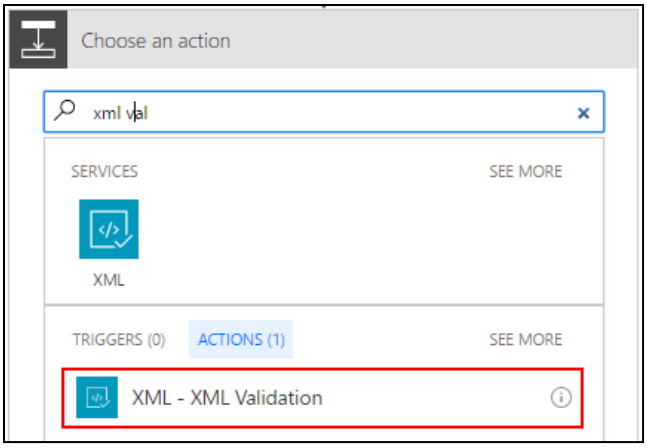
d) Select ***RootManageSharedAccessKey***
e) Copy the ***Connection String – Primary Key***

After setting up the connection, you will be able to ***select the queue*** you want to use. The same queue where we were sending the message to in the 1st lab will be used. Set an ***interval***. Be careful, because you are paying for each polling action!



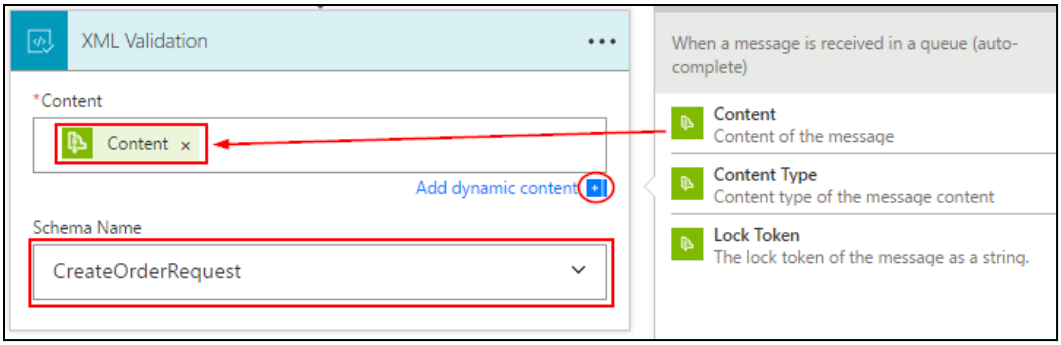
3. Now that the XML has been received, we will be able to validate it. For validation, the XSD we've just uploaded will be used.

   Add the action by searching for ***XML Validation***.



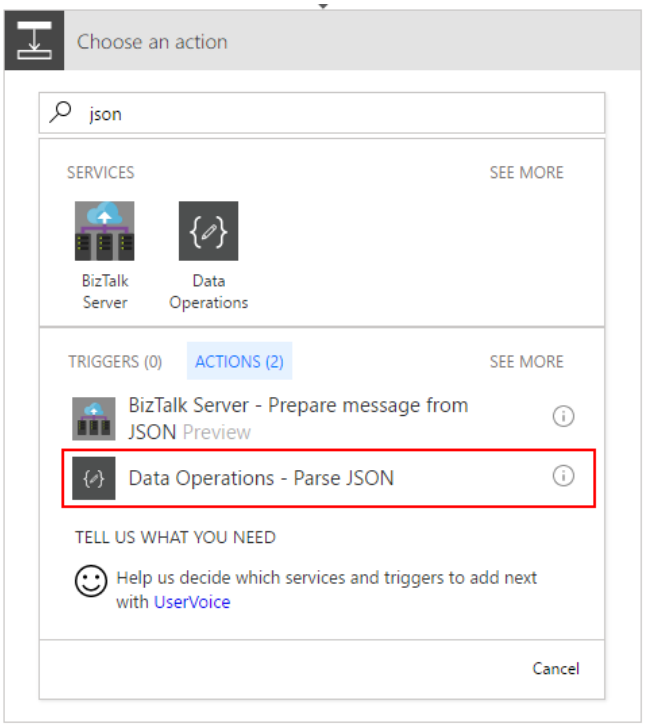
   The content just received from the queue will be validated.

   Click Content in the popup to add it to the Content field. Also ***select the XML schema*** that should be used. All schemas in the linked Integration account should be visible.

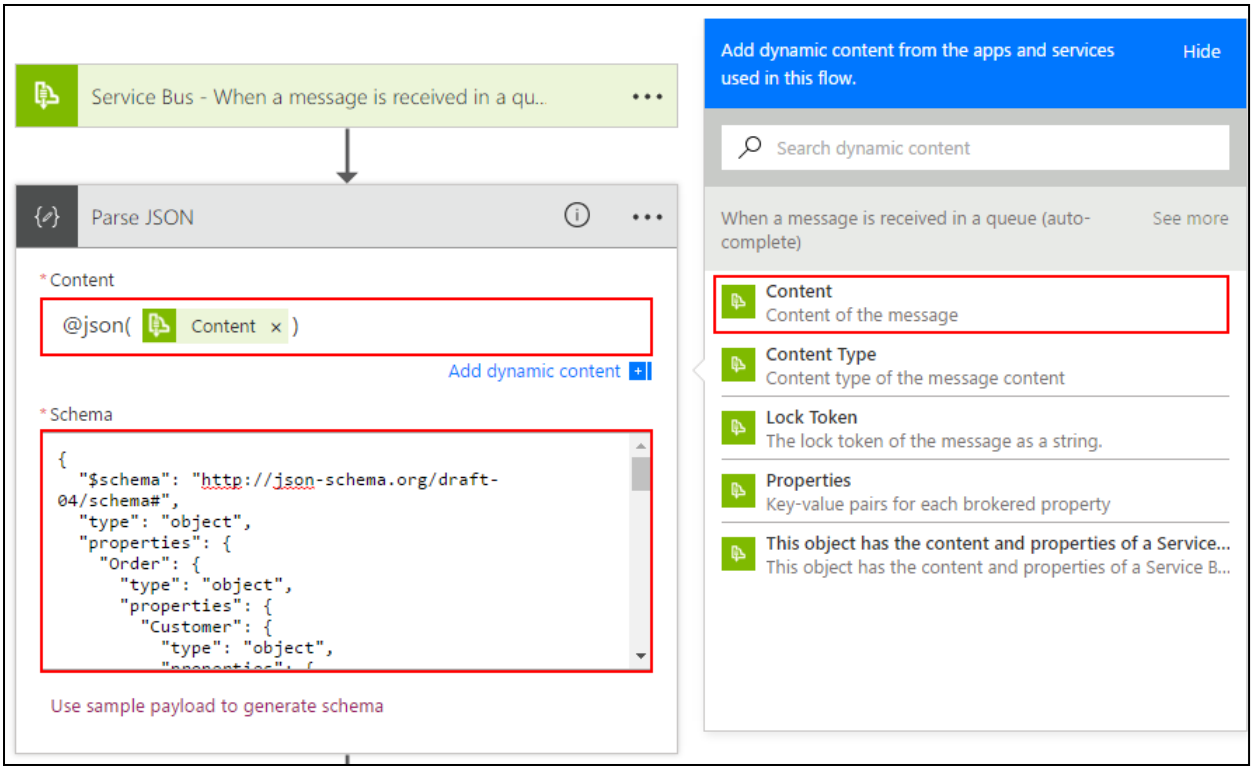


4. After receiving the XML message from the service bus queue and validating it, you'd want to parse it to JSON. Parsing it to JSON will create directly accessible properties in the logic app. Where in BizTalk everything was XML, in logic apps everything is JSON.

   Add a **Parse JSON** action to parse from XML to JSON. This action can be found under **Data Operations**.



   Converting the received XML message to JSON is a piece of cake in a logic app. Just use the **@json()** expression**.** Use the dynamic content, also called ***Content*** (that's the content of the message received from the service bus queue), as an input to the @json() expression. That should do the trick.



A JSON schema should also be added to tell the logic app what the JSON message looks like and what properties it has. You can generate a JSON schema by providing an example message at http://jsonschema.net. Or you can simply copy the schema below:

```
{
"$schema": "http://json-schema.org/draft-04/schema#",
"type": "object",
"properties": {
  "CreateOrder": {
    "type": "object",
    "properties": {
      "Order": {
        "type": "object",
        "properties": {
          "Customer": {
            "type": "object",
            "properties": {
              "CustomerNumber": {
                "type": "string"
              }
            },
            "required": [
              "CustomerNumber"
            ]
          },
          "Products": {
```

```
            "type": "object",
            "properties": {
              "Product": {
                "type": "array",
                "items": {
                  "type": "object",
                  "properties": {
                    "ProductNumber": {
                      "type": "string"
                    },
                    "Amount": {
                      "type": "string"
                    }
                  },
                  "required": [
                    "ProductNumber",
                    "Amount"
                  ]
                }
              }
            },
            "required": [
              "Product"
            ]
          },
          "OrderedDateTime": {
            "type": "string"
          }
        },
        "required": [
          "Customer",
          "Products",
          "OrderedDateTime"
        ]
      }
    },
    "required": [
      "Order"
    ]
  }
},
"required": [
  "CreateOrder"
]
}
```

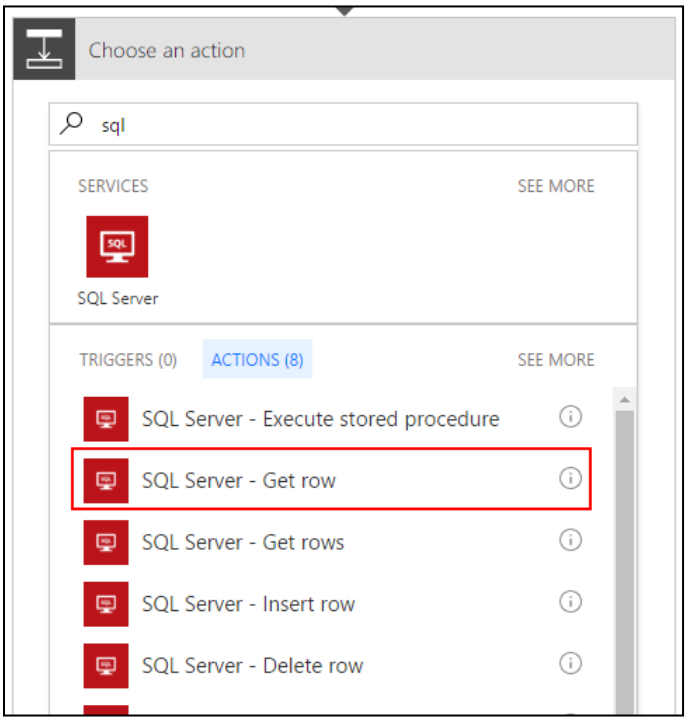
If there are any problems saving the logic app after entering the "@json(Content)" expression, delete it and use this:

```
@json(xml(base64ToString(triggerBody()?['ContentData'])))
```

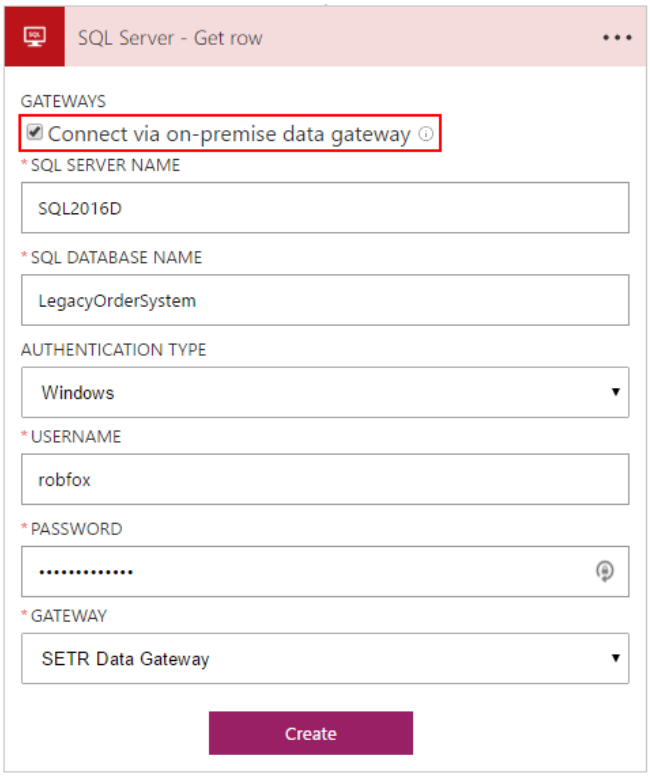
5. All the extra information we need is stored on the on-premises SQL Server. The logic app can reach it by using the On-Premises Data Gateway installed earlier.

   We will be using a simple **Get row** action to get the information we need in this scenario. This will fetch a row from a table using the primary key to select the data.

   Add the ***SQL Server – Get row*** action.

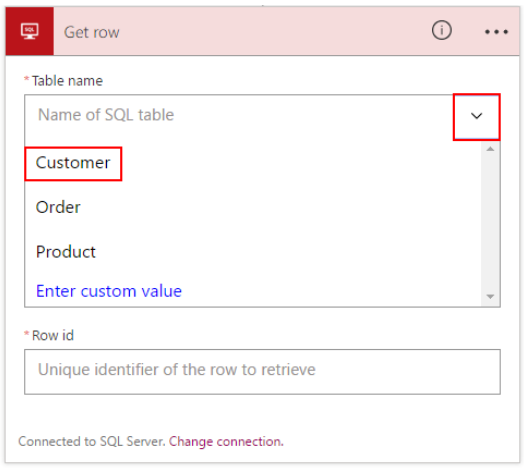


Tick the box to indicate the On-Premise Data Gateway is being used and ***fill out all the connection details***.

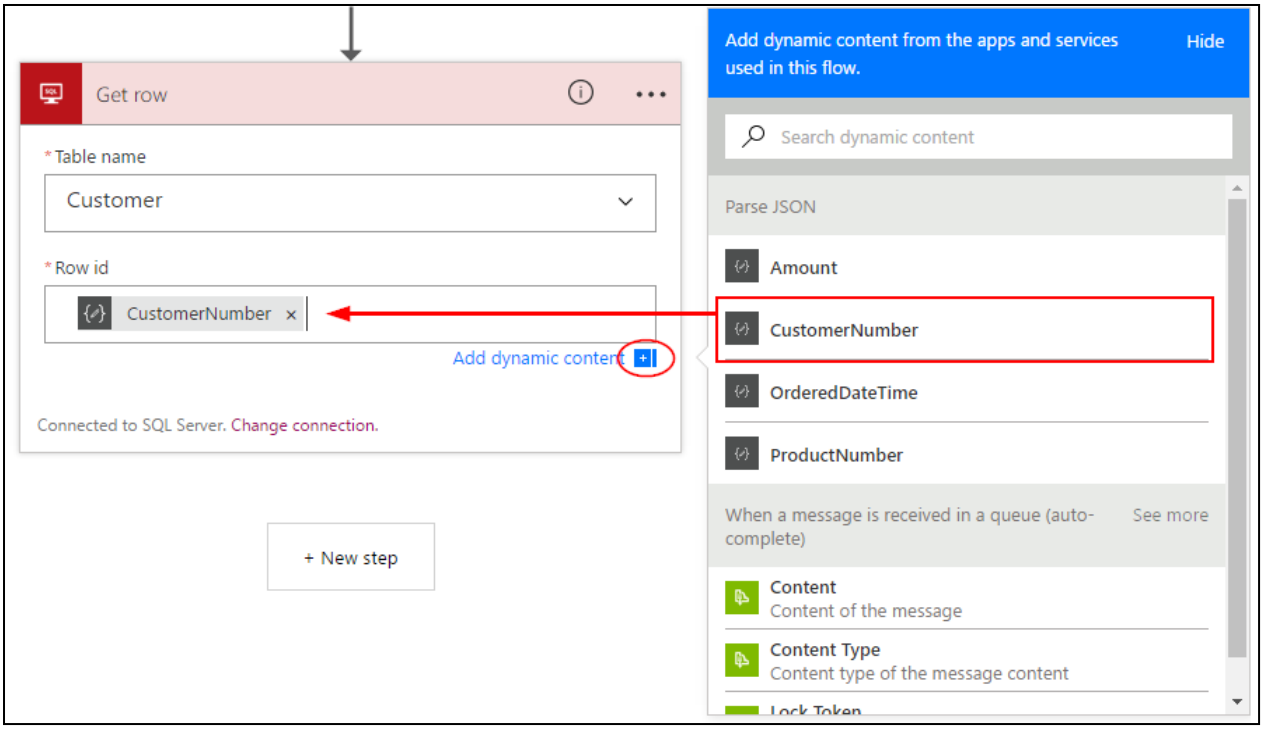


After filling out the connection details, you should be able to select the table in a list of tables that are present in the selected database.

Go ahead and ***select the Customer table***.



For the ***Row id*** the customer number in the request message should be used. From the dynamic content ***select the CustomerNumber***, which can be found between the "Parse JSON" fields.



6. The data from the request message and the data obtained from the on-premises SQL Server should now be combined into one message. The easiest way to do that, is using the **Compose JSON action**.

   Add the Compose JSON action to the logic app. The only input it requires is a JSON message. Below is an example wire-framed message that can be used to build this message.

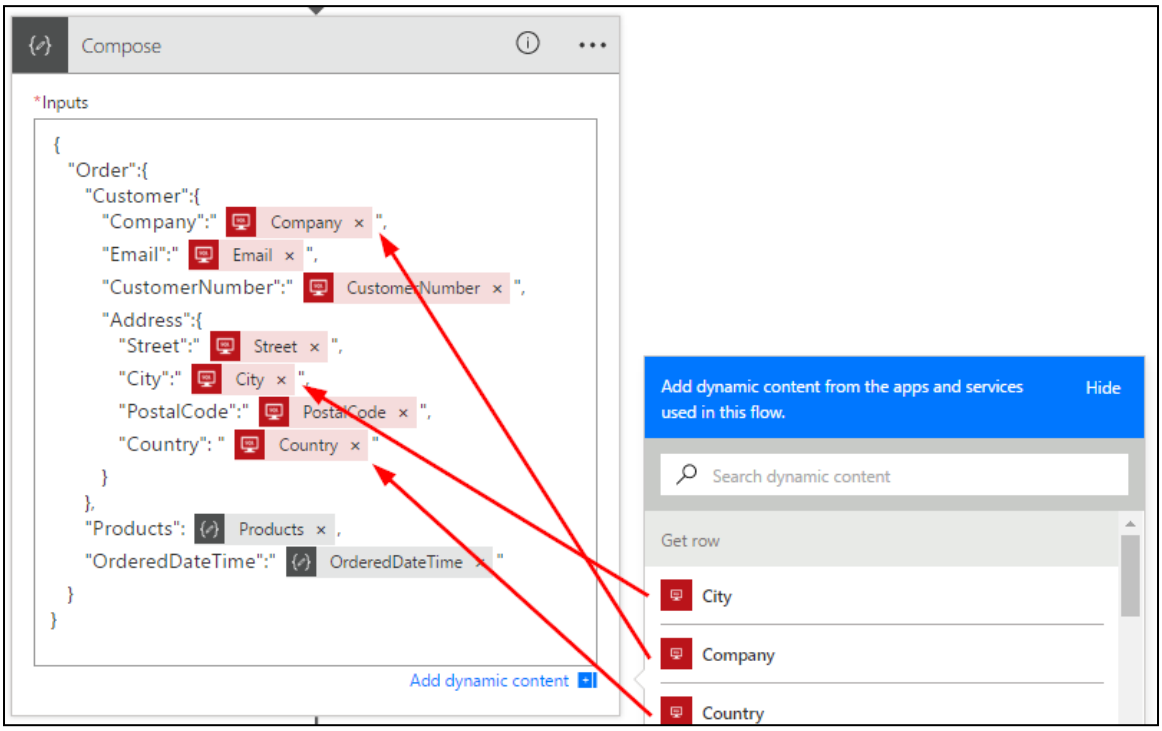
   Copy and paste the below code into the Compose shape:

```
{
   "Order":{
      "Customer":{
         "Company": "",
         "Email": "",
         "CustomerNumber": "",
         "Address":{
            "Street": "",
            "City": "",
            "PostalCode": "",
            "Country": ""
         }
      },
      "Products": ,
      "OrderedDateTime":""
   }
}
```

After copying it, there's still some values to be filled out. This means re-using values we got in the request message, plus the data received from the on-premises SQL Server.

Open the Add dynamic content dialog if it is not opened yet. **Select each value** you want and ***put them in the right place***. For the products array, just put the entire ***Products*** variable in there.

In the end, the message should resemble the following:



Test the Solution

We have not configured a place to send the message yet. The next lab will continue, where this lab left off. There's no need for it to be able to test the outcome, since Logic Apps come with extensive tracking capabilities.

1. Since the request message is coming from a queue, we are only able to test it, by putting a message on the queue ourselves.

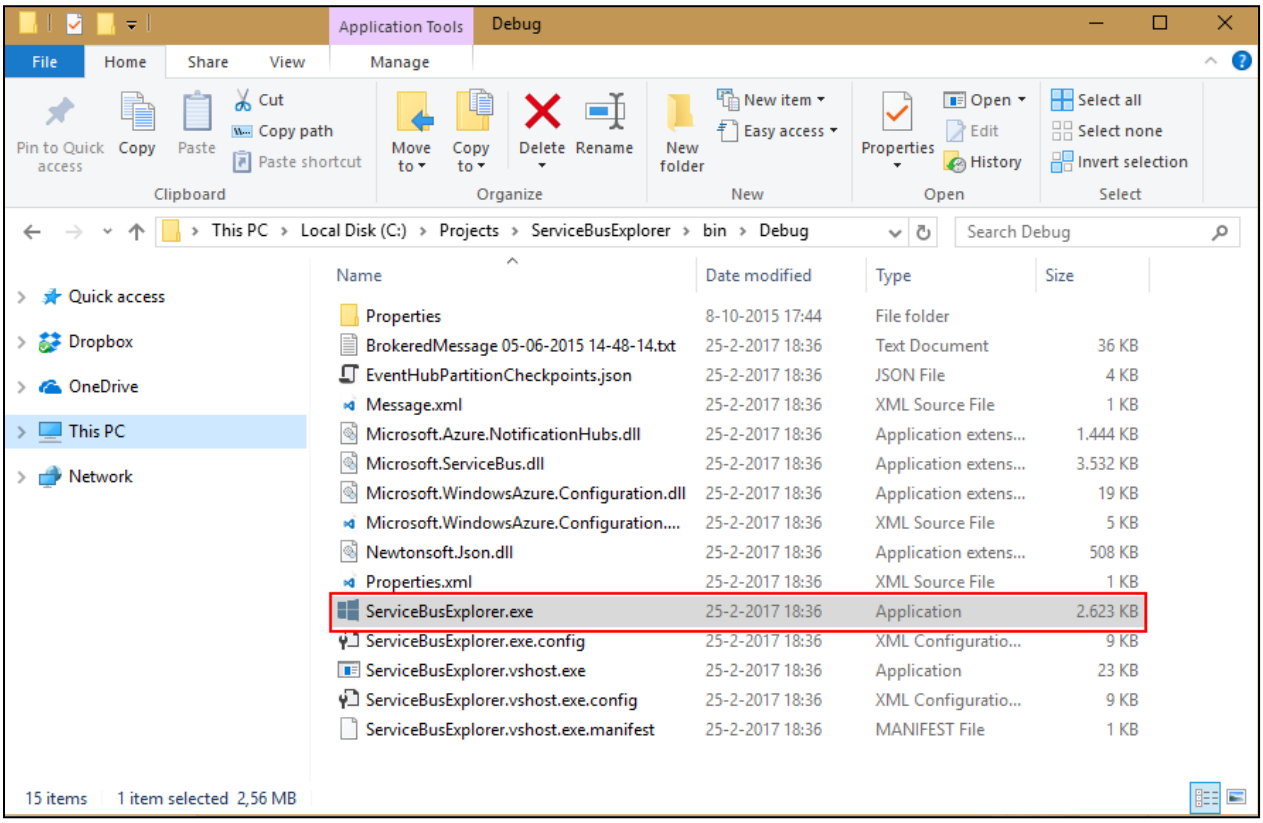
   If you have not already Installed Service Bus Explorer, please do so by downloading it and follow **Step I**. Otherwise continue to **Step II**.

   I. Download Service Bus Explorer 2.6

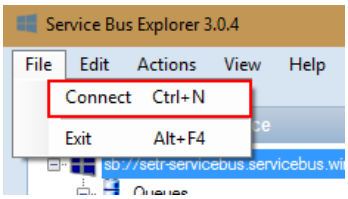
   Unpack the downloaded zip to an appropriate place (e.g. c:\projects).

You can download it over here: https://code.msdn.microsoft.com/windowsapps/Service-Bus-Explorer-f2abca5a.

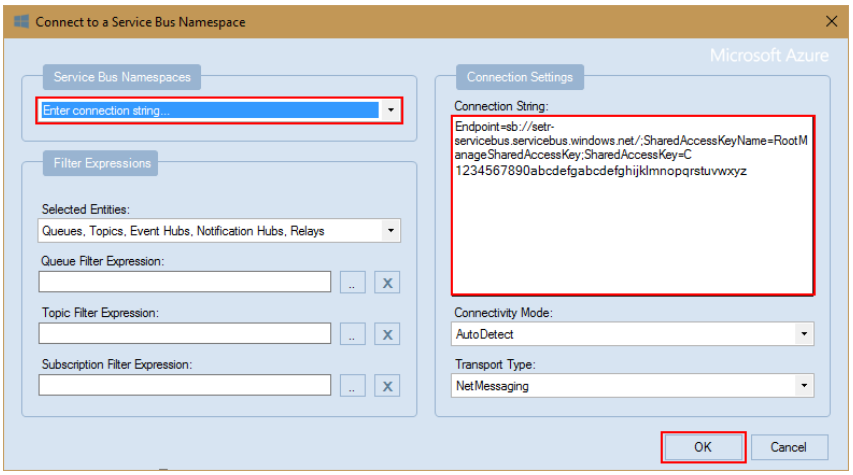
After unpacking, ***navigate to the bin/debug folder*** and start ***ServiceBusExplorer.exe***.

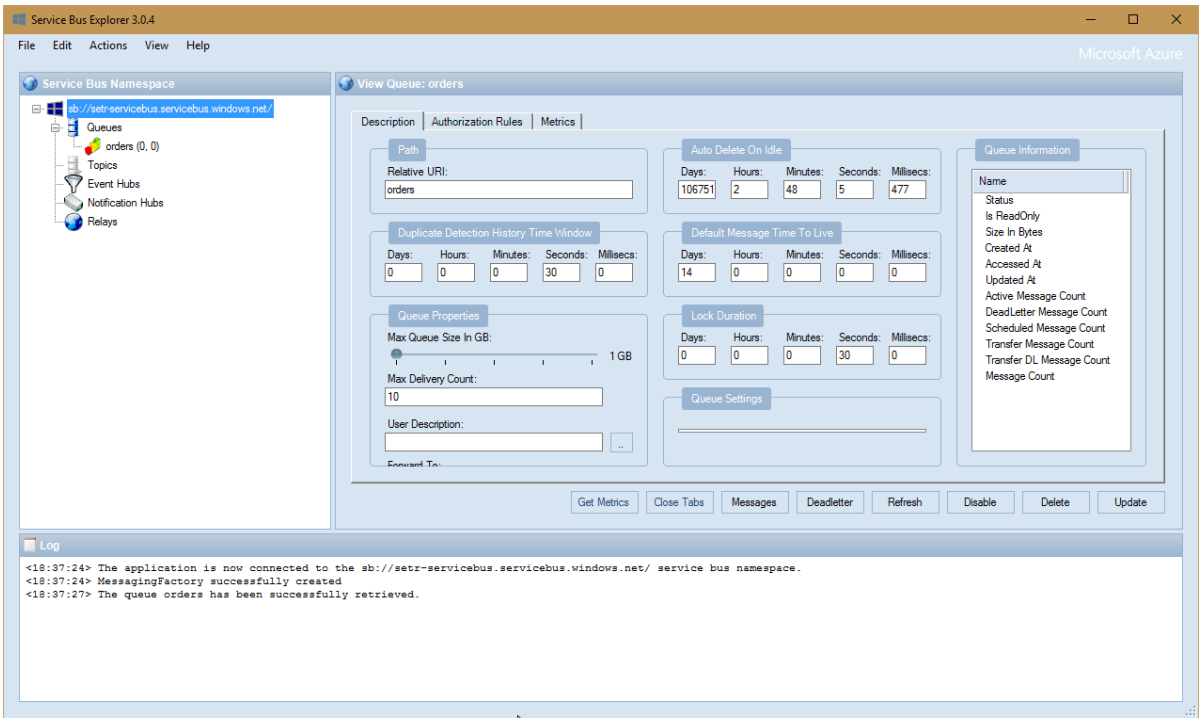


Setup a connection by choosing ***File > Connect***.



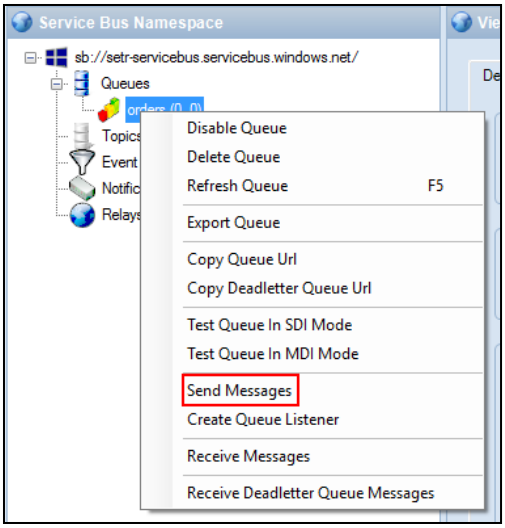
Select ***Enter connection string…*** Fill out the ***connection string*** to your service bus namespace and ***click OK***.



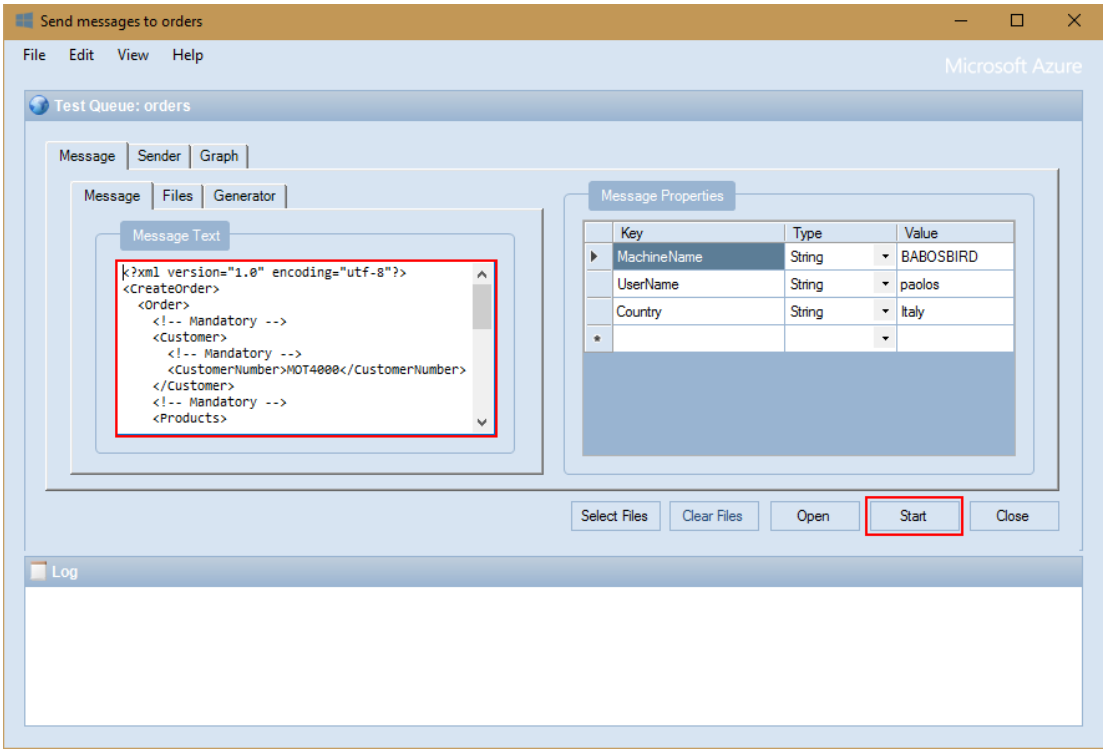


II. Send the message to the queue

Open Service Bus Explorer and ***right click the orders queue***. Choose ***Send Messages***.



In the dialog window that appears, the messages you want to send can be copied in the ***Message Text*** field. We do not care about the ***Message Properties***, so please leave them be.



To thoroughly test the validation you can also put a wrong message on the queue and see the results of that.

Go ahead and send both a correct and a incorrect message on the queue.

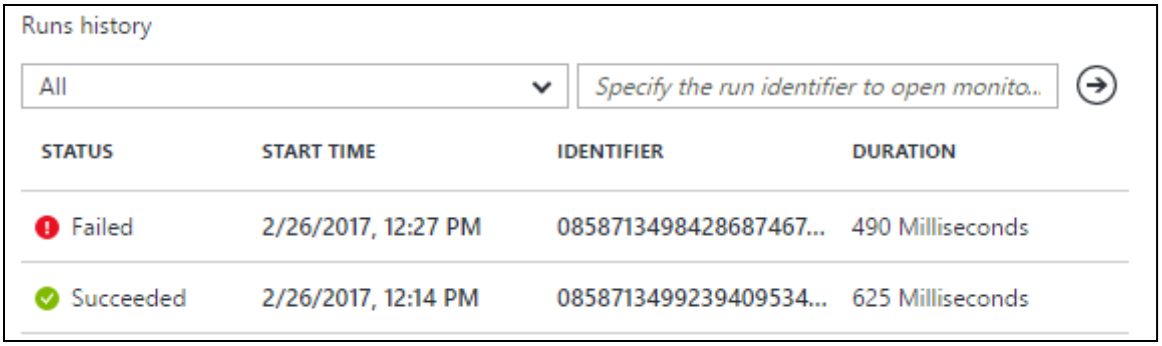
Correct Message:

```
<CreateOrder>
    <Order>
    <Customer>
        <CustomerNumber>C0001</CustomerNumber>
    </Customer>
    <Products>
        <Product>
            <ProductNumber>1000</ProductNumber>
            <Amount>1</Amount>
        </Product>
        <Product>
            <ProductNumber>2000</ProductNumber>
            <Amount>5</Amount>
        </Product>
    </Products>
    <OrderedDateTime>2016-11-20T14:26:00</OrderedDateTime>
    </Order>
</CreateOrder>
```

Incorrect Message (date has wrong format):

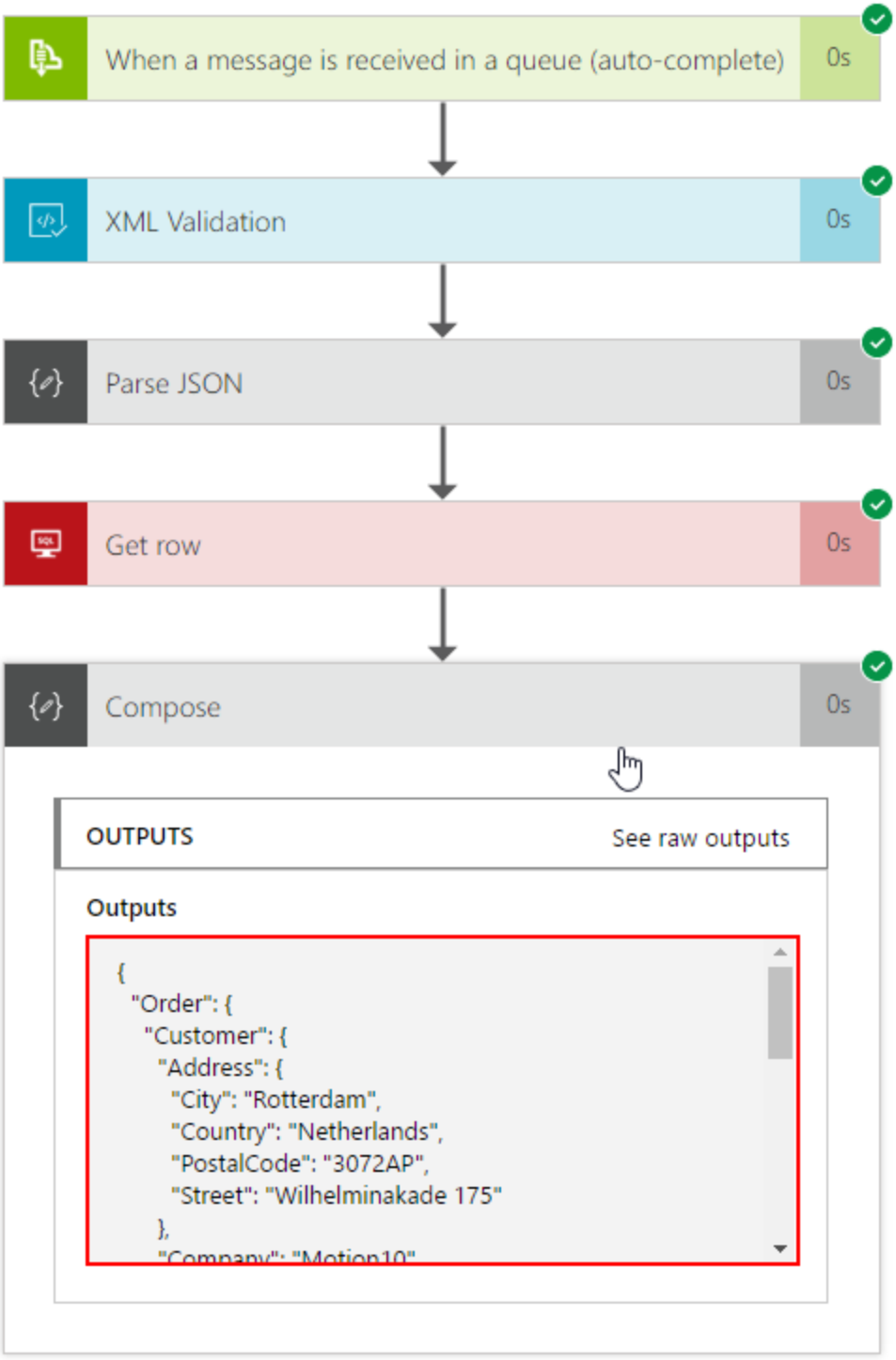
```
<CreateOrder>
    <Order>
    <Customer>
        <CustomerNumber>C0001</CustomerNumber>
    </Customer>
    <Products>
        <Product>
            <ProductNumber>1000</ProductNumber>
            <Amount>1</Amount>
        </Product>
        <Product>
            <ProductNumber>2000</ProductNumber>
            <Amount>5</Amount>
        </Product>
    </Products>
    <OrderedDateTime>20-11-2016T14:26:00</OrderedDateTime>
</CreateOrder>
```

Now check you runs in the overview blade. Check both the ***succeeded run*** and the ***failed run*** by selecting them.



Succeeded run

A correct run will show only green checkmarks. Like below. Check the output in the Compose shape to know for sure whether you've done the job correctly or not.



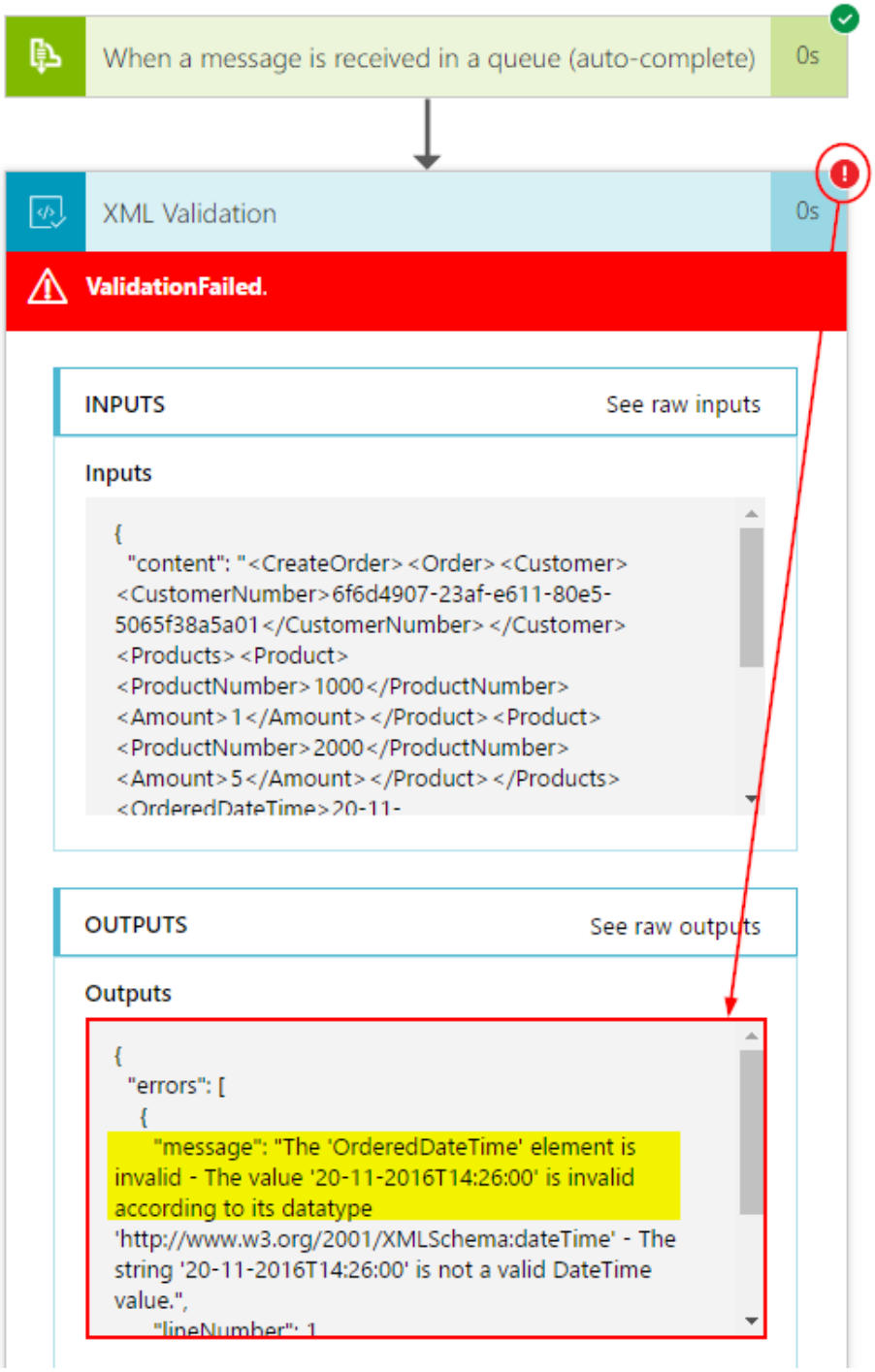
The correct output should resemble the following message. Notice that the order of JSON elements is not important, as that is different from XML.

```
{
```

```
  "Order": {
    "Customer": {
      "Address": {
        "City": "Rotterdam",
        "Country": "Netherlands",
        "PostalCode": "3072AP",
        "Street": "Wilhelminakade 175"
      },
      "Company": "Motion10",
      "CustomerNumber": "6f6d4907-23af-e611-80e5-5065f38a5a01",
      "Email": "rob.fox@motion10.com"
    },
    "OrderedDateTime": "2016-11-20T14:26:00",
    "Products": {
      "Product": [
        {
          "ProductNumber": "1000",
          "Amount": "1"
        },
        {
          "ProductNumber": "2000",
          "Amount": "5"
        }
      ]
    }
  }
}
```


Failed run

When a run has failed, you'll notice an exclamation mark. Open the action to get to know what exactly went wrong. As you can see, the datetime validation failed in this instance.



Lab 3 - Logic Apps + BizTalk 2016

Objective

In this third lab, we will be creating another Logic App, which will be called from the Logic App in our previous lab via a HTTP request. This logic app will be calling SETR's on premise BizTalk by using the Logic App adapter for BizTalk 2016. In BizTalk, the order will be stored in the database, and an order with the customer type will be generated. Once the order is generated, BizTalk will place it in a Service Bus topic, with a parameter indicating if this is an order for a business or a consumer customer. The topic will have 2 subscriptions for these types of customers.

Prerequisites

- Azure Subscription
- Service Bus Explorer
- Google Chrome Postman
- BizTalk 2016 installed and configured including IIS and WCF-SQL adapter
- SQL Server 2016 installed
- On premises data gateway installed (done in lab 2)
- LegacyOrderSystem database imported, which can be downloaded from here, and should be restored in SQL Server (done in lab 2)
- Grant execute permission to BizTalk Application Users on stored procedures in LegacyOrderSystem database

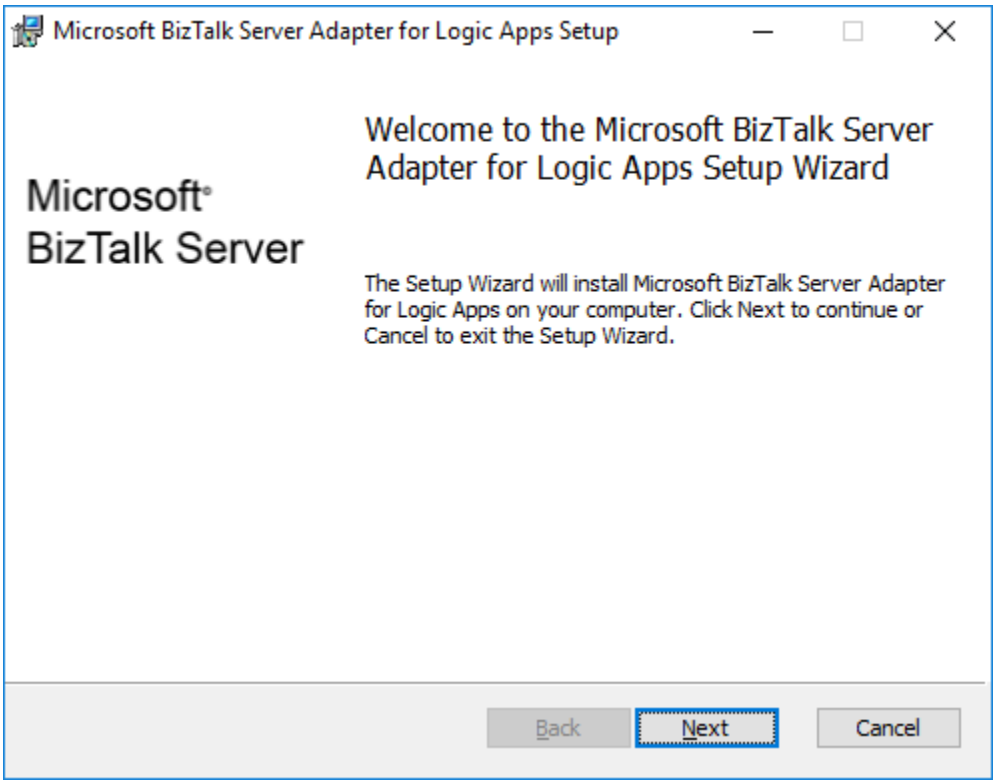
Steps

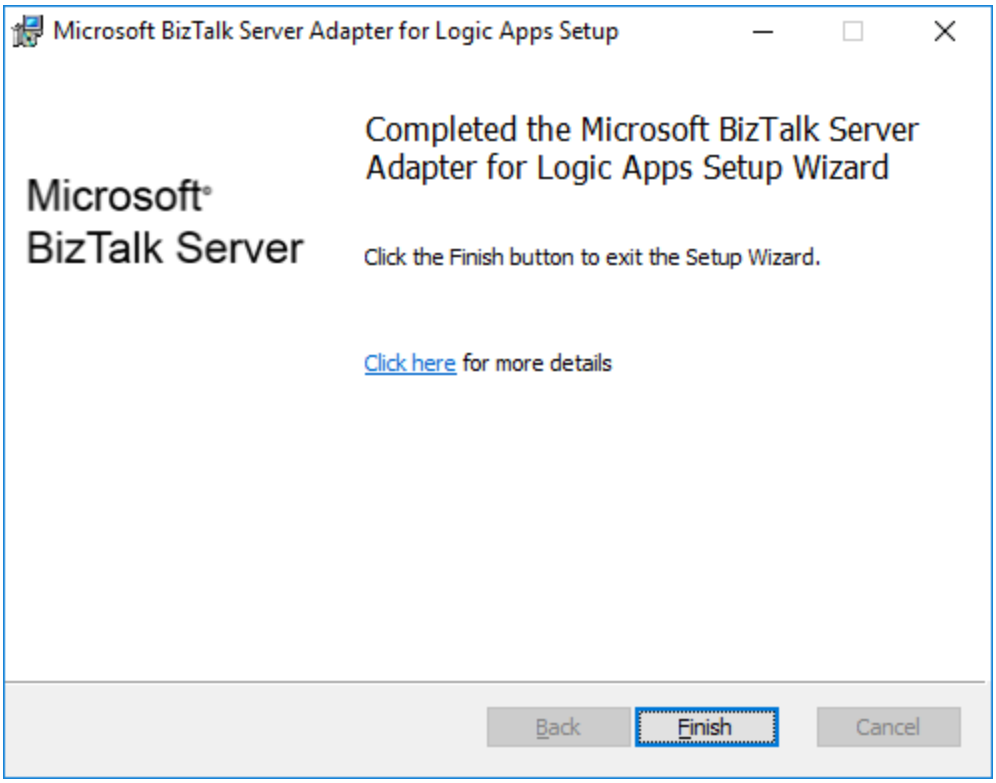
To build the solution in this lab you have to follow the steps described in this section. From a high level view the steps are:

1. Install and configure BizTalk LogicAppAdapter
2. Create Service Bus namespace
3. Install and configure BizTalk adapter
4. Create Azure Logic App

Install Logic Apps adapter

We will start by installing the new Logic Apps adapter. Download the adapter and start the installation. Follow the wizard, and install with default settings.

