



Labs

TOMASSO GROENENDIJK

ELDERT GROOTENBOER

STEF-JAN WIGGERS

ROB FOX



Contents

Naming Conventions.....	5
Lab 1 - API Management + API Apps.....	6
Objective	6
Prerequisites	6
Steps.....	6
Create Service Bus Namespace.....	7
Create an API Management instance	9
Create an Azure API App	12
Test your API App locally.....	18
Publish API App to Azure	21
Manage your API in Azure API Management.....	28
Import the Order API in API Management.....	28
Add the Order API to the Starter product.....	29
Call the Order API from the Developer Portal	31
Lab 2 - Service Bus + Enterprise Integration Pack + On-premises Data Gateway.....	35
Objective	35
Prerequisites	36
Steps.....	36
Install and configure the On-Premises Data Gateway.....	37
Register the gateway in Azure	45
Import the LegacyOrderSystem data in the on-premises SQL Server	47
Create a schema for validation in Visual Studio 2015	52
Create Integration Account.....	56
Upload schema to the Integration Account.....	58
Provision a Logic App	59
Associate the Integration Account with the Logic App.....	60
Build Logic App Definition.....	61
Test the Solution.....	70
Lab 3 - Logic Apps + BizTalk 2016.....	79
Objective	79
Prerequisites	79

 **GLOBAL INTEGRATION BOOTCAMP**

Steps.....	79
Install Logic Apps adapter	80
Created IIS applications for Logic App adapter.....	83
Add Management application	84
Add BizTalk ReceiveService application	85
Create Service Bus Namespace.....	87
BizTalk application	90
Import	90
Create Receive Location from logic app	92
Create Send Port to database	100
Create Receive Location from database	105
Create Send Port to Service Bus	113
Create Logic App which communicates with on premises BizTalk	121
Testing the lab.....	126
Lab 4 - Logic Apps + Azure Functions.....	131
Objective	131
Prerequisites	131
Steps.....	132
Create Storage Account	133
Create Storage Container	135
Create Storage Table.....	137
Provision the Function App.....	142
Building a Function.....	144
Provision a Logic App	149
Building a Logic App Definition	152
Test the Solution	167
IoT Hub + Stream Analytics + DocumentDB + PowerBI (IoT).....	171
Objective	171
Prerequisites	171
Steps.....	171
Create IoT Hub	172
Create DocumentDB	174

 **GLOBAL INTEGRATION BOOTCAMP**

Register device	175
Create simulated device	179
Create Stream Analytics job.....	182
Add input.....	182
Add outputs	183
Set query	187
Testing the lab.....	190

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><##>	gibme17-rgrp-pc01
Service Bus	gib<loc>17-sbus-<ini><##> ↑ 	gibme17-sbus-pc01
Storage Account	gib<loc>17st<ini><##>	gibme17stpc01
API Management	gib<loc>17<ini><##>	gibme17pc01
API App	gib<loc>17<ini><##>ordersapi	gibme17pc01ordersapi
Function App	gib<loc>17-func-<ini><##>	gibme17-func-pc01
Logic Apps Lab 1	gib<loc>17-logic-<ini><##>-validatemaporder	gibme17-logic-pc01-validatemaporder
Logic Apps Lab 2	gib<loc>17-logic-<ini><##>- storeorderonprem	gibme17-logic-pc01-storeorderonprem
Logic Apps Lab 3	gib<loc>17-logic-<ini><##>-procbusinesscustorder	gibme17-logic-pc01-procbusinesscustorder
IoT Hub	gib<loc>17-ioth-<ini><##>	gibme17-ioth-pc01
Document DB	gib<loc>17-docdb-<ini><##>	gibme17-docdb-pc01

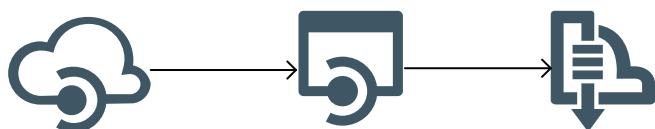
- * Replace <loc> for a 3 char acronym of your location
- * Replace <ini> for your 2 initials
- * Replace <##> for 2 random numbers for uniqueness

GLOBAL INTEGRATION BOOTCAMP

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

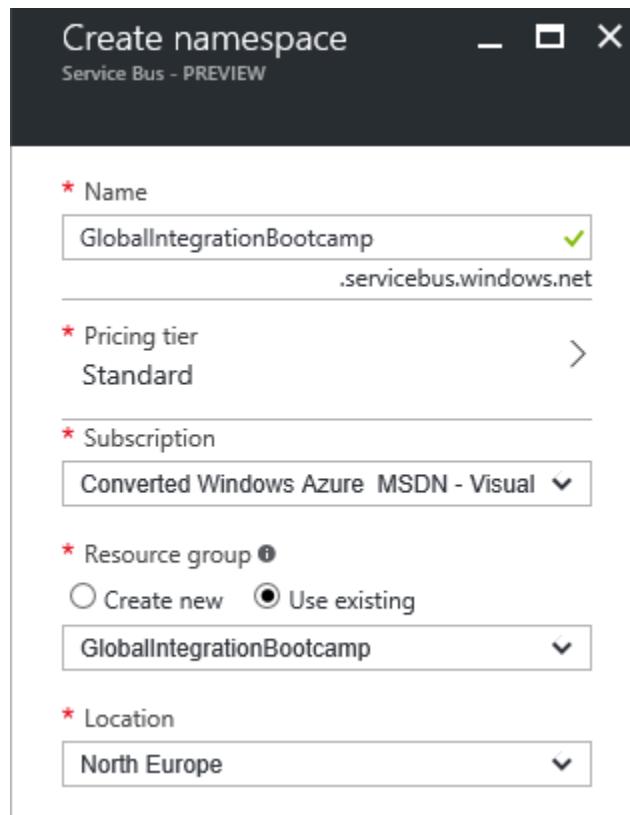
To build the solution in this lab you have to follow the steps described in this section. From a high level view the steps are:

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

GLOBAL INTEGRATION BOOTCAMP

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)



The screenshot shows the 'Create namespace' dialog in the Azure portal. The title bar says 'Create namespace' and 'Service Bus - PREVIEW'. The form fields are as follows:

- Name:** GlobalIntegrationBootcamp (with a green checkmark)
- Pricing tier:** Standard
- Subscription:** Converted Windows Azure MSDN - Visual
- Resource group:** Create new (radio button) / Use existing (radio button selected, checked)
- Resource group:** GlobalIntegrationBootcamp
- Location:** North Europe

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.

GLOBAL INTEGRATION BOOTCAMP

The screenshot shows the Azure portal interface for creating a Shared access policy. On the left, there's a sidebar with 'Pricing tier Standard' and 'Connection Strings' selected. The main area shows a table for 'Shared access policies' with one row selected: 'RootManageSharedAccessKey' with 'Manage, Send, Listen' permissions. To the right, the details for this policy are shown, including its name, claims (Manage, Send, Listen checked), and keys (Primary Key: sWd8S9..., Secondary Key: kYZ760Zt...). Connection strings are also listed: Endpoint=sb://globalintegrationbootcamp.servicebus.windows.net/ and Endpoint=sb://globalintegrationbootcamp.servicebus.windows.net/.DefaultQueue.

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

The screenshot shows the 'Connect to a Service Bus Namespace' dialog. In the 'Service Bus Namespaces' section, the 'Enter connection string...' field contains the previously copied connection string: Endpoint=sb://globalintegrationbootcamp.servicebus.windows.net/;sharedAccessKeyName=RootManageSharedAccessKey;sharedAccessKey=sWd8S9...;etc. The 'Connection Settings' section shows the endpoint, connectivity mode (AutoDetect), and transport type (NetMessaging). At the bottom are 'OK' and 'Cancel' buttons.

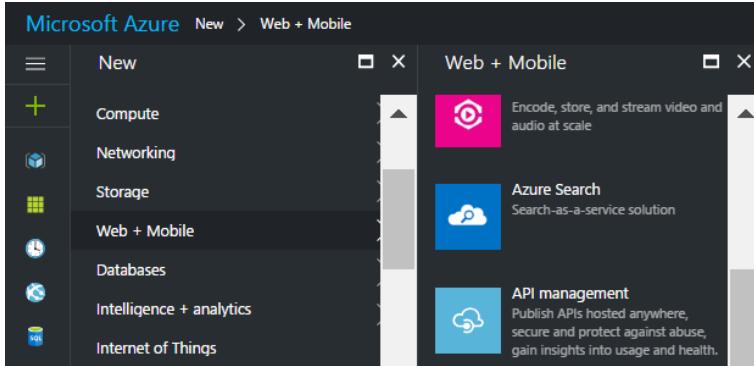
Next add a queue called **neworders**, and keep all the default settings.

GLOBAL INTEGRATION BOOTCAMP

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**.

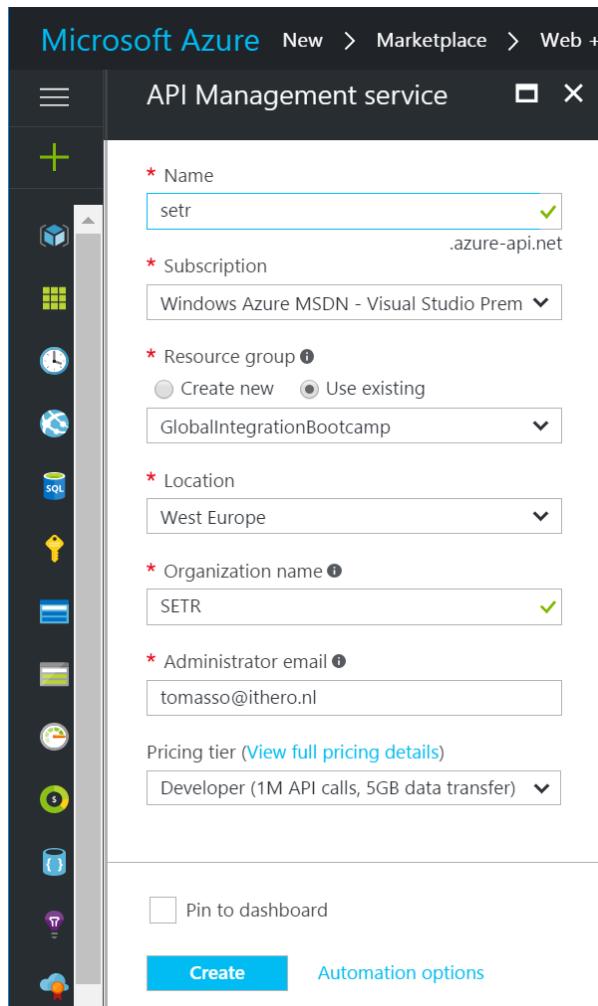


2. For Name, specify a unique sub-domain name to use for the service URL.
Choose the desired Subscription, Resource group and Location for your service instance.
Enter SETR for the Organization Name, and enter your email address in the Administrator E-Mail field.

Note

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

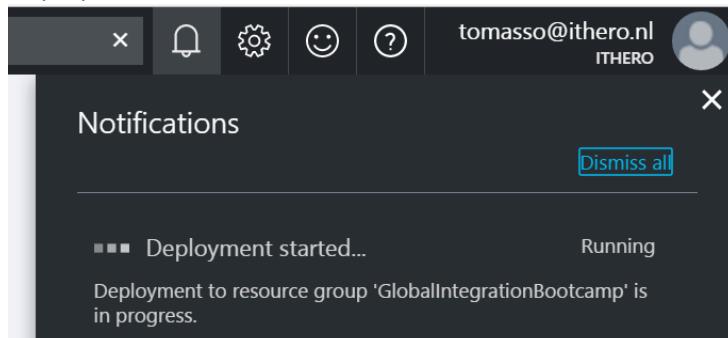
GLOBAL INTEGRATION BOOTCAMP



Note

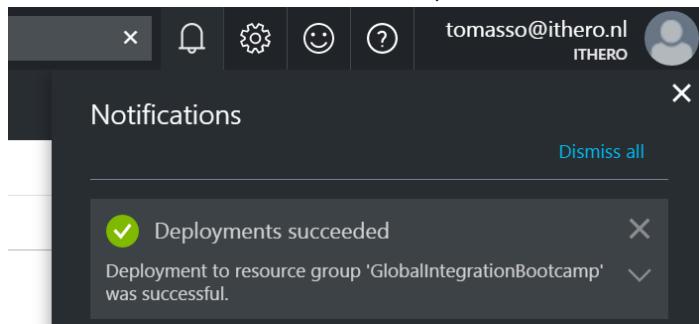
API Management service instances are available in three tiers: Developer, Standard, and Premium. You can complete this lab by using the Developer tier.

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



GLOBAL INTEGRATION BOOTCAMP

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



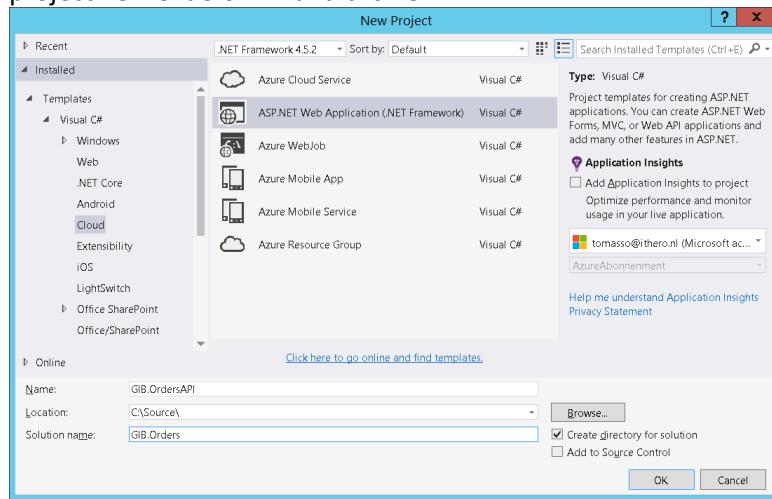
The next step is to import the Order API.

GLOBAL INTEGRATION BOOTCAMP

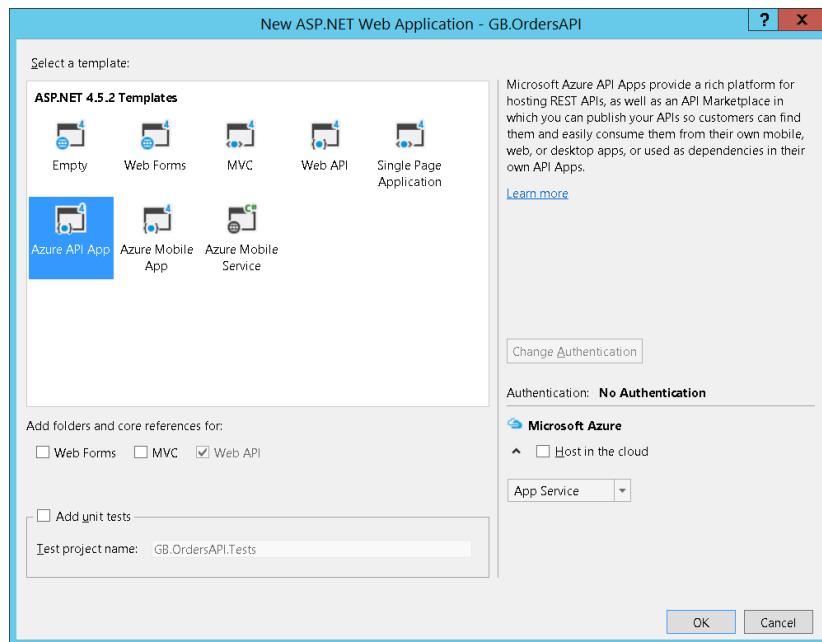
Create an Azure API App

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

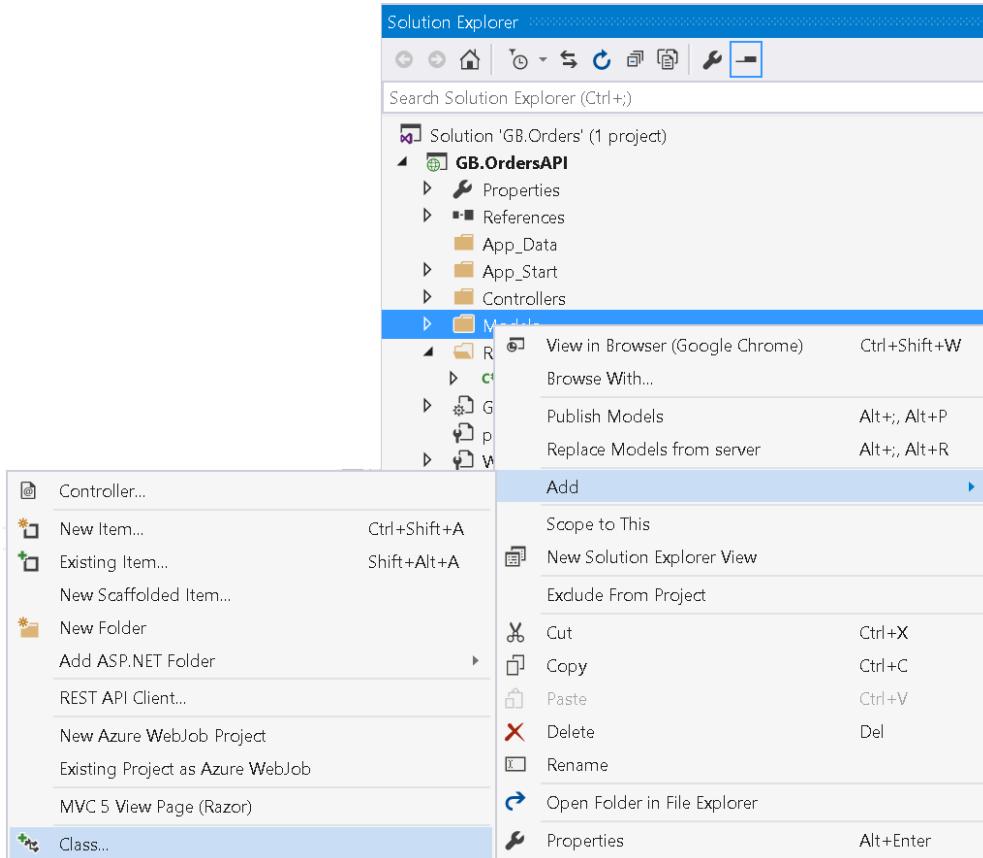
GLOBAL INTEGRATION BOOTCAMP

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=RootManageSharedAccessK
ey;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";

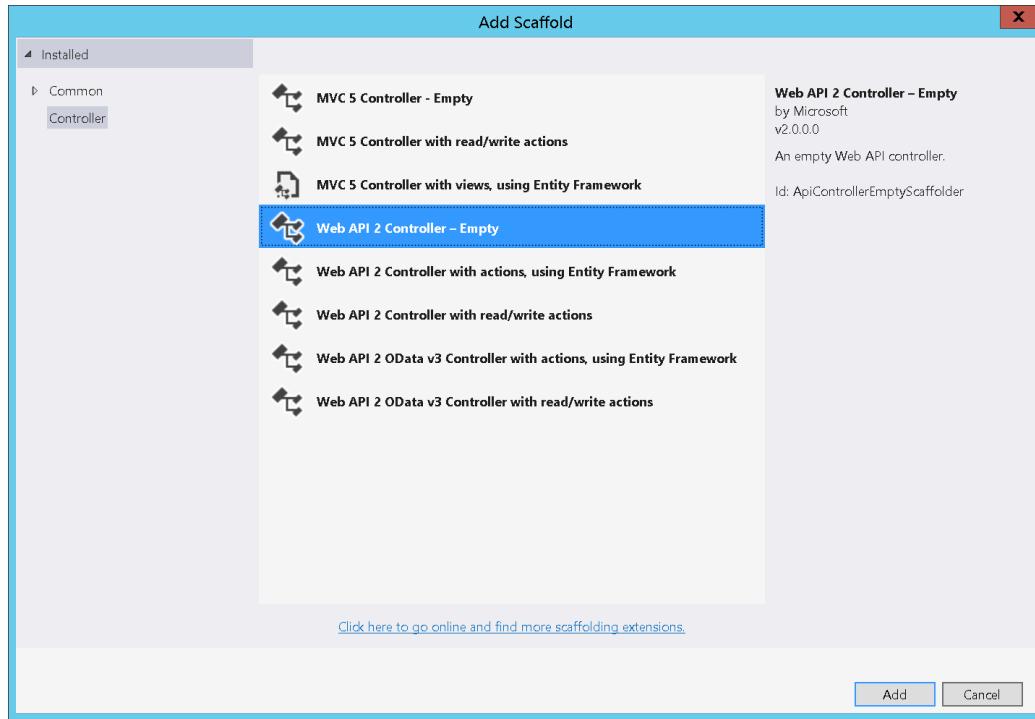
        var client =
QueueClient.CreateFromConnectionString(connectionString, queueName);
        var message = new BrokeredMessage(xmlObject);

        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**.

GLOBAL INTEGRATION BOOTCAMP



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 repository = new
ServiceBusRepository();

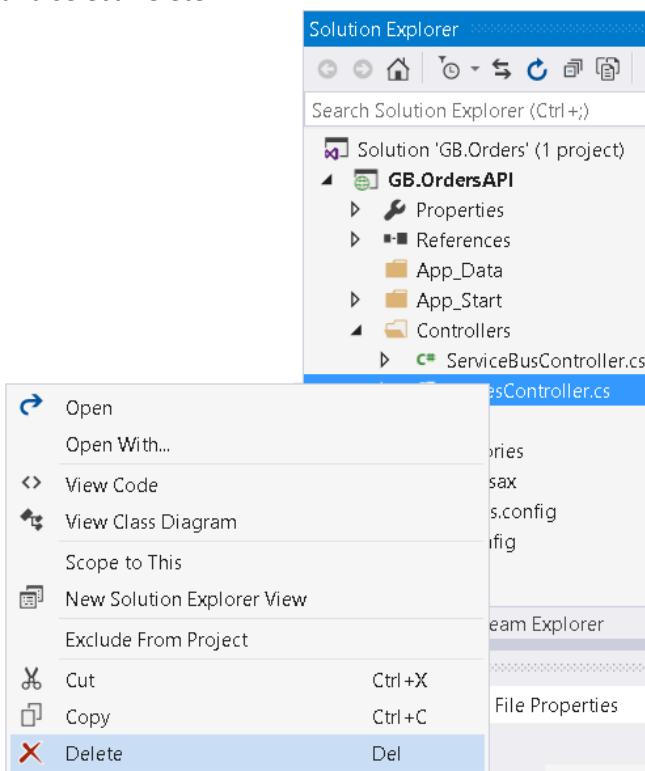
```

GLOBAL INTEGRATION BOOTCAMP

```
        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"  
  }  
}
```

GLOBAL INTEGRATION BOOTCAMP

6. Click **Try it out**.

The screenshot shows the Swagger UI interface for a POST operation to the '/api/Orders' endpoint. The request body is set to a JSON object:

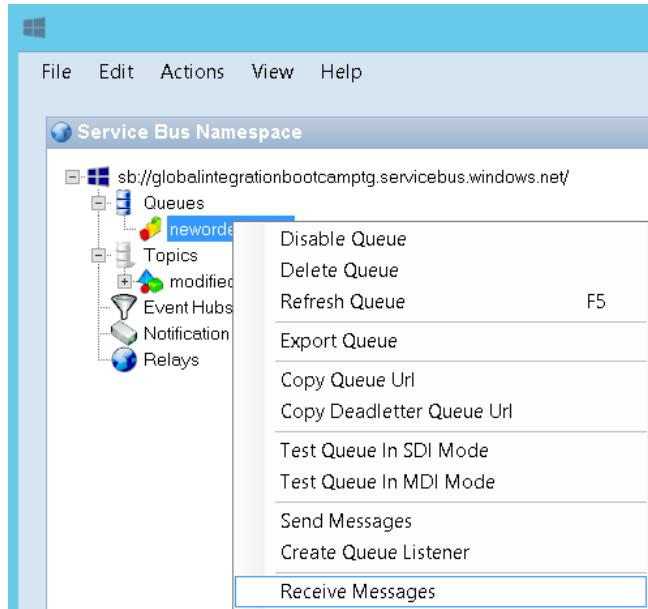
```
{  
  "Order": {  
    "Customer": {  
      "CustomerNumber": "A1000"  
    },  
    "Products": [  
      {}  
    ]  
  }  
}
```

The response content type is set to application/json. The response messages table shows a 201 Created status code with the reason 'Created'. A 'Try it out!' button is visible at the bottom left.

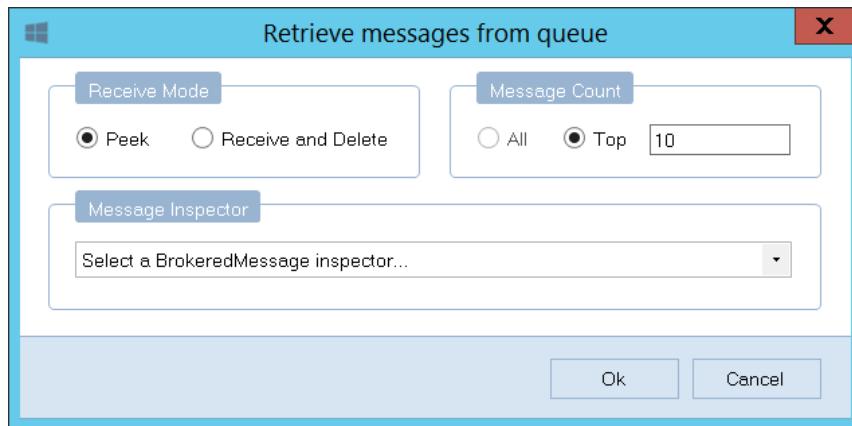
The Order API returns an HTTP 200 response code that indicates success.

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.

GLOBAL INTEGRATION BOOTCAMP

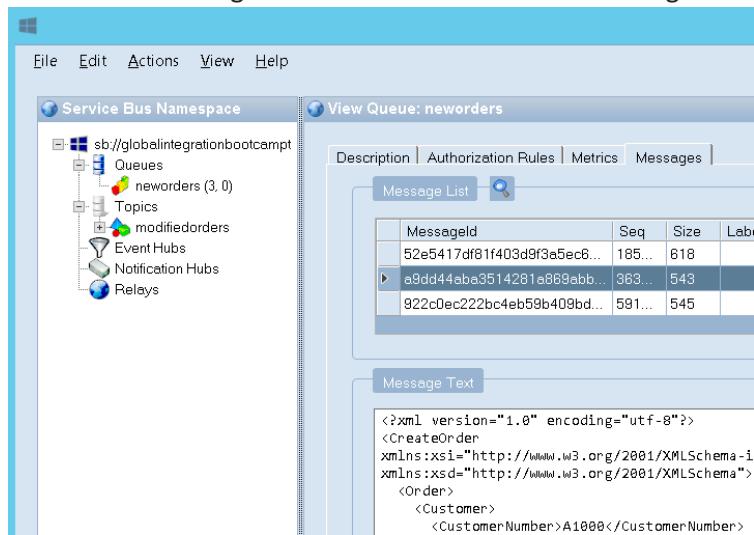


8. Select the Peek radio button and click on Ok.



GLOBAL INTEGRATION BOOTCAMP

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



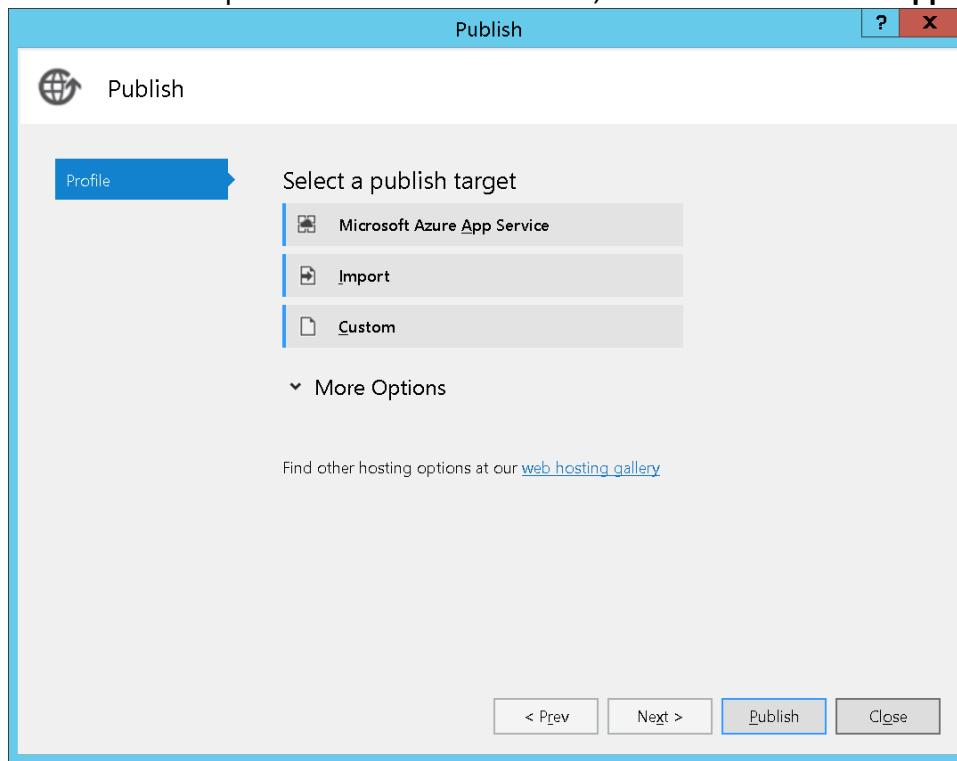
Publish API App to Azure

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.

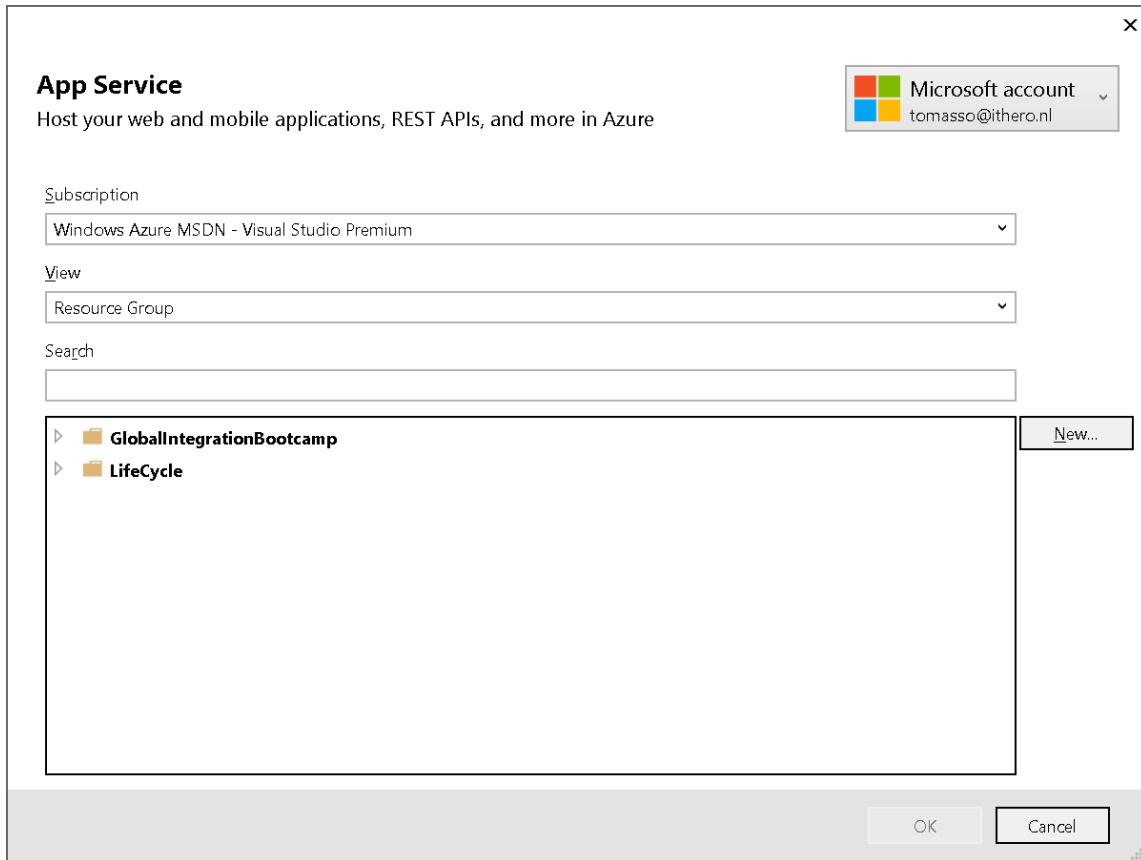
1. In **Solution Explorer**, right-click the Order API project, and then click **Publish**.

GLOBAL INTEGRATION BOOTCAMP

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

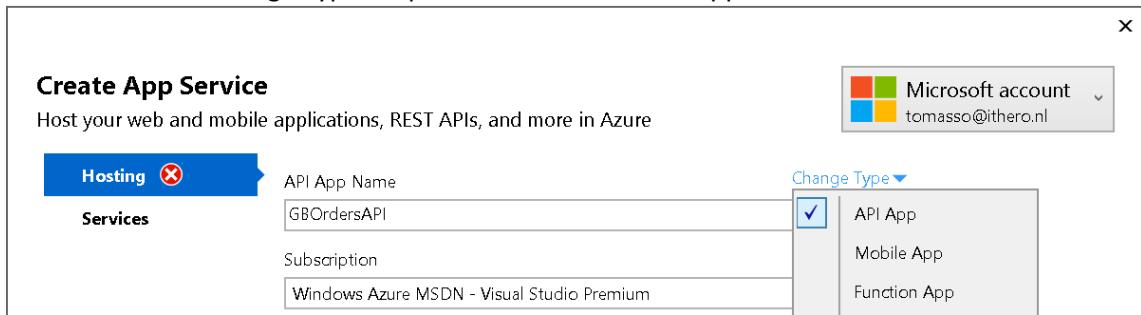


3. Sign in to your Azure account if you have not already done so, or refresh your credentials if they're expired.
4. In the App Service dialog box, choose the Azure Subscription you want to use, and then click New.



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.



- 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.



GLOBAL INTEGRATION BOOTCAMP

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.

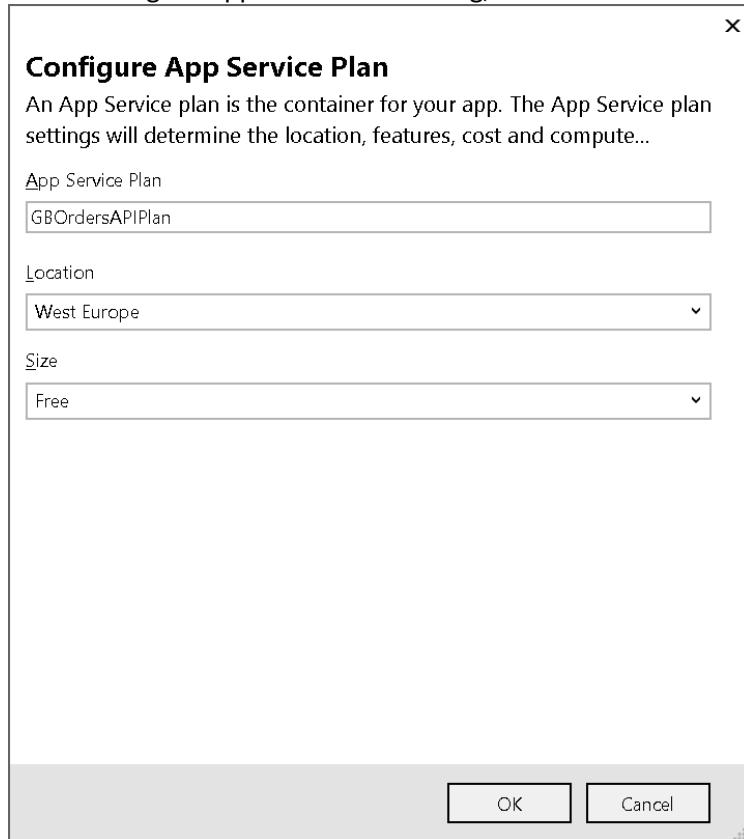
This setting specifies which Azure datacenter your app will run in. Choose a location close to you to minimize [latency](#).

10. In the Size drop-down, click Free.

For this tutorial, the free pricing tier will provide sufficient performance.

 **GLOBAL INTEGRATION BOOTCAMP**

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



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

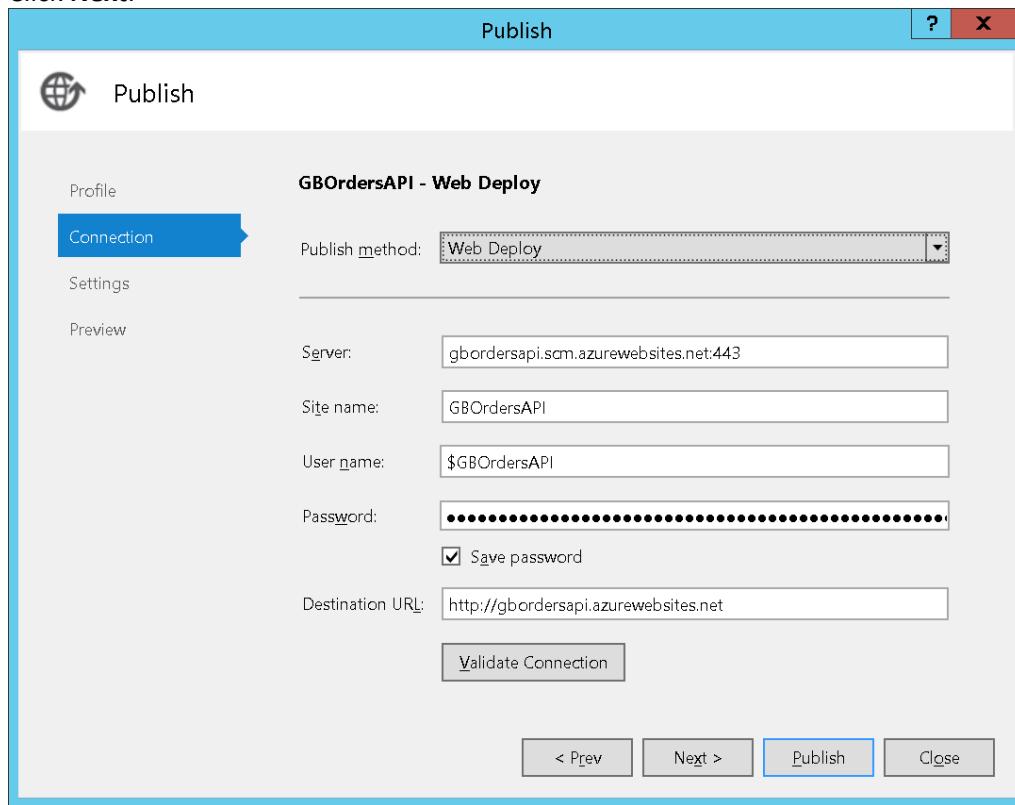
Visual Studio creates the API app and a publish profile that has all of the required settings for the API app. Then it opens the **Publish Web** wizard, which you'll use to deploy the project.

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**).

GLOBAL INTEGRATION BOOTCAMP

13. Click Next.



14. Click Next.

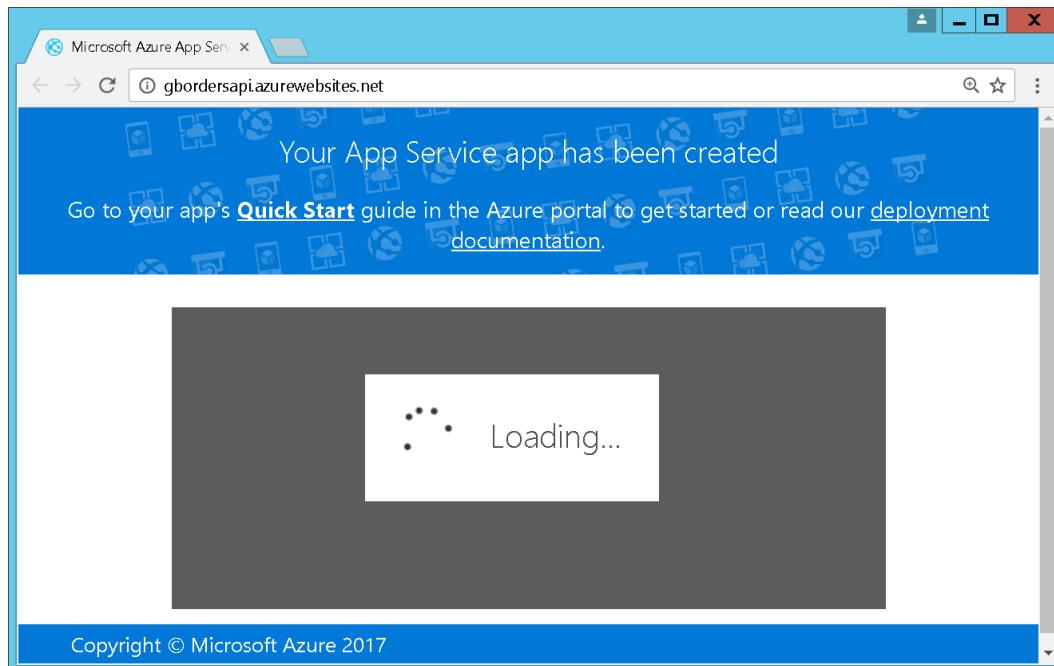
15. Click Publish.

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.

The screenshot shows the 'Output' window with the 'Build' dropdown selected. The window displays deployment logs:

