Measuring Occupancy with Delta Controls O3 Sense, Azure IoT, and ICONICS

Contents

Copyright and Confidentiality

By accessing and using the installation instructions (the "instructions") you acknowledge and agree, on your behalf and on behalf of the person, entity or other organization on whose behalf you are accessing the instructions, that neither Microsoft, ICONICS, Delta Controls, nor any of its service providers, including, without limitation, any system integrator or independent software vendor: (1) makes any representations or warranties of any kind, either express, implied, statuary or otherwise with respect to the instructions, including the accuracy, completeness or usefulness thereof; and (2) shall be liable for damages of any kind, under any legal theory, arising out of or in connection with your election to follow or use, or inability to follow or use, the instructions, including any direct, indirect, incidental, special, punitive or consequential damages, or for loss of use, loss of profits, loss of data, loss of business, or loss of privacy or security, even if foreseeable, arising out of or in connection with your election to follow or use, or inability to follow or use, the instructions. You further acknowledge and agree that your use of the instructions, whether directly or indirectly, is at your own risk and that you expressly assume all risk in connection with your use of the instructions. If you do not agree to the foregoing, you may not access or use the instructions.

Authors

- Spyros Sakellariadis, Microsoft Corporation
- Maksym Mushkin, Microsoft Corporation
- Zhi Wei Li, Director of Innovation & Incubation Solutions, ICONICS
- Gamal Mustapha, Director of Product Management, Delta Controls Inc.

1 Introduction

Monitoring the occupancy of spaces in commercial buildings and spaces has many benefits. Obvious scenarios include security, safety, and energy conservation – is there someone in the building when it is supposed to be empty, is there someone on a construction site when it is not safe, or is a room being heated when it is not in use? This document is being written during the coronavirus pandemic, and monitoring occupancy has taken on an additional importance. Which spaces in an office building are occupied and will need to be sanitized after the occupants leave?

Monitoring occupancy poses a couple of technical challenges which need to be overcome. First, detecting the presence of someone in a space can be done using motion, audio, heat, or visual sensors, but on their own each are subject to false readings – is the motion due to the wind, or a cat, is the heat due to a portable heater or is the occupant present but not moving. Second, just detecting the presence of someone is not adequate, as you need that information to be analyzed and appropriate action taken.

In the following sections we describe using an occupancy sensing solution from Delta Controls connected to the Microsoft Azure cloud and using a couple of different technologies from Microsoft and ICONICS to analyze the data.

2 Infrastructure overview

2.1 On-premises infrastructure

In the setup described in this paper, we are using a Delta Controls O3™ Sense to monitor room occupancy with a combination of temperature, humidity, motion, sound, and light sensors. It has a hardwired connection to a Windows 10 computer and communicates over BACnet/IP with an ICONICS IoTWorX application running on that computer. In turn, the IoTWorX application communicates over the Internet to applications in the Microsoft Azure cloud. The physical configuration is shown in Figure 1:

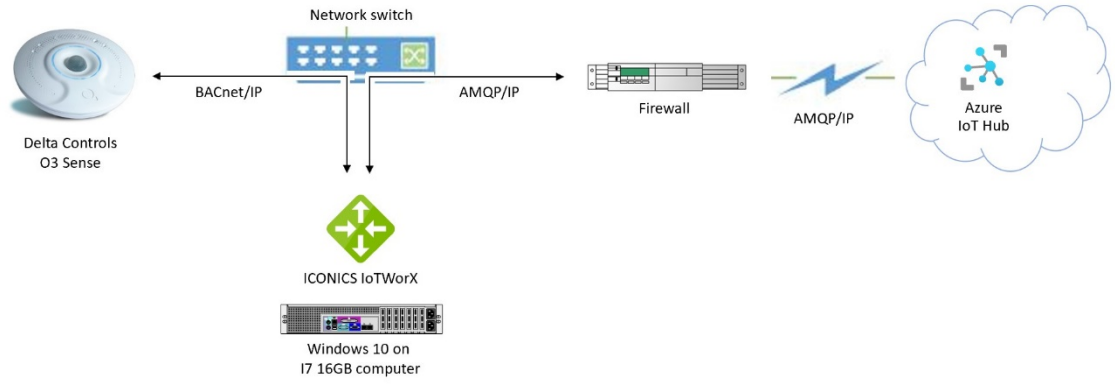

Figure 1 Physical Layout

In this configuration we use IoTWorX to read and write values from and to the O3 Sense. Specifically, we configure IoTWorX to perform the following functions:

1. Connect to the O3 Sense via BACnet
2. Request values of certain objects on the O3 every minute.
3. Reformat the data into a prescribed format .
4. Transmit that data to Azure IoT Hub.

2.2 Cloud infrastructure

After the data arrives in Azure IoT Hub, we use Azure IoT Hub Message Routing to route the data to an Event Hub based upon the origin and type of data. We then use an Azure Function to read the incoming data stream and write it to a SQL database, and use Power BI to display the current value and historical trends. Finally, we also use modules of ICONICS GENESIS64 to analyze and display the data. The overall flow is shown in Figure 2:

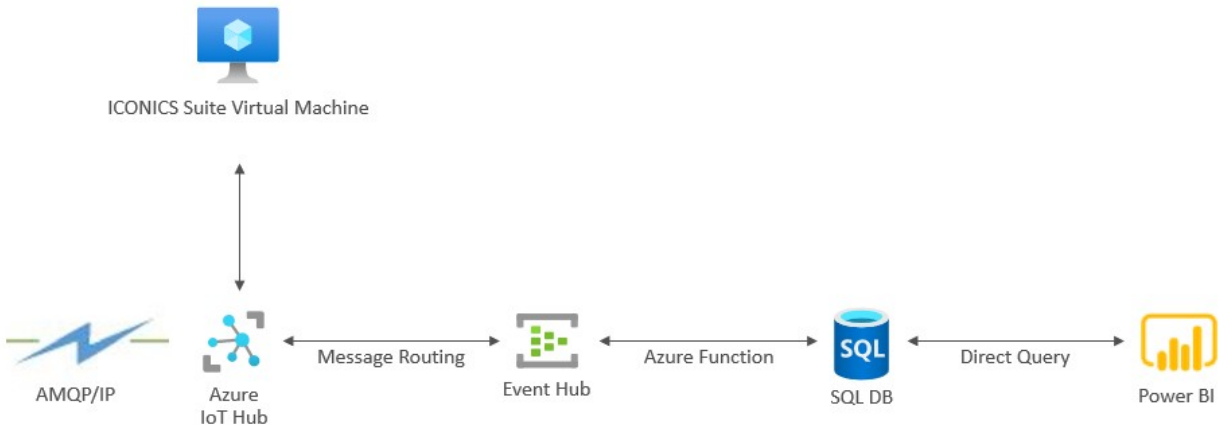

Figure 2 Software components

The following sections contain a description of how to configure the IoTWorX gateway and the Azure components to monitor the occupancy and other elements detected by the O3 Sense.

3 Configuring Azure prerequisites

3.1 Azure Resource Group

This article assumes the reader has basic knowledge of Microsoft cloud products and services and understands how to create and configure resources. Consequently, only descriptions or diagrams of the final configuration will be included, not step-by-step instructions.

The example described here uses various Azure services, deployed in a single resource group shown below. We called the resource group **IoT_projects** when creating this configuration. The final set of services looked like the following:

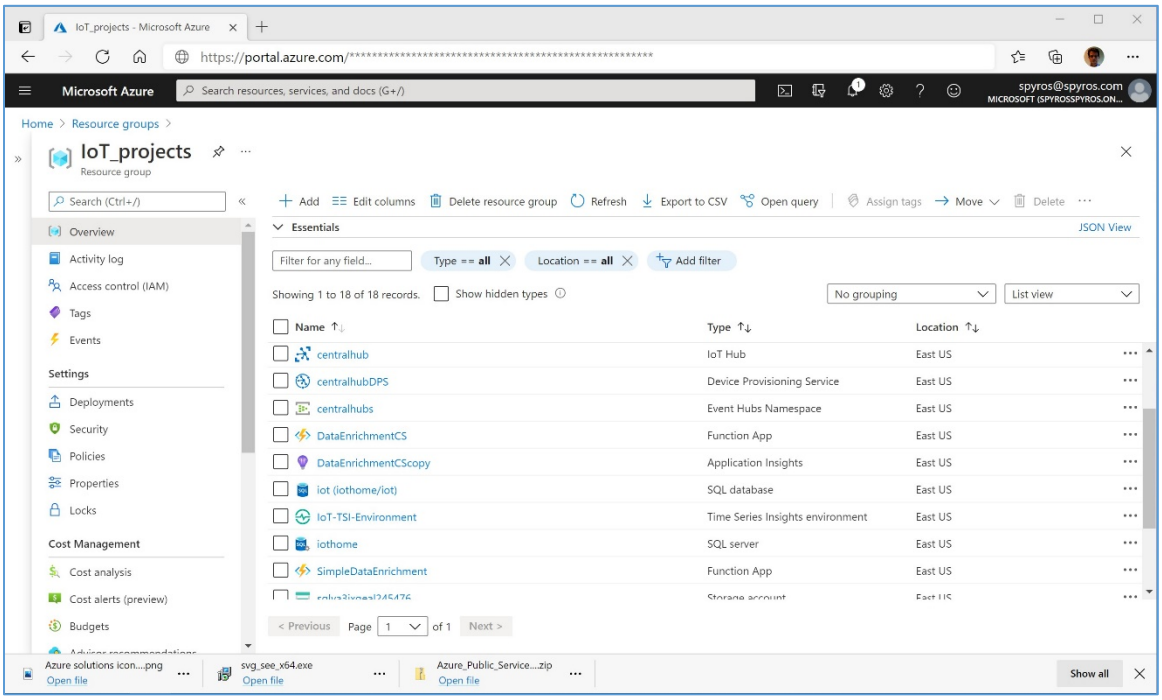

Figure 3: Azure Resource Group

The key services we will use in this solution are the following:

Resource	Type	Function
centralhub	Azure IoT Hub	Receive data from the O3 Sense
iot	SQL database	Store data received from the O3 Sense
iothome	SQL server	Holds SQL database
DataEnrichmentCS	Function App	Writes data from IoT Hub to SQL Server

3.2 Azure IoT Hub

The first task after creating the empty resource group is to create an Azure IoT Hub to receive the data from the O3 Sense. In the Azure portal select **+ Create a resource,** select the **Internet of Things** category, and click on **IoT Hub**. To create the environment used in this example, set the parameters as follows:

Settings	Value
Subscription	Enter your Azure IoT subscription name. In our example, this is **Subscription-1**.
Resource Group	Enter **IoT_projects**.
Region	Select the region where you have created the IoT Hub. In our example, this is **East US**.
IoT Hub Name	Enter **centralhub.**

Next, select the **Built-in endpoints** category, and create a couple of consumer groups for use by different readers of the data:

- Delta1
- Delta2

Next, from the left menu select **IoT Devices**, then select **+ New** at the top of the page to create a new device. Add the following:

Name	Value
Device ID	Enter **IoTWorX**.

Finally, note the following parameters for the IoT Hub, which will be needed later:

Parameter	Value
Host name	From **Overview tab**
IoT Hub primary connection string	From **Shared Access policies** à **iothubowner**
Device primary connection string	From **IoT devices** à **IoTWorX**

3.3 Event Hub

Next, we need an Event Hub to which we will route a subset of the data coming into IoT Hub. In the Azure portal select **+ Create a resource,** enter **Event Hubs** in the search category, click on **Event Hubs** and **Create**. To create the environment used in this example, set the parameters as follows:

Settings	Value
Subscription	Enter your Azure IoT subscription name. In our example, this is **Subscription-1**.
Resource Group	Enter **IoT_projects**.
Namespace name	Enter **centralhubs**
Location	Select the region where you have created the IoT Hub. In our example, this is **East US**.
Pricing tier	Select **Standard.** Do not select Basic, as Basic allows only one consumer group and we need two in order to use Visual Studio to view data coming into the Event Hub.

Click **Review + create.** Once the Event Hub is created, go to the resource. From the left menu, select **Event Hubs** and click **+ Event Hub** at the top of the page. To create the environment used in this example, set the parameters as follows:

Settings	Value
Name	Enter **iotworx**.

3.4 Azure SQL Server

Prior to installing the on-premises components, we also created a SQL database and tables to store the data. In the Azure portal select **+ Create a resource** and select the **SQL Database** category to bring up the **Create SQL Database** page. To create the environment used in this example, set the parameters as follows:

Azure Service	Value
SQL Server	Enter **iothome.database.windows.net**.
SQL Database	Enter **iot**.

