Microsoft Cloud Workshop

Internet of Things
Hands-on lab step-by-step
June 2020

Contents

Internet of Things hands-on lab step-by-step

If you have not yet completed the steps to set up your environment in Before the hands-on lab setup guide, you will need to do that before proceeding.

Abstract and learning objectives

In this hands-on lab, you will construct an end-to-end IoT solution simulating high velocity data emitted from smart meters and analyzed in Azure. You will design a lambda architecture, filtering a subset of the telemetry data for real-time visualization on the hot path, and storing all the data in long-term storage for the cold path.

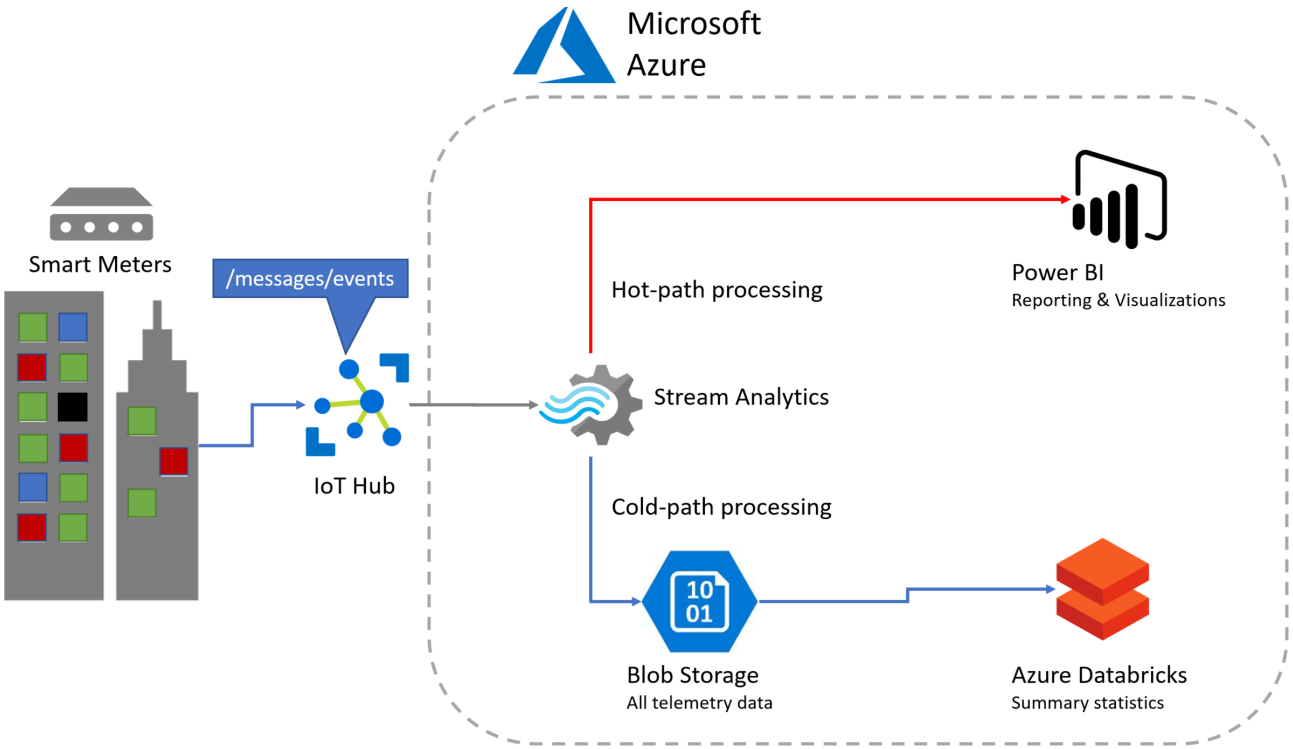
At the end of this hands-on lab, you will be better able to build an IoT solution implementing device registration with the IoT Hub Device Provisioning Service and visualizing hot data with Power BI.

Overview

Fabrikam provides services and smart meters for enterprise energy (electrical power) management. Their **You-Left-The-Light-On** service enables the enterprise to understand their energy consumption.

Solution architecture

Below is a diagram of the solution architecture you will build in this lab. Please study this carefully, so you understand the whole of the solution as you are working on the various components.



Messages are ingested from the Smart Meters via IoT Hub and temporarily stored there. A Stream Analytics job pulls telemetry messages from IoT Hub and sends the messages to two different destinations. There are two Stream Analytics jobs, one that retrieves all messages and sends them to Blob Storage (the cold path), and another that selects out only the important events needed for reporting in real time (the hot path). Data entering the hot path will be reported on using Power BI visualizations and reports. For the cold path, Azure Databricks can be used to apply the batch computation needed for the reports at scale.

Other alternatives for processing of the ingested telemetry would be to use an HDInsight Storm cluster, a WebJob running the EventProcessorHost in place of Stream Analytics, or HDInsight running with Spark streaming. Depending on the type of message filtering being conducted for hot and cold stream separation, IoT Hub Message Routing might also be used, but this has the limitation that messages follow a single path, so with the current implementation, it would not be possible to send all messages to the cold path, while simultaneously sending some of the same messages into the hot path. An important limitation to keep in mind for Stream Analytics is that it is very restrictive on the format of the input data it can process: the payload must be UTF8 encoded JSON, UTF8 encoded CSV (fields delimited by commas, spaces, tabs, or vertical pipes), or AVRO, and it must be well-formed. If any devices transmitting telemetry cannot generate output in these formats (e.g., because they are legacy devices), or their output can be not well formed at times, then alternatives that can better deal with these situations should be investigated. Additionally, any custom code or logic cannot be embedded with Stream Analytics---if greater extensibility is required, the alternatives should be considered.

> **Note**: The preferred solution is only one of many possible, viable approaches.

Requirements

- Microsoft Azure subscription must be pay-as-you-go or MSDN.
  - Trial subscriptions will not work.
- A virtual machine configured with:
  - Visual Studio Community 2019 or later
  - Azure SDK 2.9 or later (Included with Visual Studio)
- A running Azure Databricks cluster (see Before the hands-on lab)

Exercise 1: IoT Hub provisioning

Duration: 15 minutes

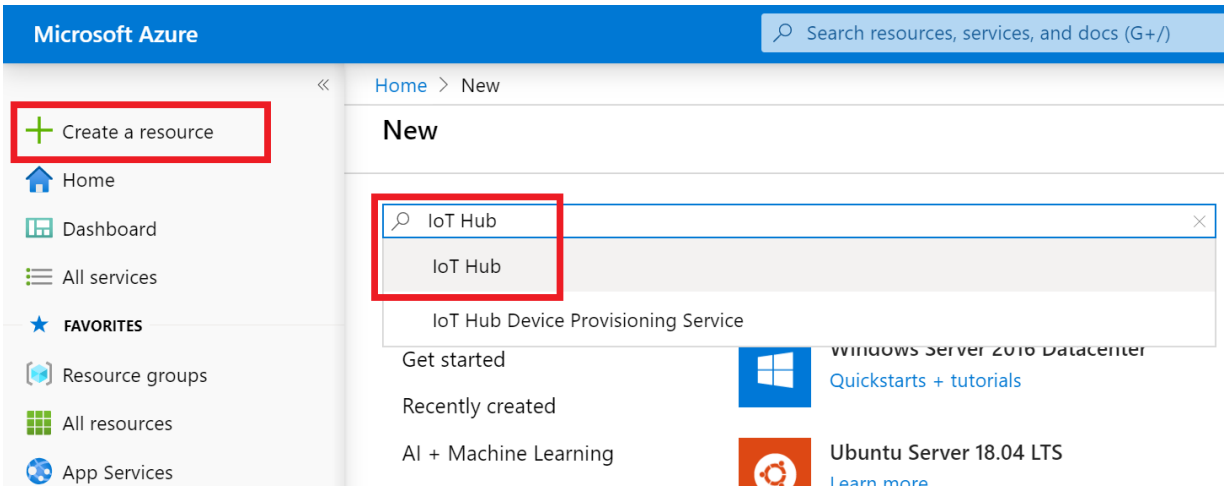
In your architecture design session with Fabrikam, it was agreed that you would use an Azure IoT Hub to manage both the device registration and telemetry ingest from the Smart Meter Simulator. Your team also identified the Microsoft provided Device Explorer project that Fabrikam can use to view the list and status of devices in the IoT Hub registry.

Task 1: Provision IoT Hub

In these steps, you will provision an instance of IoT Hub.

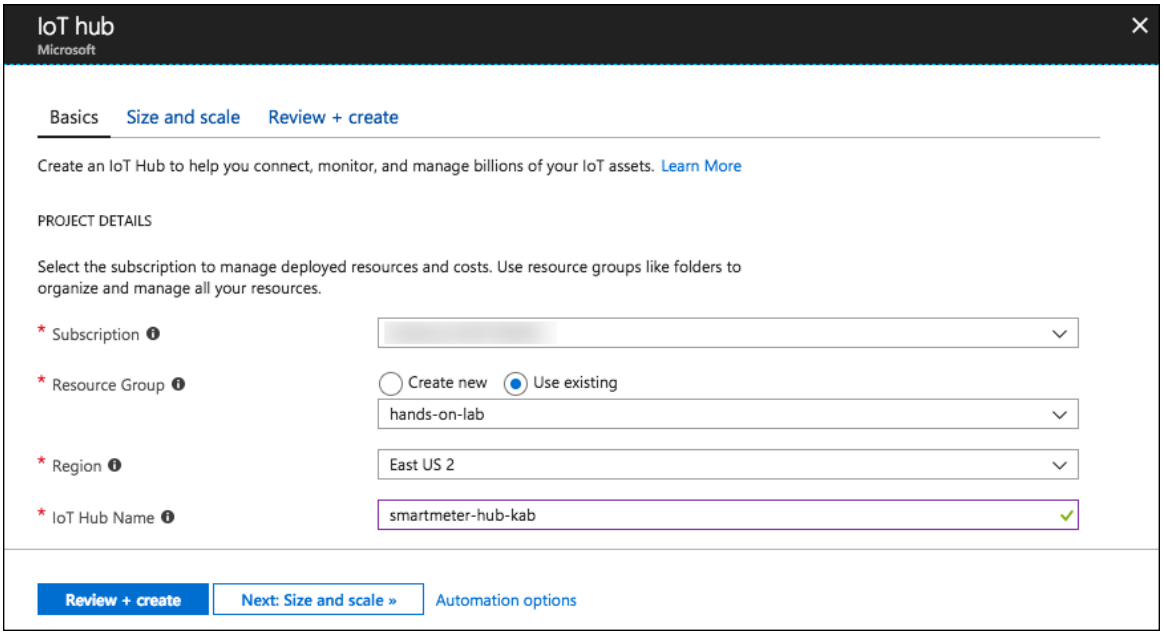
1. In your browser, navigate to the Azure portal, select **+Create a resource** in the navigation pane, enter `iot` into the **Search the Marketplace** box.

2. Select **IoT Hub** from the results, and then select **Create**.



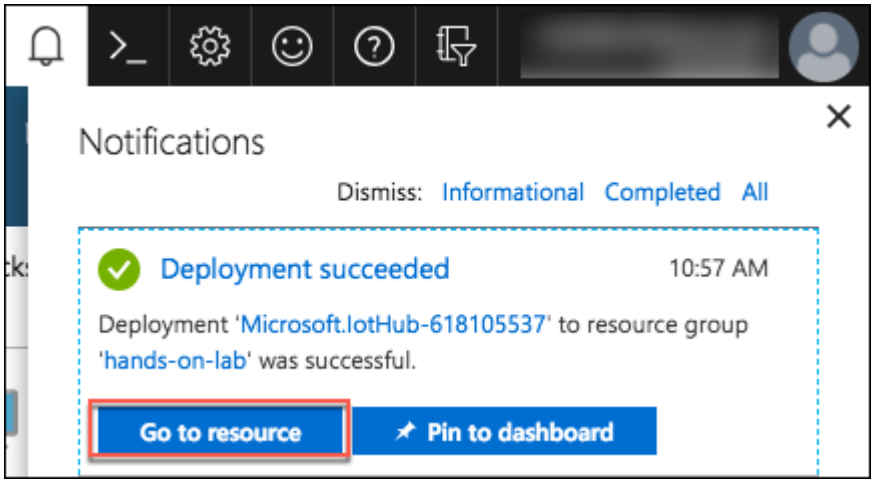
3. On the **IoT Hub** blade **Basics** tab, enter the following:

   - **Subscription**: Select the subscription you are using for this hands-on lab.
   - **Resource group**: Choose Use existing and select the **hands-on-lab-SUFFIX** resource group.
   - **Region**: Select the location you are using for this hands-on lab.
   - **IoT Hub Name**: Enter a unique name, such as `smartmeter-hub-SUFFIX`.

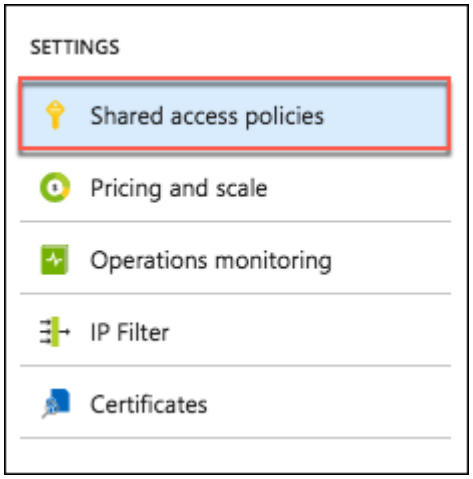


- Select **Next: Size and Scale**.
- On the **Size and scale** tab, accept the default Pricing and scale tier of **S1: Standard tier**, and select **Review + create**.
- Select **Create** on the **Review + create** blade.

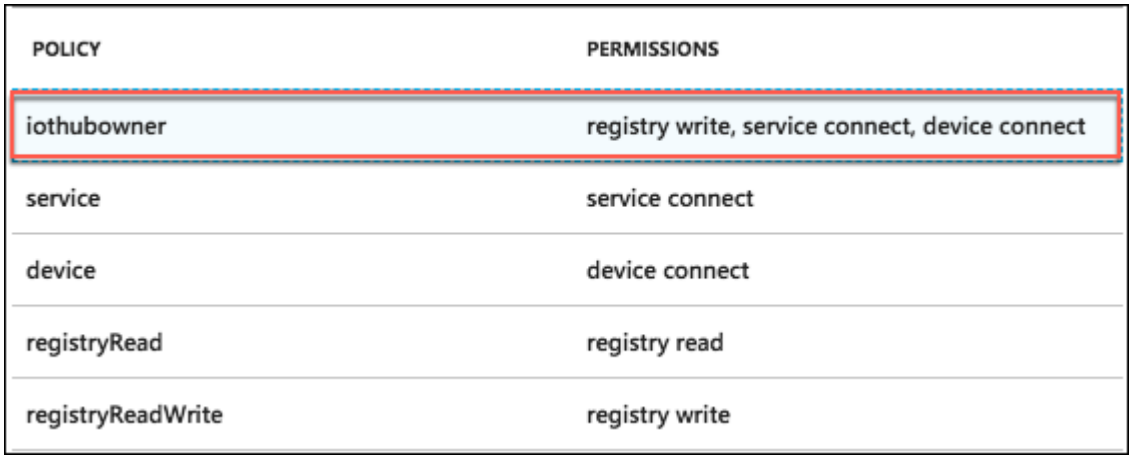
4. When the IoT Hub deployment is completed, you will receive a notification in the Azure portal. Select **Go to resource** in the notification.



5. From the **IoT Hub's Overview** blade, select **Shared access policies** under **Settings** on the left-hand menu.

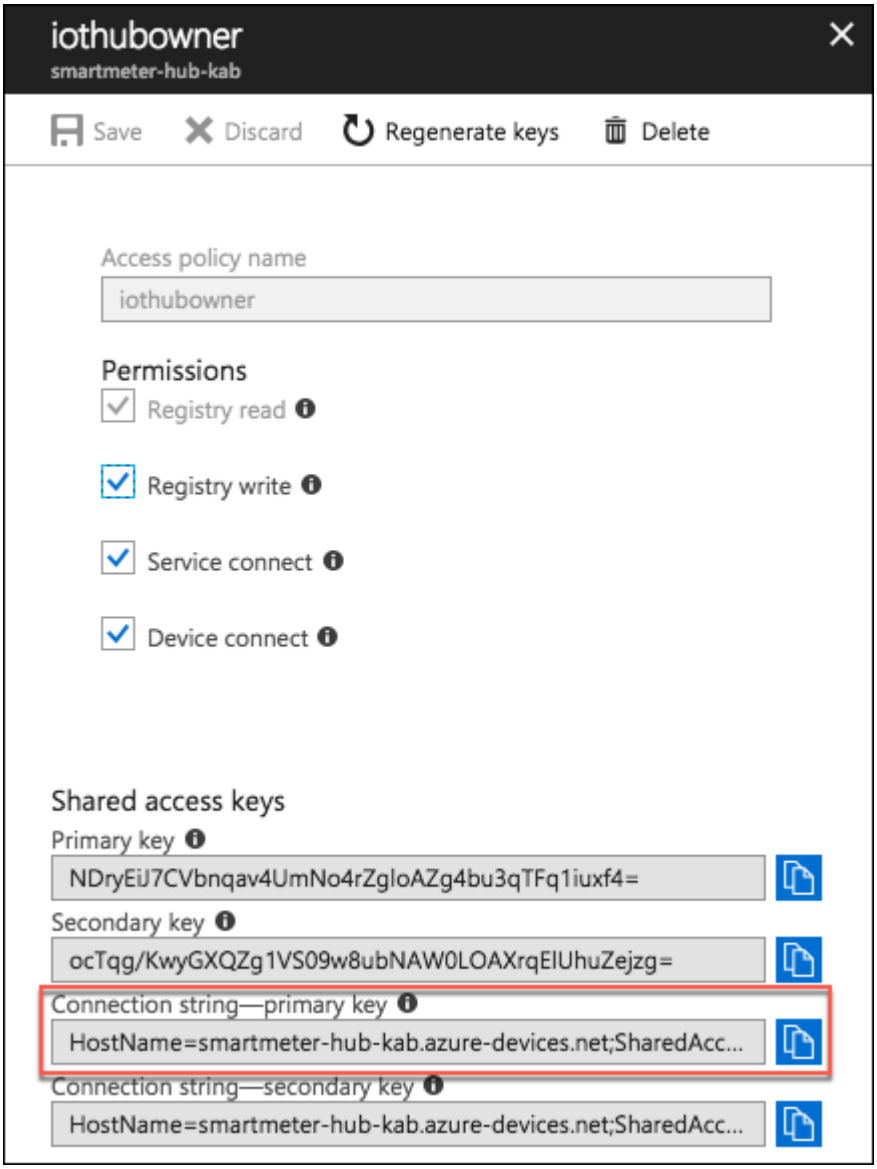


6. Select **iothubowner** policy.



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

7. In the **iothubowner** blade, select the **Copy** button to the right of the **Connection string - primary key** field. You will need this connection string value in the next exercise.



Exercise 2: Completing the Smart Meter Simulator

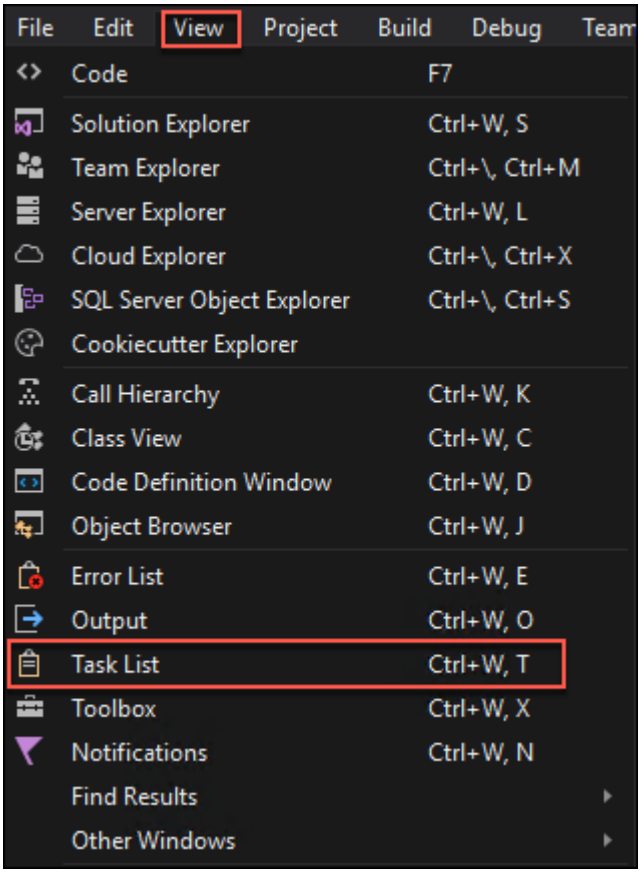
Duration: 60 minutes

Fabrikam has left you a partially completed sample in the form of the Smart Meter Simulator solution. You will need to complete the missing lines of code that deal with device registration management and device telemetry transmission that communicate with your IoT Hub.

Task 1: Implement device management with the IoT Hub

1. In **Visual Studio** on your **Lab VM**, use **Solution Explorer** to open the file `DeviceManager.cs`.

2. From the Visual Studio **View** menu, choose **Task List**.



3. In the **Task List**, you will see a list of **TODO** tasks, where each task represents one line of code that needs to be completed. Complete the line of code below each **TODO** using the code below as a reference.

4. The following code represents the completed tasks in **DeviceManager.cs**:

```
class DeviceManager
{
    static string connectionString;
    static RegistryManager registryManager;

    public static string HostName { get; set; }

    public static void IotHubConnect(string cnString)
    {
        connectionString = cnString;

        //TODO: 1.Create an instance of RegistryManager from connectionString
        registryManager = RegistryManager.CreateFromConnectionString(connectionS

        var builder = IotHubConnectionStringBuilder.Create(cnString);

        HostName = builder.HostName;
    }

    /// <summary>
```

```
        /// Register a single device with the IoT hub. The device is initially regis
        /// disabled state.
        /// </summary>
        /// <param name="connectionString"></param>
        /// <param name="deviceId"></param>
        /// <returns></returns>
        public async static Task<string> RegisterDevicesAsync(string connectionStrir
        {
            //Make sure we're connected
            if (registryManager == null)
                IotHubConnect(connectionString);

            //TODO: 2.Create new device
            Device device = new Device(deviceId);

            //TODO: 3.Initialize device with a status of Disabled
            //Enabled in a subsequent step
            device.Status = DeviceStatus.Disabled;

            try
            {
                //TODO: 4.Register the new device
                device = await registryManager.AddDeviceAsync(device);
            }
            catch (Exception ex)
            {
                if (ex is Microsoft.Azure.Devices.Common.Exceptions.DeviceAlreadyExi
                    ex.Message.Contains("DeviceAlreadyExists"))
                {
                    //TODO: 5.Device already exists, get the registered device
                    device = await registryManager.GetDeviceAsync(deviceId);

                    //TODO: 6.Ensure the device is disabled until Activated later
                    device.Status = DeviceStatus.Disabled;

                    //TODO: 7.Update IoT Hubs with the device status change
                    await registryManager.UpdateDeviceAsync(device);
                }
                else
                {
                    MessageBox.Show($"An error occurred while registering one or mor
                }
            }

            //return the device key
            return device.Authentication.SymmetricKey.PrimaryKey;
        }

        /// <summary>
        /// Activate an already registered device by changing its status to Enabled.
        /// </summary>
```

```
/// <param name="connectionString"></param>
/// <param name="deviceId"></param>
/// <param name="deviceKey"></param>
/// <returns></returns>
public async static Task<bool> ActivateDeviceAsync(string connectionString,
{
    //Server-side management function to enable the provisioned device
    //to connect to IoT Hub after it has been installed locally.
    //If device id and device key are valid, Activate (enable) the device.

    //Make sure we're connected
    if (registryManager == null)
        IotHubConnect(connectionString);

    bool success = false;
    Device device;

    try
    {
        //TODO: 8.Fetch the device
        device = await registryManager.GetDeviceAsync(deviceId);

        //TODO: 9.Verify the device keys match
        if (deviceKey == device.Authentication.SymmetricKey.PrimaryKey)
        {
            //TODO: 10.Enable the device
            device.Status = DeviceStatus.Enabled;

            //TODO: 11.Update IoT Hubs
            await registryManager.UpdateDeviceAsync(device);

            success = true;
        }
    }
    catch(Exception)
    {
        success = false;
    }

    return success;
}

/// <summary>
/// Deactivate a single device in the IoT Hub registry.
/// </summary>
/// <param name="connectionString"></param>
/// <param name="deviceId"></param>
/// <returns></returns>
public async static Task<bool> DeactivateDeviceAsync(string connectionStrin
{
    //Make sure we're connected
```

```
        if (registryManager == null)
            IotHubConnect(connectionString);

        bool success = false;
        Device device;

        try
        {
            //TODO: 12.Lookup the device from the registry by deviceId
            device = await registryManager.GetDeviceAsync(deviceId);

            //TODO: 13.Disable the device
            device.Status = DeviceStatus.Disabled;

            //TODO: 14.Update the registry
            await registryManager.UpdateDeviceAsync(device);

            success = true;
        }
        catch (Exception)
        {
            success = false;
        }

        return success;
    }

    /// <summary>
    /// Unregister a single device from the IoT Hub Registry
    /// </summary>
    /// <param name="connectionString"></param>
    /// <param name="deviceId"></param>
    /// <returns></returns>
    public async static Task UnregisterDevicesAsync(string connectionString, str
    {
        //Make sure we're connected
        if (registryManager == null)
            IotHubConnect(connectionString);

            //TODO: 15.Remove the device from the Registry
            await registryManager.RemoveDeviceAsync(deviceId);
    }

    /// <summary>
    /// Unregister all the devices managed by the Smart Meter Simulator
    /// </summary>
    /// <param name="connectionString"></param>
    /// <returns></returns>
    public async static Task UnregisterAllDevicesAsync(string connectionString)
    {
        //Make sure we're connected
```

```
            if (registryManager == null)
               IotHubConnect(connectionString);

            for(int i = 0; i <= 9; i++)
            {
                string deviceId = "Device" + i.ToString();

                //TODO: 16.Remove the device from the Registry
                await registryManager.RemoveDeviceAsync(deviceId);
            }
        }
    }
```

> **Note**: Be sure you only replace code in the **DeviceManager** class and not any other code in the file.

