Quick start guide

Cloud connectivity to Azure IoT Hub

Connecting to Azure IoT hub via Azure SDK

Beijer ELECTRONICS

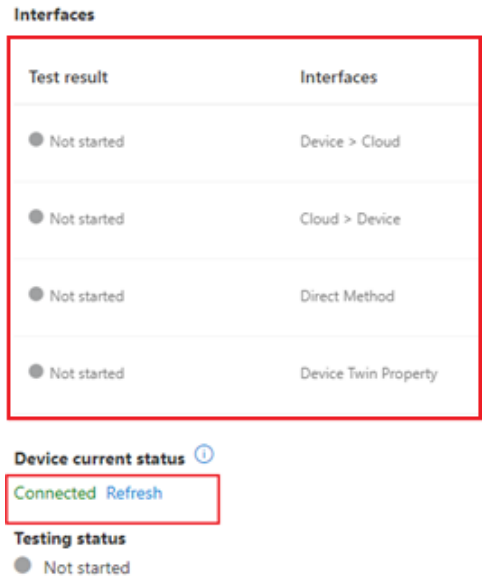
1 Function and area of use

This document provides guidelines for working with Azure IoT Hub Device Provisioning Service (DPS) and Azure IoT hub service.

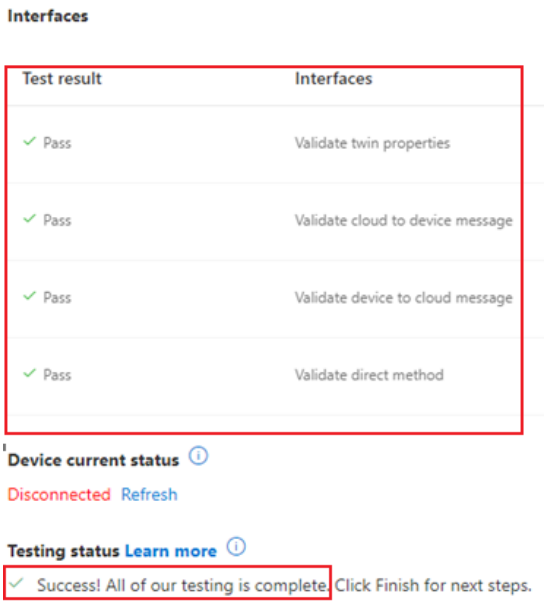
This document explains how to enable zero-touch, just-in-time provisioning to the right IoT hub and connect to Azure IoT Hub on the iX Runtime platform using Azure IoT SDKs and how to interact with IoT Hub like sending D2C messages, receiving and read C2D messages, report to Device Twins desired properties and direct method invocations.

This iX Runtime application sample also has been proved workable through the tests of the Azure Certified Device program. The test items are as follows:

1. Connecting to IoT Hub via the Device Provisioning Service (DPS)



2. Device-to-Cloud message
3. Cloud-to-Device message
4. Direct Method
5. Device Twins



2 About this start-up document

This Startup document should not be considered as a complete manual. It is an aid to be able to Startup a normal application quickly and easily.

Use the following hardware, software, drivers and utilities in order to obtain a stable application:

In this document we have used following software and hardware

- iX Developer 2.40 SP4 / SP5
- PC iX Runtime, including C2 Series

For further information refer to

- GitHub - Azure/azure-iot-sdk-csharp: A C# SDK for connecting devices to Microsoft Azure IoT services
- Use the Azure portal to create an IoT Hub | Microsoft Docs
- IoT Hub DPS Demo (microsoft.com)
- Azure IoT Hub device-to-cloud options | Microsoft Docs
- Azure IoT Hub cloud-to-device options | Microsoft Docs
- Azure IoT Hub Device Provisioning Service (DPS) Documentation | Microsoft Docs
- Download Azure SDKs and Tools | Microsoft Azure
- Install and use Azure IoT explorer | Microsoft Docs
- Beijer Electronics knowledge database, HelpOnline

This document and other Startup documents can be obtained from our homepage. Please use the address *support.europe@beijerelectronics.com* for feedback about our start-up documents.

3 Table of Contents

4 Downloading the Azure SDK

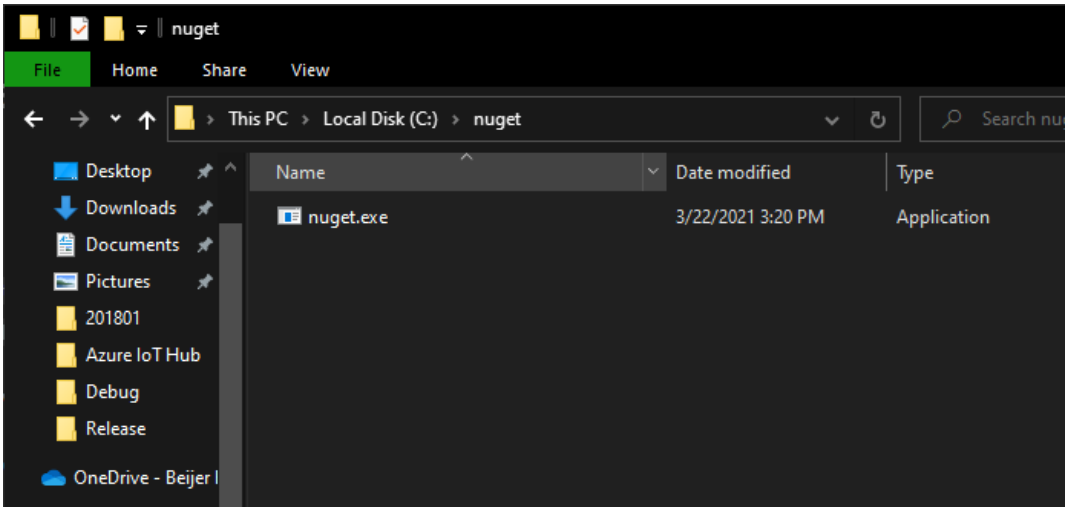
This sample shows how to use the Azure IoT SDK for .NET to connect to Azure IoT Hub. First of all, we need to get the SDK from NuGet.

4.1 Prerequisites

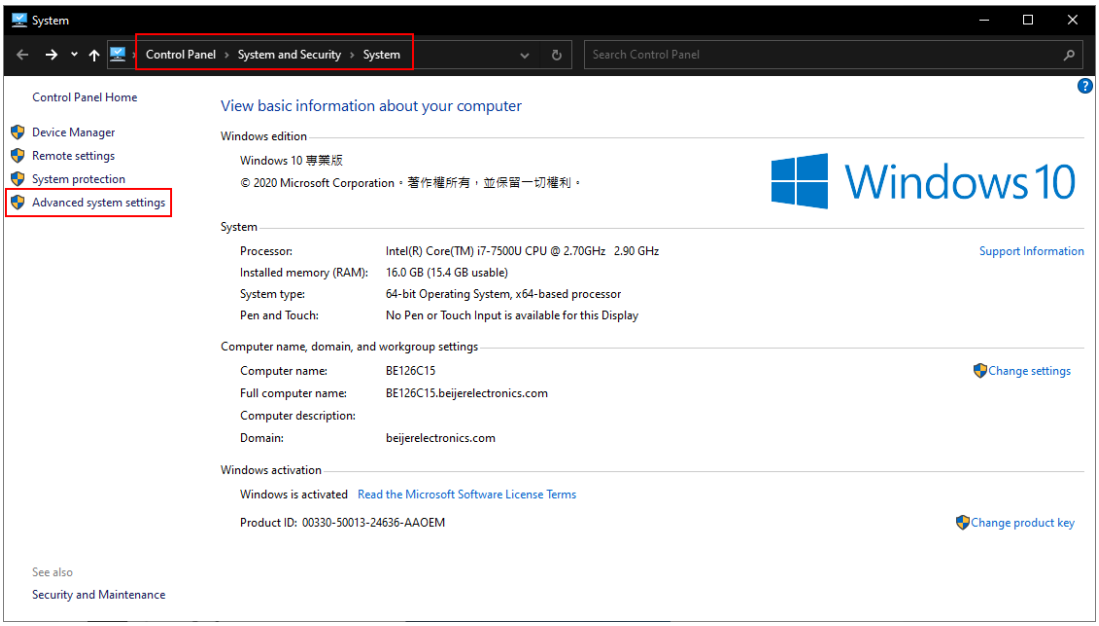
There are several ways to download a package from NuGet. For more information about it, please refer to "Install NuGet client tools". However, we only want to use the libraries (assemblies) in an iX Runtime application. It is better to get the package via nuget.exe CLI, please refer to "Manage packages using the nuget.exe CLI".

The article mentions that "Install the nuget.exe CLI by downloading it from nuget.org, saving that .exe file to a suitable folder, and adding that folder to your PATH environment variable."

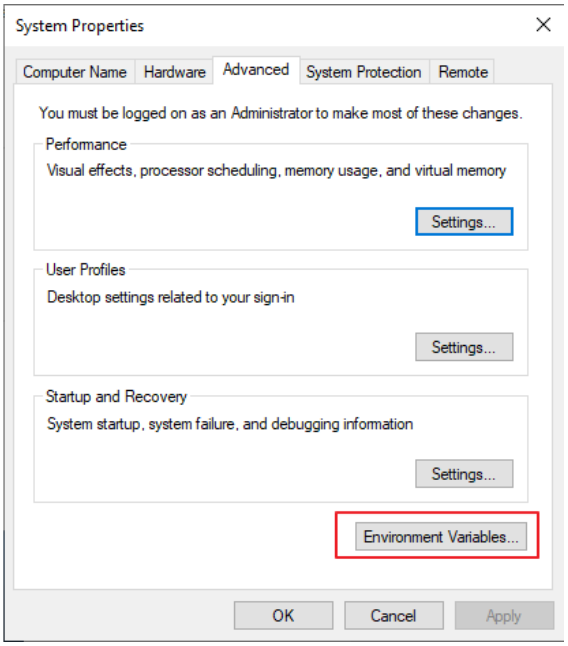
1. After downloading nuget.exe CLI, put it to a suitable folder. For example, here we created a folder called nuget in C disk to demonstrate.



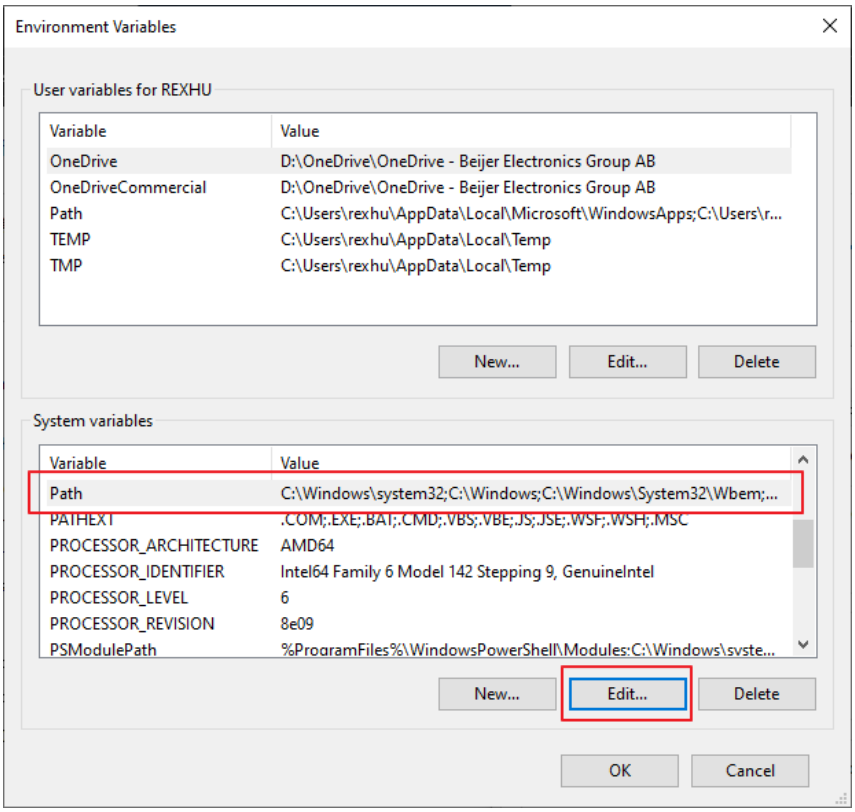
2. Adding the folder to PATH environment variable
   1. Click "Advanced system settings"



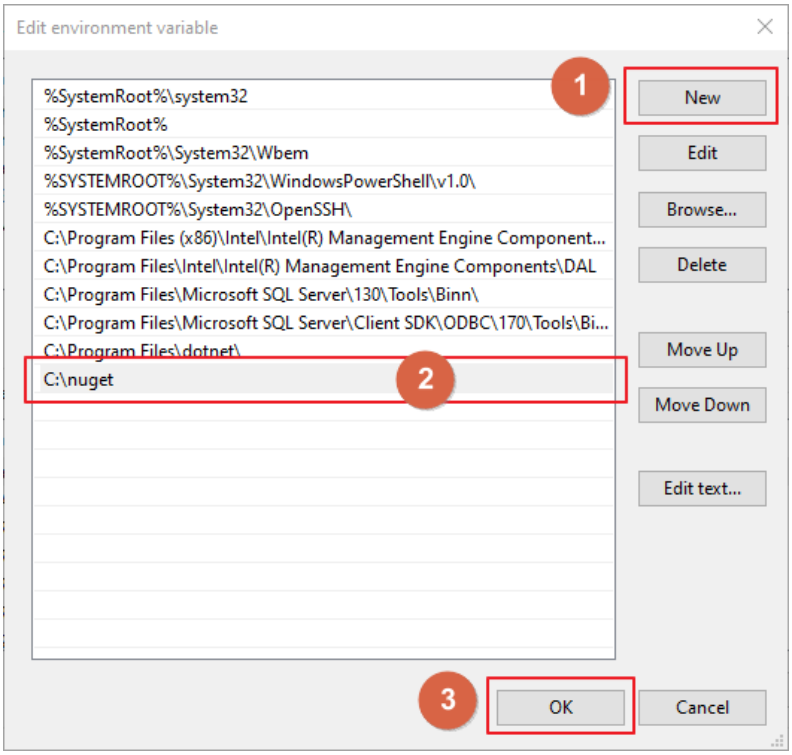
2. Click “Environment Variables” on “Advanced” tab of System Properties dialog.