The completed deployment is shown here:

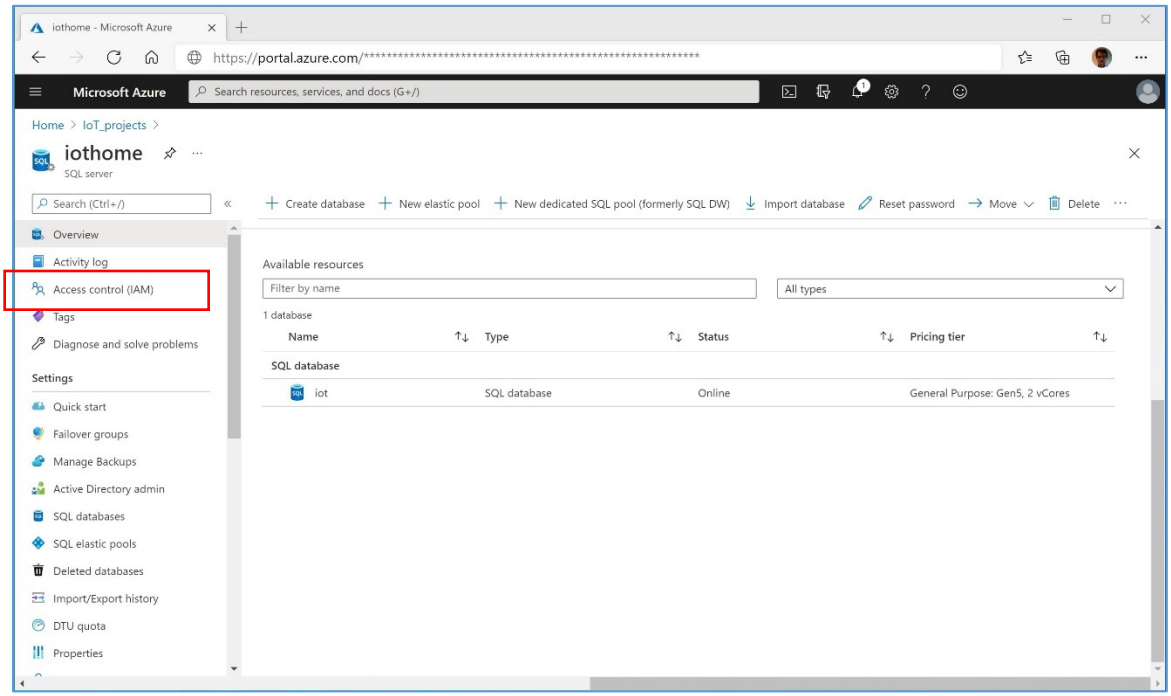

Figure 4: SQL Server overview

Next, we need to create the table in the database. On your desktop launch SQL Server Management Studio and select **File → Connect Object Explorer**. Enter **iothome.database.windows.net** for the name of the database and enter your SQL authentication credentials. Select the **iot** database, click **New Query**, and run the following query to create a table to hold the data from the O3 Sense.

```
CREATE TABLE [dbo].[Telemetry](
	[Building] [varchar](50) NOT NULL,
	[Parameter] [varchar](50) NOT NULL,
	[Value] [float] NULL,
	[TimeStamp] [datetime] NULL
) ON [PRIMARY]
GO
```

In SQL Server Management Studio:

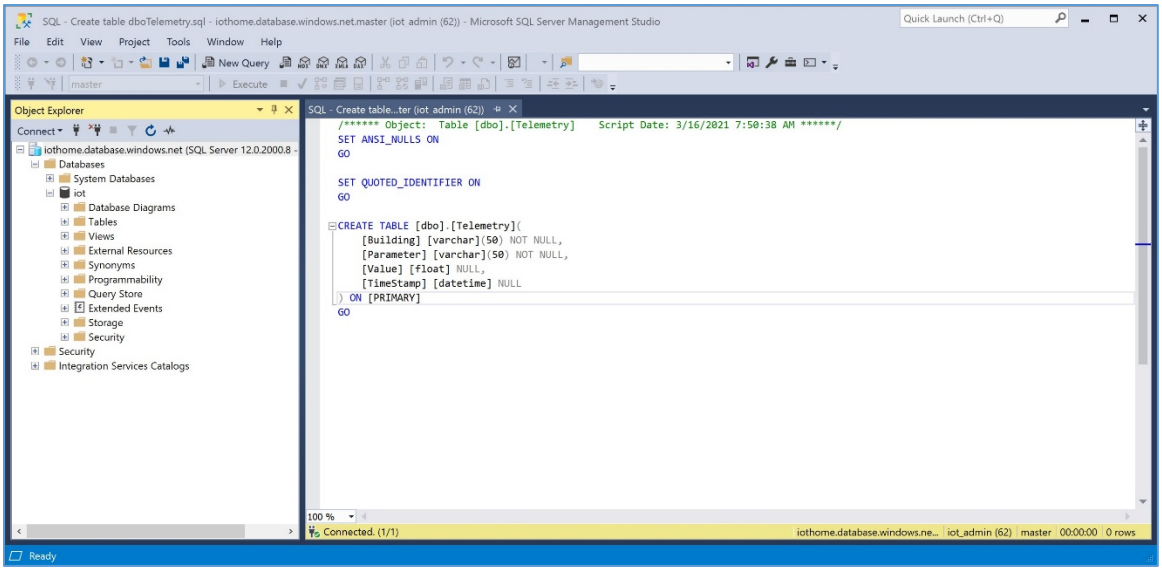

Figure 5: SQL Server Create Table

The **Telemetry** table will hold the data as it comes from the sensor, storing each measurement in a separate record. We also need a way to see all the measurements at a moment in time, which is complicated if each measurement is in a separate record. To do this we create a view. Select the **iot** database, click **New Query**, and run the following query to create the view:

```
CREATE VIEW [dbo].[v_O3_Pivot] AS
SELECT *
FROM
  (SELECT *
   FROM
     (SELECT PVTS.*
      FROM
        (SELECT CONVERT(date, TIMESTAMP) AS Date, Building,TimeStamp,[Humidity],
                [Occupant_temperature],[Internal_temperature],[IR_temperature],
                [Temperature_setpoint],[Acoustic_occupancy],
                [Acoustic_occupancy_threshhold], [Audio_retrigger_period],
                [Audio_sensitivity],[Audio_inactivity_period],[Light_level],
                [Light_level_setpoint],[Motion_sensor],[Occupancy],[Sound_level],
                [Sound_volume]
         FROM
           (SELECT t1.*
            FROM Telemetry t1) AS SourceTable PIVOT(MAX(Value)
            FOR PARAMETER IN([Light_level], [Light_level_setpoint], [Motion_sensor],
                [Sound_level], [Humidity], [Temperature_setpoint],
                [Occupant_Temperature], [Internal_temperature], [IR_temperature],
                [Acoustic_occupancy], [Occupancy], [Sound_volume],
                [Acoustic_occupancy_threshhold], [Audio_retrigger_period],
                [Audio_sensitivity], [Audio_inactivity_period]))
                AS PivotTable) AS PVTS
      WHERE [Building] IS NOT NULL AND [Temperature_setpoint] IS NOT NULL AND
      [Audio_sensitivity] IS NOT NULL) AS PVTSI) AS PVTSIP
GO
```

In SQL Server Management Studio:

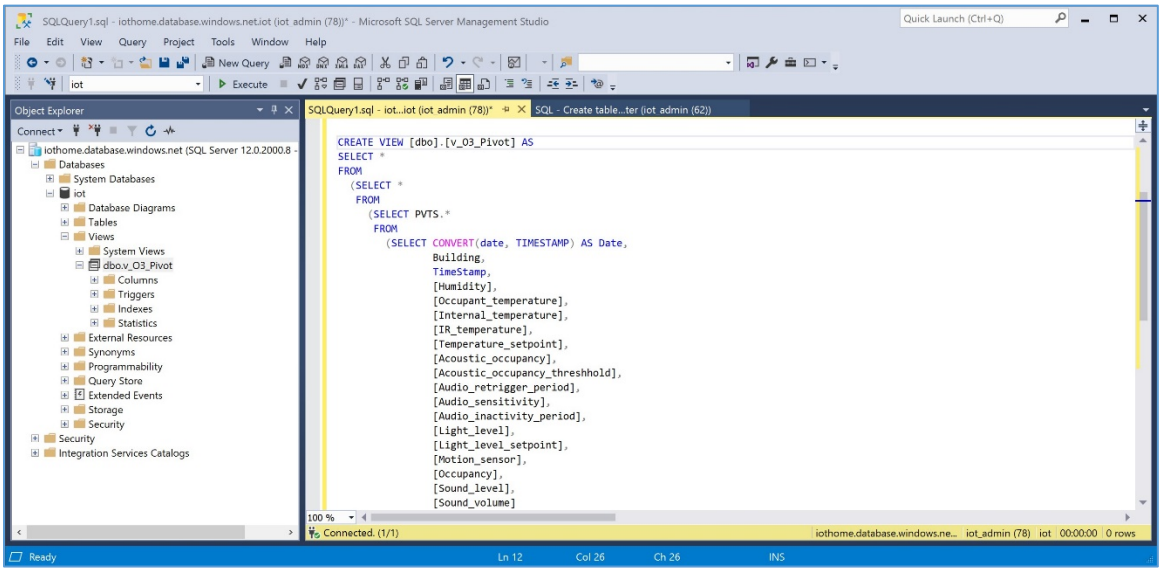

Figure 6: SQL Server Create View

Finally, we need to get the connection string for the database, which we will use later in an Azure Function. In the Azure portal, select the SQL database **iot**. In the left pane, select **Connection strings**. Note the **ADO.NET** connection string.

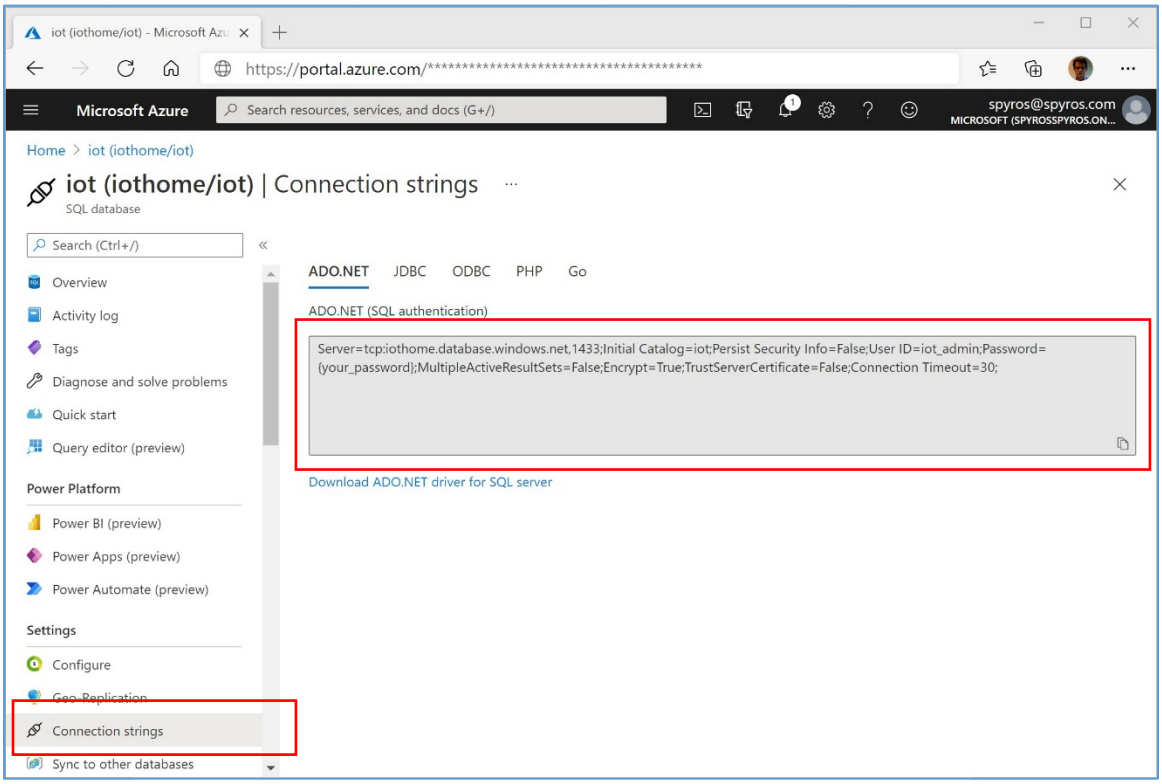

Figure 7: SQL database connection strings

It should look like the following:

Connection string	Value
ADO.NET	`Server=tcp:iothome.database.windows.net,1433;Initial Catalog=iot;Persist Security Info=False;User ID=iot_admin;Password={your_password};MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=False;Connection Timeout=30;`

When the string is needed later, you will need to insert the password created for the database in the place of **[your password]** in the string above.