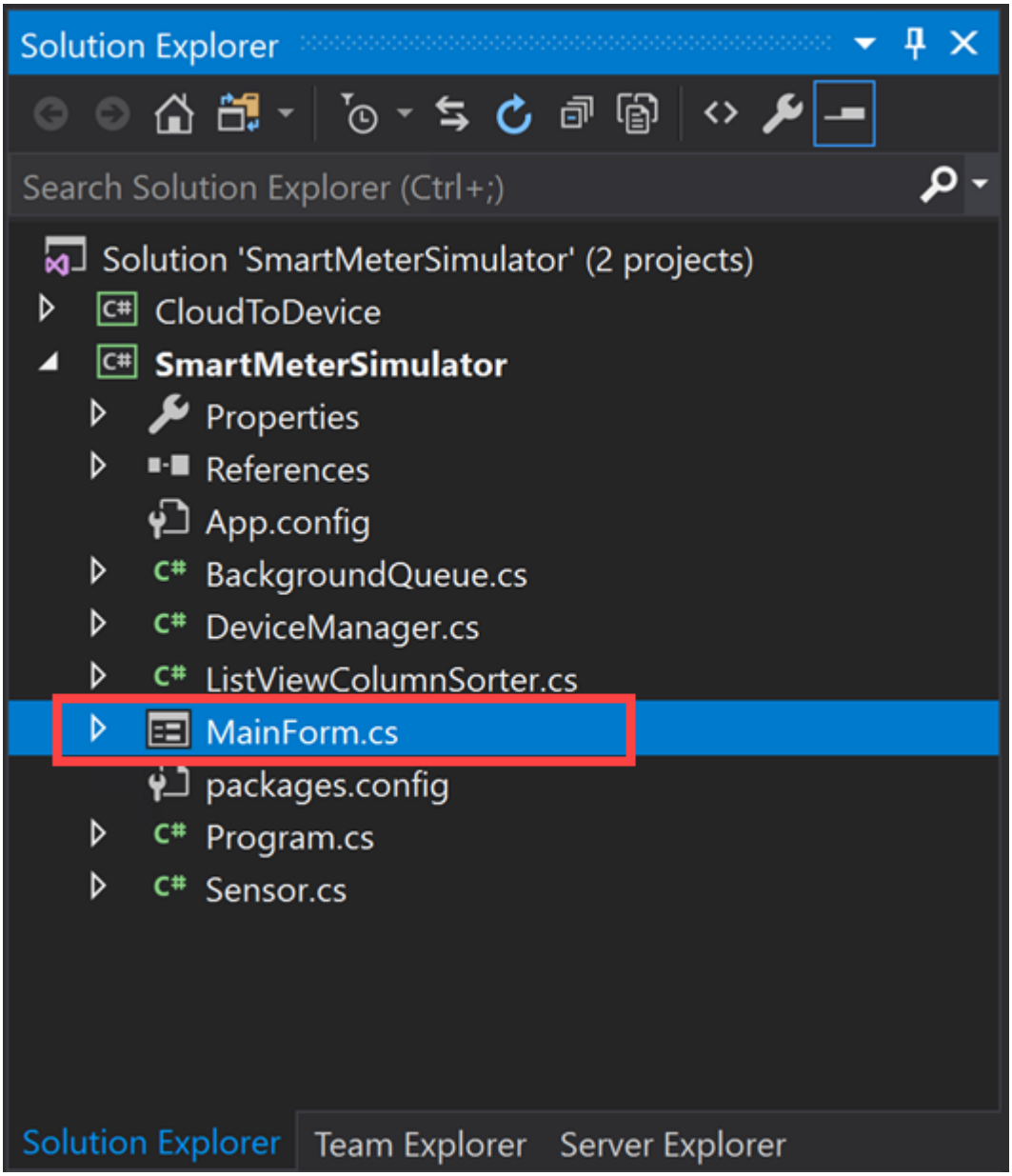
5. Save **DeviceManager.cs**.

Task 2: Configure the IoT Hub connection string

You will want to avoid entering the IoT Hub connection string every time the project is run. To do this, you can set this value as the default text for the connection string text box in the application. Follow these steps to configure the connection string:

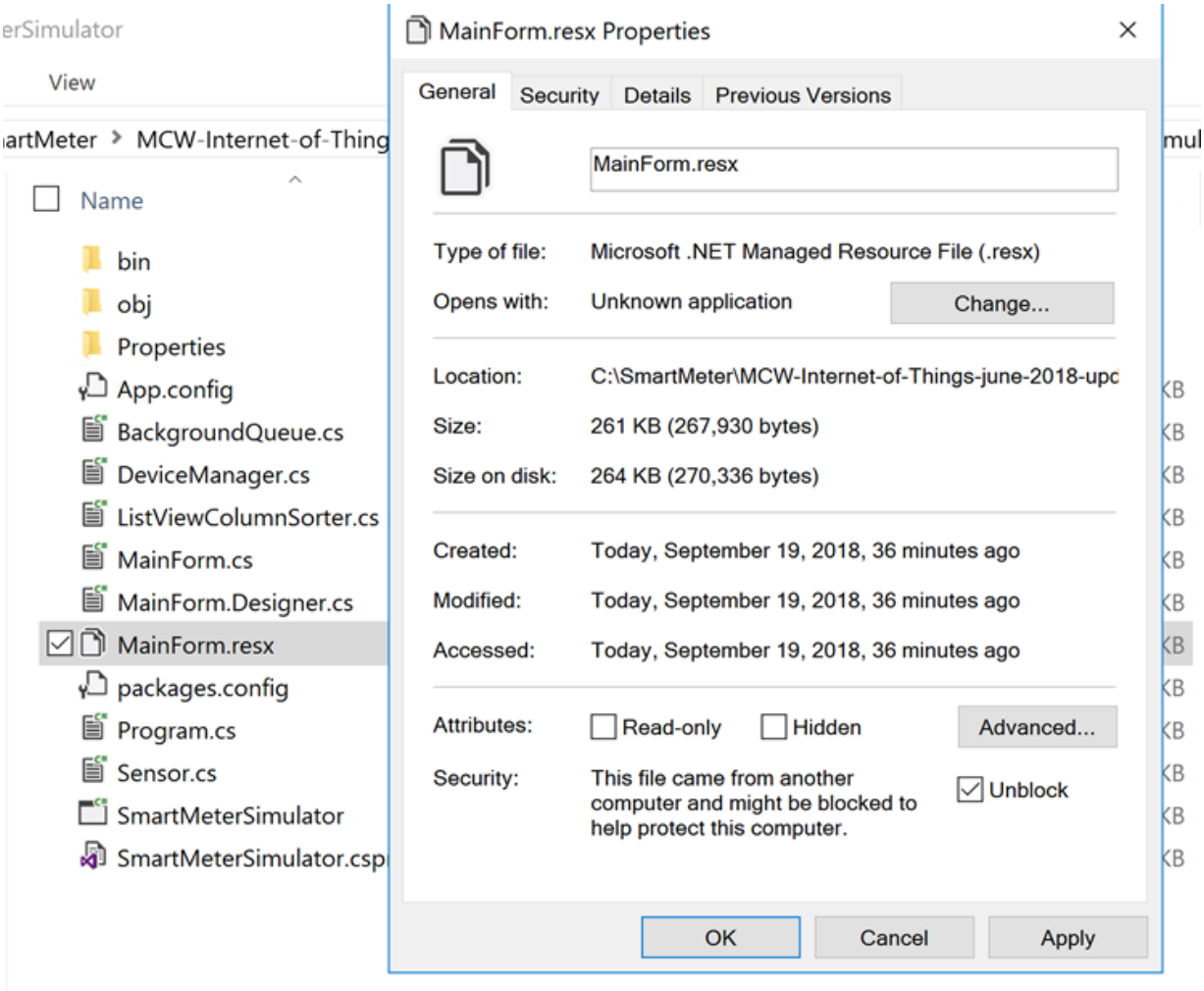
1. Return to the **SmartMeterSimulator** solution in **Visual Studio** on your **Lab VM**.

2. In the **Solution Explorer**, expand the **SmartMeterSimulator** project and double-click **MainForm.cs** to open it. (If the Solution Explorer is not in the upper-right corner of your Visual Studio instance, you can find it under the View menu in Visual Studio.)



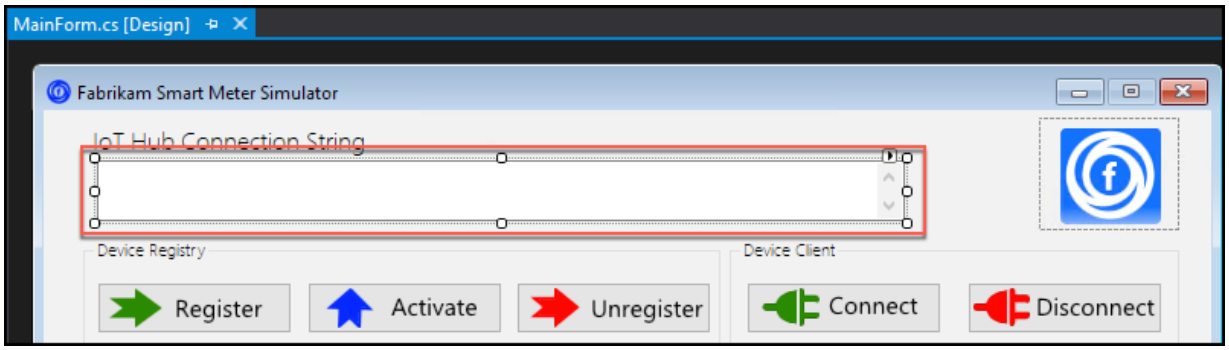
> **Note**: If the file does not open. One of the project files may be blocked.

- Open **Windows Explorer** and navigate to the starter project folder: **C:\SmartMeter\Hands-on lab\lab-files\starter-project\SmartMeterSimulator**.
- Right-click on the **MainForm.resx** file, then select **Properties**.
- Check the **Unblock** checkbox on the bottom of the **General** tab, then select **Apply** then **OK**.

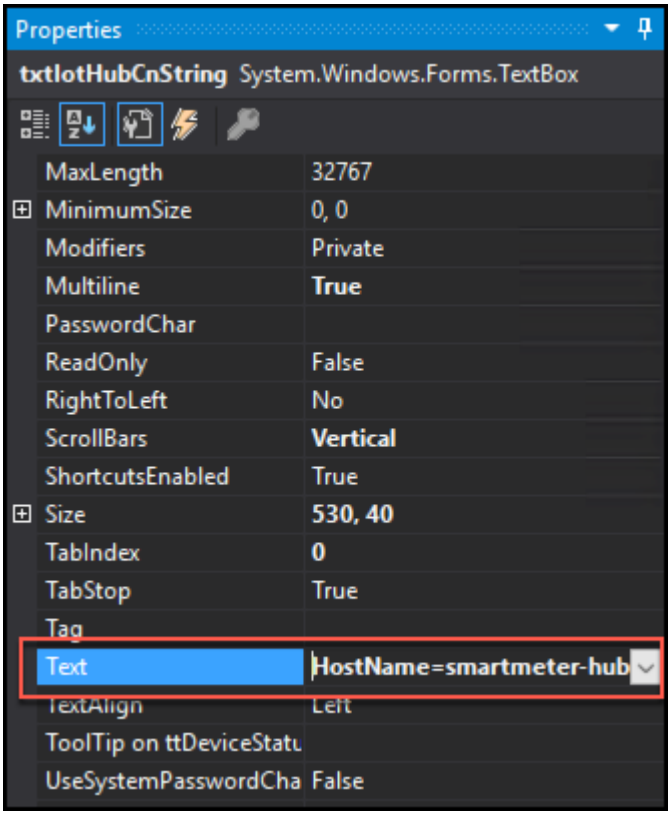


- Close and reopen **Visual Studio**. Re-open the **MainForm.cs** file.

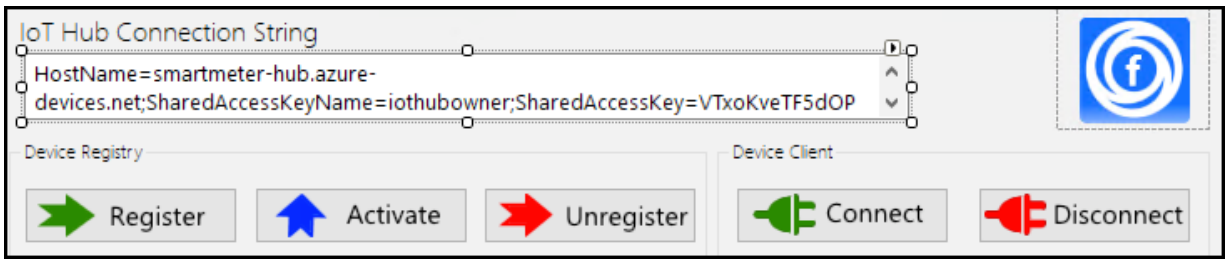
3. In the **Windows Forms designer surface**, select the **IoT Hub Connection String TextBox**.



4. In the **Properties** panel, scroll until you see the **Text** property. Paste your **IoT Hub connection string** value copied from Exercise 1, Task 1, Step 7 of the previous exercise into the value for the **Text** property. (If the properties window is not visible below the Solution Explorer, right-click the TextBox, and select **Properties**.)



5. Your connection string should now be present every time you run the **Smart Meter Simulator**.



6. Save **MainForm.cs**.

Task 3: Implement the communication of telemetry with IoT Hub

1. Open **Sensor.cs** from the **Solution Explorer**, and complete the **TODO** items indicated within the code that are responsible for transmitting telemetry data to the IoT Hub, as well as receiving data from IoT Hub.

2. The following code shows the completed result:

```
class Sensor
 {
     private DeviceClient _DeviceClient;
     private string _IotHubUri { get; set; }
     public string DeviceId { get; set; }
```

```
public string DeviceKey { get; set; }
public DeviceState State { get; set; }
public string StatusWindow { get; set; }
public string ReceivedMessage { get; set; }
public double? ReceivedTemperatureSetting { get; set; }
public double CurrentTemperature
{
    get
    {
        double avgTemperature = 70;
        Random rand = new Random();
        double currentTemperature = avgTemperature + rand.Next(-6, 6);

        if (ReceivedTemperatureSetting.HasValue)
        {
            // If we received a cloud-to-device message that sets the tempe
            currentTemperature = ReceivedTemperatureSetting.Value;
        }

        if(currentTemperature <= 68)
            TemperatureIndicator = SensorState.Cold;
        else if(currentTemperature > 68 && currentTemperature < 72)
            TemperatureIndicator = SensorState.Normal;
        else if(currentTemperature >= 72)
            TemperatureIndicator = SensorState.Hot;

        return currentTemperature;
    }
}
public SensorState TemperatureIndicator { get; set; }

public Sensor(string iotHubUri, string deviceId, string deviceKey)
{
    _IotHubUri = iotHubUri;
    DeviceId = deviceId;
    DeviceKey = deviceKey;
    State = DeviceState.Registered;
}
public void InstallDevice(string statusWindow)
{
    StatusWindow = statusWindow;
    State = DeviceState.Installed;
}

/// <summary>
/// Connect a device to the IoT Hub by instantiating a DeviceClient for tha
/// </summary>
public void ConnectDevice()
{
    //TODO: 17. Connect the Device to Iot Hub by creating an instance of De
    _DeviceClient = DeviceClient.Create(_IotHubUri, new DeviceAuthenticatic
```

```
        //Set the Device State to Ready
        State = DeviceState.Ready;
    }
    public void DisconnectDevice()
    {
        //Delete the local device client
        _DeviceClient = null;

        //Set the Device State to Activate
        State = DeviceState.Activated;
    }

    /// <summary>
    /// Send a message to the IoT Hub from the Smart Meter device
    /// </summary>
    public async void SendMessageAsync()
    {
        var telemetryDataPoint = new
        {
            id = DeviceId,
            time = DateTime.UtcNow.ToString("o"),
            temp = CurrentTemperature
        };

        //TODO: 18.Serialize the telemetryDataPoint to JSON
        var messageString = JsonConvert.SerializeObject(telemetryDataPoint);

        //TODO: 19.Encode the JSON string to ASCII as bytes and create new Mes
        var message = new Message(Encoding.ASCII.GetBytes(messageString));

        //TODO: 20.Send the message to the IoT Hub
        var sendEventAsync = _DeviceClient?.SendEventAsync(message);
        if (sendEventAsync != null) await sendEventAsync;
    }

    /// <summary>
    /// Check for new messages sent to this device through IoT Hub.
    /// </summary>
    public async void ReceiveMessageAsync()
    {
        try
        {
            Message receivedMessage = await _DeviceClient?.ReceiveAsync();
            if (receivedMessage == null)
            {
                ReceivedMessage = null;
                return;
            }

            //TODO: 21.Set the received message for this sensor to the string \
```

```
                ReceivedMessage = Encoding.ASCII.GetString(receivedMessage.GetBytes
                if(double.TryParse(ReceivedMessage, out var requestedTemperature))
                {
                    ReceivedTemperatureSetting = requestedTemperature;
                }
                else
                {
                    ReceivedTemperatureSetting = null;
                }

                // Send acknowledgement to IoT Hub that the has been successfully p
                // The message can be safely removed from the device queue. If some
                // that prevented the device app from completing the processing of
                // IoT Hub delivers it again.

                //TODO: 22.Send acknowledgement to IoT hub that the message was pro
                await _DeviceClient?.CompleteAsync(receivedMessage);
            }
            catch (NullReferenceException ex)
            {
                // The device client is null, likely due to it being disconnected s
                System.Diagnostics.Debug.WriteLine("The DeviceClient is null. This
            }
        }
    }
```

> **Note**: Be sure you only replace the **Sensor** class and not any other code in the file.

3. Save **Sensor.cs**.

Note If you face issues at this point and you are not able to move forward, close the solution go to the folder **C:\SmartMeter\MCW-Internet-of-Things-master\Hands-on lab\sos_bkupfiles**

Copy all the files and replace the five files in the directory below

C:\SmartMeter\MCW-Internet-of-Things-master\Hands-on lab\lab-files\starter-project\SmartMeterSimulator

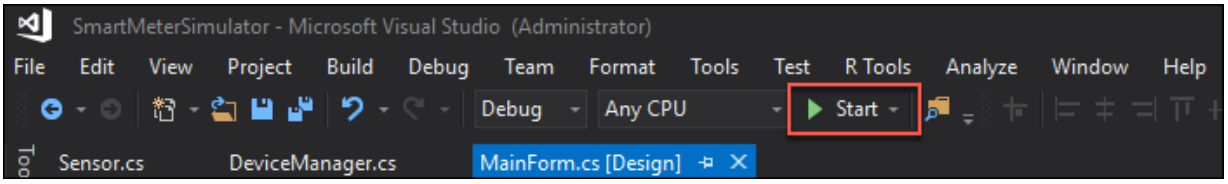
After replacing the files, open the solution again, right click on **SmartMeterSimulator.sln**, select **Open With**, **Visual Studio 2019**, go to **Build** menu, **Build Solution**. Go back to the **Task 2** and modify the connection of you solution, to connect to your current IoT Hub. After this step you are ready to continue with **Task 4**.

Task 4: Verify device registration and telemetry

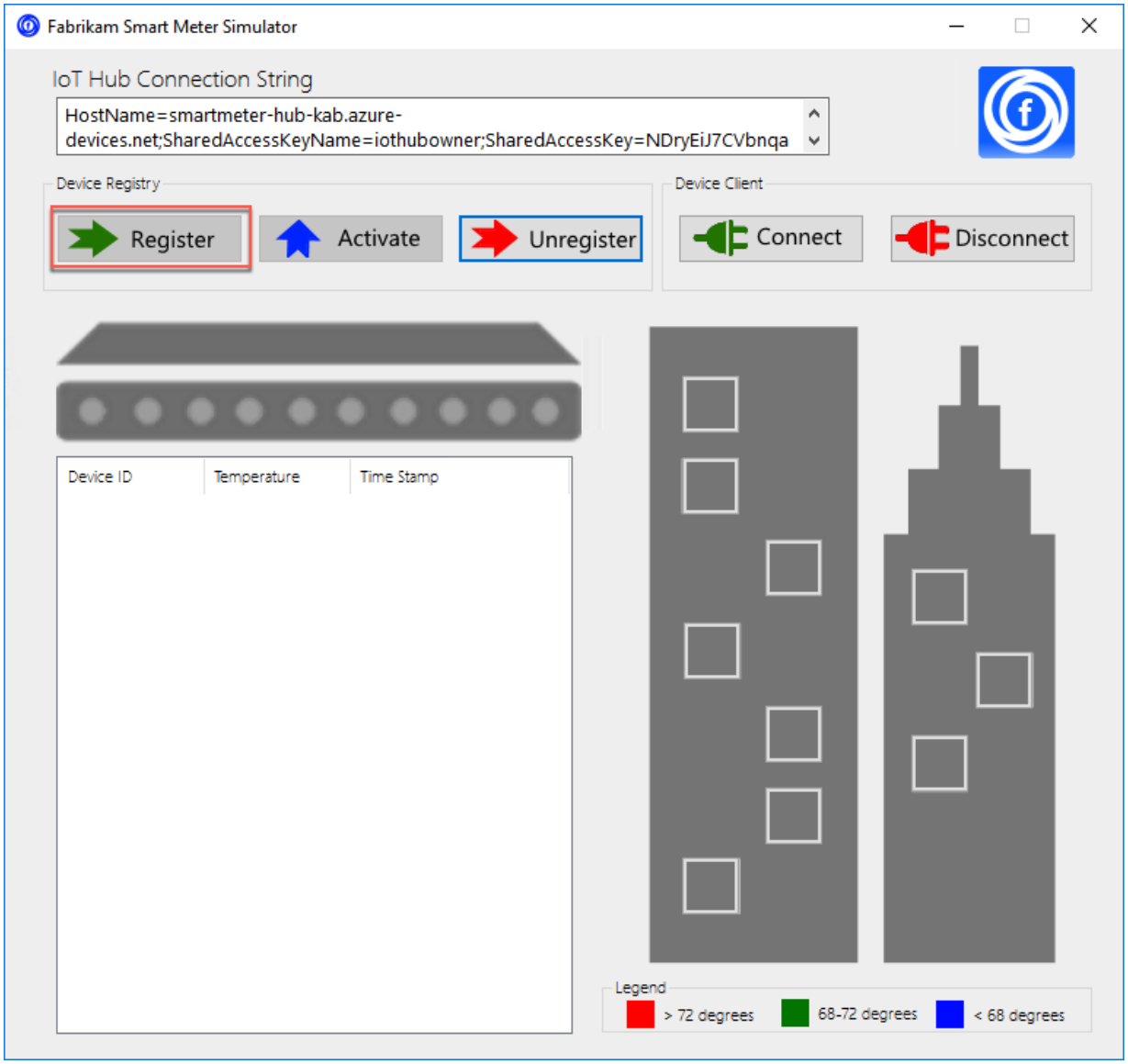
In this task, you will build and run the Smart Meter Simulator project.

1. In **Visual Studio** select **Build** from the Visual Studio menu, then select **Build Solution**.

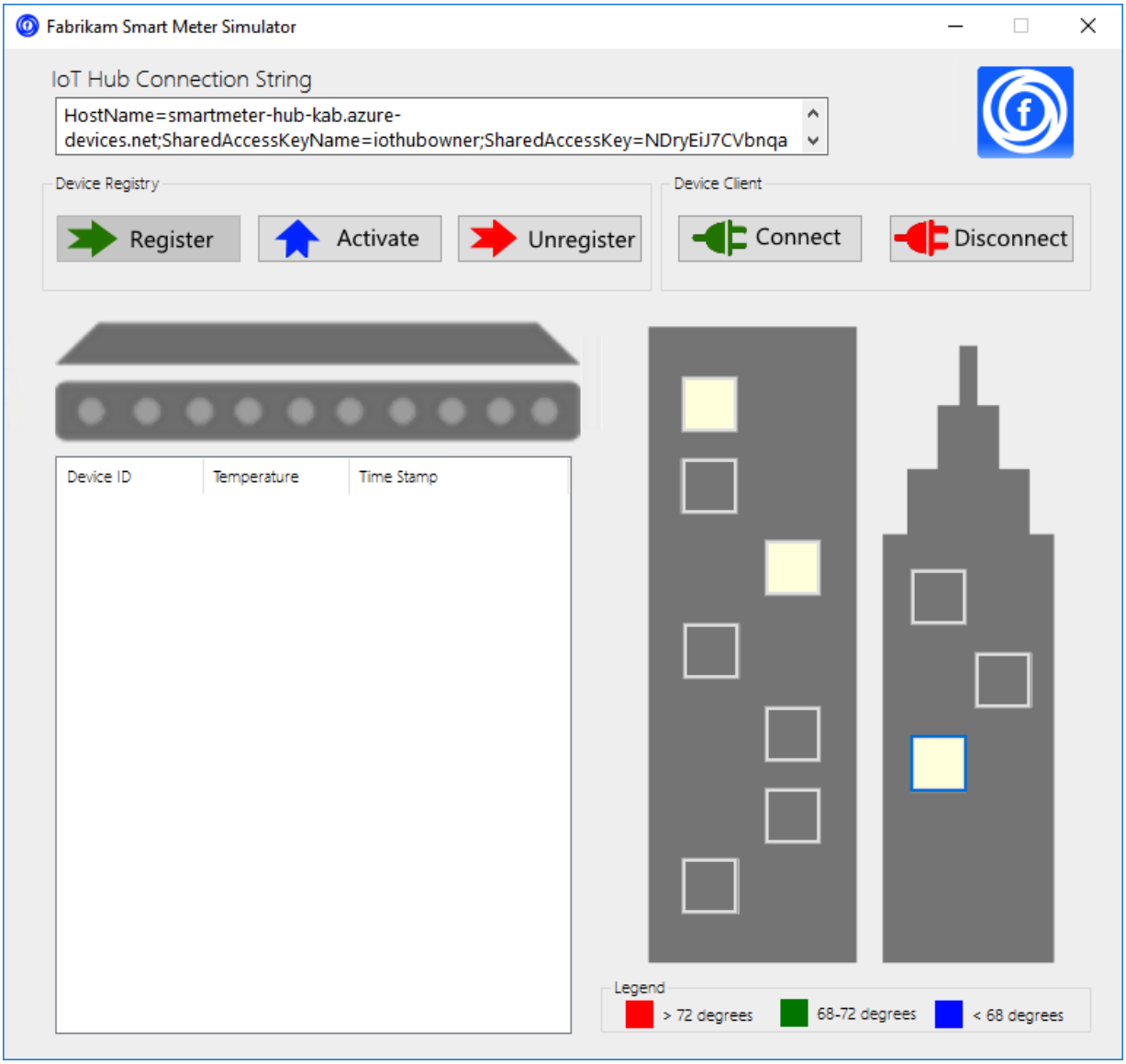
2. Run the **Smart Meter Simulator**, by selecting the green **Start** button on the Visual Studio toolbar.