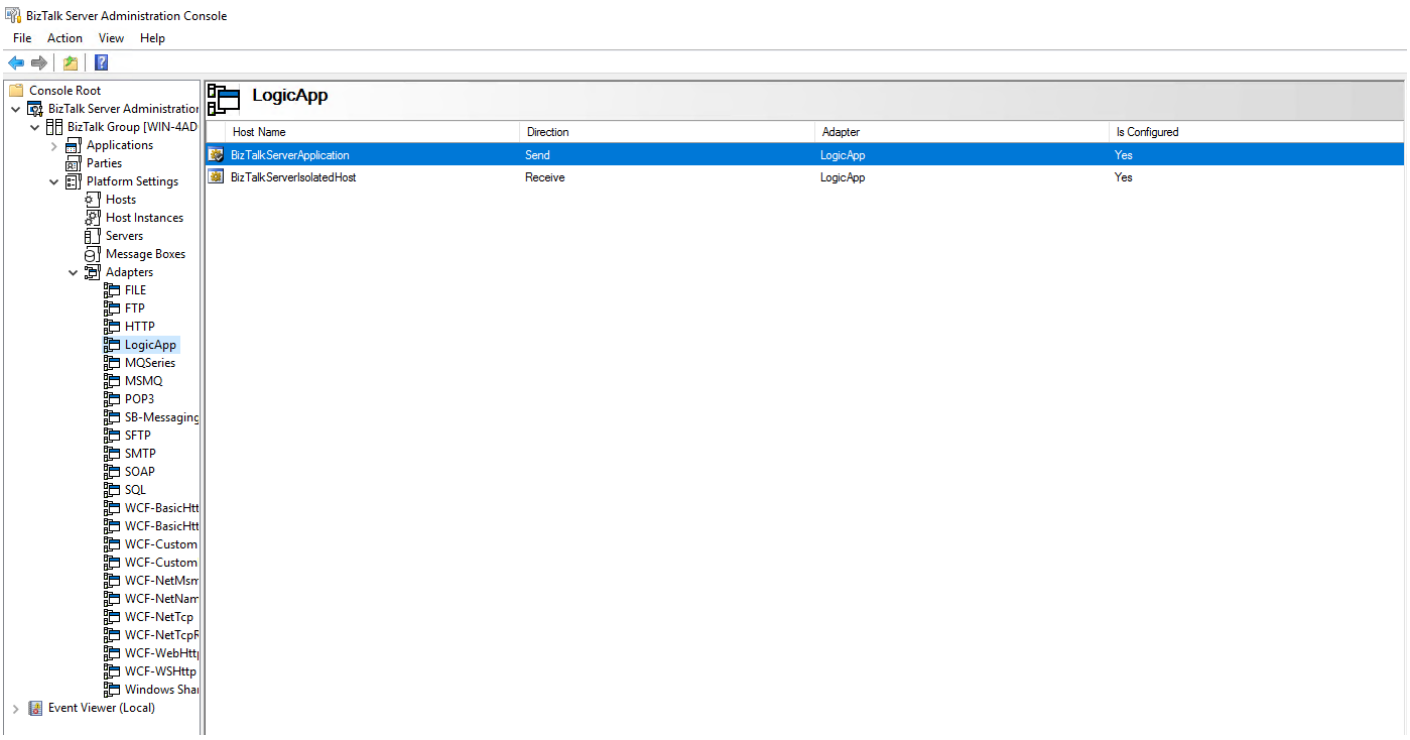




Once installed, the following will have been done automatically.

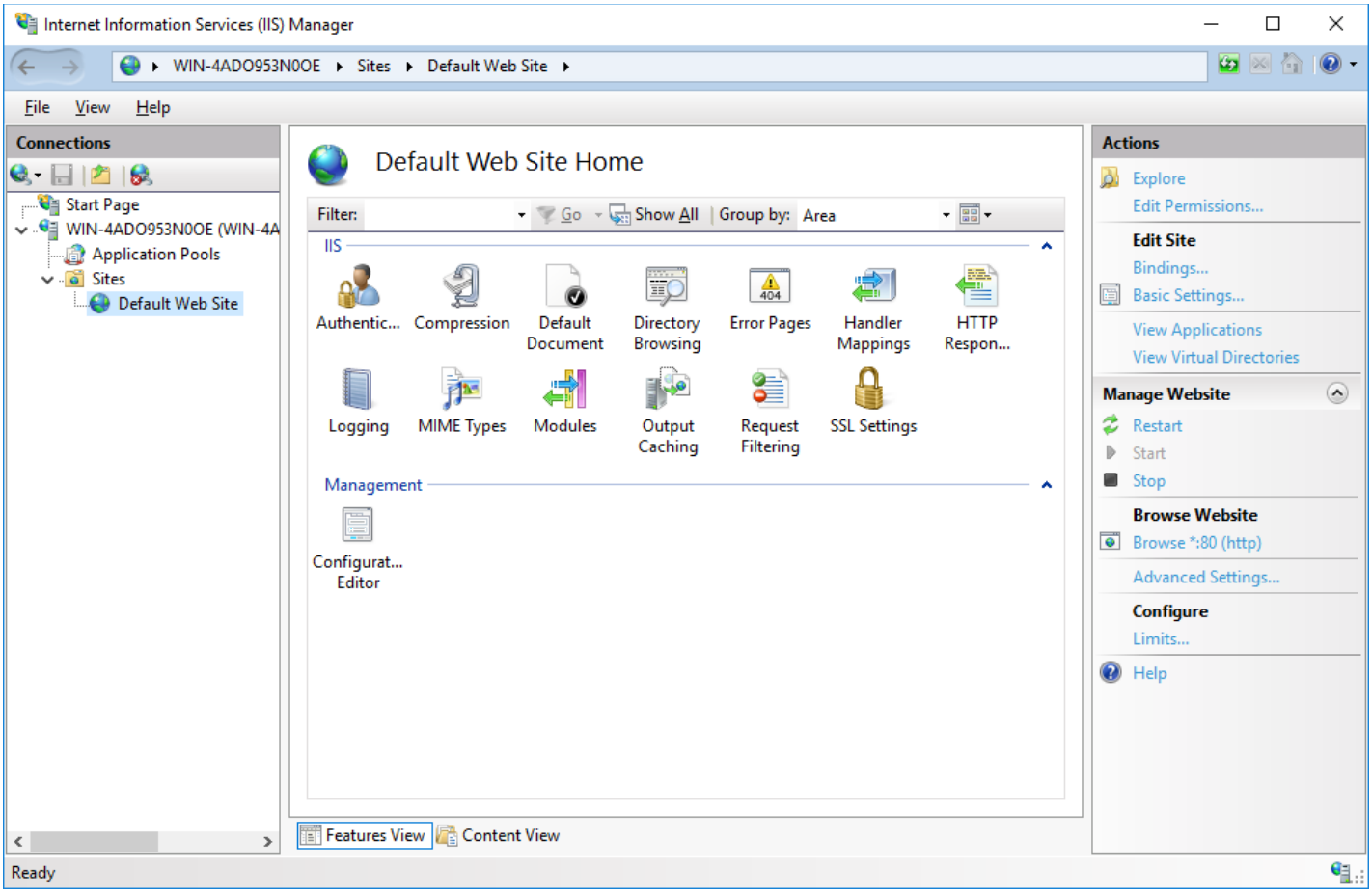
- The LogicApp adapter has been added to BizTalk
- The send handler has been created, and uses the default host (most probably BizTalkServerApplication)
- The receive handler has been created as a WCF service, and uses the BizTalkServerIsolatedHost host
- The Program Files (x86)\Microsoft BizTalk Server 2016\LogicApp Adapter folder has been created, and includes two services: Management and ReceiveService

The Management service is used by the BizTalk Connector in a logic app to connect to BizTalk Server using the data gateway, to retrieve the ports and message types exposed by BizTalk. The ReceiveService is used by the BizTalk Connector in a logic app when you enter the receive location. This service is responsible for receiving the messages from the logic app in BizTalk. Open the BizTalk Administration Console to check if the LogicApp adapter is configured correctly.

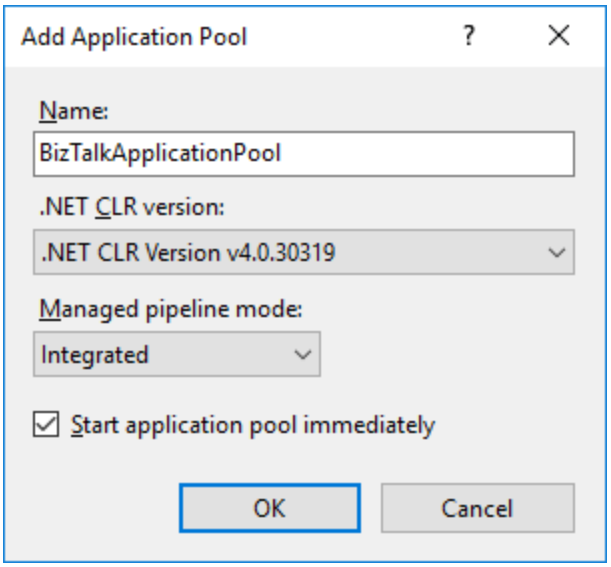


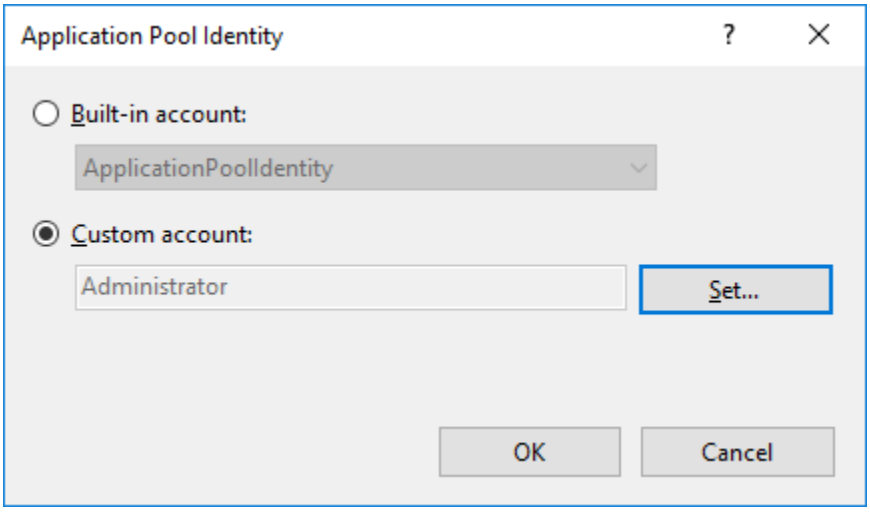
Created IIS applications for Logic App adapter

Now that the Logic App adapter has been installed, we need to configure the IIS applications which were installed. For both these applications, we need to create a WCF application in IIS. Start by opening the Internet Information Services Manager.



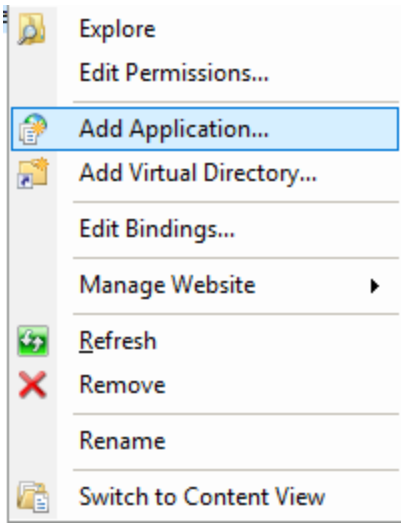
Create an application pool with same rights as BizTalk service users, which will host the IIS applications.

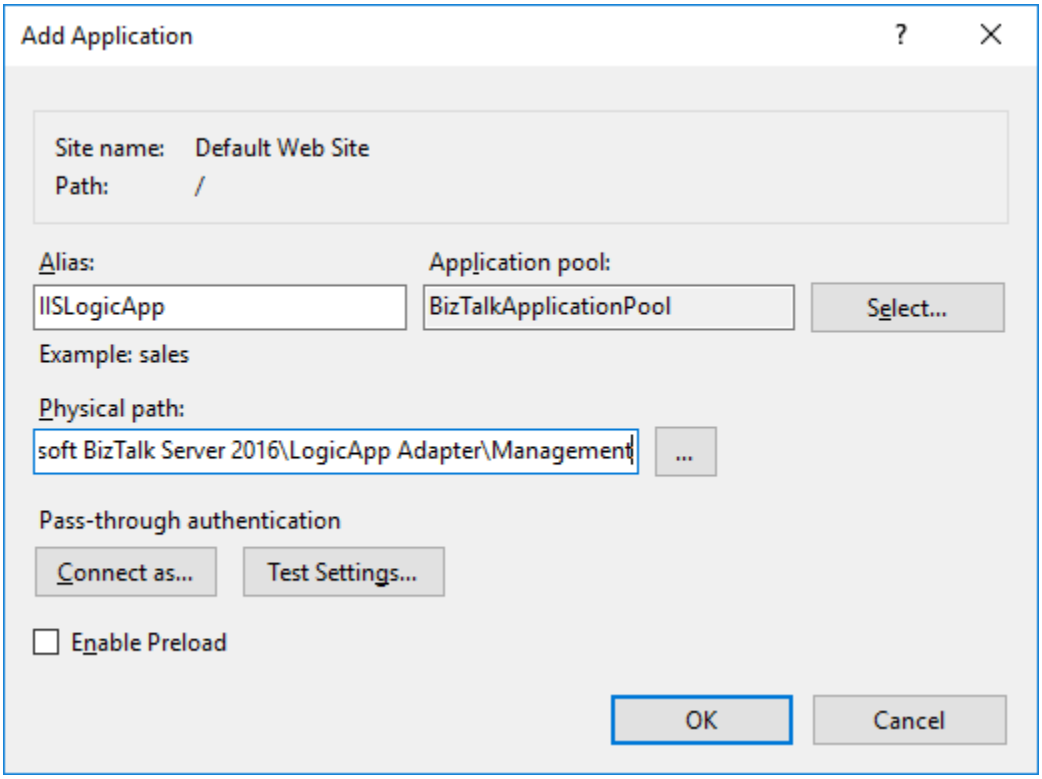




Add Management application

Now back in the IIS Administration console, add an application, and set the physical path set to C:\Program Files (x86)\Microsoft BizTalk Server 2016\LogicApp Adapter\Management. Make sure to select the application pool we created earlier.

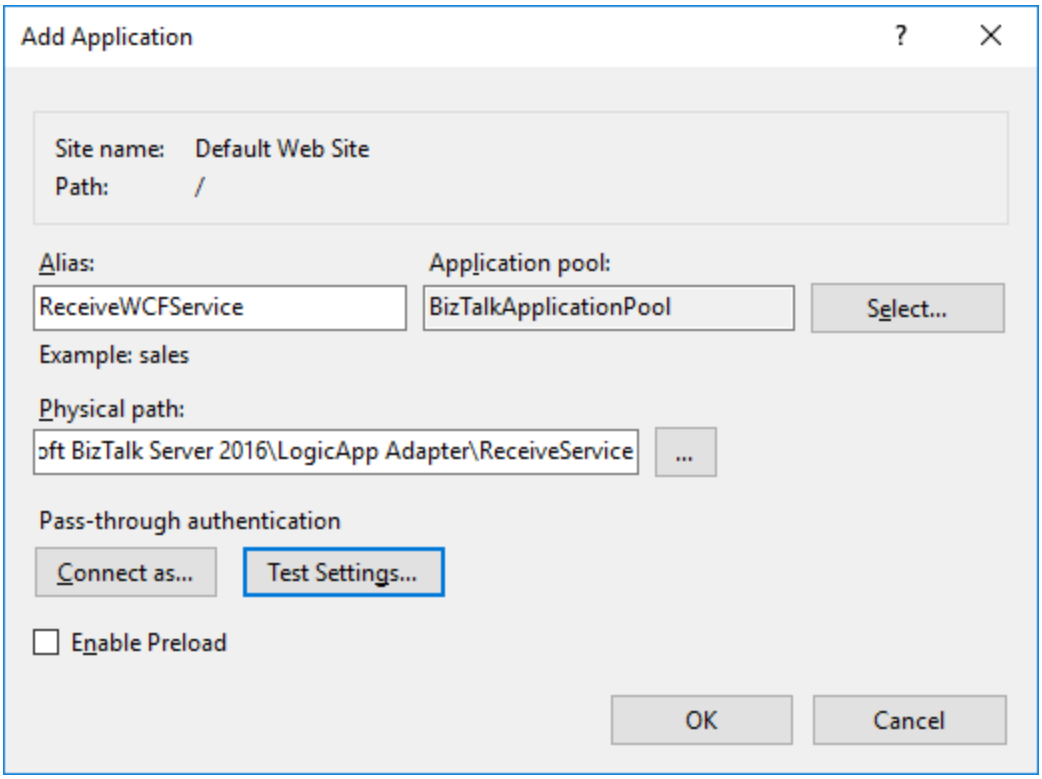




Test application by going to http://localhost/IISLogicApp/Schemas?api-version=2016-10-26, which should download a JSON file with the ports and message types from BizTalk.

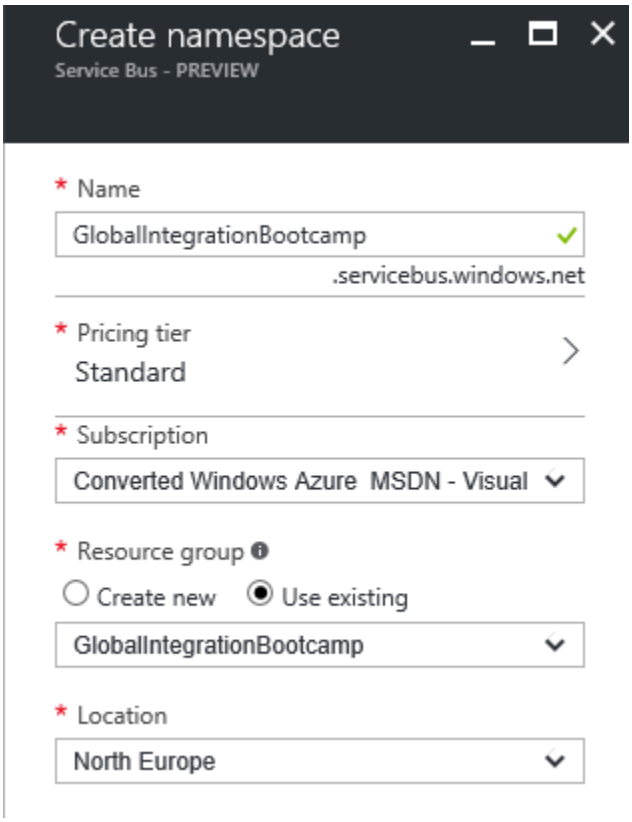
Add BizTalk ReceiveService application

Once again in the IIS Administration console, create another application, this time with the physical path set to C:\Program Files (x86)\Microsoft BizTalk Server 2016\LogicApp Adapter\ReceiveService. Here we also need to make sure to select the application pool we created earlier.

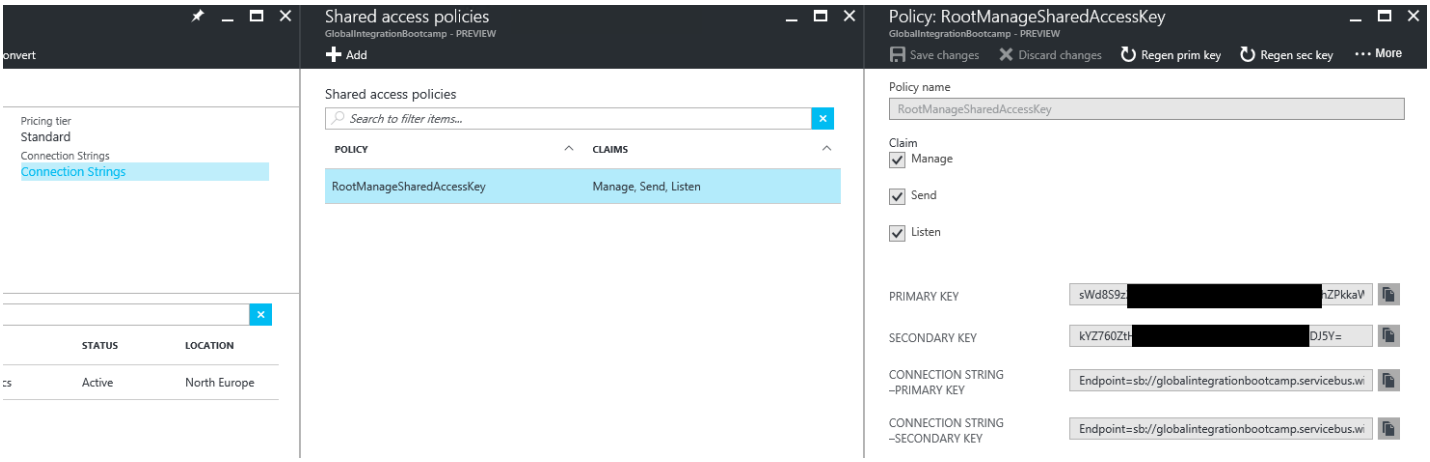


Create Service Bus Namespace

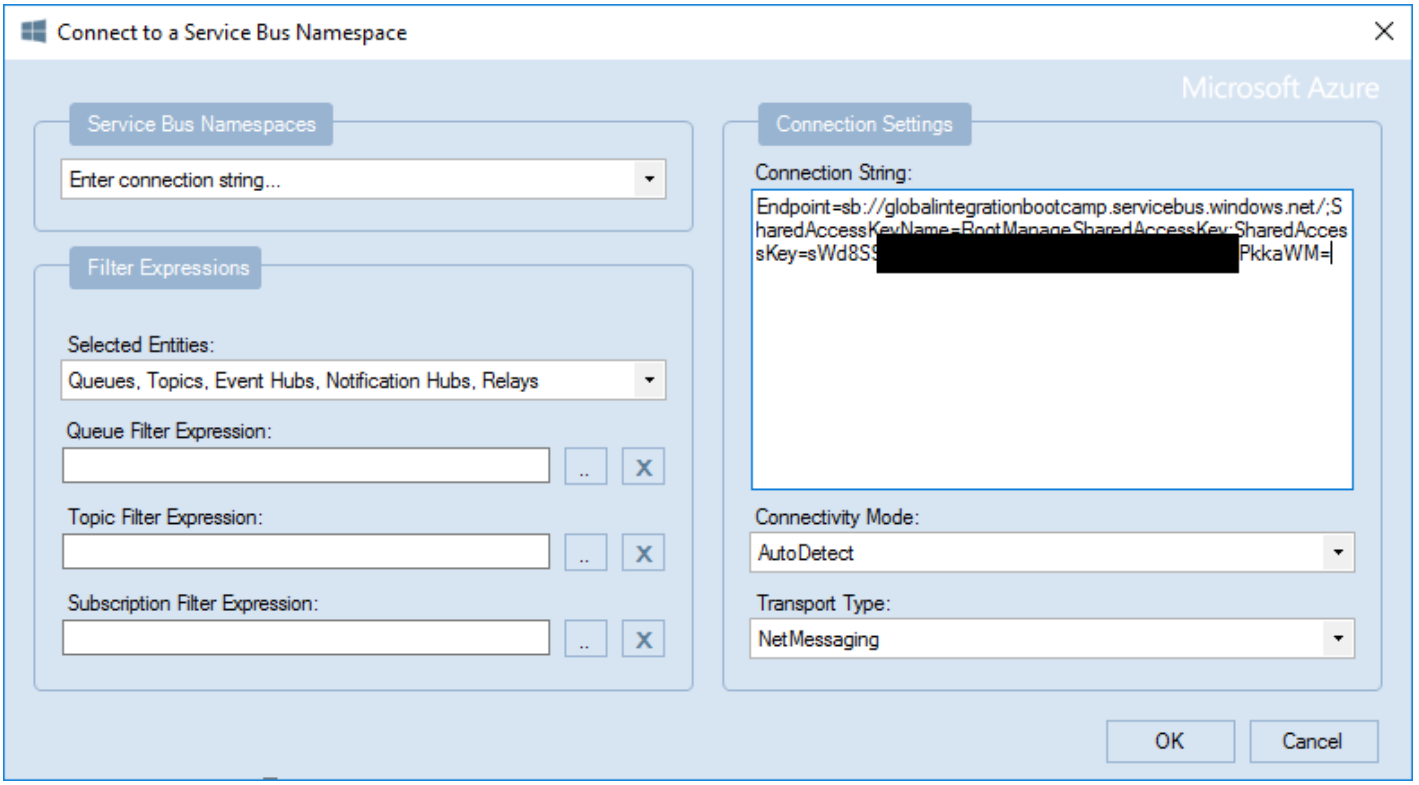
As we will be sending our outgoing messages to a Service Bus topic, we will also need to create this in Azure. You can either use the Service Bus namespace which was created in lab 1 (do make sure it is in the **Standard** tier), or go to the Service Bus blade in the portal, and add a new namespace (this should be a unique namespace). This namespace should be created on the **Standard** tier, as this is the tier from where topics are included as well.



Once the namespace has been created, we will need to add the topic and its subscriptions. For this we will use Service Bus Explorer, a great tool when working with Azure Service Bus. Start by retrieving the connection string for the **RootManageSharedAccessKey**, which we will use to manage our namespace from Service Bus Explorer.

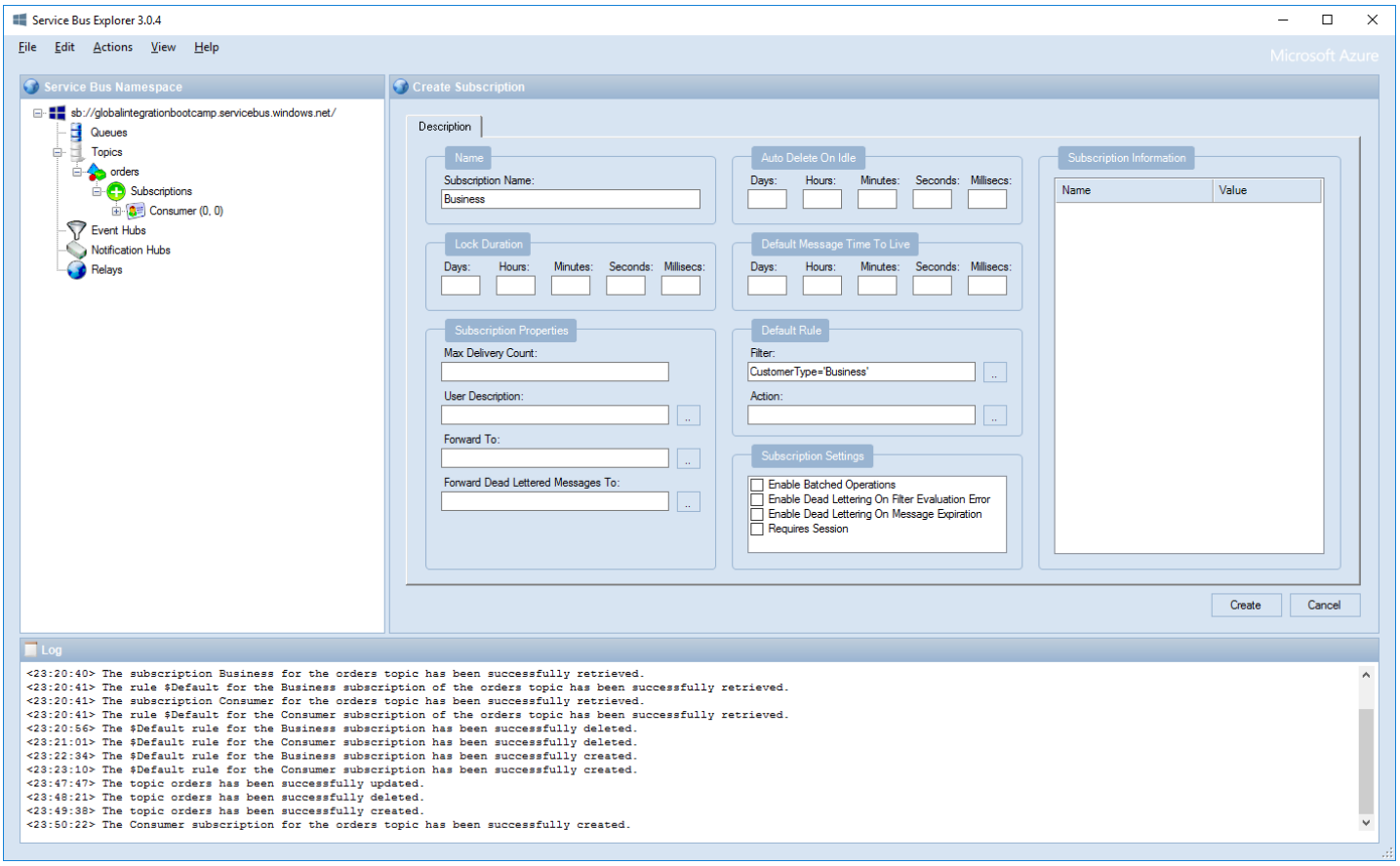


Now start Service Bus Explorer, and connect using the connection string we just retrieved.



Next add a topic called **orders**, and keep all the default settings. Once the topic has been created, we will add the subscriptions for the customer types. These subscriptions will be named **Business** and **Consumer**. When creating the subscription, make sure to set the filter as well.

- Consumer: **CustomerType = 'Consumer'**
- Business: **CustomerType = 'Business'**

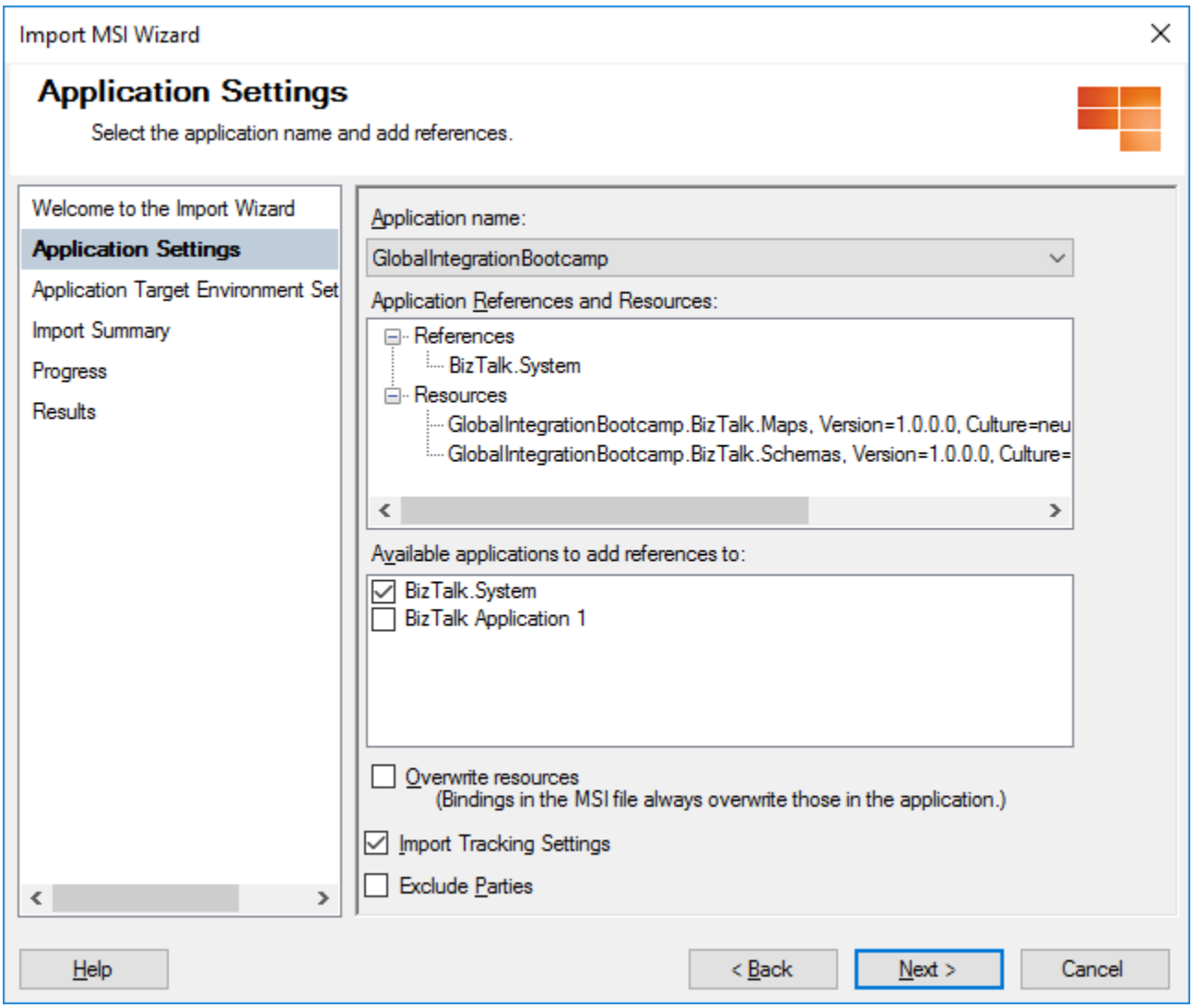


BizTalk application

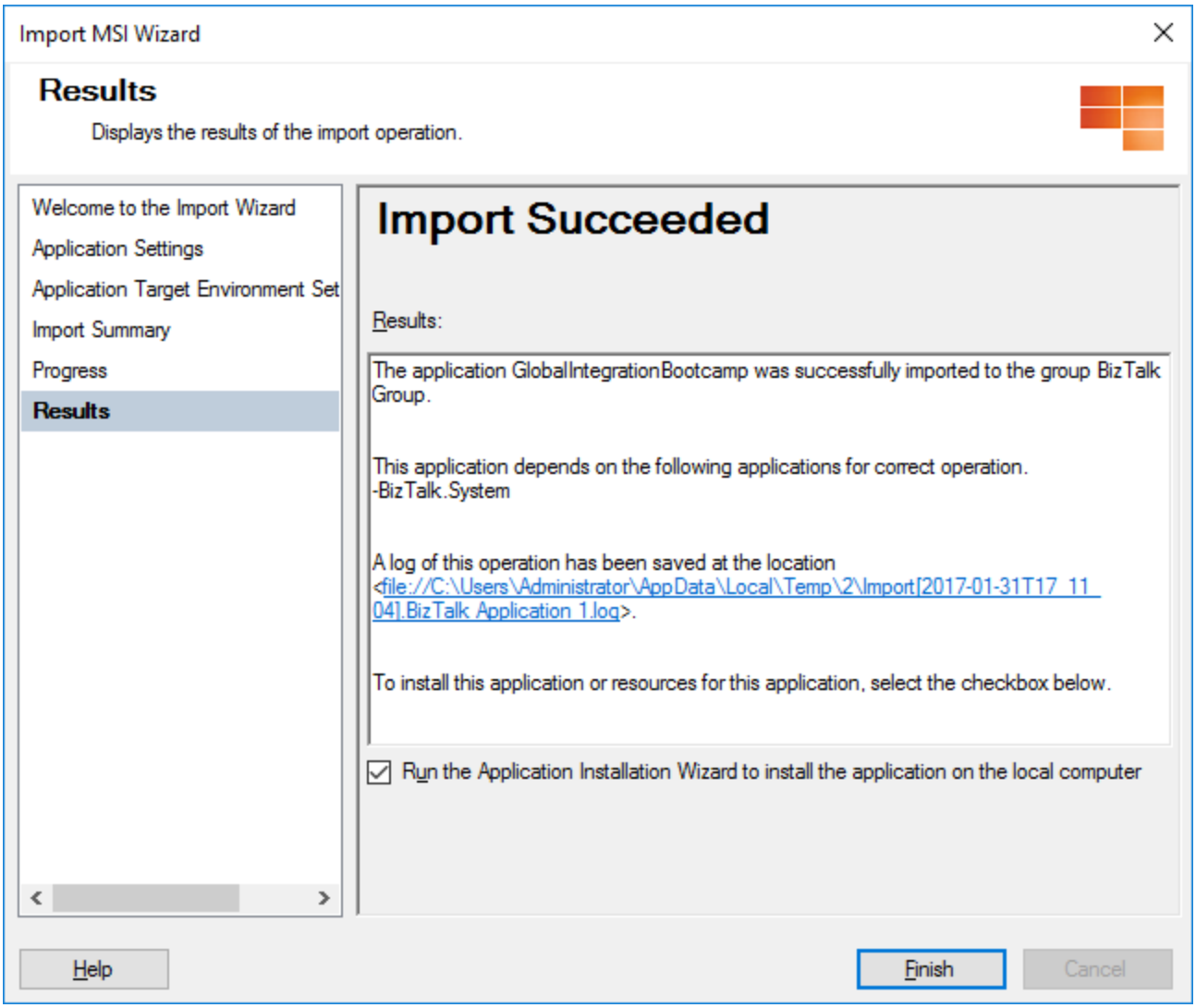
An application has been provided for this lab, which can be downloaded from here. This application contains the schemas, maps and pipelines needed for this lab.

Import

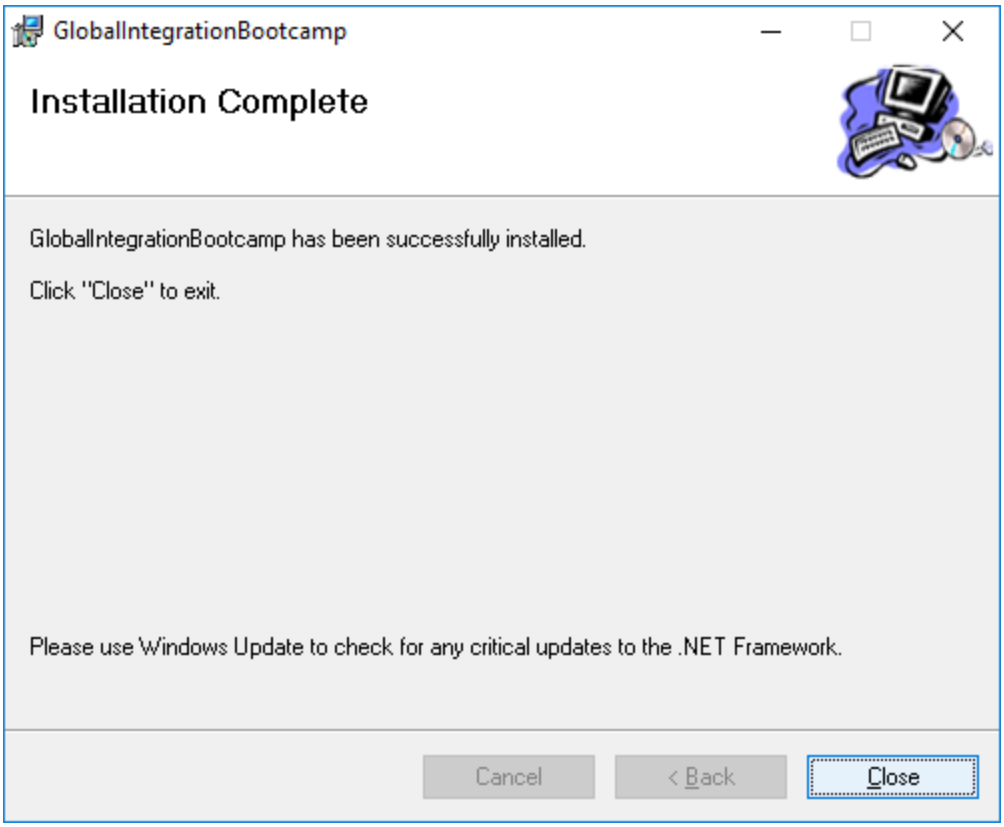
Import the application in the BizTalk Server Administration console, using all default values.



Once imported, make sure to also install the application.



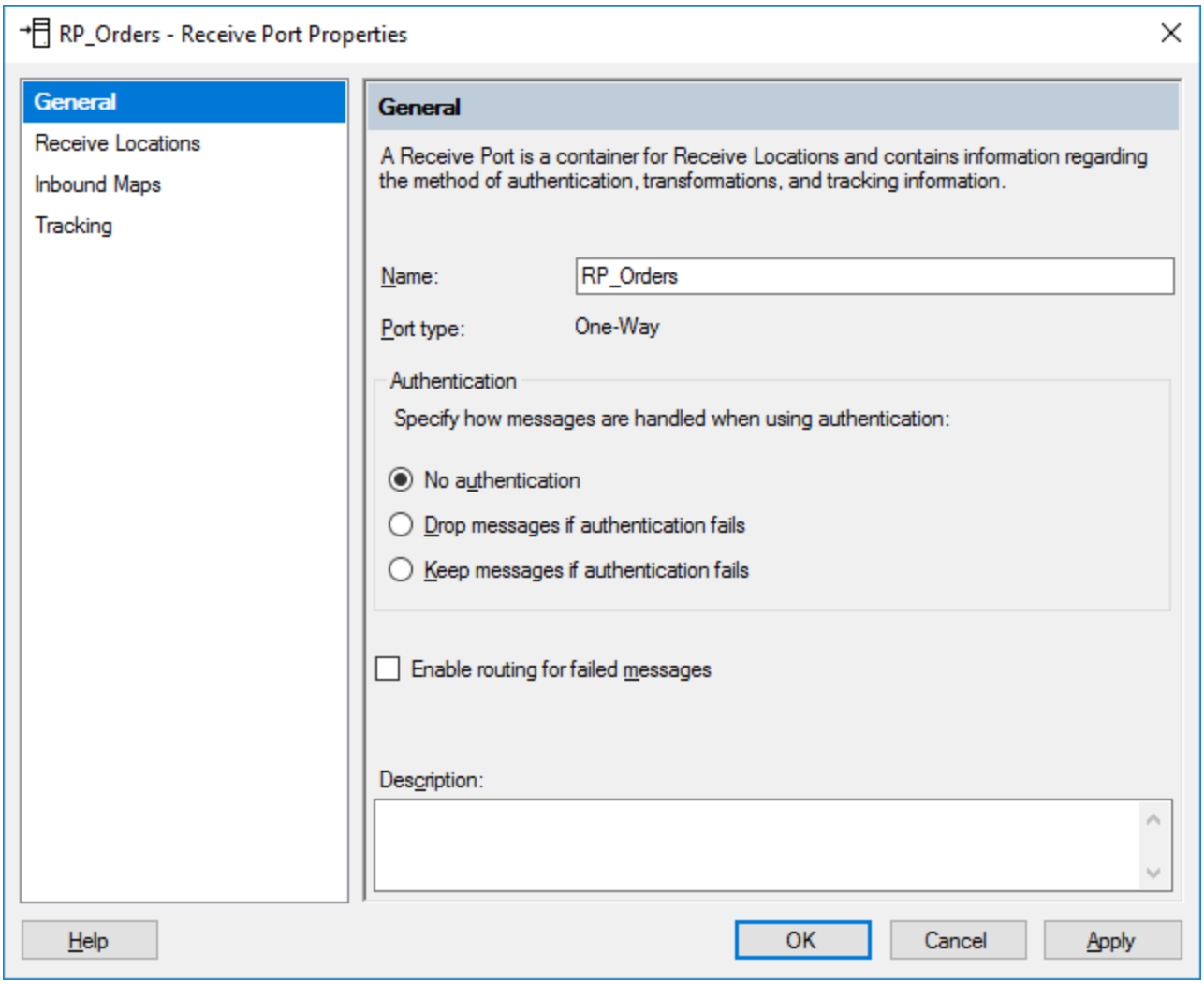
For the installation, once again use all default settings.



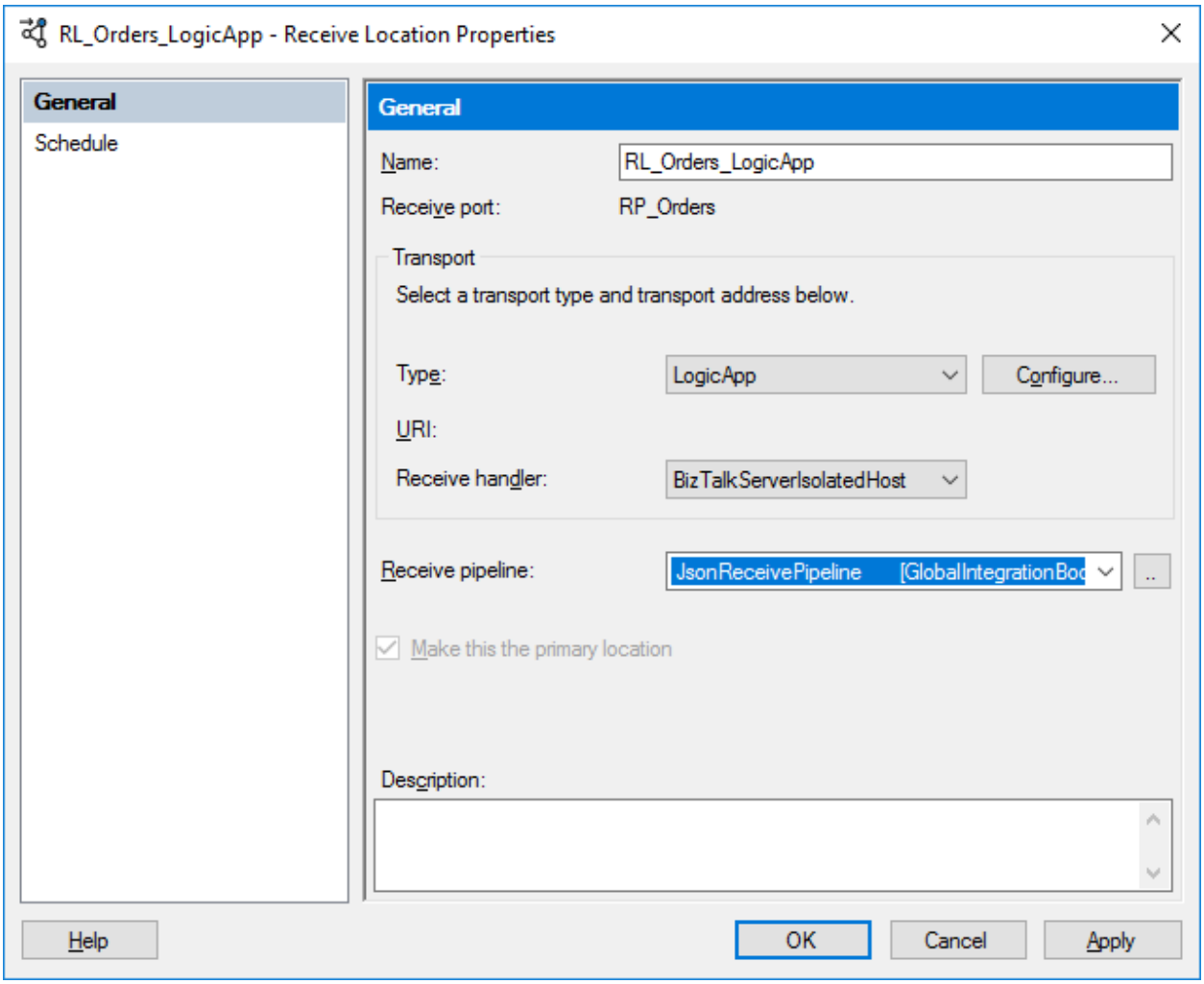
After the application has been installed, remember to restart your host instance and do an iisreset.

Create Receive Location from logic app

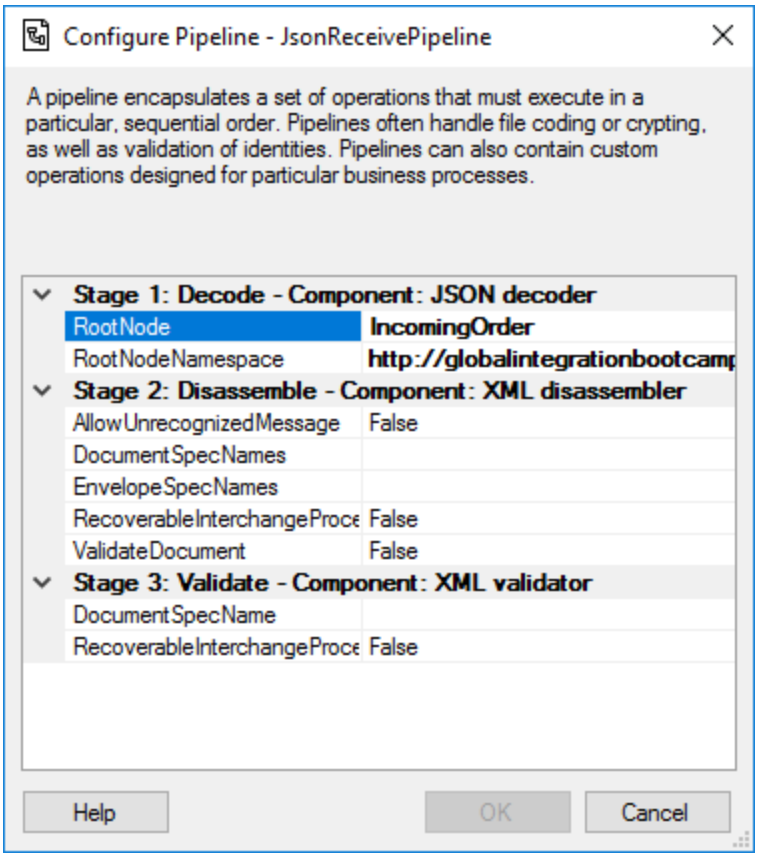
Now that the application is installed, we can add our ports to it. We will start with the receive location which will be called by our Logic App to send us an order.



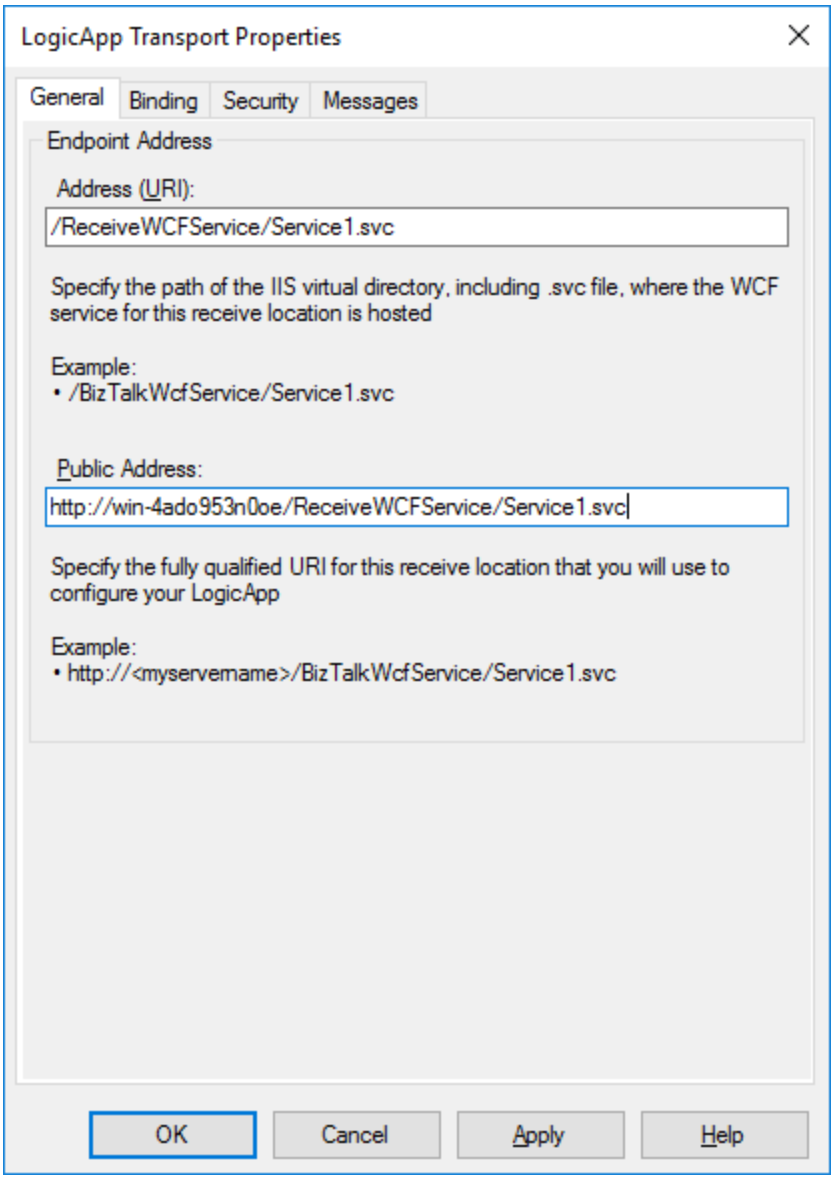
Select the LogicApp adapter, and the JsonReceivePipeline.