3. Choose “Path” in “System variables” and click “Edit”.



4. Append the folder path to it.



4.2 Downloading the Azure IoT SDK package from nuget

Next step, we can use the "install" command to download the package.

```
nuget install <packageID> -Version <version> -OutputDirectory <path>
```

The following picture shows what libraries and their version are used in this sample. You could use a later version if you know a .NET Standard version mapping to .NET Framework versions well. (.NET Standard | Common APIs across all .NET implementations (microsoft.com))

GitHub - Azure/azure-iot-sdk-csharp: A C# SDK for connecting devices to Microsoft Azure IoT services

Recommended NuGet packages

Package Name	Release Version	Pre-release Version
For Device-to-Cloud message Microsoft.Azure.Devices.Client	nuget v1.37.2	nuget v1.38.0-preview-001
Microsoft.Azure.Devices	nuget v1.34.0	nuget v1.35.0-preview-001
Common code for Azure IoT Device and Service SDKs Microsoft.Azure.Devices.Shared	nuget v1.28.1	nuget v1.29.0-preview-001
Microsoft.Azure.Devices.Provisioning.Client	nuget v1.17.1	nuget v1.18.0-preview-001
For device provisioning Microsoft.Azure.Devices.Provisioning.Transport.Amqp	nuget v1.14.1	nuget v1.15.0-preview-001
Microsoft.Azure.Devices.Provisioning.Transport.Http	nuget v1.13.1	nuget v1.14.0-preview-001
Microsoft.Azure.Devices.Provisioning.Transport.Mqtt	nuget v1.15.1	nuget v1.16.0-preview-001
Microsoft.Azure.Devices.Provisioning.Service	nuget v1.17.1	nuget v1.18.0-preview-001
If your hardware supports TPM for the secure connection, then you can also download and use it. Microsoft.Azure.Devices.Provisioning.Security.Tpm	nuget v1.13.1	nuget v1.14.0-preview-001
Microsoft.Azure.Devices.DigitalTwin.Client	N/A	nuget v1.0.0-preview-001
Microsoft.Azure.Devices.DigitalTwin.Service	N/A	nuget v1.0.0-preview-001

For example, we created a folder called mypackages under C:\nuget\. And then, we execute the install command.

```
command prompt(cmd)>nuget install Microsoft.Azure.Devices.Client -Version
1.37.2 -OutputDirectory C:\nuget\mypackages
```

```
command prompt(cmd)>nuget install
 Microsoft.Azure.Devices.Provisioning.Client -Version 1.17.1
 -OutputDirectory C:\nuget\mypackages
```

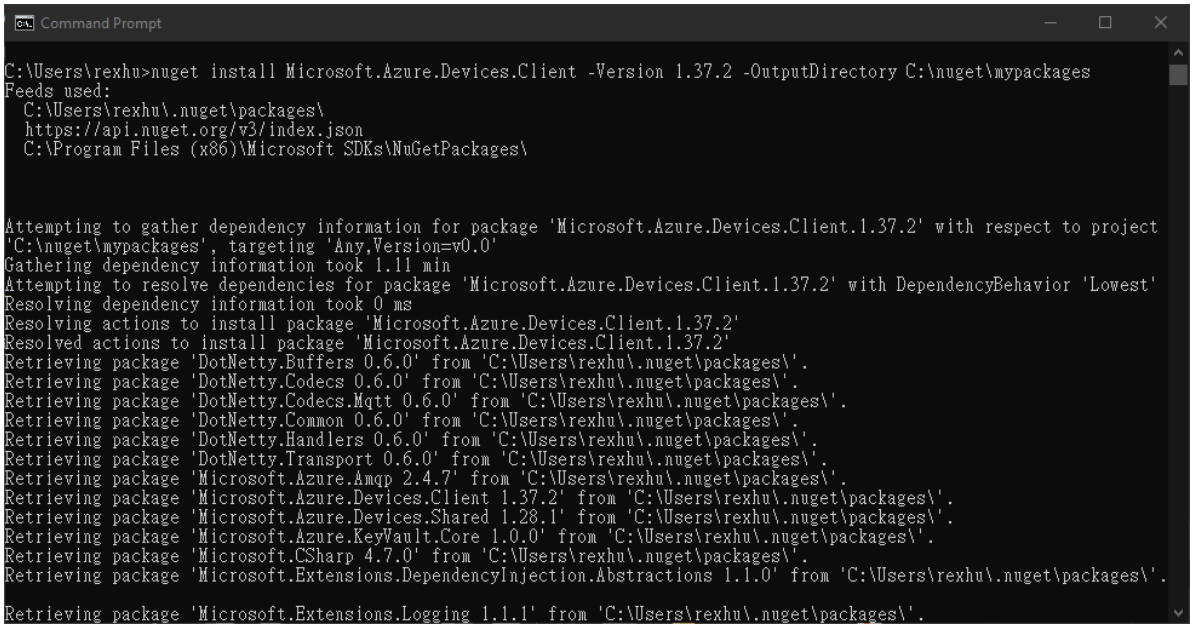
```
command prompt(cmd)>nuget install
 Microsoft.Azure.Devices.Provisioning.Transport.Amqp -Version 1.14.1
 -OutputDirectory C:\nuget\mypackages
```

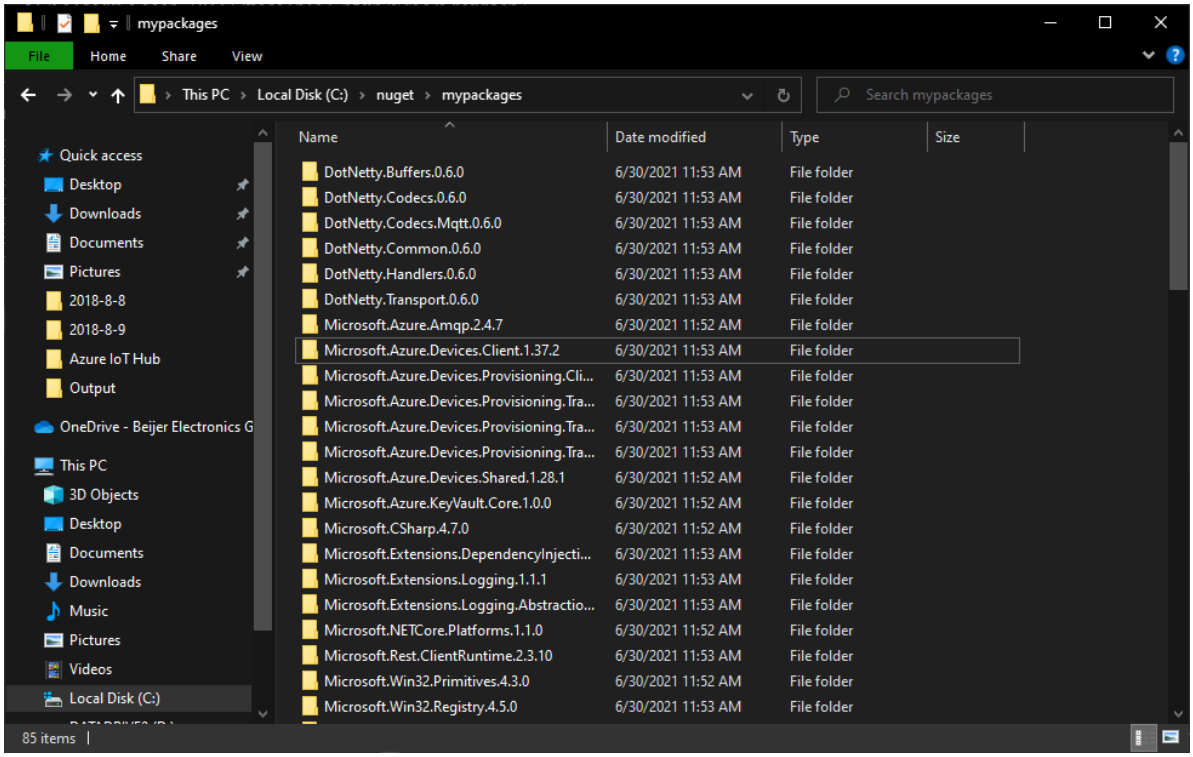
```
command prompt(cmd)>nuget install
 Microsoft.Azure.Devices.Provisioning.Transport.Http -Version 1.13.1
 -OutputDirectory C:\nuget\mypackages
```

```
command prompt(cmd)>nuget install
 Microsoft.Azure.Devices.Provisioning.Transport.Mqtt -Version 1.15.1
 -OutputDirectory C:\nuget\mypackages
```

Then, you will get the library and dependencies.

(Microsoft.Azure.Devices.Shared would be installed automatically since it is a dependency of other libraries.)



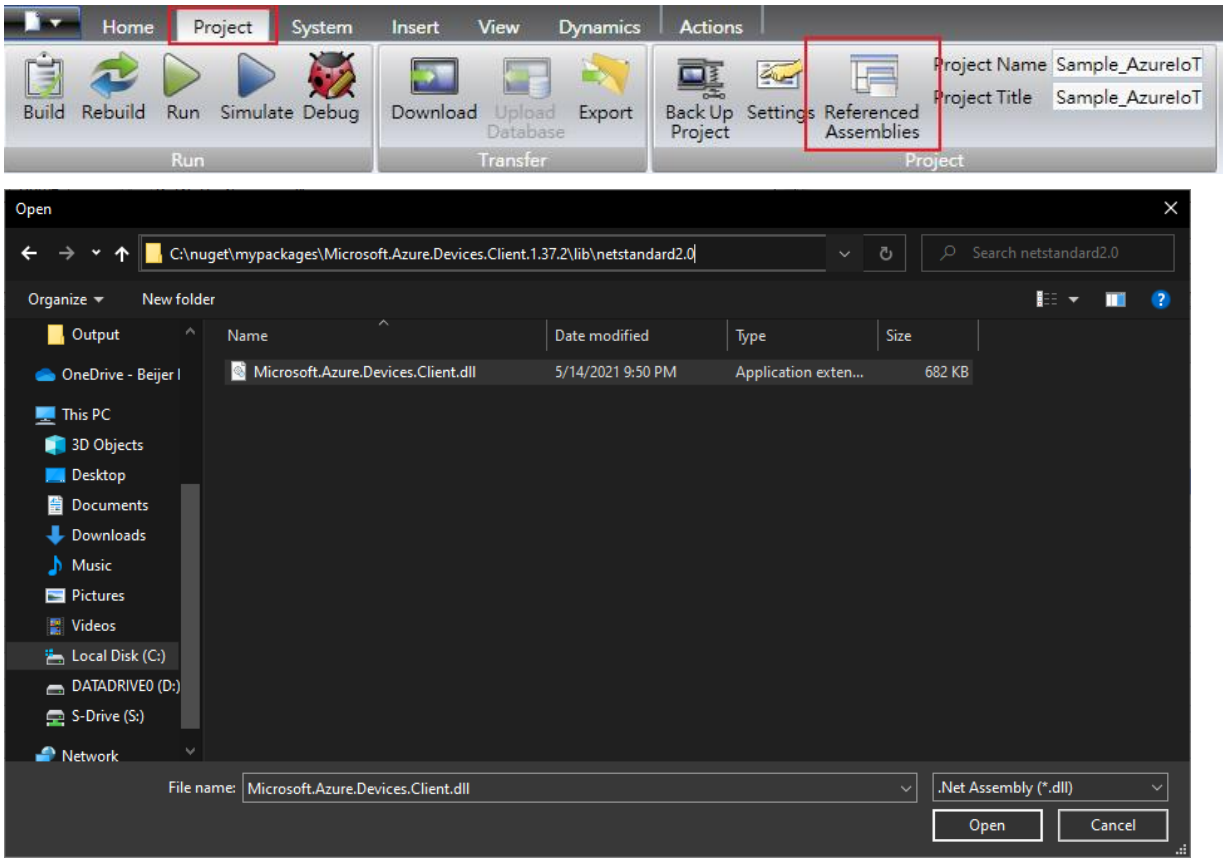


5 Use the library in an iX Runtime application

Before interacting with Azure IoT cloud services, you should reference the SDKs and dependencies assemblies.

5.1 Referenced Assemblies:

Open your iX project and click the "Referenced Assemblies" on the "Project" tab to add the references.



Following assemblies should be referenced in your iX project.