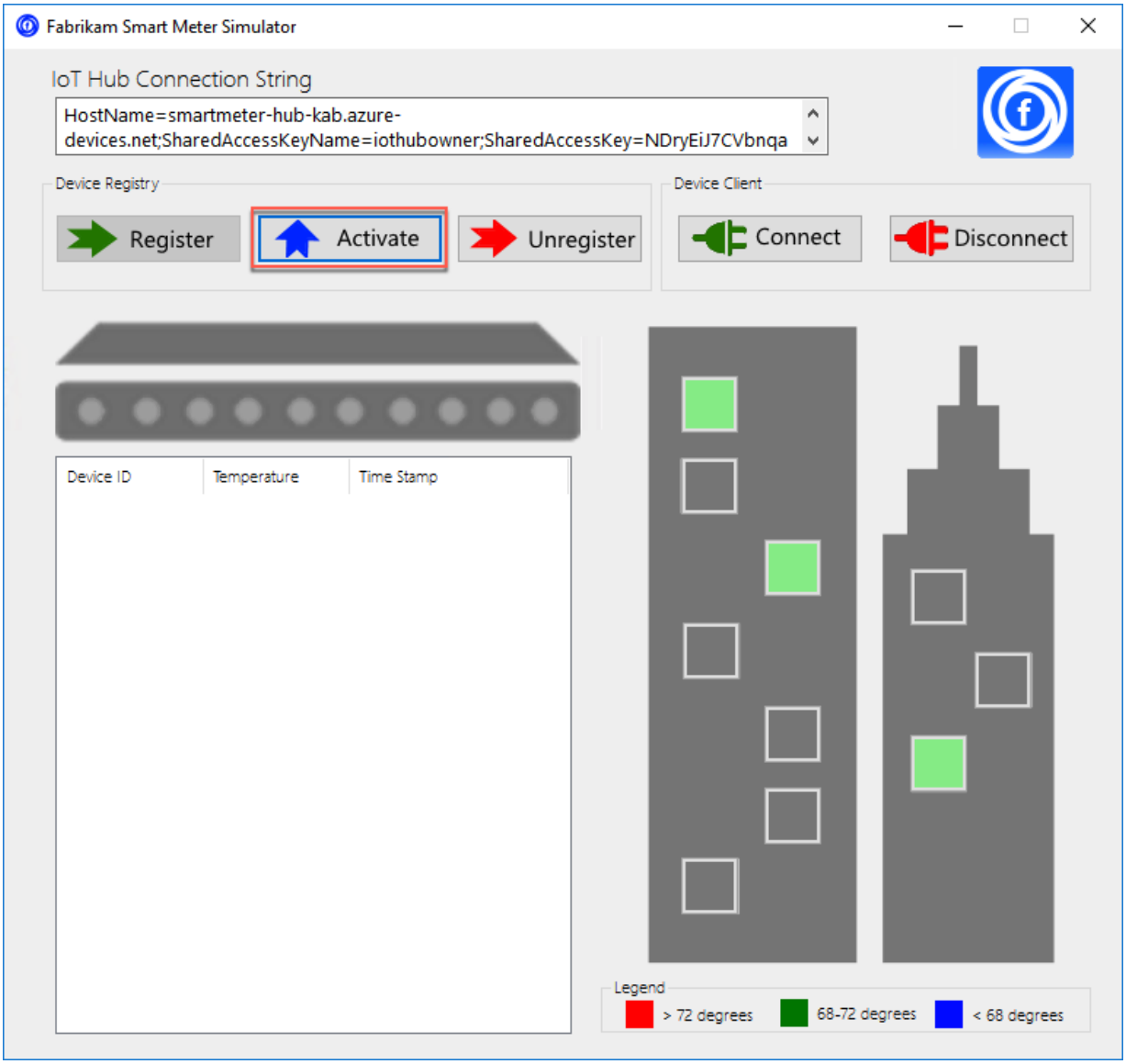
3. Select **Register** on the **Smart Meter Simulator** dialog, which should cause the windows within the building to change from black to gray.



4. Select a few of the windows. Each represents a device for which you want to simulate device installation. The selected windows should turn yellow.

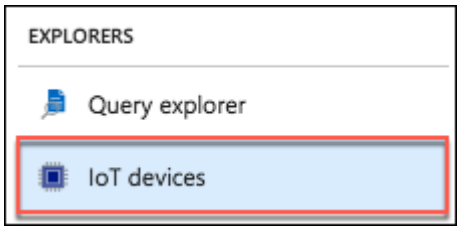


5. Select **Activate** to simulate changing the device status from disabled to enabled in the IoT Hub Registry. The selected windows should turn green.



6. At this point, you have registered 10 devices (the gray windows) but activated only the ones you selected (in green). To view this list of devices, you will switch over to the **Azure Portal**, and open the **IoT Hub** you provisioned.

7. From the **IoT Hub** blade, select **IoT Devices** under **Explorers** on the left-hand menu.

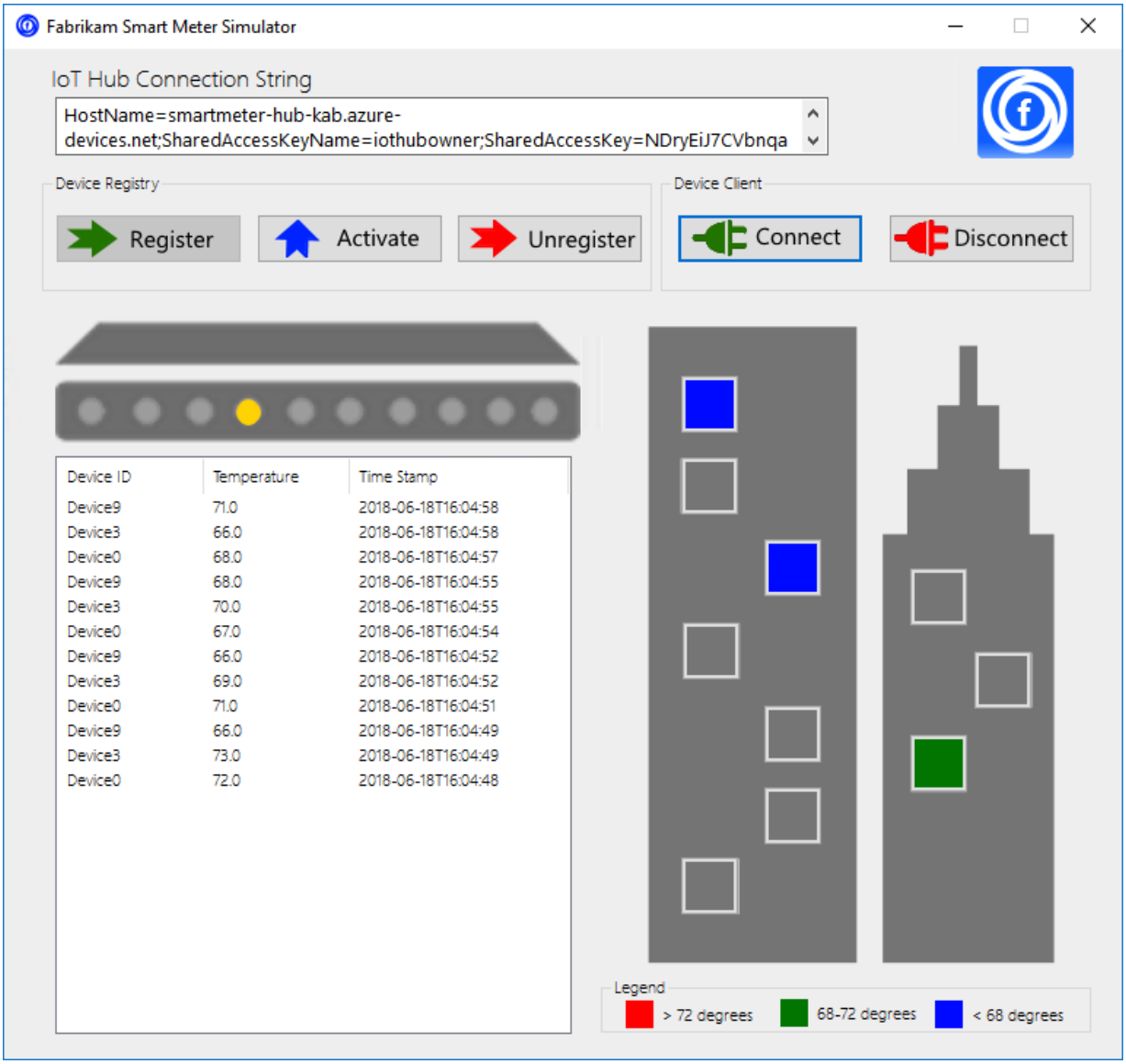


8. You should see all 10 devices listed, with the ones that you activated having a status of **enabled**.

DEVICE ID	STATUS	LAST ACTIVITY	LAST STATUS UPDATE	AUTHENTICATION TYPE
Device0	Disabled			Sas
Device1	Enabled		Thu Mar 21 2019 17:33:48...	Sas
Device2	Disabled			Sas
Device3	Disabled			Sas
Device4	Enabled		Thu Mar 21 2019 17:33:47 ...	Sas
Device5	Enabled		Thu Mar 21 2019 17:33:47 ...	Sas
Device6	Enabled		Thu Mar 21 2019 17:33:48...	Sas
Device7	Enabled		Thu Mar 21 2019 17:33:48...	Sas
Device8	Enabled		Thu Mar 21 2019 17:33:47 ...	Sas
Device9	Disabled			Sas

9. Return to the **Smart Meter Simulator** window.

10. Select **Connect**. Within a few moments, you should begin to see activity as the windows change color, indicating the smart meters are transmitting telemetry. The grid on the left will list each telemetry message transmitted and the simulated temperature value.



11. Allow the smart meter to continue to run. (Whenever you want to stop the transmission of telemetry, select the **Disconnect** button.)

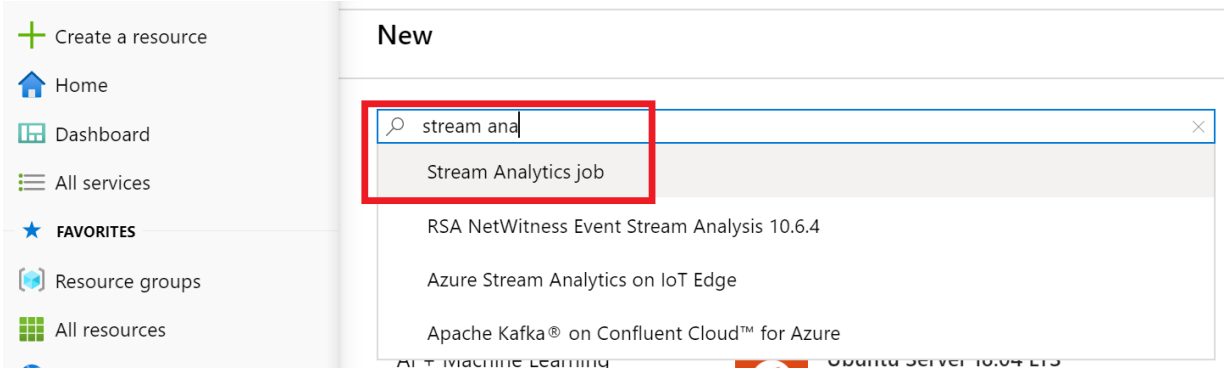
Exercise 3: Hot path data processing with Stream Analytics

Duration: 45 minutes

Fabrikam would like to visualize the "hot" data showing the average temperature reported by each device over a 5-minute window in Power BI.

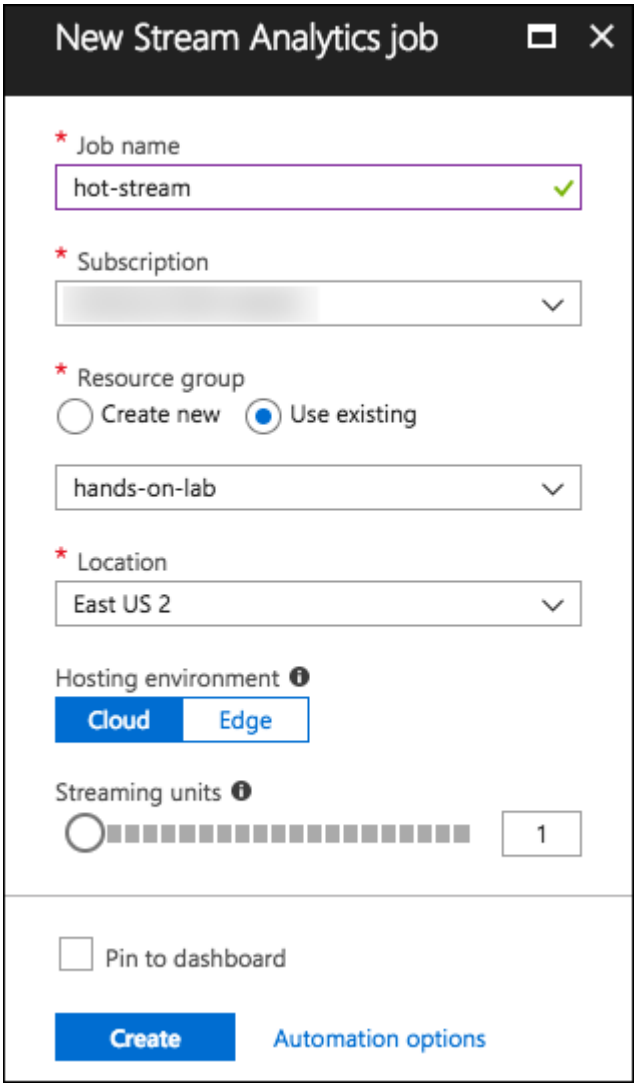
Task 1: Create a Stream Analytics job for hot path processing to Power BI

1. In the Azure Portal, select + **Create a resource**, enter `stream analytics` into the **Search the Marketplace** box, select **Stream Analytics job** from the results, and select **Create**.



2. On the New Stream Analytics Job blade, enter the following:

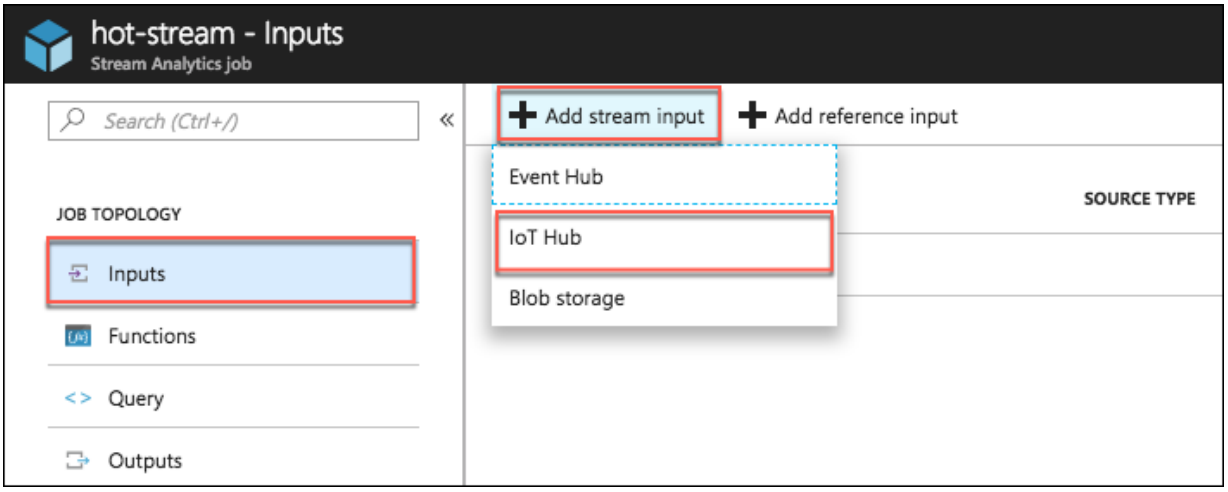
    - **Job name**: Enter `hot-stream`
    - **Subscription**: Select the subscription you are using for this hands-on lab.
    - **Resource group**: Choose Use existing and select the **hands-on-lab-SUFFIX** resource group.
    - **Location**: Select the location you are using for resources in this hands-on lab.
    - **Hosting environment**: Select **Cloud**.
    - **Streaming units**: Change the value to `1` by sliding the slider all the way left.



3. Select **Create**.

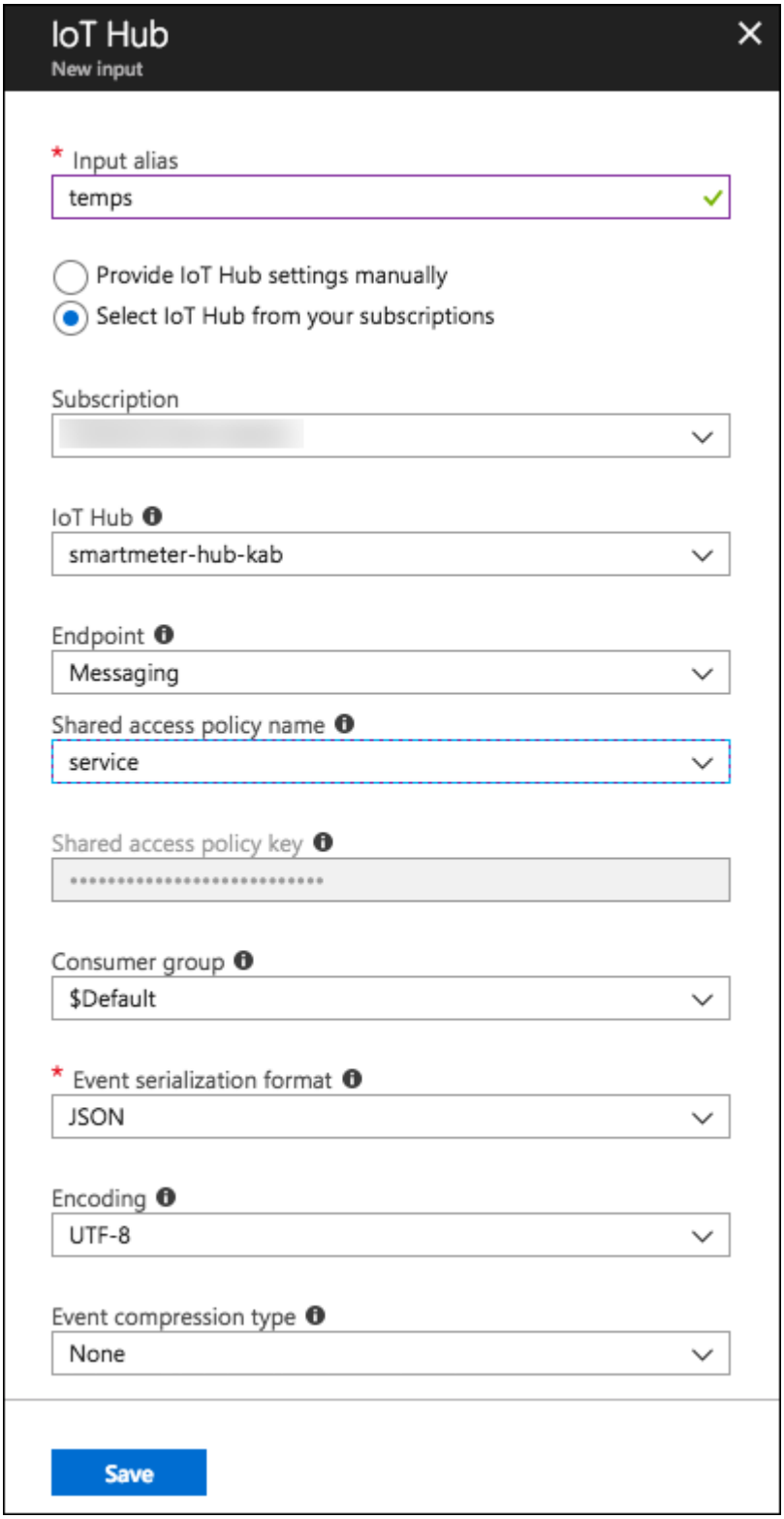
4. Once provisioned, navigate to your new **Stream Analytics job** in the portal.

5. On the **Stream Analytics job** blade, select **Inputs** from the left-hand menu, under **Job Topology**, then select **+Add stream input**, and select **IoT Hub** from the dropdown menu to add an input connected to your IoT Hub.



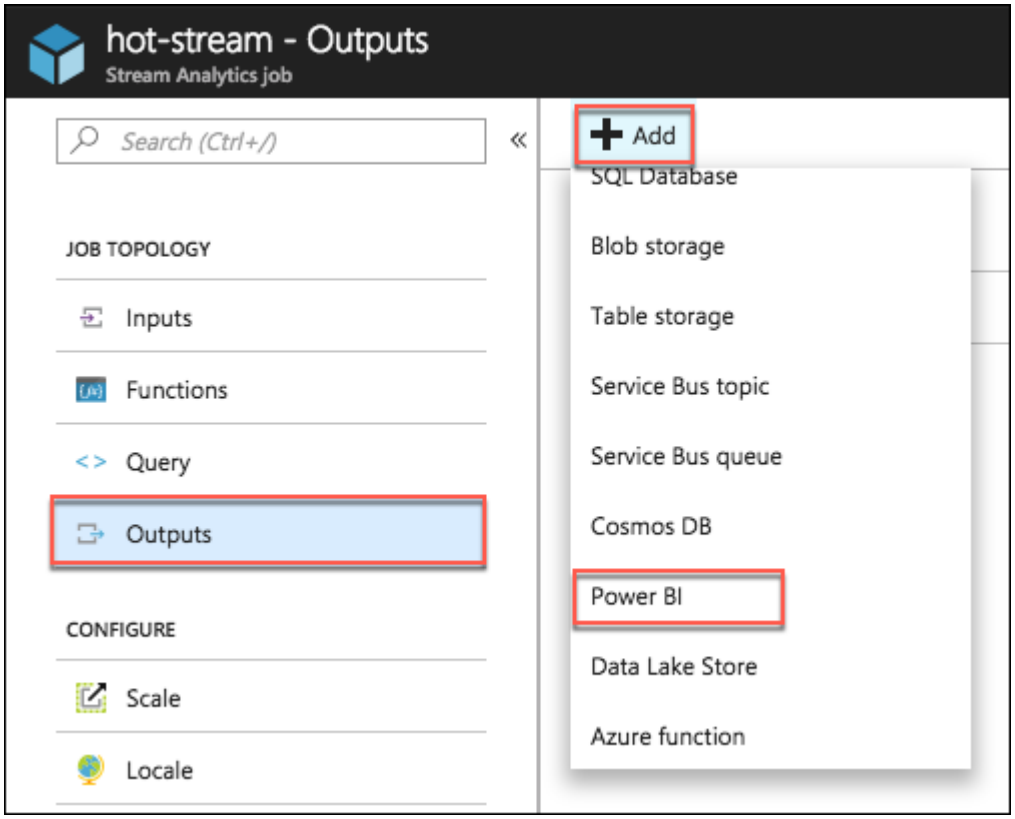
6. On the **New Input** blade, enter the following:

    - **Input alias**: Enter `temps`
    - Choose **Select IoT Hub from your subscriptions**.
    - **Subscription**: Select the subscription you are using for this hands-on lab.
    - **IoT Hub**: Select the **smartmeter-hub-SUFFIX** IoT Hub.
    - **Endpoint**: Select **Messaging**.
    - **Shared access policy name**: Select **service**.
    - **Consumer Group**: Leave set to **$Default**.
    - **Event serialization format**: Select **JSON**.
    - **Encoding**: Select **UTF-8**.
    - **Event compression type**: Leave set to **None**.

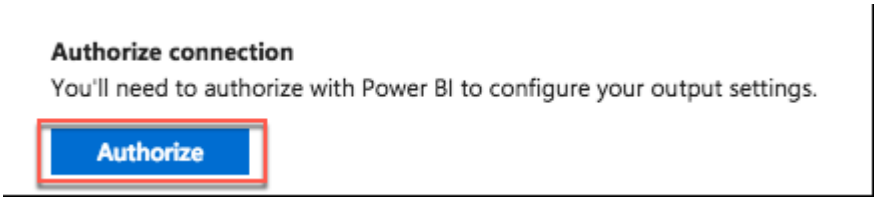


7. Select **Save**.

8. Next, select **Outputs** from the left-hand menu, under **Job Topology**, and select **+ Add**, then select **Power BI** from the drop-down menu.

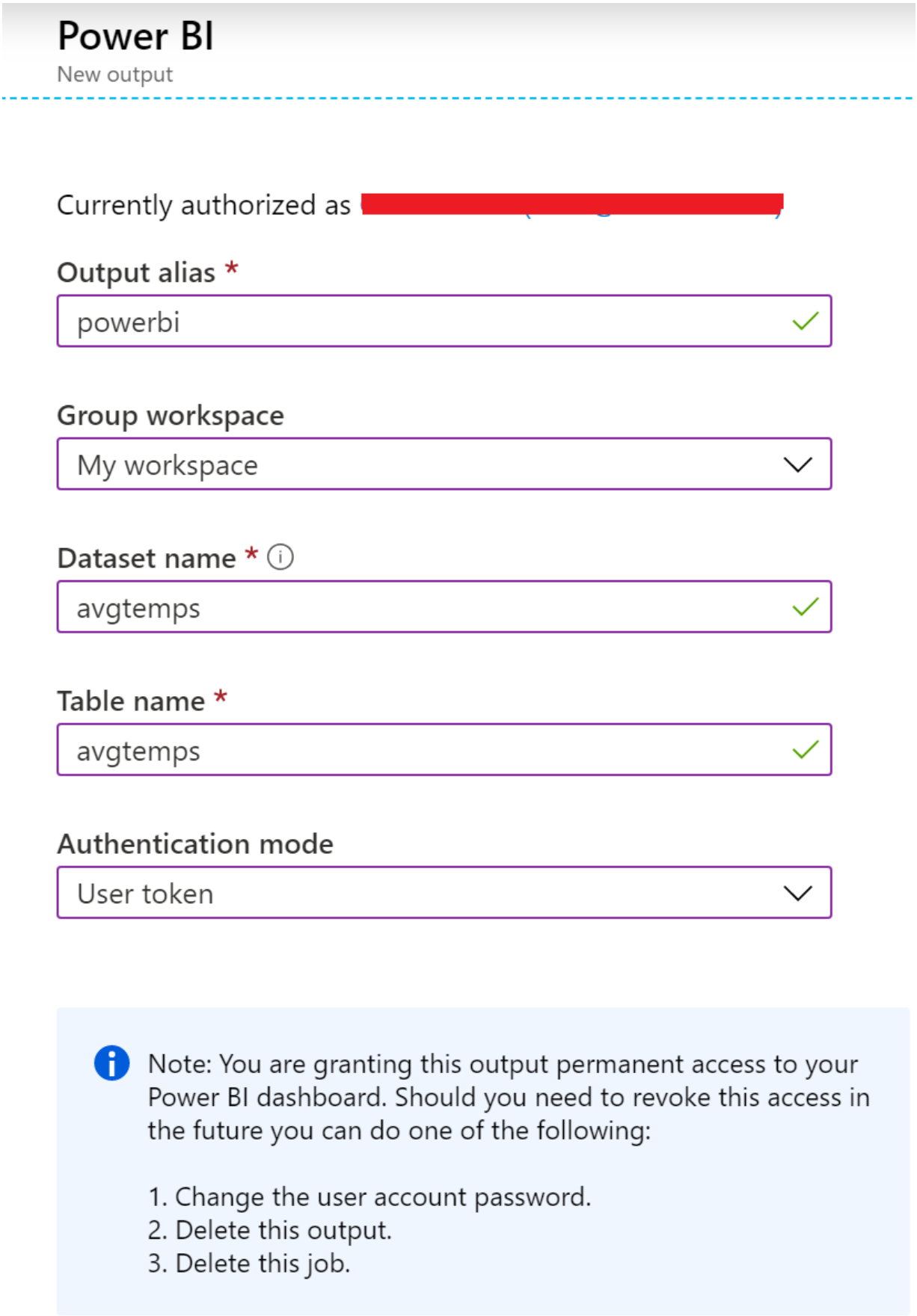


9. In the **Power BI** blade, select **Authorize** to authorize the connection to your Power BI account. When prompted in the popup window, enter the account credentials you used to create your Power BI account in Before the hands-on lab setup guide, Task 1.