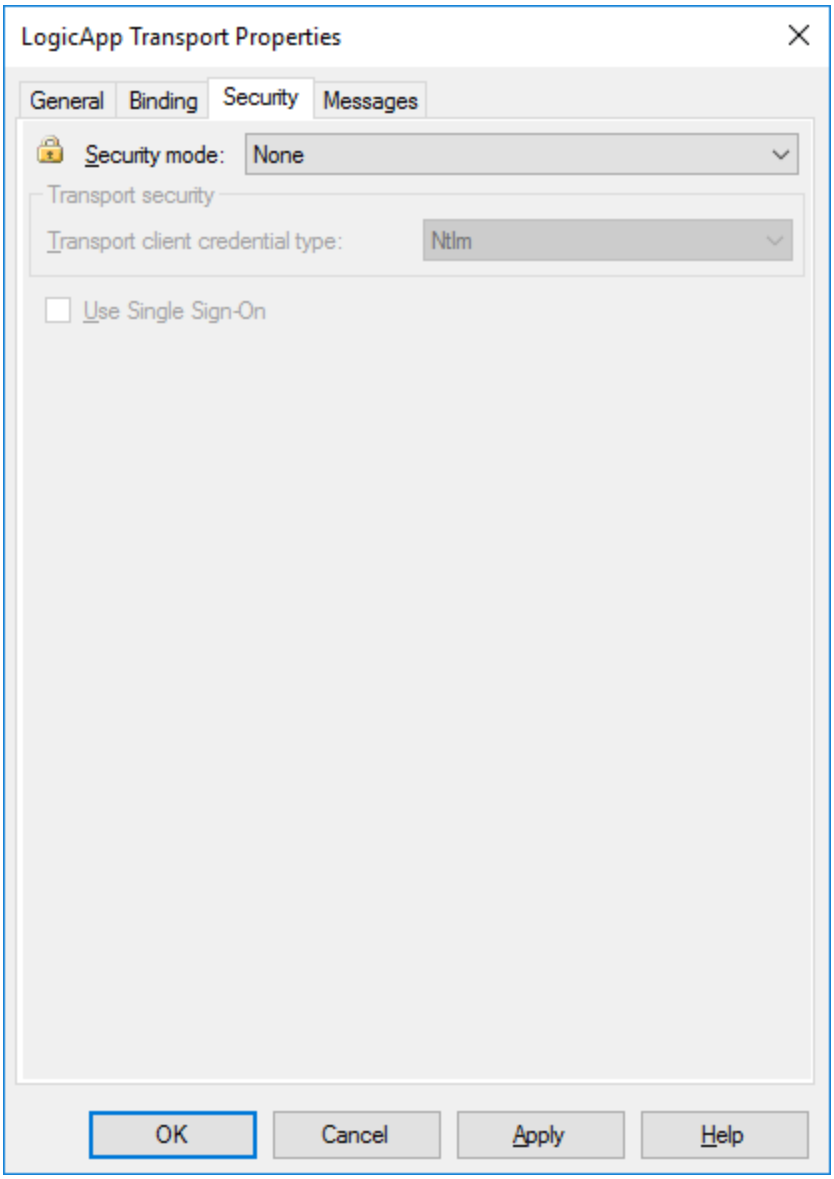
As our Logic App will send a JSON message, we will need to decode this and add our rootnode.
Set the RootNode attribute to **IncomingOrder** and the RootNodeNamespace to **http://globalintegrationbootcamp.com/Orders/V1-0** (properties for pipeline can be opened by clicking the ellipses button next to the pipeline drop-down).

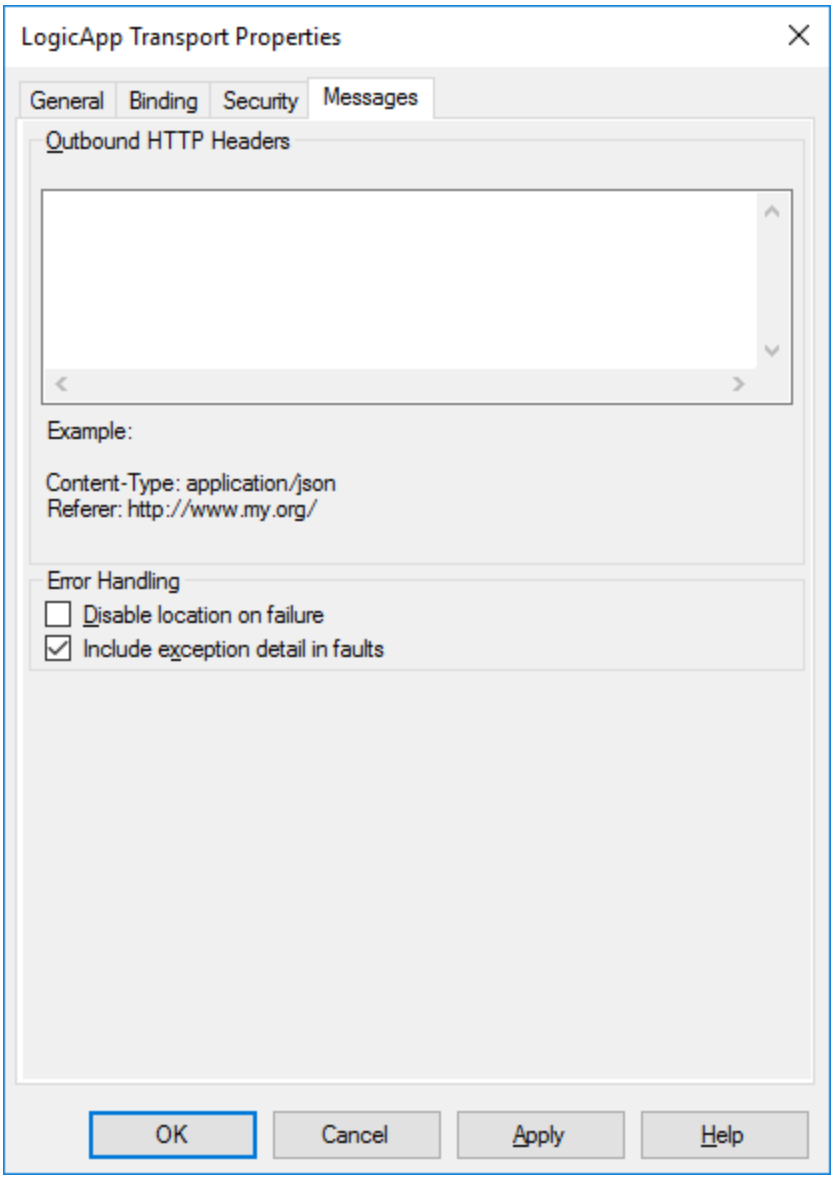


In the configuration of the LogicApp adapter (can be opened by clicking the **Configure…** button next to the port Type which we set to LogicApp), specify the address of the Receive service we created earlier. For the public address, this includes the local server name.

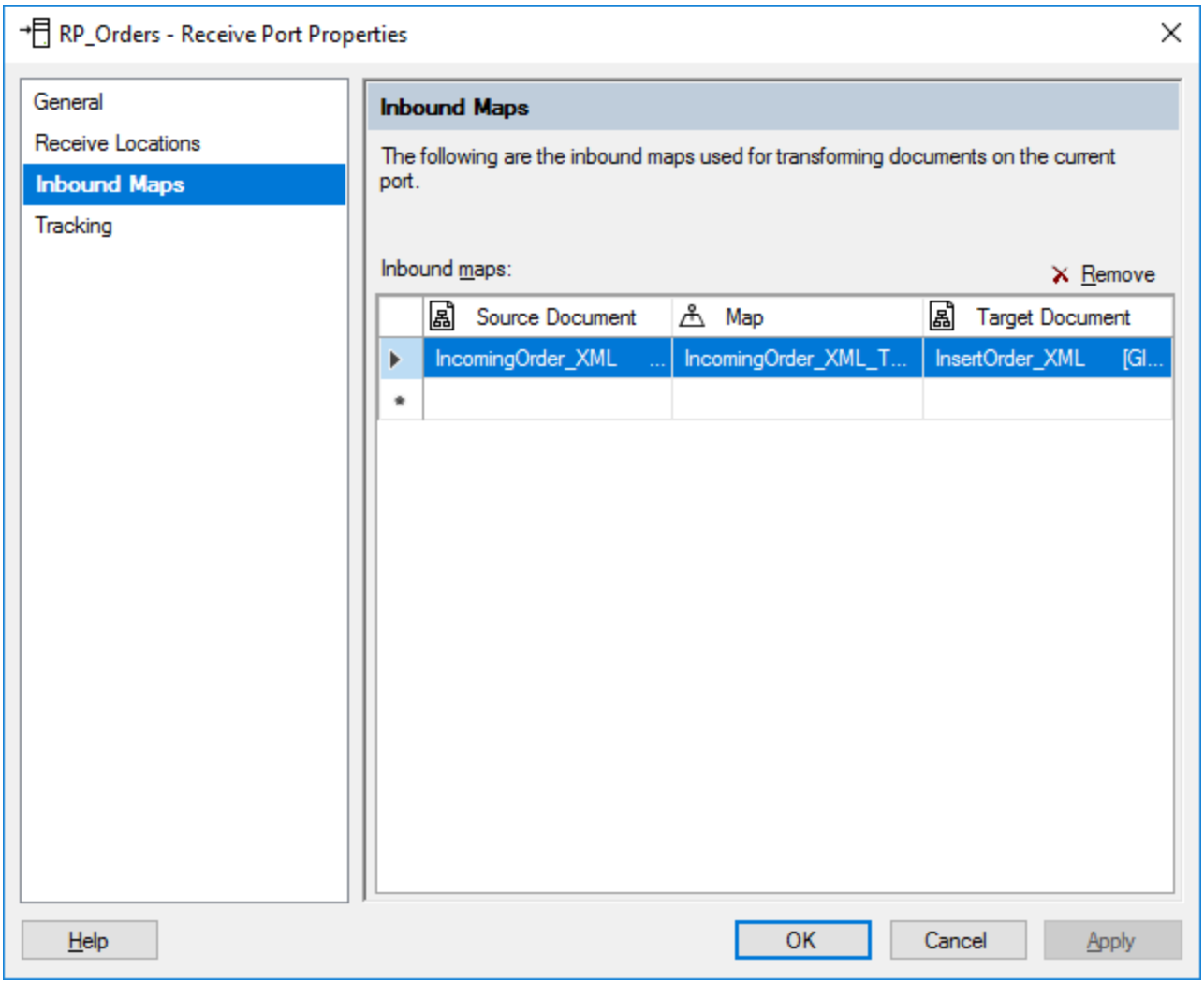


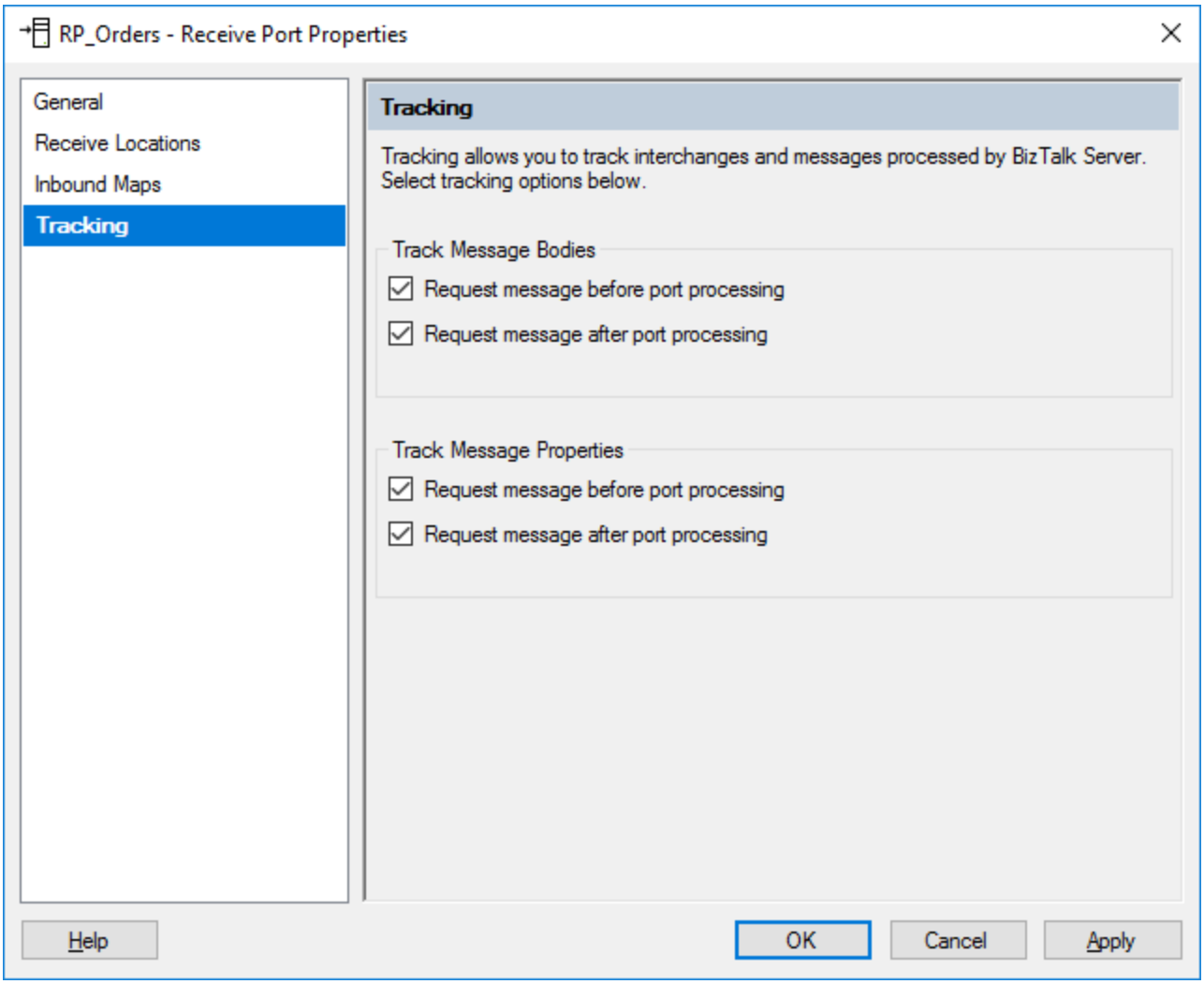
For this lab we will not use security, of course when doing this in production scenarios you will want to set this up.





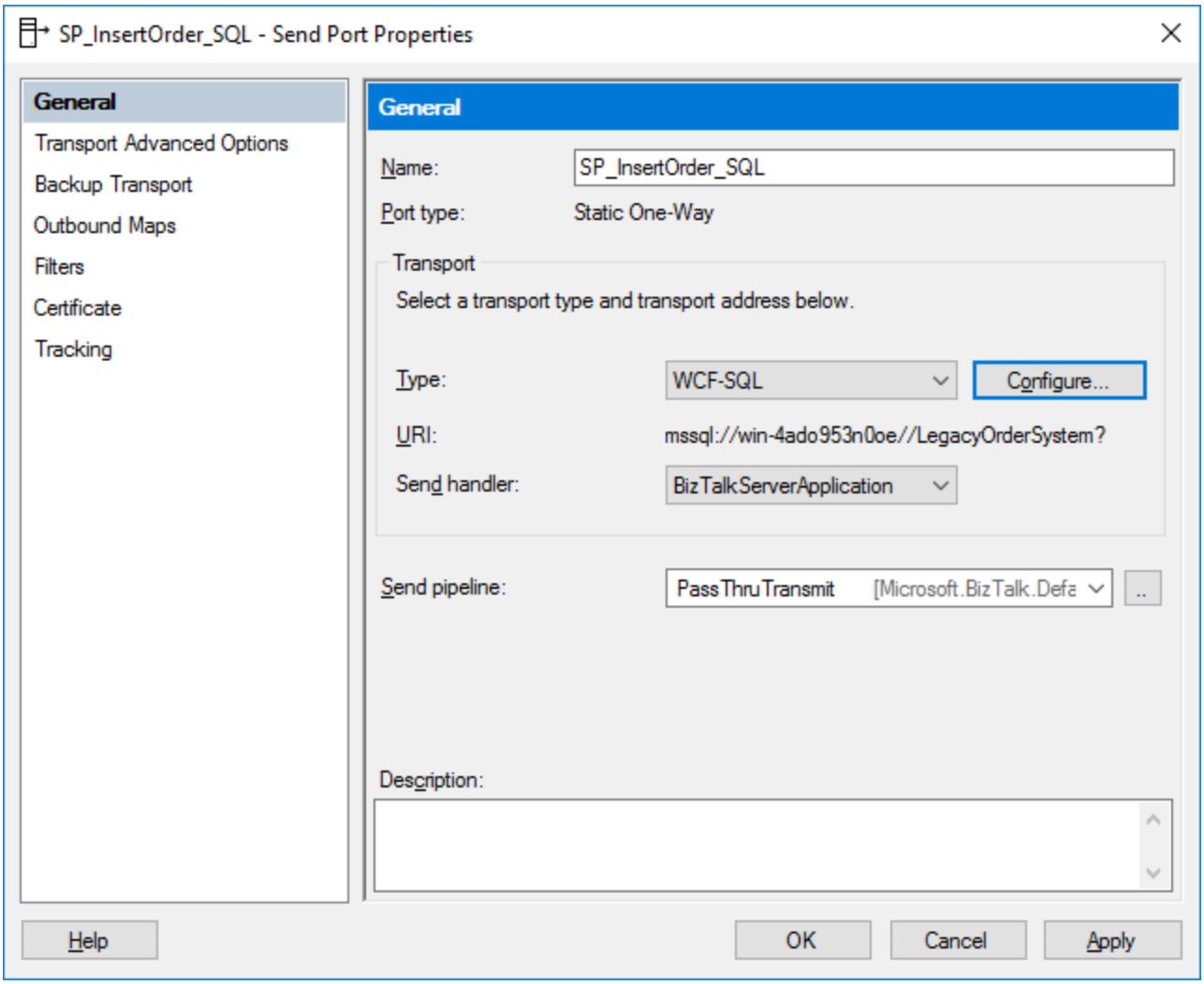
Add the map which will convert our incoming order to an order to be inserted into the database.



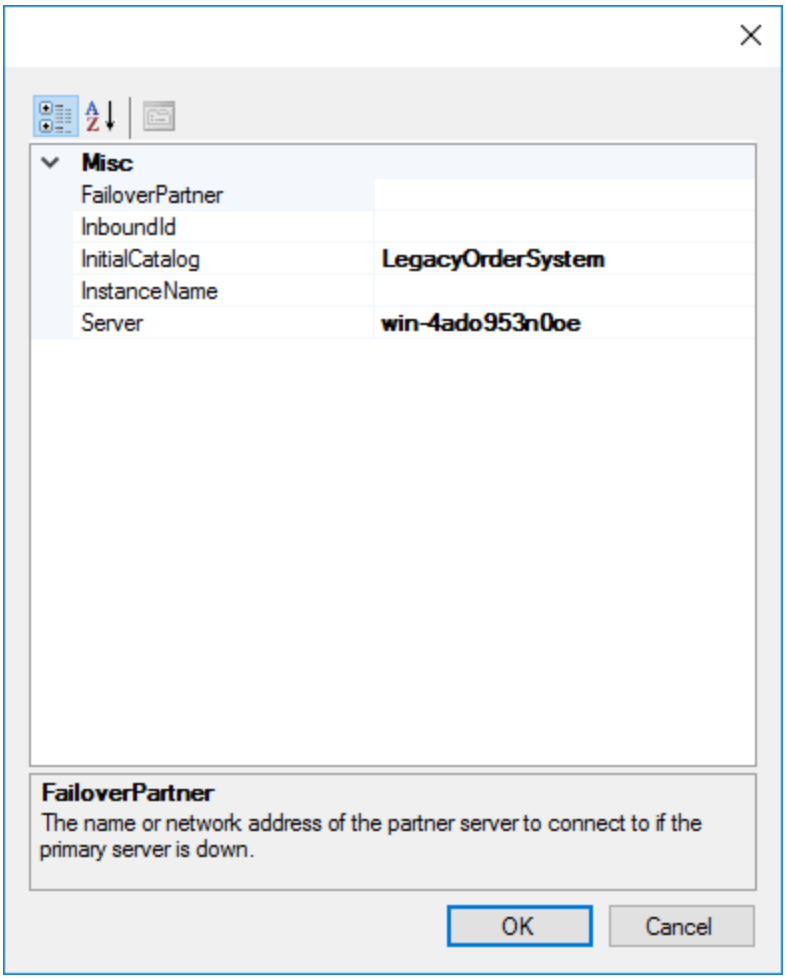


Create Send Port to database

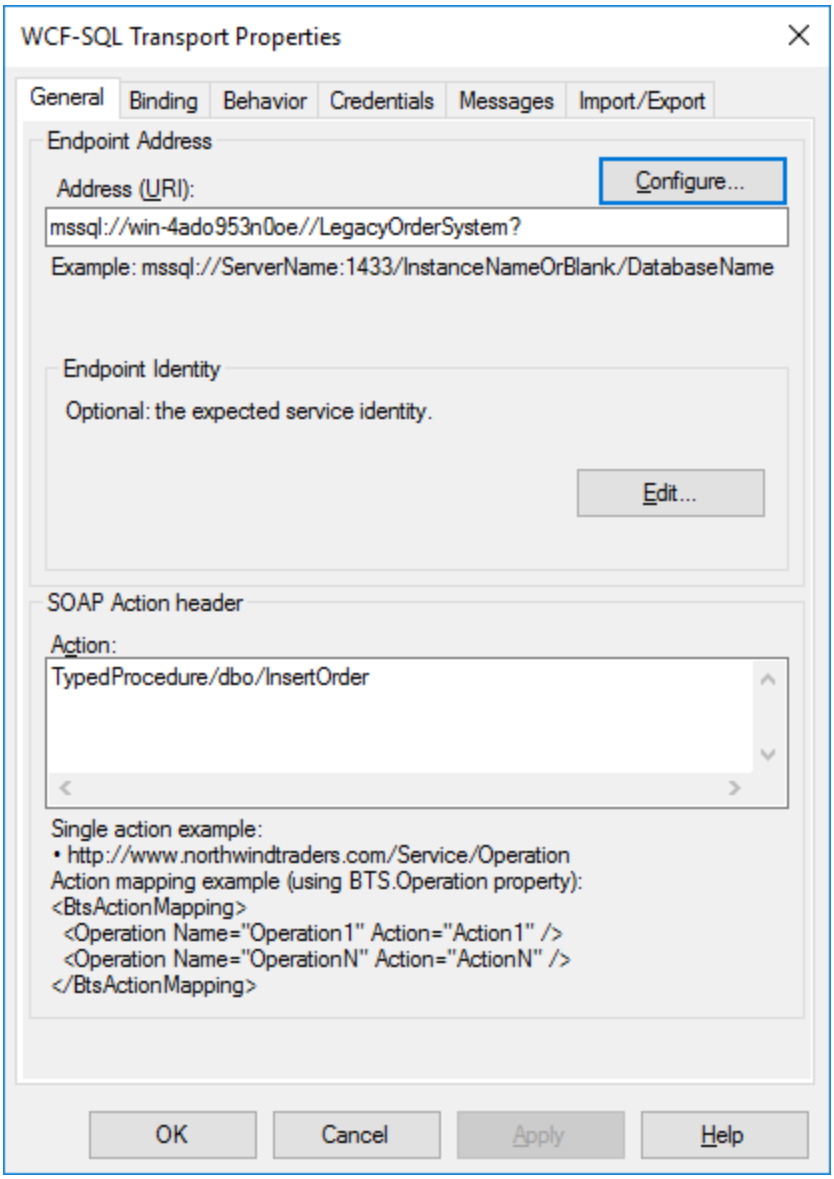
Next we will create a send port which uses the WCF-SQL adapter to insert the incoming order into the database.



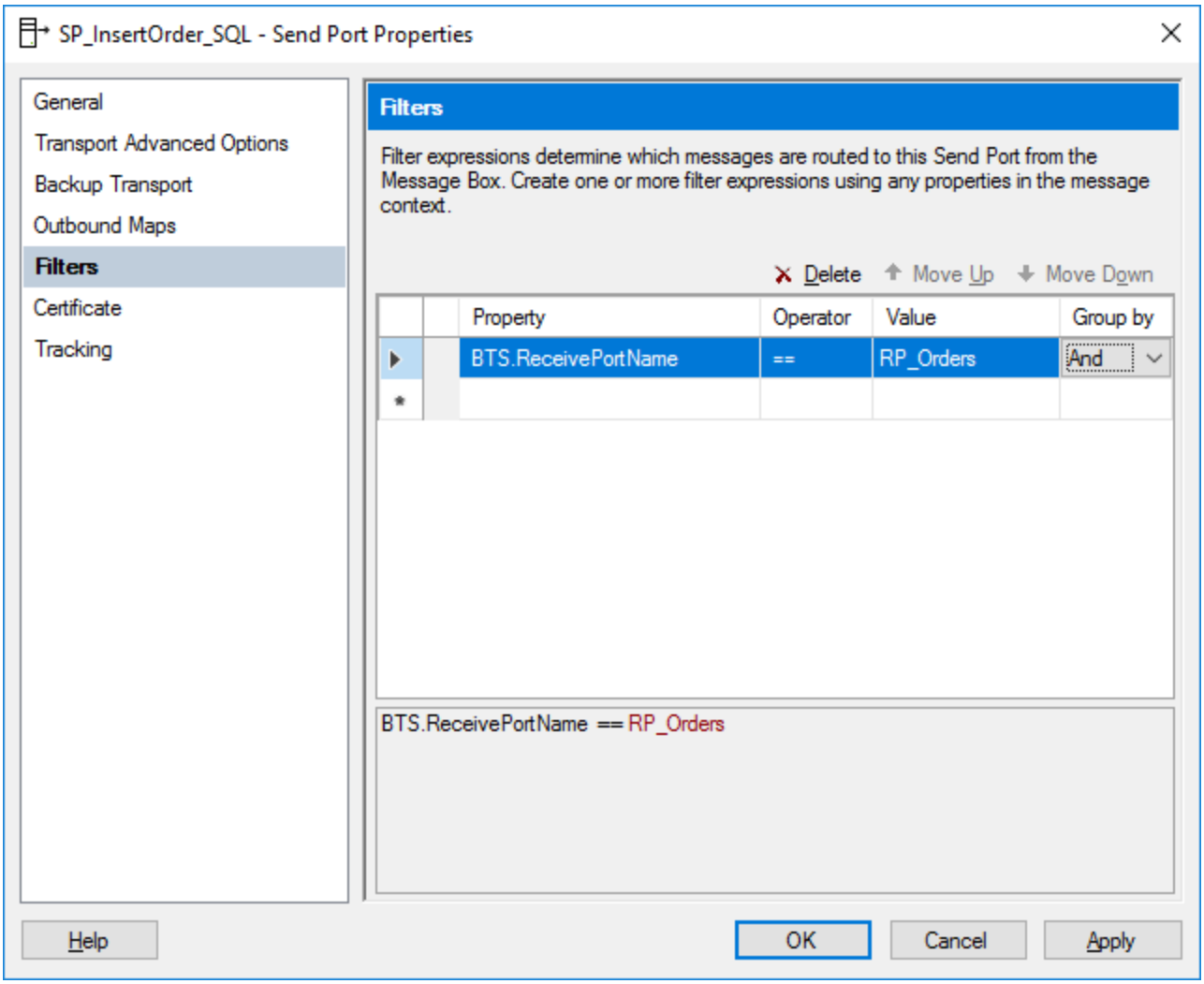
Make sure to specify your local server name (or **localhost** or **.**) and database name when configuring the adapter.

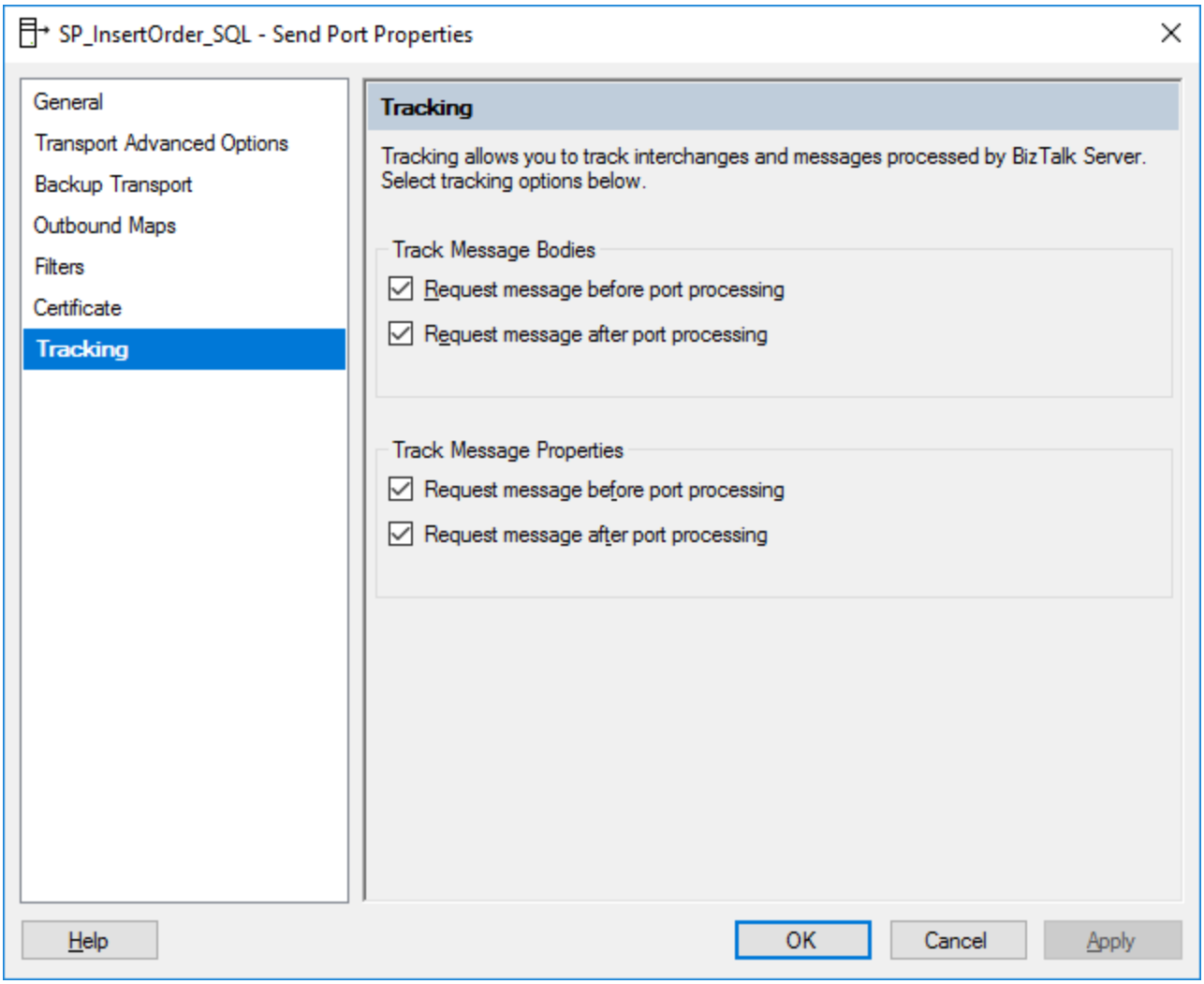


As we will be using a stored procedure to insert the order into the database, set the **Action** of the adapter to **TypedProcedure/dbo/InsertOrder**.



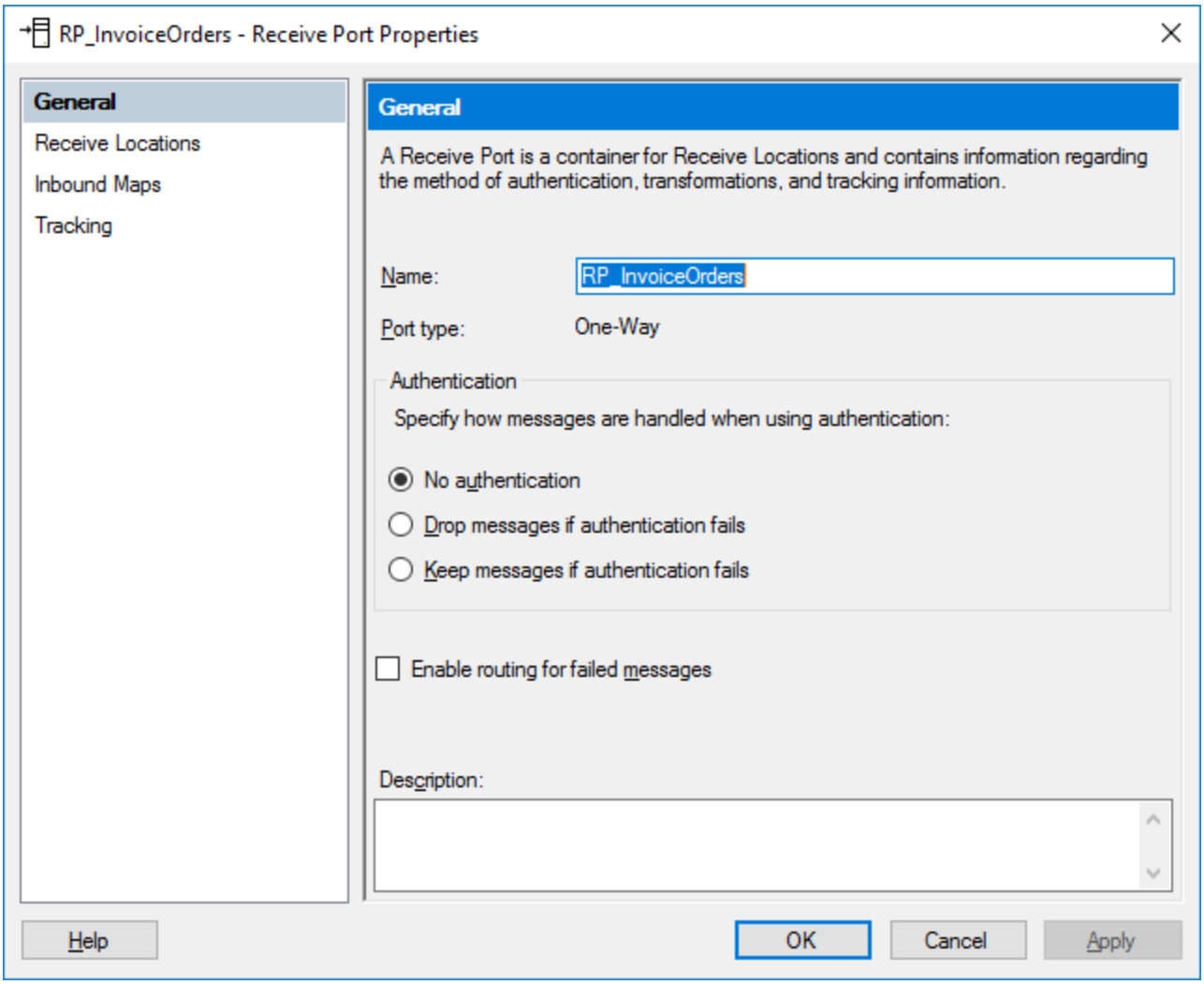
Set the filter to subscribe to the incoming messages.

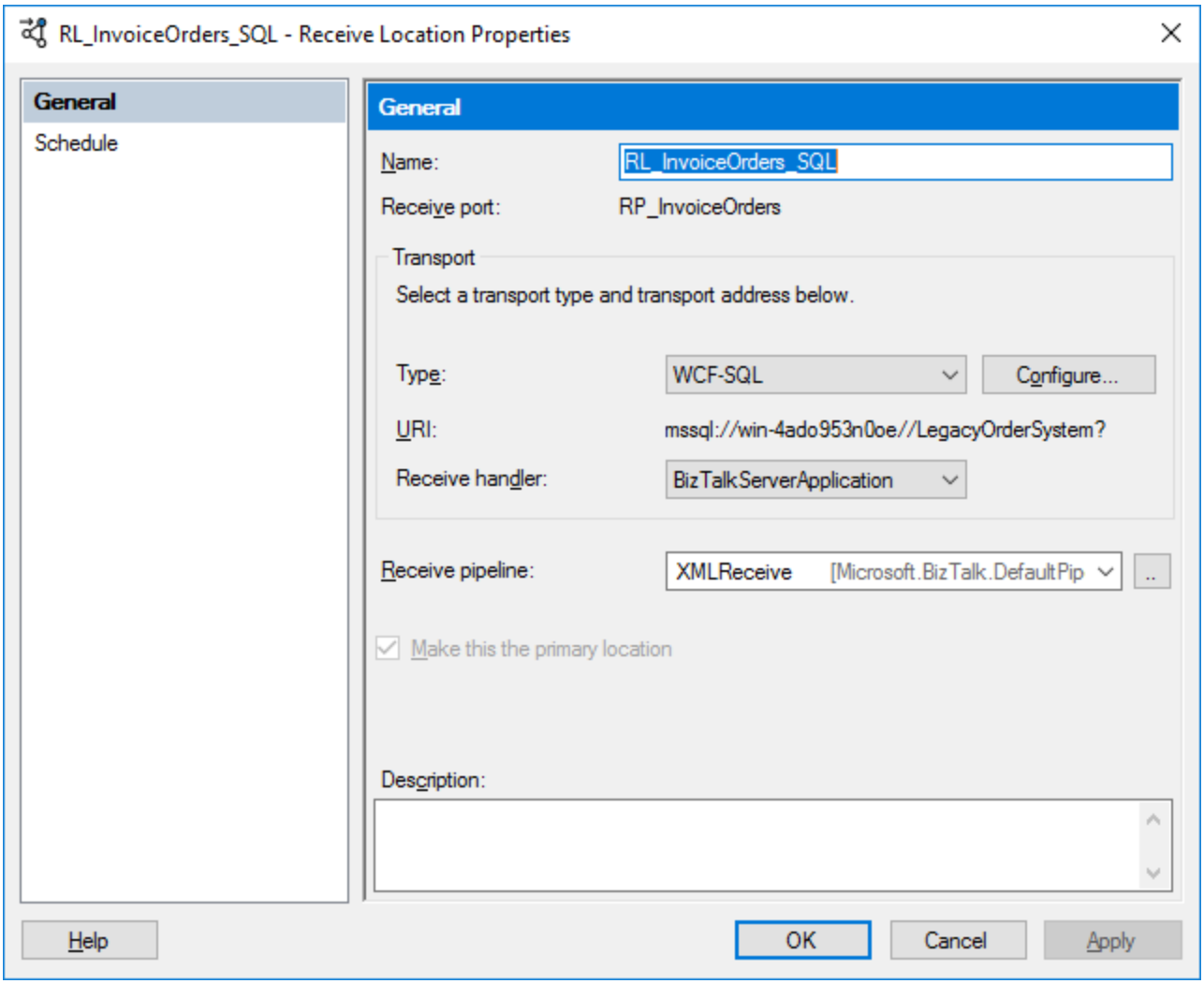




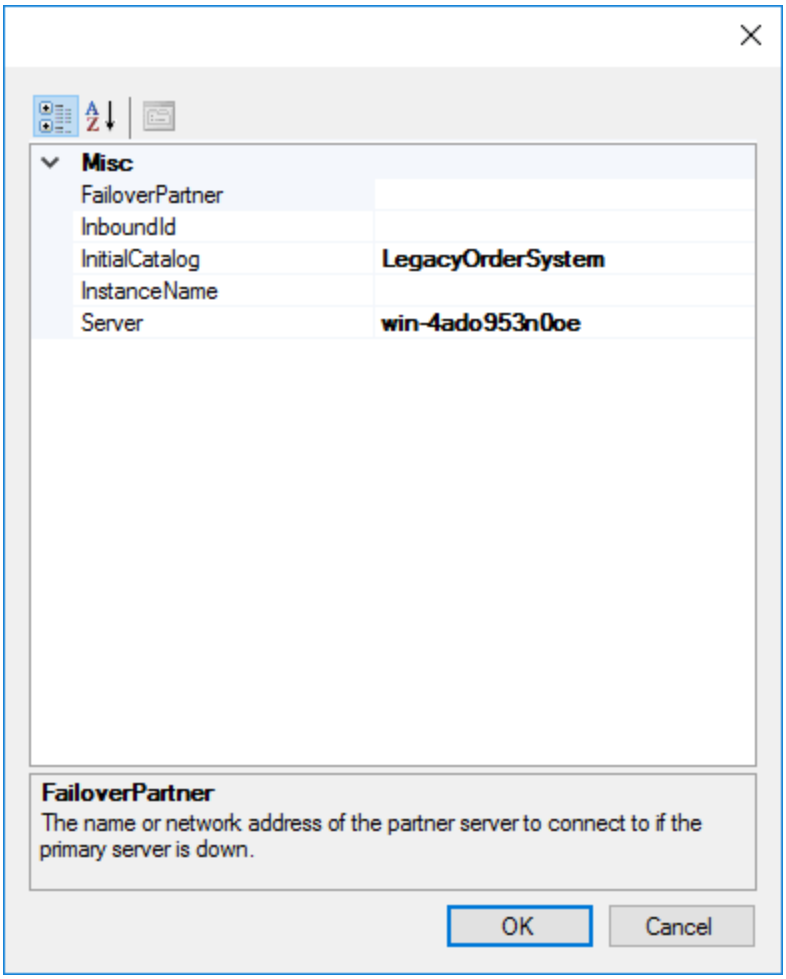
Create Receive Location from database

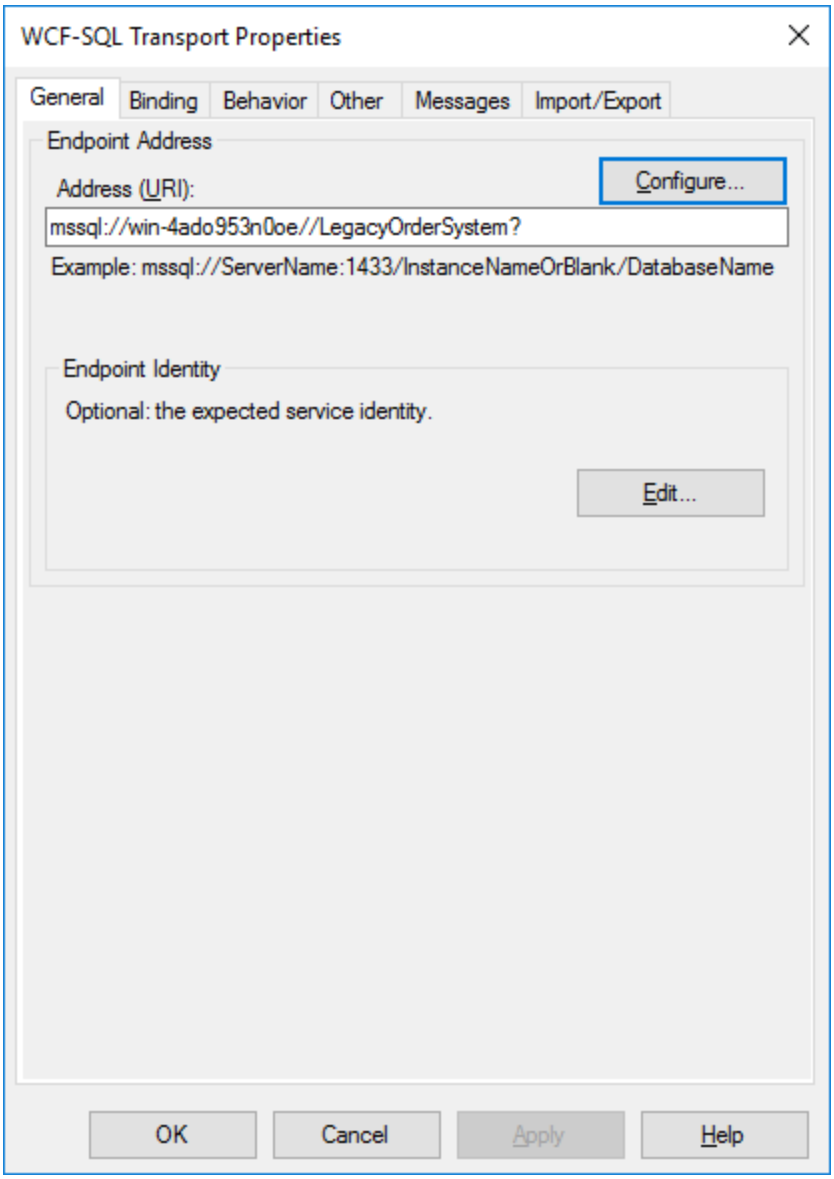
We will now add a receive port which will poll our database to check for orders on which the customer type has been set.





Once again, set the local server name and database in the adapter properties.



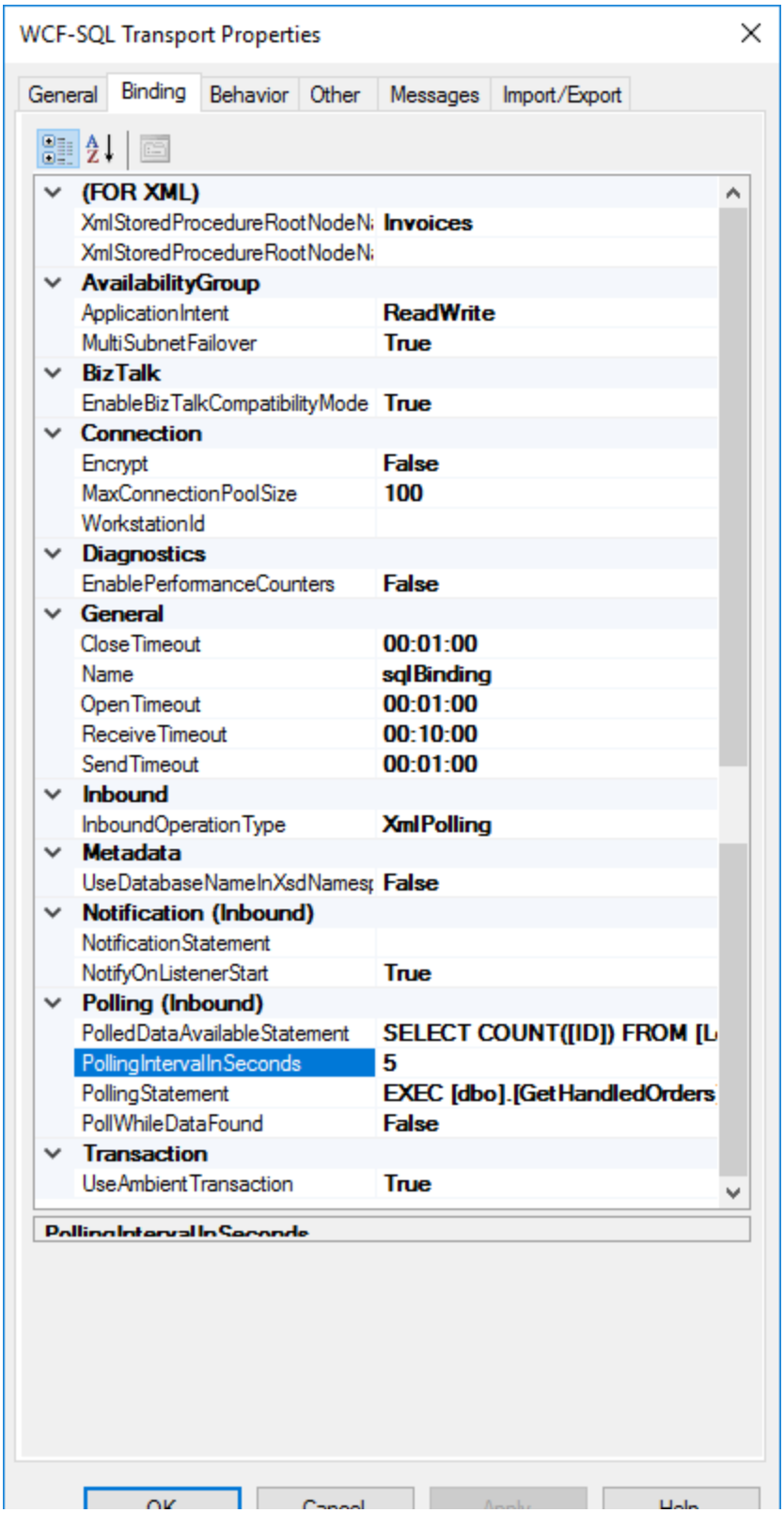


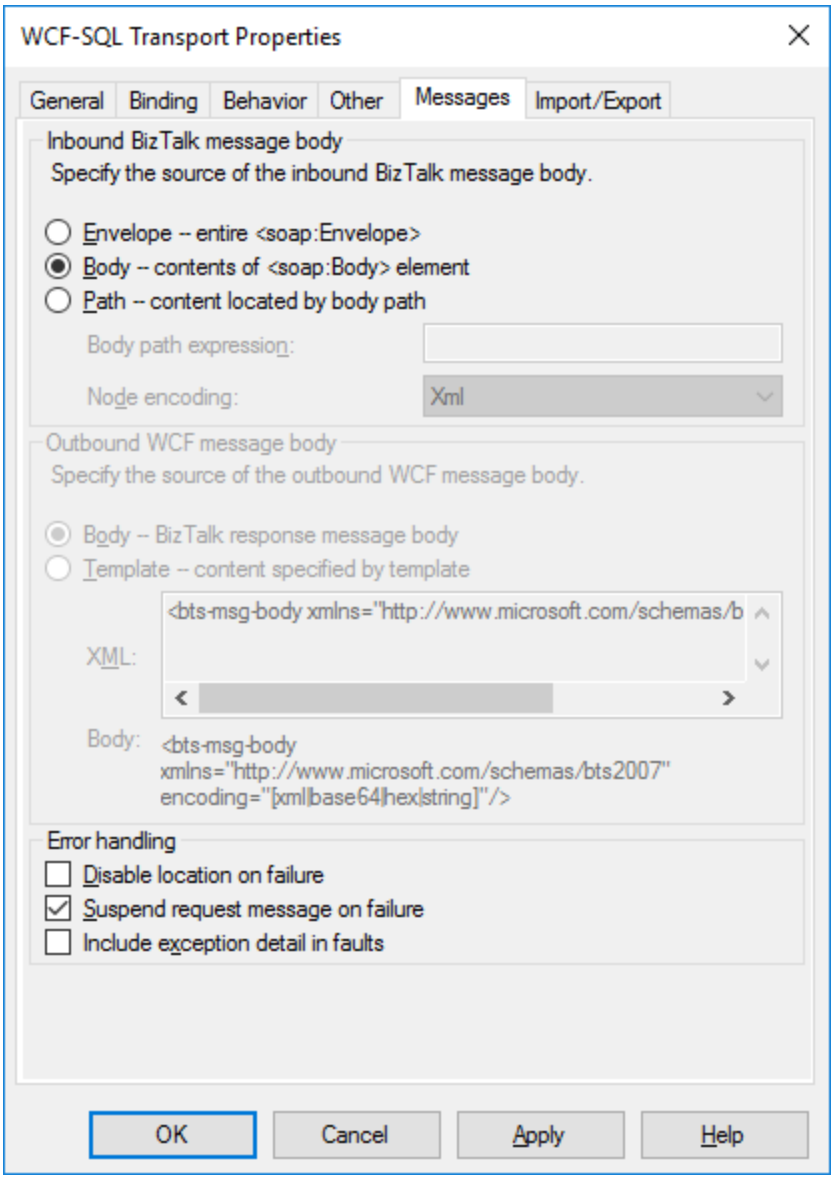
Set the InboundOperationType to **XmlPolling**, to retrieve the results from the stored procedure as an XML message. The polling available statement will be set to the following statement, which checks if any new rows are available.