```
1> GB.OrdersAPI -> C:\Source\Motion10\GB.Orders\GB.OrdersAPI\bin\GB.OrdersAPI.dll
2>----- Publish started: Project: GB.OrdersAPI, Configuration: Release Any CPU -----
2>Transformed Web.config using C:\Source\Motion10\GB.Orders\GB.OrdersAPI\Web.Release.config into obj\Release\
2>Auto ConnectionString Transformed obj\Release\TransformWebConfig\transformed\Web.config into obj\Release\cs
2>Copying all files to temporary location below for package/publish:
2>obj\Release\Package\PackageTmp.
2>Start Web Deploy Publish the Application/package to https://gbordersapi.scm.azurewebsites.net/msdeploy.axd?
2>Adding ACL's for path (GBOrdersAPI)
2>Adding ACL's for path (GBOrdersAPI)
2>Publish Succeeded.
2>Web App was published successfully http://gbordersapi.azurewebsites.net/
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
===== Publish: 1 succeeded, 0 failed, 0 skipped =====
```

GLOBAL INTEGRATION BOOTCAMP



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](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.

GLOBAL INTEGRATION BOOTCAMP

Manage your API in Azure API Management

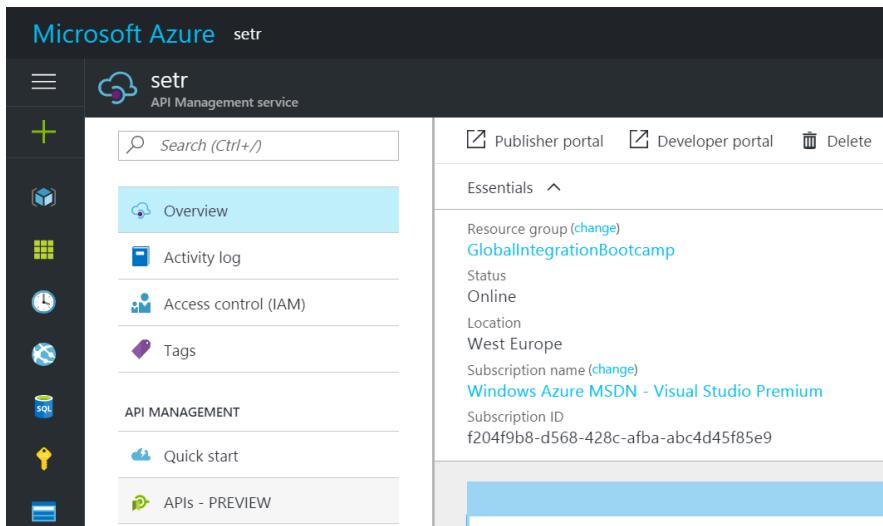
Import the Order API in API Management

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.

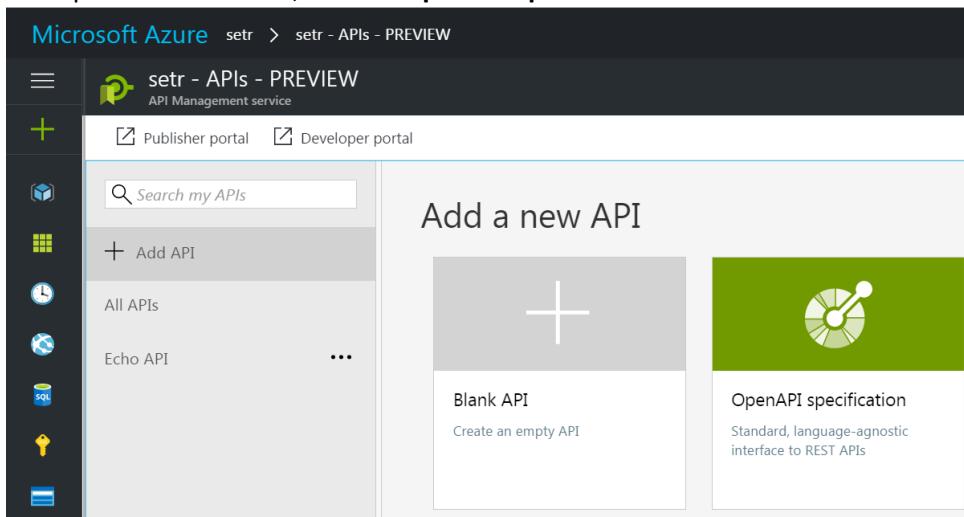
Note

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**.



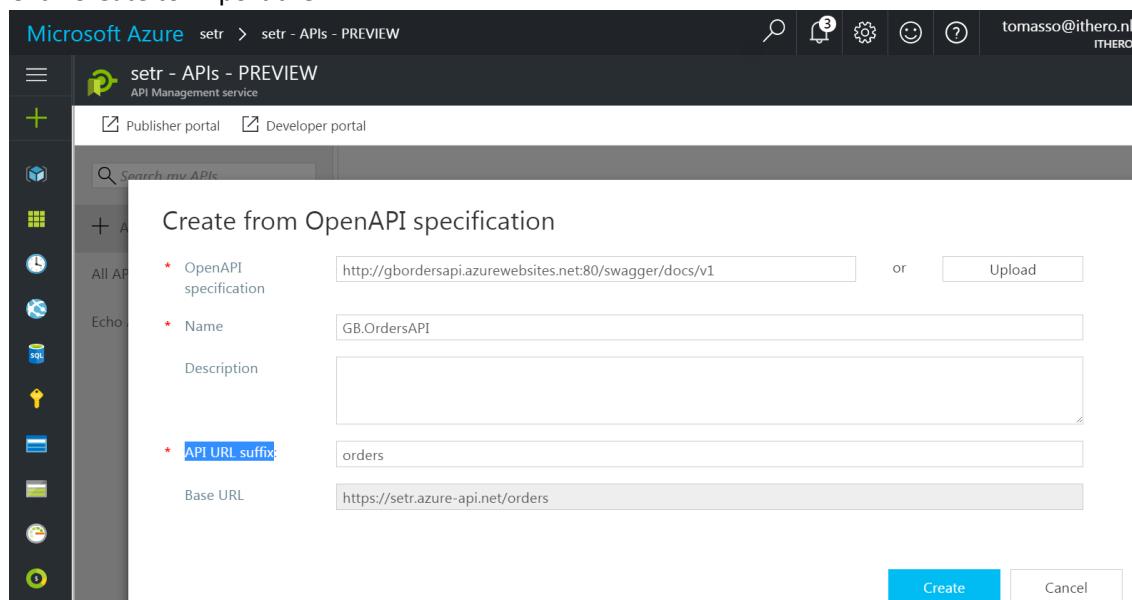
GLOBAL INTEGRATION BOOTCAMP

3. Perform the following steps to configure the Order API:

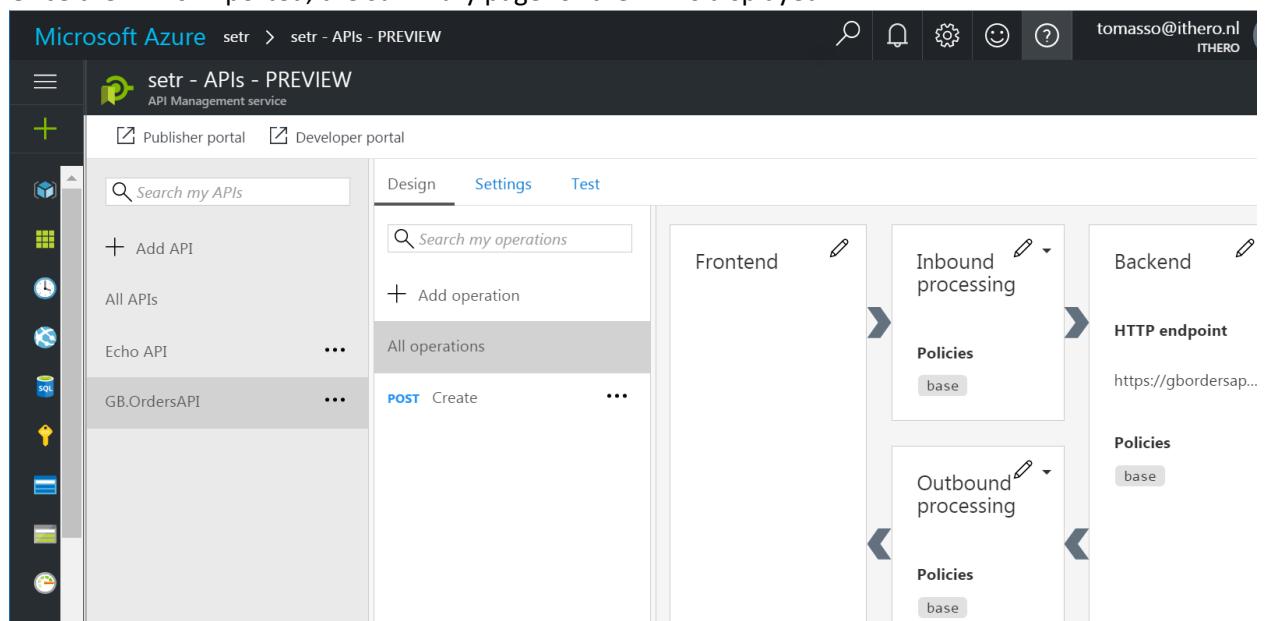
Enter the metadata swagger url from the Order API into the OpenAPI specification text box, and click the Tab button to select the next field.

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**

GLOBAL INTEGRATION BOOTCAMP

- **Unlimited**

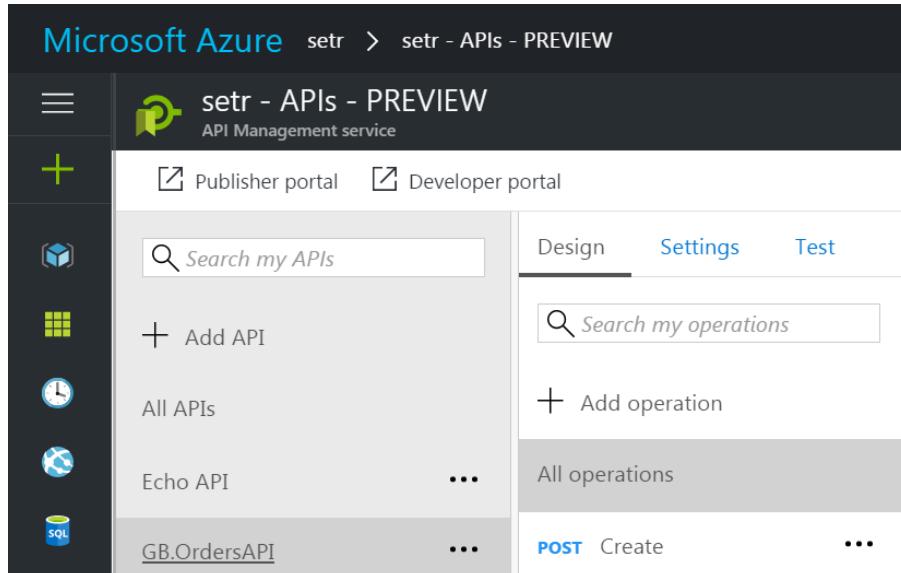
In order to make calls to an API, developers must first subscribe to a product that gives them access to it. Developers can subscribe to products in the developer portal, or administrators can subscribe developers to products in the publisher portal. You are an administrator since you created the API 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.

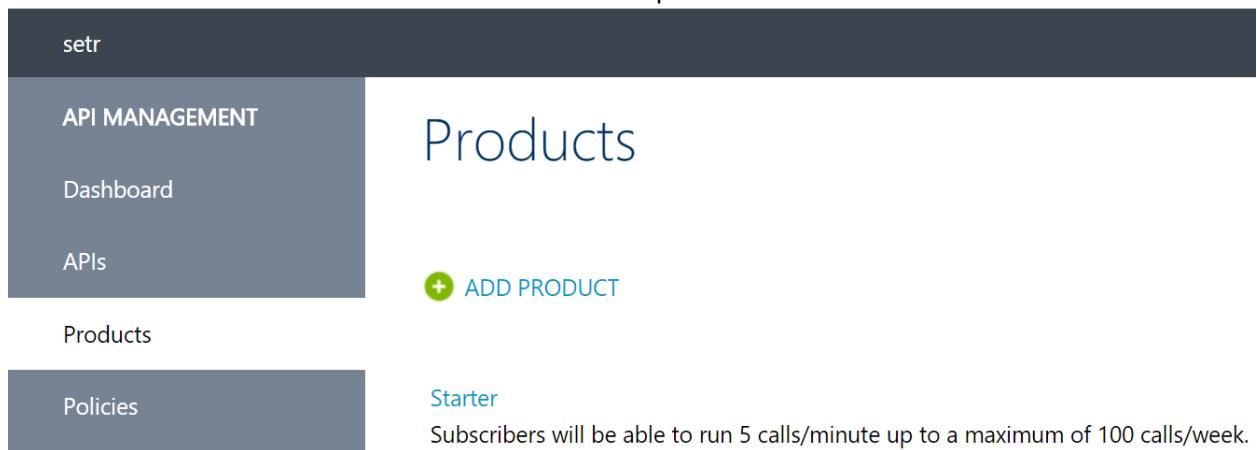
Note

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.



GLOBAL INTEGRATION BOOTCAMP

3. The **Starter Product** page contains four links for configuration: **Summary**, **Settings**, **Visibility**, and **Subscribers**. The **Summary** tab is where you can add an API.

Click on the ADD API TO PRODUCT button

Product - Starter



Published Subscribers will be able to run 5 calls
Subscription approvals not required

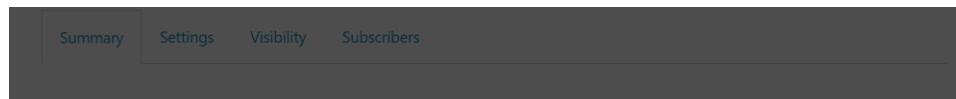
UNPUBLISH
 DELETE

APIs

The following APIs are part of your product:

ADD API TO PRODUCT

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



Add APIs to the product

GB.OrdersAPI

Save **Cancel**

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.

GLOBAL INTEGRATION BOOTCAMP

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

The screenshot shows a dark-themed interface. At the top right, there is a navigation bar with links for "Developer portal", "Help", "Sign out", and a user icon. On the left, there is a sidebar with a "setr" logo and two main sections: "API MANAGEMENT" and "Dashboard". The main content area is titled "Product - Starter".

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

The screenshot shows the "APIS" section of the interface. The main title is "SETR API". Below it, there is a navigation bar with links for "HOME", "APIS", "PRODUCTS", "APPLICATIONS", and "ISSUES".

APIs

[Echo API](#)

[GB.OrdersAPI](#)

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**.

The screenshot shows the "Create" operation for the "GB.OrdersAPI". It features a blue header bar with "POST" and "Create" buttons. Below the header, the title "Create" is displayed, followed by a "Try it" button. A "Request URL" field contains the value "https://setr.azure-api.net/orders/api/Orders".

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

GLOBAL INTEGRATION BOOTCAMP

```
2 "Order": {  
3   "Customer": {  
4     "CustomerNumber": "DEV01"  
5   },  
6   "Products": [  
7     {  
8       "ProductNumber": 10000,  
9       "Amount": 1  
10      }  
11    ],  
12   "OrderedDateTime": "2017-03-07T12:03:18.605Z"  
13 }  
14 }
```

Authorization

Subscription key

Primary-9954...

x ▾

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

GLOBAL INTEGRATION BOOTCAMP

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

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

Send

Response Trace

Response status

200 OK

Response latency

7366 ms

Response content

```
Pragma: no-cache  
Ocp-Apim-Trace-Location: https://apimgmtstcp3vhxqgaqrmda.blob.  
v=2015-07-08&sr=b&sig=GpFjT6zKPOmyqJubvn%2FtkwOJSbjItN79%2Ff0pE  
29d5e142db8a6e8fa
```

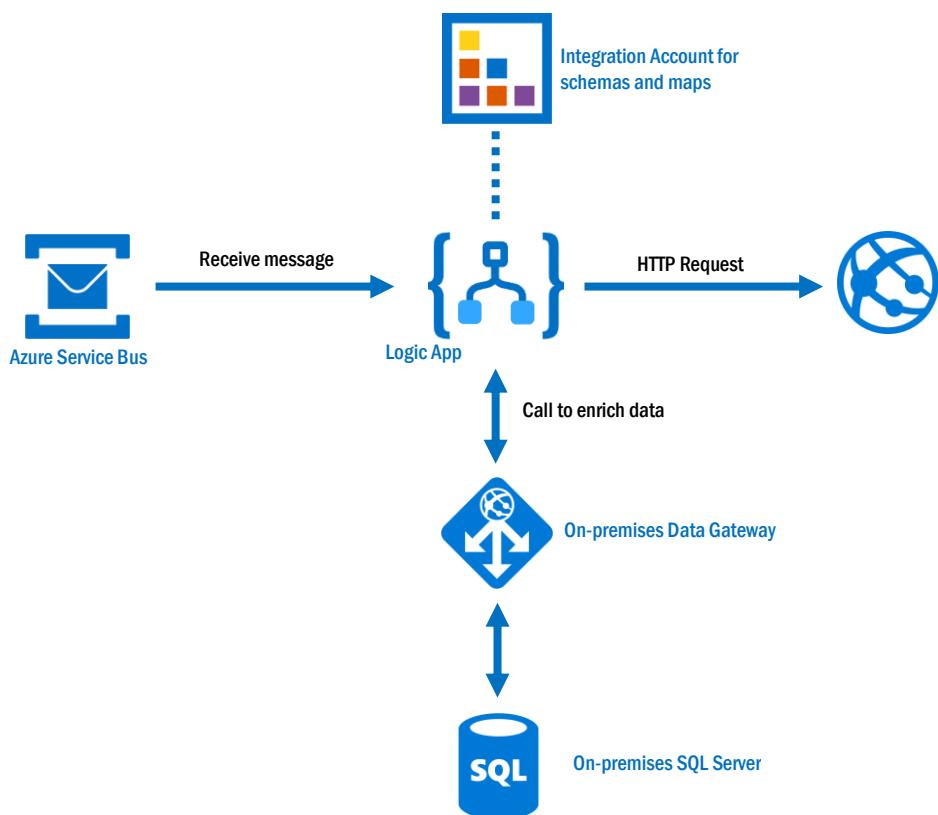
GLOBAL INTEGRATION BOOTCAMP

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

GLOBAL INTEGRATION BOOTCAMP

Install and configure the On-Premises Data Gateway

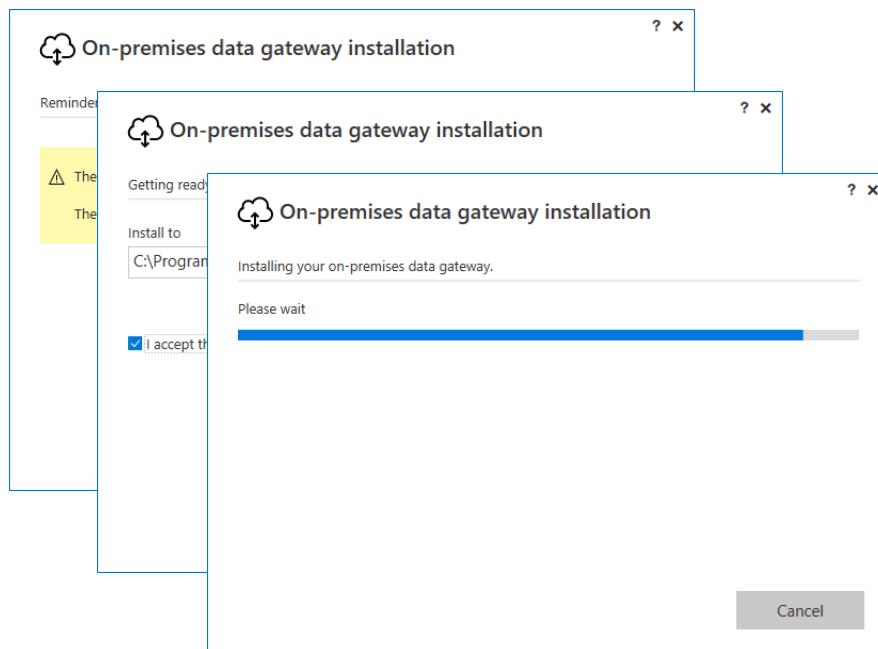
Since we are going to use the On-Premises Data Gateway, the gateway should be installed and configured first.

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.

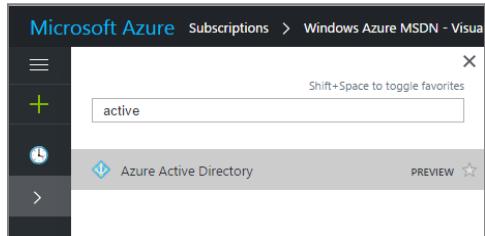
If you are using a work or school account, you can proceed to **step II**.

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

GLOBAL INTEGRATION BOOTCAMP

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 onmicrosoft.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.

GLOBAL INTEGRATION BOOTCAMP

The screenshot shows the Azure Active Directory Management Experience (Preview) interface. The left sidebar has 'Overview' selected. The main area displays a chart titled 'Enterprise applications' showing 'USERS SIGN INS' from Jan 15 to Feb 5. The right side shows a 'User' configuration dialog with fields for Name (set to 'On-Premises Data Gateway'), User name (set to 'opdgw@examplehotmail.onmicrosoft.com'), and Profile (set to 'Not configured').

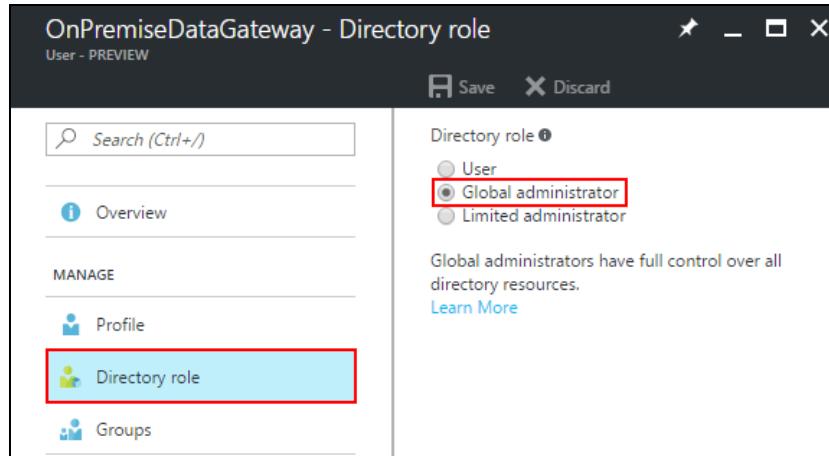
- Set the appropriate rights for the user.

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).

The screenshot shows the 'Users and groups - All users' list in the Azure Active Directory Management Experience (Preview). The 'All users' option in the left sidebar is selected. The main area lists users, with one row for 'OnPremiseDataGateway' highlighted by a red box. The user details show the name 'OnPremiseDataGateway' and the user name 'opdgw@examplehotmail.onmicrosoft.com'.

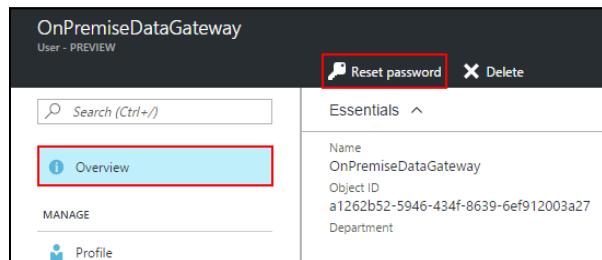
GLOBAL INTEGRATION BOOTCAMP



- 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.

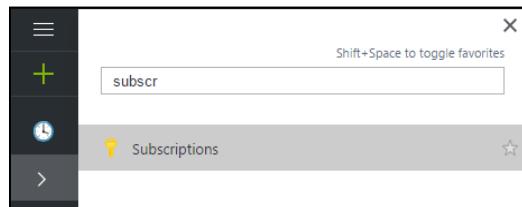
Make sure you copy it or save it for later!



- 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.



GLOBAL INTEGRATION BOOTCAMP

USER	ROLE	ACCESS	...
RF	Rob Fox example@hotmail.com	User Access Admin.. Inherited	...
Subscription admins	Owner	Inherited	...

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.

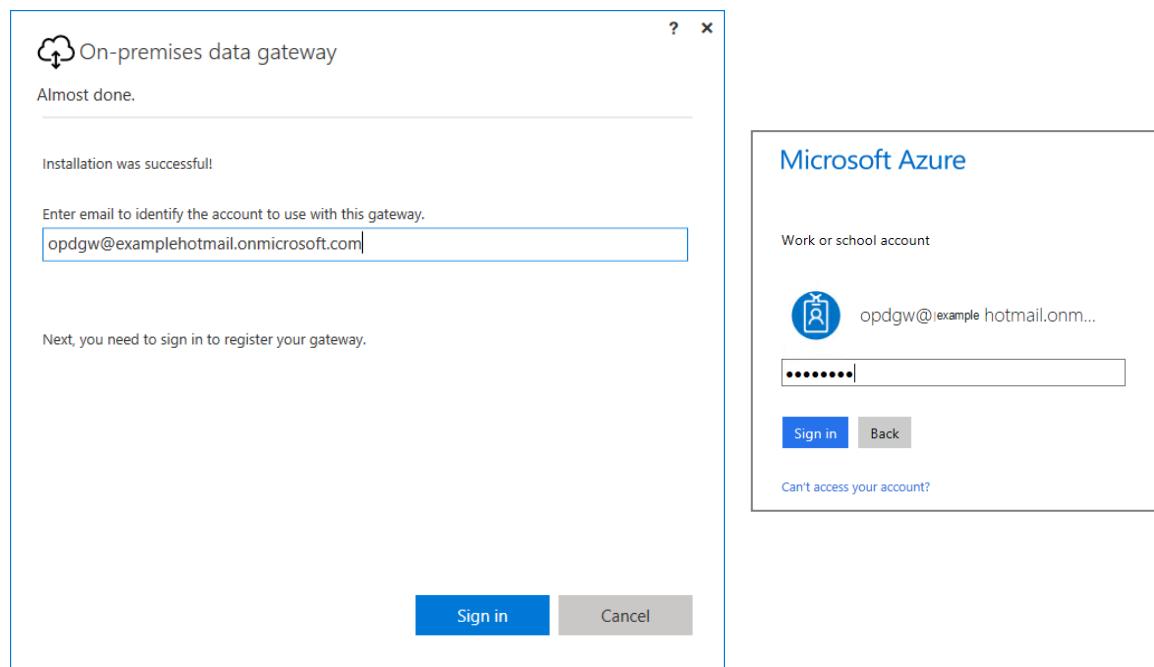
To change the password, open an in-cognito (or in private) browser window, or you can choose to sign out of the portal. After opening the new in-cognito browser window, or after signing out, navigate to <https://portal.azure.com> and login with the On-Premises Data Gateway user by supplying the e-mail 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

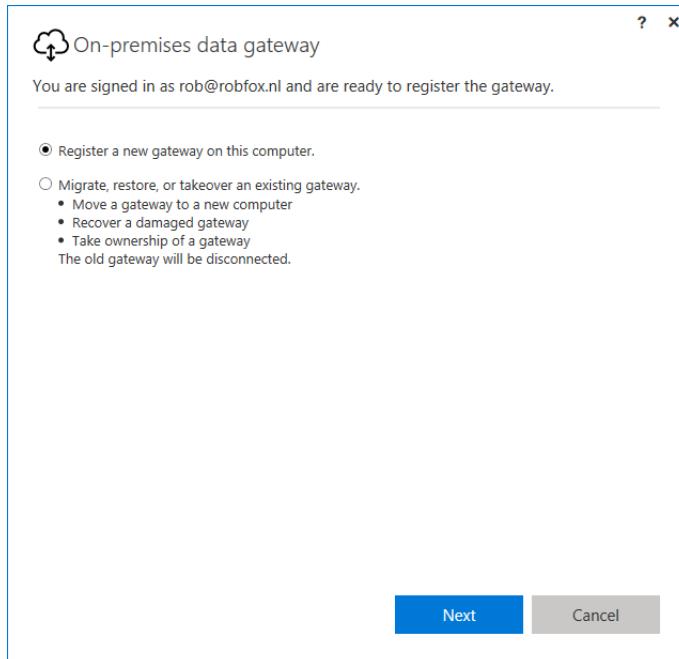
- a. On the on-premises machine containing SQL server, start the On-Premises Data Gateway configuration. This should start automatically after installation.

Login with your work or school account.



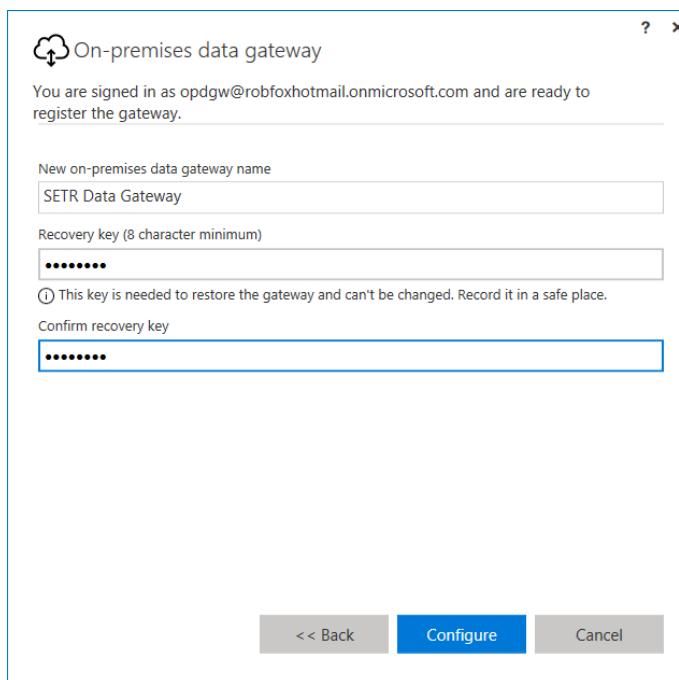
- b. On the second screen choose **Register a new Gateway on this computer**.

GLOBAL INTEGRATION BOOTCAMP



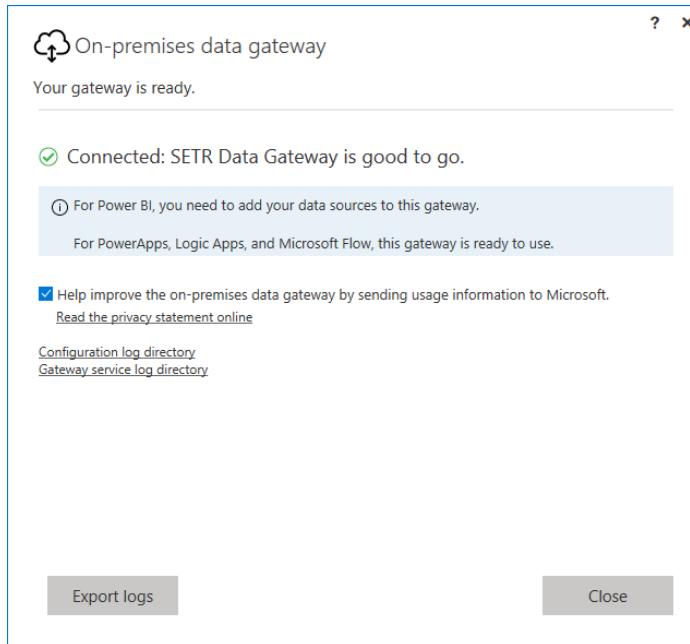
- c. In the next screen enter the **name of your gateway** and enter a **recovery key**. These values are totally up to you.

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



GLOBAL INTEGRATION BOOTCAMP

- 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.



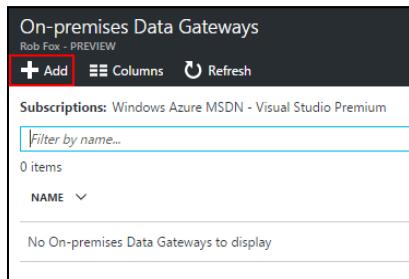
GLOBAL INTEGRATION BOOTCAMP

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.

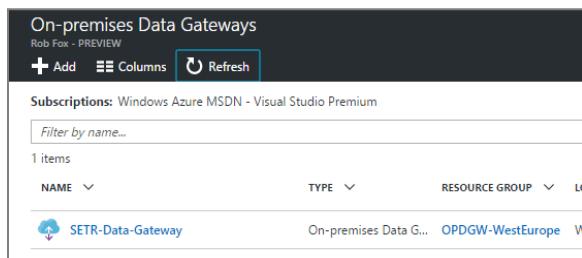
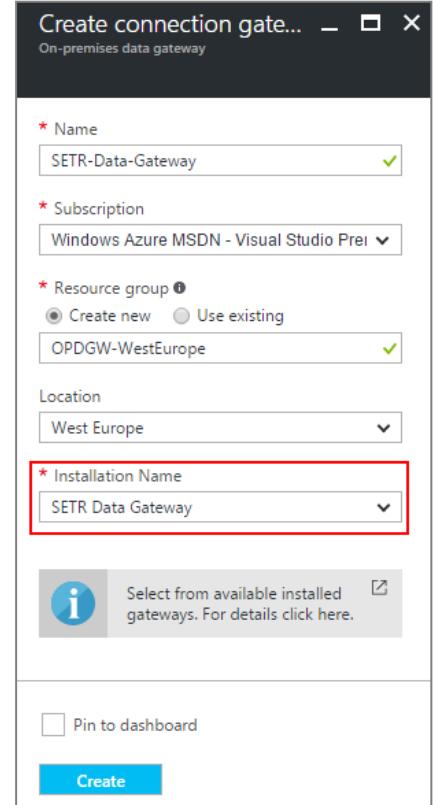


3. Now give your On-Premises Data Gateway a suitable **name** to identify it in Azure. Select your **subscription** and a **Resource group** (you can add a new one if you want). Select the desired **Location**. I.e. don't select West-Europe if you are in Australia ☺. Please note that the Gateway should 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.



GLOBAL INTEGRATION BOOTCAMP

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.

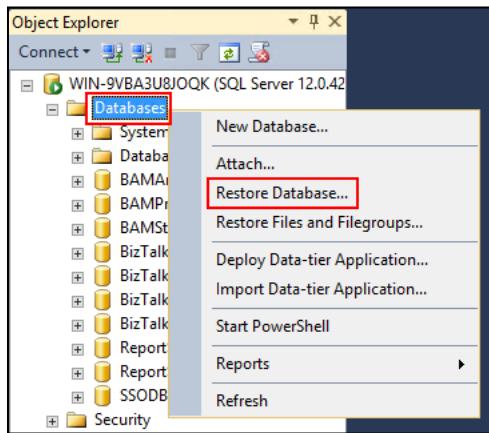
The data used for this lab can be downloaded from [here](#).

After downloading, unpack the zip file on the machine with SQL Server installed.

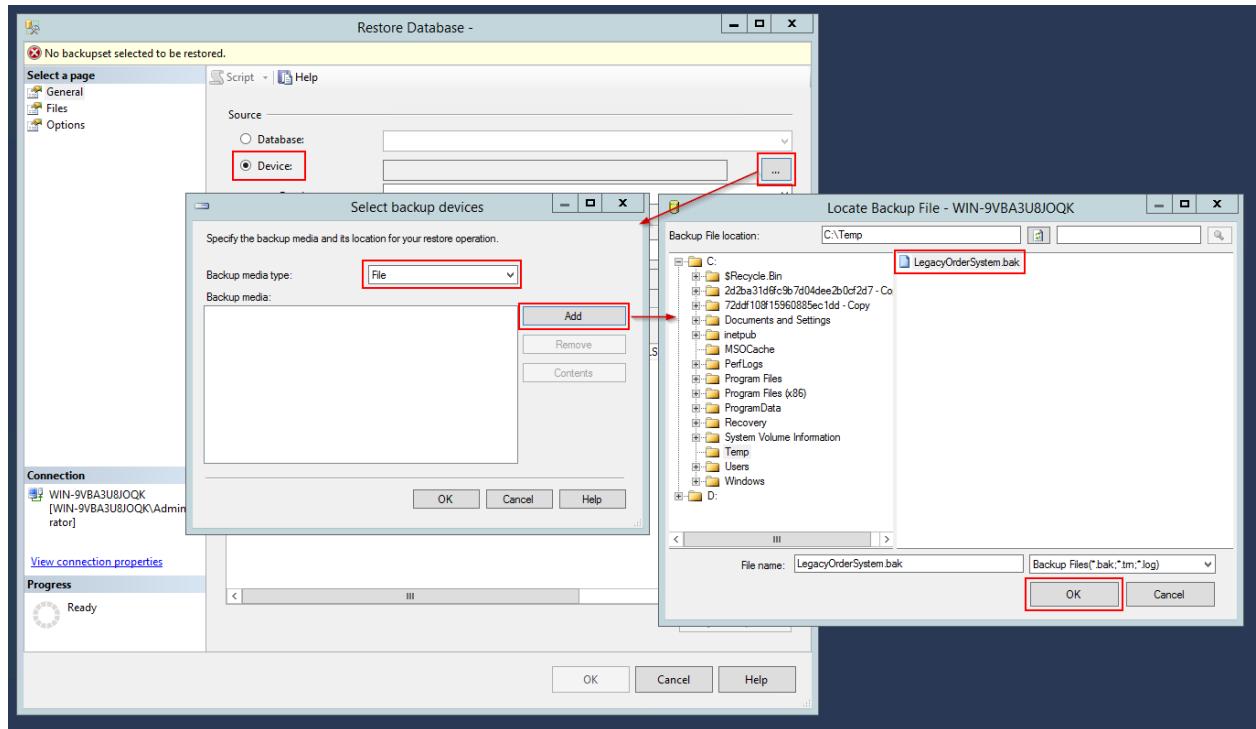
2. Restore the backup

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

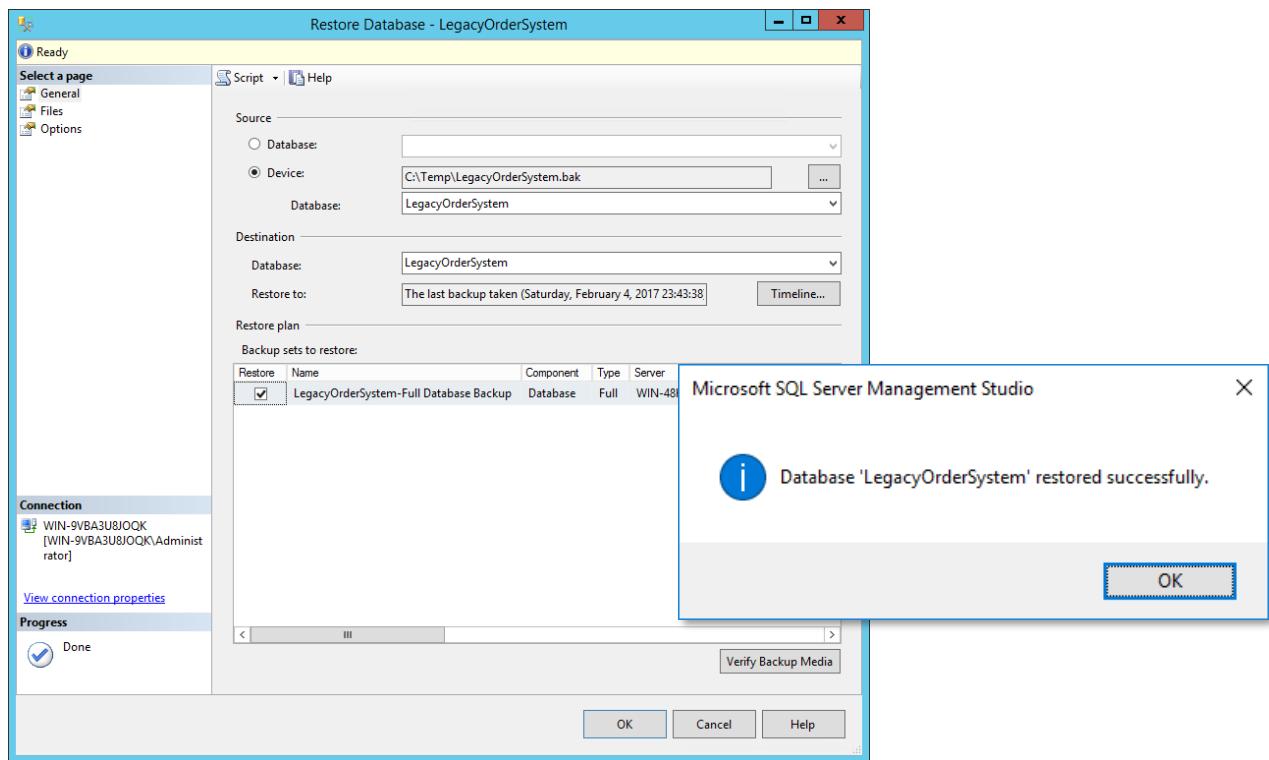
- b. In Object Explorer, **click right on Databases** and select restore



- c. Now, in the General page, select **Device** and **click the ellipses (...)**. 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.

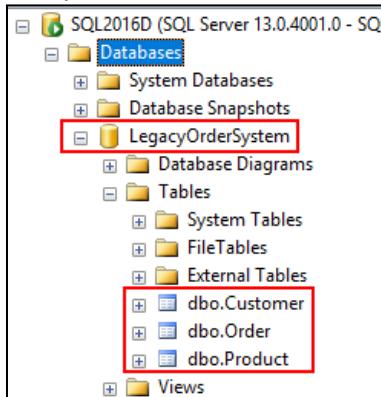


- d. The result should look like the window below. Now click **OK** to restore the backup.



GLOBAL INTEGRATION BOOTCAMP

- 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
```



GLOBAL INTEGRATION BOOTCAMP

```

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

```



GLOBAL INTEGRATION BOOTCAMP

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

END

GLOBAL INTEGRATION BOOTCAMP

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:

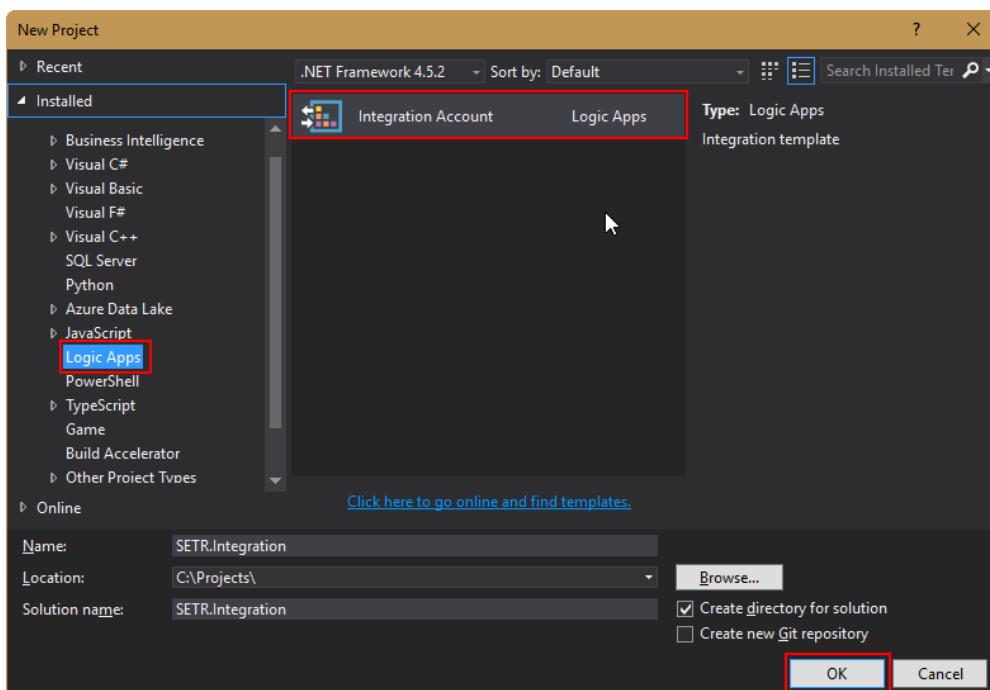
- <https://www.visualstudio.com/downloads/>

To download the Enterprise Integration tools, go here:

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

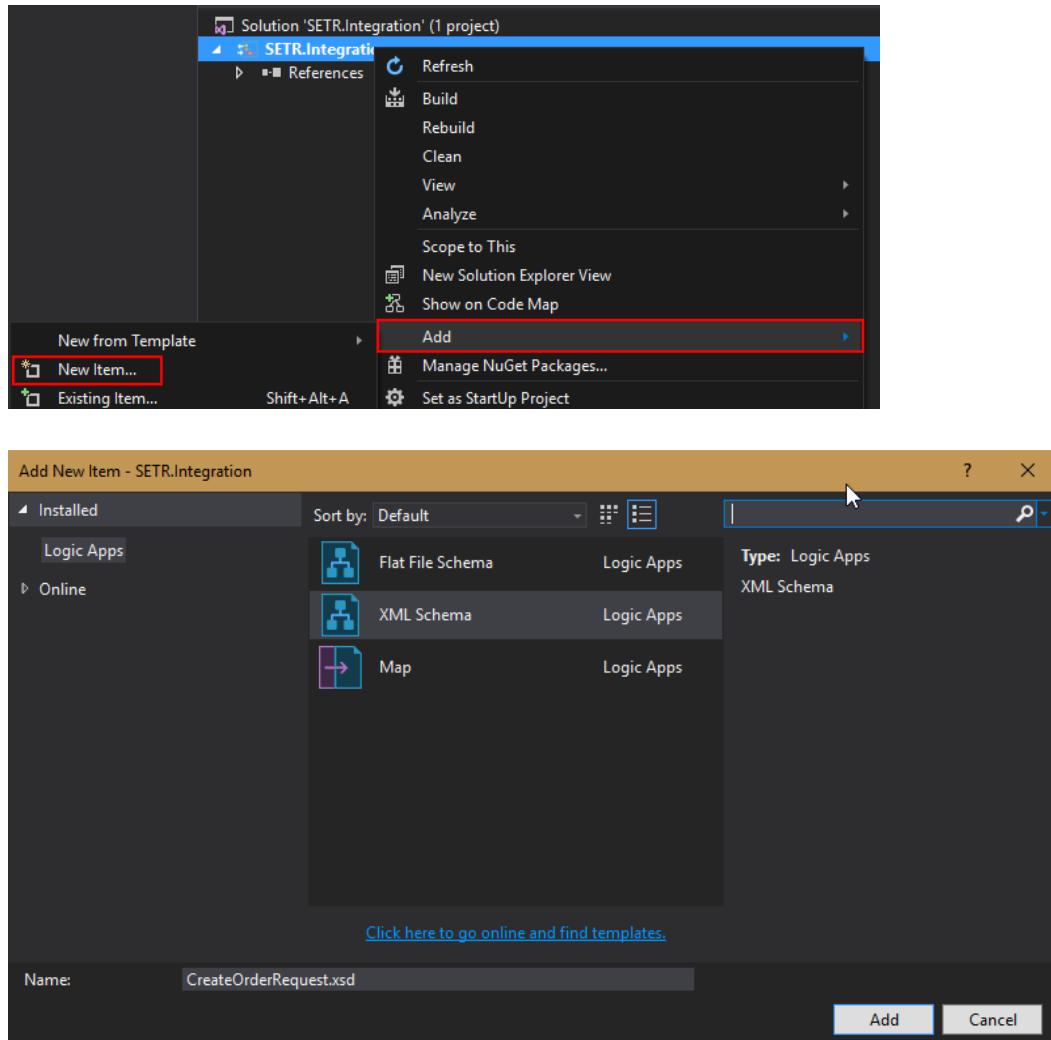
In this lab, it's only intended to get familiar with the Integration accounts. A schema will be created to validate the message that's being received from the service bus queue.

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.