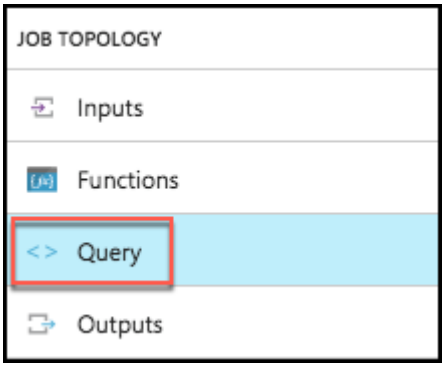
10. Once authorized, enter the following:

    - **Output alias**: Set to `powerbi`
    - For the remaining Power BI settings, enter the following:
        - **Group Workspace**: Select the default, **My Workspace**.
        - **Dataset Name**: Enter `avgtemps`
        - **Table Name**: Enter `avgtemps`
        - **Authentication mode**: Select **User token**.



11. Select **Save**.

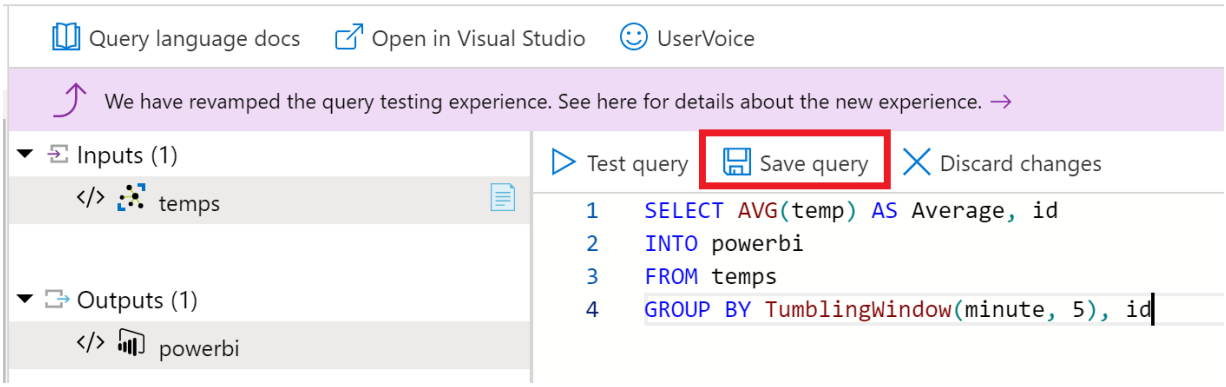
12. Next, select **Query** from the left-hand menu, under **Job Topology**.



13. In the **Query** text box, paste the following query.

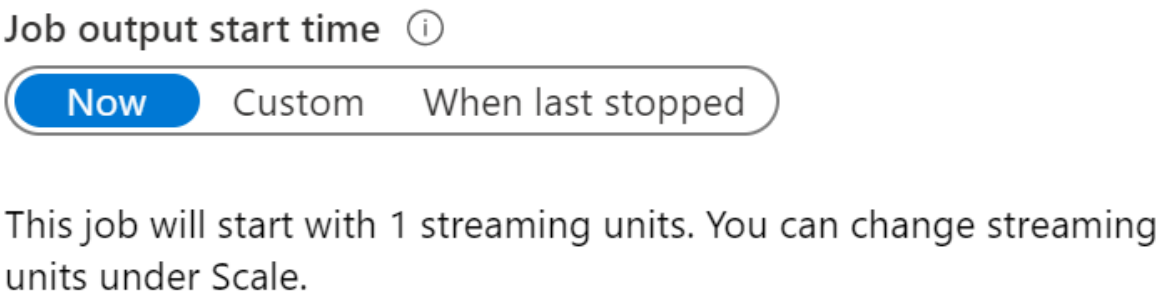
```
SELECT AVG(temp) AS Average, id
INTO powerbi
FROM temps
GROUP BY TumblingWindow(minute, 5), id
```

14. Select **Save query**.



15. Return to the **Overview** blade on your **Stream Analytics job** and select **Start**.

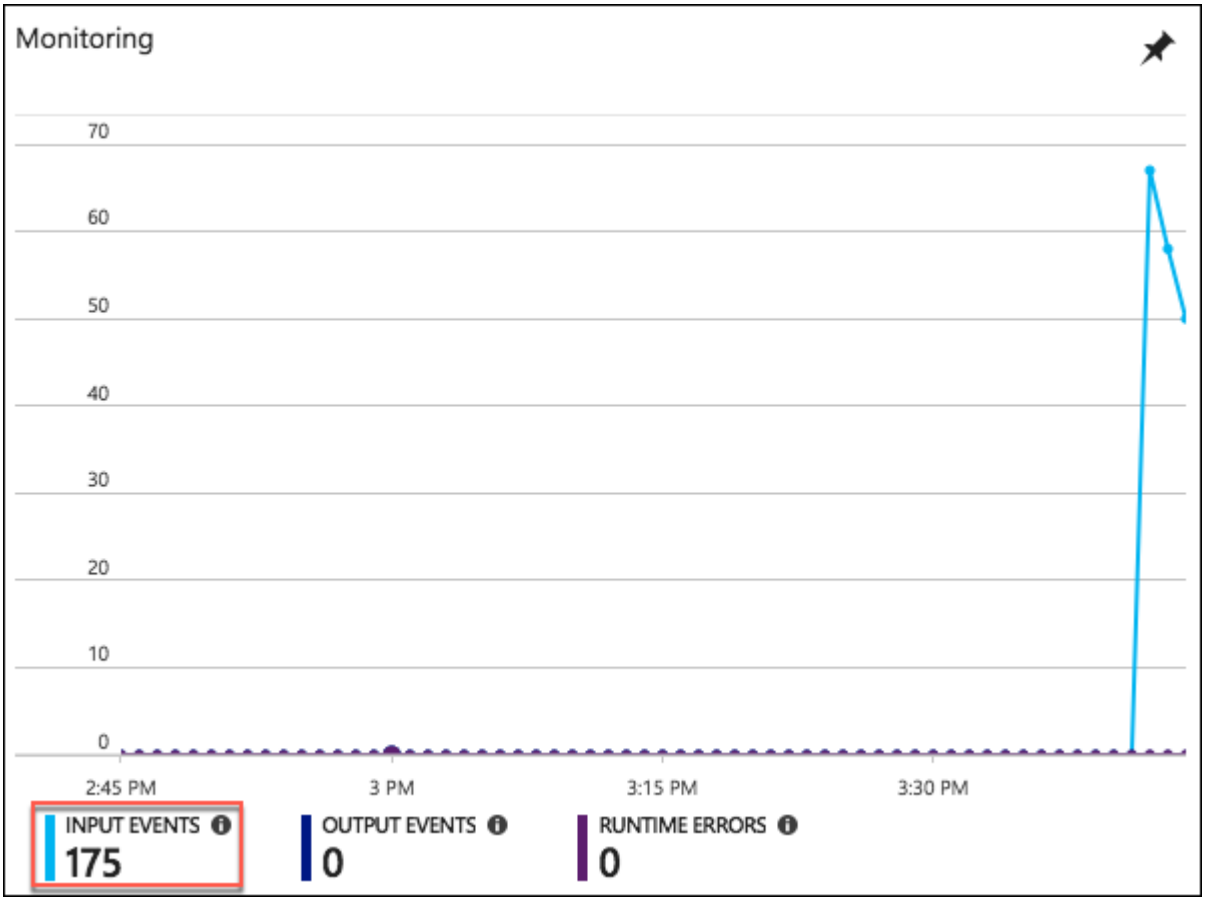
16. In the **Start job** blade, select **Now** (the job will start processing messages from the current point in time onward).



17. Select **Start**.

18. Allow your Stream Analytics Job a few minutes to start.

19. Once the Stream Analytics Job has successfully started, verify that you are showing a non-zero amount of **Input Events** on the **Monitoring** chart on the **Overview** blade. You may need to reconnect your devices on the **Smart Meter Simulator** and let it run for a while to see the events.

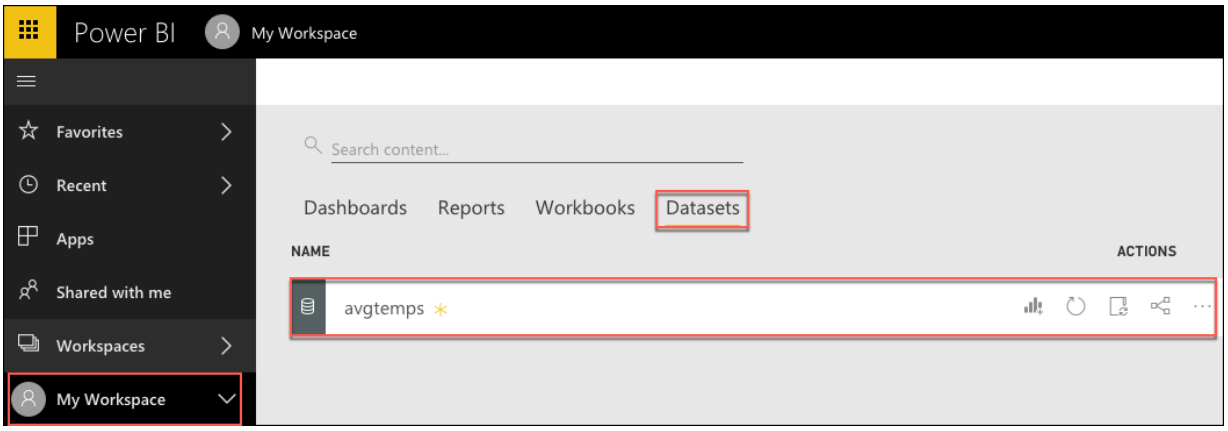


Task 2: Visualize hot data with Power BI

1. Sign in to your Power BI subscription (https://app.powerbi.com) to see if data is being collected.

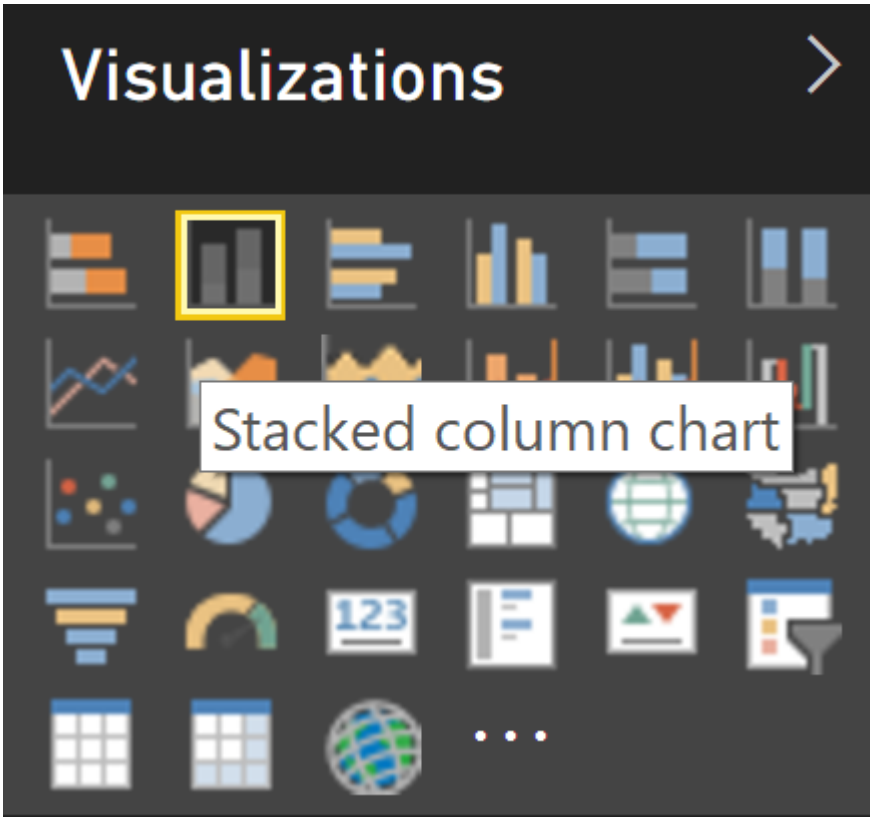
2. Select **My Workspace** on the left-hand menu, then select the **Datasets tab**, and locate the **avgtemps** dataset from the list.

   > **Note:** Sometimes it takes few minutes for the dataset to appear in the Power BI Dataset tab under **My Workspace**

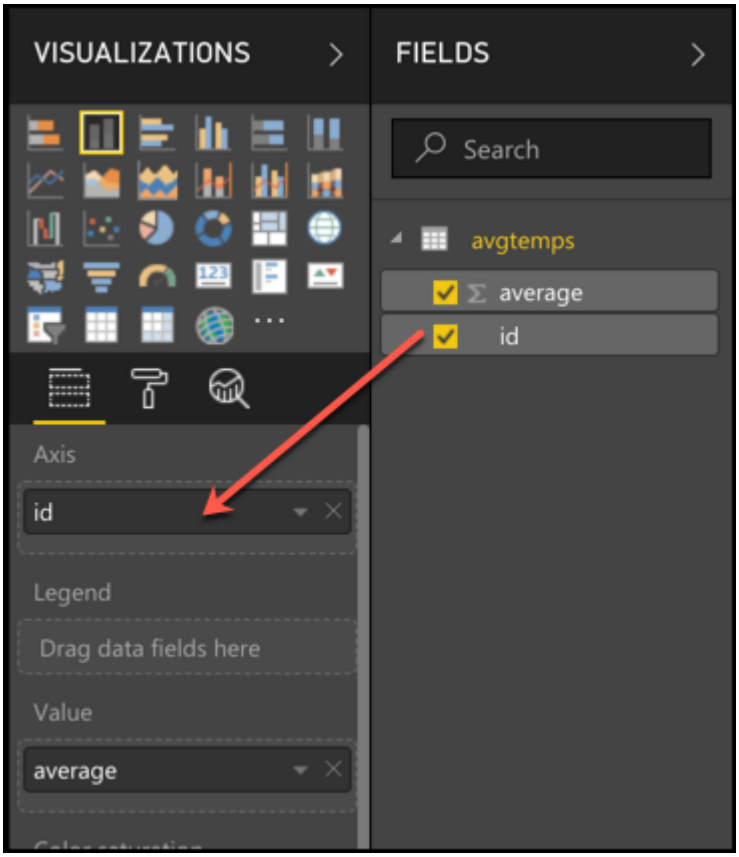


3. Select the **Create Report** button under the **Actions** column.

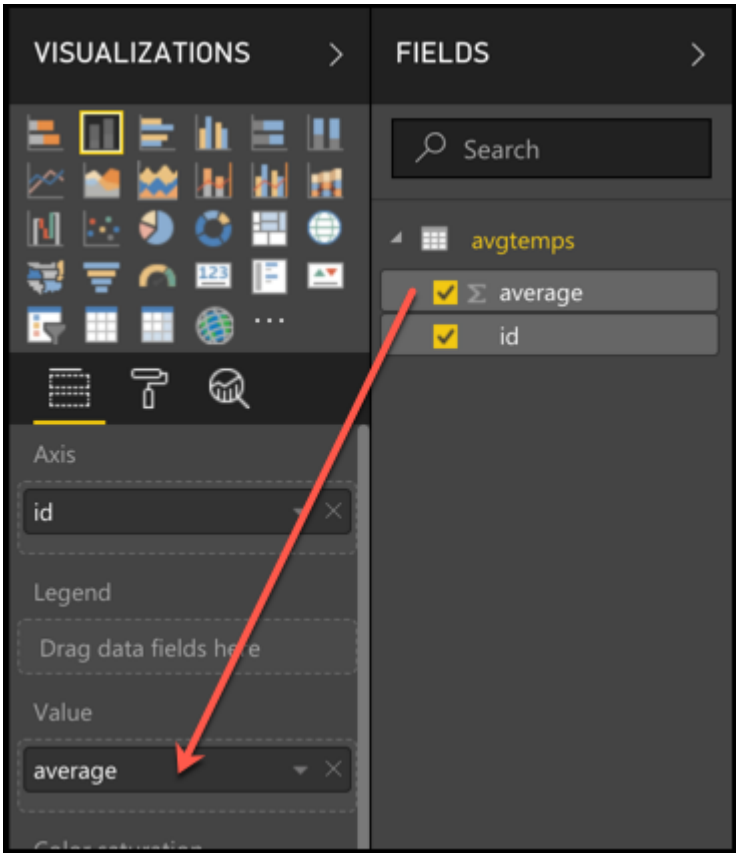
4. On the **Visualizations** palette, select **Stacked column chart** to create a chart visualization.



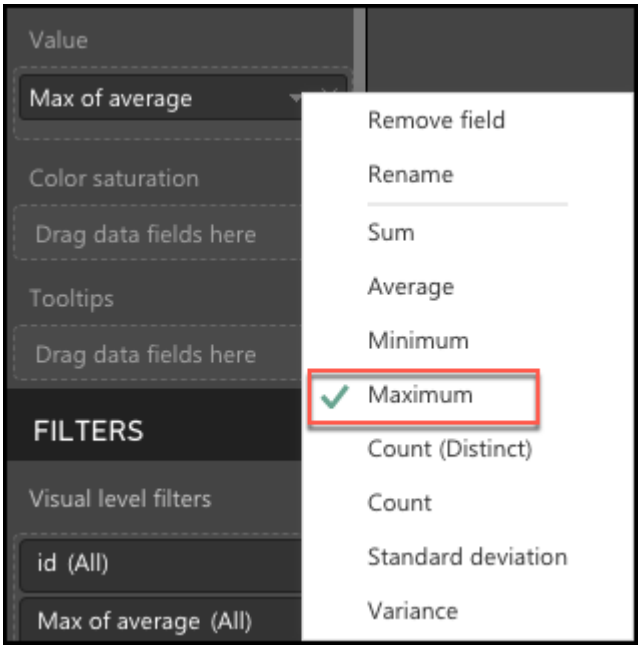
5. In the **Fields** listing, drag the **id** field, and drop it into the **Axis** field.



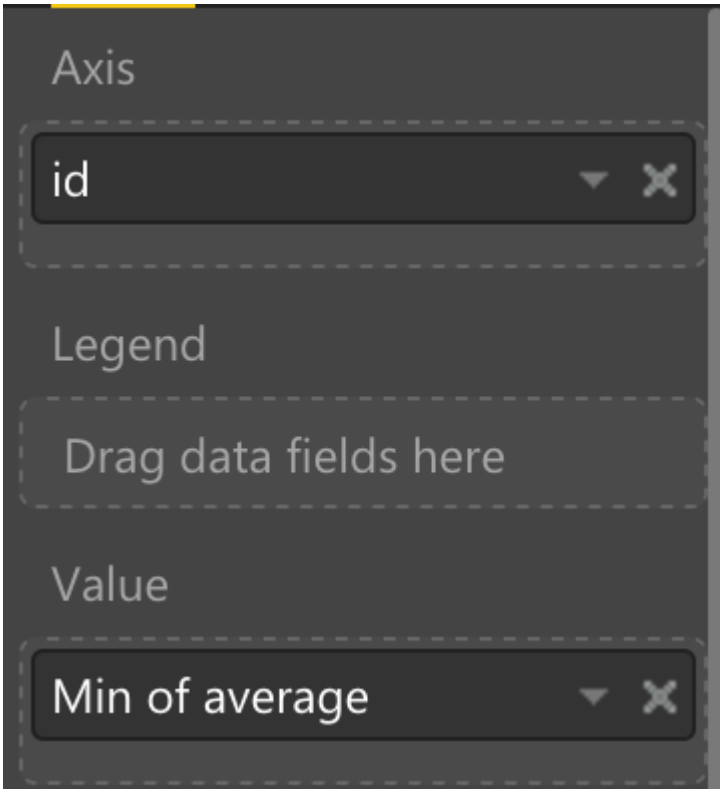
6. Next, drag the **average** field and drop it into the **Value** field.



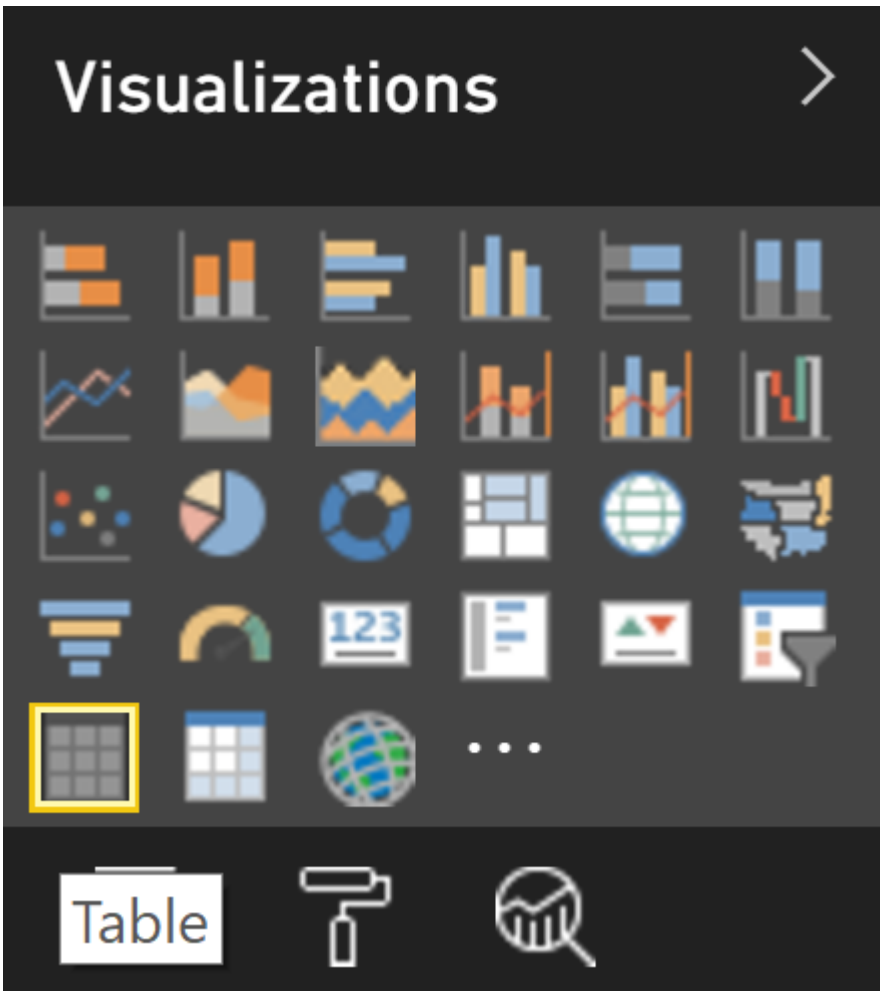
7. Now, set the **Value** to **Max of average**, by selecting the down arrow next to **average**, and select **Maximum**.



8. Repeat steps 5-8, this time adding a Stacked Column Chart for **Min of average**. (You may need to select on any area of white space on the report designer surface to deselect the Max of average by id chart visualization.)

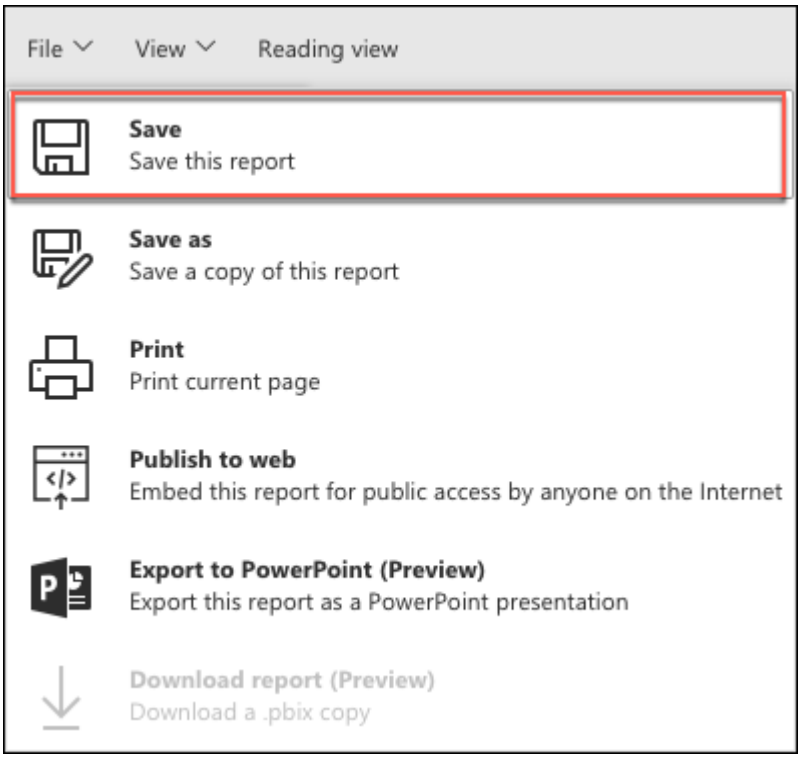


9. Next, add a **table visualization**.

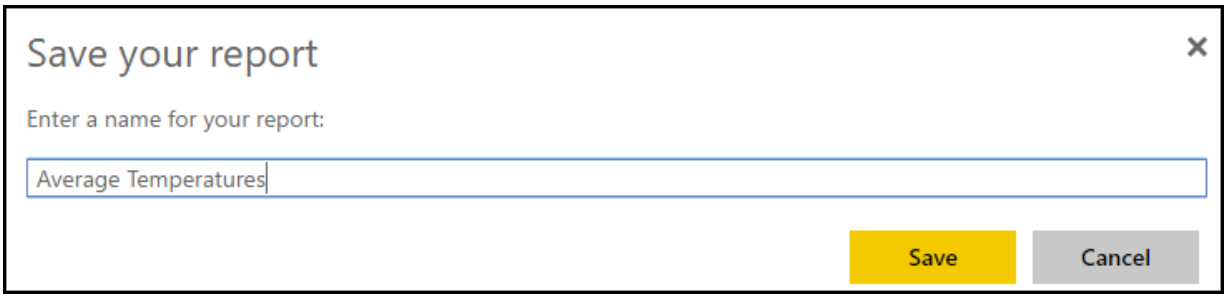


10. Set the values to **id** and **Average of average**, by dragging and dropping both fields in the **Values** field, then selecting the dropdown next to **average**, and selecting **Average**.