SELECT COUNT([ID]) FROM [LegacyOrderSystem].[dbo].[Order] WHERE [CustomerType] IS NOT NULL AND [InvoiceSent] IS NULL

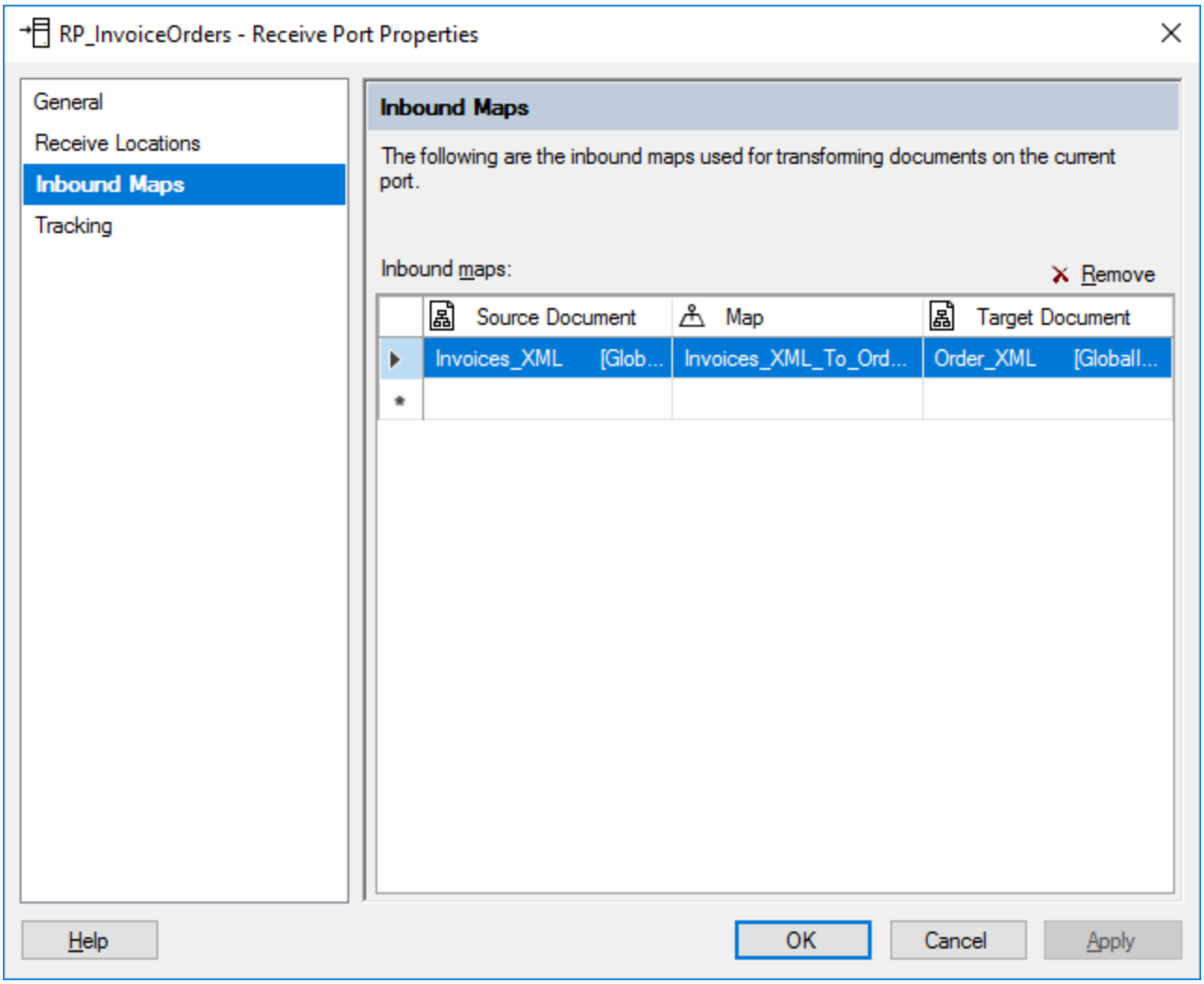
The polling statement will be set to execute the stored procedure.

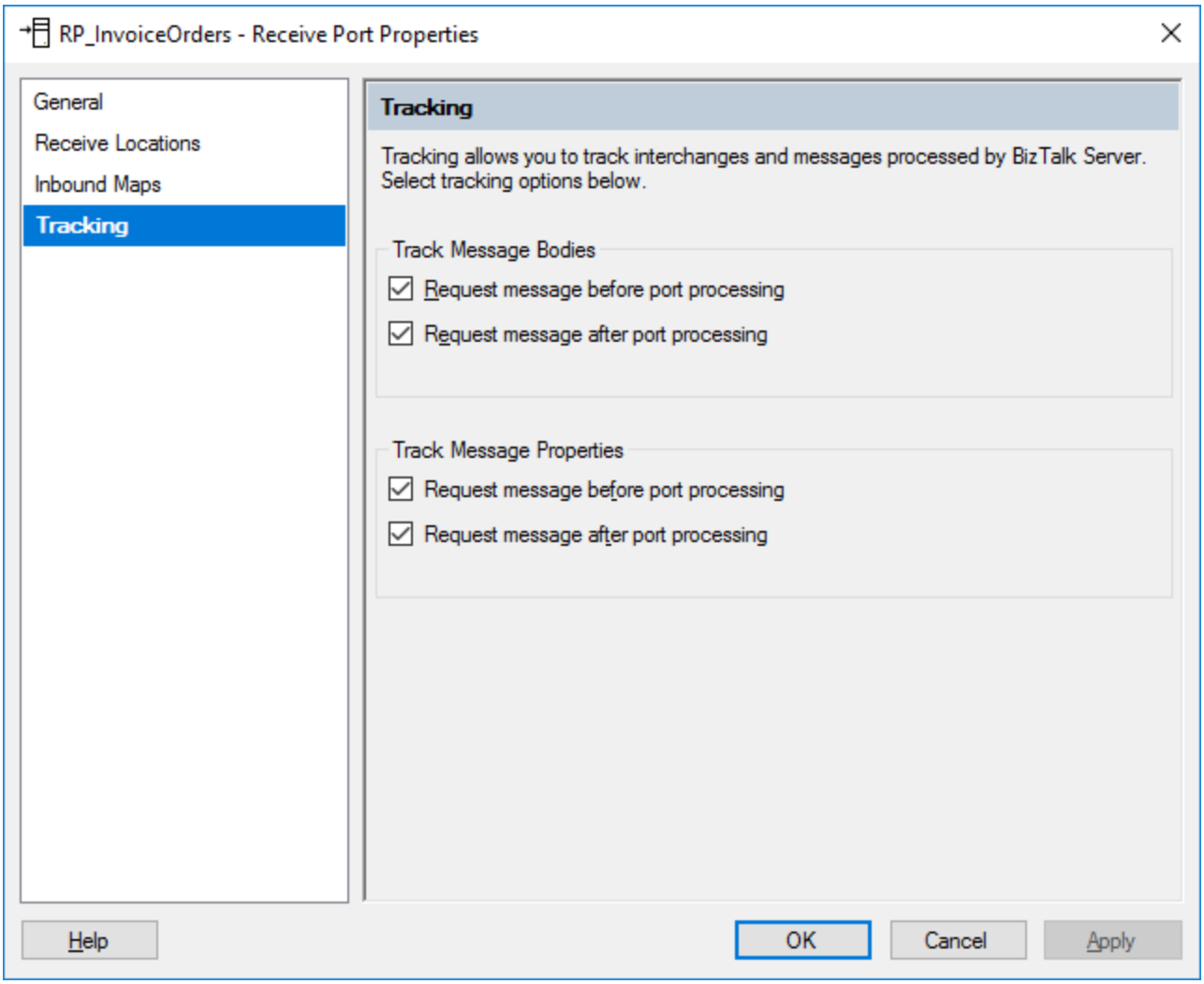
EXEC [dbo].[GetHandledOrders]





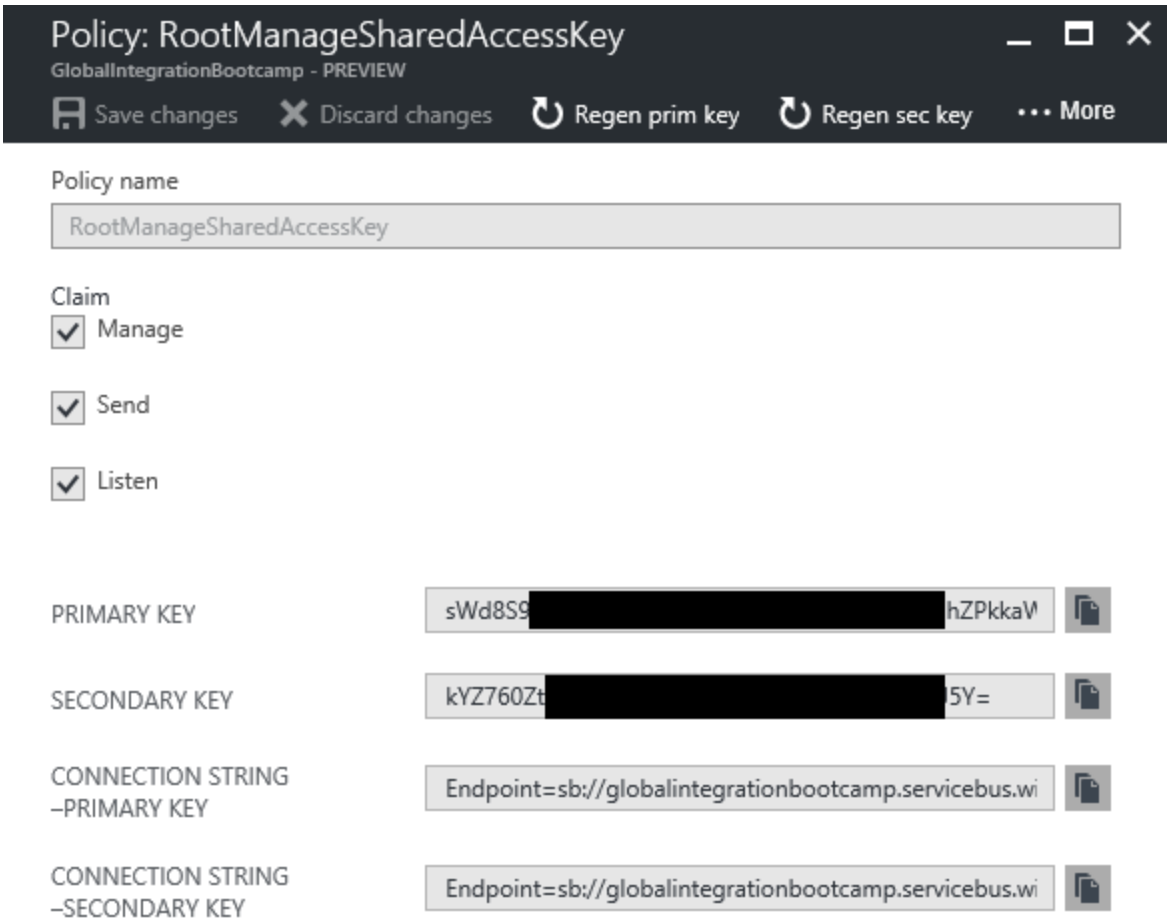
Our map will map these updated orders to our outgoing order message.



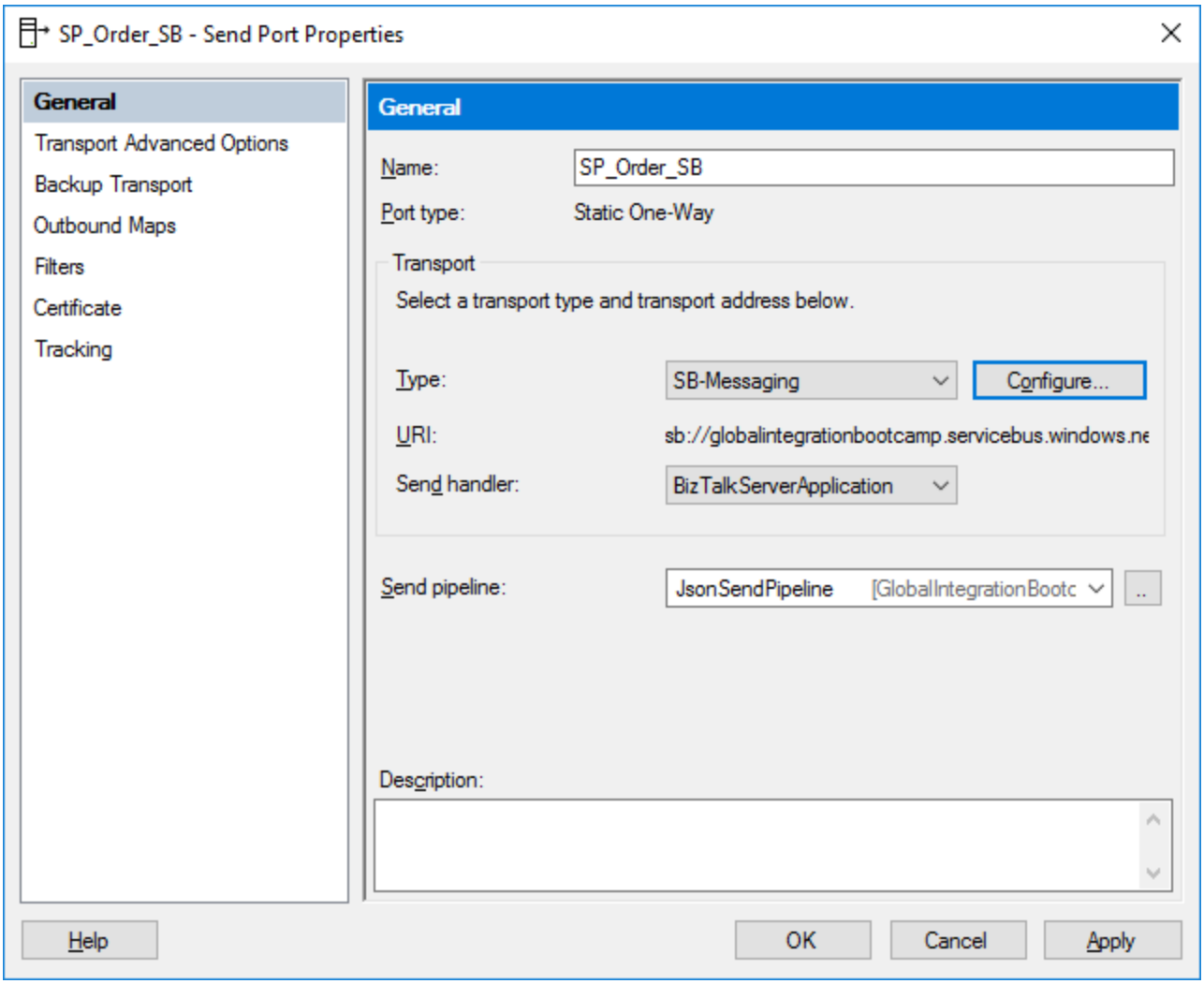


Create Send Port to Service Bus

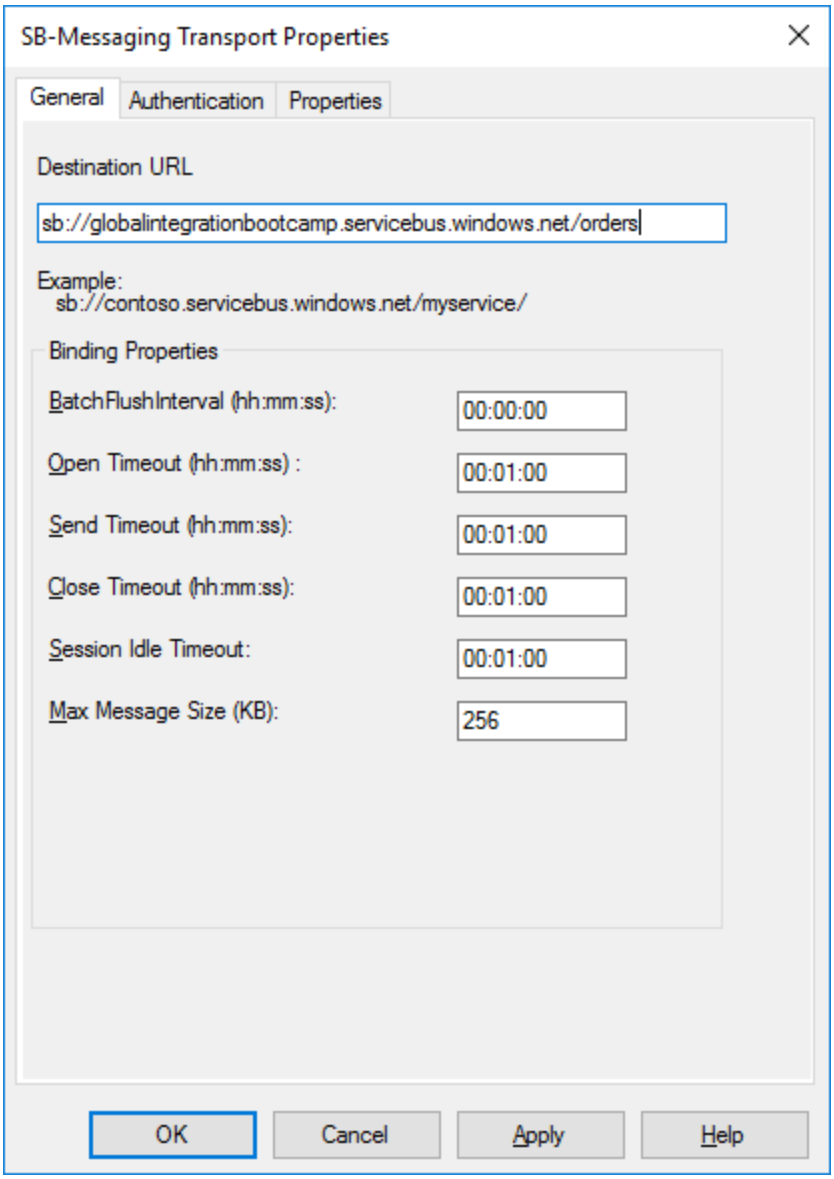
Our last port to be created will be the send port which will send our outgoing order message towards the Service Bus topic. Retrieve a key (primary or secondary key) from the portal, as we will use this to authenticate. In this lab we will use the RootManageSharedAccessKey, but in real-life scenarios we would of course use a dedicated send key for this.



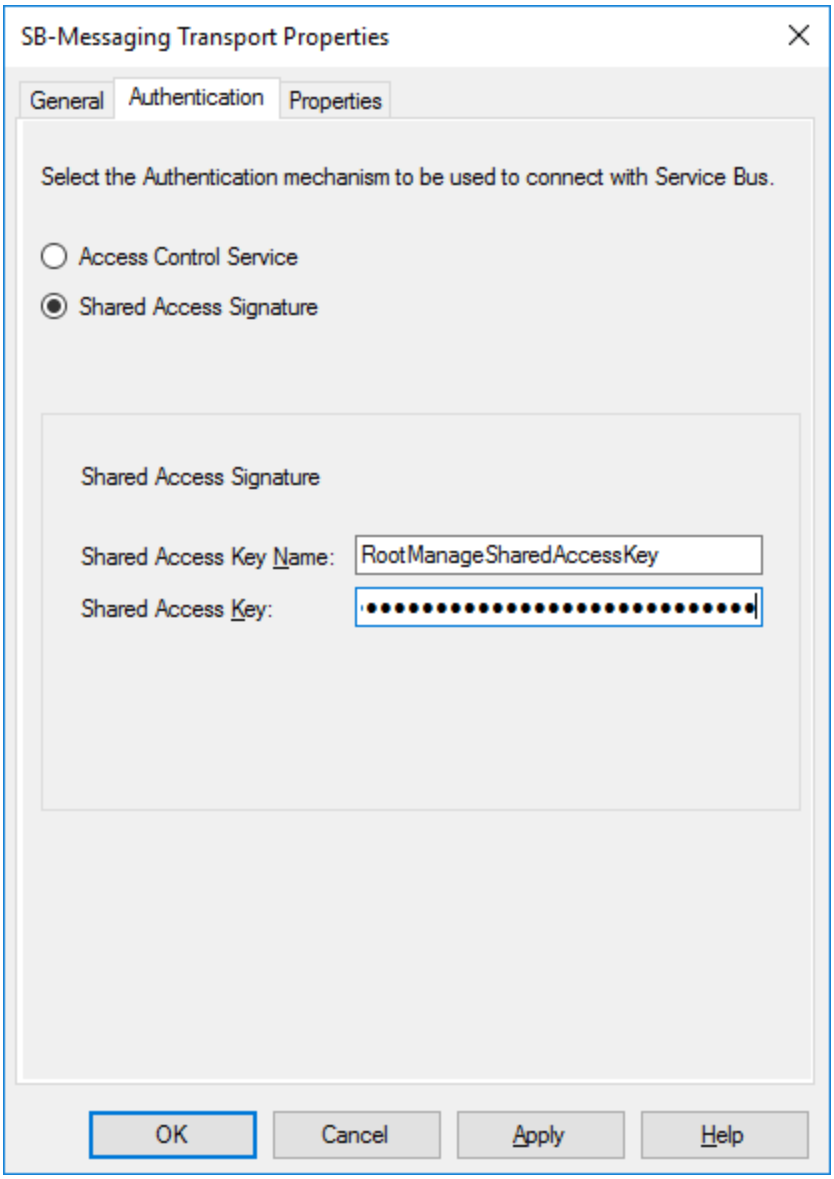
Select the **SB-Messaging** adapter, and set the Send pipeline to the JsonSendPipeline.



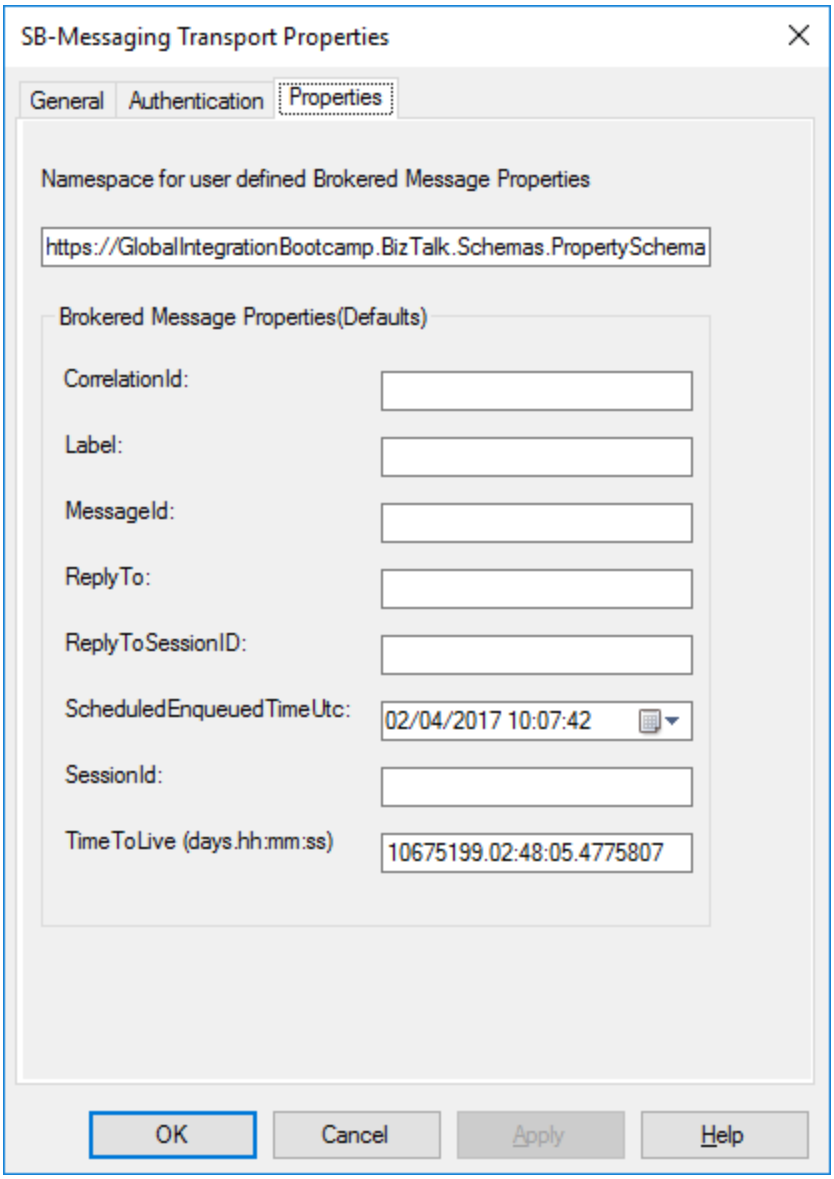
In the adapter properties, set the correct namespace in the destination URL, and add the name of the topic as the suffix.



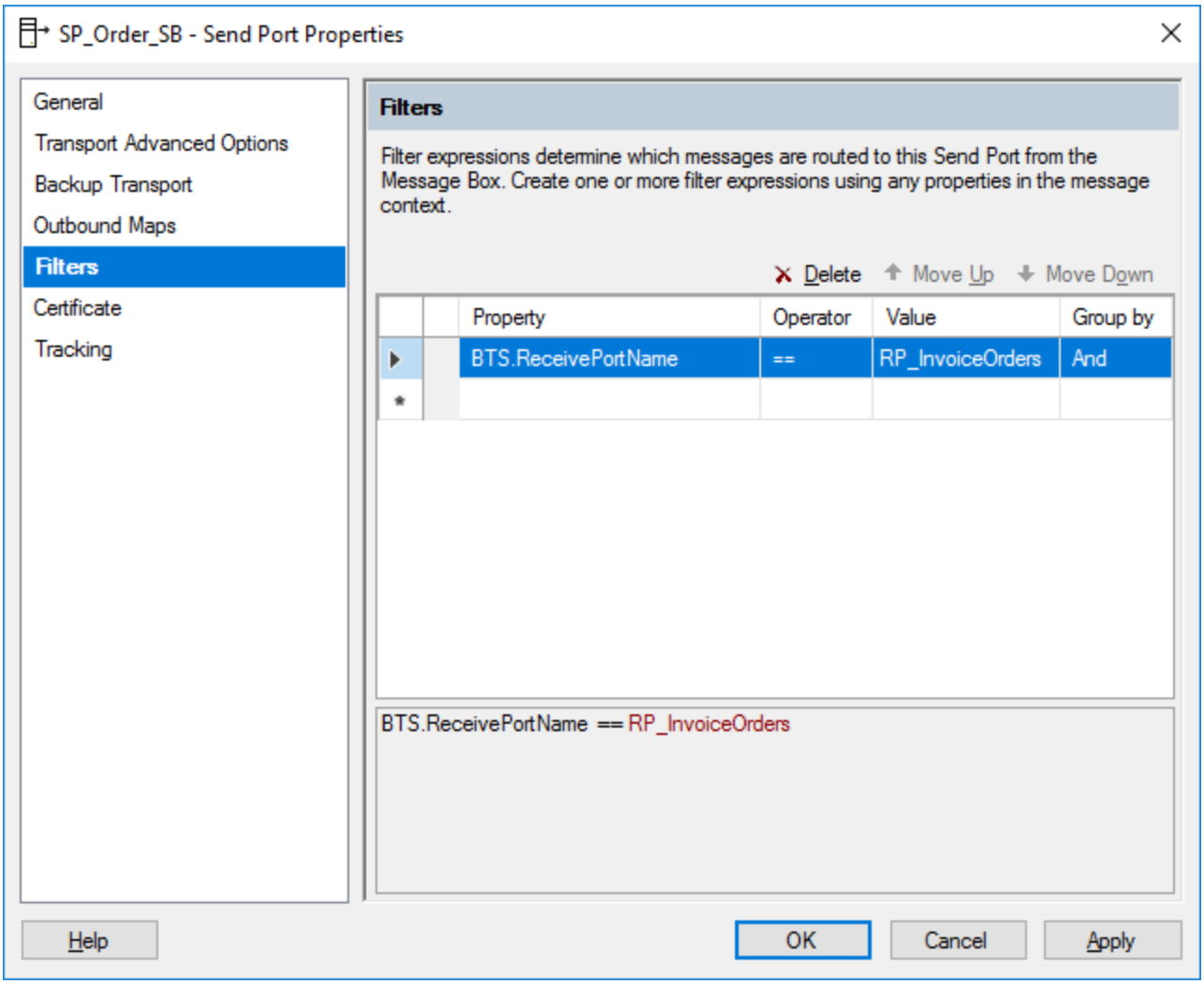
Set the authentication name and key we retrieved earlier.

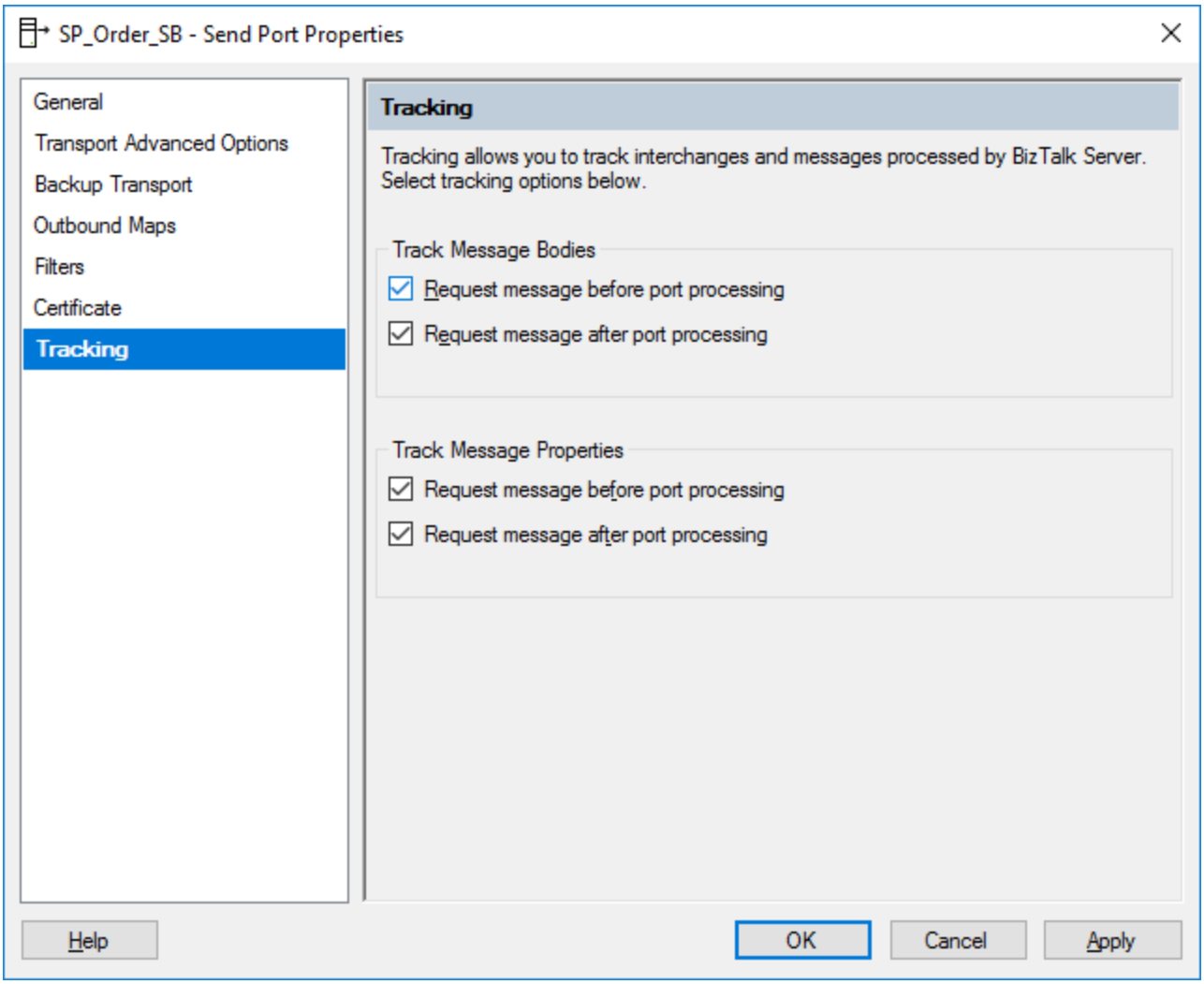


To be able to route our messages to the correct subscription on the topic, we have added the CustomerType property in a property schema, which we will use here by setting the namespace to **https://GlobalIntegrationBootcamp.BizTalk.Schemas.PropertySchema**.



Subscribe to the messages we received from the database.

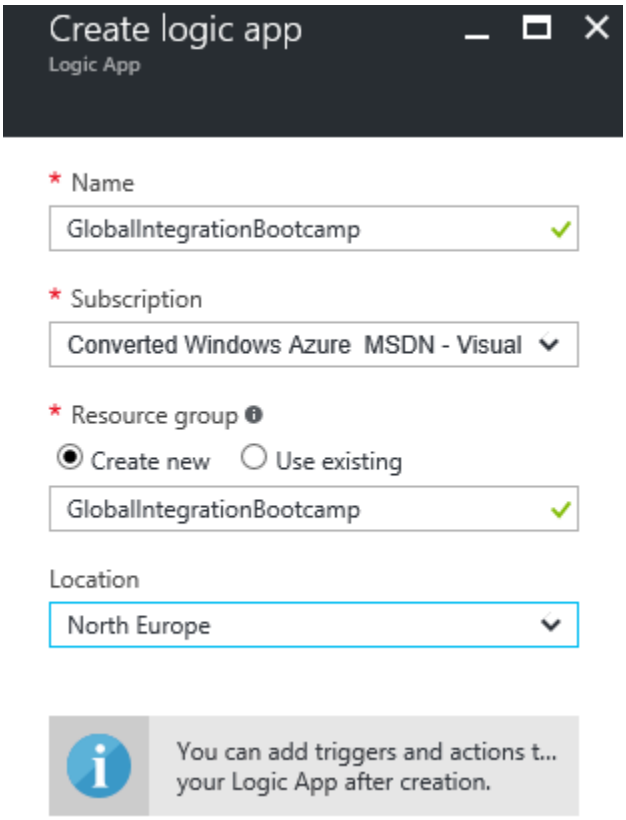




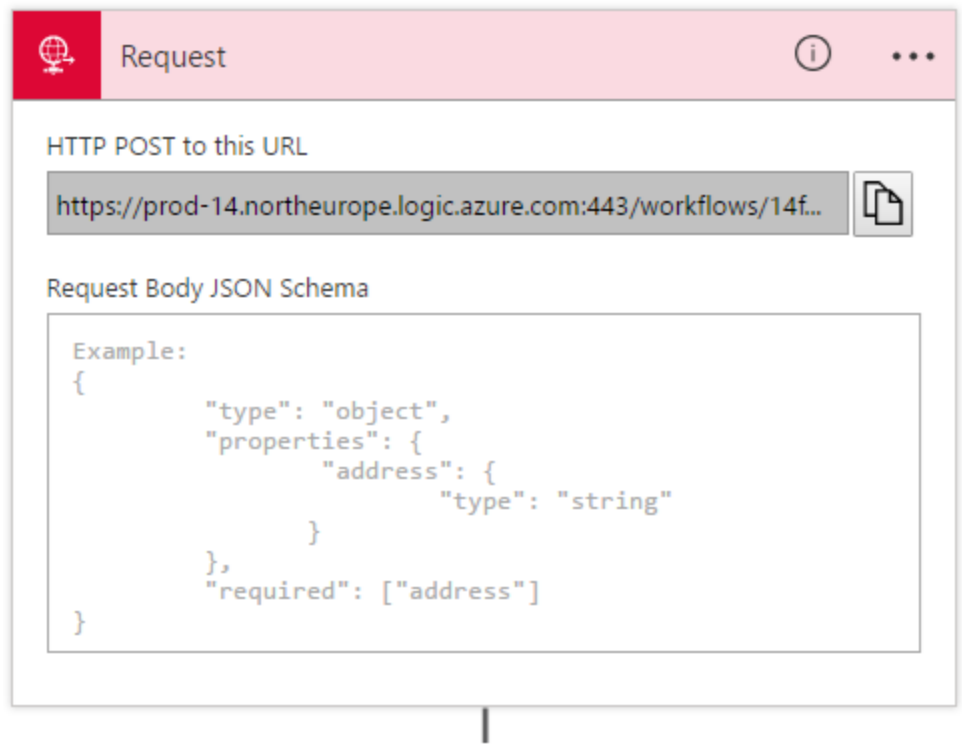
Once everything has been set up, start the BizTalk application.

Create Logic App which communicates with on premises BizTalk

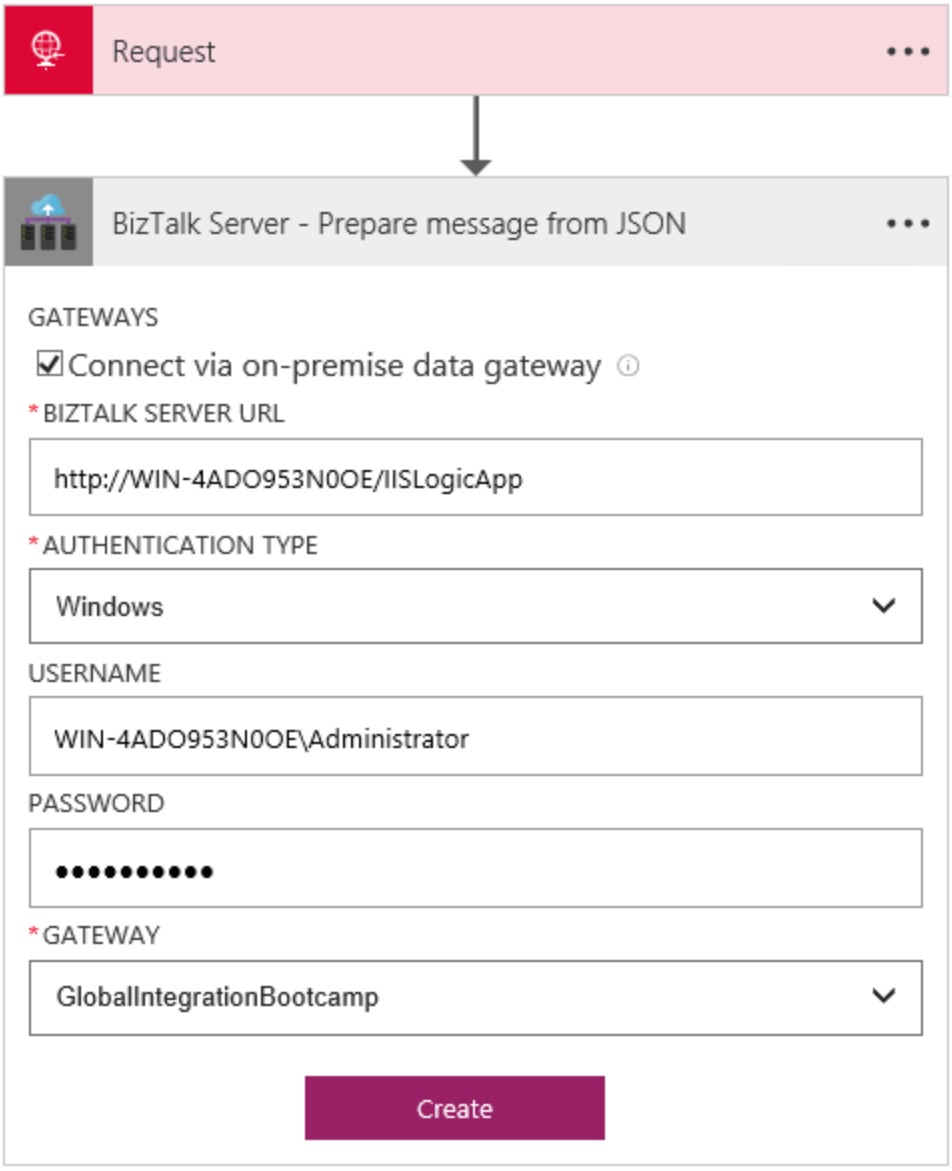
Now that we have set up our BizTalk solution, we will create a new Logic App which will send messages our BizTalk. Open the Azure portal, go to the Logic Apps blade, and create a new Logic App.



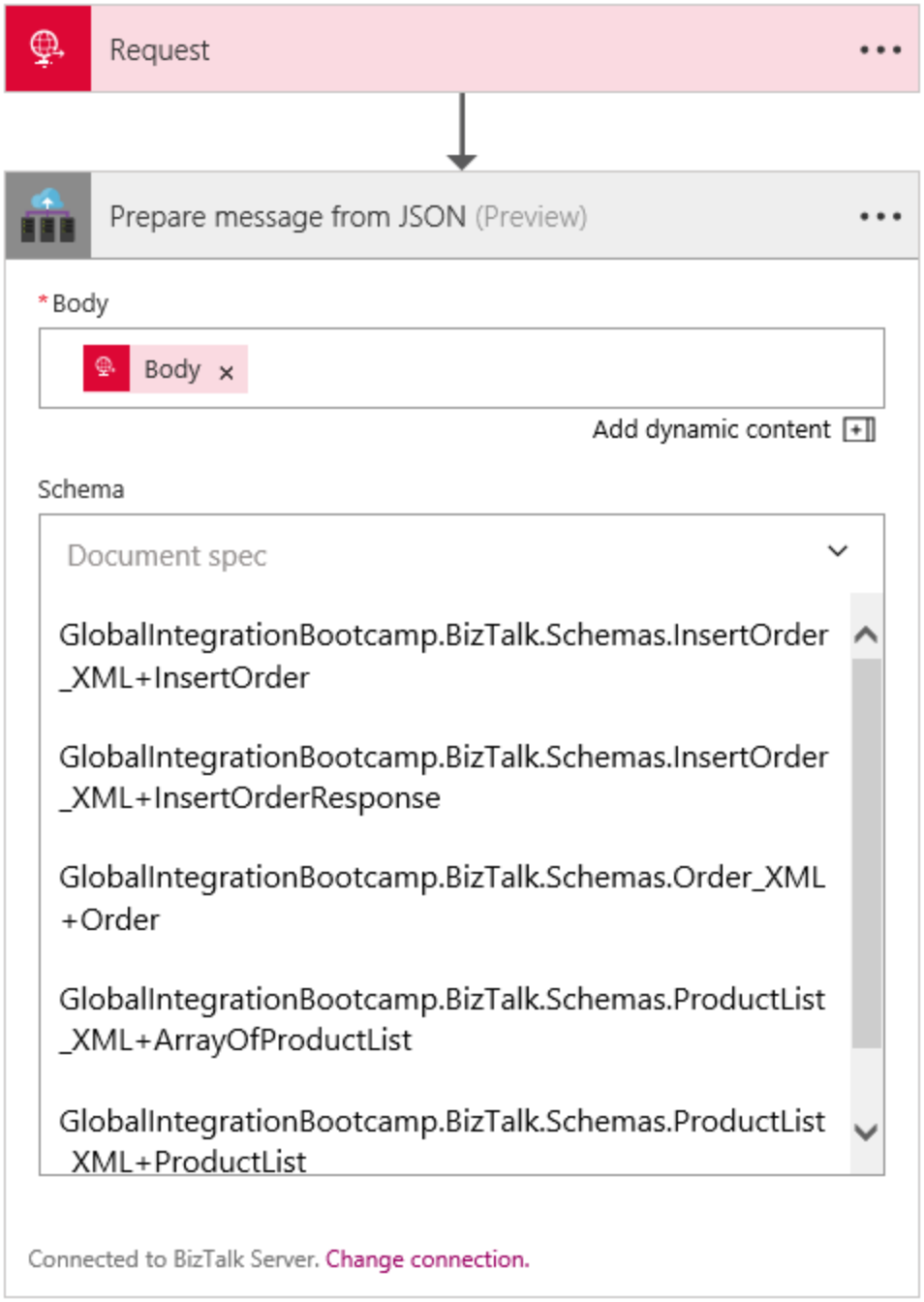
In the designer choose to create a new Logic App from a blank template, and add a **Request** trigger. As we will not access the contents of this message, but instead forward the entire body, it is not needed to set a JSON schema.



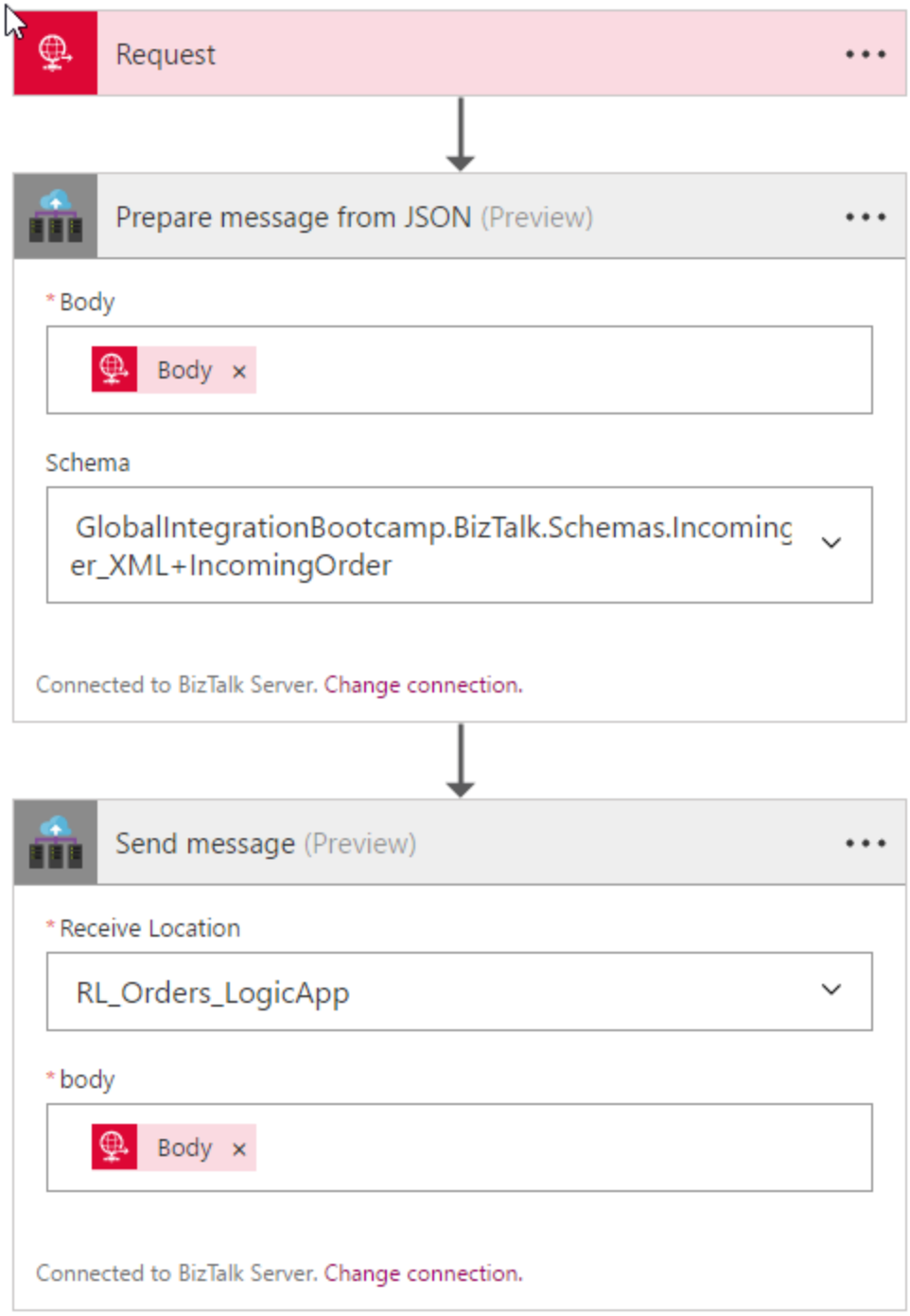
Next add a **Prepare message from JSON** action. You can use the existing gateway from our previous lab here, or create a new gateway.



With the gateway selected, set the body of the message to the incoming body. The BizTalk connector will query our on premise BizTalk server, and retrieve the message types installed.



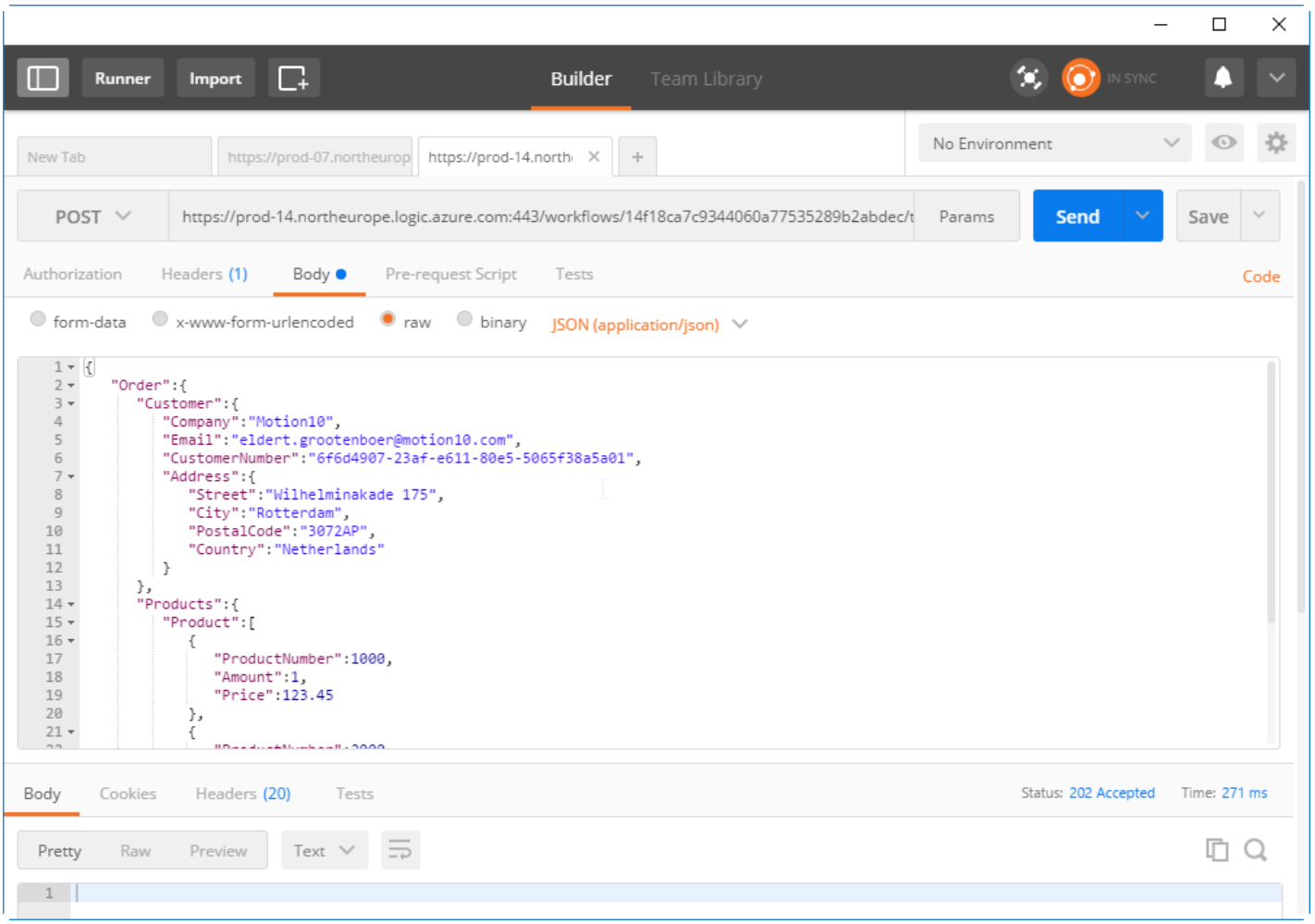
Set the message type to the **IncomingOrder**, and add another action which will send the message to our on-premise BizTalk. Finally save the application, which will generate the URL for the Request trigger.