Notice: Normally, we could use .NET Standard 2.0 or an earlier version for .NET Framework 4.8. **However, referencing DotNetty.*.dll should use .NET 4.5 DLLs. If you use NET Standard 1.3 DLLs, you will get exceptions while the application running.**

For more information about .NET Standard, please find the following links:

.NET Standard/versions.md at master · dotnet/standard · GitHub

.NET Standard | Common APIs across all .NET implementations (microsoft.com)

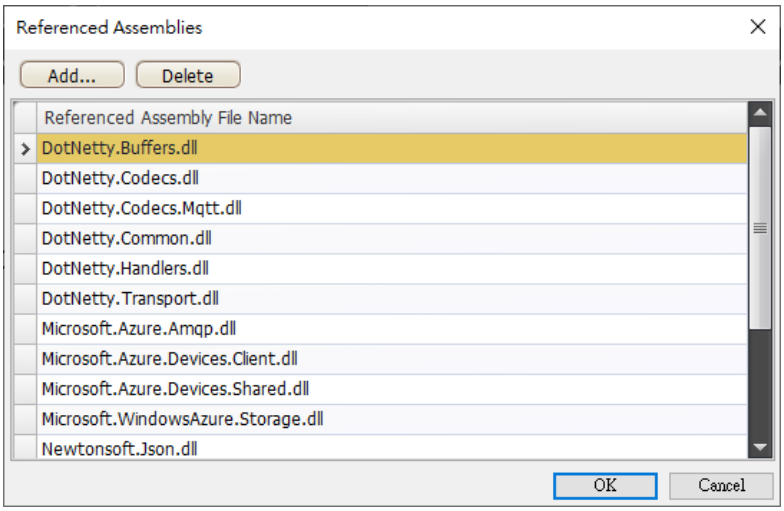
1. **DotNetty.Buffers.dll (Version 0.6.0 net45)**
2. **DotNetty.Codecs.dll (Version 0.6.0 net45)**
3. **DotNetty.Codecs.Mqtt.dll (Version 0.6.0 net45)**
4. **DotNetty.Transport.dll (Version 0.6.0 net45)**

5. **DotNetty.Common.dll (Version 0.6.0 net45)**
6. **DotNetty.Handlers.dll (Version 0.6.0 net45)**
7. Microsoft.Azure.Amqp.dll (Version 2.4.7 netstandard2.0)
8. Microsoft.Azure.Devices.Client.dll (Version 1.37.2 netstandard2.0)
9. Microsoft.Azure.Devices.Shared.dll (Version 1.28.1 netstandard2.0)
10. Microsoft.Azure.KeyVault.Core.dll (Version 1.0.0 net40)
11. Microsoft.WindowsAzure.Storage.dll (Version 9.3.2 netstandar1.3)
12. Microsoft.Extensions.Logging.Abstractions.dll (Version 1.1.1 netstandard1.1)
13. Microsoft.Extensions.Logging.dll (Version 1.1.1 netstandard1.1)
14. Newtonsoft.Json.dll (Version 12.0.3 netstandard2.0)
15. System.Runtime.CompilerServices.Unsafe.dll (Version 4.5.2 netstandard2.0)

The following DLLs are additional assemblies for IoT Hub Device Provision Service (DPS):

For optimizing iX project size, you should only reference only the protocols used by your application.

16. Microsoft.Azure.Devices.Provisioning.Client.dll (Version 1.17.1 netstandard2.0)
17. Microsoft.Azure.Devices.Provisioning.Transport.Amqp.dll (Version 1.14.1 netstandard2.0)
18. Microsoft.Azure.Devices.Provisioning.Transport.Http.dll (Version 1.13.1 netstandard2.0)
19. Microsoft.Azure.Devices.Provisioning.Transport.Mqtt.dll (Version 1.15.1 netstandard2.0)



5.2 Tags

We defined tags in the iX project for this sample to easily change the connection info of IoT Hub device and Device Provisioning Service.

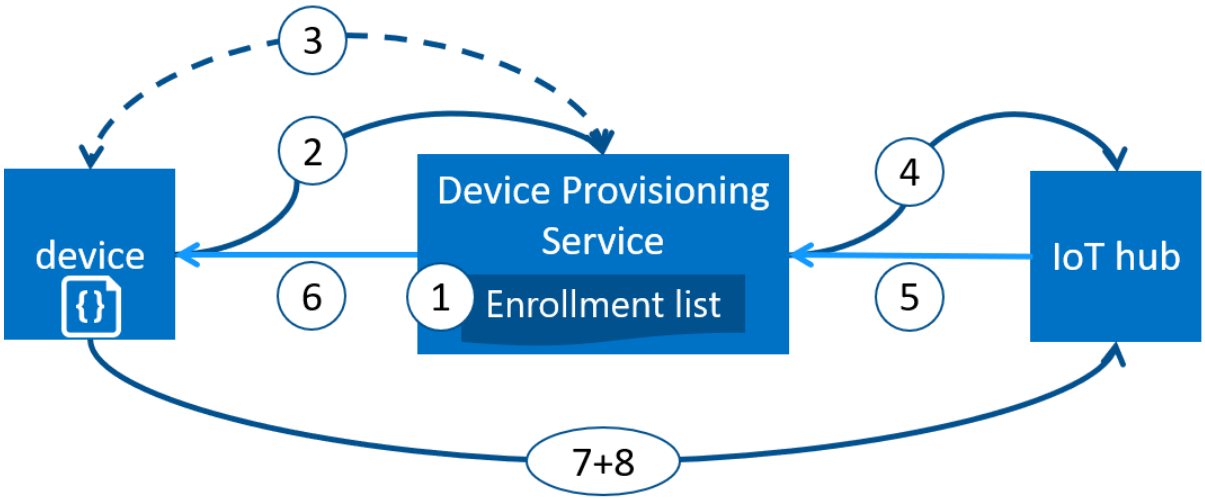
Mandatory Tag of IoTHubDeviceSettings class

Tag Name	Global Data Type	Description
DeviceId	STRING	The Device ID for the IoT device: Device identity used for device authentication and access control
DevicePrimaryKey	STRING	The Primary Key for the IoT device: The primary symmetric shared access key is stored in base64 format
IoTHubUri	STRING	Host Name/URI for the IoT Hub: It is contained in the connection string. Connection string based on primary key used in API call which allows the device to communicate with IoT Hub
DpsIdScope	STRING	The ID Scope of the Device Provisioning Service (DPS)
GlobalDeviceEndpoint	STRING	The Global device endpoint URI. of the Device Provisioning Service (DPS)
IndividualEnrollmentPrimaryKey	STRING	The PrimaryKey of an individual enrollment created in the Device Provisioning Service (DPS) The primary symmetric shared access key is stored in base64 format
IndividualEnrollmentSecondaryKey	STRING	The SecondaryKey of an individual enrollment created in the Device Provisioning Service (DPS) The secondary symmetric shared access key stored is in base64 format
EnrollmentGroupPrimaryKey	STRING	The PrimaryKey of an enrollment group created in the Device Provisioning Service (DPS) The primary symmetric shared access key is stored in base64 format
EnrollmentGroupSecondaryKey	STRING	The SecondaryKey of an enrollment group created in the Device Provisioning Service (DPS) The secondary symmetric shared access key stored is in base64 format
RegistrationId	STRING	- The Registration ID of an individual enrollment - For an enrollment group, it would be used by each device to compute its derived device key. And it also would be used as the IoT Hub Device ID.

Other tags are not necessary. They just are used to build the demonstrations. You can refer to the demos to find out how they work.

5.3 Using Device Provisioning Service for provisioning device

If you want to use Azure IoT Hub Device Provisioning Service (DPS) to provision devices to IoT Hubs, this sample demonstrates how to use DPS via Azure IoT SDKs. You can learn more about the concepts of DPS by reviewing Overview of Azure IoT Hub Device Provisioning Service | Microsoft Docs.



This sample assumes that you already have an enrollment list. So, it is involved from ◯,2 to ◯,8 steps shown in the above picture.

◯,1 Device manufacturer adds the device registration information to the enrollment list in the Azure portal.

◯,2 **Device contacts the DPS endpoint set at the factory. The device passes the identifying information to DPS to prove its identity.**

◯,3 DPS validates the identity of the device by validating the registration ID and key against the enrollment list entry using either a nonce challenge (Trusted Platform Module) or standard X.509 verification (X.509).

This sample is using a symmetric key for validation. You can use TPM or X.509 if your application needs. Notice: TPM is only supported when your hardware supports it.

◯,4 DPS registers the device with an IoT hub and populates the device's desired twin state.

◯,5 The IoT hub returns device ID information to DPS.

◯,6 **DPS returns the IoT hub connection information to the device. The device can now start sending data directly to the IoT hub.**

◯,7 The device connects to IoT hub.

◯,8 The device gets the desired state from its device twin in IoT hub.

But, only the 2nd, 3rd and 6th steps happen on your devices.

Therefore, what you need to do is contacting the DPS to request for provisioning device. Once it is validated, the devices will get the IoT Hub connection information.

1. Add namespaces according to your own application, e.g.

```
using Microsoft.Azure.Devices.Client;
using Microsoft.Azure.Devices.Provisioning.Client;
using Microsoft.Azure.Devices.Provisioning.Client.Transport;
using Microsoft.Azure.Devices.Shared;
```

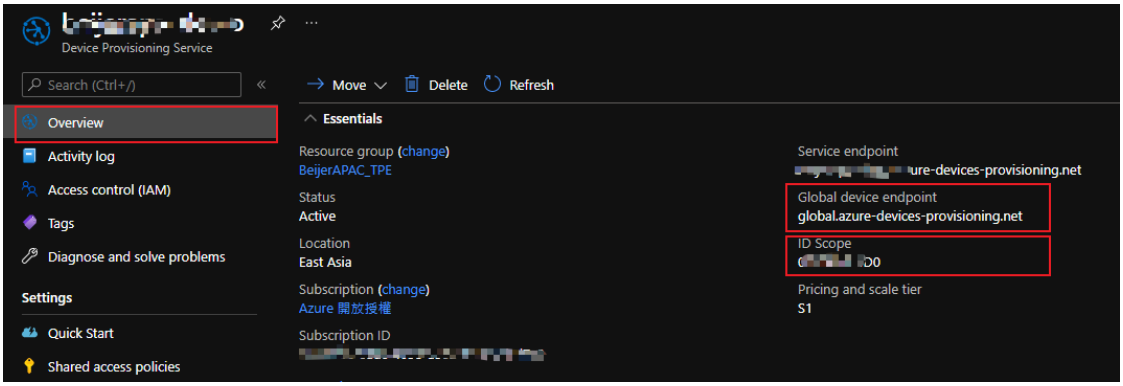
2. Leave IoT Hub related variables (Tags) blank

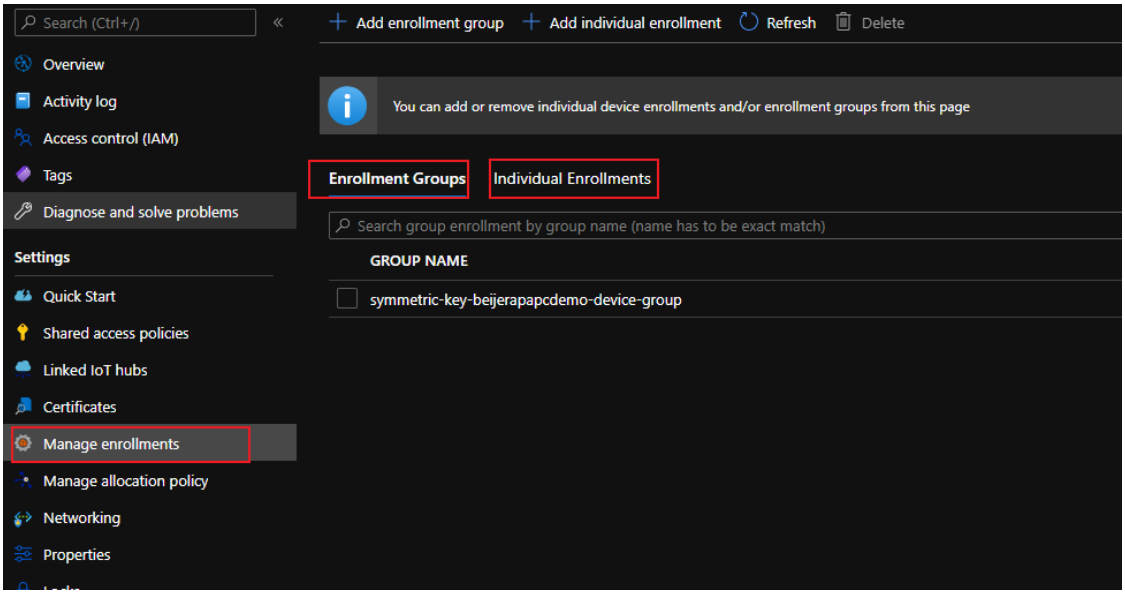
   e.g. DeviceId, DevicePrimaryKey and IoTHubUri

3. Assign a value to DPS related variables (Tags)

   e.g. DpsIdScope, GlobalDeviceEndpoint, RegistrationId, IndividualEnrollmentPrimaryKey, IndividualEnrollmentSecondaryKey, EnrollmentGroupPrimaryKey and EnrollmentGroupSecondaryKey

   If you are also in the device manufacturer/operator role, this means you are the one who created the enrollment list. You can find the Endpoint, ID Scope and enrollments in your DPS via Azure Portal. (Obviously, you know where to find them because you created them. Here is just for a newbie.) For learning about roles and operations, please see [IoT Hub Device Provisioning Service - Roles and operations | Microsoft Docs].