3.5 VM based SQL Server

An alternative is to use a pre-existing SQL server, either local or installed in a VM you already have. In that case, create the database and tables as in the previous section. Once created, construct the connection string that you will need later as follows:

Connection string	Value
ADO.NET	`Server=tcp:<DNS name of the VM>,1433;Initial Catalog=iot;Persist Security Info=False;User ID=iot_admin;Password={your_password};MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=True;Connection Timeout=30;`

There are two differences from the connection string used if the SQL Server is an Azure SQL Server. First, the Server name is not the DNS name of the SQL Server, it is the DNS name of the VM. Second, you need to change TrustServerCertificate to True.

4 Configuring the O3 Sense

This guide from Delta Controls describes how to install and set up the O3 Sense. To set up the O3, you will need an Android or iOS device with the O3 Setup app installed. You can get the app from Google Play or the App Store.

Key steps to configure the O3 are as follows:

1. Open the O3 Setup app and select Continue to enter Lite Mode.
2. In the lower right corner of the screen, select Connect.
3. Select your O3 to initiate a connection over Bluetooth O3 units are displayed in the order of signal strength.
4. Once the connection is initiated, select Verify. The O3 should play a sound and the light ring flashes blue.
5. Select *Yes, connect to this hub*. Data loads from the hub and the status changes to Connected.
6. You can now view device information and sensor data from the hub in the Diagnostics tab.
7. After connecting to the hub, select the Settings tab.

8. By default, the O3 is set to DHCP. If you want to assign a static IP address to the O3, select the pencil icon next to Network, select Static, enter the IP settings, then select Save.

By default, the O3 is set to BACnet Ethernet. If you want to change the protocol to BACnet/IP, select the pencil icon next to BACnet. Select IP, then select Save. The BACnet device ID and UDP port can also be changed if desired.
When finished, the setup app should show a screen like this:

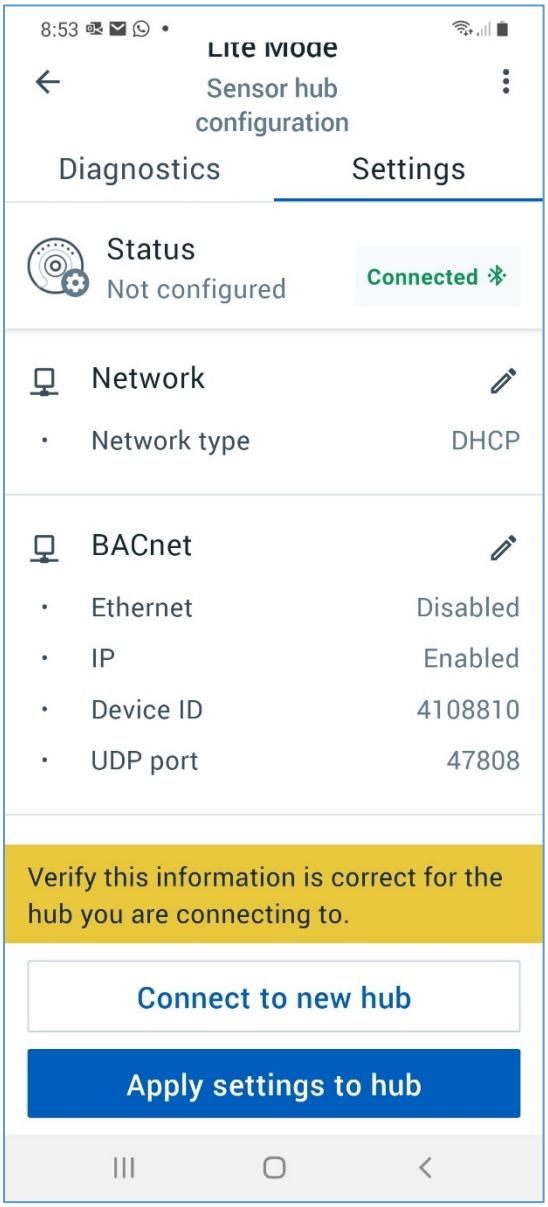

Figure 8: O3 configuration in mobile app

Note the device ID and UDP Port in this app – you will need later. Click **Apply settings to hub**, then click on the **Diagnostics** tab to see additional information:

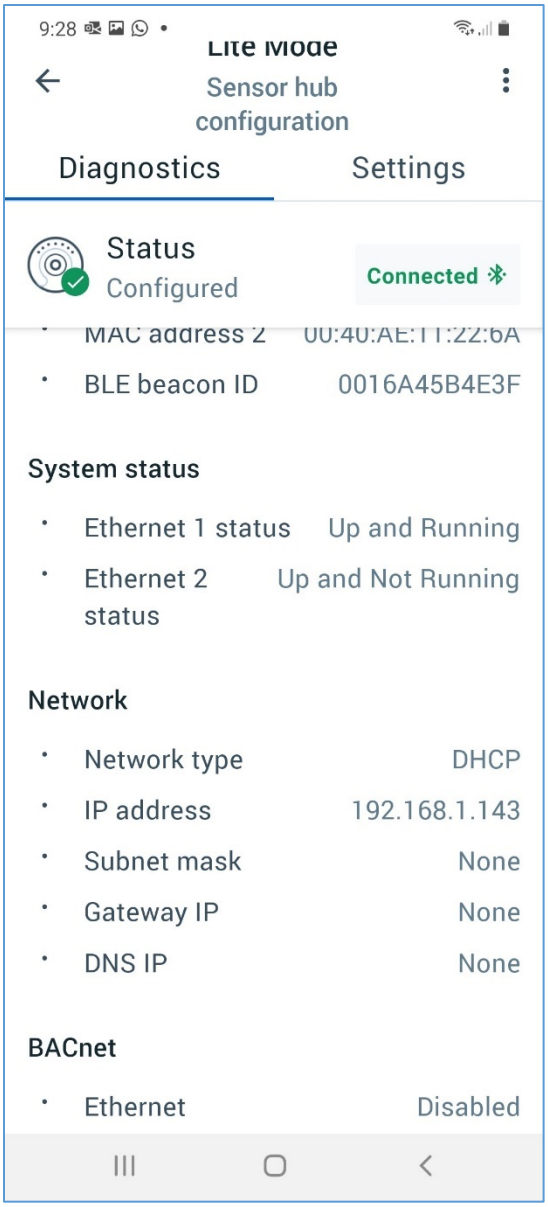

Figure 9: Diagnostics for the O3

Note the IP address – you will need it in the next section.

5 Configuring ICONICS IoTWorX to push data from the O3 Sense to Azure

Installing and configuring ICONICS IoTWorX is covered in detail in a previously published document on the ICONICS website, Using IoTWorX as a Gateway (below referred to as *Using IoTWorX*).

Follow the instructions in that document for using ICONICS Workbench, inserting the values shown below instead of those shown in *Using IoTWorX*.

5.1 Specify how to access the O3

Follow the instructions in *Using IoTWorX,* Section 3.1 to create an entry for the O3, using the following settings:

Parameter	Value
Name	Enter **Delta O3**.
Channel Type	Select **BACnetIP.**
IP address	Enter the **IP address** from the **Diagnostics** tab in the O3 mobile app.
UDP port	Enter the **UDP port** from the **Settings** tab in the O3 mobile app.

Check the **Enabled** checkbox in the **Port Settings** section. When this step is complete, the configuration in ICONICS Workbench should look like that shown below:

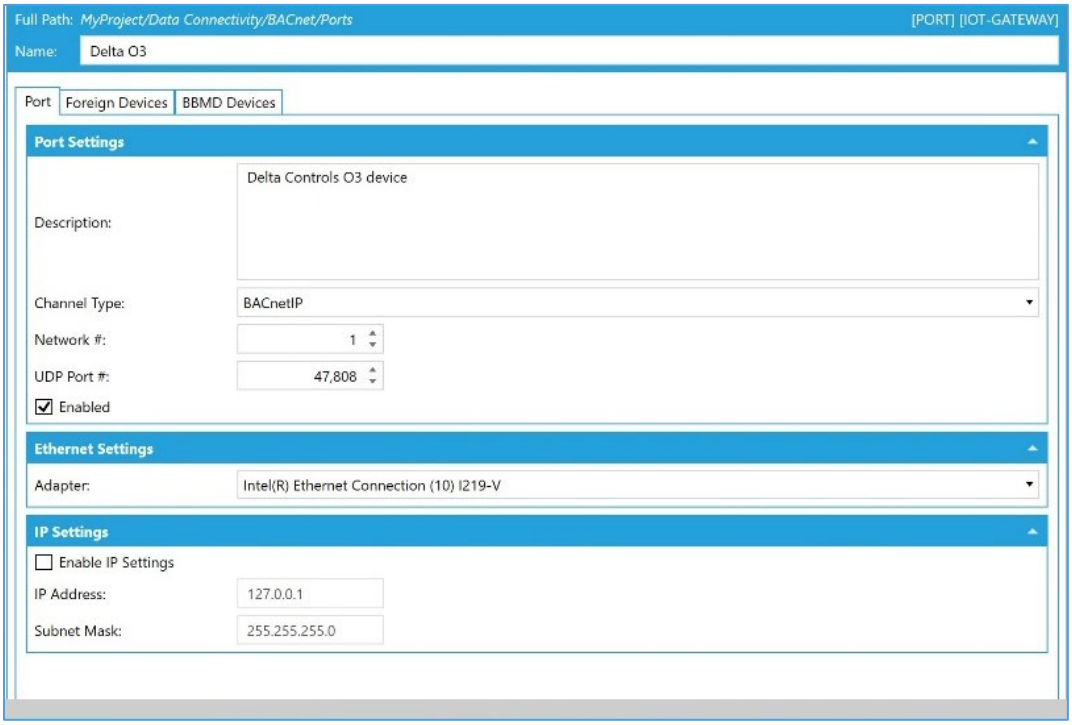

Figure 10 Port connecting to the O3

5.2 Discover devices

The data points collected by the O3 are listed in the following document: BACnet Application Guide. The ones we collect in this article are:

Name	Description
Occupant Temperature	Temperature at 1 m (3 ft) above the floor. This is a composite value derived from the O3's internal temperature sensors and the IR temperature sensor. Range: 0°C to 59°C (32°F to 138°F).
IR Temperature	Average temperature of surfaces in the O3's field of view. Range: 0°C to 59°C (32°F to 138°F).
Internal Temperature	Temperature at ceiling height. Range: 0°C to 59°C (32°F to 138°F).
Temperature Setpoint	User-entered temperature from mobile app. Measured by user at occupant height.
Occupancy Audio Retrigger Period	The amount of time (in seconds) that activity sounds can cause the hub to remain in the occupied state after motion is detected. Default value is 1200 seconds (20 minutes). Measured from most recent motion detection event.
Occupancy Inactivity Period	The amount of time (in seconds) it takes the O3 to return to the unoccupied state when no motion and no audio activity is detected. Default value is 300 seconds (5 minutes).
Acoustic Occupancy Threshold	The acoustic activity level based on the background noise level. Read-only.
Light Level	Brightness of ambient light (lx or ft-candle).
Occupant Humidity	Humidity at 1 m (3 ft) above floor. This is calculated from the occupant temperature and internal humidity using psychrometrics. Range: 0% to 100%.
Occupancy	Combined (motion + sound) occupancy signal. Active state when motion and sound is detected. See How Occupancy Works for more details.
Motion Sensor	Motion occupancy signal. Active state when motion is detected.
Acoustic Occupancy	Acoustic occupancy signal. Active state when acoustic activity level (AI10) is above the internal acoustic occupancy threshold (AV38).
Motion Sensitivity	Controls the sensitivity of the PIR sensor to changes in movement levels within the detection area. 100% = maximum sensitivity.
Occupancy Audio Sensitivity	Controls the sensitivity of the acoustic occupancy sensor to changes in audio levels within the detection area. 100% = maximum sensitivity.
Sound Level	Level of ambient noise (dB SPL). Unfiltered audio level across the entire spectrum.
Light Level Setpoint	(Optional) User-entered light level from mobile app. Records the light level read by the hub (AI12) when the lighting in the space is set to the desired brightness. This setpoint can be retrieved later by the control system to set the feedback loop, etc.