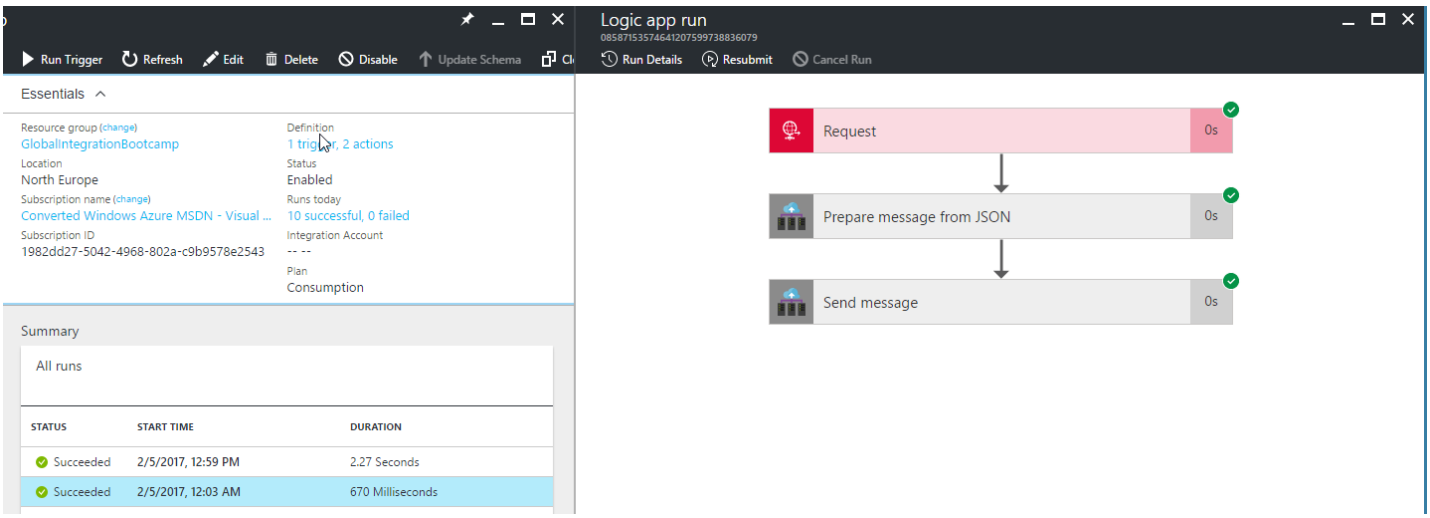
Testing the lab

To test this lab, open PostMan and set the URL to the URL which was created for our Request trigger, use **POST** verb, and set the content type to **application/json**. Use the following message to test this lab.

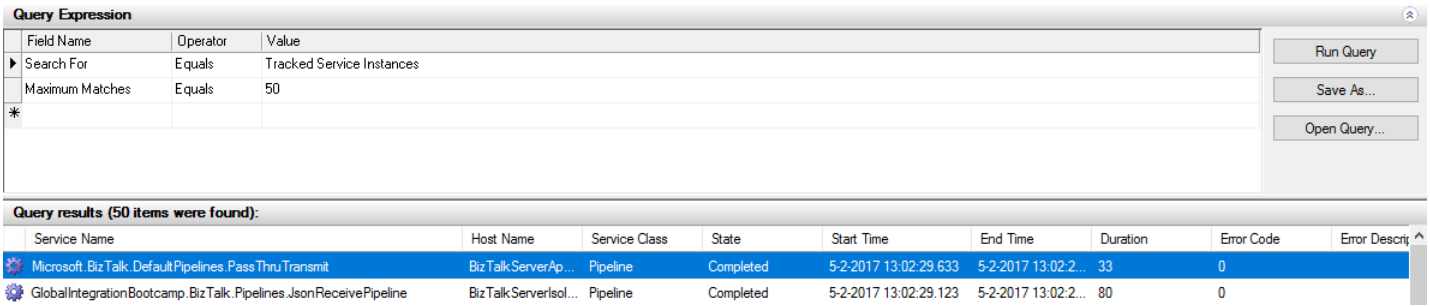
```
{
   "Order":{
      "Customer":{
         "Company":"Motion10",
         "Email":"eldert.grootenboer@motion10.com",
         "CustomerNumber":"6f6d4907-23af-e611-80e5-5065f38a5a01",
         "Address":{
            "Street":"Wilhelminakade 175",
            "City":"Rotterdam",
            "PostalCode":"3072AP",
            "Country":"Netherlands"
         }
      },
      "Products":{
         "Product":[
            {
               "ProductNumber":1000,
               "Amount":1,
               "Price":123.45
            },
            {
               "ProductNumber":2000,
               "Amount":5,
               "Price":456.78
            }
         ]
      },
      "OrderedDateTime":"2016-11-20T14:26:00",
      "TotalInvoiceAmount":2407.35
   }
}
```



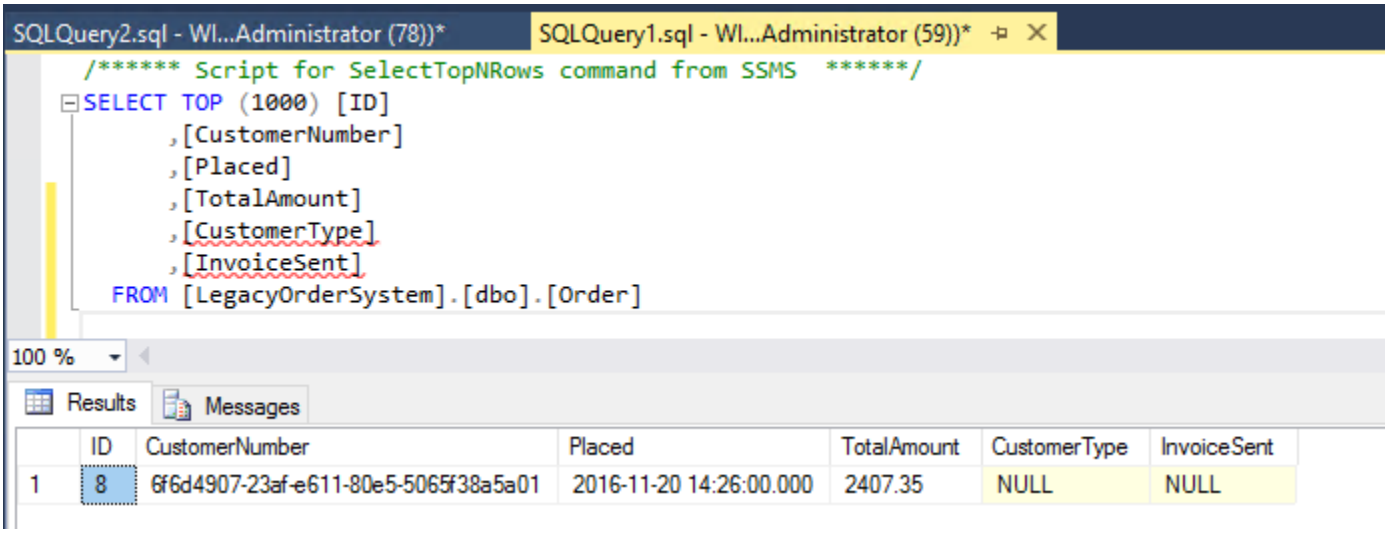
After sending the message, you can check your Logic App in the portal, which should now have run successfully.



Next go to your local server, where you can check BizTalk to see our message has been received.

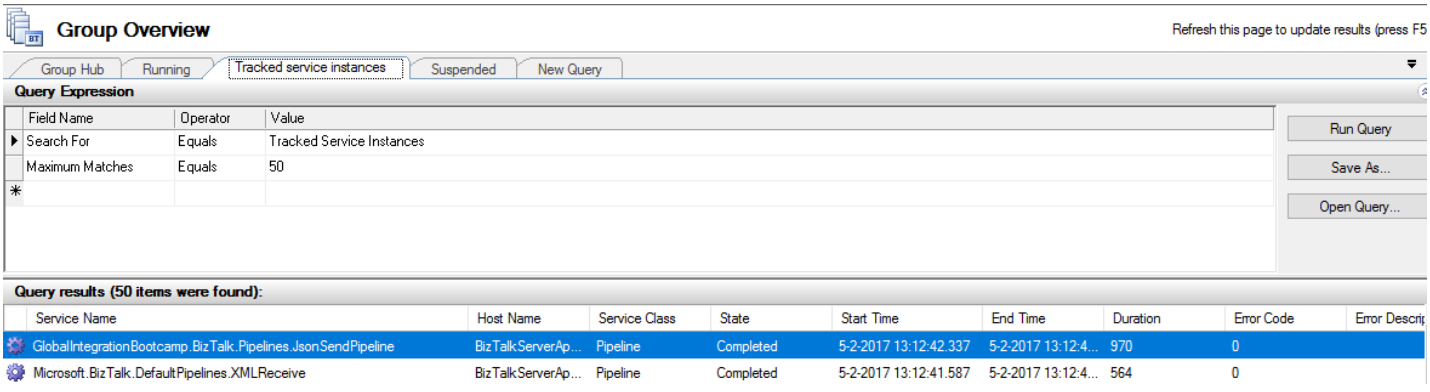


And the order will now be in our database. As you will notice, the CustomerType field is set to **NULL**, we will update this in the next step.

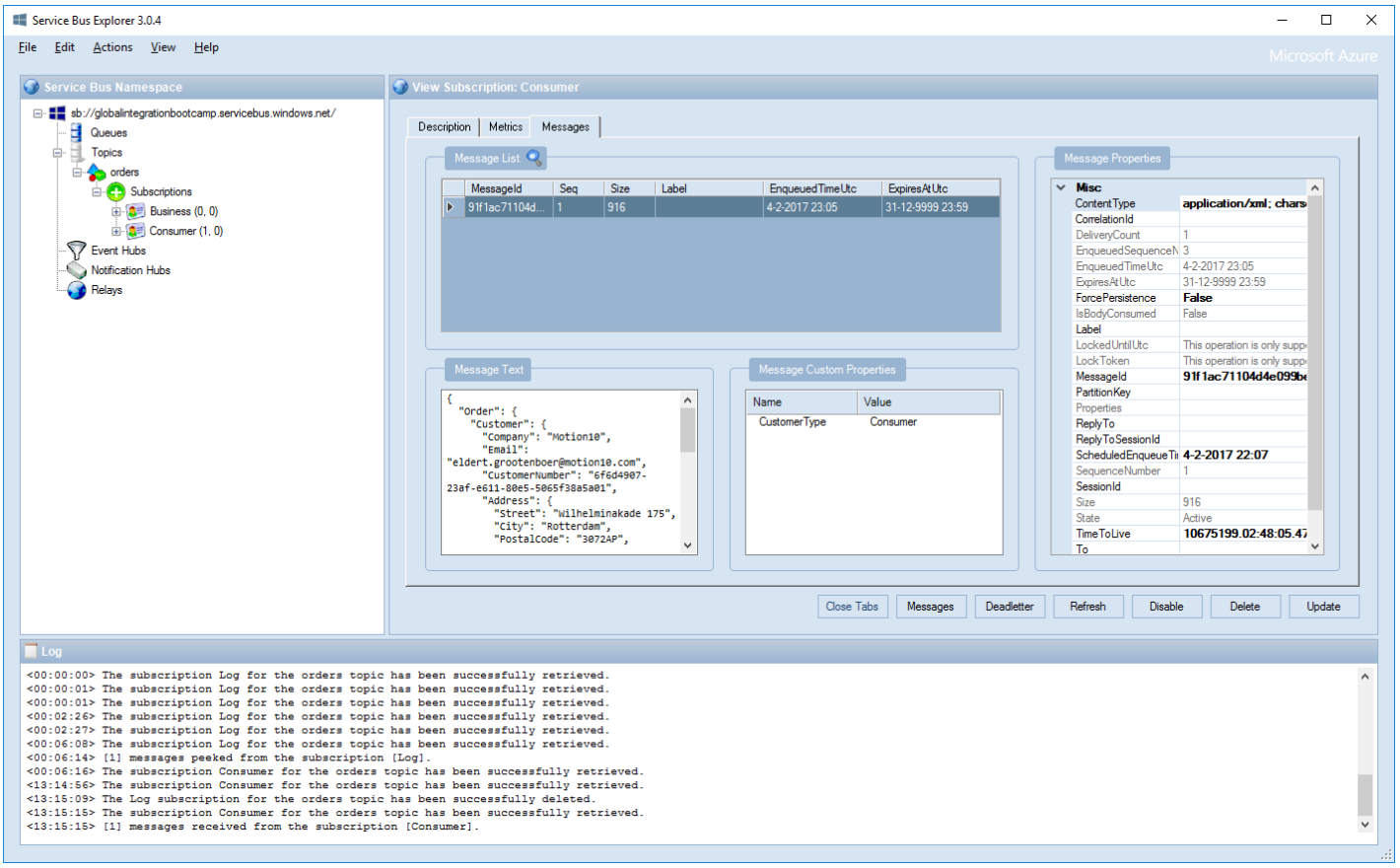


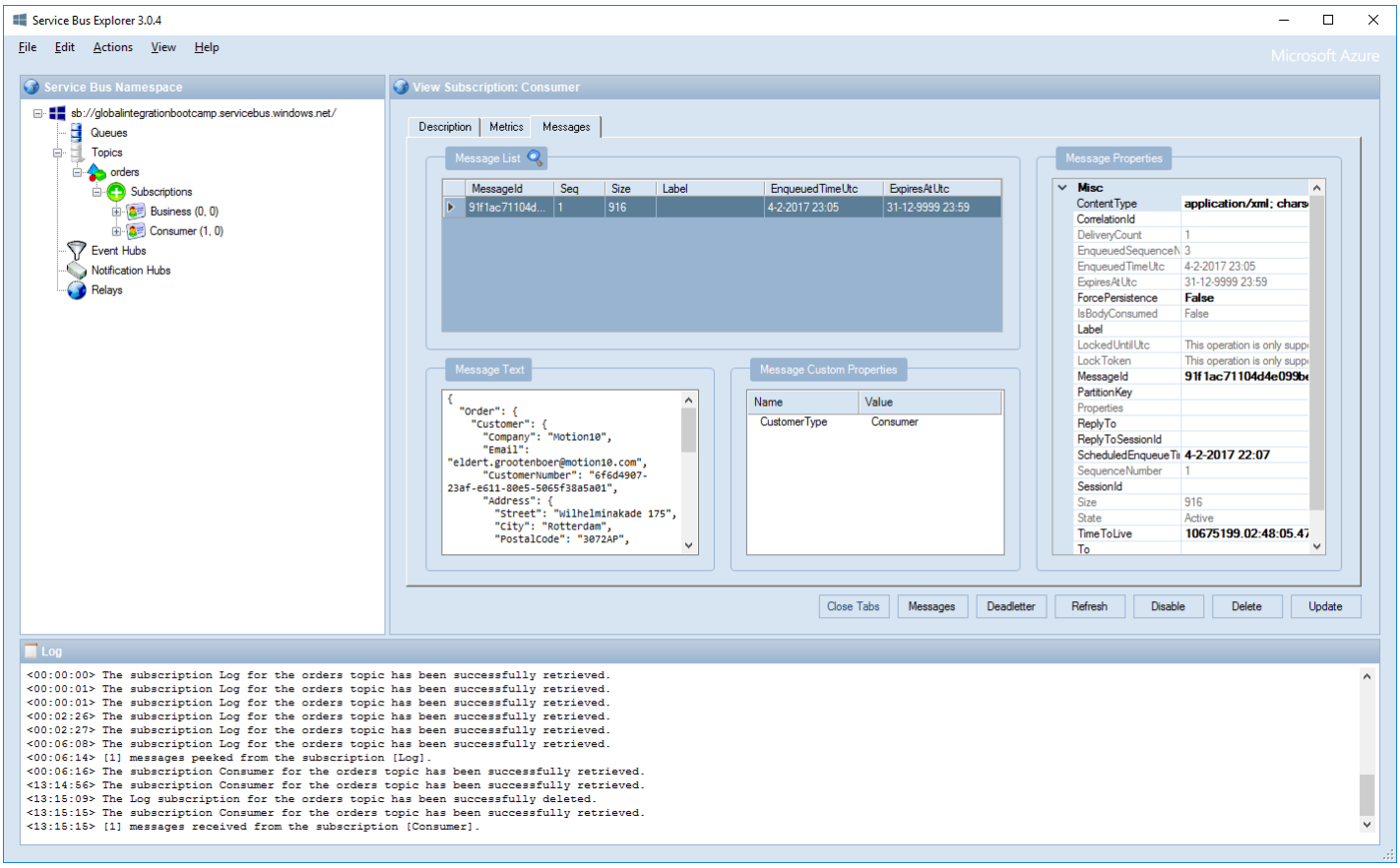
Use the following query to update the order in the database, this will update the CustomerType and trigger the receive port.

UPDATE [LegacyOrderSystem].[dbo].[Order] SET CustomerType = 'Consumer'



We can now check Service Bus Explorer to see the message received on the topic.

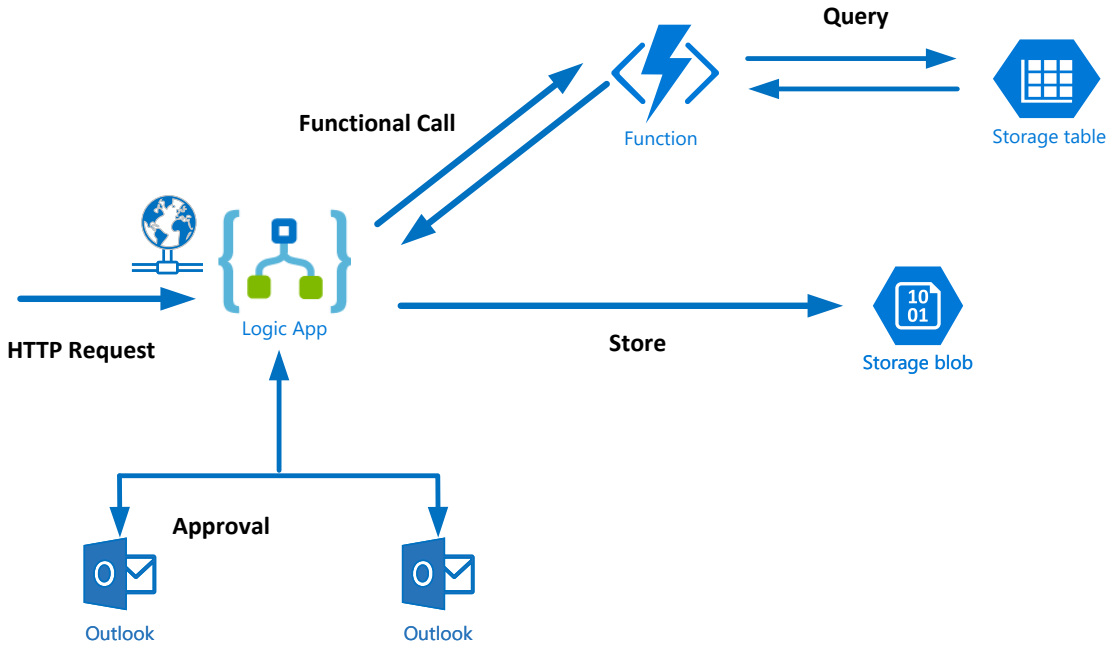




Lab 4 - Logic Apps + Azure Functions

Objective

In this fourth lab, we will be receiving the orders from the business customer's topic in a new **Logic App**, and check the total amount of the invoice. In case the customer placed a large order (over **$50000**), we will create a task for one of our sales employees to contact the customer to verify the order. In case the order is correct, the invoice will be emailed to the customer. The **Logic App** will then call a **function**, in which we will check a **storage table** to determine how much discount the customer will be given (based on the total order amount), and finally will place a file on **blob storage**, which will be used by an employee to refund the customer.



Prerequisites

- Azure Subscription
- Azure Storage Explorer: http://storageexplorer.com/
- Google Chrome Postman
- Outlook.com account
- TableCustomerDiscount.csv which can be downloaded from here

Steps

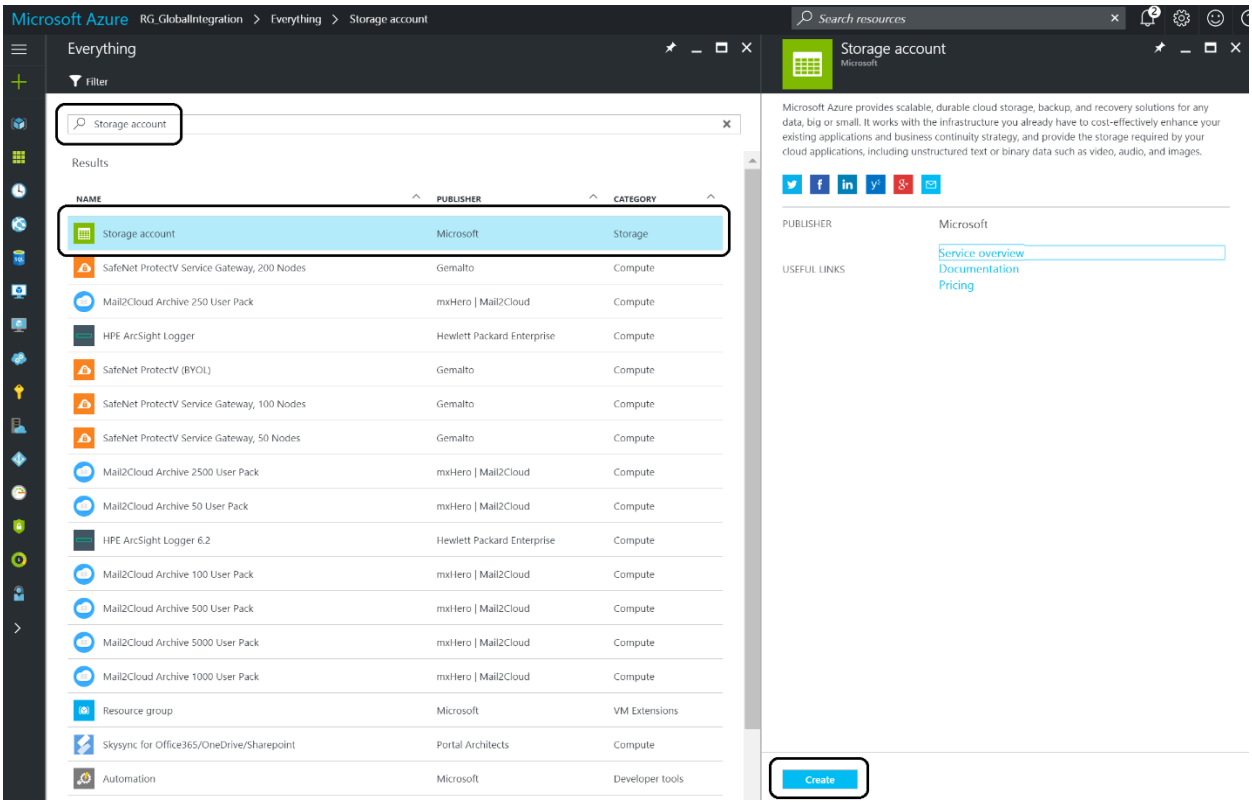
To build the solution in this lab you have to follow the steps described in this section. From a high level view the steps are:

1. Create Storage Account
2. Create Storage Blob Container
3. Create Storage Table
4. Provision the Function App
5. Build a custom function
6. Provision a Logic App
7. Build Logic App Definition
8. Test the Solution

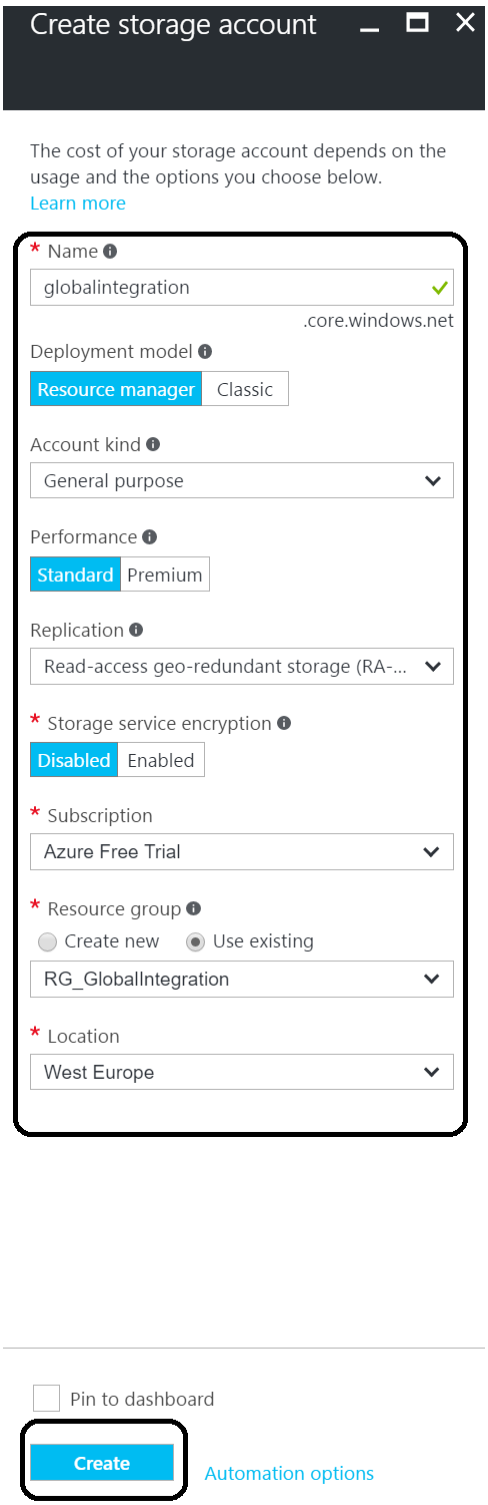
Create Storage Account

The first step in building the solution in this lab is to provision a storage account in Azure. We will be needing storage for setting up our reference table (Table Storage) and storing the order request message in Blob Storage.

1. Go to the Azure Portal: https://portal.azure.com/
2. Login into the Azure portal with your account.
3. In the Market Place enter storage account and select it from the list as shown below.



4. Click **Create**.

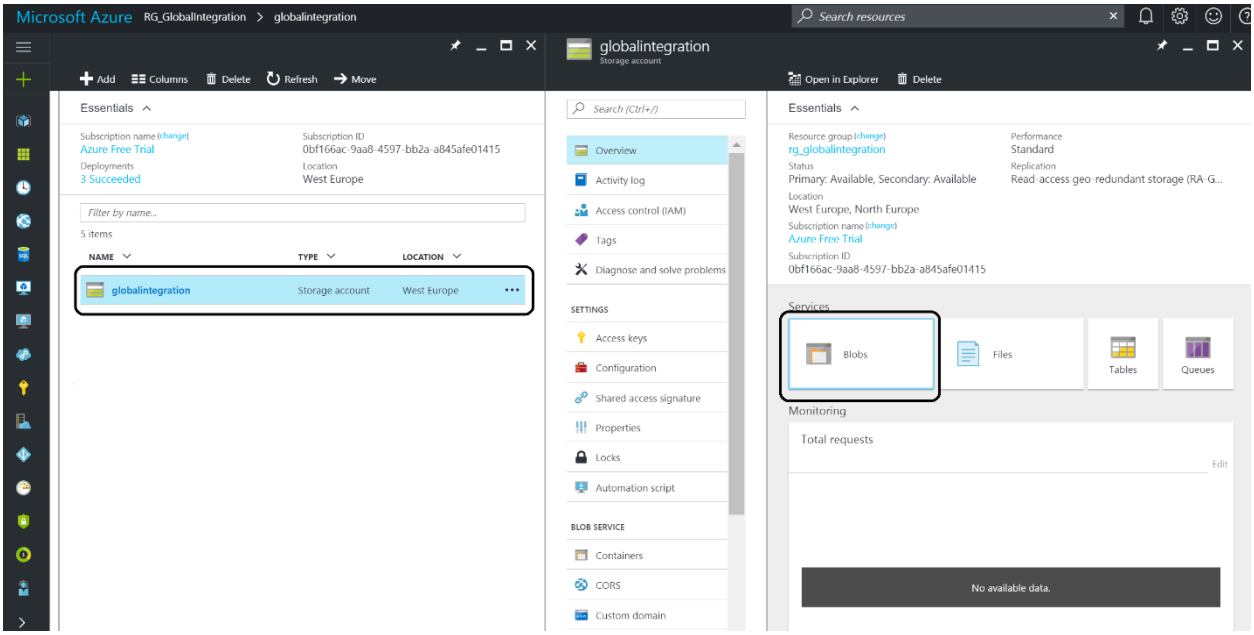


5. Specify a **name**, a **Resource Group** (you can create a new one here if you haven't created a resource group yet) and a **location**. Subsequently, click on **Create**.

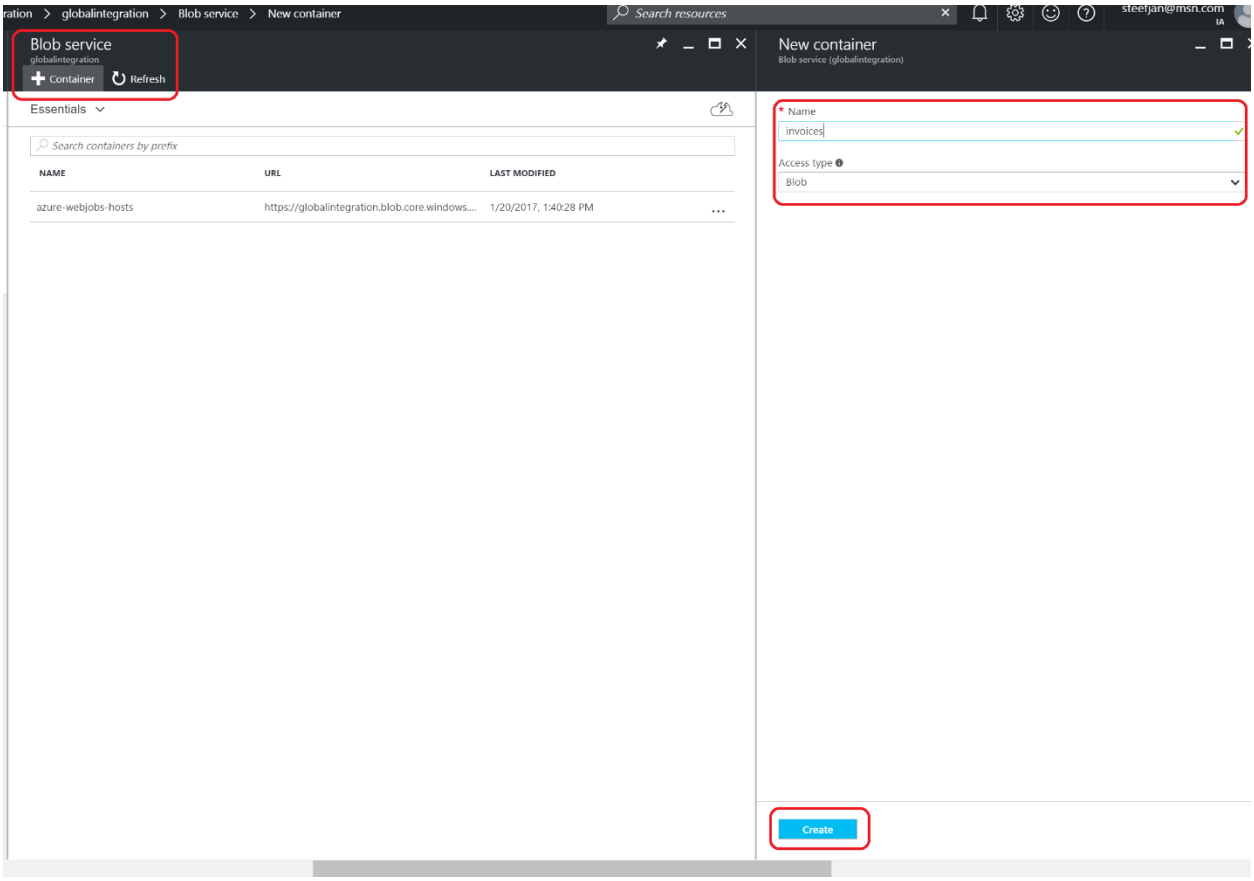
Create Strorage Container

Once the storage account has been provisioned you can navigate to it and click on it.

1. In the storage account click on **Blobs**.



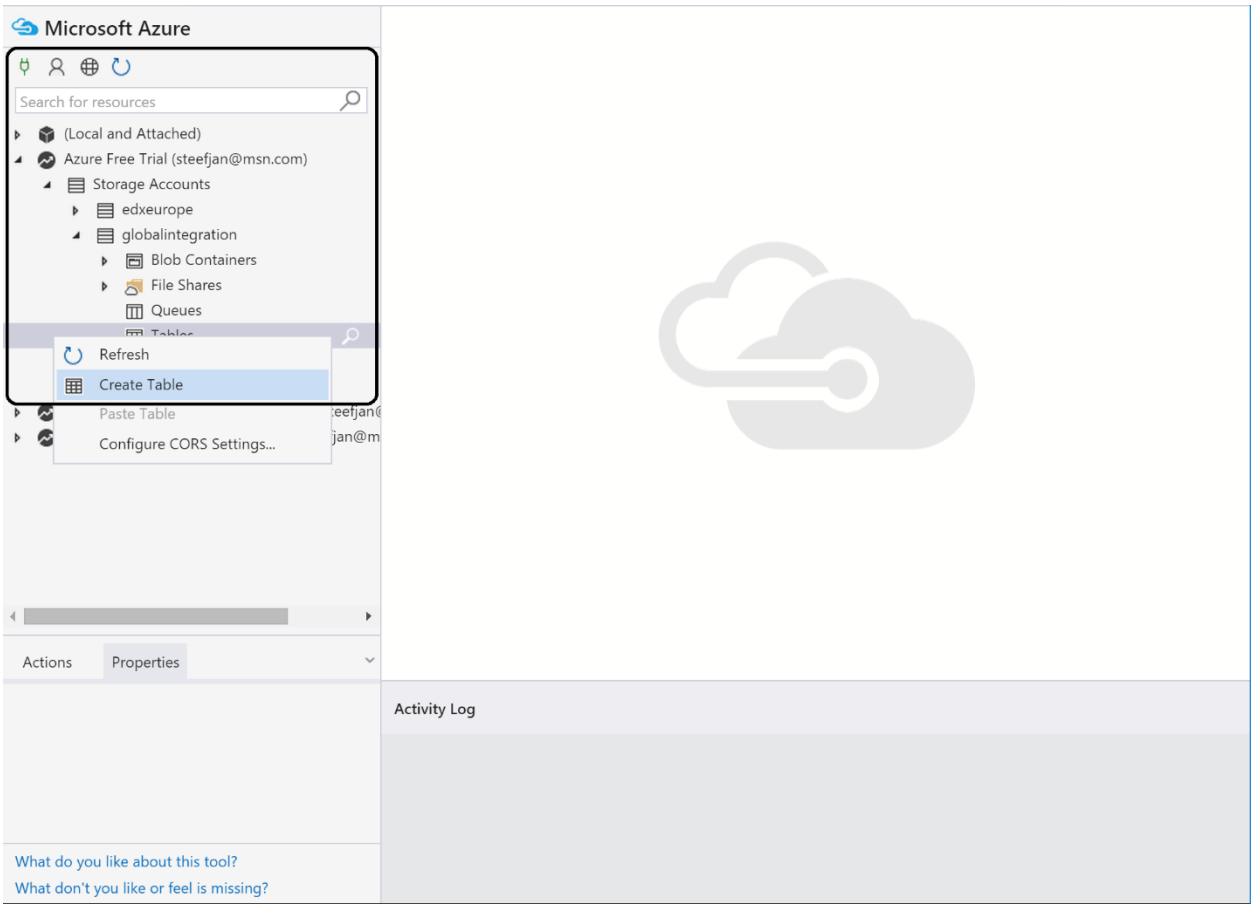
2. Click on **+ Container** and specify the name and Access Type: **Blob**.



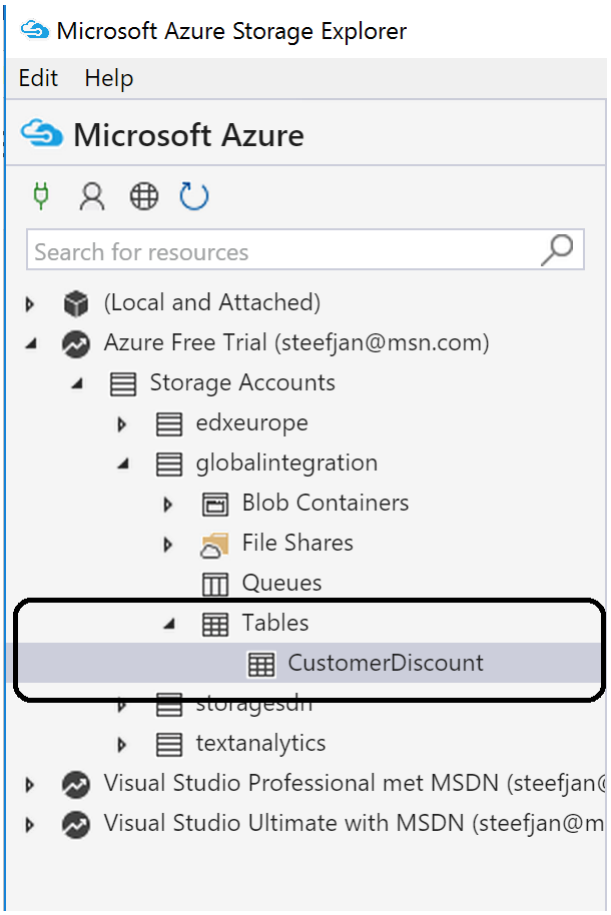
Create Storage Table

To create a storage table will use the **Azure Storage Explorer**, which can be downloaded from http://storageexplorer.com/.

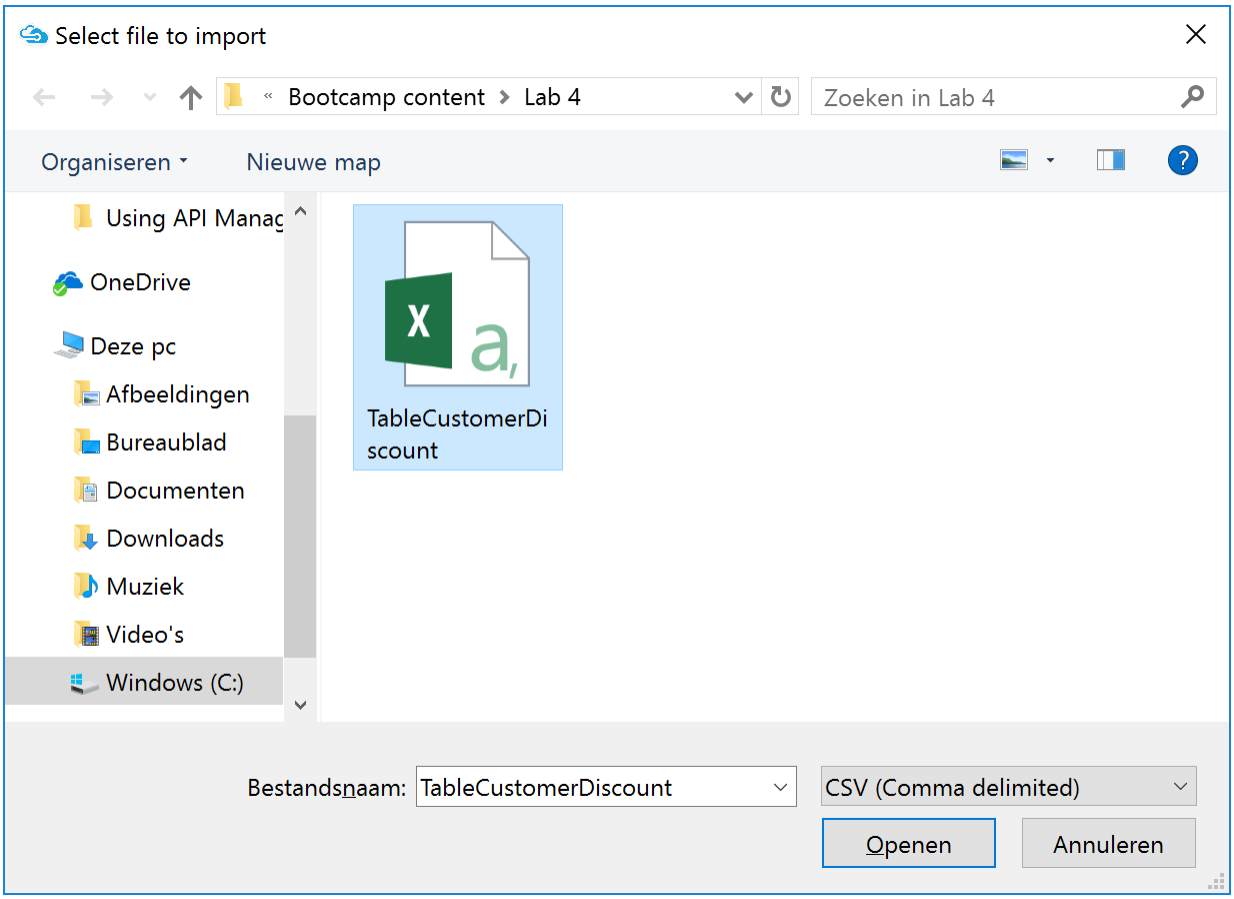
1. Install the tool, and login into your subscription.
2. Navigate to your storage account.
3. Select Tables
4. Right click Tables and click Create Table.



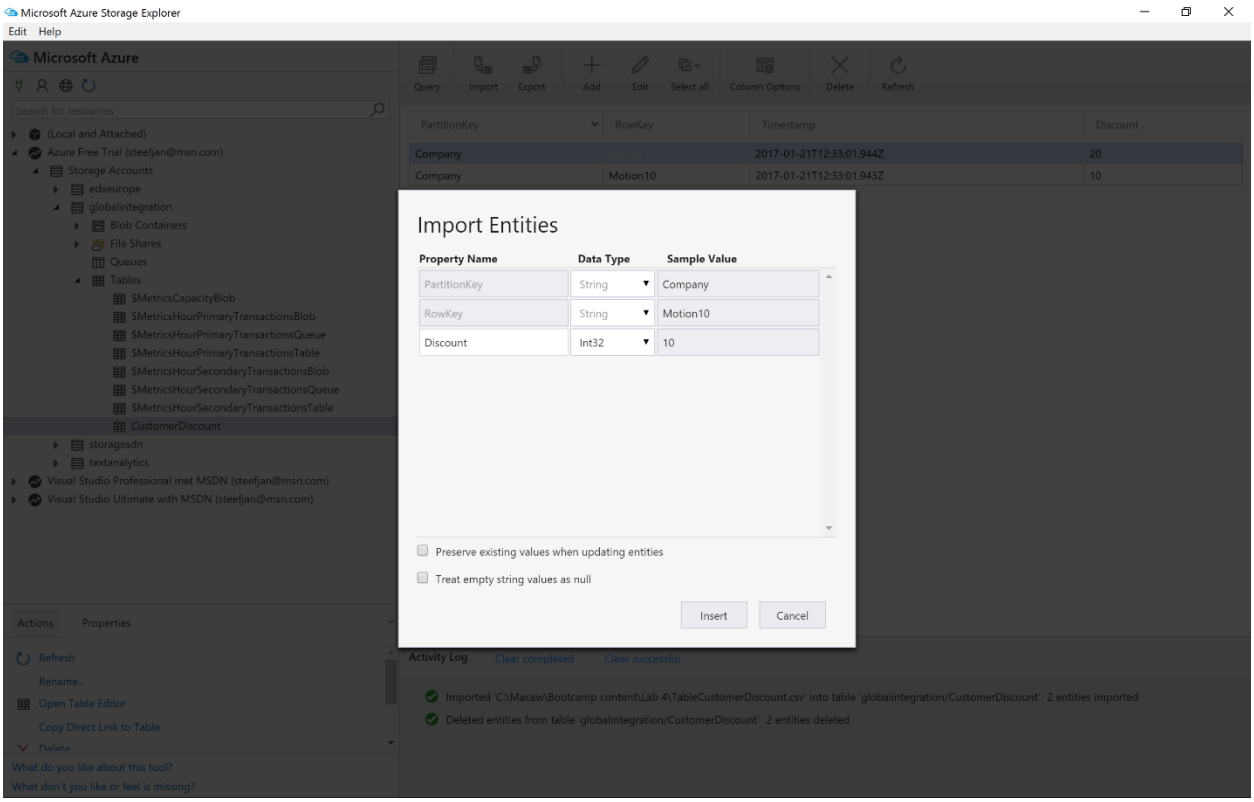
5. Specify a name for the table.



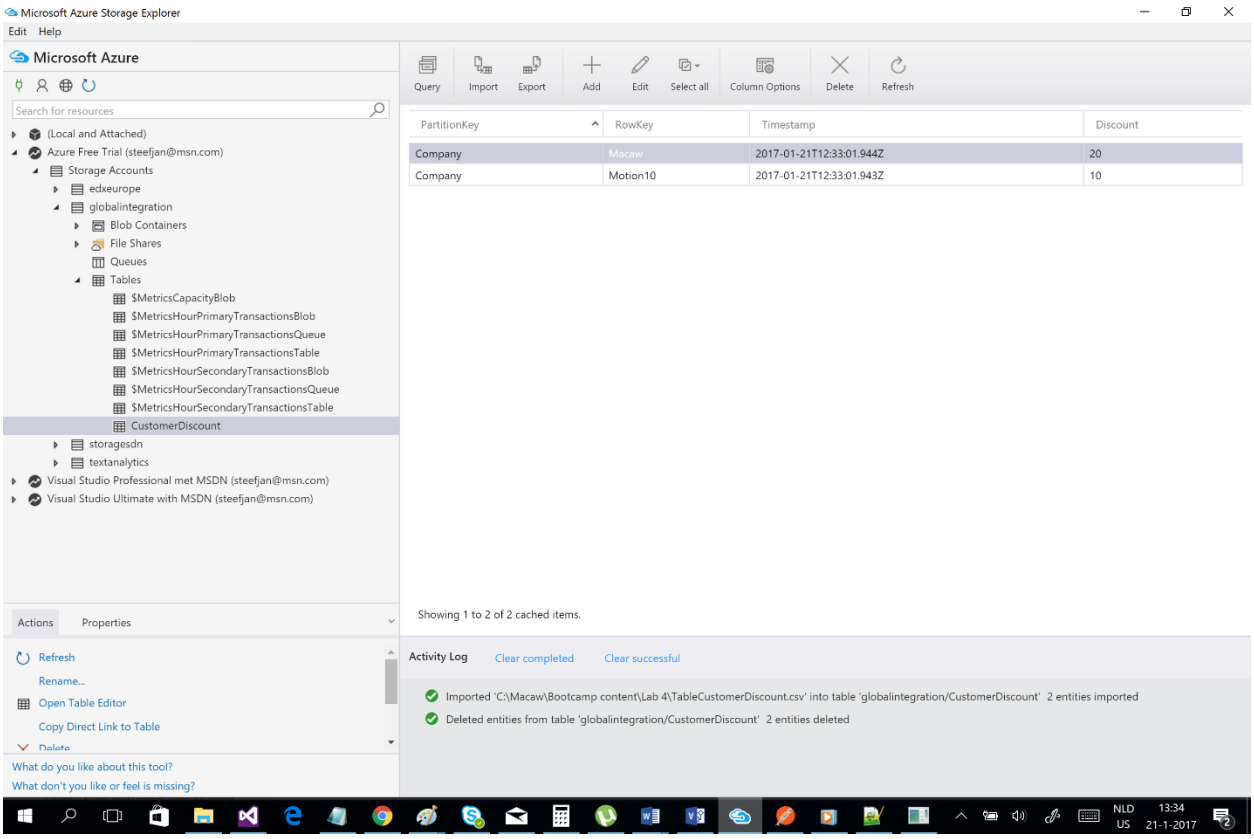
6. Select the **table**.
7. Click on **Import Entities** from file.
8. Select the **TableCustomerDiscount.csv** you downloaded previously.



9. You will see a window popup like below.



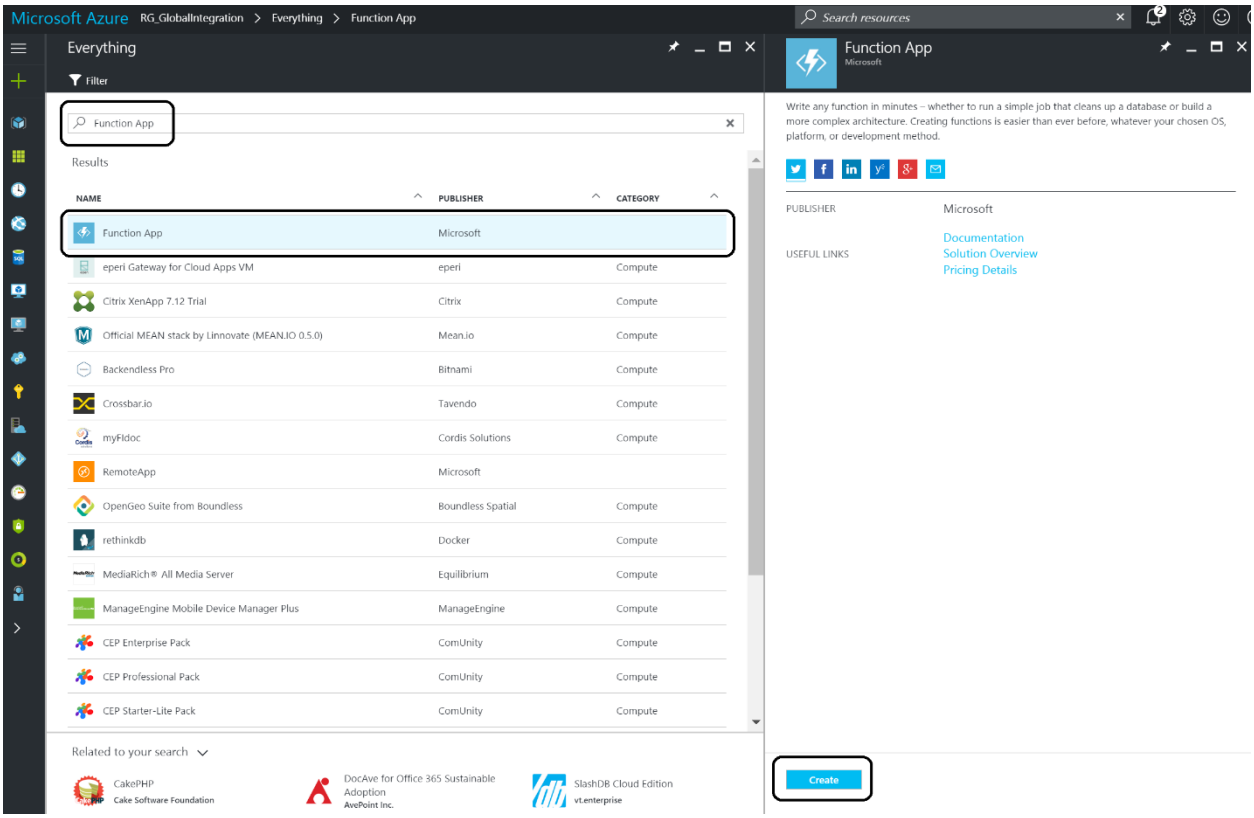
10. Click on **Insert**.
11. **Table** will be loaded with data.