GLOBAL INTEGRATION BOOTCAMP



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>
    </Products>
  </Order>
</CreateOrder>
```

```

        </Product>
    <Product>
        <ProductNumber>Integer</ProductNumber>
        <Amount>2</Amount>
    </Product>
</Products>
<!-- Mandatory --&gt;
&lt;OrderedDateTime&gt;DateTime&lt;/OrderedDateTime&gt;
&lt;/Order&gt;
&lt;/CreateOrder&gt;
</pre>

```

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: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:schema>

```

```
</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.

GLOBAL INTEGRATION BOOTCAMP

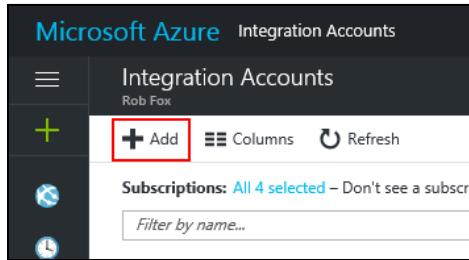
Create Integration Account

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.

GLOBAL INTEGRATION BOOTCAMP

Integration Account X

* Name: SETR ✓

* Subscription: Windows Azure MSDN - Visual Studio Prem... ▼

* Resource group: Create new Use existing
IA-WE-SETR ✓

* Location: West Europe ▼

* Pricing Tier: Free ▼

Pin to dashboard

Create Automation options

5. Navigate to the Integration Account to verify the deployment

SETR Integration Account

Search (Ctrl+F)

Overview

Access control (IAM) Tags

SETTINGS

- Callback URL
- Schemas
- Maps
- Certificates
- Partners
- Agreements
- Properties
- Locks
- Automation script

MONITORING

Diagnostics logs

Delete

Essentials ^

Resource group (change)
[IA-WE-SETR](#)

Name: SETR

Location: West Europe

Subscription name (change)
[Windows Azure MSDN - Visual Studio Premi...](#)

Subscription ID

Components

Schemas	Maps	Certificates
0	0	0
Partners	Agreements	
0	0	

GLOBAL INTEGRATION BOOTCAMP

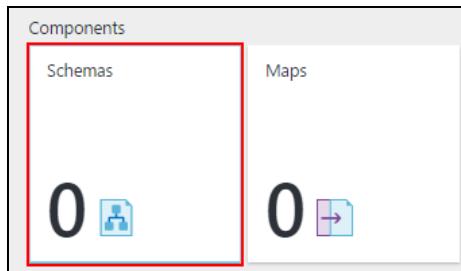
Upload schema to the Integration Account

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.

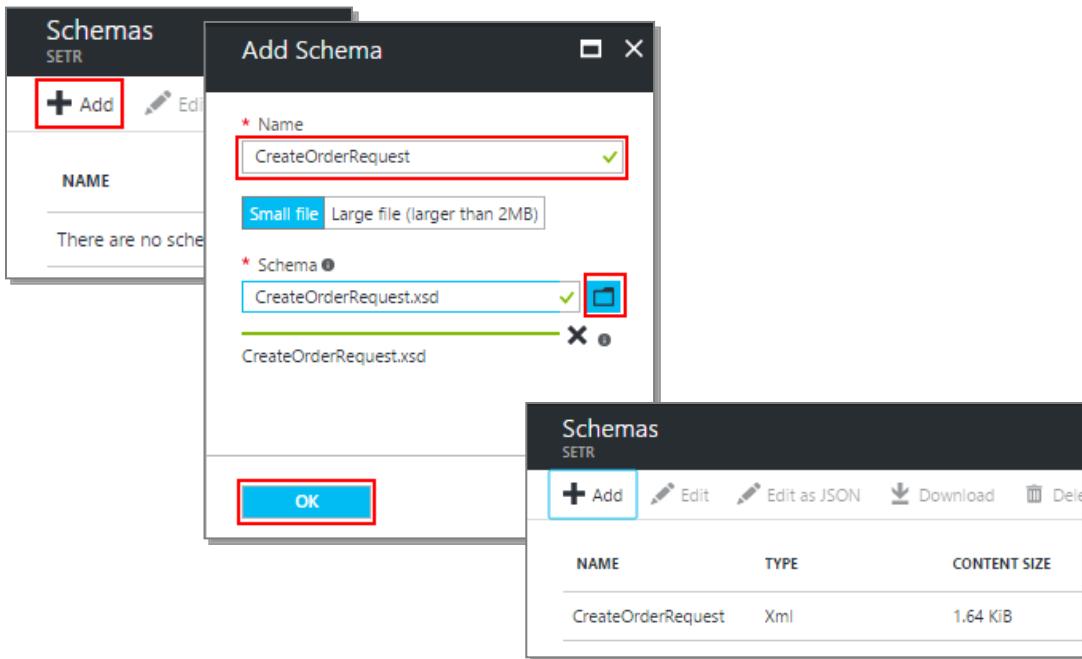
The only artifact we've created is the XSD that is going to be used for validation of the data received.

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.



GLOBAL INTEGRATION BOOTCAMP

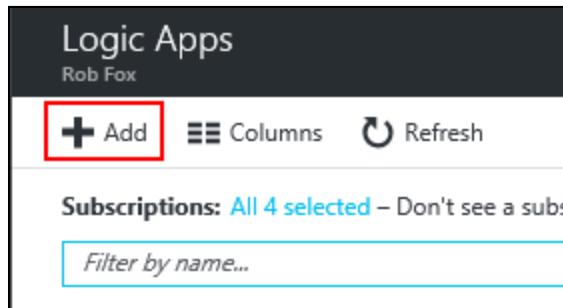
Provision a Logic App

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

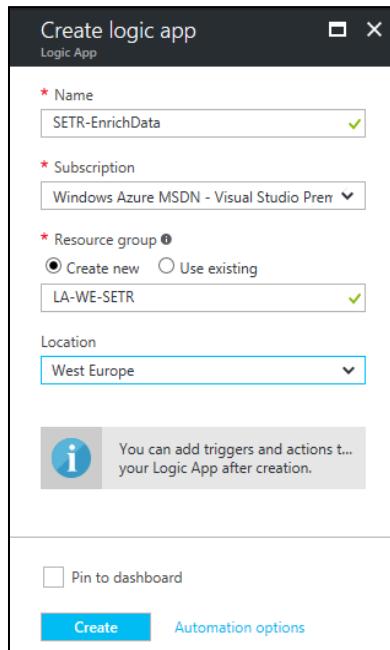
<https://portal.azure.com>

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***.



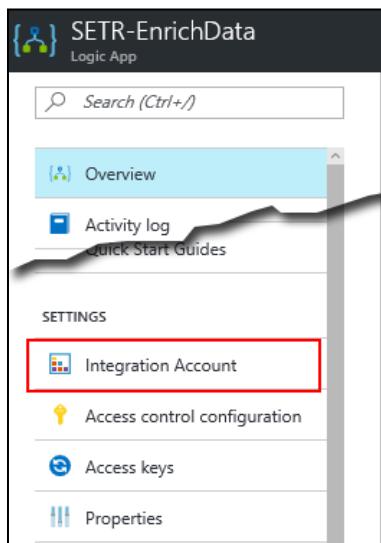
GLOBAL INTEGRATION BOOTCAMP

Associate the Integration Account with the Logic App

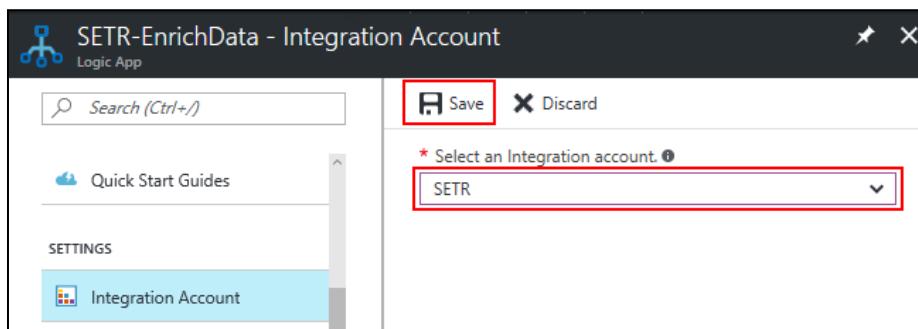
1. Navigate to the Logic App you've just created

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**.

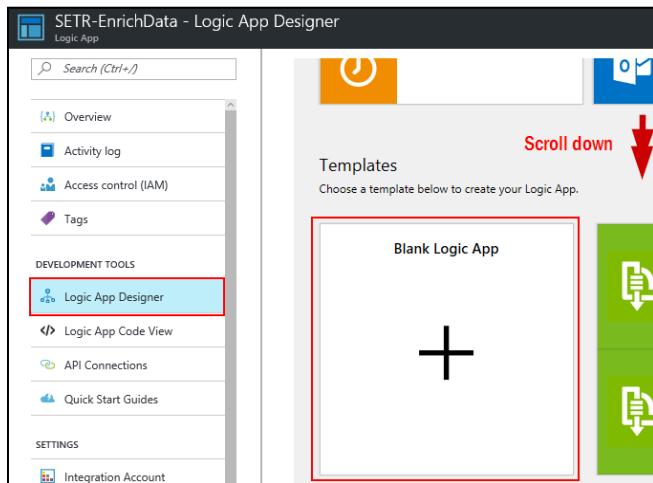


GLOBAL INTEGRATION BOOTCAMP

Build Logic App Definition

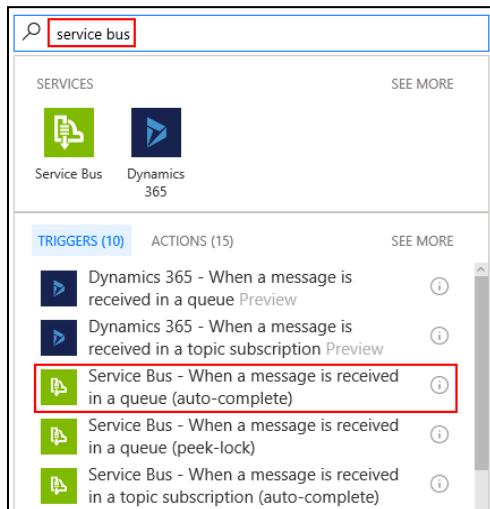
And now for the fun part: building the logic app definition.

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)**.



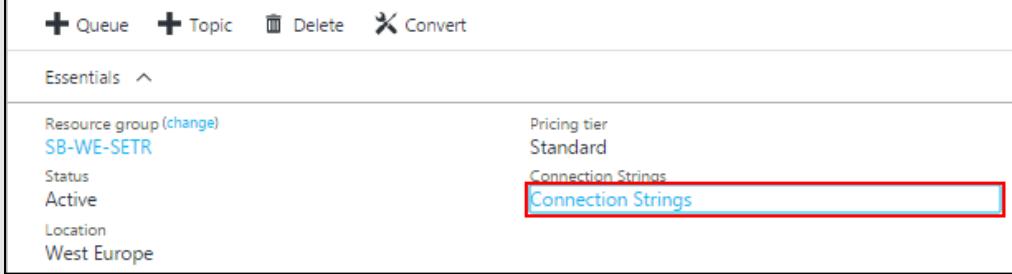
Now fill out a **connection name** and the **connection string**. This is the connection used in the 1st lab. Click Create.



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

FINIDING THE SERVICE BUS CONNECTION STRING

- Navigate to the **Service Bus blade**
- Open the service bus namespace** used for these labs
- Click on **Connection Strings**



The screenshot shows the 'Essentials' section of a Service Bus blade. It includes fields for Resource group (SB-WE-SETR), Status (Active), Location (West Europe), Pricing tier (Standard), and a 'Connection Strings' link which is highlighted with a red box.

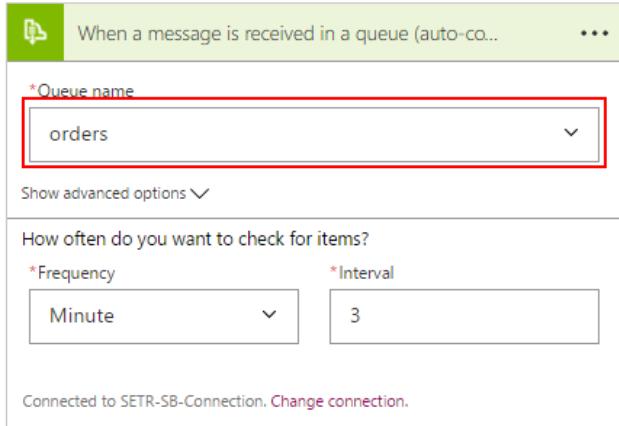
- Select **RootManageSharedAccessKey**
- Copy the **Connection String – Primary Key**

CONNECTION STRING–
PRIMARY KEY

Endpoint=sb://setr-servicebus.servicebus.windows.net/
[Copy](#)

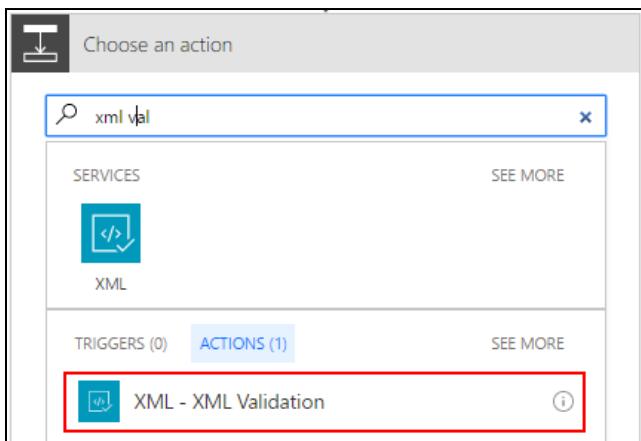
After setting up the connection, you will be able to **select the queue** you want to use. The same queue where we were sending the message to in the 1st lab will be used. Set an **interval**. Be careful, because you are paying for each polling action!

GLOBAL INTEGRATION BOOTCAMP



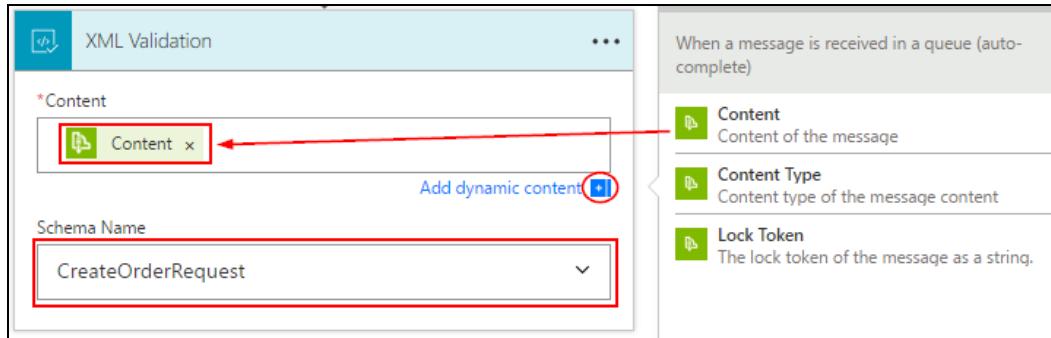
3. Now that the XML has been received, we will be able to validate it. For validation, the XSD we've just uploaded will be used.

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



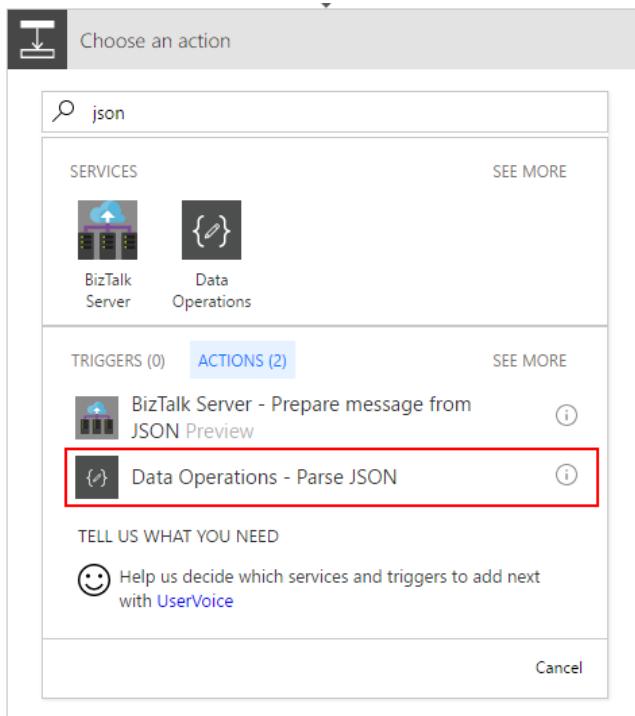
The content just received from the queue will be validated.

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.



- 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.

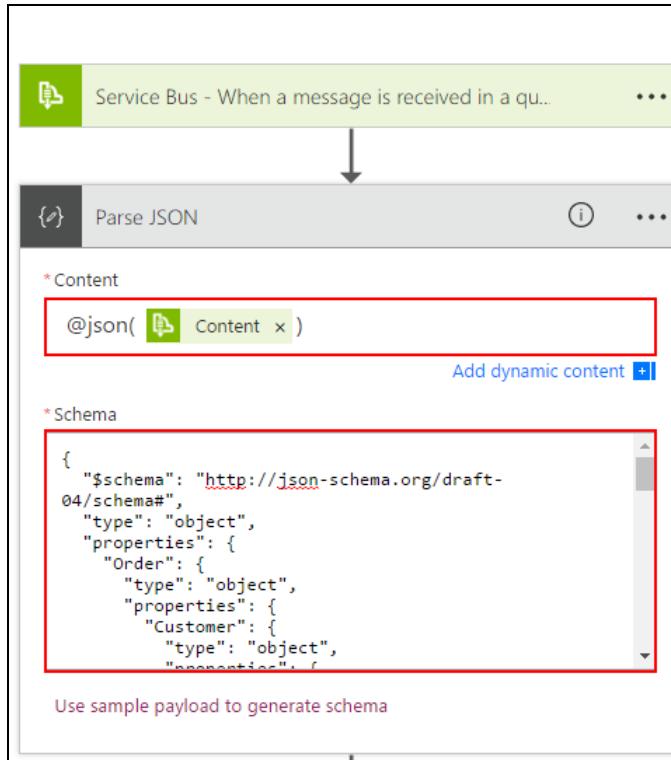
Add a **Parse JSON** action to parse from XML to JSON. This action can be found under **Data Operations**.



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.



GLOBAL INTEGRATION BOOTCAMP



Add dynamic content from the apps and services used in this flow. Hide

Search dynamic content

When a message is received in a queue (auto-complete) See more

- Content** Content of the message
- Content Type** Content type of the message content
- Lock Token** The lock token of the message as a string.
- Properties** Key-value pairs for each brokered property
- This object has the content and properties of a Service...** This object has the content and properties of a Service B...

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": "array"
        }
      }
    }
  }
}
```

GLOBAL INTEGRATION BOOTCAMP

```
"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:

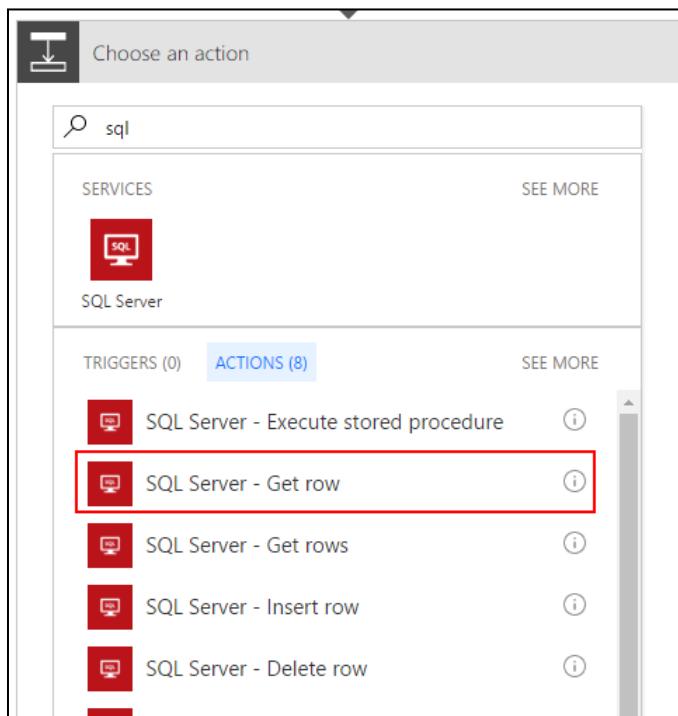
```
@json(xml(base64ToString(triggerBody()?['ContentData'])))
```

GLOBAL INTEGRATION BOOTCAMP

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***.

GLOBAL INTEGRATION BOOTCAMP

SQL Server - Get row

GATEWAYS

Connect via on-premise data gateway ⓘ

* SQL SERVER NAME

SQL2016D

* SQL DATABASE NAME

LegacyOrderSystem

AUTHENTICATION TYPE

Windows

* USERNAME

robbox

* PASSWORD

.....

* GATEWAY

SETR Data Gateway

Create

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**.

Get row

* Table name

Name of SQL table

Customer

Order

Product

Enter custom value

* Row id

Unique identifier of the row to retrieve

Connected to SQL Server. Change connection.

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.

GLOBAL INTEGRATION BOOTCAMP

The screenshot shows the Microsoft Logic App designer interface. On the left, a 'Get row' step is selected, with its configuration pane open. The 'Table name' is set to 'Customer' and the 'Row id' is set to 'CustomerNumber'. A red arrow points from the 'CustomerNumber' input field to a red circle around the 'Add dynamic content' button. To the right, a 'Parse JSON' panel is open, showing a list of fields: 'Amount', 'CustomerNumber' (which is also highlighted with a red box), 'OrderedDateTime', and 'ProductNumber'. Below this, there's a section for 'When a message is received in a queue (auto-complete)' with options for 'Content', 'Content Type', and 'Lock Token'.

- 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:

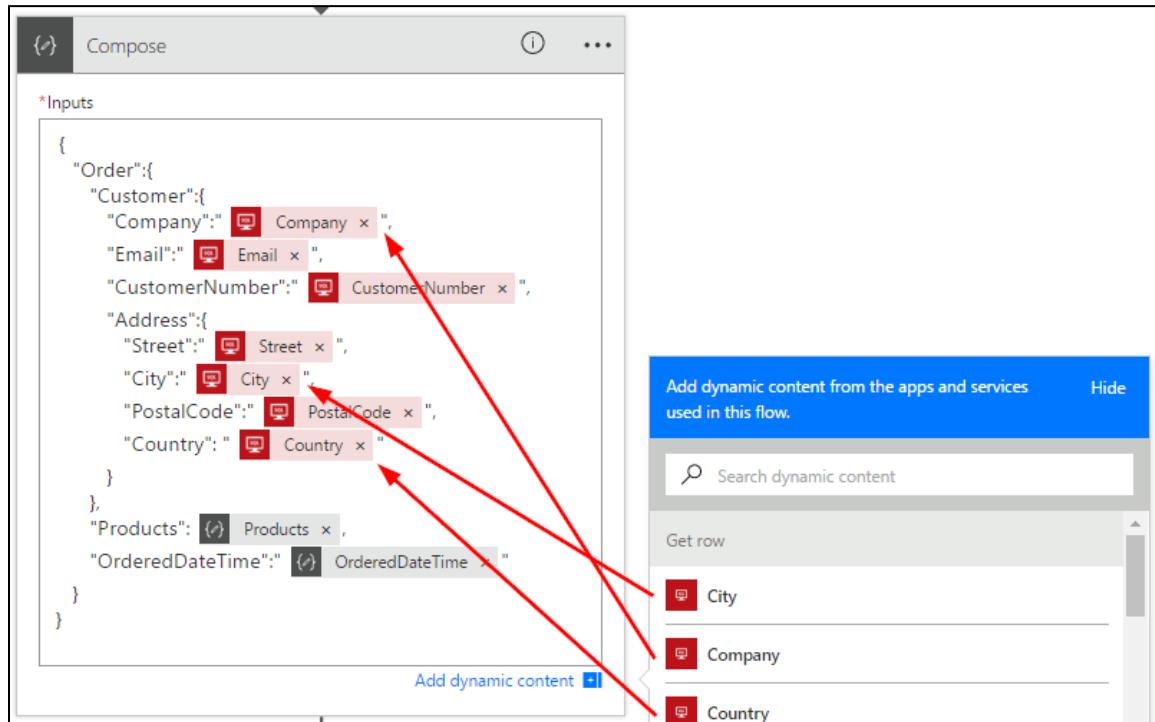
```
{
  "Order": {
    "Customer": {
      "Company": "",
      "Email": "",
      "CustomerNumber": "",
      "Address": {
        "Street": "",
        "City": "",
        "PostalCode": "",
        "Country": ""
      }
    },
    "Products": ,
    "OrderedDateTime": ""
  }
}
```

GLOBAL INTEGRATION BOOTCAMP

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**.

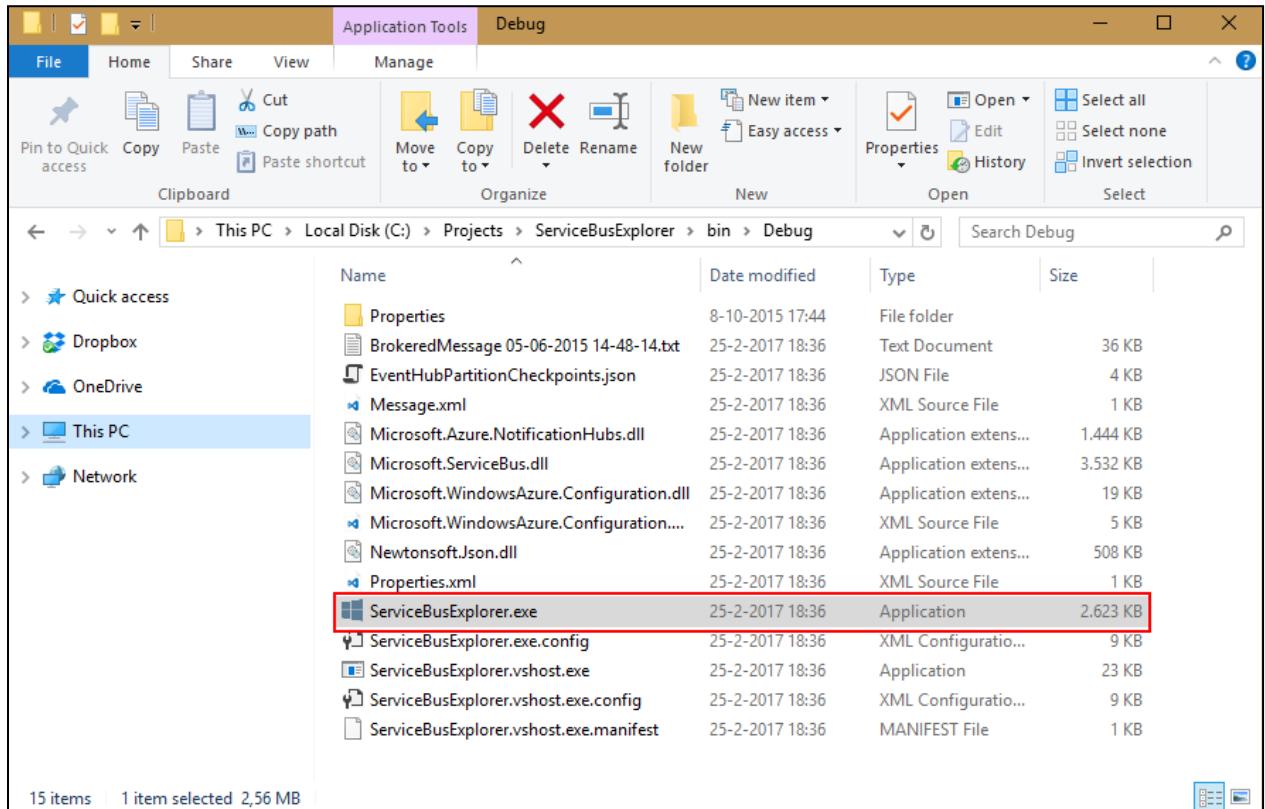
I. Download Service Bus Explorer 2.6

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

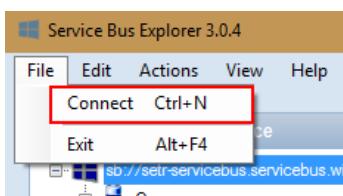
GLOBAL INTEGRATION BOOTCAMP

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*.

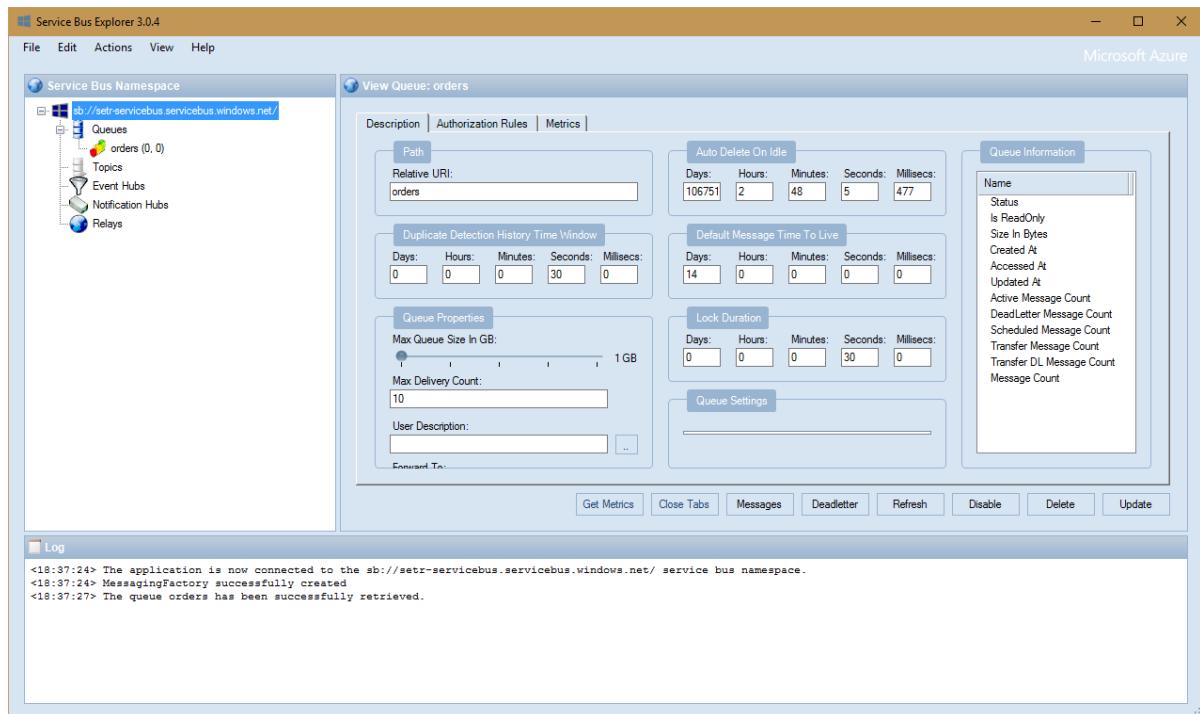
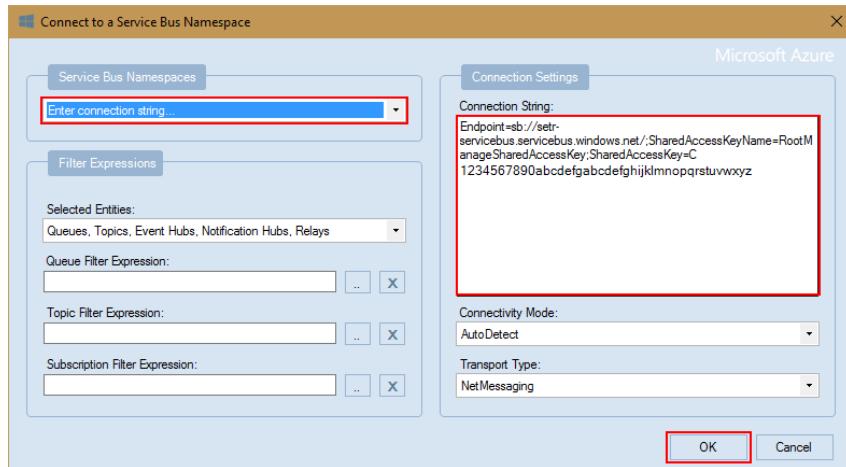


Setup a connection by choosing *File > Connect*.



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

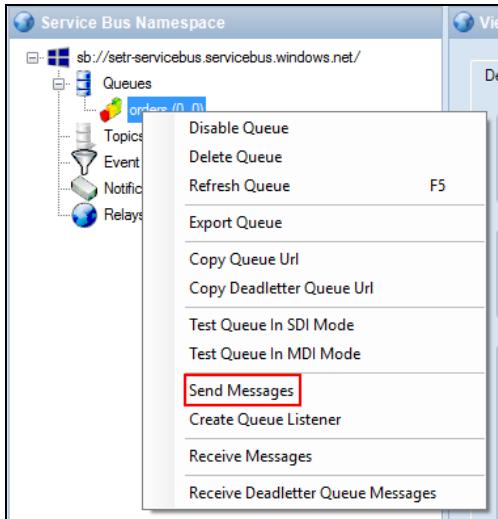
GLOBAL INTEGRATION BOOTCAMP



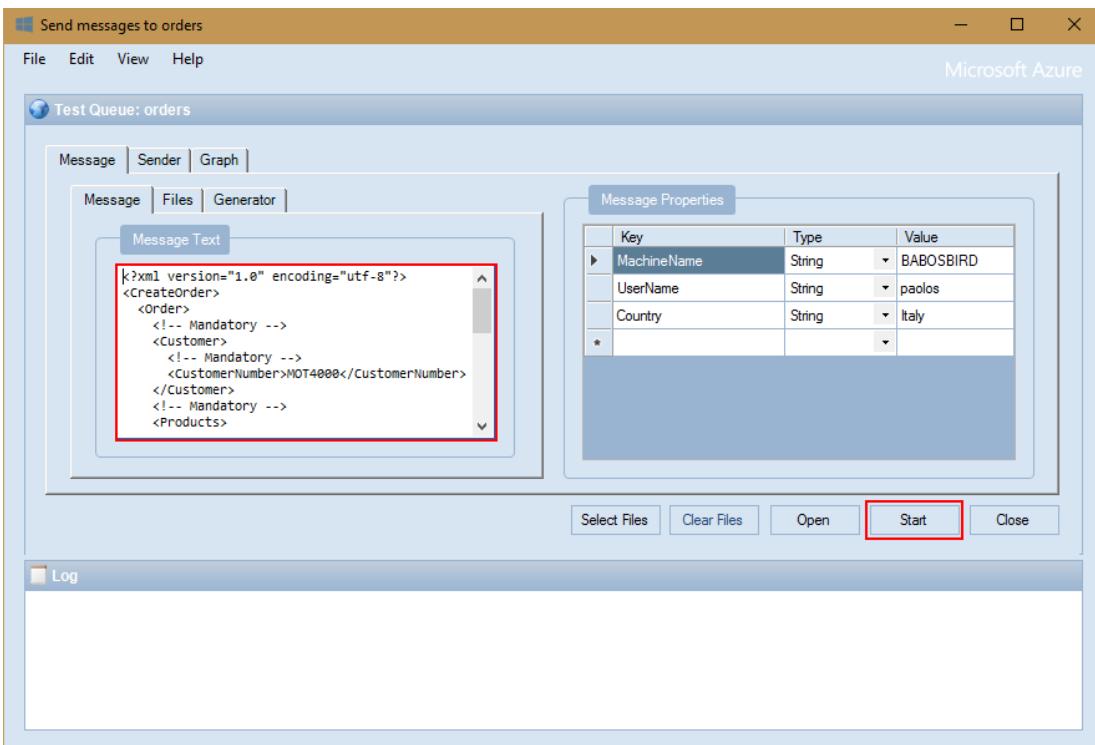
II. Send the message to the queue

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

GLOBAL INTEGRATION BOOTCAMP



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.



GLOBAL INTEGRATION BOOTCAMP

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>
  </Order>
</CreateOrder>
```

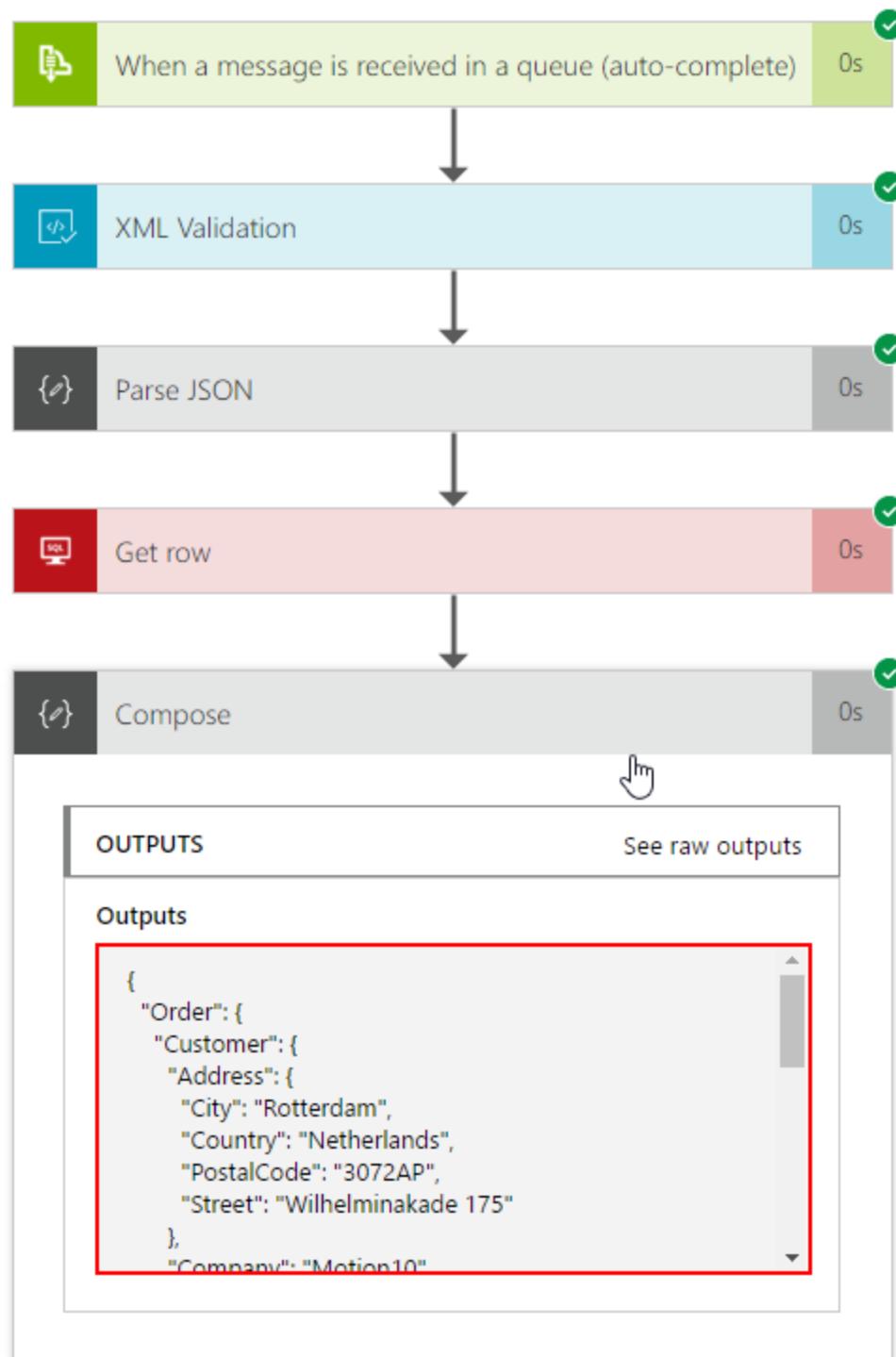
Now check you runs in the overview blade. Check both the **succeeded run** and the **failed run** by selecting them.

 **GLOBAL INTEGRATION BOOTCAMP**

Runs history				
STATUS	START TIME	IDENTIFIER	DURATION	
Failed	2/26/2017, 12:27 PM	0858713498428687467...	490 Milliseconds	
Succeeded	2/26/2017, 12:14 PM	0858713499239409534...	625 Milliseconds	

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.



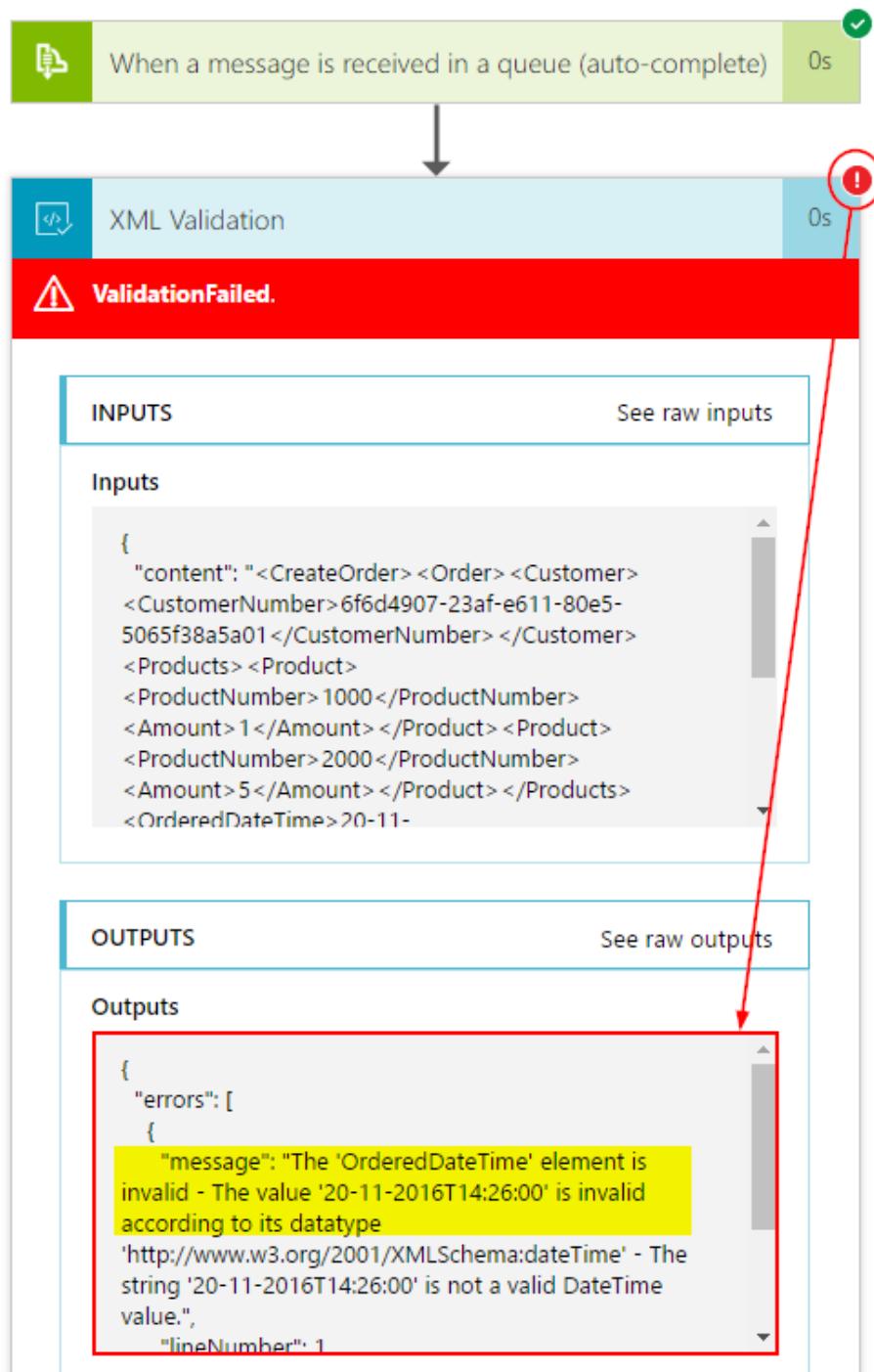
The correct output should resemble the following message. Notice that the order of JSON elements is not important, as that is different from XML.