To identify the data points on the O3 follow the instructions in *Using IoTWorX,* Section 3.2.2, Add multiple devices through a network scan. When this step is complete, your configuration in Workbench should look like this:

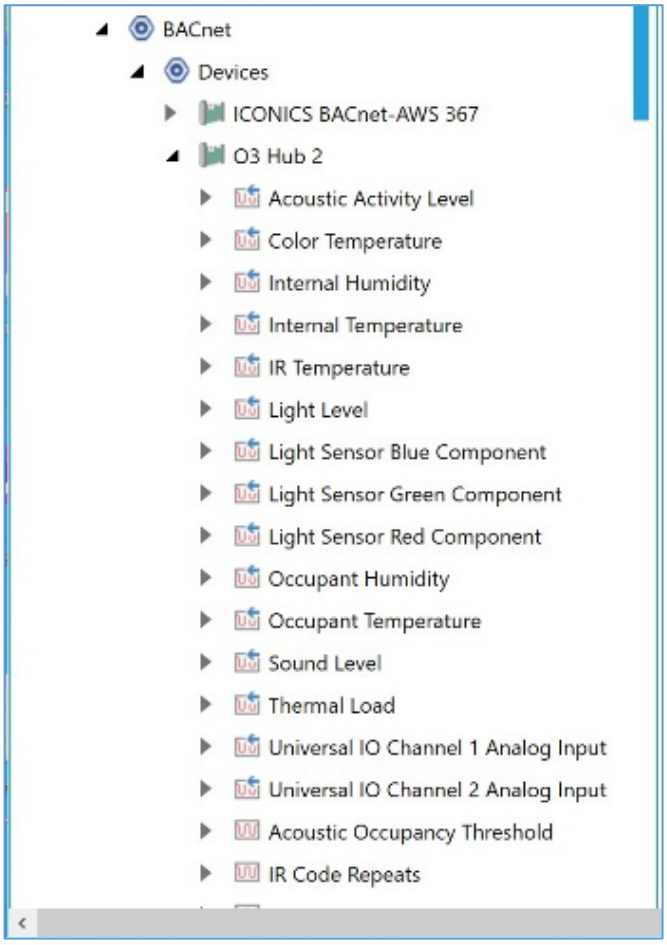

Figure 11 Devices and objects discovered on the O3

5.3 Create a publish list

For this test installation, enter **Delta O3 Publist** for the name of the Publish List and select the points listed above. When this step is complete, your publish list configuration in ICONICS Workbench should look like this:

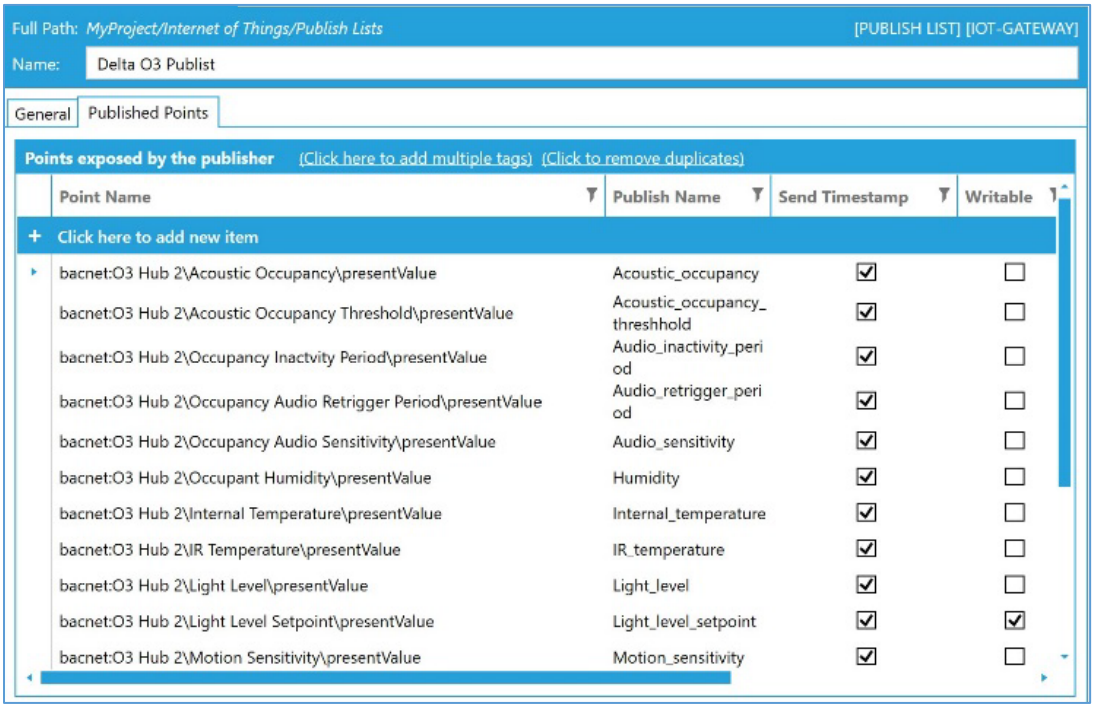

Figure 12 Publish List with selected objects of interest

It is useful to enter a Publish Name manually in the Publish Name column, as that will make it easier to parse the data in Azure later.

5.4 Create a custom encoder

For this setup, create a custom encoder called **CustomJSONEncoder**.
In the General Settings section, select **One value for each message** for the Message Type.
In the Value Format enter:

```
{
  "id": "%PUBLISHNAME%",
  "v": "%VALUE%",
  "q": "%STATUS.GOOD%",
  "t": "%NOWUTC.TEXT%"
}
```

When this step is complete, your configuration in Workbench should look like this:

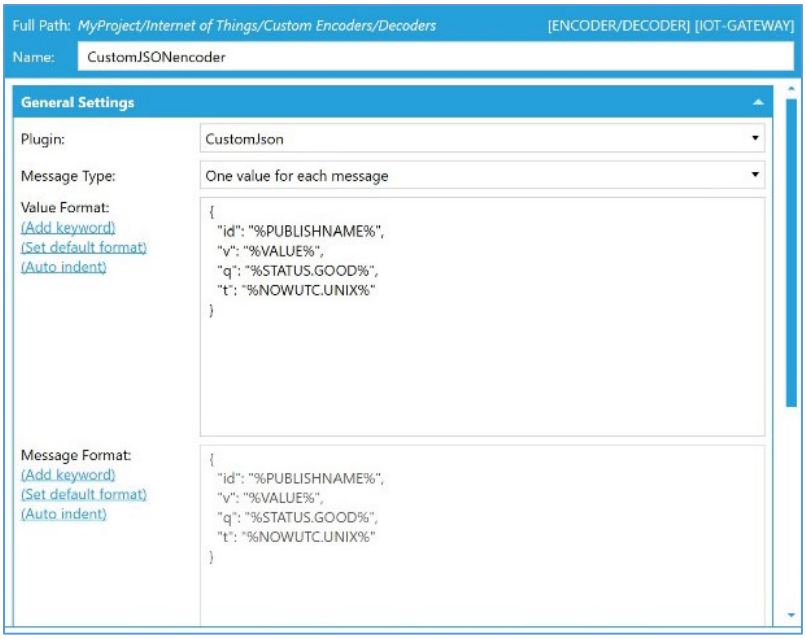

Figure 13 Custom encoder

5.5 Create a publisher connection

For this setup, enter **To_CentralHub** for the name of the Publisher Connection.
Uncheck the Enable compatibility checkbox.
For the Encoder, enter **CustomJSONEncoder**.
For the Publish List, enter **Delta O3 PubList**.
For the Connection String enter the **Device primary connection string** noted in Section 3.2 above.

When this step is complete, your configuration in Workbench should look like this:

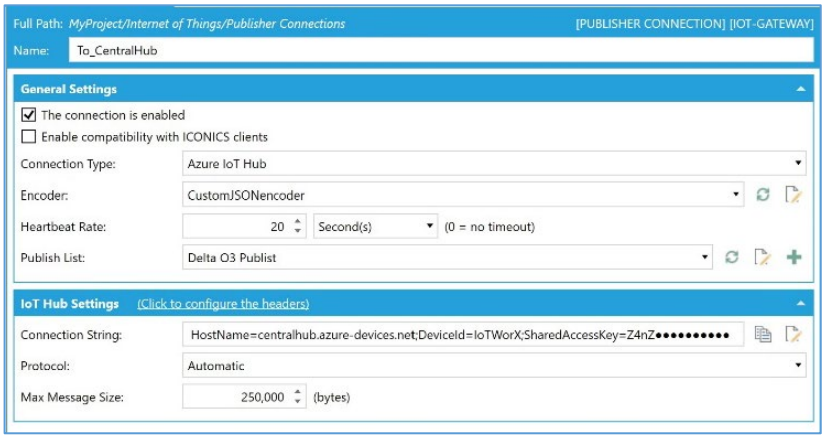

Figure 14 Publisher Connection

After you create and save the Publisher Connection, click the button in the top menu bar to start or restart the Publisher Service. At this point, data should start flowing to Azure IoT Hub, which you can confirm first by locally viewing the data being sent by IoTWorX and then by viewing the data received at the hub.

5.6 Viewing data sent by IoTWorX

To visualize the data being sent by IoTWorX, launch the Data Explorer application in the ICONICS Tools folder in the computer's Start Menu. Navigate to **My Computer → Data Connectivity → BACnet → O3 Hub 2** and click on **Occupant Temperature**. You should see a **Present Value** for the temperature:

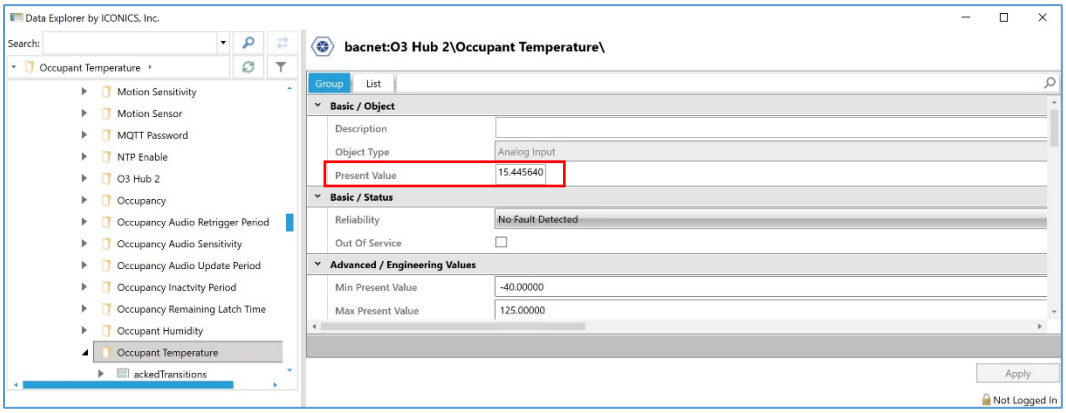

Figure 15: Viewing data collected by IoTWorX using ICONICS Data Explorer on the gateway machine

5.7 Viewing data received by IoT Hub

See Install and use Azure IoT explorer for step-by-step instructions for using the Azure IoT explorer tool to monitor incoming data. Upon launching Azure IoT Explorer, enter the **IoT Hub primary connection string** noted in Section 3.2 above.

If IoTWorX and IoT Hub are configured as described in this article, after navigating to **centralhub → Devices → IoTWorX → Telemetry** and clicking **Start**, data should be seen in the main window:

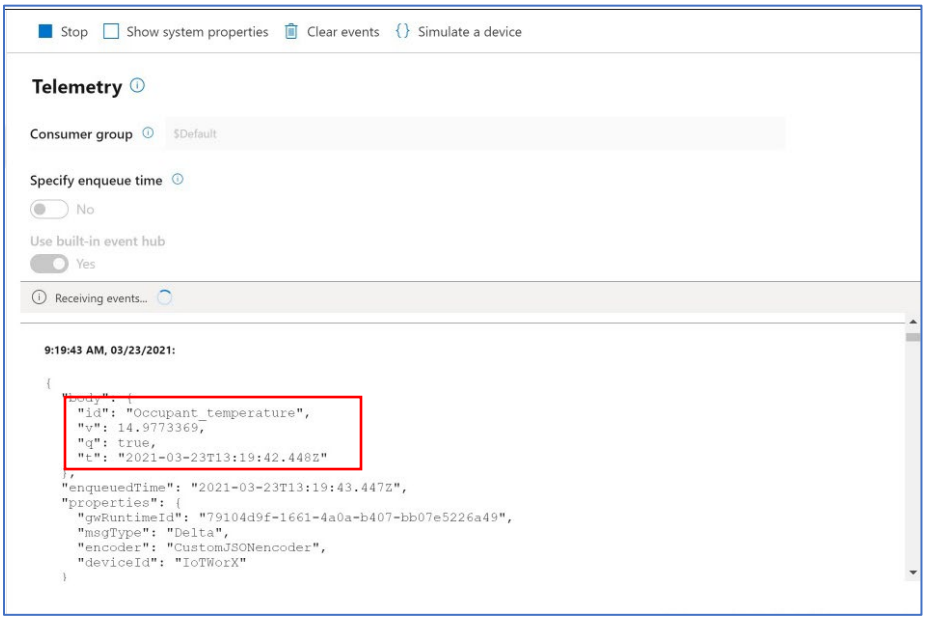

Figure 16: Viewing data received by IoT Hub

In this screen capture we see again the value of the Occupant Temperature collected by the O3 Sense, this time as it is received at the IoT Hub.

6 Routing data from IoT Hub to Event Hub

Typically, you would have many devices send data to the same IoT Hub, so we need a way to filter the incoming data for that from just the O3.