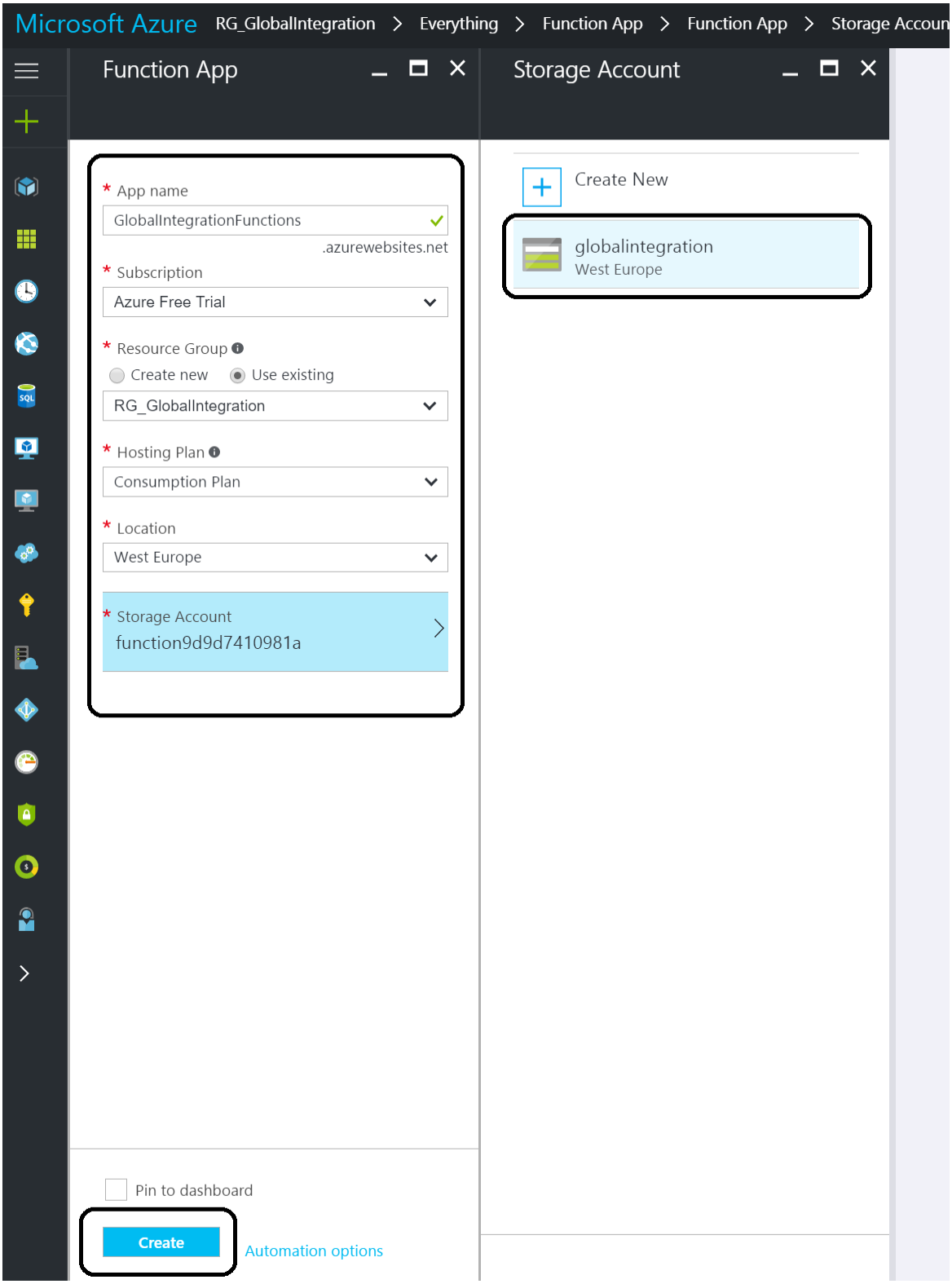
Provision the Function App

The third step in building up our solution is provision a **Function App**. A **Function App** is a container for your functions. The functions can be built with a browser using either C#, JavaScript, or some of the other languages. The code you create can be run and tested in the function app environment.

1. In the Market Place enter **Function App**.
2. Select the **Function App** and **Click Create**.



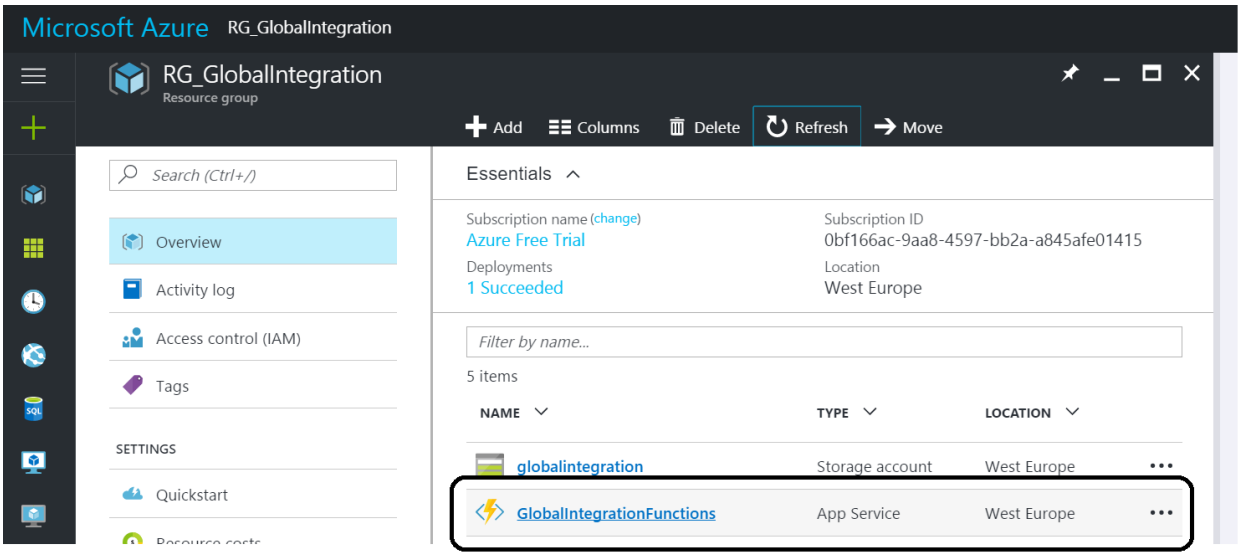
3. Specify the details are shown below. You can select the storage account your created in the previous steps.



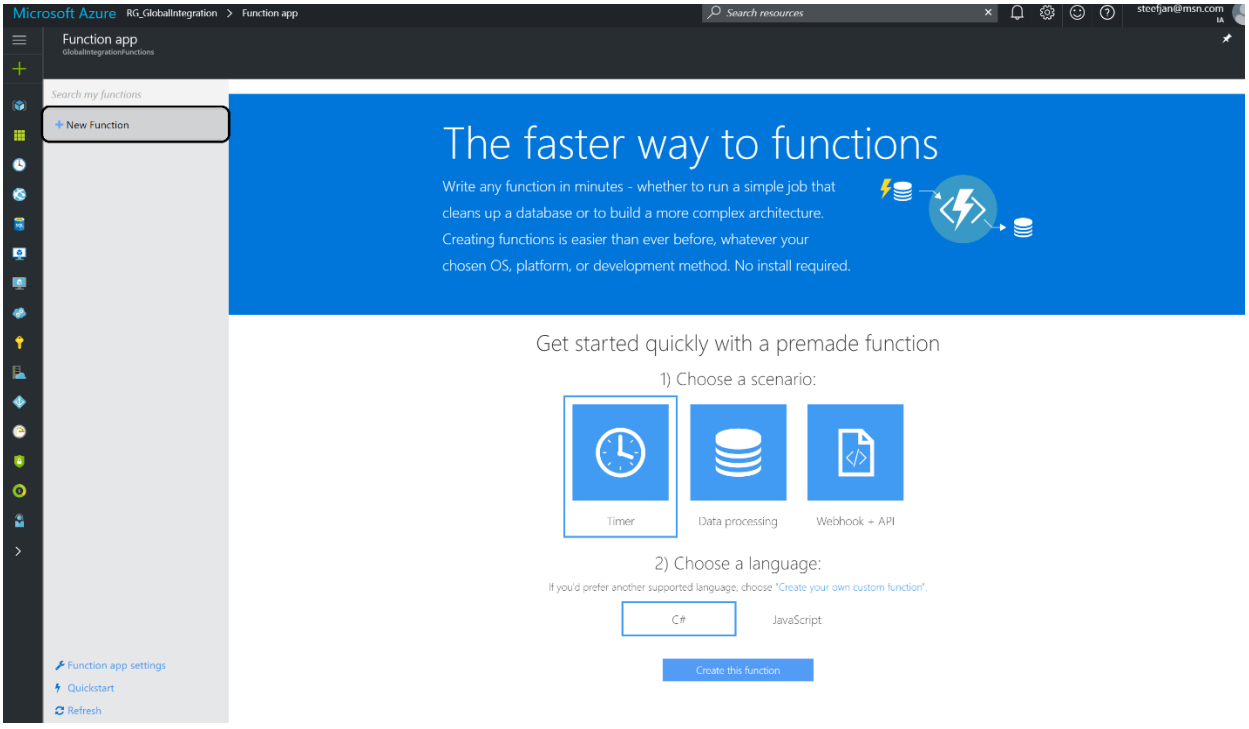
Building a Function

Once the function app is provisioned you can add function to it, build and test it.

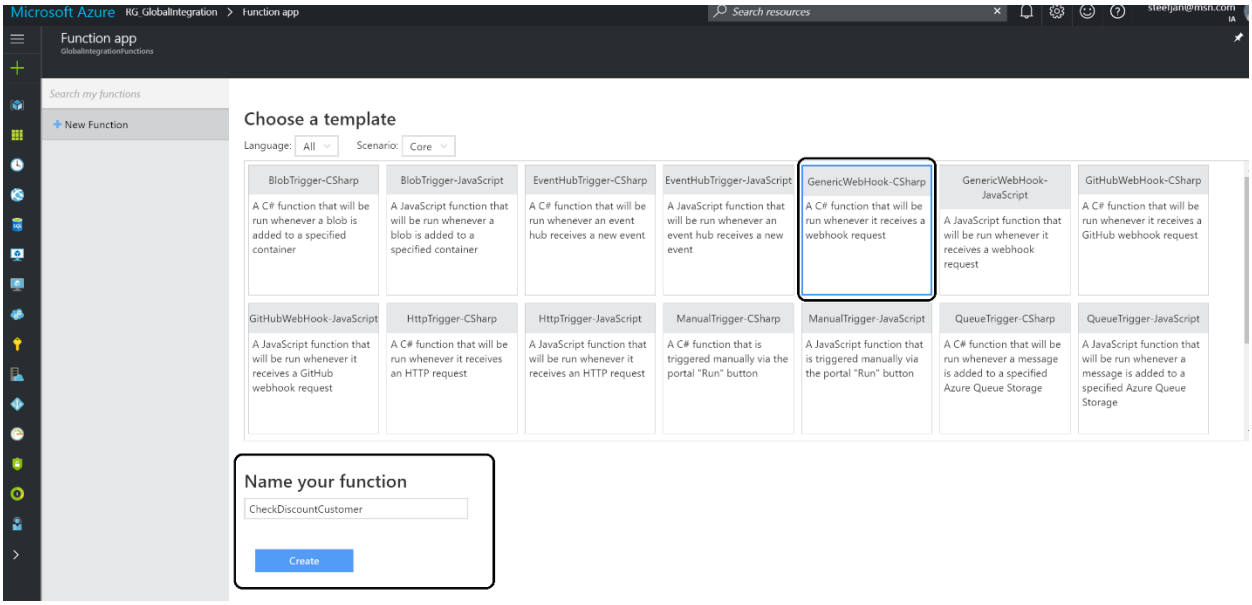
15. Select the **Function App**.



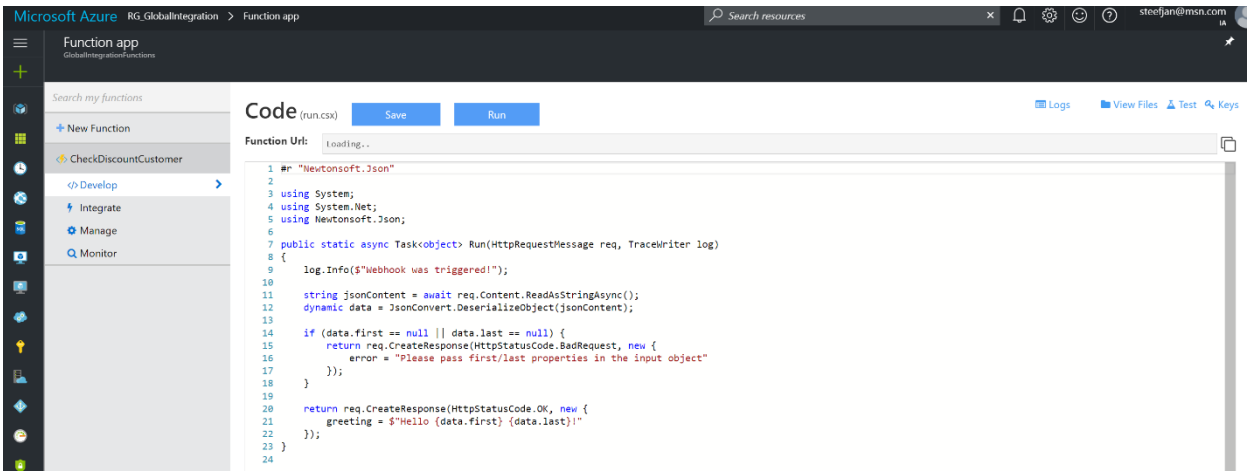
16. In the function app click **+ New Function**.



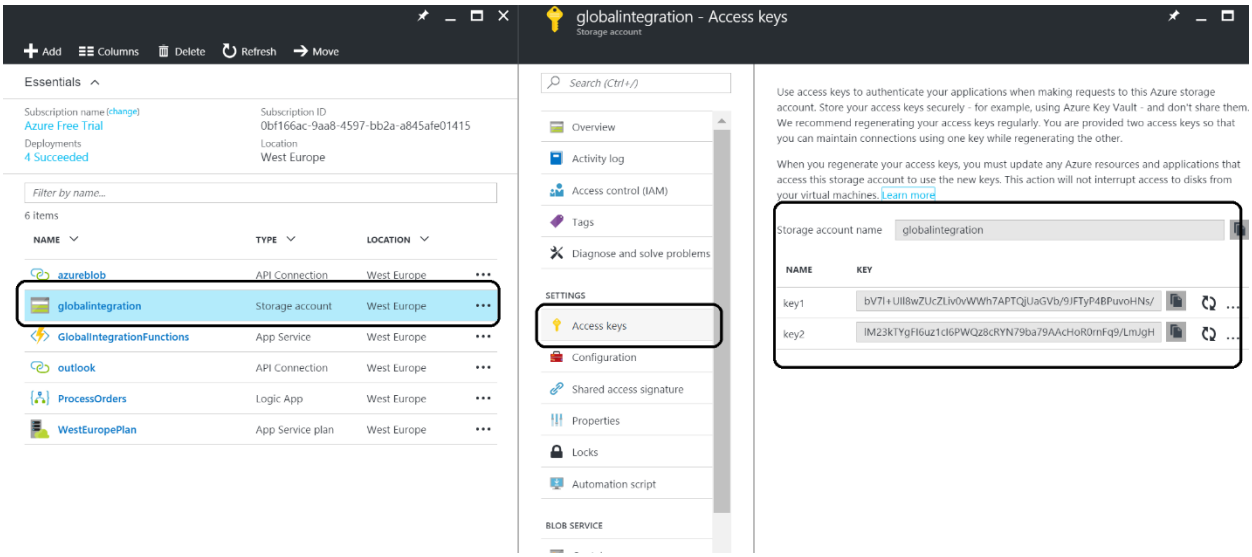
17. Select **GenericWebHook-CSharp** and specify a name for your function.



18. Now a new pane will appear with some default code.



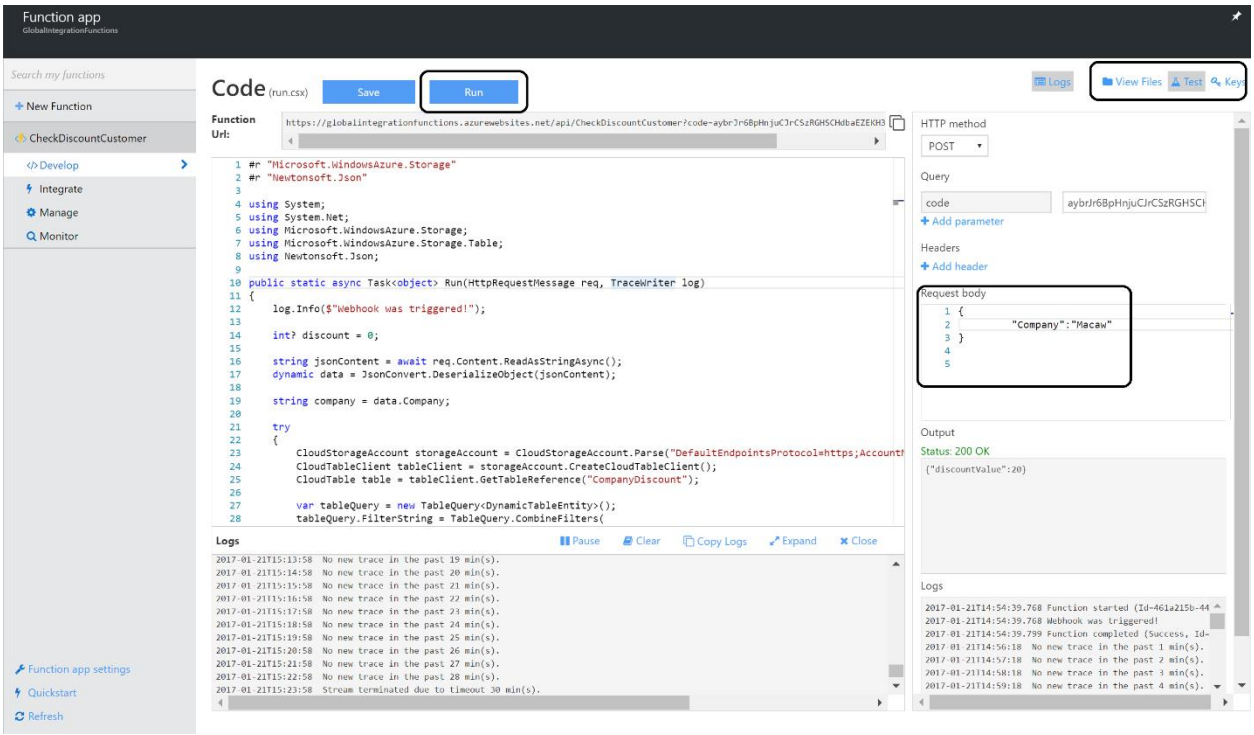
19. You will change code with the code below. This code will connect to your storage account, therefore you'll need to know your account name and key.




```
#r "Microsoft.WindowsAzure.Storage"
#r "Newtonsoft.Json"

using System;
using System.Net;
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.Table;
using Newtonsoft.Json;

public static async Task<object> Run(HttpRequestMessage req, TraceWriter log)
{
    log.Info($"Webhook was triggered!");

    int? discount = 0;

    string jsonContent = await req.Content.ReadAsStringAsync();
    dynamic data = JsonConvert.DeserializeObject(jsonContent);

    string company = data.Company;

    // Here we will connect to our storage account
    try
    {
        CloudStorageAccount storageAccount =
CloudStorageAccount.Parse("DefaultEndpointsProtocol=https;AccountName=<your
accountname>;AccountKey=<your account key>");
        CloudTableClient tableClient =
storageAccount.CreateCloudTableClient();
        CloudTable table = tableClient.GetTableReference("CustomerDiscount");

       //Create a filter expression
        var tableQuery = new TableQuery<DynamicTableEntity>();
        tableQuery.FilterString = TableQuery.CombineFilters(
                TableQuery.GenerateFilterCondition("PartitionKey",
QueryComparisons.Equal, "Company"),
                TableOperators.And,
```

```
                TableQuery.GenerateFilterCondition("RowKey",
QueryComparisons.Equal, company));

        // Loop through the results, displaying information about the entity.
        foreach (DynamicTableEntity entity in table.ExecuteQuery(tableQuery))
        {
            var item = entity.Properties;
            discount = item["Discount"].Int32Value;
        }
    }
    catch (Exception ex)
    {
        log.Info($"Webhook exception :" + ex.Message);
    }

    return req.CreateResponse(HttpStatusCode.OK, new
    {
        discountValue = discount
    });
}
```

20. Hit **Save** in the top bar.
21. Click on **Tests** in right top corner.

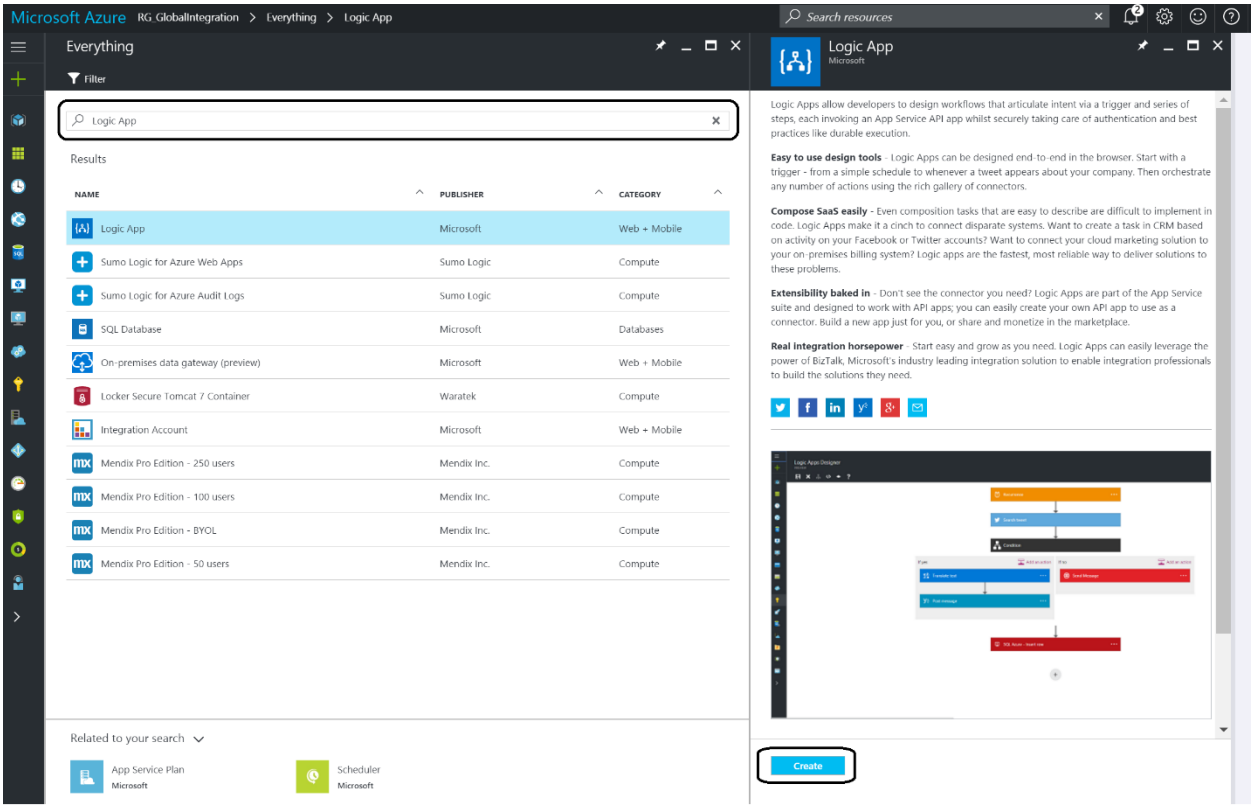


22. Change **Request body** to what is in the picture above.
23. Click **Run**.
24. Explore **Logs** and the **Output**.

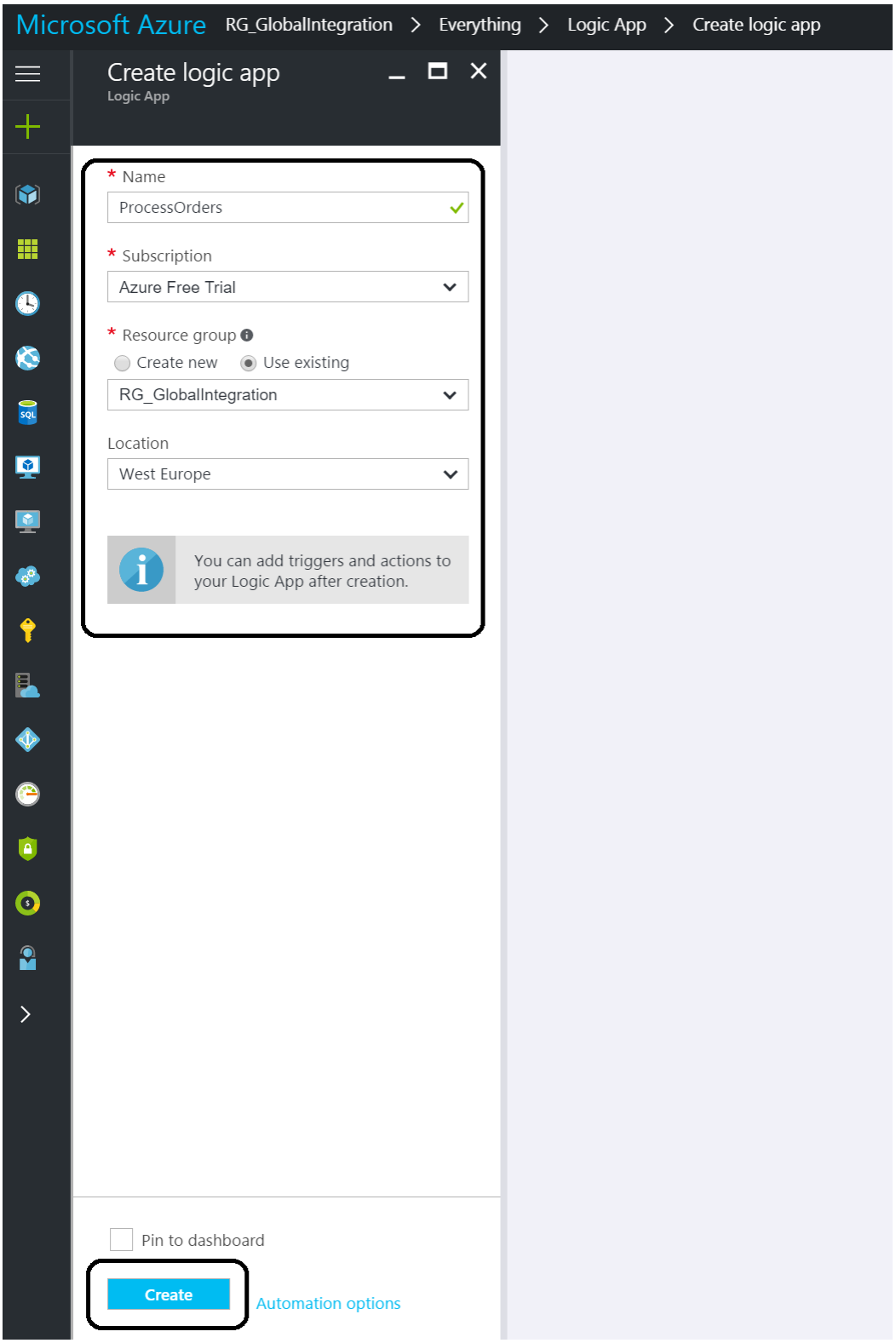
Provision a Logic App

The following steps describe how to provision a Logic App.

1. In the new Azure Portal click the **+**, navigate to **Web + Mobile** and subsequently click **Logic App**. Or in the search in the marketplace type Logic App. It's easy to find the Azure Service in the portal to provision.

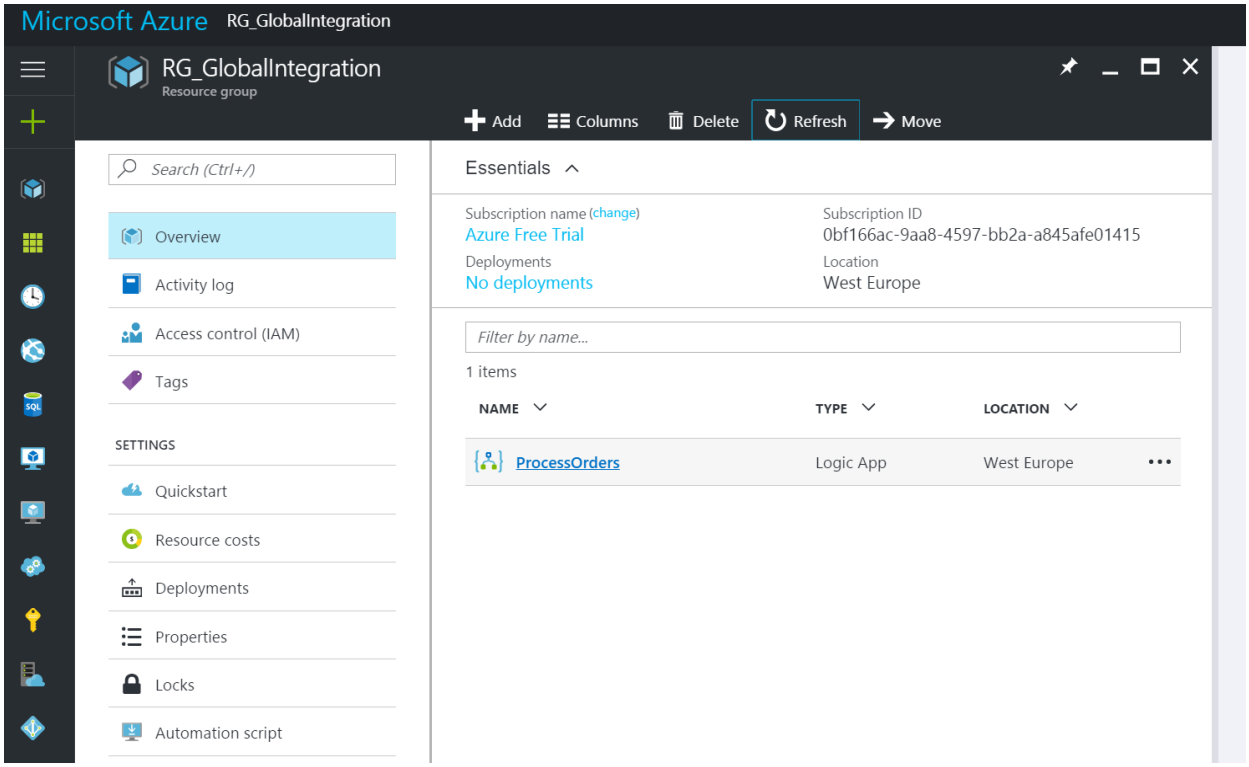


2. The next step is to specify name of your **Logic App**, the subscription (in case you would have multiple subscriptions), the resource group the Logic App should belong to and location i.e. which Azure datacenter. And subsequently decide if you want to see the Logic App on the dashboard or not and click **Create**. You wait until the Logic App is provisioned to be able to proceed with building up the flow (trigger and actions).



3. Once the **Logic App** is provisioned, you have setup a service, also known as iPaas (integration Platform as a Service). The **Logic App** is fully managed by Microsoft Azure and the only thing you

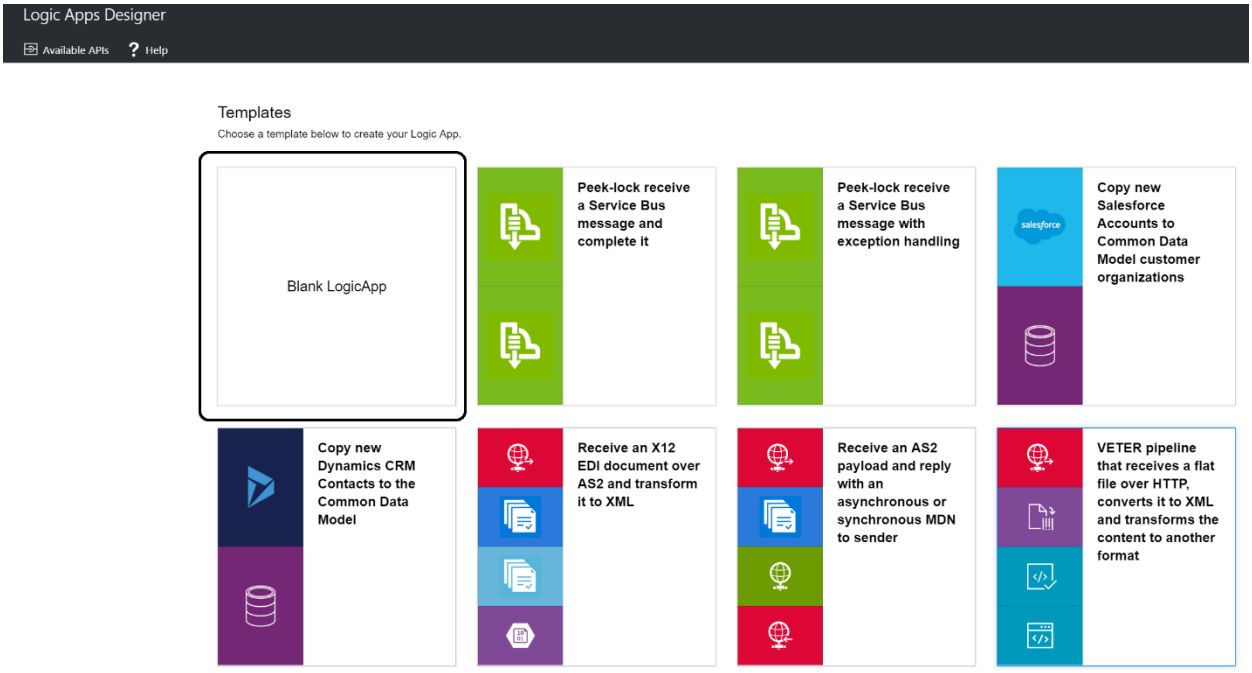
need to do is add the logic i.e. specify the trigger and defining the actions, which we will do in the next step.



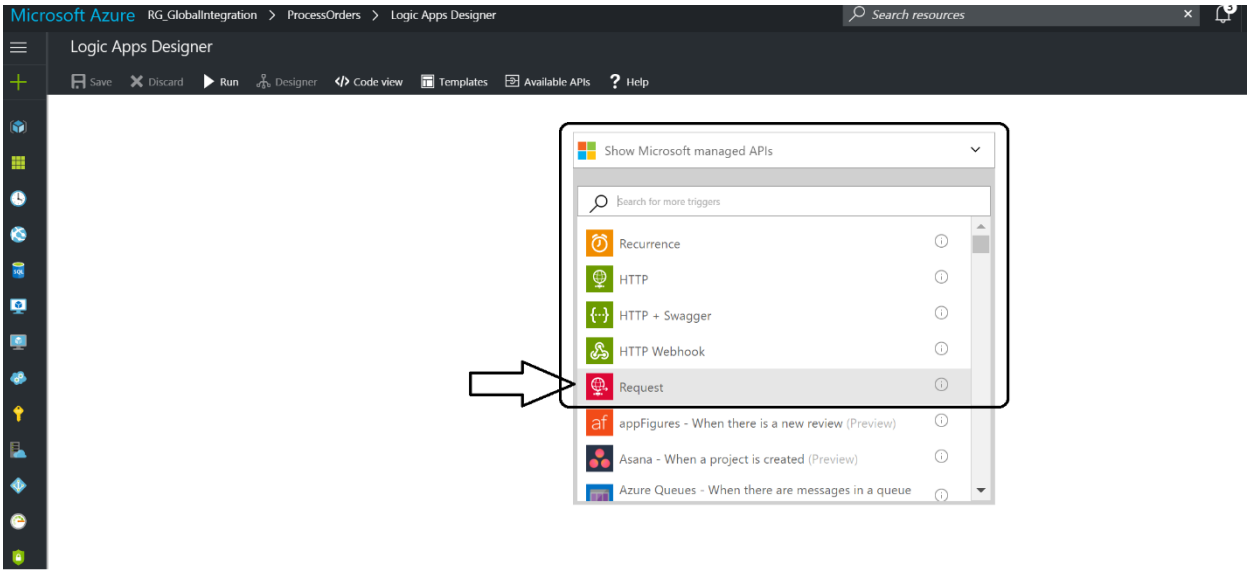
Building a Logic App Definition

In the following steps, we will build our Logic App to support our solution.

1. Once the **Logic App** is provisioned, you can click on it, choose **Blank Logic App** and you will have access the **Logic Apps designer**.



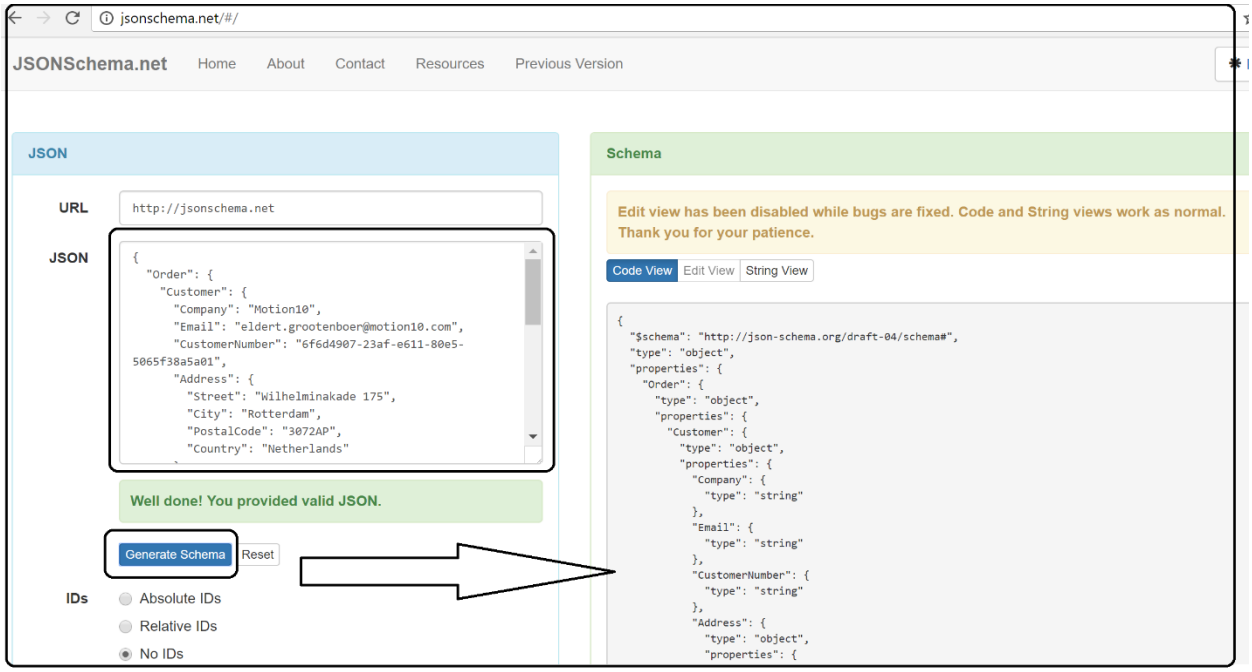
2. In the Logic App designer, you can add a trigger. You select various triggers like **HTTP**, **recurrence**, **WebHook**, etcetera (see Workflow Actions and Triggers). In this lab, it's the **HTTP trigger** (Request) for testing purposes (or the **Service Bus** if you want to connect from the previous lab), which you have to provide with a schema of the payload it can accept/expect.



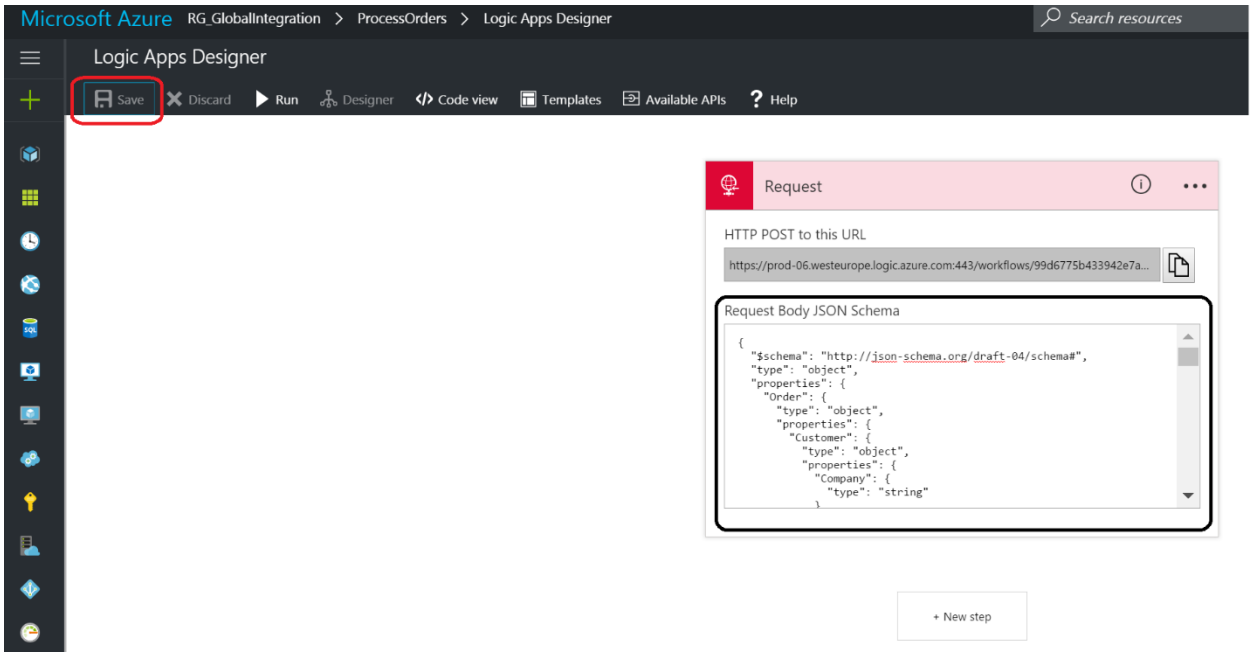
3. The schema can be generated directly from the designer, or using JsonSchema.net, a tool that automatically generates JSON schema from JSON according to the IETF JSON Schema Internet Draft Version 4. JSON Schema will be automatically generated in three formats: editable, code view, and string. Paste the code below in the JSON window as shown below:

```
{
   "Order":{
      "Customer":{
         "Company":"Motion10",
         "Email":"eldert.grootenboer@motion10.com",
         "CustomerNumber":"6f6d4907-23af-e611-80e5-5065f38a5a01",
         "Address":{
            "Street":"Wilhelminakade 175",
            "City":"Rotterdam",
            "PostalCode":"3072AP",
            "Country":"Netherlands"
         }
      },
      "Products":{
         "Product":[
            {
               "ProductNumber":1000,
               "Amount":1,
               "Price":123.45
            },
            {
               "ProductNumber":2000,
               "Amount":5,
               "Price":456.78
            }
         ]
      },
      "OrderedDateTime":"2016-11-20T14:26:00",
```

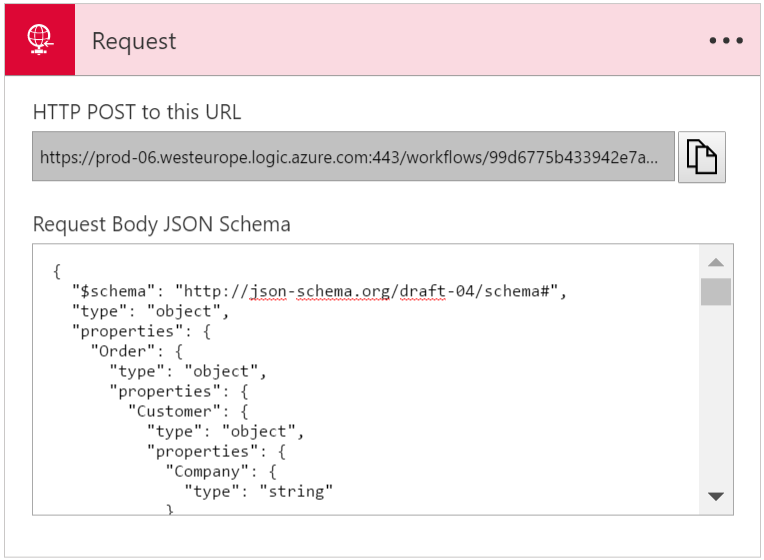
```
        "TotalInvoiceAmount":2407.35
    }
}
```



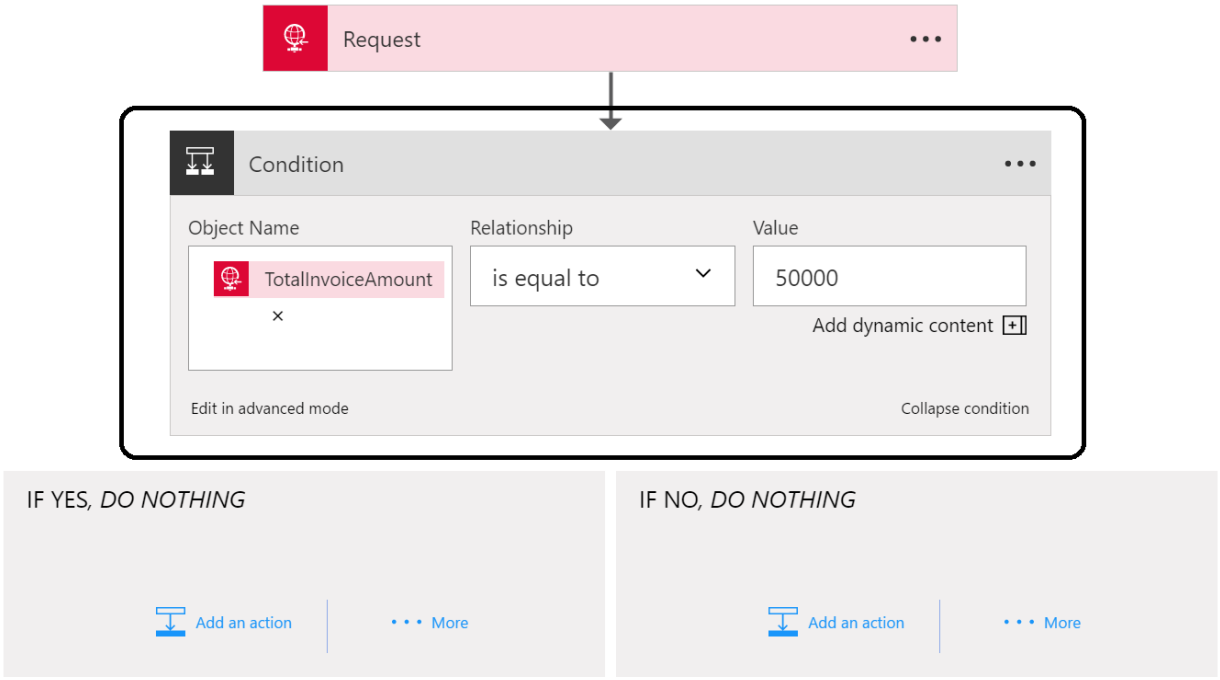
4. The generated schema can be pasted into the **Request Body JSON Schema** part of the HTTP Trigger. Note that the URL is generated after the Logic App has been saved for the first time.



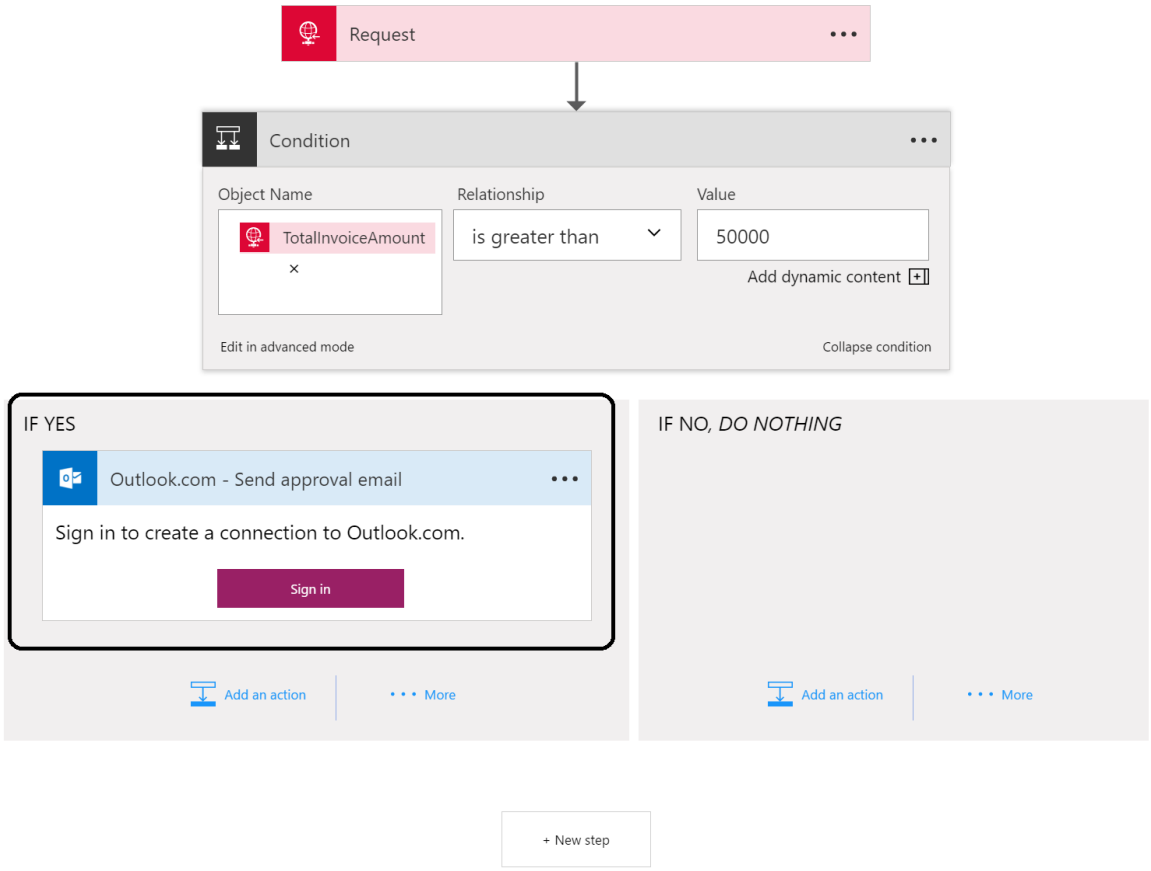
5. The next step is to add a **condition**.