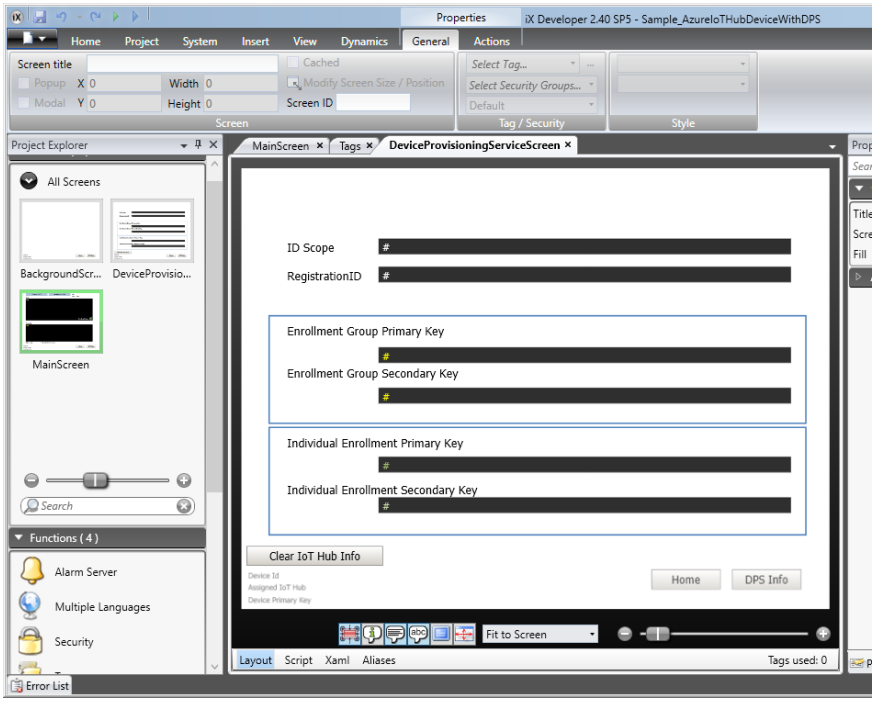


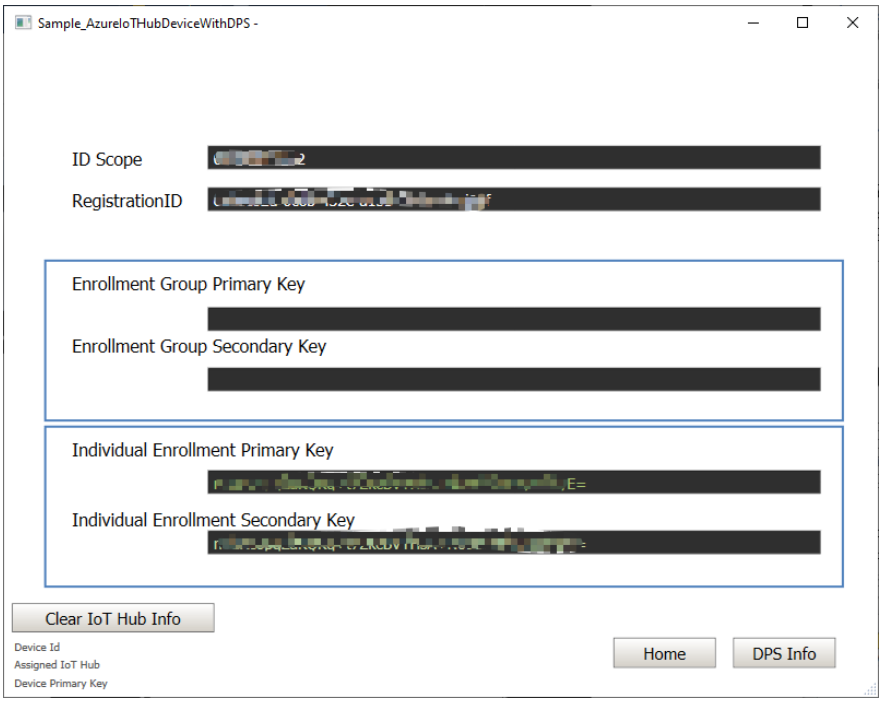


If you use an individual enrollment, you just need to assign a value for IndividualEnrollmentPrimaryKey and IndividualEnrollmentSecondaryKey and leave EnrollmentGroupPrimaryKey and EnrollmentGroupSecondaryKey blank, and vice versa. If you use an enrollment group, you just need to assign a value for EnrollmentGroupPrimaryKey and EnrollmentGroupSecondaryKey and leave IndividualEnrollmentPrimaryKey and IndividualEnrollmentSecondaryKey blank.

Or, you can delete individual enrollment or enrollment group tags and code.

Here is using a Screen to demonstrate assign DPS information and they would be stored in Non-Volatile tags. You can see the DeviceProvisioningServiceScreen of this sample.





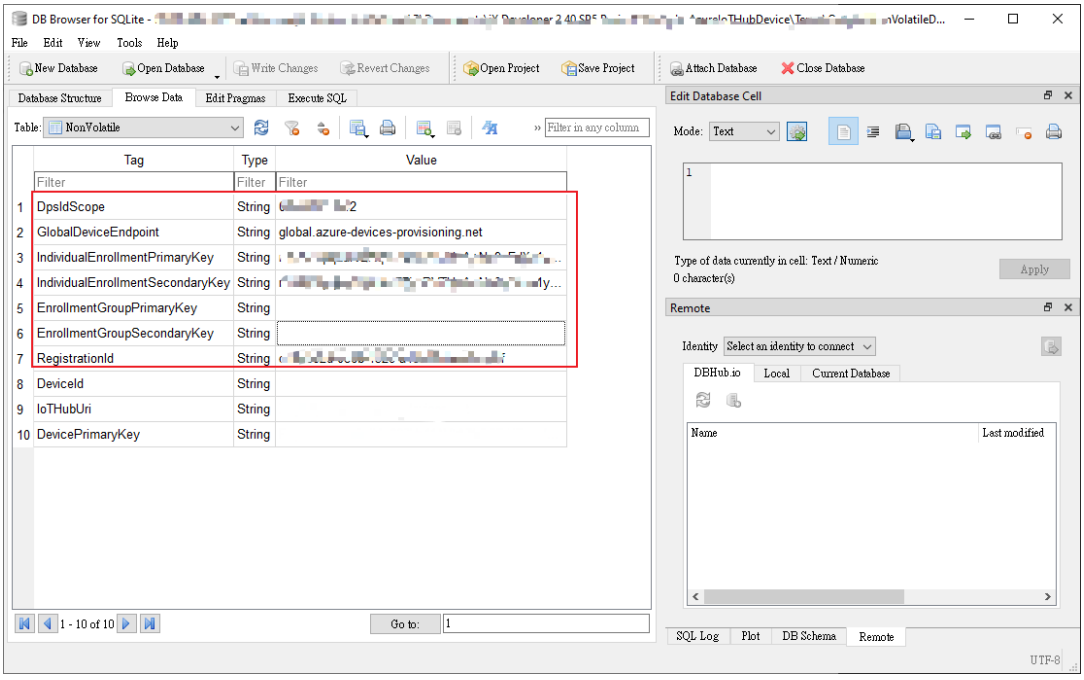
In a real case, DPS information might be stored in a file or the Windows registry. Your iX application needs to read a file or registry to get DPS information. If you use an enrollment group, you could also consider assigning information to the initial value of tags and auto-generating a Registration ID of each device. Of course, you could also manually set different initial values of individual enrollment for each device before downloading/exporting iX Runtime applications.

Tag			Controllers		Others
Name	Data Type	Access ...	Data Type	Controll...	Initial Value
DeviceId	STRING	ReadWrite	DEFAULT		
DevicePrimaryKey	STRING	ReadWrite	DEFAULT		
IoTHubUri	STRING	ReadWrite	DEFAULT		
Message	STRING	ReadWrite	DEFAULT		
DpsIdScope	STRING	ReadWrite	DEFAULT		[redacted]
GlobalDeviceEndpoint	STRING	ReadWrite	DEFAULT		global.azure-devices-provisioning.net
IndividualEnrollmentPrimaryKey	STRING	ReadWrite	DEFAULT		[redacted]
IndividualEnrollmentSecondary...	STRING	ReadWrite	DEFAULT		[redacted]
EnrollmentGroupPrimaryKey	STRING	ReadWrite	DEFAULT		
EnrollmentGroupSecondaryKey	STRING	ReadWrite	DEFAULT		
RegistrationId	STRING	ReadWrite	DEFAULT		[redacted]

Since we make these tags to be Non-Volatile for storing data in an SQLite file of the application. Another way for configuring DPS tags is by modifying the SQLite database. After the application's first startup is completed, the application database would be created. Then, closing the application would let you be able to modify database. You can use an open source tool, [DB Browser for SQLite (sqlitebrowser.org)], to manipulate an SQLite file.

The Non-Volatile database file is located where your application deploys to. Normally, it is located to iX Runtime project default path . For example, "C:\Users\Public\Documents\Beijer Electronics AB\iX Developer Runtime 2.40 SP5\Project**NonVolatileDatabase.db**".

Please see the below picture on "editing the **NonVolatile** table of the **NonVolatileDatabase.db** database".



4. Program to register to an IoT Hub via DPS and get IoT Hub information

```
if (!String.IsNullOrWhiteSpace(Globals.Tags.EnrollmentGroupPrimaryKey.Value)
    && !String.IsNullOrWhiteSpace(Globals.Tags.EnrollmentGroupSecondaryKey.Value))
{
    // Group enrollment flow, the primary and secondary keys are derived
    // from the enrollment group keys and from the desired registration id
    primaryKey = ComputeDerivedSymmetricKey(
        Convert.FromBase64String(Globals.Tags.EnrollmentGroupPrimaryKey.Value), registrationId);
    secondaryKey = ComputeDerivedSymmetricKey(
        Convert.FromBase64String(Globals.Tags.EnrollmentGroupSecondaryKey.Value), registrationId);
}
else if (!String.IsNullOrWhiteSpace(Globals.Tags.IndividualEnrollmentPrimaryKey.Value)
    && !String.IsNullOrWhiteSpace(Globals.Tags.IndividualEnrollmentSecondaryKey.Value))
{
    // Individual enrollment flow, the primary and secondary keys are
    // the same as the individual enrollment keys
    primaryKey = Globals.Tags.IndividualEnrollmentPrimaryKey.Value;
    secondaryKey = Globals.Tags.IndividualEnrollmentSecondaryKey.Value;
}
else
{
    System.Diagnostics.Debug.WriteLine(@"Invalid configuration provided
        , must provide group enrollment keys or individual enrollment keys");
    return false;
}

using (var security = new SecurityProviderSymmetricKey(registrationId, primaryKey, secondaryKey))
// Select one of the available transports:
// To optimize for size, reference only the protocols used by your application.
using (var transport = new ProvisioningTransportHandlerAmqp(TransportFallbackType.TcpOnly))
// using (var transport = new ProvisioningTransportHandlerHttp())
// using (var transport = new ProvisioningTransportHandlerMqtt(TransportFallbackType.TcpOnly))
// using (var transport = new ProvisioningTransportHandlerMqtt(TransportFallbackType.WebSocketOnly))
{
    ProvisioningDeviceClient provClient =
        ProvisioningDeviceClient.Create(gobalDeviceEndpoint, id Scope, security, transport);
```

- Using symmetric key attestation for authentication and enrollment verification
- Register the device

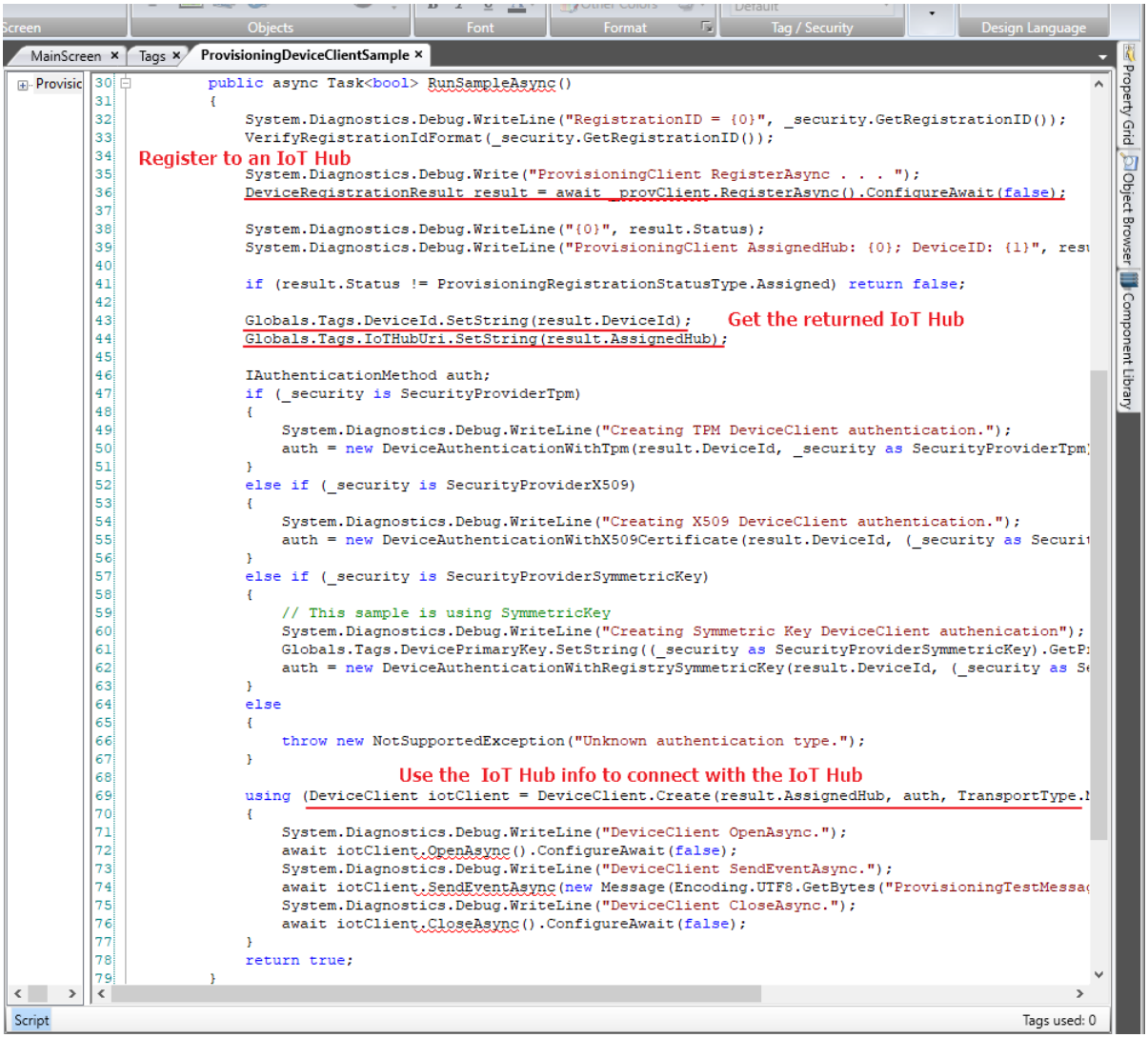
```
DeviceRegistrationResult result =
        await _provClient.RegisterAsync().ConfigureAwait(false);
```