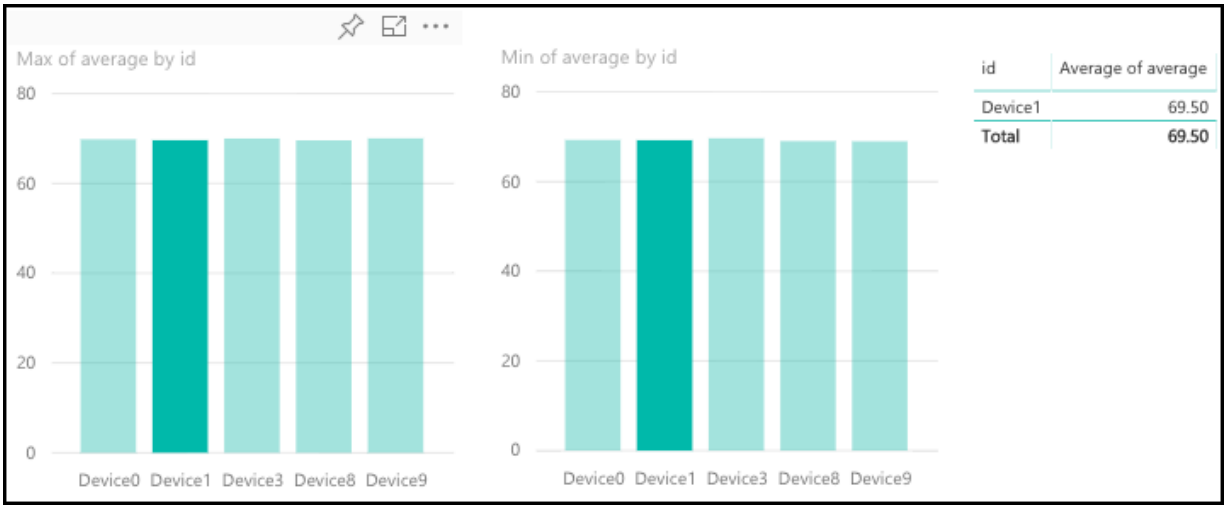
11. Save the report.



12. Enter the name `Average Temperatures`, and select **Save**.



13. Within the report, select one of the columns to see the data for just that device.



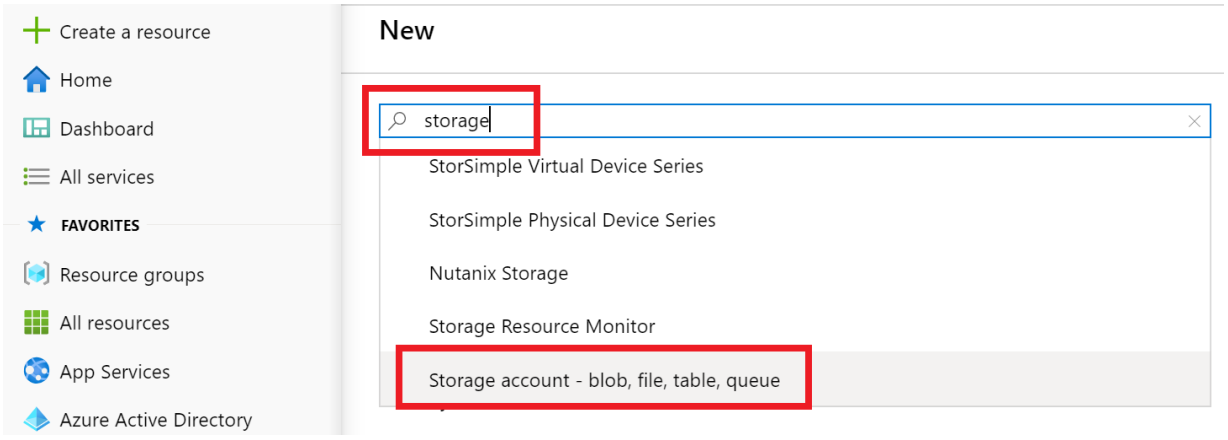
Exercise 4: Cold path data processing with Azure Databricks

Duration: 60 minutes

Fabrikam would like to be able to capture all the "cold" data into scalable storage so that they can summarize it periodically using a Spark SQL query.

Task 1: Create a Storage account

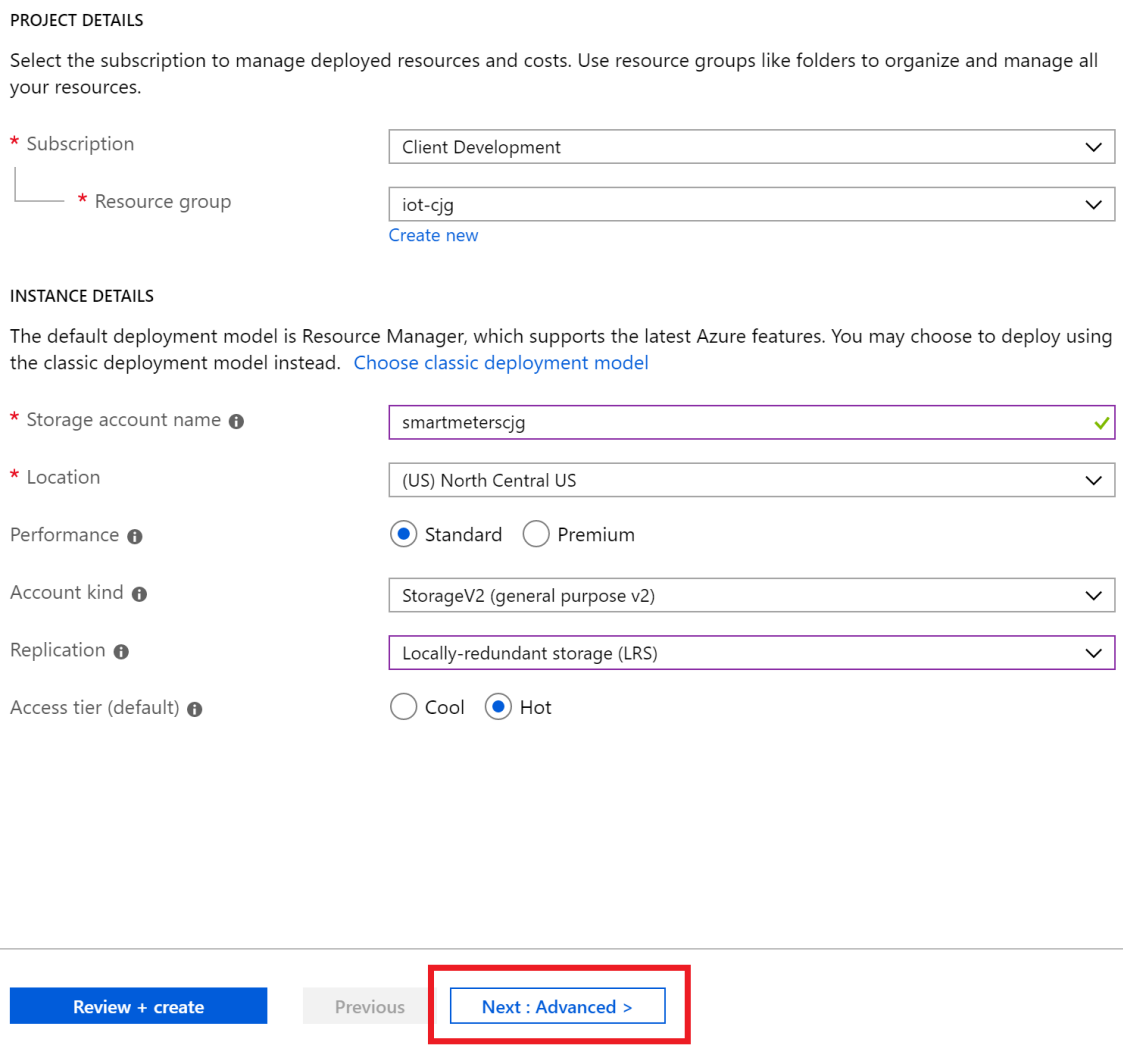
1. In the Azure portal, select + **Create a resource**, enter `storage account` into the **Search the Marketplace** box, select **Storage account** from the results, and select **Create**.



2. In the Create storage account blade, enter the following:

   - **Subscription**: Select the subscription you are using for this hands-on lab.
   - **Resource group**: Choose Use existing and select the **hands-on-lab-SUFFIX** resource group.
   - **Storage account name**: Enter `smartmetersSUFFIX`

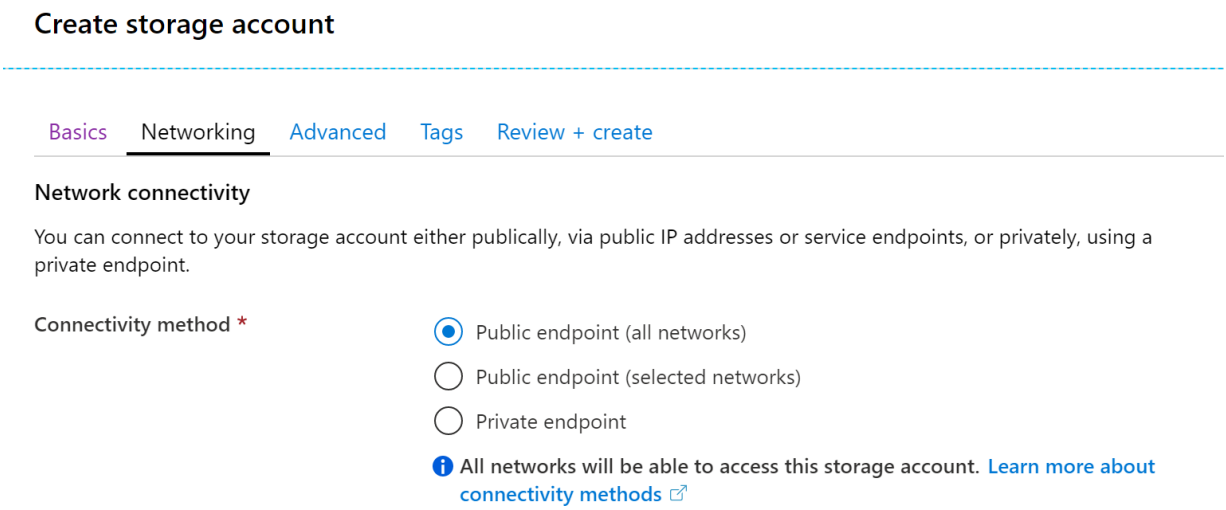
- **Location**: Select the location you are using for resources in this hands-on lab.
- **Performance**: Select **Standard**.
- **Account kind**: Select **StorageV2 (general purpose v2)**.
- **Replication**: Select **Locally-redundant storage (LRS)**.
- **Access tier (default)**: Select **Hot**.



3. Select **Next: Networking >**.

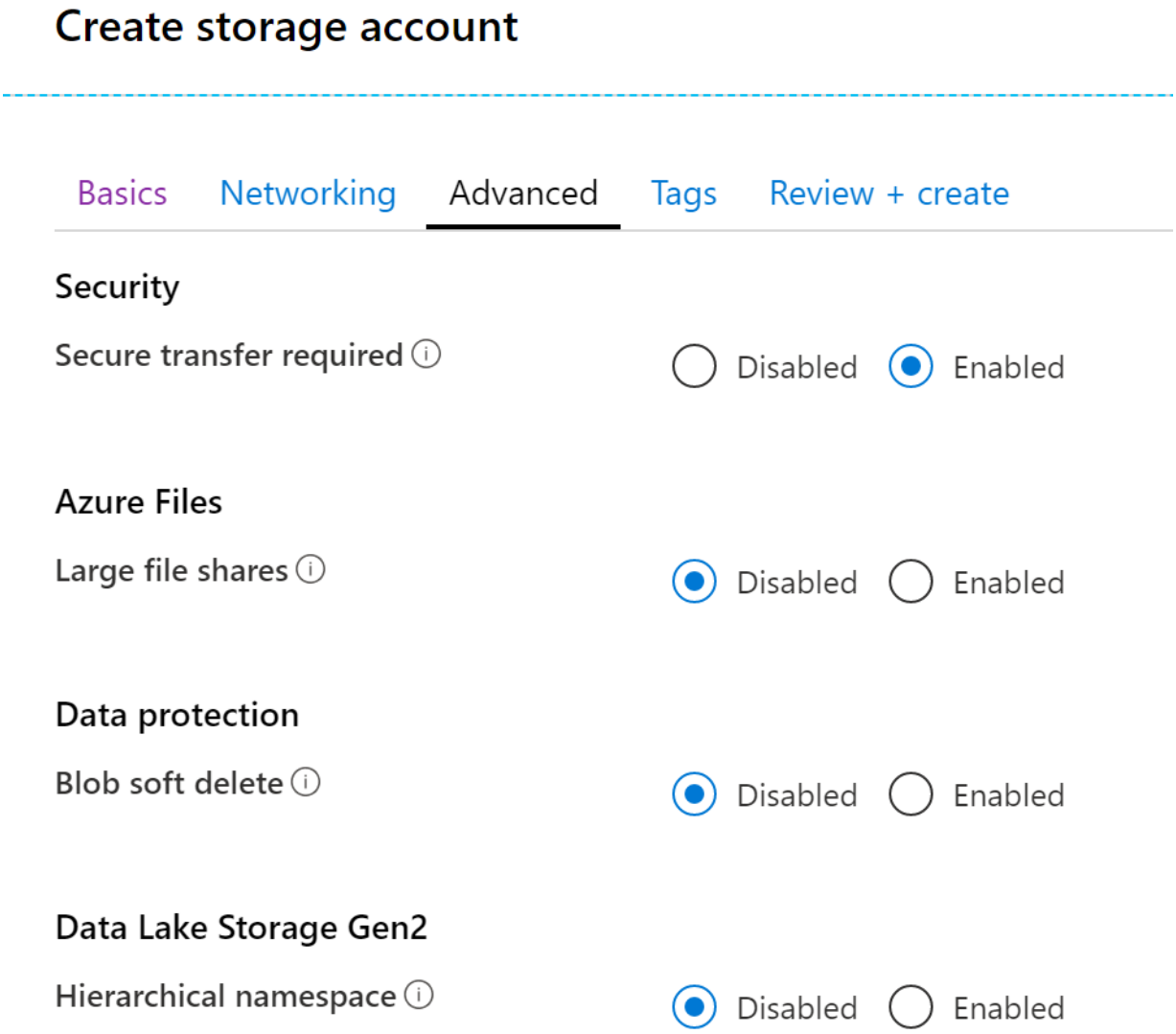
4. Select **Next: Advanced >**.

   - **Connectivity method**: Select **Public endpoint (all networks)**.



5. In the **Advanced** tab, select the following:

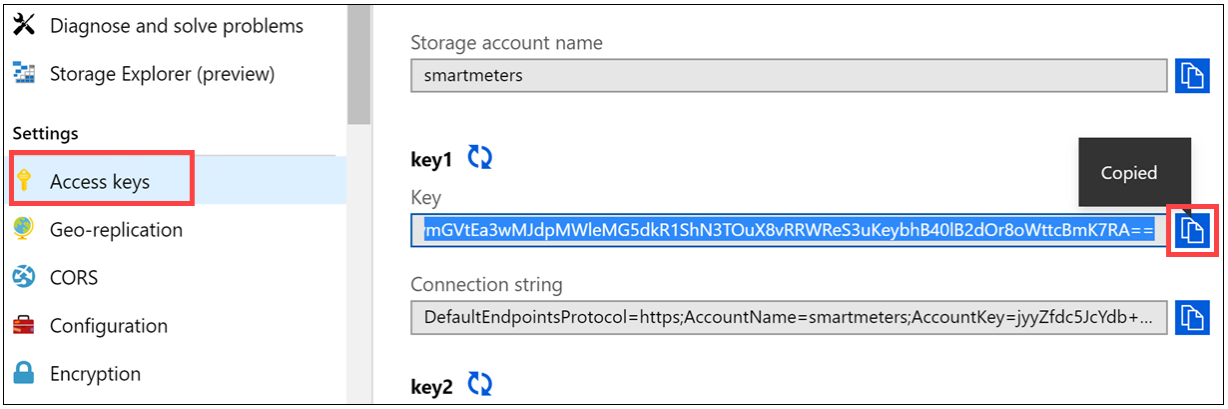
   - **Secure transfer required**: Select **Disabled**.



6. Select **Review + create**.

7. In the **Review + create** tab, select **Create**.

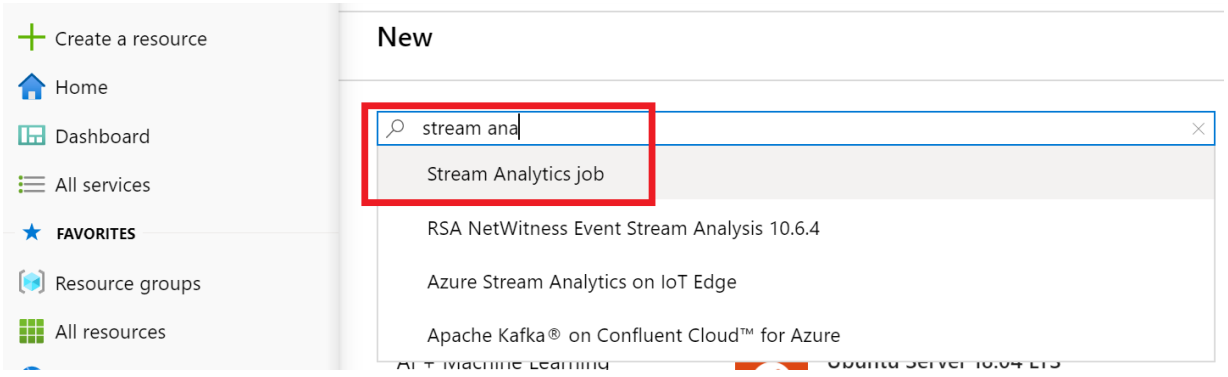
8. Once provisioned, navigate to your storage account, select **Access keys** from the left-hand menu, and copy the **key1** Key value into a text editor, such as Notepad, for later use.



Task 2: Create the Stream Analytics job for cold path processing

To capture all metrics for the cold path, set up another Stream Analytics job that will write all events to Blob storage for analysis with Azure Databricks.

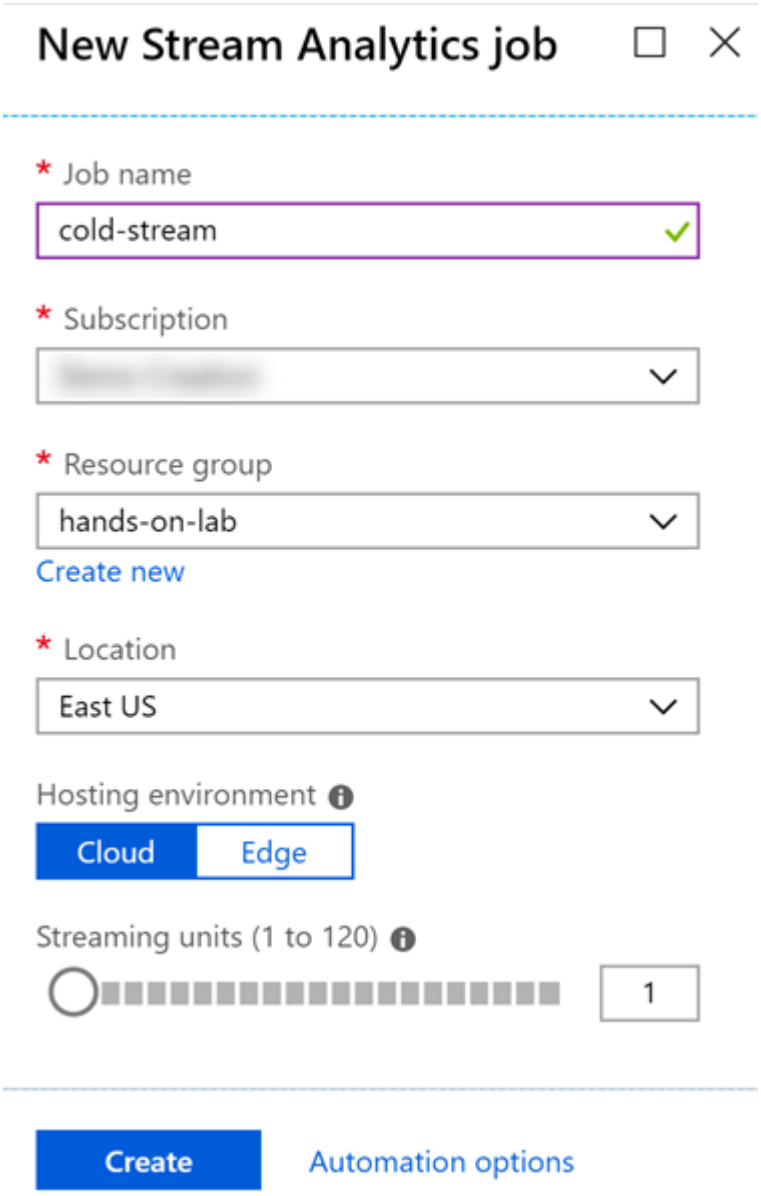
1. In the Azure Portal, select **+ Create a resource**, enter `stream analytics` into the **Search the Marketplace** box, select **Stream Analytics job** from the results, and select **Create**.



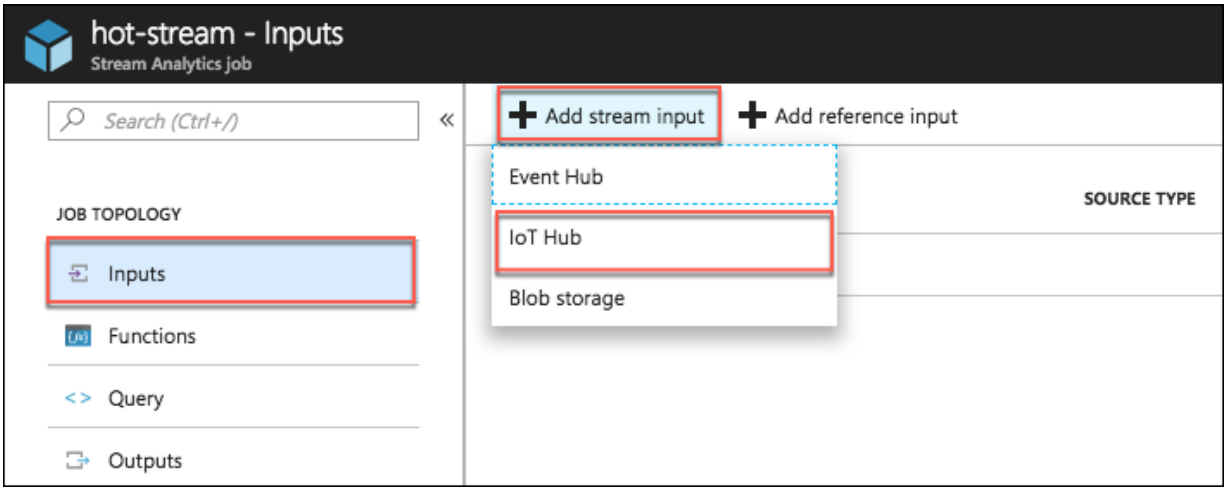
2. On the New Stream Analytics Job blade, enter the following:

   - **Job name**: Enter `cold-stream`
   - **Subscription**: Select the subscription you are using for this hands-on lab.
   - **Resource group**: Select the **hands-on-lab-SUFFIX** resource group.
   - **Location**: Select the location you are using for resources in this hands-on lab.

- **Hosting environment**: Select **Cloud**.
- **Streaming units**: Drag the slider all the way to the left to select `1` streaming unit.

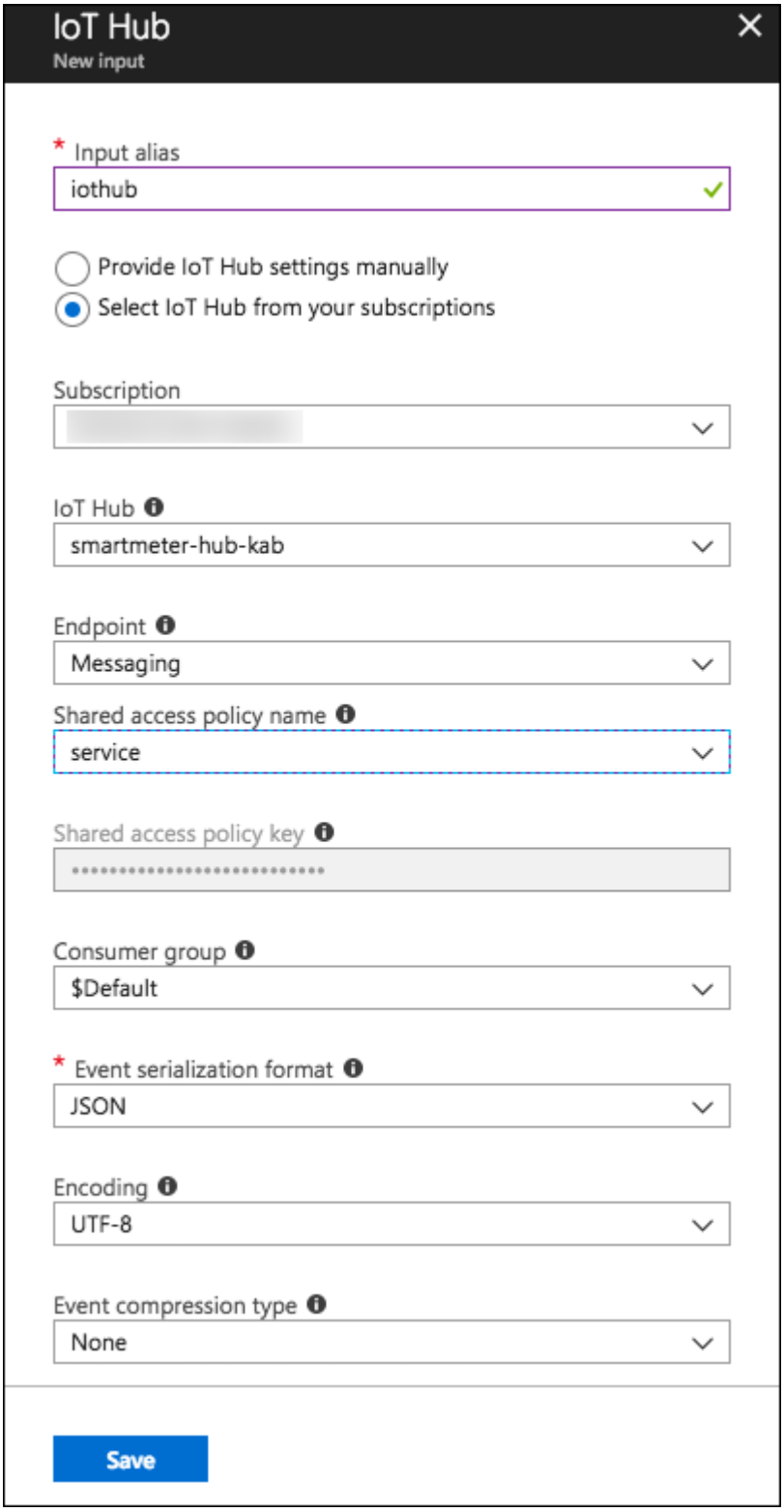


3. Select **Create**.
4. Once provisioned, navigate to your new **Stream Analytics job** in the portal.
5. On the **Stream Analytics job** blade, select **Inputs** from the left-hand menu, under **Job Topology**, then select **+Add stream input**, and select **IoT Hub** from the dropdown menu to add an input connected to your IoT Hub.