{

```
"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.



The screenshot shows a sequence of steps in an integration tool:

- Step 1:** A green step labeled "When a message is received in a queue (auto-complete)" with a duration of "0s".
- Step 2:** An "XML Validation" step with a duration of "0s". This step has failed, indicated by a red status bar with the message "ValidationFailed." and a red exclamation mark icon.
- Step 3:** The "INPUTS" section shows the XML message being validated. The XML content is:

```
{
  "content": "<CreateOrder><Order><Customer>
<CustomerNumber>6f6d4907-23af-e611-80e5-
5065f38a5a01</CustomerNumber></Customer>
<Products><Product>
<ProductNumber>1000</ProductNumber>
<Amount>1</Amount></Product><Product>
<ProductNumber>2000</ProductNumber>
<Amount>5</Amount></Product></Products>
<OrderedDateTime>20-11-
```
- Step 4:** The "OUTPUTS" section shows the validation error. The error message is:

```
{
  "errors": [
    {
      "message": "The 'OrderedDateTime' element is
invalid - The value '20-11-2016T14:26:00' is invalid
according to its datatype
'http://www.w3.org/2001/XMLSchema:dateTime' - The
string '20-11-2016T14:26:00' is not a valid DateTime
value.",
      "lineNumber": 1
    }
  ]
}
```

A red box highlights the error message, and a red arrow points from the validation failure in the XML Validation step to this error message.

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

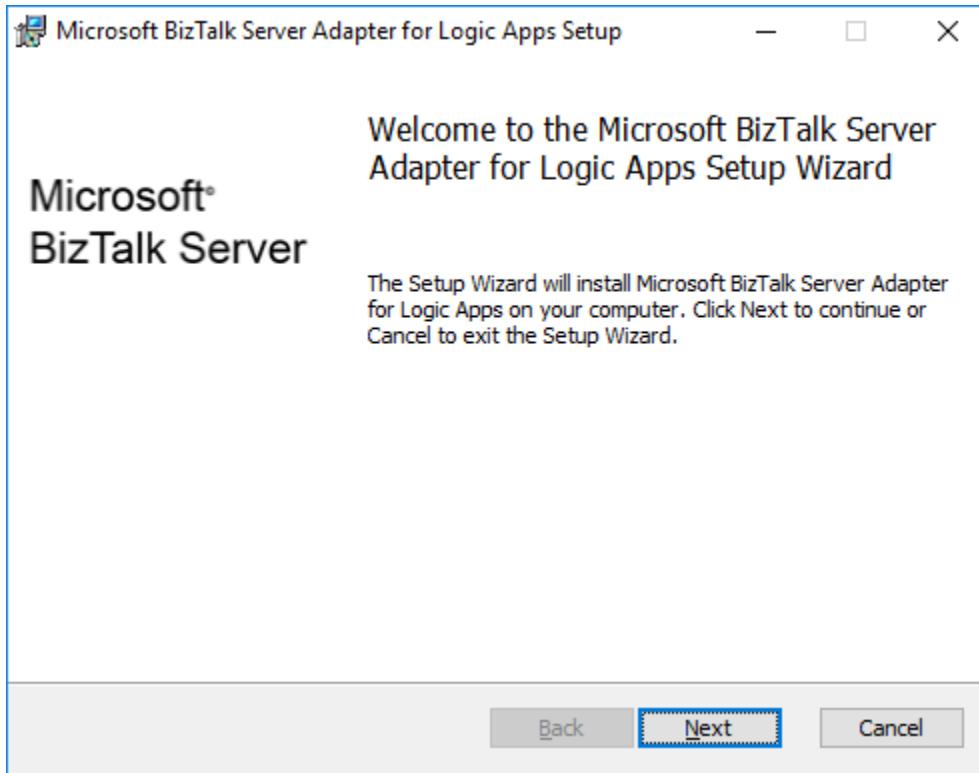
To build the solution in this lab you have to follow the steps described in this section. From a high level view the steps are:

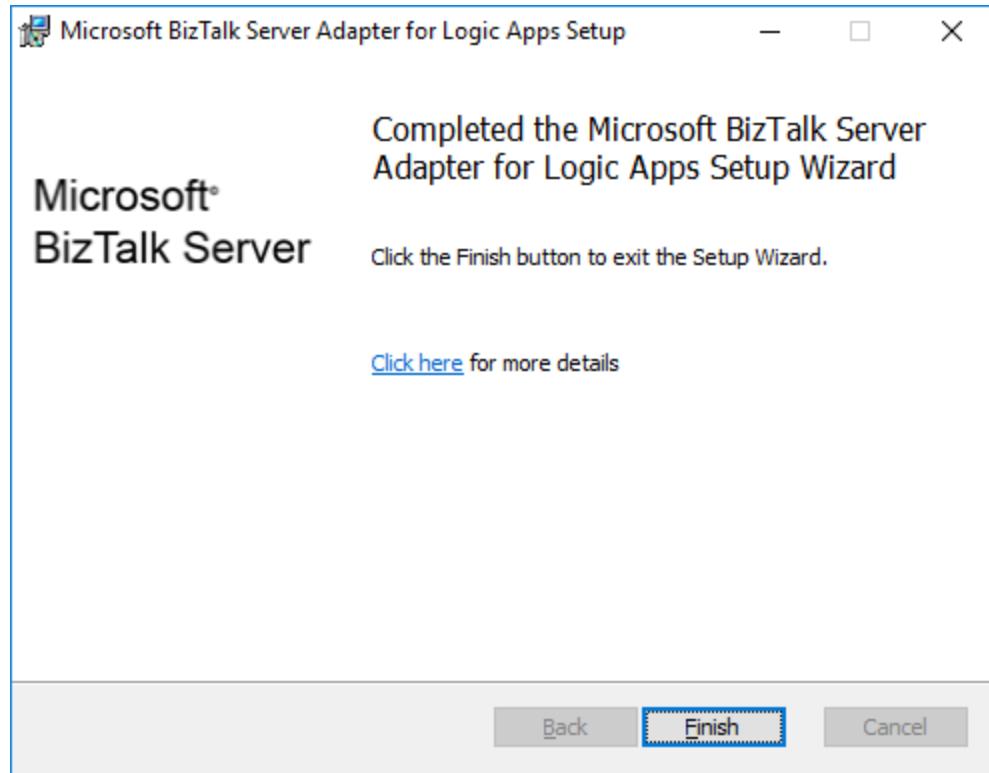
1. Install and configure BizTalk LogicAppAdapter
2. Create Service Bus namespace
3. Install and configure BizTalk adapter
4. Create Azure Logic App

GLOBAL INTEGRATION BOOTCAMP

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.



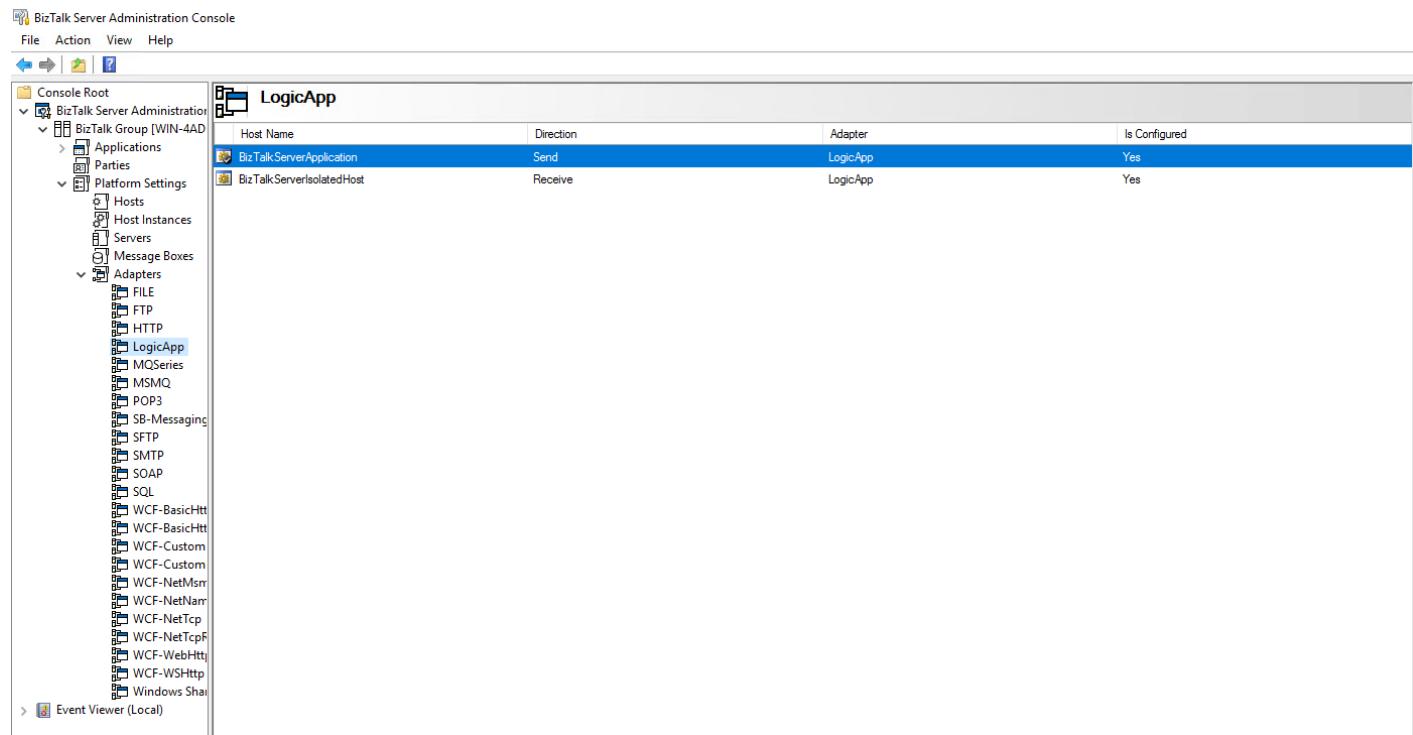


Once installed, the following will have been done automatically.

- 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.

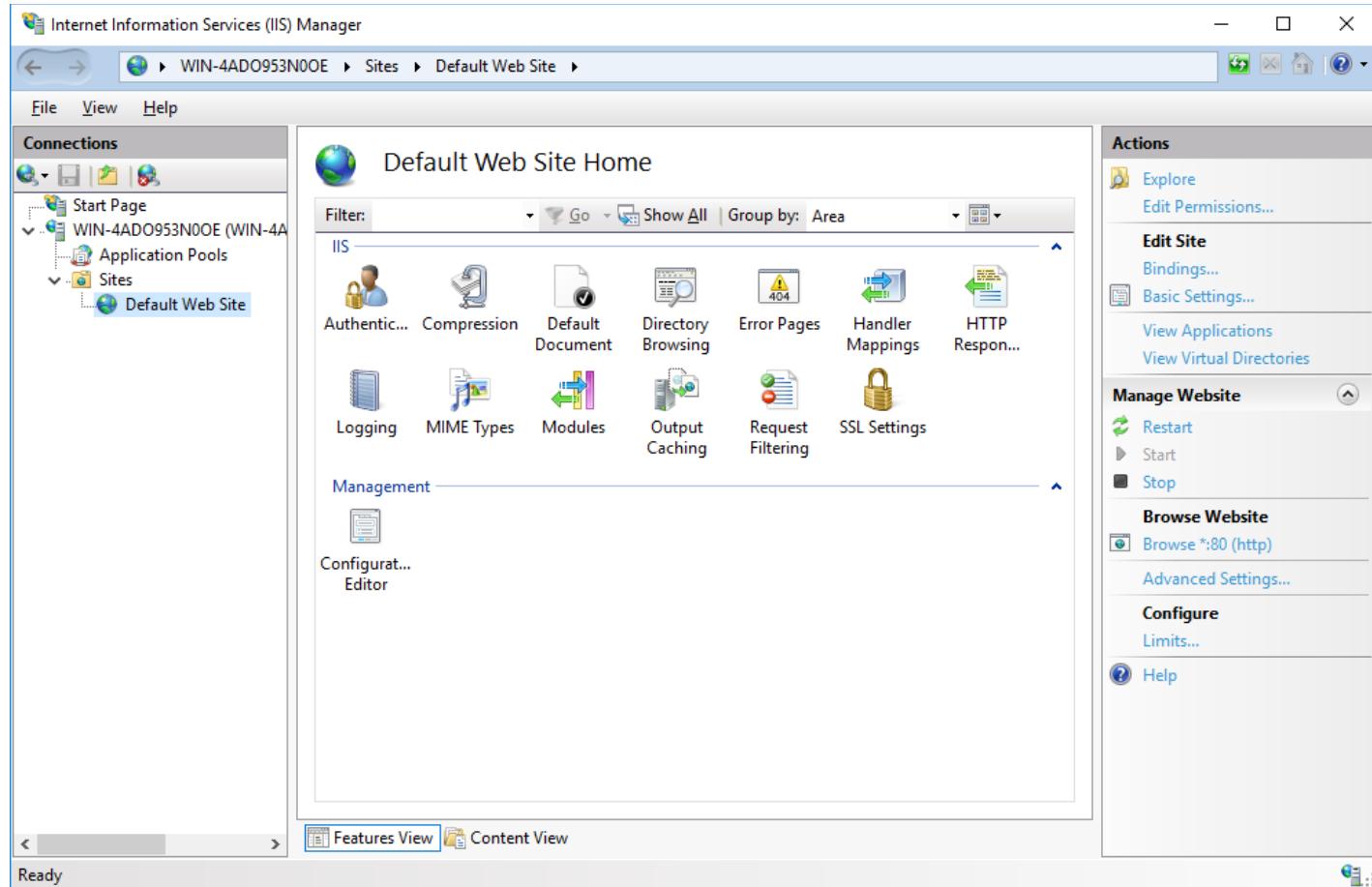
GLOBAL INTEGRATION BOOTCAMP



GLOBAL INTEGRATION BOOTCAMP

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.

GLOBAL INTEGRATION BOOTCAMP

Add Application Pool

Name:

.NET CLR version:

Managed pipeline mode:

Start application pool immediately

Application Pool Identity

Built-in account:

Custom account:

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.

GLOBAL INTEGRATION BOOTCAMP

The screenshot shows the IIS Administration console interface. A context menu is open with the following options:

- Explore
- Edit Permissions...
- Add Application... (highlighted)
- Add Virtual Directory...
- Edit Bindings...
- Manage Website ▾
- Refresh
- Remove
- Rename
- Switch to Content View

The "Add Application..." option is highlighted with a blue border. Below it, the "Add Application" dialog is displayed:

Add Application

Site name: Default Web Site
Path: /

Alias: IISLogicApp Application pool: BizTalkApplicationPool Select...

Example: sales

Physical path:
soft BizTalk Server 2016\LogicApp Adapter\Management ...

Pass-through authentication

Connect as... Test Settings...

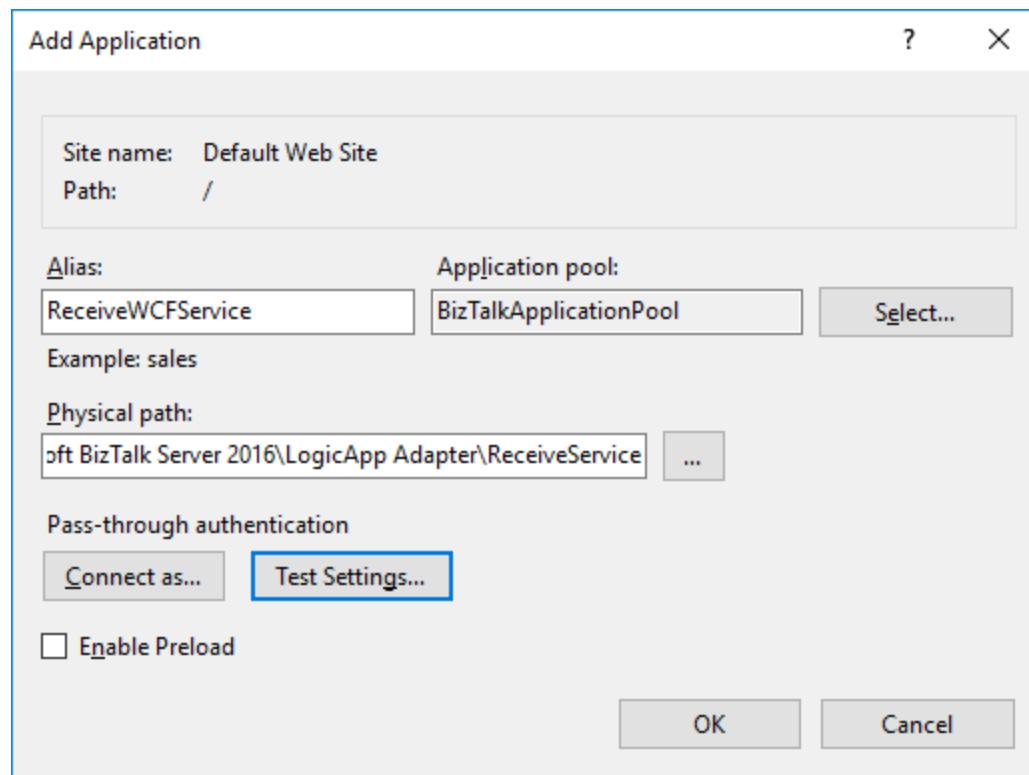
Enable Preload

OK Cancel

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

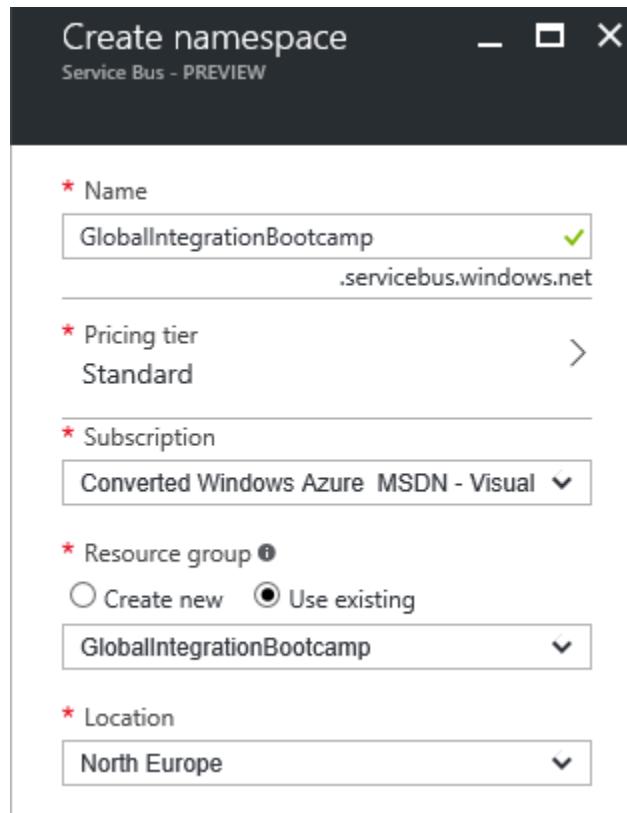
Once again in the IIS Administration console, create another application, this time with the physical path set to C:\Program Files (x86)\Microsoft BizTalk Server 2016\LogicApp Adapter\ReceiveService. Here we also need to make sure to select the application pool we created earlier.



GLOBAL INTEGRATION BOOTCAMP

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.



The screenshot shows the 'Create namespace' dialog for Service Bus - PREVIEW. It includes fields for Name (GlobalIntegrationBootcamp), Pricing tier (Standard), Subscription (Converted Windows Azure MSDN - Visual), Resource group (Use existing, GlobalIntegrationBootcamp), and Location (North Europe).

Name	GlobalIntegrationBootcamp
Pricing tier	Standard
Subscription	Converted Windows Azure MSDN - Visual
Resource group	<input checked="" type="radio"/> Use existing GlobalIntegrationBootcamp
Location	North Europe

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.

GLOBAL INTEGRATION BOOTCAMP

The screenshot shows the Azure portal interface for managing a Service Bus Namespace. On the left, there's a sidebar with 'Pricing tier Standard' and 'Connection Strings' selected. The main area is titled 'Shared access policies' for 'GlobalIntegrationBootcamp - PREVIEW'. It lists a single policy named 'RootManageSharedAccessKey' with the claim 'Manage, Send, Listen' selected. To the right, detailed information about the policy is shown, including its primary and secondary keys, and its connection strings.

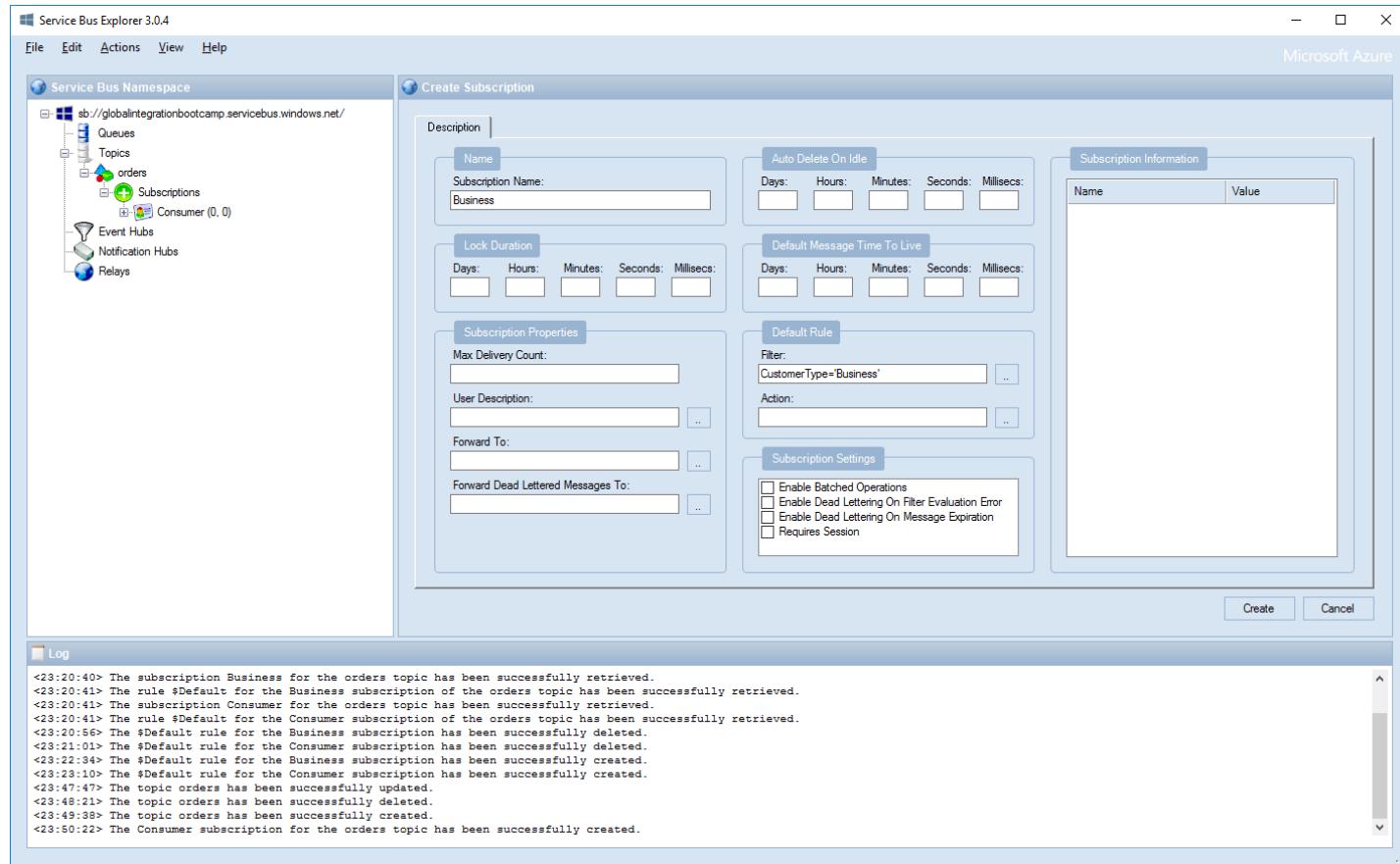
Now start Service Bus Explorer, and connect using the connection string we just retrieved.

The screenshot shows the 'Connect to a Service Bus Namespace' dialog from Service Bus Explorer. In the 'Service Bus Namespaces' section, there's a dropdown labeled 'Enter connection string...'. In the 'Connection Settings' section, the 'Connection String:' field contains the copied connection string: `Endpoint=sb://globalintegrationbootcamp.servicebus.windows.net/;SharedAccessKeyName=RootManageSharedAccessKey;SharedAccessKey=sWd8S9z[REDACTED]hZPkkaWM=[REDACTED]`. At the bottom right, there are 'OK' and 'Cancel' buttons.

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'**

GLOBAL INTEGRATION BOOTCAMP



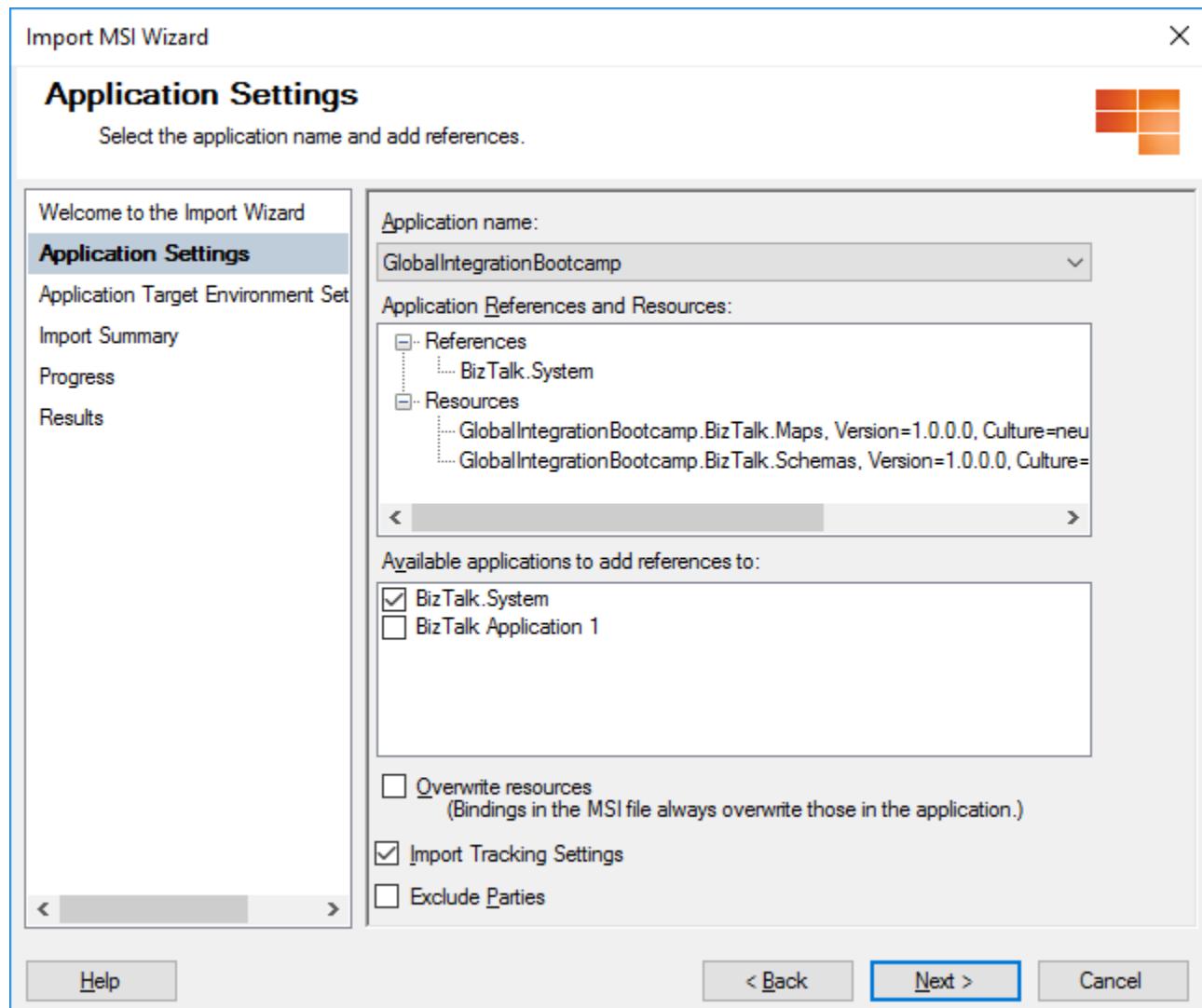
GLOBAL INTEGRATION BOOTCAMP

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.

Import MSI Wizard X

Results

Displays the results of the import operation.



Welcome to the Import Wizard

Application Settings

Application Target Environment Set

Import Summary

Progress

Results

< >

Import Succeeded

Results:

The application GlobalIntegrationBootcamp was successfully imported to the group BizTalk Group.

This application depends on the following applications for correct operation.
-BizTalk.System

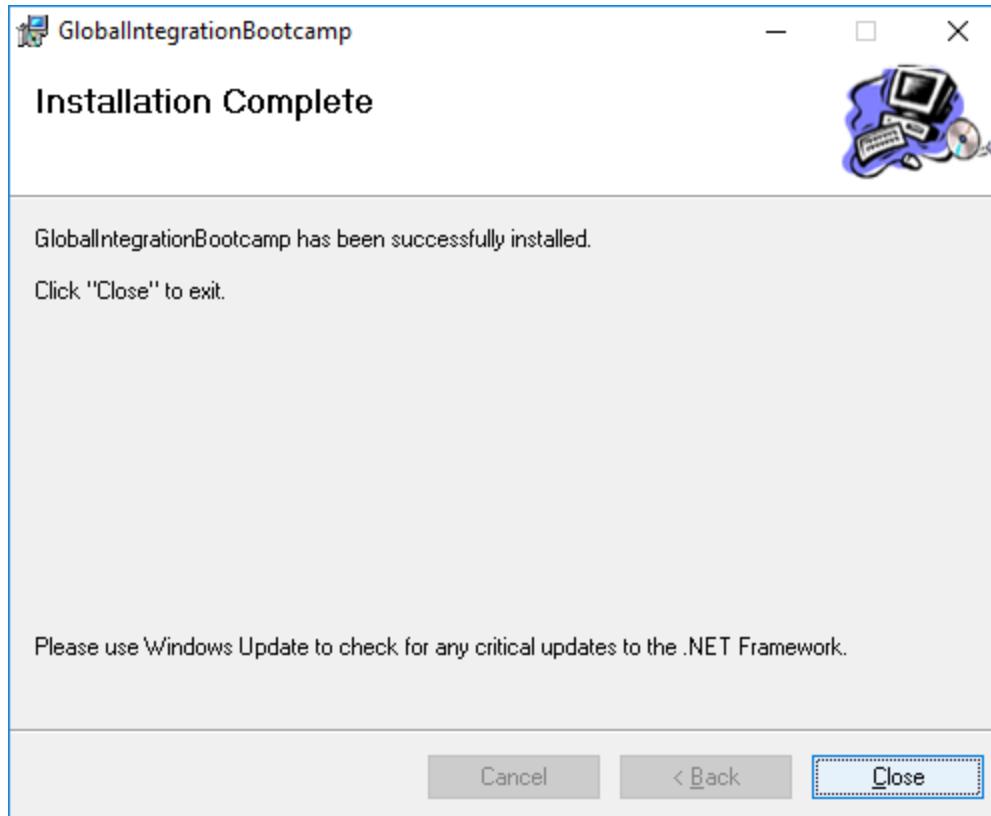
A log of this operation has been saved at the location
file:///C:/Users/Administrator/AppData/Local/Temp/2/Import/2017-01-31T17_11_041.BizTalk Application 1.log.

To install this application or resources for this application, select the checkbox below.

Run the Application Installation Wizard to install the application on the local computer

Help Finish Cancel

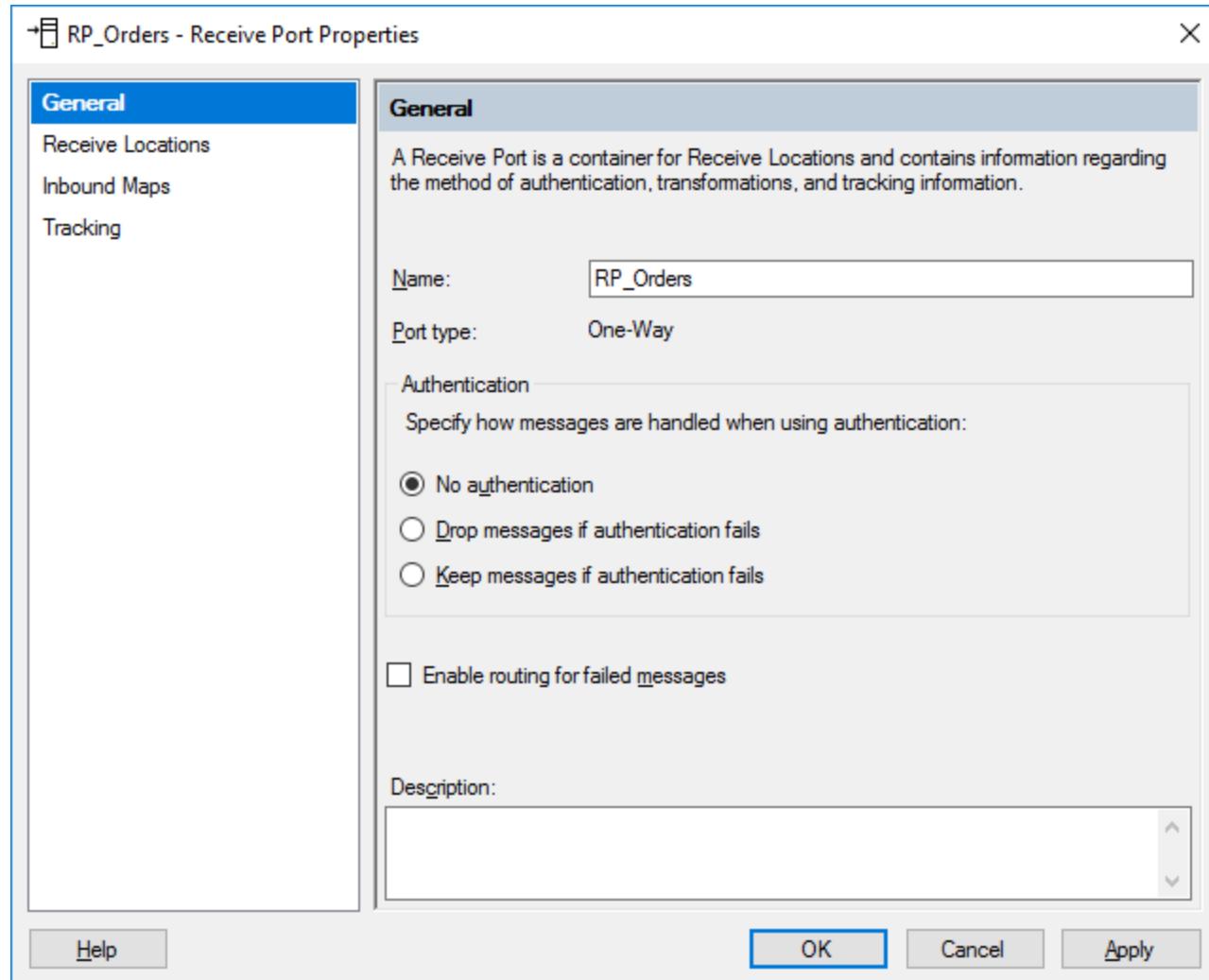
For the installation, once again use all default settings.



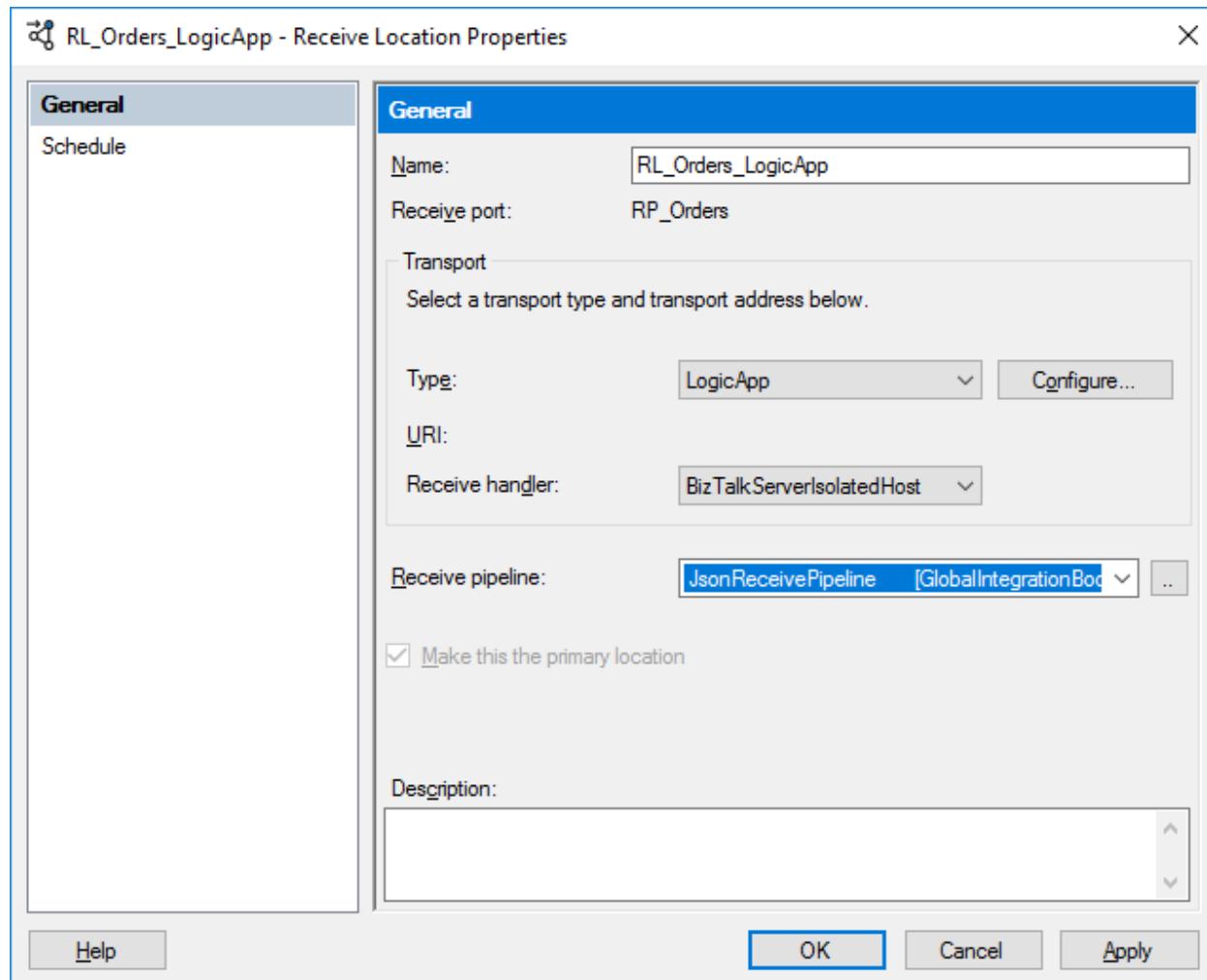
After the application has been installed, remember to restart your host instance and do an iisreset.

Create Receive Location from logic app

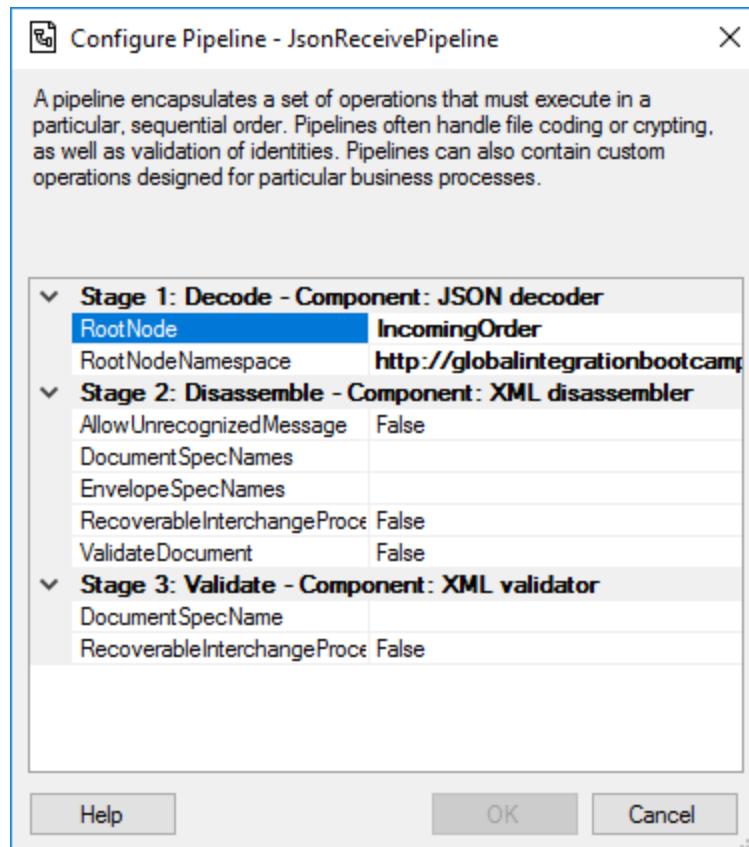
Now that the application is installed, we can add our ports to it. We will start with the receive location which will be called by our Logic App to send us an order.



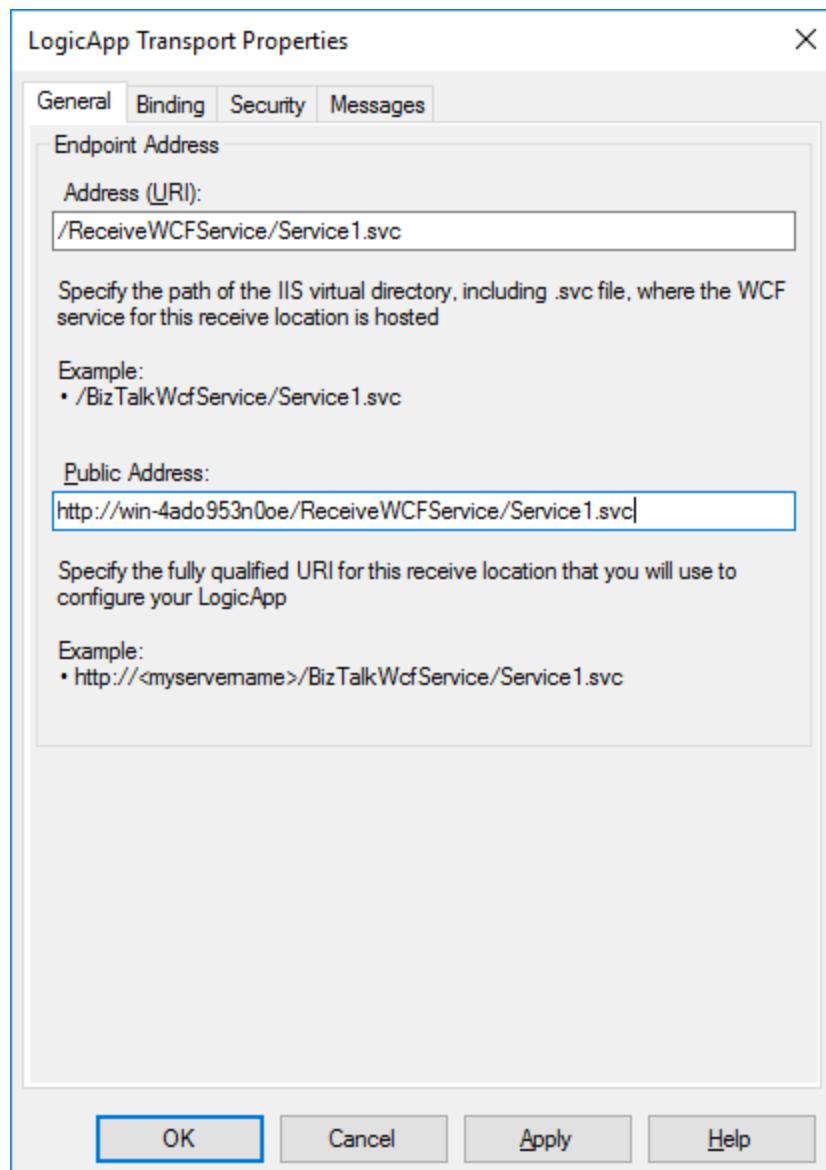
Select the LogicApp adapter, and the JsonReceivePipeline.



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 **RootNamespace** to