- Upon successful registration, a unique device ID and IoT Hub endpoint are returned to the registration application for communicating with IoT Hub.
- Save the device ID, IoT Hub endpoint and symmetric key to tags. Next time application startup could connect to IoT hub directly.

```
if (result.Status != ProvisioningRegistrationStatusType.Assigned)
        return false;
Globals.Tags.DeviceId.SetString(result.DeviceId);
Globals.Tags.IoTHubUri.SetString(result.AssignedHub);
Globals.Tags.DevicePrimaryKey.SetString(
        (_security as SecurityProviderSymmetricKey).GetPrimaryKey());
IAuthenticationMethod auth = new
        DeviceAuthenticationWithRegistrySymmetricKey(result.DeviceId,
        (_security as SecurityProviderSymmetricKey).GetPrimaryKey());
```

- Now, the device can connect with IoT hub.

```
using (DeviceClient iotClient = DeviceClient.Create(result.AssignedHub
        , auth, TransportType.Mqtt))
{
        await iotClient.OpenAsync().ConfigureAwait(false);
        await iotClient.SendEventAsync(new
               Message(Encoding.UTF8.GetBytes("ProvisioningTestMessage")))
               .ConfigureAwait(false);
        await iotClient.CloseAsync().ConfigureAwait(false);
}
```

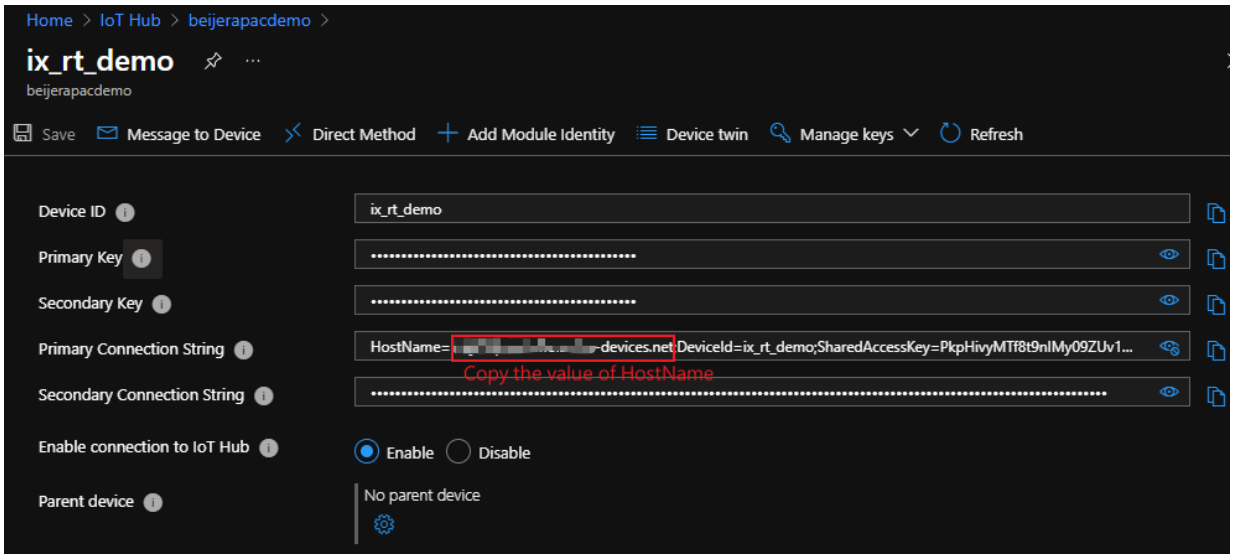


Notice: There are some red error and warning marks wavy underlines in the picture. You do not need to worry aboot them. For more information about it, please see 6 Suggestions

5.4 Send telemetry (Device-to-Cloud messages) to Azure IoT Hub

Once you execute the provisioning device process, you already have IoT Hub connection information. In this sample, the information would be stored in non-volatile tags.

If you only want to send telemetry to an existing Device Id of an IoT Hub, you can get the IoT Hub information from Azure Portal. And manually set them to corresponding tags.



1. Add namespaces according to your own application, e.g.

```
using Microsoft.Azure.Devices.Client;
```

This sample also uses JSON.NET (Json.NET - Newtonsoft) for Serializing and Deserializing JSON (Serializing and Deserializing JSON - newtonsoft.com).

```
using Newtonsoft.Json;
```

2. Assign a value to IoT Hub related variables (Tags)

If you do not use DPS to register device, you can copy the device ID, IoT Hub endpoint and primary key (symmetric key) from Azure portal before deploying (downloading) an iX application to an iX Runtime device.

Tag			Controllers		Others
Name	Data Type	Acces...	Data Type	Controll...	Initial Value
DeviceId	STRING	ReadWrite	DEFAULT		
DevicePrimaryKey	STRING	ReadWrite	DEFAULT		
IoTHubUri	STRING	ReadWrite	DEFAULT		
Message	STRING	ReadWrite	DEFAULT		

- Copy the Device ID from Azure portal and paste on the initial value of DeviceId tag.

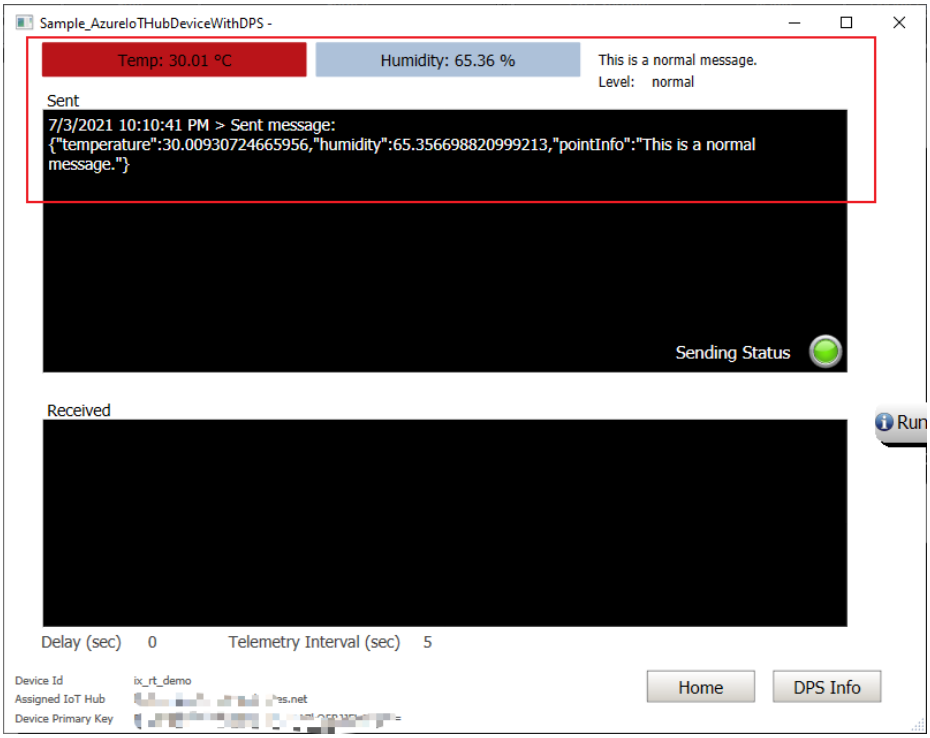
- Copy the Device Primary Key from Azure portal and paste on the initial value of DevicePrimaryKey tag.
- Copy the IoT Hub URI from Azure portal and paste on the initial value of IoTHubUri tag.

3. Compose and send your own message of telemetry.

```
var telemetryDataPoint = new
{
      temperature = Globals.Tags.CurrentTemperature.Value.Double,
      humidity = Globals.Tags.CurrentHumidity.Value.Double,
      pointInfo = Globals.Tags.InfoString.Value.String
};
// Serialize the telemetry data and convert it to JSON.
var telemetryDataString = JsonConvert.SerializeObject(telemetryDataPoint);
var message = new Message(Encoding.UTF8.GetBytes(telemetryDataString))
{
      ContentType = "application/json",
      ContentEncoding = "utf-8",
};
// Add one property to the message.
message.Properties.Add("level", Globals.Tags.LevelValue.Value.String);
// Submit the message to the hub.
await _deviceClient.SendEventAsync(message);
```

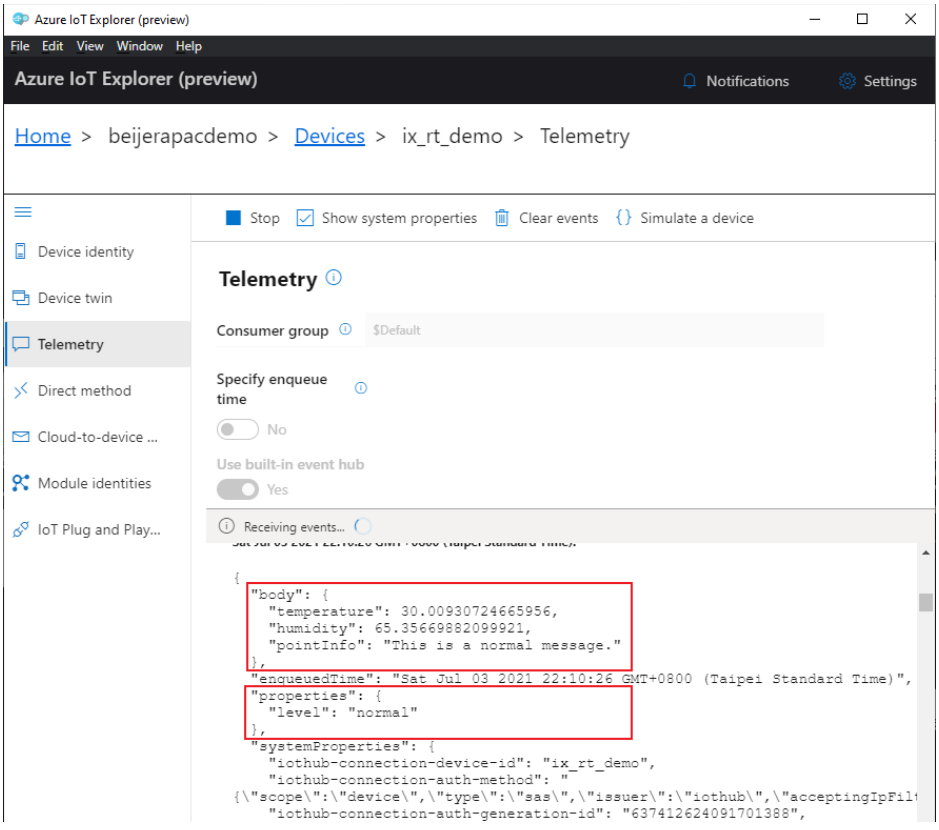
4. Testing and monitoring D2C messages

Run the sample

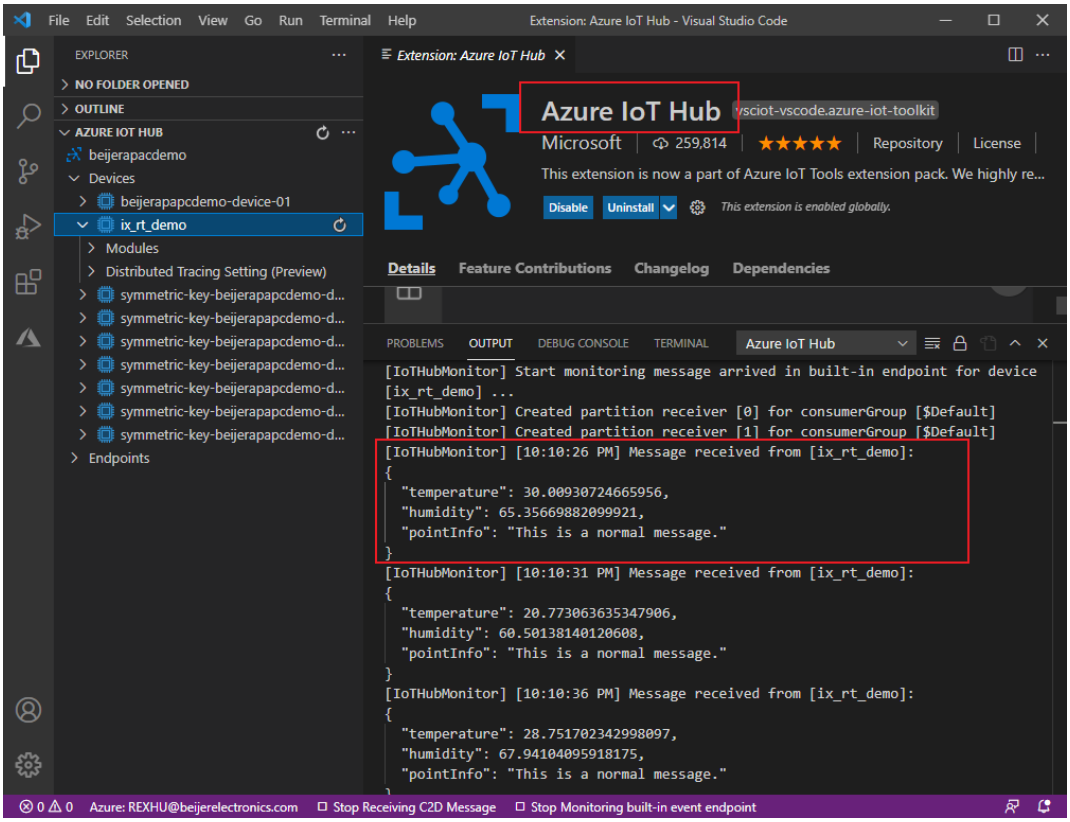


Using Azure IoT Explorer to monitor.