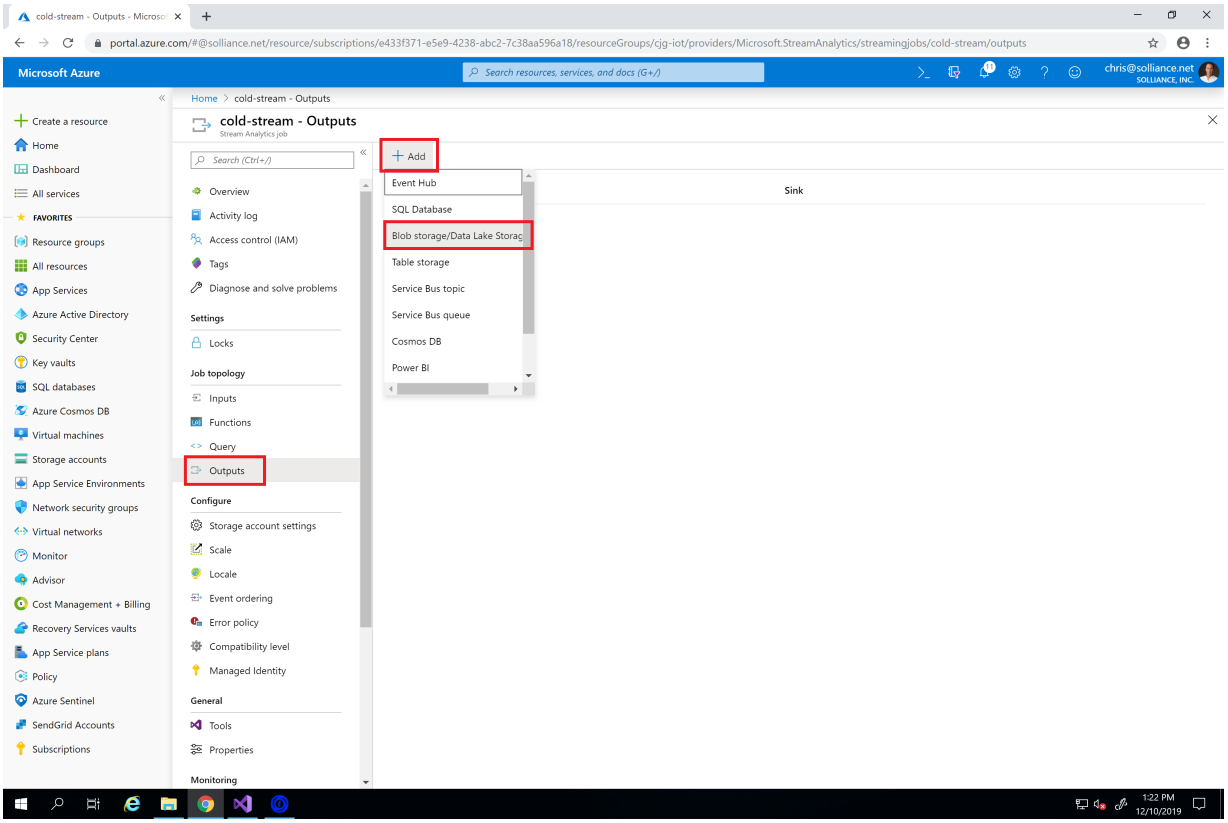
6. On the **New Input** blade, enter the following:

   - **Input alias**: Enter `iothub`
   - Choose **Select IoT Hub from your subscriptions**.
   - **Subscription**: Select the subscription you are using for this hands-on lab.
   - **IoT Hub**: Select the **smartmeter-hub-SUFFIX** IoT Hub.
   - **Endpoint**: Select **Messaging**.
   - **Shared access policy name**: Select **service**.
   - **Consumer Group**: Leave set to **$Default**.
   - **Event serialization format**: Select **JSON**.
   - **Encoding**: Select **UTF-8**.
   - **Event compression type**: Leave set to **None**.



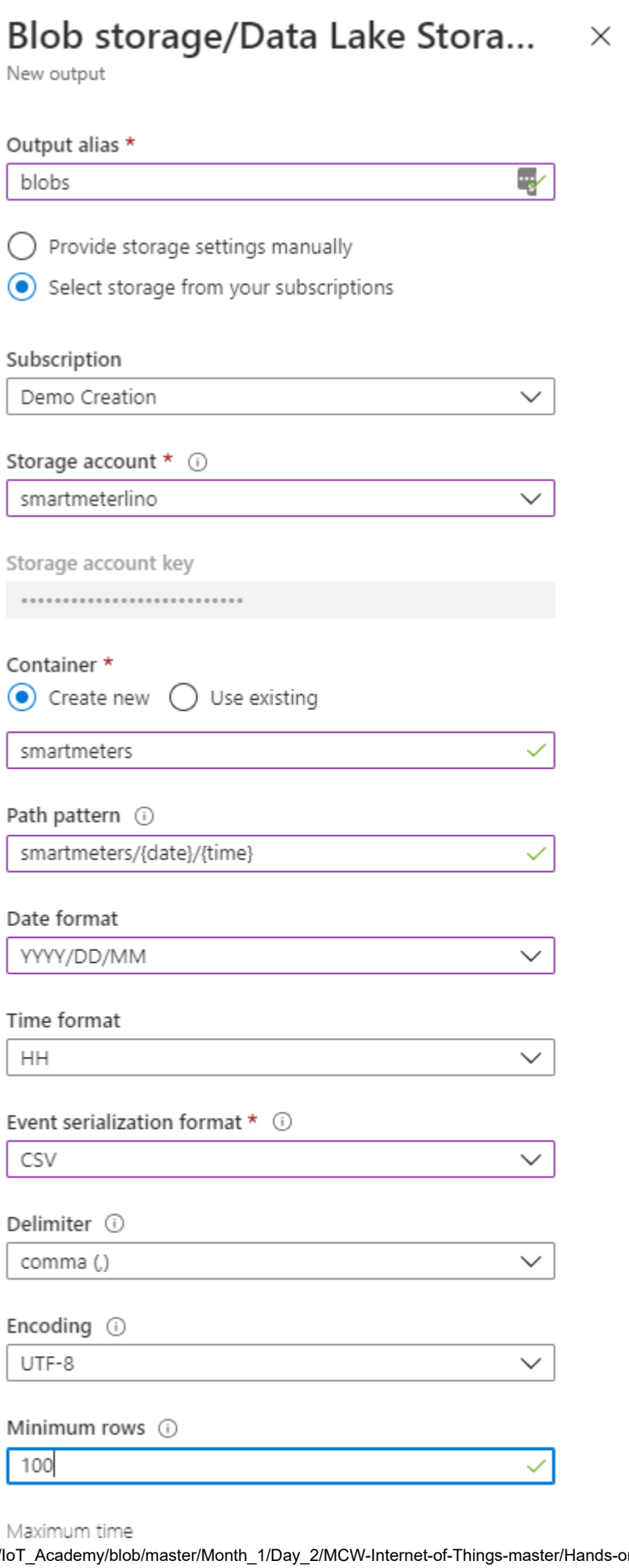
7. Select **Save**.

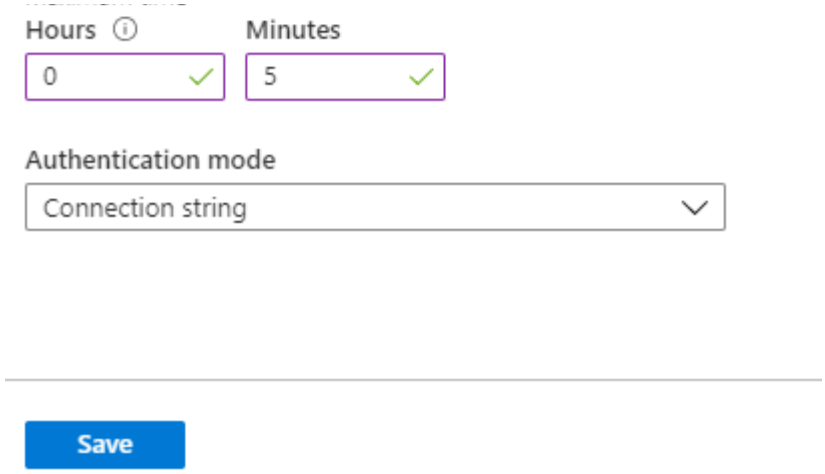
8. Next, select **Outputs** from the left-hand menu, under **Job Topology**, and select **+ Add**, then select **Blob storage/Data Lake Storage Gen2** from the drop-down menu.



9. On the **Blob storage** output blade, enter the following:

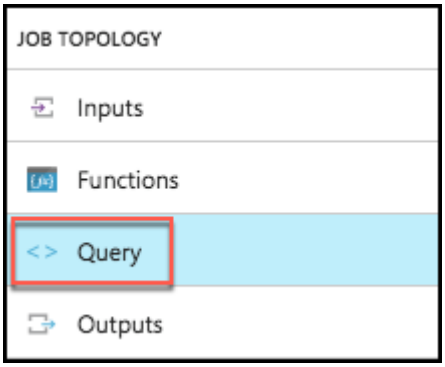
    - **Output alias**: Set to `blobs`
    - Choose **Select blob storage from your subscriptions**.
    - **Subscription**: Select the subscription you are using for this hands-on lab.
    - **Storage account**: Select the **smartmetersSUFFIX** storage account you created in the previous task.
    - **Container**: Choose **Create new** and enter `smartmeters`
    - **Path pattern**: Enter `smartmeters/{date}/{time}`
    - **Date format**: Select **YYYY-DD-MM**.
    - **Time format**: Select **HH**.
    - **Event serialization format**: Select **CSV**.
    - **Delimiter**: Select **comma (,)**.
    - **Encoding**: Select **UTF-8**.


Blob storage/Data Lake Stora...
New output
Output alias *
blobs
Provide storage settings manually
Select storage from your subscriptions
Subscription
Demo Creation
Storage account *
smartmeterlino
Storage account key
Container *
Create new
Use existing
smartmeters
Path pattern
smartmeters/{date}/{time}
Date format
YYYY/DD/MM
Time format
HH
Event serialization format *
CSV
Delimiter
comma (,)
Encoding
UTF-8
Minimum rows
100
Maximum time



10. Select **Save**.

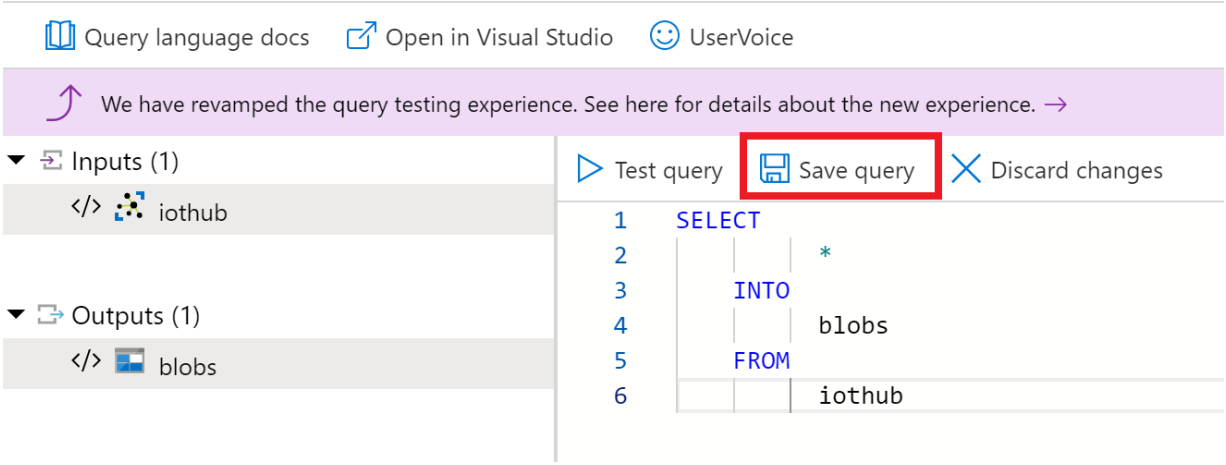
11. Next, select **Query** from the left-hand menu, under **Job Topology**.



12. In the **Query** text box, paste the following query.

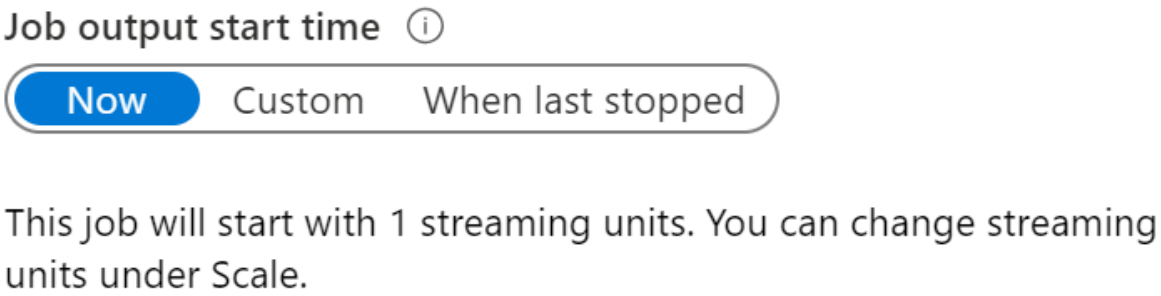
```
SELECT
      *
INTO
      blobs
FROM
      iothub
```

13. Select **Save query**, and **Yes** when prompted with the confirmation.



14. Return to the **Overview** blade on your **Stream Analytics job** and select **Start**.

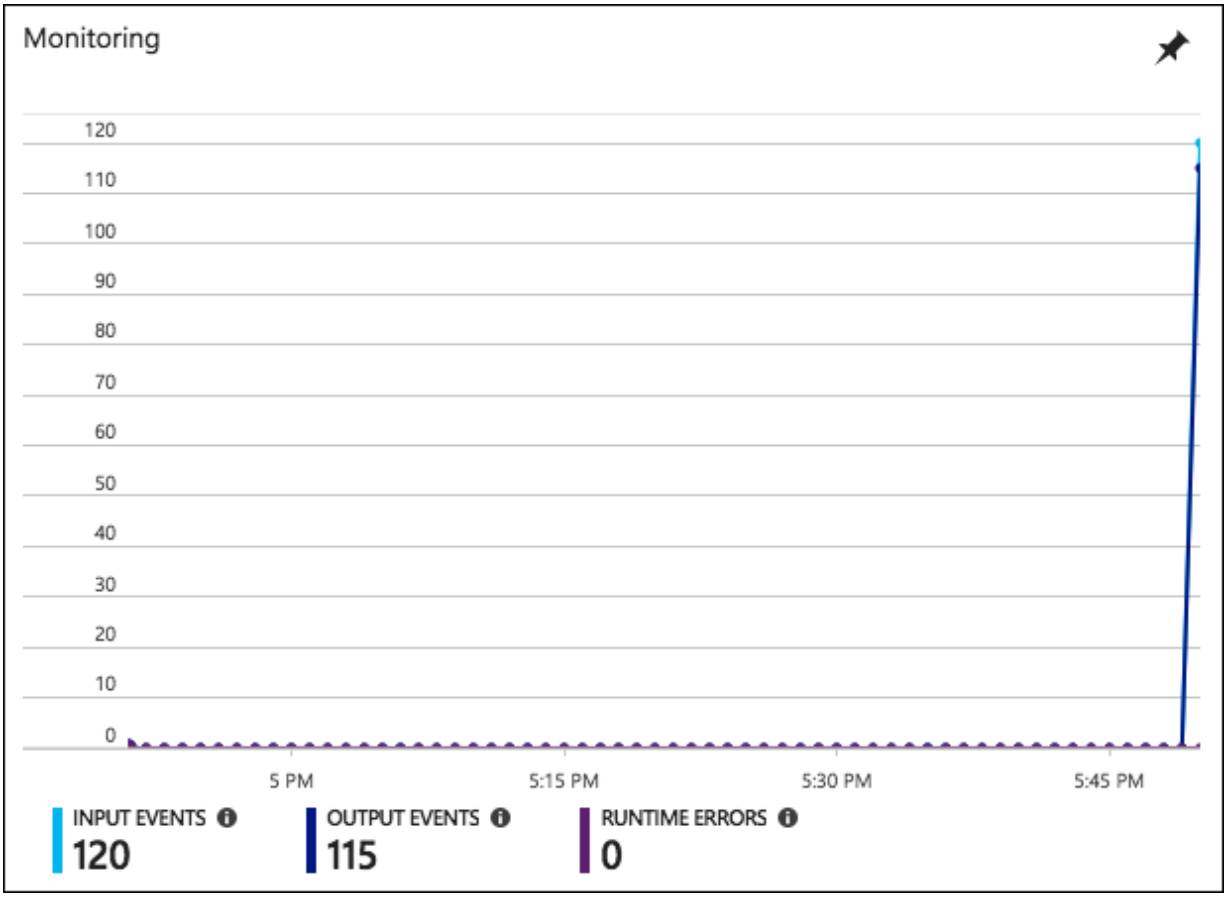
15. In the **Start job** blade, select **Now** (the job will start processing messages from the current point in time onward).



16. Select **Start**.

17. Allow your Stream Analytics Job a few minutes to start.

18. Once the Stream Analytics Job has successfully started, verify that you are showing a non-zero amount of **Input Events** on the **Monitoring** chart on the **Overview** blade. You may need to reconnect your devices on the **Smart Meter Simulator** and let it run for a while to see the events.

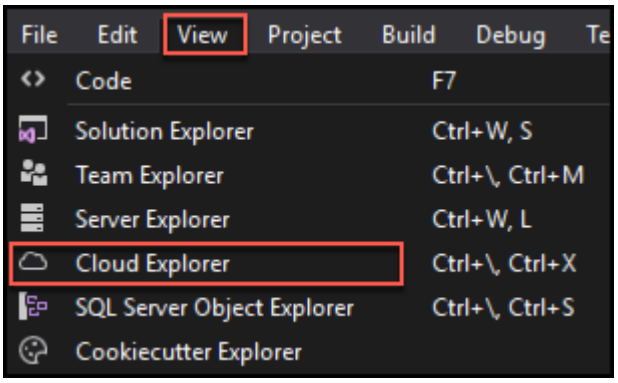


Task 3: Verify CSV files in blob storage

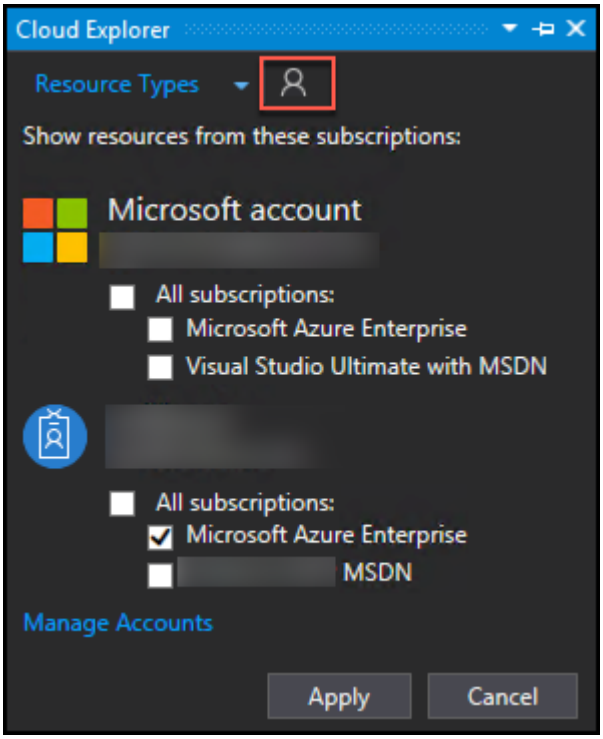
In this task, we are going to verify that the CSV file is being written to blob storage.

> **Note**: This can be done via Visual Studio or using the Azure portal. For this lab, we will perform the task using Visual Studio.

1. Within **Visual Studio** on your **Lab VM**, select the **View** menu, then select **Cloud Explorer**.



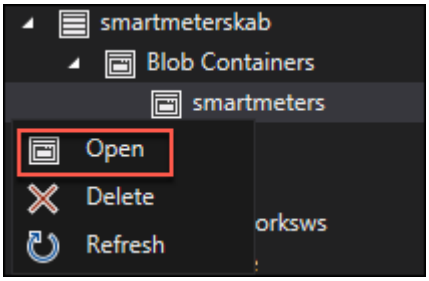
2. In **Cloud Explorer**, select **Account Management**, and connect to your Microsoft Azure Subscription.



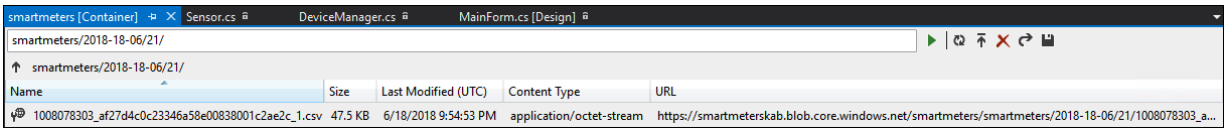
3. If prompted, sign into your Azure account.

4. Allow Cloud Explorer about 30 seconds to load your subscription resources.

5. Expand your **Azure subscription**, then expand **Storage Accounts**, expand the **smartmetersSUFFIX** storage account, then expand the **Blob containers** node, then right-click the **smartmeters** container, and select **Open**. It may take a few moments to load your storage accounts.



6. Verify files are being written to Blob storage (the files should be located underneath the **smartmeters** container).

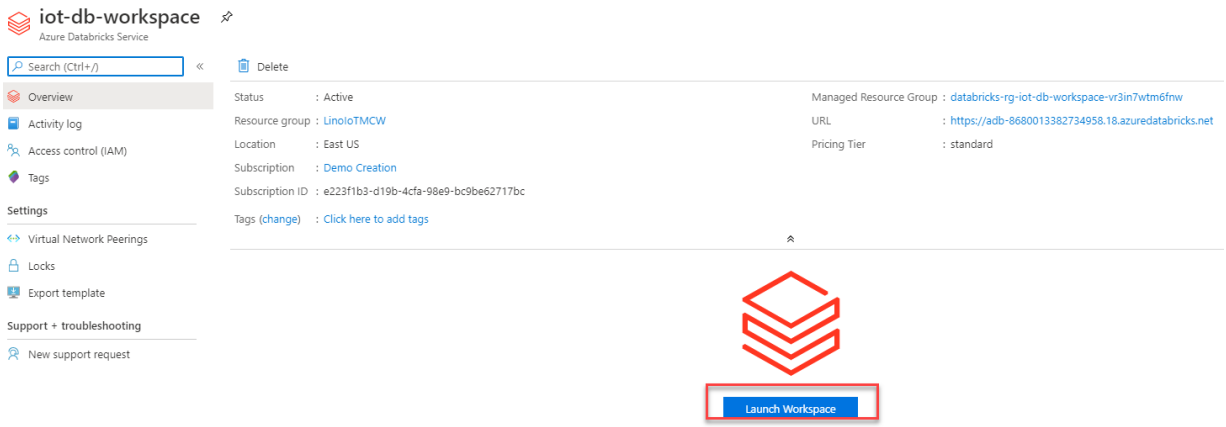


Task 4: Process with Spark SQL

In this task, you will create a new Databricks notebook to perform some processing and visualization of the cold path data using Spark.

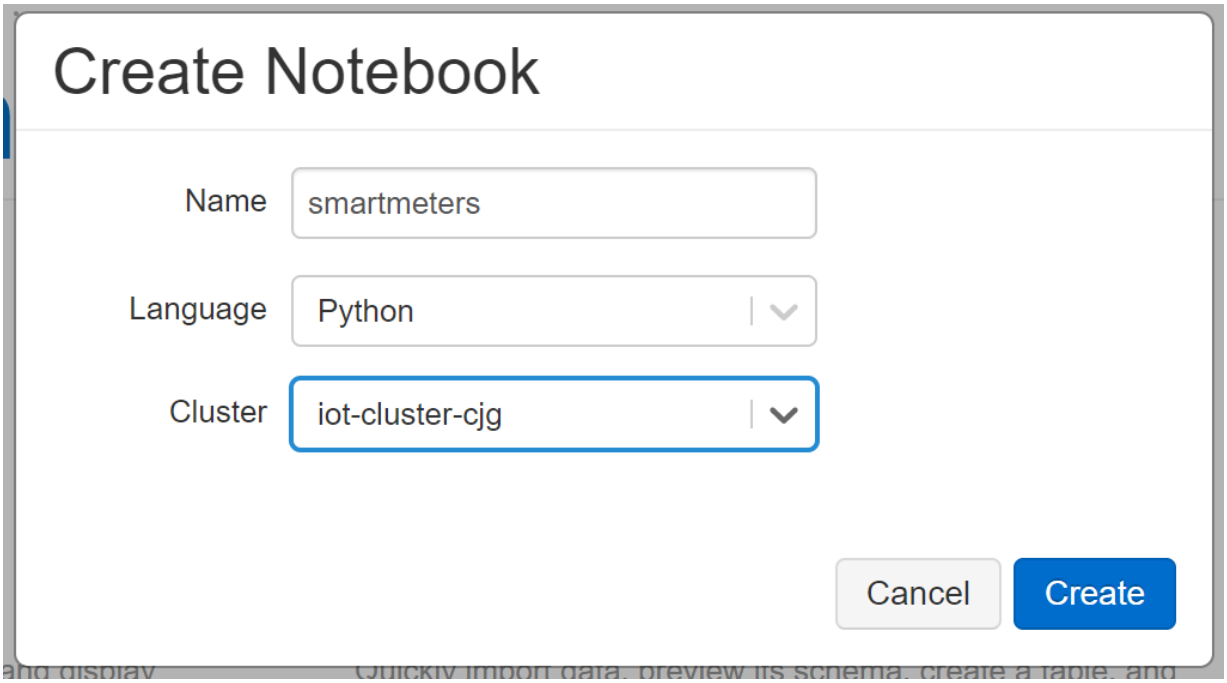
> **Note**: The complete Databricks notebook can be found in the Databricks-notebook folder of the GitHub repo associated with this hands-on lab, should you need to reference it for troubleshooting.