6.1 Creating a filter for the data

First, we need to create an attribute on the incoming data by which to filter it. To do this, we add a property to the Azure device twin for the device as configured in the IoT Hub. In the Azure portal, select the IoT Hub **centralhub** and click on **IoT devices** and select the **IoTWorX** device. On the **IoTWorX** device page, click on **Device twin**:

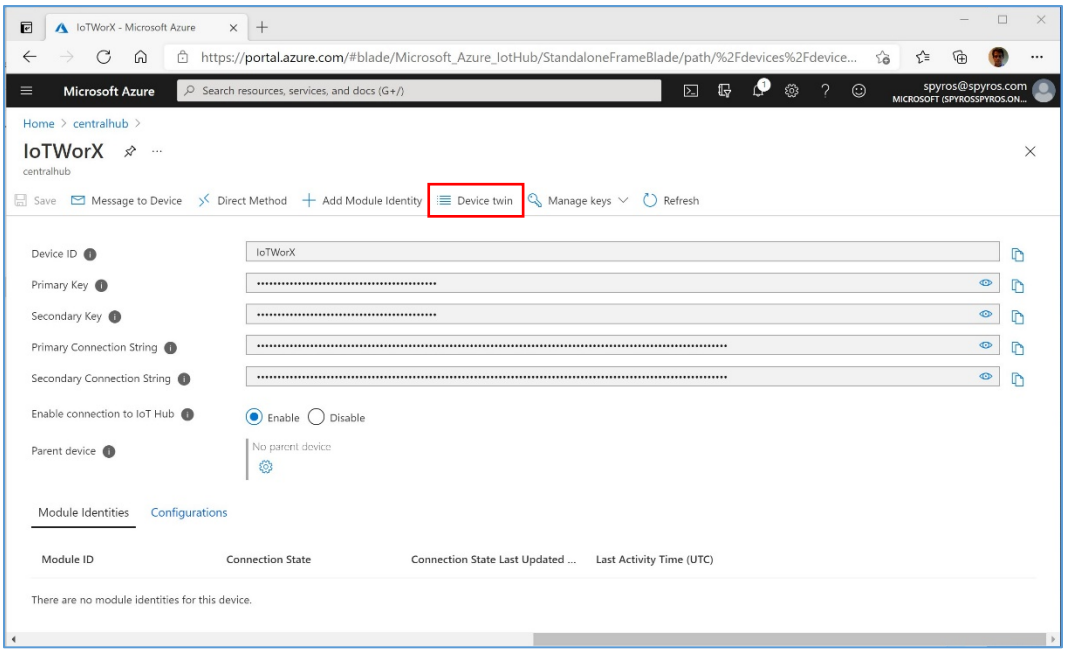

Figure 17: Azure IoT Hub Device Twin

On the next screen, note the value of the deviceID. This was automatically created for the Twin when we created the IoT device was created in IoT Hub:

```
"deviceId": "IoTWorX"
```

Note that you can also see this deviceId in IoT Explorer, show in Figure 16: Viewing data received by IoT Hub, last line. Now we can add tags section with device location if you want to use Device Twin Data Enrichment functionality. In the portal add the following:

```
"tags": {
        "deviceBuilding": "4630"
},
```

So that it looks like this:

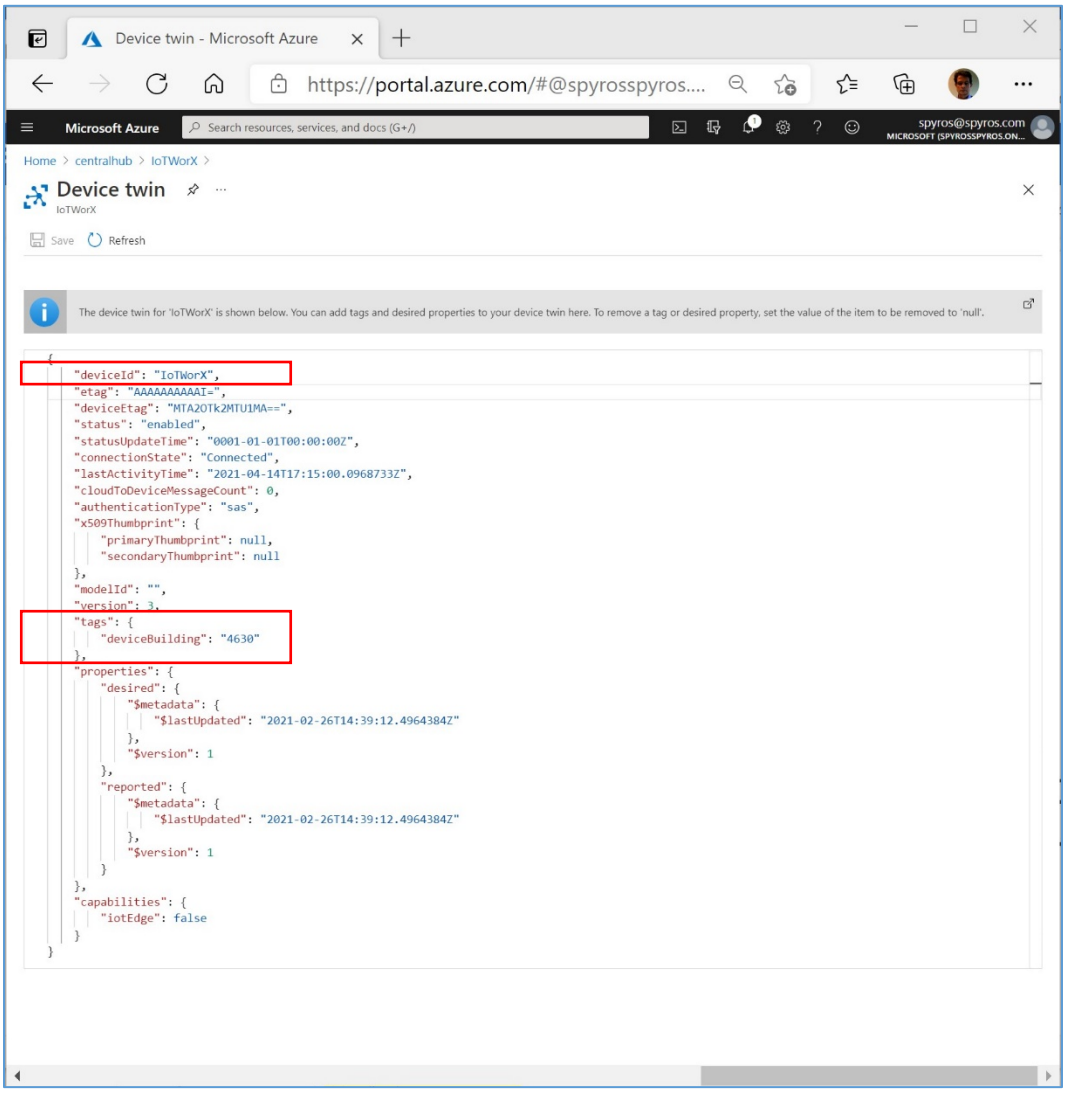

Figure 18: Azure IoT Hub Device Twin properties

6.2 Configuring routing and data enrichment

Next, we configure IoT Hub message routing for data with the device twin tag of **IoTWorX** to the Event Hub we created earlier. In the Azure portal, select the IoT Hub **centralhub** and click on **Message routing** in the left menu.

In the **Enrich messages** tab, add a message enrichment with the following parameters:

Parameter	Value
Name	Enter **deviceBuilding**.
Value	Enter **$twin.tags.deviceBuilding.**
Endpoint	Select **iotworx** in the dropdown, **Event Hubs** section**.**

In the portal it should look like this:

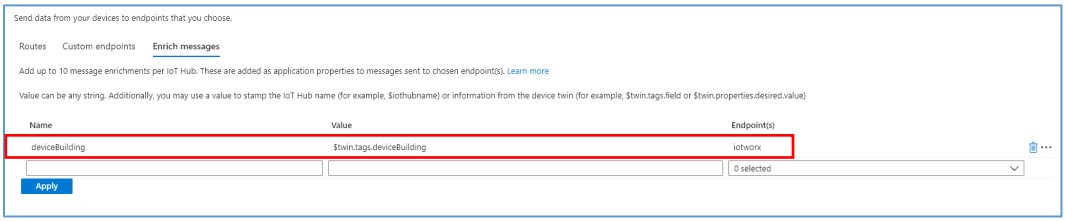

Figure 199: Azure IoT Hub Data Enrichment

Next, to add the route we want, we need to create a Custom Endpoint first. Select **Custom endpoints** tab and click **+ Add.**

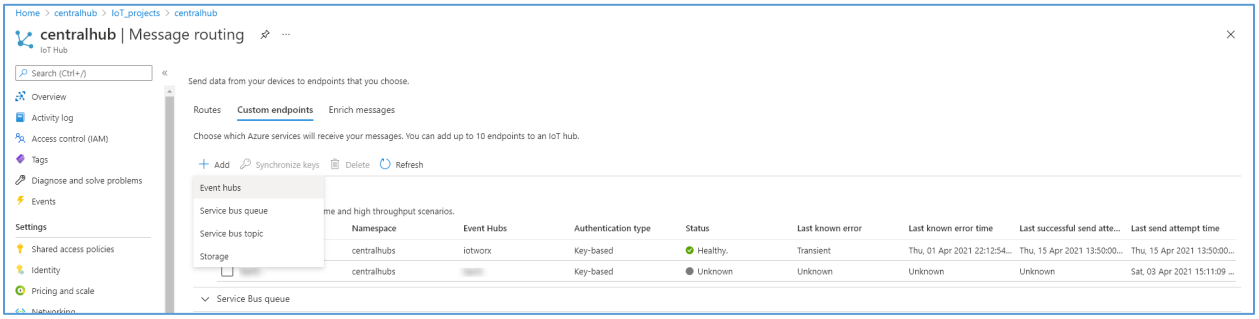

Figure 20: Azure IoT Hub Custom Endpoints

On the next page, select Event Hub namespace and Instance created previously and click **Create:**

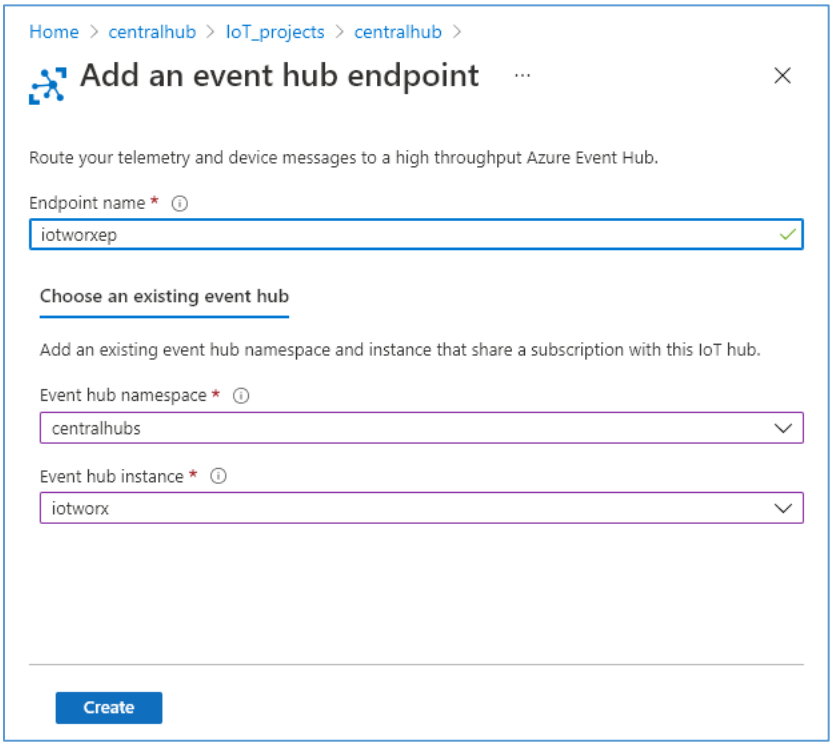

Figure 21: Azure IoT Hub Custom Endpoints creation

Now we are ready to create new Route, select the **Routes** tab, and click **+ Add.**

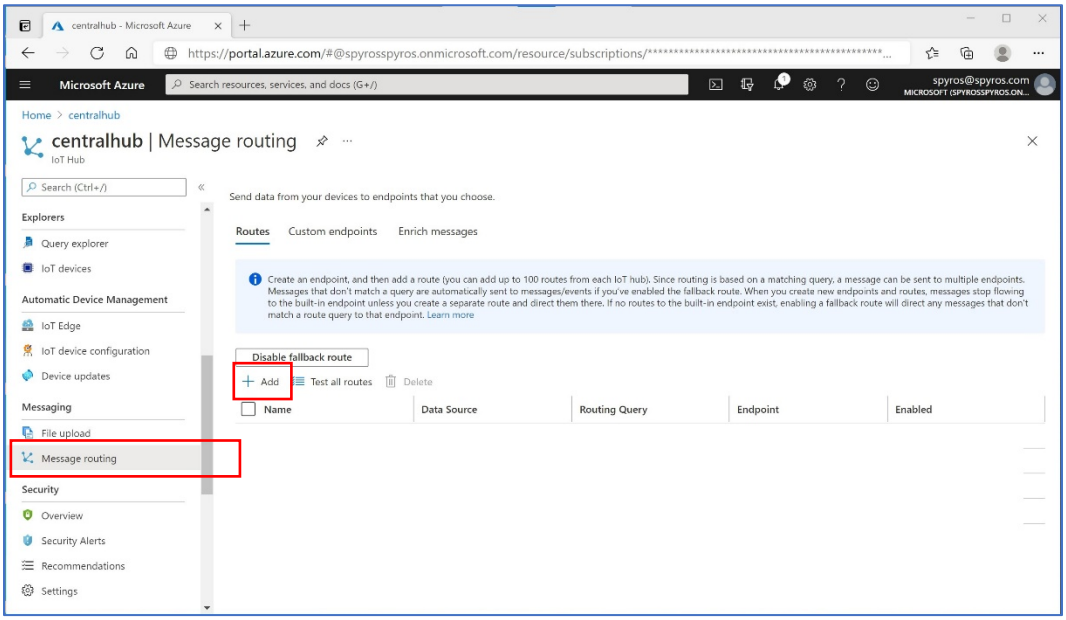

Figure20: Message routing

To create the environment used in this example, set the parameters as follows:

Parameter	Value
Name	Enter **iotworxroute**.
Endpoint	Click the down arrow and select **iotworx**.
Routing query	Enter **$twin.deviceId = 'IoTWorX'**

6.3 Viewing data received by Event Hub

To monitor the data received from the IoT Hub by the Event Hub, we will use Microsoft Visual Studio. First download and install Visual Studio Code, then the Azure Event Hub Explorer. Open Visual Studio Code and follow these steps.