<http://globalintegrationbootcamp.com/Orders/V1-0> (properties for pipeline can be opened by clicking the ellipses button next to the pipeline drop-down).

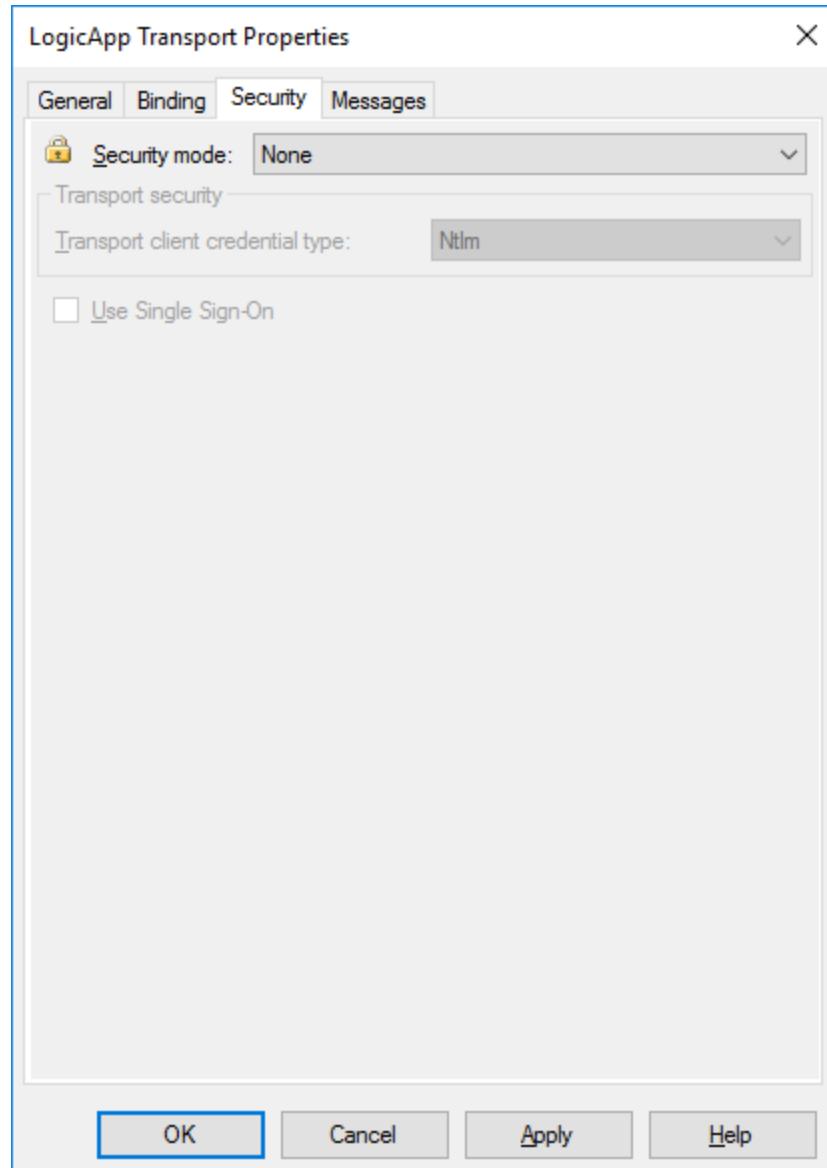


In the configuration of the LogicApp adapter (can be opened by clicking the **Configure...** button next to the port Type which we set to LogicApp), specify the address of the Receive service we created earlier. For the public address, this includes the local server name.

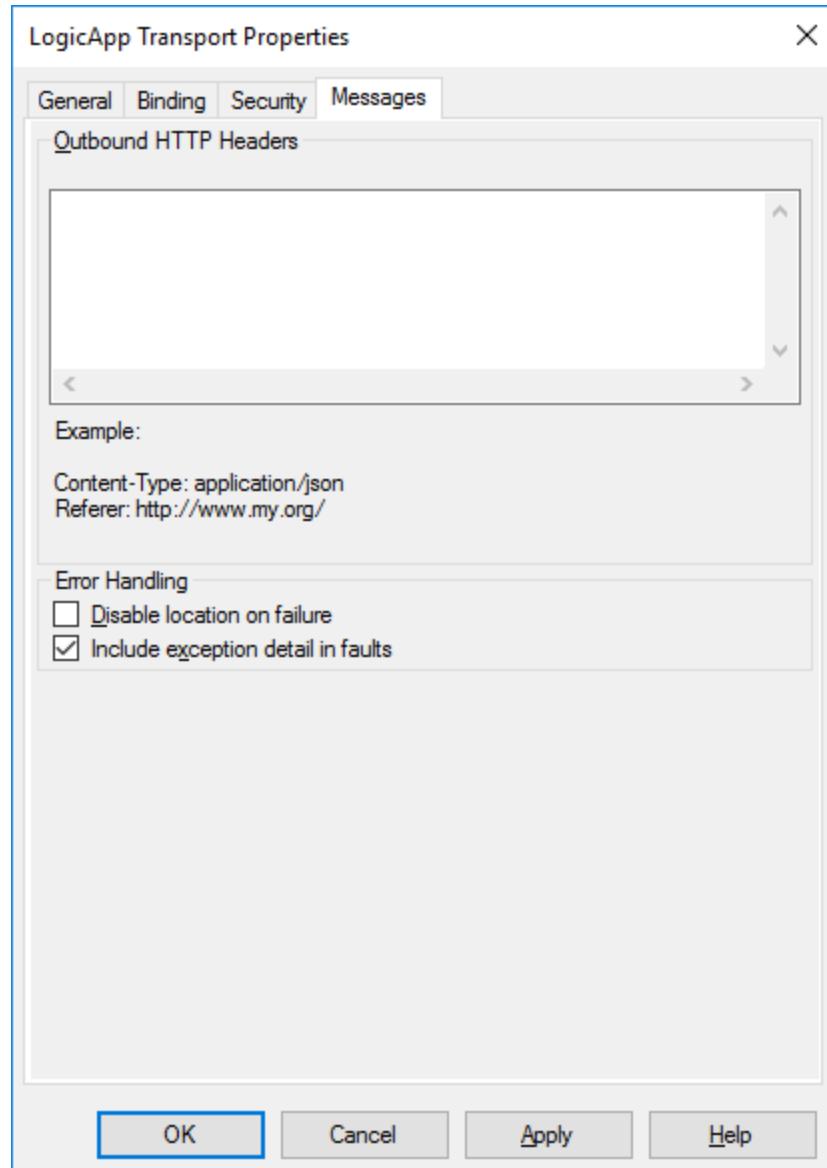
 **GLOBAL INTEGRATION BOOTCAMP**

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

GLOBAL INTEGRATION BOOTCAMP



GLOBAL INTEGRATION BOOTCAMP



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

RP_Orders - Receive Port Properties X

General
Receive Locations
Inbound Maps
Tracking

Inbound Maps

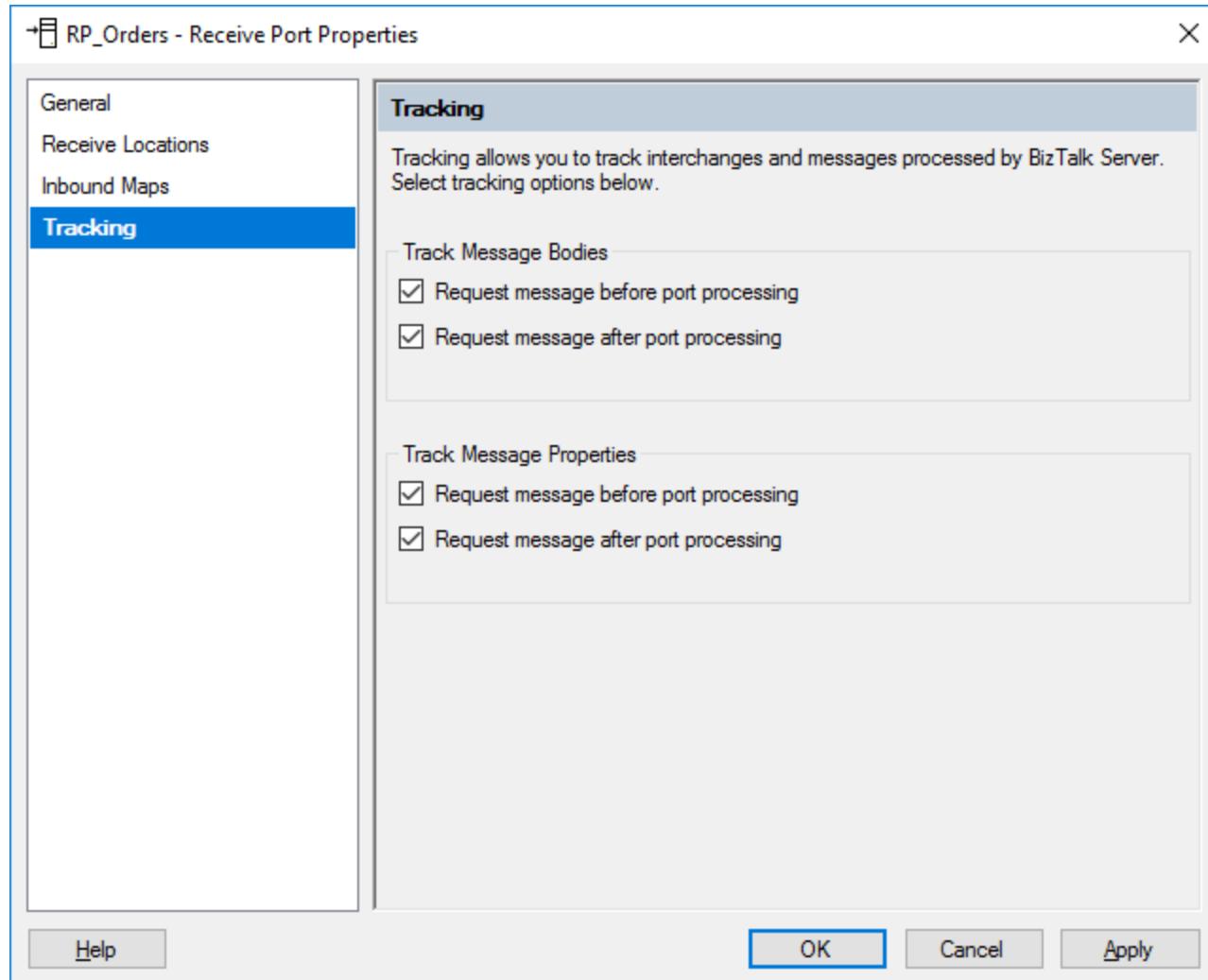
The following are the inbound maps used for transforming documents on the current port.

Inbound maps:

X Remove

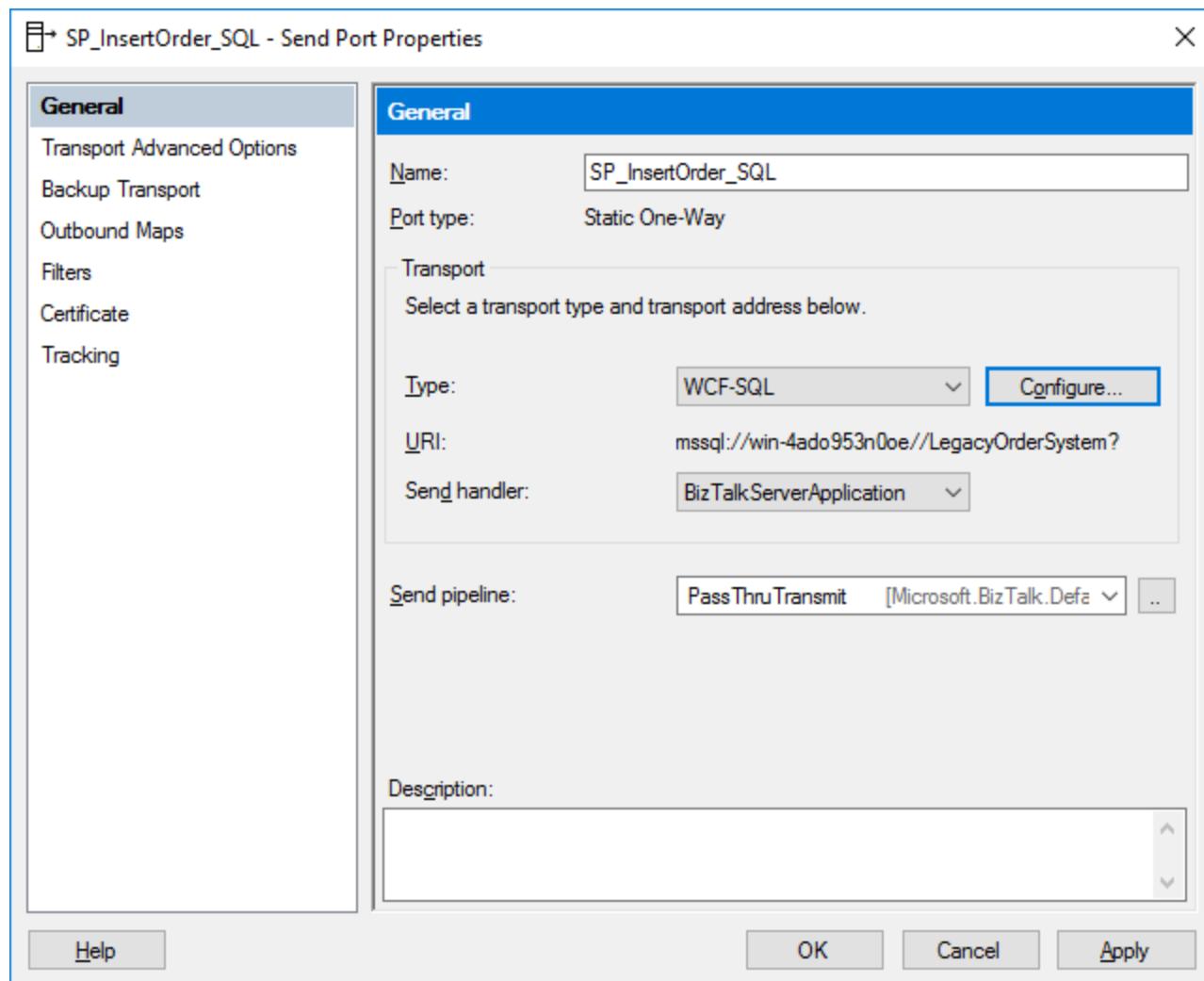
Source Document	Map	Target Document
IncomingOrder_XML ...	IncomingOrder_XML_T...	InsertOrder_XML [Gl...
*		

Help OK Cancel Apply

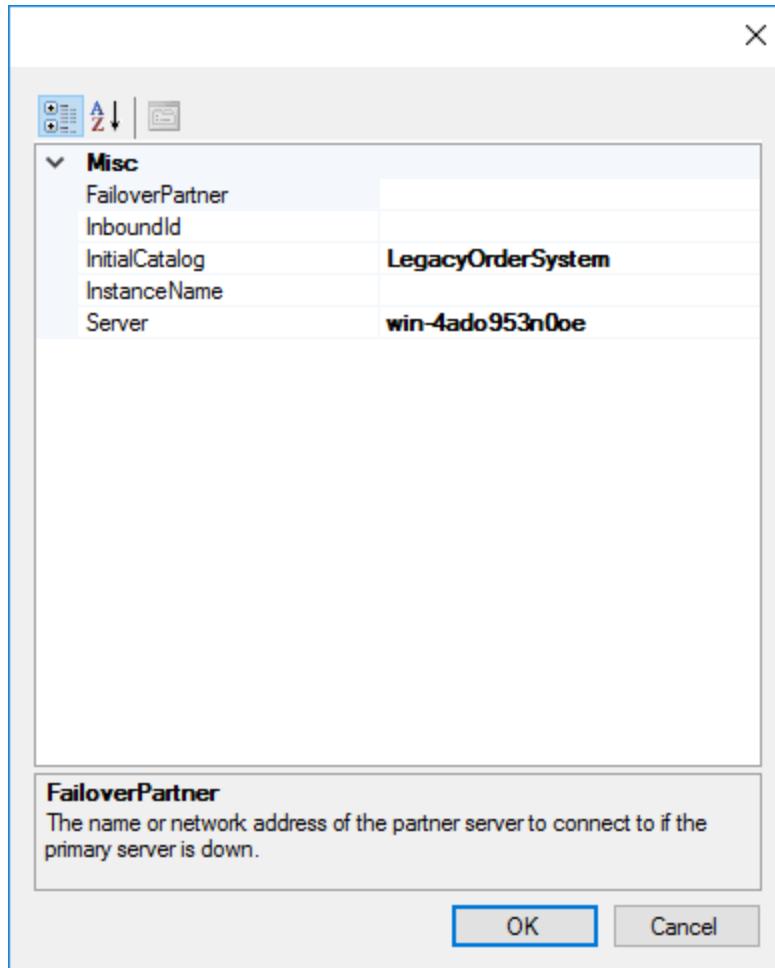


Create Send Port to database

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

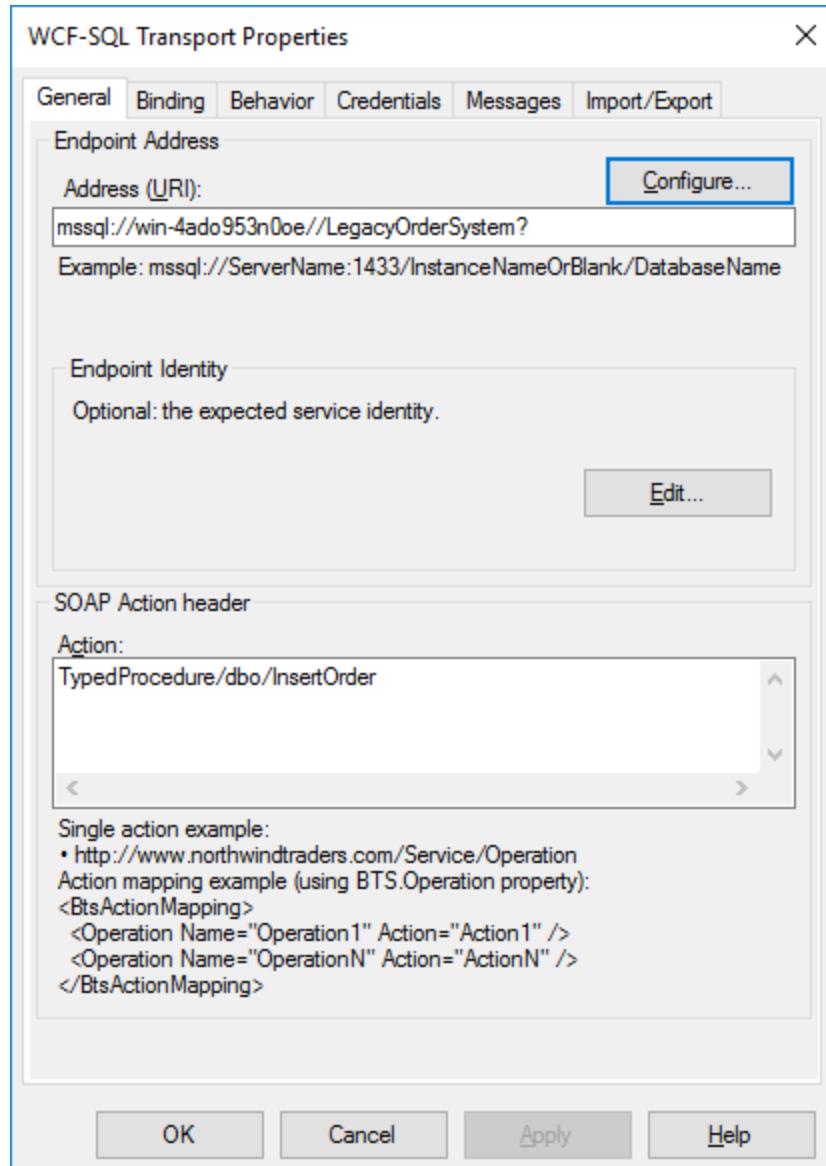


Make sure to specify your local server name (or **localhost** or **.**) and database name when configuring the adapter.



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**.

GLOBAL INTEGRATION BOOTCAMP



Set the filter to subscribe to the incoming messages.

SP_InsertOrder_SQL - Send Port Properties

General
Transport Advanced Options
Backup Transport
Outbound Maps
Filters
Certificate
Tracking

Filters

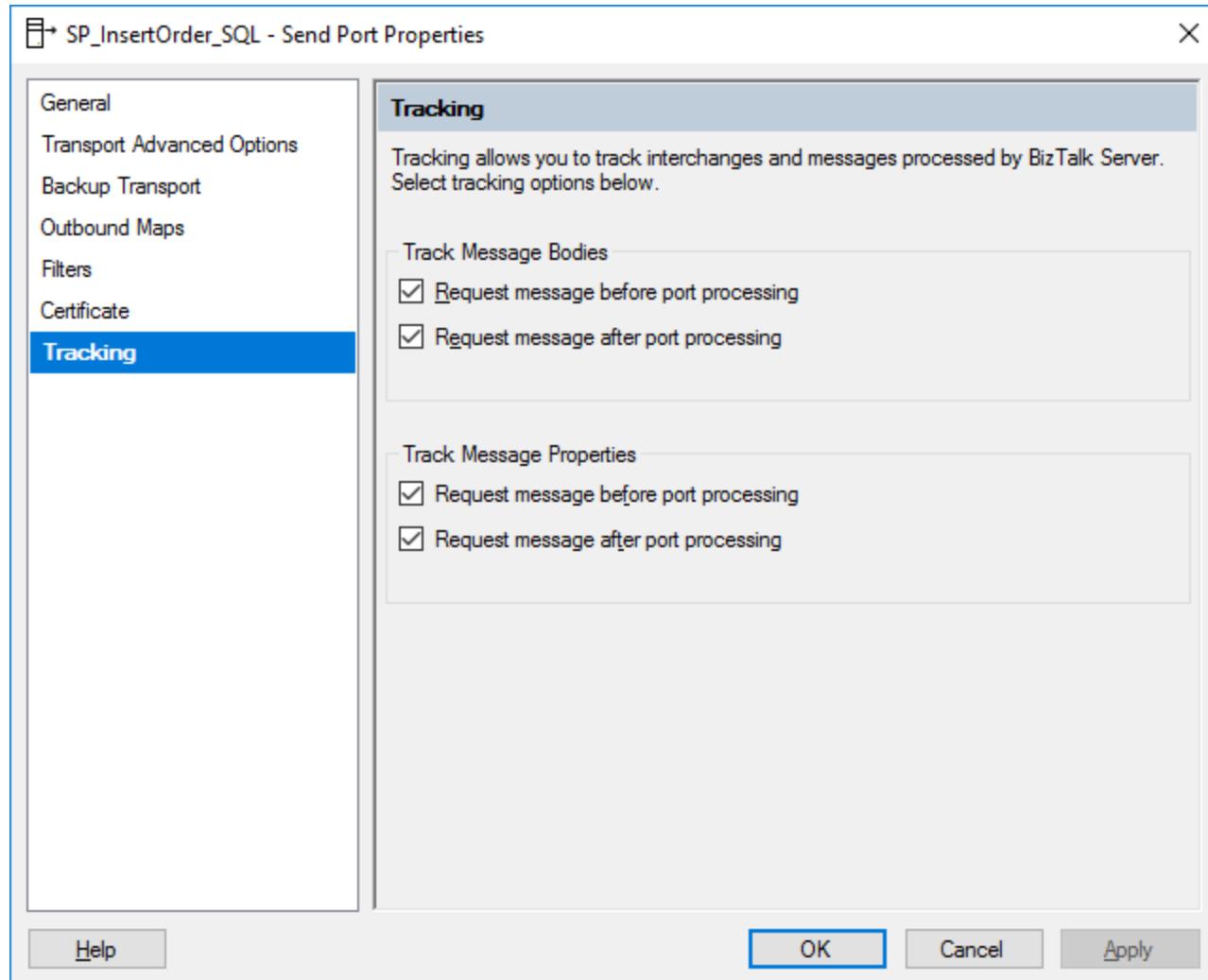
Filter expressions determine which messages are routed to this Send Port from the Message Box. Create one or more filter expressions using any properties in the message context.

X Delete ↑ Move Up ↓ Move Down

	Property	Operator	Value	Group by
▶	BTS.ReceivePortName	==	RP_Orders	And
*				

BTS.ReceivePortName == RP_Orders

Help OK Cancel Apply



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.

 **GLOBAL INTEGRATION BOOTCAMP**

RP_InvoiceOrders - Receive Port Properties X

General

Receive Locations
Inbound Maps
Tracking

General

A Receive Port is a container for Receive Locations and contains information regarding the method of authentication, transformations, and tracking information.

Name:

Port type: One-Way

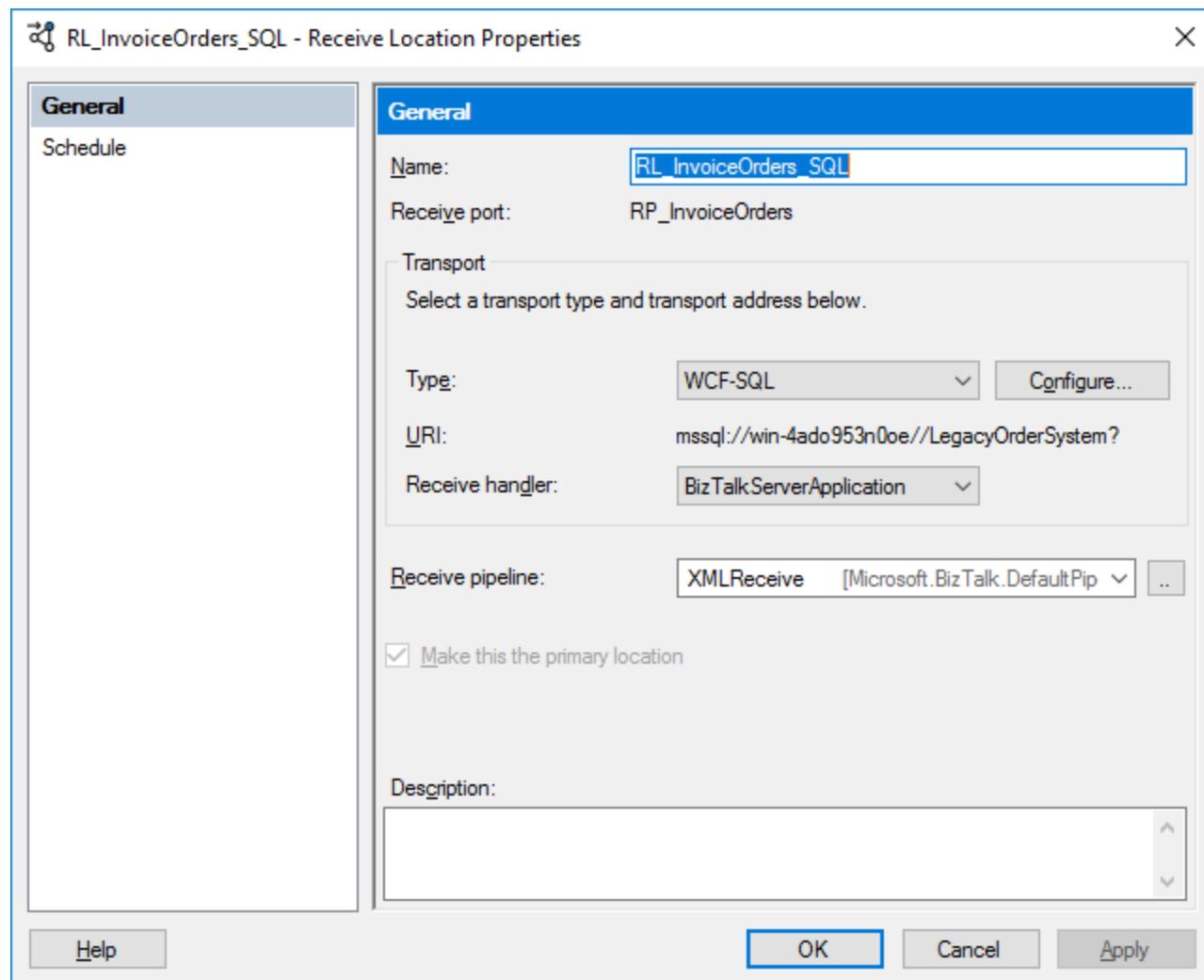
Authentication

Specify how messages are handled when using authentication:

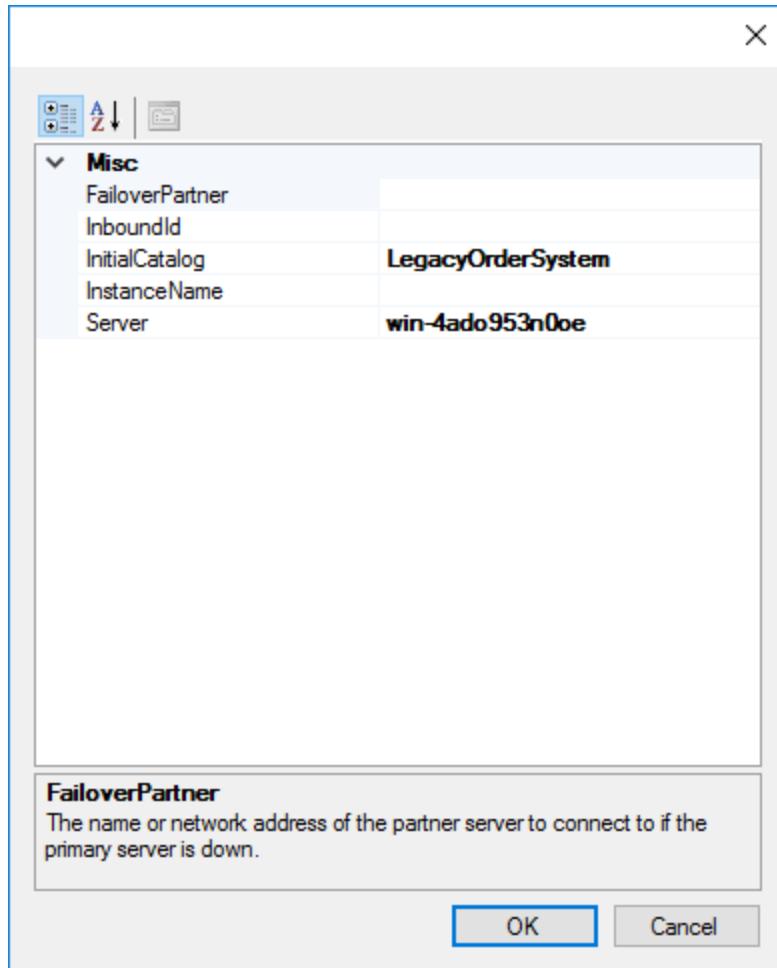
No authentication
 Drop messages if authentication fails
 Keep messages if authentication fails

Enable routing for failed messages

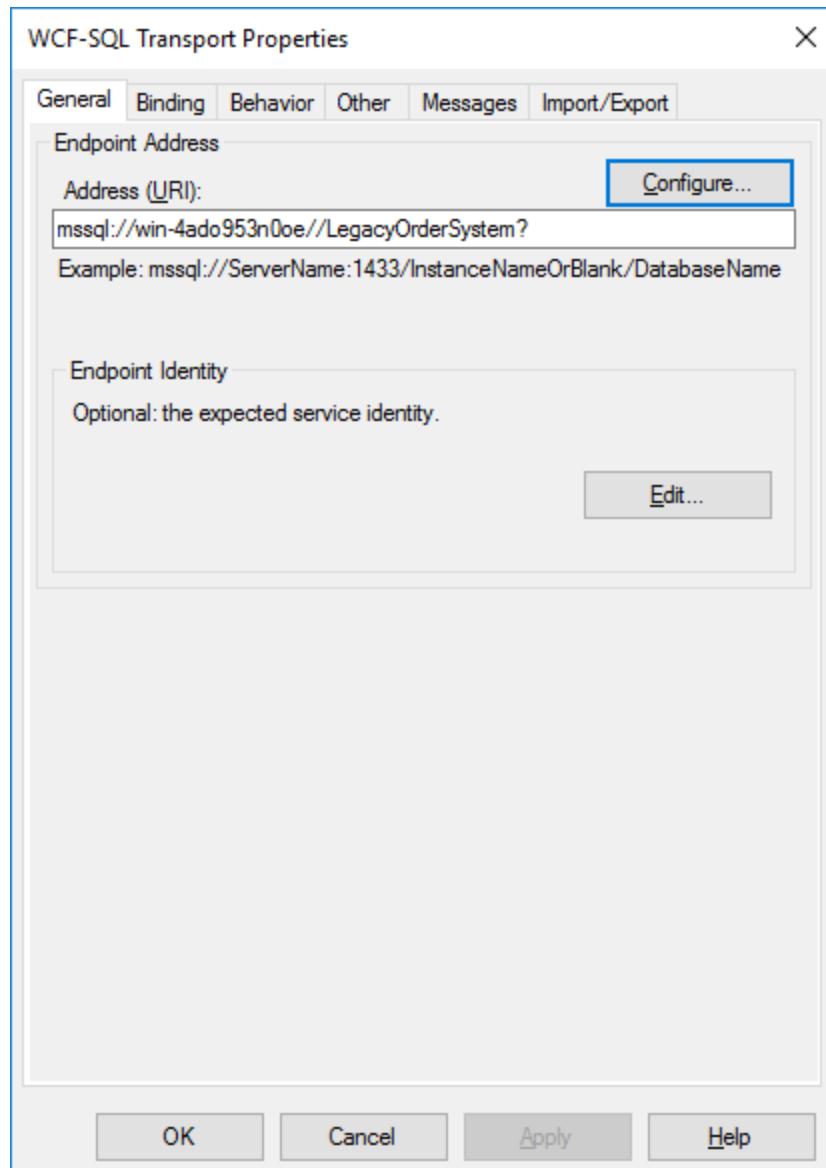
Description:



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



GLOBAL INTEGRATION BOOTCAMP



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]

WCF-SQL Transport Properties

General Binding Behavior Other Messages Import/Export

(FOR XML)
 XmlStoredProcedureRootNodeName: **Invoices**
 XmlStoredProcedureRootNodeName:

AvailabilityGroup
 ApplicationIntent: **ReadWrite**
 MultiSubnetFailover: **True**

BizTalk
 EnableBizTalkCompatibilityMode: **True**

Connection
 Encrypt: **False**
 MaxConnectionPoolSize: **100**
 WorkstationId:

Diagnostics
 EnablePerformanceCounters: **False**

General
 CloseTimeout: **00:01:00**
 Name: **sqlBinding**
 OpenTimeout: **00:01:00**
 ReceiveTimeout: **00:10:00**
 SendTimeout: **00:01:00**

Inbound
 InboundOperationType: **XmlPolling**

Metadata
 UseDatabaseNameInXsdNames: **False**

Notification (Inbound)
 NotificationStatement:
 NotifyOnListenerStart: **True**

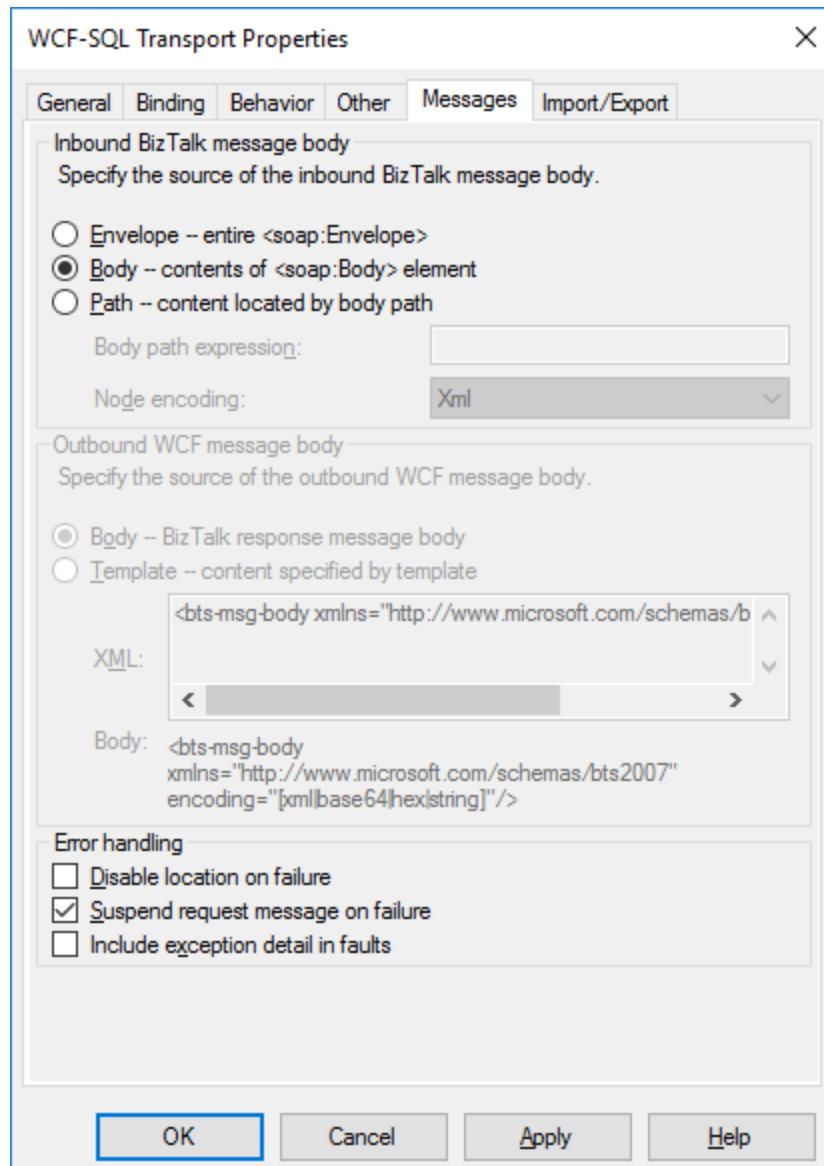
Polling (Inbound)
 PolledDataAvailableStatement: **SELECT COUNT([ID]) FROM [Log] WHERE [ID] > @LastID**
 PollingIntervalInSeconds: **5**
 Polling Statement: **EXEC [dbo].[GetHandledOrders]**
 PollWhileDataFound: **False**

Transaction
 UseAmbient Transaction: **True**

PollingIntervalInSeconds

OK Cancel Apply Help

GLOBAL INTEGRATION BOOTCAMP



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

RP_InvoiceOrders - Receive Port Properties X

General
Receive Locations
Inbound Maps
Tracking

Inbound Maps

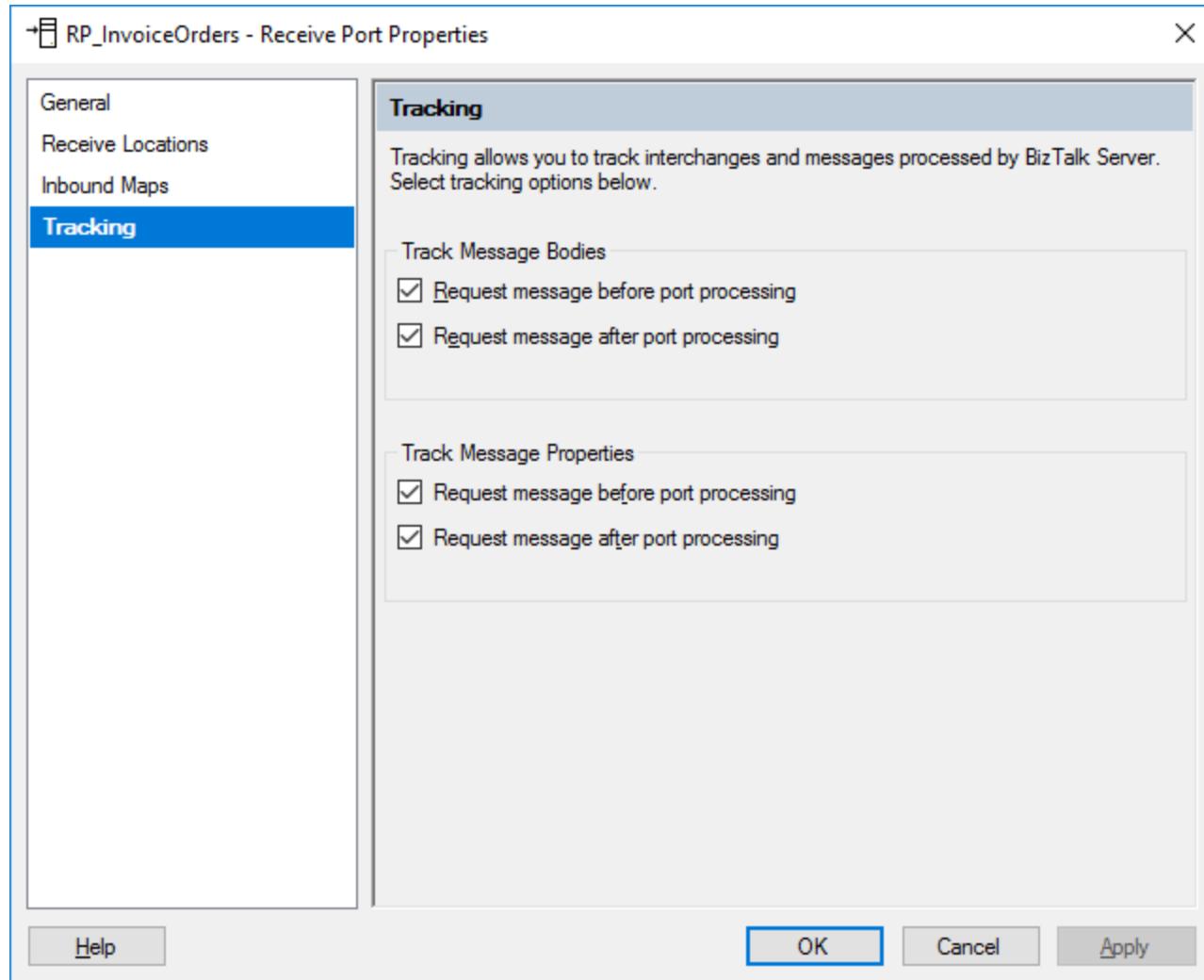
The following are the inbound maps used for transforming documents on the current port.

Inbound maps:

X Remove

	Source Document	Map	Target Document
▶	Invoices_XML [Glob...	Invoices_XML_To_Ord...	Order_XML [Global...
*			

Help OK Cancel Apply



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.

GLOBAL INTEGRATION BOOTCAMP

Policy: RootManageSharedAccessKey

GlobalIntegrationBootcamp - PREVIEW

Save changes Discard changes Regen prim key Regen sec key More

Policy name
RootManageSharedAccessKey

Claim

Manage

Send

Listen

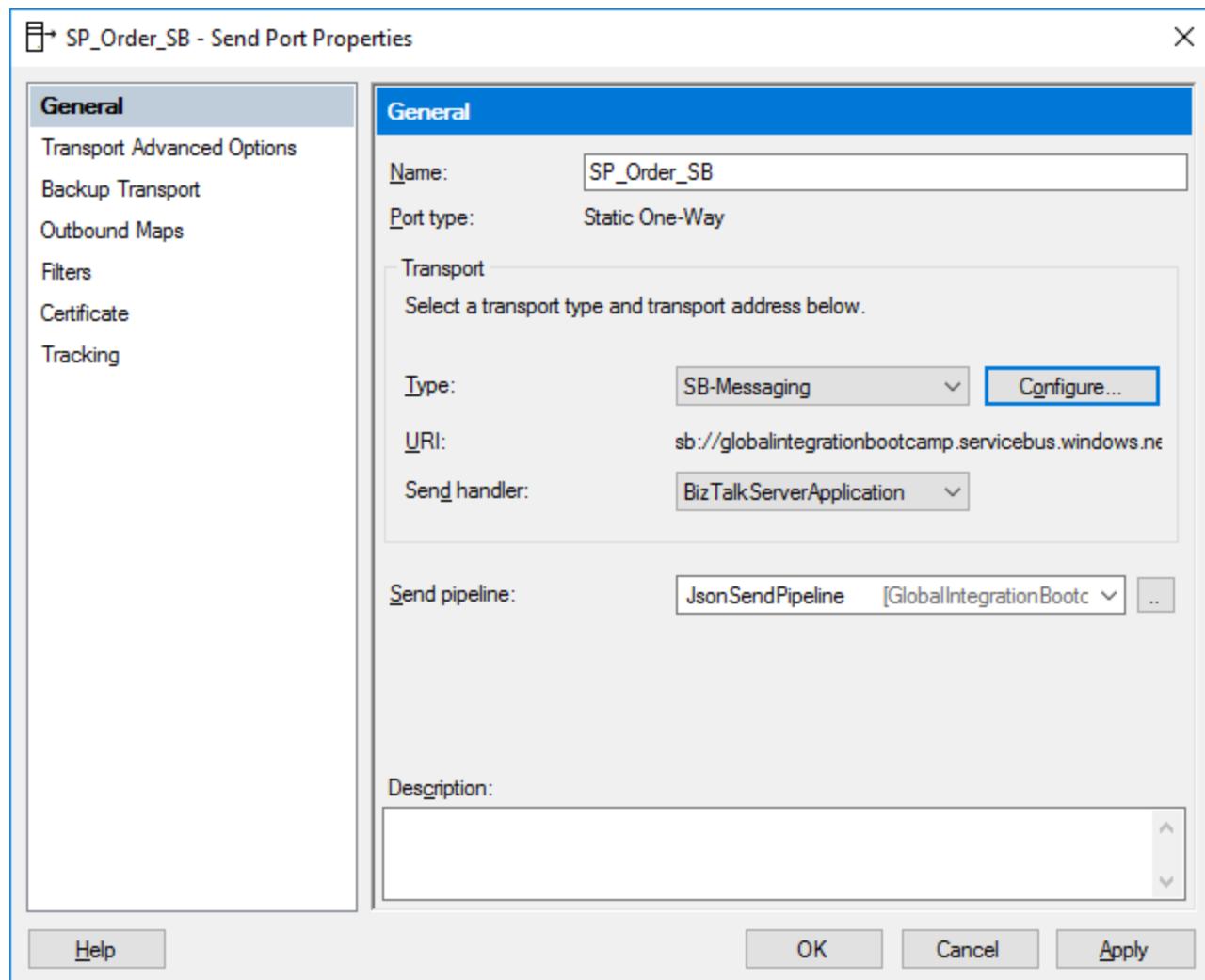
PRIMARY KEY

SECONDARY KEY

CONNECTION STRING
-PRIMARY KEY

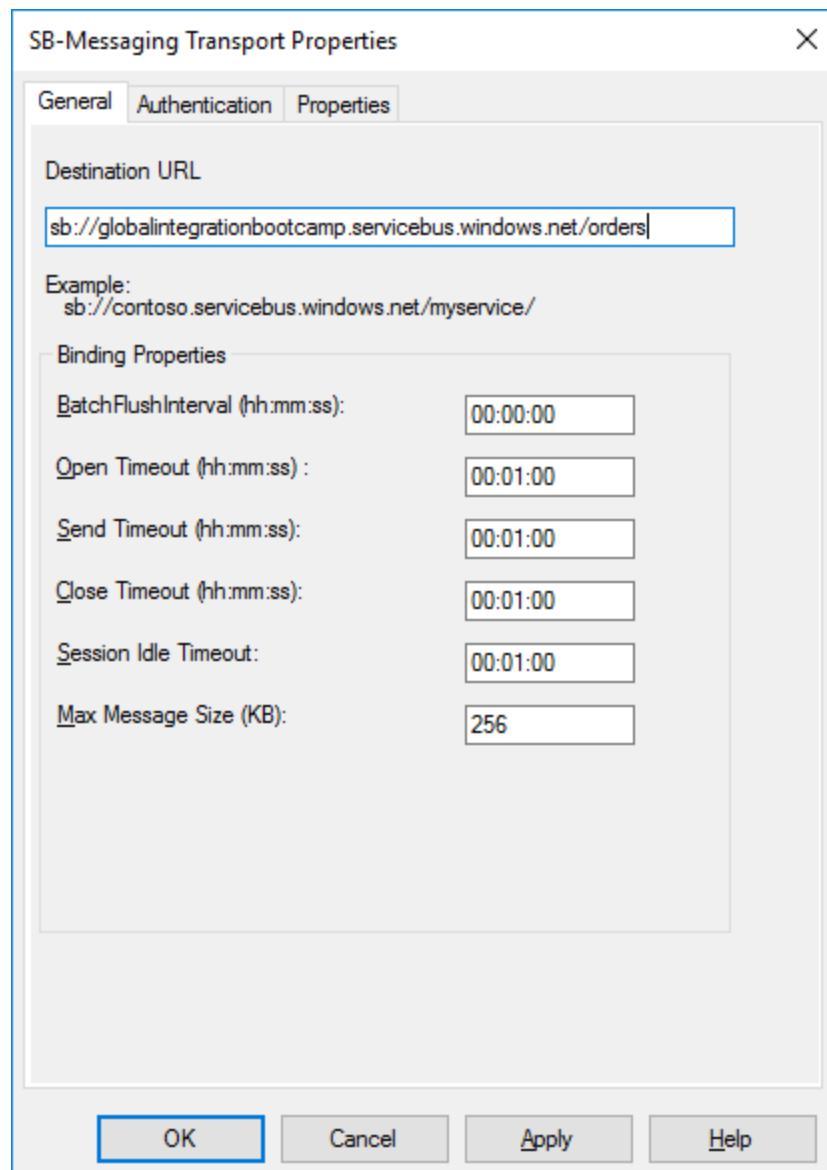
CONNECTION STRING
-SECONDARY KEY

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

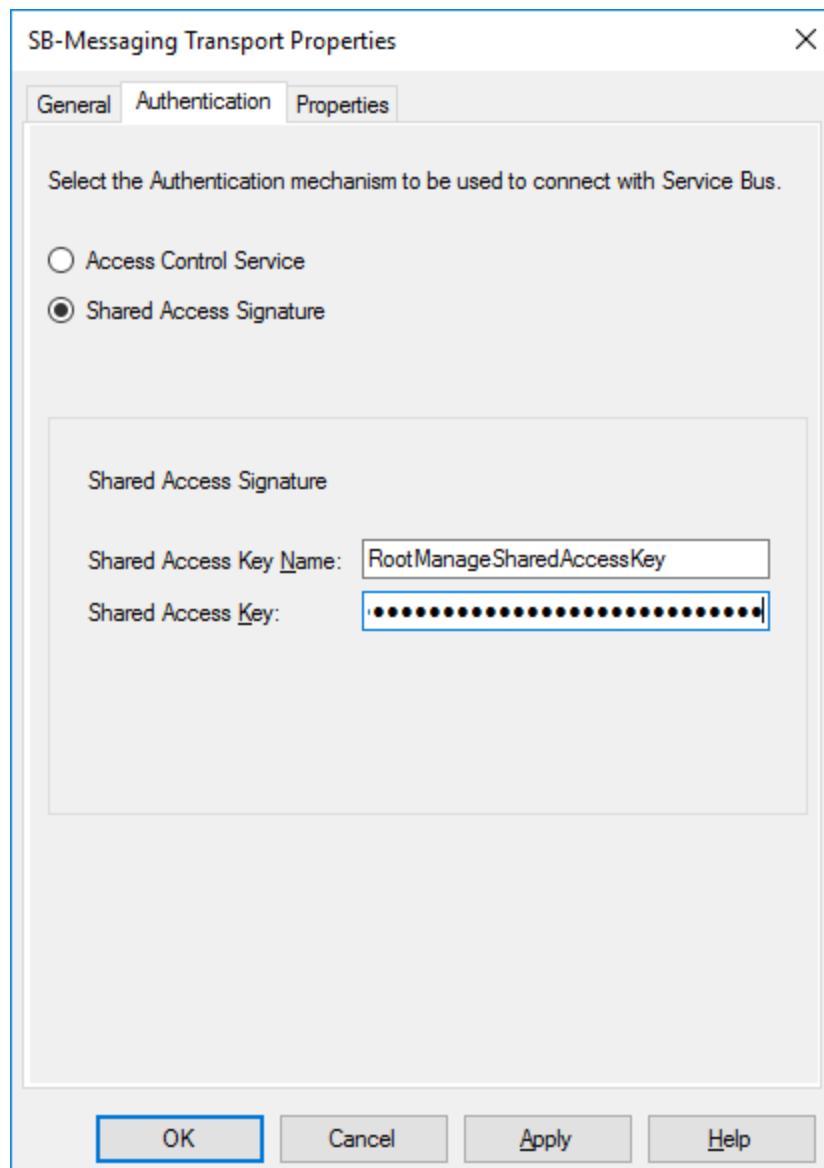


In the adapter properties, set the correct namespace in the destination URL, and add the name of the topic as the suffix.

GLOBAL INTEGRATION BOOTCAMP

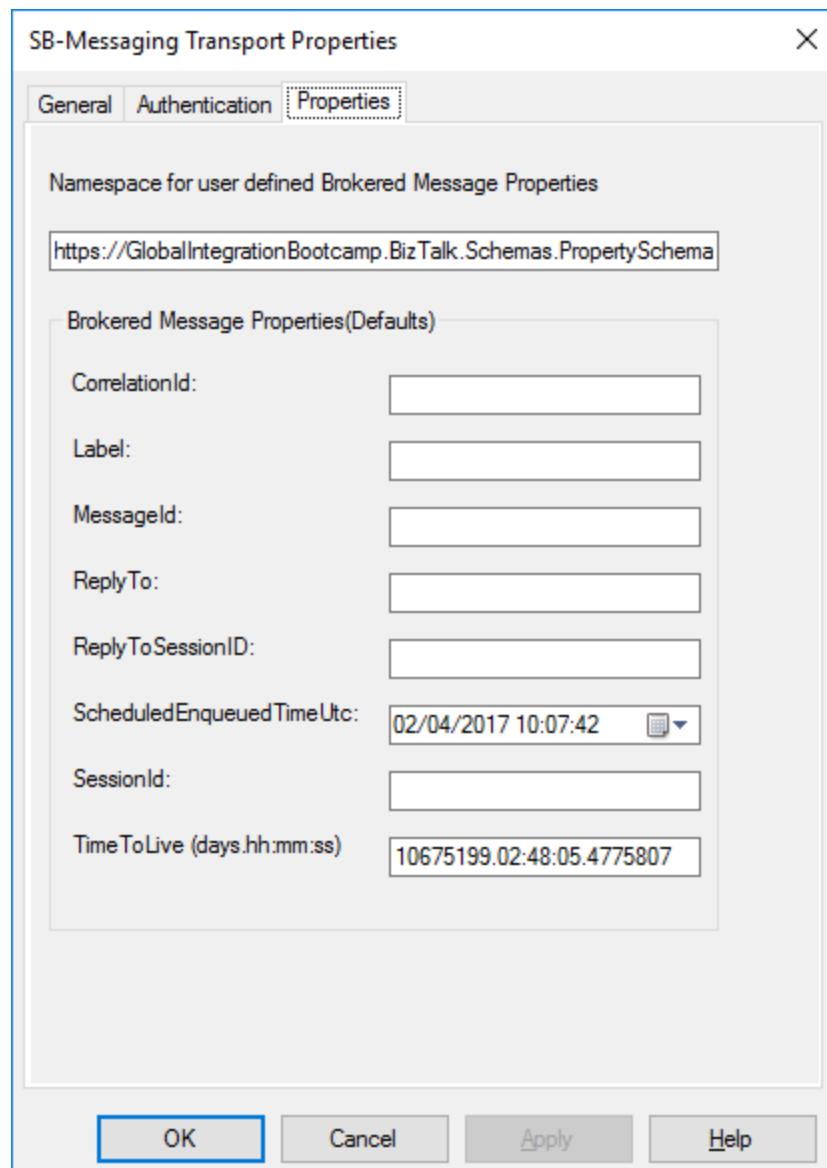


Set the authentication name and key we retrieved earlier.



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>.

GLOBAL INTEGRATION BOOTCAMP



Subscribe to the messages we received from the database.

SP_Order_SB - Send Port Properties X

General
Transport Advanced Options
Backup Transport
Outbound Maps
Filters
Certificate
Tracking

Filters

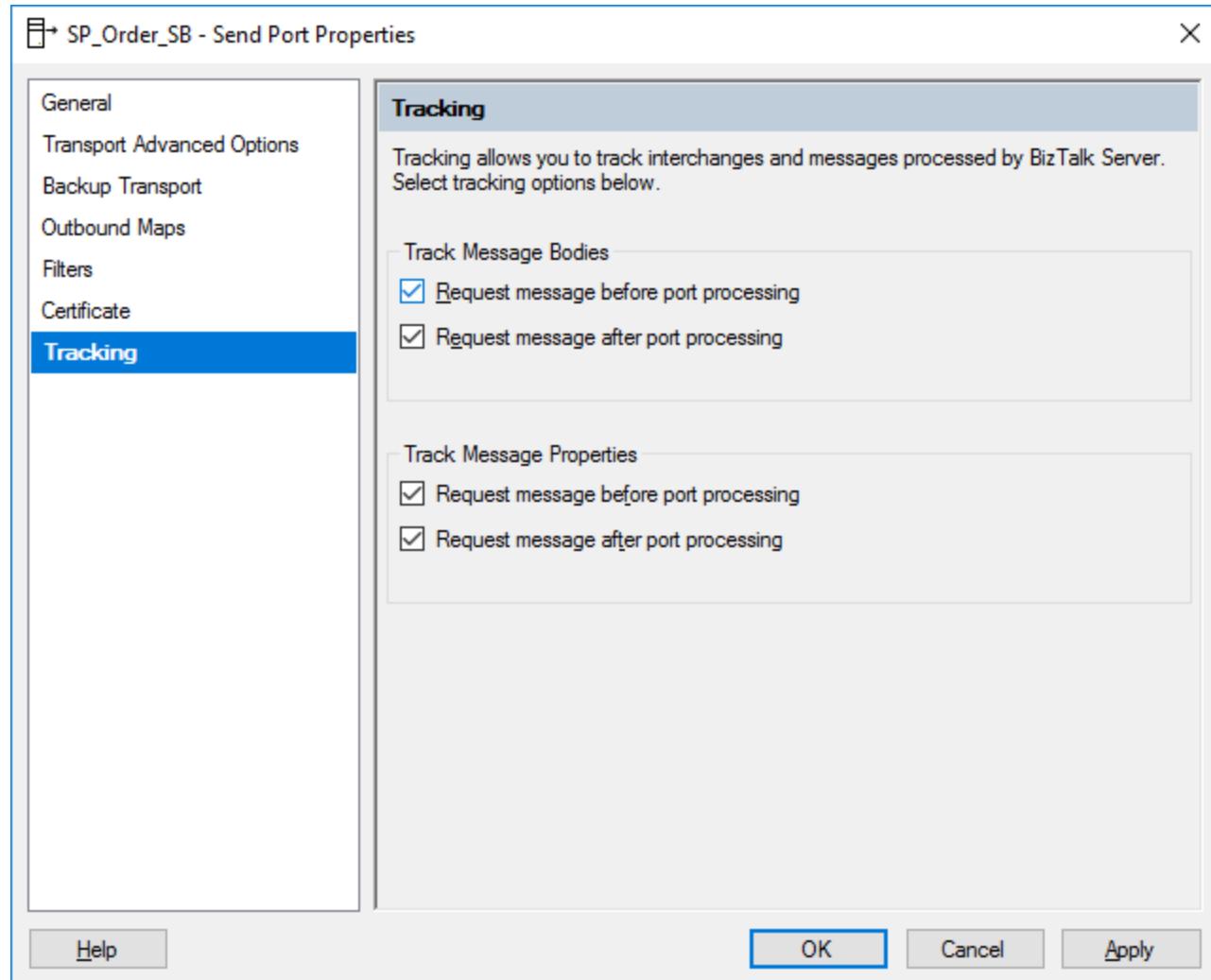
Filter expressions determine which messages are routed to this Send Port from the Message Box. Create one or more filter expressions using any properties in the message context.

✖ Delete ⌂ Move Up ⌄ Move Down

	Property	Operator	Value	Group by
▶	BTS.ReceivePortName	==	RP_InvoiceOrders	And
*				

BTS.ReceivePortName == RP_InvoiceOrders

Help OK Cancel Apply

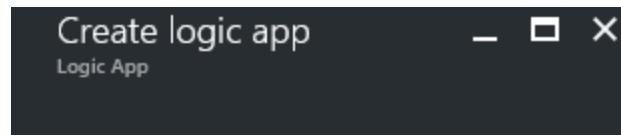


Once everything has been set up, start the BizTalk application.

GLOBAL INTEGRATION BOOTCAMP

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 to our BizTalk. Open the Azure portal, go to the [Logic Apps blade](#), and create a new Logic App.



* Name
GlobalIntegrationBootcamp ✓

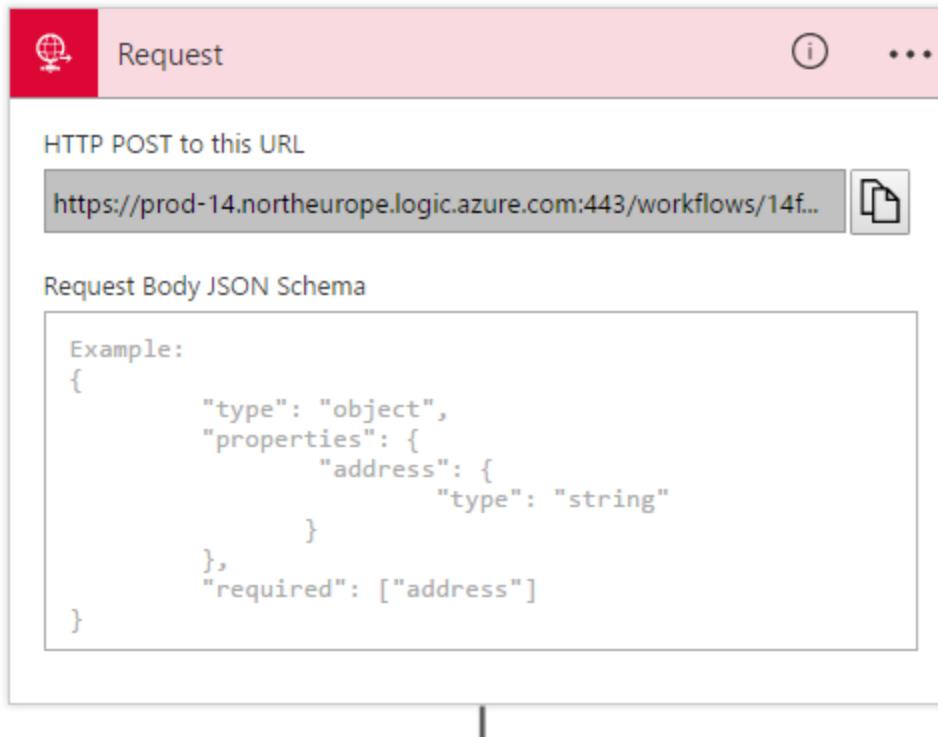
* Subscription
Converted Windows Azure MSDN - Visual ✓

* Resource group ⓘ
 Create new Use existing
GlobalIntegrationBootcamp ✓

Location
North Europe

 You can add triggers and actions to your Logic App after creation.

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.



The screenshot shows the 'Request' blade of the Microsoft Azure Logic Apps interface. At the top, there's a red header bar with a globe icon, the word 'Request', and three dots for more options. Below the header, the URL for the API is displayed as 'HTTP POST to this URL' followed by <https://prod-14.northeurope.logic.azure.com:443/workflows/14f...>. To the right of the URL is a copy icon. Underneath the URL, the 'Request Body JSON Schema' section is shown. It contains an example schema in JSON format:

```
Example:  
{  
    "type": "object",  
    "properties": {  
        "address": {  
            "type": "string"  
        }  
    },  
    "required": ["address"]  
}
```

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

GLOBAL INTEGRATION BOOTCAMP

The diagram illustrates a process flow. It starts with a red box labeled "Request" containing a globe icon. An arrow points down to a grey box labeled "BizTalk Server - Prepare message from JSON" containing a server icon. Both boxes have three dots in the top right corner.

BizTalk Server - Prepare message from JSON

...
...

GATEWAYS

Connect via on-premise data gateway ⓘ

* BIZTALK SERVER URL
http://WIN-4ADO953N0OE/IISLogicApp

* AUTHENTICATION TYPE
Windows

USERNAME
WIN-4ADO953N0OE\Administrator

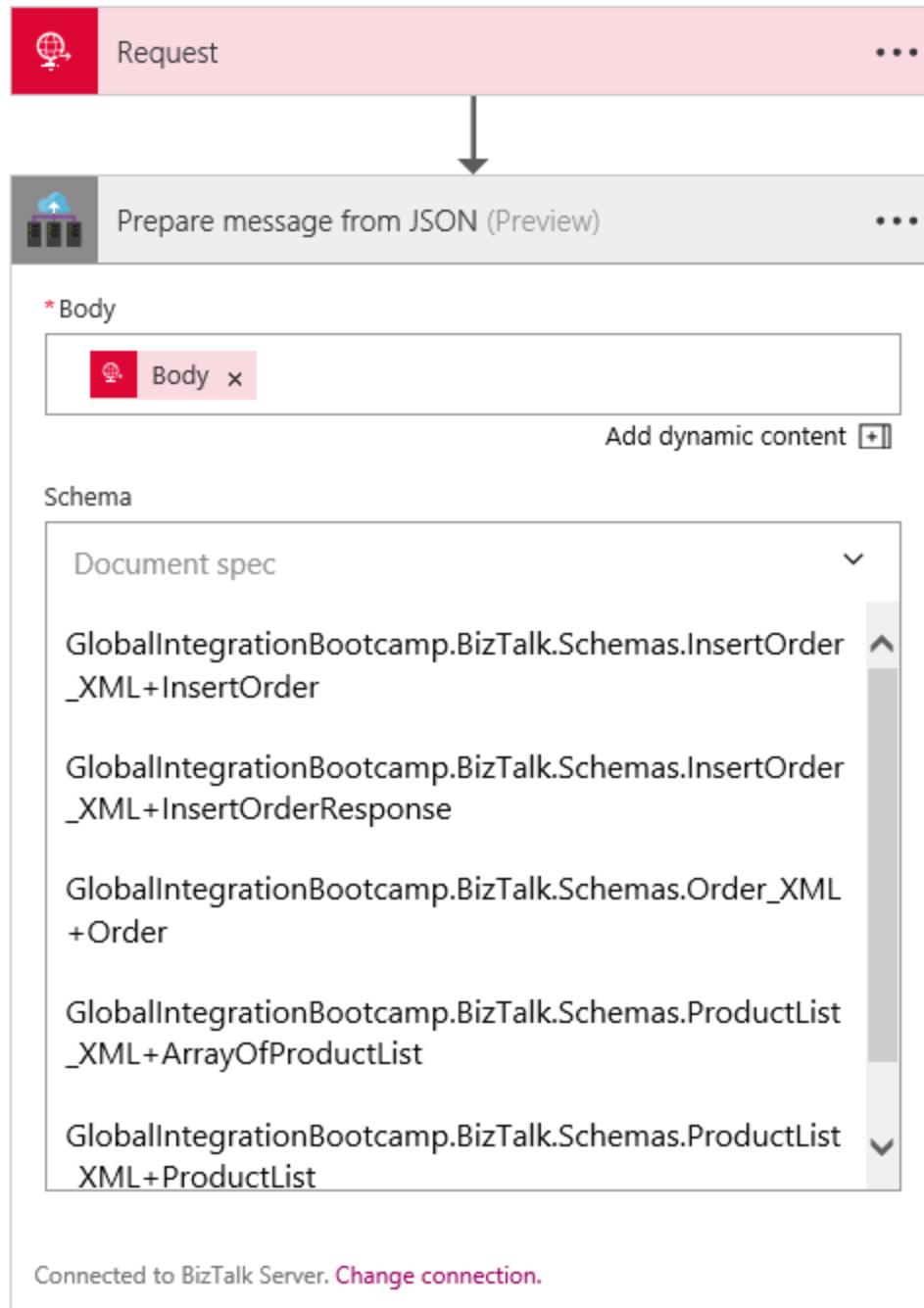
PASSWORD

* GATEWAY
GlobalIntegrationBootcamp

Create

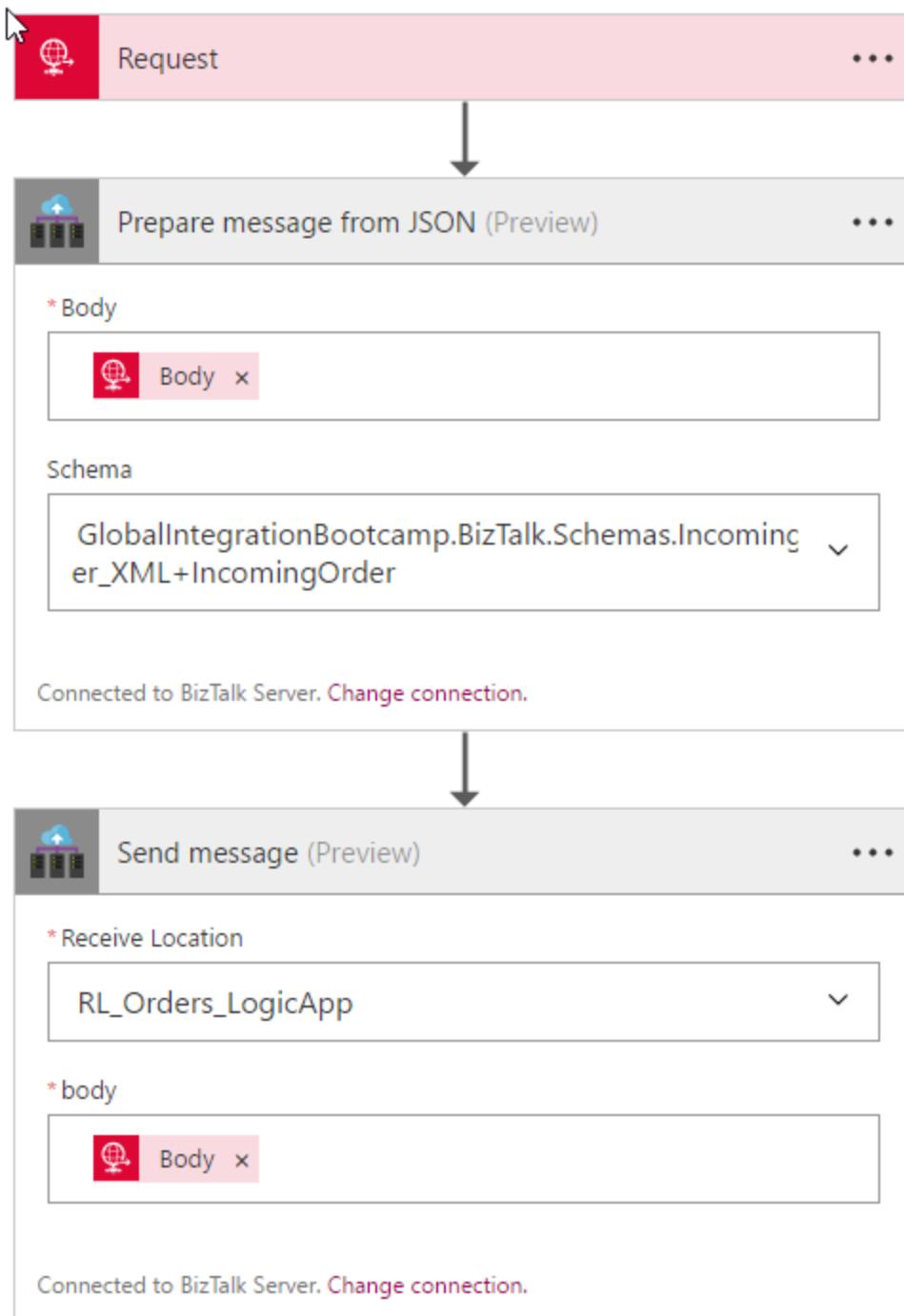
With the gateway selected, set the body of the message to the incoming body. The BizTalk connector will query our on premise BizTalk server, and retrieve the message types installed.

GLOBAL INTEGRATION BOOTCAMP



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.

GLOBAL INTEGRATION BOOTCAMP



 **GLOBAL INTEGRATION BOOTCAMP**

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  
    }  
}
```

GLOBAL INTEGRATION BOOTCAMP

The screenshot shows the Postman application interface. At the top, there are tabs for 'Runner', 'Import', and 'Builder' (which is selected). Below the tabs, there are tabs for 'New Tab', 'https://prod-07.northeurope...', 'https://prod-14.north...', and a '+' button. To the right of these tabs, it says 'No Environment' with a dropdown arrow. On the far right are icons for 'IN SYNC', a bell, and a gear.

In the main area, there's a 'POST' dropdown set to 'https://prod-14.northeurope.logic.azure.com:443/workflows/14f18ca7c9344060a77535289b2abdec...' and a 'Params' button. To the right are 'Send' and 'Save' buttons.

Below the URL, there are tabs for 'Authorization', 'Headers (1)', 'Body' (which is selected), 'Pre-request Script', and 'Tests'. Under 'Body', there are radio buttons for 'form-data', 'x-www-form-urlencoded', 'raw', 'binary', and 'JSON (application/json)' (which is selected).

The 'Body' section contains a JSON payload:

```
1 [{}]
2 "Order": {
3     "Customer": {
4         "Company": "Motion10",
5         "Email": "eldert.grootenboer@motion10.com",
6         "CustomerNumber": "6f6d4907-23af-e611-80e5-5065f38a5a01",
7         "Address": {
8             "Street": "Wilhelminakade 175",
9             "City": "Rotterdam",
10            "PostalCode": "3072AP",
11            "Country": "Netherlands"
12        }
13    },
14    "Products": {
15        "Product": [
16            {
17                "ProductNumber": 1000,
18                "Amount": 1,
19                "Price": 123.45
20            },
21            {
22                ...
23            }
24        ]
25    }
26]
```

Below the JSON payload, there are tabs for 'Body', 'Cookies', 'Headers (20)', and 'Tests'. To the right, it says 'Status: 202 Accepted' and 'Time: 271 ms'. At the bottom, there are buttons for 'Pretty', 'Raw', 'Preview', 'Text', and a search icon.

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

The screenshot shows the Azure Logic App portal. At the top, there are buttons for 'Run Trigger', 'Refresh', 'Edit', 'Delete', 'Disable', 'Update Schema', and 'Cancel Run'. To the right, it says 'Logic app run' and '08587153574641207599738836079'.

On the left, there's a sidebar with 'Essentials' and 'Summary' sections. The 'Essentials' section shows details like 'Resource group (change) GlobalIntegrationBootcamp', 'Location North Europe', 'Subscription name (change) Converted Windows Azure MSDN - Visual ...', and 'Subscription ID 1982dd27-5042-4968-802a-c9b9578e2543'. The 'Summary' section shows 'All runs' and a table of runs:

STATUS	START TIME	DURATION
Succeeded	2/5/2017, 12:59 PM	2.27 Seconds
Succeeded	2/5/2017, 12:03 AM	670 Milliseconds

On the right, there's a flowchart titled 'Logic app run' showing three steps: 'Request', 'Prepare message from JSON', and 'Send message', all with a duration of 0s and a green checkmark.

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



GLOBAL INTEGRATION BOOTCAMP

Query Expression

Field Name	Operator	Value
► Search For	Equals	Tracked Service Instances
Maximum Matches	Equals	50
*		

Run Query
Save As...
Open Query...

Query results (50 items were found):

Service Name	Host Name	Service Class	State	Start Time	End Time	Duration	Error Code	Error Description
Microsoft.BizTalk.DefaultPipelines.PassThruTransmit	BizTalkServerAp...	Pipeline	Completed	5-2-2017 13:02:29.633	5-2-2017 13:02:2...	33	0	
GlobalIntegrationBootcamp.BizTalk.Pipelines.JsonReceivePipeline	BizTalkServerIsol...	Pipeline	Completed	5-2-2017 13:02:29.123	5-2-2017 13:02:2...	80	0	

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.

SQLQuery2.sql - WI...Administrator (78)* SQLQuery1.sql - WI...Administrator (59)*

```
***** Script for SelectTopNRows command from SSMS *****
SELECT TOP (1000) [ID]
    ,[CustomerNumber]
    ,[Placed]
    ,[TotalAmount]
    ,[CustomerType]
    ,[InvoiceSent]
FROM [LegacyOrderSystem].[dbo].[Order]
```

100 %

Results Messages

ID	CustomerNumber	Placed	TotalAmount	CustomerType	InvoiceSent
1	8	2016-11-20 14:26:00.000	2407.35	NULL	NULL

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'`

Group Overview

Refresh this page to update results (press F5)

Group Hub Running Tracked service instances Suspended New Query

Query Expression

Field Name	Operator	Value
► Search For	Equals	Tracked Service Instances
Maximum Matches	Equals	50
*		

Run Query
Save As...
Open Query...

Query results (50 items were found):

Service Name	Host Name	Service Class	State	Start Time	End Time	Duration	Error Code	Error Description
GlobalIntegrationBootcamp.BizTalk.Pipelines.JsonSendPipeline	BizTalkServerAp...	Pipeline	Completed	5-2-2017 13:12:42.337	5-2-2017 13:12:4...	970	0	
Microsoft.BizTalk.DefaultPipelines.XMLReceive	BizTalkServerAp...	Pipeline	Completed	5-2-2017 13:12:41.587	5-2-2017 13:12:4...	564	0	

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

GLOBAL INTEGRATION BOOTCAMP

The screenshot shows the Service Bus Explorer interface. On the left, the Service Bus Namespace tree includes Queues, Topics, and the orders topic, which contains Subscriptions (Business (0, 0) and Consumer (1, 0)). The main area displays the 'View Subscription: Consumer' window. It has tabs for Description, Metrics, and Messages, with the Messages tab selected. The Message List table shows one message: MessageId 91f1ac71104d..., Seq 1, Size 916, EnqueuedTimeUtc 4-2-2017 23:05, and ExpiresAtUtc 31-12-9999 23:59. Below the table is a Message Text pane showing JSON data for an order, and a Message Custom Properties pane showing CustomerType: Consumer. To the right is a large Message Properties pane with various properties like ContentType (application/xml; charset=utf-8), CorrelationId (91f1ac71104d4e099bc...), and ForcePersistence (False). At the bottom are buttons for Close Tabs, Messages, Deadletter, Refresh, Disable, Delete, and Update. A Log pane at the bottom shows several log entries related to the subscription.

```

<00:00:00> The subscription Log for the orders topic has been successfully retrieved.
<00:00:01> The subscription Log for the orders topic has been successfully retrieved.
<00:00:01> The subscription Log for the orders topic has been successfully retrieved.
<00:02:26> The subscription Log for the orders topic has been successfully retrieved.
<00:02:27> The subscription Log for the orders topic has been successfully retrieved.
<00:06:08> The subscription Log for the orders topic has been successfully retrieved.
<00:06:14> [1] messages peeked from the subscription [Log].
<00:06:16> The subscription Consumer for the orders topic has been successfully retrieved.
<13:14:56> The subscription Consumer for the orders topic has been successfully retrieved.
<13:15:09> The Log subscription for the orders topic has been successfully deleted.
<13:15:15> The subscription Consumer for the orders topic has been successfully retrieved.
<13:15:15> [1] messages received from the subscription [Consumer].

```

GLOBAL INTEGRATION BOOTCAMP

The screenshot shows the Service Bus Explorer interface. On the left, the Service Bus Namespace tree includes 'sb://globalintegrationbootcamp.servicebus.windows.net/' with Queues, Topics, and the 'orders' topic selected. The 'orders' topic has 'Subscriptions' (Business (0, 0) and Consumer (1, 0)). The main area is titled 'View Subscription: Consumer' and shows a 'Message List' tab with one message. The message details are as follows:

MessageId	Seq	Size	Label	EnqueuedTimeUtc	ExpiresAtUtc
91f1ac71104d...	1	916		4-2-2017 23:05	31-12-9999 23:59

The 'Message Text' pane displays the JSON message body:

```
{
  "Order": {
    "Customer": {
      "Company": "Motion10",
      "Email": "eldert.grootenboer@motion10.com",
      "CustomerNumber": "6f6d4907-23af-e611-80e5-5065f38a5a01",
      "Address": {
        "Street": "Wilhelminakade 175",
        "City": "Rotterdam",
        "PostalCode": "3872AP",
      }
    }
  }
}
```

The 'Message Custom Properties' pane shows:

Name	Value
CustomerType	Consumer

The 'Message Properties' pane on the right lists various properties:

- Msc**
 - ContentType: application/xml; charset=utf-8
 - CorrelationId: 91f1ac71104d4e099b
 - DeliveryCount: 1
 - EnqueuedSequenceNumber: 3
 - EnqueuedTimeUtc: 4-2-2017 23:05
 - ExpiresAtUtc: 31-12-9999 23:59
 - ForcePersistence: False
 - IsBodyConsumed: False
 - Label:
 - LockedUntilUtc: This operation is only supported for死锁.
 - LockToken: This operation is only supported for死锁.
 - MessageId: 91f1ac71104d4e099b
 - PartitionKey:
 - Properties:
 - ReplyTo:
 - ReplyToSessionId: 4-2-2017 22:07
 - ScheduledEnqueueTimeUtc: 4-2-2017 22:07
 - SequenceNumber: 1
 - SessionId: 916
 - State: Active
 - TimeToLive: 10675199.02:48:05.47
 - To:

Below the main tabs are buttons: Close Tabs, Messages, Deadletter, Refresh, Disable, Delete, Update. A 'Log' pane at the bottom shows the following log entries:

```

<00:00:00> The subscription Log for the orders topic has been successfully retrieved.
<00:00:01> The subscription Log for the orders topic has been successfully retrieved.
<00:00:01> The subscription Log for the orders topic has been successfully retrieved.
<00:02:26> The subscription Log for the orders topic has been successfully retrieved.
<00:02:27> The subscription Log for the orders topic has been successfully retrieved.
<00:06:08> The subscription Log for the orders topic has been successfully retrieved.
<00:06:14> [1] messages peeked from the subscription [Log].
<00:06:16> The subscription Consumer for the orders topic has been successfully retrieved.
<13:14:56> The subscription Consumer for the orders topic has been successfully retrieved.
<13:15:09> The Log subscription for the orders topic has been successfully deleted.
<13:16:15> The subscription Consumer for the orders topic has been successfully retrieved.
<13:16:15> [1] messages received from the subscription [Consumer].

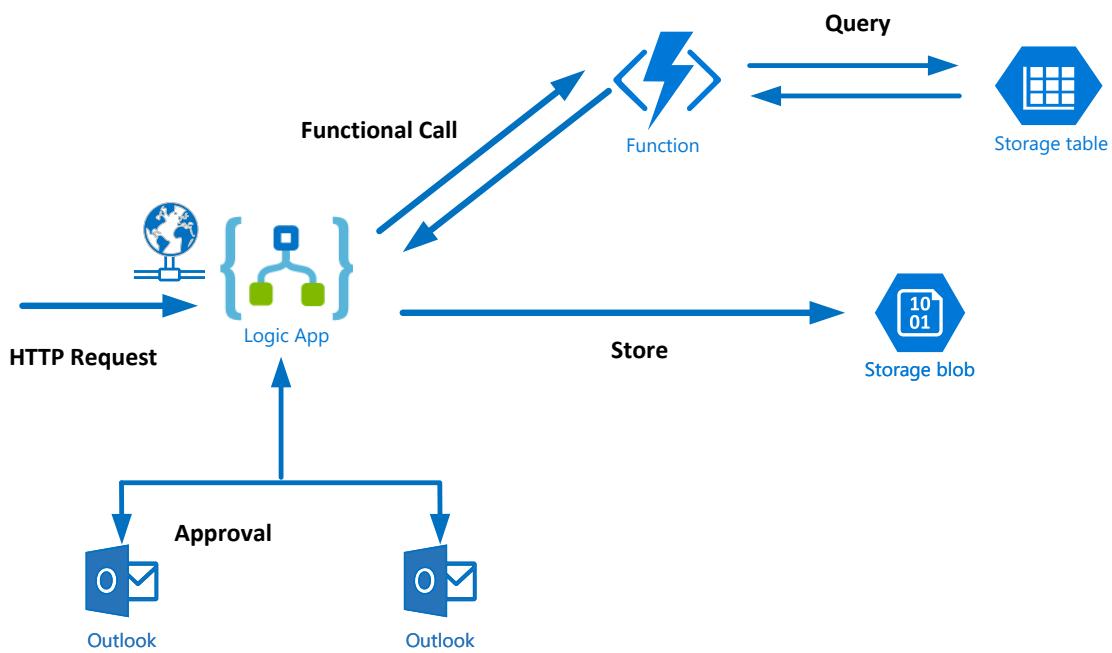
```

GLOBAL INTEGRATION BOOTCAMP

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

GLOBAL INTEGRATION BOOTCAMP

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.

Microsoft Azure RG GlobalIntegration > Everything > Storage account

Storage account

Microsoft Azure provides scalable, durable cloud storage, backup, and recovery solutions for any data, big or small. It works with the infrastructure you already have to cost-effectively enhance your existing applications and business continuity strategy, and provide the storage required by your cloud applications, including unstructured text or binary data such as video, audio, and images.

PUBLISHER Microsoft

NAME Storage account

RESULTS

NAME	PUBLISHER	CATEGORY
Storage account	Microsoft	Storage
SafeNet ProtectV Service Gateway, 200 Nodes	Gemalto	Compute
Mail2Cloud Archive 250 User Pack	mxHero Mail2Cloud	Compute
HPE ArcSight Logger	Hewlett Packard Enterprise	Compute
SafeNet ProtectV (BYOL)	Gemalto	Compute
SafeNet ProtectV Service Gateway, 100 Nodes	Gemalto	Compute
SafeNet ProtectV Service Gateway, 50 Nodes	Gemalto	Compute
Mail2Cloud Archive 2500 User Pack	mxHero Mail2Cloud	Compute
Mail2Cloud Archive 50 User Pack	mxHero Mail2Cloud	Compute
HPE ArcSight Logger 6.2	Hewlett Packard Enterprise	Compute
Mail2Cloud Archive 100 User Pack	mxHero Mail2Cloud	Compute
Mail2Cloud Archive 500 User Pack	mxHero Mail2Cloud	Compute
Mail2Cloud Archive 5000 User Pack	mxHero Mail2Cloud	Compute
Mail2Cloud Archive 1000 User Pack	mxHero Mail2Cloud	Compute
Resource group	Microsoft	VM Extensions
Skysync for Office365/OneDrive/Sharepoint	Portal Architects	Compute
Automation	Microsoft	Developer tools

USEFUL LINKS

Service overview Documentation Pricing

Create

4. Click **Create**.

GLOBAL INTEGRATION BOOTCAMP

Create storage account

The cost of your storage account depends on the usage and the options you choose below.

[Learn more](#)

* Name ⓘ
globalintegration ✓
.core.windows.net

Deployment model ⓘ
Resource manager Classic

Account kind ⓘ
General purpose

Performance ⓘ
Standard Premium

Replication ⓘ
Read-access geo-redundant storage (RA...)

* Storage service encryption ⓘ
Disabled Enabled

* Subscription
Azure Free Trial

* Resource group ⓘ
 Create new Use existing
RG_GlobalIntegration

* Location
West Europe

Pin to dashboard

Create

Automation options

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**.

GLOBAL INTEGRATION BOOTCAMP

Create Storage Container

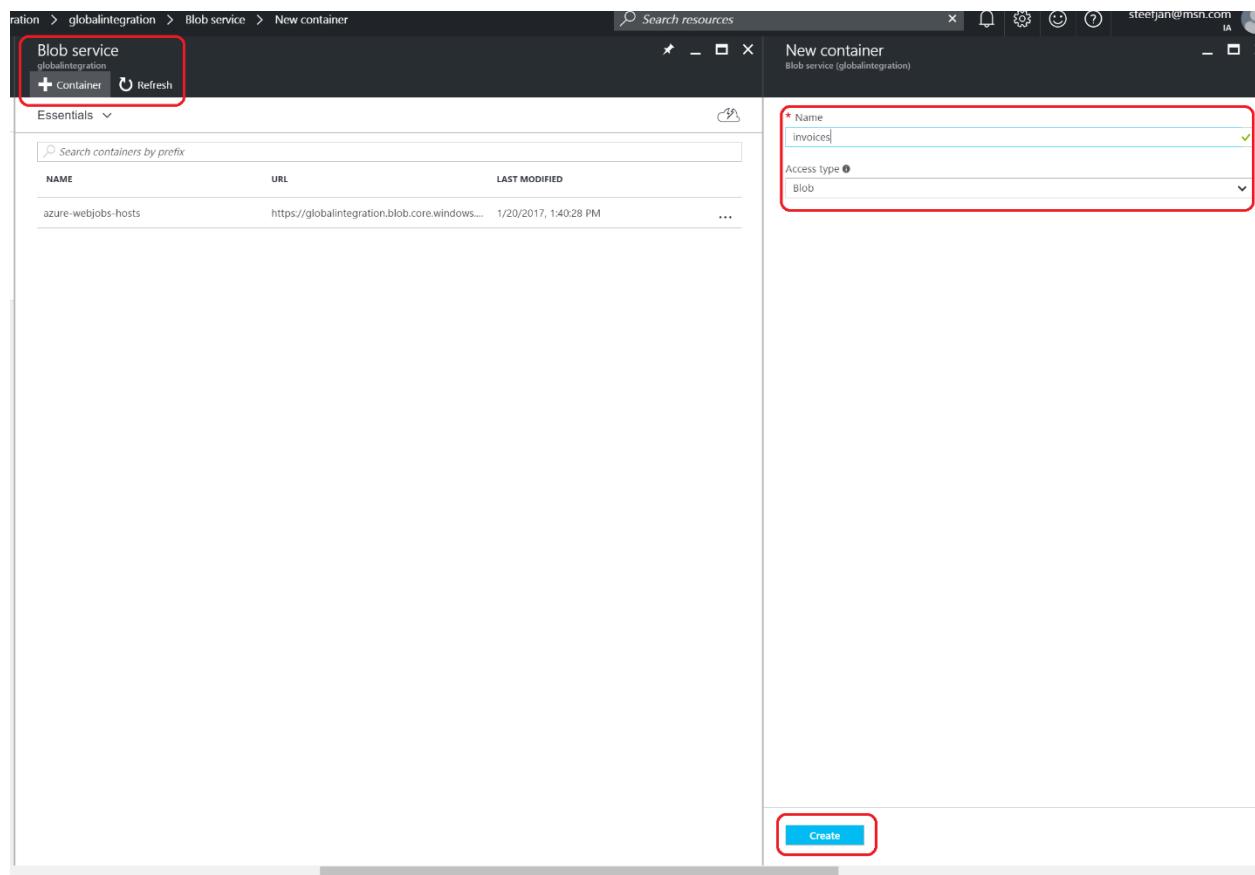
Once the storage account has been provisioned you can navigate to it and click on it.

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

The screenshot shows the Microsoft Azure portal interface for a storage account named 'globalintegration'. On the left, there's a sidebar with various icons. The main area is divided into two panes. The left pane displays the 'Essentials' section for the storage account, including details like Subscription name (Azure Free Trial), Subscription ID (0bf166ac-9aa8-4597-bb2a-a845afe01415), and Location (West Europe). It also shows a table with one item: 'globalintegration' (Storage account, West Europe). The right pane is titled 'globalintegration Storage account' and contains sections for 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', 'SETTINGS' (with options like Access keys, Configuration, Shared access signature, Properties, Locks, Automation script), 'BLOB SERVICE' (Containers, CORS, Custom domain), and 'Services' (Blobs, Files, Tables, Queues). The 'Blobs' service is highlighted with a blue border. Below the services is a 'Monitoring' section showing 'Total requests' with a note 'No available data.'

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

GLOBAL INTEGRATION BOOTCAMP

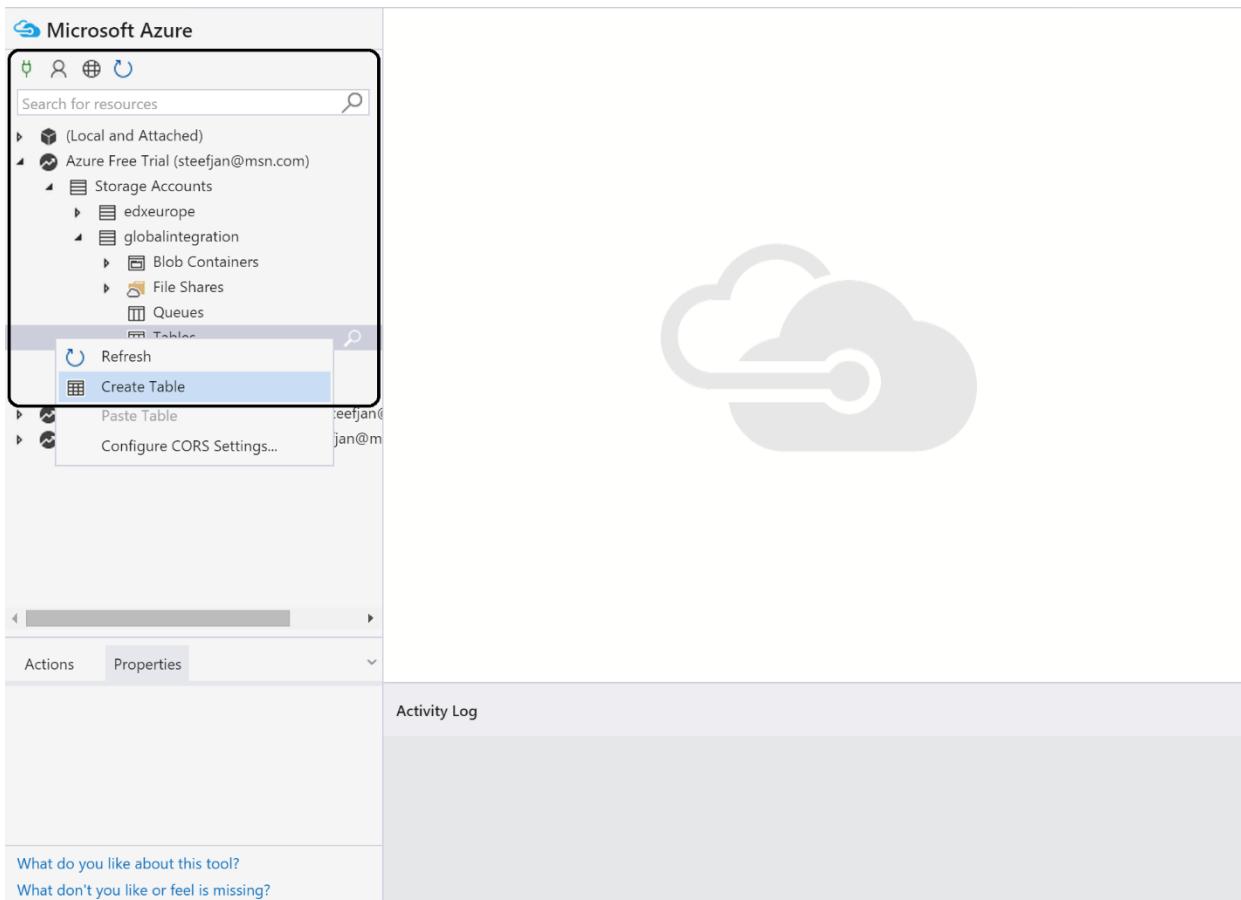


GLOBAL INTEGRATION BOOTCAMP

Create Storage Table

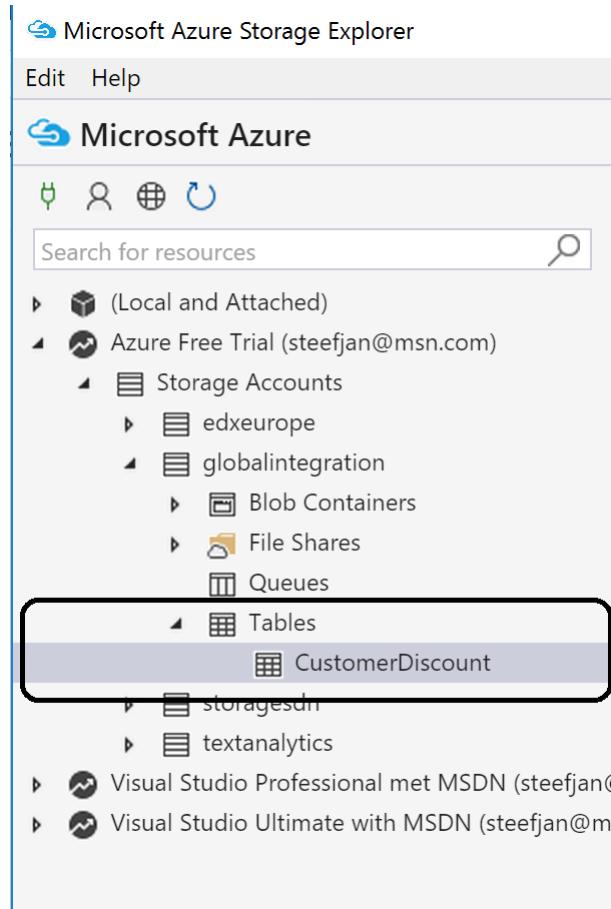
To create a storage table will use the **Azure Storage Explorer**, which can be downloaded from <http://storageexplorer.com/>.