For more information about Azure IoT Explorer, please refer to this link: Install and use Azure IoT explorer | Microsoft Docs.

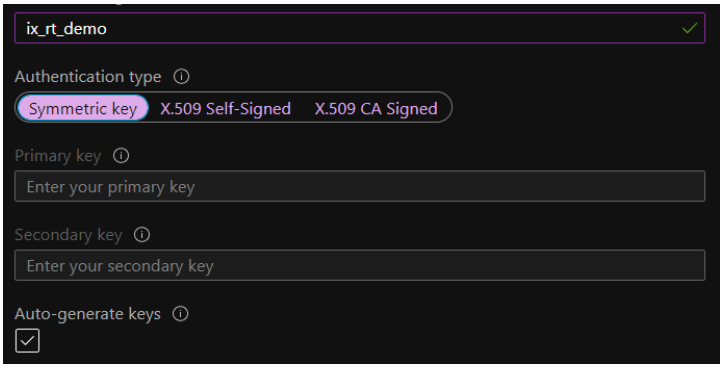


Using another tool, Visual Studio Code with Azure IoT Hub extension, to monitor.



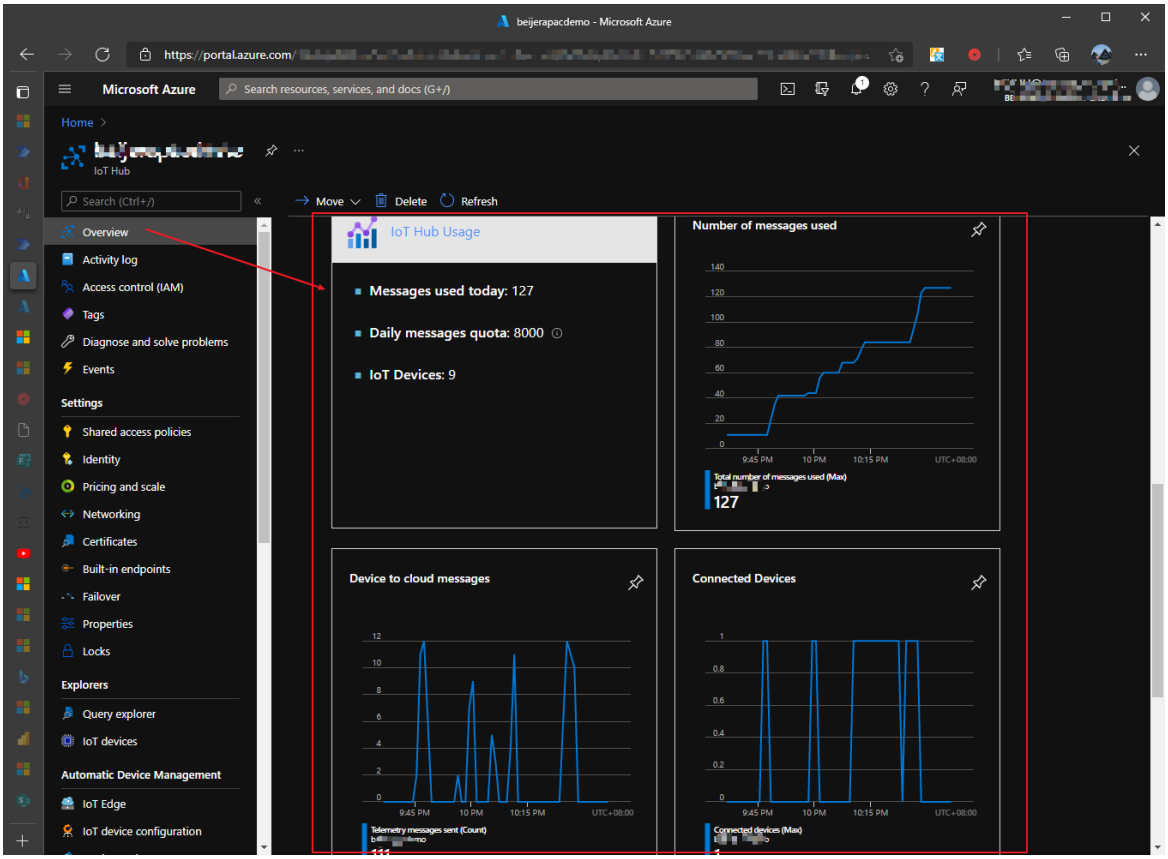
Notice: This sample uses Symmetric Key as below pictures. For X.509, please refer to Control access to IoT Hub - Supported X.509 certificates.

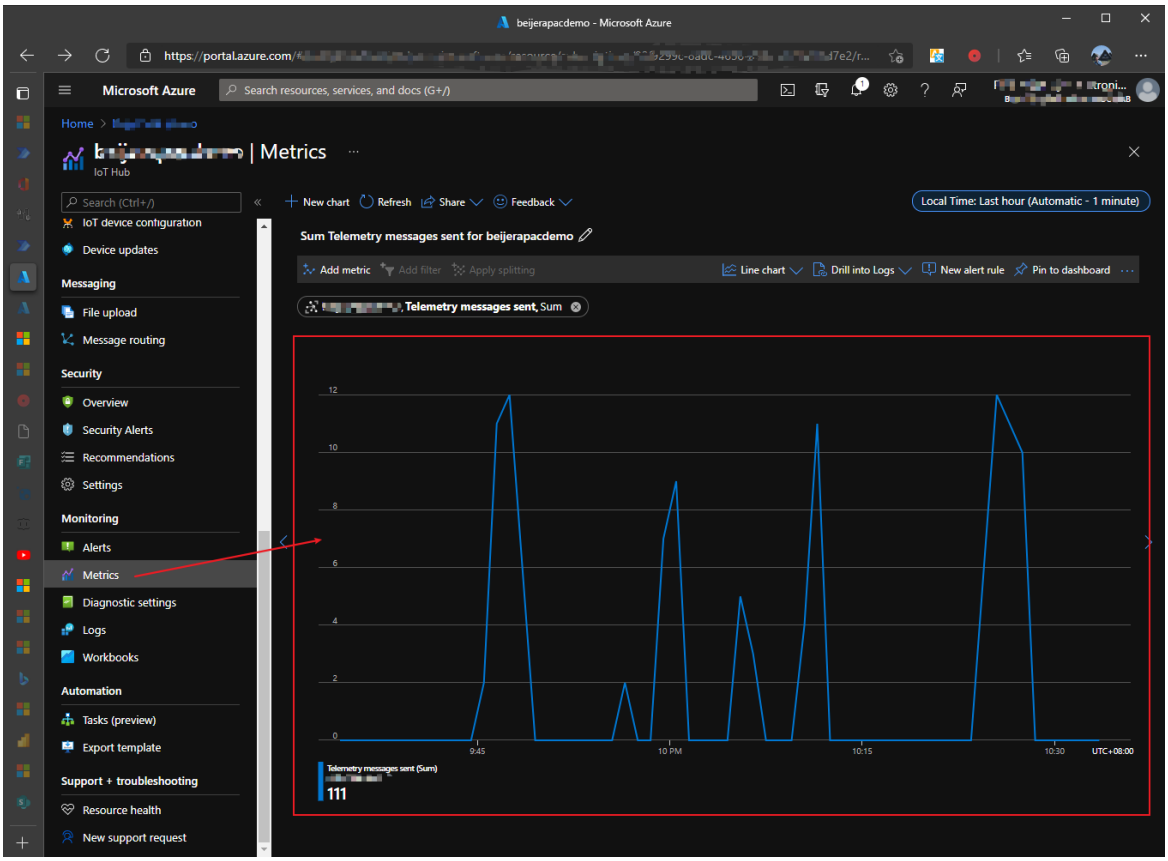
And refer to Azure IoT Samples for C# (.NET) for coding details.




```
private static DeviceClient _deviceClient;

public static DeviceClient DeviceClient
{
    get
    {
        if (_deviceClient == null)
        {
            _deviceClient = DeviceClient.Create(
                Globals.Tags.IoTHubUri.Value
                , new DeviceAuthenticationWithRegistrySymmetricKey(
                  Globals.Tags.DeviceId.Value
                  , Globals.Tags.DevicePrimaryKey.Value)
                , TransportType.Mqtt);
        }
        return _deviceClient;
    }
}
```

When the iX application is running, you can see the number of messages used on the IoT Hub resource overview page or metrics page of Azure portal.





5. Read Device-to-Cloud messages from the built-in endpoint

   After sending device-to-cloud messages from the device, you can read the messages from the built-in endpoint or add other endpoints to utilize messages for your back-end applications. For example, you can assume Azure IoT Explorer is your dashboard back-end app. It presents visualization to uses.

   For more information, please refer to below links:

   Read device-to-cloud messages from the built-in endpoint

   Use message routes and custom endpoints for device-to-cloud messages

5.5 Receive and read Cloud to Device messages

If you not only want to send device-to-cloud messages, but also want to interact with devices from back-end applications. You can learn about how to achieve this goal from Azure IoT Hub cloud-to-device options | Microsoft Docs. Here is the abstract: IoT Hub provides three options for device apps to expose functionality to a back-end app:

- **Direct methods** for communications that require immediate confirmation of the result. Direct methods are often used for interactive control of devices such as turning on a fan.
- **Twin's desired properties** for long-running commands intended to put the device into a certain desired state. For example, set the telemetry sending interval to 30 minutes.
- **Cloud-to-device messages** for one-way notifications to the device app.

Following chapters will demonstrate how to implement them in an iX application one by one in our samples. All of them are assuming the device (_deviceClient object) is already connecting with IoT Hub.

For receiving and read Cloud-to-Device messages, there are two methods active polling and passive notification.

1. Add namespaces according to your own application, e.g.

```
using Microsoft.Azure.Devices.Client;
```

2. Active polling: This can be used to check if there is any not received/unread message in IoT Hub after an iX application staupup.

```
Message receivedMessage =
      await _deviceClient.ReceiveAsync(timeout).ConfigureAwait(false);
// Do something for processing a message
ProcessReceivedMessage(receivedMessage);
// Deletes a received message from the device queue
await _deviceClient.CompleteAsync(receivedMessage).ConfigureAwait(false);
```

3. Passive notification: This can be used at application runtime.
   - Write a callback method:

```
private async Task OnC2dMessageReceived(Message receivedMessage
      , object userContext)
{
      // Do something for processing a message
      ProcessReceivedMessage(receivedMessage);
      // Deletes a received message from the device queue
      await _deviceClient.CompleteAsync(receivedMessage).ConfigureAwait(false);
}
```

   - Register the callback method to the DeviceClient object:

```
await _deviceClient.SetReceiveMessageHandlerAsync(OnC2dMessageReceived,
      _deviceClient).ConfigureAwait(false);
```

4. Processing a received message:

```
private void ProcessReceivedMessage(Message receivedMessage)
{
      // Get message string
      string messageData =
            Encoding.ASCII.GetString(receivedMessage.GetBytes());
      // User set application properties can be retrieved from the
      // Message.Properties dictionary.
      foreach (KeyValuePair<string, string> prop in receivedMessage.Properties)
      {
            // Do something for your application e.g. set a delay time
            if (prop.Key == "delay")
            {
                  int delay = 0;
                  if (int.TryParse(prop.Value, out delay))
                        Globals.Tags.Delay.SetAnalog(delay);
            }
      }
}
```

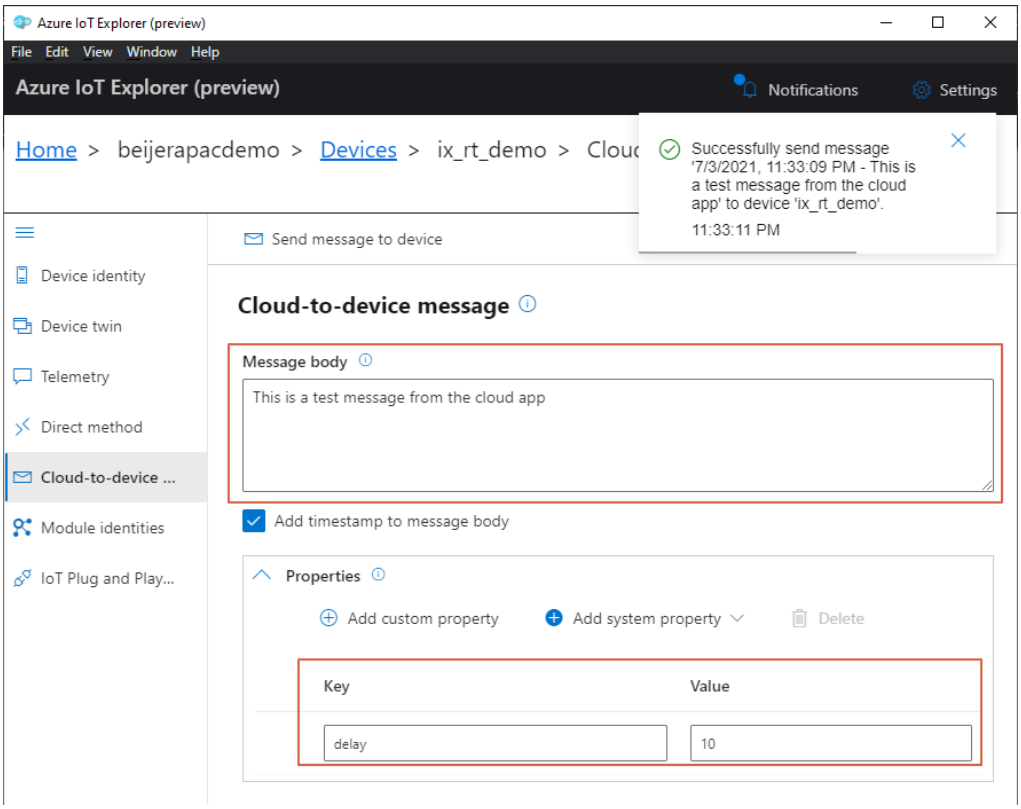
5. Unsubscribe a callback when a device application does not want to receive notifications

```
// This is how one can unsubscribe a callback for C2D messages using a null
// callback handler.
_deviceClient.SetReceiveMessageHandlerAsync(null, _deviceClient)
      .ConfigureAwait(false).GetAwaiter().GetResult();
```