1. Select **View → Extensions → Azure Event Hub Explorer.**
2. Select **View → Command Palette → Event Hub: Select Event Hub**

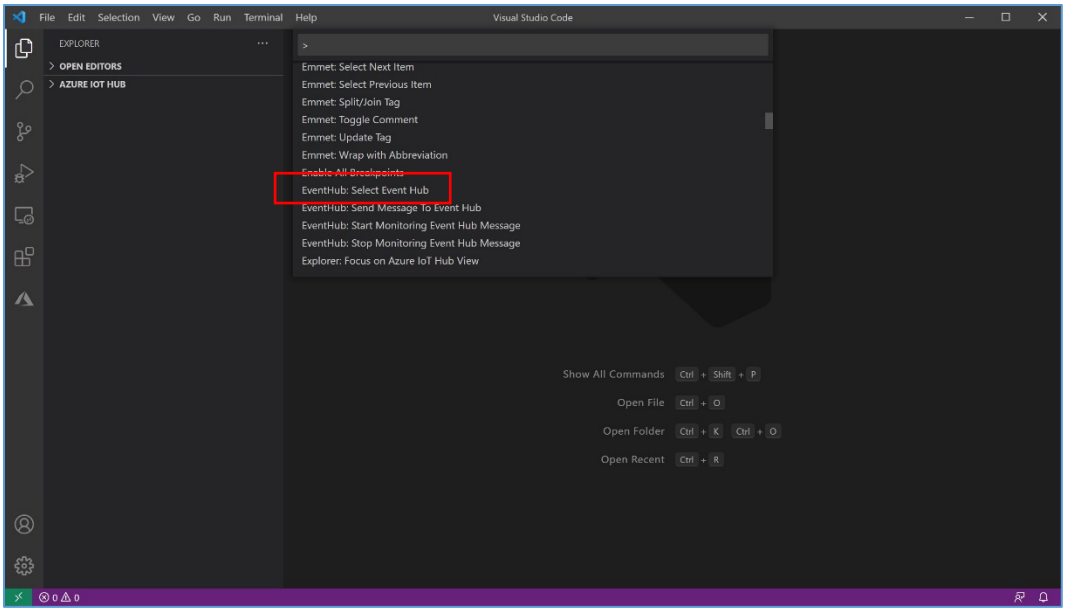

Figure 21: Select Event Hub

3. From the drop-down select subscription **Subscription-1**.
4. From the drop-down select resource group **iotprojects.**
5. From the drop-down select event hub namespace **centralhubs.**
6. From the drop-down select event hub **iotworx.**
7. From the top menu select **View → Command Palette → Event Hub: Start monitoring.**

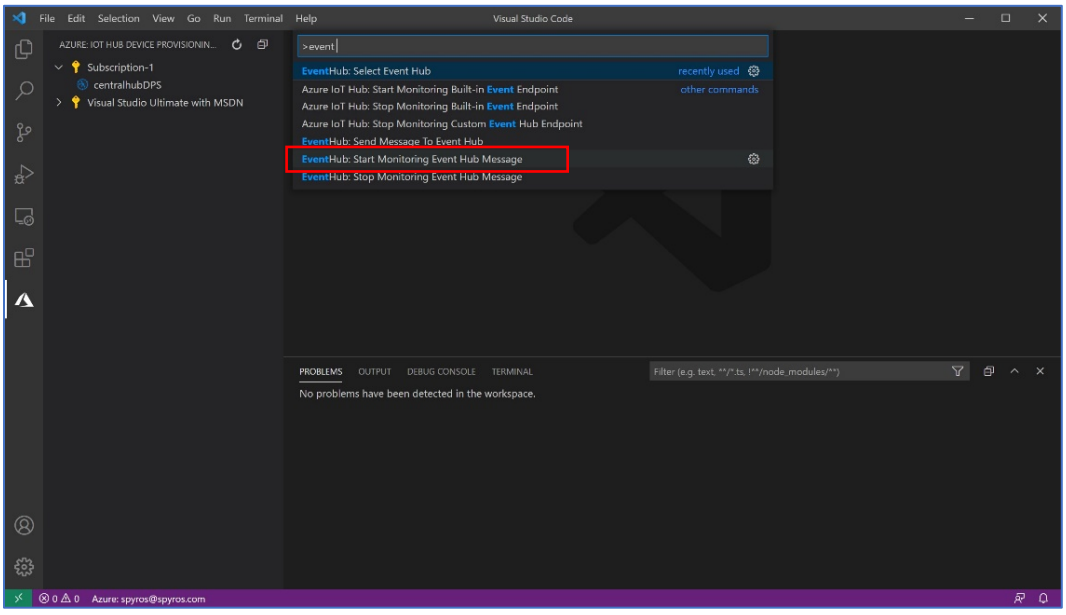

Figure 202: Start monitoring Event Hub

At this point, data should start appearing:

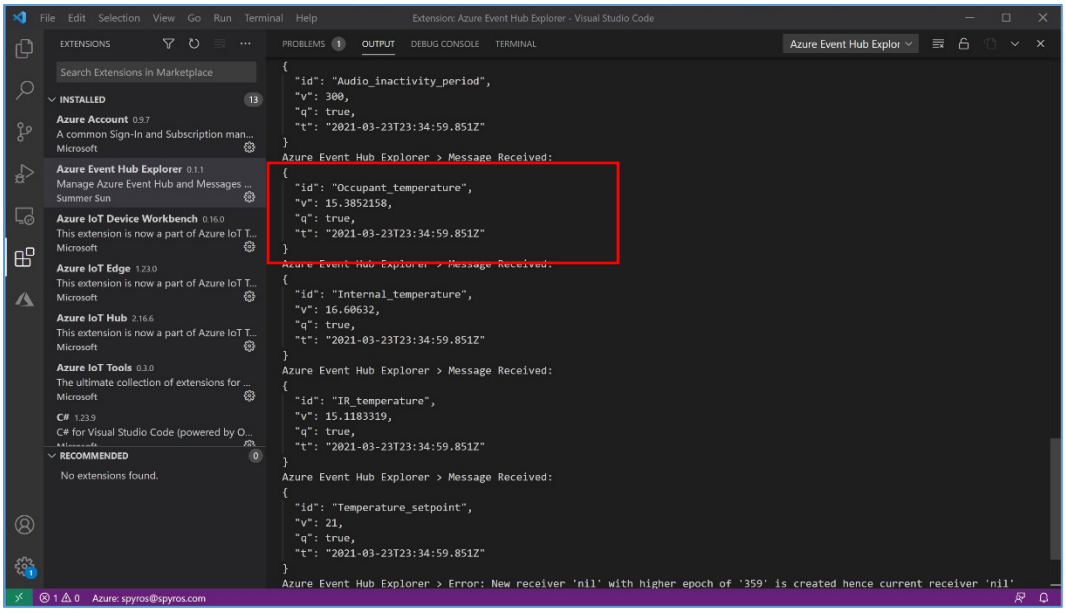

Figure 213: Data arriving in Event Hub

In this screen capture we see again the value of the Occupant_temperature collected by the O3, this time it is received at the Event Hub.

7 Configuring an Azure Function to push data from Event Hub to SQL Server

There are multiple ways to write the streaming data to SQL Server. In a previous whitepaper, Monitoring Building Air Quality, we describe the steps to do this from IoT Hub with an Azure Stream Analytics job. In the section below we show how to do this from the Event Hub created above in a more cost-efficient manner using an Azure Function, though this way is more complex to set up and requires some coding skills.

7.1 Creating the Function App

In the Azure portal select **+ Create a resource** and select the **Function App** category. To create the environment used in this example, on the **Basics** page set the parameters as follows:

Setting	Value
Subscription	Enter your Azure IoT subscription name. In our example, this is **Subscription-1**.
Resource Group	Enter **IoT_projects**.
Function App name	Enter **DataEnrichmentCS**.
Publish	Select **Code**.
Runtime stack	Select **.NET**
Version	Select **3.1**.
Region	Select the region where you have created the IoT Hub. In our example, this is **East US**.

Select **Next : Hosting**. On the **Hosting** page, accept the defaults.

Select **Next : Monitoring**. On the **Monitoring** page, turn off **Application Insights**.

Select **Review + Create**, then **Create** to deploy the function app.

7.2 Specifying configuration values

When the deployment is complete, select **Go To Resource**. From the left menu select **Configuration**, then select **+ New application setting** in the right-hand pane. Add the following:

Name	Value
ConnectionString	Enter the connection string for the SQL Server database **iot**, noted above. Edit to include the password you selected for the SQL Server.

We will use this variable in the function we are about to create.

7.3 Creating the Function

Next, we create a function. When the deployment is complete, select **Go To Resource**. From the left menu select **Functions**, then select **+ Add** from the top menu. In the **Add Function** window, set the parameters as follows:

Setting	Value
Develop environment	Select **Develop in portal.**
Template	Select **Azure Event Hub trigger.**
New Function	Enter **IoTWorXToSQL**
Event Hub connection	Select recently created EventHub connection from the list (not IoT Hub EventHub endpoint)
Event Hub name	Enter **iotworx**.
Event Hub consumer group	Enter **tosql**

Click **Add** to create the function. Once created, click on **IoTWorXToSQL** in the list on the right to open the function page. Click **Code + Test** in the left menu, select **run.csx** in the drop-down at the top and replace the code in the window with the following and click **Save** (formatting below modified to fit to page)**.**

```
#r "System.Data.Common"
#r "Microsoft.Azure.EventHubs"
#r "Newtonsoft.Json"

using System;
using System.Text;
using System.Data;
using System.Data.SqlClient;
using Microsoft.Azure.EventHubs;

using Newtonsoft.Json;

public static async Task Run(EventData[] events, ILogger log)
{
    var exceptions = new List<Exception>();

    string cs = Environment.GetEnvironmentVariable("ConnectionString");

    if(string.IsNullOrEmpty(cs)) {
            log.LogError("DB Connection string is not defined!");
    }
```

(Continued on next page)

```
    foreach (EventData eventData in events)
    {
        try
        {
            string messageBody = Encoding.UTF8.GetString(eventData.Body.Array, eventData.Body.Offset, eventData.Body.Count);
            Message m = JsonConvert.DeserializeObject<Message>(messageBody);
             var deviceBuilding = "UnknownBuilding";
             if(eventData.Properties.ContainsKey("deviceBuilding")){
                deviceBuilding = eventData.Properties["deviceBuilding"].ToString();
            }
            log.LogInformation($"device Building is: {deviceBuilding}");
            var insertScript = $"INSERT INTO [dbo].[Telemetry] ([Building],[Parameter],[Value],[TimeStamp]) VALUES (@Building,
@Parameter, @Value, @Date)";
            using (SqlConnection connection = new SqlConnection(cs))
            {
                SqlCommand command = new SqlCommand(insertScript, connection);
                command.Parameters.AddWithValue("@Building", deviceBuilding);
                command.Parameters.AddWithValue("@Parameter", m.id);
                command.Parameters.AddWithValue("@Value", m.v);
                command.Parameters.AddWithValue("@Date", m.t);
                try{
                    connection.Open();
                    var rows = command.ExecuteNonQuery();
                }
                catch(Exception ex){
                    log.LogError(ex.Message);
                }
            }
            log.LogInformation($"C# Event Hub trigger function processed a message: {messageBody}");
            await Task.Yield();
        }
        catch (Exception e)
        {
            exceptions.Add(e);
        }
    }
    if (exceptions.Count > 1)  throw new AggregateException(exceptions);
    if (exceptions.Count == 1)  throw exceptions.Single();
}
public class Message{
    public string id {get;set;}
    public double v {get;set;}
    public DateTime t {get; set;}
}
```

Finally, select **function.json** in the drop-down at the top. It should look like this:

```
{
  "bindings": [
    {
      "name": "events",
      "connection": "centralhubs_RootManageSharedAccessKey_EVENTHUB2",
      "eventHubName": "iotworx",
      "consumerGroup": "tosql",
      "cardinality": "many",
      "direction": "in",
      "type": "eventHubTrigger"
    }
  ]
}
```

Note that the configuration file specifies **iotworx,** the Event Hub from which the function will read data.