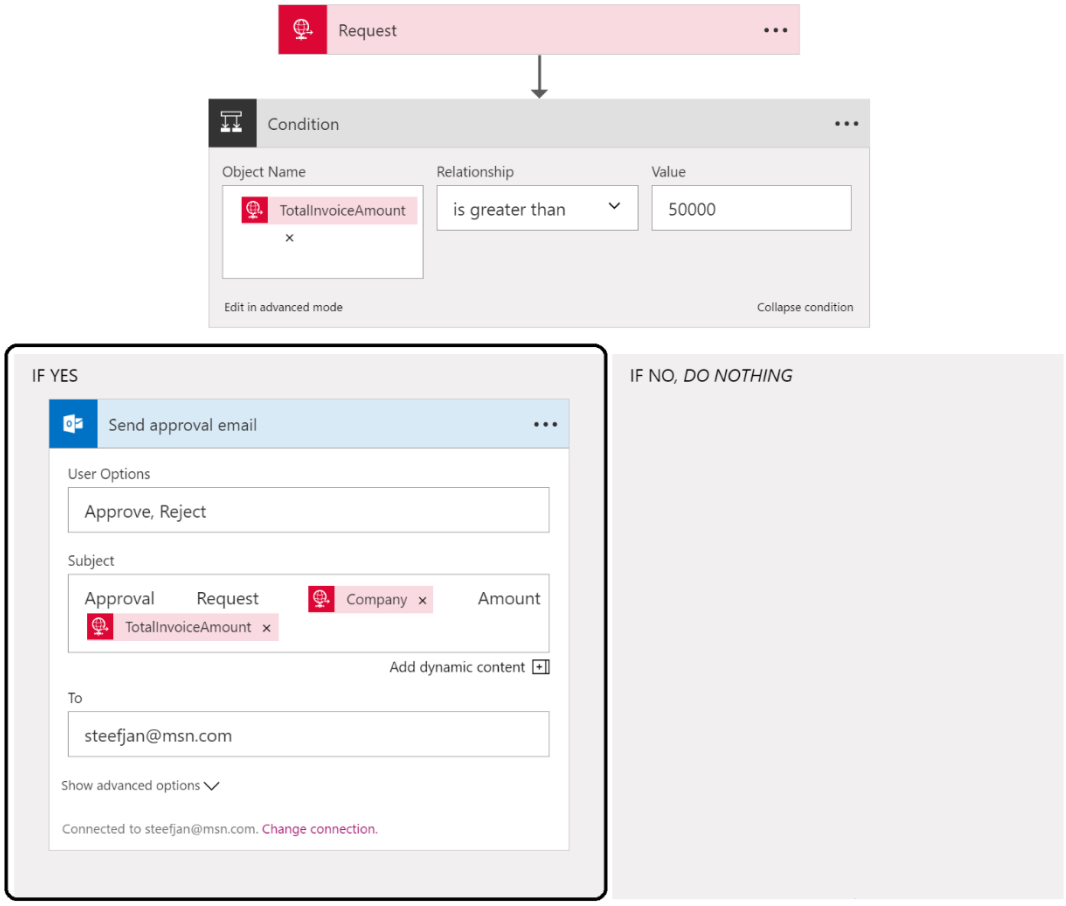
6. The **condition** will be to verify if the amount is higher than **50000**.



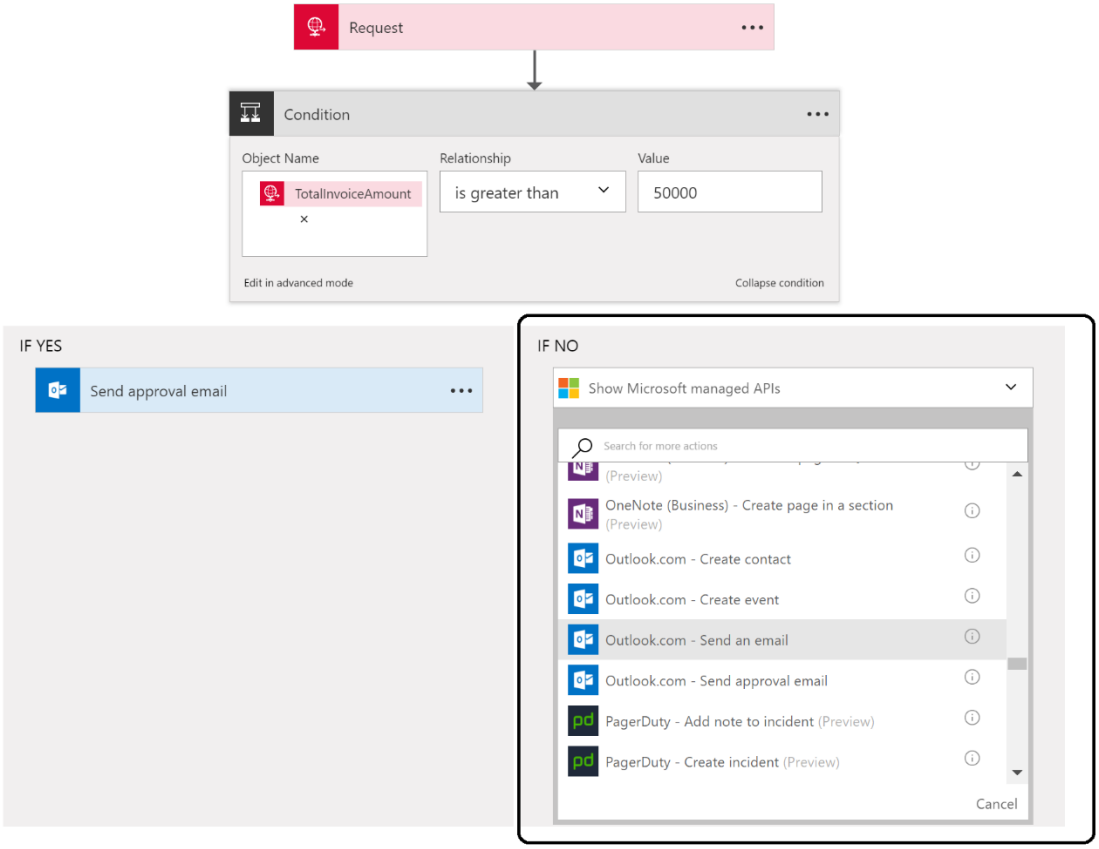
7. Add an **Outlook** (Send Approval Mail) **action** to the left branch.



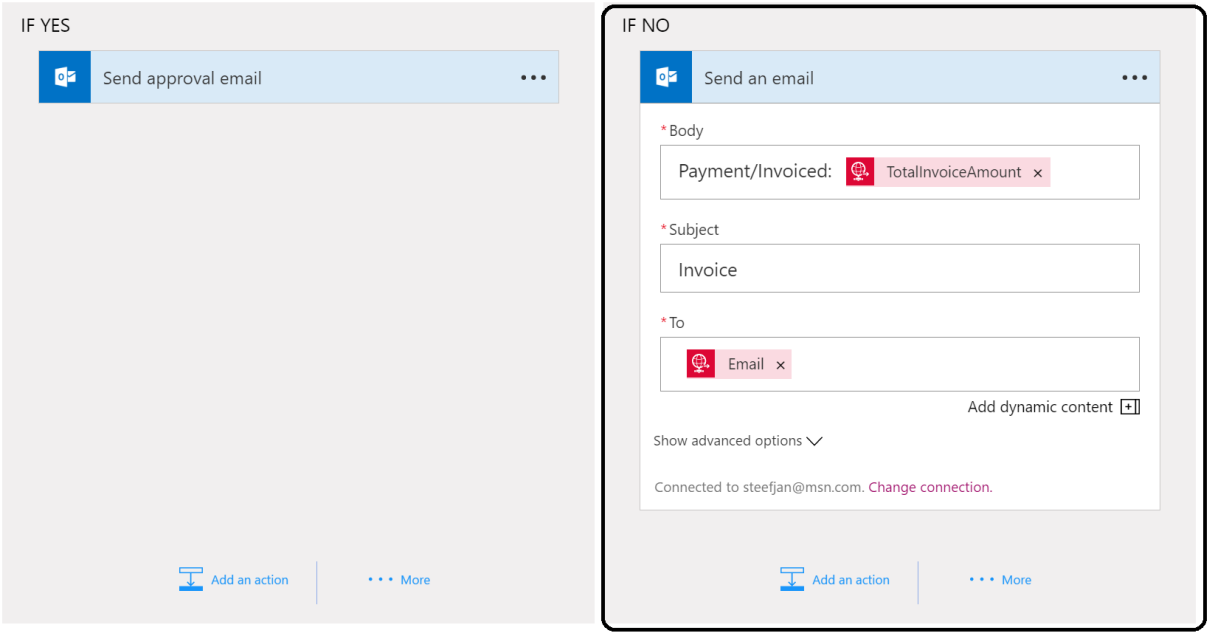
8. **Login** to your **Outlook.com** account. If you do not have one you can create one easily.



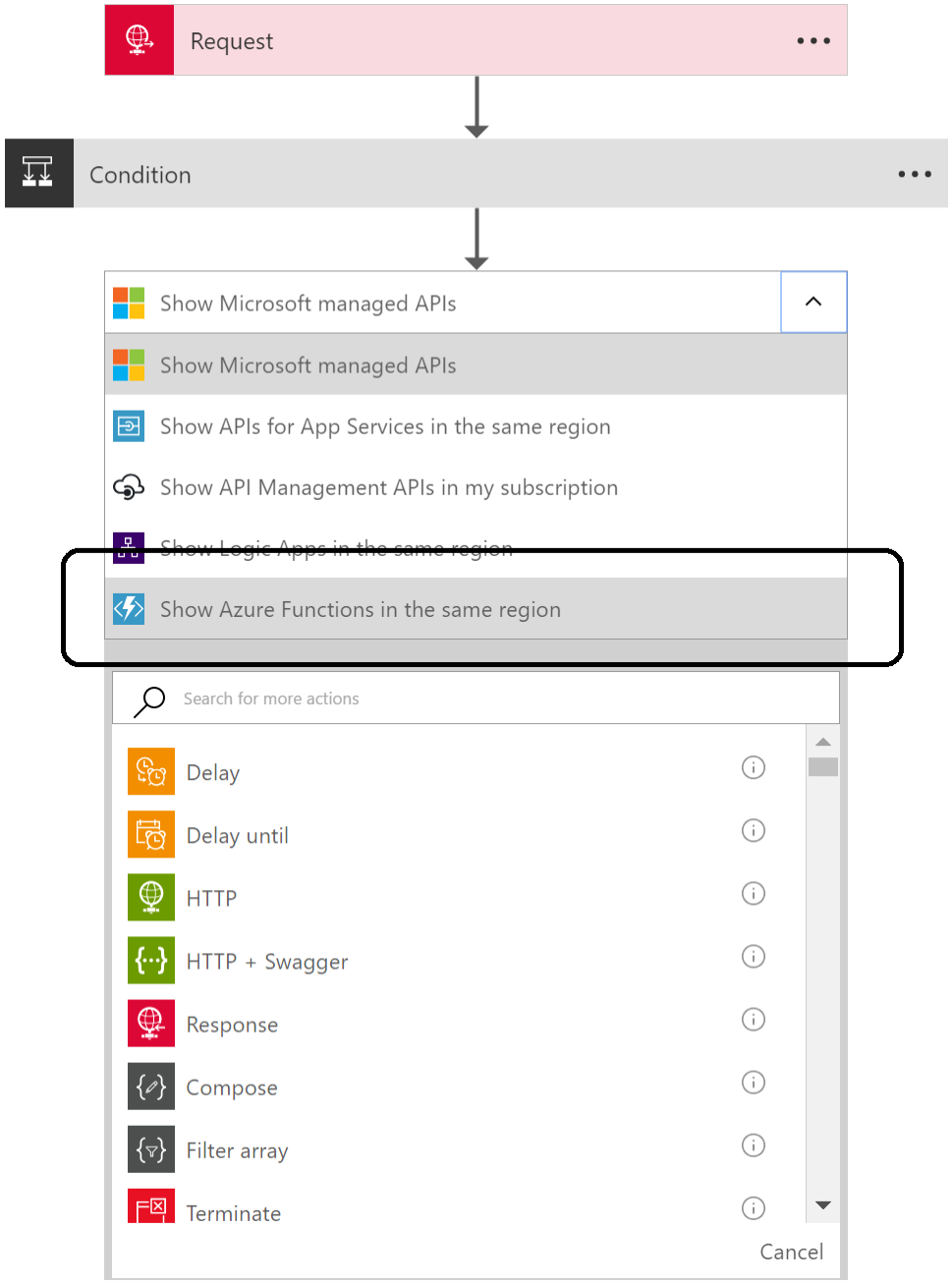
9. After the connection to your outlook has been established you can specify the subject like above and add your email address in the **To** field.



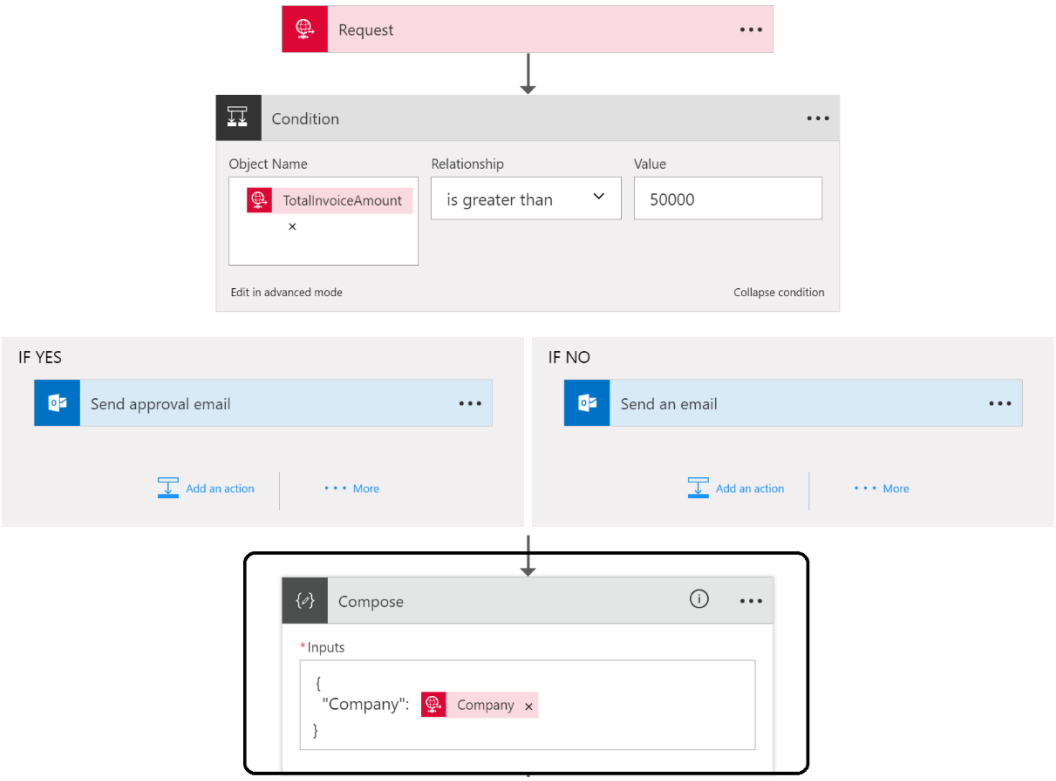
10. In the right branch (IF NO) you can add an Outlook.com Send an email action using your earlier established connection. You do not need to login again.



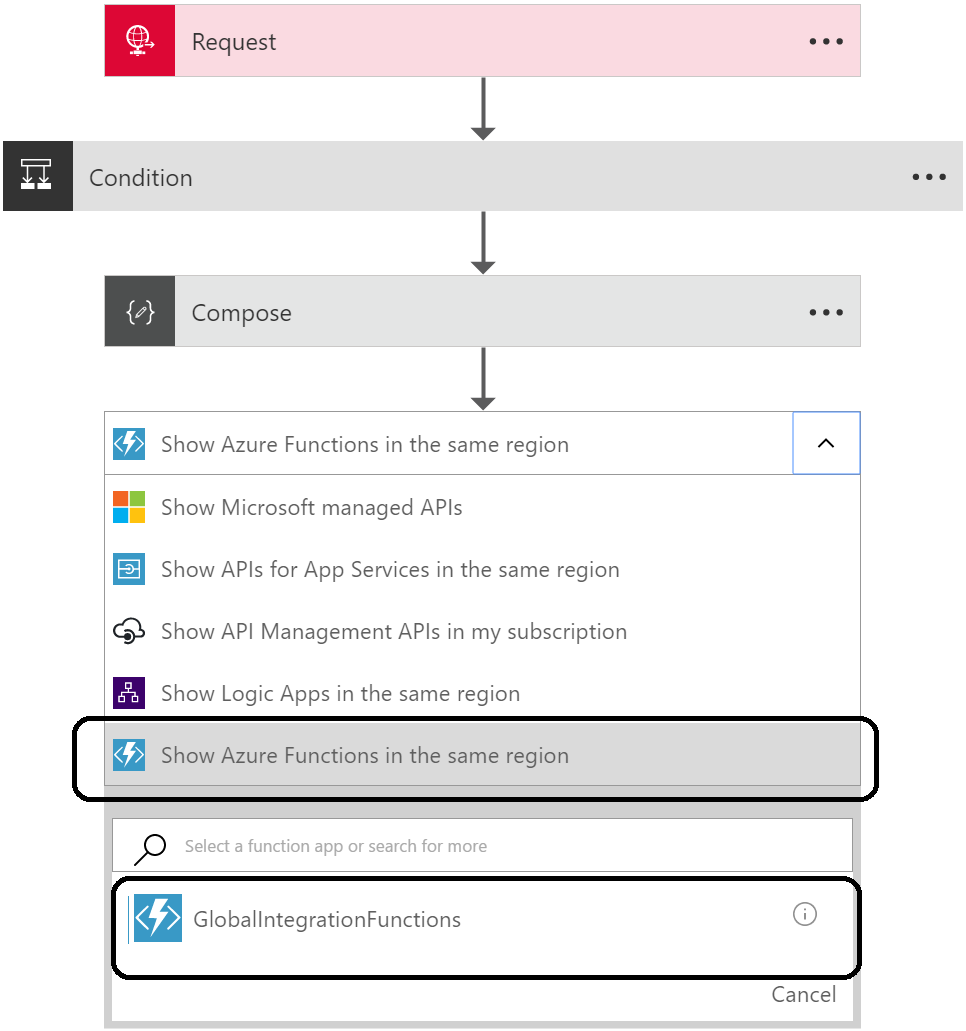
11. You can specify the body as shown above, the subject and the **To** email (which is email to your email address).
12. Add another action below the condition branches
13. Select **Show Azure Functions** in the same region.



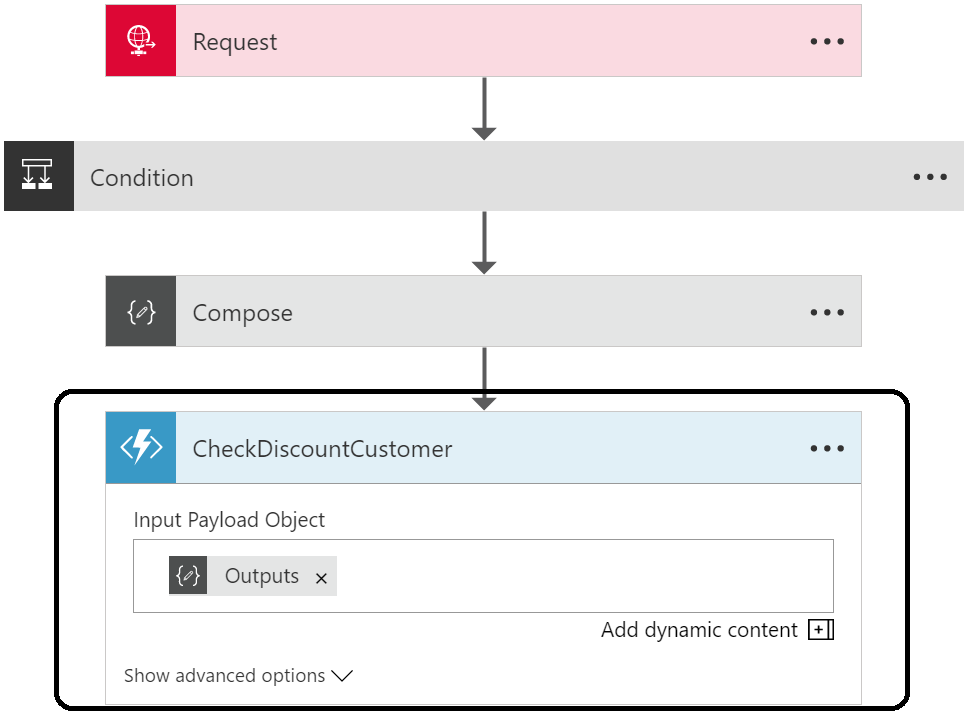
14. Add an **Compose** action below the conditions.



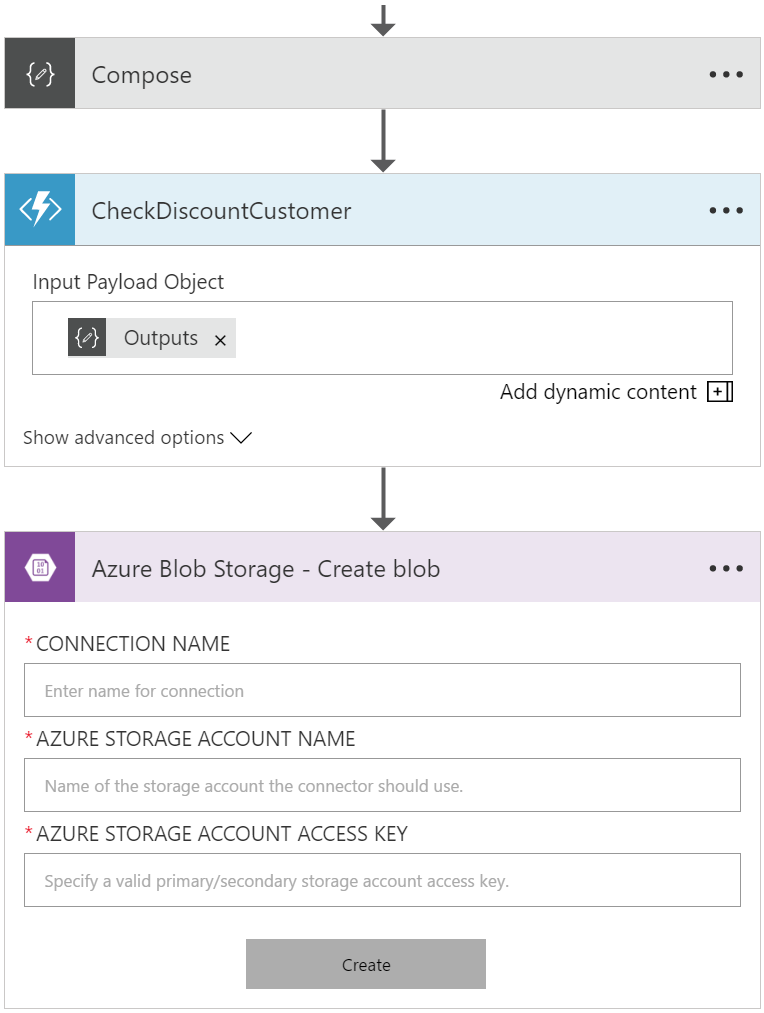
15. In the **Compose** Window create the input as above.
16. Add an action below the **Compose**.
17. Select your function app that will appear.



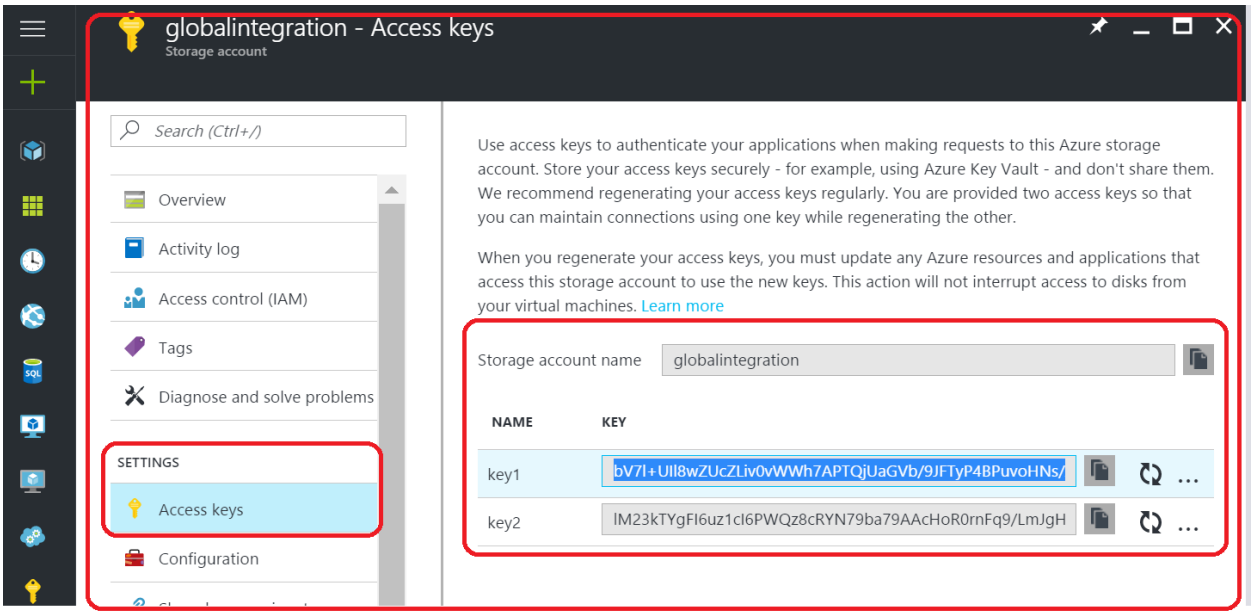
18. Select your custom created function.
19. Drag the **Output** from the **Compose** in the **Input Payload Object**.

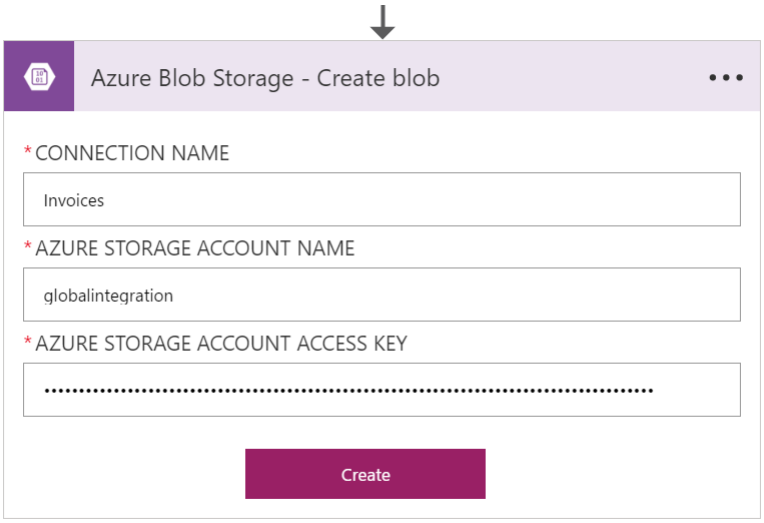


20. Add another action **Azure Blob Storage - Create Blob**.

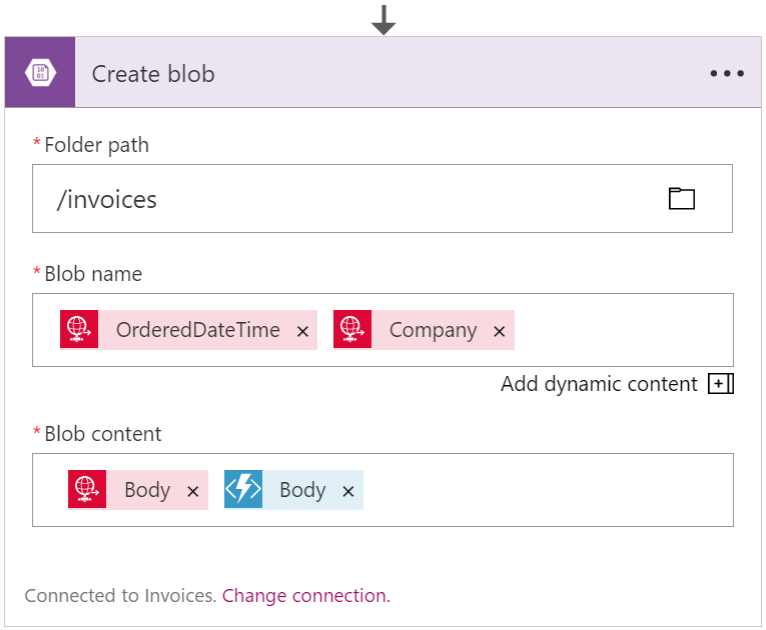


21. Specify the connection **Name**, **Storage Account** and **Key**. You can obtain the Account and Key from your previously created storage account.





22. Click **Create**.
23. Now you can select the path i.e. container for the blobs (invoices), blob name and define the content.

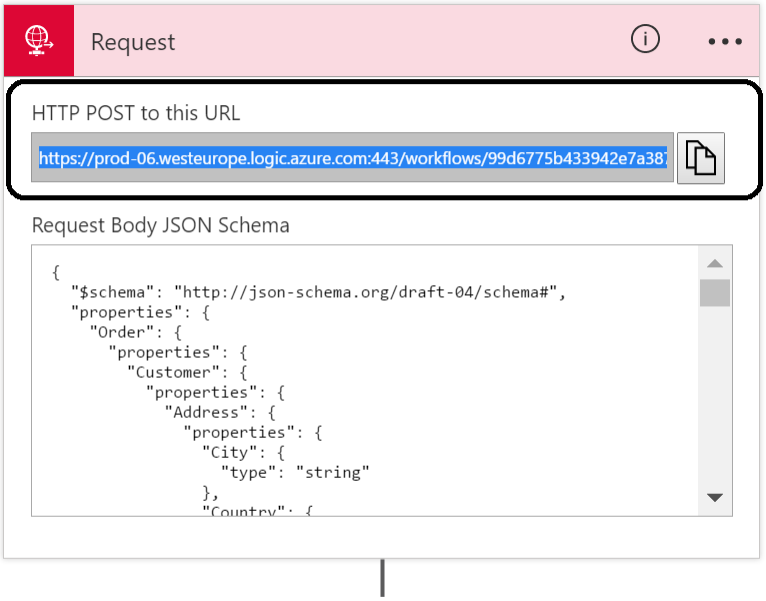


24. **Save** the Logic App Definition.

Test the Solution

Our solution has completely setup and can be tested now. And to test the solution you can use Postman, which can be obtained through online.

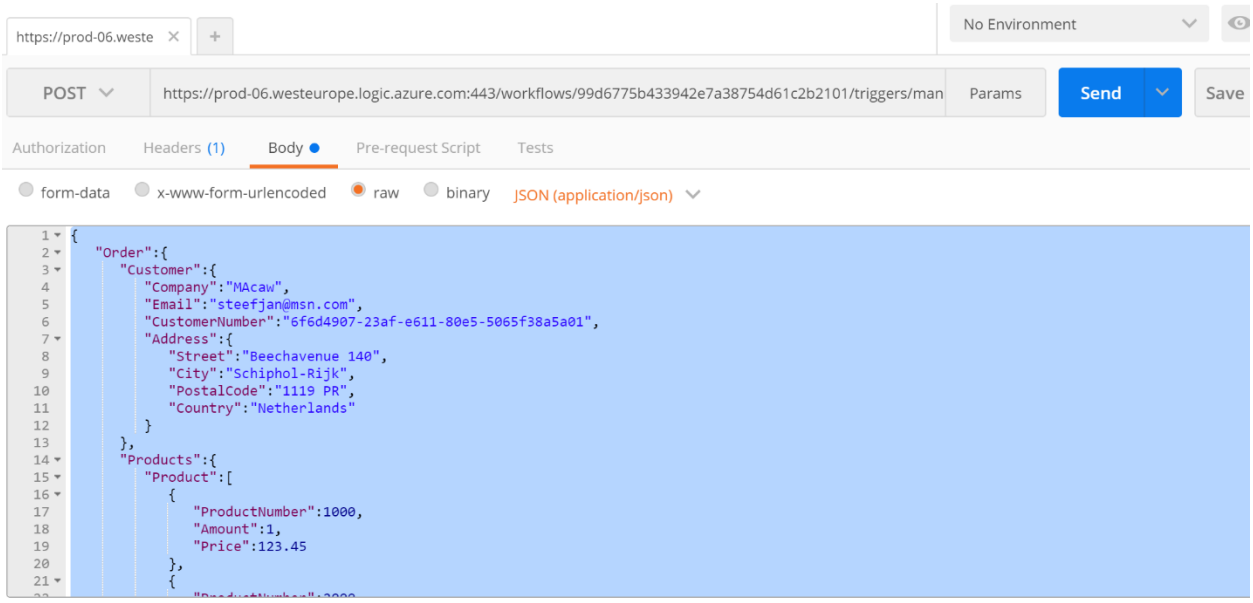
1. To be able to post anything to the **Logic App** you'll need to open up the **Logic App Designer** and copy the **URL**.



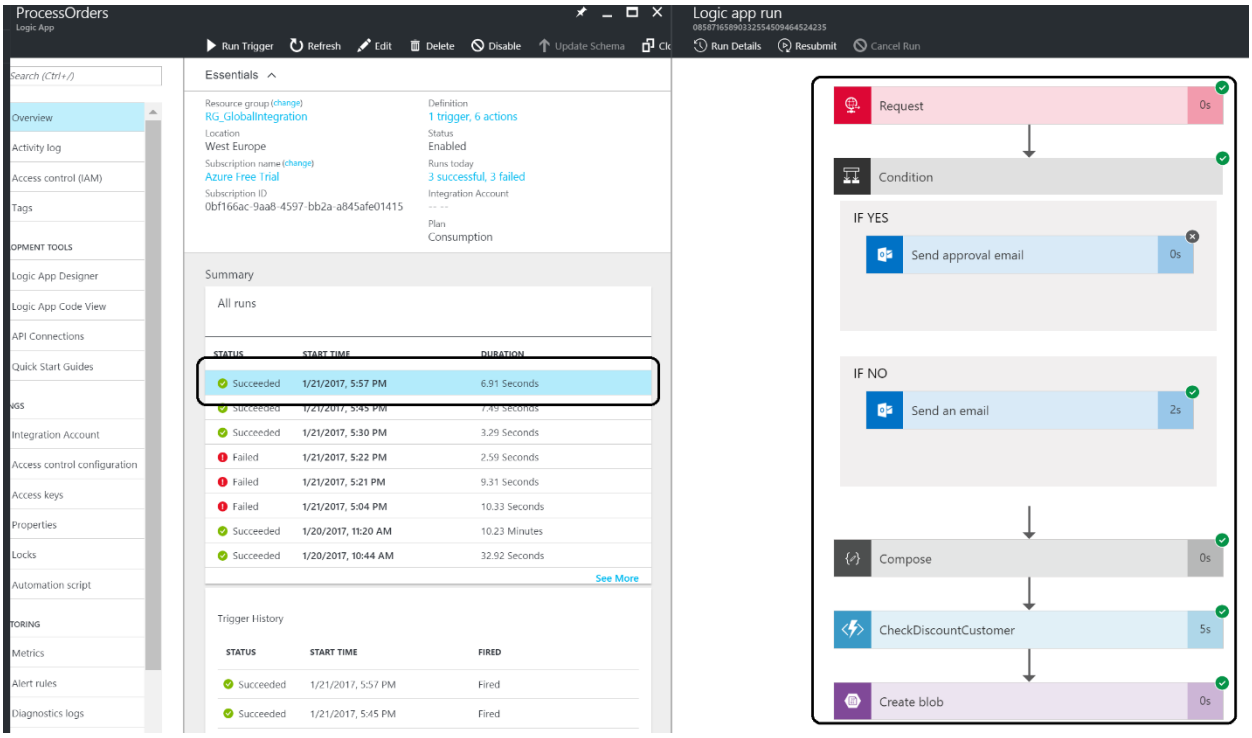
2. Open **postman**.
3. Copy address in address bar, select **POST** Method, for the body choose **raw** and content type **application/json**.
4. Post the following as body content i.e. you can change **TotalInvoiceAmount** to an amount below 50000, and specify your own email address!

```
{
   "Order":{
      "Customer":{
         "Company":"Macaw",
         "Email":"steefjan@msn.com",
         "CustomerNumber":"6f6d4907-23af-e611-80e5-5065f38a5a01",
         "Address":{
            "Street":"Beechavenue 140",
            "City":"Schiphol-Rijk",
            "PostalCode":"1119 PR",
            "Country":"Netherlands"
         }
      },
      "Products":{
```

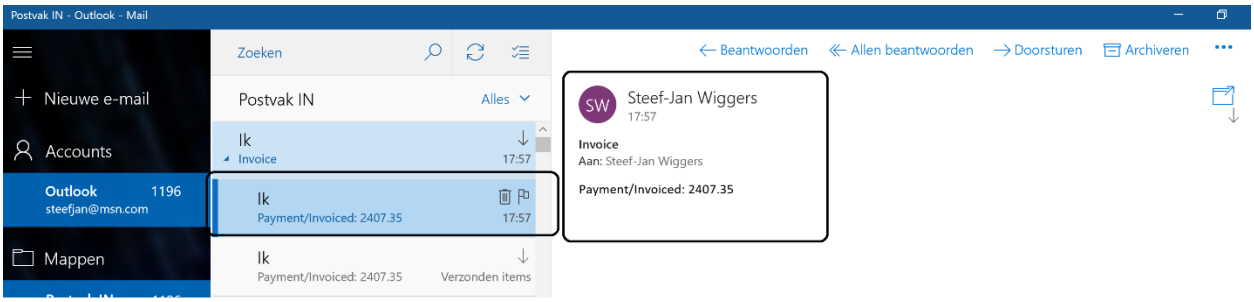
```
            "Product":[
                {
                    "ProductNumber":1000,
                    "Amount":1,
                    "Price":123.45
                },
                {
                    "ProductNumber":2000,
                    "Amount":5,
                    "Price":456.78
                }
            ]
        },
        "OrderedDateTime":"2016-11-20T14:26:00",
        "TotalInvoiceAmount":2407.35
    }
}
```



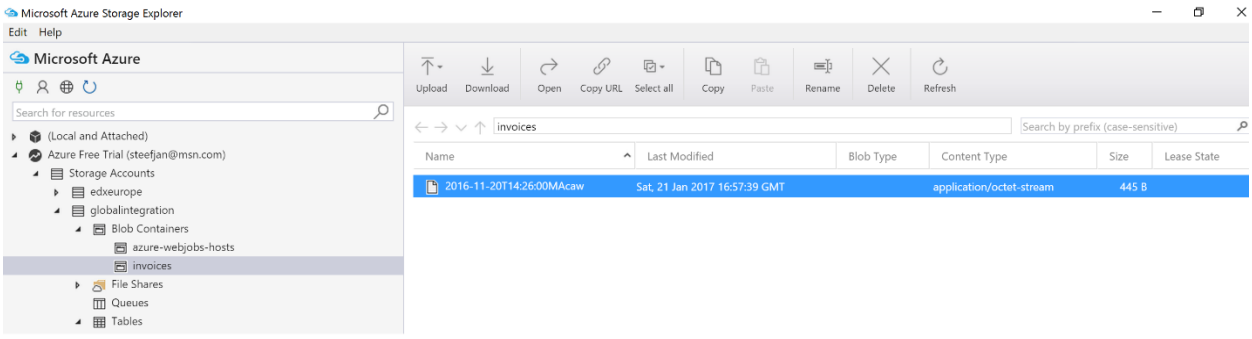
5. Click **Send**.
6. Switch over to **Azure Portal**.
7. Go to your **Logic App**.
8. Click on the last run and examine the steps.



9. Look into your **mail box** and see if you have an **email**.



10. Check using the **Azure Explorer** if there's a blob in the invoices container.



11. **Click** on the blob and examine the contents and look for the discount.
12. Repeat the test with an amount above **50000**.

13. You'll have to look into your mailbox to see an approval mail, which you can **approve**. Please note that the HTTP trigger has a time-out, so if this takes to long, the Logic App can get into a timeout. Using the Service Bus trigger this will not be an issue.

IoT Hub + Stream Analytics + DocumentDB + PowerBI (IoT)

Objective

In this final lab, we will simulate the trucks which deliver the packages at the customers of SETR. The drivers will have a device on which they can choose the following options at every customer:

- Package was delivered successfully
- Customer was not present
- Customer did not accept the package

The devices will all communicate with IoT Hub, where the messages will be collected. We will use Stream Analytics to process the messages from the IoT Hub, and send this to PowerBI for live visualizations, as well as DocumentDB for storing all data.

Prerequisites

- Azure subscription
- PowerBI account
- Device Explorer

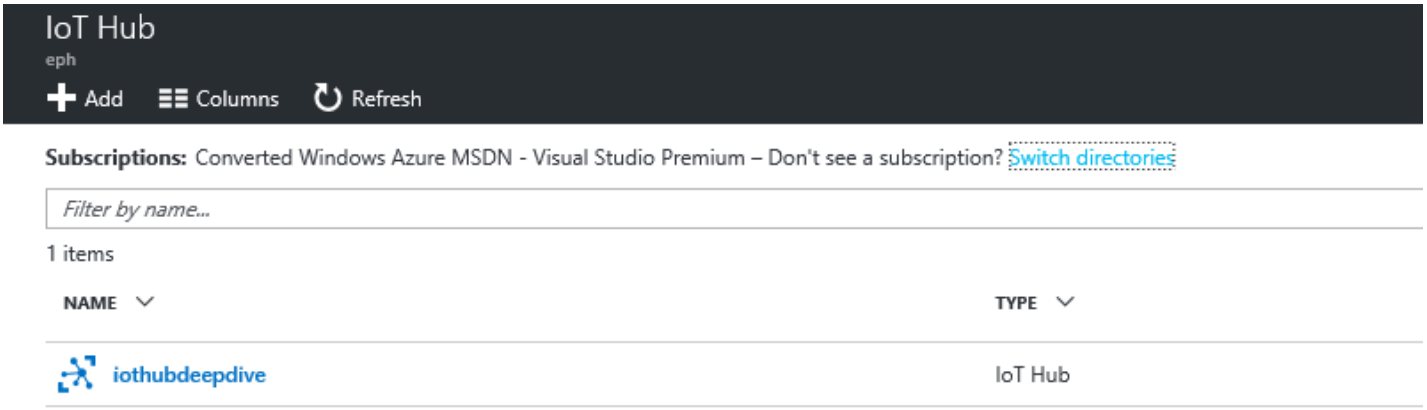
Steps

To build the solution in this lab you have to follow the steps described in this section. From a high level view the steps are:

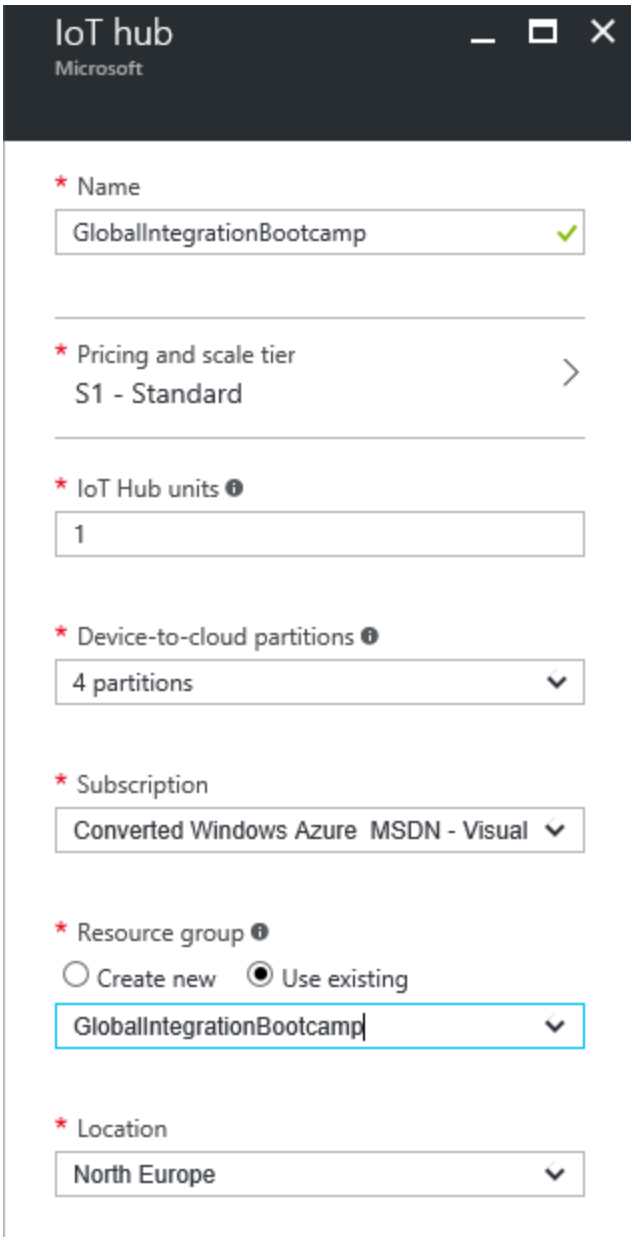
9. Create IoT Hub
10. Create DocumentDB
11. Register device
12. Create simulated device
13. Create Stream Analytics job

Create IoT Hub

IoT Hub can be used for bi-directional communication between Azure and billions of devices. We will use IoT Hub here to send messages from our (simulated) device into Azure. Go to the IoT Hub blade in the Azure portal, and create a new IoT Hub.

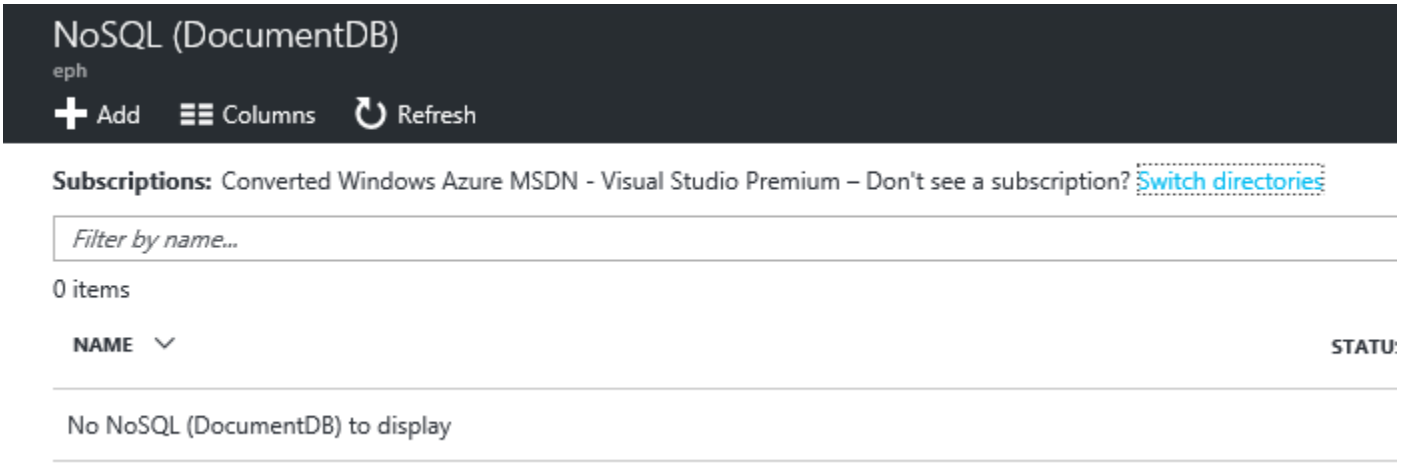


You can create one free IoT Hub in your subscription which is great for testing, or you can use one of the paid priceplans if you want to be able to handle more events.

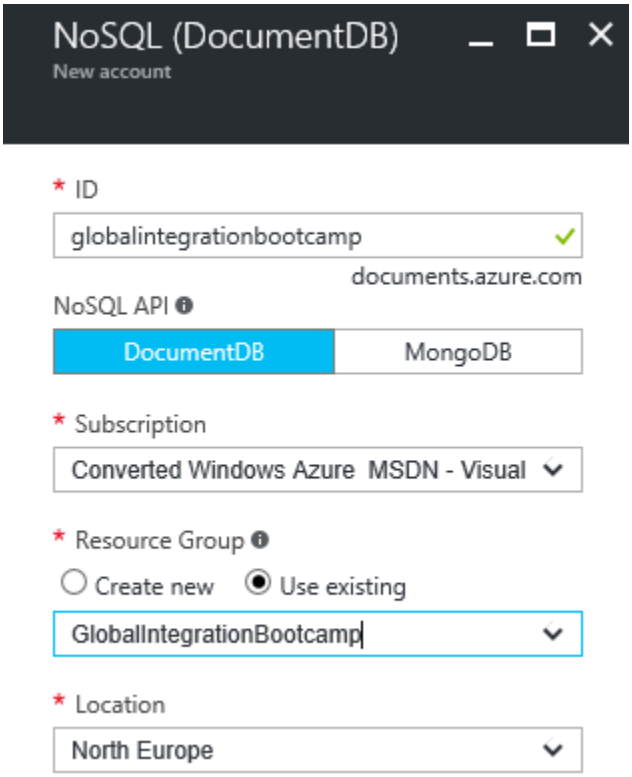


Create DocumentDB

DocumentDB is Azure's NoSQL solution, which is ideal in case you need to store large amounts of data and need a performance when working with this data. Go to the DocumentDB blade in the Azure portal, and create a new DocumentDB.