1. In the Azure portal, navigate to the **Azure Databricks** resource you created in the Before the hands-on lab setup guide exercises, and select **Launch Workspace**.



2. On the **Azure Databricks** landing page, create a new notebook by selecting **New Notebook** under **Common Tasks**.

3. In the **Create Notebook** dialog, enter `smartmeters` as the **Name** and select **Python** as the **Language**, then select **Create**.



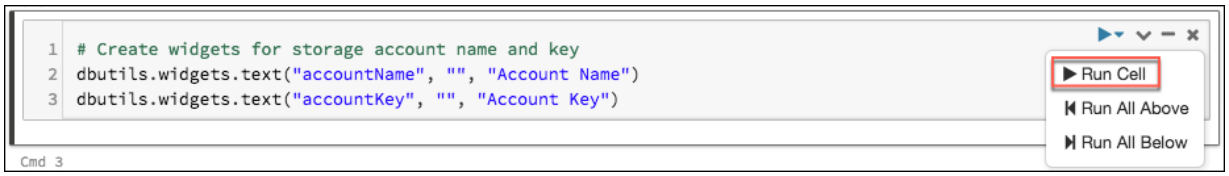
> **Note**: If your cluster is stopped, you can select the down arrow next to your attached cluster name, and select Start Cluster from the menu, then select Confirm when prompted.

4. In the **first cell** of your **Databricks notebook** (referred to as a paragraph in notebook jargon), enter the following **Python code** that creates widgets in the notebook for entering your **Azure storage account name** and **key**.

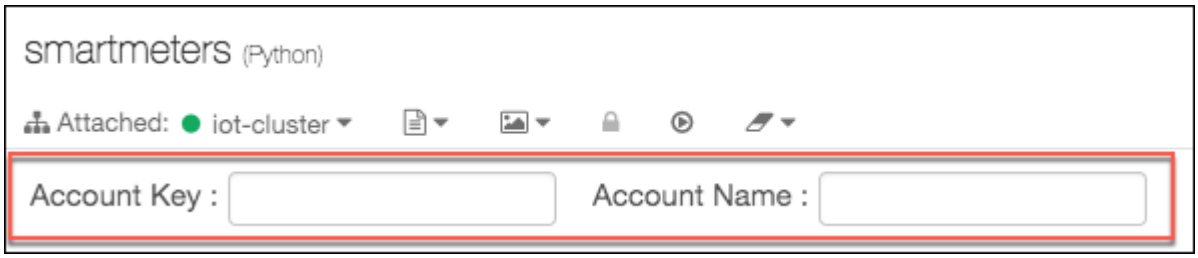
```
# Create widgets for storage account name and key
dbutils.widgets.text("accountName", "", "Account Name")
dbutils.widgets.text("accountKey", "", "Account Key")
```

> **Note**: Make sure to be aware of any indents\tabs. Python treats indents\tabs with specific syntactical meaning.

5. Now, select the **Run** button on the right side of the cell and select **Run cell**.



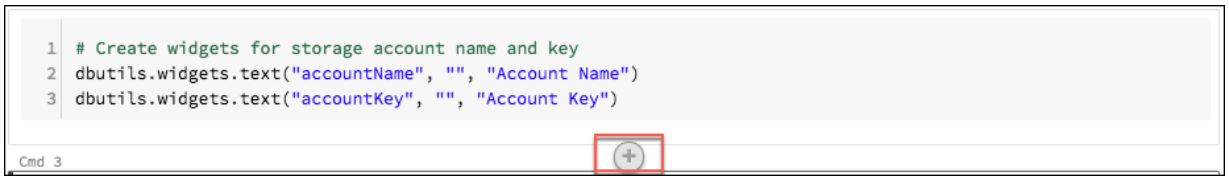
6. When the cell finishes executing, you will see the **Account Key** and **Account Name** widgets appear at the top of the notebook, just below the toolbar.



7. You will also notice a message at the bottom of the cell, indicating that the cell execution completed, and the amount of time it took.

8. Enter your **Azure Storage account key** into the **Account Key widget text box**, and your **Azure storage account name** into the **Account Name widget text box**. These values can be obtained from the **Access keys** blade in your storage account.

9. At the bottom of the **first cell**, select the + button to insert a new cell below it.



10. In the **new cell**, paste the following code that will assign the values you entered into the widgets you created above into variables that will be used throughout the notebook.

```
# Get values entered into widgets
accountName = dbutils.widgets.get("accountName")
accountKey = dbutils.widgets.get("accountKey")
```

11. Run the cell.

12. **Insert a new cell** into the notebook, and paste the following code to mount your blob storage account into Databricks File System (DBFS), then **run** the cell.

```
# Mount the blob storage account at /mnt/smartmeters. This assumes your containe
if not any(mount.mountPoint == '/mnt/smartmeters' for mount in dbutils.fs.mounts
  dbutils.fs.mount(
  source = "wasbs://smartmeters@" + accountName + ".blob.core.windows.net/smartm
  mount_point = "/mnt/smartmeters",
  extra_configs = {"fs.azure.account.key." + accountName + ".blob.core.windows.n
```

> **Note**: Mounting Azure Blob storage directly to DBFS allows you to access files as if they were on the local file system. Once your blob storage account is mounted, you can access them with Databricks Utilities, `dbutils.fs` commands.

13. **Insert a new cell** and paste the code below to see how **dbutils.fs.ls** can be used to list the files and folders directly below the **smartmeters** folder.

```
# Inspect the file structure
display(dbutils.fs.ls("/mnt/smartmeters/"))
```

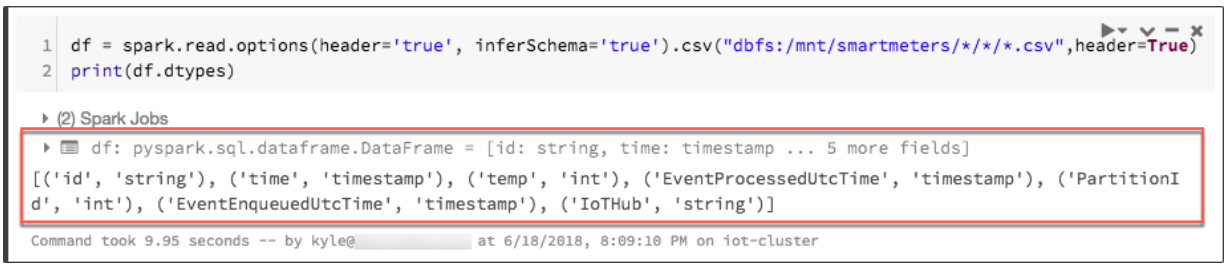
14. **Run** the cell.

15. You know from inspecting the files in the storage container that the files are contained within a folder structure resembling, **smartmeters/YYYY-MM-DD/HH**. You can use wildcards to obfuscate the date and hour folders, as well as the file names, and access all the files in all the folders. **Insert another cell** into the notebook, paste the following code, and **run** the cell to load the data from the files in blob storage into a **Databricks Dataframe**.

```
# Create a Dataframe containing data from all the files in blob storage, regardl
df = spark.read.options(header='true', inferSchema='true').csv("dbfs:/mnt/smartm
print(df.dtypes)
```

> **Note**: In some rare cases, you may receive an error that the **dbfs:/mnt/smartmeters/*/*/*.csv** path is incorrect. If this happens, change the path in the cell to the following: **dbfs:/mnt/smartmeters/*/*/*/*/*.csv**

16. The cell above also outputs the value of the **df.dtypes** property, which is a list of the data types of the columns added to the **Dataframe**, similar to the following:



17. **Insert another cell** and run the following code to view the first 10 records contained in the **Dataframe**.

```
df.show(10)
```

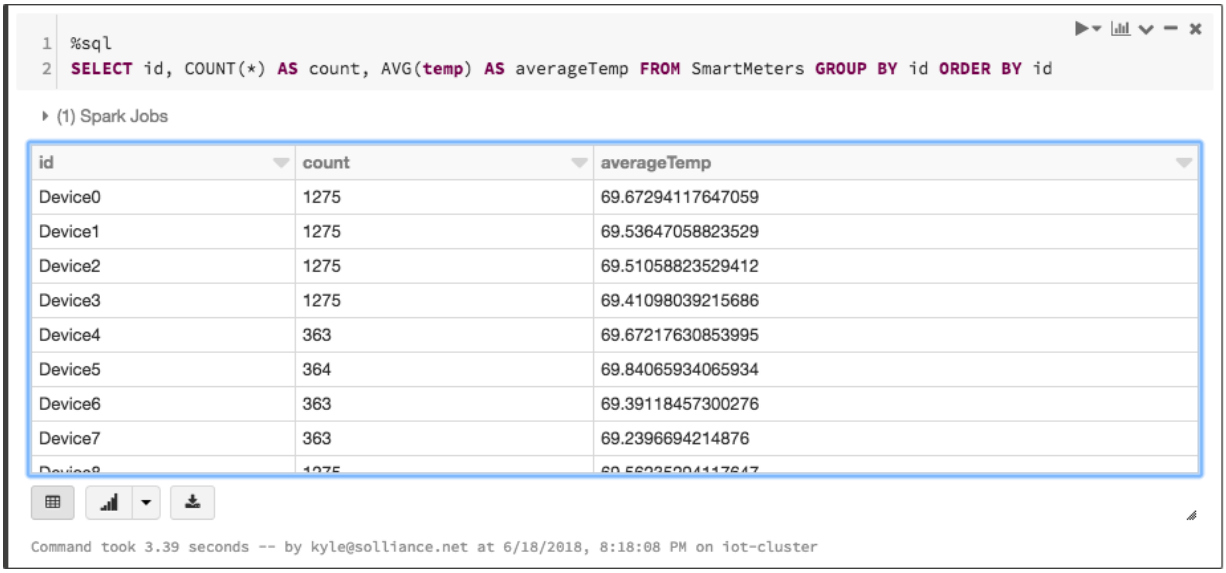
18. Now, you can save the **Dataframe** to a **global table** in **Databricks**. This will make the table accessible to all users and clusters in your Databricks workspace. **Insert a new cell** and **run** the following code.

```
df.write.mode("overwrite").saveAsTable("SmartMeters")
```

19. Now, you will use the `%sql` magic command to change the language of the next cell to **SQL** from the notebook's default language, Python, then execute a SQL command to aggregate the SmartMeter data by average temperature. Paste the following code into **a new cell**, and **run** the cell.

```
%sql
SELECT id, COUNT(*) AS count, AVG(temp) AS averageTemp FROM SmartMeters GROUP BY
```

20. The output from the SQL command should resemble the following table:



21. Now, execute the same command in **a new cell**, this time using **Spark SQL** so you can save the summary data into a Dataframe. Copy and **execute** the following code into **a new cell**:

```
# Query the table to create a Dataframe containing the summary
summary = spark.sql("SELECT id, COUNT(*) AS count, AVG(temp) AS averageTemp FROM

# Save the new pre-computed table
summary.write.mode("overwrite").saveAsTable("DeviceSummary")
```

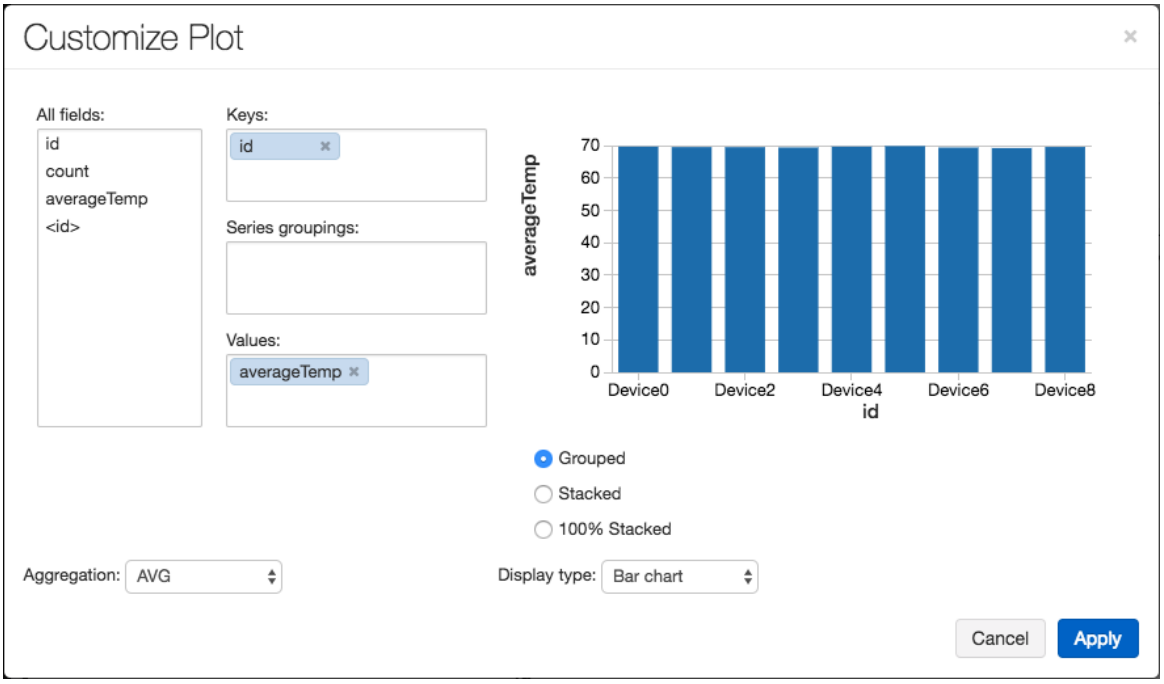
22. Next, query from this summary table by **executing** the following query in **a new cell**:

```
%sql
SELECT * FROM DeviceSummary
```

23. **Below** the results table, notice the area to change the visualization for tabular output. Select the **Bar** button, and then select **Plot Options**.

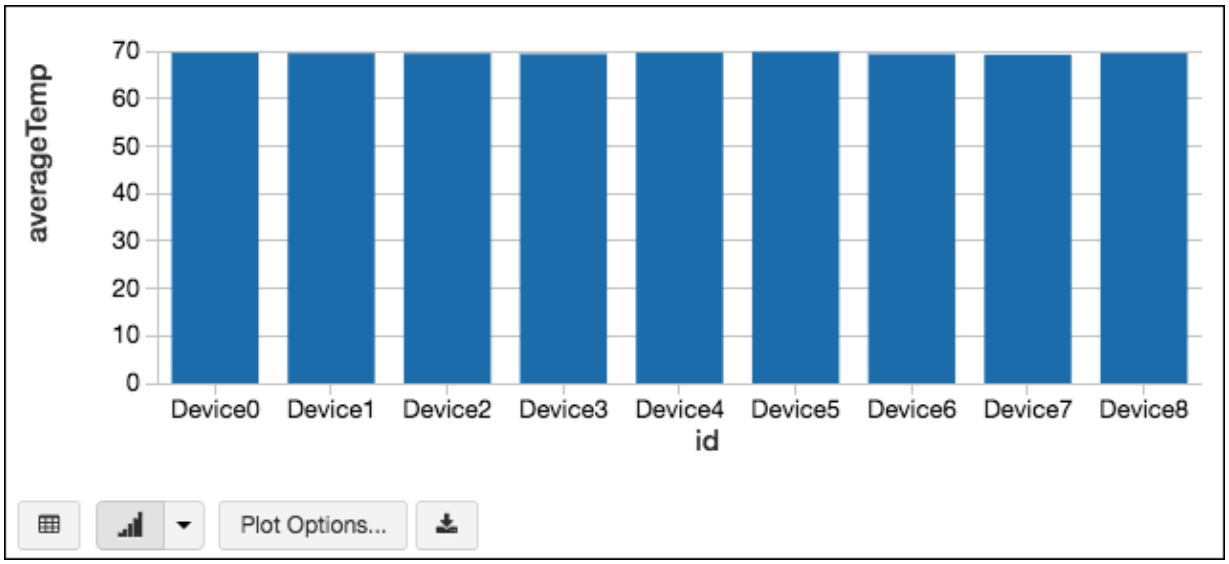
24. In the **Customize Plot** dialog, ensure the following are set:

- **Keys**: **id**
- **Values**: **averageTemp**
- **Aggregation**: Select **AVG**
- Select **Grouped** as the chart type.
- **Display type**: Select **Bar chart**.



25. Select **Apply**.

26. Observe the results graphed as a column chart, where each column represents a device's average temperature.



Exercise 5: Sending commands to the IoT devices

Duration: 20 minutes

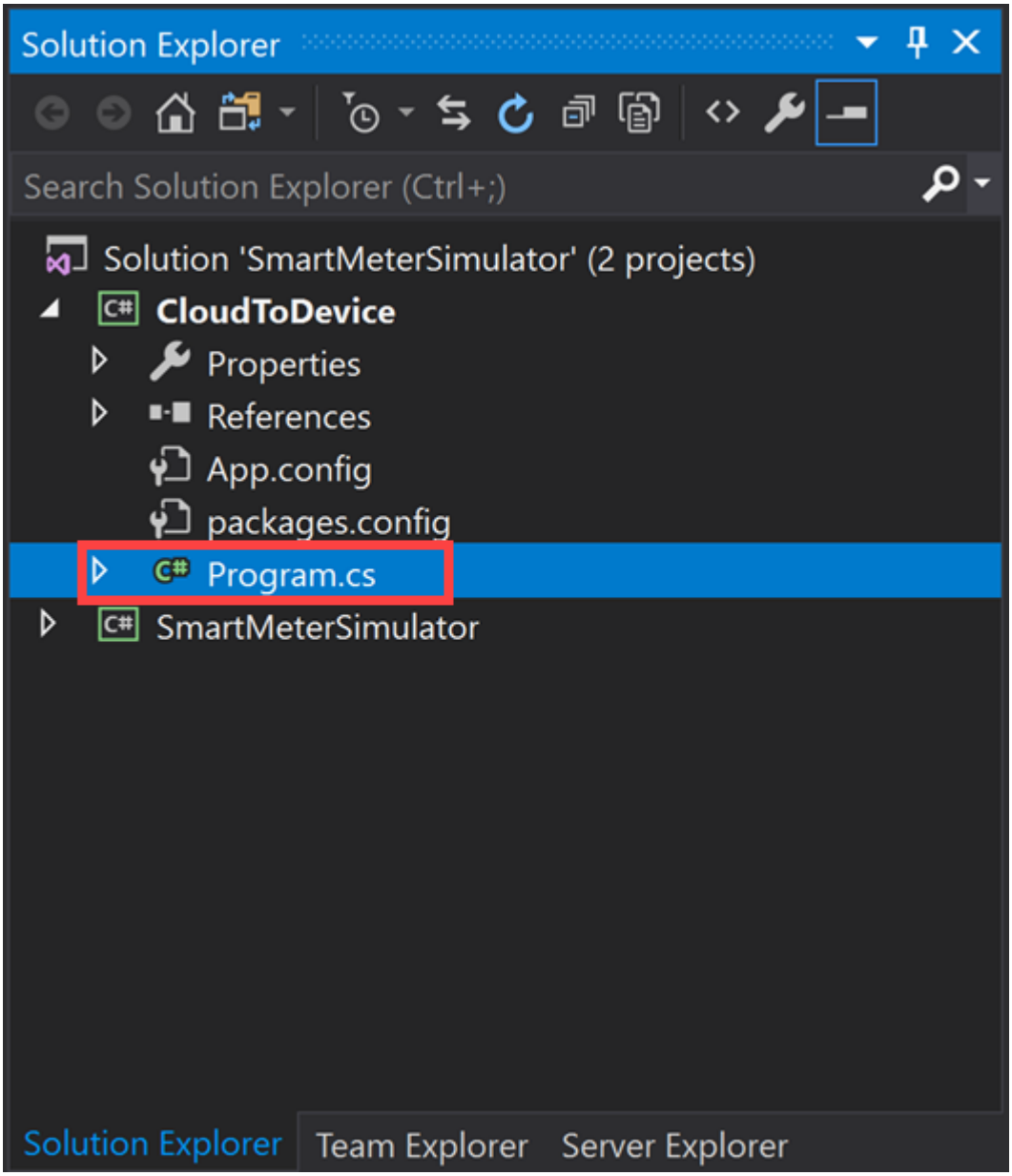
Fabrikam would like to send commands to devices from the cloud in order to control their behavior. In this exercise, you will send commands that control the temperature settings of individual devices.

Task 1: Add your IoT Hub connection string to the CloudToDevice console app

This console app is configured to connect to IoT Hub using the same connection string you use in the SmartMeterSimulator app. Messages are sent from the console app to IoT Hub, specifying a device by its ID, for example **Device1**. IoT Hub then transmits that message to the device when it is connected. This is called a **cloud-to-device message**. The console app is not directly connecting to the device and sending it the message. All messages flow through IoT Hub where the connections and device state are managed.

1. Return to the **SmartMeterSimulator** solution in **Visual Studio** on your **Lab VM**.

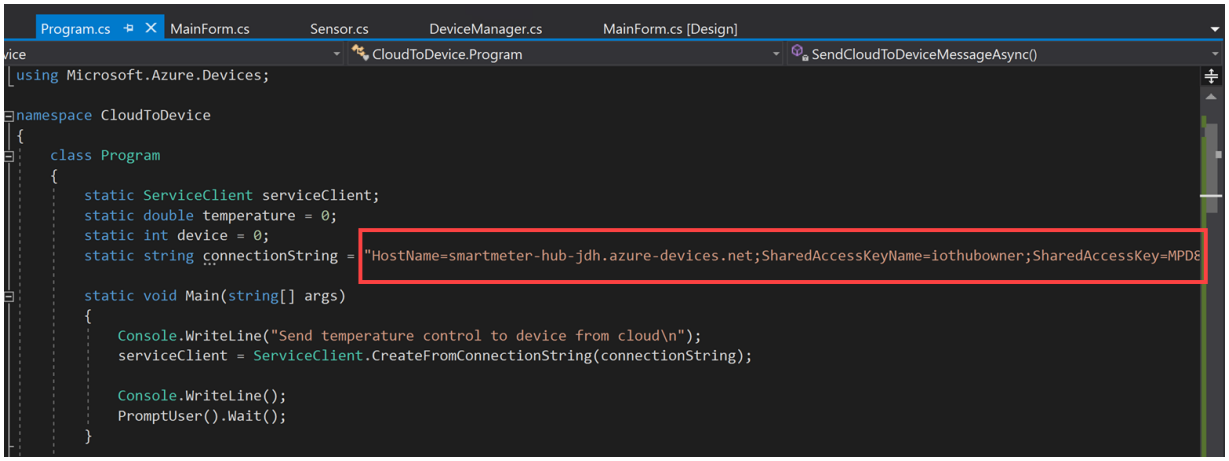
2. In the **Solution Explorer**, expand the **CloudToDevice** project and double-click **Program.cs** to open it. (If the Solution Explorer is not in the upper-right corner of your Visual Studio instance, you can find it under the View menu in Visual Studio.)



3. Replace **YOUR-CONNECTION-STRING** on line 15 with your IoT Hub connection string. This is the same string you added to the Main form in the SmartMeterSimulator earlier. The line you need to update looks like this:

```
static string connectionString = "YOUR-CONNECTION-STRING";
```

After updating, your **Program.cs** file should look similar to the following:

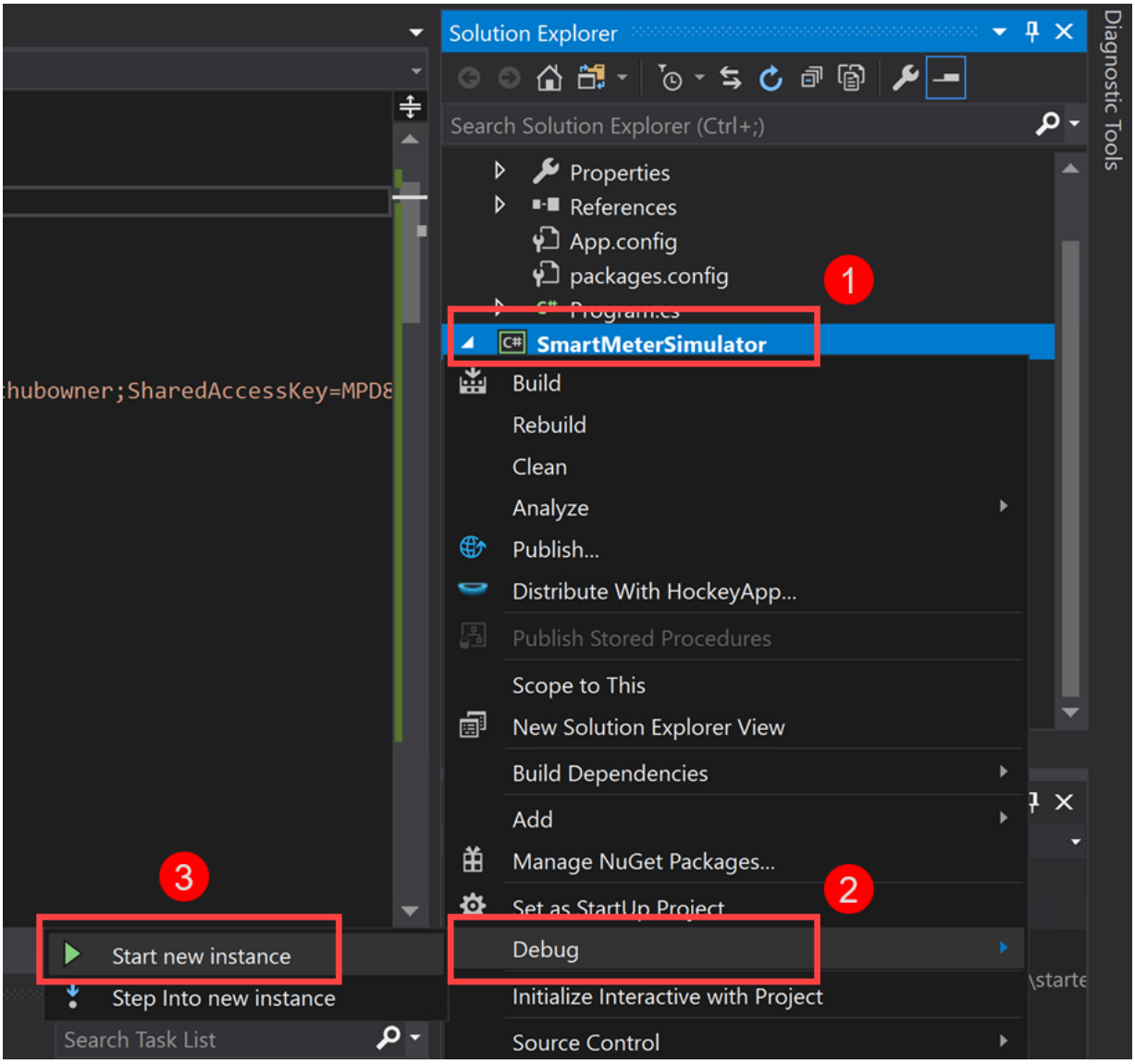


4. Save the file.

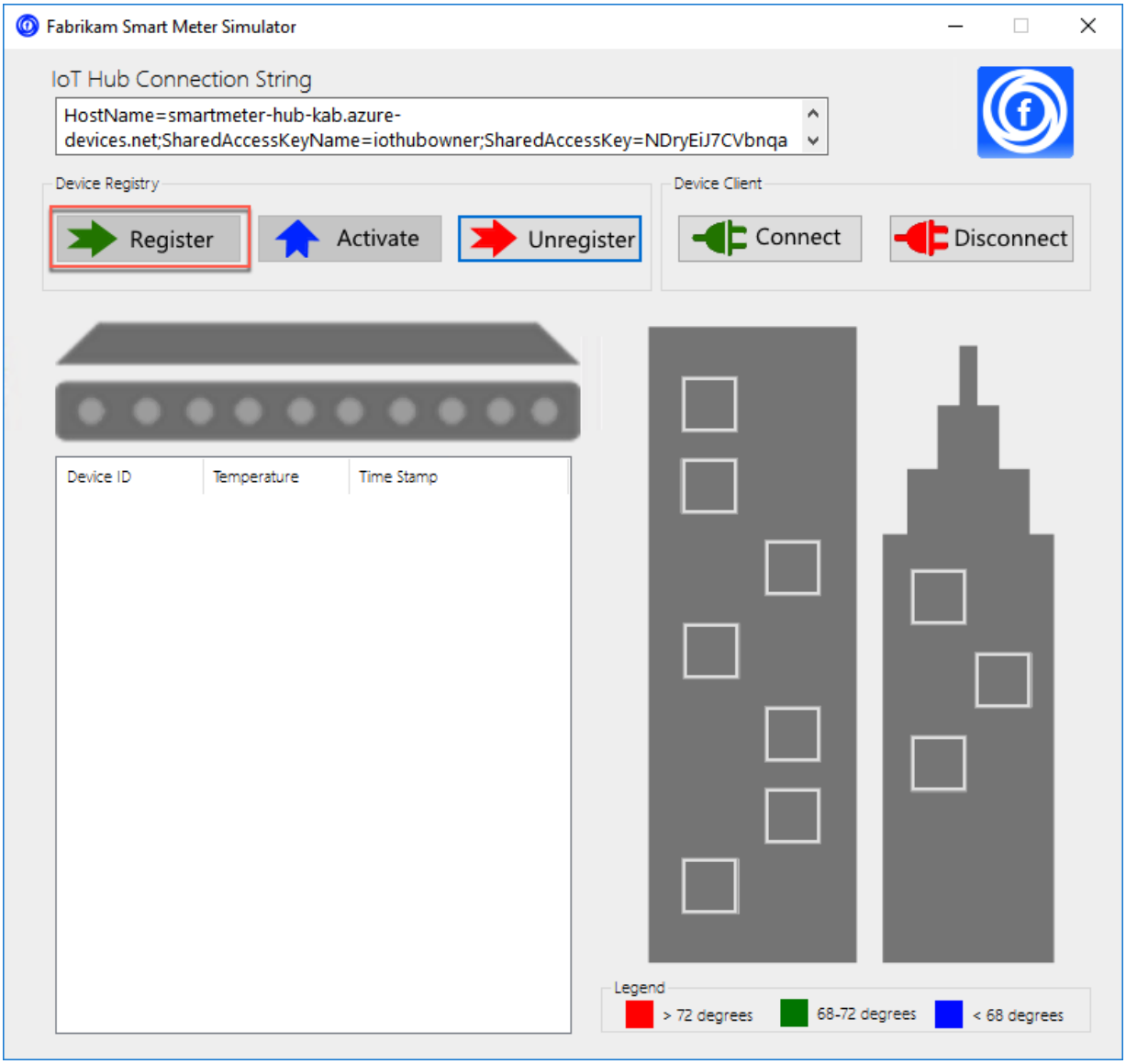
Task 2: Run the device simulator

In this task, you will register, activate, and connect all devices. You will then leave the simulator running so that you can launch the console app and start sending cloud-to-device messages.