The **IoTWorXToSQL** function is called every time a message or a batch of messages arrives at the **iotworx** Event Hub and inserted into the Telemetry table of the SQL database. Briefly the code above works as follows:

Line starting	Function
`string cs =`	Identifies the SQL database **iot**, getting it from upon the variable **ConnectionString** noted in Section 3.4 above.
`Message m =`	Identifies the IoT Hub **centralhub**, getting it from **function.json**
`var insertScript =`	Writes a record to the SQL database, mapping the attributes in the records arriving at the IoT Hub to the fields in the SQL table
`public class Message{`	Identifies the attributes of the record arriving at the Event Hub

7.4 Viewing data received by SQL Server

To verify that the function is working correctly launch SQL Server Management Studio on your desktop, connect to **iothome,** right click on the **iot** database**,** and select **New Query**. Enter and execute the following query to see the data pushed to SQL Server:

```
SELECT * FROM [dbo].[Telemetry] order by TimeStamp desc
```

The results in SSMS:

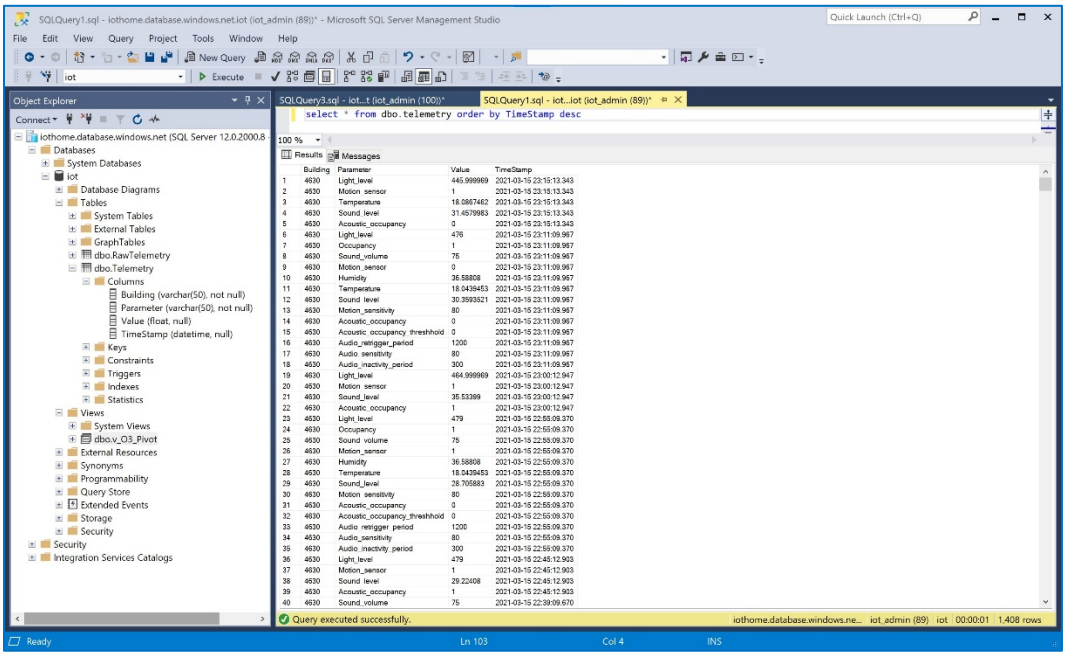

Figure 22: SQL Server Telemetry listing

Note that each record is from a single message from the O3, for example a record for Occupant_Temperature, a record for Light_level. We can also display all the records at a specific time by executing the SQL view we created earlier. Right click on the **iot** database and select **New Query**. Enter and execute the following query to see the data in the view:

```
SELECT * FROM [dbo].[v_O3_Pivot] order by TimeStamp desc
```

The results in SSMS:

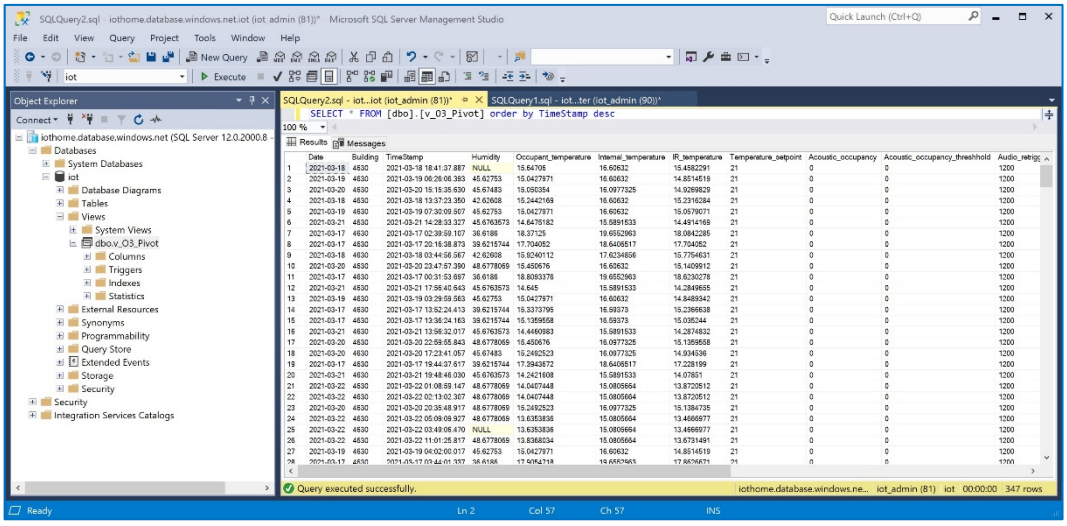

Figure 23: SQL view

Note here that all the values collected by the O3 at a specific time are stored in a single record. This will make it easier to use analysis tools to display the data.

8 Alternative: push data from Event Hub to Azure Table Storage

If you do not need the functionality and power of SQL Server, a cost-effective alternative is to push the data to Azure Table Storage. To do this, we use a function like that used above. Follow the steps in Sections 7.1 and 7.2 above, and then continue as follows.

8.1 Creating the Function

Next, we create a function. When the deployment is complete, select **Go To Resource**. From the left menu select **Functions**, then select **+ Add** from the top menu. In the **Add Function** window, set the parameters as follows:

Setting	Value
Develop environment	Select **Develop in portal.**
Template	Select **Azure Event Hub trigger.**
New Function	Enter **EventHubToTable**
Event Hub connection	Select **centralhub_events_IOTHUB**
Event Hub consumer group	Select **totablestorage**

Click **Add** to create the function. Once created, click on **EventHubToTable** in the list on the right to open the function page. Click **Integration**, to bring up the wire frame:

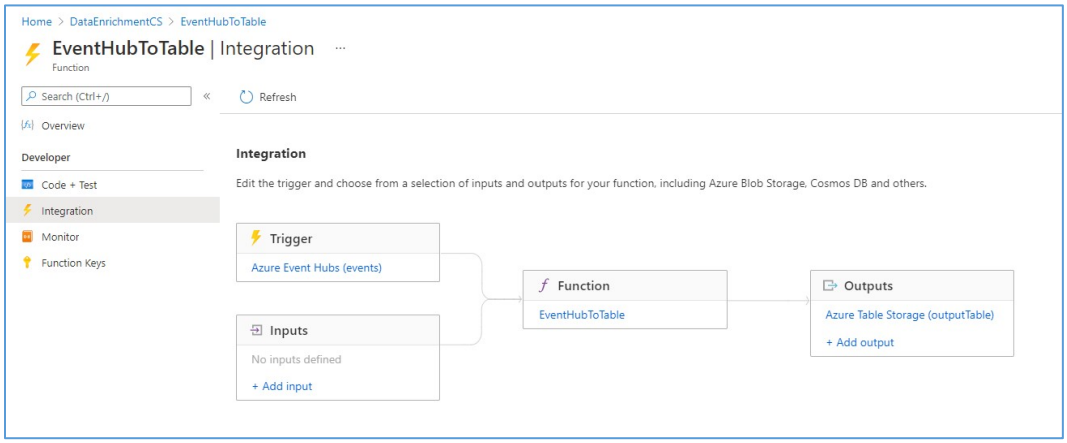

Figure 24: Integration

Click on **+ Add output**, and enter the following values:

Setting	Value
Binding Type	Select **Azure Table Storage.**
Table parameter name	Enter **outputTable.**
Table name	Enter **Telemetry.**
Storage account connection	Select **storageaccountiotpr96cc_STORAGE**

The **Edit Output** box should look like this:

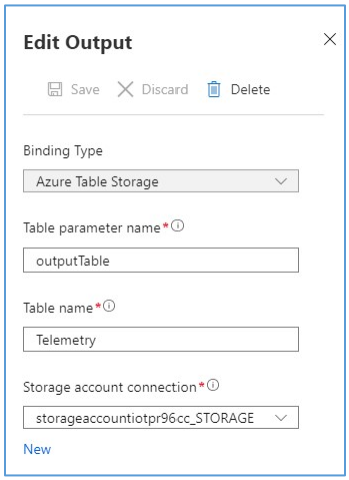

Figure 25: Edit output

Click **Save** to finish the configuration. Next, Click **Code + Test** in the left menu and select **function.json** in the drop-down at the top. The JSON should contain the information from the **Create Function** wizard and the **Create Output** wizard:

```
{
  "bindings": [
    {
      "type": "eventHubTrigger",
      "name": "events",
      "direction": "in",
      "eventHubName": "iotworx",
      "cardinality": "many",
      "connection": "centralhubs_RootManageSharedAccessKey_EVENTHUB3",
      "consumerGroup": "totablestorage"
    },
    {
      "name": "outputTable",
      "direction": "out",
      "type": "table",
      "tableName": "Telemetry",
      "connection": "storageaccountiotpr96cc_STORAGE"
    }
  ]
}
```

Figure 26: Function.JSON for EventHubToTable function

Note that the input ("direction": "in") specifies **iotworx,** the Event Hub from which the function will read data and the output ("direction": "out") specifies **Telemetry**, the storage table to which the function will write the data.

Next, select **run.csx** in the drop-down at the top, and replace the code in the window with the following and click **Save** (formatting below modified to fit to page):

```
#r "Microsoft.Azure.EventHubs"
#r "Newtonsoft.Json"
using System;
using System.Text;
using Microsoft.Azure.EventHubs;
using Newtonsoft.Json;
public static async Task Run(EventData[] events, ICollector<TelemetryItem> outputTable, ILogger log)
{
    var exceptions = new List<Exception>();
    foreach (EventData eventData in events)
    {
        try
        {
            string messageBody = Encoding.UTF8.GetString(eventData.Body.Array, eventData.Body.Offset,
eventData.Body.Count);
            Message m = JsonConvert.DeserializeObject<Message>(messageBody);

            log.LogInformation($"C# Event Hub trigger function processed a message: {messageBody}");

            DateTimeOffset offsetDate = new DateTimeOffset(m.t);
            long unixTimeStamp = offsetDate.ToUnixTimeSeconds();
            outputTable.Add(
                new TelemetryItem(){
                    PartitionKey = $"PugetSound-WestCampus-SpyrosLab-{m.id}",
                    RowKey = unixTimeStamp.ToString(),
                    id = m.id,
                    v = m.v,
                    t = m.t
                }
            );
            await Task.Yield();
        }
        catch (Exception e)
        {
            // We need to keep processing the rest of the batch - capture this exception and continue.
            // Also, consider capturing details of the message that failed processing so it can be processed
            // again later.
            exceptions.Add(e);
        }
    }
```

(Continued on next page)

```
    // Once processing of the batch is complete, if any messages in the batch failed processing throw an
    // exception so that there is a record of the failure.

    if (exceptions.Count > 1)
        throw new AggregateException(exceptions);

    if (exceptions.Count == 1)
        throw exceptions.Single();
}
public class Message{
    public string id {get;set;}
    public double v {get;set;}
    public DateTime t {get; set;}
}

public class TelemetryItem : Message{
    public string PartitionKey {get; set;}
    public string RowKey {get; set;}
}
```

8.2 Viewing data received by Azure Table Storage

To verify that the function is working correctly, from the Azure portal select **storageaccountiotpr96cc**, then from the left menu select **Storage Explorer (preview) → TABLES → Telemetry**. This should show data in the **Telemetry** table specified in **Function.json:**

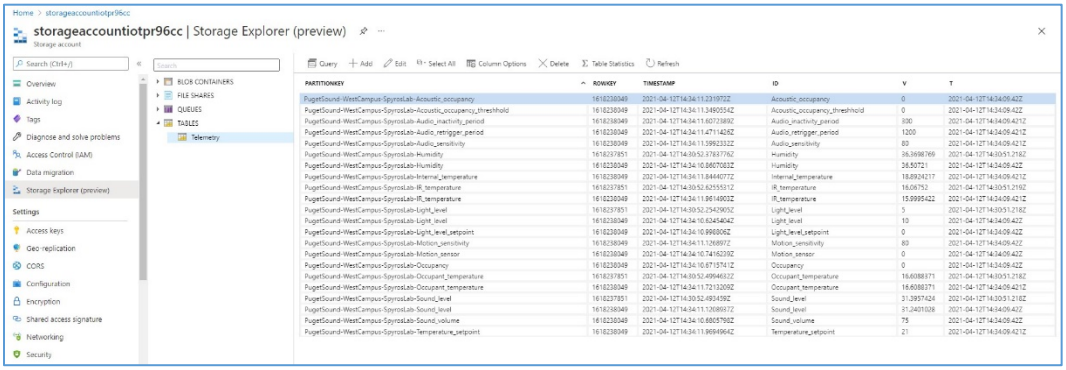

Figure 27: Storage Explorer showing data in Telemetry table

If you have applications which can access Azure Table Storage and you do not need the functionality and scale provided by SQL Server, this is a more cost-efficient method to capture the data.