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.

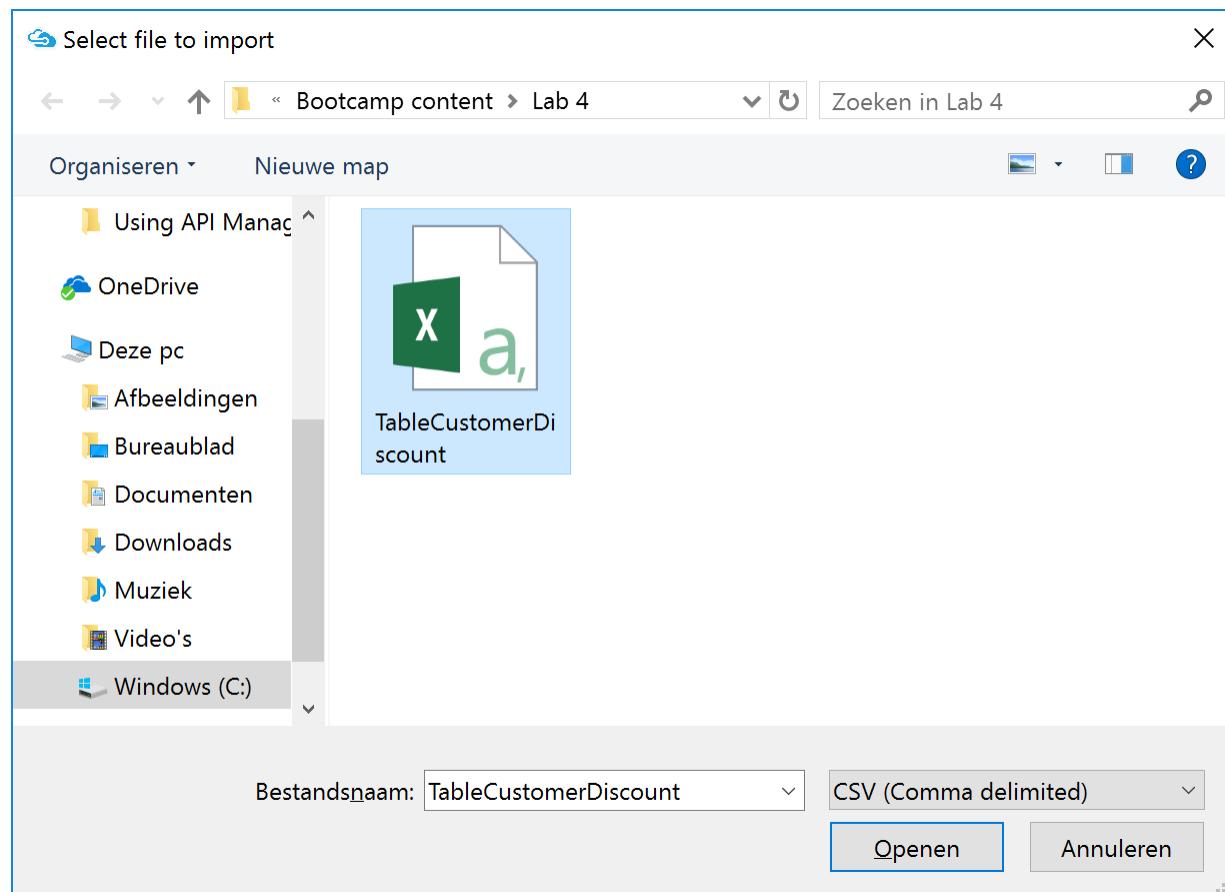


5. Specify a name for the table.

GLOBAL INTEGRATION BOOTCAMP



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.

GLOBAL INTEGRATION BOOTCAMP

Property Name	Data Type	Sample Value
PartitionKey	String	Company
RowKey	String	Motion10
Discount	Int32	10

Activity Log

- Imported 'C:\Macaw\Bootcamp content\Lab 4\TableCustomerDiscount.csv' into table 'globalintegration/CustomerDiscount' 2 entities imported
- Deleted entities from table 'globalintegration/CustomerDiscount' 2 entities deleted

10. Click on Insert.

11. Table will be loaded with data.

GLOBAL INTEGRATION BOOTCAMP

Microsoft Azure Storage Explorer

Microsoft Azure

Search for resources

PartitionKey	RowKey	Timestamp	Discount
Company	Macaw	2017-01-21T12:33:01.944Z	20
Company	Motion10	2017-01-21T12:33:01.943Z	10

Actions Properties

Showing 1 to 2 of 2 cached items.

Activity Log

- Imported 'C:\Macaw\Bootcamp content\Lab 4\TableCustomerDiscount.csv' into table 'globalintegration\CustomerDiscount' 2 entities imported
- Deleted entities from table 'globalintegration\CustomerDiscount' 2 entities deleted

Refresh Rename... Open Table Editor Copy Direct Link to Table Delete

What do you like about this tool? What don't you like or feel is missing?

13:34 NLD US 21-1-2017

GLOBAL INTEGRATION BOOTCAMP

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**.

The screenshot shows the Microsoft Azure Marketplace interface. The search bar at the top contains the text "Function App". The results list shows several items, with "Function App" by Microsoft highlighted. The right pane displays detailed information about the selected app, including its publisher (Microsoft) and useful links like Documentation, Solution Overview, and Pricing Details. A "Create" button is visible at the bottom right of the right pane.

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

GLOBAL INTEGRATION BOOTCAMP

Microsoft Azure RG_GlobalIntegration > Everything > Function App > Function App > Storage Account

Function App

Storage Account

* App name: GlobalIntegrationFunctions.azurewebsites.net

* Subscription: Azure Free Trial

* Resource Group: RG_GlobalIntegration

* Hosting Plan: Consumption Plan

* Location: West Europe

* Storage Account: function9d9d7410981a

Create New

globalintegration
West Europe

Pin to dashboard

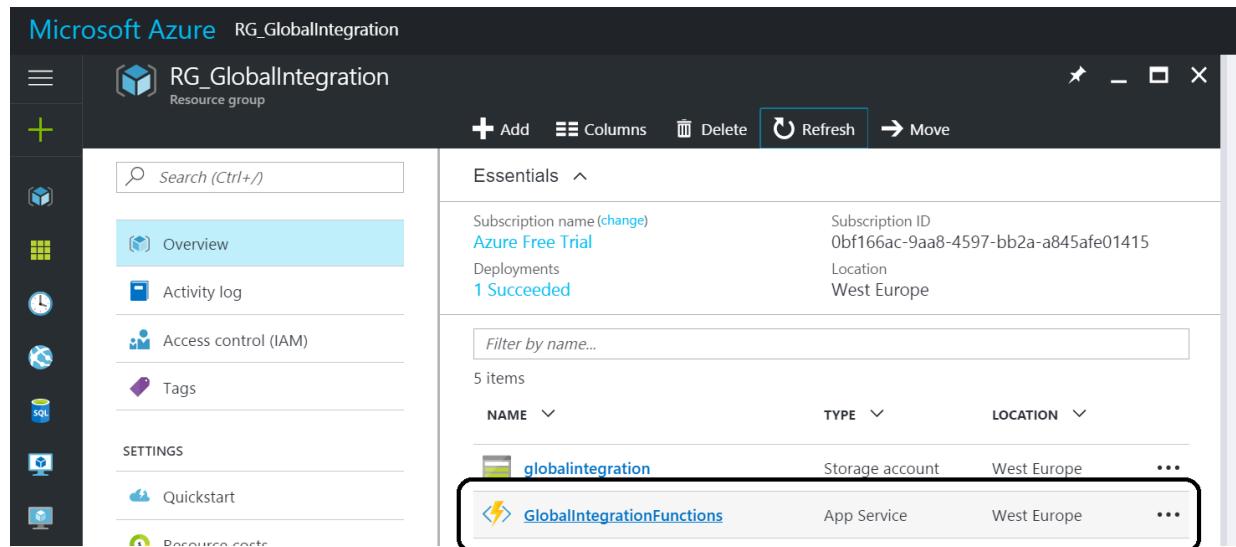
Create

GLOBAL INTEGRATION BOOTCAMP

Building a Function

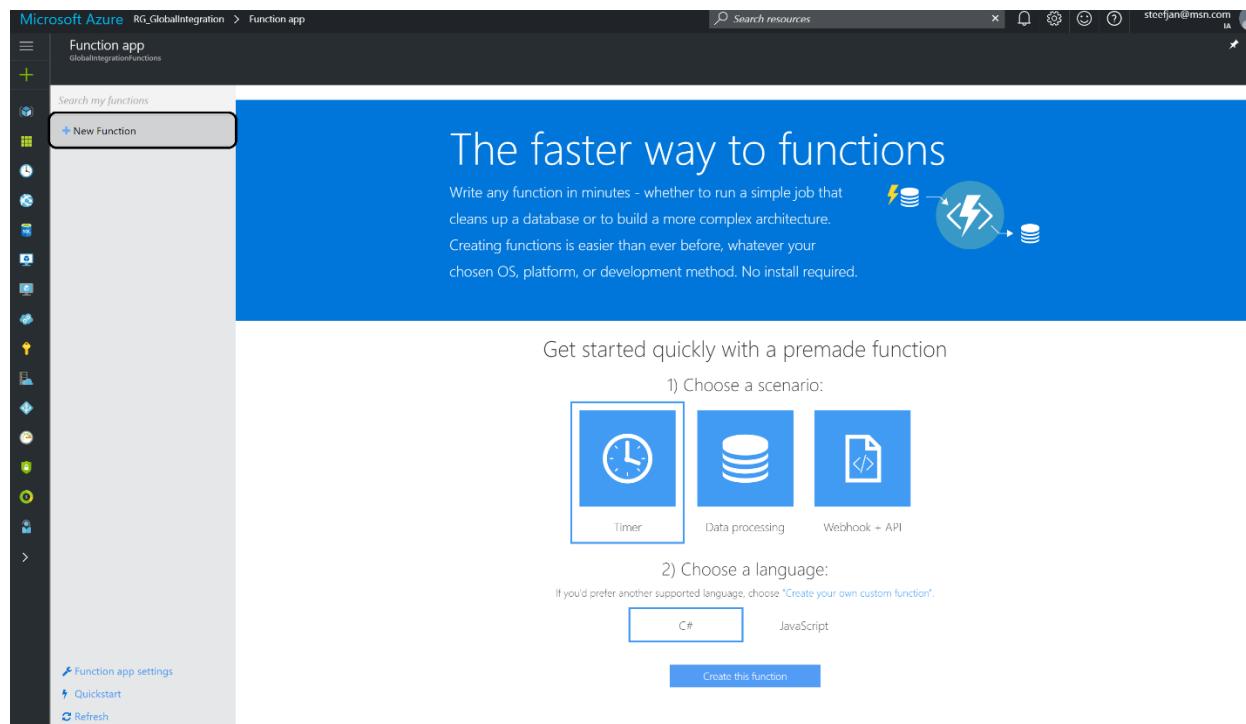
Once the function app is provisioned you can add function to it, build and test it.

15. Select the **Function App**.



The screenshot shows the Microsoft Azure portal interface. On the left, there's a sidebar with various icons for different services like Storage, Functions, and Logic Apps. The main area is titled "RG_GlobalIntegration" under "Resource group". It has a "Search (Ctrl+/" input field. Below it are links for "Overview", "Activity log", "Access control (IAM)", and "Tags". A "SETTINGS" section includes "Quickstart" and "Resource costs". The right side is titled "Essentials" and shows deployment details: "Subscription name (change) Azure Free Trial", "Subscription ID 0bf166ac-9aa8-4597-bb2a-a845afe01415", "Deployments 1 Succeeded", and "Location West Europe". A table lists resources: "globalintegration" (Storage account, West Europe), and "GlobalIntegrationFunctions" (App Service, West Europe). The "GlobalIntegrationFunctions" row is highlighted with a red box.

16. In the function app click + New Function.



The screenshot shows the "Function app" blade for "globalintegrationfunctions". The left sidebar has "Function app" selected. A "Search my functions" input field and a "+ New Function" button are visible. The main area has a blue header with the text "The faster way to functions" and subtext about creating functions quickly. It shows three icons: a timer for "Timer", a database for "Data processing", and a file for "Webhook + API". Below this, it says "Get started quickly with a premade function" and "1) Choose a scenario:" followed by the three icons. It then says "2) Choose a language:" with "C#" and "JavaScript" buttons, and a "Create this function" button at the bottom.

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

GLOBAL INTEGRATION BOOTCAMP

Microsoft Azure RG_GlobalIntegration > Function app

Search my functions

+ New Function

Choose a template

Language: All Scenario: Core

BlobTrigger-CSharp	BlobTrigger-JavaScript	EventHubTrigger-CSharp	EventHubTrigger-JavaScript	GenericWebHook-CSharp	GenericWebHook-JavaScript	GitHubWebHook-CSharp
A C# function that will be run whenever a blob is added to a specified container	A JavaScript function that will be run whenever a blob is added to a specified container	A C# function that will be run whenever an event hub receives a new event	A JavaScript function that will be run whenever an event hub receives a new event	A C# function that will be run whenever it receives a webhook request	A JavaScript function that will be run whenever it receives a webhook request	A C# function that will be run whenever it receives a GitHub webhook request
GitHubWebHook-JavaScript	HttpTrigger-CSharp	HttpTrigger-JavaScript	ManualTrigger-CSharp	ManualTrigger-JavaScript	QueueTrigger-CSharp	QueueTrigger-JavaScript
A JavaScript function that will be run whenever it receives a GitHub webhook request	A C# function that will be run whenever it receives an HTTP request	A JavaScript function that will be run whenever it receives an HTTP request	A C# function that is triggered manually via the portal "Run" button	A JavaScript function that is triggered manually via the portal "Run" button	A C# function that will be run whenever a message is added to a specified Azure Queue Storage	A JavaScript function that will be run whenever a message is added to a specified Azure Queue Storage

Name your function
CheckDiscountCustomer

Create

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

Microsoft Azure RG_GlobalIntegration > Function app

Search my functions

+ New Function

CheckDiscountCustomer

Code (run.csx) Save Run

Function Url: Loading..

```

1 #r "Newtonsoft.Json"
2
3 using System;
4 using System.Net;
5 using Newtonsoft.Json;
6
7 public static async Task<object> Run(HttpRequestMessage req, TraceWriter log)
8 {
9     log.Info("Webhook was triggered!");
10
11    string jsonContent = await req.Content.ReadAsStringAsync();
12    dynamic data = JsonConvert.DeserializeObject(jsonContent);
13
14    if (data.first == null || data.last == null) {
15        return req.CreateResponse(HttpStatusCode.BadRequest, new {
16            error = "Please pass first/last properties in the input object"
17        });
18    }
19
20    return req.CreateResponse(HttpStatusCode.OK, new {
21        greeting = $"Hello {data.first} {data.last}!"
22    });
23 }
24

```

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.

GLOBAL INTEGRATION BOOTCAMP

The screenshot shows the Azure portal interface. On the left, there's a list of resources under 'Essentials', including 'Subscription name (change)', 'Azure Free Trial', 'Deployments (4 Succeeded)', and a list of 6 items. The 'globalintegration' storage account is selected. On the right, the 'Access keys' blade is open for this storage account. It contains a search bar, a navigation menu with links like 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', 'SETTINGS' (which is currently selected), 'Configuration', 'Shared access signature', 'Properties', 'Locks', and 'Automation script'. Under 'SETTINGS', the 'Access keys' section is highlighted. It shows two keys: 'key1' with the value 'bV7l+Ulw7UcZLivvWWWh7APTQlJaGVb/9fTyP4BpuvoHNs/' and 'key2' with the value 'IM23KTYgFl6uz1cl6PWQz8cRYN79ba79AAchHoR0rnfq9/LmJgH'. Each key has a copy icon next to it.

```
#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,

```

GLOBAL INTEGRATION BOOTCAMP

```

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

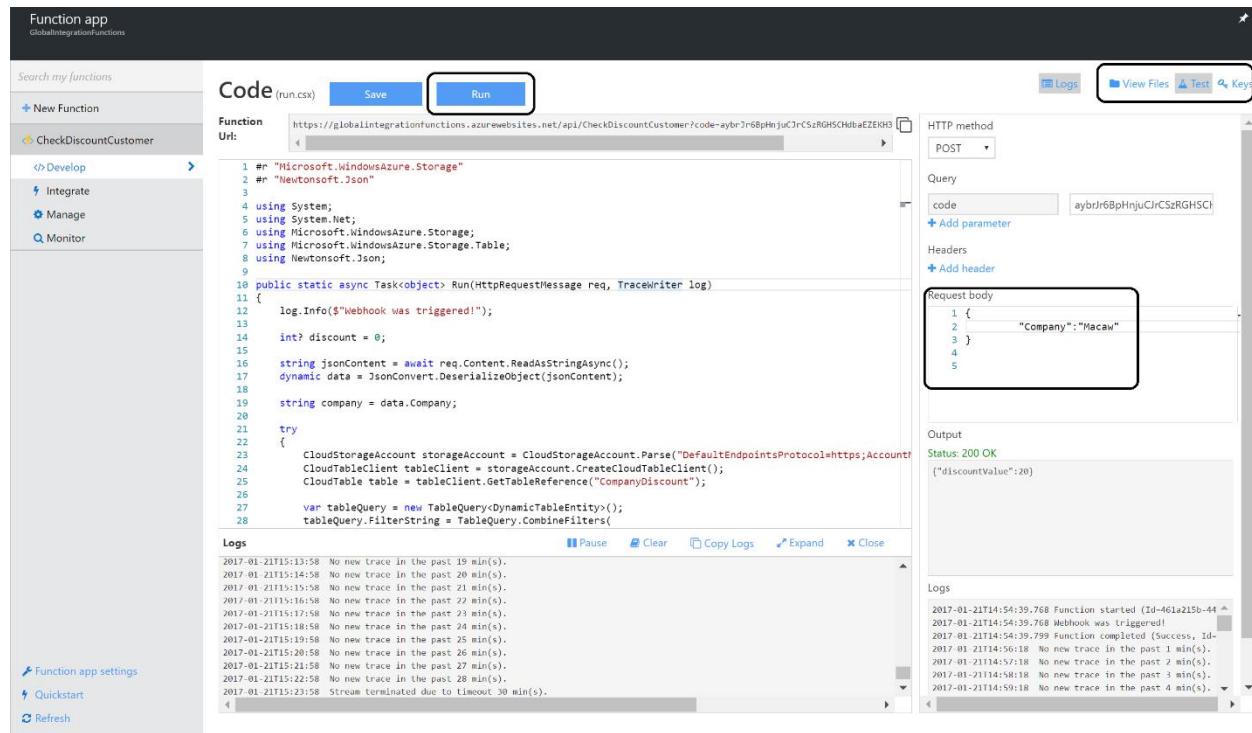
    // Loop through the results, displaying information about the entity.
    foreach (DynamicTableEntity entity in table.ExecuteQuery(tableQuery))
    {
        var item = entity.Properties;
        discount = item["Discount"].Int32Value;
    }
}
catch (Exception ex)
{
    log.Info($"Webhook exception :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

GLOBAL INTEGRATION BOOTCAMP

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.

The screenshot shows the Microsoft Azure portal interface. On the left, there is a sidebar with various icons. The main area has a search bar at the top with the text 'Search resources'. Below it, there are two tabs: 'Everything' and 'Logic App'. The 'Logic App' tab is selected, showing a search result for 'Logic App' with a count of 10 results. The results table has columns for NAME, PUBLISHER, and CATEGORY. The first result is 'Logic App' by Microsoft, categorized under 'Web + Mobile'. To the right of the search results, there is a detailed view of the 'Logic App' service, including its description, usage statistics, and a preview of the Logic App Designer interface. The designer interface shows a flow with several triggers and actions connected by arrows. At the bottom of the designer window, there is a blue 'Create' button.

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).

GLOBAL INTEGRATION BOOTCAMP

The screenshot shows the 'Create logic app' wizard in the Microsoft Azure portal. The top navigation bar indicates the path: Microsoft Azure > RG_GlobalIntegration > Everything > Logic App > Create logic app. On the left, a sidebar lists various Azure services with icons: Functions, Logic Apps, API Management, Container Registry, App Service, Storage, Key Vault, and others. The main form is titled 'Create logic app' and contains the following fields:

- Name:** ProcessOrders (marked with a red asterisk)
- Subscription:** Azure Free Trial
- Resource group:** RG_GlobalIntegration (radio buttons for 'Create new' and 'Use existing' are shown, with 'Use existing' selected)
- Location:** West Europe

A note at the bottom of the form says: "You can add triggers and actions to your Logic App after creation." At the bottom of the wizard, there is a checkbox for "Pin to dashboard" and a large blue "Create" button, which is highlighted with a black border.

- 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

GLOBAL INTEGRATION BOOTCAMP

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

The screenshot shows the Microsoft Azure portal interface for a Resource Group named 'RG_GlobalIntegration'. The left sidebar lists various service icons, and the main pane displays the 'Essentials' section for the resource group. Key details shown include:

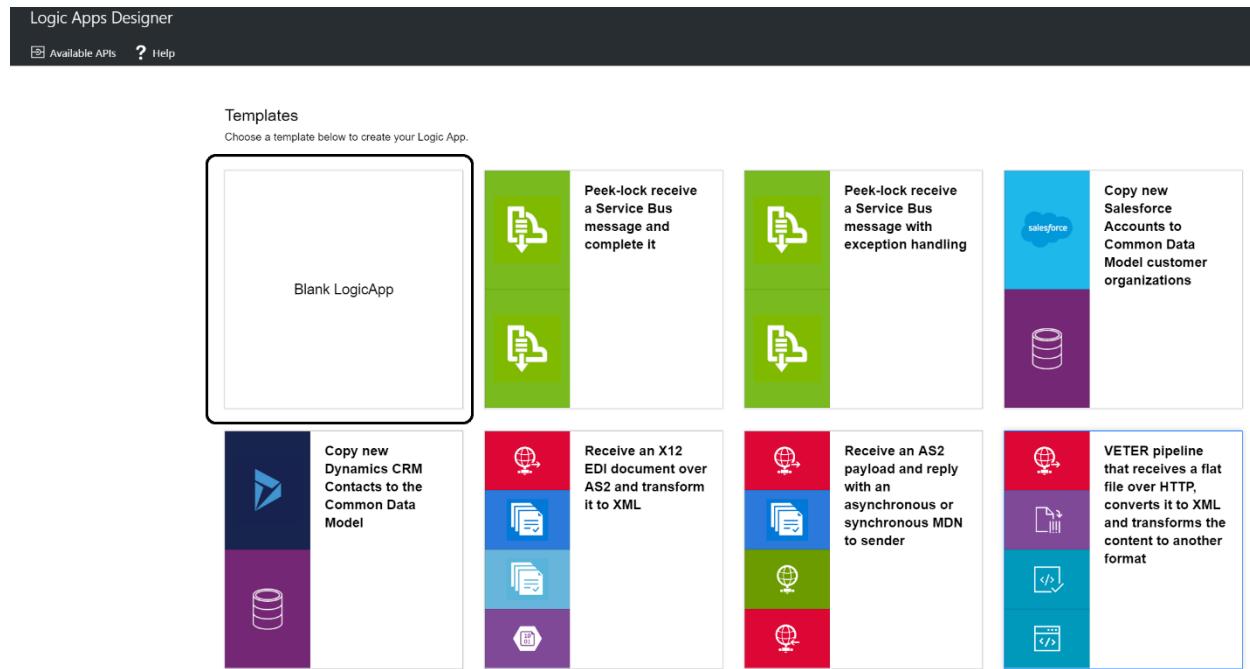
- Subscription name: Azure Free Trial
- Subscription ID: 0bf166ac-9aa8-4597-bb2a-a845afe01415
- Location: West Europe
- Deployments: No deployments
- Logic App: ProcessOrders (NAME, TYPE: Logic App, LOCATION: West Europe)

GLOBAL INTEGRATION BOOTCAMP

Building a Logic App Definition

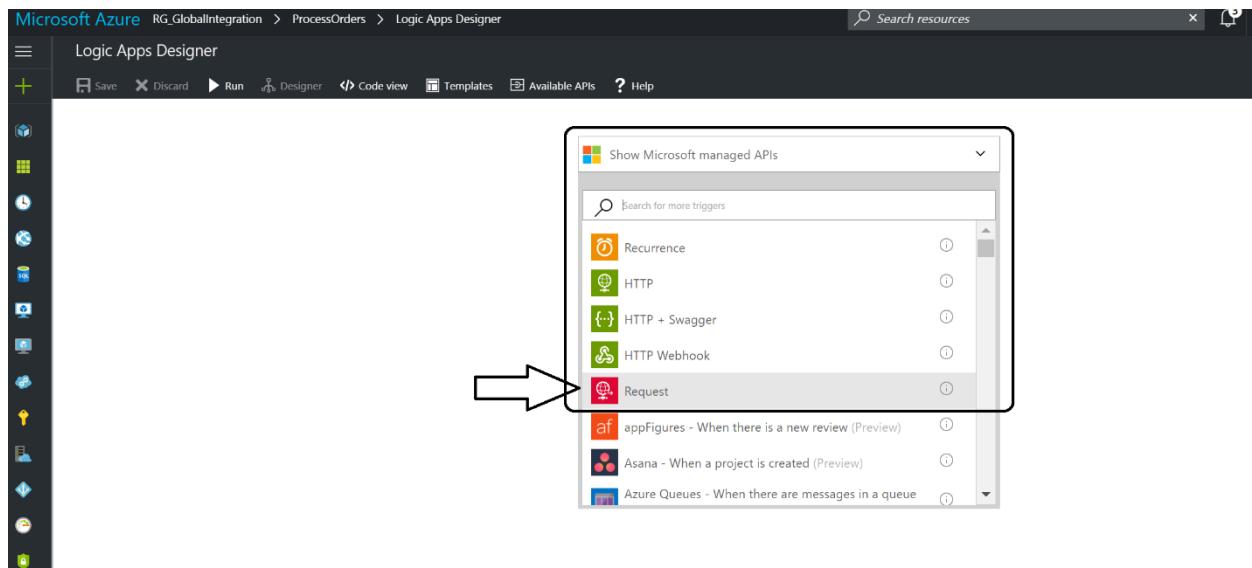
In the following steps, we will build our Logic App to support our solution.

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



- 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.

GLOBAL INTEGRATION BOOTCAMP



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",  
        "Status": "New"  
    }  
}
```

GLOBAL INTEGRATION BOOTCAMP

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

The screenshot shows the jsonschema.net website. On the left, under 'JSON', a sample JSON object is pasted:

```
{
  "Order": {
    "Customer": {
      "Company": "Motion10",
      "Email": "eldert.grootenboer@motion10.com",
      "CustomerNumber": "6f6d4907-23af-e611-80e5-5065f38a5ab1",
      "Address": {
        "Street": "Wilhelminakade 175",
        "City": "Rotterdam",
        "PostalCode": "3072AP",
        "Country": "Netherlands"
      }
    }
  }
}
```

A green box at the bottom says 'Well done! You provided valid JSON.' Below it are buttons for 'Generate Schema' and 'Reset'. To the right, under 'Schema', the generated JSON schema is shown:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "Order": {
      "type": "object",
      "properties": {
        "Customer": {
          "type": "object",
          "properties": {
            "Company": {
              "type": "string"
            },
            "Email": {
              "type": "string"
            },
            "CustomerNumber": {
              "type": "string"
            },
            "Address": {
              "type": "object",
              "properties": {
                "Street": {
                  "type": "string"
                },
                "City": {
                  "type": "string"
                },
                "PostalCode": {
                  "type": "string"
                },
                "Country": {
                  "type": "string"
                }
              }
            }
          }
        }
      }
    }
  }
}
```

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.

The screenshot shows the Microsoft Azure Logic Apps Designer interface. The top navigation bar includes 'Microsoft Azure', 'RG_GlobalIntegration', 'ProcessOrders', 'Logic Apps Designer', and a 'Search resources' bar. The main area is titled 'Logic Apps Designer' and shows a 'Save' button highlighted with a red box. Below it is a 'Request' card with the title 'HTTP POST to this URL' and a URL field containing 'https://prod-06.westeurope.logic.azure.com:443/workflows/99d6775b433942e7a...'. A large text input field labeled 'Request Body JSON Schema' contains the previously generated JSON schema:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "Order": {
      "type": "object",
      "properties": {
        "Customer": {
          "type": "object",
          "properties": {
            "Company": {
              "type": "string"
            }
          }
        }
      }
    }
  }
}
```

GLOBAL INTEGRATION BOOTCAMP

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

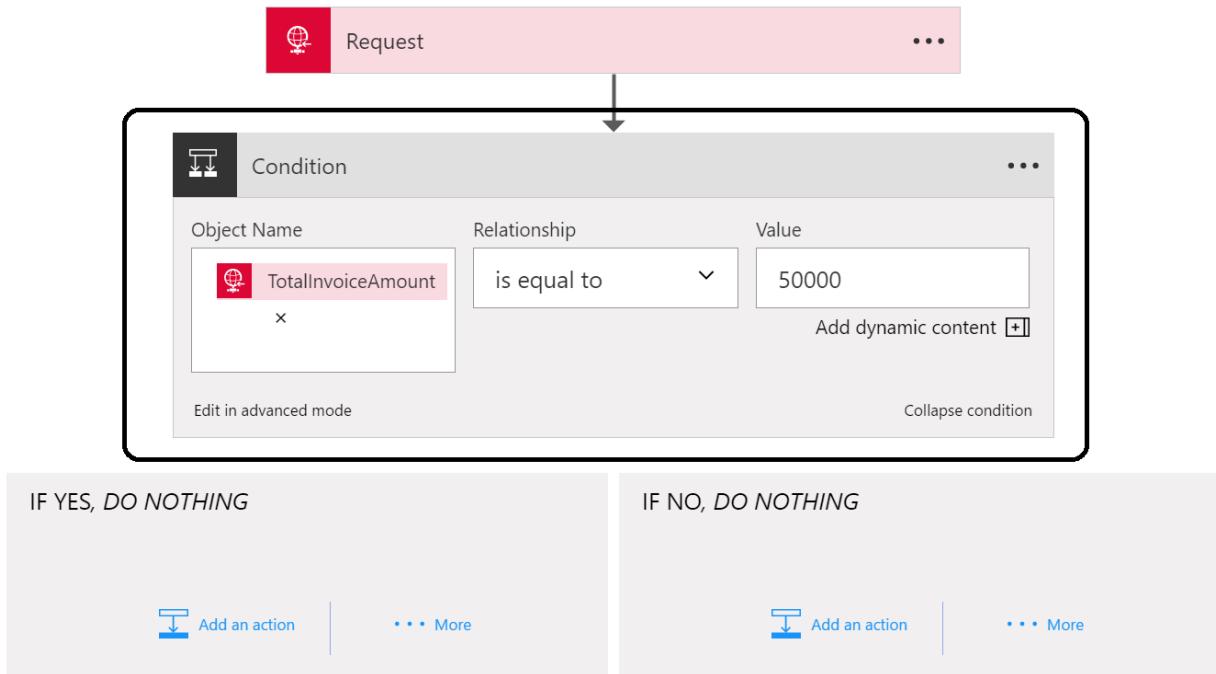
The screenshot shows the Microsoft Logic App designer interface. At the top, there's a red header bar with a globe icon, the word "Request", and three dots. Below the header, the title "HTTP POST to this URL" is displayed, followed by a text input field containing the URL <https://prod-06.westeurope.logic.azure.com:443/workflows/99d6775b433942e7a...>. To the right of the URL is a copy icon. Underneath the URL input, the section "Request Body JSON Schema" is visible, showing a partial JSON schema definition:

```
{  
  "$schema": "http://json-schema.org/draft-04/schema#",  
  "type": "object",  
  "properties": {  
    "Order": {  
      "type": "object",  
      "properties": {  
        "Customer": {  
          "type": "object",  
          "properties": {  
            "Company": {  
              "type": "string"  
            }  
          }  
        }  
      }  
    }  
  }  
}
```

At the bottom of the interface, there's a blue footer bar with three buttons: "Add an action" (with a downward arrow icon), "Add a condition" (with a double downward arrow icon), and "More" (with three dots). Above these buttons is a white button labeled "+ New step".

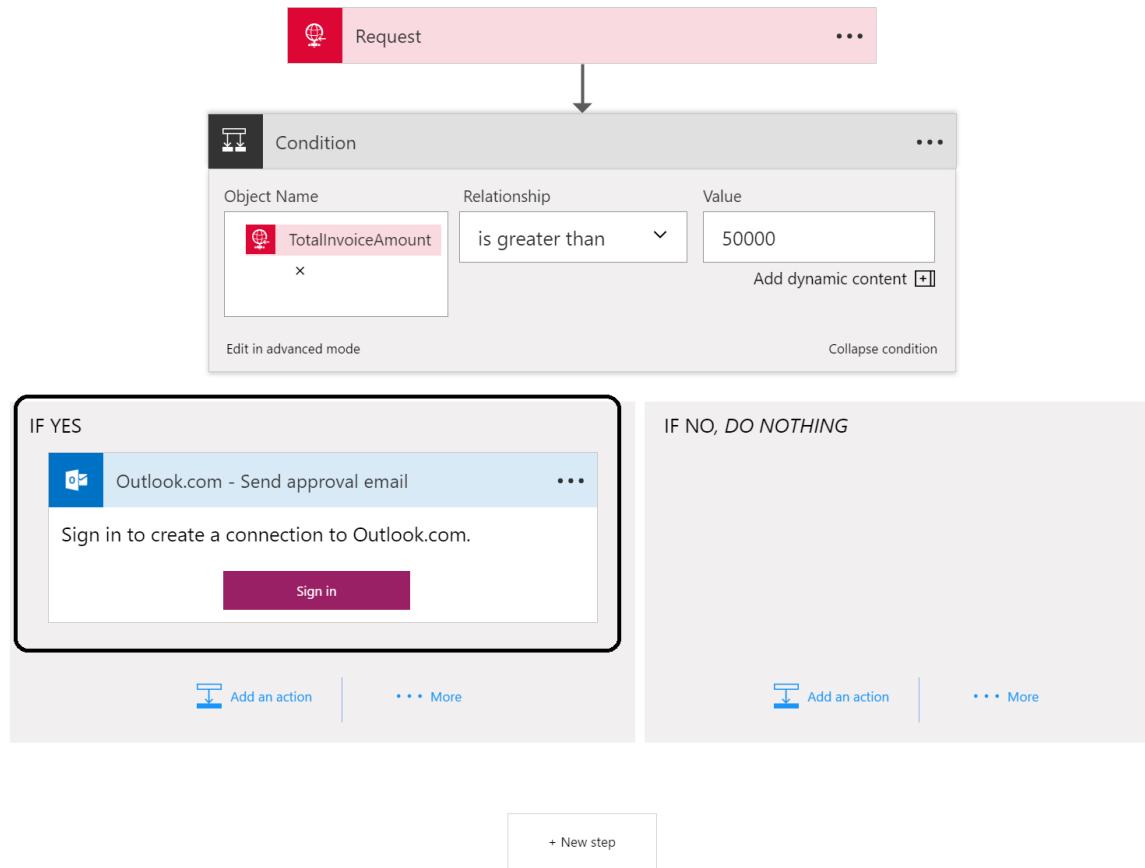
6. The **condition** will be to verify if the amount is higher than **50000**.

GLOBAL INTEGRATION BOOTCAMP



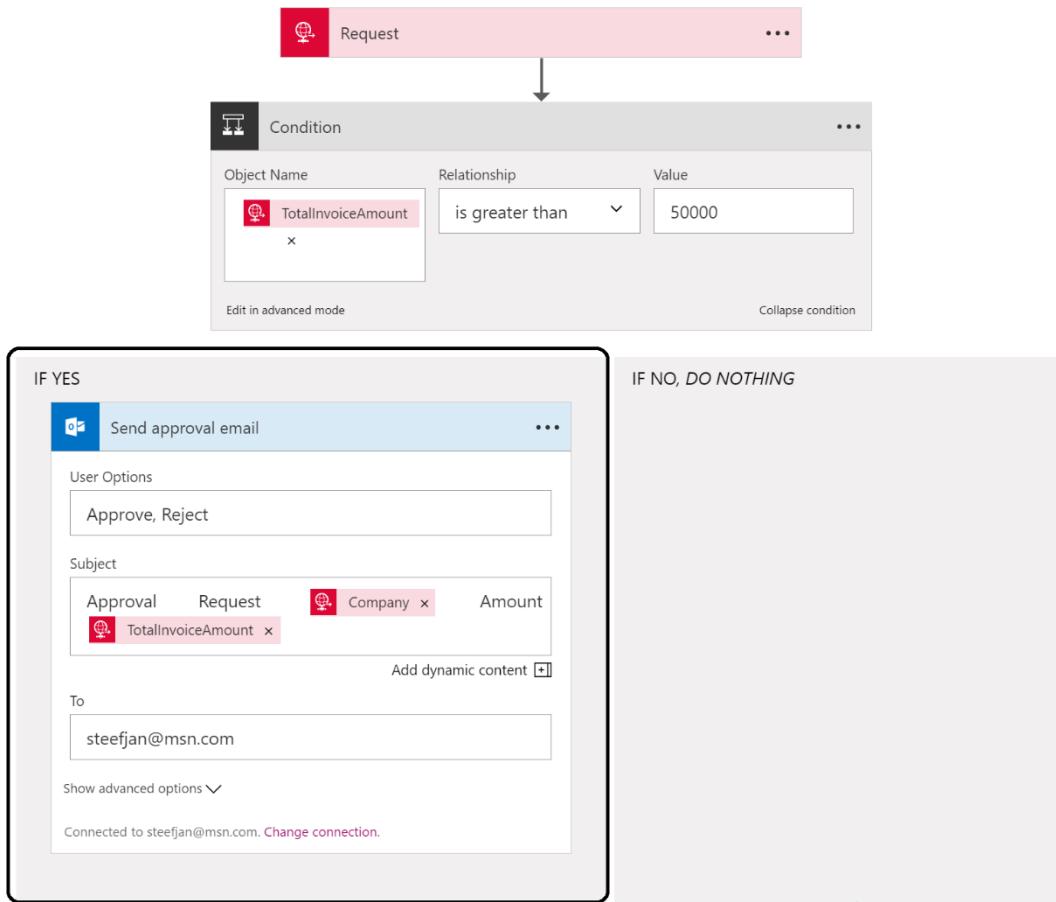
7. Add an **Outlook (Send Approval Mail)** **action** to the left branch.

GLOBAL INTEGRATION BOOTCAMP



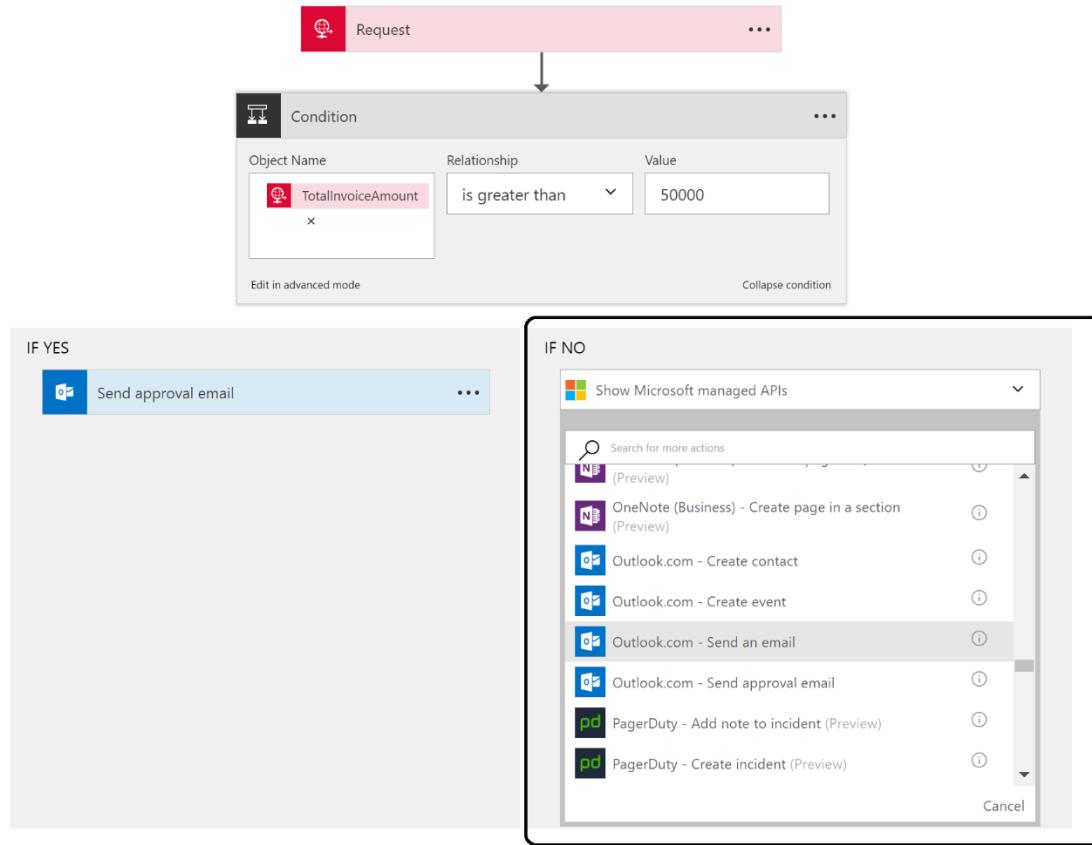
8. **Login** to your **Outlook.com** account. If you do not have one you can create one easily.

GLOBAL INTEGRATION BOOTCAMP

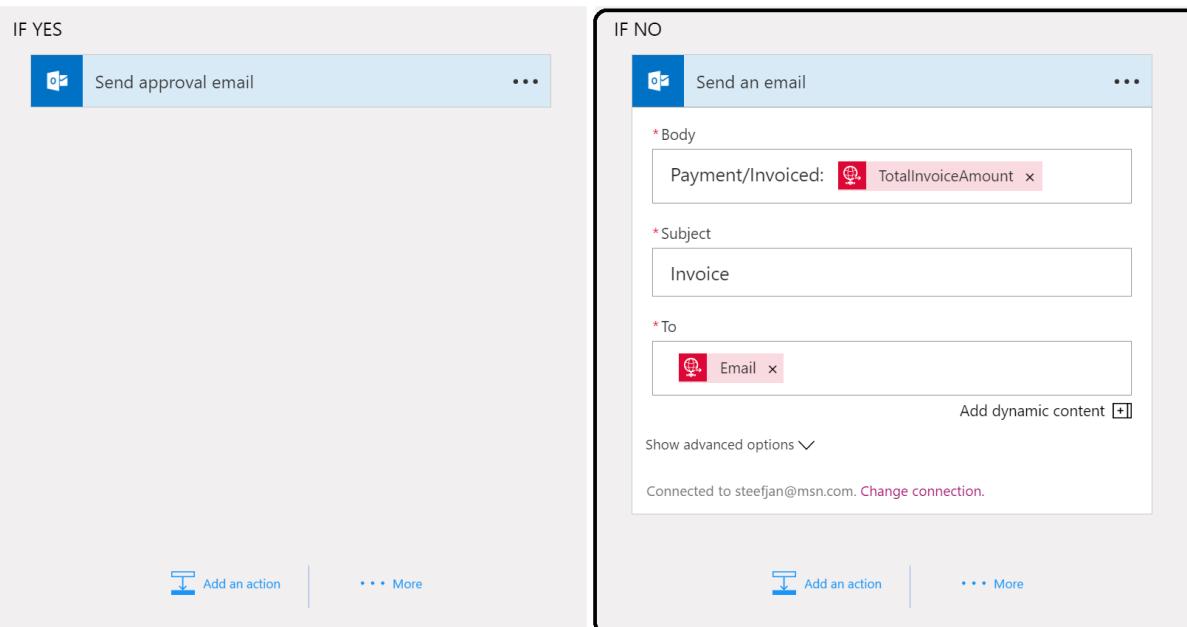


9. After the connection to your outlook has been established you can specify the subject like above and add your email address in the **To** field.