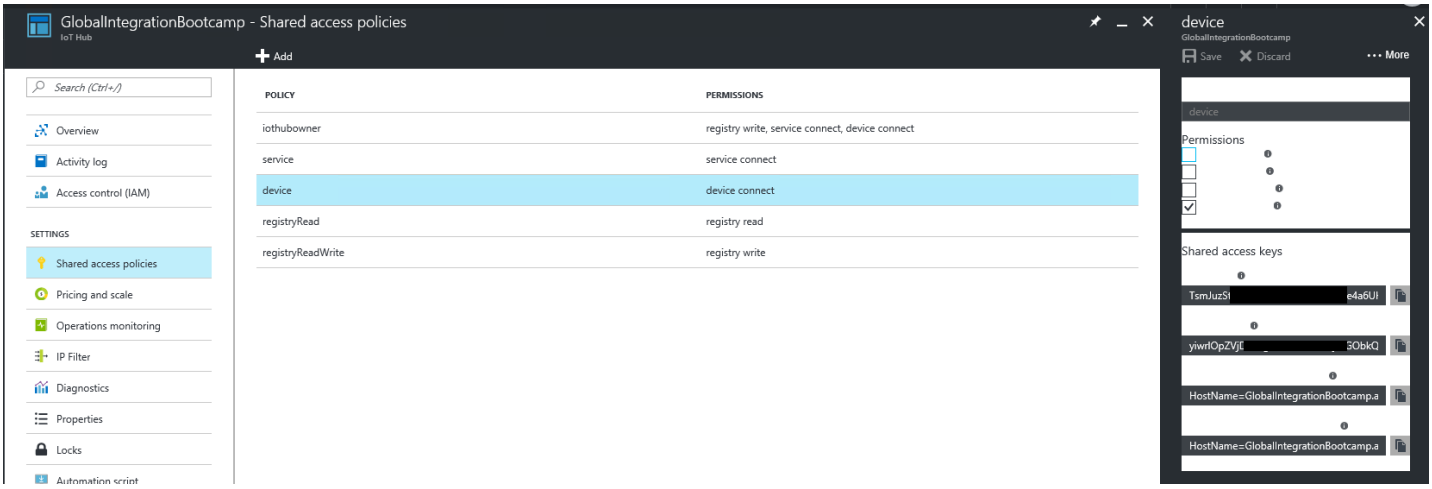


For this lab, we will use a DocumentDB backend.

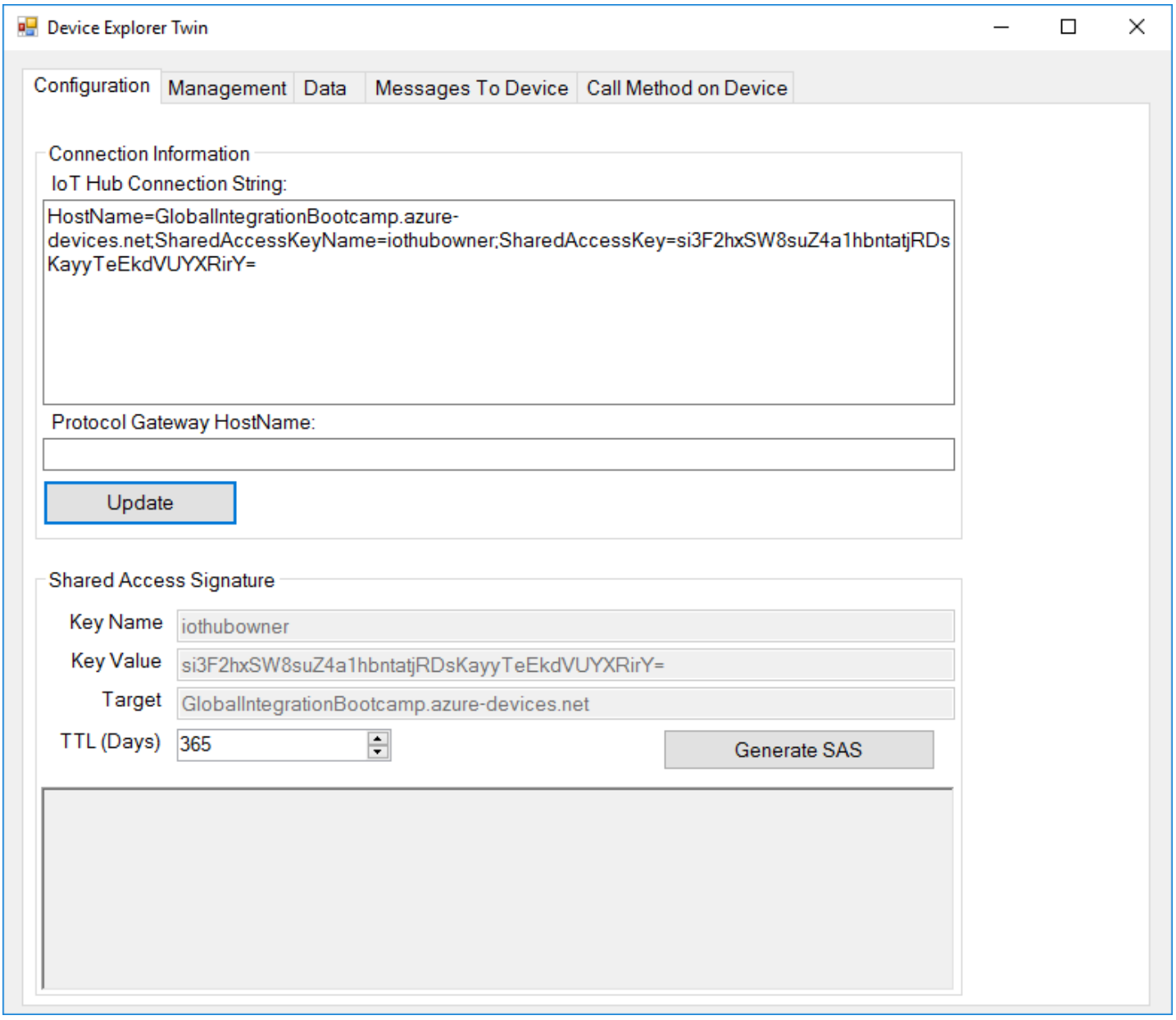


Register device

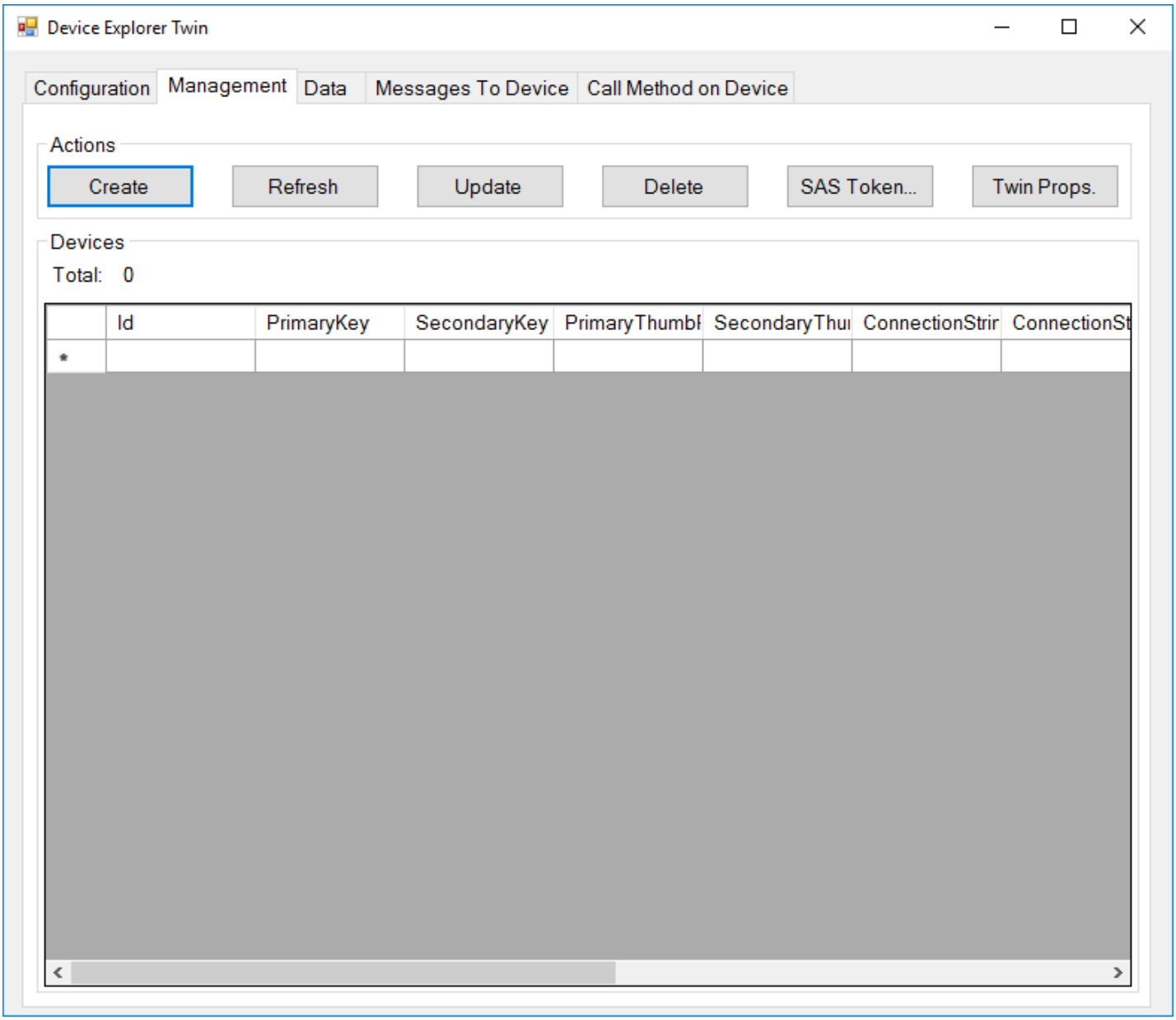
Before a device can send its messages to our IoT Hub, it has to be registered. For this, we will use Device Explorer, a sample application which can be downloaded from GitHub which allows us to work with our IoT Hub. Get the connection string for the iothubowner policy, which we will use to connect with Device Explorer.



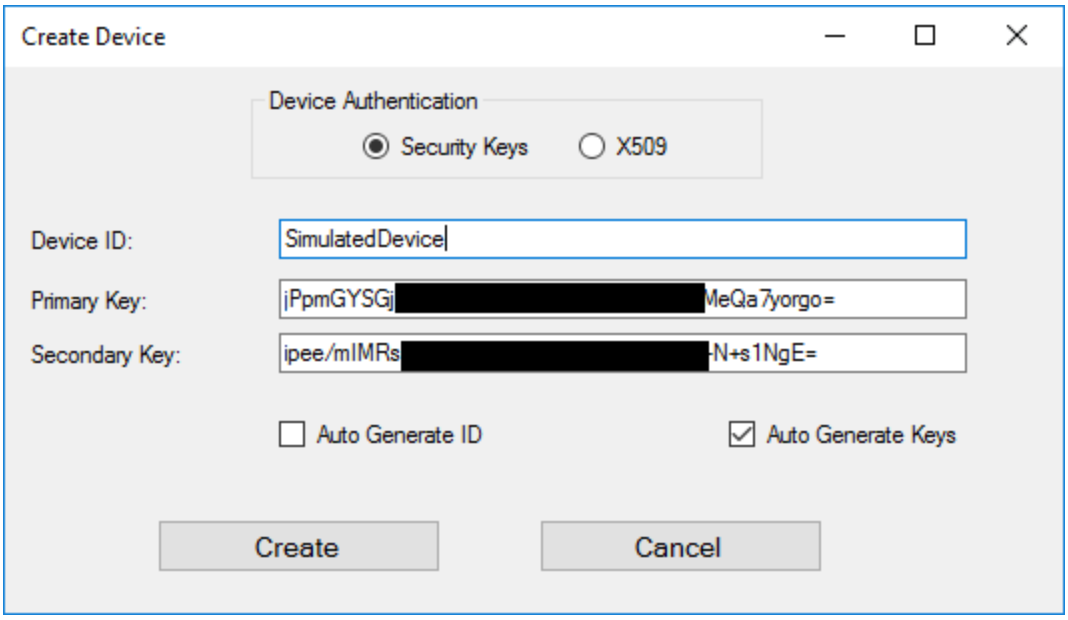
Connect to the IoT Hub using the connection string we just retrieved.



On the management tab, create a new device.

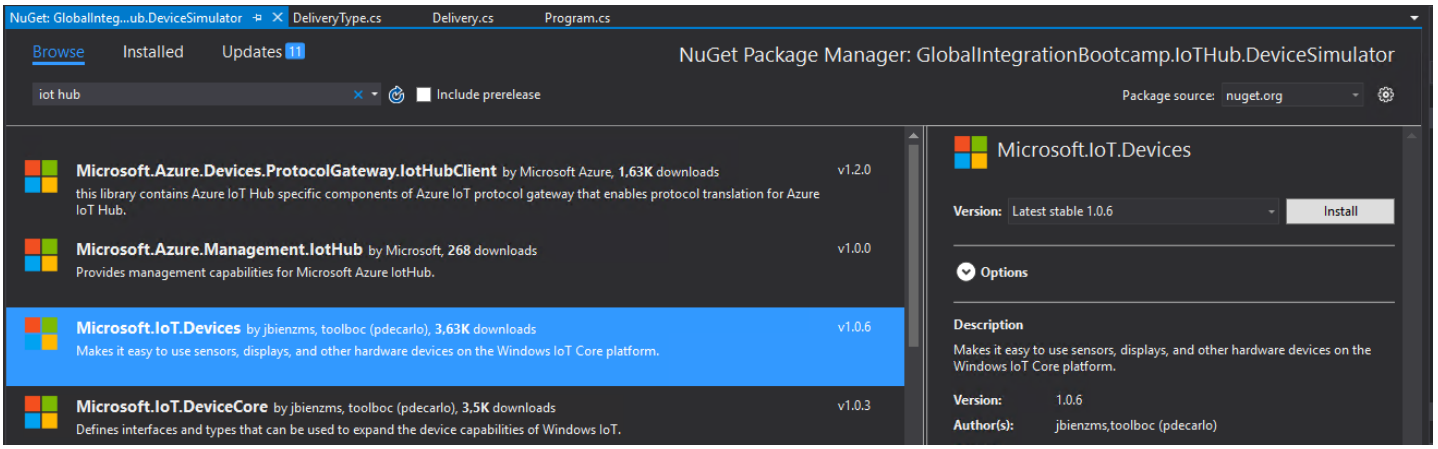


Set the name of the device, and get the primary key, we will use this later on in the code for connecting our simulated device.



Create simulated device

Create a new **Console Application** project in Visual Studio, and add a reference to the **Microsoft.IoT.Devices** NuGet library.



Create a new enum in the project, which will be used to set the result of the delivery.

```
namespace GlobalIntegrationBootcamp.IoTHub.DeviceSimulator
{
    /// <summary>
    /// Possible results for a delivery.
    /// </summary>
    public enum DeliveryResult
    {
        PackageDelivered,
        CustomerNotHome,
        CustomerDidNotAcceptPackage
    }
}
```

Next create a struct, which will represent a delivery with the prior created types of delivery result.

```
using System;

namespace GlobalIntegrationBootcamp.IoTHub.DeviceSimulator
{
    /// <summary>
    /// Class representing a delivery.
    /// </summary>
    public struct Delivery
    {
        /// <summary>
        /// Customer ID.
        /// </summary>
        public Guid Customer;

        /// <summary>
        /// Date / time delivery was done.
        /// </summary>
```

```
        public DateTime DeliveryDateTime;

        /// <summary>
        /// Result of the delivery.
        /// </summary>
        public DeliveryResult DeliveryResult;
    }
}
```

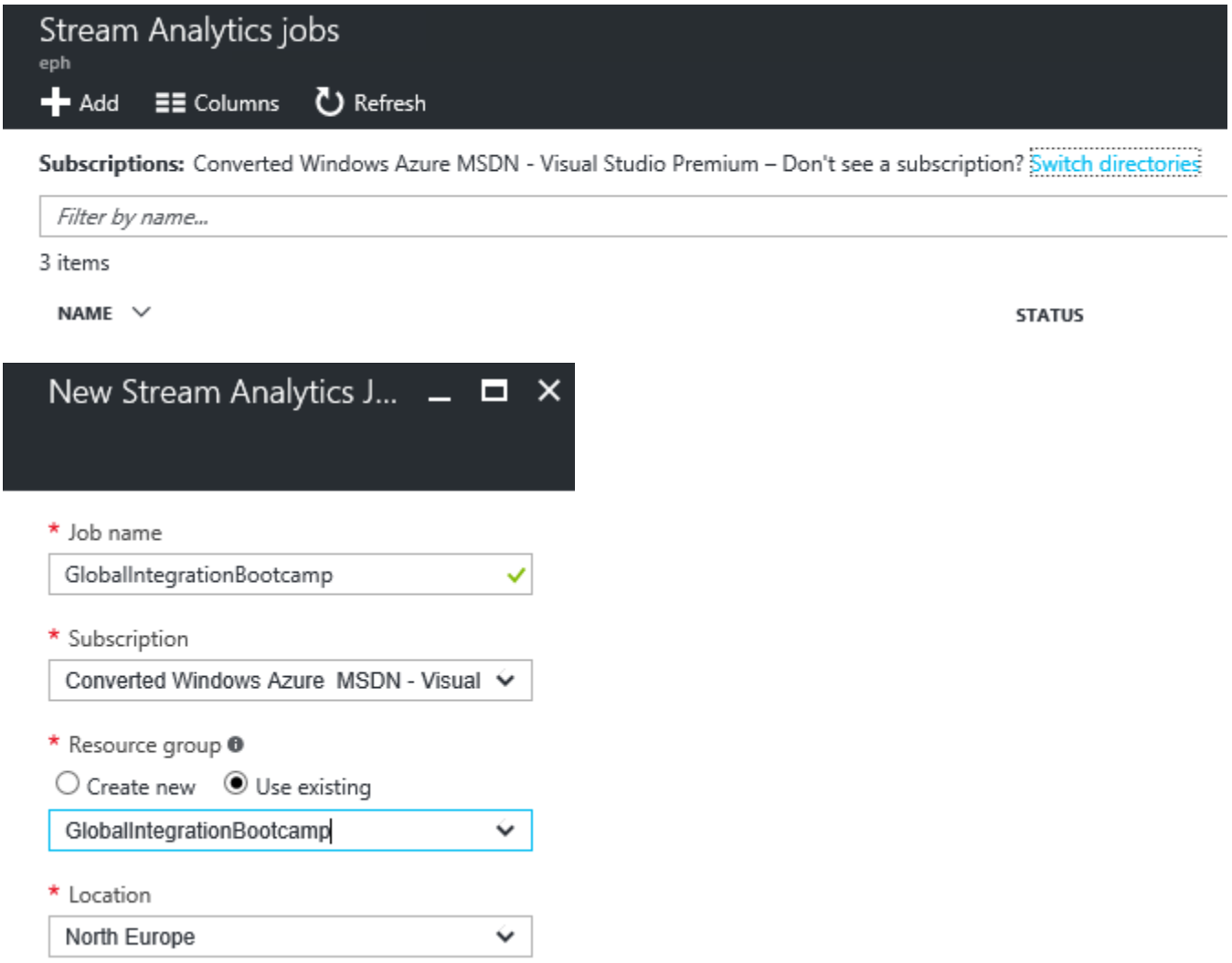
And finally, create the application which will send the messages to our IoT Hub. In the connection string, use the name you gave your IoT Hub. For the key, use the name and primary key of the device you registered previously.

```
using System;
using System.Diagnostics;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

using Microsoft.Azure.Devices.Client;

using Newtonsoft.Json;

namespace GlobalIntegrationBootcamp.IoTHub.DeviceSimulator
{
    public class Program
    {
        /// <summary>
        /// Client used to communicate with IoT Hub.
        /// </summary>
        private static DeviceClient _deviceClient;

        public static void Main(string[] args)
        {
            // Create the client
            _deviceClient =
DeviceClient.Create("GlobalIntegrationBootcamp.azure-devices.net",
                new
DeviceAuthenticationWithRegistrySymmetricKey("SimulatedDevice",
"OjR3mUq9LtBrJ8zdzOOkxxxxxzX8HHaJUQLlnyz+8PFrg="));

            // Send messages
            SendMessages().Wait();
        }

        /// <summary>
        /// Send messages to IoT Hub.
        /// </summary>
        private static async Task SendMessages()
        {
            while (true)
            {
                // Result for the delivery
                DeliveryResult deliveryResult;
```

```
                // Create delivery result
                switch (DateTime.Now.Second)
                {
                    case 10:
                        deliveryResult =
DeliveryResult.CustomerDidNotAcceptPackage;
                        break;
                    case 20:
                    case 40:
                        deliveryResult = DeliveryResult.CustomerNotHome;
                        break;
                    default:
                        deliveryResult = DeliveryResult.PackageDelivered;
                        break;
                }

                // Create delivery
                var delivery = new Delivery { Customer = Guid.NewGuid(),
DeliveryDateTime = DateTime.Now, DeliveryResult = deliveryResult };

                // Create message
                var message = new
Message(Encoding.UTF8.GetBytes(JsonConvert.SerializeObject(delivery)));

                // Send message to IoT Hub
                Console.WriteLine($"Sending message:
{JsonConvert.SerializeObject(delivery)}");
                await _deviceClient.SendEventAsync(message);

                // Wait before sending next message
                Thread.Sleep(3000);
            }
        }
    }
}
```

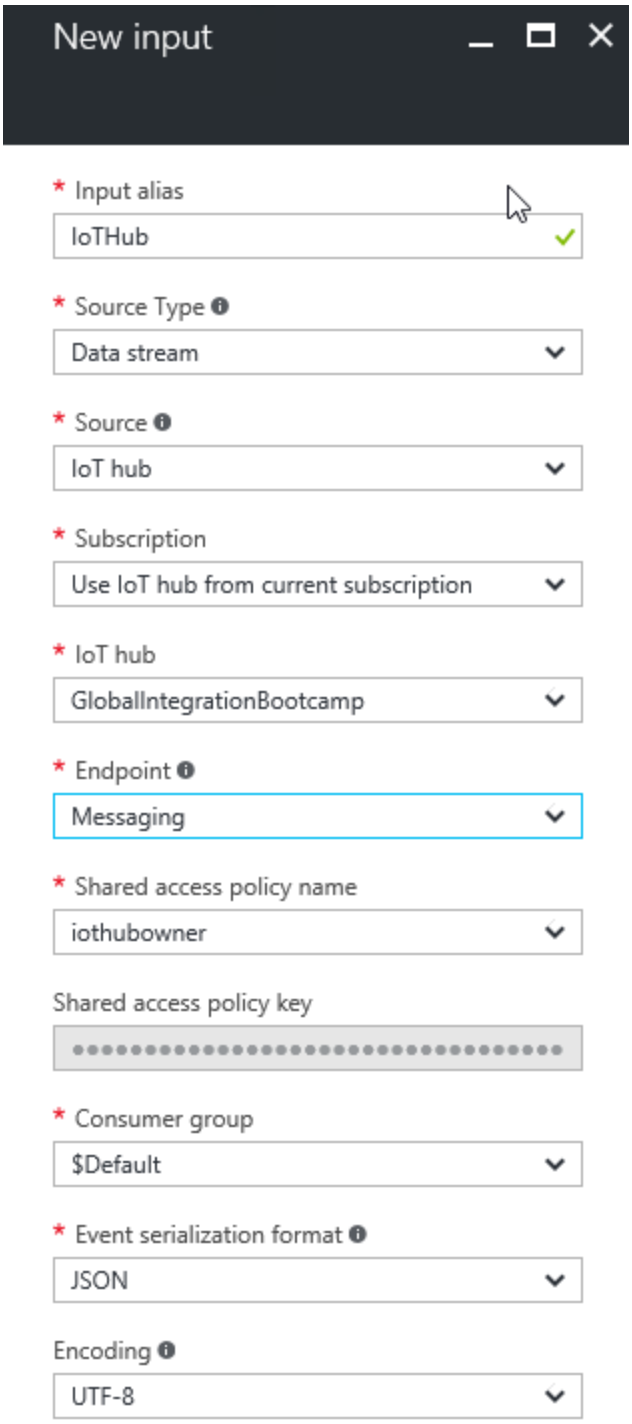
Create Stream Analytics job

We will use Stream Analytics to process the messages sent to our IoT Hub. Go to the Stream Analytics blade in the Azure portal and create a new Stream Analytics job.



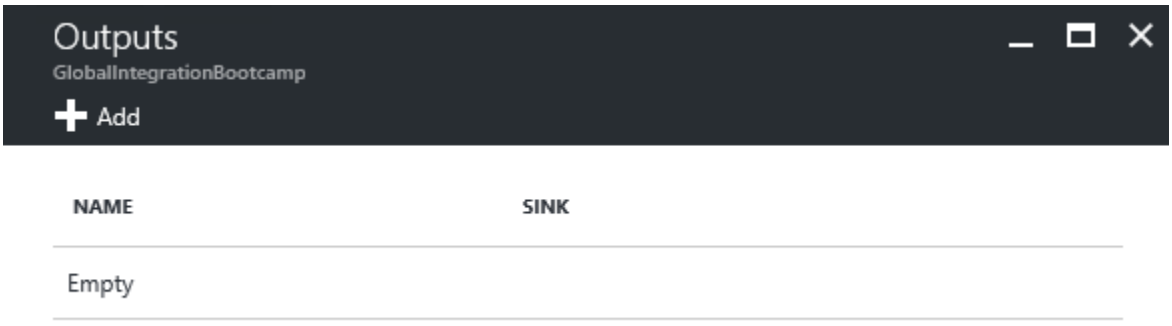
Add input

Once the job has been created, we will add the input from IoT hub. Make sure to select IoT Hub we previously created, and use the iothubowner SAS policy for connecting to it.



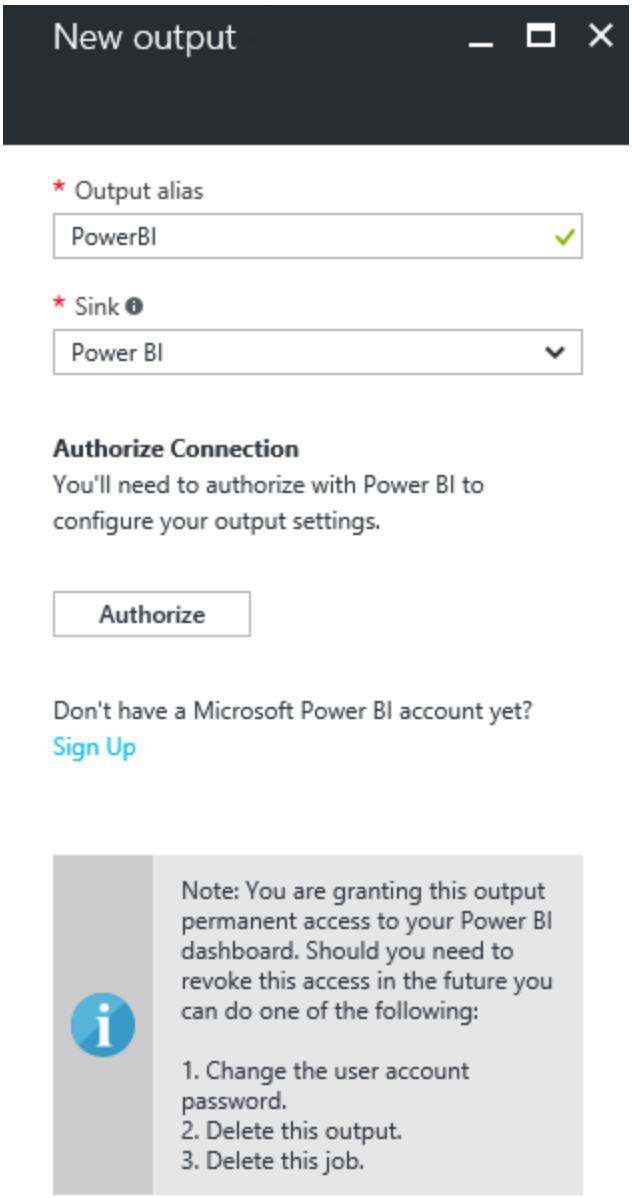
Add outputs

We will now add our two outputs.

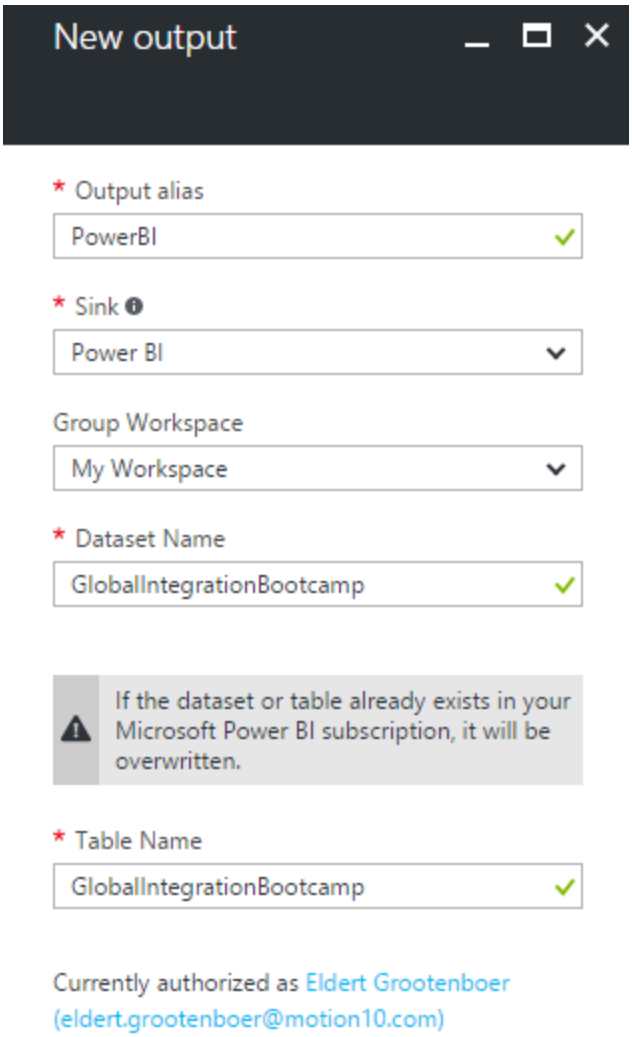


PowerBI

Add a PowerBI output, and authorize it to connect with your PowerBI account.

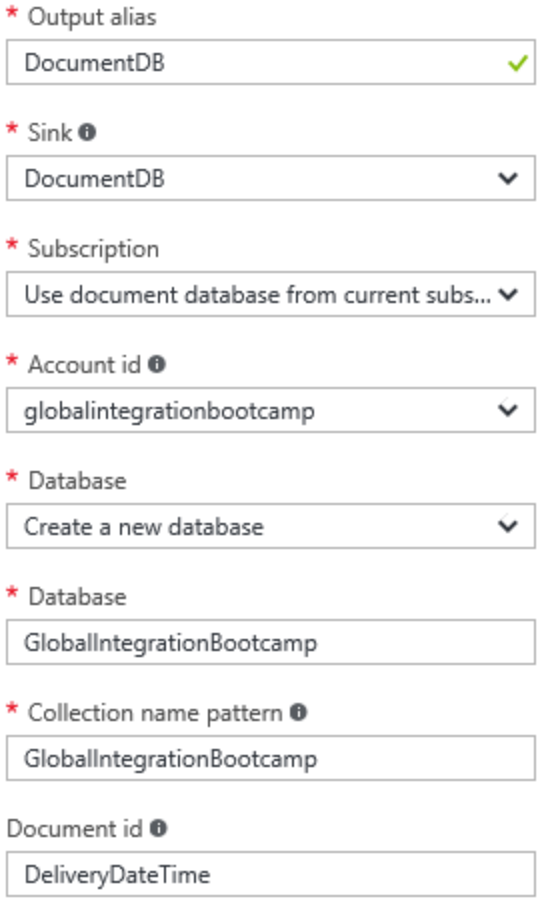


Create a dataset and table in your preferred workspace.



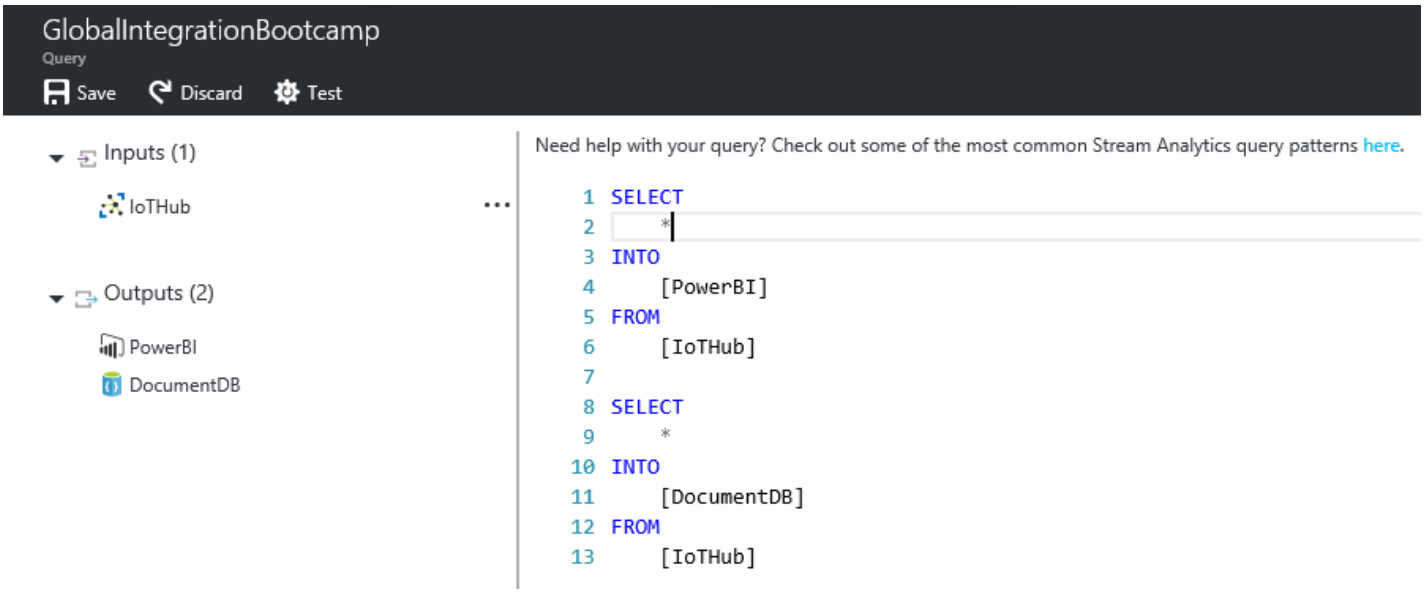
DocumentDB

Add another output, this time to the DocumentDB we created earlier. As we did not create a database, let Stream Analytics create one for us.

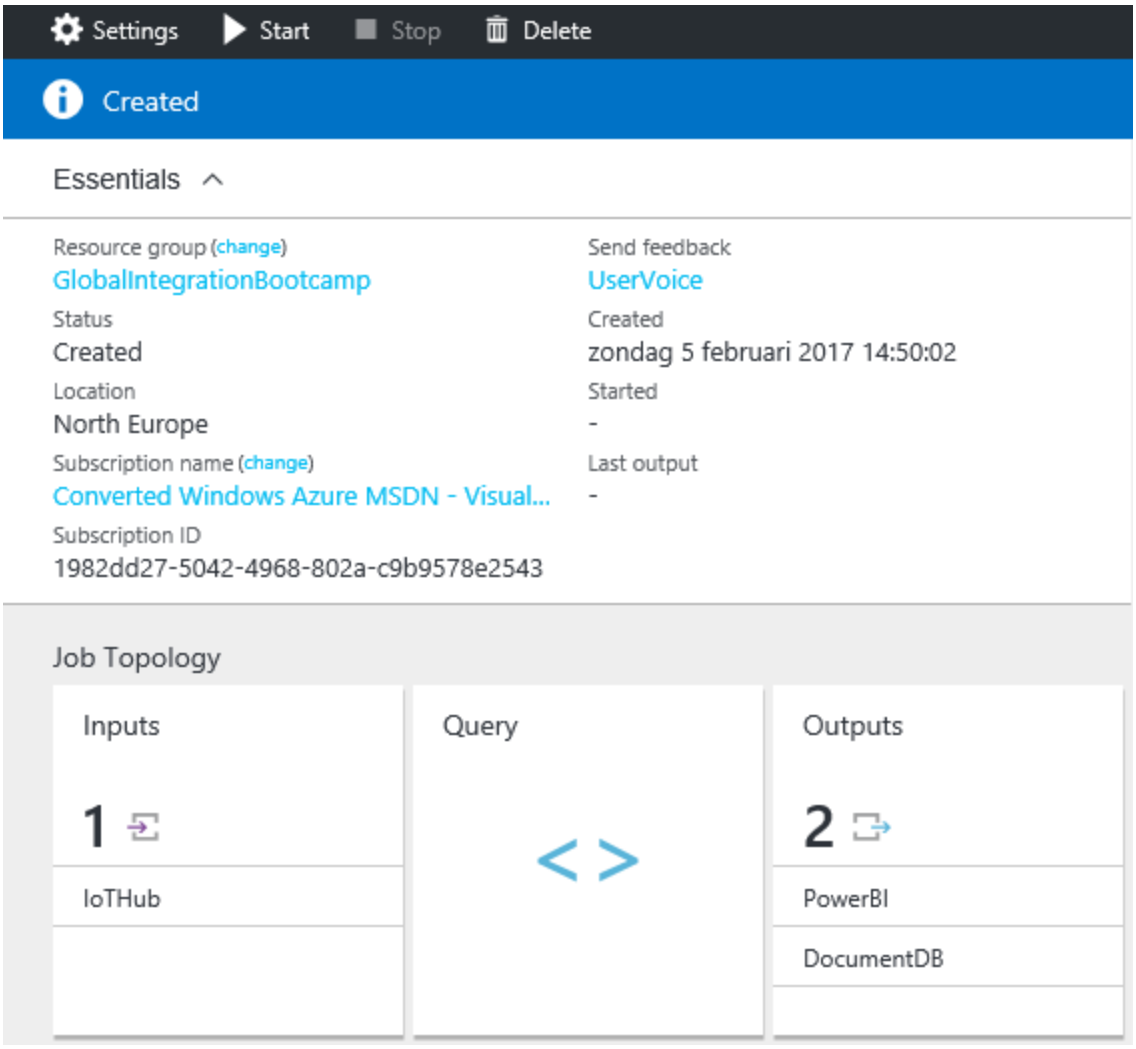


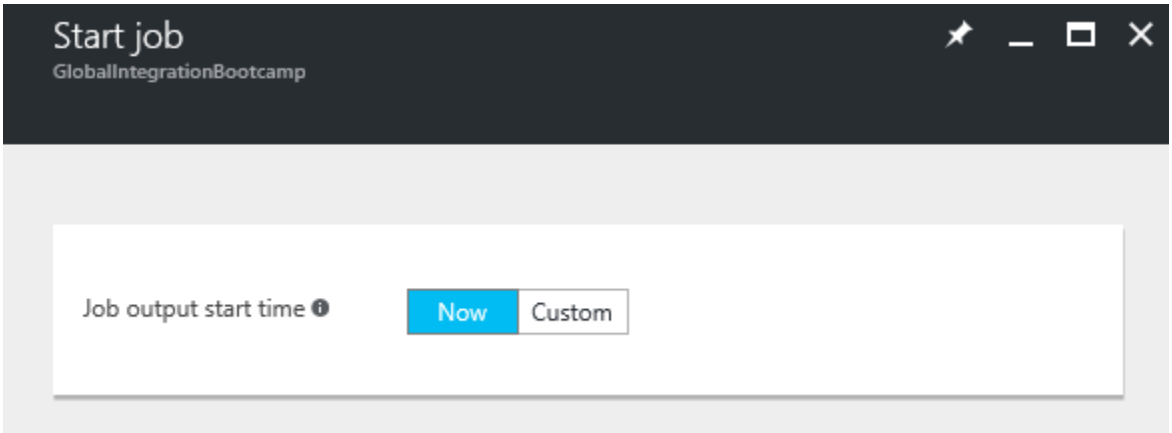
Set query

Now that we have our input and outputs set up, we can create our query. In this lab we will simply sent everything that comes in to both our outputs, but you could also do data transformation here.



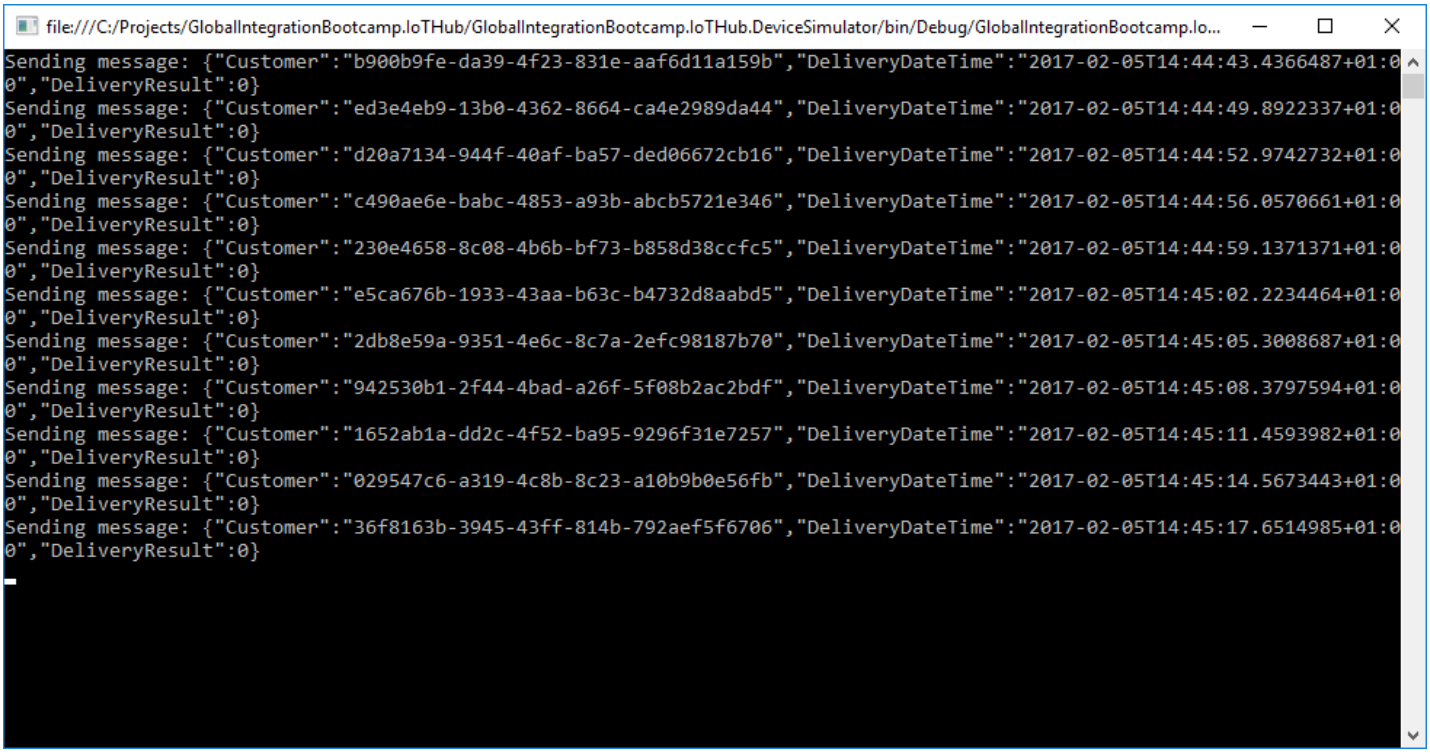
Now that we have our Stream Analytics job all set up, we will start the job.



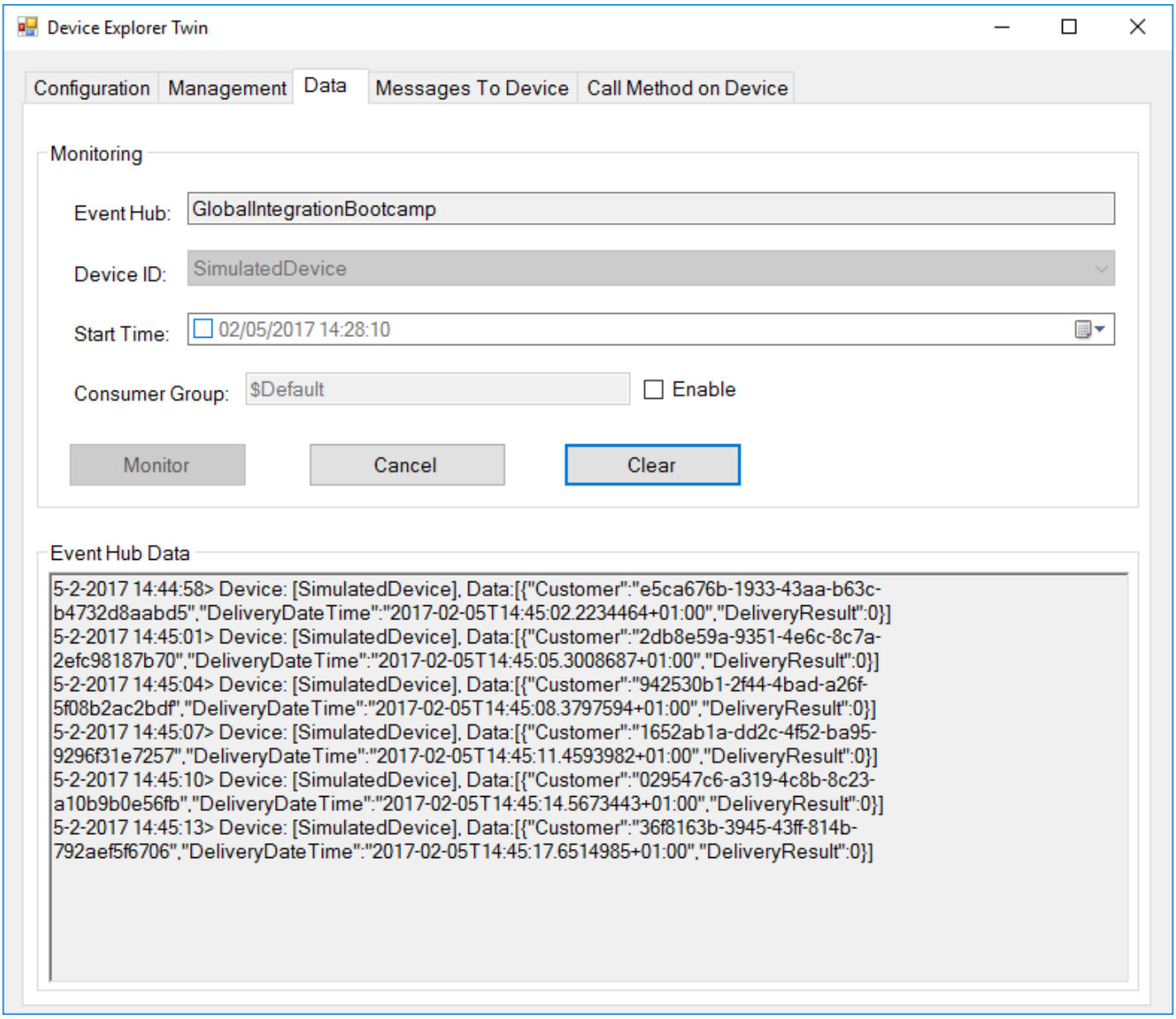


Testing the lab

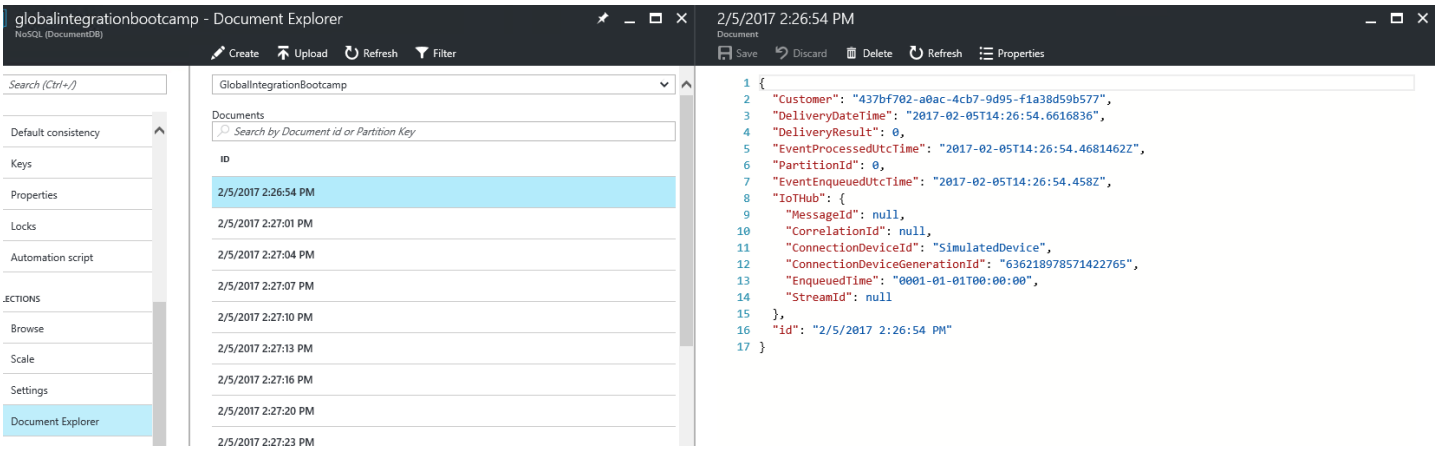
To test this lab, start the console application we created.



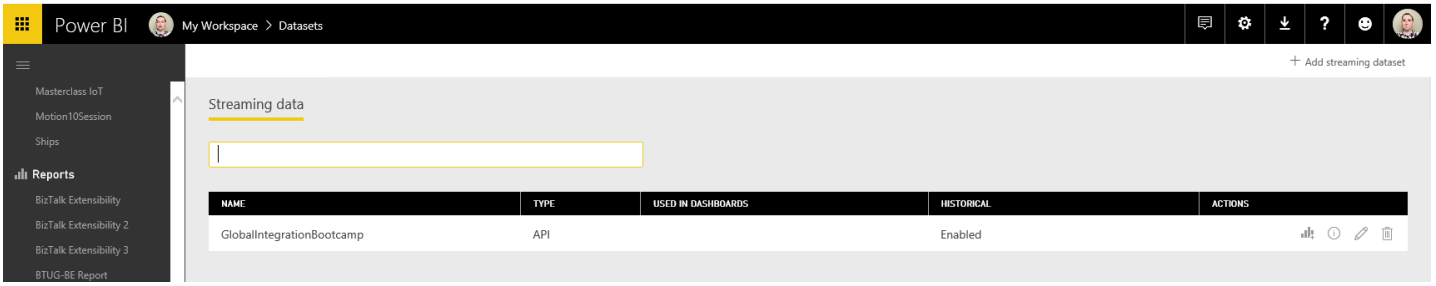
If you want to make sure the messages are arriving at IoT Hub, this can be checked in Device Explorer.
Do not forget to stop monitoring once we have assured the messages are arriving.



We can now go to our DocumentDB in the Azure Portal, and using the Document Explorer, we can see our messages and their content.



In PowerBI, a new Streaming Dataset has been created, on which we can create new reports.



You can now create your own report. With the following report you can check how many deliveries were accepted, rejected or could not be delivered.