9 Creating a Power BI application to display the data

Once you have the data in SQL or in Table Storage, you can build an Azure dashboard to display the data in real time. It is beyond the scope of this paper to describe this in detail, but the basic steps on one way to do this are as described below. You may need assistance from an IT/ICT professional familiar with SQL and Power BI to complete these steps.

Here is an example of a Power BI report pulling the data from the SQL table:

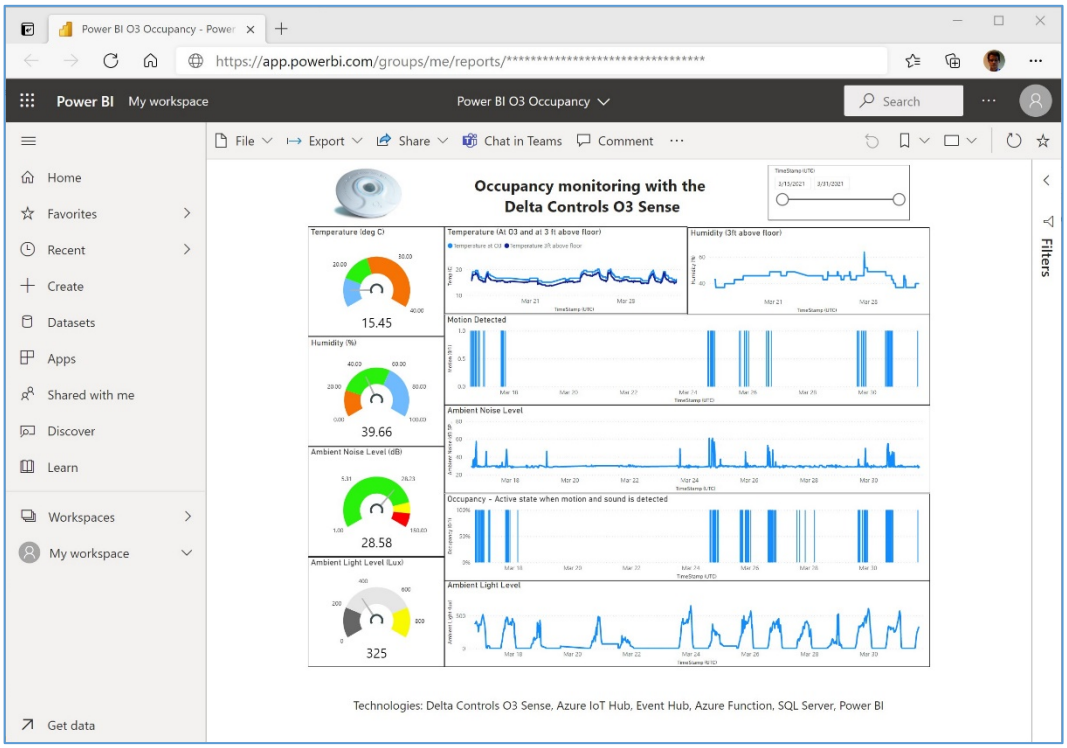

Figure 28 PowerBI.com report

To create such a dashboard, you would use the free Power BI desktop application, and do the following:

1. Specify the connection string for the SQL database and table, noted in Section 3.4 above.
2. Specify the query against the database.
3. Specify the chart type (on the right in the image we have some Line Charts, on the left some examples of a third-party Power BI gauge widget downloaded from the store).
4. Specify the axes.
5. Add any text or JPG.

6. Publish the chart to http://powerbi.com.
7. Share the workspace to authorized users.

There are multiple tutorials on the Internet on setting up Power BI dashboards and reports, which you can find easily with a simple search.

10 Using GENESIS64 as a no code client

ICONICS GENESIS64 can be used as a no code client to the published O3 Sense data. To do so, the following configuration needs to be setup in an Azure virtual machine with GENESIS64 installed.

Start by deploying the latest version of ICONICS Suite VM offer from the Azure Marketplace.
As of this writing, the latest version of ICONICS Suite is version 10.97, available here:
https://azuremarketplace.microsoft.com/en-us/marketplace/apps/iconics.iconics-suite-1097?tab=Overview

10.1 Create a custom encoder

To decode the published O3 data, we must first setup the custom encoder that instructs GENESIS64 how to understand the published data.

To set this up, in **Workbench → Internet of Things**, right click on **Custom Encoders/Decoders**, choose **Add Encoder/Decoder,** and create a custom encoder like in Section 5.4 above, name it as "Delta O3 Encoder" and define the Value Format as follows:

```
{
  "id": "%PUBLISHNAME%",
  "v": "%VALUE%",
  "q": "%STATUS.GOOD%",
  "t": "%NOWUTC.TEXT%"
}
```

When this step is complete, your configuration in Workbench should look like this:

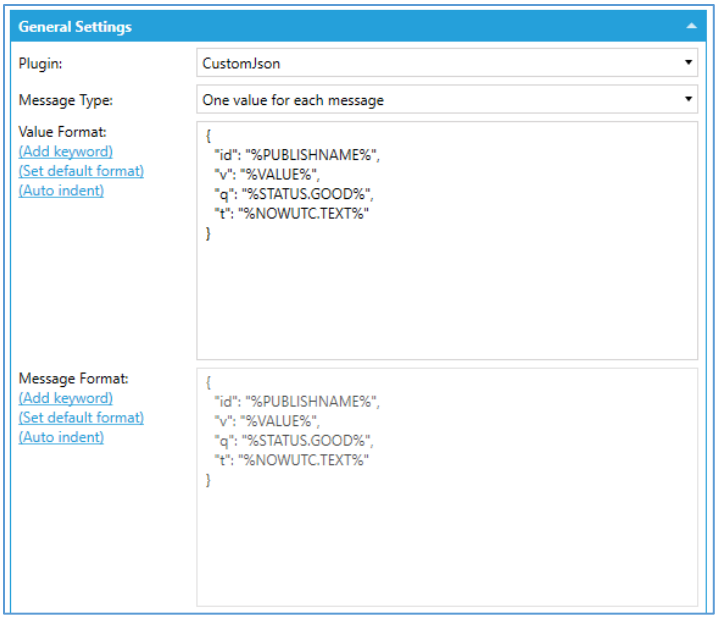

Figure 29: Encoder

10.2 Create a subscriber connection

To start receiving published data from the O3, we have first to subscribe to the IoT Hub with a subscriber connection.

To set this up, in **Workbench → Internet of Things**, right click on **Subscriber Connections**, choose **Add Subscriber Connection,** and give the subscriber connection a name, for example **Delta_O3_Hub**.

Set up the general settings of the subscriber connection as shown below:

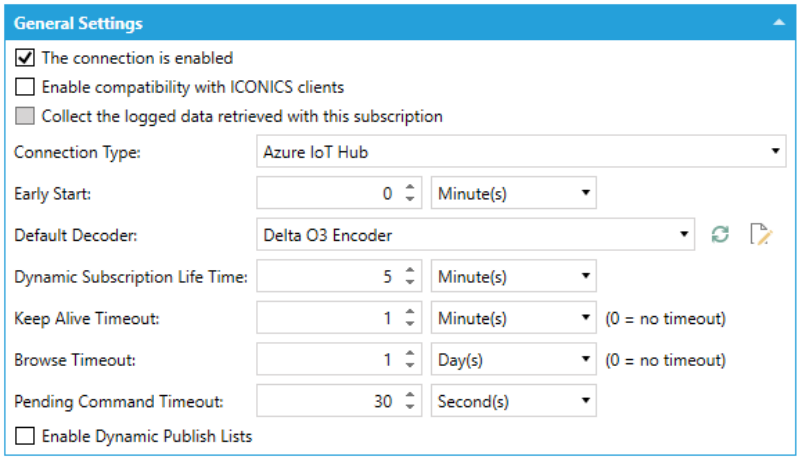

Figure 10 Subscriber Connection General Settings

Leave the Datasets Support section with default values.

In the IoT Hub Settings section, enter the appropriate connection strings from section 3.2. Click Apply to save the configuration and start the Subscriber service.

10.3 Visualize and interact with published data

To visualize data from the subscribed IoT Hub, start the Data Explorer application and browse under the My Computer → Internet of Things branch like that shown in Figure 12:

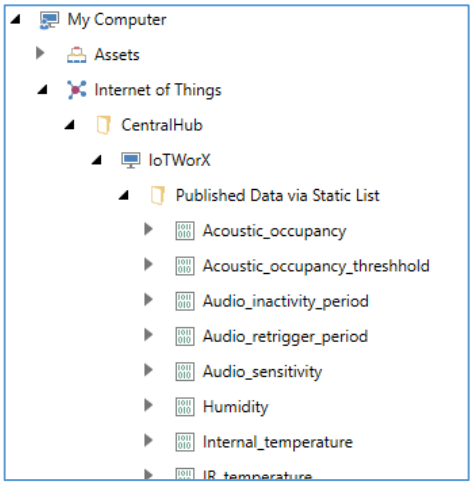

Figure 12 Browsing for published data

Select the desired published data point and you should see the data values in a table to the right like that shown in Figure 13:

Name	Value	Timestamp	Quality
Acoustic_occupancy	0	3/29/2021 2:05 PM	Good
Acoustic_occupancy_threshhold	0	3/29/2021 2:05 PM	Good
Audio_inactivity_period	300	3/29/2021 2:05 PM	Good
Audio_retrigger_period	1200	3/29/2021 2:05 PM	Good
Audio_sensitivity	80	3/29/2021 2:05 PM	Good
Humidity	42.74052	3/29/2021 2:05 PM	Good
Internal_temperature	17.678875	3/29/2021 2:05 PM	Good
IR_temperature	16.4829483	3/29/2021 2:05 PM	Good
Light_level	299	3/29/2021 2:05 PM	Good
Light_level_setpoint	0	3/29/2021 2:05 PM	Good
Motion_sensitivity	80	3/29/2021 2:05 PM	Good
Motion_sensor	0	3/29/2021 2:05 PM	Good
Occupancy	0	3/29/2021 2:05 PM	Good
Occupant_temperature	16.510643	3/29/2021 2:05 PM	Good
Sound_level	29.33028	3/29/2021 2:05 PM	Good
Sound_volume	75	3/29/2021 2:05 PM	Good
Temperature_setpoint	21	3/29/2021 2:05 PM	Good

Figure 13 Published data values

The frequency of data updates and availability will be dependent on the publish rate set on the IoTWorX gateway.

10.4 Organizing data with ICONICS AssetWorX

AssetWorX is a digital twins module in the ICONICS Suite. Data received from IoTWorX can easily be organized into a logical structure and extended with history, alarms, and faults.

By leveraging features like equipment classes, a template for the Delta O3 can be defined and used to deploy multiple instances of the device in bulk.

Learn about AssetWorX on the ICONICS Institute here: https://iconics.com/Resources/ICONICS-Institute/Units/Asset-Organization

10.5 Create an IoT dashboard

An engaging and dynamic IoT dashboard can easily be created with ICONICS' visualization capabilities. Here is an example of such a dashboard:

Figure 30 ICONICS IoT Dashboard

11 Next steps

If you have successfully completed the above steps you have an end-to-end example of remotely monitoring the Delta O3 Sense. With the available data, more sophisticated environment monitoring solutions can be built.