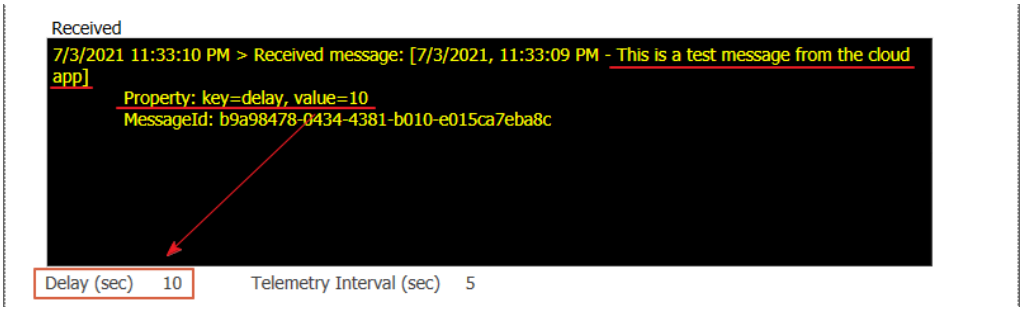
6. Testing and monitoring C2D messages

   Before sending a C2D message, nothing appears on the Received region of the device application.

Using Azure IoT Explorer to send a message and its property.

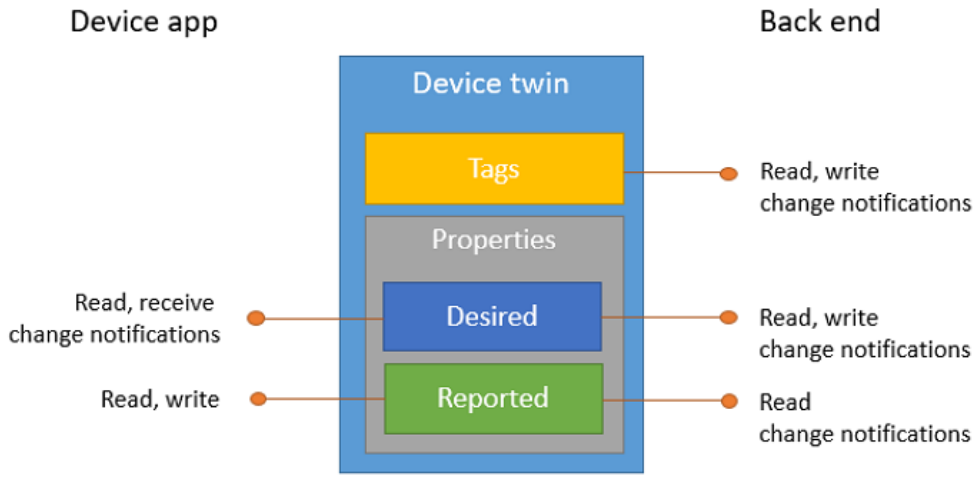


The device application received and read the message and changed its delay state from 0 sec to 10 secs.



5.6 Get update of Device Twins and send report to it

The below diagram is a quote from [Understand Azure IoT Hub device twins | Microsoft Docs]



The following example shows a device twin JSON document:

```
{
    "deviceId": "devA",
    "etag": "AAAAAAAAAAc=",
    "status": "enabled",
    "statusReason": "provisioned",
    "statusUpdateTime": "0001-01-01T00:00:00",
    "connectionState": "connected",
    "lastActivityTime": "2015-02-30T16:24:48.789Z",
    "cloudToDeviceMessageCount": 0,
    "authenticationType": "sas",
    "x509Thumbprint": {
        "primaryThumbprint": null,
```

```
            "secondaryThumbprint": null
        },
        "version": 2,
        "tags": {
            "$etag": "123",
            "deploymentLocation": {
                "building": "43",
                "floor": "1"
            }
        },
        "properties": {
            "desired": {
                "telemetryConfig": {
                    "sendFrequency": "5m"
                },
                "$metadata" : {...},
                "$version": 1
            },
            "reported": {
                "telemetryConfig": {
                    "sendFrequency": "5m",
                    "status": "success"
                },
                "batteryLevel": 55,
                "$metadata" : {...},
                "$version": 4
            }
        }
    }
```

The device to our sample also has its own Device Twins JSON document.

1. Add namespaces according to your own application, e.g.

```
using Microsoft.Azure.Devices.Shared;
using Microsoft.Azure.Devices.Client;
using Newtonsoft.Json;
```

2. Write a callback method for handling desired property updated event

```
// A type for Serializing and Deserializing JSON
private class TelemetrySetting
{
      public int telemetryInterval { get; set; }
}

private async Task OnDesiredPropertyChangedAsync(TwinCollection
      desiredProperties, object userContext)
{
      var telemetrySetting = JsonConvert.DeserializeObject<TelemetrySetting>
            (desiredProperties.ToJson());
      Globals.Tags.TelemetryInterval.SetAnalog(
            telemetrySetting.telemetryInterval);
      TwinCollection reportedProperties = new TwinCollection();
      reportedProperties["telemetryInterval"] =
            Globals.Tags.TelemetryInterval.Value.Int;
      // Push reported property changes up to the service.
      await _deviceClient.UpdateReportedPropertiesAsync(reportedProperties)
            .ConfigureAwait(false);
}
```

3. Register the callback handler to the DeviceClient object

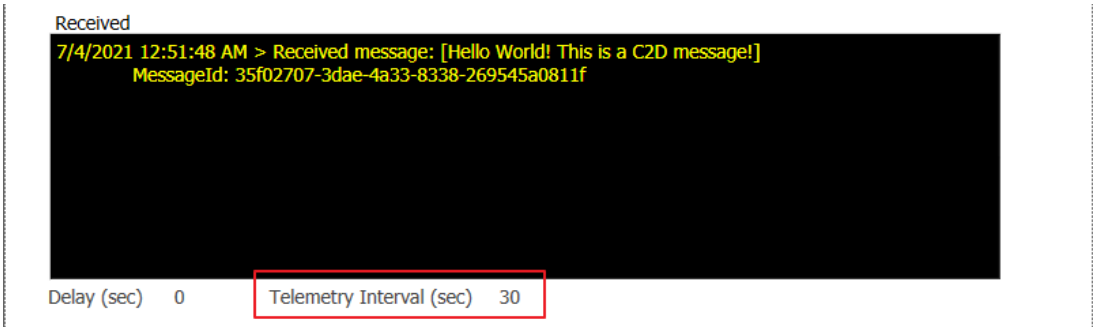
```
await _deviceClient.SetDesiredPropertyUpdateCallbackAsync(
      OnDesiredPropertyChangedAsync, null).ConfigureAwait(false);
```

4. Unsubscribe a callback when a device application does not want to receive notifications of changes in the desired properties.

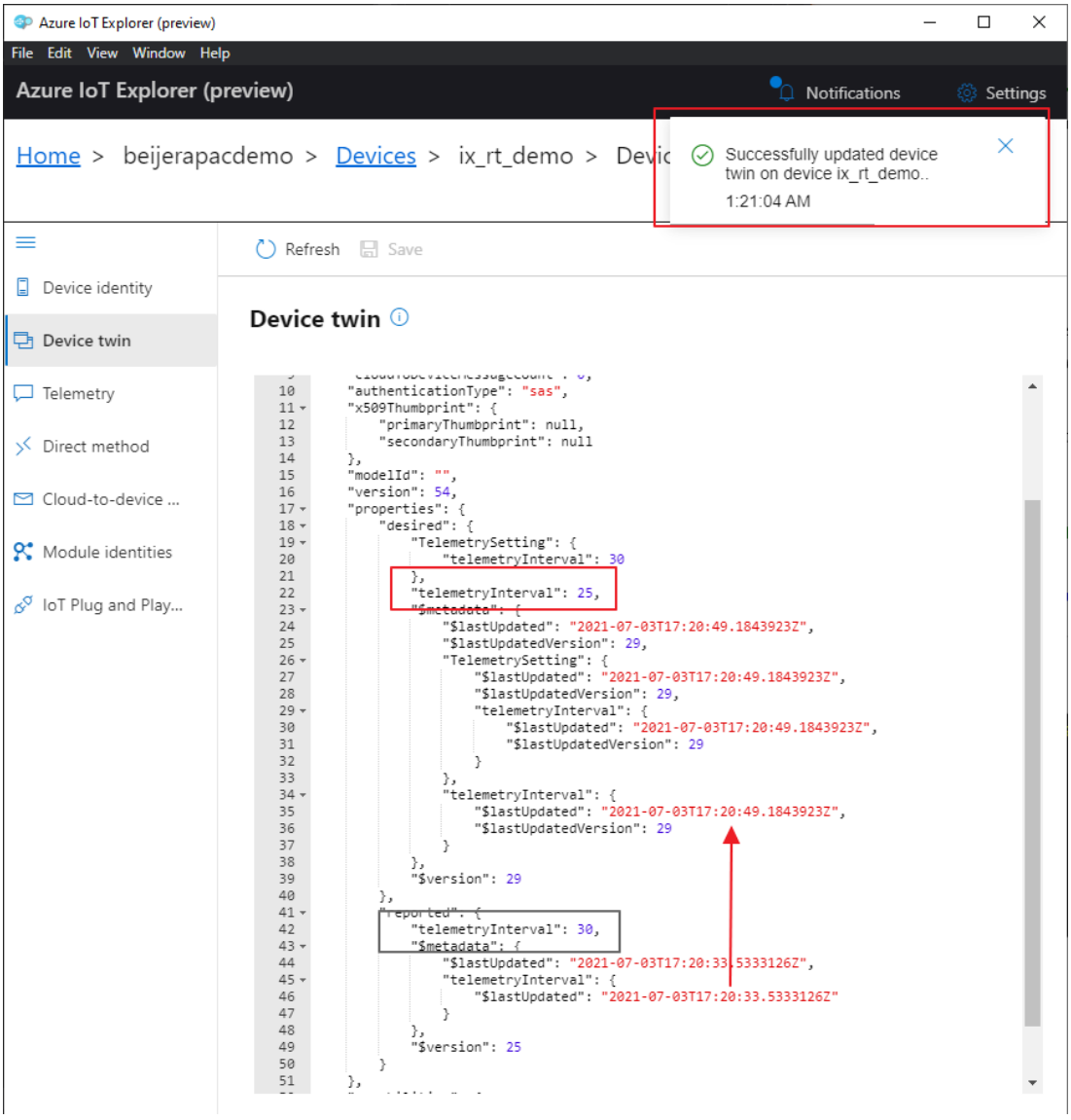
```
// This is how one can unsubscribe a callback for properties using a null
// callback handler.
_deviceClient.SetDesiredPropertyUpdateCallbackAsync(null, null)
      .ConfigureAwait(false).GetAwaiter().GetResult();
```

5. Testing and monitoring Device Twins

   Before changing Device Twins, the value of the interval tag is 30 secs.