GLOBAL INTEGRATION BOOTCAMP

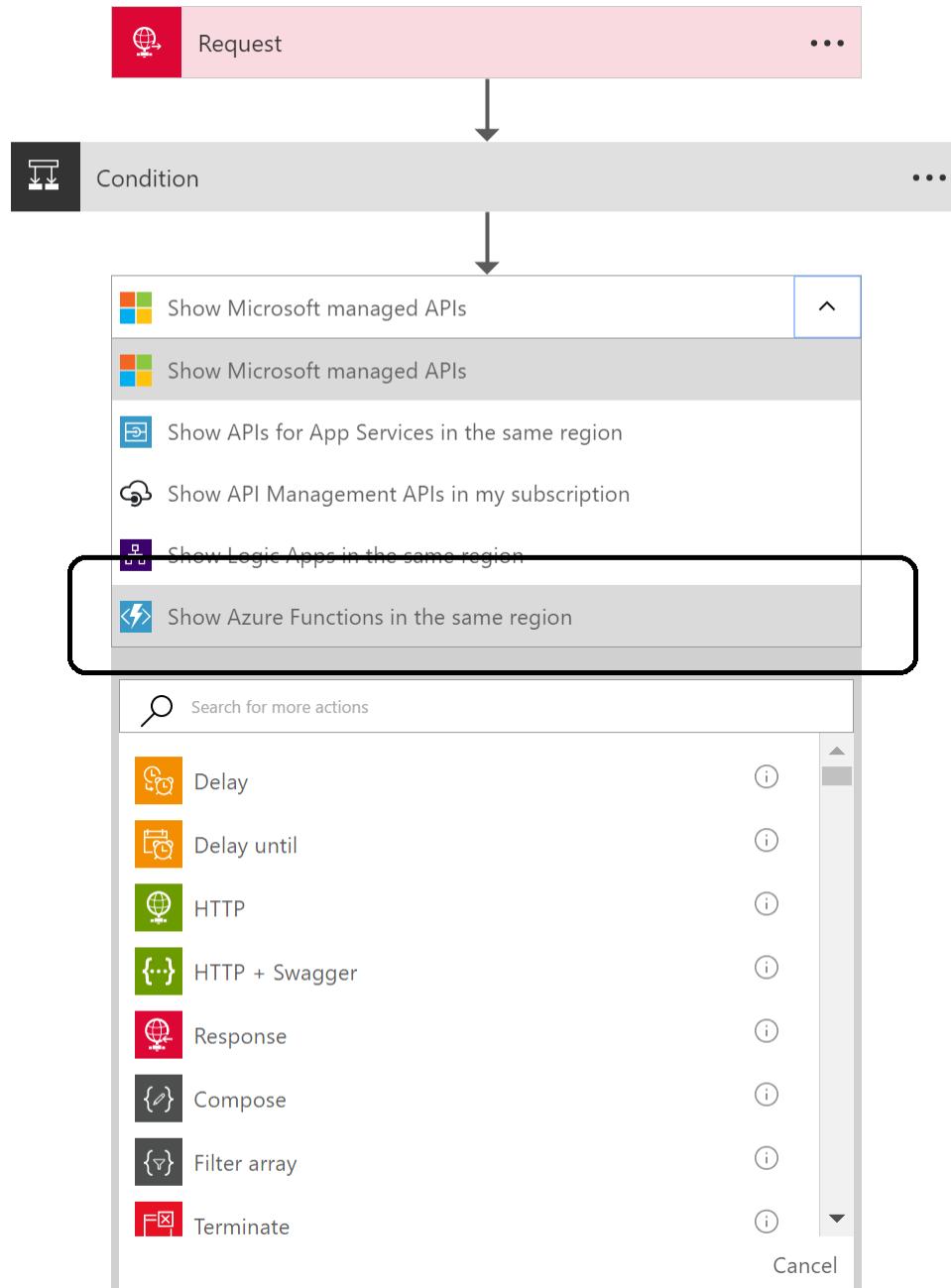


10. In the right branch (IF NO) you can add an Outlook.com Send an email action using your earlier established connection. You do not need to login again.



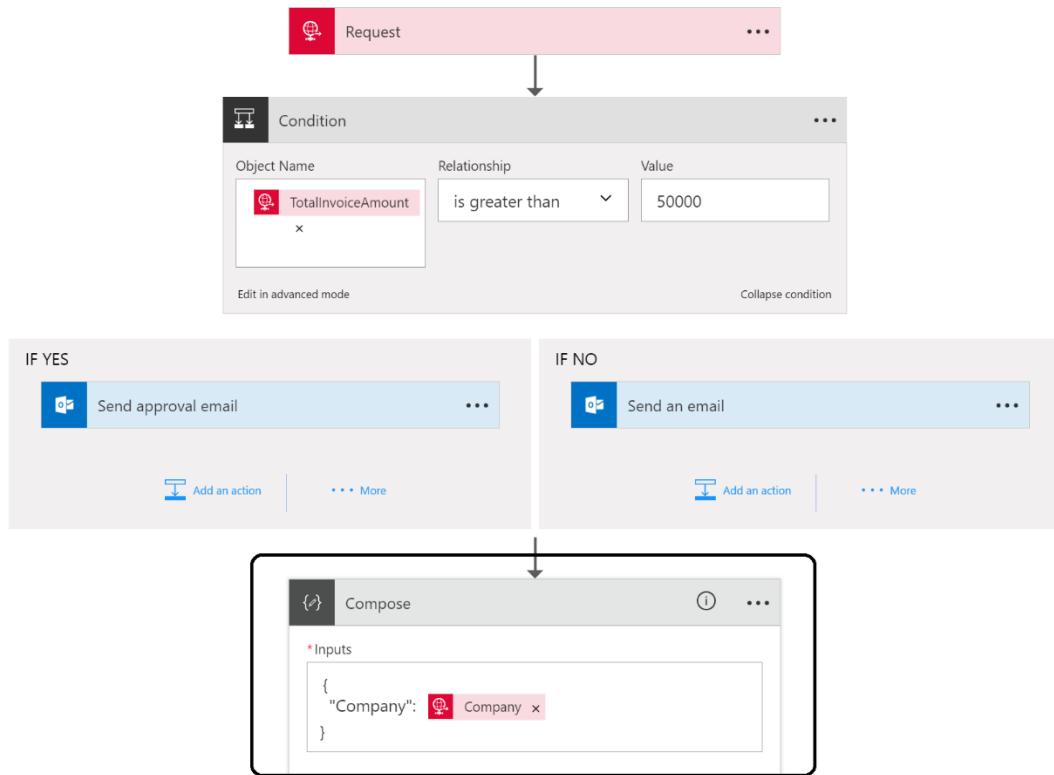
GLOBAL INTEGRATION BOOTCAMP

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.

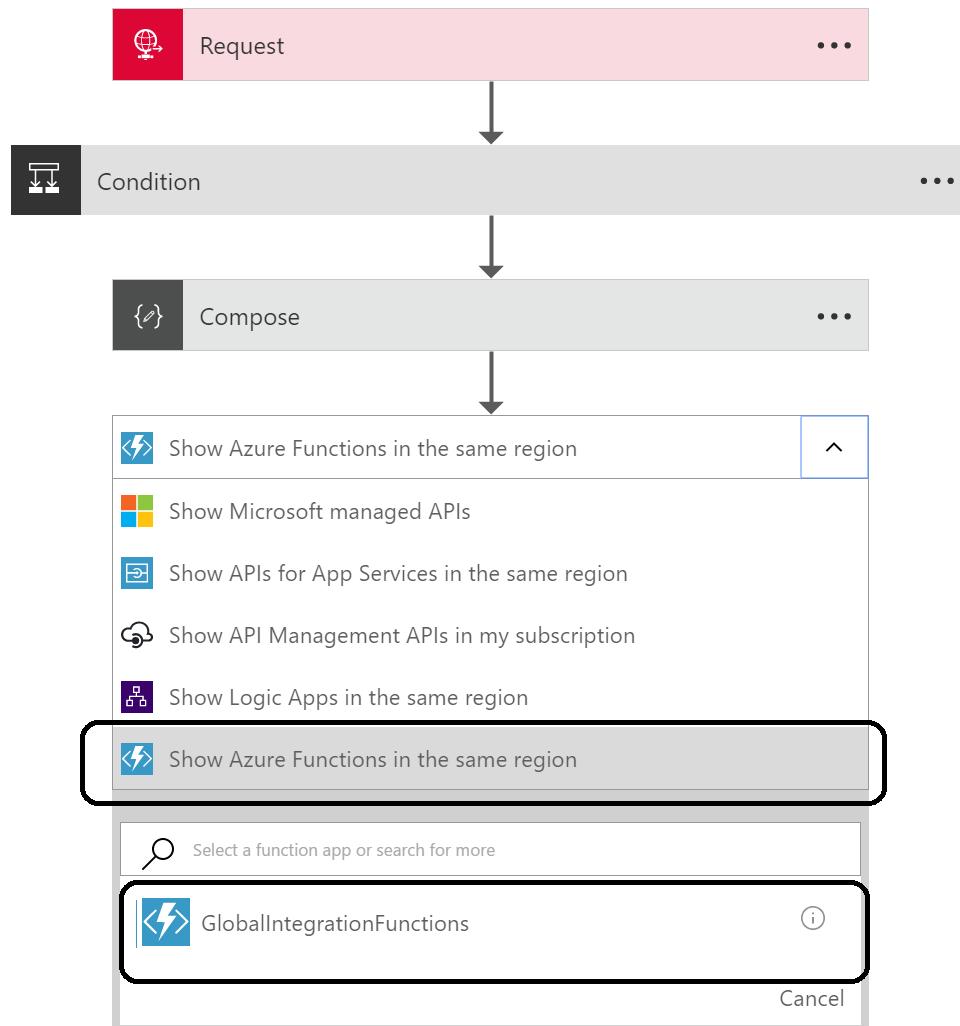


14. Add an **Compose** action below the conditions.

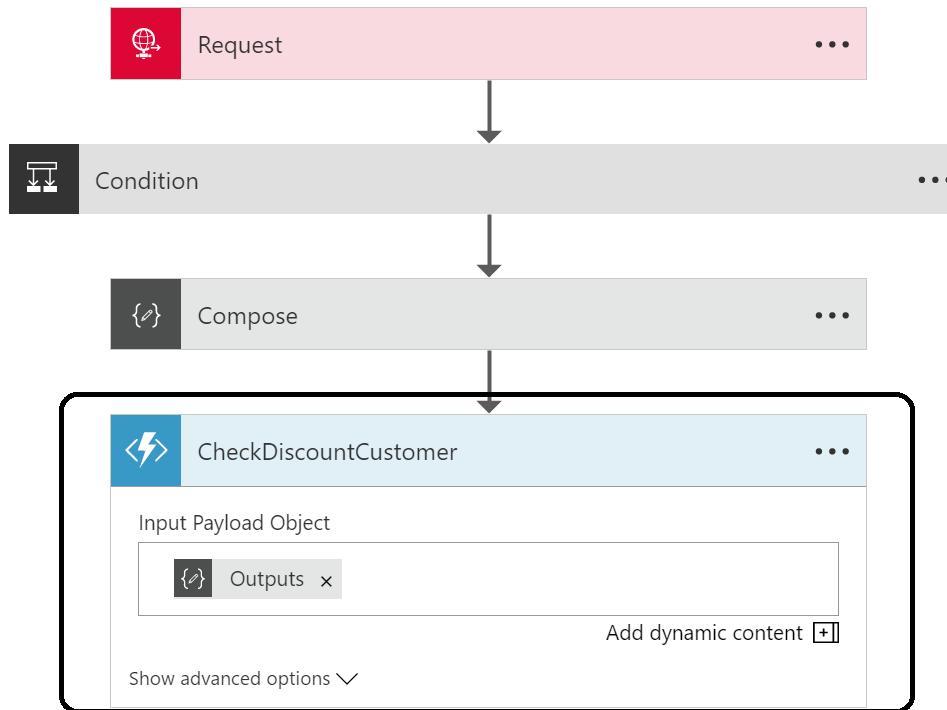
GLOBAL INTEGRATION BOOTCAMP



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.

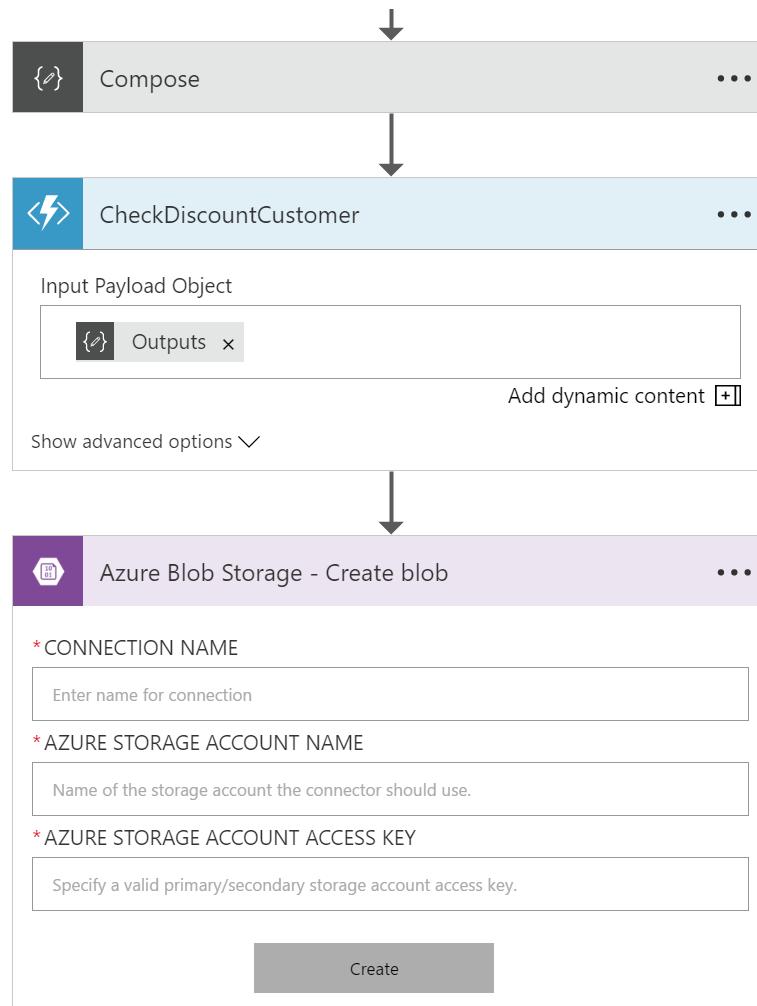


18. Select your custom created function.
19. Drag the **Output** from the **Compose** in the **Input Payload Object**.



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

GLOBAL INTEGRATION BOOTCAMP



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



GLOBAL INTEGRATION BOOTCAMP

globalintegration - Access keys
Storage account

Search (Ctrl+ /)

- Overview
- Activity log
- Access control (IAM)
- Tags
- Diagnose and solve problems

SETTINGS

- Access keys** (highlighted with a red box)
- Configuration

Storage account name: globalintegration

NAME	KEY
key1	bV7I+Ull8wZUcZL.../9JFTyP4BPu...Hns/
key2	IM23kTYgFl6uz1cl6PWQz8cRYN79ba79AA...Hns/

Azure Blob Storage - Create blob

* CONNECTION NAME
Invoices

* AZURE STORAGE ACCOUNT NAME
globalintegration

* AZURE STORAGE ACCOUNT ACCESS KEY
.....

Create

22. Click **Create**.

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

GLOBAL INTEGRATION BOOTCAMP

The screenshot shows the Microsoft Logic App designer interface. At the top, there is a purple header bar with a downward arrow icon. Below the header, the main area is titled "Create blob". On the left, there is a "Folder path" input field containing "/invoices" with a "Remove" button. In the center, there is a "Blob name" input field containing "OrderedDateTime" and "Company" separated by a space, with a "Remove" button. To the right of the blob name input is a "Add dynamic content" button. Below the blob name input is a "Blob content" input field containing "Body" and "Body" separated by a space, with a "Remove" button. At the bottom of the main area, a message says "Connected to Invoices. [Change connection.](#)".

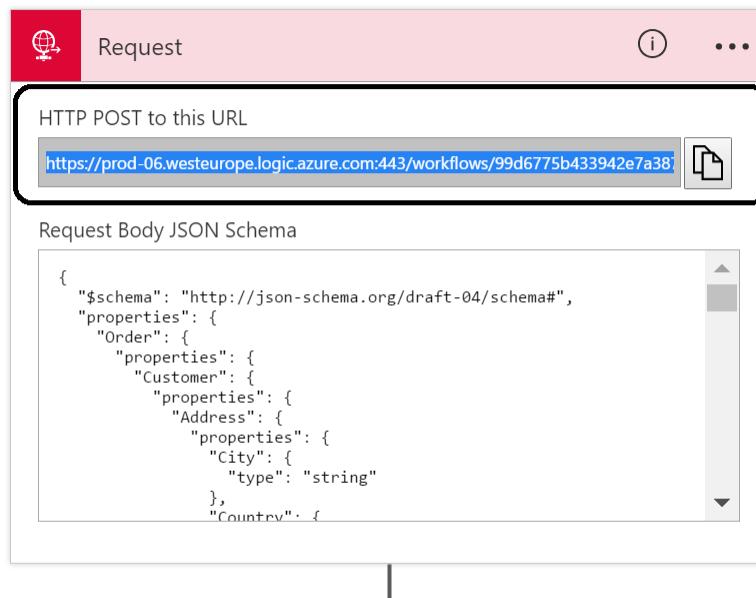
24. **Save the Logic App Definition.**

GLOBAL INTEGRATION BOOTCAMP

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": {  
                "Name": "Dell XPS 15",  
                "Quantity": 1,  
                "UnitPrice": 1200  
            }  
        }  
    }  
}
```

GLOBAL INTEGRATION BOOTCAMP

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

The screenshot shows the Postman application interface. At the top, the URL is https://prod-06.weste... and the environment is set to 'No Environment'. Below the URL bar, there are tabs for 'POST' (selected), 'Headers (1)', 'Body' (selected), 'Pre-request Script', and 'Tests'. The 'Body' tab is currently active, showing the JSON payload for the POST request. The payload is a multi-line JSON object representing an order with products and their details. The JSON is formatted with line numbers from 1 to 21.

```
1 {  
2     "Order":{  
3         "Customer":{  
4             "Company":"Macaw",  
5             "Email":"steefjan@msn.com",  
6             "CustomerNumber":"6f6d4907-23af-e611-80e5-5065f38a5a01",  
7             "Address":{  
8                 "Street":"Beechavenue 140",  
9                 "City":"Schiphol-Rijk",  
10                "PostalCode":"1119 PR",  
11                "Country":"Netherlands"  
12            }  
13        },  
14        "Products":{  
15            "Product": [  
16                {  
17                    "ProductNumber":1000,  
18                    "Amount":1,  
19                    "Price":123.45  
20                },  
21                {  
22                    "ProductNumber":2000,  
23                    "Amount":5,  
24                    "Price":456.78  
25                }  
26            ]  
27        }  
28    }  
29}
```

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.

GLOBAL INTEGRATION BOOTCAMP

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**.

 **GLOBAL INTEGRATION BOOTCAMP**

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.

 **GLOBAL INTEGRATION BOOTCAMP**

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

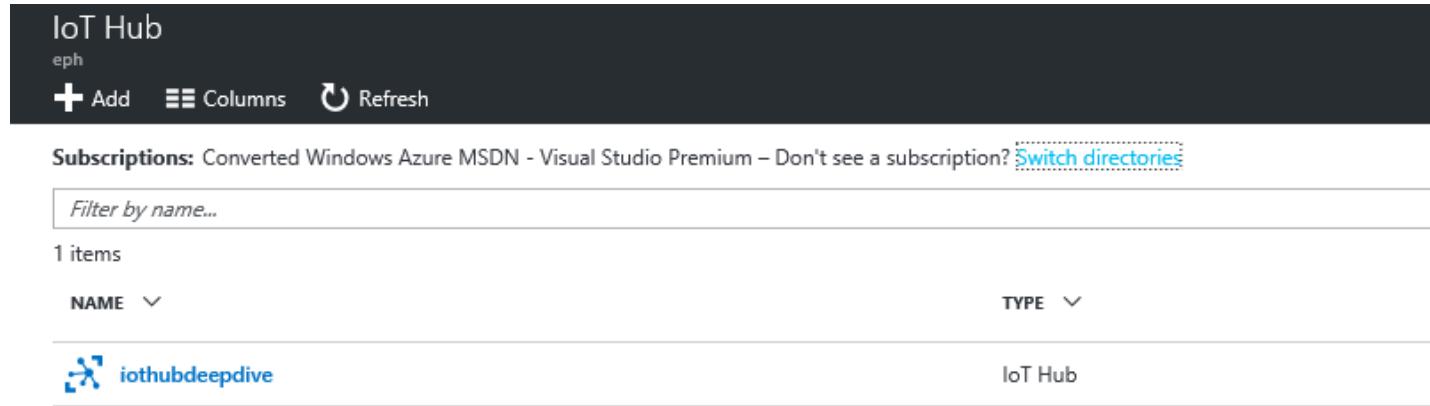
To build the solution in this lab you have to follow the steps described in this section. From a high level view the steps are:

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

GLOBAL INTEGRATION BOOTCAMP

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.



The screenshot shows the 'IoT Hub' blade in the Azure portal. At the top, there's a header with the title 'IoT Hub' and a subtitle 'eph'. Below the header are three buttons: '+ Add', 'Columns', and 'Refresh'. A message 'Subscriptions: Converted Windows Azure MSDN - Visual Studio Premium – Don't see a subscription? [Switch directories](#)' is displayed. A search bar with the placeholder 'Filter by name...' is present. Below the search bar, it says '1 items'. There are two columns: 'NAME' and 'TYPE'. A single item is listed: 'iothubdeepdive' (with its icon) under 'NAME' and 'IoT Hub' under 'TYPE'.

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.

GLOBAL INTEGRATION BOOTCAMP

IoT hub - □ ×

Microsoft

* Name
GlobalIntegrationBootcamp ✓

* Pricing and scale tier >
S1 - Standard

* IoT Hub units ⓘ
1

* Device-to-cloud partitions ⓘ
4 partitions

* Subscription
Converted Windows Azure MSDN - Visual ▼

* Resource group ⓘ
 Create new Use existing
GlobalIntegrationBootcamp ▼

* Location
North Europe ▼

GLOBAL INTEGRATION BOOTCAMP

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.

The screenshot shows the 'NoSQL (DocumentDB)' blade in the Azure portal. At the top, there are buttons for 'Add' and 'Refresh'. Below that, a section titled 'Subscriptions:' shows 'Converted Windows Azure MSDN - Visual Studio Premium' with a link to 'Switch directories'. A search bar labeled 'Filter by name...' is present. The main area displays '0 items' and a table header with columns 'NAME' and 'STATUS'. A message at the bottom states 'No NoSQL (DocumentDB) to display'.

For this lab, we will use a DocumentDB backend.

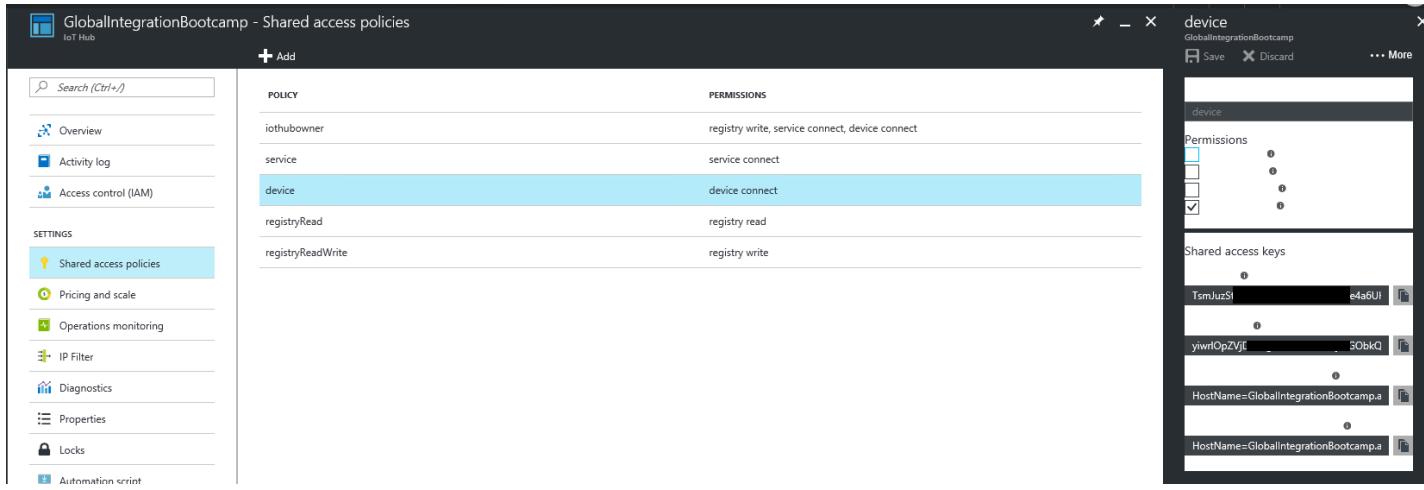
The screenshot shows the 'New account' blade for creating a DocumentDB instance. It includes fields for 'ID' (set to 'globalintegrationbootcamp'), 'Subscription' (set to 'Converted Windows Azure MSDN - Visual'), 'Resource Group' (radio button selected for 'Use existing', value 'GlobalIntegrationBootcamp'), and 'Location' (set to 'North Europe').

★ ID	globalintegrationbootcamp
NoSQL API	DocumentDB
★ Subscription	Converted Windows Azure MSDN - Visual
★ Resource Group	<input checked="" type="radio"/> Use existing Create new
Location	North Europe

GLOBAL INTEGRATION BOOTCAMP

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.



The screenshot shows the Azure IoT Hub Shared access policies page. On the left, there's a sidebar with options like Overview, Activity log, Access control (IAM), Pricing and scale, Operations monitoring, IP Filter, Diagnostics, Properties, Locks, and Automation script. The 'Shared access policies' option is selected and highlighted in blue. The main area displays a table of shared access policies:

POLICY	PERMISSIONS
iothubowner	registry write, service connect, device connect
service	service connect
device	device connect
registryRead	registry read
registryReadWrite	registry write

To the right of the table, there's a panel titled 'device' with sections for 'Permissions' (showing checkboxes for registry write, service connect, and device connect, with 'device connect' checked) and 'Shared access keys'. It lists two keys: 'TsmJuzSl' (e4a6UJ) and 'yiwrlOpZVjl' (5ObkQ). Below each key are fields for 'HostName' containing 'GlobalIntegrationBootcamp.a'.

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

GLOBAL INTEGRATION BOOTCAMP

The screenshot shows the 'Device Explorer Twin' application window. At the top, there is a navigation bar with tabs: Configuration, Management, Data, Messages To Device, and Call Method on Device. The 'Management' tab is currently selected.

Connection Information

IoT Hub Connection String:

```
HostName=GlobalIntegrationBootcamp.azure-devices.net;SharedAccessKeyName=iothubowner;SharedAccessKey=si3F2hxSW8suZ4a1hbntatjRDsKayyTeEkdVUYXRirY=
```

Protocol Gateway HostName:

Update

Shared Access Signature

Key Name: iothubowner

Key Value: si3F2hxSW8suZ4a1hbntatjRDsKayyTeEkdVUYXRirY=

Target: GlobalIntegrationBootcamp.azure-devices.net

TTL (Days): 365

Generate SAS

On the management tab, create a new device.

GLOBAL INTEGRATION BOOTCAMP

The screenshot shows the 'Device Explorer Twin' application window. At the top, there is a menu bar with tabs: Configuration, Management, Data, Messages To Device, and Call Method on Device. Below the menu, there is a toolbar with buttons for Actions: Create, Refresh, Update, Delete, SAS Token..., and Twin Props. Under the Actions section, there is a heading 'Devices' and a message 'Total: 0'. Below this, there is a table with the following columns: Id, PrimaryKey, SecondaryKey, PrimaryThumbprint, SecondaryThumbprint, ConnectionString, and ConnectionString2. A single row is present in the table, starting with an asterisk (*) in the Id column.

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.

 **GLOBAL INTEGRATION BOOTCAMP**

Create Device

Device Authentication

Security Keys X509

Device ID:

Primary Key: [REDACTED]

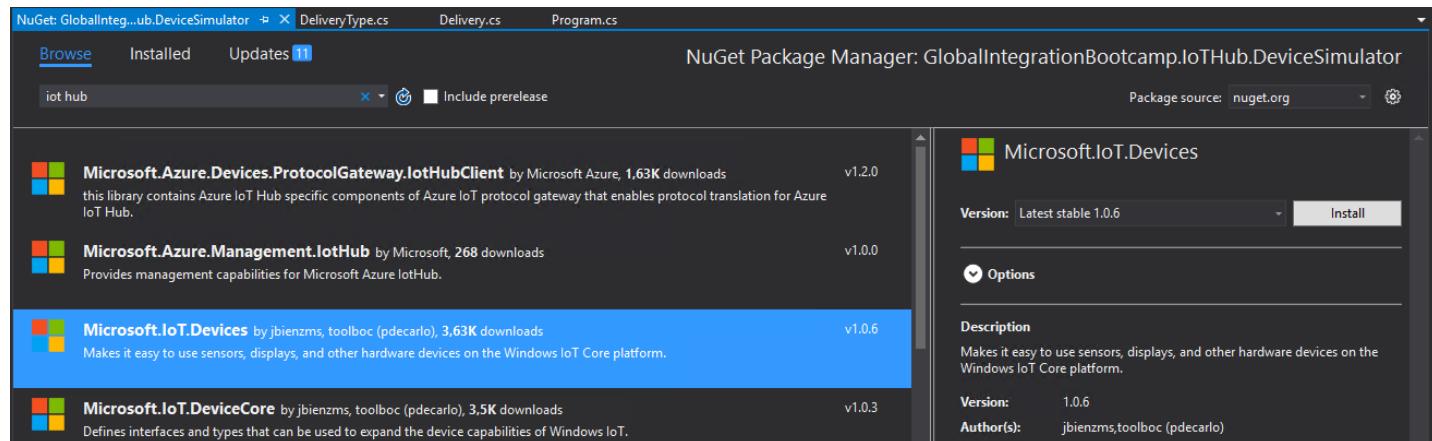
Secondary Key: [REDACTED]

Auto Generate ID Auto Generate Keys

GLOBAL INTEGRATION BOOTCAMP

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>
    }
}
```

 **GLOBAL INTEGRATION BOOTCAMP**

```
    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",
"OjR3mUq9LtBrJ8zdz0OkxxxxxX8HJaJUQLlnyz+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);
}
}
}
```

GLOBAL INTEGRATION BOOTCAMP

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.

Stream Analytics jobs
eph

+ Add Columns Refresh

Subscriptions: Converted Windows Azure MSDN - Visual Studio Premium – Don't see a subscription? [Switch directories](#)

Filter by name...

3 items

NAME	STATUS
New Stream Analytics J...	

* Job name
GlobalIntegrationBootcamp ✓

* Subscription
Converted Windows Azure MSDN - Visual

* Resource group ⓘ
 Create new Use existing
GlobalIntegrationBootcamp

* Location
North Europe

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.

GLOBAL INTEGRATION BOOTCAMP

New input - □ ×

* Input alias
IoTHub ✓

* Source Type ⓘ
Data stream

* Source ⓘ
IoT hub

* Subscription
Use IoT hub from current subscription

* IoT hub
GlobalIntegrationBootcamp

* Endpoint ⓘ
Messaging

* Shared access policy name
iothubowner

Shared access policy key

* Consumer group
\$Default

* Event serialization format ⓘ
JSON

Encoding ⓘ
UTF-8

Add outputs

We will now add our two outputs.

 **GLOBAL INTEGRATION BOOTCAMP****Outputs**

GlobalIntegrationBootcamp

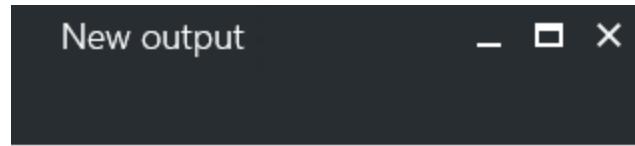
+ Add**- □ ×****NAME****SINK**

Empty

PowerBI

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

GLOBAL INTEGRATION BOOTCAMP



* Output alias

PowerBI ✓

* Sink ⓘ

Power BI ▾

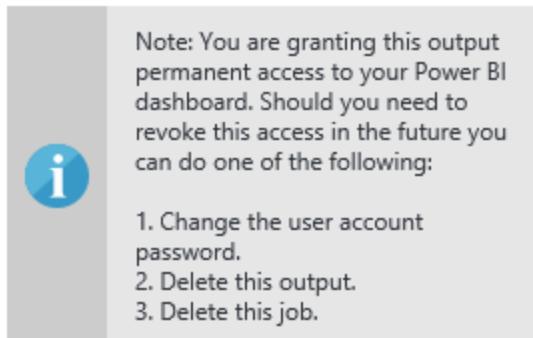
Authorize Connection

You'll need to authorize with Power BI to configure your output settings.

[Authorize](#)

Don't have a Microsoft Power BI account yet?

[Sign Up](#)

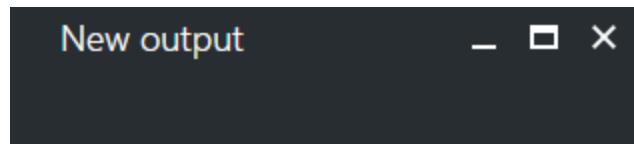


Note: You are granting this output permanent access to your Power BI dashboard. Should you need to revoke this access in the future you can do one of the following:

1. Change the user account password.
2. Delete this output.
3. Delete this job.

Create a dataset and table in your preferred workspace.

GLOBAL INTEGRATION BOOTCAMP



* Output alias

PowerBI ✓

* Sink ⓘ

Power BI ▾

Group Workspace

My Workspace ▾

* Dataset Name

GlobalIntegrationBootcamp ✓



If the dataset or table already exists in your Microsoft Power BI subscription, it will be overwritten.

* Table Name

GlobalIntegrationBootcamp ✓

Currently authorized as Eldert Grootenboer
(eldert.grootenboer@motion10.com)

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.

GLOBAL INTEGRATION BOOTCAMP

* Output alias

 ✓

* Sink ⓘ

 ▼

* Subscription

 ▼

* Account id ⓘ

 ▼

* Database

 ▼

* Database

* Collection name pattern ⓘ

Document id ⓘ

Set query

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

GLOBAL INTEGRATION BOOTCAMP

The screenshot shows the Azure Stream Analytics job configuration interface. At the top, there's a navigation bar with 'GlobalIntegrationBootcamp' and 'Query'. Below it are buttons for 'Save', 'Discard', and 'Test'. The main area has two sections: 'Inputs (1)' and 'Outputs (2)'. Under 'Inputs (1)', there's one item: 'IoTHub'. Under 'Outputs (2)', there are two items: 'PowerBI' and 'DocumentDB'. To the right of these sections is a large text area containing a Stream Analytics query:

```
1 SELECT  
2 *  
3 INTO [PowerBI]  
4 FROM [IoTHub]  
5  
6 SELECT  
7 *  
8 INTO [DocumentDB]  
9 FROM [IoTHub]
```

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

GLOBAL INTEGRATION BOOTCAMP

Settings Start Stop Delete

Created

Essentials ^

Resource group (change)	Send feedback
GlobalIntegrationBootcamp	UserVoice
Status	Created
Created	zondag 5 februari 2017 14:50:02
Location	Started
North Europe	-
Subscription name (change)	Last output
Converted Windows Azure MSDN - Visual...	-
Subscription ID	
1982dd27-5042-4968-802a-c9b9578e2543	

Job Topology

Inputs	Query	Outputs
1	<>	2
IoTHub		PowerBI
		DocumentDB

Start job

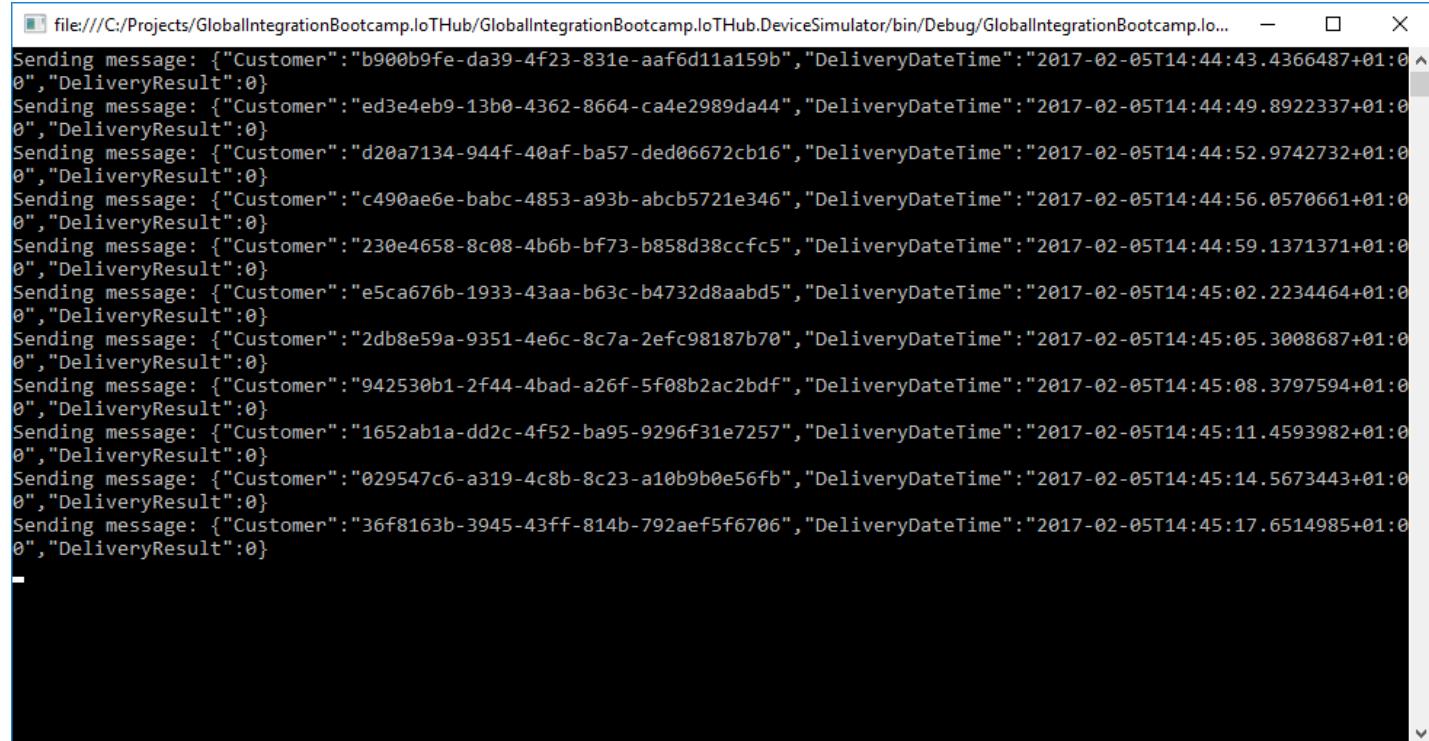
GlobalIntegrationBootcamp

Job output start time [i](#) Now Custom

GLOBAL INTEGRATION BOOTCAMP

Testing the lab

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



```
file:///C:/Projects/GlobalIntegrationBootcamp.IoTHub/GlobalIntegrationBootcamp.IoTHub.DeviceSimulator/bin/Debug/GlobalIntegrationBootcamp.lo... - □ X
Sending message: {"Customer":"b900b9fe-da39-4f23-831e-aaf6d11a159b", "DeliveryDateTime":"2017-02-05T14:44:43.4366487+01:00", "DeliveryResult":0}
Sending message: {"Customer":"ed3e4eb9-13b0-4362-8664-ca4e2989da44", "DeliveryDateTime":"2017-02-05T14:44:49.8922337+01:00", "DeliveryResult":0}
Sending message: {"Customer":"d20a7134-944f-40af-ba57-ded06672cb16", "DeliveryDateTime":"2017-02-05T14:44:52.9742732+01:00", "DeliveryResult":0}
Sending message: {"Customer":"c490ae6e-babc-4853-a93b-abcb5721e346", "DeliveryDateTime":"2017-02-05T14:44:56.0570661+01:00", "DeliveryResult":0}
Sending message: {"Customer":"230e4658-8c08-4b6b-bf73-b858d38ccfc5", "DeliveryDateTime":"2017-02-05T14:44:59.1371371+01:00", "DeliveryResult":0}
Sending message: {"Customer":"e5ca676b-1933-43aa-b63c-b4732d8ab5", "DeliveryDateTime":"2017-02-05T14:45:02.2234464+01:00", "DeliveryResult":0}
Sending message: {"Customer":"2db8e59a-9351-4e6c-8c7a-2efc98187b70", "DeliveryDateTime":"2017-02-05T14:45:05.3008687+01:00", "DeliveryResult":0}
Sending message: {"Customer":"942530b1-2f44-4bad-a26f-5f08b2ac2bdf", "DeliveryDateTime":"2017-02-05T14:45:08.3797594+01:00", "DeliveryResult":0}
Sending message: {"Customer":"1652ab1a-dd2c-4f52-ba95-9296f31e7257", "DeliveryDateTime":"2017-02-05T14:45:11.4593982+01:00", "DeliveryResult":0}
Sending message: {"Customer":"029547c6-a319-4c8b-8c23-a10b9b0e56fb", "DeliveryDateTime":"2017-02-05T14:45:14.5673443+01:00", "DeliveryResult":0}
Sending message: {"Customer":"36f8163b-3945-43ff-814b-792aef5f6706", "DeliveryDateTime":"2017-02-05T14:45:17.6514985+01:00", "DeliveryResult":0}
```

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.

GLOBAL INTEGRATION BOOTCAMP

The screenshot shows the 'Device Explorer Twin' application window. At the top, there are tabs: Configuration, Management, Data, Messages To Device, and Call Method on Device. The 'Data' tab is selected. Below the tabs, there's a 'Monitoring' section with the following fields:

- Event Hub: GlobalIntegrationBootcamp
- Device ID: SimulatedDevice
- Start Time: 02/05/2017 14:28:10
- Consumer Group: \$Default Enable

At the bottom of this section are three buttons: Monitor, Cancel, and Clear (which is highlighted with a blue border).

Below the monitoring section is a large text area titled 'Event Hub Data' containing the following log entries:

```
5-2-2017 14:44:58> Device: [SimulatedDevice], Data:[{"Customer": "e5ca676b-1933-43aa-b63c-b4732d8aab5", "DeliveryDateTime": "2017-02-05T14:45:02.2234464+01:00", "DeliveryResult": 0}]\n5-2-2017 14:45:01> Device: [SimulatedDevice], Data:[{"Customer": "2db8e59a-9351-4e6c-8c7a-2efc98187b70", "DeliveryDateTime": "2017-02-05T14:45:05.3008687+01:00", "DeliveryResult": 0}]\n5-2-2017 14:45:04> Device: [SimulatedDevice], Data:[{"Customer": "942530b1-2f44-4bad-a26f-5f08b2ac2bd", "DeliveryDateTime": "2017-02-05T14:45:08.3797594+01:00", "DeliveryResult": 0}]\n5-2-2017 14:45:07> Device: [SimulatedDevice], Data:[{"Customer": "1652ab1a-dd2c-4f52-ba95-9296f31e7257", "DeliveryDateTime": "2017-02-05T14:45:11.4593982+01:00", "DeliveryResult": 0}]\n5-2-2017 14:45:10> Device: [SimulatedDevice], Data:[{"Customer": "029547c6-a319-4c8b-8c23-a10b9b0e56fb", "DeliveryDateTime": "2017-02-05T14:45:14.5673443+01:00", "DeliveryResult": 0}]\n5-2-2017 14:45:13> Device: [SimulatedDevice], Data:[{"Customer": "36f8163b-3945-43ff-814b-792aef5f6706", "DeliveryDateTime": "2017-02-05T14:45:17.6514985+01:00", "DeliveryResult": 0}]
```

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

GLOBAL INTEGRATION BOOTCAMP

The screenshot shows the Azure Document Explorer interface. On the left, there's a sidebar with options like 'Search (Ctrl+)', 'Default consistency', 'Keys', 'Properties', 'Locks', 'Automation script', 'OPTIONS', 'Browse', 'Scale', 'Settings', and 'Document Explorer'. The main area has a search bar and a list of documents. One document, 'GlobalIntegrationBootcamp' with ID '2/5/2017 2:26:54 PM', is selected and expanded. Its JSON content is displayed on the right:

```

1 {
2   "Customer": "437bf702-a0ac-4cb7-9d95-f1a38d59b577",
3   "DeliveryDateTime": "2017-02-05T14:26:54.6616836",
4   "DeliveryResult": 0,
5   "EventProcessedUtcTime": "2017-02-05T14:26:54.4681462Z",
6   "PartitionId": 0,
7   "EventEnqueuedUtcTime": "2017-02-05T14:26:54.458Z",
8   "IoTHub": {
9     "MessageId": null,
10    "CorrelationId": null,
11    "ConnectionDeviceId": "SimulatedDevice",
12    "ConnectionDeviceGenerationId": "636218978571422765",
13    "EnqueuedTime": "0001-01-01T00:00:00",
14    "StreamId": null
15  },
16  "id": "2/5/2017 2:26:54 PM"
17 }

```

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

The screenshot shows the Power BI service interface. On the left, there's a navigation pane with 'My Workspace > Datasets'. Under 'Datasets', it says 'Streaming data' and lists one dataset: 'GlobalIntegrationBootcamp'. The main area shows a table with columns: NAME, TYPE, USED IN DASHBOARDS, and HISTORICAL. The 'USED IN DASHBOARDS' column shows 'Enabled'. On the right, there are various icons for managing datasets.

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.

The screenshot shows a Power BI report. At the top, there are navigation buttons: File, View, Reading view, Explore, Text Box, Shapes, Visual Interactions, Refresh, Duplicate this page, Save, and more. The main area contains a bar chart titled 'Count of Customer by DeliveryResult'. The Y-axis ranges from 0 to 100, and the X-axis shows categories 0, 1, and 2. The chart shows values approximately 85 for category 0, 5 for category 1, and 0 for category 2. To the right of the chart is a 'Visualizations' pane with various chart and table icons, and a 'Fields' pane listing fields from the 'GlobalIntegrationBootcamp' dataset, including Customer, DeliveryDateT..., DeliveryResult (selected), EventEnqueue..., EventProcess..., IoTHub, and PartitionId.