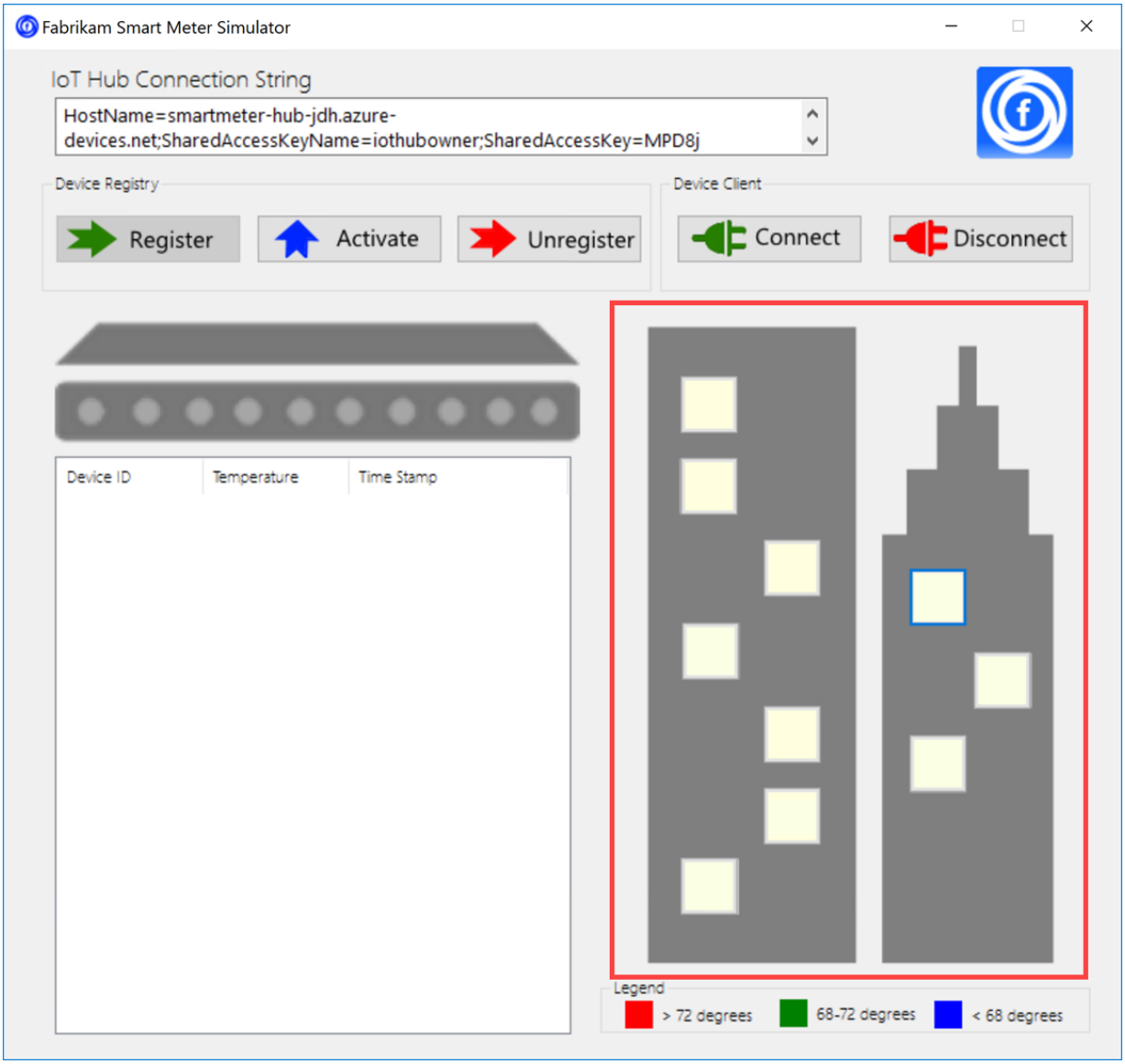
1. Within the **SmartMeterSimulator** Visual Studio solution, right-click the **SmartMeterSimulator** project, select **Debug**, then select **Start new instance** to run the device simulator.



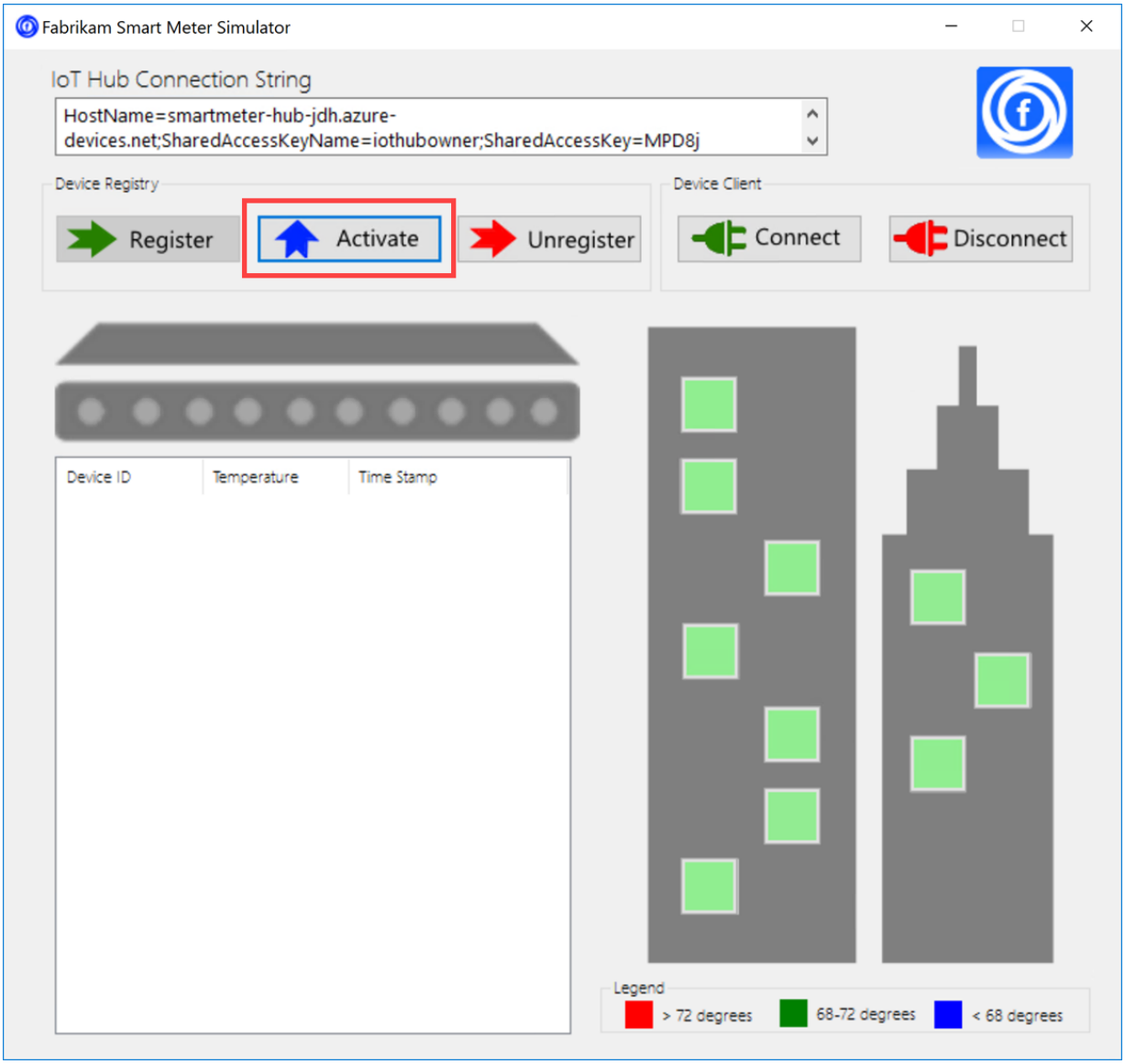
2. Select **Register** on the **Smart Meter Simulator** dialog, which should cause the windows within the building to change from black to gray.



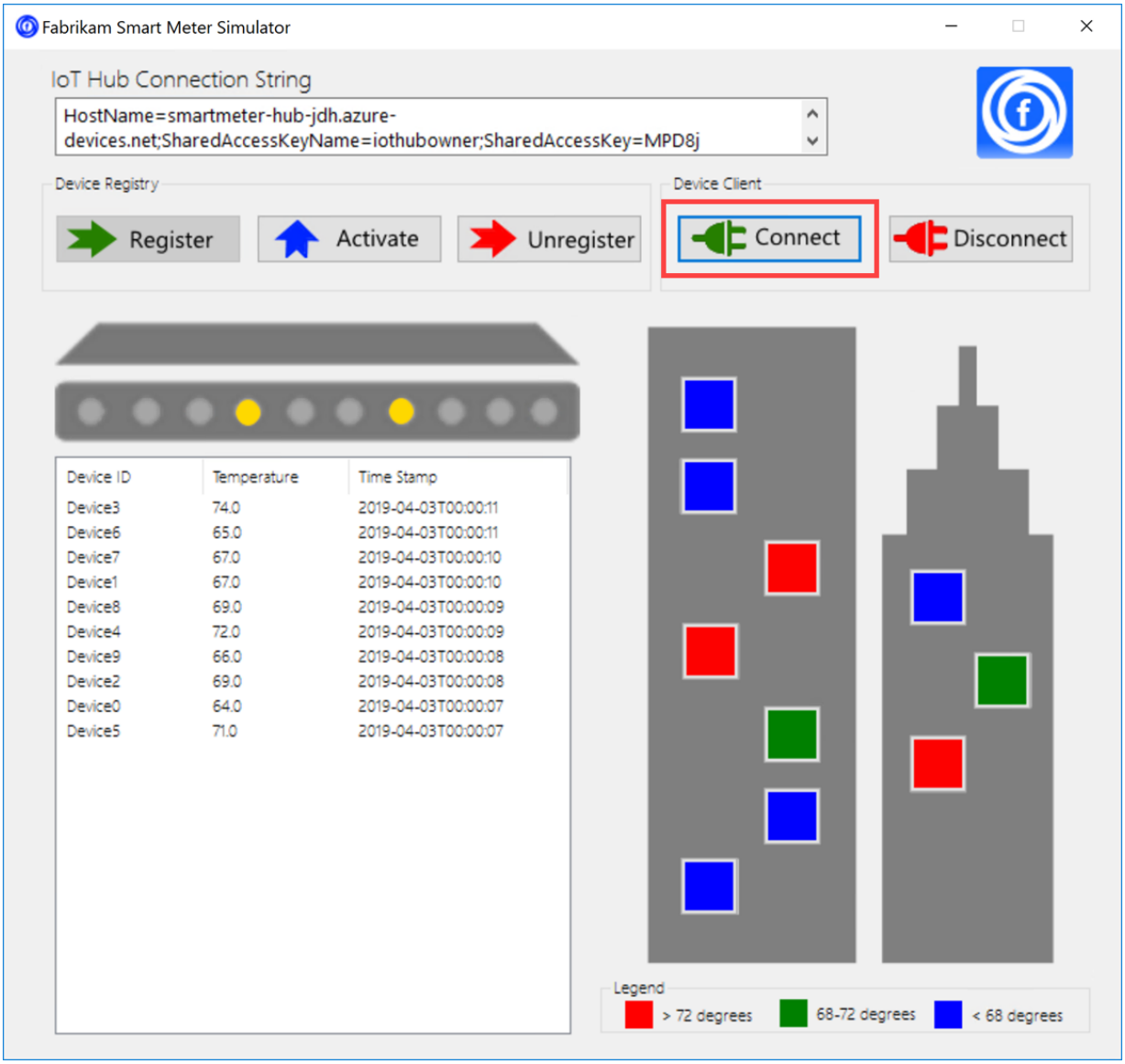
3. Select **all** of the windows. Each represents a device for which you want to simulate device installation. The selected windows should turn yellow.



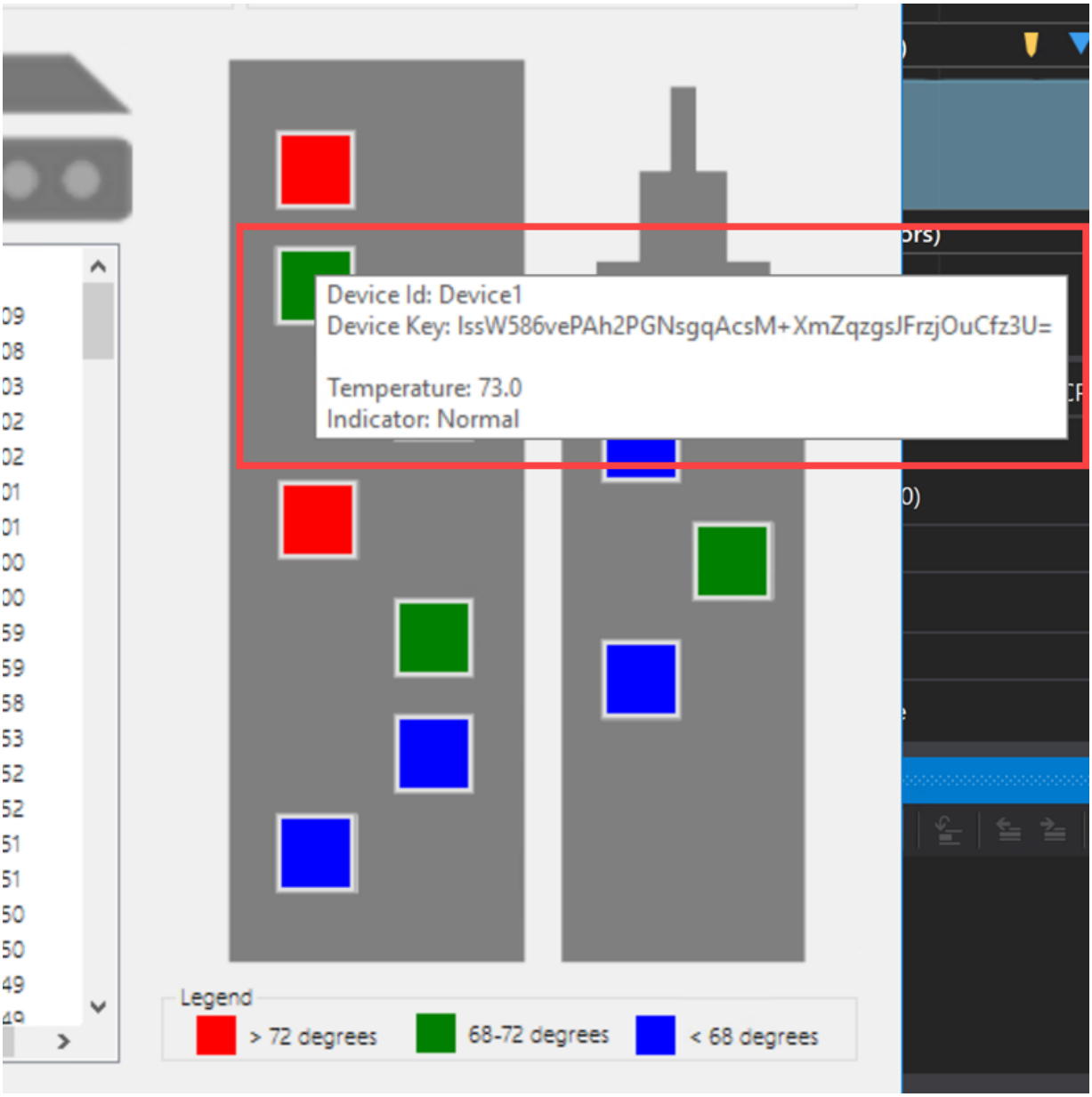
4. Select **Activate** to simulate changing the device status from disabled to enabled in the IoT Hub Registry. The selected windows should turn green.



5. Select **Connect**. Within a few moments, you should begin to see activity and the windows change color, indicating the smart meters are transmitting telemetry. The grid on the left will list each telemetry message transmitted and the simulated temperature value.



6. Hover over one of the windows. You will see a dialog display information about the associated device, including the Device ID (in this case, **Device1**), Device Key, Temperature, and Indicator. The legend on the bottom shows the indicator displayed for each temperature range. The Device ID is important when sending cloud-to-device messages, as this is how we will target a specific device when we remotely set the desired temperature. Keep the Device ID values in mind when sending the messages in the next task.



7. Allow the smart meter to continue to run.

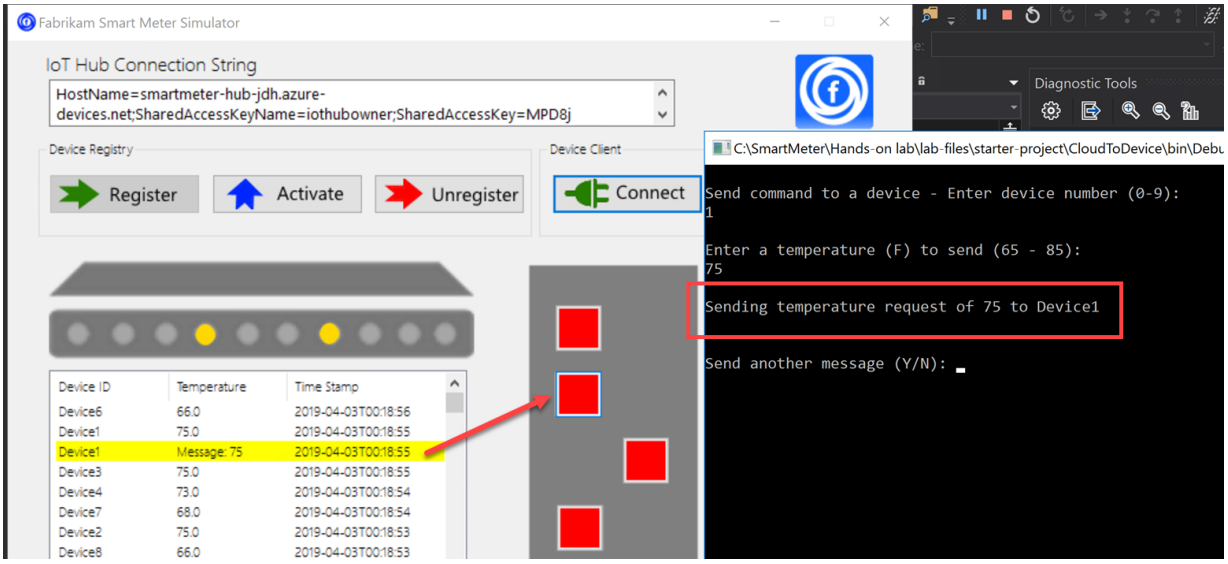
Task 3: Run the console app and send cloud-to-device messages

In this task, you will run the console app to send desired temperature settings to specific devices and observe the simulated device receiving and reacting to the message.

1. Within the **SmartMeterSimulator** Visual Studio solution, right-click the **CloudToDevice** project, select **Debug**, then select **Start new instance** to run the console app.

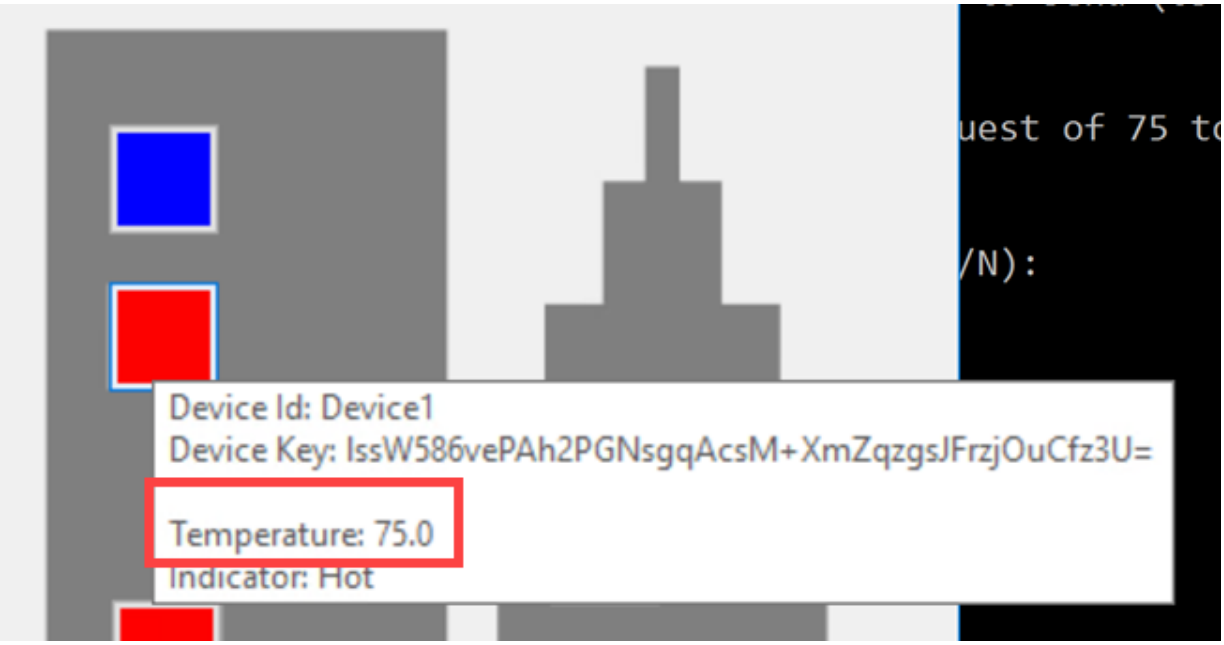
2. In the **console window**, enter a **device number** when prompted. Accepted values are 0-9, since there are 10 devices whose IDs begin with 0. You can hover over the windows in the **Smart Meter Simulator** to view the Device IDs. When you enter a number, such as `5`, then a message will be sent to **Device5**.

3. Now enter a **temperature value** between 65 and 85 degrees (F) when prompted. If you set a value above 72 degrees, the window will turn red. If the value is set between 68 and 72 degrees, it will turn green. Values below 68 degrees will turn the window blue. Once you set a value, the device will remain at that value until you set a new value, rather than randomly changing.



If you run the **Smart Meter Simulator** side-by-side with the **console app**, you can observe the message logged by the Smart Meter Simulator within seconds. This message appears with a yellow background and displays the temperature request value sent to the device. In our case, we sent a request of 75 degrees to Device1. The console app indicates that it is sending the temperature request to the indicated device.

4. Hover over the device to which you sent the message. You will see that its temperature is set to the value you requested through the console app.



5. In the console window, you can enter `Y` to send another message. Experiment with setting the temperature on other devices and observe the results.

After the hands-on lab

Duration: 10 mins

In this exercise, you will delete any Azure resources that were created in support of the lab. You should follow all steps provided after attending the Hands-on lab to ensure your account does not continue to be charged for lab resources.

Task 1: Delete the resource group

1. Using the Azure portal, navigate to the Resource group you used throughout this hands-on lab by selecting Resource groups in the left menu.

2. Search for the name of your research group and select it from the list.

3. Select Delete in the command bar and confirm the deletion by re-typing the Resource group name, and selecting Delete.

You should follow all steps provided *after* attending the Hands-on lab.