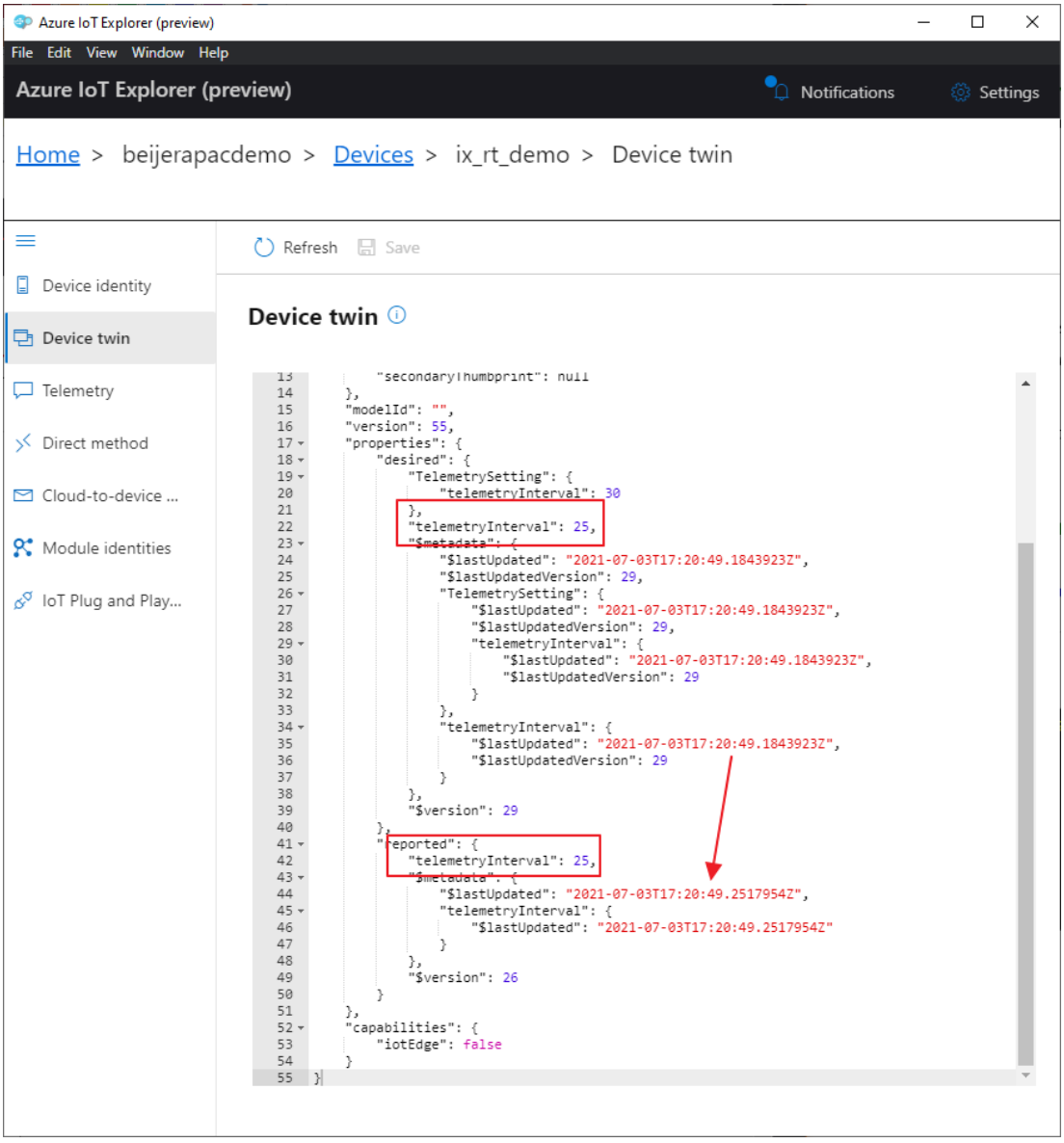


   Changing the subitem (telemetryInterval) value of desired properties of desired of Device Twins from 30 to 25.



Then, the device is notified of the desired property update event. It changes the telemetry interval from 30 secs to 25 secs and reports to the IoT hub.

After a moment, click "Refresh" and the subitem (telemetryInterval) value of reported properties changed.



5.7 Invoke direct methods from IoT Hub

"IoT Hub gives you the ability to invoke direct methods on devices from the cloud. Direct methods represent a request-reply interaction with a device similar to an HTTP call in that they succeed or fail immediately (after a user-specified timeout). This approach is useful for scenarios where the course of immediate action is different depending on whether the device was able to respond." This is a quote from Understand Azure IoT Hub direct methods | Microsoft Docs.

Let's invoke a direct method from a back-end app.

1. Add namespaces according to your own application, e.g.

```
using Microsoft.Azure.Devices.Client;
using Newtonsoft.Json;
```

2. Write callback methods for dealing with direct method invocations. Here we have three direct methods in our sample. Therefore, we write three handlers to them.

```
// Types for Serializing and Deserializing JSON
private class DeviceData
{
      public string Name { get; set; }
}
private class DeviceSetting
{
      public int delay { get; set; }
}

// For "start" direct method to set a delay time
private static Task<MethodResponse> SetDelay(MethodRequest methodRequest
      , object userContext)
{
      var deviceSetting = JsonConvert.DeserializeObject<DeviceSetting>(
            methodRequest.DataAsJson);
      if (deviceSetting != null)
      {
            Globals.Tags.DirectMethod_Delay.SetAnalog(deviceSetting.delay);
            // Acknowlege the direct method call with a 200 success message
            string result = string.Format(
                  "{{\"result\":\"Executed direct method: {0}\"}}"
                  , methodRequest.Name);
            return Task.FromResult(
                  new MethodResponse(Encoding.UTF8.GetBytes(result), 200));
```

```
        }
        else
        {
                // Acknowlege the direct method call with a 400 error message
                string result = "{\"result\":\"Invalid parameter\"}";
                return Task.FromResult(
                        new MethodResponse(Encoding.UTF8.GetBytes(result), 400));
        }
}

// For “WriteToConsole” direct method to output messages
private Task<MethodResponse> WriteToConsoleAsync(MethodRequest methodRequest
        , object userContext)
{
        Console.WriteLine(string.Format("\t *** {0} was called.",
                methodRequest.Name));
        Console.WriteLine(string.Format("\t{0}\n", methodRequest.DataAsJson));
}

// For “GetDeviceName” direct method to get the devive’s name
private Task<MethodResponse> GetDeviceNameAsync(MethodRequest methodRequest
        , object userContext)
{
        MethodResponse retValue;
        if (userContext == null)
        {
                retValue = new MethodResponse(new byte[0], 500);
        }
        else
        {
                var deviceData = (DeviceData)userContext;
                string result = JsonConvert.SerializeObject(deviceData);
                retValue = new MethodResponse(Encoding.UTF8.GetBytes(result)
                        , 200);
        }
}
```

3. Register the callback handlers to the DeviceClient object

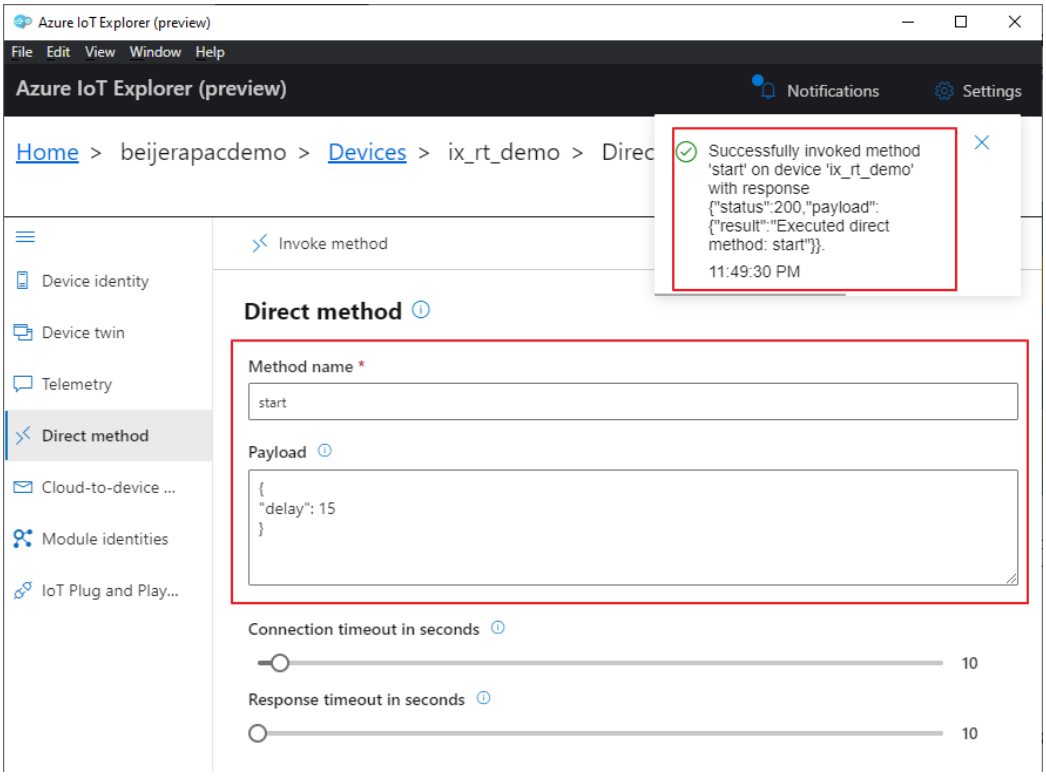
```
await _deviceClient.SetMethodHandlerAsync("start", SetDelay, null)
      .ConfigureAwait(false);
await _deviceClient.SetMethodHandlerAsync("WriteToConsole",
WriteToConsoleAsync, null)
      .ConfigureAwait(false);


await _deviceClient.SetMethodHandlerAsync(
      "GetDeviceName", GetDeviceNameAsync
      , new DeviceData { Name = "DeviceClientMethodSample" })
      .ConfigureAwait(false);
```

4. Unsubscribe a callback when a device application does not want to receive invocations

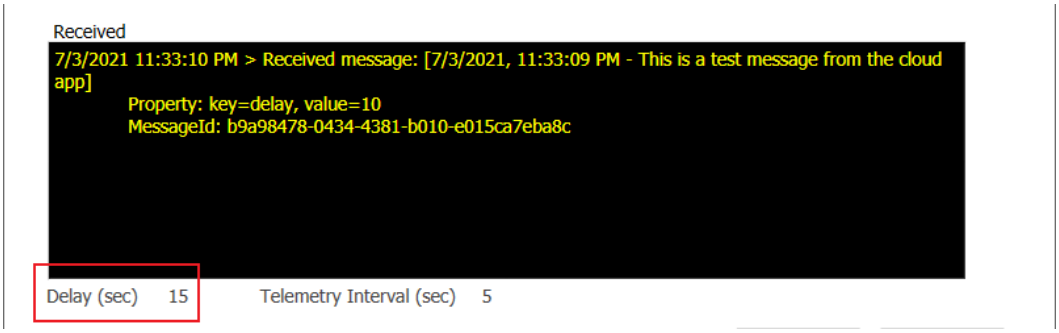
```
// You can unsubscribe from receiving a callback for direct methods by setting
// a null callback handler.
_deviceClient.SetMethodHandlerAsync("start", null, null)
      .ConfigureAwait(false).GetAwaiter().GetResult();
_deviceClient.SetMethodHandlerAsync("GetDeviceName", null, null)
      .ConfigureAwait(false).GetAwaiter().GetResult();
_deviceClient.SetMethodHandlerAsync("WriteToConsole", null, null)
      .ConfigureAwait(false).GetAwaiter().GetResult();
```

5. Testing and monitoring Device Twins

   Using Azure IoT Explorer to invoke the start method and get a 200 OK response.



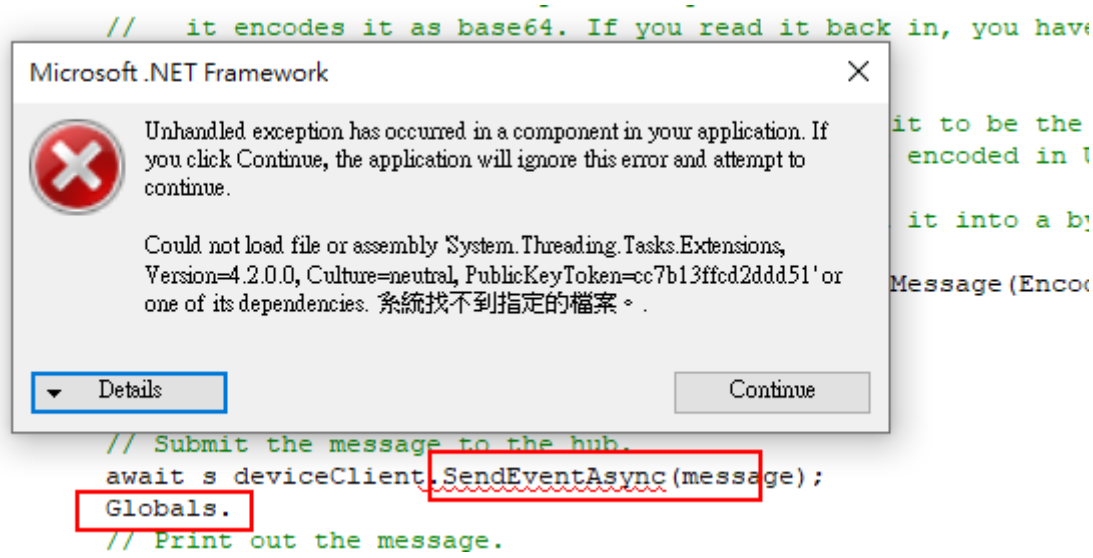
   And the device application executed the start method to set the start delay time to 15 secs.



6 Suggestions

We suggest using iX Developer SP5 or a later version and coding in Visual Studio.

The intellisense of the C# editor of iX Developer SP4 and an earlier version is bad for the new C# syntax. As below picture, it will throw an exception when typing a DOT(.) follow Globals object after using Tasks. And, it also shows the red error and warning marks wavy underlines (known as squiggles). But it can be ignored and compiled successfully. It just bothers a programmer. We suggest using an external tool (VS Code / Studio) and skip the editor when utilizing new C# syntax.



7 About Beijer Electronics

Beijer Electronics is a multinational, cross-industry innovator that connects people and technologies to optimize processes for business-critical applications. Our offer includes operator communication, automation solutions, digitalization, display solutions and support. As experts in user-friendly software, hardware

and services for the Industrial Internet of Things, we empower you to meet your challenges through leading-edge solutions.

Beijer Electronics is a Beijer Group company. Beijer Group has a sale over 1.6 billion SEK in 2019 and is listed on the NASDAQ OMX Nordic Stockholm Small Cap list under the ticker BELE.
www.beijergroup.com


China
Shanghai


Denmark
Roskilde


France
Champlan


Germany
Nürtingen


Italy
Salsomaggiore


Norway
Lier


South Korea
Geumcheon-gu


Sweden HQ
Malmö


Taiwan
Taipei City


Turkey
Istanbul


United Kingdom
Nottingham


Usa
Salt Lake City

7.1 Contact us

Global offices and distributors